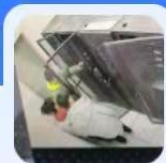


第四讲 – 性能问题





2023春GPU并行编程

群号：658334249



扫一扫二维码，加入群聊。



电子科技大学
University of Electronic Science and Technology of China

目录

- **线程束 (Warps) 与单指令多数据流指令集 (SIMD)**
- 控制分支现象 (Control Divergence) 的性能影响
- 并行规约 (Parallel Reduction)
- 内存并行性 (Memory Parallelism)



小节目标

- 了解 CUDA 线程如何在 SIMD 硬件上执行
 - 线程束的划分
 - SIMD 硬件
 - 控制分支现象

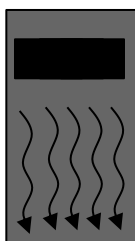


执行过程

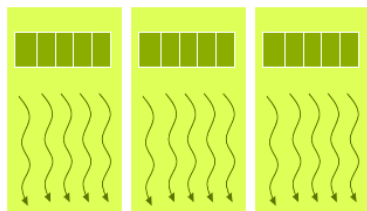
软件



线程



Thread
Block



Grid

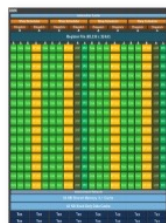


电子科技大学
University of Electronic Science and Technology of China

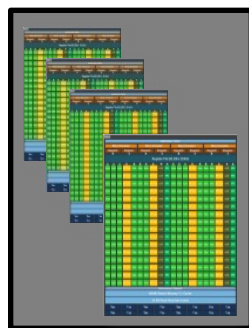
硬件



CUDA
Core



Multiprocessor



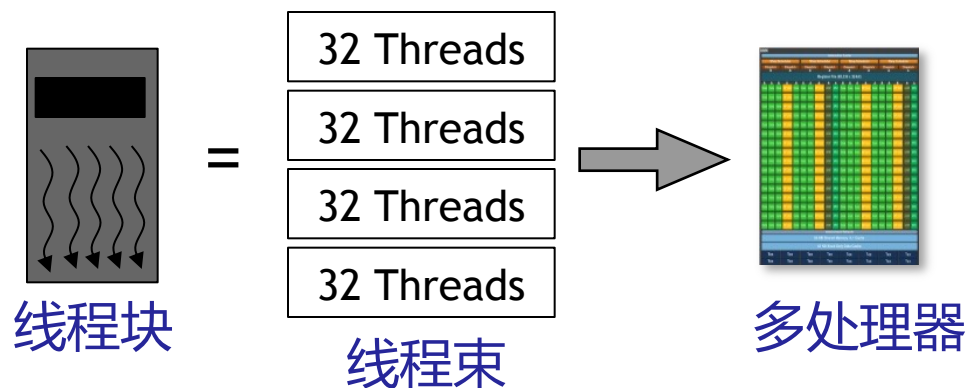
Device

CUDA Core=ALU=SP

SM=内核=逻辑架构里的CORE

当调用kernel函数时，启动起来很多线程，然后分配给硬件去执行，执行过程中要占用硬件资源。

线程束

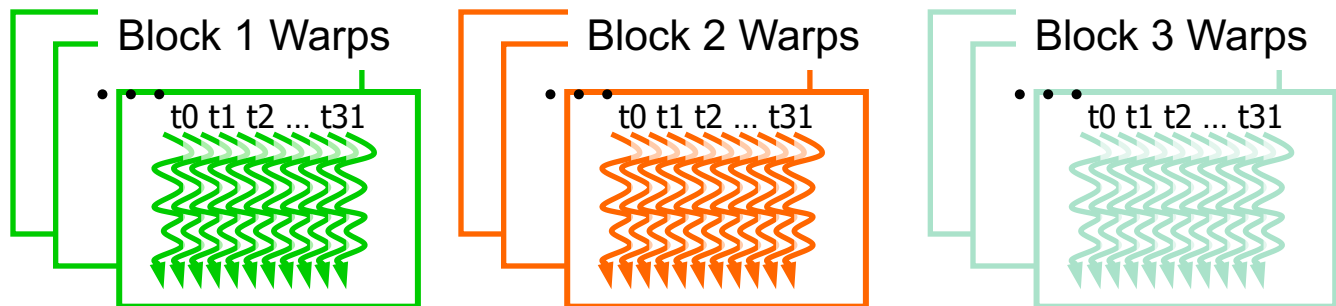


一个线程块包含若干个包含 32 线程的线程束

线程束在多处理器上以物理方式并行 (SIMD) 执行



将线程束作为调度单元

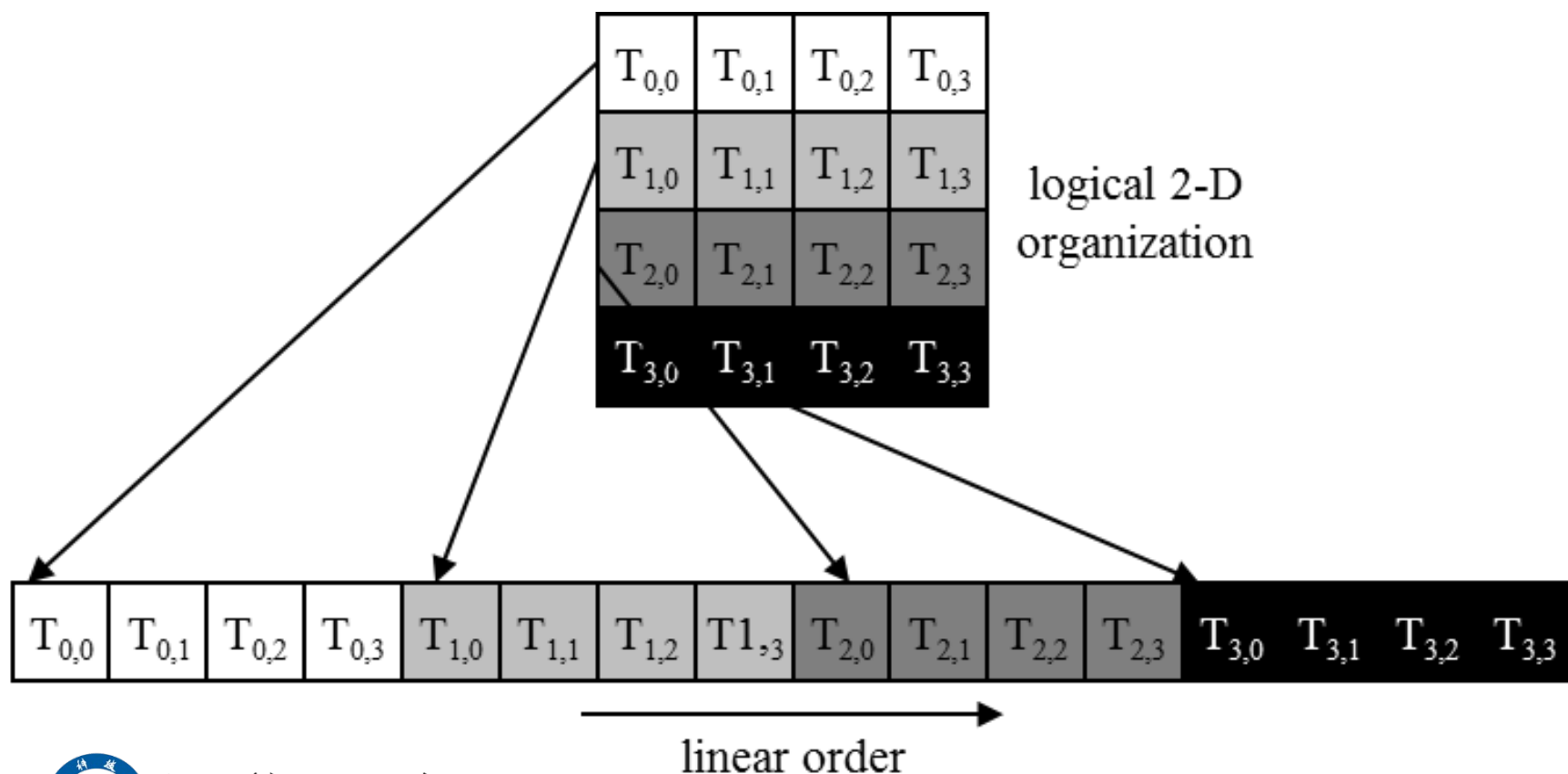


- 每个线程块拆分为包含32线程的线程束
 - 一种实现技术，不是 CUDA 编程模型的一部分
 - 线程束是 SM 中的调度单元
 - 线程束中的线程以单指令多数据 (SIMD) 方式执行
 - 线程束中的线程数量可能在未来几代中有所不同



