

COMP4007: 并行处理和体系结构

第五章： GPU计算和CUDA并行编程II

授课老师：王强、施少怀
助 教：林稳翔、刘虎成

哈尔滨工业大学（深圳）

- ▶ CUDA线程组织结构
- ▶ 基于多维度数据的线程映射
- ▶ 基于硬件的线程映射
- ▶ 示例：矩阵乘法
- ▶ GPU内存：共享内存
- ▶ 分析-并行-优化-部署[Assess, Parallelize, Optimize and Deploy, APOD]设计流程
- ▶ CUDA优化技术
 - ✎ 内核(Kernel)配置: # 线程块数, # 线程数/线程块
 - ✎ 控制流
 - ✎ 全局内存访问
 - ✎ 共享内存访问
 - ✎ 指令优化

▶ 内核执行时线程网格(Grid)会按配置被组织成 1-D、2-D 或 3-D 线程块阵列

✎ 线程网络的配置可以通过内置变量 `gridDim` 获取

☞ `gridDim.x`, `gridDim.y`, `gridDim.z`

☞ GPU计算能力小于3.0时, `x`, `y`, `z`必须小于65536

☞ GPU计算能力大于3.0时, `x`必须小于 2^{31} , 同时`y`和`z`必须小于65536

✎ 每个线程块由内置变量 `blockIdx` 标识

☞ `blockIdx.x`, `blockIdx.y`, `blockIdx.z`

✎ 线程块中线程的最大数量

☞ 计算能力为1.x时为512

☞ 计算能力为2.x或以上时为1024

¹有关计算能力的详细信息, 请参见附录 G的参考文献 [2] 。

▶ 线程块(Block)被进一步按照配置组织成1-D、2-D或3-D线程阵列

✎ 通过内置变量 `blockDim` 可以获得线程块的结构

☞ `blockDim.x`, `blockDim.y`, `blockDim.z`

☞ `blockDim.z` 必须小于或等于 64

✎ 同一线程块中的所有线程共享相同的 `blockIdx` 值

✎ 每个线程由内置变量 `threadIdx` 标识

☞ `threadIdx.x`, `threadIdx.y`, `threadIdx.z`

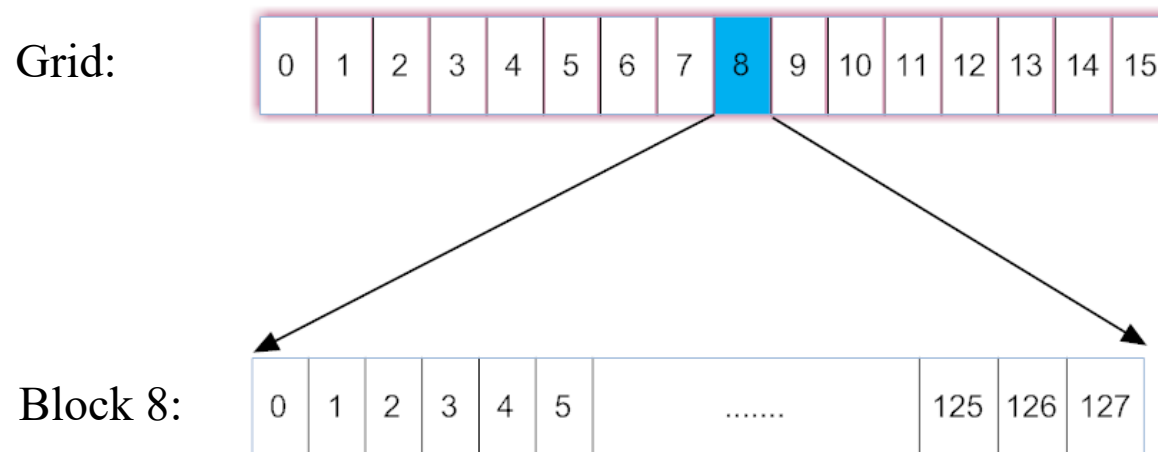
1D 示例

- ▶ 如果要创建一个有 16 个 1D 线程块的网格，每个 1D 线程块中有 128 个线程：

✎ `dim3 dimBlock(128, 1, 1); /* 设置 y和z为1, 可得到1D线程块 */`

✎ `dim3 dimGrid(16, 1, 1); /* 设置y和z为1, 可得到1D网格 */`

✎ `vecAddkernel<<<dimGrid, dimBlock>>>(...);`



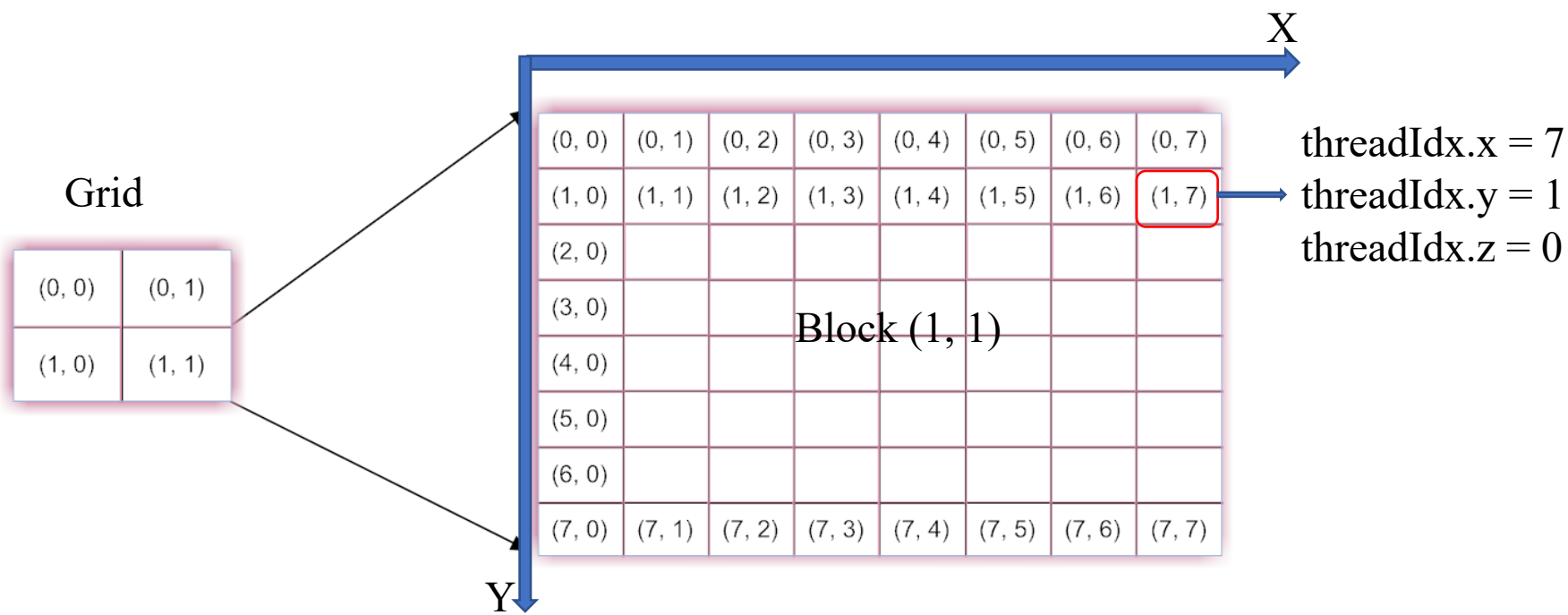
2D 示例

- ▶ 如果想创建一个拥有 2×2 个线程块的二维网格，每个线程块有 8×8 个线程：

✎ `dim3 dimBlock(8, 8, 1); /* 设置z为1, 可得到2D线程块 */`

✎ `dim3 dimGrid(2, 2, 1); /* 设置z为1, 可得到2D网格 */`

✎ `vecAddkernel<<<dimGrid, dimBlock>>>(...);`



基于多维度数据的线程映射



- ▶ 通常会基于数据的天然属性选择使用1D, 2D或3D线程组织结构
- ▶ 例如: 图像是2D像素阵列
 - ✎ 使用由2D线程块组成的2D网格表示较为方便
