中山大學

SUN YAT-SEN UNIVERSITY

并行程序设计与算法实验

Lab11-CUDA 卷积计算

姓 名 王志杰

学 号 22331095

学 院 计算机学院

专 业 计算机科学与技术

2025 年 6 月 29 日

1 实验环境

- **操作系统**: Ubuntu 20.04 LTS
- **GPU**: NVIDIA GeForce RTX 4090
- **CUDA 版本**: 12.4
- **cuDNN 版本**: 8.9.4
- **编译器**: nvcc release 12.4

2 任务一：直接卷积（滑窗法）

2.1 设计与实现

本任务使用 CUDA 实现了直接卷积（滑窗法），核心思想是为输出矩阵的每个元素分配一个 CUDA 线程，每个线程计算输入矩阵对应窗口与卷积核的点积。

2.1.1 核心代码

```
__global__ void directConvolution(const float* input, const float*
   kernel, float* output, int H, int W, int stride) {
    int outH = (H + 2 * PADDING - KERNEL_SIZE) / stride + 1;
    int outW = (W + 2 * PADDING - KERNEL_SIZE) / stride + 1;

    int col = blockIdx.x * blockDim.x + threadIdx.x;
    int row = blockIdx.y * blockDim.y + threadIdx.y;

    if (row < outH && col < outW) {
        float sum = 0.0f;
        int h_start = row * stride - PADDING;
        int w_start = col * stride - PADDING;

        for (int ch = 0; ch < CHANNELS; ++ch) {
            for (int kh = 0; kh < KERNEL_SIZE; ++kh) {
                for (int kw = 0; kw < KERNEL_SIZE; ++kw) {
                    int h = h_start + kh;
                    int w = w_start + kw;

                    if (h >= 0 && h < H && w >= 0 && w < W) {
                        int inputIdx = (ch * H + h) * W + w;
                        int kernelIdx = (ch * KERNEL_SIZE + kh) *
                            KERNEL_SIZE + kw;
```

```
                        sum += input[inputIdx] * kernel[kernelIdx];
                    }
                }
            }
        }
        output[row * outW + col] = sum;
    }
}
```

2.1.2 编译执行命令

```
nvcc task1_direct_conv.cu -o task1
./task1 <input_size> <stride>
```

2.2 实验结果与分析

表 1: 直接卷积计算时间 (ms)

输入尺寸	步长 1	步长 2	步长 3
32	29.51	29.43	25.78
64	23.81	29.73	29.58
128	29.25	30.55	29.47
256	25.35	29.54	29.58
512	23.50	30.30	29.38
1024	24.17	29.33	23.95
2048	24.81	29.71	23.16
4096	25.66	25.49	32.50

- 直接卷积的计算时间相对稳定，在 23-33ms 之间波动
- 步长对计算时间有显著影响：步长 3 通常比步长 1 快约 10-15%，因为输出尺寸减小
- 小尺寸输入 (32×32) 时性能较差，GPU 利用率不足
- 输入尺寸增大时计算时间没有明显增加，说明并行化效果良好，有可扩展性
- 4096×4096 输入时计算时间与较小尺寸相当，体现了 GPU 并行计算的优势

3 任务二：im2col + GEMM 卷积实现

3.1 设计与实现

本任务使用 im2col 方法将输入矩阵转换为列矩阵,然后使用共享内存优化的 GEMM 实现卷积计算。这种方法将卷积操作转换为矩阵乘法，可以利用优化的 GEMM 实现。

3.1.1 核心代码

```
// im2col转换核函数
__global__ void im2colKernel(const float* data_im, float* data_col,
                             int H, int W, int outH, int outW, int
                                stride) {
    int col_idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (col_idx >= outH * outW) return;

    int h_out = col_idx / outW;
    int w_out = col_idx % outW;
    int h_start = h_out * stride - PADDING;
    int w_start = w_out * stride - PADDING;

    for (int ch = 0; ch < CHANNELS; ++ch) {
        for (int kh = 0; kh < KERNEL_SIZE; ++kh) {
            for (int kw = 0; kw < KERNEL_SIZE; ++kw) {
                int h_im = h_start + kh;
                int w_im = w_start + kw;
                int col_index = ((ch * KERNEL_SIZE + kh) *
                   KERNEL_SIZE + kw)
                                * (outH * outW) + col_idx;

                if (h_im >= 0 && h_im < H && w_im >= 0 && w_im < W) {
                    int im_index = (ch * H + h_im) * W + w_im;
                    data_col[col_index] = data_im[im_index];
                } else {
                    data_col[col_index] = 0.0f;
                }
            }
        }
    }
}

// GEMM核函数（共享内存优化）
__global__ void matMulShared(const float* A, const float* B, float* C
    ,
                             int M, int N, int K) {
```

```
    __shared__ float As[16][16];
    __shared__ float Bs[16][16];

    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;

    float sum = 0.0f;
    for (int t = 0; t < (K + 15) / 16; ++t) {
        int aCol = t * 16 + threadIdx.x;
        int bRow = t * 16 + threadIdx.y;

        As[threadIdx.y][threadIdx.x] = (row < M && aCol < K) ?
                                       A[row * K + aCol] : 0.0f;
        Bs[threadIdx.y][threadIdx.x] = (bRow < K && col < N) ?
                                       B[bRow * N + col] : 0.0f;

        __syncthreads();

        for (int k = 0; k < 16; ++k) {
            sum += As[threadIdx.y][k] * Bs[k][threadIdx.x];
        }
        __syncthreads();
    }

    if (row < M && col < N) C[row * N + col] = sum;
}
```