多维线程块中的线程束

- 按行主序，线程块线性化为一维
 - 按照先 x 维，次 y 维，后 z 维的顺序



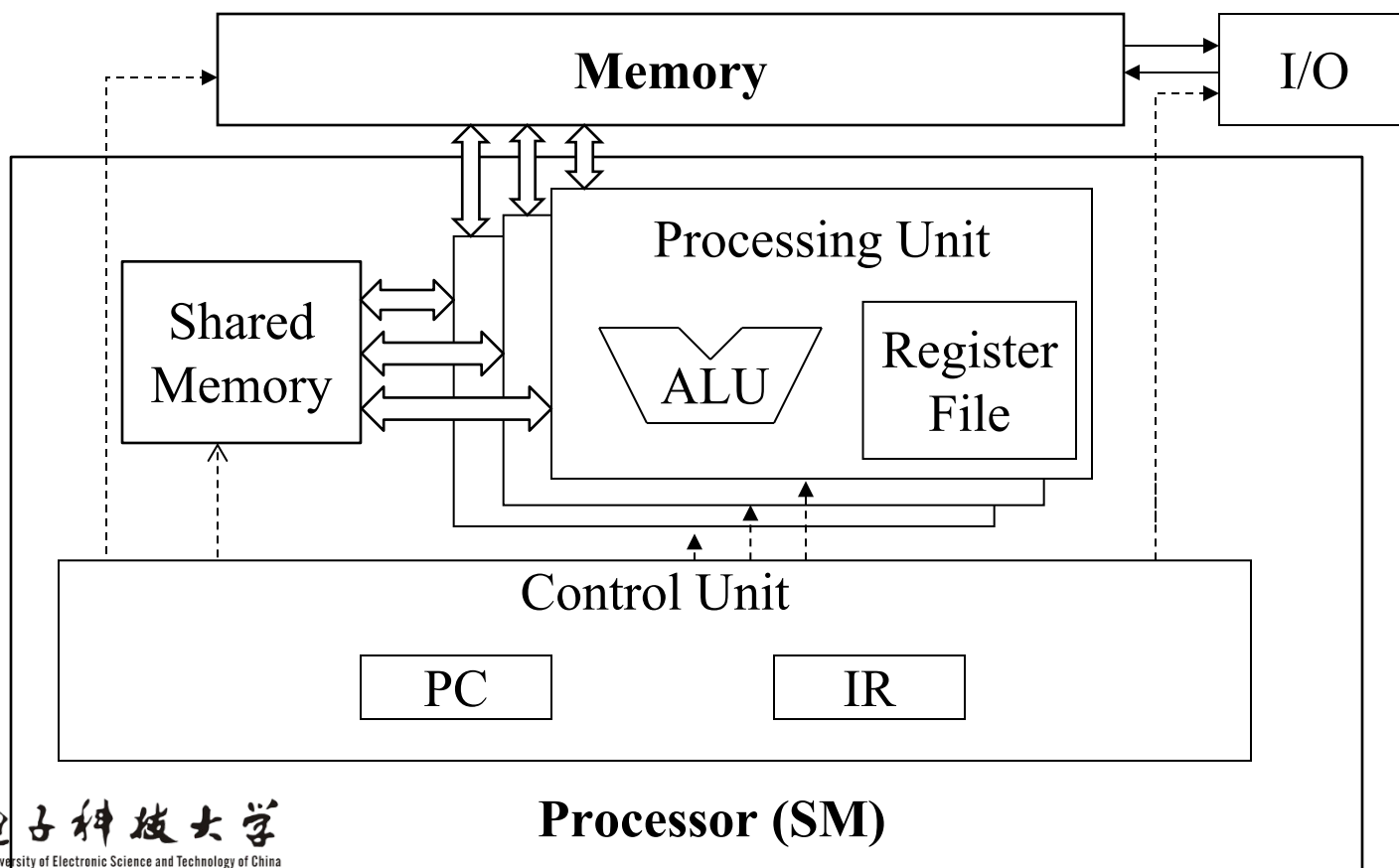
线程块在线性化后被划分

- 线性化的线程块被划分
 - 线程束中的线程索引是连续的并且增加的
 - 从线程束 0 中的线程 0 开始计数
- 划分方案在设备之间是一致的
 - 因此，可以在控制流中使用这一知识点
 - 然而，线程束的确切大小可能会因代而异
- 线程束内或线程束之间的任何顺序都不具参考性
 - 如果线程之间存在任何依赖关系，则必须执行 `__syncthreads()` 才能获得正确的结果（稍后会详细介绍）。



流式多处理器是 SIMD 处理器

- 用于指令获取、解码和控制的控制单元在多个处理单元之间共享
 - 控制开销最小化



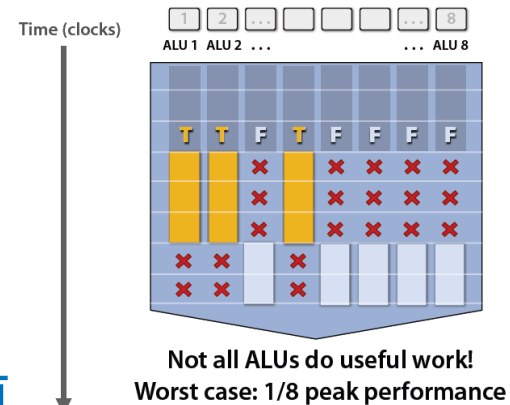
控制分支

- 当线程束中的线程通过判断分支执行不同的代码流路径时，就会发生控制分支 (Control Divergence) 现象

- 一些采用 then 路径，另一些采用 if 语句的 else 路径
- 一些线程的循环迭代次数与其他线程不同

- 采用不同路径的线程的执行在当前 GPU 中被串行化

- 线程束中的线程所采用的控制路径一次遍历一条，直到不再有新的控制路径。考虑嵌套控制流语句时，不同路径的数量可能很大。



```
<unconditional  
shader code>  
  
if (x > 0) {  
    y = pow(x, exp);  
    y *= Ks;  
    refl = y + Ka;  
} else {  
    x = 0;  
    refl = Ka;  
}  
  
<resume unconditional  
shader code>
```



例子 – 控制分支

- 当分支或循环条件是线程索引的函数时，可能会出现控制分支现象

- 示例：

```
if (threadIdx.x > 2) { }
```

```
If (blockIdx.x > 2) { }
```



示例：向量加法内核

Device Code

```
// Compute vector sum  $C = A + B$   
// Each thread performs one pair-wise addition  
  
__global__  
void vecAddKernel(float* A, float* B, float* C,  
    int n)  
{  
    int i = threadIdx.x + blockDim.x * blockIdx.x;  
    if(i < n) C[i] = A[i] + B[i];  
}
```



针对包含 1,000 个元素的向量进行分析

- 假设每个线程块的大小是 256 个线程
 - 每个线程块包含 8 个线程束
- 在线程块 0, 1, 2 中的所有线程均在有限范围内
 - i 值的范围为 0 到 767
 - 这三个线程块中有 24 个线程束, 都不会发生控制分支现象
- 线程块 3 中的大部分线程束不会发生控制分支现象
 - 线程束 0 ~ 6 中的线程均在有效范围内, 因此不会有控制分支
- 线程块 3 中的一个线程束将发生控制分支
 - 线程 992 ~ 999 均在有效范围内
 - 线程 1000 ~ 1023 在有效范围之外
- 串行化对控制分支的影响很小
 - 每 32 个线程束中会有 1 个存在控制分支现象
 - 对于性能的影响很有可能会小于 3%



目录

- 线程束 (Warps) 与单指令多数据流指令集 (SIMD)
- **控制分支现象 (Control Divergence) 的性能影响**
- 并行规约 (Parallel Reduction)
- 内存并行性 (Memory Parallelism)



小节目标

- 分析控制分支对性能的影响
 - 边界条件检查
 - 控制分支取决于数据

控制分支对性能的影响

- 边界条件检查对于并行代码的完整功能和稳健性至关重要
 - 平铺矩阵乘法内核有许多边界条件检查
 - 然而，这些检查可能会导致性能显著下降
 - 例如以下分片加载代码中：

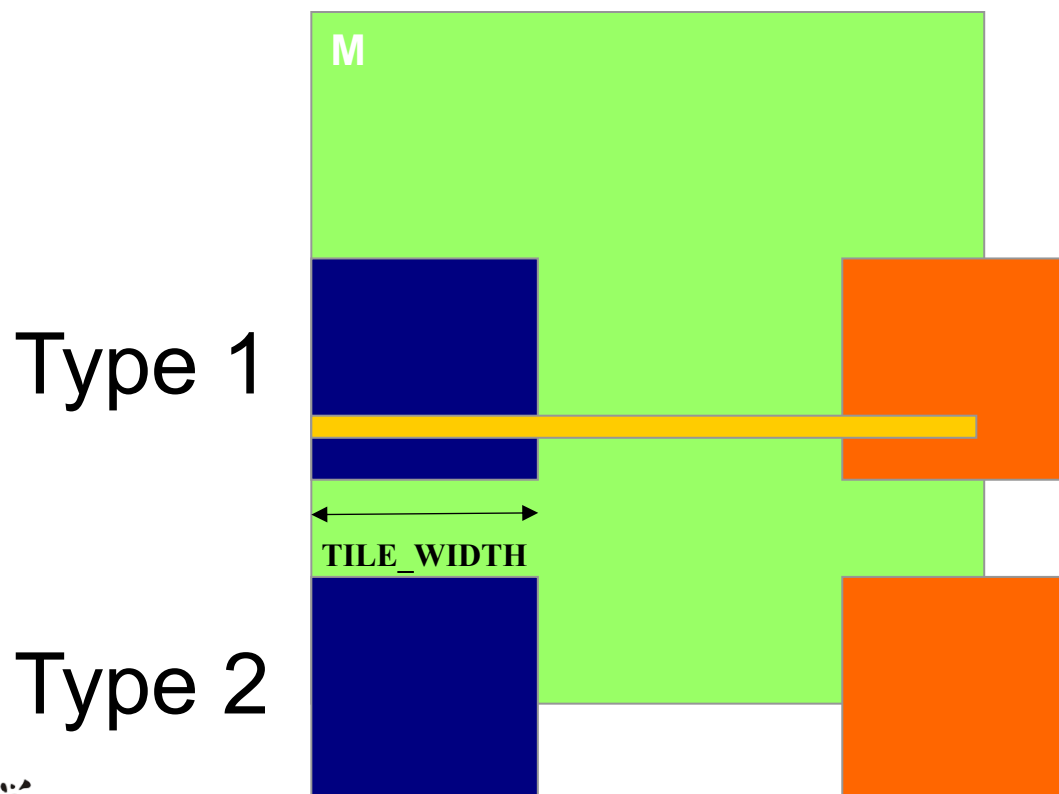
```
if(Row < Width && p * TILE_WIDTH+tx < Width) {  
    ds_M[ty][tx] = M[Row * Width + p * TILE_WIDTH + tx];  
} else {  
    ds_M[ty][tx] = 0.0;  
}
```

```
if (p*TILE_WIDTH+ty < Width && Col < Width) {  
    ds_N[ty][tx] = N[(p*TILE_WIDTH + ty) * Width + Col];  
} else {  
    ds_N[ty][tx] = 0.0;
```



载入矩阵 M 分片时的两类线程块

- Type 1. 直到最后一个阶段，其分片都在有效范围内的线程块。
- Type 2. 其分片部分在有效范围之外的线程块



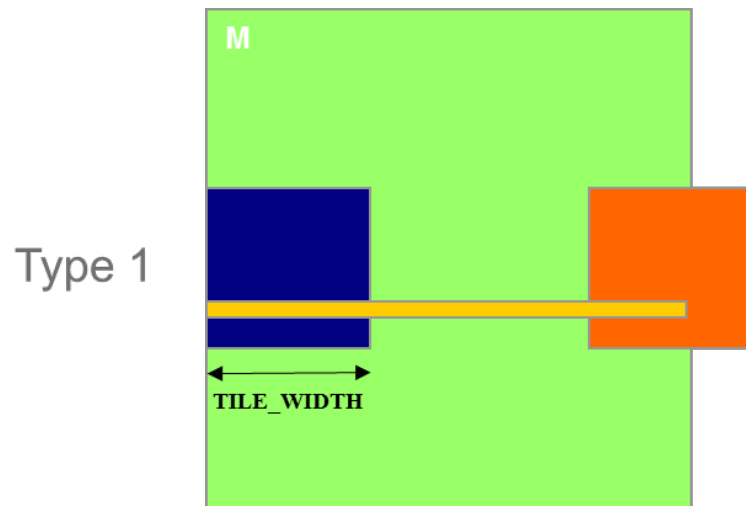
控制分支影响分析

- 假设有 **16x16** 的分片和线程块
- 每个线程块有 8 个线程束 (256/32)
- 假设有 **100x100** 的方阵
- 每个线程会历经 7 个阶段 (ceiling of 100/16)
- 有 49 个线程块 (每个维度上有 7 个)



