GPU Acceleration with CUDA

项目编号：project5

学　　号：	**12210158**
姓　　名：	吴同馨
指导老师：	于仕琪
完成时间：	2024-06-01

目 录

第1章 项目要求分析

1.1 概述

根据项目作业要求“Project5.pdf”文件，本项目的要求是用 CUDA 编写程序，比较 CPU 和 GPU 下矩阵乘法的速度。

1.2 项目要求依据

Project5.pdf 文档。

1.3 开发语言要求

C 语言。

1.4 实现功能要求

1. 使用 CUDA 实现 B = a * A + b；
2. 使用 cuBlas 实现 B = a * A + b；
3. 使用 openBlas 实现 B = a * A + b；
4. 记录上述方法的性能并进行比较；
5. 实现关于 GPU 的有趣功能。

1.5 项目提交内容

1. 源代码: .cu, .h 文件格式
2. report.pdf 文件格式

1.6 项目提交时间

截止时间：2024 年 6 月 2 日 23:59

第2章 项目要点分析

2.1 概述

本项目的主要目标是比较 CPU 和 GPU 下矩阵运算的速度。

一、首先设定优化标准。在矩阵乘法计算中， project3 中实现了在 CPU 中使用 openBlas 等矩阵计算方法的测定，在本项目中可以以此为标准。

二、计算从硬件方面入手，使用 GPU 进行运算。本项目使用 NVIDIA GPU 进行矩阵运算，测试和比较 GPU、CPU 的性能。

综上所述并结合作业要求中提到的方法，矩阵计算的基本思路如下：

- CUDA 的使用；
- cuBlas 的使用；
- openBlas 的使用；

2.2 CUDA 的学习

GPU 可以很好的提高矩阵运算的速度，CUDA 是 NVIDIA 推出的一个并行计算平台和编程模型，它允许开发者使用 C/C++编写程序，并在 NVIDIA 的 GPU 进行通用计算。

根据查找的有关 CUDA 的资料，我了解了 CUDA 相关知识，如下：

- 组成：CUDA 包含 CUDA 指令集架构（ISA）以及 GPU 内部的并行计算引擎。
- CUDA 在以下三个方面有很好的优化：

内存优化：减少全局内存访问，使用共享内存。

计算优化：使用分块或流水线技术减少等待时间；使用并行规约算法加速归约操作；利用并行算法和技术。

线程和块管理：优化线程块规模，确保 SM 的活跃状态；负载均衡，确保所有 GPU 上的线程都得到充分利用。

- CUDA 主要应用场景：

深度学习与机器学习：CUDA 特别适合于深度学习应用，因为深度学习模型训练涉

及大量的矩阵运算和数据并行处理。

科学计算：如物理、化学、生物学等领域的模拟和计算。

图像与视频处理：加速图像处理任务，如实时视频转换和裸眼立体成像技术。

- cuBlas 与 CUDA 的关系

cuBLAS是CUDA平台上的一个数学库，专门用于执行基本线性代数子程序（BLAS，Basic Linear Algebra Subprograms）中的操作，如矩阵乘法、向量点积等。

它提供了一系列高性能的 BLAS 函数，这些函数经过优化，可以充分利用 GPU 的并行计算能力。cuBLAS 支持单精度、双精度和复数的线性代数运算，并且是 NVIDIA GPU 上执行这些运算的首选库。

关于 CUDA 的使用方法详见附录 2。

第3章 性能测试约定

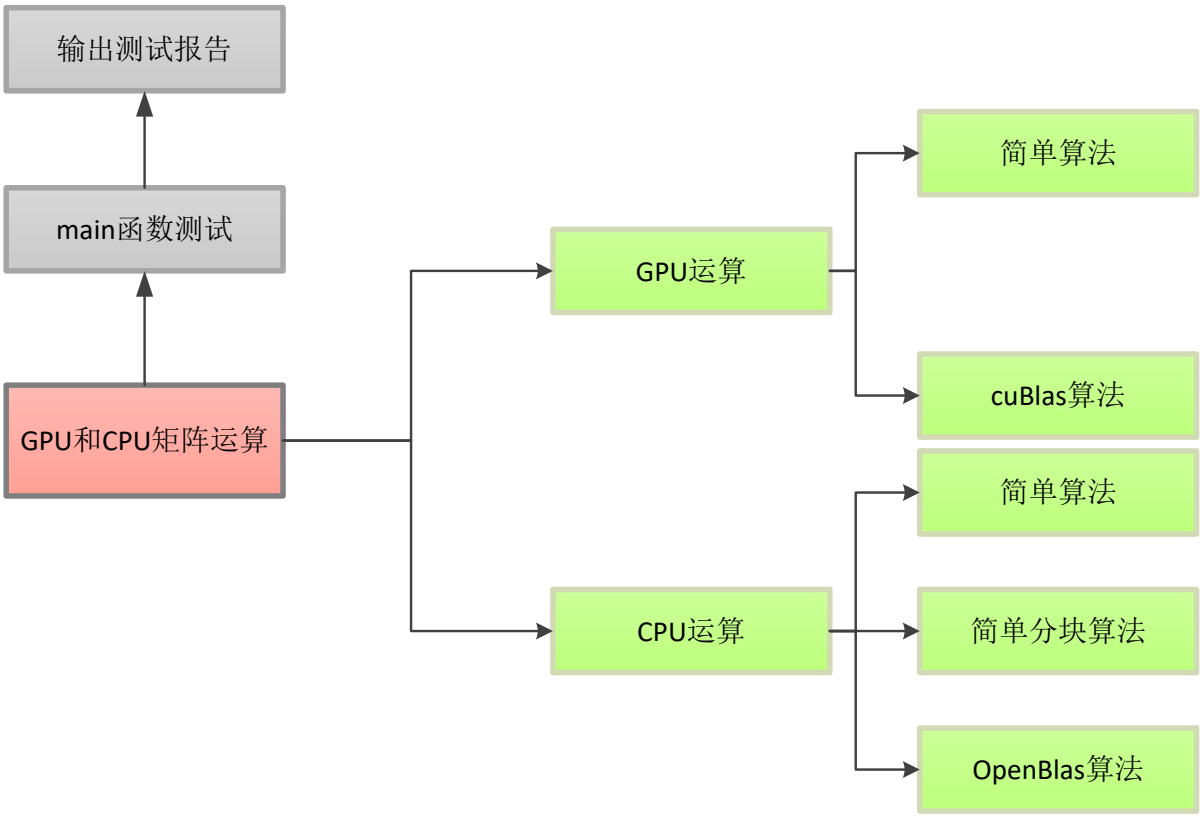
No	约定	内容
1	矩阵类型	正方形
2	矩阵大小	4096
3	矩阵值	0.0-2.0
4	单次测试时间单位	毫秒 ms
5	测时起点	矩阵运算前
6	测时终点	矩阵运算后
7	机器	学校远端服务器
8	单次测试记录	矩阵大小、用时
9	算法速度比较标准	均指 4096*4096 矩阵大小
10	矩阵运算	B 矩阵 = a 标量 * A 矩阵 + b 标量

