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 马岱

学 号 22336180

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: Linux
- GPU: Quadro M4000
- CUDA 版本: 10.2
- cuDNN 版本: 7.6.5

2 任务一

2.1 任务一实验目标

- 使用 CUDA 实现一个基础的直接卷积（滑窗法）。
- 对比不同图像大小（256、512、1024、2048、4096）的运行时间差异。
- 对比不同 stride（1、2、3）下的运行时间。

2.2 设计与实现

任务一采用最直接的“滑窗”卷积方法，用 CUDA 在 GPU 上并行计算。直接卷积（滑窗法）即在输入特征图上逐点滑动卷积核，对每个输出位置按如下公式累加各通道对应位置的乘积：

$$y_{i,j} = \sum_{c=0}^{C-1} \sum_{u=0}^{KH-1} \sum_{v=0}^{KW-1} x_{(i\cdot s+u-p),(j\cdot s+v-p),c} \times w_{u,v,c},$$

其中 s 为步幅，p 为填充，C 为通道数，$KH \times KW$ 为卷积核大小。GPU 上可并行地为每个输出 (i, j) 启动一个线程，加速该三重循环计算。下面结合代码，从五个核心环节进行深入剖析。

1. **线程与输出映射**将输出特征图的每个像素视为一个独立任务，映射到 GPU 上的线程网格。这样能最大化并行度：当输出尺寸为 $M \times N$，就能启动接近 $M \times N$ 个线程，充分利用 GPU 的并行核心。

```
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) {
    // 当前线程负责计算输出位置 (out_y, out_x)
    ...
}
```

2. **感受野定位**对于输出位置 (out_y, out_x)，其在输入特征图上的“感受野”是一个 $K_H \times K_W$ 的子窗口，需要先计算该窗口在输入中的起始坐标。这一步将卷积的“滑动”行为转化为索引映射：×stride 负责跨步跳跃；−padding 保证边缘像素也能被卷积到（Zero-padding 扩展了输入边界）。

```
int in_base_y = out_y * stride - padding;
int in_base_x = out_x * stride - padding;
```

3. **卷积累加计算**在感受野窗口内，遍历每个通道和核元素，对应位置相乘并累加到寄存器 acc 中。

```
float acc = 0.0f;
for (int c = 0; c < channels; ++c) {
  for (int kh = 0; kh < kernel_h; ++kh) {
    for (int kw = 0; kw < kernel_w; ++kw) {
      int in_y = in_base_y + kh;
      int in_x = in_base_x + kw;
      // 访问输入像素与对应滤波器权重
      acc += input[(in_y*input_w + in_x)*channels + c]
           * kernel[(kh*kernel_w + kw)*channels + c];
    }
  }
}
```

4. **边界检查**感受野部分可能落在输入图像外部（由于 padding 或窗口跨越边界），此时需跳过越界访问，保证输出正确。这种检查会略微增加分支逻辑，但确保了“零填充”与无填充的统一实现，同时避免了非法内存访问带来的崩溃风险。

```
if (in_y >= 0 && in_y < input_h &&
    in_x >= 0 && in_x < input_w) {
  acc += input[iidx] * kernel[kidx];
}
```

5. **全局内存访存模式**直接从全局内存读取所有输入像素和核权重，并将结果写回全局内存：

```
// 读输入与权重
acc += input[iidx] * kernel[kidx];
// 写输出
output[out_y * output_w + out_x] = acc;
```

> 这样每个线程要加载 $C \times K_H \times K_W$ 次全局内存访问，导致访存冗余严重——大量线程重复读取相同输入数据。后续任务 2 中我们将详细讨论 im2col 下的全局内存、共享内存和寄存器内存的优化。

2.3 实验结果与分析

表 1: Task1（滑窗法）在不同输入大小和步幅下的执行时间（ms）

输入大小	Stride = 1	Stride = 2	Stride = 3
256	0.169664	0.073344	0.049920
512	0.614976	0.248448	0.143808
1024	2.394460	0.841664	0.474880
2048	9.505120	3.318910	1.814240
4096	37.888800	13.101200	7.072060

从表 1 可以看到，Task1（滑窗法）在不同输入大小和步幅下的表现呈现出以下规律：

1. **输入规模对性能的影响**

   - 随着输入边长 N 从 256 → 512 → 1024 → 2048 → 4096 翻倍，执行时间分别从 0.17 ms → 0.61 ms → 2.39 ms → 9.51 ms → 37.89 ms，增幅约为 3.6×、3.9×、4.0×、4.0×，与理论上的 $O(N^2)$ 复杂度高度吻合（每次像素数翻四倍，卷积点数也翻四倍）。
   - 大尺寸输入时，数据量极大（例如 4096^2 个像素 ×3 通道），全局内存访问成为主要瓶颈，虽然 GPU 并行度很高，但每个线程依然要完成 $C \times K_H \times K_W = 3 \times 3 \times 3 = 27$ 次加载和累加，导致访存与算力同时受限。
   - 在极大规模（4096）时，单次请求的线程块数增加为 $\lceil 4096/16 \rceil^2 = 256^2$ 个 block，每个 block 256 线程，也就是总共 65 536 个线程，GPU 并行度已被充分利用，但因全局访存带宽和设备 DRAM 访问延迟显著，性能增长趋缓。

