中山大学计算机院本科生实验报告

(2024学年秋季学期)

课程名称：高性能计算程序设计

实验	Cuda并行程序设计	专业(方向)	信息与计算科学
学号	**22336313**	**姓名**	**郑鸿鑫**
Email	zhenghx57@mail2.sysu.edu.cn	**完成日期**	2024/12/15

1. 实验目的

针对CUDA编程和深度学习中的卷积神经网络（CNN）操作。实验内容包括使用CUDA实现通用矩阵乘法（GEMM）的并行版本、利用CUBLAS库进行矩阵乘法、在GPU上实现直接卷积和使用im2col方法进行卷积操作，以及使用NVIDIA cuDNN库进行卷积操作。通过这些任务，实验旨在提升对并行计算、矩阵运算和卷积技术的理解，并评估不同实现方法的性能，从而深入掌握高性能计算在深度学习领域的应用。

2. 实验过程和核心代码

> **子任务1** 通过CUDA实现通用矩阵乘法（Lab1）的并行版本，CUDA Thread Block size从32增加至512，矩阵规模从512增加至8192。

编写通用矩阵乘法的cuda核函数如下所示：

```
__global__ void matrixMulKernel(float *A, float *B, float *C, int N, int K) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    int x = idx % N; // 输出矩阵C的列索引
    int y = idx / N; // 输出矩阵C的行索引

    if (x < N && y < K) {
        float sum = 0.0f;
        for (int i = 0; i < N; ++i) {
            sum += A[y * N + i] * B[i * K + x];
        }
        C[y * N + x] = sum;
    }
}
```

一些分配资源的操作：

```
//主机端内存分配
float *h_A = new float[M * N];
float *h_B = new float[N * K];
float *h_C = new float[M * K];
//随机数初始化数据
srand(time(0));
for (int i = 0; i < M * N; ++i) {
    h_A[i] = static_cast<float>(rand()) %10;
}
for (int i = 0; i < N * K; ++i) {
    h_B[i] = static_cast<float>(rand()) %10;
}
//设备端内存分配
float *d_A, *d_B, *d_C;
cudaMalloc((void **)&d_A, M * N * sizeof(float));
cudaMalloc((void **)&d_B, N * K * sizeof(float));
cudaMalloc((void **)&d_C, M * K * sizeof(float));
//数据拷贝到设备端
cudaMemcpy(d_A, h_A, M * N * sizeof(float), cudaMemcpyHostToDevice);
cudaMemcpy(d_B, h_B, N * K * sizeof(float), cudaMemcpyHostToDevice);
//结果拷贝回主机
cudaMemcpy(h_C, d_C, M * K * sizeof(float), cudaMemcpyDeviceToHost);
//释放内存
cudaFree(d_A);
cudaFree(d_B);
cudaFree(d_C);
delete[] h_A;
delete[] h_B;
delete[] h_C;
```

编译运行，用小规模矩阵来检验矩阵乘法的正确性：

```
ehpc@519cef0053f6:~/Desktop/Cuda$ nvcc -o matrix_multiply matrix_multiply.cu
ehpc@519cef0053f6:~/Desktop/Cuda$ ./matrix_multiply
Matrix A:
      6.0       7.0       3.0       3.0
      9.0       0.0       7.0       1.0
      0.0       0.0       7.0       7.0
      7.0       8.0       6.0       6.0
Matrix B:
      2.0       7.0       2.0       5.0
      2.0       4.0       7.0       5.0
      4.0       6.0       8.0       1.0
      8.0       9.0       1.0       5.0
Matrix C:
     62.0     115.0      88.0      83.0
     54.0     114.0      75.0      57.0
     84.0     105.0      63.0      42.0
    102.0     171.0     124.0     111.0
Run time:  0.0 s
Matrix sizes:  4
Block sizes:3
```

> **子任务2** 通过NVDIA的矩阵计算函数库CUBLAS计算矩阵相乘，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

分配资源的操作与上面类似，关键代码如下：

```
//声明并初始化一个cublas库的操作句柄
cublasHandle_t handle;
cublasCreate(&handle);
//alpha和beta分别用于矩阵乘法中的标量乘法和累加操作。alpha设置为1.0（意味着A和B的乘积不会被额外缩放），beta设置为0.0（意味着在计算新的C之前，旧的C
float alpha = 1.0f;
float beta = 0.0f;
auto start = std::chrono::high_resolution_clock::now();
// 执行矩阵乘法
cublasSgemm(handle, CUBLAS_OP_N, CUBLAS_OP_N, N, M, K, &alpha, d_B, N, d_A, K, &beta, d_C, N);
cudaDeviceSynchronize();
```

