高性能计算应用实践实验报告（lab6&9）

2024 秋 苏涵 2023311B25

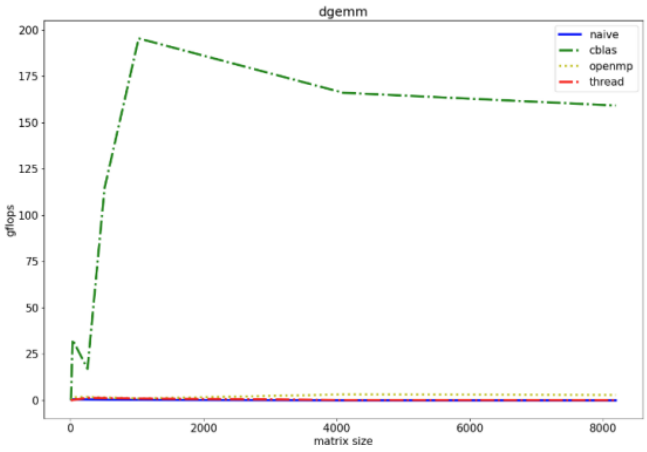
实验环境

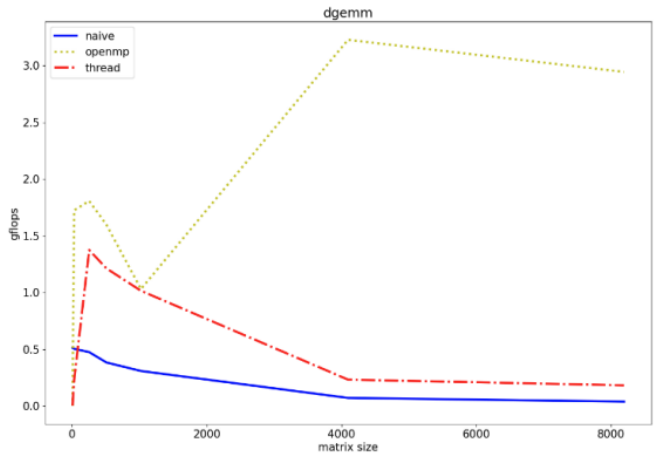
OS：Ubuntu 22.04.3 LTS
gcc：(Ubuntu 11.4.0-1ubuntu1~22.04) 11.4.0
cpu 型号：11th Gen Intel(R) Core(TM) i7-11370H
频率：3.30GHz
核数：4
内存大小：16GB
显卡型号：NVIDIA RTX A2000 Laptop GPU

lab6

数据分析

(因为之前没有保存 lab6 的文件，所以 lab6 部分的图片是直接截图的不是特别清晰，可移步只有 lab6 版本的实验报告)





当矩阵规模为 8*8 时，naive 的 gflops 大于其它方法，其它情况下 gflops 从大到小

排列依次为 cblas、openmp、thread、naive。且 cblas 的 gflops 远大于其他三种方法。

实现方式介绍与核心代码

naïve

使用三重嵌套循环来计算矩阵乘法。外层两个循环遍历结果矩阵 C 的每个元素，内层循环计算对应元素的和

naive_gemm.c
```c
#include "defs.h"

void MY_MMult(int m, int n, int k, double *a, int lda,
              double *b, int ldb,
              double *c, int ldc)
{
  int i, j, p;

  for (i = 0; i < m; i++) /* Loop over the rows of C */
  {
    for (j = 0; j < n; j++) /* Loop over the columns of C */
    {
      for (p = 0; p < k; p++)
      {
        C(i, j) = C(i, j) + A(i, p) * B(p, j);
      }
    }
  }
}
```