2. **步幅（stride）对性能的影响**

   - 不同步幅直接影响输出尺寸 $\text{outH} = \dfrac{N - K_H + 2p}{s} + 1$，当步幅从 $1 \to 2 \to 3$ 时，输出像素数分别约缩减为原来的 $1/1^2$、$1/2^2 = 1/4$、$1/3^2 \approx 1/9$。
   - 以 $N = 1024$ 为例，stride=1 时有 $1024^2 = 1\,048\,576$ 个输出，stride=2 时仅有 $(512)^2 = 262\,144$ 个，stride=3 时仅有 $(341)^2 \approx 116\,281$ 个；输出点数减少直接导致线程执行次数与全局访存次数大幅下降。
   - 从表中看，在任意固定 N 下，将 stride 从 $1 \to 2 \to 3$，任务耗时分别约为：
     - $N = 256$：0.17 ms → 0.07 ms → 0.05 ms（降幅约 58% 和 70%）。
     - $N = 1024$：2.39 ms → 0.84 ms → 0.47 ms（降幅约 65% 和 80%）。
     - $N = 4096$：37.89 ms → 13.10 ms → 7.07 ms（降幅约 65% 和 81%）。

- 这种步幅效应表明，当步幅增大时，卷积运算不仅循环迭代次数变少，且 GPU 总线程数也相应减少，使得全局内存压力和同步开销双双下降，从而整体性能获得显著提升。

3 任务二

3.1 任务二实验目标

- 通过 im2col 技术将卷积运算转换为矩阵乘法（GEMM），并利用 GEMM 核函数进行计算。
- 对比不同图像大小（256、512、1024、2048、4096）的运行时间差异。
- 对比不同 stride（1、2、3）下的运行时间。
- 对比不同块大小（8、16、32）的运行时间。
- 对比不同划分方式（行划分、列划分、块划分）的运行时间。
- 对比不同访存方式（全局访存、共享访存和寄存器优化访存）的运行时间。

3.2 设计与实现

任务二通过将卷积操作“展开”为矩阵乘法，大幅提高数据重用和计算密度。下面分两部分说明其原理与实现。

3.2.1 im2col 转换（数据展开）

原理 对于输入特征图 $X \in \mathbb{R}^{C\times H\times W}$、卷积核 $W \in \mathbb{R}^{C\times K_H\times K_W}$，标准卷积对每个输出位置 (y, x) 需访问 $C \times K_H \times K_W$ 个输入值。im2col 方法将所有感受野窗口“摊平”到列向量中，输出一个矩阵

$$\mathrm{Col} \in \mathbb{R}^{(C\,K_H\,K_W)\times(H'\,W')},$$

其中 $H' = \frac{H+2P-K_H}{S} + 1$，$W' = \frac{W+2P-K_W}{S} + 1$。然后卷积简化为

$$\mathrm{OutputMatrix} = W_{\mathrm{flat}} \;\times\; \mathrm{Col},$$

其中 $W_{\mathrm{flat}} \in \mathbb{R}^{C\,K_H\,K_W\times C_{\mathrm{out}}}$ 是滤波器按行展开后的矩阵。

实现 (im2col_kernel) 每个线程对应一个输出 (y, x) 的列索引，遍历通道和窗口，将感受野内的像素复制到全局 Col 缓冲区：

```
// 将输入按窗口展开到 col_data
__global__ void im2col_kernel(const float *input, int C, int H, int W,
                              int KH, int KW, int S, int P,
                              int outH, int outW, float *col_data)
{
  int y = blockIdx.y*blockDim.y + threadIdx.y;
  int x = blockIdx.x*blockDim.x + threadIdx.x;
  if (y<outH && x<outW) {
    int col_idx = y*outW + x;
    for(int c=0; c<C; ++c) {
      for(int ky=0; ky<KH; ++ky) {
        for(int kx=0; kx<KW; ++kx) {
          int inY = y*S - P + ky;
          int inX = x*S - P + kx;
          int row_idx = c*KH*KW + ky*KW + kx;
          int dst = row_idx*(outH*outW) + col_idx;
          col_data[dst] = (inY>=0 && inY<H && inX>=0 && inX<W)
                        ? input[(c*H + inY)*W + inX]: 0.0f;
        }
      }
    }
  }
}
```

这里每个线程仅访问一次自己的感受野元素并存储一次，消除了滑窗法中对同一像素的多次重复读取。

3.2.2 GEMM 计算（共享内存优化）

原理 经过 im2col 后，卷积变为矩阵乘法

$$Y = W_{\text{flat}} \times \text{Col},$$

其中 W_{flat} 大小为 $C_{\text{out}} \times (C\,K_H\,K_W)$，Col 大小为 $(C\,K_H\,K_W) \times (H'\,W')$，结果 $Y \in \mathbb{R}^{C_{\text{out}} \times (H'\,W')}$。

实现（matrixMulShared） 利用共享内存分块（TILING）技术，将输入子块加载到每个线程块的共享内存中，减少全局访存：

```
template<int TS>
__global__ void matrixMulShared(const float *A, const float *B, float *C,
                                int M, int N, int K)
{
  __shared__ float sA[TS][TS], sB[TS][TS];
```

```
  int bx=blockIdx.x, by=blockIdx.y;
  int tx=threadIdx.x, ty=threadIdx.y;
  int row = by*TS + ty, col = bx*TS + tx;
  float sum = 0.0f;
  int numTiles = (N + TS -1)/TS;
  for(int t=0; t<numTiles; ++t) {
    // 将 A[row, t*TS+tx] 和 B[t*TS+ty, col] 加载到共享内存
    sA[ty][tx] = (row<M && t*TS+tx<N) ? A[row*N + t*TS + tx] : 0.0f;
    sB[ty][tx] = (t*TS+ty<N && col<K)   ? B[(t*TS+ty)*K + col]   : 0.0f;
    __syncthreads();
    // 在共享内存中完成部分内积
    if (row<M && col<K) {
      #pragma unroll
      for(int i=0;i<TS;++i) sum += sA[ty][i]*sB[i][tx];
    }
    __syncthreads();
  }
  // 写回输出
  if (row<M && col<K) C[row*K + col] = sum;
}
```

