第十一次实验 CUDA实现卷积算子

学号	姓名
20319045	刘冠麟

任务描述

任务一

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，kernel channel(depth)设置为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。注：实验的卷积操作不需要考虑bias(b)，bias设置为0。

输入

Input matrix size和Kernel size，例如 32 和 3，输入从256*256增加至4096*4096或者输入从32*32增加至512*512.

输出

输出卷积结果以及计算时间。

任务二

使用im2col方法实现CUDA并行卷积，将图像中每个需要进行卷积的窗口平铺为矩阵，并利用并行矩阵乘法模块实现卷积。

输入

Input matrix size和Kernel size，例如 32 和 3，输入从256*256增加至4096*4096或者输入从32*32增加至512*512.

输出

输出卷积结果以及计算时间。

任务三

NVIDIA cuDNN是用于深度神经网络的GPU加速库。它强调性能、易用性和低内存开销。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。

实验过程与核心代码

任务一：滑动窗口法卷积

卷积核函数

首先计算出当前CUDA线程计算的像素位置，每个线程计算一个输出图像中的特定像素。通过CUDA的内置变量 `blockIdx`、`blockDim` 和 `threadIdx` 计算当前线程处理的像素位置 `(x, y)` 以及通道 `c`，然后检查当前线程处理的像素位置和通道是否在图像的有效范围内。

最然后是**卷积计算**：对每个通道的3x3卷积核进行遍历：

- `kx` 和 `ky` 是卷积核的坐标。
- `kc` 是卷积核的通道索引。
- 计算指定步幅步幅 `stride` 时输入图像中对应位置的像素坐标 `(inX, inY)`。
- 对于每个卷积核位置 `(kx, ky, kc)`，将输入图像像素乘以卷积核权重，然后累加到 `sum` 变量中。

最后将累加的卷积结果 `sum` 存储到输出图像的对应位置，就可以的得到当前CUDA线程所要计算的像素位置的像素值。

```
__global__ void convolutionKernel(const float* input, float* output, const float*
kernel,
                          int width, int height, int channels, int paddedWidth, int
paddedHeight, int stride) {
    int x = blockIdx.x * blockDim.x + threadIdx.x;
    int y = blockIdx.y * blockDim.y + threadIdx.y;
    int c = blockIdx.z * blockDim.z + threadIdx.z;

    if (x < width && y < height && c < channels) {
        float sum = 0.0f;
        for (int kc = 0; kc < channels; ++kc) {
            for (int ky = 0; ky < 3; ++ky) {
                for (int kx = 0; kx < 3; ++kx) {
                    int inX = x * stride + kx;
                    int inY = y * stride + ky;
                    if (inX >= 0 && inX < paddedWidth && inY >= 0 && inY <
paddedHeight) {
                        sum += input[(inY * paddedWidth + inX) * channels + kc] *
kernel[c * 3 * 3 * 3 + (kc * 3 + ky) * 3 + kx];
                    }
                }
            }
        }
        output[(y * width + x) * channels + c] = sum;
    }
}
```