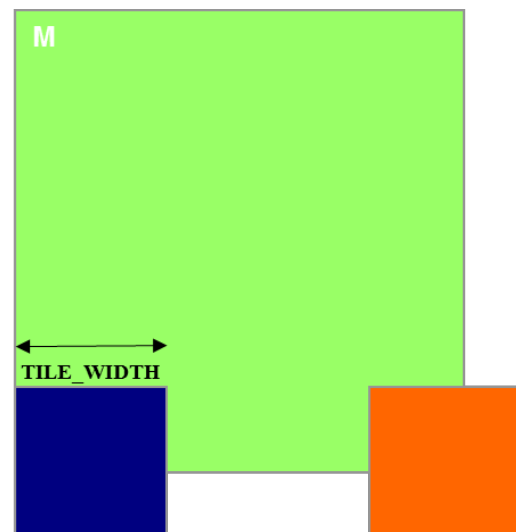
载入矩阵 M 分片时的控制分支

- 假设有 16×16 的分片和线程块
- 每个线程块有 8 个线程束 ($256/32$)
- 假设有 100×100 的方阵
- 每个线程会历经 7 个阶段 (ceiling of $100/16$)
- 有 42 (6×7) 个 Type 1 的线程块，一共包含 336 (8×42) 个线程束
- 它们都有 7 个阶段，因此有 2,352 (336×7) 个线程束-阶段 (warp-phases)
- 线程束只在它们最后一个阶段存在控制分支
- 336 个线程束-阶段存在控制分支



载入矩阵 M 分片时的控制分支 (Type 2)

- Type 2 : 被分配去加载分片底部元素的 7 个线程块, 一共 56 (8×7) 个线程束
- 它们都有 7 个阶段, 有 392 (56×7) 个线程束-阶段
- 每个 Type 2 的线程块的前 2 个线程束处于有效范围直到最后一个阶段
- 剩下的 6 个线程束处于有效范围之外
- 因此, 只有 14 (2×7) 个线程束-阶段存在控制分支

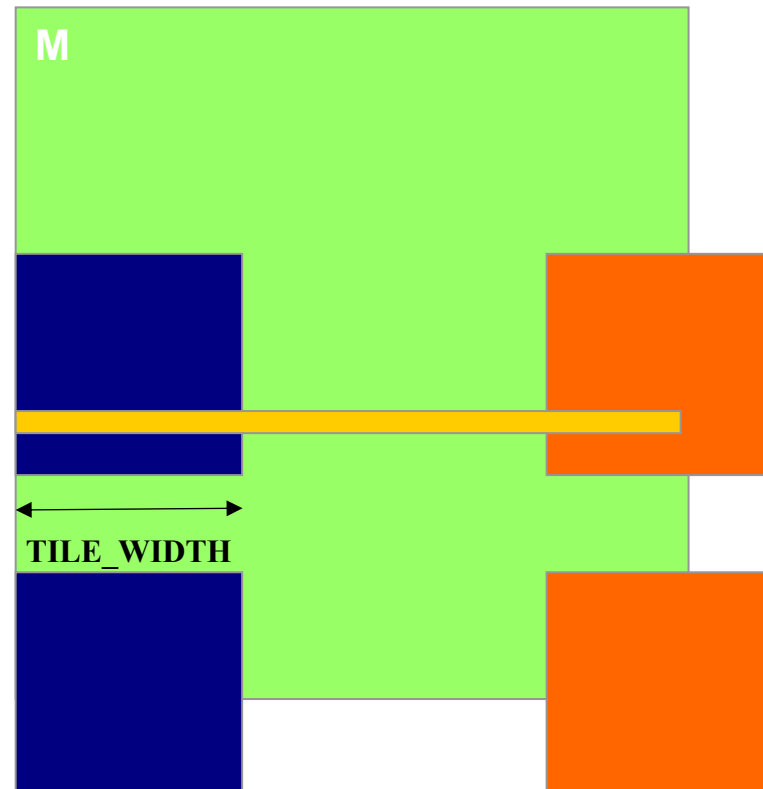


控制分歧的总体影响

- Type 1 线程块：2352 个线程束-阶段中有 336 个存在控制分支
- Type 2 线程块：392 个线程束-阶段中有 14 个存在控制分支
- 预计性能影响小于 12% ($350/2,944$ 或 $(336+14)/(2352+14)$)

Type 1

Type 2



补充

- 加载矩阵 N 的分片时控制分支的影响在计算上有些不同，留作练习
- 预计的性能影响取决于数据
 - 对于更大的矩阵，影响会明显更小
- 一般来说，控制分支对大型输入数据集的边界条件检查的影响应该是不重要的
 - 应该毫不犹豫地使用边界检查来确保完整的功能
- 内核充满控制流结构这一事实并不意味着控制分支会大量发生
- 稍后我们将介绍一些自然会导致控制分支的算法模式（例如并行归约）。



目录

- 线程束 (Warps) 与单指令多数据流指令集 (SIMD)
- 控制分支现象 (Control Divergence) 的性能影响
- **并行规约 (Parallel Reduction)**
- 内存并行性 (Memory Parallelism)



小节目标

- 学习并行归约模式
 - 一类重要的并行计算
 - 工作效率分析
 - 资源效率分析

“划分与总结”

- 处理大型输入数据集的常用策略
 - 数据集中的待处理元素没有必需的顺序（结合律和交换律）
 - 将数据集划分为更小的块
 - 让每个线程处理一个块
 - 使用归约树 (Reduction Tree) 将每个块的结果汇总为最终答案
- 例如，Google 和 Hadoop MapReduce 框架都支持这种策略
- 我们现在将专注于归约树步骤



归约计算是什么？

- 使用 “归约运算” 将一组输入值汇总为一个值
 - Max
 - Min
 - Sum
 - Product
- 常与用户定义的归约运算函数一起使用，只要该运算
 - 满足结合律和交换律
 - 具有明确定义的标识值（例如，0 表示 Sum）
 - 例如，用户可以为 3D 坐标数据集提供自定义 “max” 函数，其中每个坐标数据元组的大小是与原点的距离。



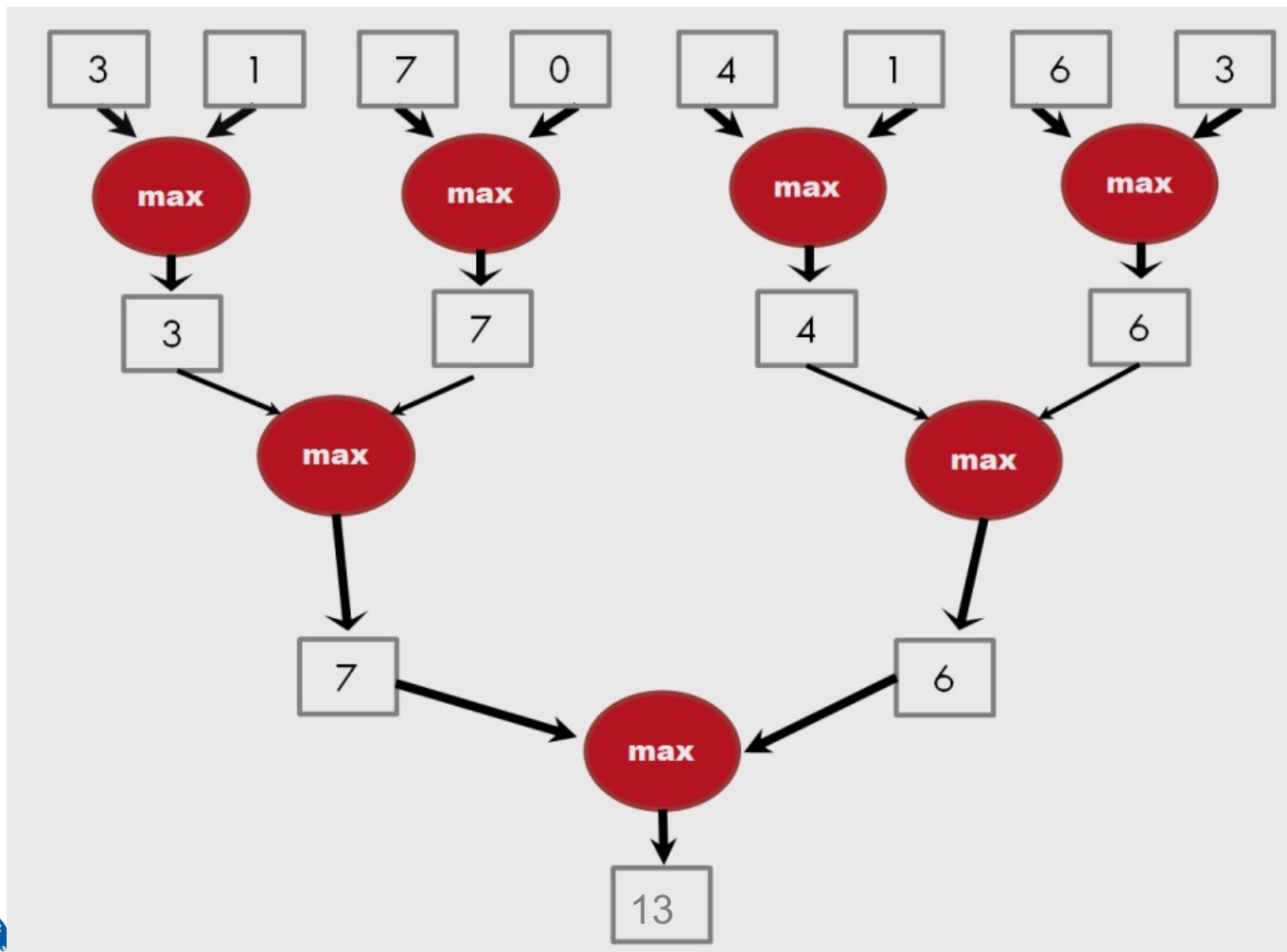
一个有效的串行归约算法 $O(N)$

- 根据归约操作，将结果**初始化**为对应的标识值
 - Max：尽可能小的值
 - Min：尽可能大的值
 - Sum：0
 - Product：1
- **遍历**输入并在结果值和当前输入值之间进行归约运算
 - 对 N 个输入值进行归约操作
 - 算法复杂度为 $O(N)$
 - 这是一种**计算效率高**的算法



并行归约树算法

以 $\log(N)$ 步执行 $N-1$ 次操作



快速分析

- 对于 N 个输入值，归约树执行
 - $(1/2)N + (1/4)N + (1/8)N + \dots (1/N)N = (1 - (1/N))N = N-1$ 个操作
 - 在 $\text{Log}(N)$ 步中 – 1,000,000 个输入值需要 20 步
 - 假设我们有足够的执行资源
 - 平均并行度 (Average Parallelism) = $(N-1)/\text{Log}(N)$
 - 当 $N = 1,000,000$ 时, 平均并行度为 50,000
 - 然而，资源需求的峰值为 500,000
 - 因此，这不是资源有效的 (resource efficient)
- 这是一种**计算高效率 (work efficient)** 的并行算法
 - 完成的工作量与高效的串行算法相当
 - 许多并行算法效率并不高



