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 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 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

2.1 朴素实现

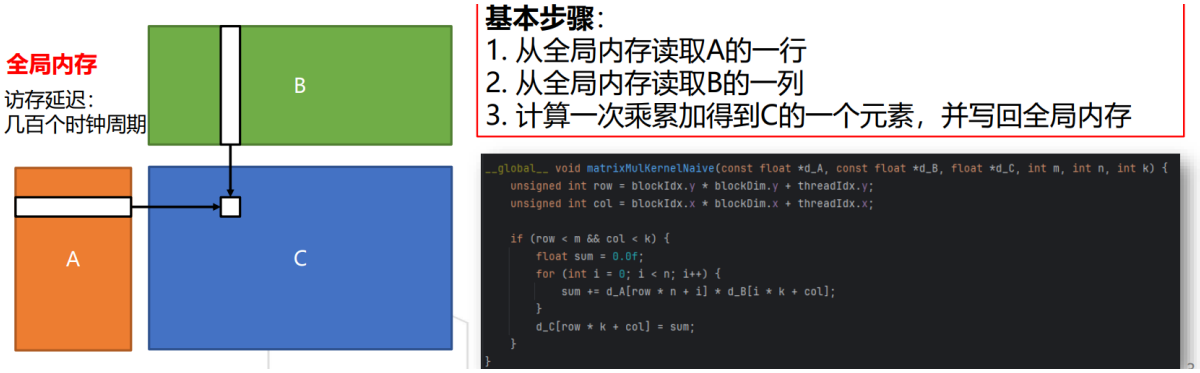

图 1: 朴素实现

朴素实现方式每个 GPU 线程计算输出矩阵的一个元素，通过遍历输入矩阵的行和列进行乘加运算，最终将结果写入全局内存。在实验中，我具体分为了行划分、列划分和快划分来实现。首先我们来介绍行划分。

2.1.1 行划分

具体而言，每个线程负责计算结果矩阵 C 中的一个元素 $C[row][col]$，其中 row 表示行号，col 表示列号。该核函数首先根据线程块编号（`blockIdx`）和线程编号（`threadIdx`），结合线程块尺寸（`blockDim`），计算出当前线程对应的位置索引。然后使用局部变量 `sum` 对当前线程负责计算的矩阵元素值进行累加。依次将矩阵 A 的第 row 行与矩阵 B 的第 col 列对应元素相乘，并将乘积累加，完成对应矩阵元素的点积操作。由于 CUDA 中矩阵以一维数组形式存储，且采用行优先（Row-major）顺序，矩阵 A 和 B 中元素的访问地址通过偏移量计算获得，分别为 $A[row \times n + j]$ 和 $B[j \times k + col]$。

```
__global__ void matMulRow(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 j = 0; j < n; ++j) {
            sum += A[row * n + j] * B[j * k + col];
        }
        C[row * k + col] = sum;
    }
}
```

2.1.2 列划分

列划分的实现方式与行划分类似，只不过任务划分的维度不同。在该方法中，每个线程同样负责计算结果矩阵 C 中的一个元素 $C[row][col]$，但线程的索引计算方式与行划分略有区别。具体而言，将 `threadIdx.x` 分配用于计算行号，将 `threadIdx.y` 分配用于计算列号，线程块编号的计算方式也相应调整为：行号由 `blockIdx.y * blockDim.x + threadIdx.x` 计算，列号由 `blockIdx.x * blockDim.y + threadIdx.y` 计算。

计算核心部分与行划分方法保持一致，依旧采用朴素矩阵乘法的方式，使用一个局部变量 `sum`，对矩阵 A 的第 row 行与矩阵 B 的第 col 列进行逐元素乘法，再进行累加，最终将累加结果写入矩阵 C 对应位置。由于 CUDA 中矩阵以一维数组形式、行优先顺序存储，矩阵元素的访问方式仍然为 $A[row \times n + j]$ 和 $B[j \times k + col]$。

```
__global__ void matMulCol(const float *A, const float *B, float *C,
                          int m, int n, int k) {
    // 这里我们把 threadIdx.x 用于行，threadIdx.y 用于列
    int row = blockIdx.y * blockDim.x + threadIdx.x;  // note: blockDim.x
    int col = blockIdx.x * blockDim.y + threadIdx.y;  // note: blockDim.y
    if (row < m && col < k) {
        float sum = 0.0f;
        for (int j = 0; j < n; ++j) {
            sum += A[row * n + j] * B[j * k + col];
        }
        C[row * k + col] = sum;
    }
}
```