thread

将矩阵乘法任务分配到多个线程上，每个线程负责计算矩阵的一部分，最终合并结果

pthreads_gemm.c
```c
#include "defs.h"
#include <pthread.h>

typedef struct {
    int start_row;
    int end_row;
    double* A;
    double* B;
    double* C;
    int m;
    int n;
} ThreadData;


void* matrix_multiply(void* arg) {
    ThreadData* data = (ThreadData*)arg;
    int start_row = data->start_row;
    int end_row = data->end_row;
    double* A = data->A;
    double* B = data->B;
    double* C = data->C;
    int n = data->n;

    for (int i = start_row; i < end_row; i++) {
        for (int j = 0; j < n; j++) {
            for (int k = 0; k < n; k++) {
                C[i*n+j] = C[i*n+j] + A[i*n+k] * B[k*n+j];
            }
        }
    }
    pthread_exit(NULL);
}

void MY_MMult(int m, int n, int k, double *a, int lda,
              double *b, int ldb,
              double *c, int ldc)
{
    int rows_per_thread = m / NUM_THREADS;
    pthread_t threads[NUM_THREADS];
    ThreadData thread_data[NUM_THREADS];

     for (int i = 0; i < NUM_THREADS;i++) {
            thread_data[i].start_row = i * rows_per_thread;
            thread_data[i].end_row = (i + 1) * rows_per_thread;
            thread_data[i].A = a;
            thread_data[i].B = b;
            thread_data[i].C = c;
            thread_data[i].n = n;
            pthread_create(&threads[i], NULL, matrix_multiply, (void*)&thread_data[i]);
        }

        for (int i = 0; i < NUM_THREADS; i++) {
            pthread_join(threads[i], NULL);
        }
}
```

cblas

使用 BLAS 库中的 cblas_dgemm 函数，自动处理矩阵乘法的计算

```c
// cblas_gemm.c > ...
#include <cblas.h>
#include <stdio.h>

void MY_MMult(int m, int n, int k, double *a, int lda,
              double *b, int ldb,
              double *c, int ldc)
{
  double alpha = 1.0;
  double beta = 1.0;

  cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans, m, n, k, alpha, a, lda, b, ldb, beta, c, ldc);

}
```

openmp

使用 OpenMP 并行化矩阵乘法，通过添加指令，利用多核处理器加速计算

```c
// openmp_gemm.c > ...
#include "defs.h"
#include<omp.h>

void MY_MMult(int m, int n, int k, double *a, int lda,
              double *b, int ldb,
              double *c, int ldc)
{
    #pragma omp parallel for
    for (int i = 0; i < m; i++) /* Loop over the rows of C */
    {
        for (int j = 0; j < n; j++) /* Loop over the columns of C */
        {
            for (int p = 0; p < k; p++)
            {
                C(i, j) = C(i, j) + A(i, p) * B(p, j);
            }
        }
    }
}
```

lab3 问题

(1)

```
echo "version = 'naive_gemm';
```

可通过查看 makefile 中 new 或运行时输出 version 判断调用的是哪个函数

```
$(BUILD_DIR)/test_MMult.x: $(OBJS) defs.h
    $(LINKER) $(OBJS) $(LDFLAGS) -o $@
```

```
OBJS  := $(BUILD_DIR)/util.o $(BUILD_DIR)/REF_MMult.o $(BUILD_DIR)/test_MMult.o $(BUILD_DIR)/$(NEW).o
```

```
NEW  := naive_gemm
```

