中山大學
SUN YAT-SEN UNIVERSITY

并行程序设计与算法实验

Lab9-CUDA **矩阵转置**

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 5 月 21 日

1 实验目的

- 熟悉 CUDA 线程层次结构 (grid、block、thread) 和观察 warp 调度行为。
- 掌握 CUDA 内存优化技术 (共享内存、合并访问)。
- 理解线程块配置对性能的影响。

2 实验内容

2.1 CUDA 并行输出

1. 创建 n 个线程块，每个线程块的维度为 $m \times k$。
2. 每个线程均输出线程块编号、二维块内线程编号。例如：
   - “Hello World from Thread (1, 2) in Block 10!”
   - 主线程输出 “Hello World from the host!”。
   - 在 main 函数结束前，调用 `cudaDeviceSynchronize()`。
3. 完成上述内容，观察输出，并回答线程输出顺序是否有规律。

2.1.1 代码分析

为了实现 CUDA 并行输出，我们需要先创建 n 个线程块，每个线程块的维度为 $m \times k$。`dim3 blocks(n)` 将网格中块的数量设置为 n，而 `dim3 threads(m, k)` 定义了每个块内线程的二维尺寸为 $m \times k$。通过 `<<<blocks, threads>>>` 语法将 `hello_world` kernel 推送到 GPU 上并行执行，每个线程都会按照刚才定义的索引打印消息。

```
    dim3 blocks(n);
    dim3 threads(m, k);
    hello_world<<<blocks, threads>>>();
    printf("Hello World from the host!\n");
```

为了实现并行输出，我们需要完成 CUDA 核心代码 `kernel` 函数。在 `kernel` 定义中，`__global__` 关键字说明 `hello_world` 会被 GPU 上的线程并行调用。函数内部通过内建变量 `blockIdx.x`、`threadIdx.x`、`threadIdx.y` 分别获取当前线程所在块的索引和块内二维线程索引，再利用 `printf` 将三者打印出来。

```
__global__ void hello_world() {
    int blockId = blockIdx.x; // 块索引
    int threadId_x = threadIdx.x; // 线程索引
```

```
    int threadId_y = threadIdx.y;
    printf("Hello World from Thread (%d, %d) in Block %d!\n",
           threadId_x, threadId_y, blockId);
}
```

在 kernel 启动后，cudaDeviceSynchronize() 会阻塞主机线程，直到 GPU 上所有任务完成，确保后续的错误检查能够检测到 kernel 执行中的异常。调用 cudaGetLastError() 并检查返回值，若发生错误则打印详细的错误描述并退出。最后，cudaDeviceReset() 用于清理 CUDA 运行时分配的资源，确保程序结束时 GPU 环境被安全重置。

```
    // 等待 GPU 完成
    cudaDeviceSynchronize();
    // 检查错误
    cudaError_t err = cudaGetLastError();
    if (err != cudaSuccess) {
        printf("CUDA error: %s\n", cudaGetErrorString(err));
        return 1;
    }
    // 释放 GPU 资源
    cudaDeviceReset();
```

2.2 使用 CUDA 实现矩阵转置及优化

1. 使用 CUDA 完成并行矩阵转置。
2. 随机生成 $N \times N$ 的矩阵 A。
3. 对其进行转置得到 A^T。
4. 分析不同线程块大小、矩阵规模、访存方式 (全局内存访问，共享内存访问)、任务/数据划分和映射方式，对程序性能的影响。
5. 实现并对比以下两种矩阵转置方法：
   - 仅使用全局内存的 CUDA 矩阵转置。
   - 使用共享内存的 CUDA 矩阵转置，并考虑优化存储体冲突。

2.2.1 全局内存版本代码分析

为了实现全局内存的 CUDA 函数，首先通过计算 bx = blockIdx.x * blockDim.x 和 by = blockIdx.y * blockDim.y 得到当前线程块在整矩阵中的子块起始坐标，然后利用线程索引 tx = threadIdx.x 和 ty = threadIdx.y 确定该线程负责处理的元素在

