

中山大学计算机学院本科生实验报告

(2024 学年秋季学期)

课程名称：高性能计算程序设计基础

批改人：

实验	CUDA 程序设计	专业（方向）	信息与计算科学
学号	22336049	姓名	陳日康
Email	chenih5@mail2.sysu.edu.cn	完成日期	2025 年 1 月 13 日

1. 实验内容

- 通过 CUDA 实现通用矩阵乘法（Lab1）的并行版本。
- 通过 NVIDIA 的矩阵计算函数库 CUBLAS 计算矩阵相乘，并与任务 1 的矩阵乘法进行性能比较和分析。
- 在 GPU 上实现神经网络中的卷积操作。
- 使用 im2col 方法结合任务 1 实现的 GEMM（通用矩阵乘法）实现卷积操作。
- 使用 cuDNN 提供的卷积方法进行卷积操作，并与任务 1 的矩阵乘法进行性能比较和分析。

2. 实验过程和核心代码

(1). 通过 CUDA 实现通用矩阵乘法（Lab1）的并行版本。

(i). CUDA 矩阵乘法

```
__global__ void matrixMulKernel(float* A, float* B, float* C, int N) {
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;

    if (row < N && col < N) {
        float sum = 0.0f;
        for (int k = 0; k < N; k++) {
            sum += A[row * N + k] * B[k * N + col];
        }
        C[row * N + col] = sum;
    }
}
```

CUDA 核函数 `matrixMulKernel` 用于完成矩阵乘法运算，核函数会以并行方式运行在大量线程上，每个线程负责计算结果矩阵 `C` 中的一个元素。在 CUDA 中，线程分布在网格（Grid）和线程块（Block）中，每个线程有一个唯一的全局索引，用于确定它处理的数据位置。每个线程利用自己的全局索引 `row` 和 `col` 计算 `C[row][col]`。由于线程块的维度可能与矩阵大小不完全匹配，需要检查 `row` 和 `col` 是否超出矩阵范围。如果超出，则当前线程不参与计算。

(ii). 内存管理与数据传输

```
// 分配设备内存
float *d_A, *d_B, *d_C;
cudaMalloc((void**)&d_A, size);
cudaMalloc((void**)&d_B, size);
cudaMalloc((void**)&d_C, size);

// 拷贝数据到设备
cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);
```

```

// 设置 CUDA 网格和块大小
dim3 dimBlock(TILE_SIZE, TILE_SIZE); // 32 x 32 的线程块, 512 个线程/块
dim3 dimGrid((N + TILE_SIZE - 1) / TILE_SIZE, (N + TILE_SIZE - 1) / TILE_SIZE);

// 记录时间
auto start = std::chrono::high_resolution_clock::now();

// 启动 CUDA 核函数
matrixMulKernel<<<dimGrid, dimBlock>>>(d_A, d_B, d_C, N);

// 同步设备
cudaDeviceSynchronize();

// 记录结束时间
auto end = std::chrono::high_resolution_clock::now();
std::chrono::duration<float> duration = end - start;

// 拷贝结果回主机
cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);

```

cudaMalloc 为 GPU 分配 A、B、C 的全局内存。每个矩阵的内存大小为 size。cudaMemcpy 将主机内存中的 A、B 数据拷贝到设备内存。线程块大小是 32×32 ，即每个线程块有 1024 个线程。网格大小计算，对矩阵维度进行整除（上取整）以容纳所有元素。执行核函数 matrixMulKernel，每个线程负责计算 C 中的一个元素。cudaDeviceSynchronize 确保所有设备端操作完成。cudaMemcpy 将计算结果 C 从设备内存拷贝回主机内存。

(2). 通过 NVIDIA 的矩阵计算函数库 CUBLAS 计算矩阵相乘，并与任务 1 和任务 2 的矩阵乘法进行性能比较和分析。

矩阵规模从 512 增加至 8192，并与任务 1 和任务 2 的矩阵乘法进行性能比较和分析，如果性能不如 CUBLAS，思考并文字描述可能的改进方法。

(i). cublasSgemm

```

cublasSgemm(handle, CUBLAS_OP_N, CUBLAS_OP_N, size, size, size, &alpha,
            d_A, size, d_B, size, &beta, d_C, size);