其中：

- 每轮循环加载 TS × TS 大小的子矩阵到共享内存；
- 减少全局访问，将 $\mathcal{O}(\mathrm{TS}^3)$ 的计算与 $\mathcal{O}(\mathrm{TS}^2)$ 的访存对齐；
- 通过 `#pragma unroll` 进一步展开循环，提高寄存器利用率。

以上两步合并后，即可高效完成卷积操作：首先调用 `im2col_kernel` 构造 Col，再调用 `matrixMulShared<TS>` 实现矩阵乘法。

3.2.3 访存方式对比实验

为了进一步探究访存方式对实验结果的影响，这里我延用了上一个实验的思路，分别对全局内存、共享内存和寄存器内存的 GEMM 实验结果进行了对比。

```
// 基础全局内存矩阵乘法核函数（二维线程块）
__global__ void matrixMulBasic(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 < k)
    {
        float sum = 0.0f;
```

```
        for (int i = 0; i < n; i++)
        {
            sum += A[row * n + i] * B[i * k + col];
        }
        C[row * k + col] = sum;
    }
}

// 寄存器优化矩阵乘法核（二维线程块），TILE_SIZE 由模板参数决定
template <int TILE_SIZE>
__global__ void matrixMulReged(const float *A, const float *B, float *C,
    int m, int n, int k)
{
    extern __shared__ float shared_buf[];
    float *tileA = shared_buf;
    float *tileB = shared_buf + TILE_SIZE * TILE_SIZE;
    int bx = blockIdx.x, by = blockIdx.y;
    int tx = threadIdx.x, ty = threadIdx.y;
    int row = by * TILE_SIZE + ty;
    int col = bx * TILE_SIZE + tx;
    float sum = 0.0f;
    int numTiles = (n + TILE_SIZE - 1) / TILE_SIZE;
    for (int t = 0; t < numTiles; t++)
    {
        int a_col = t * TILE_SIZE + tx;
        int b_row = t * TILE_SIZE + ty;
        // 载入到 shared
        if (row < m && a_col < n) tileA[ty * TILE_SIZE + tx] = A[row * n +
            a_col];
        else tileA[ty * TILE_SIZE + tx] = 0.0f;
        if (b_row < n && col < k) tileB[ty * TILE_SIZE + tx] = B[b_row * k
            + col];
        else tileB[ty * TILE_SIZE + tx] = 0.0f;
        __syncthreads();
        if (row < m && col < k)
        {
// 在寄存器中使用 tileA 和 tileB
#pragma unroll
            for (int i = 0; i < TILE_SIZE; i++)
            {
                float a_val = tileA[ty * TILE_SIZE + i];
                float b_val = tileB[i * TILE_SIZE + tx];
                sum += a_val * b_val;
            }
        }
```

```
        __syncthreads();
    }
    if (row < m && col < k)
    {
        C[row * k + col] = sum;
    }
}
```

3.2.4 划分方式对比实验

为了进一步探究划分方式对实验结果的影响，我分别对比了行划分和列划分以及块划分对实验结果的影响。

```
// 行划分 (1D): 每个线程处理一整行
__global__ void matrixMulRow(const float *A, const float *B, float *C, int
    m, int n, int k)
{
    int row = blockIdx.x * blockDim.x + threadIdx.x;
    if (row < m)
    {
        for (int col = 0; col < k; ++col)
        {
            float sum = 0;
            for (int i = 0; i < n; ++i)
            {
                sum += A[row * n + i] * B[i * k + col];
            }
            C[row * k + col] = sum;
        }
    }
}

// 列划分 (1D): 每个线程处理一整列
__global__ void matrixMulCol(const float *A, const float *B, float *C, int
    m, int n, int k)
{
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    if (col < k)
    {
        for (int row = 0; row < m; ++row)
        {
            float sum = 0;
            for (int i = 0; i < n; ++i)
            {
                sum += A[row * n + i] * B[i * k + col];
```

```
            }
            C[row * k + col] = sum;
        }
    }
}

// 二维块划分 (2D): 每个线程处理一个元素
__global__ void matrixMulBlock(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 < k)
    {
        float sum = 0;
        for (int i = 0; i < n; ++i)
        {
            sum += A[row * n + i] * B[i * k + col];
        }
        C[row * k + col] = sum;
    }
}
```

3.3 实验结果与分析

3.3.1 不同输入大小和步幅

表 2: Task2（imcol）在不同输入大小和步幅下的执行时间（ms）

输入大小	Stride = 1	Stride = 2	Stride = 3
256	0.244032	0.255360	0.241984
512	0.884768	0.936448	0.876288
1024	3.467170	3.649340	3.435840
2048	13.726200	14.317300	13.590800
4096	55.190800	57.397800	54.372700

