中山大學
SUN YAT-SEN UNIVERSITY

并行程序设计与算法实验

Lab10-CUDA **并行矩阵乘法**

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 28 日

1 实验目的

- 理解 CUDA 编程模型 (Grid、Block、Thread) 及其在矩阵乘法中的应用。
- 学习 GPU 内存优化技术。

2 实验内容

- 实现基础矩阵乘法
- 优化矩阵乘法：共享内存，分块技术
- 测量不同实现的运行时间

3 程序简要分析

本程序实现了三种不同优化级别的 CUDA 矩阵乘法核函数：

- **朴素实现**（`matMulNaive`）：每个线程计算 C 中一个元素，直接遍历整个内积，内存访问无优化。
- **共享内存优化**（`matMulShared<TILE>`）：先将 A、B 子矩阵块加载到 `__shared__` 缓冲区，再在片内线程协同完成积累，减少全局内存访存次数。
- **寄存器分块优化**（`matMulReg`）：将内积展开（手动 4 元展开）以提高寄存器复用，减轻内存带宽压力。

核心调度逻辑：根据输入参数 `N`、`TILE`、`partition`、`mode` 构造合适的 `grid` 与 `block`，并在运行时选择对应的核函数执行完毕后使用 CUDA 事件记录耗时。

3.1 关键代码片段

Listing 1: 核函数调用与调度逻辑

```
// 设置线程块与网格
dim3 block(TILE, TILE);
dim3 grid;
switch(part){
  case ROW:    grid = dim3((N+TILE-1)/TILE, 1);          break;
  case COL:    grid = dim3(1, (N+TILE-1)/TILE);          break;
  default:     grid = dim3((N+TILE-1)/TILE,(N+TILE-1)/TILE);
}
```

```
// 计时开始
cudaEventRecord(start);
// 根据 mode 选择核函数
switch(mode){
  case 0:  matMulNaive<<<grid,block>>>(dA,dB,dC,N); break;
  case 1:  // 共享内存优化
    if (TILE==16) matMulShared<16><<<grid,block>>>(dA,dB,dC,N);
    break;
  case 2:  matMulReg<<<grid,block>>>(dA,dB,dC,N);    break;
}
cudaDeviceSynchronize();
// 计时结束
cudaEventRecord(stop);
```

3.2 运行方式

在 Linux 终端中执行以下步骤:

1. 编译:

   ```
   nvcc matrix_mul.cu -o matrix_mul -O3
   ```

2. 运行举例:

   ```
   ./matrix_mul 1024 16 2 1
   ```

   其中参数含义:

   - `1024`: 矩阵规模 N;
   - `16`: 线程块大小 $TILE$;
   - `2`: 按 “数据块” 方式划分 (0= 行、1= 列、2= 块);
   - `1`: 选择 “共享内存优化” 核函数 (0= 朴素、1= 共享、2= 寄存器)。

4 实验结果与分析

4.1 不同实现方法的性能对比

表 1: 按行划分：性能对比 (时间单位: ms)

矩阵规模 (N)	线程块大小	朴素实现	基于共享内存优化	基于寄存器分块优化
512	8×8	78.185	78.648	77.124
	16×16	78.195	106.739	77.588
	32×32	65.283	65.603	62.858
1024	8×8	63.274	75.646	76.905
	16×16	63.689	64.610	59.287
	32×32	66.757	70.463	78.178
2048	8×8	76.734	67.677	75.951
	16×16	76.778	66.178	68.491
	32×32	66.808	68.425	77.531

表 2: 按列划分：性能对比 (时间单位: ms)

矩阵规模 (N)	线程块大小	朴素实现	基于共享内存优化	基于寄存器分块优化
512	8×8	77.570	77.845	96.143
	16×16	65.890	65.811	77.669
	32×32	69.312	65.621	67.372
1024	8×8	61.407	59.453	60.680
	16×16	63.065	60.577	63.023
	32×32	99.228	64.170	76.690
2048	8×8	77.401	76.710	63.978
	16×16	65.271	72.518	64.927
	32×32	77.117	65.002	66.377

表 3: 按数据块划分：性能对比 (时间单位: ms)

矩阵规模 (N)	线程块大小	朴素实现	基于共享内存优化	基于寄存器分块优化
512	8×8	76.940	75.823	78.356
	16×16	77.526	77.732	60.689
	32×32	77.334	77.597	62.067
1024	8×8	64.874	65.073	69.722
	16×16	62.788	61.298	65.507
	32×32	77.332	77.760	75.531
2048	8×8	82.592	68.092	80.993
	16×16	79.201	92.273	81.420
	32×32	81.747	68.142	68.626

分析性能差异的原因：

4.2 性能差异原因分析

- 结合 CUDA 内存模型和矩阵乘法原理，分析造成观察到的性能差异的可能原因。
  回答： CUDA 全局内存带宽相对寄存器和共享内存延迟较高。
  - **朴素实现**：每次访问 A、B 元素都从全局内存读取，缺乏数据重用，导致带宽瓶颈，随着 N 增大性能下降缓慢。
  - **共享内存优化**：将子块加载到每个块的共享内存中，块内所有线程可以重用同一数据，显著减少全局访存次数。但如果 tile 大小与块大小不匹配或共享内存使用超限，会导致分配不足或降低并发块数。
  - **寄存器分块优化**：利用寄存器展开手动积累，寄存器访问延迟最低，适合计算密集型场景。但过度展开可能导致寄存器压力过大，降低可并发的线程数，影响吞吐量。
- 如何选择合适的线程块大小以提高占用率?
  回答：
  - 线程块大小应是 32 或其整数倍，以避免浪费线程。
  - 根据 GPU 架构查询最多可驻留的线程数（如 2048 或 1536），选择块大小使得多个块可同时驻留，例如 $256 = 8 \times 32$ 或 $512 = 16 \times 32$。
  - 较大 tile 增加共享内存和寄存器需求，可能降低并发块数；较小 tile 则全局访存增多。

- 思考如果按不同的方式划分（例如，按行、列、数据块划分），可能会对性能和实现复杂度带来什么影响?
  回答：
  - **按行划分**：每个 block 负责一定数量的行，grid.y=1，grid.x 足够大。优点是简单；缺点是当 N 很大时，单维度网格可能超过最大 grid.x 限制，且每个线程块内维度不平衡，load imbalance。
  - **按列划分**：类似按行，grid.x=1，grid.y 多。同样可能出现维度越界和负载不均。
  - **按数据块（二维划分）**：grid.x 和 grid.y 都分片，最常用。均衡分配工作，每块大小可控，并便于在块内实现共享内存优化。但实现需要计算二维索引，稍微复杂一些。
- 何时应该优先考虑使用哪种存储?
  回答：
  - **寄存器**：私有于线程，访问最快。
  - **共享内存**：块内线程可共享，延迟低于全局内存。
  - **全局内存**：容量大但延迟高，适合一次性读取或输出结果后不再重用的数据。
  - **常量 / 只读缓存**：适用于不变且跨线程共享的小数据（例如卷积核系数），可提供比全局内存更低延迟的只读缓存访问。