

第三讲 – 内存与数据划分



目录

- 内存访问效率 (Memory Access Efficiency)
- 分块矩阵乘法 (Tiled Matrix Multiplication)
- 分块矩阵乘法内核 (Tiled Matrix Multiplication Kernel)
- 处理分块中的边界条件 (Boundary Conditions)
- 任意大小矩阵的分块内核 (Tiled Kernel)

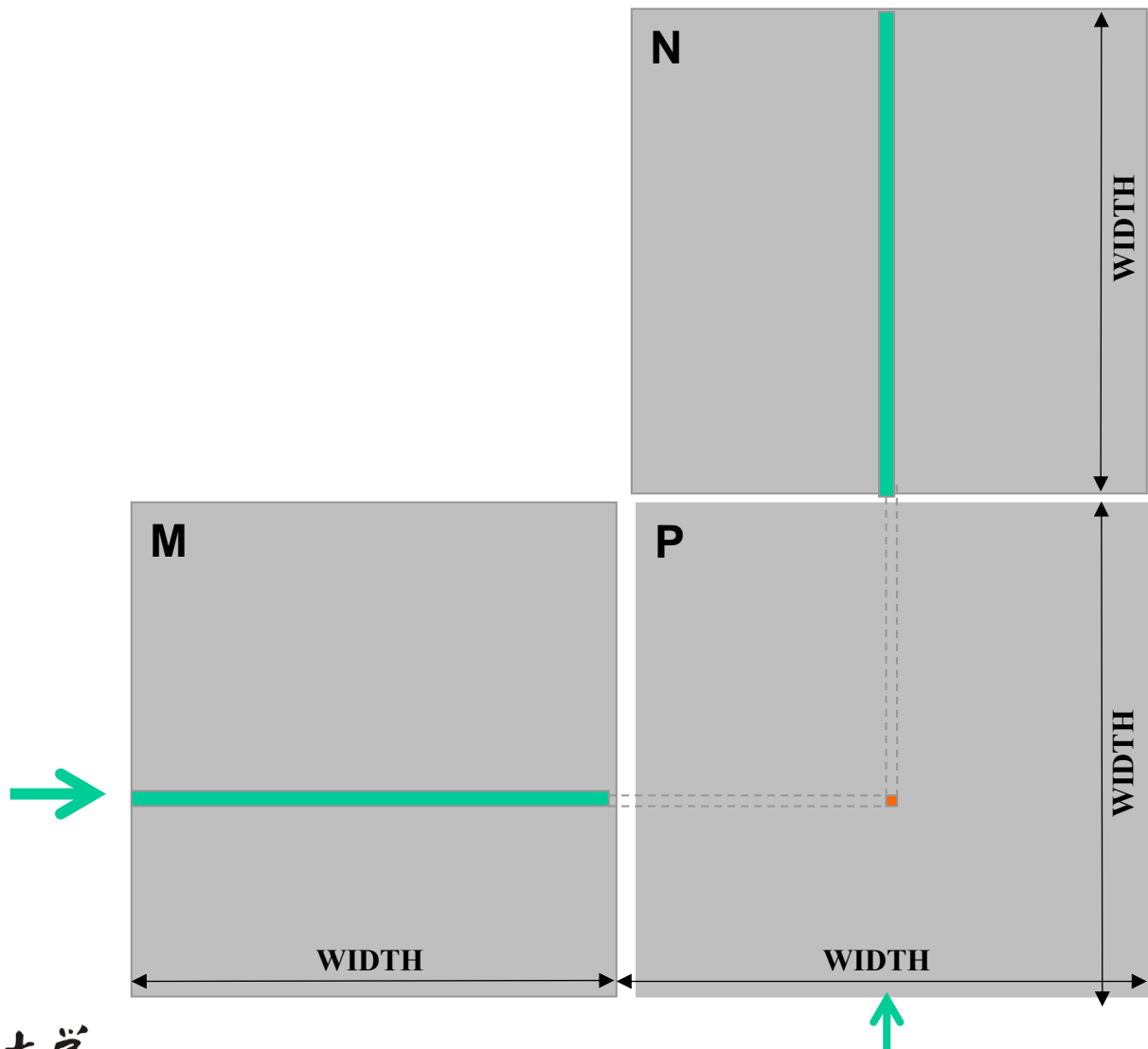


小节目标

- 学习在并行程序中有效使用 CUDA 内存类型
 - 内存访问效率的重要性
 - 寄存器 (Register)、共享内存 (Shared Memory)、全局内存 (Global Memory)
 - 范围 (Scope) 和周期 (Lifetime)



示例 – 矩阵乘法



简单的矩阵乘法

```
__global__ void MatrixMulKernel(float* M, float* N, float* P, int Width)
{
    // Calculate the row index of the P element and M
    int Row = blockIdx.y*blockDim.y+threadIdx.y;

    // Calculate the column index of P and N
    int Col = blockIdx.x*blockDim.x+threadIdx.x;

    if ((Row < Width) && (Col < Width)) {
        float Pvalue = 0;
        // each thread computes one element of the block sub-matrix
        for (int k = 0; k < Width; ++k) {
            Pvalue += M[Row*Width+k]*N[k*Width+Col];
        }
        P[Row*Width+Col] = Pvalue;
    }
}
```



在GPU上的性能评估

- 计算/通信比 **CGMA ratio** (Compute to Global Memory Access ratio)：每次访问 CUDA 内核区域内的全局内存时执行的浮点计算次数。
- 计算/通信比越大，表示在GPU上的性能越好。

矩阵乘法的计算/通信比是多少呢？



矩阵乘法的计算/通信比计算

- 所有线程为获得输入矩阵元素而访问全局内存
 - 每次浮点加法需要一次内存访问 (4 bytes)
 - 计算/通信比为1
 - 内存带宽/浮点计算速度 (FLOPS) 为 4B/s
- 假设GPU拥有
 - 1500 GFLOPS 的峰值浮点计算速度 , 200 GB/s 的 DRAM 带宽
 - 峰值 FLOPS 速度 $4 \times 1500 = 6000$ GB/s
 - 200 GB/s 的内存带宽将执行速度限制在 50 GFLOPS
- 这将执行速度限制为设备峰值浮点执行率的 **3.3% (50/1500)** !
- 需要大幅减少内存访问以接近 1,500 GFLOPS
- 需要达到 30 的计算/通信比 , 来达到峰值的 1,500 GFLOPS

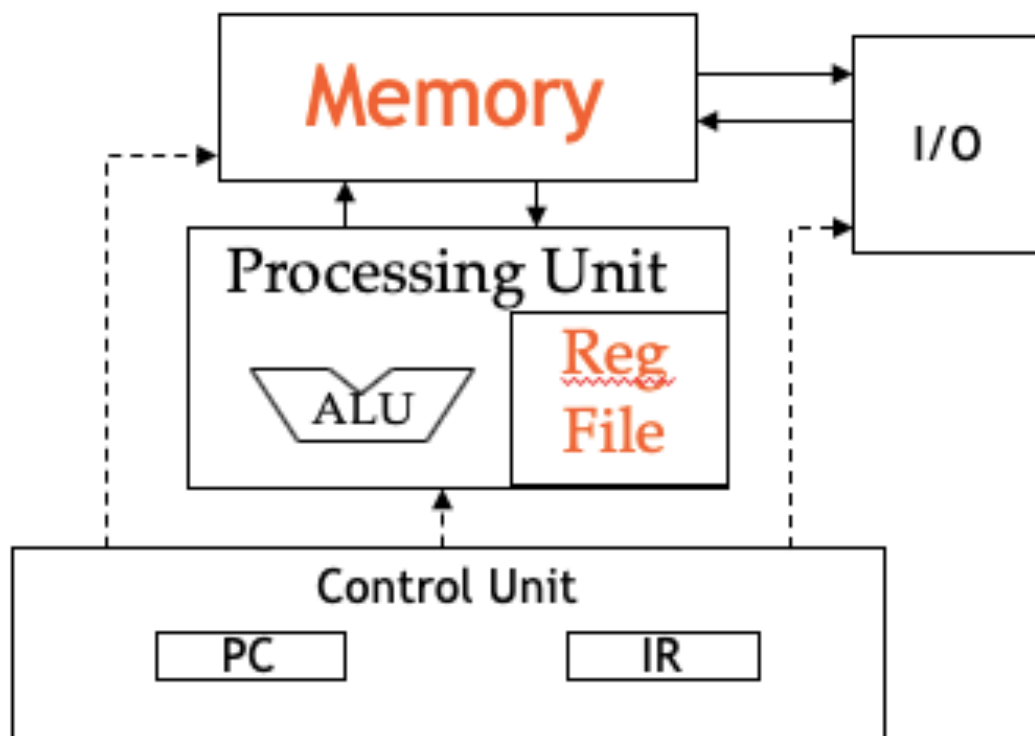


提问

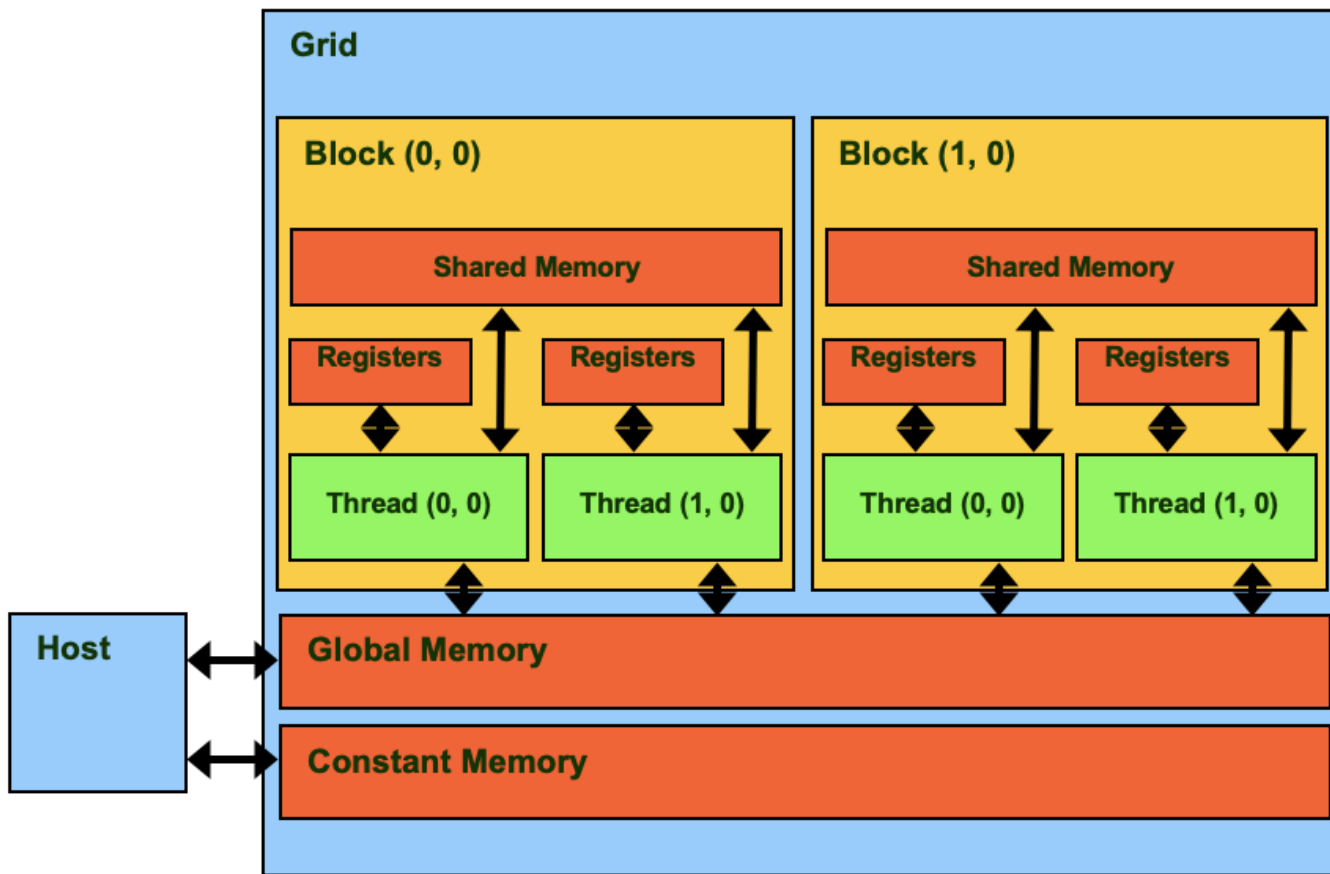
- 如何提升内存访问效率？
 - 提高计算量To increase calculation
 - 提高利用率To improve utilization
 - 合理利用内存层次结构
 - 全局内存
 - 共享内存
 - 寄存器



Von-Neumann 模型中的内存和寄存器



CUDA 内存视图



声明 CUDA 变量

Variable declaration	Memory	Scope	Lifetime
int LocalVar;	register	thread	thread
__device__ __shared__ int SharedVar;	shared	block	block
__device__ int GlobalVar;	global	grid	application
__device__ __constant__ int ConstantVar;	constant	grid	application

- 与 `__shared__` 或 `__constant__` 一起使用时 ,
`__device__` 是可选的
- 自动变量 (Automatic variables) 存在寄存器中

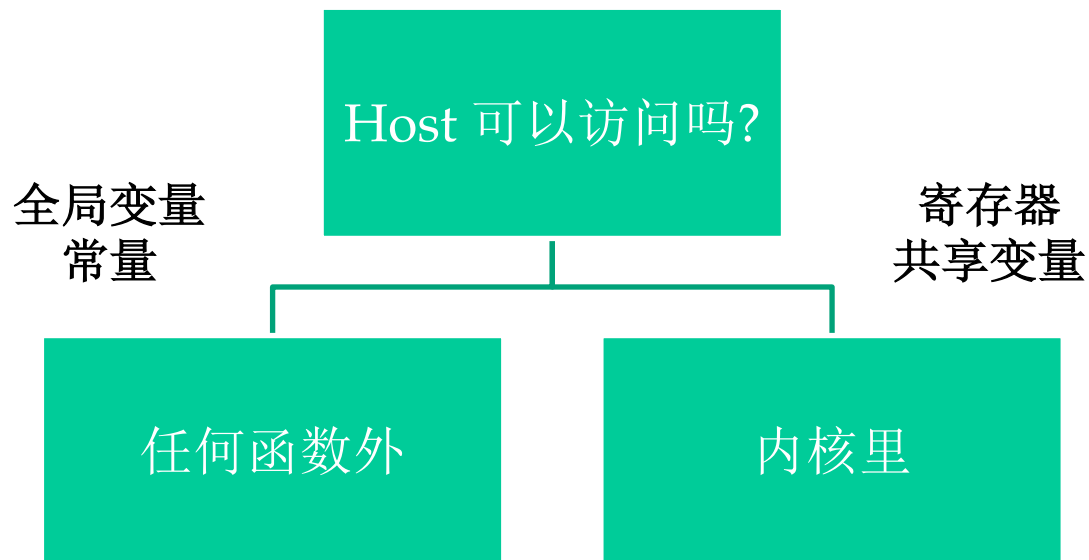


声明共享内存变量

```
void blurKernel(unsigned char * in, unsigned char * out,  
int w, int h)  
{  
    __shared__ float ds_in[TILE_WIDTH][TILE_WIDTH];  
  
    ...  
}
```



在哪里声明变量?