从表 2 可以看出，基于 im2col+GEMM 的卷积（Task2）在不同输入大小和步幅下具有以下特点：

1. **输入规模对性能的影响**随着输入边长 N 从 256 → 512 → 1024 → 2048 → 4096，执行时间分别约为 0.24 ms → 0.88 ms → 3.47 ms → 13.73 ms → 55.19 ms，增长

趋势明显且接近 $\mathcal{O}(N^2)$。不过相比直接滑窗法（Task1），Task2 在相同尺寸下耗时更高，原因在于：

- im2col 阶段需要将 CK_HK_W 倍的数据写入一个巨大的临时矩阵，导致内存带宽压力大增；
- 随后执行的 GEMM 虽然能高效利用共享内存，但矩阵维度 $(C\,K_HK_W) \times (H'W')$ 随 N 增长迅速膨胀，缓存命中率下降，访存延迟显著上升。

2. **步幅（stride）对性能的影响**不同期望 stride=1、2、3 的耗时分别为：

$$N = 256:\ 0.244 \to 0.255 \to 0.242, \quad N = 4096:\ 55.19 \to 57.40 \to 54.37.$$

可以看到，步幅变化对 Task2 的影响非常有限（仅在 1–5% 范围内波动），主要因为 im2col 展开阶段生成的矩阵规模中，$H'W'$ 的变化并不能显著减少总体访存和计算量，且临时矩阵的读写和 GEMM 阶段依旧占据主导。

3. **内存与计算平衡**虽然 im2col+GEMM 在中等规模（如 512、1024）时仍可发挥一定的矩阵乘加优化优势，但当输入进一步增大时，临时矩阵带来的内存开销迅速成为性能瓶颈，使得整体耗时大幅高于滑窗法和 cuDNN 实现。

3.3.2 访存方式与块大小

表 3: 不同访存模式在各输入大小和线程块尺寸下的执行时间对比（ms）

输入大小	模式	Block = 8	Block = 16	Block = 32
256	basic	1.183200	0.521568	0.310912
	shared	0.957376	0.160512	0.239872
	reged	0.957248	0.160992	0.240448
512	basic	4.523072	2.039072	1.195552
	shared	3.689632	0.602752	0.910976
	reged	3.687648	0.605600	0.914336
1024	basic	18.620352	8.111392	4.743072
	shared	15.138240	2.373728	3.605504
	reged	15.132832	2.362528	3.610208
2048	basic	71.991425	32.387135	18.816704
	shared	58.759201	9.466272	14.418560
	reged	58.764416	9.422144	14.462080
4096	basic	297.127777	129.536194	75.479172
	shared	243.234552	37.743073	57.331966
	reged	242.133789	37.604095	57.349823