第4章 开发工具

项目	内容	参照
操作系统	ubuntu	-
开发语言	C/C++	-
集成开发工具	vi + g++ + nvcc + gdb	-
项目构建工具	cmake	-
CUDA	GPU 计算	参考附录
cuBlas	cuBlas 库	参考附录
OpenBlas	OpenBlas 库	-

第5章 程序结构与项目构建

5.1 程序逻辑结构



程序结构图

程序结构分为三部分：使用 CPU 进行矩阵运算的 3 种方法和使用 GPU 进行矩阵运算的 2 种方法；关于 GPU 的相关探索。

5.2 程序目录结构

以下是本项目的目录结构，采用 cmake 进行构建管理。其中 src 目录下的程序将被编译成静态库 libmm.a，test 目录下 benchtest.cpp 调用 libmm.a 库中各个计算函数。

```
project05
    |---01c                    <!-- C/C++程序 -->
        |---bin                <!-- 执行程序被 cmake 拷贝至本目录 -->
        |---build              <!-- 编译 CPU 矩阵计算结果 -->
        |---src
            |--- CMakeLists.txt        <!-- 编译 openBlas.c -->
            |--- cuBlas.cu             <!—GPU 矩阵计算 -->
            |--- openBlas.c            <!—CPU 矩阵计算 -->
            |--- lowPass.cu            <!—图像模糊(黑白) -->
            |--- lowPic.cu            <!—图像模糊(彩色)-->
```

5.3 CMakeLists.txt 文件

- src 子目录 CMakeLists.txt

```
# 查找 OpenBLAS 库和头文件
find_library(OPENBLAS_LIBRARY NAMES libopenblas.so HINTS /usr/lib/x86_64-
linux-gnu)
find_path(OPENBLAS_INCLUDE_DIR NAMES cblas.h HINTS /usr/include)
find_package(OpenMP REQUIRED)
find_package(OpenCL REQUIRED)

# 包含 OpenBLAS 的头文件
include_directories(${OPENBLAS_INCLUDE_DIR})
# 添加头文件搜索路径
include_directories(${CMAKE_CURRENT_SOURCE_DIR})
# 添加源文件列表，并编译为库
set(SRC matrix_openBlas.c
    matrixMul.c
    matrixMul_SIMD.c
    #matrixMul_ARM.c
    matrixMul_openMP.c
    matrixMul_threads.c
    matrixMul_opencl.c
    )
set(SRC_H matrixMul.h matrixMul_opencl.h)
# 设置库的名字
add_library(mm STATIC ${SRC} ${SRC_H})
# 链接 OpenBLAS 库
```

```
target_link_libraries(mm ${OPENBLAS_LIBRARY})
target_link_libraries(mm  OpenMP::OpenMP_C)
```

第6章 源代码阐述

6.1 概述

本章记述源代码、测试数据以及说明。

6.2 常用宏定义

- 源代码

1. 运行时间测量

```
#define TIME_START gettimeofday(&t_start, NULL);
#define TIME_END(name) gettimeofday(&t_end, NULL); \
                elapsedTime = (t_end.tv_sec - t_start.tv_sec) * 1000.0;   \
                elapsedTime += (t_end.tv_usec - t_start.tv_usec) / 1000.0;
\
                printf(#name " Time = %f ms.\n", elapsedTime);
```

使用该宏进行每次矩阵计算时间的测量，减少重复代码。

2. 初始化矩阵

```
#define FUNC_INIT \
    if (A == NULL || A->data == NULL) \
        return NULL; \
    Matrix *B = matrix_malloc(A->rows, A->cols);\
        if (A->rows != B->rows && A->cols != B->cols) \
        return NULL; \
    if (B == NULL) \
        return NULL;
```