使用共享内存的滑动卷积核函数

使用CUDA并行库的共享内存来加速卷积操作，每个线程首先将其负责的输入图像数据块加载到共享内存中，包括必要的边缘数据，以确保卷积核能够正确处理边界。

每个线程将对应的输入图像数据加载到共享内存的中心位置，然后再使用同步线程 `__syncthreads()` 确保所有线程在加载数据到共享内存后再进行后续计算。

线程同步操作确保所有线程都完成共享内存的加载后再进行卷积计算，最终将卷积结果存储到输出图像的相应位置。

```
void convolutionWithDifferentStrides(const float* input, float* output, const
float* kernel, int width, int height, int channels,int stride, int block_size) {
    int newHeight = (height - 1) * stride + 3;
    int newWidth = (width - 1) * stride + 3;

    float* d_input, * d_output, * d_kernel;
    cudaMalloc((void**)&d_input, newWidth * newHeight * channels *
sizeof(float));
    cudaMalloc(&d_output, width * height * channels * sizeof(float));
    cudaMalloc(&d_kernel, 3 * 3 * 3 * 3 * sizeof(float));

    float* h_paddedInput = pad_image(input, width, height, channels, newWidth,
newHeight, 0);

    cudaMemcpy(d_input, h_paddedInput, newWidth * newHeight * channels *
sizeof(float), cudaMemcpyHostToDevice);
    cudaMemcpy(d_kernel, kernel, 3 * 3 * 3 * 3 * sizeof(float),
cudaMemcpyHostToDevice);

    dim3 blockSize(block_size, block_size, 3);
    dim3 gridSize((width + blockSize.x - 1) / blockSize.x, (height + blockSize.y
- 1) / blockSize.y, 1);

    convolutionKernel << <gridSize, blockSize >> > (d_input, d_output, d_kernel,
width, height, channels, newWidth, newHeight, stride);
    cudaDeviceSynchronize();

    cudaMemcpy(output, d_output, width * height * channels * sizeof(float),
cudaMemcpyDeviceToHost);

    // 释放显存
    delete[] h_paddedInput;
    cudaFree(d_input);
    cudaFree(d_output);
    cudaFree(d_kernel);
}
```

任务二：Img2col卷积

Img2col函数

Img2col卷积的思想是将输入图像数据重排为适合卷积操作的矩阵形式。可以在使用矩阵乘法实现卷积时用于加速卷积操作。主要将输入图像的数据重排成一个矩阵，其中每一列包含一个卷积核窗口的展开内容。每个线程根据索引 `idx` 计算其对应的输入图像的像素位置，最后将输入图像中相应位置的像素值复制到输出矩阵中。

```
__global__ void img2col(float* input, float* output, int h, int w, int stride) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    int k = 3;  // 卷积核大小

    int n = (h - k) / stride + 1;  // 输出矩阵的行数
    int m = (w - k) / stride + 1;  // 输出矩阵的列数
    int total_size = n * m * k * k * 3;
```

```
    if (idx < total_size) {
        int channel = idx % 3;
        int kx = (idx / 3) % k;
        int ky = (idx / 3 / k) % k;
        int w_out_idx = (idx / 3 / k / k) % m;
        int h_out_idx = (idx / 3 / k / k / m) % n;

        int w_in_idx = w_out_idx * stride + kx;
        int h_in_idx = h_out_idx * stride + ky;

        output[idx] = input[(h_in_idx * w + w_in_idx) * 3 + channel];
    }
}
```

实现矩阵相乘的核函数

计算矩阵乘法，首先计算出线程函数负责的像素位置，然后计算矩阵乘法循环的最里层的for循环，遍历矩阵 A的当前行和矩阵 B 的当前列对应的所有元素，计算它们的乘积并累加到 value上：

```
__global__ void matrixMulKernel(float* A, float* B, float* C, int size_1, int
size_2, int size_3) {
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    if (row < size_1 && col < size_3) {
        float value = 0.0f;
        for (int k = 0; k < size_2; k++) {
            value += A[row * size_2 + k] * B[k * size_3 + col];
        }
        C[row * size_3 + col] = value;
    }
}
```

任务三：cuDNN并行卷积

cuDNN计算卷积

首先创建并初始化cuDNN句柄：

```
cudnnHandle_t cudnn;
cudnnCreate(&cudnn);
```

然后读取输入图像，并将其从主机内存拷贝到GPU内存：

```
float* host_image = read_img("image.jfif");
int image_size = width * height * channels * sizeof(float);
float* device_image;
cudaMalloc(&device_image, image_size);
cudaMemcpy(device_image, host_image, image_size, cudaMemcpyHostToDevice);
```