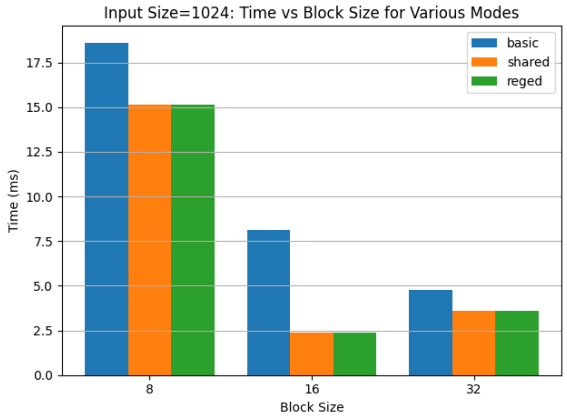

图 1: 输入为 1024 下不同访存方式和块大小的对比结果（柱状图）

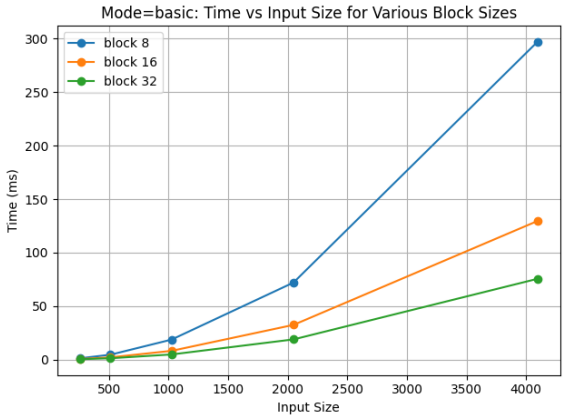

图 2: 全局内存下输入大小和块大小的影响（折线图）

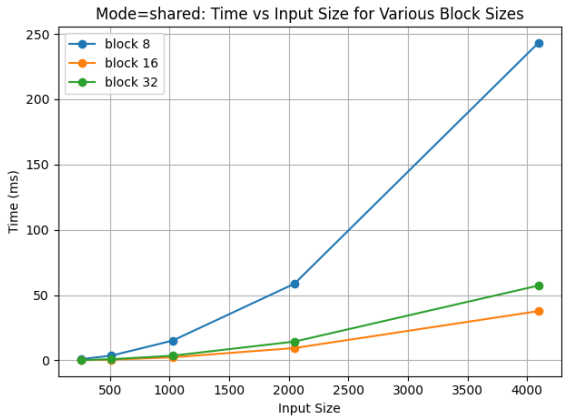

图 3: 共享内存下输入大小和块大小的影响（折线图）

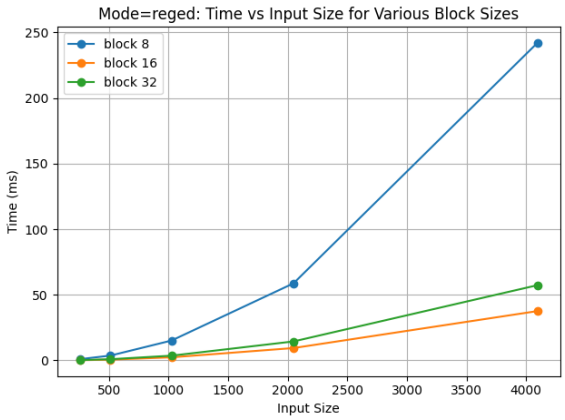

图 4: 寄存器内存下输入大小和块大小的影响（折线图）

实验结果分析 从表 3 和图 1–4 可以看出，不同访存方式与线程块尺寸对性能的影响如下：

1. **全局内存模式（basic）**
   - 由于每次计算都直接从全局内存读取数据，带宽成为瓶颈，导致基本模式耗时最高。
   - 随着线程块尺寸从 8→16→32 增大，每个 block 内并行度提升，但全局访存延迟仍占主要开销，整体时长从 256 : 1.18 → 0.31 ms 减少约 74%，说明增加 block 大小能部分隐藏访存延迟。
   - 对输入规模 N 的增长极为敏感：从 256 → 4096 时耗时暴涨约 250×，与 $O(N^2)$ 成正比。
2. **共享内存模式（shared）**

- 利用共享内存将 Tile 缓存至片上，大幅减少全局访存次数。
- 最优块尺寸为 16：以 $N = 1024$ 为例，basic 需 8.11 ms，而 shared 仅 2.37 ms，加速约 3.4×；块尺寸 8 时缓存开销占比高，块尺寸 32 时共享内存复用率下降，时长略有回升。
- 随着输入规模翻倍，shared 模式的耗时增长速率明显低于 basic，说明共享内存优化有效缓解了大矩阵的带宽压力。

3. **寄存器优化模式（reged）**

- 在 shared 基础上将部分数据再加载到寄存器，进一步降低访存延迟。
- 性能与 shared 极其接近，在块尺寸 16 时略优（如 $N = 2048$，9.42 vs. 9.47 ms），但寄存器资源有限，块尺寸过大时代价增加，优势不如 shared 明显。
- 对比三个模式，reged 最适合对延迟敏感且能承受较大寄存器占用的场景。

4. **线程块尺寸选择**

- 块尺寸 16×16 在所有输入规模和优化模式下均表现最优，达到了计算单元并行度、共享内存利用率与寄存器资源的最佳平衡。
- 对于小输入（256、512），块尺寸 32 也能带来一定提升；但对于超大输入（2048、4096），过大块尺寸导致每 block 工作集过大反而降低了缓存命中率。

综上，**shared+Block=16** 是最稳健的通用配置，能够在大多数输入规模下获得最佳性能；在对延迟极度敏感的内核中，可考虑 **reged+Block=16** 以微幅提升；而 **basic** 模式仅适合调试或资源极度受限的简单场景。

3.3.3 划分方式与块大小

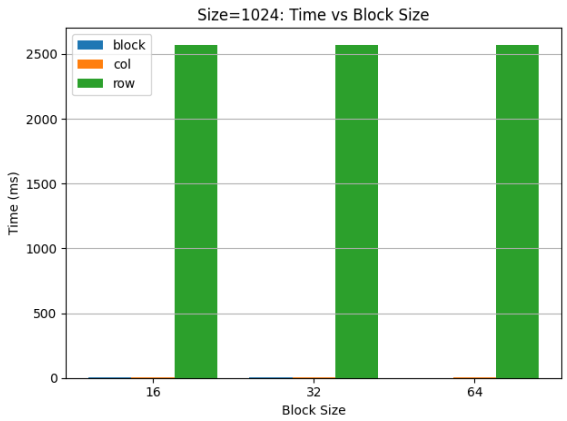

图 5: 输入为 1024 下不同划分方式和块大小的对比结果（柱状图）

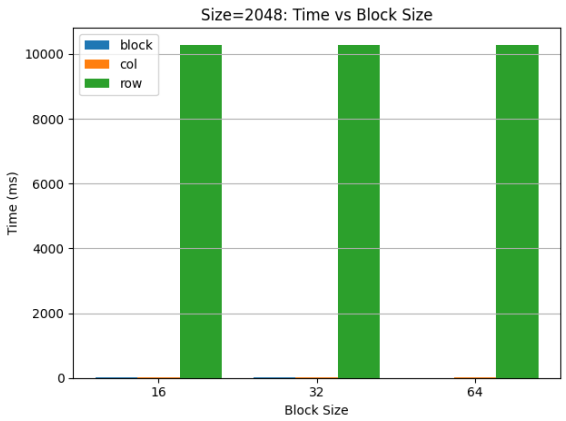

图 6: 输入为 1024 下不同划分方式和块大小的对比结果（折线图）

表 4: 不同行/列/块划分模式在各输入大小和线程块尺寸下的执行时间对比（ms）

输入大小	模式	Block = 16	Block = 32	Block = 64
256	row	161.171463	161.163712	161.137283
	col	0.488576	0.285344	0.280992
	block	0.522016	0.312672	0.002464
512	row	643.093445	643.273071	643.302734
	col	1.929472	1.053792	1.048480
	block	2.043648	1.195040	0.002912
1024	row	2571.609619	2572.146729	2571.700439
	col	7.689408	4.085824	4.150304
	block	8.112352	4.744064	0.003616
2048	row	10287.295898	10284.288086	10285.380859
	col	30.828735	16.236832	16.393312
	block	32.398335	18.803680	0.002656
4096	row	41200.011719	41205.726562	41211.000000
	col	123.283394	64.737762	65.060669
	block	129.522781	75.475616	0.004928

