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 李源卿

学 号 22336128

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。

2.2 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。

```
Hello World from thread (14, 7) in Block (14)
Hello World from thread (15, 7) in Block (14)
Hello World from thread (0, 10) in Block (1)
Hello World from thread (1, 10) in Block (1)
Hello World from thread (2, 10) in Block (1)
Hello World from thread (3, 10) in Block (1)
Hello World from thread (4, 10) in Block (1)
Hello World from thread (5, 10) in Block (1)
Hello World from thread (6, 10) in Block (1)
Hello World from thread (7, 10) in Block (1)
Hello World from thread (8, 10) in Block (1)
Hello World from thread (9, 10) in Block (1)
Hello World from thread (10, 10) in Block (1)
Hello World from thread (11, 10) in Block (1)
Hello World from thread (12, 10) in Block (1)
Hello World from thread (13, 10) in Block (1)
Hello World from thread (14, 10) in Block (1)
Hello World from thread (15, 10) in Block (1)
Hello World from thread (0, 11) in Block (1)
Hello World from thread (1, 11) in Block (1)
Hello World from thread (2, 11) in Block (1)
Hello World from thread (3, 11) in Block (1)
Hello World from thread (4, 11) in Block (1)
Hello World from thread (5, 11) in Block (1)
Hello World from thread (6, 11) in Block (1)
Hello World from thread (7, 11) in Block (1)
Hello World from thread (8, 11) in Block (1)
Hello World from thread (9, 11) in Block (1)
Hello World from thread (10, 11) in Block (1)
Hello World from thread (11, 11) in Block (1)
Hello World from thread (12, 11) in Block (1)
Hello World from thread (13, 11) in Block (1)
Hello World from thread (14, 11) in Block (1)
Hello World from thread (15, 11) in Block (1)
Hello World from thread (0, 8) in Block (14)
Hello World from thread (1, 8) in Block (14)
```

图 1: HelloWorld 程序输出

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。

关键代码

```
__global__ void hello_world_kernel() {
    int x = threadIdx.x + blockIdx.x * blockDim.x;
    int y = threadIdx.y + blockIdx.y * blockDim.y;
    printf("Hello World from thread (%d, %d) in Block (%d)\n",
        threadIdx.x, threadIdx.y, blockIdx.x);
}
    ...
```

```
dim3 threadsPerBlock(16, 16);
dim3 blocksInGrid(16);
```

我们可以注重分析 block 编号为 1 的线程块：

- 该 block 中线程输出按 y 坐标分组出现：先是 y=10 的所有 x(0-15)，然后是 y=11 的所有 x(0-15)
- 每个 y 组内线程按 x 坐标顺序输出（从 0 到 15）
- 与 block 14 的线程输出交错出现

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。

- **Warp 组织线程**：我们可以看到程序调用了 y 坐标为 10 和 11 的所有线程，总共 32 个，符合 Warp 的组织线程的数量。
  同时，我们发现程序调用了 y 坐标为 10 和 11 的所有线程（x 编号为 0-15），说明 warp 组织线程时是从 x 的方向将线程展平的。
- **非严格顺序调度**：虽然 y=10 和 y=11 的线程按顺序输出，但整体上 block 1 和 block 14 的线程输出交错出现，说明：
  - GPU 以 warp 为单位进行调度
  - 执行顺序受硬件调度器控制，不保证严格的线性顺序
- **SIMT 执行特性**：同一 warp 内的线程执行相同指令（printf），但输出顺序可能受线程索引影响，体现了单指令多线程的执行特点。

3.2 CUDA 矩阵转置及优化

3.2.1 正确性分析

我选择输出前 5 行 5 列来验证正确性。

```
    ...

printMatrix(h_in, 5, n);

printMatrix(h_out, 5, n);//param: 输出的矩阵、输出前几行几列、矩阵大小
    ...
```

结果如下：

```
Original matrix (top-left 5x5):
  0   1   2   3   4
16384 16385 16386 16387 16388
32768 32769 32770 32771 32772
49152 49153 49154 49155 49156
65536 65537 65538 65539 65540

Transposed matrix (top-left 5x5):
  0 16384 32768 49152 65536
  1 16385 32769 49153 65537
  2 16386 32770 49154 65538
  3 16387 32771 49155 65539
  4 16388 32772 49156 65540
```

图 2: trans 程序输出

3.2.2 不同实现方法的性能对比

优化版本 1 是我自己想的，主要是更改访问顺序来解决存储体冲突。下面是我的思路：

存储体冲突的原因在于，一个存储体能存储 32 个矩阵元素，那么，如果按照图 3 中的箭头去依次分配元素给 thread，就会造成一个 warp 中的 16 个线程访问同一个存储体，而另外 16 个线程访问另外一个存储体，会产生冲突。

我们可以分析一下矩阵元素在存储体中的排布：(0,0) 存储在 bank0...(0,15) 存储在 bank15，(1,0) 存储在 bank16...(1,15) 存储在 bank31...（以此类推）。

刚才的问题在于，(0,0) 和 (2,0) 在同一个存储体，分别被 thread0 和 thread2 同时访问。那么我们不妨改变一下访问的映射关系，把每一行的元素一个个分给 thread 们，也就是让 thread0 访问 (0,0)(bank0)，让 thread1 访问 (1,0)(bank1)..... 我觉得这样就不会有存储体冲突了