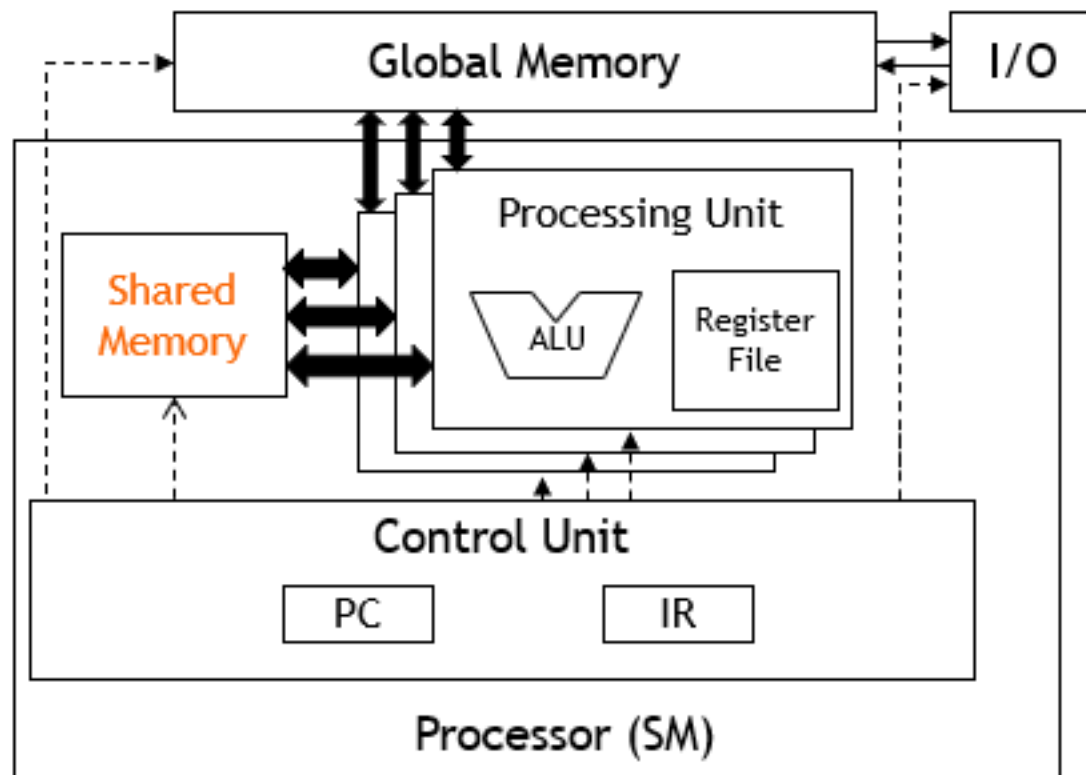


CUDA 中的共享内存 (Shared Memory)

- 一种特殊类型的内存，其内容在内核源代码中明确定义和使用
 - 每个 SM 中有 1 个共享内存
 - 较之全局内存，有着更高的访问速度（在延迟和吞吐量方面）
 - 生命周期 - 线程块，对应线程完成终止执行后内容会消失
 - 通过内存 load/store 命令来访问共享内存
 - 一种 Scratchpad Memory



CUDA 内存的硬件视图



目录

- 内存访问效率 (Memory Access Efficiency)
- **分块矩阵乘法 (Tiled Matrix Multiplication)**
- 分块矩阵乘法内核 (Tiled Matrix Multiplication Kernel)
- 处理分块中的边界条件 (Boundary Conditions)
- 任意矩阵维度的分块内核 (Tiled Kernel)