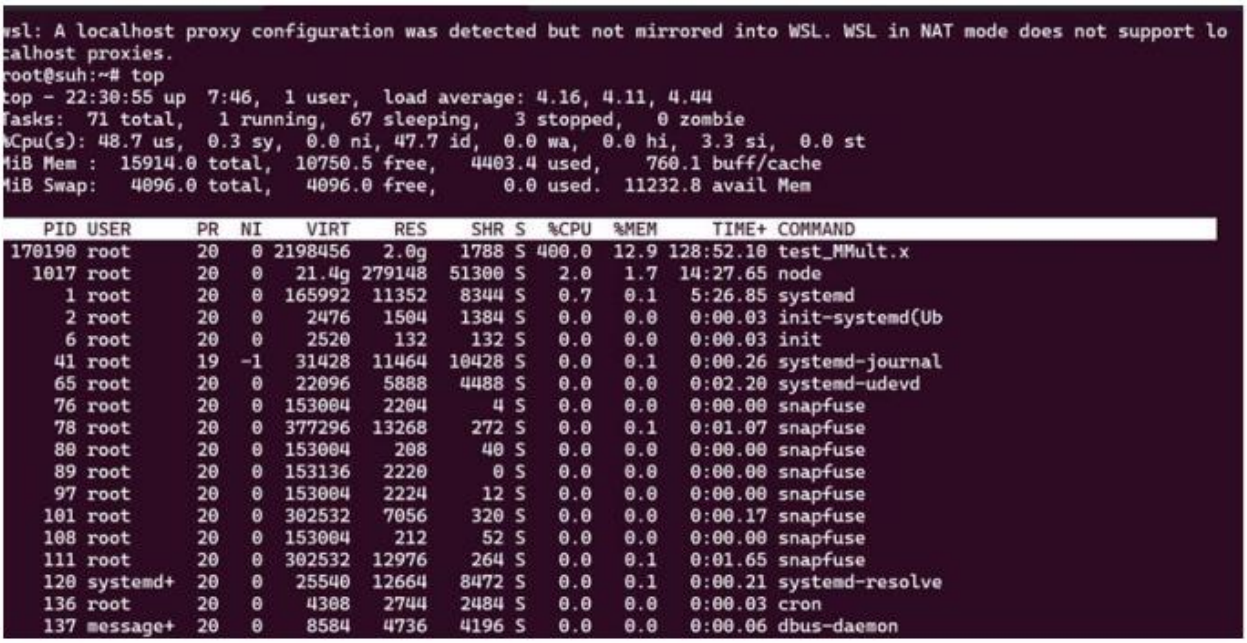
图中代码指定了使用 new 中的 MY_MMult 函数

(2)

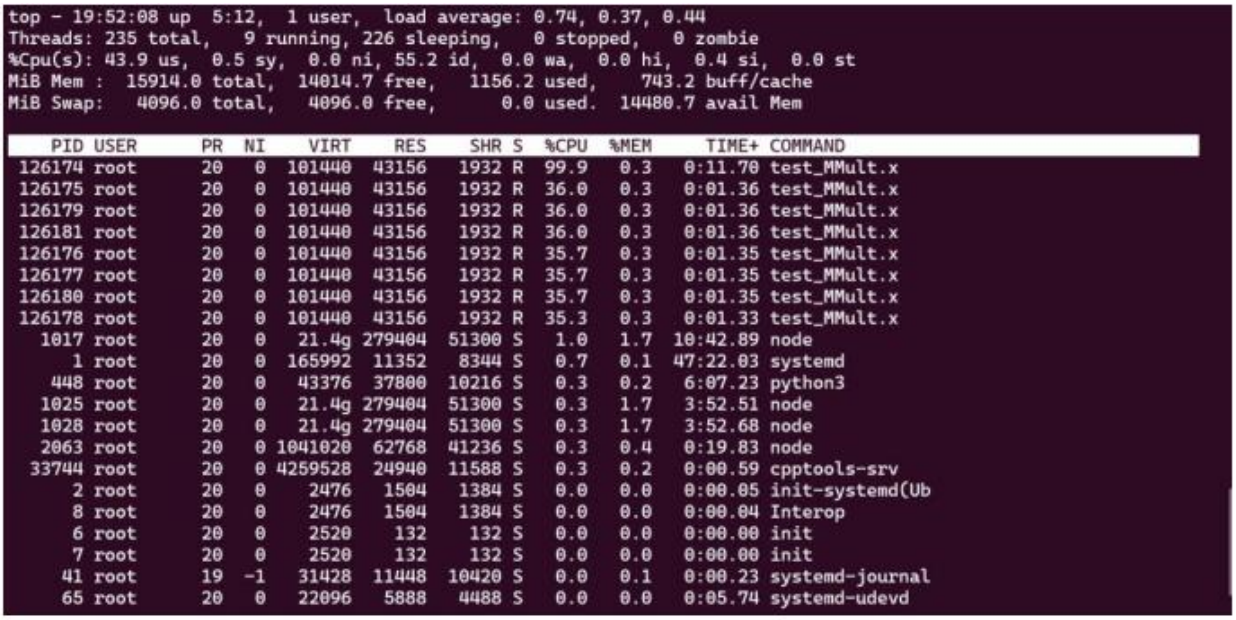
```
33      @echo "date = ``date``;" > $(DATA_DIR)/output_$(NEW).m
34      @echo "version = '$(NEW)';" >> $(DATA_DIR)/output_$(NEW).m
35      $(BUILD_DIR)/test_MMult.x >> $(DATA_DIR)/output_$(NEW).m
```