所以有下面的代码：

```
int row = by + ty;
int col = bx + tx;
if (row < n && col < n) {
    smem[ty*BDIM + tx] = in[row * n + col];
}
__syncthreads();
if (row < n && col < n) {
    out[col * n + row] = smem[ty*BDIM + tx];
}
```

优化版本 2 就是按照老师讲的思路：分配共享内存的时候多加一列，打乱矩阵元素在存

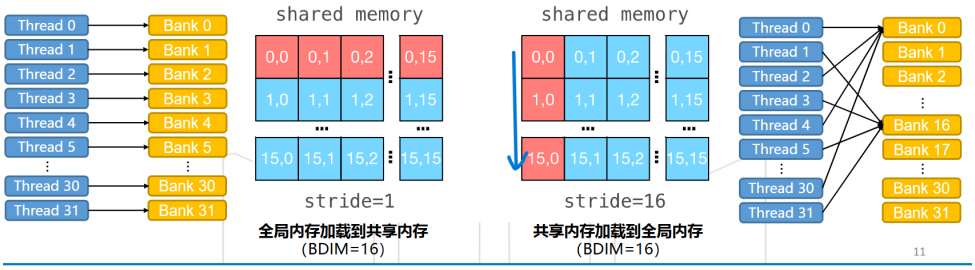

图 3: 冲突原因

储体中的排列。

下面我将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	优化版本 1	优化版本 2
512	8×8	0.009873	0.011587	0.010315	0.010779
	16×16	0.011199	0.010640	0.010724	0.010875
	32×32	0.012310	0.012183	0.012963	0.010320
1024	8×8	0.010590	0.010228	0.010749	0.010412
	16×16	0.010595	0.011088	0.010934	0.010467
	32×32	0.013219	0.011646	0.012723	0.010704
2048	8×8	0.009914	0.010524	0.010520	0.010486
	16×16	0.010818	0.010747	0.010572	0.010557
	32×32	0.012362	0.012070	0.012736	0.010811
16384	8×8	0.013191	0.012427	0.013074	0.012300
	16×16	0.014253	0.011709	0.013826	0.011185
	32×32	0.016478	0.013745	0.017241	0.010062

3.2.3 性能结果分析

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？

- **线程块大小对性能的影响**:
  - 较小线程块（如 8×8）在小规模矩阵（如 $N = 512$ 或 1024）上表现更好。例如，在 $N = 512$ 规模下，全局内存版本时间为 0.009873 ms。这是线程块小，分出来的线程块就多，线程块数目应当大于 SM 数量才能较好发挥 GPU 性能。

  - 随着线程块增大（如 32×32），在小规模矩阵上时间可能增加（如 $N = 512$ 规模下全局内存版本时间达 0.012310 ms），表明块增大会导致块数目减少，有的 SM 甚至可能会空闲。
  - 较大线程块（如 32×32）在更大规模矩阵（如 $N = 16384$）上优势显著，因为能更好地利用 GPU 的 warp 调度和并行资源，减少内核启动开销和块间同步成本。

- **矩阵规模对性能的影响**:
  - 随着矩阵规模增加（从 $N = 512$ 到 16384），运行时间总体呈上升趋势。例如，全局内存版本在 $N = 16384$ 规模下时间高达 0.016478 ms。
  - 这是由于数据量增大导致全局内存访问同步延迟（由于存储体冲突需要同步线程）增加和并行任务复杂度提升。
  - 优化版本（如优化版本 2）能有效缓解此影响，通过共享内存和优化访问模式、解决存储体冲突减轻延迟问题。

- **最优配置**: 矩阵规模 $N = 16384$、线程块大小 32×32、优化版本 2（时间最低，0.010062 ms）。

- **原因**:
  - 较大线程块与大矩阵匹配，线程块内的线程数和线程块的数目处于一个合适的大小，既可以充分被 SM 调用来隐藏延迟，SM 内部的活跃线程数也较多。
  - 优化版本 2 充分利用共享内存、减少 bank 冲突（通过调整数据访问模式确保合并访问），所以性能上较优。

- **一个问题**: 我还没有想明白我的方式为什么没有解决问题，得到的性能和没有解决存储体冲突类似甚至更差。

- **问题解决了**: 我的方式解决了存储体冲突，但是顾此失彼，在写入时不是合并写入，导致更多的时间开销。

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**:
  - 较大块（如 32×32) 能处理更大数据，减少块数量并提高计算密度（如在 $N = 16384$ 规模下，32×32 线程块优化版本 2 性能最优）。
  - 划分应与 GPU 架构特性（如共享内存大小和 warp 调度）精细匹配，才能最大化并行效率和内存带宽。

- **映射方式对性能的影响**:
  - 共享内存版本通常性能优于全局内存版本（如 $N = 16384$ 规模、16×16 块下，共享内存版本时间 0.011709 ms vs. 全局内存版本 0.014253 ms），因为它减少了高延迟全局内存访问。
  - 但不当的映射（如 bank 冲突）会劣化性能（我的优化版本 1 可能是一个不恰当的映射）。
  - 优化版本 2 通过改进数据映射（转置时使用共享内存填充以避免 bank 冲突）实现了较好的性能。