子块中的偏移位置。这样，每个线程读入位置为 `in[(by+ty)*n + (bx+tx)]` 并将其写回到转置后的位置 `out[(bx+tx)*n + (by+ty)]`，实现了行列下标的互换。

在读全局内存时，线程块内所有线程对于固定的行坐标 `y = by+ty` 会根据 `tx` 按照连续地址依次访问元素，因而能充分利用全局内存的合并读特性，获得较高的带宽。但在写全局内存阶段，写入坐标中的行由 `bx+tx` 决定，当 `tx` 发生变化时，不同线程写入的列位置是连续的，这导致写操作无法按行主序合并，因而写带宽会受到严重制约。

```
__global__ void transpose(float* out, float* in, int n) {
    int bx = blockIdx.x * blockDim.x;
    int by = blockIdx.y * blockDim.y;
    int tx = threadIdx.x;
    int ty = threadIdx.y;
    out[(bx + tx) * n + by + ty] = in[(by + ty) * n + bx + tx];
}
```

由于该实现对每个元素都直接进行一次全局读和一次全局写，且写操作不共线，随着矩阵规模增大，写带宽瓶颈会显著限制整体性能。

2.2.2 共享内存版本代码分析

共享内存版本通过引入 `__shared__` 关键字在每个线程块内分配了一块 $B \times B$ 的共享内存区域，用于暂存从全局内存中读取的数据。首先，线程根据 `blockIdx` 和 `threadIdx` 计算出其在全局矩阵中的位置 `(x, y)`，并将 `in[y*n + x]` 的值读入共享内存中的 `smem[ty * BDIM + tx]`，保证线程以连续方式访问全局内存，从而实现全局内存读取的 coalescing。在所有线程读入数据后，使用 `__syncthreads()` 实现线程同步，确保共享内存写入已完成。

随后，线程将共享内存中转置后的位置 `smem[tx * BDIM + ty]` 写入全局内存对应的转置位置 `out[y2 * n + x2]`。由于此时访问的是共享内存，其访问速度远快于全局内存，可以显著降低读写延迟。同时，在写回全局内存阶段，各线程再次沿行主序访问不同位置，使得写操作同样具备较好的内存合并特性，相比原始的全局版本性能有显著提升。

```
__global__ void transpose_shared(float* out, float* in, int n){
    __shared__ float smem[BDIM*BDIM];
    int B = blockDim.x;                 // 实际块大小
    int bx = blockIdx.x * B;
    int by = blockIdx.y * B;
    int tx = threadIdx.x, ty = threadIdx.y;
    int x = bx + tx, y = by + ty;
    if (x < n && y < n) {
```

```
        smem[ty*BDIM + tx] = in[y*n + x];
    }
    __syncthreads();
    // 转置写回
    int x2 = by + tx, y2 = bx + ty;
    if (x2 < n && y2 < n) {
        out[y2*n + x2] = smem[tx*BDIM + ty];
    }
}
```

然而，由于共享内存是由多个 memory bank 组成的，当多个线程访问处于同一个 bank 的地址时会发生 bank conflict，导致访问串行化，从而降低共享内存性能。

2.2.3 共享内存（优化）版本代码分析

优化版本通过在共享内存二维数组的第二维上多申请一列，即将共享内存声明为 `smem[BDIM*(BDIM+1)]`，从而显式打破 bank conflict 的访问模式。由于 CUDA 的共享内存以列为 bank 分布，当所有线程读取 `ty*pitch + tx` 或写入 `tx*pitch + ty` 时，增加的一列使得每行起始地址不再落入相同 bank，保证线程间对共享内存的访问分布在不同 bank，从而实现真正的并行访问。

```
__global__ void transpose_shared_opt(float* out, float* in, int n){
    // 在第二维多分配一列，消除 bank conflict
    __shared__ float smem[BDIM*(BDIM+1)];
    int B = blockDim.x;
    int bx = blockIdx.x * B;
    int by = blockIdx.y * B;
    int tx = threadIdx.x, ty = threadIdx.y;
    int pitch = BDIM + 1;  // 真正的行距
    int x = bx + tx, y = by + ty;
    if (x < n && y < n) {
        smem[ty*pitch + tx] = in[y*n + x];
    }
    __syncthreads();
    // 转置写回
    int x2 = by + tx, y2 = bx + ty;
    if (x2 < n && y2 < n) {
        out[y2*n + x2] = smem[tx*pitch + ty];
    }
}
```