最后结束时需要销毁句柄，释放资源：

```
cublasDestroy(handle);
```

编译运行，用小规模矩阵来检验cublas实现矩阵乘法的正确性：

```
ehpc@519cef0053f6:~/Desktop/Cuda$ nvcc -o cublas_matrix_multiply cublas_matrix_multiply.cu -lcublas
ehpc@519cef0053f6:~/Desktop/Cuda$ ./cublas_matrix_multiply
Matrix A:
      9.0       8.0       0.0       5.0
      4.0       7.0       3.0       1.0
      4.0       5.0       7.0       3.0
      7.0       2.0       9.0       2.0
Matrix B:
      5.0       0.0       2.0       9.0
      0.0       6.0       6.0       5.0
      0.0       8.0       6.0       8.0
      2.0       5.0       1.0       3.0
Matrix C:
     55.0      73.0      71.0     136.0
     22.0      71.0      69.0      98.0
     26.0     101.0      83.0     126.0
     39.0      94.0      82.0     151.0
Run time:  0.0 s
Matrix sizes:  4
```

> **子任务3** 用直接卷积的方式对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，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。

定义直接计算卷积的cuda核函数：

```
__global__ void convolutionKernel(const float* input, const float* kernel, float* output,
                                  int padding_size, int kernel_size, int output_size, int stride, int depth) {
    int x = blockIdx.x * blockDim.x + threadIdx.x;
    int y = blockIdx.y * blockDim.y + threadIdx.y;
    if (x < output_size && y < output_size) {
        float value = 0.0f;
        // 遍历通道
        for (int c = 0; c < depth; ++c) {
            // 遍历卷积核
            for (int ky = 0; ky < kernel_size; ++ky) {
                for (int kx = 0; kx < kernel_size; ++kx) {
                    // 计算输入图像中的位置
                    int ix = x * stride + kx;
                    int iy = y * stride + ky;
                    if (ix >= 0 && ix < padding_size && iy >= 0 && iy < padding_size) {
                        int input_idx = (c * padding_size * padding_size) + (iy * padding_size + ix);
                        int kernel_idx = (c * kernel_size * kernel_size) + (ky * kernel_size + kx);
                        value += input[input_idx] * kernel[kernel_idx];
                    }
                }
            }
        }
        output[y * output_size + x] = value;
    }
}
```

main函数中关键代码展示（分配资源的代码省略，与上文类似）

```
//定义参数：
const int input_size = 8;
const int kernel_size = 3;
const int stride = 2;
const int padding = 1;
const int depth = 3;
const int padding_size = input_size + 2 * padding; // 包含填充后的大小
const int output_size = (input_size - kernel_size + 2 * padding) / stride + 1; // 输出矩阵的尺寸
//初始化输入数据和卷积核
//初始化输入数据（带填充）
srand(time(0));
for (int c = 0; c < depth; ++c) {
    for (int y = 0; y < padding_size; ++y) {
        for (int x = 0; x < padding_size; ++x) {
            int index = c * padding_size * padding_size + y * padding_size + x;
            if (x < padding || x >= input_size + padding || y < padding || y >= input_size + padding) {
                h_input[index] = 0.0f; // 填充区域
            } else {
                h_input[index] = static_cast<float>(rand() % 10); // 非填充区域
            }
        }
    }
}
// 初始化卷积核
for (int i = 0; i < kernel_size * kernel_size * depth; ++i) {
    h_kernel[i] = static_cast<float>(rand() % 5 + 1); // 随机权重
}
// 启动卷积核函数
    convolutionKernel<<<gridDim, blockDim>>>(d_input, d_kernel, d_output,
                                             padding_size, kernel_size, output_size, stride, depth);
```