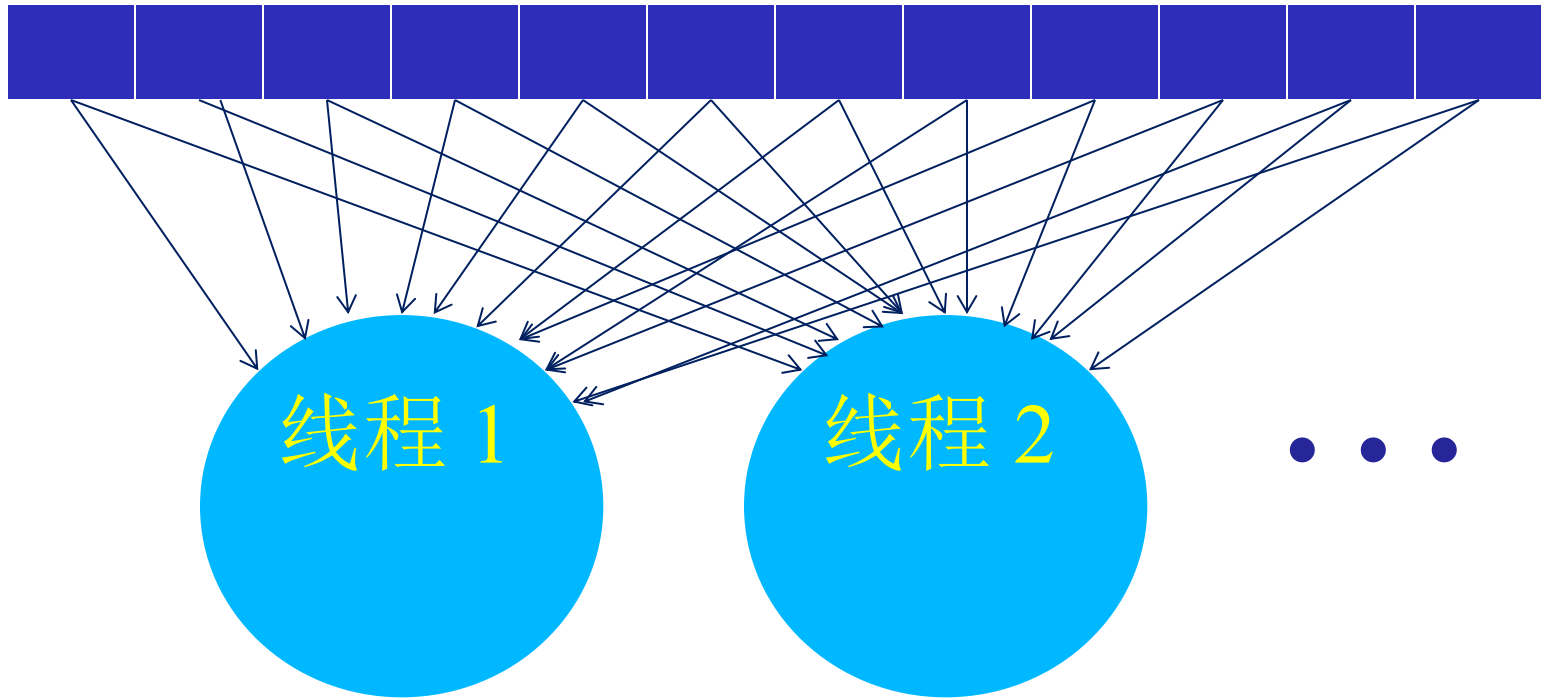
小节目标

- 了解分块并行算法的动机和思路
 - 减少内存带宽对并行内核性能的限制影响
 - 分块算法和屏障同步

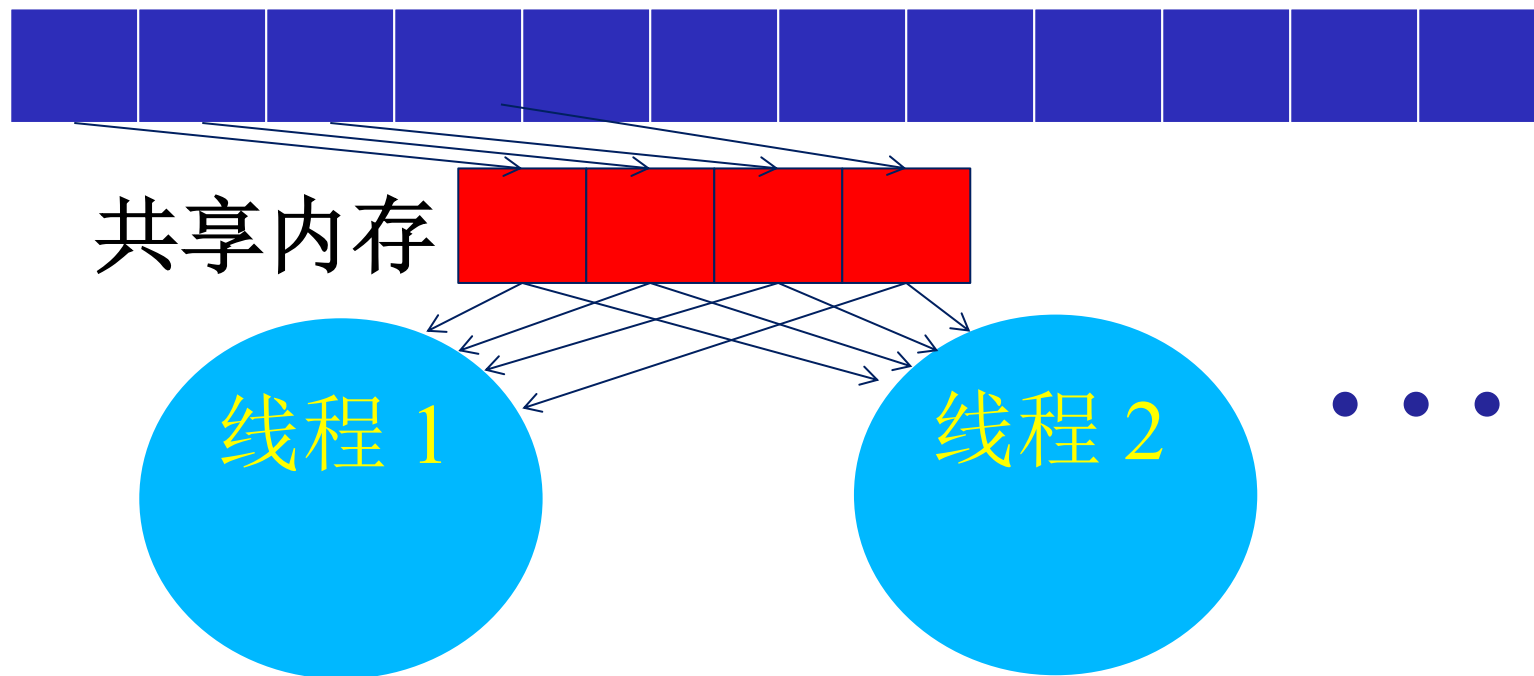


矩阵乘法内核的全局内存访问模式

全局内存



分块（ Tiling ）/阻塞（ Blocking ）

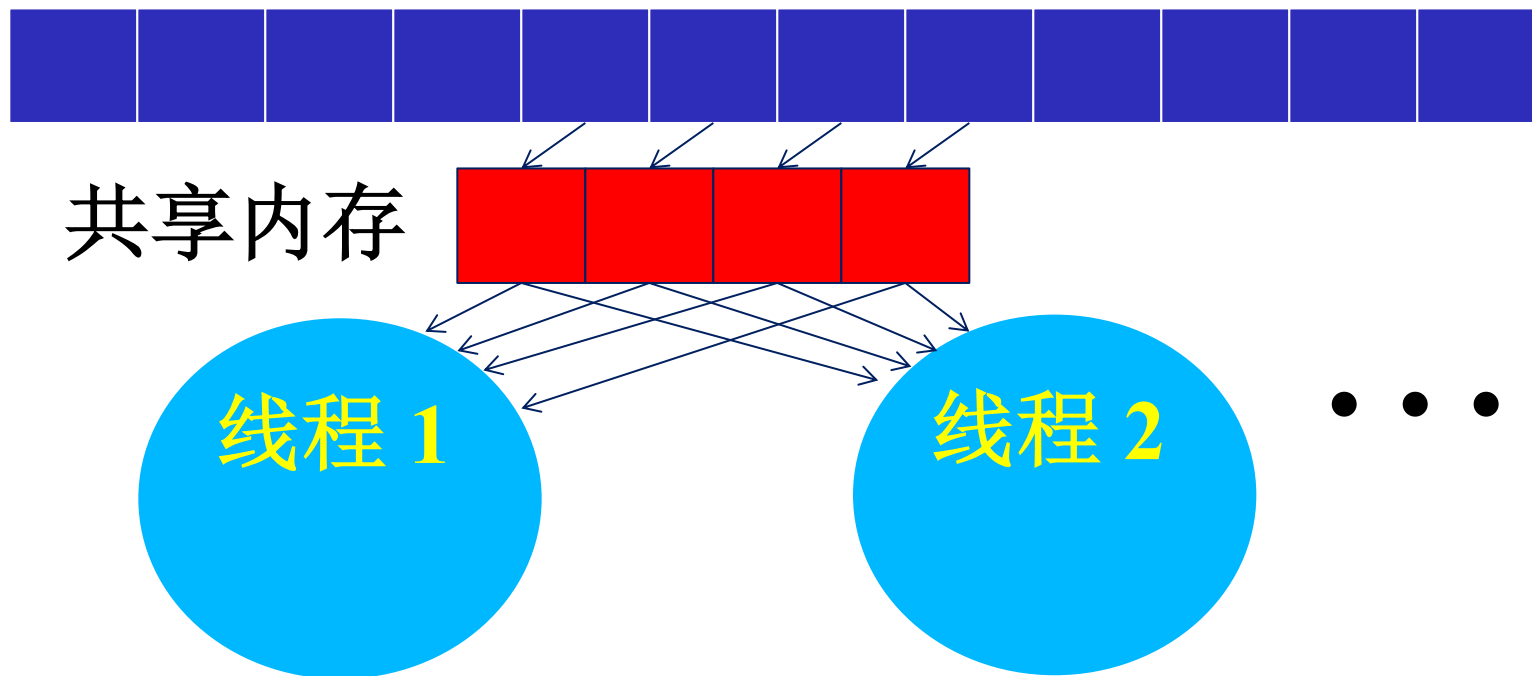


将全局内存内容划分为多个分片

在每个时间点将线程的计算集中在一个或少量的分片上



分块（ Tiling ）/阻塞（ Blocking ）



分块的基本概念

- 在拥堵的交通系统中，减少车辆可以大大改善拥堵
 - 通勤者之间的拼车
 - 用于全局内存访问的分块
 - 驾驶员 = 访问其内存数据操作数的线程
 - 车辆 = 内存访问的请求



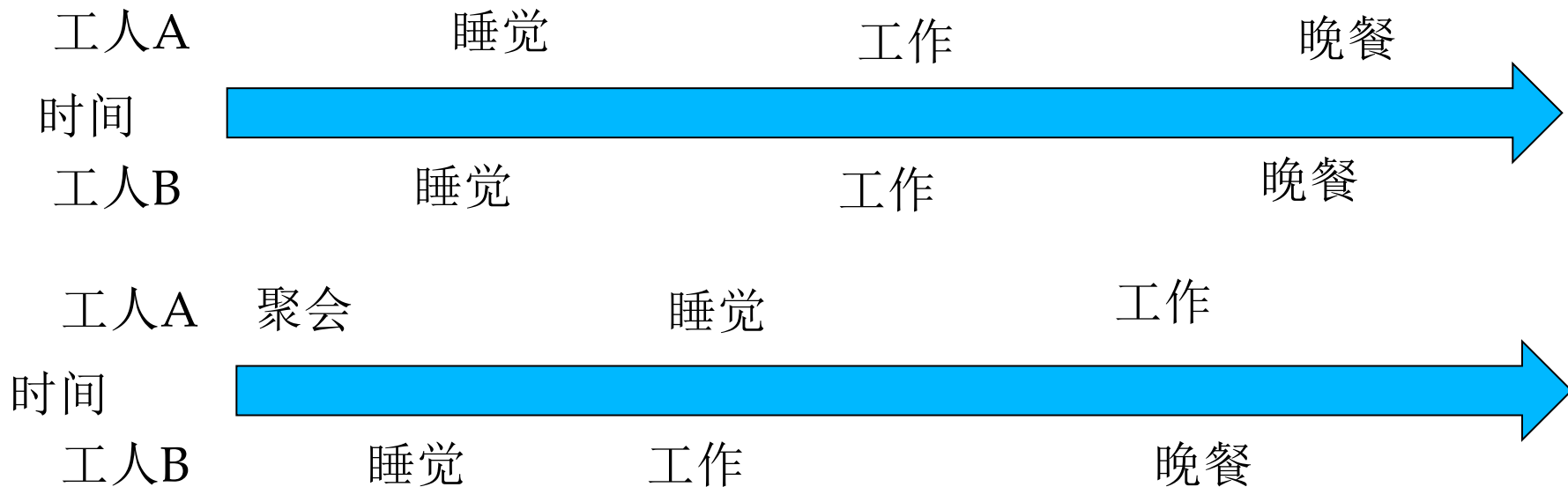
一些计算对分块操作更具挑战性

- 一些拼车方案可能比其他拼车方案更容易
 - 拼车者有更相似的工作安排
 - 一些车辆更适合拼车
- 分块也存在类似的挑战



拼车需要同步

- 当拼车者有相似的时间表时，拼车方案有正向效果

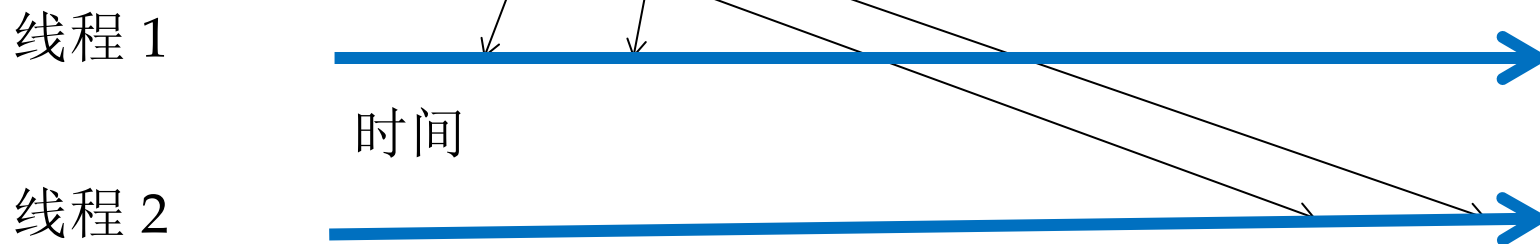
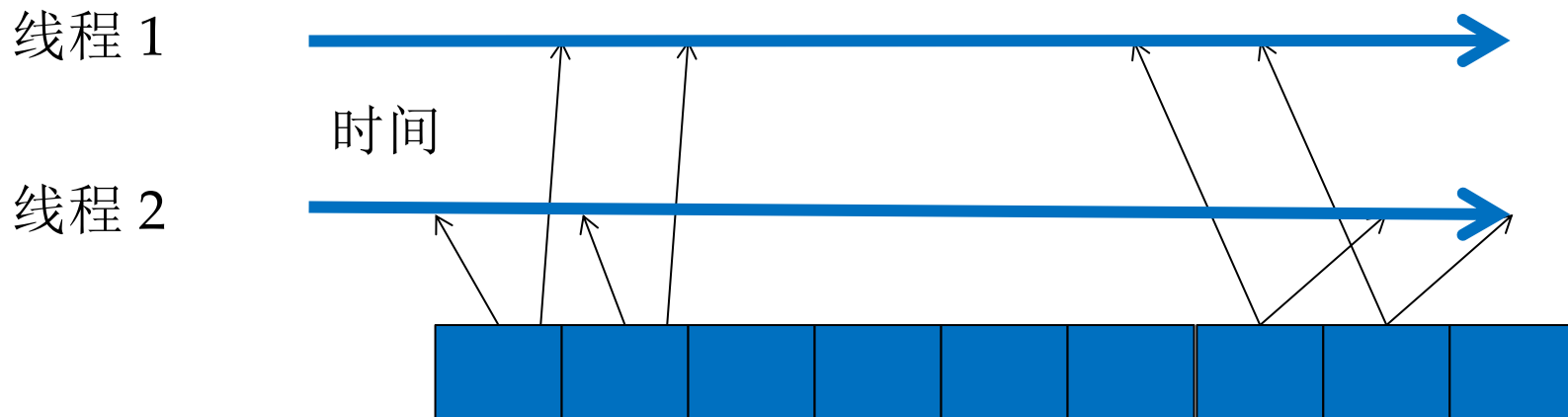


- 当拼车者的计划存在很大差异时，拼车方案难以起效



分块也是如此

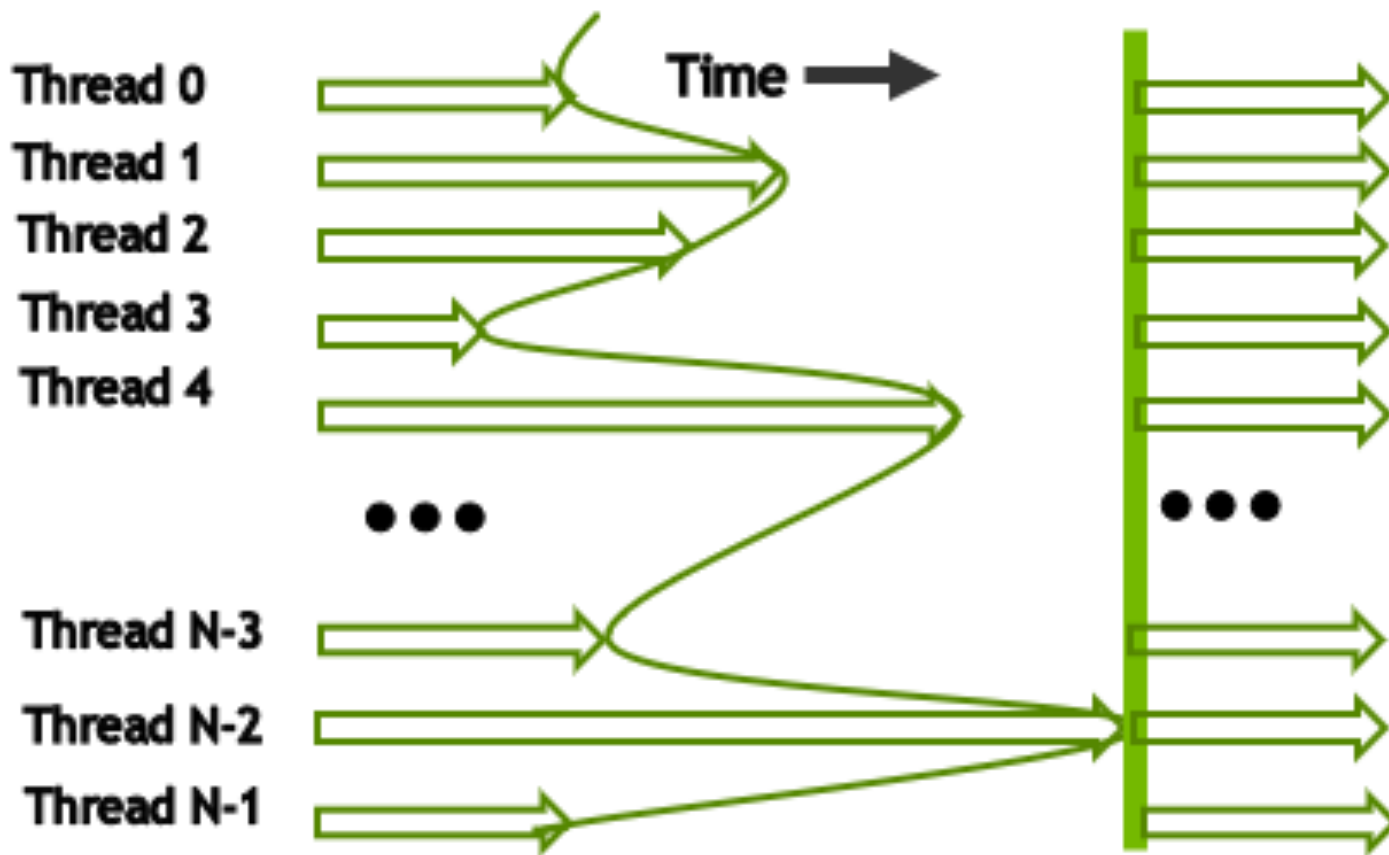
- 有效果：当线程有着相似的访问时间



- 没有效果：当线程有着极具差异的访问时间



线程同步



分块技术中的同步

- 识别由多个线程访问的全局内存的分片
- 将分片内容从全局内存加载到共享内存
- 使用**同步**确保所有线程都准备好开始阶段
- 让多个线程从共享内存中访问它们的数据
- 使用**同步**确保所有线程都完成了当前阶段
- 移至下一个分片



目录

- 内存访问效率 (Memory Access Efficiency)
- 分块矩阵乘法 (Tiled Matrix Multiplication)
- **分块矩阵乘法内核 (Tiled Matrix Multiplication Kernel)**
- 处理分块中的边界条件 (Boundary Conditions)
- 任意矩阵维度的分块内核 (Tiled Kernel)