```

调用 cuBLAS 库中的 cublasSgemm 完成矩阵乘法。参数 CUBLAS_OP_N 表示矩阵不转置。使用公式 $C = \alpha \cdot A \cdot B + \beta \cdot C$ 计算结果， $\alpha = 1.0f$ ， $\beta = 0.0f$ ，即结果为普通的矩阵乘法。

(ii). 思考：

如果普通 CUDA 矩阵乘法的性能显著低于 cuBLAS，可能存在未充分利用共享内存、线程块分配不合理等问题。为了缩小与 cuBLAS 性能的差距，可以考虑以下改进方法：引入共享内存优化，将矩阵的子块加载到共享内存中，减少全局内存访问次数；优化线程分配和块划分：根据矩阵大小动态选择线程块和网格配置，确保 GPU 计算资源的高效利用。

CUDA 矩阵乘法与 cuBLAS 矩阵乘法在性能上的差距主要源于两者在底层优化和实现方式上的差异。cuBLAS 是 NVIDIA 提供的高度优化的库，它针对矩阵乘法操作（如 cublasSgemm）进行了专业的调优。因此，cuBLAS 通常能充分发挥 GPU 的计算能力，具有较高效率。而普通 CUDA 矩阵乘法代码往往难以在所有这些方面达到 cuBLAS 的优化水平。

(3). 在 GPU 上实现神经网络中的卷积操作。

通过 CUDA 实现直接卷积（滑窗法），输入从 256 增加至 4096 或者输入从 32 增加至 512。输入：Input 和 Kernel(3x3)。用直接卷积的方式对 Input 进行卷积，只需要实现 2D，height*width，通道 channel(depth)设置为 3，Kernel (Filter)大小设置为 3*3，步幅(stride)分别设置为 1，2，3，可能需要通过填充(padding)配合步幅(stride)完成 CNN 操作。实验的卷积操作不需要考虑 bias(b)，bias 设置为 0。

(i). direct_convolution_2d 核函数

```
__global__ void direct_convolution_2d(
    float* input, float* kernel, float* output,
    int input_h, int input_w, int stride, int padding, int output_h, int output_w) {

    int out_x = blockIdx.x * blockDim.x + threadIdx.x;
    int out_y = blockIdx.y * blockDim.y + threadIdx.y;

    if (out_x < output_w && out_y < output_h) {
        float result = 0.0f;

        for (int c = 0; c < 3; ++c) { // Iterate over channels
            for (int ky = 0; ky < 3; ++ky) {
                for (int kx = 0; kx < 3; ++kx) {
                    int in_x = out_x * stride + kx - padding;
                    int in_y = out_y * stride + ky - padding;

                    if (in_x >= 0 && in_x < input_w && in_y >= 0 && in_y < input_h) {
                        result += input[(c * input_h + in_y) * input_w + in_x] *
                                   kernel[(c * 3 + ky) * 3 + kx];
                    }
                }
            }
        }
        output[(out_y * output_w + out_x)] = result;
    }
}
```

函数是 CUDA 核函数，用于执行二维卷积操作。其输入包括 3 通道输入图像和 3*3 卷积核，计算后将结果存储在输出矩阵中。每个线程负责计算输出矩阵中的一个元素。通过 blockIdx 和 threadIdx，每个线程的唯一索引 out_x 和 out_y 用于定位输出矩阵中的位置。若线程负责的索引超出输出矩阵范围 (out_x >= output_w || out_y >= output_h)，直接返回，确保访问内存的合法性。使用三重嵌套循环进行卷积计算，逐个遍历每个通道、每个核元素，执行逐元素相乘并累加。通过 in_x 和 in_y 的合法性检查（即是否在输入矩阵范围内）来实现边界上的零填充。