编译运行，用小规模矩阵来检验直接卷积法的正确性：

```
ehpc@519cef0053f6:~/Desktop/Cuda$ nvcc -o direct_convolution direct_convolution.cu
ehpc@519cef0053f6:~/Desktop/Cuda$ ./direct_convolution
Inpu(include padding)t:
Channel 0:
      0.0       0.0       0.0       0.0       0.0       0.0       0.0       0.0
      0.0       6.0       9.0       1.0       0.0       2.0       8.0       0.0
      0.0       8.0       9.0       3.0       0.0       6.0       4.0       0.0
      0.0       1.0       3.0       6.0       4.0       0.0       4.0       0.0
      0.0       4.0       3.0       8.0       8.0       6.0       2.0       0.0
      0.0       2.0       2.0       6.0       5.0       3.0       8.0       0.0
      0.0       3.0       9.0       9.0       6.0       9.0       3.0       0.0
      0.0       0.0       0.0       0.0       0.0       0.0       0.0       0.0
Channel 1:
      0.0       0.0       0.0       0.0       0.0       0.0       0.0       0.0
      0.0       5.0       0.0       4.0       8.0       2.0       1.0       0.0
      0.0       4.0       3.0       6.0       0.0       9.0       6.0       0.0
      0.0       6.0       3.0       0.0       5.0       4.0       8.0       0.0
      0.0       9.0       8.0       2.0       6.0       3.0       5.0       0.0
      0.0       6.0       7.0       6.0       7.0       3.0       8.0       0.0
      0.0       1.0       0.0       8.0       5.0       8.0       2.0       0.0
      0.0       0.0       0.0       0.0       0.0       0.0       0.0       0.0
Channel 2:
      0.0       0.0       0.0       0.0       0.0       0.0       0.0       0.0
      0.0       8.0       4.0       7.0       5.0       5.0       6.0       0.0
      0.0       3.0       3.0       9.0       5.0       8.0       3.0       0.0
      0.0       5.0       0.0       2.0       7.0       8.0       7.0       0.0
      0.0       4.0       4.0       4.0       1.0       3.0       0.0       0.0
      0.0       9.0       4.0       2.0       9.0       2.0       1.0       0.0
      0.0       1.0       2.0       7.0       8.0       9.0       4.0       0.0
      0.0       0.0       0.0       0.0       0.0       0.0       0.0       0.0
Kernel:
Channel 0:
      5.0       4.0       4.0
      1.0       1.0       4.0
      2.0       4.0       4.0
Channel 1:
      1.0       1.0       2.0
      3.0       3.0       3.0
      5.0       4.0       2.0
Channel 2:
      2.0       5.0       1.0
      4.0       4.0       5.0
      2.0       5.0       3.0
Convolution Result:
   223.0     269.0
   267.0     377.0
Input_size:6
srtide:3
Total Execution Time: 2155 ms
```

> **子任务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
> 定义参数和分配资源的代码跟上面类似，此处不再展示：
> 编写im2col核函数如下（将输入矩阵根据每个卷积核窗口展成一个kernel_size * kernel_size行，
> output_size * output_size列的一个矩阵，其中每一个卷积核窗口中的输入元素展开作为其中的一列）

```
__global__ void im2colKernel(const float* input, float* col, int padding_size, int kernel_size,
                             int stride, int padding, int output_size, int depth) {
    int tid = blockIdx.x * blockDim.x + threadIdx.x; // 每个线程负责一个卷积核窗口
    if (tid < output_size * output_size) {
        int out_x = tid % output_size; // 输出矩阵的列索引
        int out_y = tid / output_size; // 输出矩阵的行索引
        for (int c = 0; c < depth; ++c) {
            for (int ky = 0; ky < kernel_size; ++ky) {
                for (int kx = 0; kx < kernel_size; ++kx) {
                    int in_x = out_x * stride + kx;
                    int in_y = out_y * stride + ky;
                    int col_index = ((c * kernel_size * kernel_size + ky * kernel_size + kx) * output_size * output_size) + tid;
                    int input_index = (c * padding_size * padding_size) + (in_y * padding_size + in_x);
                    col[col_index] = input[input_index];
                }
            }
        }
    }
}
```

将通用矩阵乘法核函数稍作修改，让其可以累加多通道的结果作为最终结果，传入矩阵乘法的矩阵分别为行向量:1行kernel_size * kernel_size列的卷积核矩阵，还有im2col的输出：kernel_size * kernel_size行,output_size * output_size列的列向量矩阵,具体代码如下所示：

```
// 矩阵乘法核函数（按通道累加）
__global__ void matrixMulKernel(const float* A, const float* B, float* C,
                                int depth, int kernel_size, int output_size) {
    int out_x = blockIdx.x * blockDim.x + threadIdx.x;
    int out_y = blockIdx.y * blockDim.y + threadIdx.y;
    if (out_x < output_size && out_y < output_size) {
        int output_index = out_y * output_size + out_x;
        float sum = 0.0f;
        for (int c = 0; c < depth; ++c) {
            for (int ky = 0; ky < kernel_size; ++ky) {
                for (int kx = 0; kx < kernel_size; ++kx) {
                    int kernel_index = c * kernel_size * kernel_size + ky * kernel_size + kx;
                    int col_index = ((c * kernel_size * kernel_size + ky * kernel_size + kx) * output_size * output_size) + output_index;
                    sum += A[kernel_index] * B[col_index];
                }
            }
        }
        C[output_index] = sum;
    }
}
```