2.2.4 时间记录代码分析

在这里为了能够更好地记录 cuda 编程所用的时间，我调用了 cuda 专门的统计时间的函数库，代码如下：

```
    cudaEvent_t start, stop;
    cudaEventCreate(&start);
    cudaEventCreate(&stop);
    cudaEventRecord(start, 0);
    transpose_shared_opt<<<grid, block>>>(d_B, d_A, N);
    cudaEventRecord(stop, 0);
    cudaEventSynchronize(stop);
    float ms;
    cudaEventElapsedTime(&ms, start, stop);
```

3 实验结果与分析

3.1 CUDA Hello World 并行输出

3.1.1 实验现象

描述实验观察到的现象，例如线程输出的顺序等。可以粘贴部分关键的运行截图或输出文本。

```
=== CUDA Hello World ===
Hello World from the host!
Hello World from Thread (0, 0) in Block 6!
Hello World from Thread (1, 0) in Block 6!
Hello World from Thread (0, 1) in Block 6!
Hello World from Thread (1, 1) in Block 6!
Hello World from Thread (0, 2) in Block 6!
Hello World from Thread (1, 2) in Block 6!
Hello World from Thread (0, 3) in Block 6!
Hello World from Thread (1, 3) in Block 6!
Hello World from Thread (0, 0) in Block 7!
Hello World from Thread (1, 0) in Block 7!
Hello World from Thread (0, 1) in Block 7!
Hello World from Thread (1, 1) in Block 7!
Hello World from Thread (0, 2) in Block 7!
Hello World from Thread (1, 2) in Block 7!
Hello World from Thread (0, 3) in Block 7!
Hello World from Thread (1, 3) in Block 7!
Hello World from Thread (0, 0) in Block 5!
Hello World from Thread (1, 0) in Block 5!
```

图 1: 线程输出情况 1

```
Hello World from Thread (0, 0) in Block 9!
Hello World from Thread (1, 0) in Block 9!
Hello World from Thread (0, 1) in Block 9!
Hello World from Thread (1, 1) in Block 9!
Hello World from Thread (0, 2) in Block 9!
Hello World from Thread (1, 2) in Block 9!
Hello World from Thread (0, 3) in Block 9!
Hello World from Thread (1, 3) in Block 9!
Hello World from Thread (0, 0) in Block 0!
Hello World from Thread (1, 0) in Block 0!
Hello World from Thread (0, 1) in Block 0!
Hello World from Thread (1, 1) in Block 0!
Hello World from Thread (0, 2) in Block 0!
Hello World from Thread (1, 2) in Block 0!
Hello World from Thread (0, 3) in Block 0!
Hello World from Thread (1, 3) in Block 0!
Hello World from Thread (0, 0) in Block 3!
Hello World from Thread (1, 0) in Block 3!
Hello World from Thread (0, 1) in Block 3!
Hello World from Thread (1, 1) in Block 3!
```

图 2: 线程输出情况 2

- **主机打印在前，设备输出在后**
  - Kernel 调用之后，主机端立即执行 `printf("Hello World from the host!\n")`，而 CUDA Kernel 为异步启动，直到 `cudaDeviceSynchronize()` 前，主机不会等待设备执行完成。因此，主机输出总是率先显示，随后才是设备端线程的输出。
- **Block 之间的执行顺序无全局规律**
  - 从输出中观察到 Block 的编号顺序如：$6 \to 7 \to 5 \to 8 \to 1 \to 2 \to 4 \to 9 \to 0 \to 3$，表明 Block 被调度至不同的 SM 并行执行，各 SM 的输出依照自身执行完成和缓冲区刷新时序交错输出，无法预测全局 Block 顺序。
- **同一 Block 内线程输出几乎有序**
  - 以 Block 6 为例，其线程输出顺序为：