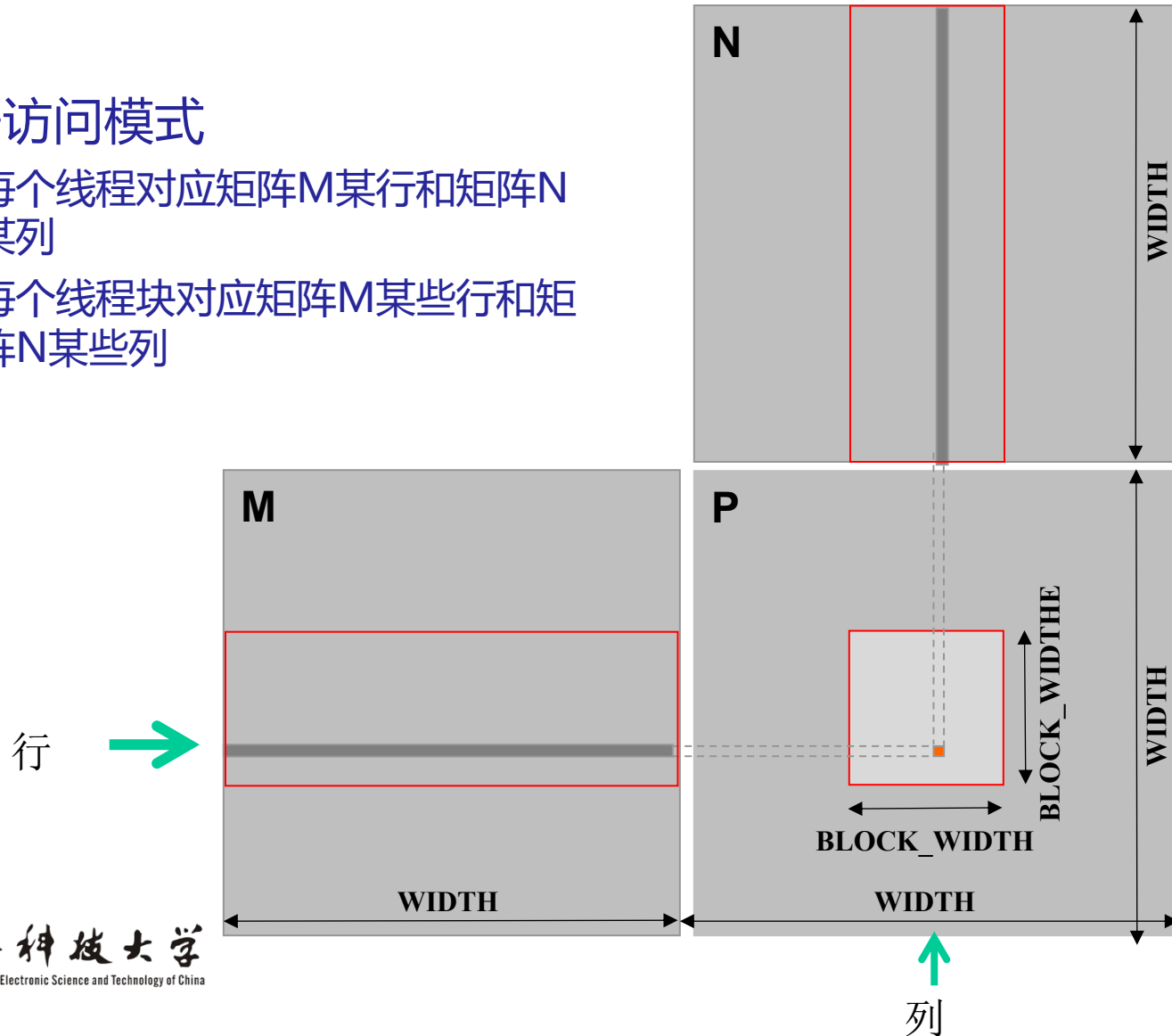
小节目标

- 了解矩阵乘法的分块并行算法的设计
 - 载入分片
 - 分阶段执行
 - 屏障同步



矩阵乘法

- 数据访问模式
 - 每个线程对应矩阵M某行和矩阵N某列
 - 每个线程块对应矩阵M某些行和矩阵N某些列



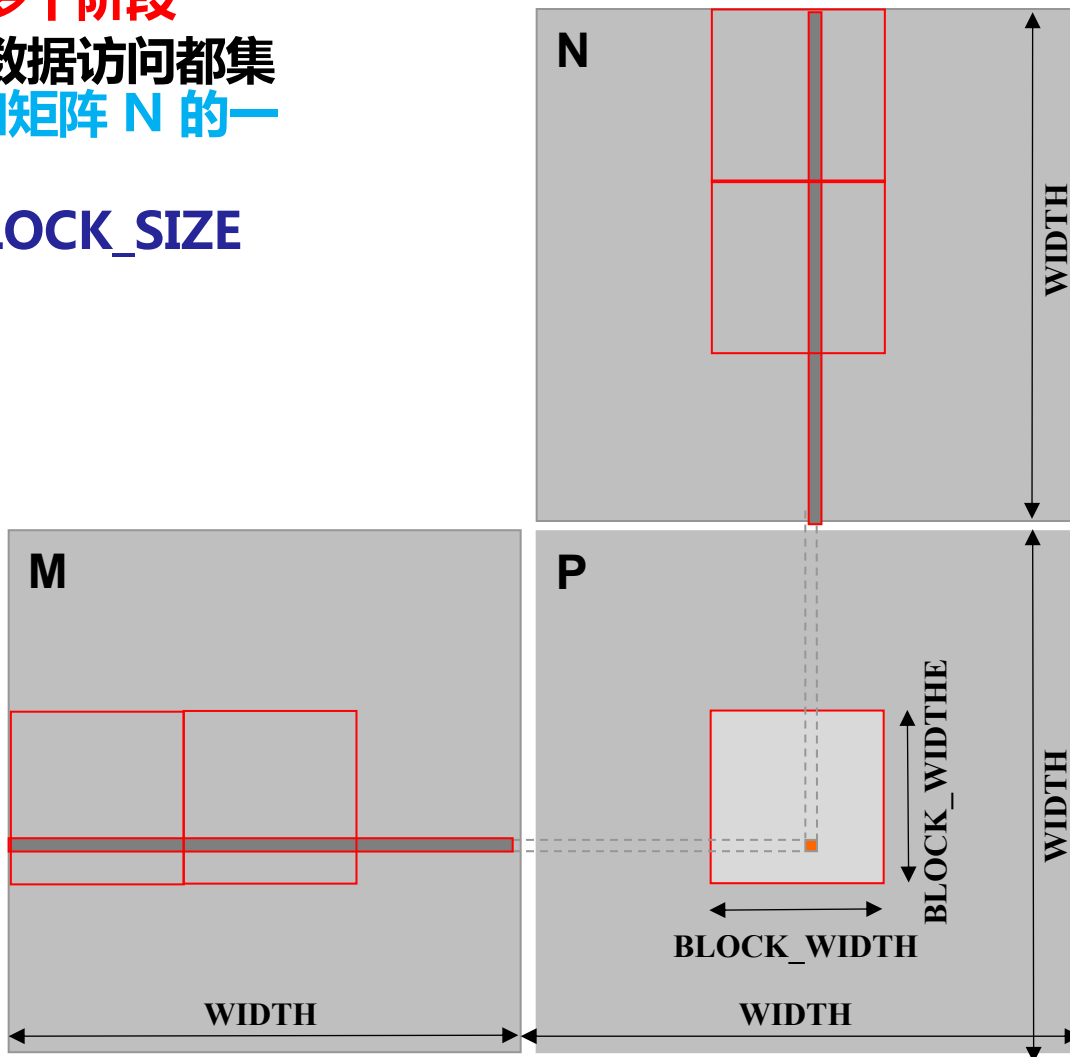
分块矩阵乘法

- 将每个线程的执行分解为**多个阶段**
- 这样线程块在每个阶段的数据访问都集中在**矩阵 M 的一个分片和矩阵 N 的一个分片**
- 分片在每个维度中都有 **BLOCK_SIZE** 个元素

$$A = \begin{pmatrix} A_{11} & \cdots & A_{1r} \\ \vdots & \cdots & \vdots \\ A_{s1} & \cdots & A_{sr} \end{pmatrix} B = \begin{pmatrix} B_{11} & \cdots & B_{1r} \\ \vdots & \cdots & \vdots \\ B_{t1} & \cdots & B_{tr} \end{pmatrix}$$

$$AB = \begin{pmatrix} C_{11} & \cdots & C_{1r} \\ \vdots & \cdots & \vdots \\ C_{s1} & \cdots & C_{sr} \end{pmatrix}$$

$$C_{ij} = \sum_{k=1}^t A_{ik} B_{kj} \quad (i = 1, \dots, s; j = 1, \dots, r)$$

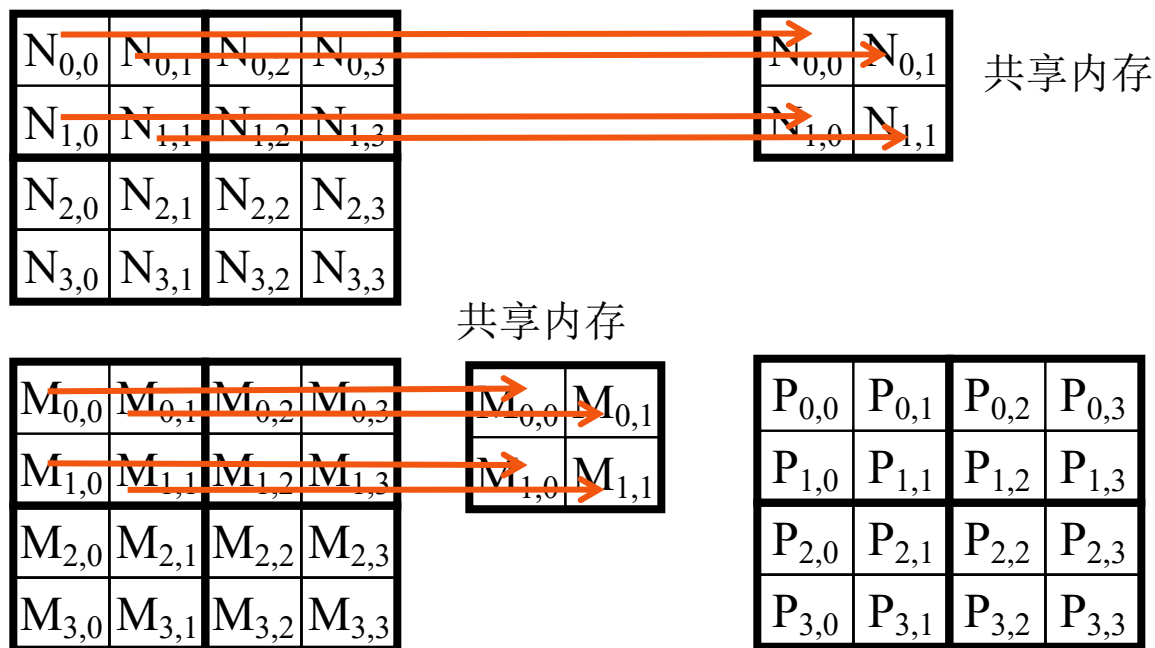


载入分片

- 同一线程块中的所有线程都参与
 - 在分块操作代码中，每个线程加载矩阵 M 的一个元素和矩阵 N 的一个元素



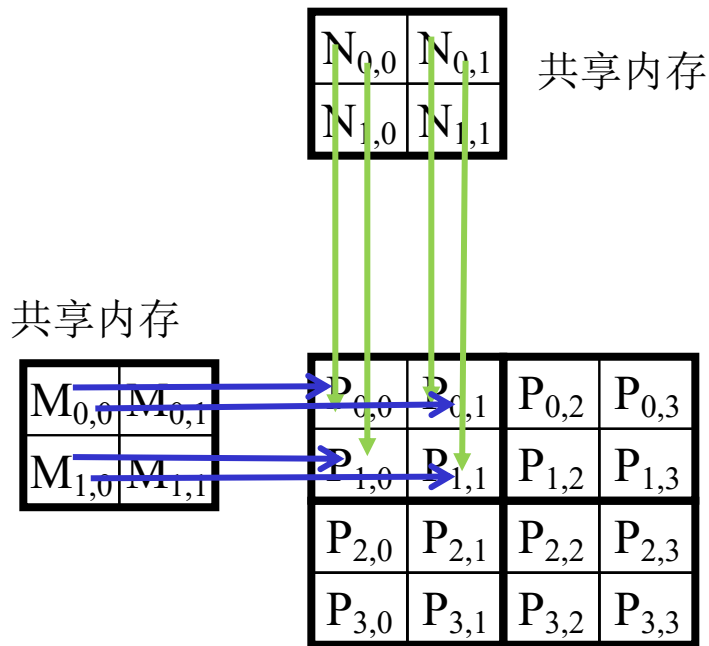
阶段 0 : 载入块(0 , 0)



阶段 0 : 计算块(0 , 0)(第0次迭代)

$N_{0,0}$	$N_{0,1}$	$N_{0,2}$	$N_{0,3}$
$N_{1,0}$	$N_{1,1}$	$N_{1,2}$	$N_{1,3}$
$N_{2,0}$	$N_{2,1}$	$N_{2,2}$	$N_{2,3}$
$N_{3,0}$	$N_{3,1}$	$N_{3,2}$	$N_{3,3}$