编译运行，用小规模矩阵来检验im2col实现卷积法的正确性：

```
ehpc@519cef0053f6:~/Desktop/Cuda$ nvcc -o im2col_convolution im2col_convolution.cu
ehpc@519cef0053f6:~/Desktop/Cuda$ ./im2col_convolution
Inpu(include padding)t:
Channel 0:
      0.0       0.0       0.0       0.0       0.0       0.0       0.0       0.0
      0.0       2.0       8.0       7.0       5.0       0.0       1.0       0.0
      0.0       7.0       8.0       4.0       7.0       3.0       0.0       0.0
      0.0       9.0       8.0       5.0       0.0       6.0       7.0       0.0
      0.0       3.0       2.0       7.0       3.0       7.0       4.0       0.0
      0.0       5.0       2.0       9.0       1.0       0.0       0.0       0.0
      0.0       8.0       2.0       0.0       6.0       7.0       1.0       0.0
      0.0       0.0       0.0       0.0       0.0       0.0       0.0       0.0
Channel 1:
      0.0       0.0       0.0       0.0       0.0       0.0       0.0       0.0
      0.0       7.0       7.0       1.0       1.0       4.0       4.0       0.0
      0.0       3.0       3.0       5.0       0.0       6.0       1.0       0.0
      0.0       0.0       1.0       4.0       9.0       4.0       1.0       0.0
      0.0       3.0       2.0       3.0       4.0       5.0       4.0       0.0
      0.0       6.0       3.0       8.0       7.0       1.0       8.0       0.0
      0.0       0.0       8.0       5.0       1.0       2.0       9.0       0.0
      0.0       0.0       0.0       0.0       0.0       0.0       0.0       0.0
Channel 2:
      0.0       0.0       0.0       0.0       0.0       0.0       0.0       0.0
      0.0       7.0       7.0       5.0       2.0       0.0       3.0       0.0
      0.0       4.0       0.0       4.0       8.0       9.0       1.0       0.0
      0.0       1.0       4.0       5.0       4.0       8.0       0.0       0.0
      0.0       0.0       4.0       5.0       1.0       3.0       9.0       0.0
      0.0       9.0       3.0       9.0       4.0       6.0       1.0       0.0
      0.0       5.0       6.0       1.0       0.0       8.0       1.0       0.0
      0.0       0.0       0.0       0.0       0.0       0.0       0.0       0.0
Kernel:
Channel 0:
      3.0       4.0       3.0
      0.0       4.0       2.0
      3.0       5.0       8.0
Channel 1:
      8.0       2.0       6.0
      0.0       2.0       2.0
      7.0       3.0       8.0
Channel 2:
      6.0       4.0       3.0
      6.0       0.0       0.0
      9.0       6.0       6.0
Convolution Result :
  208.00    352.00
  263.00    509.00
Input_size:6
srtide:3
Total Execution Time: 2344 ms
```

> **子任务5** 使用cuDNN提供的卷积方法进行卷积操作，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。

关键代码如下，其中参数设定和资源分配等类似上面，省略

```
// 初始化 cuDNN
cudnnHandle_t cudnn;
checkCUDNN(cudnnCreate(&cudnn));
// 创建 Tensor 描述符
cudnnTensorDescriptor_t input_desc, output_desc;
checkCUDNN(cudnnCreateTensorDescriptor(&input_desc));
checkCUDNN(cudnnCreateTensorDescriptor(&output_desc));
// 输入 Tensor 描述
checkCUDNN(cudnnSetTensor4dDescriptor(
    input_desc, CUDNN_TENSOR_NCHW, CUDNN_DATA_FLOAT,
    batch_size, depth, input_size, input_size));
// 输出 Tensor 描述
checkCUDNN(cudnnSetTensor4dDescriptor(
    output_desc, CUDNN_TENSOR_NCHW, CUDNN_DATA_FLOAT,
    batch_size, output_channels, output_size, output_size));
// 卷积描述符
cudnnConvolutionDescriptor_t conv_desc;
checkCUDNN(cudnnCreateConvolutionDescriptor(&conv_desc));
checkCUDNN(cudnnSetConvolution2dDescriptor(
    conv_desc, padding, padding, stride, stride,
    1, 1, CUDNN_CROSS_CORRELATION, CUDNN_DATA_FLOAT));
// 卷积核描述符
cudnnFilterDescriptor_t filter_desc;
checkCUDNN(cudnnCreateFilterDescriptor(&filter_desc));
checkCUDNN(cudnnSetFilter4dDescriptor(
    filter_desc, CUDNN_DATA_FLOAT, CUDNN_TENSOR_NCHW,
    output_channels, depth, kernel_size, kernel_size));
// 获取卷积算法
cudnnConvolutionFwdAlgo_t algo;
checkCUDNN(cudnnGetConvolutionForwardAlgorithm(
    cudnn, input_desc, filter_desc, conv_desc, output_desc,
    CUDNN_CONVOLUTION_FWD_PREFER_FASTEST, 0, &algo));
// 获取工作空间大小
size_t workspace_size = 0;
checkCUDNN(cudnnGetConvolutionForwardWorkspaceSize(
    cudnn, input_desc, filter_desc, conv_desc, output_desc,
    algo, &workspace_size));
```