(ii). run_convolution 运行卷积函数

```
void run_convolution(int input_h, int input_w, int stride) {
    const int channels = 3;
    const int kernel_size = 3;
    const int padding = (stride - 1) / 2; // Same padding calculation

    int output_h = (input_h + 2 * padding - kernel_size) / stride + 1;
    int output_w = (input_w + 2 * padding - kernel_size) / stride + 1;

    size_t input_size = input_h * input_w * channels * sizeof(float);
    size_t kernel_size_bytes = kernel_size * kernel_size * channels * sizeof(float);
    size_t output_size = output_h * output_w * sizeof(float);

    float* h_input = (float*)malloc(input_size);
    float* h_kernel = (float*)malloc(kernel_size_bytes);
    float* h_output = (float*)malloc(output_size);
}
```

```

// Initialize input and kernel with random values
for (int i = 0; i < input_h * input_w * channels; ++i) h_input[i] = static_cast<float>(rand() % 10 + 1);
for (int i = 0; i < kernel_size * kernel_size * channels; ++i) h_kernel[i] = static_cast<float>(rand() % 5 + 1);

float* d_input, * d_kernel, * d_output;
CHECK_CUDA_CALL(cudaMalloc(&d_input, input_size));
CHECK_CUDA_CALL(cudaMalloc(&d_kernel, kernel_size_bytes));
CHECK_CUDA_CALL(cudaMalloc(&d_output, output_size));

CHECK_CUDA_CALL(cudaMemcpy(d_input, h_input, input_size, cudaMemcpyHostToDevice));
CHECK_CUDA_CALL(cudaMemcpy(d_kernel, h_kernel, kernel_size_bytes, cudaMemcpyHostToDevice));

dim3 block_dim(16, 16);
dim3 grid_dim((output_w + block_dim.x - 1) / block_dim.x,
              (output_h + block_dim.y - 1) / block_dim.y);

auto start = std::chrono::high_resolution_clock::now(); // Start timer
direct_convolution_2d<<<grid_dim, block_dim>>>(&
    d_input, d_kernel, d_output,
    input_h, input_w, stride, padding, output_h, output_w);
CHECK_CUDA_CALL(cudaDeviceSynchronize()); // Wait for GPU to finish
auto end = std::chrono::high_resolution_clock::now(); // End timer

CHECK_CUDA_CALL(cudaMemcpy(h_output, d_output, output_size, cudaMemcpyDeviceToHost));

```

函数配置并运行卷积操作，包括内存分配、初始化、数据拷贝、核函数调用、结果打印等完整流程。根据输入矩阵尺寸、卷积核大小、步幅和填充计算输出矩阵尺寸。进行内存分配和初始化，在主机端分配输入、卷积核和输出矩阵的内存。使用随机值初始化输入和卷积核矩阵。在设备端分配输入、卷积核和输出矩阵的内存。使用 `cudaMemcpy` 将主机数据传输到设备。配置线程块和网格的维度，调用核函数执行卷积运算，并使用 `cudaDeviceSynchronize` 确保核函数执行完成。

(4). 使用 `im2col` 方法结合任务 1 实现的 GEMM（通用矩阵乘法）实现卷积操作。

输入从 256 增加至 4096 或者输入从 32 增加至 512，具体实现的过程可以参考下面的图片和参考资料。输入：Input 和 Kernel (Filter)。用 `im2col` 的方式对 Input 进行卷积，这里只需要实现 2D，`height*width`，通道 `channel(depth)` 设置为 3，Kernel (Filter) 大小设置为 3*3。实验的卷积操作不需要考虑 `bias(b)`，`bias` 设置为 0，步幅(`stride`)分别设置为 1, 2, 3。输出卷积结果和时间。

(i). `__global__ void gemmKernel`

```

__global__ void gemmKernel(const float* A, const float* B, float* C, int M, int N, int K) {
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;

    if (row < M && col < N) {
        float value = 0.0;
        for (int k = 0; k < K; ++k) {
            value += A[row * K + k] * B[k * N + col];
        }
        C[row * N + col] = value;
    }
}

