中山大学计算机学院

本科生实验报告

（2025学年春季学期）

课程名称: 并行程序设计

实验	CUDA 矩阵乘法	专业(方向)	计算机科学与技术
学号	--	姓名	--
Email	--	完成日期	2025.6.5

实验目的

- CUDA并行矩阵乘法

实验过程和核心代码

首先初始化矩阵数据, 根据输入来分配对应大小的空间, 在cpu上初始化A, B矩阵并复制到对应cuda的矩阵中. 由于需要验证cuda矩阵乘法的正确性, 所以在cpu需要两个结果矩阵来分别存储cpu矩阵乘法的结果(`cpu_C`)和cuda计算的矩阵乘法结果(`cuda_C`).

```
// initialize host A, B
float* h_A = new float[n * n];
float* h_B = new float[n * n];
randomInitMatrix(h_A, n * n);
randomInitMatrix(h_B, n * n);

// cup result and cuda result
float* cpu_C = new float[n * n];
float* cuda_C = new float[n * n];

// cuda malloc
float* d_A;
float* d_B;
float* d_C;
cudaMalloc(&d_A, n * n * sizeof(float));
cudaMalloc(&d_B, n * n * sizeof(float));
cudaMalloc(&d_C, n * n * sizeof(float));

// cpu 初始化值复制到 cuda 中
cudaMemcpy(d_A, h_A, n * n * sizeof(float), cudaMemcpyHostToDevice);
cudaMemcpy(d_B, h_B, n * n * sizeof(float), cudaMemcpyHostToDevice);
```

接下来需要根据输入来设置线程块大小. 我们让每个线程负责一个位置的结果计算, 所以可以根据线程块大小来计算出grid的大小. 设置这两个参数来调用矩阵乘法核函数.

```
// 设置线程块大小与grid大小, 每个线程计算一个位置的矩阵乘法结果
dim3 blockDim(block_size, block_size);
dim3 gridDim(n / block_size, n / block_size);
```

在本次实验中, 我编写了三种不同的矩阵乘法核函数, 分别为**基础矩阵乘法核函数**、**共享内存分块矩阵乘法核函数**以及**循环展开矩阵乘法核函数**.

基础矩阵乘法核函数

首先计算出矩阵负责计算的结果矩阵的位置:

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

根据矩阵乘法的特性, 该位置的结果为:

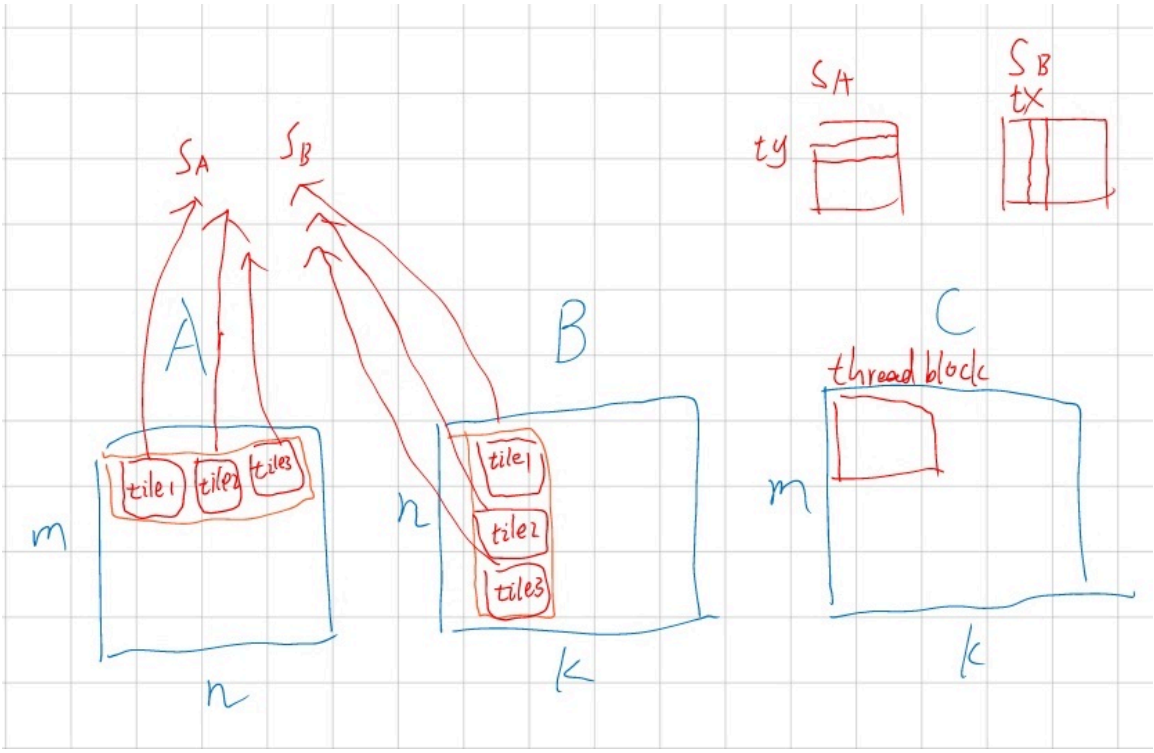
$$C_{row,col} = \sum_{i=1}^{n} A_{row,i} \times B_{i,col}$$