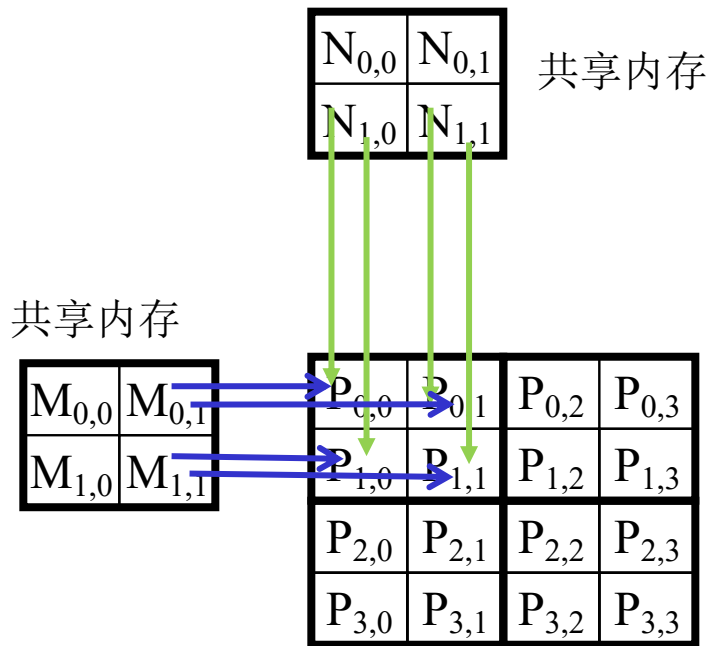
$M_{0,0}$	$M_{0,1}$	$M_{0,2}$	$M_{0,3}$
$M_{1,0}$	$M_{1,1}$	$M_{1,2}$	$M_{1,3}$
$M_{2,0}$	$M_{2,1}$	$M_{2,2}$	$M_{2,3}$
$M_{3,0}$	$M_{3,1}$	$M_{3,2}$	$M_{3,3}$



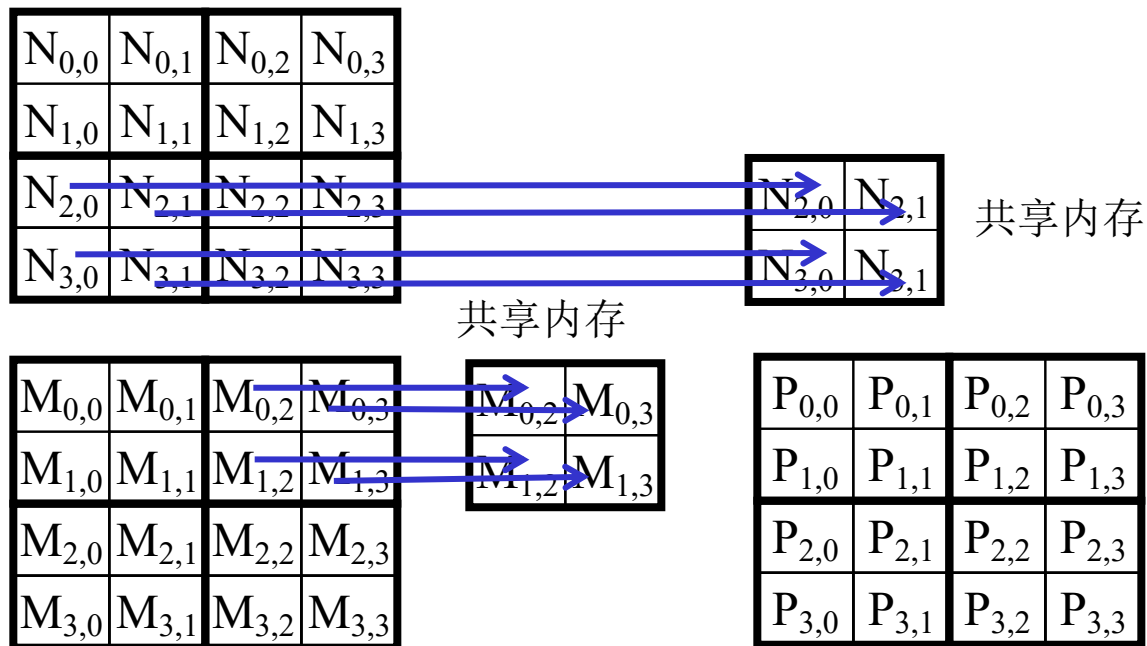
阶段 0 : 计算块(0 , 0)(第1次迭代)

$N_{0,0}$	$N_{0,1}$	$N_{0,2}$	$N_{0,3}$
$N_{1,0}$	$N_{1,1}$	$N_{1,2}$	$N_{1,3}$
$N_{2,0}$	$N_{2,1}$	$N_{2,2}$	$N_{2,3}$
$N_{3,0}$	$N_{3,1}$	$N_{3,2}$	$N_{3,3}$

$M_{0,0}$	$M_{0,1}$	$M_{0,2}$	$M_{0,3}$
$M_{1,0}$	$M_{1,1}$	$M_{1,2}$	$M_{1,3}$
$M_{2,0}$	$M_{2,1}$	$M_{2,2}$	$M_{2,3}$
$M_{3,0}$	$M_{3,1}$	$M_{3,2}$	$M_{3,3}$



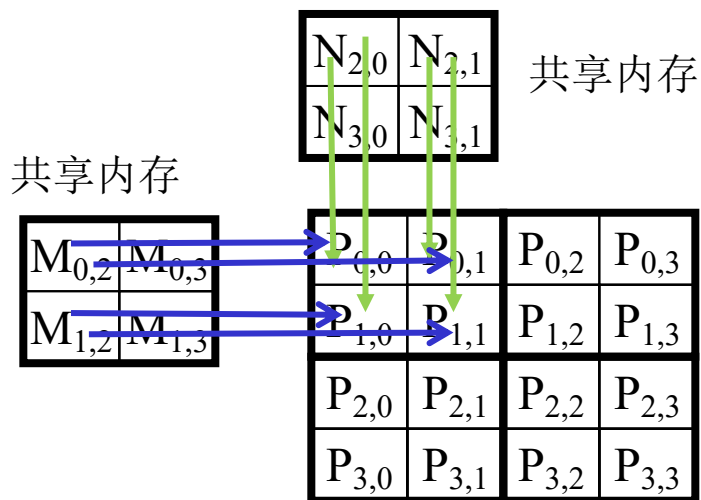
阶段 1：载入块(0,0)



阶段 1：计算块(0,0)(第0次迭代)

$N_{0,0}$	$N_{0,1}$	$N_{0,2}$	$N_{0,3}$
$N_{1,0}$	$N_{1,1}$	$N_{1,2}$	$N_{1,3}$
$N_{2,0}$	$N_{2,1}$	$N_{2,2}$	$N_{2,3}$
$N_{3,0}$	$N_{3,1}$	$N_{3,2}$	$N_{3,3}$

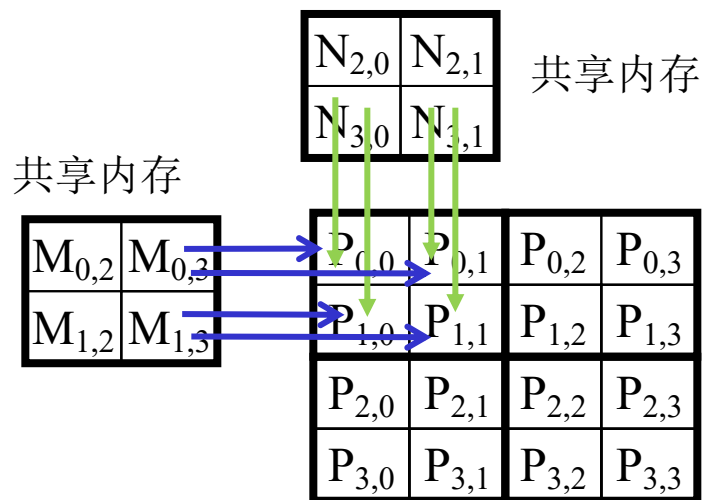
$M_{0,0}$	$M_{0,1}$	$M_{0,2}$	$M_{0,3}$
$M_{1,0}$	$M_{1,1}$	$M_{1,2}$	$M_{1,3}$
$M_{2,0}$	$M_{2,1}$	$M_{2,2}$	$M_{2,3}$
$M_{3,0}$	$M_{3,1}$	$M_{3,2}$	$M_{3,3}$



阶段 1：计算块(0,0) (第1次迭代)

$N_{0,0}$	$N_{0,1}$	$N_{0,2}$	$N_{0,3}$
$N_{1,0}$	$N_{1,1}$	$N_{1,2}$	$N_{1,3}$
$N_{2,0}$	$N_{2,1}$	$N_{2,2}$	$N_{2,3}$
$N_{3,0}$	$N_{3,1}$	$N_{3,2}$	$N_{3,3}$

$M_{0,0}$	$M_{0,1}$	$M_{0,2}$	$M_{0,3}$
$M_{1,0}$	$M_{1,1}$	$M_{1,2}$	$M_{1,3}$
$M_{2,0}$	$M_{2,1}$	$M_{2,2}$	$M_{2,3}$
$M_{3,0}$	$M_{3,1}$	$M_{3,2}$	$M_{3,3}$



示例：执行阶段

	Phase 0			Phase 1		
thread _{0,0}	M_{0,0} ↓ Mds _{0,0}	N_{0,0} ↓ Nds _{0,0}	PValue _{0,0} += Mds _{0,0} *Nds _{0,0} + Mds _{0,1} *Nds _{1,0}	M_{0,2} ↓ Mds _{0,0}	N_{2,0} ↓ Nds _{0,0}	PValue _{0,0} += Mds _{0,0} *Nds _{0,0} + Mds _{0,1} *Nds _{1,0}
thread _{0,1}	M_{0,1} ↓ Mds _{0,1}	N_{0,1} ↓ Nds _{1,0}	PValue _{0,1} += Mds _{0,0} *Nds _{0,1} + Mds _{0,1} *Nds _{1,1}	M_{0,3} ↓ Mds _{0,1}	N_{2,1} ↓ Nds _{0,1}	PValue _{0,1} += Mds _{0,0} *Nds _{0,1} + Mds _{0,1} *Nds _{1,1}
thread _{1,0}	M_{1,0} ↓ Mds _{1,0}	N_{1,0} ↓ Nds _{1,0}	PValue _{1,0} += Mds _{1,0} *Nds _{0,0} + Mds _{1,1} *Nds _{1,0}	M_{1,2} ↓ Mds _{1,0}	N_{3,0} ↓ Nds _{1,0}	PValue _{1,0} += Mds _{1,0} *Nds _{0,0} + Mds _{1,1} *Nds _{1,0}
thread _{1,1}	M_{1,1} ↓ Mds _{1,1}	N_{1,1} ↓ Nds _{1,1}	PValue _{1,1} += Mds _{1,0} *Nds _{0,1} + Mds _{1,1} *Nds _{1,1}	M_{1,3} ↓ Mds _{1,1}	N_{3,1} ↓ Nds _{1,1}	PValue _{1,1} += Mds _{1,0} *Nds _{0,1} + Mds _{1,1} *Nds _{1,1}

time →



示例：执行阶段

	Phase 0			Phase 1		
thread _{0,0}	M_{0,0} ↓ Mds _{0,0}	N_{0,0} ↓ Nds _{0,0}	PValue _{0,0} += Mds_{0,0} *Nds _{0,0} + Mds _{0,1} *Nds _{1,0}	M_{0,2} ↓ Mds _{0,0}	N_{2,0} ↓ Nds _{0,0}	PValue _{0,0} += Mds _{0,0} *Nds _{0,0} + Mds _{0,1} *Nds _{1,0}
thread _{0,1}	M_{0,1} ↓ Mds _{0,1}	N_{0,1} ↓ Nds _{1,0}	PValue _{0,1} += Mds_{0,0} *Nds _{0,1} + Mds _{0,1} *Nds _{1,1}	M_{0,3} ↓ Mds _{0,1}	N_{2,1} ↓ Nds _{0,1}	PValue _{0,1} += Mds _{0,0} *Nds _{0,1} + Mds _{0,1} *Nds _{1,1}
thread _{1,0}	M_{1,0} ↓ Mds _{1,0}	N_{1,0} ↓ Nds _{1,0}	PValue _{1,0} += Mds _{1,0} *Nds _{0,0} + Mds _{1,1} *Nds _{1,0}	M_{1,2} ↓ Mds _{1,0}	N_{3,0} ↓ Nds _{1,0}	PValue _{1,0} += Mds _{1,0} *Nds _{0,0} + Mds _{1,1} *Nds _{1,0}
thread _{1,1}	M_{1,1} ↓ Mds _{1,1}	N_{1,1} ↓ Nds _{1,1}	PValue _{1,1} += Mds _{1,0} *Nds _{0,1} + Mds _{1,1} *Nds _{1,1}	M_{1,3} ↓ Mds _{1,1}	N_{3,1} ↓ Nds _{1,1}	PValue _{1,1} += Mds _{1,0} *Nds _{0,1} + Mds _{1,1} *Nds _{1,1}

time



目录

- 内存访问效率 (Memory Access Efficiency)
- 分块矩阵乘法 (Tiled Matrix Multiplication)
- 分块矩阵乘法内核 (Tiled Matrix Multiplication Kernel)
- **处理分块中的边界条件 (Boundary Conditions)**
- 任意矩阵维度的分块内核 (Tiled Kernel)