2.1.3 块划分

块划分方式将输出矩阵 C 划分为若干个子矩阵块（Tile），每个线程块（Block）负责计算输出矩阵中的一个子矩阵块内的所有元素，线程块内的每个线程对应子矩阵块中的一个具体元素。这种方法通过调整线程块的尺寸（tile_x, tile_y）来控制任务划分的粒度。

在该实现方式中，首先根据线程块编号（blockIdx）和线程块尺寸（tile_x, tile_y）计算出当前线程块所负责的子矩阵块在输出矩阵中的起始行、列编号（blockRow, blockCol）。然后，结合线程块内线程编号（threadIdx）计算当前线程在全局矩阵 C 中的行、列索引（row, col）。之后，和前两种方法一样，使用局部变量 sum，对矩阵 A 的第 row 行与矩阵 B 的第 col 列进行逐元素乘法并累加，完成对应矩阵元素的点积计算。

```
__global__ void matMulGlobalTile(const float* A, const float* B, float* C,
                                 int m, int n, int k, int tile_x, int tile_y) {
    int blockRow = blockIdx.y * tile_y;
    int blockCol = blockIdx.x * tile_x;
    int row = blockRow + threadIdx.y;
    int col = blockCol + threadIdx.x;
    if (row < m && col < k) {
        float sum = 0.0f;
        for (int j = 0; j < n; ++j) {
            sum += A[row * n + j] * B[j * k + col];
        }
        C[row * k + col] = sum;
    }
}
```

2.2 共享内存

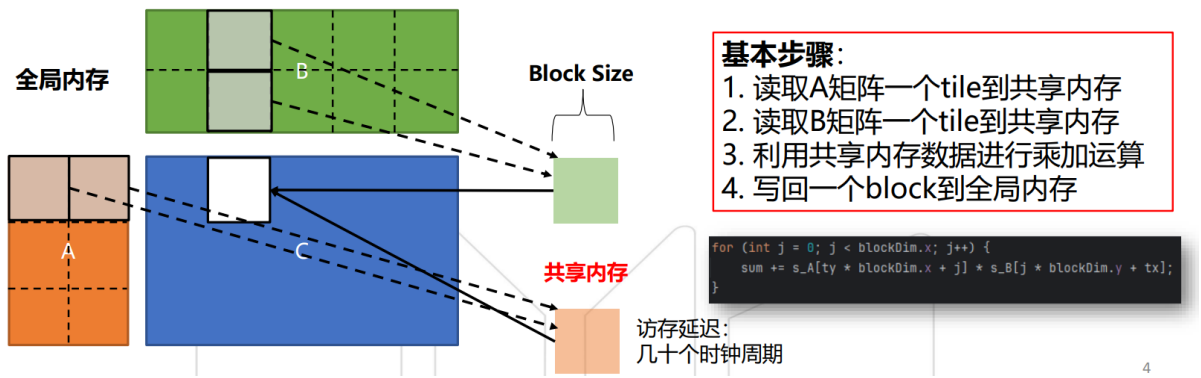

图 2: 共享内存

共享内存的原理是将频繁访问的全局内存数据缓存在速度更快的共享内存中，通过分块计算实现数据重用，减少全局内存访问次数。在本实现中，使用了两个大小为 TILE_DIM × TILE_DIM 的共享内存数组 tileA 和 tileB，分别用于缓存矩阵 A 的子块和矩阵 B 的子块。具体过程如下：

首先，根据线程块编号（blockIdx）和线程内编号（threadIdx）计算当前线程在输出矩阵 C 中的行、列索引（row, col）。随后，按照 Tile 划分方式，矩阵乘法被划分为若干个子块（Tile）累加。内层循环变量 t 控制当前加载第 t 块子矩阵。每次循环时，线程块内所有线程协同将矩阵 A 的当前子块和矩阵 B 的当前子块分别加载到共享内存中的 tileA 和 tileB 中。加载时，考虑矩阵边界，若越界则填充 0。

