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 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 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。

```
(openr1) (base) root@0a2d09754970:~/nfs/parallel computing/lab9# ./matrix_transpose 2 4 4
=== CUDA 矩阵转置实验 ===

=== 任务1: CUDA Hello World ===
Hello World from the host!
\Hello World from Thread (0, 0) in Block 1!
Hello World from Thread (1, 0) in Block 1!
Hello World from Thread (2, 0) in Block 1!
Hello World from Thread (3, 0) in Block 1!
Hello World from Thread (0, 1) in Block 1!
Hello World from Thread (1, 1) in Block 1!
Hello World from Thread (2, 1) in Block 1!
Hello World from Thread (3, 1) in Block 1!
Hello World from Thread (0, 2) in Block 1!
Hello World from Thread (1, 2) in Block 1!
Hello World from Thread (2, 2) in Block 1!
Hello World from Thread (3, 2) in Block 1!
Hello World from Thread (0, 3) in Block 1!
Hello World from Thread (1, 3) in Block 1!
Hello World from Thread (2, 3) in Block 1!
Hello World from Thread (3, 3) in Block 1!
Hello World from Thread (0, 0) in Block 0!
Hello World from Thread (1, 0) in Block 0!
Hello World from Thread (2, 0) in Block 0!
Hello World from Thread (3, 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 (2, 1) in Block 0!
Hello World from Thread (3, 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 (2, 2) in Block 0!
Hello World from Thread (3, 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 (2, 3) in Block 0!
Hello World from Thread (3, 3) in Block 0!
```

图 1: 任务 1：CUDA Hello World 的输出

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。

- 回答：在我的运行中，参数设置如下，可以参考我的运行截图1
  - 第 1 个参数 (2): n = 线程块数量，创建 2 个线程块
  - 第 2 个参数 (4): m = 每个线程块的 x 维度，4 个线程
  - 第 3 个参数 (4): k = 每个线程块的 y 维度，4 个线程

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：先执行 Block 1 的所有线程，再执行 Block 0 的所有线程。每个块内的线程按 (x,y) 顺序执行：固定 y，x 从 0 递增到 3（如 (0,0) → (3,0)），然后 y 递增，重复 x 的顺序（如 (0,1) → (3,1)）。

  块内顺序是由 CUDA 线程索引的行优先线性化决定的，线程 ID = x + y * blockDim.x。而块间顺序理论上不确定。

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	0.006	0.005	0.005
	16×16	0.005	0.005	0.005
	32×32	0.010	0.010	0.009
1024	8×8	0.011	0.010	0.010
	16×16	0.013	0.008	0.007
	32×32	0.032	0.030	0.030
2048	8×8	0.034	0.034	0.034
	16×16	0.049	0.025	0.018
	32×32	0.117	0.113	0.111

3.2.2 不同实现方法的核心代码

1. 全局内存版本

```
__global__ void transposeGlobal(float* input, float* output, int n) {
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    int row = blockIdx.y * blockDim.y + threadIdx.y;

    if (row < n && col < n) {
        output[col * n + row] = input[row * n + col];
```

```
    }
}
```

直接从全局内存读取并写入，存在大量不合并的内存访问。

2. 共享内存版本

```
__global__ void transposeShared(float* input, float* output, int n) {
    __shared__ float tile[32][32];

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

    // 读取到共享内存
    if (row < n && col < n) {
        tile[threadIdx.y][threadIdx.x] = input[row * n + col];
    }

    __syncthreads();

    // 转置后的坐标
    int new_col = blockIdx.y * blockDim.y + threadIdx.x;
    int new_row = blockIdx.x * blockDim.x + threadIdx.y;

    if (new_row < n && new_col < n) {
        output[new_col * n + new_row] = tile[threadIdx.x][threadIdx.y];
    }
}
```

使用 32×32 共享内存块缓存数据，减少全局内存访问，但存在 bank conflicts。

3. 优化共享内存版本

```
__global__ void transposeSharedOptimized(float* input, float* output, int
    __shared__ float tile[32][33]; // 增加一列避免bank conflicts

    // ... 其余逻辑相同
}
```

通过将共享内存大小从 32×32 改为 32×33，避免 bank conflicts，提升访问效率。

3.2.3 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:

- **线程块大小影响**：16×16 的线程块配置在大多数情况下性能最优，特别是在中大规模矩阵（1024 和 2048）中表现突出。32×32 配置在大矩阵时性能下降明显，8×8 配置在小矩阵时表现良好但扩展性不佳。
- **矩阵规模影响**:随着矩阵规模增加,不同实现方法的性能差异逐渐显现。在 2048×2048 矩阵中，优化共享内存版本比全局内存版本快约 3 倍（16×16 配置下）。
- **最优配置**：2048 矩阵 +16×16 线程块 + 优化共享内存版本，执行时间仅 0.018ms。这是因为 16×16 配置在保证充分并行度的同时避免了资源竞争，且优化版本消除了 bank conflicts。

2. 讨论任务/数据划分和映射方式对性能的影响。

回答:

- **数据局部性**：共享内存版本通过数据划分，将 32×32 数据块载入共享内存，显著改善了数据局部性，减少了全局内存访问延迟。
- **内存访问模式**：全局内存版本的内存合并效率低下，而共享内存版本通过重新组织内存访问模式，实现了更好的内存带宽利用。
- **Bank conflicts 消除**：优化版本通过增加 padding（32×33 vs 32×32），避免了共享内存 bank conflicts，进一步提升了内存访问效率，特别在大规模矩阵中效果显著。

注：实验报告格式参考本模板，可在此基础上进行修改；实验代码以 zip 格式另提交；最终提交内容包括实验报告 (pdf 格式) 和实验代码 (zip 压缩包格式)