```

函数在 GPU 上执行，完成矩阵乘法的核心逻辑。计算当前线程处理的元素位置 `row` 和 `col`，`blockIdx` 和 `threadIdx` 分别 CUDA 中的块索引和线程索引，用于确定线程的位置。`blockDim` 是每个块的线程数量。检查线程是否在合法范围内，确保线程不会访问超出矩阵范围的位置。累积求和完成矩阵乘法，每个线程计算矩阵 C 的一个元素，遍历矩阵 A 的当前行和矩阵 B 的当前列，完成点积计算。最后写入结果到矩阵 C 中。

(5). 使用 cuDNN 提供的卷积方法进行卷积操作，并与任务 1 的矩阵乘法进行性能比较和分析。

记录其相应 Input 的卷积时间，与自己实现的卷积操作进行比较。如果性能不如 cuDNN，用文字描述可能的改进方法。

(i). CHECK_CUDA 与 CHECK_CUDNN 宏

```
#define CHECK_CUDA(call) {\
    cudaError_t err = call;\
    if (err != cudaSuccess) {\
        std::cerr << "CUDA error at " << __FILE__ << ":" << __LINE__ << " - " << cudaGetErrorString(err) << std::endl;\
        exit(EXIT_FAILURE);\
    }\
}\

#define CHECK_CUDNN(call) {\
    cudnnStatus_t err = call;\
    if (err != CUDNN_STATUS_SUCCESS) {\
        std::cerr << "cuDNN error at " << __FILE__ << ":" << __LINE__ << " - " << cudnnGetErrorString(err) << std::endl;\
        exit(EXIT_FAILURE);\
    }\
}
```

用于检查 CUDA 和 cuDNN 函数调用是否返回错误。若发生错误，打印错误信息并终止程序。CHECK_CUDA 验证 CUDA API 调用是否成功（如 cudaMalloc、cudaMemcpy 等）。CHECK_CUDNN 验证 cuDNN API 调用是否成功（如 cudnnSetTensor4dDescriptor、cudnnConvolutionForward 等）。

(ii). cudnnCreate 系列函数

```
CHECK_CUDNN(cudnnCreate(&cudnn));\
CHECK_CUDNN(cudnnCreateTensorDescriptor(&inputDesc));\
CHECK_CUDNN(cudnnCreateTensorDescriptor(&outputDesc));\
CHECK_CUDNN(cudnnCreateFilterDescriptor(&filterDesc));\
CHECK_CUDNN(cudnnCreateConvolutionDescriptor(&convDesc));
```

创建 cuDNN 的各种资源句柄和描述符，用于管理张量、卷积核和卷积操作的属性。cudnnCreate 创建 cuDNN 全局上下文句柄，用于后续 API 调用。cudnnCreateTensorDescriptor 创建张量描述符，用于指定输入和输出的形状、数据类型和内存布局。cudnnCreateFilterDescriptor 创建卷积核描述符（filterDesc），用于指定卷积核的形状和数据类型。cudnnCreateConvolutionDescriptor 创建卷积描述符（convDesc），定义卷积的超参数（如步幅、填充方式等）。

(iii). cudnnSet*Descriptor 系列函数

```
// Input tensor descriptor
CHECK_CUDNN(cudnnSetTensor4dDescriptor(inputDesc, CUDNN_TENSOR_NCHW, CUDNN_DATA_FLOAT, N, C, H, W));

// Kernel descriptor
CHECK_CUDNN(cudnnSetFilter4dDescriptor(filterDesc, CUDNN_DATA_FLOAT, CUDNN_TENSOR_NCHW, K, C, R, S));

// Convolution descriptor
CHECK_CUDNN(cudnnSetConvolution2dDescriptor(convDesc, pad_h, pad_w, stride_h, stride_w, 1, 1, CUDNN_CROSS_CORRELATION, CUDNN_DATA_FLOAT));
```

为张量、卷积核和卷积操作的描述符设置具体属性。cudnnSetTensor4dDescriptor 设置输入和输出张量的形状、数据类型，以及存储格式。cudnnSetFilter4dDescriptor 设置卷积核的形状和数据类型。cudnnSetConvolution2dDescriptor 设置卷积参数，包括填充、步幅，以及卷积类型。