通过 makefile 中图中代码，将运行后的数据写入到_data/output_MMult0.m

thread 运行截图



openmp 运行截图



```
top - 19:52:08 up  5:12,  1 user,  load average: 0.74, 0.37, 0.44
Threads: 235 total,   9 running, 226 sleeping,   0 stopped,   0 zombie
%Cpu(s): 43.9 us,  0.5 sy,  0.0 ni, 55.2 id,  0.0 wa,  0.0 hi,  0.4 si,  0.0 st
MiB Mem :  15914.0 total,  14014.7 free,   1156.2 used,    743.2 buff/cache
MiB Swap:   4096.0 total,   4096.0 free,      0.0 used.  14480.7 avail Mem

    PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
 126174 root      20   0  101440  43156   1932 R  99.9   0.3   0:11.70 test_MMult.x
 126175 root      20   0  101440  43156   1932 R  36.0   0.3   0:01.36 test_MMult.x
 126179 root      20   0  101440  43156   1932 R  36.0   0.3   0:01.36 test_MMult.x
 126181 root      20   0  101440  43156   1932 R  36.0   0.3   0:01.36 test_MMult.x
 126176 root      20   0  101440  43156   1932 R  35.7   0.3   0:01.35 test_MMult.x
 126177 root      20   0  101440  43156   1932 R  35.7   0.3   0:01.35 test_MMult.x
 126180 root      20   0  101440  43156   1932 R  35.7   0.3   0:01.35 test_MMult.x
 126178 root      20   0  101440  43156   1932 R  35.3   0.3   0:01.33 test_MMult.x
   1017 root      20   0   21.4g 279404  51300 S   1.0   1.7  10:42.89 node
      1 root      20   0  165992  11352   8344 S   0.7   0.1  47:22.03 systemd
    448 root      20   0   43376  37800  10216 S   0.3   0.2   6:07.23 python3
   1025 root      20   0   21.4g 279404  51300 S   0.3   1.7   3:52.51 node
   1028 root      20   0   21.4g 279404  51300 S   0.3   1.7   3:52.68 node
   2063 root      20   0 1041020  62768  41236 S   0.3   0.4   0:19.83 node
  33744 root      20   0 4259528  24940  11588 S   0.3   0.2   0:00.59 cpptools-srv
      2 root      20   0    2476   1504   1384 S   0.0   0.0   0:00.05 init-systemd(Ub
      8 root      20   0    2476   1504   1384 S   0.0   0.0   0:00.04 Interop
      6 root      20   0    2520    132    132 S   0.0   0.0   0:00.00 init
      7 root      20   0    2520    132    132 S   0.0   0.0   0:00.00 init
     41 root      19  -1   31428  11448  10420 S   0.0   0.1   0:00.23 systemd-journal
     65 root      20   0   22096   5888   4488 S   0.0   0.0   0:05.74 systemd-udevd
```

```
systemd─┬─2*[agetty]
        ├─cron
        ├─dbus-daemon
        ├─init-systemd(Ub─┬─SessionLeader───Relay(940)─┬─4*[cpptools-srv───7*[{cpptools-srv}]]
        │                 │                           └─sh───sh───sh───node─┬─node───12*[{node}]
        │                 │                                                 ├─node─┬─cpptools───17*[{cpptools}+
        │                 │                                                 │      ├─node───6*[{node}]
        │                 │                                                 │      ├─pet
        │                 │                                                 │      ├─python3─┬─python3
        │                 │                                                 │      │         └─2*[{python3}]
        │                 │                                                 │      └─11*[{node}]
        │                 │                                                 ├─node─┬─bash
        │                 │                                                 │      └─11*[{node}]
        │                 │                                                 └─10*[{node}]
        │                 ├─SessionLeader───Relay(91608)───node───6*[{node}]
        │                 ├─SessionLeader───Relay(91618)───node───6*[{node}]
        │                 ├─SessionLeader───Relay(112933)───bash───test_MMult.x───7*[{test_MMult.x}]
        │                 ├─SessionLeader───Relay(114129)───bash───pstree
        │                 ├─SessionLeader───Relay(116605)───bash───top
        │                 ├─init───{init}
        │                 ├─login───bash
        │                 └─{init-systemd(Ub}
        ├─networkd-dispat
        ├─packagekitd───2*[{packagekitd}]
        ├─polkitd───2*[{polkitd}]
        ├─rsyslogd───3*[{rsyslogd}]
        ├─snapd───13*[{snapd}]
        ├─5*[snapfuse───2*[{snapfuse}]]
        ├─snapfuse───5*[{snapfuse}]
        ├─2*[snapfuse───4*[{snapfuse}]]
        ├─subiquity-serve───python3.10─┬─python3
        │                              └─5*[{python3.10}]
        ├─systemd───(sd-pam)
        ├─systemd-journal
        ├─systemd-logind
        ├─systemd-resolve
        ├─systemd-udevd
        └─unattended-upgr───{unattended-upgr}
```