每个矩阵计算函数都用到了这个宏定义进行初始化，减少重复代码。

6.3 CUDA 实现

6.3.1 矩阵定义

- 源代码

```
typedef struct Matrix{
    size_t rows;
    size_t cols;
    float * data; // CPU memory
```

```
    float * data_device; //GPU mememory
} Matrix;
```

- 说明

由于数据需要通过 cuda 从 CPU 传到 GPU，因此定义了 data 和 data_device 分别用于 CPU 和 GPU，使用 cudaMemcpy 从 CPU 中获得值传递到 GPU 后再在 GPU 上进行运算。

6.3.2 opGPU 函数

```
__global__ void opKernel(const float * input1, const float input2, const
float input3, float * output, size_t len){
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    if(i < len){
        output[i] = input2 * input1[i] + input3;
    }
}

Matrix* opGPU(const Matrix * A,const float a,const float b){
    FUNC_INIT
    cudaError_t ecode = cudaSuccess;
    size_t len = A->rows * A->cols;

    cudaMemcpy(A->data_device, A->data, sizeof(float)*len,
cudaMemcpyHostToDevice);//将 CPU 中数据复制到 GPU
    opKernel<<<(len + 255) / 256, 256>>>(A->data_device, a, b,
B->data_device, len);//调用 opKernel

    if ((ecode = cudaGetLastError()) != cudaSuccess){
        fprintf(stderr, "CUDA Error: %s\n", cudaGetErrorString(ecode));
        return NULL;
    }
    cudaMemcpy(B->data, B->data_device, sizeof(float)*len,
cudaMemcpyDeviceToHost);

    return B;
}
```

- 说明

B = a * A + b 是将 A 矩阵的每个元素都乘以 a 再加上 b，因此在使用 CUDA 的 kernal 运算时它的时间复杂度可以优化成 O(n)。在 CPU 上运行时我使用 opCPU 函数和它保持一致进行比较。

6.3.3 cuBlas 函数

```
Matrix * cuBlas(const Matrix* A,const float a,const float b) {
    FUNC_INIT
    cublasHandle_t handle;
    cublasCreate(&handle);
    Matrix* ma = matrix_malloc(A->rows,A->cols);
    iniMatrix(ma,a);
    // 将矩阵 A 和 B 从主机传输到设备
    cudaMemcpy(A->data_device, A->data, A->rows * A->cols * sizeof(float),
cudaMemcpyHostToDevice);
    cudaMemcpy(ma->data_device, ma->data, ma->rows * ma->cols *
sizeof(float), cudaMemcpyHostToDevice);

    const float alpha = 1.0f;//不对结果进行缩放
    const float beta = 0.0f;

    cublasStatus_t status = cublasSgemm(handle, CUBLAS_OP_N, CUBLAS_OP_N,
              A->rows, A->cols, A->cols, &alpha,
              A->data_device, A->rows, ma->data_device, ma->cols, &beta,
              B->data_device, B->rows);
    if (status != CUBLAS_STATUS_SUCCESS) {
        printf("CUBLAS error: %d\n", status);
        cublasDestroy(handle);
        matrix_free(ma);
        return NULL;
    }

    Matrix* mb = matrix_malloc(B->rows,B->cols);
    setMatrix(mb, b);
    cudaMemcpy(mb->data_device, mb->data, mb->rows * mb->cols *
sizeof(float), cudaMemcpyHostToDevice);

    size_t len = A->rows * A->cols;
    addKernel<<<(len+255)/256, 256>>>(A->data_device, mb->data_device,
B->data_device, len);
    // 将结果从设备传输回主机
    cudaMemcpy(B->data, B->data_device, B->rows * B->cols * sizeof(float),
cudaMemcpyDeviceToHost);

    cublasDestroy(handle);
    matrix_free(ma);
    matrix_free(mb);
    return B;
```

```
}
```

- 说明

由于 cuBlas 只支持矩阵乘法的计算，因此我在计算时将 a 标量转换成对角矩阵，b 标量转换成元素全为 b 的矩阵进行计算。相比于使用 CUDA 的 kernal，这种计算方法在计算 B = a * A + b 时增加了耗时。为了更准确地对比运行时间，我在 opCPU2 函数里使用分块矩阵乘法，在 openBlas 函数里也采用了这种运算方式以此比较运行时间。

6.4 CPU 实现

- 源代码（部分）

```
Matrix * openBlas(const Matrix * A,const float a,const float b) {
    FUNC_INIT
    Matrix* ma = matrix_malloc(A->rows,A->cols);
    iniMatrix(ma,a);
    matmul_plain_bk(A,ma);
    cblas_sgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans,
                A->cols,ma->rows, B->rows, 1.0f,
                A->data, B->rows, ma->data, B->rows,
                0.0f, B->data,B->rows);

    Matrix* mb = matrix_malloc(A->rows,A->cols);
    setMatrix(mb,b);

    for(size_t i = 0;i < A->rows * A->cols;i++) {
        B->data[i] += mb->data[i];
    }
    return B;
}
```