    ```
    Thread (0, 0), (1, 0), (0, 1), (1, 1), (0, 2), (1, 2), (0, 3), (1, 3)
    ```

  - 该顺序体现了线程在两维 `threadIdx` 中，按 `ty`（行）优先，`tx`（列）次序排列。CUDA 执行 `printf` 时，依据 Warp 内线程编号（`threadIdx.y * blockDim.x + threadIdx.x`）线性展开调度，因此 Block 内的输出具有一定顺序性。
- **Warp 级并行与输出缓冲交错**
  - 同一 Block 内多个 Warp 并行执行，交替触发 `printf`，但由于线程编号集中（尤其 Block 较小时，一个 Block 可能只占用一个 Warp），所以大多数 Block 内输出呈现“行优先”顺序。
  - 若 Block 中线程数超出一个 Warp（如 $16 \times 16 = 256$ 线程），则同一 Block 中的多个小批次（Warp）分批次输出，每个批次内部依旧保持“行优先”顺序，但 Warp 之间输出顺序不固定。

3.1.2 结果分析

线程输出顺序是否有规律？为什么？结合 CUDA 线程调度机制进行解释。

- 回答：整体来看，设备端线程的 `printf` 输出顺序是非确定性的。不同 Block 的输出顺序没有固定模式，Block ID 的打印顺序在多次运行中都会变化。同一个 Block 内，线程在小规模 Block（一个或少数 Warp）中常显现为按行主序（`ty` 再 `tx`）的输出，但在 Block 较大、跨 Warp 时会出现分段交错。

- 原因：CUDA Kernel 调用是异步的，主机端的 `printf("Hello World from the host!")` 会在设备端输出之前完成。设备上所有线程的输出先被缓存在 GPU 内部，直到调用 `cudaDeviceSynchronize()` 后才刷入主机标准输出，因而主机打印总在前。

- 不同 Block 被分派到多个 SM 并行调度，SM 处理 Block 的顺序取决于硬件空闲情况和调度策略，无法按 Block ID 增序或减序执行，因此全局输出顺序不可预测。

- 每个 Block 内的线程按线性编号 $laneId = ty \times \texttt{blockDim.x} + tx$ 分配到若干个 Warp（每 Warp 32 线程）。同一 Warp 内线程通常会以连续的编号顺序并行执行 `printf`，因而在小 Block（线程数 $\leq$32）或同 Warp 范围内会看到“行主序”输出。但多个 Warp 并行时，其输出被交替缓冲，导致局部有序、跨 Warp 则交错。

3.2 CUDA 矩阵转置及优化

3.2.1 不同实现方法的性能对比

1. 展示不同矩阵转置实现（仅全局内存、使用共享内存、优化共享内存访问）在不同矩阵规模 (N) 和不同线程块大小下的运行时间。可以根据你的实验设置更改表格的矩阵规模、线程块大小。

表 1: 矩阵转置性能对比 (时间单位: ms)

矩阵规模 (N)	线程块大小	全局内存版本	共享内存版本	优化共享内存版本
512	8×8	0.057	0.055	0.054
	16×16	0.054	0.056	0.054
	32×32	0.063	0.057	0.059
1024	8×8	0.044	0.046	0.044
	16×16	0.051	0.046	0.046
	32×32	0.061	0.051	0.036
2048	8×8	0.101	0.085	0.072
	16×16	0.090	0.061	0.046
	32×32	0.165	0.067	0.055
8192	8×8	0.720	0.771	0.728
	16×16	0.975	0.710	0.709
	32×32	2.041	0.877	0.791
16384	8×8	2.746	2.730	2.758
	16×16	3.492	2.732	2.713
	32×32	7.510	3.385	2.971