 - ✎ 每个线程处理一个像素: 通过 `blockDim`、`blockIdx` 和 `threadIdx` 可以轻松计算出像素的位置

示例: 76×62分辨率的图片表示

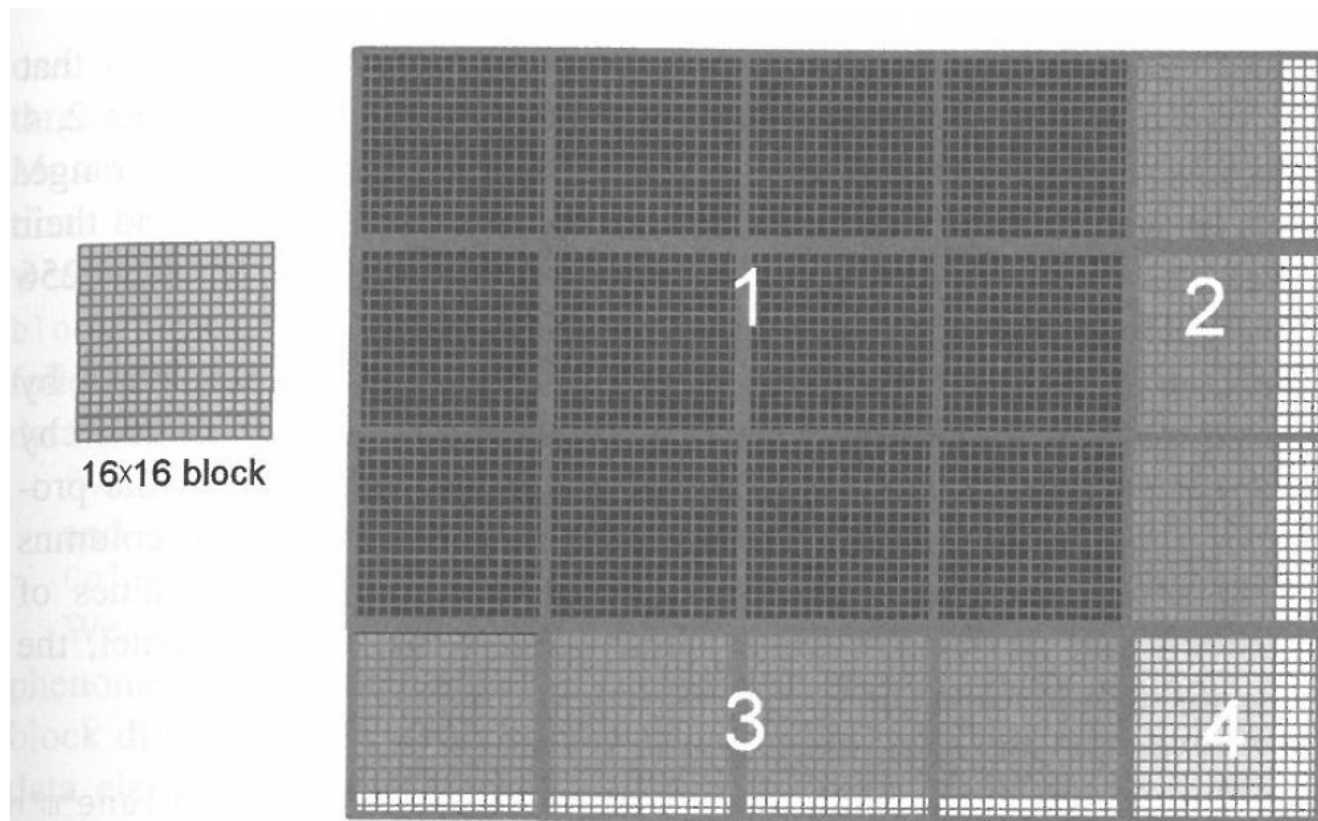


FIGURE 4.5

Covering a 76×62 picture with 16×16 blocks.

图片大小:
 76×62




线程块大小:
 16×16

共需要 5×4 个线程
块!

最终生成 80×64
个线程, 其中一
些会被浪费。

参考文献1中的图4.5

- ▶ 假设图像的分辨率为 $n \times m$

```
 dim3 dimGrid(ceil(n/16.0), ceil(m/16.0), 1);  
 dim3 dimBlock(16, 16, 1);  
 pictureKernel<<<dimGrid, dimBlock>>>(...);
```



线程如何在 GPU 上执行？

为线程块分配资源

- ▶ 内核启动后，CUDA 会生成相应的线程网格
- ▶ 线程首先会被组织成多个线程块，然后以线程块为单位分配给执行资源（即流式多处理器SM）
 - ✎需求：需要执行多个线程块，比如总共 N 个块
 - ✎资源：数量有限的 SM，如 P 个SM
 - ✎每个 SM 将平均执行 N/P 个区块
- ▶ 活动线程块（或“常驻区块”）
 - ✎一个 SM 可以同时处理多个线程块。我们称之为“活动线程块”
 - ✎当前的活动线程块结束后，一组新的线程块将分配给 SM，并成为新的活动线程块

▶ 线程块被分配给SM后会被进一步划分为warps

✎ 每个`warp`包括 32个线程

✎ Warp 0: 线程 0 – 31

✎ Warp 1: 线程 32 – 63

✎ Warp 2: 线程 64 - 95

✎

▶ Warp是线程在SM中调度的基本单位

✎ Warp中的 32 个线程遵循SIMD模型

✎ 获取一条指令后，warp中的所有 32 个线程均同时执行该指令

多warp的优势: 延迟隐藏



▶ 通过多warp可隐藏延迟

- ✎ warp 执行的指令需要等待数据时, 该warp会“休眠”

- ☞ 全局存储器的延迟时间可达数百个 GPU 周期

- ✎ 数据准备就绪的另一个warp将被选中执行

- ☞ 为了充分利用执行硬件

- ✎ 零开销的线程调度

- ☞ 选择新的活动warp执行的开销很小

▶ 一般来说, 应该为每个 SM 调度“足够的”活动Warp

- ▶ 每个 SM 的活动线程块/warps/线程数量有限。

计算能力	1.x	2.x	3.x	5.0	7.0
每个 SM 的最大活动线程块数	8	8	16	32	32
每个 SM 的最大活动warp数	24 → 32	48	64	64	64
每个 SM 的最大活动线程数	768 → 1024	1536	2048	2048	2048
每个线程块包含的最大线程数	512	1024	1024	1024	1024

备注：其他限制因素包括寄存器和共享内存，这此将在后面讨论

示例



- ▶ 假设计算能力为 3.0
- ▶ 情况 1: 线程块大小为 $8 \times 8 = 64$
 - ✎ 最多可以有多少个活动线程?
- ▶ 情况 2: 线程块大小为 $16 \times 16 = 256$
 - ✎ 最多可以有多少个活动线程?
- ▶ 情况3: 线程块大小为 $32 \times 32 = 1024$
 - ✎ 最多可以有多少个活动线程?
- ▶ 情况4: 线程块大小为 $64 \times 64 = 4096$
 - ✎ 将会发生什么呢?