小节目标

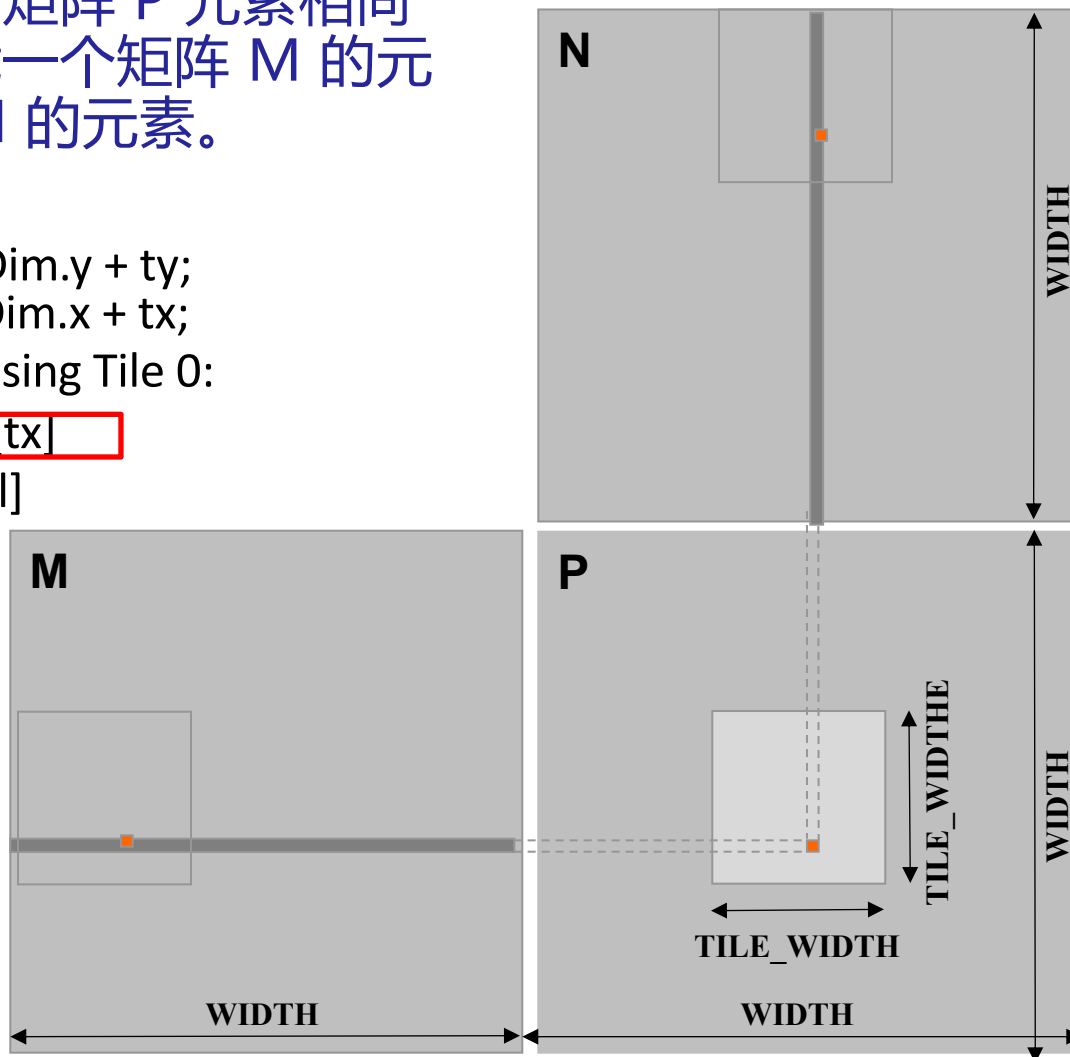
- 学习编写分块矩阵乘法内核
 - 加载和使用分片进行矩阵乘法
 - 屏障同步，共享内存
 - 资源安排
 - 为简单起见，假设矩阵的维度是分片大小的倍数

载入矩阵M的分片0(阶段 0)

- 让每个线程在与矩阵 P 元素相同的相对位置加载一个矩阵 M 的元素和一个矩阵 N 的元素。

```
int Row = by * blockDim.y + ty;  
int Col = bx * blockDim.x + tx;  
2D indexing for accessing Tile 0:
```

`M[Row][tx]`
`N[ty][Col]`



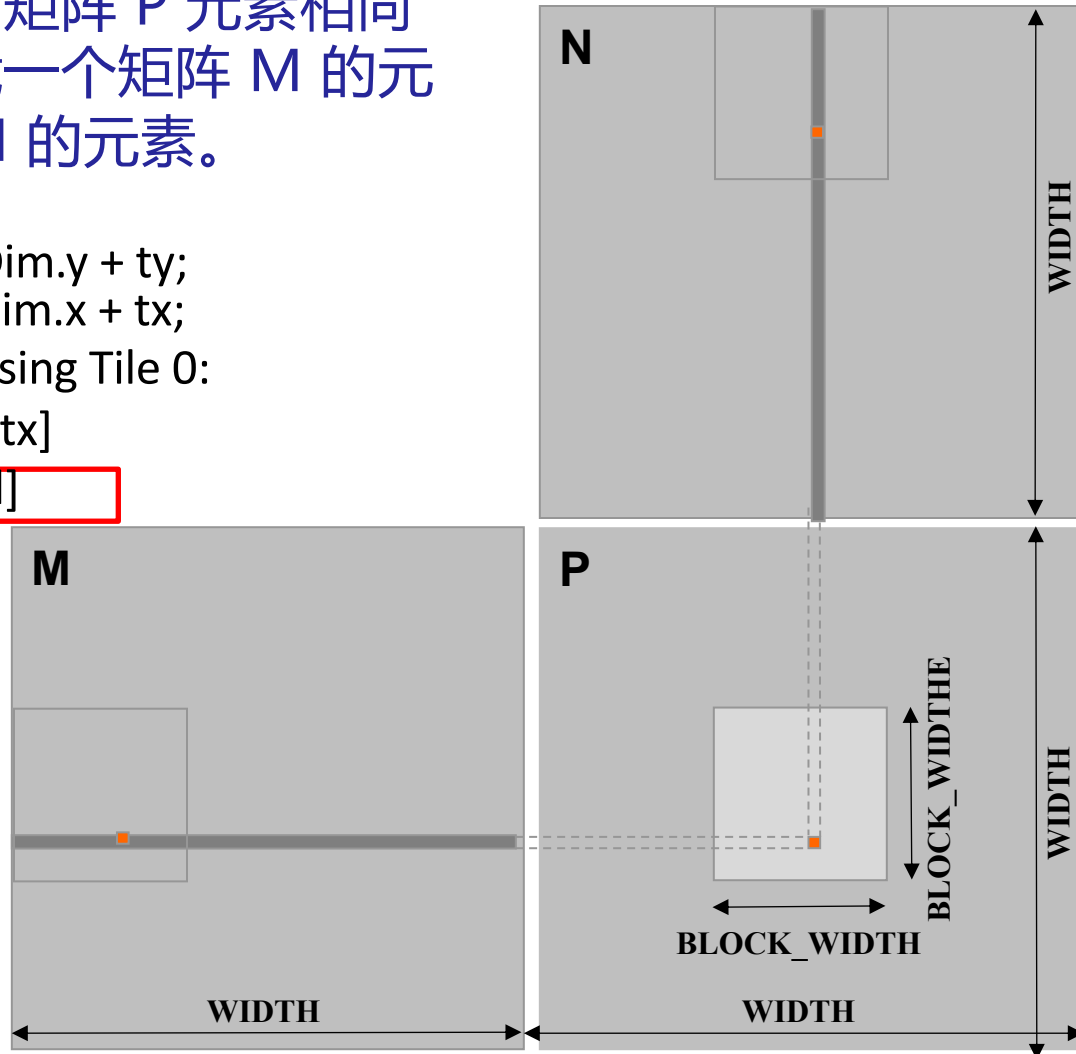
载入矩阵N的分片0(阶段 0)

- 让每个线程在与矩阵 P 元素相同的相对位置加载一个矩阵 M 的元素和一个矩阵 N 的元素。

```
int Row = by * blockDim.y + ty;  
int Col = bx * blockDim.x + tx;  
2D indexing for accessing Tile 0:
```

M[Row][tx]

N[ty][Col]

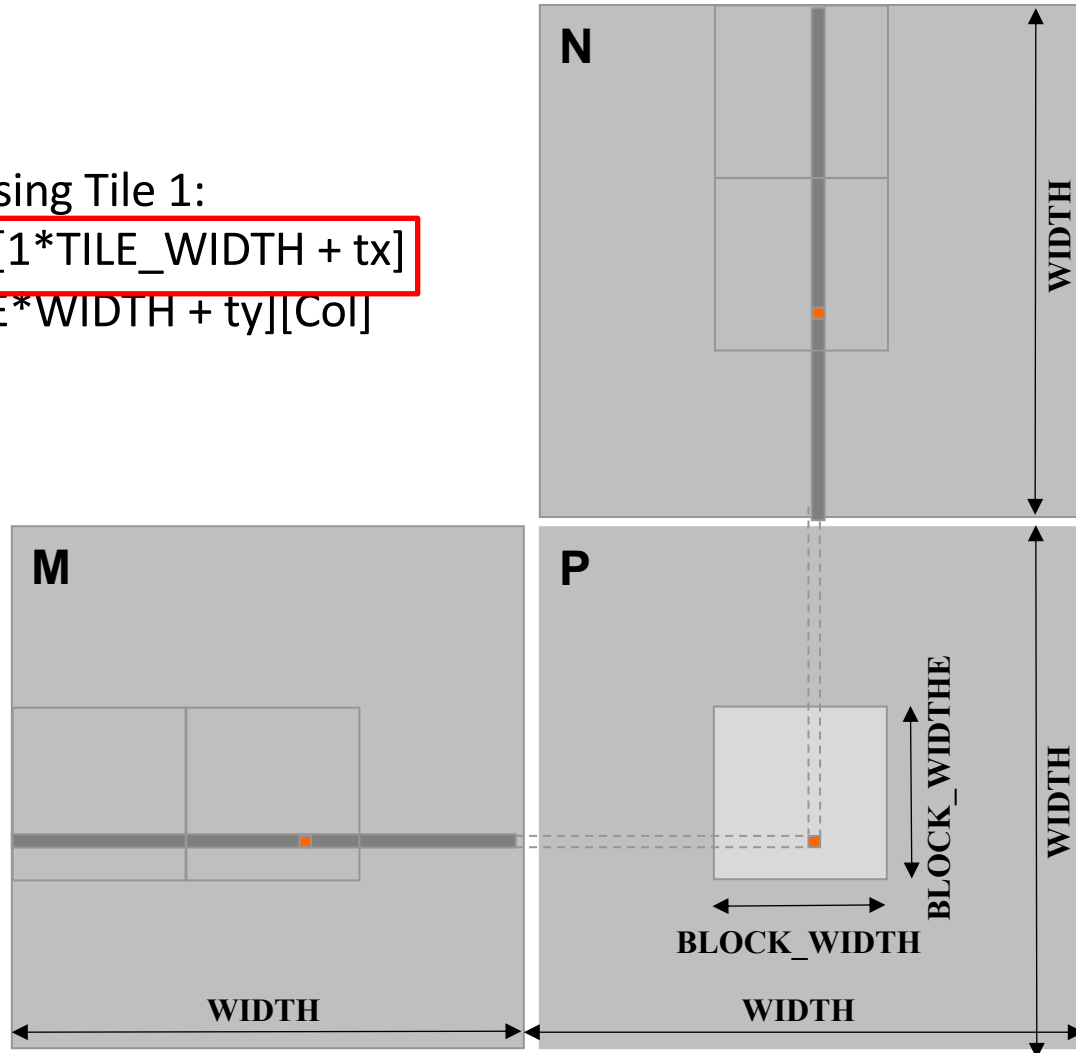


载入矩阵M的分片1(阶段 1)

2D indexing for accessing Tile 1:

$M[\text{Row}][1 * \text{TILE_WIDTH} + \text{tx}]$

$N[1 * \text{TILE} * \text{WIDTH} + \text{ty}][\text{Col}]$

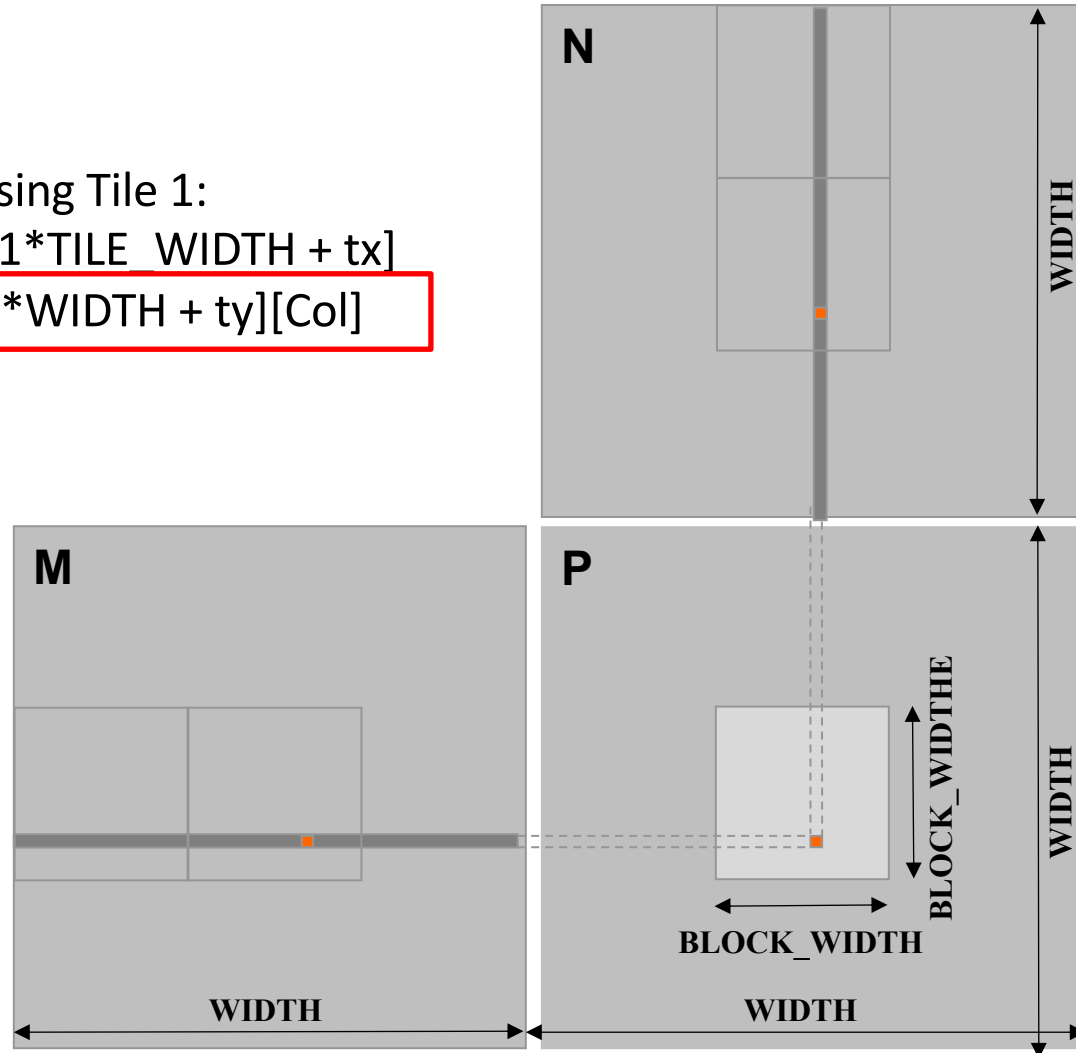


载入矩阵N的分片1(阶段 1)

2D indexing for accessing Tile 1:

$M[\text{Row}][1 * \text{TILE_WIDTH} + \text{tx}]$

$N[1 * \text{TILE} * \text{WIDTH} + \text{ty}][\text{Col}]$



分块矩阵乘法内核

```
__global__ void MatrixMulKernel(float* M, float* N, float* P, Int Width)
{
    __shared__ float ds_M[TILE_WIDTH][TILE_WIDTH];
    __shared__ float ds_N[TILE_WIDTH][TILE_WIDTH];

    int bx = blockIdx.x;  int by = blockIdx.y;
    int tx = threadIdx.x; int ty = threadIdx.y;

    int Row = by * blockDim.y + ty;
    int Col = bx * blockDim.x + tx;
    float Pvalue = 0;

    // Loop over the M and N tiles required to compute the P element
    for (int p = 0; p < WIDTH/TILE_WIDTH; ++p) {
        // Collaborative loading of M and N tiles into shared memory
        ds_M[ty][tx] = M[Row*Width + p*TILE_WIDTH+tx];
        ds_N[ty][tx] = N[(p*TILE_WIDTH+ty)*Width + Col];
        __syncthreads();

        for (int i = 0; i < TILE_WIDTH; ++i) Pvalue += ds_M[ty][i] * ds_N[i][tx];
        __syncthreads();
    }
    P[Row*Width+Col] = Pvalue;
}
```



分块矩阵乘法内核

```
__global__ void MatrixMulKernel(float* M, float* N, float* P, Int Width)
{
    __shared__ float ds_M[TILE_WIDTH][TILE_WIDTH];
    __shared__ float ds_N[TILE_WIDTH][TILE_WIDTH];

    int bx = blockIdx.x;  int by = blockIdx.y;
    int tx = threadIdx.x; int ty = threadIdx.y;

    int Row = by * blockDim.y + ty;
    int Col = bx * blockDim.x + tx;
    float Pvalue = 0;

    // Loop over the M and N tiles required to compute the P element
    for (int p = 0; p < WIDTH/TILE_WIDTH; ++p) {
        // Collaborative loading of M and N tiles into shared memory
        ds_M[ty][tx] = M[Row*Width + p*TILE_WIDTH+tx];
        ds_N[ty][tx] = N[(p*TILE_WIDTH+ty)*Width + Col];
        __syncthreads();

        for (int i = 0; i < TILE_WIDTH; ++i) Pvalue += ds_M[ty][i] * ds_N[i][tx];
        __syncthreads();
    }
    P[Row*Width+Col] = Pvalue;
}
```



分块矩阵乘法内核

```
__global__ void MatrixMulKernel(float* M, float* N, float* P, Int Width)
{
    __shared__ float ds_M[TILE_WIDTH][TILE_WIDTH];
    __shared__ float ds_N[TILE_WIDTH][TILE_WIDTH];

    int bx = blockIdx.x;  int by = blockIdx.y;
    int tx = threadIdx.x; int ty = threadIdx.y;

    int Row = by * blockDim.y + ty;
    int Col = bx * blockDim.x + tx;
    float Pvalue = 0;

    // Loop over the M and N tiles required to compute the P element
    for (int p = 0; p < WIDTH/TILE_WIDTH; ++p) {
        // Collaborative loading of M and N tiles into shared memory
        ds_M[ty][tx] = M[Row*Width + p*TILE_WIDTH+tx];
        ds_N[ty][tx] = N[(p*TILE_WIDTH+ty)*Width + Col];
        __syncthreads();

        for (int i = 0; i < TILE_WIDTH; ++i) Pvalue += ds_M[ty][i] * ds_N[i][tx];
        __syncthreads();
    }
    P[Row*Width+Col] = Pvalue;
}
```



课堂测试：

```
// Loop over the M and N tiles required to compute the P element
for (int p = 0; p < WIDTH/TILE_WIDTH; ++p) {
    // Collaborative loading of M and N tiles into shared memory
    ds_M[ty][tx] = M[Row*Width + p*TILE_WIDTH+tx];
    ds_N[ty][tx] = N[(p*TILE_WIDTH+ty)*Width + Col];
    __syncthreads();

    for (int i = 0; i < TILE_WIDTH; ++i) Pvalue += ds_M[ty][i] * ds_N[i][tx];
    __syncthreads();
}
P[Row*Width+Col] = Pvalue;
}
```

请给出第一个和第二个__syncthreads()函数的作用。



分片(线程块)大小注意事项

- 每个**线程块**应该包含很多线程
 - TILE_WIDTH 为 16 的线程块包含 $16 \times 16 = 256$ 个线程
 - TILE_WIDTH 为 32 的线程块包含 $32 \times 32 = 1024$ 个线程
- 当线程块宽度为16时，在每个阶段，每个线程块从全局内存中进行 $2 \times 256 = 512$ 次浮点载入，用于 $256 * (2 \times 16) = 8,192$ 次乘法/加法运算。
- 计算/通信比 CGMA=?
- 当线程块宽度为32时，在每个阶段，每个线程块从全局内存中进行 $2 \times 1024 = 2048$ 次浮点载入，用于 $1024 * (2 \times 32) = 65,536$ 次乘法/加法运算。（每次内存载入用于 32 次浮点操作）
- 计算/通信比 CGMA=?



共享内存和多线程

- 每个 SM 都有 16KB 的共享内存
 - 共享内存大小取决于实现方案
 - 当 $TILE_WIDTH = 16$ 时
 - 每个线程块使用的共享内存大小为 $2*256*4B = 2KB$
 - 对于 16KB 的共享内存，可以满足最多8个线程块执行
 - 当 $TILE_WIDTH = 32$ 时
 - 每个线程块使用的共享内存大小为 $2*32*32*4\text{ Byte} = 8K\text{ Byte}$ ，允许 2 个线程块同时处于执行状态
 - 但是，目前的GPU 中每个 SM 的1536 个线程。SM 的线程数限制将使每个 SM 的线程块数减少到 1 个！
- 每次 `__syncthread()` 能减少线程块中激活线程的数量
 - 更多的线程块可能是有利的



目录

- 内存访问效率 (Memory Access Efficiency)
- 分块矩阵乘法 (Tiled Matrix Multiplication)
- 分块矩阵乘法内核 (Tiled Matrix Multiplication Kernel)
- 处理分块中的边界条件 (Boundary Conditions)
- **任意矩阵维度的分块内核 (Tiled Kernel)**



小节目标

- 学习面向任意大小矩阵的分块矩阵乘法
 - 边界条件检查
 - 正则化分片内容
 - 矩形矩阵

处理任意大小的矩阵

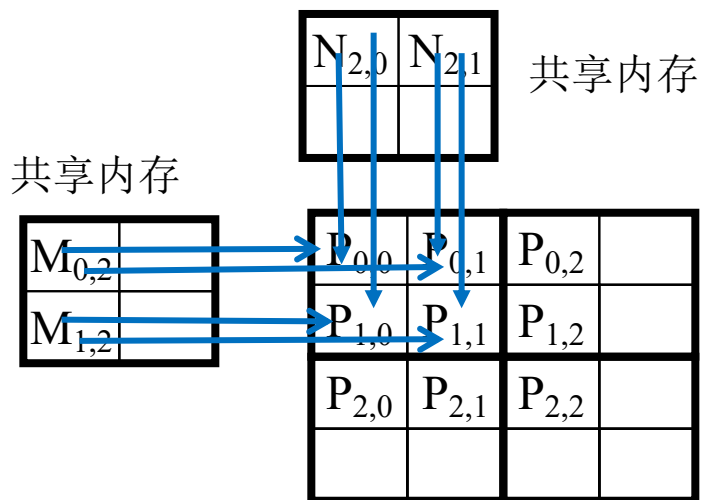
- 到目前为止，提出的分块矩阵乘法内核只能处理尺寸（Width）是分片宽度（TILE_WIDTH）倍数的方阵
 - 然而，实际应用程序需要处理任意大小的矩阵。
 - 可以将行和列填充（添加元素）成瓦片大小的倍数，但会产生大量的空间和数据传输时间开销。
- 我们将采取一种不一样的方法。



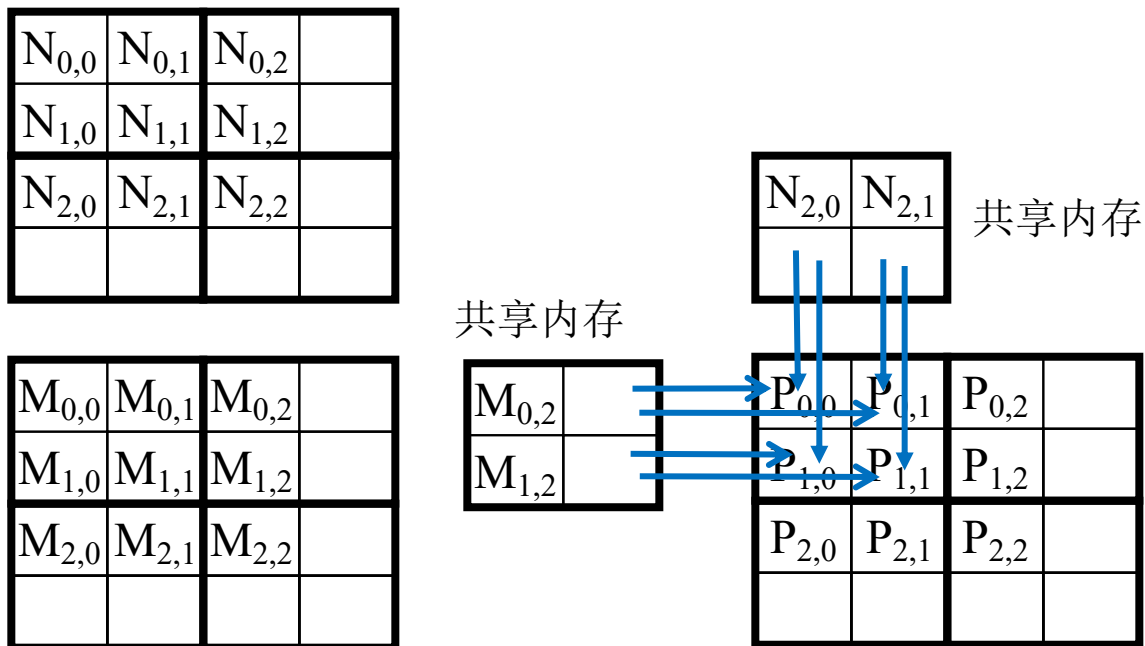
阶段 1：计算块(0,0)(第0次迭代)

$N_{0,0}$	$N_{0,1}$	$N_{0,2}$	
$N_{1,0}$	$N_{1,1}$	$N_{1,2}$	
$N_{2,0}$	$N_{2,1}$	$N_{2,2}$	

$M_{0,0}$	$M_{0,1}$	$M_{0,2}$	
$M_{1,0}$	$M_{1,1}$	$M_{1,2}$	
$M_{2,0}$	$M_{2,1}$	$M_{2,2}$	



阶段 1：计算块(0,0)(第1次迭代)