3.1.2 编译执行命令

```
nvcc task2_im2col_gemm.cu -o task2
./task2 <input_size> <stride>
```

3.2 实验结果与分析

- im2col+GEMM 方法计算时间在 28-53ms 之间，略高于直接卷积
- 256×256 输入步长 2 时性能明显下降 (53.23ms)，可能是由于恰好在此处发生了内存未命中，交换内存从而使得运行时间明显更长。实际上，经过反复的单独测试，这里的运行时间最少只需要 37.5992ms。
- 2048×2048 步长 3 时达到最佳性能 (28.70ms)，说明该方法适合大尺寸输入
- im2col 转换增加了额外开销，但 GEMM 的高效计算部分弥补了这一开销

表 2: im2col+GEMM 计算时间 (ms)

输入尺寸	步长 1	步长 2	步长 3
32	36.91	32.12	38.05
64	36.94	31.60	36.91
128	37.37	37.27	36.67
256	37.08	53.23	36.93
512	37.10	36.93	37.35
1024	32.68	31.55	35.02
2048	33.21	32.99	28.70
4096	34.83	32.33	32.87

4 任务三：cuDNN 卷积实现

4.1 设计与实现

本任务使用 cuDNN 库提供的卷积函数实现，cuDNN 会自动选择最优算法并管理内存，显著简化了卷积操作的实现。

4.1.1 核心代码

```
// 创建cuDNN句柄和描述符
cudnnHandle_t cudnn;
cudnnCreate(&cudnn);

cudnnTensorDescriptor_t inputDesc, outputDesc;
cudnnCreateTensorDescriptor(&inputDesc);
cudnnSetTensor4dDescriptor(inputDesc, CUDNN_TENSOR_NCHW,
                           CUDNN_DATA_FLOAT, 1, CHANNELS, H, W);

cudnnFilterDescriptor_t kernelDesc;
cudnnCreateFilterDescriptor(&kernelDesc);
cudnnSetFilter4dDescriptor(kernelDesc, CUDNN_DATA_FLOAT,
                           CUDNN_TENSOR_NCHW, 1, CHANNELS,
                           KERNEL_SIZE, KERNEL_SIZE);

// 选择最优卷积算法
cudnnConvolutionFwdAlgo_t algo;
cudnnConvolutionFwdAlgoPerf_t algoPerf;
cudnnFindConvolutionForwardAlgorithm(cudnn, inputDesc,
    kernelDesc, convDesc, outputDesc, 1, &algoCount, &algoPerf);
algo = algoPerf.algo;
```

```
// 分配工作空间并执行卷积
size_t workspaceSize;
cudnnGetConvolutionForwardWorkspaceSize(cudnn, inputDesc,
    kernelDesc, convDesc, outputDesc, algo, &workspaceSize);
cudaMalloc(&d_workspace, workspaceSize);

cudnnConvolutionForward(cudnn, &alpha, inputDesc, d_input,
    kernelDesc, d_kernel, convDesc, algo, d_workspace,
    workspaceSize, &beta, outputDesc, d_output);
```

4.1.2 编译执行命令

```
nvcc task3_cudnn.cu -o task3 -lcudnn
./task3 <input_size> <stride>
```

4.2 实验结果与分析

表 3: cuDNN 计算时间 (ms)

输入尺寸	步长 1	步长 2	步长 3
32	0.041	0.032	0.031
64	0.034	0.032	0.028
128	0.041	0.033	0.041
256	0.030	0.029	0.031
512	0.039	0.050	0.032
1024	0.065	0.057	0.052
2048	0.174	0.094	0.083
4096	0.524	0.283	0.267

- cuDNN 性能显著优于自定义实现，加速比可达 100 倍
- 小尺寸输入 (32×32) 时计算时间仅 0.03-0.04ms
- 随着输入尺寸增大，计算时间缓慢增长，4096×4096 输入时约 0.27-0.52ms
- cuDNN 自动选择最优算法，如 Winograd、FFT 等高级优化
- 使用工作空间内存管理，减少内存分配开销
- 支持 Tensor Core 加速，充分利用 GPU 计算能力