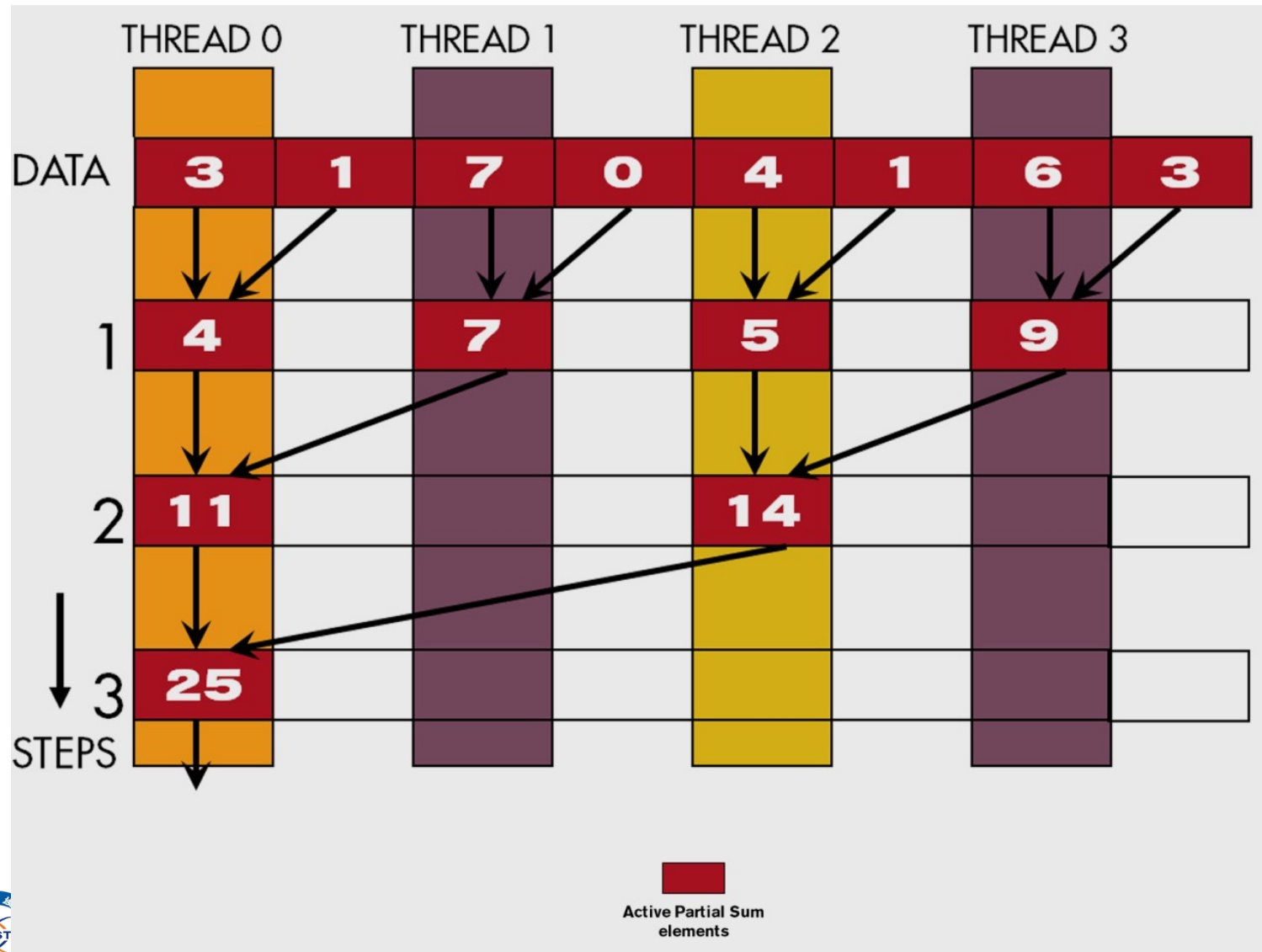
如何使用CUDA来实现呢？

并行求和归约

- 并行执行
 - 每一步中，计算线程对两个值进行求和操作
 - 每次迭代，参与计算的线程数量**减半**
 - 当有 n 个元素时，需要 $\log(n)$ 步，需要 $n/2$ 个线程
- 假设使用**共享内存**进行 **in-place** 归约
 - 原始向量在设备全局内存中
 - 共享内存用于保存部分求和向量 (partial sum vector)
 - 一开始，部分求和向量只是原始向量
 - 每一步都使部分求和向量更接近总和
 - 最终总和将在部分求和向量的元素 0 中
 - 由于读取和写入部分的部分求和值，减少了全局内存流量
 - 线程块的大小将 n 限制在小于或等于 2,048



示例：并行求和归约



用于数据映射的线程 (Naive Thread)

- 每个线程负责部分求和向量的偶数索引位置 (责任位置 location of responsibility)
- 每一步之后，不再需要一半的线程
- 输入之一始终来自责任位置
- 在每一步中，其中一个输入来自越来越远的距离



一个简单的线程块设计

- 每个线程块需要 $2 \times \text{BlockDim.x}$ 个输入元素
- 每个线程向共享内存中载入 2 个元素

```
__shared__ float partialSum[2*BLOCK_SIZE];
```

```
unsigned int t = threadIdx.x;
```

```
unsigned int start = 2*blockIdx.x*blockDim.x;
```

```
partialSum[t] = input[start + t];
```

```
partialSum[blockDim.x+t] = input[start + blockDim.x+t];
```



归约步骤

```
for (unsigned int stride = 1;
     stride <= blockDim.x;  stride *= 2)
{
    __syncthreads();
    if (t % stride == 0)
        partialSum[2*t] += partialSum[2*t+stride];
}
```

为什么我们需要 __syncthreads() ?



屏障同步

- 需要 `__syncthreads()` 以确保在进行下一步之前已经生成了每个版本的部分求和的所有元素



总体而言

- 在内核执行结束时，每个线程块中的线程 0 将 `partialSum[0]` 中的线程块之和写入以 `blockIdx.x` 为索引的向量中
- 如果原始向量非常大，可能会有大量这样的和
 - 主机代码可能会迭代并启动另一个内核
- 如果总和的数量很少，主机可以简单地将数据传回并将它们加在一起
- 或者，每个线程块的线程 0 可以使用原子操作来累积得到全局求和变量中。



目标

- 学习编写更好的归约内核
 - 提高资源效率
 - 改进的线程到数据映射
 - 减少控制分支



关于朴素归约内核的一些发现

- 在每次迭代中，每个线程束将依次遍历两个控制流路径
 - 执行加法的线程和不执行加法的线程
 - 不执行加法的线程仍然消耗执行资源
- 在第一步之后将执行一半或更少的线程
 - 第一步后禁用所有奇数索引线程
 - 在第 5 步之后，每个线程块中的整个线程束将无法通过 if 测试，资源利用率低但没有发散
 - 这可以持续一段时间，最多再增加 5 个步骤（stride = 64、128、256、512、1024），其中每个活动的线程束只有一个生产线程，直到线程块中的所有线程束都失效

```
if (t % stride == 0)
    partialSum[2*t] +=
    partialSum[2*t+stride];
```