- 说明

OpenBlas 同样使用矩阵乘法对 B = a * A + b 进行计算，可以作为一个测量标准。

第7章 测时结果对比与分析

7.1 原始测试结果

- 使用 CPU 运算的结果

```
opCPU Time = 61.281000 ms.
  Result = [1.0, 0.9, 1.2, ..., 0.4]
opCPU2 Time = 12162.327000 ms.
  Result = [0.4, 0.7, 1.0, ..., 1.0]
openBlas Time = 12141.641000 ms.
  Result = [0.4, 0.4, 0.4, ..., 0.4]
```

- 使用 GPU 运算的结果

```
You have 4 cuda devices.
You are using device 2.
opGPU Time = 56.636000 ms.
  Result = [0.5, 0.5, 0.6, ..., 1.0]
cuBlas Time = 202.889000 ms.
  Result = [0.9, 1.0, 0.4, ..., 0.8]
```

- 使用 openCL 运算的结果（GPU：Intel 集成显卡 Iris(R) Xe Graphics）

这里使用了 project3 中我实现的 openCL 方法运行出的结果，这里不再赘述相同代码，下面是矩阵大小 4096 时的运行结果：

```
BM_OpenCL         4096       23329.358000 ms
```

可以看出使用远端服务器的英伟达 GPU 比我的英特尔笔记本 GPU 快很多，快约 115 倍。

7.2 矩阵计算耗时对比表格

benchmark 数据:	单位 ms	机器：学校远端服务器
被测函数/矩阵尺寸	4096	备注
opCPU	61.28	普通算法
opCPU2	12,162.33	分块矩阵乘法
openBlas	12,141.64	openblas
opGPU	56.64	CUDA
cuBlas	202.89	cuBlas