lab9

任务 1

1. 问题分析

通过不同的矩阵计算顺序和数据布局，可以改变性能。考虑的计算顺序包括：ijk、ikj、jik、jki、kij 和 kji。不同的顺序会影响缓存利用率、内存访问模式等，从而影响性能。此外，向量化 SIMD、矩阵分块和数据重排等技术也是优化 GEMM 性能的关键手段。

2. 方案设计及实现方法

本实验中，我在 how-to-optimize-gemm 框架添加了以下几种方案的 MY_MMult 函数

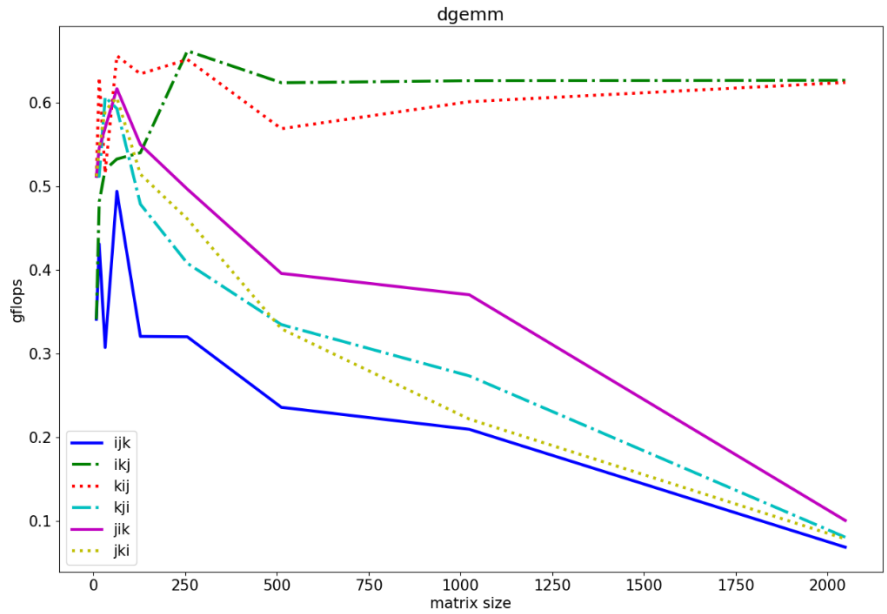
- 调整计算顺序：调换 for 循环语句顺序，编写 ijk、ikj、jki、jik、kij、kji 顺序的 GEMM 实现，并记录不同规模矩阵每种顺序的执行时间。
- 向量化 SIMD：利用 SIMD 指令集进行计算（用了 avx2），进行了向量化。
- 矩阵分块：由于 B 不是内存连续的，程序就要读取多次 B 矩阵的数据，这样数据存取将成为整个程序 gflops 上升的瓶颈。故将原始矩阵划分为 64x64 的小矩阵，分别计算后再合并结果，优化缓存使用，减少内存访问延迟。
- 数据重排：在分块后重新排列 A、B 矩阵数据，确保数据访问的连续性。