完成共享内存加载后，通过 __syncthreads() 确保所有线程完成数据加载，再由线程块内所有线程进行子块内逐元素乘法累加操作。内层累加完毕后，再次 __syncthreads()，确保所有线程完成当前子块计算，再进入下一子块加载及计算过程。循环结束后，每个线程将计算好的结果写入输出矩阵 C 对应位置。由于共享内存提高了数据重用率，显著减少了对全局内存的访问次数，从而提升了整体矩阵乘法性能。

```
__global__ void matMulBlockShared(const float *A, const float *B, float *C,
                                  int m, int n, int k) {
    __shared__ float tileA[TILE_DIM][TILE_DIM];
    __shared__ float tileB[TILE_DIM][TILE_DIM];
    int row = blockIdx.y * TILE_DIM + threadIdx.y;
    int col = blockIdx.x * TILE_DIM + threadIdx.x;
    float sum = 0.0f;
    for (int t = 0; t < (n + TILE_DIM - 1) / TILE_DIM; ++t) {
        if (row < m && t * TILE_DIM + threadIdx.x < n)
            tileA[threadIdx.y][threadIdx.x] =
                A[row * n + t * TILE_DIM + threadIdx.x];
        else
            tileA[threadIdx.y][threadIdx.x] = 0.0f;

        if (col < k && t * TILE_DIM + threadIdx.y < n)
            tileB[threadIdx.y][threadIdx.x] =
                B[(t * TILE_DIM + threadIdx.y) * k + col];
        else
            tileB[threadIdx.y][threadIdx.x] = 0.0f;
        __syncthreads();
        for (int i = 0; i < TILE_DIM; ++i)
            sum += tileA[threadIdx.y][i] * tileB[i][threadIdx.x];
        __syncthreads();
    }
```

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

2.3 寄存器分块技术

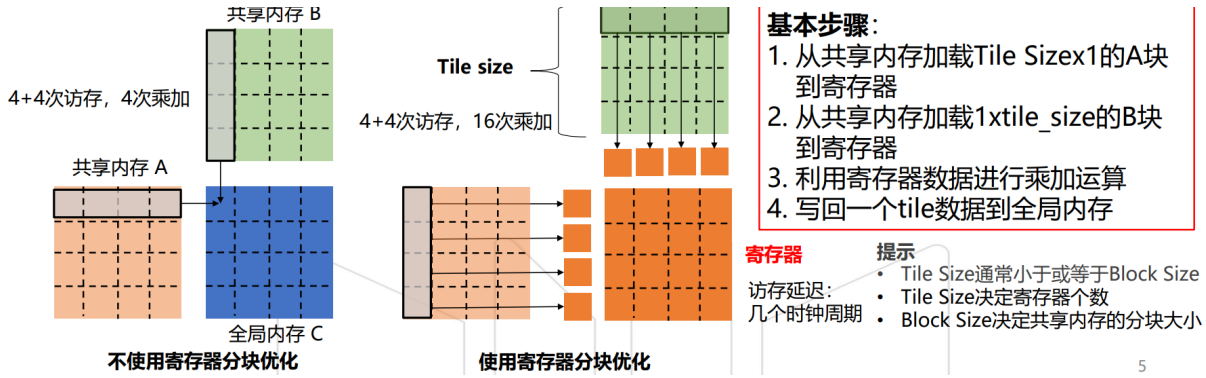

图 3: 寄存器分块技术

寄存器分块的原理是将矩阵划分为多个子块（Tile），每个线程块负责计算一个子块区域内的结果矩阵。为了减少对共享内存的频繁访问，在每次 Tile 块内的计算过程中，线程将需要频繁访问的子块数据从共享内存中加载至寄存器，利用寄存器高速缓存进行乘加运算，从而提升访存效率和计算访存比。