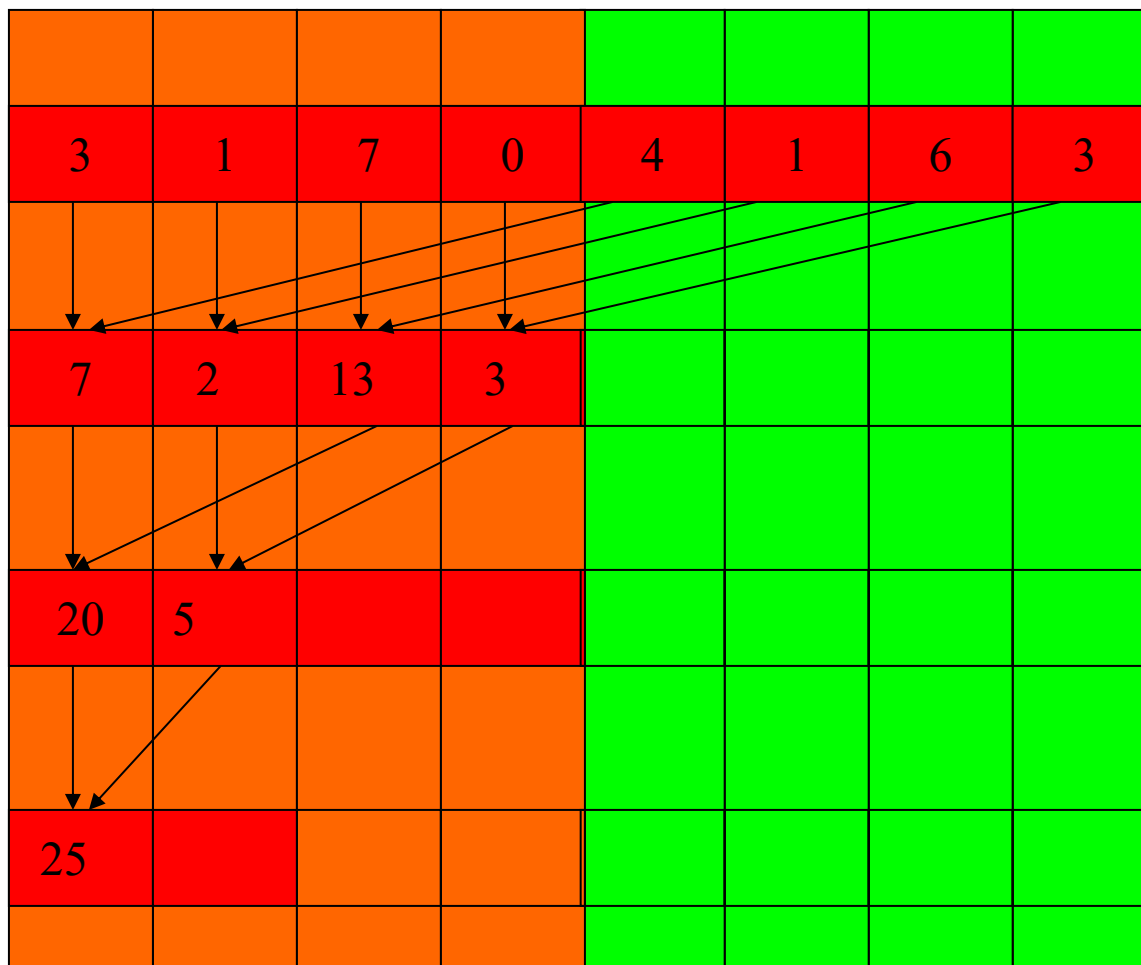


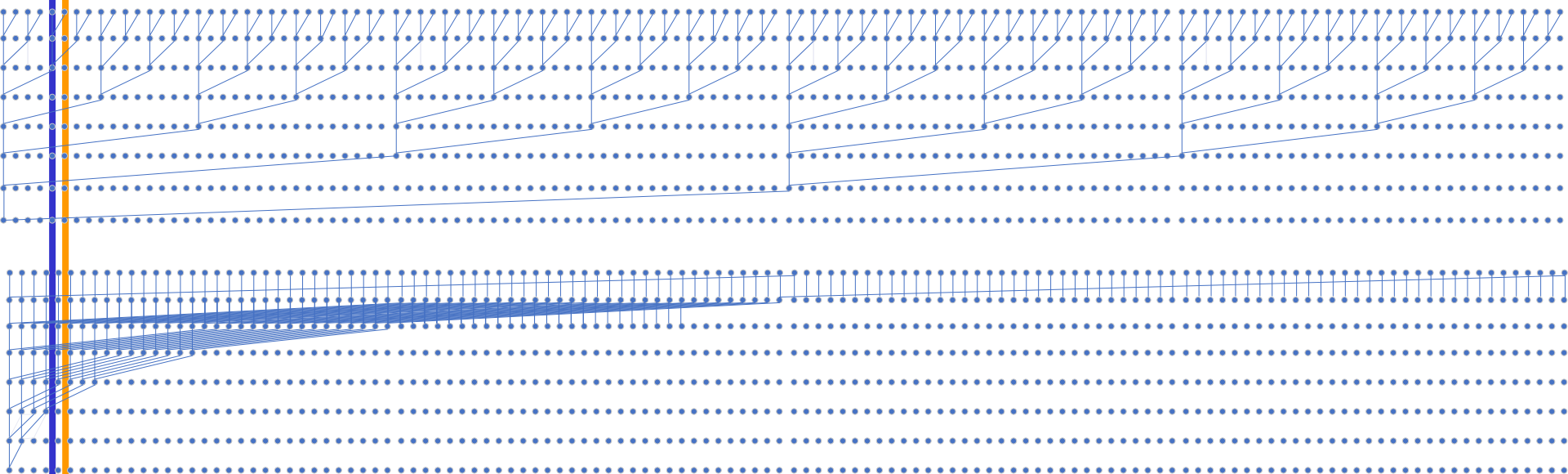
线程索引的使用是非常重要的

- 在某些算法中，可以改变索引的使用以改善发散行为
 - 交换和结合运算符
- 始终将部分求和结果压缩到 `partialSum[]` 数组的前面位置
- 保持活动线程连续



示例：4-线程





电子科技大学
University of Electronic Science and Technology of China

更好的归约内核

```
for (unsigned int stride = blockDim.x;  
    stride > 0;  stride /= 2)  
{  
    __syncthreads();  
    if (t < stride)  
        partialSum[t] += partialSum[t+stride];  
}
```



快速分析

- 对于一个包含 1024 个线程的线程块
 - 前 6 步中没有发散现象
 - 每一步有 1024、512、256、128、64、32 个连续线程处于活动状态
 - 每个线程束中的所有线程都处于活动状态或全部处于非活动状态
 - 后 5 步中仍将有发散现象



目录

- 线程束 (Warps) 与单指令多数据流指令集 (SIMD)
- 控制分支现象 (Control Divergence) 的性能影响
- 并行规约 (Parallel Reduction)
- **内存并行性 (Memory Parallelism)**



小节目标

- 了解内存带宽是影响大规模并行处理器性能的一级因素
 - DRAM 迸发、组和通道
 - 所有概念也适用于现代多核处理器



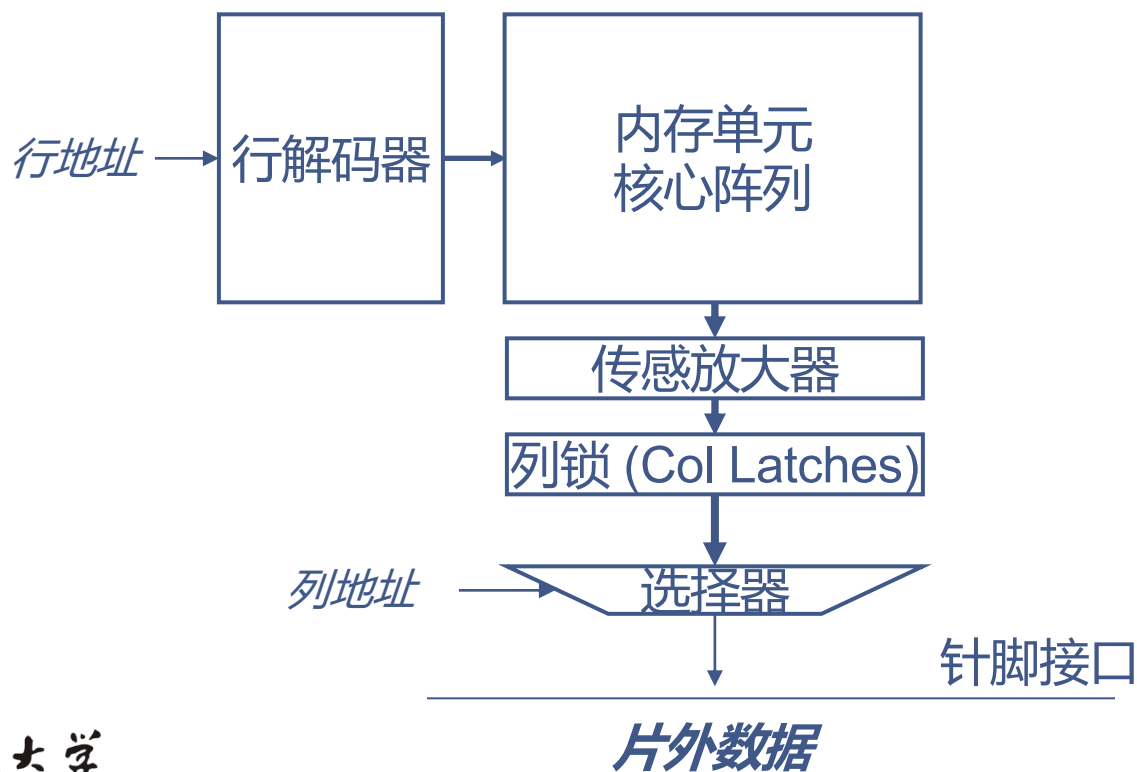
全局内存 (DRAM) 带宽

- Ideal
- Reality

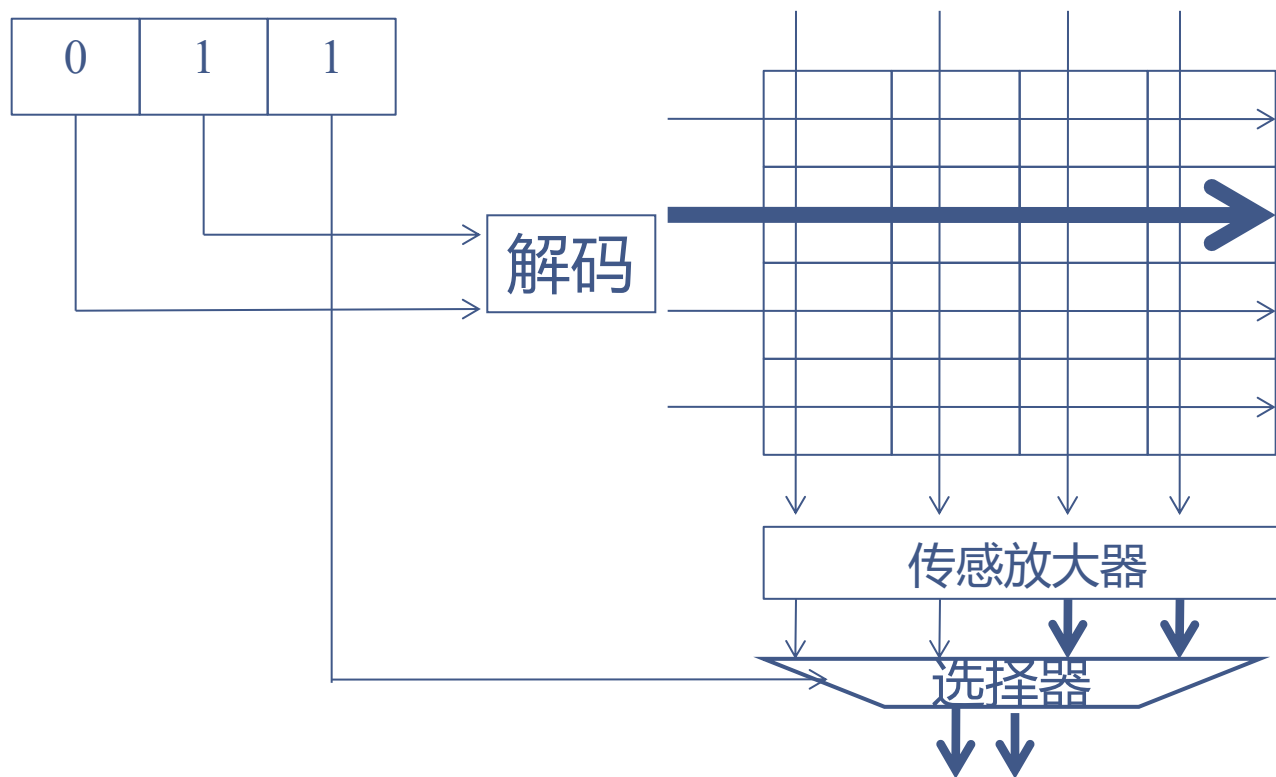


DRAM 核心阵列

- 每个 DRAM 核心阵列大约有 16M 位
- 每个位存储在一个由一个晶体管（1T1C）制成的微型电容器中

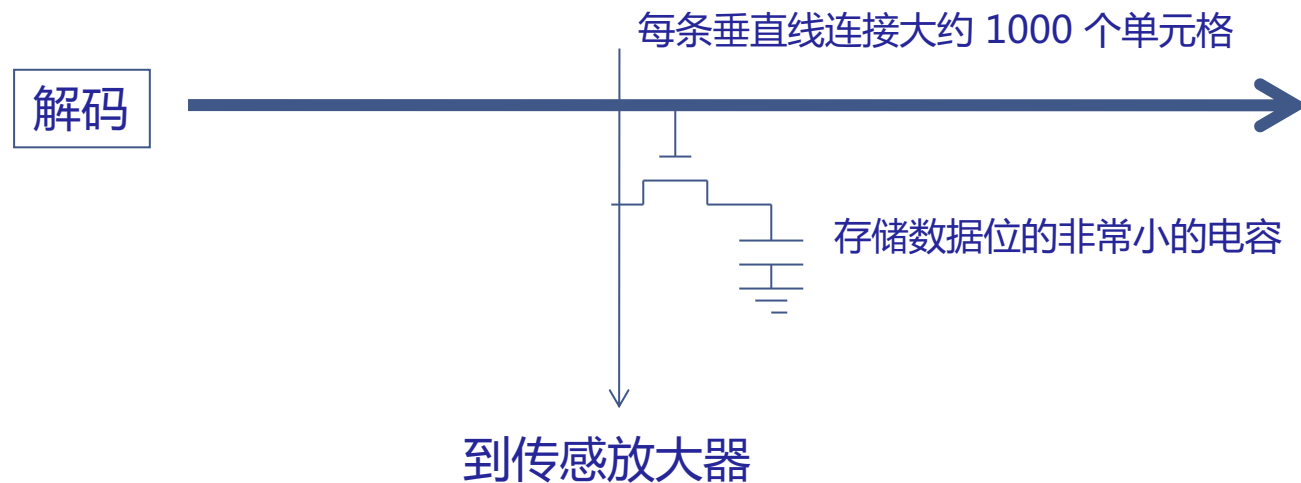


一个示例的 (8x2-bit) DRAM 核心阵列



DRAM 核心阵列——很慢

- 从核心阵列中的单元读取是一个非常缓慢的过程
 - DDR：核心速度 = $\frac{1}{2}$ 接口速度
 - DDR2/GDDR3：核心速度 = $\frac{1}{4}$ 接口速度
 - DDR3/GDDR4：核心速度 = $\frac{1}{8}$ 接口速度
 - ...未来可能会更糟