3.2.2 结果分析

1. 根据实验结果，总结线程块大小、矩阵规模对程序性能的影响。哪种配置下性能最优？为什么？

回答:从表中数据可以看出,共享内存版本(Shared)和优化共享内存版本(Shared_opt)均优于全局内存版本（Global），尤其是在较大规模矩阵上差距更明显。进一步分析:

- 对于中小规模矩阵（$N = 512$、1024），当线程块大小取 32×32 时，Shared_opt 版本分别取得了 0.059 ms 和 0.036 ms 的最快成绩。这是因为较大的线程块能够在一次 kernel 调用中处理更多数据，减少 kernel 启动及同步开销；同时共享内存无 bank 冲突，访存带宽利用率最高。
- 对于大规模矩阵（$N = 2048$、8192、16384），Global 版本的时间随矩阵规模急剧上升（如 $N = 16384$ 时达到 7.510 ms），而 Shared_opt 仅为 2.971 ms，优势更加突出。说明共享内存优化对海量数据处理的稳定加速效果。
- 较小线程块（8×8、16×16）在大规模矩阵上性能回落，原因在于更多块的启动和边界检查带来额外开销；而 32×32 的块尺寸能够最大化利用每个 SM 的并行度并摊薄固定开销。

因此，**在所有测试场景下，将线程块配置为 32×32 并使用优化后的共享内存版本可获得最优性能**，主要原因包括:

- 较大线程块摊薄了 kernel 启动和同步的固定开销;
- 共享内存打散（padding）后无 bank 冲突，确保读写都为共线访问;
- 数据在共享内存中复用率高，极大降低了全局内存带宽压力。

2. 讨论任务/数据划分和映射方式对性能的影响。

任务/数据划分对性能的影响

回答: 本实验中，任务与数据的划分方式主要体现在线程块大小 `B` 的选取上，即将整个 $N \times N$ 矩阵划分为若干个 $B \times B$ 的子块，交由 GPU 上不同线程块（block）并行处理。代码中通过 `dim3 block(B, B)` 和 `dim3 grid((N+B-1)/B, (N+B-1)/B)` 实现。

实验表明，块大小直接影响 GPU SM 的资源利用率和访存效率。较小的 `B` 值（8、16）导致线程块数量增加，kernel 启动开销增大，且每个线程块内部的访存操作分散，访存合并效果较差。而较大的 `B` 值（32）则可以有效减少线程块数量，提高单个线程块内的访存连续性，减轻启动开销，并让更多访存操作合并执行。根据实验结果，$B = 32$ 时在 $N = 2048$ 矩阵下取得了最佳性能，此时线程数量充足、访存合并良好、并行度与硬件资源利用达到平衡。

映射方式对性能的影响

回答：映射方式主要指线程如何访问全局内存或共享内存，并决定访存性能。本实验共实现了三种映射方式：全局内存直接映射、共享内存映射、以及消除 bank conflict 的共享内存优化映射。

首先，全局内存直接映射版本中，每个线程直接从 `in[y*n + x]` 读取一个元素，再写入 `out[(x)*n + y]`。由于 GPU 全局内存带宽大、延迟高，且写操作非连续（转置导致行主序数据在内存中变成列访问），访存合并效果极差，导致大量 global memory transaction，实验中性能最低。

其次，普通共享内存映射版本中，每个线程先将自己负责的 `in[y*n + x]` 元素读入 `__shared__ float smem[BDIM*BDIM]` 中，然后同步所有线程，再完成转置写回。这种方式大幅减少了全局内存访存次数（每线程块只需两次 global 访存），共享内存访问延迟极低，性能显著提升。但由于共享内存按 bank 分布，转置写回阶段 `smem[tx*BDIM + ty]` 容易出现 bank conflict，限制了带宽利用率。

最后，优化共享内存映射将共享内存数组第二维多加一列`smem[BDIM*(BDIM+1)]`改变物理排布，避免同一 warp 内多线程访问同一 bank，消除了 bank conflict。实验数据显示，该版本在 $B = 32$ 下较普通共享内存版本进一步提速，访存冲突完全消除，带宽利用率最优。