内核函数

/* d_Pin指向图片数据*/

```
__global__ void PictureKernel(float *d_Pin, float *d_Pout, int n, int m)
{
    int Row = blockIdx.y * blockDim.y + threadIdx.y;
    int Col = blockIdx.x * blockDim.x + threadIdx.x;

    if( (Row < m) && (Col < n) )
        d_Pout[Row * n + Col] = 2 * d_Pin[Row * n + Col];
}
```

查找线程对应的输入数据



考虑目标线程:

✎ 线程所属的本地线程块上方有 blockIdx.y 行线程块

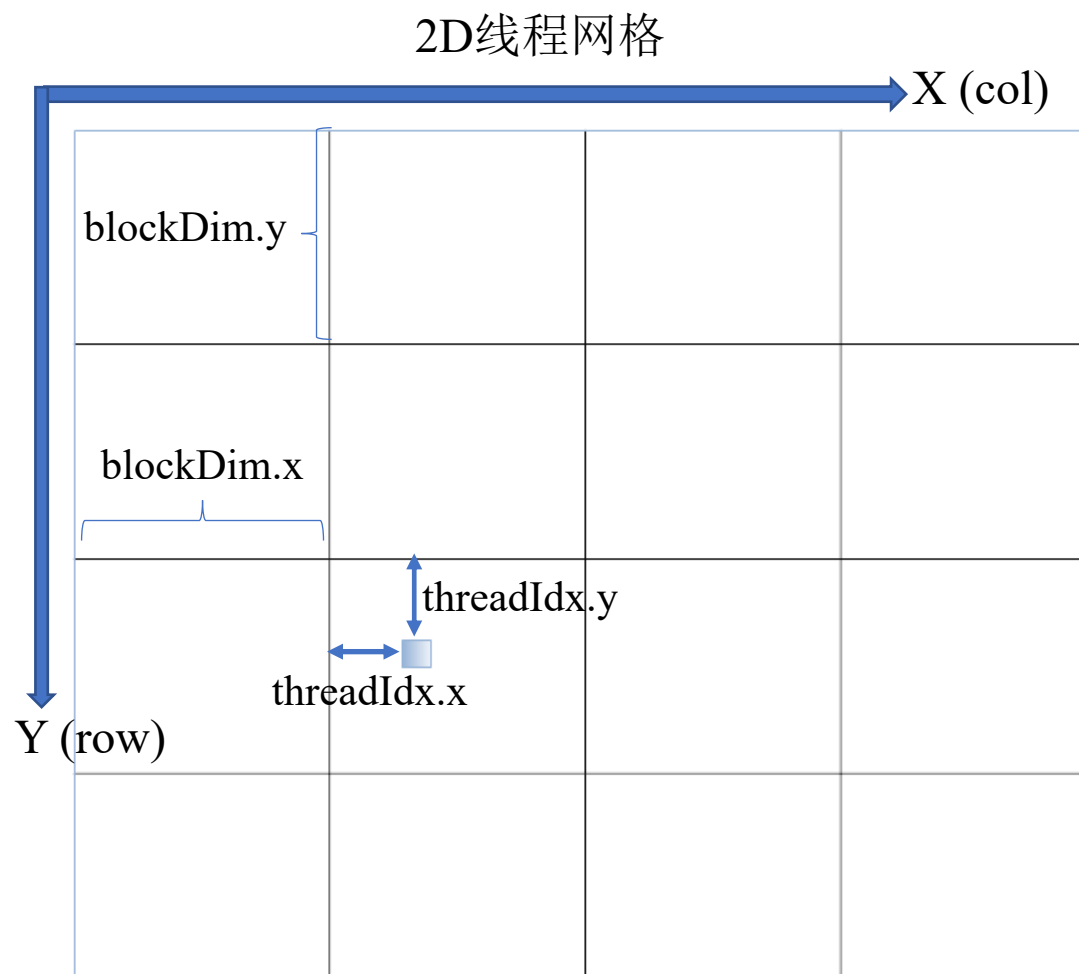


$$\text{Row} = \text{blockIdx.y} * \text{blockDim.y} + \text{threadIdx.y}$$

✎ 线程所属的本地线程块左边有 blockIdx.x 个线程块



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





示例：矩阵乘法