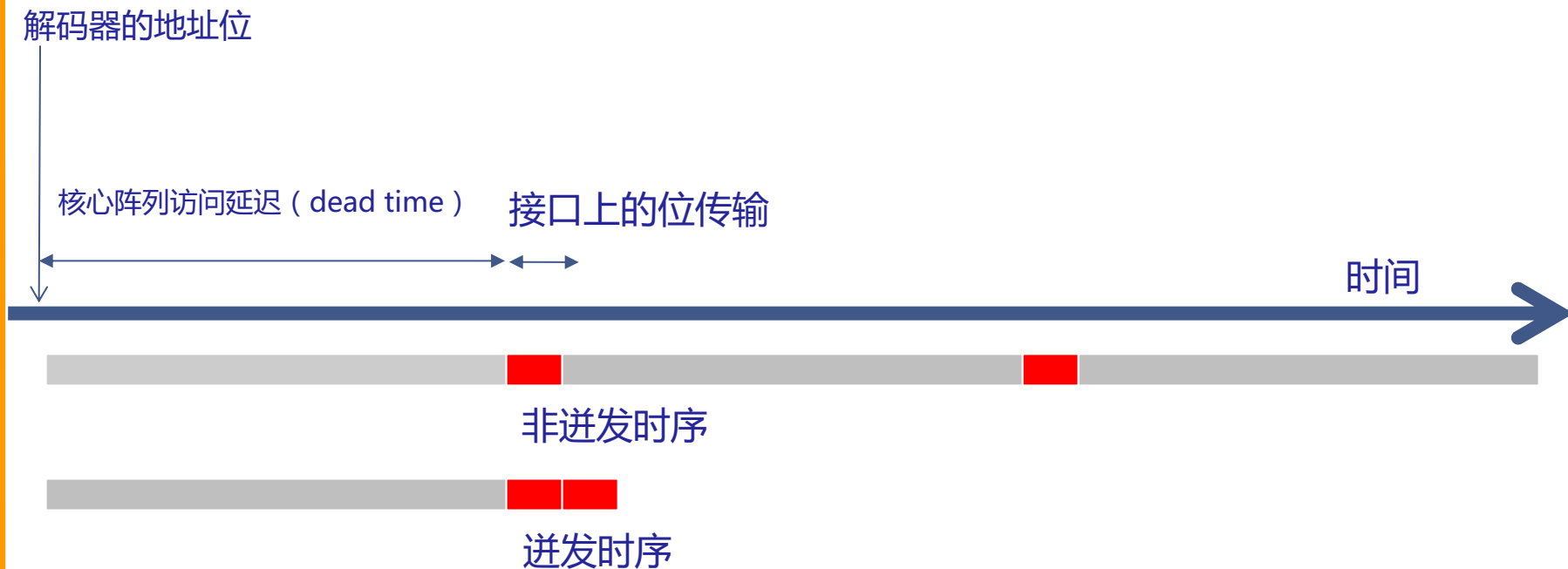


DRAM 进发

- 对于时钟频率为接口速度 $1/N$ 的 DDR{2,3} SDRAM 内核：
 - 将同一行的 DRAM 位一次加载 ($N \times$ 接口宽度) 到内部缓冲区，然后以接口速度以 N 步传输
 - DDR3/GDDR4: 缓冲区宽度 = $8X$ 接口宽度



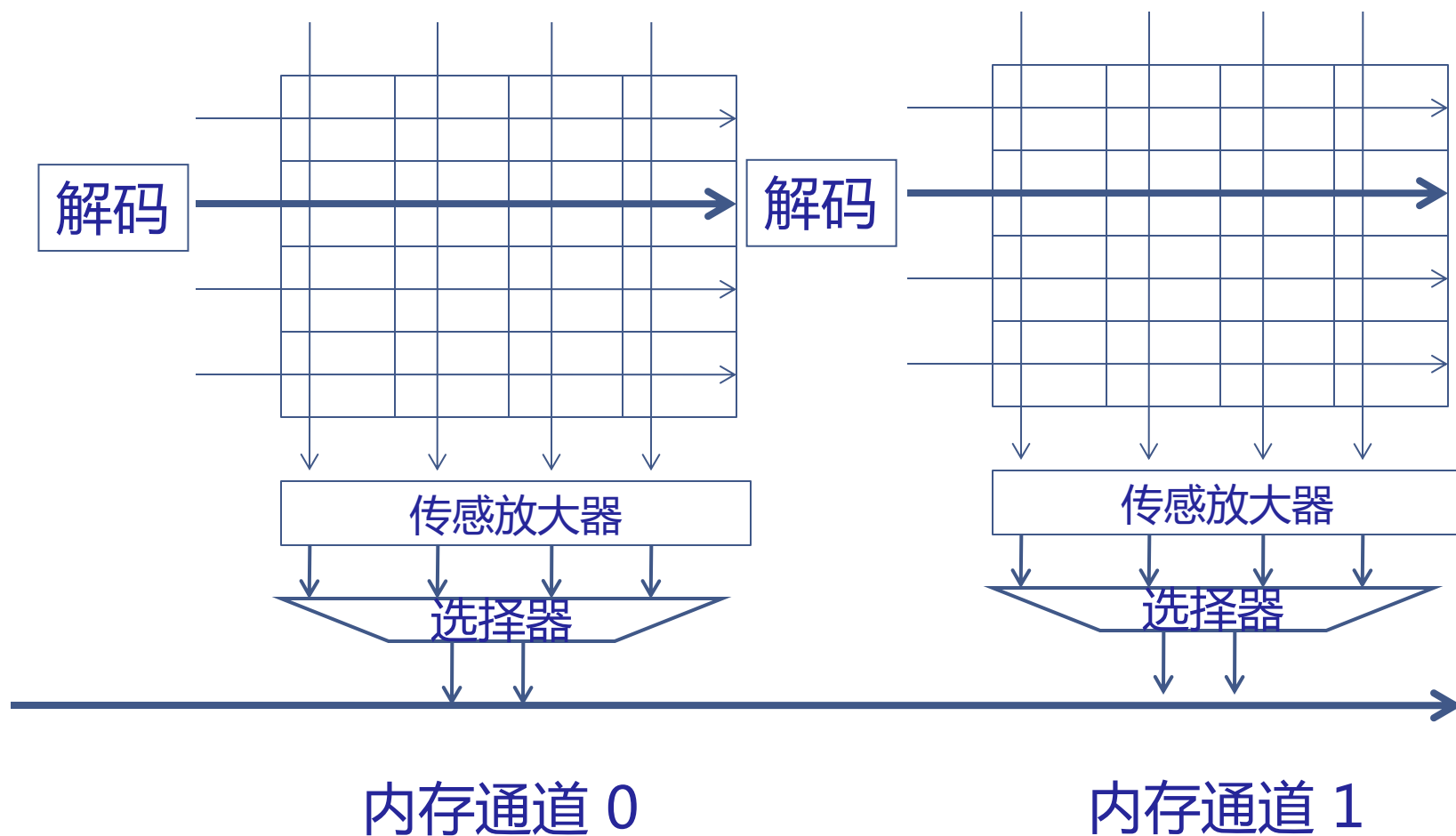
示例：DRAM 迸发时序(Burst mode)



现代 DRAM 系统设计为始终以迸发模式访问。
迸发字节被传输到处理器，但在访问不是顺序位置时被丢弃。



多个 DRAM 内存通道



多通道内存的 DRAM 迸发



单内存通道的迸发时序，死区时间在接口上



多内存通道的迸发时序，减少死区时间

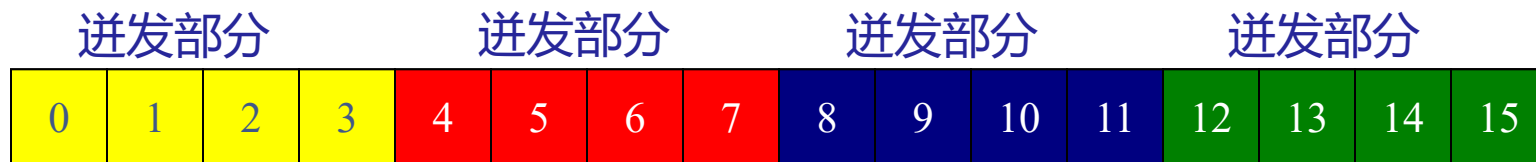


GPU 片外内存子系统

- NVIDIA GTX280 GPU:
 - 全局内存带宽峰值 = 141.7GB/s
- 全局内存 (GDDR3) 接口 @ 1.1GHz
 - (核心频率 @ 276Mhz)
 - 对于典型的 64 位接口，我们只能维持大约 17.6 GB/s (DDR - 每个时钟 2 次传输)
 - 我们需要更多的带宽 (141.7 GB/s) —— 因此需要 8 个内存通道