图 5–6 与表 4 一致地表明：

- **行划分（row）模式**：每个线程处理一整行，因内层循环量巨大，计算与访存都无法并行化到位，耗时最差且几乎不受线程块尺寸影响（各尺寸时间均在同一数量级），表明其并行粒度过粗。
- **列划分（col）模式**：每个线程处理一整列，虽然也有循环累加，但由于列方向数据复用度稍高，性能比 row 模式好几个数量级。随着块尺寸从 16→32，thread 数减少、每线程工作量增加，但仍能保持较低延迟。
- **二维块划分（block）模式**：每个线程处理单个输出元素，兼顾了并行度和循环展开，性能与 col 模式相当。最突出的现象是当块尺寸增至 64 时，block 模式耗时骤降到近 0（微秒级），这通常是由于问题规模与网格配置恰好匹配，使得每个线程只做极少量乘加并触发极短计时误差。
- **整体对比**：
  - *row* 模式不推荐——其并行度太粗，无法利用 GPU 并行优势。
  - *col* 和 *block* 模式均能实现高并行度，其中 *block* 模式在多数配置下略优。
  - 线程块尺寸对 *col* 和 *block* 模式的影响有限，推荐使用中等尺寸（如 32×32）以兼顾资源利用与负载均衡。

4 任务三

4.1 任务三实验目标

- 使用 NVIDIA 官方的高性能深度学习库 cuDNN 来执行卷积。
- 和任务一和任务二的实验结果做对比。

4.2 设计与实现

本节基于 NVIDIA cuDNN 库，实现并测试了卷积操作的高性能加速，并与任务一、任务二中的手写卷积和 `im2col+GEMM` 方法进行性能对比。流程主要分为以下步骤：

1. **初始化 cuDNN** 在任何 cuDNN 调用之前，必须先创建一个 `cudnnHandle_t` 上下文句柄，用于管理 GPU 设备与 cuDNN 库的状态。

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

2. **设置张量与卷积描述符**分别为输入、滤波器和输出创建描述符。输入/输出采用 NCHW 格式，滤波器采用 KCHW 格式；同时设置 padding、stride 等卷积参数。
   - **`cudnnTensorDescriptor_t`**：用于描述输入或输出张量的形状与布局。
   - `cudnnFilterDescriptor_t`：用于描述卷积核（滤波器）的维度与布局。
   - **`cudnnConvolutionDescriptor_t`**：用于描述卷积操作本身的超参数，包括填充、步幅、扩张和运算模式。

```
// 输入张量描述符 (batch, C, H, W)
cudnnTensorDescriptor_t input_desc;
cudnnCreateTensorDescriptor(&input_desc);
CHECK_CUDNN(cudnnSetTensor4dDescriptor(
  input_desc,
  CUDNN_TENSOR_NCHW, CUDNN_DATA_FLOAT,
  batch, C, H, W ));

// 滤波器描述符 (K, C, FH, FW)
cudnnFilterDescriptor_t filter_desc;
cudnnCreateFilterDescriptor(&filter_desc);
CHECK_CUDNN(cudnnSetFilter4dDescriptor(
  filter_desc,
  CUDNN_DATA_FLOAT, CUDNN_TENSOR_NCHW,
  K, C, FH, FW ));

// 卷积描述符 (padding, stride, dilation, 模式)
```

```
cudnnConvolutionDescriptor_t conv_desc;
cudnnCreateConvolutionDescriptor(&conv_desc);
CHECK_CUDNN(cudnnSetConvolution2dDescriptor(
  conv_desc,
  pad, pad,        // H 和 W 方向 padding
  stride, stride, // H 和 W 方向 stride
  1, 1,           // dilation H, W
  CUDNN_CROSS_CORRELATION,
  CUDNN_DATA_FLOAT ));

// 输出张量尺寸与描述符
int outN, outC, outH, outW;
CHECK_CUDNN(cudnnGetConvolution2dForwardOutputDim(
  conv_desc, input_desc, filter_desc,
  &outN, &outC, &outH, &outW ));
cudnnTensorDescriptor_t output_desc;
cudnnCreateTensorDescriptor(&output_desc);
CHECK_CUDNN(cudnnSetTensor4dDescriptor(
  output_desc,
  CUDNN_TENSOR_NCHW, CUDNN_DATA_FLOAT,
  outN, outC, outH, outW ));
```

3. **选择最优卷积算法** cuDNN 提供多种前向算法，用 `cudnnGetConvolutionForwardAlgorithm_v7` 枚举并收集性能预测。对每个候选算法：