- 说明

opCPU 函数使用简单计算方法计算 B = a * A + b，时间复杂度为 O(n)；

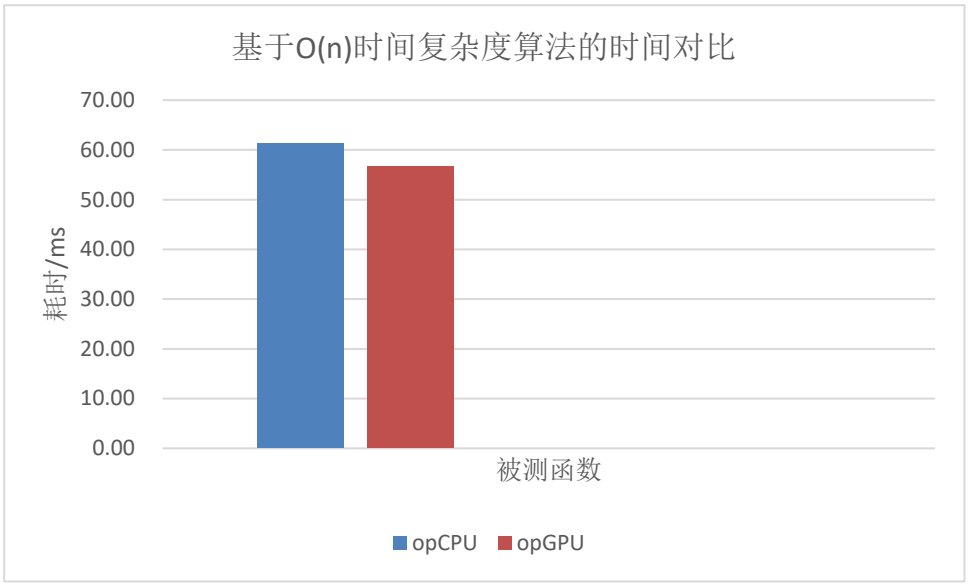
opGPU2 函数使用分块矩阵乘法和矩阵加法计算 B = a * A + b；

openBlas 函数使用 openBlas 库进行矩阵乘法计算 B = a * A + b；

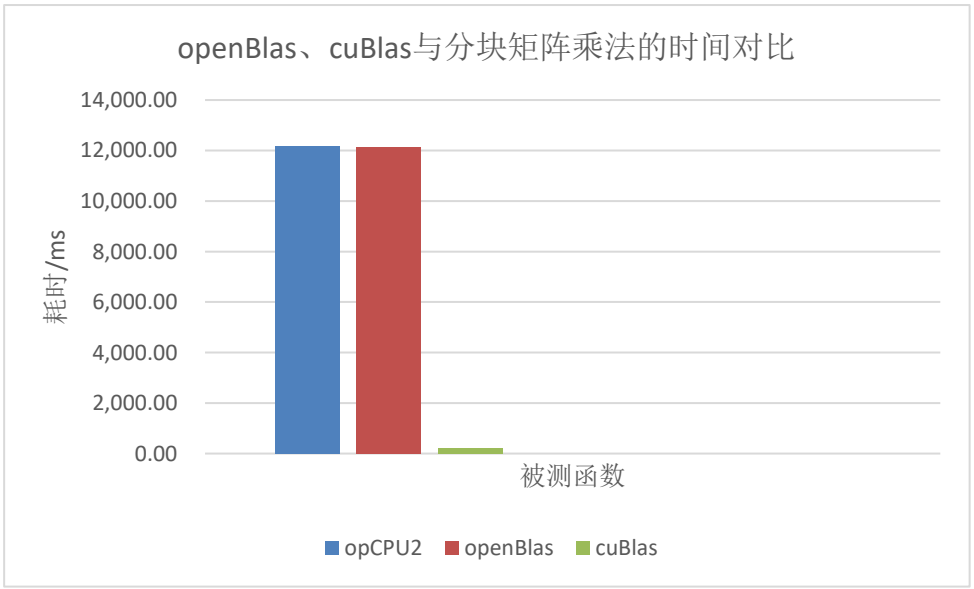
opGPU 函数使用简单计算方法计算 B = a * A + b；

cuBlas 函数使用 cuBlas 库行矩阵乘法计算和 CUDA 的 kernal 进行加法计算 B = a * A + b；

7.3 耗时对比柱状图



图一



图二

- 说明

图一是分别在 CPU 和 GPU 上使用同一种计算方法（简单线性计算）得到的对比数据；

图二是分别在 CPU 和 GPU 上使用矩阵乘法和矩阵加法进行计算得到的对比数据。

7.4 对比表格和对比柱状图的基本解读

1. 由上图图一可以看出由于时间复杂度较小，因此大小为 4096 的矩阵运算在 CPU 和 GPU 上差别不大；
2. 由上图图二可以看出使用矩阵乘法这种时间复杂度较高的运算时在较大矩阵时 GPU 计算速度比 CPU 下快很多。我认为原因有：

并行处理能力：GPU 通过执行成千上万的小型、简单的操作（如浮点运算），可以显著加速矩阵乘法等并行任务。对于矩阵乘法，GPU 可以将矩阵分割成小块或子矩阵，并将这些小块分配给不同的处理核心进行并行计算。

内存访问优化：GPU 具有多级内存，包括全局内存和共享内存等，访问速度和容量各不相同。通过将数据加载到更快的内存（如共享内存）中，GPU 可以减少访问全局内存的次数，进一步提高性能。这种内存访问的优化对于矩阵乘法这类需要大量数据访问的操作尤为重要。

计算密集型的优化：矩阵乘法是一个计算密集型的任务，需要大量的浮点运算。GPU 的架构特别适合于这种类型的任务，因为它们可以并行执行大量的简单操作。相比之下，CPU 的架构更适用于执行复杂的逻辑和分支操作。

3. CPU 下的分块运算和 openBlas 此时运行结果相差不大，说明分块算法优化结果较好。

第8章 选做 — 用 CUDA 实现图片模糊滤镜

8.1 概述

根据项目要求 3，我利用 CUDA 仿照 photoshop 做了一个图片模糊滤镜。

图片模糊滤镜需要进行傅里叶变换，因为傅里叶变换能够实现低通滤波并使图片变模糊的原因在于其能够将图像从空域转换到频域，并在频域中去除高频成分以保留低频成分，最终通过反傅里叶变换得到滤波后的图像。这种处理方式减少了图像中的细节和噪点，使图像变得更为平滑和模糊。

比如人脸边缘和背景间的波变化较快，属于高频波，如果过滤掉边缘的高频波，那么图片的人脸边缘和背景之间就会显得模糊。

傅里叶变换的原理其实很复杂，但是 CUDA 的 cufft 库可以直接实现傅里叶正反变换，因此可以直接使用，降低了计算复杂程度。具体的傅里叶变换还需要以后慢慢深入学习。

8.2 编译指令

```
nvcc -c -o low.o lowPass.cu -I/usr/include/opencv4

g++ -o l low.o -L/usr/local/cuda/include -L/usr/local/cuda/lib -
L/usr/local/cuda/lib64 -lopencv_core -lopencv_imgproc -lopencv_imgcodecs -
lcufft -lcuda -lcudadevrt -lcudart
```

编译需要根据不同机器上 openCV 和 CUDA 的库和头文件安装路径来决定。这里是在学校服务器上运行。

8.3 源代码阐述

8.3.1 模糊成黑白图片

- 实现滤波

```
using namespace cv;

__global__ void lowPass(cufftComplex *data, float d0){
    uint x = blockIdx.x * blockDim.x + threadIdx.x;
```

```
    uint y = blockIdx.y * blockDim.y + threadIdx.y;
    uint off = x * N + y;
    float dx = fabs((float)x - N / 2);
    float dy = fabs((float)y - N / 2);
    float d = sqrt(dx * dx + dy * dy);
    float filter;
    filter = 1 / (1 + (d / d0)*(d / d0));
    data[off].x *= filter;
    data[off].y *= filter;
}
```

lowPass 函数实现了一个低通滤波器作用到二维复数数据上。

- main 函数

```
    Mat img = imread("1.jpg", IMREAD_GRAYSCALE);
    img.convertTo(img, CV_32F);
```

读取灰度图像便于简单处理，方便编写程序。

```
cufftComplex* data;
    cufftHandle cu;
    cudaMallocManaged((void **)&data, N*N*sizeof(cufftComplex));
    cufftPlan2d(&cu, N, N, CUFFT_C2C);//使用 cufftPlan2d 初始化,两个 N 代表分别图
像的长宽
```