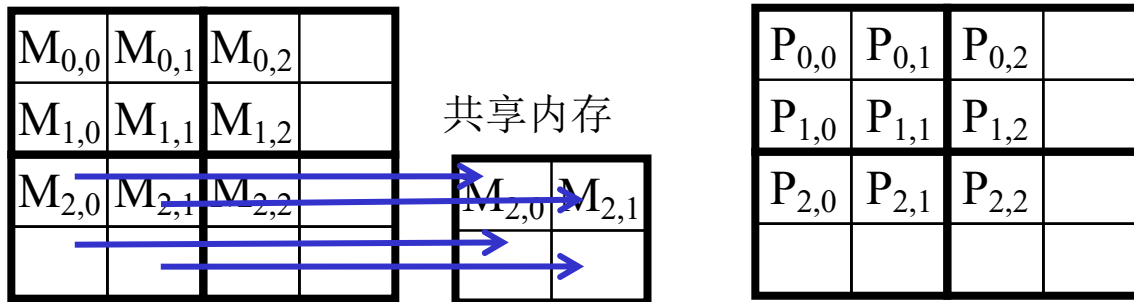
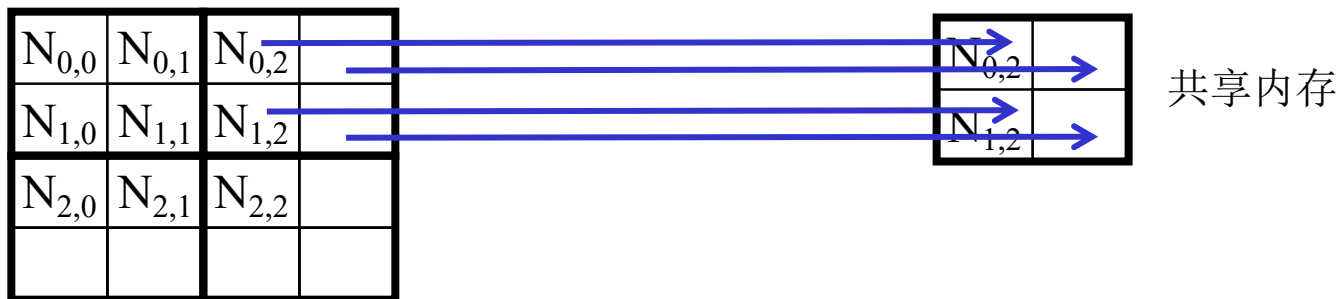


- 所有线程都需要特殊处理。
- 他们都不应该在他们的 P 元素中引入无效的贡献。



阶段0：载入块(1,1) (以3*3矩阵为例)

在加载矩阵N分片时，线程(0,1)和(1,1)需要特殊处理。

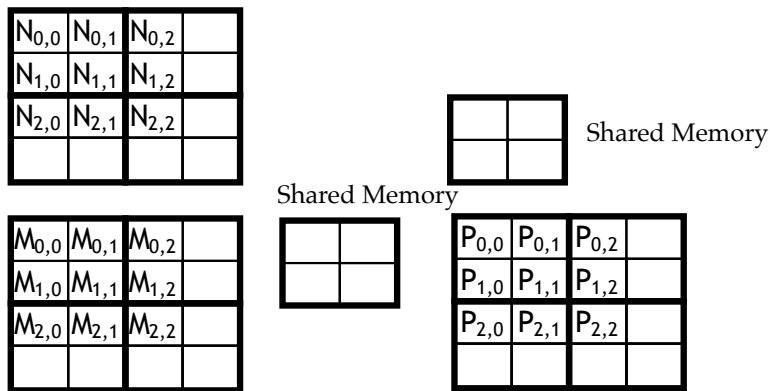


在加载矩阵M分片时，线程(1,0)和(1,1)需要特殊处理。



主要案例

- 不计算有效 P 元素但仍需要参与加载输入分片的线程
 - 阶段0中, Block(1,1), Thread(1,0)被分配去计算不存在的P[3,2]但需要参与加载分片元素N[1,2]
- 计算有效 P 元素的线程可能会在加载输入分片时尝试加载不存在的输入元素
 - 阶段1中, Block(0,0), Thread(1,0)被分配去计算有效值 P[1,0]但尝试去加载不存在的N[3,0]



一种“简单”的解决办法

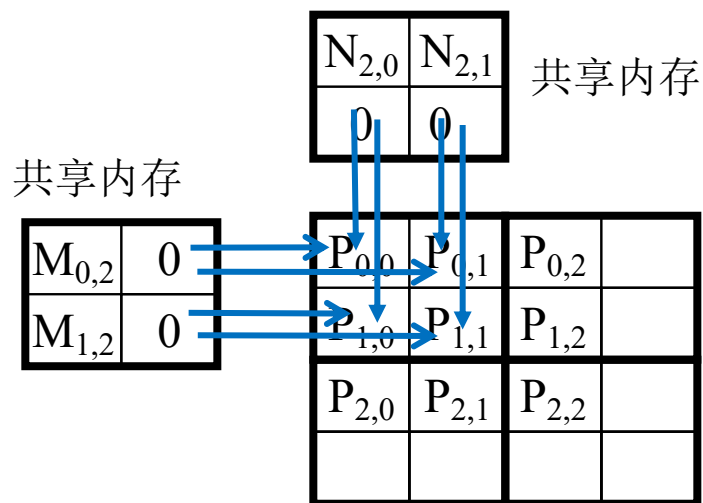
- 当一个线程要加载任何输入元素时，测试它是否在有效的索引范围内
 - 如果在，则继续加载操作
 - 如果不在，取消加载操作，写入元素0
- 基本原理：值 0 将确保连续的加载操作不会影响输出元素的最终值
- 加载输入元素的测试条件与计算输出P元素的测试条件不同
 - 不计算有效 P 元素的线程仍然可以参与输入分片元素的加载过程



阶段1：计算块(0,0)(第1次迭代)

$N_{0,0}$	$N_{0,1}$	$N_{0,2}$	
$N_{1,0}$	$N_{1,1}$	$N_{1,2}$	
$N_{2,0}$	$N_{2,1}$	$N_{2,2}$	

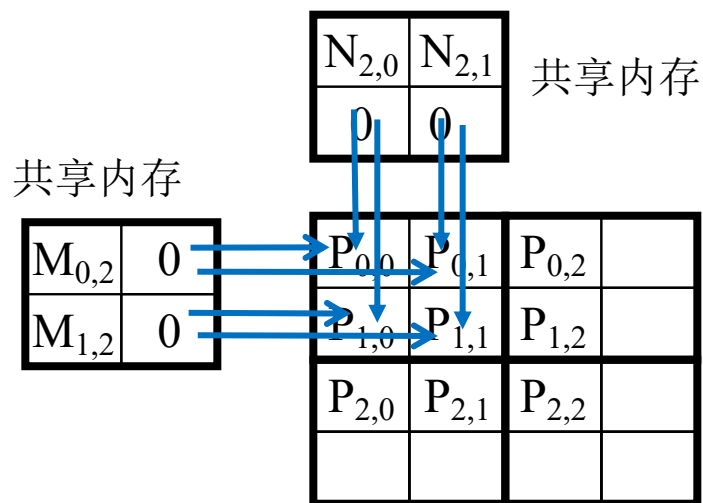
$M_{0,0}$	$M_{0,1}$	$M_{0,2}$	
$M_{1,0}$	$M_{1,1}$	$M_{1,2}$	
$M_{2,0}$	$M_{2,1}$	$M_{2,2}$	



阶段1：计算块(0,0)(第1次迭代)

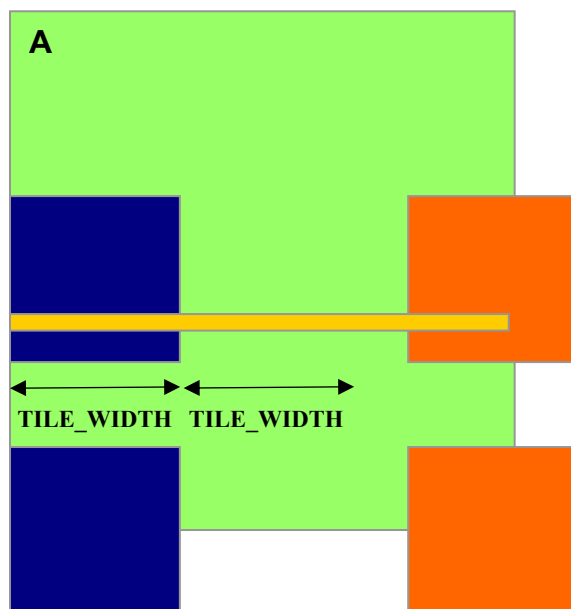
$N_{0,0}$	$N_{0,1}$	$N_{0,2}$	
$N_{1,0}$	$N_{1,1}$	$N_{1,2}$	
$N_{2,0}$	$N_{2,1}$	$N_{2,2}$	

$M_{0,0}$	$M_{0,1}$	$M_{0,2}$	
$M_{1,0}$	$M_{1,1}$	$M_{1,2}$	
$M_{2,0}$	$M_{2,1}$	$M_{2,2}$	



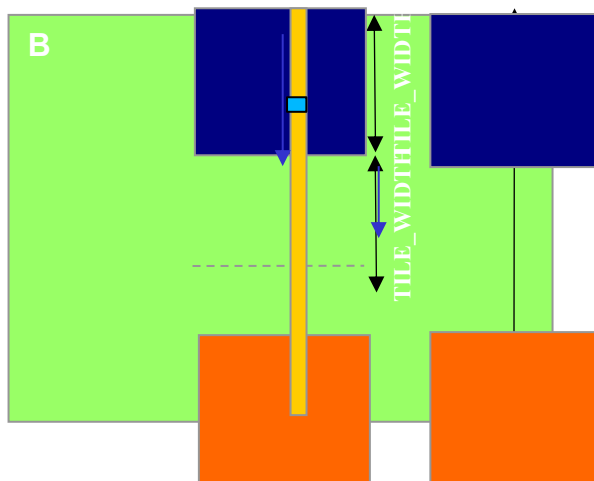
输入矩阵 M 分片的边界条件

- 每个线程载入
 - $M[\text{Row}][p \cdot \text{TILE_WIDTH} + tx]$
 - $M[\text{Row} \cdot \text{Width} + p \cdot \text{TILE_WIDTH} + tx]$
- 需要去检测
 - $(\text{Row} < \text{Width}) \ \&\& \ (p \cdot \text{TILE_WIDTH} + tx < \text{Width})$
 - 如果为真，载入矩阵 M 元素
 - 如果为假，载入值 0



输入矩阵 N 分片的边界条件

- 每个线程载入
 - $N[p * \text{TILE_WIDTH} + ty][\text{Col}]$
 - $N[(p * \text{TILE_WIDTH} + ty) * \text{Width} + \text{Col}]$
- 需要去检测
 - $(p * \text{TILE_WIDTH} + ty < \text{Width}) \ \&\& \ (\text{Col} < \text{Width})$
 - 如果为真，载入矩阵 N 元素
 - 如果为假，载入值 0



带边界检查的元素加载过程

```
• 8   for (int p = 0; p < (Width-1) / TILE_WIDTH + 1; ++p) {  
•  
•   ++   if (Row < Width && t * TILE_WIDTH + tx < Width) {  
• 9       ds_M[ty][tx] = M[Row * Width + p * TILE_WIDTH + tx];  
•   ++   } else {  
•   ++       ds_M[ty][tx] = 0.0;  
•   ++   }  
•   ++   if (p * TILE_WIDTH + ty < Width && Col < Width) {  
• 10       ds_N[ty][tx] = N[(p * TILE_WIDTH + ty) * Width + Col];  
•   ++   } else {  
•   ++       ds_N[ty][tx] = 0.0;  
•   ++   }  
• 11   __syncthreads();  
•
```



内积操作的前与后

- ++ if(Row < Width && Col < Width) {
- 12 for (int i = 0; i < TILE_WIDTH; ++i) {
- 13 Pvalue += ds_M[ty][i] * ds_N[i][tx];
- }
- 14 __syncthreads();
- 15 } /* end of outer for loop */
- ++ if (Row < Width && Col < Width)
- 16 P[Row*Width + Col] = Pvalue;
- } /* end of kernel */



一些要点

- 对于每个线程，在以下条件中，边界条件是不同的：
 - 载入矩阵 M 的元素
 - 载入矩阵 N 的元素
 - 计算以及储存输出元素
- 对于大型矩阵，控制发散现象的影响应该很小



处理一般的矩形矩阵

- 一般来说，矩阵乘法是根据矩形矩阵定义的
 - 一个 $j * k$ 的矩阵 M 和一个 $k * l$ 的矩阵 N 相乘，得到一个 $j * l$ 的矩阵 P
- 作为一种特殊的案例，方阵的矩阵乘法算法已被提出
- 内核函数需要经过泛化以处理一般的矩形矩阵
 - 参数 $Width$ 替换为三个参数： j 、 k 、 l
 - 当 $Width$ 指代矩阵 M 的高或者矩阵 P 的高时，用参数 j 来替换
 - 当 $Width$ 指代矩阵 M 的宽或者矩阵 N 的高时，用参数 k 来替换
 - 当 $Width$ 指代矩阵 N 的宽或者矩阵 P 的宽时，用参数 l 来替换



ANY QUESTIONS?