设置卷积前向操作的参数以及调用：

```
// 设置卷积前向操作的 alpha 和 beta
float alpha = 1.0f, beta = 0.0f;
// 执行卷积前向操作
checkCUDNN(cudnnConvolutionForward(
    cudnn, &alpha, input_desc, d_input, filter_desc, d_kernel,
    conv_desc, algo, workspace, workspace_size, &beta, output_desc, d_output));
```

其中上文涉及到的checkCUDNN函数为检查过程中是否有错误，这是由于我们在运行这个程序时，当矩阵大小超过512时便一直报错：

$$Segmentation fault (core dumped)$$

故添加上这个函数来调试错误，代码如下：
// 检查 cuDNN 的返回状态

```
void checkCUDNN(cudnnStatus_t status) {
    if (status != CUDNN_STATUS_SUCCESS) {
        std::cerr << "cuDNN Error: " << cudnnGetErrorString(status) << std::endl;
        exit(EXIT_FAILURE);
    }
}
```

编译运行，用小规模矩阵来检验cuDNN实现卷积法的正确性：

```
ehpc@519cef0053f6:~/Desktop/Cuda$ nvcc -o cuDNN_convolution cuDNN_convolution.cu -lcudnn
ehpc@519cef0053f6:~/Desktop/Cuda$ ./cuDNN_convolution
Inpu(include padding)t:
Channel 0:
      0.0       0.0       0.0       0.0       0.0       0.0       0.0       0.0
      0.0       3.0       6.0       7.0       5.0       3.0       5.0       0.0
      0.0       6.0       2.0       9.0       1.0       2.0       7.0       0.0
      0.0       0.0       9.0       3.0       6.0       0.0       6.0       0.0
      0.0       2.0       6.0       1.0       8.0       7.0       9.0       0.0
      0.0       2.0       0.0       2.0       3.0       7.0       5.0       0.0
      0.0       9.0       2.0       2.0       8.0       9.0       7.0       0.0
      0.0       0.0       0.0       0.0       0.0       0.0       0.0       0.0
Channel 1:
      0.0       0.0       0.0       0.0       0.0       0.0       0.0       0.0
      0.0       3.0       6.0       1.0       2.0       9.0       3.0       0.0
      0.0       1.0       9.0       4.0       7.0       8.0       4.0       0.0
      0.0       5.0       0.0       3.0       6.0       1.0       0.0       0.0
      0.0       6.0       3.0       2.0       0.0       6.0       1.0       0.0
      0.0       5.0       5.0       4.0       7.0       6.0       5.0       0.0
      0.0       6.0       9.0       3.0       7.0       4.0       5.0       0.0
      0.0       0.0       0.0       0.0       0.0       0.0       0.0       0.0
Channel 2:
      0.0       0.0       0.0       0.0       0.0       0.0       0.0       0.0
      0.0       2.0       5.0       4.0       7.0       4.0       4.0       0.0
      0.0       3.0       0.0       7.0       8.0       6.0       8.0       0.0
      0.0       8.0       4.0       3.0       1.0       4.0       9.0       0.0
      0.0       2.0       0.0       6.0       8.0       9.0       2.0       0.0
      0.0       6.0       6.0       4.0       9.0       5.0       0.0       0.0
      0.0       4.0       8.0       7.0       1.0       7.0       2.0       0.0
      0.0       0.0       0.0       0.0       0.0       0.0       0.0       0.0
Kernel:
Channel 0:
      7.0       2.0       2.0
      6.0       1.0       0.0
      6.0       1.0       5.0
Channel 1:
      9.0       4.0       9.0
      0.0       9.0       1.0
      7.0       7.0       1.0
Channel 2:
      1.0       5.0       9.0
      7.0       7.0       6.0
      7.0       3.0       6.0
Single Channel Convolution Result:
  108.00    226.00
  323.00    308.00
Input_size:6
srtide:3
cuDNN Convolution Run Time: 0.08 ms
```