定义一个 cufft 句柄，然后使用 cufftPlan2d 初始化，两个 N 代表分别图像的长宽，使用复数 CUFFT_C2C。

```
    cufftComplex* data;
    cufftHandle cu;
    cudaMallocManaged((void **)&data, N*N*sizeof(cufftComplex));
    cufftPlan2d(&cu, N, N, CUFFT_C2C);//使用 cufftPlan2d 初始化,两个 N 代表分别图
像的长宽
    for(int i = 0;i < N;i++){
        for(int j = 0;j < N;j++){
            float shift;
            if ((i + j) % 2 == 0) {
                shift = 1;
            } else {
                shift = -1;
            }
            data[i*N+j].x = img.at<float>(i,j) * shift;
        }
    }
    cufftExecC2C(cu, data, data, CUFFT_FORWARD);
    cudaDeviceSynchronize();
```

cufftComplex 为库中定义的复数类型结构体，其中成员 x 和 y 分别代表实部和虚部。

cufftExecC2C 代表执行傅里叶变换，CUFFT_FORWARD 和 CUFFT_INVERSE 分别表示正变换和反变换。

```
    cudaFree(data);
    cufftDestroy(cu);
```

最后是释放。

8.3.2 模糊成彩色图片

- Main 函数

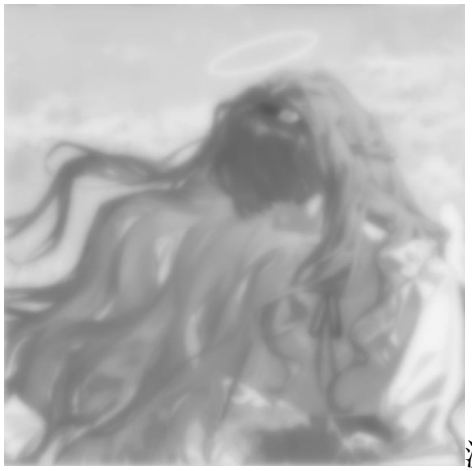
生成彩色图片的方法与生成黑白图片方法类似，只是分别计算了 3 个通道，最后再将它们合并得到结果。以下是循环中操作：

```
for (int c = 0; c < 3; ++c) {
        cudaMallocManaged((void **)&data_gpu[c], N * N *
sizeof(cufftComplex));
        cufftPlan2d(&plan[c], N, N, CUFFT_C2C);

        result_planes[c] = Mat::zeros(N, N, CV_32F);

        for (int i = 0; i < N; ++i) {

            for (int j = 0; j < N; ++j) {
                float shift = ((i + j) % 2 == 0) ? 1.0f : -1.0f;
                data_gpu[c][i * N + j].x = img_planes[c].at<float>(i, j) *
shift;
            }
        }
        cufftExecC2C(plan[c], data_gpu[c], data_gpu[c], CUFFT_FORWARD);
        cudaDeviceSynchronize();

        dim3 grid(N / THREAD, N / THREAD);
        dim3 block(THREAD, THREAD);
        lowPass<<<grid, block>>>(data_gpu[c], 20.0f);

        cufftExecC2C(plan[c], data_gpu[c], data_gpu[c], CUFFT_INVERSE);
        cudaDeviceSynchronize();

        result_planes[c].create(N, N, CV_32F);
        for (int i = 0; i < N; ++i) {
            for (int j = 0; j < N; ++j) {
                float shift = ((i + j) % 2 == 0) ? 1.0f : -1.0f;
                result_planes[c].at<float>(i, j) = data_gpu[c][i*N+j].x / (N *
N) * shift;
```

```
            }
        }
        cufftDestroy(plan[c]);
        cudaFree(data_gpu[c]);
    }
```

8.4 运行结果展示


原始图片


滤镜模糊后图片（黑白）


滤镜模糊后图片（彩色）

第9章 项目总结

本次项目通过实现 B = a * A + b，学习了 CUDA 在 GPU 上的使用方法，认识到了计算机软件与硬件在计算效率方面的相互作用。

通过本项目的实施，在前四次 project 的基础上，学习了 C/C++以及 CUDA。

1. 巩固了 openBlas 的使用方法；
2. 学习了 CUDA 的基本编程方法；
3. 学习了 cuBlas 的基本编程方法；
4. 初步学习了利用 GPU 对图片进行渲染。

通过本项目，对 GPU 编程有了更深入的学习。

通过本项目的实施，利用 CUDA 在 GPU 上进行了矩阵乘法计算，也为今后利用 GPU 进行计算准备了一些基础知识。

总体来说，本项目中，我学习到了 NvidiaGPU 编程的有关知识，尤其是学习了 CUDA 编程，结合 Project3 中实验的 openCL 编程，在 GPU 方面我初步掌握了 2 种流行的 GPU 编程方法，这两种方法之间可以相互比较，加深了对 GPU 编程的理解，收获很大。这些工具由于时间关系我只是很粗略地运用了一下，今后有机会我会更加精细地深入学习这些工具的使用。

第10章 C/C++项目综述

本次项目是本学期C/C++的最后一个项目。本学期一共做了五个项目，这五个项目除了第1个项目外，主题都是围绕深度学习中常用的矩阵运算尤其是矩阵乘法展开。这五个项目分别是：

No	项目名称	技术要点简述
1	A Simple Calculator	采用了 flex 和 bison 进行词法语法分析，并且使用了 GMP 库
2	Simple Matrix Multiplication	对比了 C 语言和 Java 矩阵乘法的运算效率
3	Improved Matrix Multiplication	通过改进矩阵乘法的算法，学习了内存访问优化、自动向量化技术、多线程并行计算等
4	A Class to Describe a Matrix	在这个项目集中使用了C++面向对象编程的封装性、继承性、多态性，以及内存软拷贝和硬拷贝的不同
5	GPU Acceleration with CUDA	采用 cuda 进行 GPU 编程