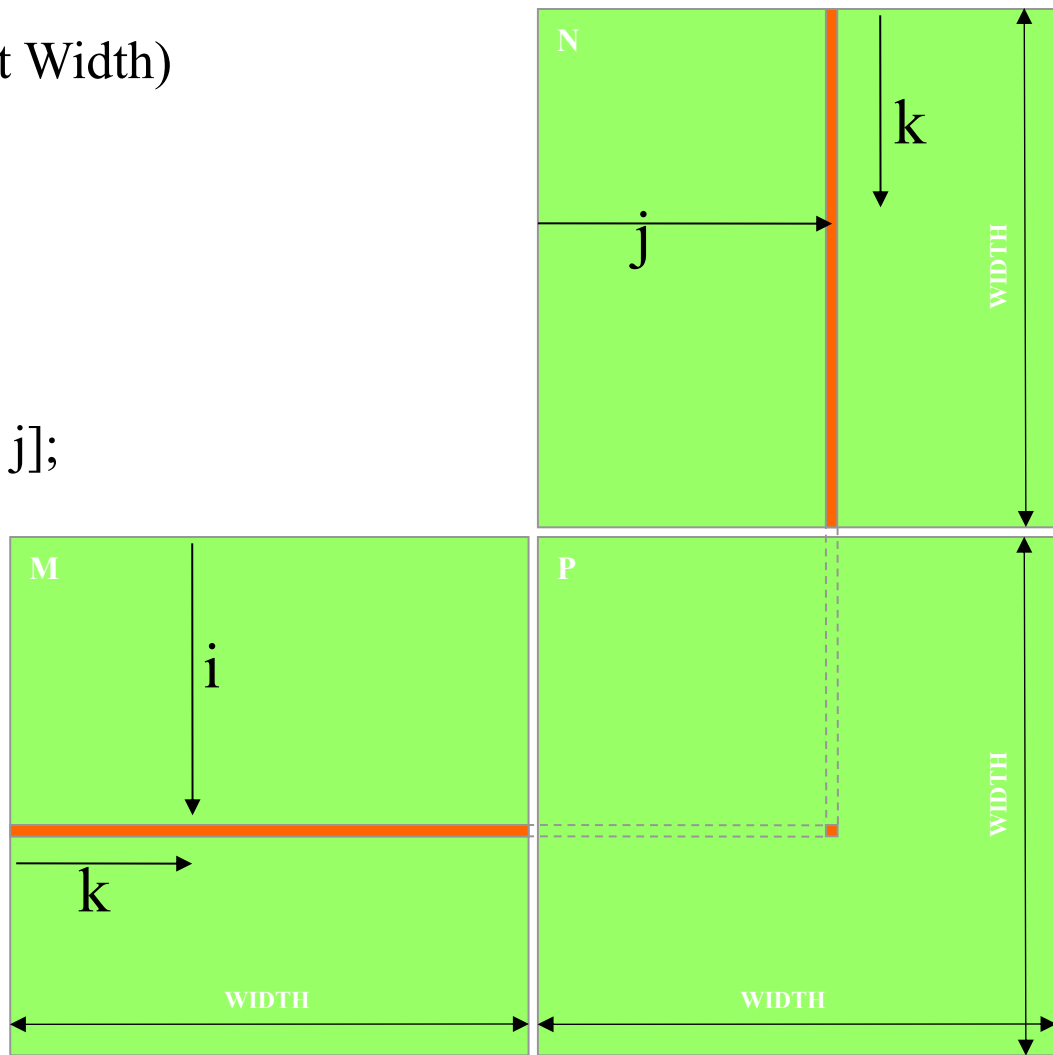
- ▶ 问题： $P = M \times N$
- ▶ M, N, P 为方阵: $WIDTH \times WIDTH$
- ▶ GPU 上的并行化
 - ✎ 矩阵 P 中每个元素的计算由 GPU 线程完成
 - ☞ 计算两个向量的点积
 - ✎ 问题：如何将线程组织成线程块？

串行方案



// CPU上的矩阵乘法

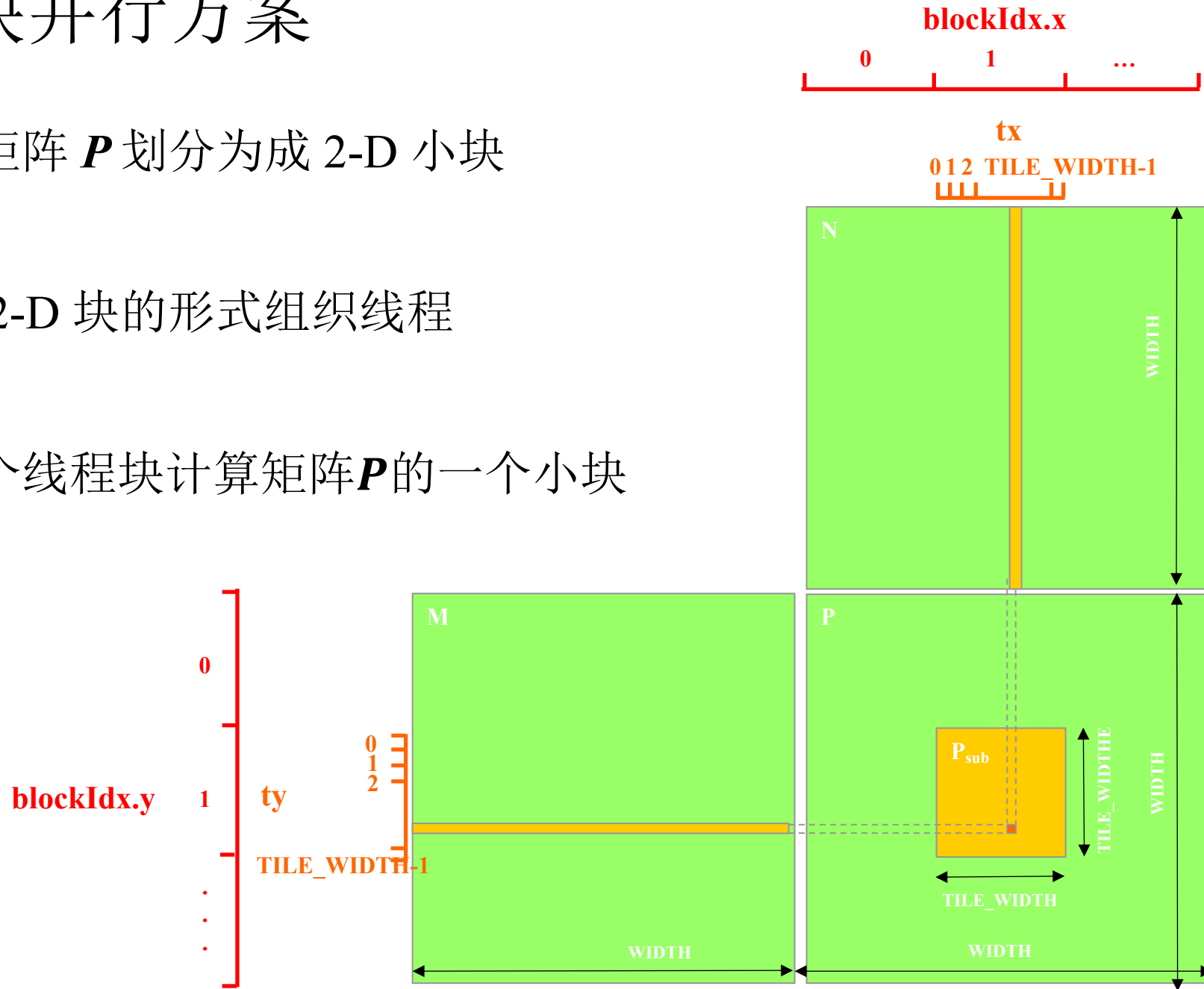
```
void MatMulOnHost(float* M, float* N, float* P, int Width)
{
    for (int i = 0; i < Width; i++)
        for (int j = 0; j < Width; j++) {
            float sum = 0.0;
            for (int k = 0; k < Width; k++) {
                sum += M[i * width + k] * N[k * width + j];
            }
            P[i * Width + j] = sum;
        }
}
```