3. 结果分析

测量记录不同方案的 gflops 时间，结果如下：

GFLOPS	ijk	ikj	kij	kji	jik	jki
8	0.3413333	0.3413333	0.5120000	0.5120000	0.5120000	0.5120000
16	0.4311579	0.4818824	0.6301538	0.5120000	0.5461333	0.5461333
32	0.3076808	0.5201270	0.5160315	0.6068148	0.5698783	0.5904144
64	0.4941451	0.5328130	0.6561802	0.5930860	0.6168094	0.6061133
128	0.3208371	0.5402942	0.6345392	0.4789112	0.5500727	0.5147017
256	0.3203562	0.6620059	0.6517322	0.4081899	0.4968009	0.4612103
512	0.2361149	0.6239971	0.5690411	0.3349251	0.3961106	0.3299095
1024	0.2098076	0.6263835	0.6013171	0.2736344	0.3705864	0.2220228
2048	0.06895994	0.6267700	0.6242805	0.08090880	0.1009096	0.07886396

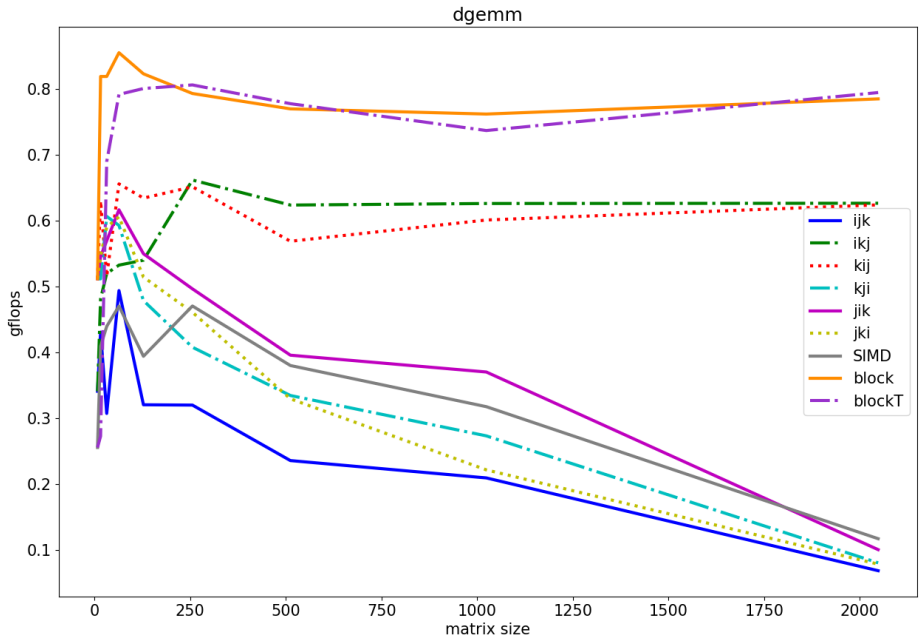
使用 python 将以上结果转化为可视化图像得：



可见当矩阵规模小的时候，波动较大且各顺序之间无较明显性能差异，当矩阵规模增大，差异明显，性能从好到差依次为 ikj、kij、jik、kji、jki、ijk。且可从图中看出矩阵规模变大时，ikj、kij 顺序性能基本保持稳定，而其它则 gflops 明显变小。