这五个项目对我这样的初学者来说真的很难，但是通过解决这些困难的问题收获了C/C++广泛的知识，比单纯地从书本学习高效了很多。以下就C/C++的这些项目做简单的总结。

10.1 C 语言基础知识

通过项目的实施，直观地感受了C基本语法，先将在项目中印象较深的基础知识总结如下：

1. 基本数据类型；
2. 赋值语句、条件语句、循环语句；
3. 函数，包括函数定义、参数、返回值；
4. 结构体；
5. 指针，实践了一维指针和二维指针，对C语言的内存分配和释放有了直观的感受；

6. 函数指针的定义和使用，函数指针的存在使得某些编程变得非常灵活；
7. 预编译，包括#define 等。

10.2 C++的基础知识

C++的大部分语法都是基于 C 语言的，而且主要是在项目 4 中运用了 C++。C++与其说是一种编程语言，不如说是一种思想，即面向对象的思想，而面向对象的思想不仅仅是编程时候有用，它也可以指导我们用一种新的眼光看待世界。这里重点对 C++面向对象编程的特性做个总结：

1. 封装性，包括属性和方法的封装，以及构造函数和析构函数；
2. 继承性，包括了基类和派生类之间的继承关系，以及这两种类的属性和方法的继承关系，主要在项目 4 中利用 ROI 设计了一个形状的抽象类以及方形 ROI 和圆形 ROI 的派生类来进行基类和派生类的继承；
3. 多态性，对虚函数、纯虚函数的定义和重写进行了实践，主要也是在项目 4 中利用上述的基类和派生类来实践纯虚函数的技术要点；
4. 模板类，在项目 4 中大量运用了模板类，一下子让很抽象且陌生的模板类变得具体和熟悉起来。

10.3 项目构建工具 cmake

在这五个项目中，都是用了项目构建工具 cmake。由于从终端输入编译指令对于复杂的项目工程来说是不现实的，这就需要使用项目构建工具 cmake 来构建 C/C++项目。有了项目构建工具，我们可以很方便的管理我们的源代码的编译。没有其它工具我们尚且可以一点点自己编程得到结果，但没有项目构建工具，则编程难以正常进行下去，所以这里单列了项目构建工具 cmake。

10.4 单元测试(Unit Test)

测试是软件工程必不可少的环节，涉及到单元测试、性能测试、系统测试等等很多测试环节，而其中单元测试是最重要的基础性测试环节，所以在项目实施过程中，在项目 1 和项目 4 中增加了对程序的单元测试。针对 C 语言程序的单元测试采用了 Cunit 进行单元测试，针对 C++程序采用 google gtest 单元测试工具。

10.5 Benchmark 测试

性能测试是另外一个最重要的基础性测试，Benchmark 测试是重要的性能测试手段。在项目实施过程中我采用了自己内置时钟进行 Benchmark 测试，或者采用了 Google Benchmark 测试。尤其在项目 2、项目 3 和项目 4 中，集中使用了这些 Benchmark 测试工具。只有通过项目实践，才能有机会实践这些测试方法。

10.6 各种工具及库的汇总

在本学期五次项目中，为了完成每个项目设定的目标，我查阅了很多的资料，大量使用了各种工具和库，极大地拓展了我的知识面。关于这些工具和库的列表如下：

No	要点	说明	项目
1	vi + g++/gcc + gdb	ubuntu 上编辑器+编译器+调试器	1
2	flex + bison	词法分析 + 语法分析	1
3	cunit	单元测试	1
4	GMP	任意精度算术库	1
5	C/C++正则表达式库	处理正则表达式	1
6	cmake	C/C++项目构建工具	2
7	编译参数-O3	建立编译参数概念，由此学习其它参数	2
8	maven	Java 项目构建工具	2
9	valgrind	C/C++内存泄漏等错误检测	2
10	google benchmark	C/C++基准测试	2
11	JMH	Java 基准测试	2
12	指针和双指针运用	双指针在内存分配上的特点	2
13	OpenBlas	基础线性代数程序集	3
14	SIMD	多数据流处理技术,能够同时对多个数据进行操作，intel avx2 指令集	3
15	ARM SIMD	neon 选项	3
16	OpenMP	共享存储并行编程	3
17	Thread	多线程编程技术	3

No	要点	说明	项目
18	OpenCL	运用 GPU 进行计算	3
19	objdump	将执行文件反汇编成汇编程序和 gcc -S -mavx matrixMul_SIMD.c 作用类似但更简单，因为 gcc 反汇编可能要参数	3
20	OpenCV	图像识别	4
21	ROI	寻找感兴趣的图像区域	4
22	重载操作符	重载内置操作符	4
23	抽象类与具体类	代码的复用和维护	4
24	GTest	测试	4
25	CUDA	并行计算架构	5
26	cuBlas	NVIDIA 提供的一个高性能的基础线性代数子程序库	5

本学期的这些项目不仅让我学到了 C/C++的基础知识，还让我学到了很多课本上没有的综合知识。能在短短一学期收获这么多有用的知识，是老师为我们精心设计和准备这些项目的结果，感谢老师！

第11章 附录 1 项目知识要点备忘清单

No	要点	说明	项目
1	CUDA	并行计算架构	5
2	cuBlas	NVIDIA 提供的一个高性能的基础线性代数子程序库	5
3	OpenBlas	基础线性代数程序集	3