3. 实验结果

子任务1

仅展示512x512的结果，详细结果见Result文件夹

```
ehpc@e693fc586066:~/Desktop/Cuda$ nvcc -o matrix_multiply matrix_multiply.cu
ehpc@e693fc586066:~/Desktop/Cuda$ ./matrix_multiply
Run time: 0.00314122 s
Matrix sizes: 512
Block sizes:32
ehpc@e693fc586066:~/Desktop/Cuda$ nvcc -o matrix_multiply matrix_multiply.cu
ehpc@e693fc586066:~/Desktop/Cuda$ ./matrix_multiply
Run time: 0.00234071 s
Matrix sizes: 512
Block sizes:64
ehpc@e693fc586066:~/Desktop/Cuda$ nvcc -o matrix_multiply matrix_multiply.cu
ehpc@e693fc586066:~/Desktop/Cuda$ ./matrix_multiply
Run time: 0.00256321 s
Matrix sizes: 512
Block sizes:128
ehpc@e693fc586066:~/Desktop/Cuda$ nvcc -o matrix_multiply matrix_multiply.cu
ehpc@e693fc586066:~/Desktop/Cuda$ ./matrix_multiply
Run time: 0.00231464 s
Matrix sizes: 512
Block sizes:256
ehpc@e693fc586066:~/Desktop/Cuda$ nvcc -o matrix_multiply matrix_multiply.cu
ehpc@e693fc586066:~/Desktop/Cuda$ ./matrix_multiply
Run time: 0.00245596 s
Matrix sizes: 512
Block sizes:512
```

完整的实验结果整理如下表所示：

Matrix Size	Block Size	Run Time (s)
512	32	0.0031412
512	64	0.00234071
512	128	0.0025632
512	256	0.00231464
512	512	0.0024559
1024	32	0.0292028
1024	64	0.0200948
1024	128	0.0244402
1024	256	0.0183831
1024	512	0.0199324
2048	32	0.234102
2048	64	0.212029
2048	128	0.200228
2048	256	0.209189
2048	512	0.207608
4096	32	1.9365
4096	64	1.81061
4096	128	1.7786
4096	256	1.77622
4096	512	1.76855

Matrix Size	Block Size	Run Time (s)
8192	32	17.4036
8192	64	13.9042
8192	128	14.144
8192	256	14.1178
8192	512	14.063

分析实验结果，我们可以观察到，对于较小的矩阵尺寸（如512x512），较大的块大小（如256或512）在某些情况下可能会因为更有效的内存访问模式和更高的资源利用率而提供更好的性能。然而，对于较大的矩阵尺寸（如2048x2048和4096x4096），较小的块大小（如32或64）通常能够提供更快的运行时间，这可能是因为较小的块大小允许更细粒度的并行计算，减少了内存访问的竞争和线程同步的开销。此外，随着矩阵尺寸的增加，运行时间的增加并非线性，这可能是由于硬件资源限制、内存带宽或算法效率等因素。这些结果强调了在CUDA编程中，性能调优是一个复杂的过程，需要考虑矩阵大小、块大小、内存访问模式和GPU资源利用率等多个因素。

子任务2

利用cublas库计算得到的运行时间如下所示：

```
ehpc@519cef0053f6:~/Desktop/Cuda$ nvcc -o cublas_matrix_multiply cublas_matrix_multiply.cu -lcublas
ehpc@519cef0053f6:~/Desktop/Cuda$ ./cublas_matrix_multiply
Matrix A:
Run time: 0.00023992 s
Matrix sizes: 512
ehpc@519cef0053f6:~/Desktop/Cuda$ nvcc -o cublas_matrix_multiply cublas_matrix_multiply.cu -lcublas
ehpc@519cef0053f6:~/Desktop/Cuda$ ./cublas_matrix_multiply
Matrix A:
Run time: 0.00120179 s
Matrix sizes: 1024
ehpc@519cef0053f6:~/Desktop/Cuda$ nvcc -o cublas_matrix_multiply cublas_matrix_multiply.cu -lcublas
ehpc@519cef0053f6:~/Desktop/Cuda$ ./cublas_matrix_multiply
Matrix A:
Run time: 0.00800789 s
Matrix sizes: 2048
ehpc@519cef0053f6:~/Desktop/Cuda$ nvcc -o cublas_matrix_multiply cublas_matrix_multiply.cu -lcublas
ehpc@519cef0053f6:~/Desktop/Cuda$ ./cublas_matrix_multiply
Matrix A:
Run time: 0.0414695 s
Matrix sizes: 4096
ehpc@519cef0053f6:~/Desktop/Cuda$ nvcc -o cublas_matrix_multiply cublas_matrix_multiply.cu -lcublas
ehpc@519cef0053f6:~/Desktop/Cuda$ ./cublas_matrix_multiply
Matrix A:
Run time: 0.336857 s
Matrix sizes: 8192
```