分块并行方案



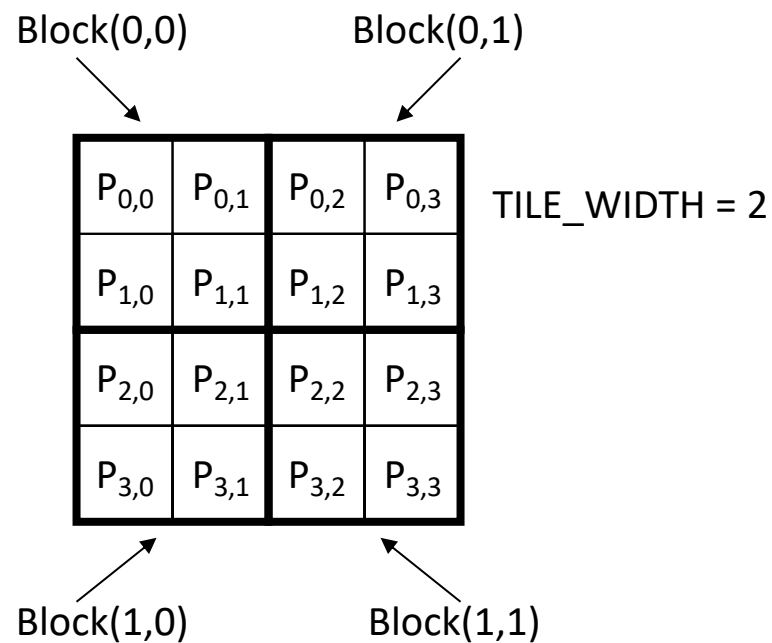
- ▶ 将矩阵 P 划分为成 2-D 小块
- ▶ 按 2-D 块的形式组织线程
- ▶ 每个线程块计算矩阵 P 的一个小块



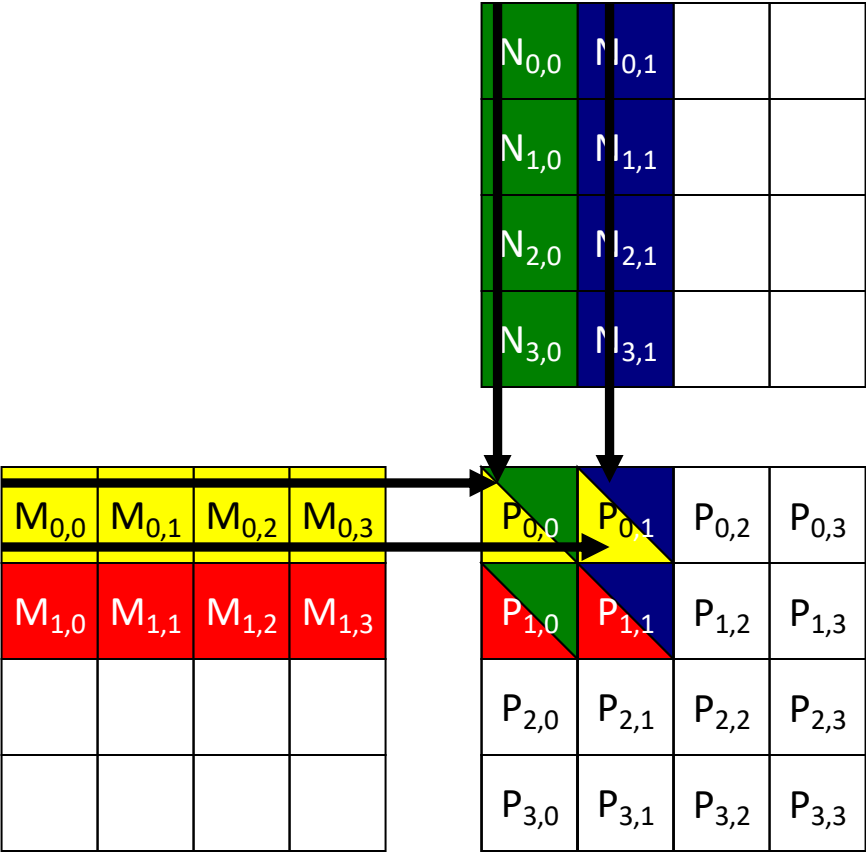
简易代码实现



- ▶ $WIDTH = 4$
- ▶ $TILE_WIDTH = 2$
- ▶ $2 \times 2 = 4$ blocks
- ▶ Each block has $2 \times 2 = 4$ threads



简易代码实现（续）



```
__global__ void MatMulKernel(float* Md, float* Nd, float* Pd, int Width)
{
    // 线程需计算的矩阵P中结果元素的行索引
    int Row = blockIdx.y*blockDim.y + threadIdx.y;
    // 线程需计算的矩阵P中结果元素的列索引

    int Col = blockIdx.x*blockDim.x + threadIdx.x;

    if ( (Row < Width) && (Col < Width) ) {
        float Pvalue = 0.0;
        // 每个线程计算矩阵P的一个元素
        for (int k = 0; k < Width; ++k)
            Pvalue += Md[Row*Width+k] * Nd[k*Width+Col];
        Pd[Row*Width+Col] = Pvalue;
    }
}
```

启动内核



```
#define TILE_WIDTH 16
```

```
// 设置内核执行时的配置
```

```
int NB = Width/TILE_WIDTH;
```

```
if (Width % TILE_WIDTH != 0) NB++;
```

```
dim3 dimGrid(NB, NB);
```

```
dim3 dimBlock(TILE_WIDTH, TILE_WIDTH);
```

```
// 启动设备上的计算线程
```

```
MatMulKernel<<<dimGrid, dimBlock>>>(Md, Nd, Pd, Width);
```

通过gettimeofday()对内核函数计时



```
#include <sys/time.h>

double cpuTime() {
    struct timeval tp;
    gettimeofday(&tp, NULL);
    return ( (double)tp.tv_sec + (double)tp.tv_usec * 1.e-6 );
}

double t_start = cpuTime();
/* 调用CUDA核函数 */
your_kernel<<<grid, block>>>(...);
/* 等待GPU完成核函数计算 */
cudaDeviceSynchronize();
double kernel_time = cpuTime() - t_start;

printf("The kernel elapsed %f seconds.\n", kernel_time);
```


性能有多好？



▶ 测试平台：一台安装windows 7操作系统的台式机

✎ CPU: Intel Core i7-3770 @3.9GHz

✎ GPU: Nvidia GT640 with 384 cores @900MHz

▶ 运行时间对比

Matrix Size	N = 512	N = 1024	N = 1536	N = 2048
GPU time	18.5ms	141ms	470.7ms	1112ms*
CPU time	181.6ms	6817ms	23863ms	67797ms
GPU Flops	14.5G	15.2G	15.4G	15.4G
CPU Flops	1.48G	0.31G	0.30G	0.25G
Speedup	9.8	48.3	50.7	61

*备注：N = 2048时，程序在Nvidia GTX780上的运行时间为112.7ms

性能足够了吗？

- ▶ GTX780 拥有 2304 个CUDA核，其理论单精度计算能力接近 4TFlops

✎ 前文的代码仅能达到 $2 \times 20483 / 112.7 \text{ms} = 152.4 \text{GFlops}$

✎ 即 GPU 利用率仅为3.8%!