第12章 附录 2 CUDA

12.1 介绍

CUDA 里面用 Grid 和 Block 作为线程组织的组织单位，一个 Grid 可包含了 N 个 Block，一个 Block 包含 N 个 thread。

CUDA 支持英伟达显卡，根据题目要求使用 CUDA 在 GPU 上进行矩阵计算。

12.2 编译方法

本项目中我采用直接编译的方式，如下：

```
nvcc -o test cuBlas.cu -lcublas
```

12.3 使用方法

```
cublasHandle_t handle;
cublasCreate(&handle);
cublasDestroy(handle);

cudaMemcpy(mb->data_device, mb->data, mb->rows * mb->cols * sizeof(float),
cudaMemcpyHostToDevice);
cudaMemcpy(B->data, B->data_device, B->rows * B->cols * sizeof(float),
cudaMemcpyDeviceToHost);
```

本项目中主要使用了 CUDA 的句柄和它的 cudaMemcpy，将 CPU 内存中数据拷贝到 GPU 中，操作后再拷贝回 CPU。

利用 CUDA 操作 GPU 的整体流程：

CPU 分配空间给 GPU（cudaMalloc）；

CPU 复制数据给 GPU（cudaMemcpy）；

CPU 加载 kernels 给 GPU 作计算；

CPU 把 GPU 的运算结果复制回来。

12.4 cuBlas 库

cuBlas 是 CUDA 的一个组成部分，它专注于基本的线性代数运算，如矩阵乘法、矩

阵向量乘法、矩阵转置等；它通过利用 GPU 的并行计算能力，为这些操作提供高性能的实现，从而显著加速大规模数值计算和科学计算任务。

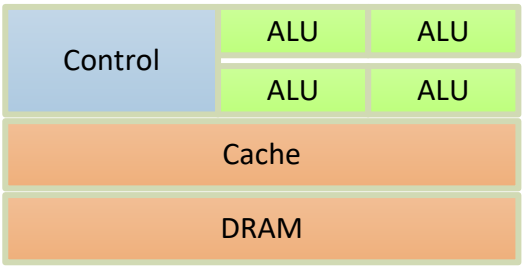
本项目中使用 cublasSgemm 函数进行矩阵乘法。

第13章 附录 3 CPU 和 GPU 的区别比较

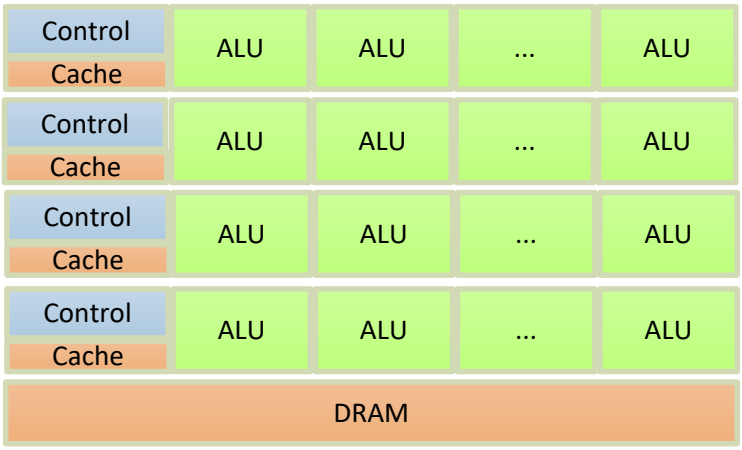
13.1 二者主要功能

CPU 更倾向于快速处理少量数据（例如算术运算：567），GPU 更擅长处理大量重复数据（例如矩阵运算：（AB）C）。因此，虽然 CPU 单次运送的时间更快，但是在处理图像处理、动漫渲染、深度学习这些需要大量重复工作负载时，GPU 优势就越显著。GPU 会延迟对性能的影响，但对于深度学习的典型任务场景，数据一般占用大块连续的内存空间，GPU 可以提供最佳的内存带宽，并且线程并行带来的延迟几乎不会造成影响。

13.2 CPU 构造



13.3 GPU 构造



13.4 二者比较

- 二者架构核心不同；
- 从上图结构可以看出，CPU 的 ALU 较复杂，但是个数较少；GPU 的计算单元相对简单，但是个数很多 CPU 的控制单元较复杂；GPU 的控制单元相对简单，分布在各计算核；
- GPU 的功能是优化数据吞吐量。它允许一次通过其内部推送尽可能多的任务，这比 CPU 一次可以处理的任务多得多。这是因为通常情况下，GPU 具有比 CPU 多得多的内核；
- CPU 是为延迟优化的，而 GPU 则是带宽优化的。CPU 更善于一次处理一项任务，而且 GPU 则可以同时处理多项任务；
- CPU 基本上是实时响应，对单任务的速度要求很高，所以就要用很多层缓存的办法来保证单任务的速度；而 GPU 往往采用的是批处理的机制，即：任务先排好队，挨个处理。
- GPU 可以即时(JIT)编译，它减少了调度并行工作负载的开销。GPU 驱动程序和运行时具有 JIT 编译功能，可以在执行之前将高级着色器代码转换为优化的设备指令，相比之下，CPU 必须坚持预编译的机器码，不能根据运行时行为自适应地重新编译。