其中A的维度为 $m \times n$, B 的维度为 $n \times k$.
由于代码中使用的是一维矩阵, 计算实现如下:

```
if(row < m && col < k) {
    float sum = 0;
    // A row all elements mul B col all elements
    for(int i = 0; i < n; ++i) {
        sum += A[row * n + i] * B[i * k + col];
    }
    C[row * k + col] = sum;
}
```

共享内存分块矩阵乘法核函数

在基础矩阵乘法核函数中, 每次都要从全局内存取A,B对应位置的值, 没有充分利用线程块的共享内存.
由于共享内存是线程块内的所有线程共享, 所以我们可以考虑分块矩阵乘法的思路. 首先声明两个二维共享内存S_A和S_B分别存储A,B的对应块. 线程块内每个线程负责从全局内存取一个位置的数据到共享内存中, 等待所有线程取完数据后在使用共享内存中的值进行计算. 计算完毕后再重复以上步骤计算下一个分块. 累加每个分块计算的结果即可得到最终的结果.



```cpp
__global__ void matrixMulSharedKernel(const float* A, const float* B, float* C, int m, int n, int k) {
    __shared__ float sharedTileA[32][32]; // 共享A矩阵分块
    __shared__ float sharedTileB[32][32]; // 共享B矩阵分块

    // 该线程负责及计算C矩阵row行col列的结果
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;

    int tx = threadIdx.x;
    int ty = threadIdx.y;

    float sum = 0.0f;

    // 分块数量
    int tile_count = (n + blockDim.x - 1) / blockDim.x;

    // 循环处理分块
    for(int tile = 0; tile < tile_count; ++tile) {
        // 当前分块线程负责加载A的位置 (row, tile * blockDim.x + tx)
        int tile_idx_A = row * n + tile * blockDim.x + tx;
        // 当前分块线程负责加载B的位置 (tile * blockDim.y + ty, col)
        int tile_idx_B = (tile * blockDim.y + ty) * k + col;

        // 加载A,B矩阵对应分块的值到共享内存中
        if(row < m && tile * blockDim.x + tx < n) {
            sharedTileA[ty][tx] = A[tile_idx_A];
        } else {
            sharedTileA[ty][tx] = 0.0f;
        }

        if(col < k && tile * blockDim.y + ty < n) {
            sharedTileB[ty][tx] = B[tile_idx_B];
        } else {
            sharedTileB[ty][tx] = 0.0f;
        }

        __syncthreads();

        // 计算子块乘法结果
        for(int i = 0; i < blockDim.x; ++i) {
            sum += sharedTileA[ty][i] * sharedTileB[i][tx];
        }

        __syncthreads();
    }

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

由于这里共享内存大小设置为了32, 所以在程序运行时线程块的大小最大为32 x 32.
在基础矩阵乘法中, 每个线程在**全局内存中从A,B矩阵分别取n次数据**. 而在共享内存分块矩阵乘法中, **每个线程在一个分块中只需要在全局内存中取一次A,B的数据**. 一共需要取分块数量, 即`(n + blockDim.x - 1) / blockDim.x`次A,B的数据.

从计算与全局内存访问比来看:

- 基础版本: 计算/访问比 = $1/2$
- 共享内存版本: 计算/访问比 = $\text{blockDim.x}/2$
  提升与内存块的大小也有关, 内存块越大, 提升越大.

循环展开矩阵乘法核函数

循环展开优化是基于A矩阵的行数据是连续的, 使用cuda内置的`float4`可以**一次性加载4个连续的`float`数据**. 计算4个数据的**指令之间也没有依赖, 可以通过多发射来提高性能**.

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

if(row < m && col < k) {
    // CUDA 内置向量类型
    float4 sum = make_float4(0.0f, 0.0f, 0.0f, 0.0f);

    // n & (~3) 向下对齐4的倍数
    int floor4 = (n & (~3));

    for(int i = 0; i < floor4; i += 4) {
        float4 a = reinterpret_cast<const float4*>(&A[row * n + i])[0];
        float4 b = make_float4(B[i * k + col], B[(i + 1) * k + col],
                               B[(i + 2) * k + col], B[(i + 3) * k + col]);
        sum.x += a.x * b.x;
        sum.y += a.y * b.y;
        sum.z += a.z * b.z;
        sum.w += a.w * b.w;
    }

    C[row * k + col] = sum.x + sum.y + sum.z + sum.w;
    // 不足4列的部分
    for(int i = floor4; i < n; ++i) {
        C[row * k + col] += A[row * n + i] * B[i * k + col];
    }
}
```