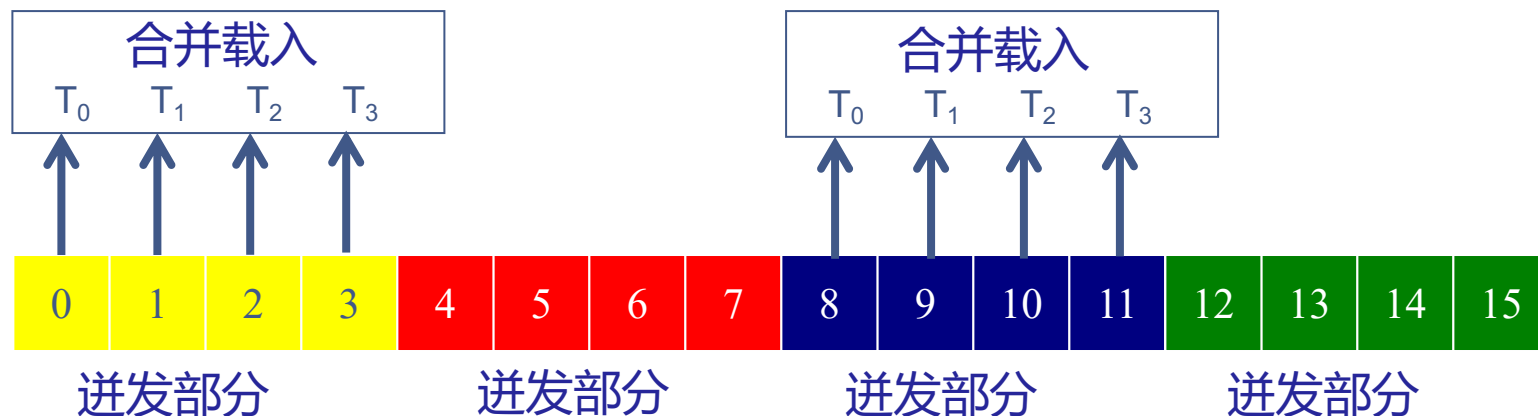


DRAM 进发 – 系统视角



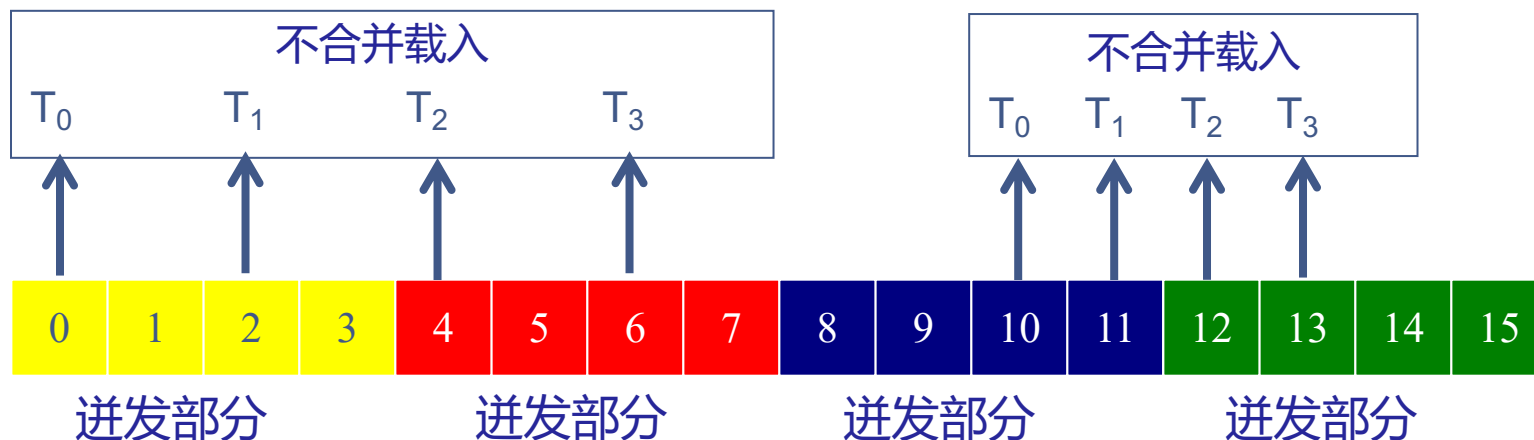
- 每个地址空间被划分为多个进发部分
 - 每当访问一个位置时，同一部分中的所有其他位置也会传送给处理器
- 基本示例：16 字节的地址空间，4 字节的进发部分
 - 事实上，我们至少有 4GB 地址空间，进发部分大小为 128 字节或更多

合并访存



- 当一个线程束的所有线程都执行一条加载指令时，如果所有访问的位置都落入同一个进发部分，那么只会发出一个DRAM请求，访问将被全部合并。

不合并的访问



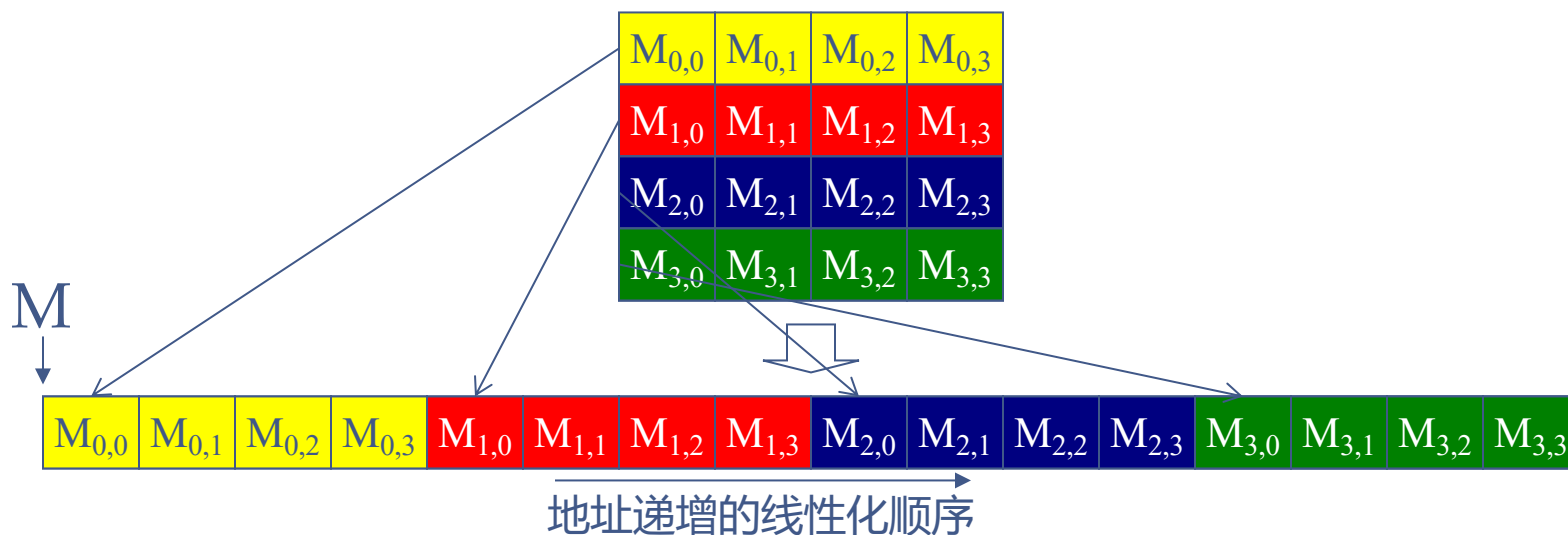
- 当访问的位置跨越进发段边界时：
 - 合并失败
 - 产生多个 DRAM 请求
 - 访问没有被全部合并
- 访问和传输的某些字节未被线程使用

如何判断一个访问是否被合并？

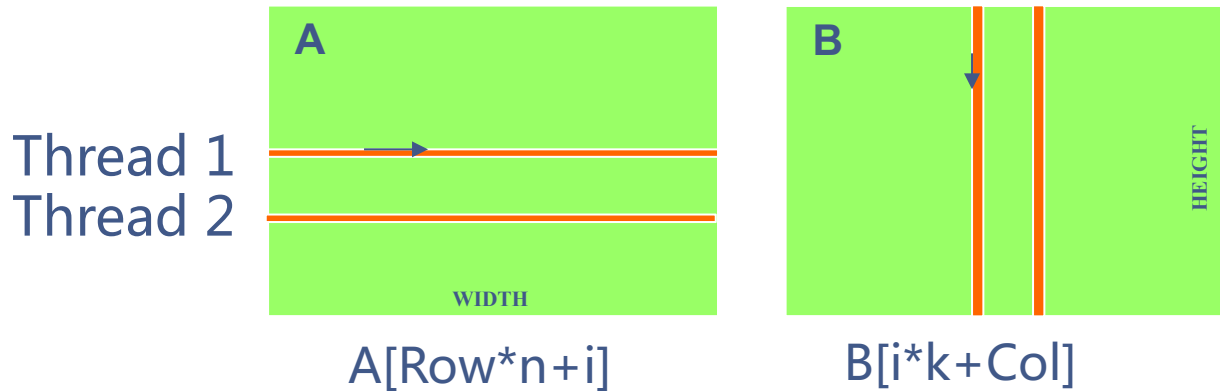
- 如果数组访问中的索引采用以下形式，则线程束中的访问是对连续位置的访问
 - $A[(\text{expression with terms independent of threadIdx.x}) + \text{threadIdx.x}]$;