- ▶ 问题出在哪儿？

✎ 提示： GT640 的显存带宽为 28.5GB/s， GTX780 的显存带宽为 288GB/s

✎ 内存访问是瓶颈！

CGMA: 计算与全局内存访问比率



```
/* 内核函数的关键部分 */  
for (int k = 0; k < Width; ++k)  
    Pvalue += Md[Row*Width+k] * Nd[k*Width+Col];
```

- ▶ 上述核函数有 $2 * \text{Width}$ 的计算量和 $2 * \text{Width}$ 的全局内存访问量
 - ✎ Md[] 和 Nd[]位于GPU 全局内存
 - ✎ Pvalue位于GPU 寄存器中，速度非常快
- ▶ CGMA 比率仅为 1

GPU全局内存

▶ GPU 全局内存容量大

✎ GTX780: 3GB

✎ Tesla K40: 12GB

▶ 带宽：GPU全局内存传输数据的速度有多快？

✎ GTX640: 28.5GB/s

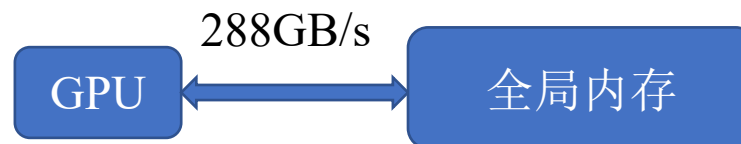
✎ GTX780: 288GB/s

✎ Tesla K40: 288GB/s

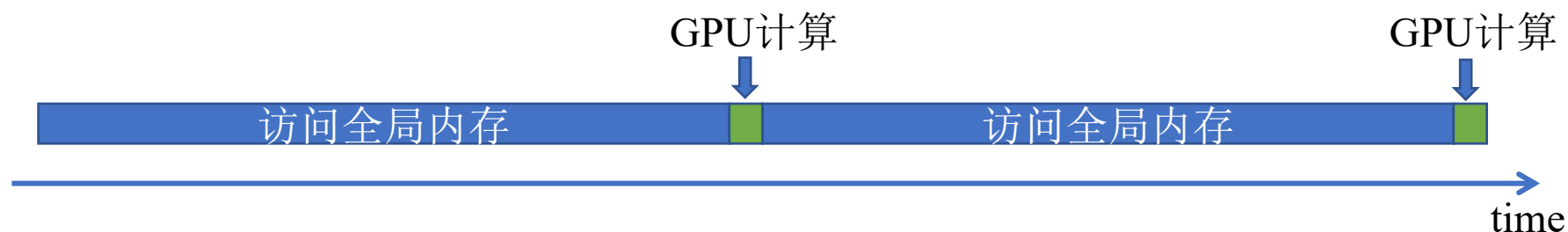
▶ 延迟：

✎ 200-400 GPU时钟周期

内存瓶颈



- ▶ 前文内核代码的 CGMA 值为 1
- ▶ 每次计算都需要从全局内存访问一次数据
- ▶ 因此，性能受到内存带宽的限制
 - ✎ 大多数时候，GPU 都无事可做！



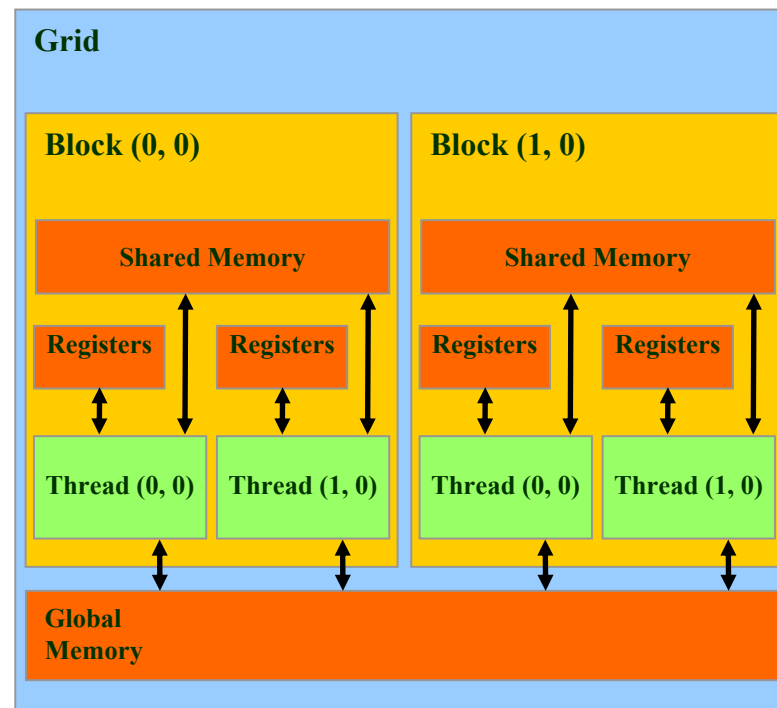
如何改进？

- ▶ 为了让 GPU 忙起来，每次全局内存访问都需要获取到更多次计算所需的数据
 - ✎即，我们应实现较高的 CGMA 比率
- ▶ CPU 利用缓存缓解内存瓶颈
- ▶ GPU利用片上“共享内存”
 - ✎最新的 GPU 还使用cache提高数据访问性能
 - ✎Cache由硬件管理，而 "共享内存 "可由程序员管理

共享内存

▶ 每个线程块都有一个共享内存

- ✎ 位于芯片上
- ✎ 带宽极高和延迟极低
- ✎ 容量有限：计算能力 2.x 或以上版本最多 48KB
- ✎ 线程块中的所有线程共享



提高 CGMA 比率的基本策略

- ▶ 准备工作：分配共享内存
- ▶ 数据加载：将数据从全局内存加载到共享内存
- ▶ 数据处理：重复使用共享内存中的数据
 - ✎ 每个数据参与的计算次数越多，CGMA 比率就越高
- ▶ 如果未结束，跳至步骤 2

回到矩阵乘法



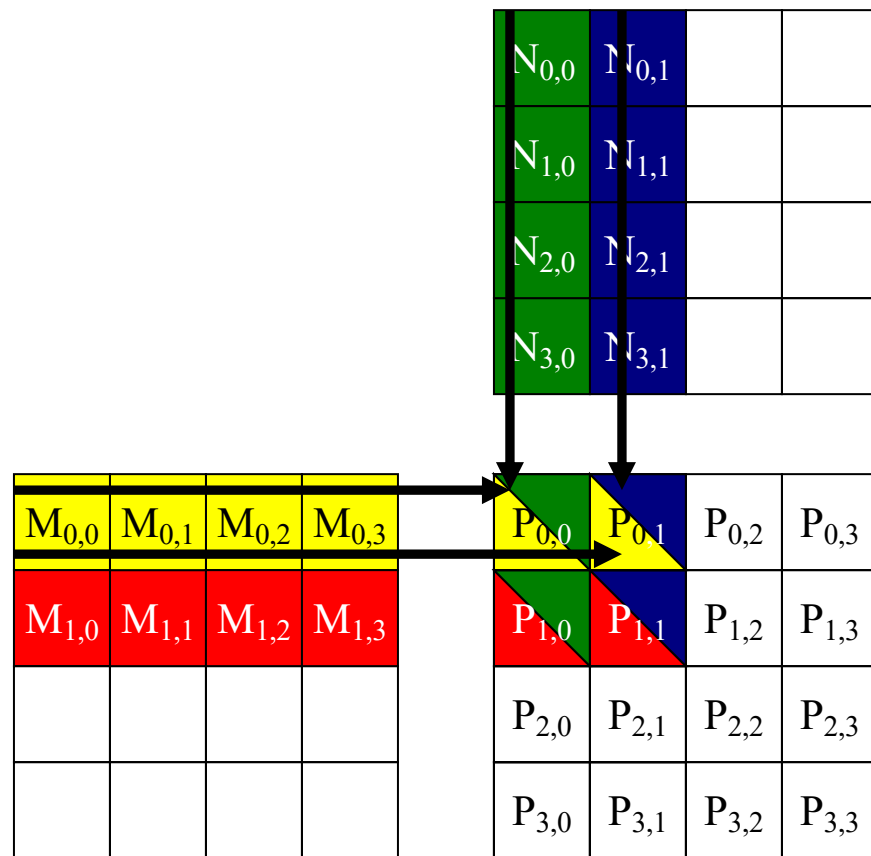
分析大小为 2×2 的线程块(0, 0)