初始化卷积核：

```
float* host_kernel = new float[kernel_size];
for (int i = 0; i < kernel_size; i++) {
    host_kernel[i] = static_cast<float>(rand()) / RAND_MAX;
}
float* device_kernel;
cudaMalloc(&device_kernel, kernel_size * sizeof(float));
cudaMemcpy(device_kernel, host_kernel, kernel_size * sizeof(float),
cudaMemcpyHostToDevice);
```

创建和设置卷积、输入、输出张量的描述符：

```
cudnnFilterDescriptor_t filter_desc;
cudnnCreateFilterDescriptor(&filter_desc);
cudnnSetFilter4dDescriptor(filter_desc, CUDNN_DATA_FLOAT, CUDNN_TENSOR_NCHW, 3,
3, 3, 3);

cudnnTensorDescriptor_t input_desc;
cudnnCreateTensorDescriptor(&input_desc);
cudnnSetTensor4dDescriptor(input_desc, CUDNN_TENSOR_NCHW, CUDNN_DATA_FLOAT, 1,
channels, height, width);

cudnnTensorDescriptor_t output_desc;
cudnnCreateTensorDescriptor(&output_desc);
cudnnSetTensor4dDescriptor(output_desc, CUDNN_TENSOR_NCHW, CUDNN_DATA_FLOAT, 1,
output_channels, output_height, output_width);

cudnnConvolutionDescriptor_t conv_desc;
cudnnCreateConvolutionDescriptor(&conv_desc);
cudnnSetConvolution2dDescriptor(conv_desc, 0, 0, 1, 1, 1, 1, CUDNN_CONVOLUTION,
CUDNN_DATA_FLOAT);
```

根据卷积描述符和输入描述符，计算输出张量的尺寸并分配相应的GPU内存：

```
int n, c, h, w;
cudnnGetConvolution2dForwardOutputDim(conv_desc, input_desc, filter_desc, &n, &c,
&h, &w);
float* device_output;
cudaMalloc(&device_output, n * c * h * w * sizeof(float));
```

然后选择卷积前向算法，并分配显存：

```
int returnedAlgoCount;
cudnnConvolutionFwdAlgoPerf_t algo;
cudnnFindConvolutionForwardAlgorithm(cudnn, input_desc, filter_desc, conv_desc,
output_desc, 1, &returnedAlgoCount, &algo);

size_t workspace_bytes = 0;
cudnnGetConvolutionForwardWorkspaceSize(cudnn, input_desc, filter_desc,
conv_desc, output_desc, algo.algo, &workspace_bytes);

void* d_workspace = nullptr;
cudaMalloc(&d_workspace, workspace_bytes);
```

最后使用选择的算法执行卷积操作，并将结果存储到输出张量中：

```
float alpha = 1.0f, beta = 0.0f;
cudnnConvolutionForward(cudnn, &alpha, input_desc, device_image, filter_desc,
device_kernel,conv_desc, algo.algo, d_workspace, workspace_bytes, &beta,
output_desc, device_output);
```

实验结果

(1) 线程块大小与图像大小影响

设stride=1

线程块大小\图像大小(N)	1024	2048	4096	8192
1	337.0713	572.9975	2461.6114	9695.7832
2	56.2780	217.3870	823.2858	3447.1588
4	30.0805	111.8365	427.4303	1920.9486
8	25.8119	98.0588	369.4751	1631.3011
16	25.1211	92.0363	382.5762	1607.7526
32	19.8384	80.2716	333.5827	1370.2600

(2) 使用共享内存影响

设stride=1

线程块大小\图像大小(N)	1024	2048	4096	8192
1	127.9533	502.0364	1950.2632	8063.1579
2	51.2823	179.9290	794.1919	3056.9137
4	31.3943	111.2477	432.8870	1756.0466
8	27.7025	101.6990	359.3886	1636.7647
16	15.6515	61.7697	215.8553	877.3529
32	15.7877	60.4892	224.356	885.625