整理如下表：

Matrix Size	Run Time (s)
512	0.00023992
1024	0.00120179
2048	0.00800789
4096	0.0414695
8192	0.336857

可以看到直接调用cublas库的时候，运行时间相比于直接实现的矩阵乘法在块大小为32时会快上很多，但是块大小超过32时，则是我们直接实现的程序运行时间更快。

改进的方法：

优化内存访问模式以减少全局内存访问，增加共享内存的使用，以及确保内存访问是连续的，以提高内存带宽的利用率。优化寄存器的使用，减少寄存器溢出到全局内存，这可以显著提高性能。研究cuBLAS库的源码，了解其内部实现和优化策略，尝试将这些策略应用到自己的实现中。选择最佳的线程块大小和网格配置也可以改进程序的运行时间。

子任务3

只展示大小为4096的矩阵步长从1到3时的运行时间，完整结果见Result文件夹：

```
ehpc@519cef0053f6:~/Desktop/Cuda$ nvcc -o direct_convolution direct_convolution.cu
ehpc@519cef0053f6:~/Desktop/Cuda$ ./direct_convolution
Input_size:4096
srtide:1
Total Execution Time: 3609 ms
ehpc@519cef0053f6:~/Desktop/Cuda$ nvcc -o direct_convolution direct_convolution.cu
ehpc@519cef0053f6:~/Desktop/Cuda$ ./direct_convolution
Input_size:4096
srtide:2
Total Execution Time: 3602 ms
ehpc@519cef0053f6:~/Desktop/Cuda$ nvcc -o direct_convolution direct_convolution.cu
ehpc@519cef0053f6:~/Desktop/Cuda$ ./direct_convolution
Input_size:4096
srtide:3
Total Execution Time: 3433 ms
```

整理如下表：

Input Size	Stride	Total Execution Time (ms)
256	1	2353
256	2	2104
256	3	2287
512	1	2351
512	2	2252
512	3	2332
1024	1	2423
1024	2	2300
1024	3	2251
2048	1	2613
2048	2	2379
2048	3	2412
4096	1	3609
4096	2	3602
4096	3	3433

input size增加时，运行时间由于运算量的增加而增加，但增加的幅度没有很大，说明程序的主要时间可能除了计算方面还有在于资源分配的开销，随着stride的增大，输出的尺寸应该会减小可以减少运算量，但是实验结果体现出来的是步长为1时时间最长，其次是步长为3，最后步长为2时运行时间最短。

> **子任务4**

只展示大小为4096的矩阵步长从1到3时的运行时间，完整结果见Result文件夹：

```
ehpc@519cef0053f6:~/Desktop/Cuda$ nvcc -o im2col_convolution im2col_convolution.cu
ehpc@519cef0053f6:~/Desktop/Cuda$ ./im2col_convolution
Input_size:4096
srtide:1
Total Execution Time: 3508 ms
ehpc@519cef0053f6:~/Desktop/Cuda$ nvcc -o im2col_convolution im2col_convolution.cu
ehpc@519cef0053f6:~/Desktop/Cuda$ ./im2col_convolution
Input_size:4096
srtide:2
Total Execution Time: 3003 ms
ehpc@519cef0053f6:~/Desktop/Cuda$ nvcc -o im2col_convolution im2col_convolution.cu
ehpc@519cef0053f6:~/Desktop/Cuda$ ./im2col_convolution
Input_size:4096
srtide:3
Total Execution Time: 2906 ms
```

Input Size	Stride	Total Execution Time (ms)
256	1	2207
256	2	2353
256	3	2339
512	1	2394
512	2	2363
512	3	2268
1024	1	2321
1024	2	2235
1024	3	1997
2048	1	2390
2048	2	2664
2048	3	2650
4096	1	3508
4096	2	3003
4096	3	2906