GFLOPS	SIMD	分块	数据重排
8	0.2560000	0.5120000	0.2560000
16	0.4096000	0.8192000	0.2730667
32	0.4398389	0.8192000	0.6898526
64	0.4706355	0.8552822	0.7919758
128	0.3944239	0.8232196	0.8008982
256	0.4707808	0.7933428	0.8063449
512	0.3804336	0.7700253	0.7779137
1024	0.3179459	0.7621390	0.7371289
2048	0.1175286	0.7851604	0.7947936

使用 python 将以上所有方案结果转化为可视化图像得:



由图像可知，SIMD 向量化优化方法有一定效果，但在矩阵规模大时仍然会变差。而矩阵分块和分块后数据重排两种方案性能与其他方法相比均有明显优化，且在规模增大时仍能保持较高性能，而在规模小时，数据重排的 gflops 比未重排的还小，规模变大时两者十分相近。

改进：

由于没有重排和重排的结果在已测矩阵规模中较接近，所以在答辩之后多增了几组测试，以下为测试结果：

GFLOPS	分块	数据重排
4096	0.4631622	0.7312742
5120	0.6954469	0.7464754
6144	0.5049016	0.7326933
7168	0.6841628	0.7357336
8192	0.4178990	0.7409358

可以看到在这几组测试中，数据重排过后的很明显比没重排的更佳，说明在矩阵规模足够大的时候重排可以带来更好的效果。

任务 2

1. 问题分析

CPU 实现可以通过多种优化方法（例如任务 1）提高性能，而 GPU 由于其高度并行的计算架构，能够提供更优的性能。该任务将 GPU 的 GEMM 实现集成到 how-to-optimize-gemm 框架中，可利用 GPU 的计算能力进行优化。

2. 方案设计及实现方法

- **环境搭建**：确认安装了 nvidia 显卡驱动，安装 CUDA 工具包，确保 CUDA 环境正

常运行，配置相应的编译器和库。

- **使用 CUDA 进行 GPU 编程**：利用 NVIDIA 的 CUDA 编程模型实现 GEMM，编写 MY_MMult 函数，充分利用 GPU 的并行计算能力。
- **集成到框架**：修改原有 makefile 文件
- **编译运行**

3. 实现过程

个人电脑有一张 Intel(R) Iris(R) Xe Graphics 和一张 NVIDIA RTX A2000 Laptop GPU 的独显，故使用个人电脑。

环境搭建：

使用 nividia-smi 查看是否已安装 nvidia 驱动，下图输出结果可知已安装

```
root@suh:~# nvidia-smi
Sat Oct  5 16:34:28 2024
+---------------------------------------------------------------------------------------+
| NVIDIA-SMI 535.86.10              Driver Version: 537.58       CUDA Version: 12.2     |
|-----------------------------------------+----------------------+----------------------+
| GPU  Name                 Persistence-M | Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp   Perf          Pwr:Usage/Cap |         Memory-Usage | GPU-Util  Compute M. |
|                                         |                      |               MIG M. |
|=========================================+======================+======================|
|   0  NVIDIA RTX A2000 Laptop GPU    On  | 00000000:F3:00.0 Off |                  N/A |
| N/A   41C    P8               4W /  33W |     81MiB /  4096MiB |      3%      Default |
|                                         |                      |                  N/A |
+-----------------------------------------+----------------------+----------------------+

+---------------------------------------------------------------------------------------+
| Processes:                                                                            |
|  GPU   GI   CI        PID   Type   Process name                            GPU Memory |
|        ID   ID                                                             Usage      |
|=======================================================================================|
|  No running processes found                                                           |
+---------------------------------------------------------------------------------------+
```