观察

1. 计算 $P_{0,0}$ 和 $P_{0,1}$ 都需要访问矩阵 M 的第一行
2. 计算 $P_{1,0}$ 和 $P_{1,1}$ 都需要访问矩阵 M 的第二行
3. 计算 $P_{0,0}$ 和 $P_{0,1}$ 都需要访问矩阵 N 的第一行
4. 计算 $P_{1,0}$ 和 $P_{1,1}$ 都需要访问矩阵 N 的第二行

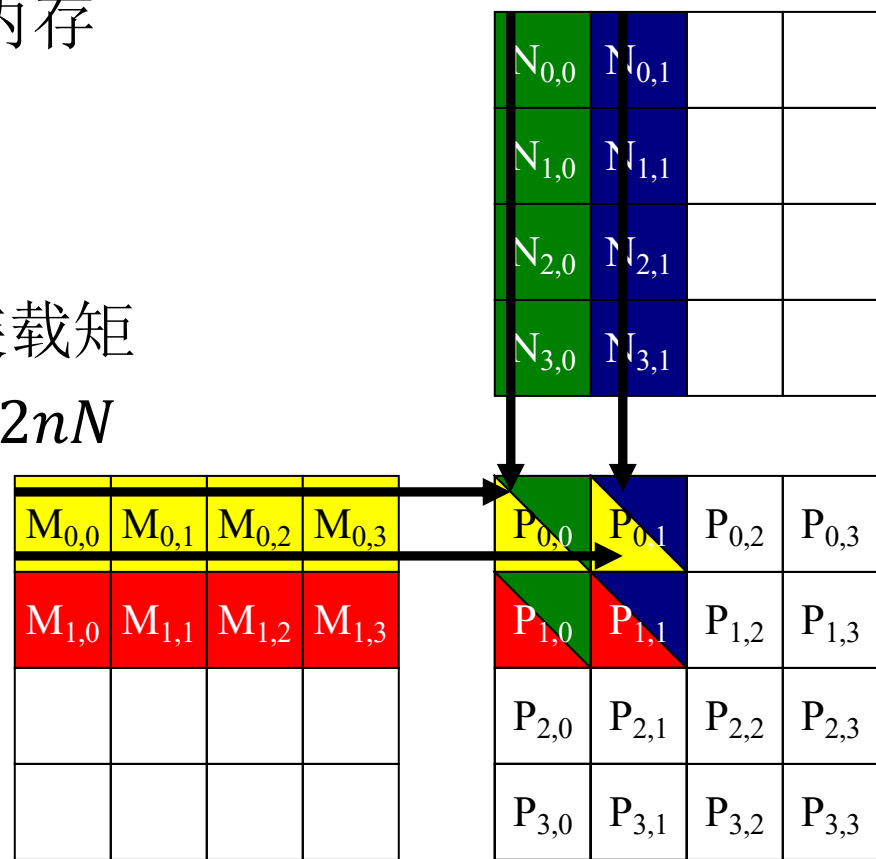
前文方案

- 每个线程块有 32 次全局内存访问
- 使用共享内存可将每个线程块的全局内存访问次数减少为16次



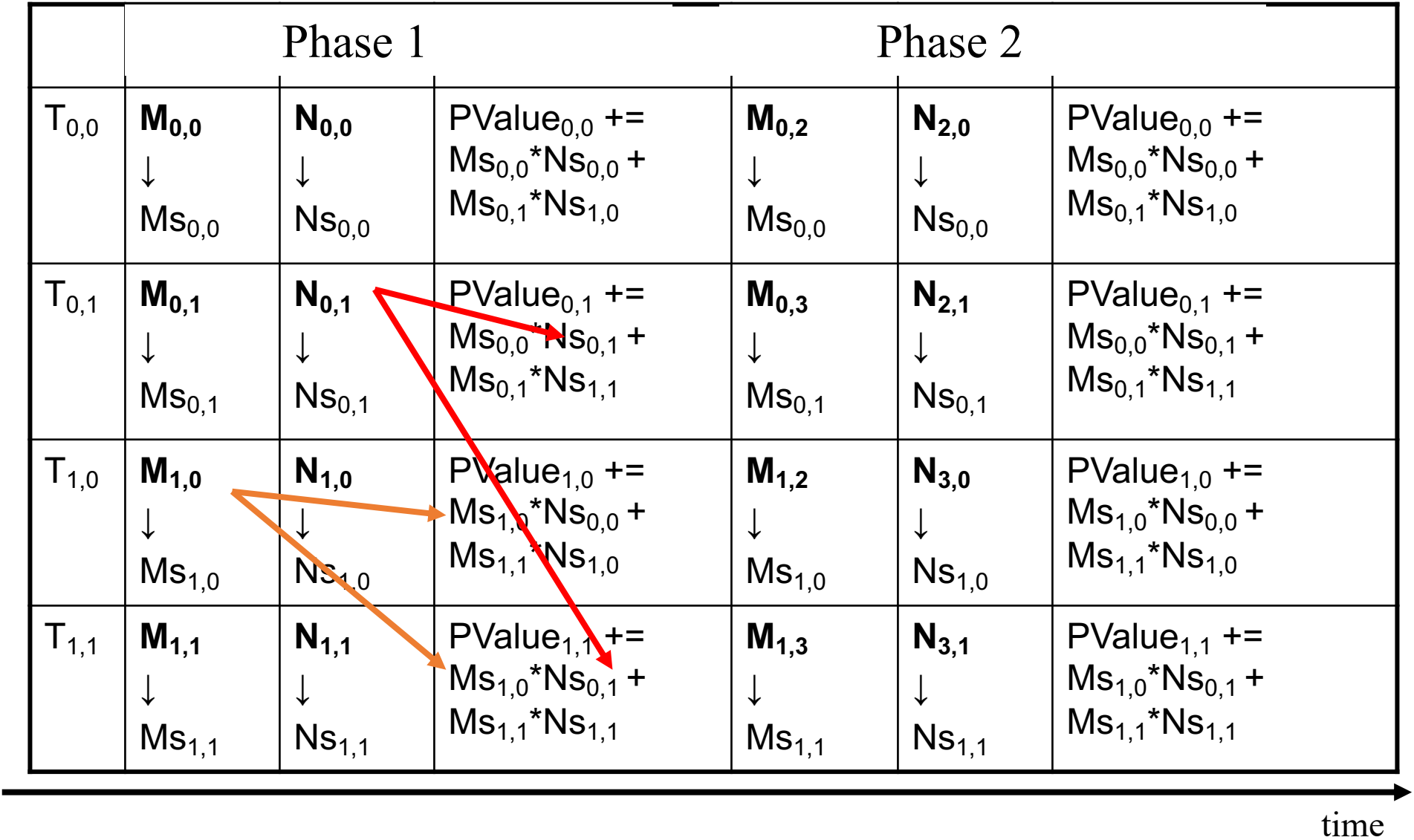
回到矩阵乘法

- 假设线程块大小为 $n \times n$ 。矩阵大小为 $N \times N$
- 之前的方案中每个线程块需进行 $2n^2N$ 次全局内存访问
- 利用共享内存，在计算时单个线程块时可只装载矩阵 M 的 n 行数据，矩阵 N 的 n 列数据，只需访问 $2nN$ 次全局内存
- 可将全局内存访问量减少 n 倍！