分析类似于上面直接卷积法，不过使用im2col方法结合矩阵乘法后，在相同尺寸和步长的情况下，运行时间都会比直接卷积法略快一些。

子任务5

由于在使用cuDNN时，将矩阵从256到512都仍然可以正常运行，但是当大小达到或超过1024时会报错：

$$Segmentation fault(coredumped)$$

故展示矩阵大小为256和512的运行时间：

```
ehpc@519cef0053f6:~/Desktop/Cuda$ nvcc -o cuDNN_convolution cuDNN_convolution.cu -lcudnn
ehpc@519cef0053f6:~/Desktop/Cuda$ nvcc -o cuDNN_convolution cuDNN_convolution.cu -lcudnn
ehpc@519cef0053f6:~/Desktop/Cuda$ ./cuDNN_convolution
Input_size:256
srtide:1
cuDNN Convolution Run Time: 80 ms
ehpc@519cef0053f6:~/Desktop/Cuda$ nvcc -o cuDNN_convolution cuDNN_convolution.cu -lcudnn
ehpc@519cef0053f6:~/Desktop/Cuda$ ./cuDNN_convolution
Input_size:256
srtide:2
cuDNN Convolution Run Time: 79 ms
ehpc@519cef0053f6:~/Desktop/Cuda$ nvcc -o cuDNN_convolution cuDNN_convolution.cu -lcudnn
ehpc@519cef0053f6:~/Desktop/Cuda$ ./cuDNN_convolution
Input_size:256
srtide:3
cuDNN Convolution Run Time: 91 ms
ehpc@519cef0053f6:~/Desktop/Cuda$ nvcc -o cuDNN_convolution cuDNN_convolution.cu -lcudnn
ehpc@519cef0053f6:~/Desktop/Cuda$ ./cuDNN_convolution
Input_size:512
srtide:1
cuDNN Convolution Run Time: 104 ms
ehpc@519cef0053f6:~/Desktop/Cuda$ nvcc -o cuDNN_convolution cuDNN_convolution.cu -lcudnn
ehpc@519cef0053f6:~/Desktop/Cuda$ ./cuDNN_convolution
Input_size:512
srtide:2
cuDNN Convolution Run Time: 86 ms
ehpc@519cef0053f6:~/Desktop/Cuda$ nvcc -o cuDNN_convolution cuDNN_convolution.cu -lcudnn
ehpc@519cef0053f6:~/Desktop/Cuda$ ./cuDNN_convolution
Input_size:512
srtide:3
cuDNN Convolution Run Time: 77 ms
```

我们猜想可能的原因包括GPU内存不足无法处理大数据量、程序内存管理错误如越界访问或重复释放、cuDNN参数配置不当，以及不兼容的CUDA和cuDNN版本或硬件限制。解决这一问题通常需要确保GPU内存充足、检查和修正内存管理逻辑、调整cuDNN参数配置，并考虑硬件和软件环境的兼容性。
整理结果如下表所示：

Input Size	Stride	cuDNN Convolution Run Time (ms)
256	1	80
256	2	79
256	3	91
512	1	104
512	2	86
512	3	77

可以看到使用cuDNN实现的卷积方法，运行时间比我们自己实现的直接卷积法和im2col结合矩阵乘法实现卷积运行的时间都要快上很多。
可能的原因包括：
cuDNN库中的卷积操作经过了高度优化，可能使用了更高效的算法，比如Winograd算法、FFT算法或者直接卷积算法的优化版本。cuDNN能够更好地利用GPU的并行计算能力，通过并行执行多个操作来减少内存访问次数和提高计算效率。cuDNN在内存管理上可能更加高效，比如通过内存池化技术来减少内存分配和释放的开销。
改进的方法：
改进内存访问模式，减少全局内存访问，增加共享内存和寄存器的使用。调整线程块的大小和数量，找到最适合当前GPU硬件的并发度。在CUDA中使用共享内存来存储卷积核和输入数据，减少内存访问延迟。研究cuDNN的实现，了解其内部优化的策略，并尝试在自己的实现中应用这些策略。

4. 实验感想

我深刻体会到了GPU并行计算在现代计算中的核心作用。通过CUDA实现通用矩阵乘法，我不仅加深了对矩阵运算和内存管理的理解，而且通过调整Thread Block大小和矩阵规模，我学习到了如何平衡计算负载和优化性能。使用CUBLAS库的经历让我认识到了专业库的强大性能和易用性，同时也激发了我探索其底层优化策略的兴趣。在信号和图像处理领域中广泛使用的卷积技术，通过直接卷积和im2col方法的实现，我更加明白了其在深度学习中的重要性。最后，cuDNN的使用让我见识到了深度学习库的高效和便捷。整个实验过程不仅提升了我的编程技能，也加深了我对高性能计算在深度学习领域应用的认识。