(iv). cudnnGetConvolutionForwardWorkspaceSize

```
CHECK_CUDNN(cudnnGetConvolutionForwardWorkspaceSize(cudnn, inputDesc, filterDesc, convDesc, outputDesc, algo, &workspaceSize));
void* d_workspace = nullptr;
if (workspaceSize > 0) {
    CHECK_CUDA(cudaMalloc(&d_workspace, workspaceSize));
}
```

获取执行卷积操作所需的工作空间大小。根据选择的卷积算法计算所需的额外工作空间大小（workspaceSize）。如果需要额外空间，则在 GPU 上分配对应的内存。

(v). cudnnConvolutionForward

```
// Perform convolution
CHECK_CUDNN(cudnnConvolutionForward(cudnn, &alpha, inputDesc, d_input, filterDesc, d_kernel, convDesc, algo, d_workspace, workspaceSize, &beta, outputDesc, d_output));
```

执行前向卷积操作，完成输入张量与卷积核的计算，得到输出张量。inputDesc 和 d_input 是输入张量描述符和数据。filterDesc 和 d_kernel 是卷积核描述符和数据。convDesc 卷积描述符，定义卷积超参数。algo 选择的卷积算法。d_workspace 和 workspaceSize 卷积计算需要的临时工作空间。outputDesc 和 d_output 输出张量描述符和结果数据。

(vi). 思考：

cuDNN 是经过高度优化的深度学习库，针对不同的硬件架构进行了深入调优，能够高效地利用内存和计算资源，同时实现最优的并行性能。如果普通的 CUDA 矩阵乘法性能不如 cuDNN，很可能是因为普通实现未能充分利用 GPU 的硬件资源，例如共享内存、寄存器等。相比之下，普通实现通常存在全局内存访问过多、线程分配不合理或者未充分利用共享内存的情况。这些问题会导致内存带宽受限和计算单元闲置，从而影响整体性能。以下是一些改进方法：

(a). 利用 Shared Memory：普通的 CUDA 矩阵乘法可能直接从全局内存中加载数据，导致大量的全局内存访问，增加了延迟。使用 CUDA 的 shared memory 将需要频繁访问的数据加载到共享内存中，减少全局内存访问。共享内存的访问速度接近寄存器速度，可以显著提高性能。

(b). 优化线程分配：普通 CUDA 实现可能没有充分利用 GPU 的并行计算能力，例如线程块大小设置不合理，导致计算资源未充分利用。合理设置 grid 和 block 的大小，确保 GPU 中每个 SM（流式多处理器）有足够的线程运行。更好的线程分配可以减少 GPU 的空闲时间，提高吞吐量。

(c). 使用寄存器优化数据存储：普通实现中可能将中间计算结果存储在全局内存中，造成不必要的内存延迟。寄存器是 GPU 上最快的存储空间，将频繁使用的中间结果存储在线程私有的寄存器中。避免频繁访问全局内存。

(d). 避免分支和分歧：条件分支会导致 warp 中部分线程等待其他线程完成不同分支的计算，从而降低并行效率。减少或消除条件分支，使用线性计算替代分支逻辑。保证同一 warp 内的线程执行相同路径的代码，避免 warp divergence。

(e). 调整数据布局（Memory Coalescing）：普通实现可能在内存访问模式上存在问题，导致全局内存访问未对齐或分散。调整矩阵的存储布局，确保线程访问的全局内存地址是连续的（即 memory coalescing）。确保线程访问遵循一致的步幅模式，减少缓存失效。高效的内存访问模式可以减少全局内存访问延迟，提高计算效率。

(f). 减少多次 Kernel 启动开销：普通实现中可能将矩阵乘法拆分为多个小核函数（kernel），导致内核启动开销过大。可以合并内核函数，减少 kernel 启动次数。在单个内核中完成更多计算任务。内核启动开销在 GPU 计算中是非计算部分的主要性能瓶颈，尤其是小规模任务时。