(3) stride的影响

块大小设置为32，使用**共享内存的滑窗卷积**

Stride\图像大小(N)	1024	2048	4096	8192
1	14.5735	61.0964	217.9195	914.0635
2	26.1970	93.8791	469.1659	1548.5135
3	48.7537	179.1332	674.8383	7481.3160

(4) img2col卷积

设stride=1

线程块大小\图像大小(N)	1024	2048	4096	8192
1	15.8776	57.2387	243.8322	899.4230
2	16.4017	58.4376	220.2681	890.6807
4	15.6323	59.0895	219.6845	962.8820
8	14.5208	56.8537	232.2652	941.8152
16	15.4908	55.3477	233.0213	950.7712
32	14.3140	62.1345	232.7388	1021.6519

(5) CuDNN卷积与滑窗、共享内存、img2col对比

设stride=1，线程块大小32：

方法	1024	2048	4096	8192
滑窗	20.2874	74.0314	316.8060	1344.8506
共享内存滑窗	15.4239	57.8259	238.7087	877.9032
Img2col	15.1679	61.5189	239.8821	1038.0535
CuDNN考虑分配存储等额外时间	313.8976	750.4553	738.7458	2261.3152
CuDNN仅考虑卷积运算时间	0.0402	0.0454	0.0388	0.0377

结果分析

共享内存与Img2col加速效果比较：

- **共享内存**和**Img2col**都显著快于使用本地内存的滑窗法。
- 其中共享内存的加速效果更加显著。初步分析是因为共享内存允许线程块中的线程共享数据，减少了对全局内存的访问，因而减少了内存访问延迟。
- **Img2col**在小块大小的情况下加速效果明显优于共享内存方法。Img2col通过将卷积操作转换为矩阵乘法来提高计算效率，并且小块大小情况下，矩阵乘法能充分利用GPU的并行计算能力。

Stride大小对卷积时间的影响：

- 实验表明，卷积时间与stride大小呈正相关关系。即stride越大卷积时间越长。
- 这是因为stride越大，输出特征图尺寸会变小，为了保持输出尺寸相同，必须对输入图像进行更大的填充，增加了计算量，降低了计算效率。

CuDNN的使用效果：

- CuDNN实现的卷积运算本身效率极高，显示出极短的卷积运算时间（如实验数据中的CuDNN仅考虑卷积运算时间）。
- 然而，CuDNN需要的前期准备操作（如创建句柄、分配内存等）耗时较长，这些额外的操作时间会导致整体运行时间增加（如实验数据中的CuDNN考虑分配存储等额外时间）。

可能的改进方法

优化内存访问模式：

- **共享内存**：充分利用共享内存，减少全局内存访问次数。
- **保证访问顺序一致性：**确保数据访问是按顺序的，以减少内存访问冲突。
- **内存对齐**：确保数据在内存中是对齐的，以提高内存访问速度。

提高计算并行度：

- **线程块和网格配置**：合理配置线程块和网格的大小，以确保每个GPU都有足够的线程来充分利用计算资源。
- **并行化卷积操作**：将更多的卷积计算任务并行化，确保每个线程都能充分工作。

使用高级算法优化：

- **Winograd算法**：在一些特定的卷积核大小和数据情况下使用Winograd算法可以减少卷积计算的乘法次数。
- **FFT卷积**：使用快速傅里叶变换FFT进行卷积计算，可以在大尺寸卷积核情况下显著加速。

减少内存拷贝开销：

- **减少主机和设备之间的内存拷贝**：尽量减少CPU和GPU之间的数据传输，将数据尽可能长时间地保存在GPU内存中。
- **统一内存**：使用CUDA的统一内存来简化内存管理，并减少显式内存拷贝操作。

优化卷积算法：

- **图像块处理**：将图像分块处理，每个块独立进行卷积计算，可以减少内存带宽占用并提高缓存命中率。
- **循环展开**：对循环进行展开以减少循环控制开销，并利用更多的寄存器和共享内存。