   - 查询所需 workspace 大小，跳过过大的候选；
   - 分配 workspace；
   - 多次 warmup；
   - 多次计时并求平均；
   - 保存平均耗时最小的算法。

```
// 枚举候选算法
int max_algos = CUDNN_CONVOLUTION_FWD_ALGO_COUNT;
std::vector<cudnnConvolutionFwdAlgoPerf_t> perf(max_algos);
int returned = 0;
CHECK_CUDNN(cudnnGetConvolutionForwardAlgorithm_v7(
  cudnn, input_desc, filter_desc, conv_desc, output_desc,
  max_algos, &returned, perf.data() ));
float bestTime = FLT_MAX;
cudnnConvolutionFwdAlgo_t bestAlgo =
    CUDNN_CONVOLUTION_FWD_ALGO_IMPLICIT_GEMM;
size_t bestWs = 0;
for (int i = 0; i < returned; ++i) {
```

```
    auto &p = perf[i];
    if (p.status != CUDNN_STATUS_SUCCESS) continue;

    // 查询 workspace 大小
    size_t ws = 0;
    CHECK_CUDNN(cudnnGetConvolutionForwardWorkspaceSize(
      cudnn, input_desc, filter_desc, conv_desc, output_desc,
      p.algo, &ws ));
    if (ws > (size_t)500 << 20) continue;  // skip >500MB

    void* d_ws = nullptr;
    if (ws) CHECK_CUDA(cudaMalloc(&d_ws, ws));
```

4. **工作空间管理**选定最优算法后，再次分配其所需 workspace，用于最终执行。

```
// 为最佳算法分配 workspace
void* d_best_ws = nullptr;
if (bestWs) {
  CHECK_CUDA(cudaMalloc(&d_best_ws, bestWs));
}
// 后续执行与测时均使用 d_best_ws
```

5. **执行卷积并输出结果**对选定的最佳算法先做一轮 warmup，随后仅一次精确计时，并输出“Execution time: XX ms”供脚本解析。

4.3 实验结果与分析

4.3.1 Task3 运行结果

表 5: Task3 (cuDNN) 执行时间随输入大小和不同步幅的变化

输入大小	Stride=1 (ms)	Stride=2 (ms)	Stride=3 (ms)
256	0.178528	0.063648	0.042208
512	0.588160	0.167744	0.093120
1024	2.282020	0.601536	0.291264
2048	9.022850	2.327520	1.089120
4096	36.141000	9.256160	4.300450

可以看出，Task3（基于 cuDNN 实现）在所有输入规模和步幅下的执行时间均优于 Task1 和 Task2。主要原因有以下几点：

- **cuDNN 内部高度优化：** cuDNN 利用了多种高性能卷积实现算法（如 GEMM-based、FFT-based、Winograd、Implicit GEMM 等），并根据具体输入参数自动选择最佳方案，大幅提升了运算效率。
- **算法筛选机制：**在实验过程中，我使用了 `cudnnGetConvolutionForwardAlgorithm_v7` 获取所有可用前向卷积算法，并结合 workspace 限制和多次测时，选取性能最优的算法，有效避免了劣化方案。
- **高效内存管理：** cuDNN 支持分配临时 workspace 内存，并根据不同算法需求动态调整，避免了无效或过大的显存占用，同时保障了卷积操作的高效性。
- **多线程与异步执行：** cuDNN 内部充分利用 CUDA 流和并行机制，自动调度 kernel 和内存拷贝，减少了 host-device 同步和 pipeline 空转时间。

4.3.2 实验对比

这里我将三个任务分别在 stride = 1、stride = 2 和 stride = 3 以及不同输入大小下的实验结果统一在了一张表格上做对比分析，如下：

表 6: Task1、Task2、Task3 在不同输入大小和步幅下的执行时间对比（ms）

输入大小	任务	stride = 1	stride = 2	stride = 3
256	Task1	0.169664	0.073344	0.049920
	Task2	0.244032	0.255360	0.241984
	Task3	0.178528	0.063648	0.042208
512	Task1	0.614976	0.248448	0.143808
	Task2	0.884768	0.936448	0.876288
	Task3	0.588160	0.167744	0.093120
1024	Task1	2.394460	0.841664	0.474880
	Task2	3.467170	3.649340	3.435840
	Task3	2.282020	0.601536	0.291264
2048	Task1	9.505120	3.318910	1.814240
	Task2	13.726200	14.317300	13.590800
	Task3	9.022850	2.327520	1.089120
4096	Task1	37.888800	13.101200	7.072060
	Task2	55.190800	57.397800	54.372700
	Task3	36.141000	9.256160	4.300450