在官网安装支持的 cuda 版本的 toolkit，再将 CUDA 的路径添加到系统的环境变量中。由以下输出可知搭建完成

```
root@suh:~# nvcc -V
nvcc: NVIDIA (R) Cuda compiler driver
Copyright (c) 2005-2021 NVIDIA Corporation
Built on Thu_Nov_18_09:45:30_PST_2021
Cuda compilation tools, release 11.5, V11.5.119
Build cuda_11.5.r11.5/compiler.30672275_0
```

核心代码实现过程及代码截图：

① 使用 cudaMalloc 在 GPU 设备上为 a、b、c 分配内存。

② 进行数据传输，使用 cudaMemcpy 将输入矩阵 a、b、c 复制到 GPU 上相应的内存位置。

③ 使用 cublasDgemm 函数执行矩阵乘法。

④ 通过再次调用 cudaMemcpy 将计算结果从 GPU 内存的矩阵 c 复制回 CPU 内存。

⑤ 用 cudaFree 释放内存，同时，调用 cublasDestroy 来销毁 cuBLAS 句柄，释放相关资源。

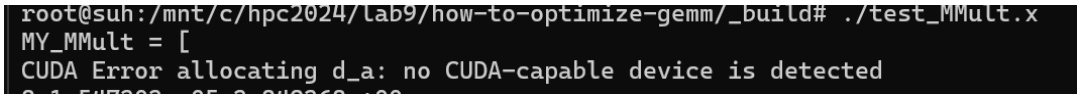
```
gpu_gemm.cu > MY_MMult(int, int, int, double *, int, double *, int, double *, int)
#include <stdio.h>
#include <stdlib.h>
#include <cuda_runtime.h>
#include <cublas_v2.h>
#include "defs.h"

extern "C" void MY_MMult(int m, int n, int k, double *a, int lda,
                         double *b, int ldb,
                         double *c, int ldc)
{
    double *d_a, *d_b, *d_c;

    cudaMalloc((void**)&d_a, lda * m * sizeof(double));
    cudaMalloc((void**)&d_b, ldb * k * sizeof(double));
    cudaMalloc((void**)&d_c, ldc * m * sizeof(double));

    cudaMemcpy(d_a, a, lda * k * sizeof(double), cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, b, ldb * n * sizeof(double), cudaMemcpyHostToDevice);
    cudaMemcpy(d_c, c, ldb * n * sizeof(double), cudaMemcpyHostToDevice);

    cublasHandle_t handle;
    cublasCreate(&handle);

    double alpha = 1.0;
    double beta = 1.0;

    cublasDgemm(handle, CUBLAS_OP_N, CUBLAS_OP_N,
                m, n, k,
                &alpha,
                d_a, lda,
                d_b, ldb,
                &beta,
                d_c, ldc);

    cudaMemcpy(c, d_c, ldc * n * sizeof(double), cudaMemcpyDeviceToHost);

    cudaFree(d_a);
    cudaFree(d_b);
    cudaFree(d_c);
    cublasDestroy(handle);
}
```

编译运行方法：

修改原有框架的 makefile 文件，使用 NVCC、增加-lcublas 等等（提交的代码中有注释标出修改部分）

遇到的问题:

编译通过，但运行时无法使用 GPU，显示 no CUDA-capable device is detected



```
root@suh:/mnt/c/hpc2024/lab9/how-to-optimize-gemm/_build# ./test_MMult.x
MY_MMult = [
CUDA Error allocating d_a: no CUDA-capable device is detected
```

上网搜集资料后尝试多种方法（修改 nvidia 控制面板全局、关掉 secure boot、使用 sudo prime-select nvidia、安装更低版本的 CUDA toolkit 等等）都无法正常运行。