具体实现时，首先将矩阵 A 和 B 按照块划分的方式加载到共享内存 tileA 和 tileB 中，然后通过`__syncthreads()`保证所有线程完成共享内存加载。随后，每个线程将当前行和列对应位置的 tileA 和 tileB 中的元素加载到寄存器变量`regA`和`regB`，在寄存器内完成 TileDim 次乘法与累加操作，避免了计算过程中对共享内存 tileA 和 tileB 的重复访存，提高了访存局部性。完成一个子块的计算后，线程再次同步，继续加载下一个子块，直至遍历完全部子块，最终将累加得到的乘积和写回全局内存中的结果矩阵 C 中。

```
__global__ void matMulRegisterTile(const float *A, const float *B, float *C,
                                   int m, int n, int k)
{
    __shared__ float tileA[TILE_DIM][TILE_DIM];
    __shared__ float tileB[TILE_DIM][TILE_DIM];
    int row = blockIdx.y * TILE_DIM + threadIdx.y;
    int col = blockIdx.x * TILE_DIM + threadIdx.x;
    float regA, regB;
    float sum = 0.0f;
```

```
    for (int t = 0; t < (n + TILE_DIM - 1) / TILE_DIM; ++t)
    {
        if (row < m && t * TILE_DIM + threadIdx.x < n)
            tileA[threadIdx.y][threadIdx.x] =
                A[row * n + t * TILE_DIM + threadIdx.x];
        else
            tileA[threadIdx.y][threadIdx.x] = 0.0f;

        if (col < k && t * TILE_DIM + threadIdx.y < n)
            tileB[threadIdx.y][threadIdx.x] =
                B[(t * TILE_DIM + threadIdx.y) * k + col];
        else
            tileB[threadIdx.y][threadIdx.x] = 0.0f;
        __syncthreads();
        // 寄存器分块优化核心部分
        for (int i = 0; i < TILE_DIM; ++i)
        {
            regA = tileA[threadIdx.y][i];
            regB = tileB[i][threadIdx.x];
            sum += regA * regB;
        }
        __syncthreads();
    }
    if (row < m && col < k)
        C[row * k + col] = sum;
}
```

3 实验结果与分析

3.1 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	2.651	1.023	1.023
	16×16	1.861	0.906	0.907
	32×32	2.399	0.829	0.834
1024	8×8	22.234	8.029	8.034
	16×16	14.932	7.078	7.077
	32×32	18.035	6.528	6.522
2048	8×8	174.459	63.933	64.019
	16×16	118.749	56.225	56.229
	32×32	142.870	51.608	51.611

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	4.474	1.762	1.029
	16×16	7.951	2.772	0.934
	32×32	14.877	4.955	0.984
1024	8×8	38.277	13.993	8.099
	16×16	62.307	22.091	7.151
	32×32	117.006	39.046	7.093
2048	8×8	283.027	110.628	64.542
	16×16	497.853	174.939	56.397
	32×32	926.561	310.436	56.440

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	2.658	1.018	1.020
	16×16	1.852	0.908	0.900
	32×32	2.393	0.831	0.832
1024	8×8	22.203	8.035	8.031
	16×16	14.902	7.078	7.076
	32×32	18.100	6.518	6.527
2048	8×8	174.622	64.015	64.026
	16×16	118.777	56.229	56.234
	32×32	142.842	51.734	51.741

3.2 分析性能差异的原因

3.2.1 性能差异

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。

- 全局内存访问不连续（行列划分）：以 512 规模、32×32 线程块为例，“按列朴素”耗时 14.877 ms，远高于“按行朴素”的 2.399 ms。这差距主要因为按列访问 A 矩阵时，线程读写不是连续地址，无法合并访存，导致全局内存耗时很大。因此，当线程块按列划分后，线程访问 A、B 中的元素可能不再按连续地址读取，导致内存访问无法完全合并，出现大量零散的全局访存事务，吞吐率直线下降。

- 共享内存与寄存器复用：同样配置下，512/32 情况下“按行共享”仅 0.829 ms，“按行寄存器”0.834 ms，和 2.399 ms 相比加速约 2.9×，说明共享内存和寄存器缓存能大幅减少全局访存。共享内存优化版将同一行或同一块子矩阵预先装入共享内存，多线程复用，大幅降低对全局内存的访问次数；寄存器分块优化则进一步将复用率最高的子矩阵元素放入寄存器，读取延迟更低。