3. 实验结果

(1). 通过 CUDA 实现通用矩阵乘法（Lab1）的并行版本

命令格式：nvcc -o lab5_1 lab5_1.cu
./lab5_1

```
Matrix A (top-left 2*2):  
0.109 0.803 ...  
0.359 0.195 ...  
...  
  
Matrix B (top-left 2*2):  
0.532 0.89 ...  
0.609 0.438 ...  
...  
  
Matrix C (top-left 2*2):  
2045.91 2052.89 ...  
2041.89 2035.48 ...  
...  
  
Execution time: 8.03473 seconds.
```

(2). 通过 NVIDIA 的矩阵计算函数库 CUBLAS 计算矩阵相乘。

命令格式：nvcc -o lab5_2 lab5_2.cu -lcublas
./lab5_2

```
Matrix A:  
84.019 39.438 ...  
61.864 48.007 ...  
...  
  
Matrix B:  
95.011 83.117 ...  
71.903 71.583 ...  
...  
  
Matrix C:  
1261494.750 1315242.000 ...  
1383085.250 1430396.500 ...  
...  
  
Execution time (512x512): 0.000 seconds  
Execution time (1024x1024): 0.001 seconds  
Execution time (2048x2048): 0.006 seconds  
Execution time (4096x4096): 0.041 seconds  
Execution time (8192x8192): 0.614 seconds
```

(3). 在 GPU 上实现神经网络中的卷积操作。

命令格式: `nvcc -o lab5_3 lab5_3.cu`
`./lab5_3`

(i). 输入为 256:

步幅为 1:

```
Running convolution with stride = 1
Input Matrix:
4 7 ...
1 7 ...
... (truncated)

Kernel Matrix:
5 3 ...
1 5 ...
... (truncated)

Output Matrix:
532 461 ...
498 441 ...
... (truncated)

Convolution runtime: 0.000225601 seconds
```

步幅为 2:

```
Running convolution with stride = 2
Input Matrix:
4 7 ...
1 7 ...
... (truncated)

Kernel Matrix:
5 3 ...
1 5 ...
... (truncated)

Output Matrix:
532 459 ...
426 426 ...
... (truncated)

Convolution runtime: 0.000146745 seconds
```

步幅为 3:

```
Running convolution with stride = 3
Input Matrix:
4 7 ...
1 7 ...
... (truncated)

Kernel Matrix:
5 3 ...
1 5 ...
... (truncated)

Output Matrix:
185 250 ...
288 426 ...
... (truncated)

Convolution runtime: 0.000177304 seconds
```


(ii). 输入为 4096:

步幅为 1:

```
Running convolution with stride = 1
Input Matrix:
4 7 ...
8 9 ...
... (truncated)

Kernel Matrix:
1 2 ...
2 1 ...
... (truncated)

Output Matrix:
436 424 ...
429 402 ...
... (truncated)

Convolution runtime: 0.0236944 seconds
```

步幅为 2:

```
Running convolution with stride = 2
Input Matrix:
4 7 ...
8 9 ...
... (truncated)

Kernel Matrix:
1 2 ...
2 1 ...
... (truncated)

Output Matrix:
436 385 ...
420 379 ...
... (truncated)

Convolution runtime: 0.00601076 seconds
```

步幅为 3:

```
Running convolution with stride = 3
Input Matrix:
4 7 ...
8 9 ...
... (truncated)

Kernel Matrix:
1 2 ...
2 1 ...
... (truncated)

Output Matrix:
219 257 ...
248 379 ...
... (truncated)

Convolution runtime: 0.00312563 seconds
```

(4). 使用 `im2col` 方法结合任务 1 实现的 GEMM（通用矩阵乘法）实现卷积操作。

命令格式: `nvcc -o lab5_4 lab5_4.cu`
`./lab5_4`

(i). 输入为 256:

步幅为 1:

```
Output matrix (first 2*2 elements):  
0.388134 0.725936 ...  
0.0502654 0.0940125 ...  
...  
Execution time: 0.000164461 seconds
```