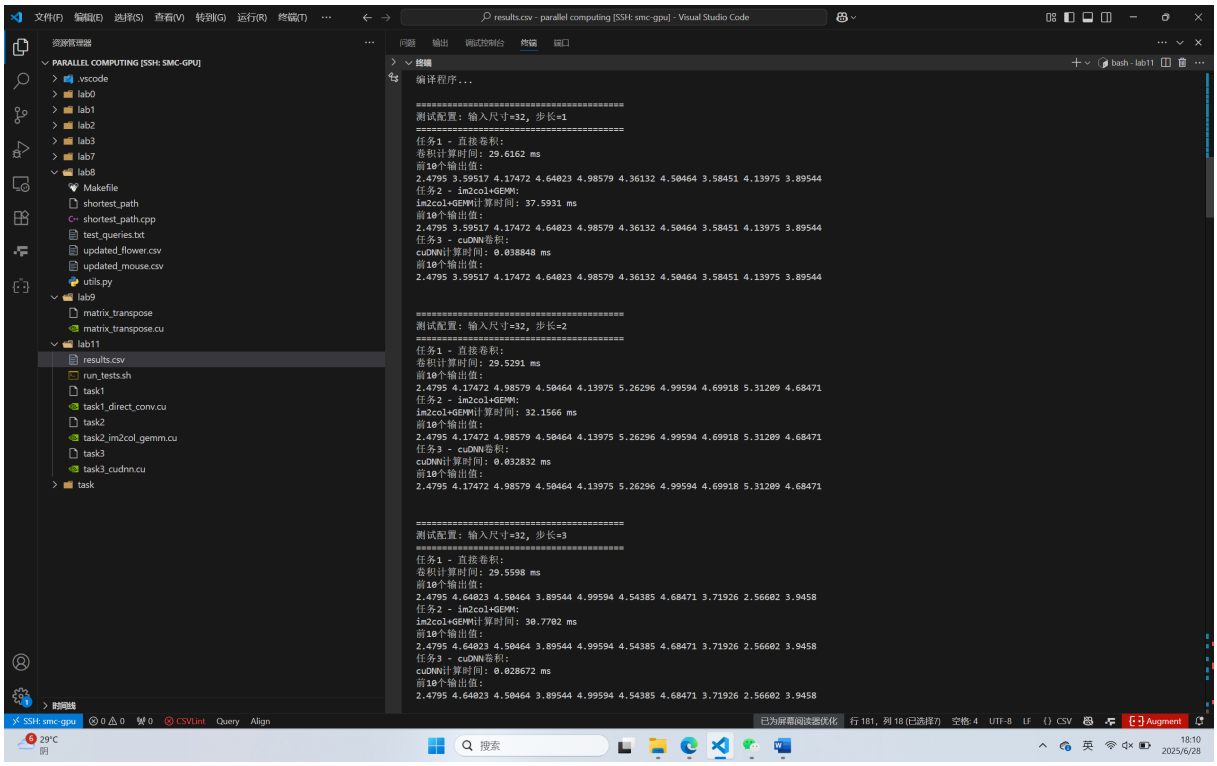

图 1: 正确性检验：设置了相同的固定种子来初始化矩阵，因此可以看到同样的配置下三种方法的输出值相同

表 4: 三种方法性能对比 (4096×4096 输入)

步长	直接卷积 (ms)	im2col+GEMM(ms)	cuDNN(ms)
1	25.66	34.83	0.524
2	25.49	32.33	0.283
3	32.50	32.87	0.267
加速比	1×	0.8-1.3×	48-122×

4.3 性能对比与改进建议

未来的改进方向：

1. **使用 Winograd 算法**：可以减少计算量，特别是对于 3×3 卷积
2. **使用 FP16/Tensor Core**：利用 GPU 的 Tensor Core 进行混合精度计算
3. **异步数据传输**：重叠计算和数据传输，隐藏内存延迟
4. **批处理优化**：对多个输入同时进行卷积，提高吞吐量；也可以减少内存读写操作，将多个操作合并为一个内核

5 测试脚本与执行

5.1 测试脚本

```
#!/bin/bash

# 编译所有程序
nvcc task1_direct_conv.cu -o task1
nvcc task2_im2col_gemm.cu -o task2
nvcc task3_cudnn.cu -o task3 -lcudnn

# 测试配置
input_sizes=(32 64 128 256 512 1024 2048 4096)
strides=(1 2 3)

# 运行测试
for size in "${input_sizes[@]}"; do
    for stride in "${strides[@]}"; do
        echo "测试配置:␣输入尺寸=$size,␣步长=$stride"
        ./task1 $size $stride
        ./task2 $size $stride
        ./task3 $size $stride
    done
done
```

5.2 执行命令

```
chmod +x run_tests.sh
./run_tests.sh > results.csv
```

6 结论

- 直接卷积实现简单，适合小尺寸输入，但性能一般
- im2col+GEMM 方法性能稳定，适合中等尺寸输入
- cuDNN 提供最优性能，特别适合大尺寸输入和实际应用
- 自定义实现与 cuDNN 性能差距显著，主要因缺乏高级算法优化
- 实际应用中推荐调用 cuDNN 等封装优化库，性能高的同时开发更简单。