3.2.2 提高占用率

如何选择合适的线程块大小以提高占用率？

- 以按行寄存器为例，512 规模时 8×8 是 1.023 ms，16×16 0.907 ms，32×32 0.834 ms。Block 大小提升确实能进一步压榨缓存复用，但从 16×16 到 32×32 提升仅 8%（0.907→0.834），而寄存器占用翻倍，性价比开始递减。对于 2048 规模，8×8 是 64.019 ms，16×16 56.229 ms，32×32 51.753 ms。提升幅度分别约 12% 和 8%。结合资源占用，16×16 往往是更优中点。

- 综合数据分析，我们可得知如果线程块太大（如 32×32=1024 线程），单块就可能占满该 SM 的所有寄存器或共享内存，导致无法同时驻留第二个块，反而吞吐率下降。较小的线程块（8×8）更易保证每个 warp（32 线程）内的访问地址连续，从而更好地利用内存合并访问；过大时，warp 内不同线程跨越的地址可能分散，降低带宽利用。

3.2.3 不同的方式划分

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？

表 4: 不同划分方式的性能与复杂度对比

划分方式	性能影响	实现复杂度
按行（Row）	• 访问连续，行主序存储时全局访存可合并 • 共享/寄存器复用同一行数据，加速明显	• 最低，无需显式同步
按列（Column）	• 列主序存储时合并访存，否则非连续访问严重降速 • 性能波动依赖数据布局	• 低，与按行类似，可能需要先转置
按数据块（Block/Tiled）	• 最大算术强度，子矩阵多次复用 • 全局访存次数最少，大矩阵加速最高	• 较高，需管理块加载、边界处理与 `__syncthreads()`

- 按行划分在 512 规模、32×32 线程块时仅需 0.829 ms（共享）/0.834 ms（寄存器），而按列朴素高达 14.877 ms，性能相差近 18×；在 1024、16×16 配置下，按块共享/寄存器约 7.08 ms，对比按行朴素的 14.932 ms，加速约 2.1×，说明块划分在大规模矩阵上数据复用更充分，性能提升更显著。
- 寄存器优化下的列划分：值得注意的是，当改用寄存器分块优化后，按列划分的劣势几乎消除；例如在 512×512、8×8 时，列寄存器优化耗时 1.029 ms，仅比行

寄存器的 1.023 ms 略高 0.006 ms，和全局内存以及共享内存列划分完全处于劣势，这表明寄存器缓存能够有效解决原本的非连续访问问题。

- 综上所述，三种划分方式各有优劣：按行划分由于与行主序存储同步，能够实现高效的全局内存合并访问，并在共享内存或寄存器优化下进一步加速。按列划分在非列主序存储场景下朴素版本性能极差，但通过寄存器分块优化后可将非连续访问问题彻底缓解，实现与按行划分类似的高性能优势，同时仍保持较低的实现复杂度；而按数据块划分虽然在大规模矩阵（如 N 1024）以及多次复用子矩阵块场景下能够最大化算术强度、最显著地降低全局访存次数，但必须在每个线程块内部显式管理子块加载、边界判断与 `__syncthreads()` 同步，使得代码量和调试难度显著增加。

3.2.4 存储

何时应该优先考虑使用哪种存储？

- 寄存器：当单个线程内部对同一数据元素进行多次读写、计算复用率极高时，应优先将该元素载入寄存器，以利用寄存器的最低访问延迟提升性能；但需注意寄存器占用会限制每个 SM 上的线程块驻留数，应注意寄存器使用量，保持合理的占用率。

- 共享内存：当一个线程块内的多个线程需要反复访问同一子矩阵区域时，使用共享内存可以将数据从全局内存加载后在块内多次复用，大幅降低全局访存次数；适用于按数据块划分的大规模矩阵乘法，但需在加载后插入 `__syncthreads()` 以保证数据一致性，并防止 bank conflict。

- 全局内存：当数据仅需访问一次或复用率极低，且寄存器和共享内存资源均已满载时，才依赖全局内存存取；必须确保线程在同一 warp 内按照连续地址访问以实现访存合并，否则将产生大量非合并事务，导致带宽利用效率骤降。