线性存储空间中的二维数组



基本矩阵乘法的两种访问模式



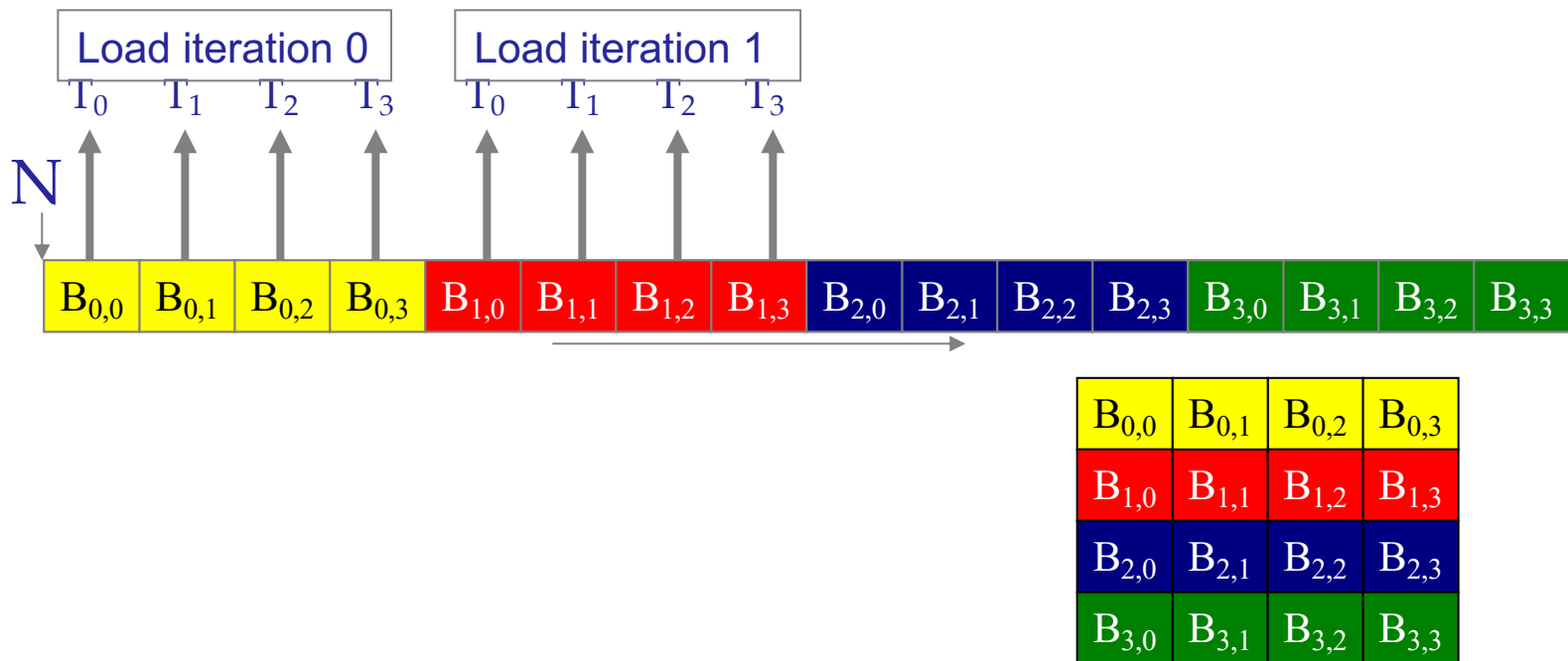
i 是内核代码内积循环中的循环计数器

A is $m \times n$, B is $n \times k$

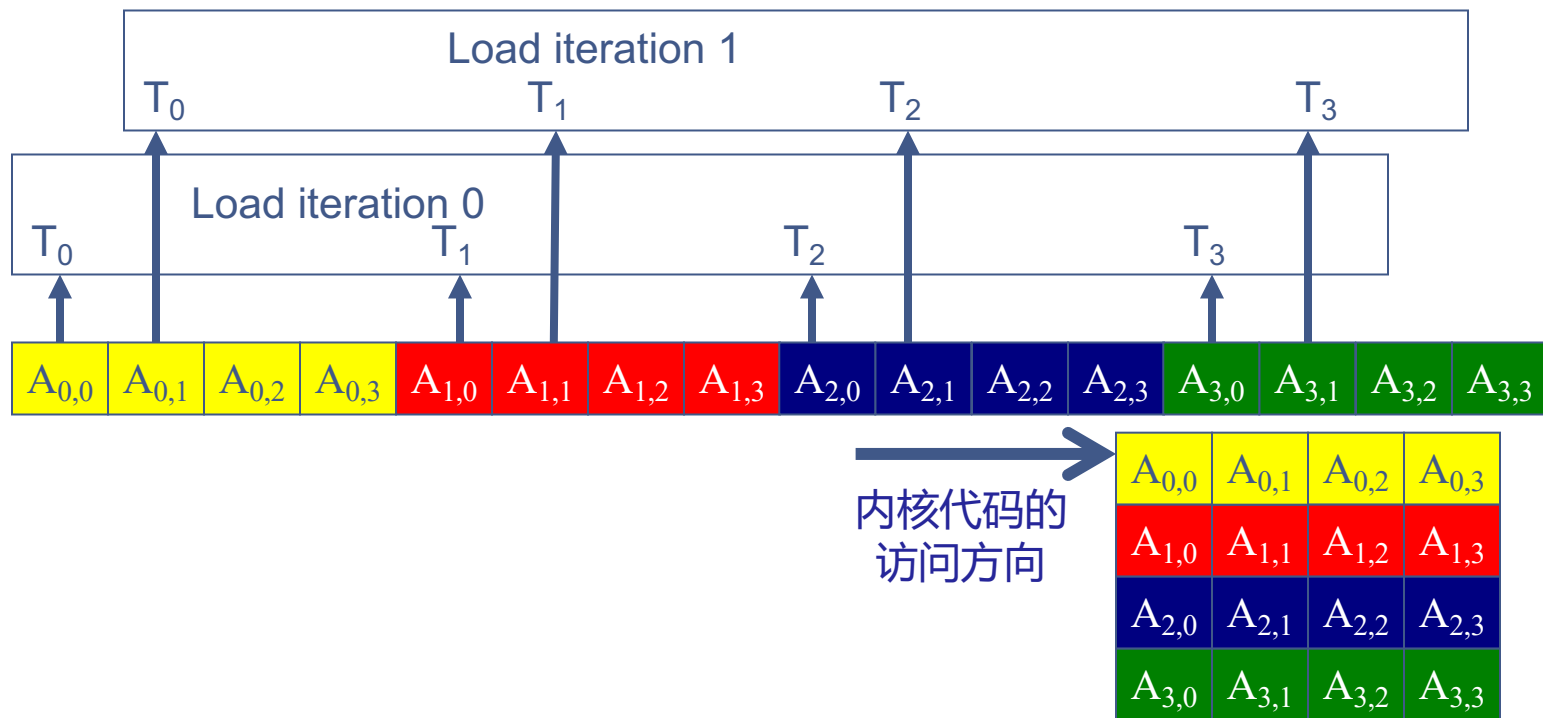
$\text{Col} = \text{blockIdx.x} \times \text{blockDim.x} + \text{threadIdx.x}$



访问被合并



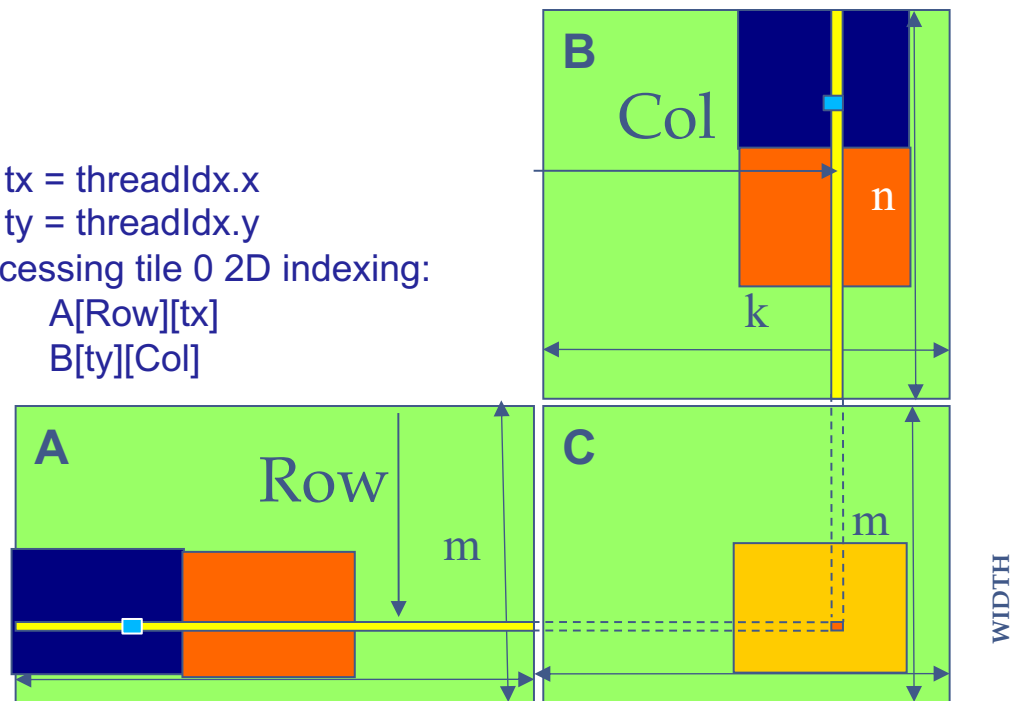
访问未被合并



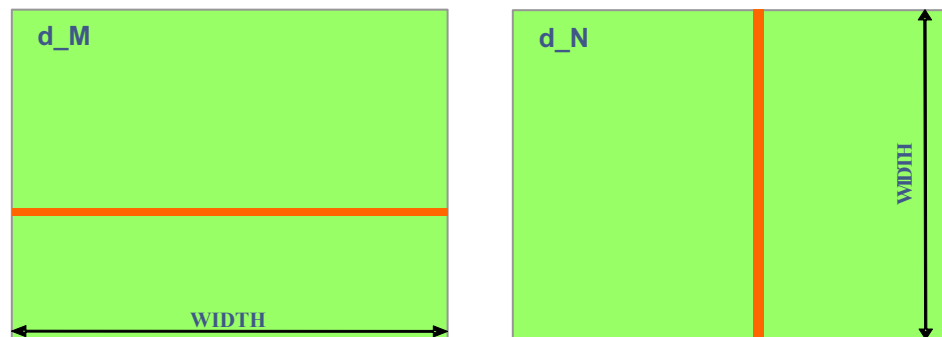
加载输入分片

让每个线程加载一个 A 元素和一个 B 元素
在同一个相对位置位置作为它的 C 元素。

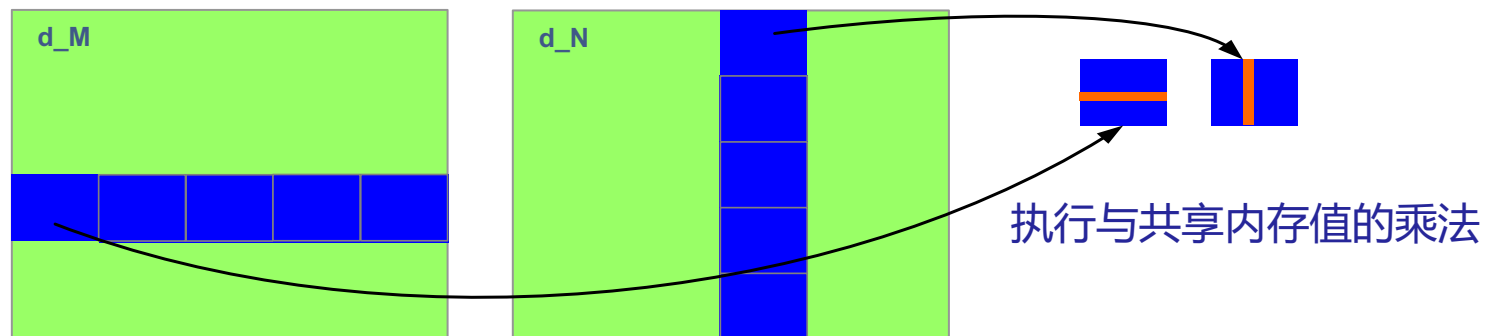
```
int tx = threadIdx.x  
int ty = threadIdx.y  
Accessing tile 0 2D indexing:  
A[Row][tx]  
B[ty][Col]
```



原始访问模式



平铺访问模式



ANY QUESTIONS?



电子科技大学
University of Electronic Science and Technology of China