步幅为 2:

```
Output matrix (first 2*2 elements):  
0.388134 0.725936 ...  
0.0502654 0.0940125 ...  
...  
Execution time: 0.000130412 seconds
```

步幅为 3:

```
Output matrix (first 2*2 elements):  
0.388134 0.725936 ...  
0.0502654 0.0940125 ...  
...  
Execution time: 0.000125439 seconds
```

(ii). 输入为 4096:

步幅为 1:

```
Output matrix (first 2*2 elements):  
0.643384 0.67227 ...  
0.0833217 0.0870625 ...  
...  
Execution time: 0.00965877 seconds
```

步幅为 2:

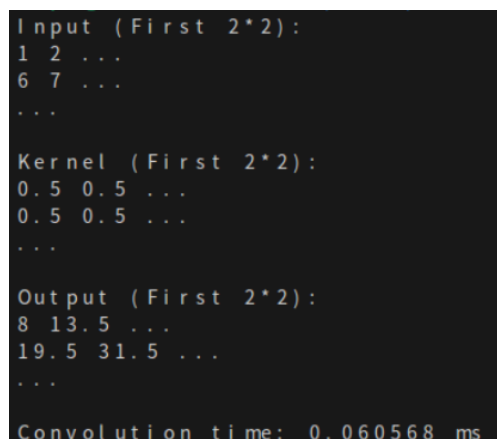
```
Output matrix (first 2*2 elements):  
0.643384 0.67227 ...  
0.0833217 0.0870625 ...  
...  
Execution time: 0.00323069 seconds
```

步幅为 3:

```
Output matrix (first 2*2 elements):  
0.643384 0.67227 ...  
0.0833217 0.0870625 ...  
...  
Execution time: 0.00166079 seconds
```

(5). 使用 cuDNN 提供的卷积方法进行卷积操作。

命令格式: `nvcc -o lab5_5 lab5_5.cu -lcudnn`
`./lab5_5`

A terminal window with a black background and white text. It displays the output of a convolution operation. The text is as follows:

```
Input (First 2*2):  
1 2 ...  
6 7 ...  
...  
  
Kernel (First 2*2):  
0.5 0.5 ...  
0.5 0.5 ...  
...  
  
Output (First 2*2):  
8 13.5 ...  
19.5 31.5 ...  
...  
  
Convolution time: 0.060568 ms
```

4. 实验感想

本次实验涉及 CUDA 矩阵乘法和卷积操作的实现与性能优化，通过与 cuBLAS 和 cuDNN 的对比，我进一步理解 GPU 的计算特性及优化策略。在实现 CUDA 矩阵乘法时，我掌握了 CUDA 线程网格与块的分配方法，以及如何通过每个线程计算一个结果矩阵元素的方式实现并行计算。相比高度优化的 cuBLAS 库，CUDA 矩阵乘法性能存在明显差距。进一步优化后，理解了共享内存和寄存器的有效使用能够显著减少全局内存访问延迟，提高计算效率。卷积操作的实现挑战性较高。

直接卷积方法通过多层嵌套循环计算，但随着输入尺寸的增加，计算复杂度迅速提升。相比之下，im2col 方法结合 GEMM 极大地提升了计算效率，使得卷积操作能够以矩阵乘法的形式实现，利用了现有高效矩阵乘法算法的优势。在实现 im2col 的过程中，我加深了对数据重排及内存布局优化的认识。最后，通过 cuDNN 完成卷积操作并与自己实现的方法进行对比，不仅见识了专业库的优化能力，也进一步认识到普通实现在全局内存访问优化、数据布局调整以及并行化程度等方面的不足。cuDNN 的出色表现归因于其充分利用了 GPU 硬件架构的特点，尤其是在计算与内存访问的优化上。这种对比促使我更清晰地认识到高效算法与硬件资源结合的重要性。

本次实验让我全面体会到 GPU 计算在高性能计算中的潜力与挑战。未来，我希望能进一步深入研究 GPU 性能优化策略，加深对高性能计算程序的认识。