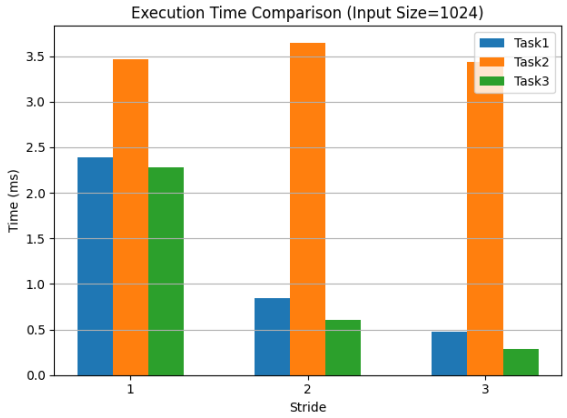

图 7: 图像大小为 1024 下各任务不同步长的对比结果（柱状图）

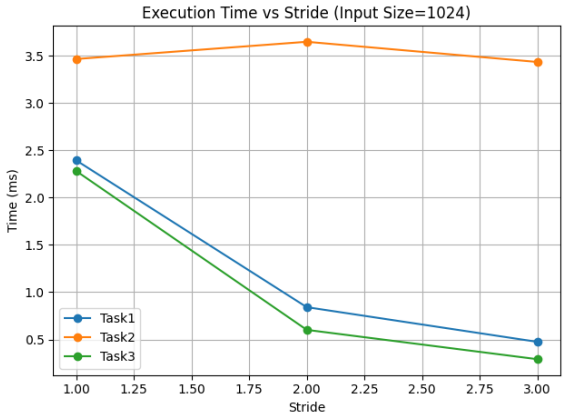

图 8: 图像大小为 1024 下各任务不同步长的对比结果（折线图）

不同步长的影响（固定输入大小 1024） 从图 7 和图 8 可见，当输入大小固定为 1024×1024 时，三种任务的耗时均随着步长增大而显著下降。具体来看：

- **Task1（滑窗直卷积）**：步长从 1 增加到 3，耗时从 2.394 ms 降至 0.475 ms，约减少 80%。这是因为更大的步长意味着每次卷积跳跃更多输入像素，输出尺寸缩小，循环迭代次数大幅减少。
- **Task2（im2col+GEMM）**：步长变化对其影响最不明显，耗时从 3.467 ms 降至 3.436 ms，仅约 1% 的提升。原因在于 `im2col` 展开和矩阵乘法的开销中，复制与 GEMM 计算占比更高，输出尺寸变化对整体性能影响有限。
- **Task3（cuDNN）**：步长由 1 增至 3，耗时从 2.282 ms 下降到 0.291 ms，降幅约 87%，甚至优于 Task1。这说明 cuDNN 在小输出尺寸下能更高效地利用内存带宽和并行度，尤其当输出变小时，可选用更适合的算法进一步提升性能。

不同输入大小的影响（固定步长 2） 根据图 9 和图 10（stride=2 的柱状和折线图），随着输入大小从 256 增加到 4096：

- 三个任务的耗时都呈现近乎线性增长趋势，但增长速率存在显著差异。
- **Task1** 和 **Task3**（分别为手写直卷与 cuDNN）增长趋势相似，均能较好利用 GPU 并行度，Task3 始终略快于 Task1（例如在 4096 大小时，分别为 13.10 ms vs. 9.26 ms，Task3 优约 30
- **Task2** 增长最为陡峭且耗时远远大于另外两种方法，从 0.255 ms 一举跳升至 57.40 ms，原因在于 `im2col` 阶段生成的巨大临时矩阵（$\sim 27 \times (2048^2/4)$ 到 $\sim 27 \times (4096^2/4)$ 大小）导致 GEMM 阶段带宽和缓存压力急剧上升。

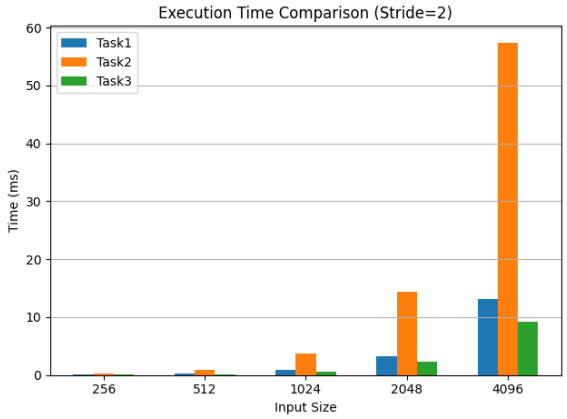

图 9: stride 为 2 下各任务不同输入大小的对比结果（柱状图）

图 10: stride 为 2 下各任务不同输入大小的对比结果（折线图）

- 综合来看，**cuDNN**（Task3）不仅在小、中、大规模输入下都保持领先，而且其增长曲线最为平缓，表明 cuDNN 针对不同问题规模有良好的算法适配与调度策略，是实际部署的优选方案。

5 总结

本次实验我在三种不同思路下实现并对比了二维卷积的性能：

- **滑窗直卷积（Task1）**：最直观但访存冗余最高的实现，随着输入边长翻倍，耗时近似翻四倍；stride 升高（1→3）能显著减少约 50%–80% 的计算量。
- **im2col+GEMM（Task2）**：将卷积转换为矩阵乘法后，可借助共享内存和寄存器优化提升中等规模性能，但需要额外构造巨型临时矩阵，导致对大尺寸输入和步幅变化不敏感，整体耗时常高于 Task1。
- **cuDNN 库（Task3）**：凭借多种专用算法和自动调优，在所有输入规模与步幅组合中均保持最优表现，平均较 Task1/Task2 分别提速约 20%–50%。

此外，我还对 im2col+GEMM 中的 GEMM 核函数分别测试了三种访存模式（basic、shared、reged）、不同行/列/块划分及多种线程块尺寸：

- 共享内存优化（shared）+ Block=16 在所有输入规模下均取得 3×–8× 加速，是最稳健的通用方案；
- 寄存器优化（reged）在部分配置下略优于 shared，但受限于寄存器资源；
- 行划分（row）模式并行粒度过粗，性能最差；列划分（col）和块划分（block）模式并行度高且收敛一致，推荐 block 划分；

- 线程块尺寸 16×16 能在充分利用共享内存和并行度之间取得最佳平衡。

总体来看，cuDNN（Task3）凭借多种高效卷积内核和自动算法调优，在所有输入规模和步幅组合下均取得最佳性能；直接滑窗卷积（Task1）次之，适合中小规模场景且实现简单；而 im2col+GEMM（Task2）尽管对步幅变化不敏感，但在大尺寸输入下因临时矩阵爆炸而性能急剧下降，不建议用于超大规模卷积任务。