实验结果

详细结果见`result.log`

运行时间单位：ms

Matrix Size	Block Size	Global Memory	Shared Memory	Loop Unrolling
128	8	0.2776	0.1102	**0.0756**
128	16	0.2401	**0.0582**	0.0831
128	32	0.2102	**0.0615**	0.0944
256	8	0.9582	0.5768	**0.3785**
256	16	0.6667	**0.2728**	0.3019
256	32	0.6578	**0.2318**	0.4360
512	8	6.5856	4.1997	**2.6697**
512	16	4.3306	**1.9180**	2.0946
512	32	3.3091	**1.6048**	2.7649
1024	8	52.4293	34.1861	**23.2854**
1024	16	33.7877	**14.9572**	17.5248
1024	32	24.6340	**12.3752**	21.4005
2048	8	387.8540	223.2421	**134.9380**
2048	16	258.9721	**100.6720**	111.3851
2048	32	196.3463	**90.9498**	149.2692

线程块大小影响

- 在全局内存实现中, 线程块大小越大运行时间越短. 可能由于线程块越大, 使线程块的数量少, 减少了调度的开销, 使性能提升.
- 在共享内存实现中, 在规模为128的矩阵线程块大小在16使性能最好, 其他矩阵规模下都是在线程块大小为32时达到了最好性能. 在矩阵规模较小时可能是由于**同步带来的性能下降大于共享内存带来的提升**, 导致线程块大小为32时性能不如线程块大小为16. 而在**矩阵规模较大时共享内存带来的提升更大**, 所以在线程块为32时达到最好的性能.
- 在循环展开实现中, 矩阵规模在128时, 线程块大小为8时达到最好性能. 其他矩阵规模下都是在线程块大小为16时达到最好性能. 可能是由于**线程块中的线程越多, 每个线程分配的寄存器越少**, 而多发射使计算数据较多, 从而导致性能下降.

矩阵规模影响

在所有的实现方式中, 随矩阵规模增大两倍, 即数据量增大4倍时, 运行时间的增大倍数都超过了4倍. 这是由于**随矩阵规模增加, 在相同线程块大小下, 线程块的数量会增加, 导致线程块的调度开销增大**, 使得运行时间增大倍率超过了数据量的增大倍率.

不同实现方式影响

表格中各行加粗值表示在相同矩阵规模与线程块大小下性能最好的实现. 可以看到在**线程块大小为8时循环展开在各矩阵规模下都有最好的性能**. 而在**其他线程块大小时, 各矩阵规模都是共享内存实现方式达到最好性能**. 这是由于在线程块较小时, 共享内存的 计算/全局访问比 更小, 不能充分发挥共享内存的优势. 而线程块较小时, 线程块中线程数量较少, 循环展开实现中寄存器的分配足够, 能够充分发挥多发射性能优势.

观察表格可以看到全局内存实现不论线程块大小与矩阵规模运行时间都是最长的. 说明共享内存与循环展开都能显著提高矩阵乘法的性能.

实验感想

通过本次实验, 我对cuda编程有了更深的了解. 如果在解决问题时能够充分利用cuda的共享内存, 就可以大大提升性能. 在**问题的不同实现方式中, 也要根据实现方式的特点来分析如何配置线程块的大小以发挥该实现方式最大的优势**. 另外, 在本实验的共享内存实现方式的计算子块结果循环中, 也可以结合循环展开来进一步探索.