新的挑战

- ▶ 共享内存的大小有限
- ▶ 同时将矩阵M的n行和矩阵N中的n列加载到共享内存是不现实
- ▶ 解决方法:
 - ✎ 将n行和n列划分为多个tile，这样可按tile将矩阵M和矩阵N加载到共享内存中
 - ✎ 计算共享内存中两个tiles
 - ✎ 将新的矩阵M和矩阵N的tiles加载到共享内存，然后重复计算直到完成



新的内核函数



```
#define TILE_WIDTH 16
__global__ void MatMulKernel(float* Md, float* Nd, float* Pd, int Width)
{
1.  __shared__ float Mds[TILE_WIDTH][TILE_WIDTH];
2.  __shared__ float Nds[TILE_WIDTH][TILE_WIDTH];

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

// Identify the row and column of the Pd element to work on
5.  int Row = by * TILE_WIDTH + ty;
6.  int Col = bx * TILE_WIDTH + tx;

7.  float Pvalue = 0;
// Loop over the Md and Nd tiles required to compute the Pd element
8.  for (int m = 0; m < Width/TILE_WIDTH; ++m) {
    // Collaborative loading of Md and Nd tiles into shared memory
9.    Mds[ty][tx] = Md[Row*Width + (m*TILE_WIDTH + tx)];
10.   Nds[ty][tx] = Nd[(m*TILE_WIDTH + ty)*Width + Col];
11.   __syncthreads();

12.   for (int k = 0; k < TILE_WIDTH; ++k)
13.     Pvalue += Mds[ty][k] * Nds[k][tx];
14.   __syncthreads();
    }
15. Pd[Row*Width + Col] = Pvalue;
}
```

新的内核函数详解 (1)



```
1. __shared__ float Mds[TILE_WIDTH][TILE_WIDTH];  
2. __shared__ float Nds[TILE_WIDTH][TILE_WIDTH];
```

- ▶ 利用关键字 `__shared__` 分配共享内存
- ▶ 每个线程块将分配相同数量的共享内存
- ▶ 将为每个线程块创建一对 `Mds[][]` 和 `Nds[][]`。
- ▶ 线程块中的所有线程都可以访问 `Mds[][]` 和 `Nds[][]`。

新的内核函数详解 (2)

```
3.  int bx = blockIdx.x;  int by = blockIdx.y;  
4.  int tx = threadIdx.x; int ty = threadIdx.y;  
  
// Identify the row and column of the Pd element to work on  
5.  int Row = by * TILE_WIDTH + ty;  
6.  int Col = bx * TILE_WIDTH + tx;
```

- ▶ 上述语句用于计算线程所需要计算的结果在矩阵 d_P 中的位置
- ▶ Row与Y轴相对应
- ▶ Col与X轴相对应

新的内核函数详解 (3)



```
7.  float Pvalue = 0;  
    // Loop over the Md and Nd tiles required to compute the Pd element  
8.  for (int m = 0; m < Width/TILE_WIDTH; ++m) {  
    .....  
    }  
15. Pd[Row*Width + Col] = Pvalue;
```

- ▶ 第7行中的Pvalue 是一个“局部变量”，通常存储在 GPU 速度最快的寄存器中

✎备注：每个 SM 的寄存器总数有限，每个线程只能使用少量寄存器

✎寄存器溢出：如果线程有本地变量过多，寄存器被用完后会使用全局内存存放一些本地变量

- ▶ 第15行，将结果 Pvalue 复制到全局内存Pd[]。

新的内核函数详解 (4)



```
8.  for (int m = 0; m < Width/TILE_WIDTH; ++m) {  
    // Collaborative loading of Md and Nd tiles into shared memory  
9.      Mds[ty][tx] = Md[Row*Width + (m*TILE_WIDTH + tx)];  
10.     Nds[ty][tx] = Nd[(m*TILE_WIDTH + ty)*Width + Col];  
11.     __syncthreads();  
    .....  
}
```

- ▶ 计算Pvalue的过程被划分为Width/TILE_WIDTH个子阶段
- ▶ 第9行和第10行，每个线程将数据从全局内存 Md[] 和 Nd[] 加载到共享内存 Mds[][] 和 Nds[][] 中
 - ✎ 每个线程块的共享内存 Mds[][]和Nds[][]将被完全填满
- ▶ 第11行，__syncthreads() 为屏障函数，用于确保线程块中的所有线程都已完成数据加载
 - ✎ 线程块中有多个线程且可能在不同的时间被调度，因此不同的线程可能在不同的时间完成第10行



新的内核函数详解 (5)

```
8.  for (int m = 0; m < Width/TILE_WIDTH; ++m) {  
    .....  
12.      for (int k = 0; k < TILE_WIDTH; ++k)  
13.          Pvalue += Mds[ty][k] * Nds[k][tx];  
14.      __syncthreads();  
    }
```

- ▶ 共享内存中的数据准备就绪后，第12-13行将被执行

✎ 运算所需的数据都在共享内存中，因此数据访问效率比旧版本更高

- ▶ 第 14 行确保线程块中的所有线程都完成当前阶段的计算

✎ 共享内存可在下一阶段重复使用

新内核的运行结果

▶ 运行于 GTX640 上

Matrix Size	N = 512	N = 1024	N = 1536	N = 2048
全局内存版本	18.5ms	141ms	470.7ms	1112ms
共享内存版本	7ms	50.7ms	172.3ms	394.6ms
Speedup	2.64	2.78	2.73	2.82

展开循环

```
#pragma unroll  
for (int k = 0; k < TILE_WIDTH; ++k)  
    Pvalue += Mds[ty][k] * Nds[k][tx];
```

- ▶ 不展开(unrolling)会导致每次乘法和加法都需要更多额外指令
- ▶ 通过展开, “for循环” 被 “TILE_WIDTH” 句代码替代

Matrix Size	N = 512	N = 1024	N = 1536	N = 2048
Without unrolling	7ms	50.7ms	172.3ms	394.6ms
With unrolling	5.5ms	39.1ms	131.9ms	303.3ms
Speedup	1.27	1.30	1.31	1.30

内存会限制并行性

- ▶ 寄存器和共享内存速度快但大小有限
- ▶ 如果单个线程使用大量寄存器和/或共享内存，“活动线程”的数量会减少，从而可能导致性能低下
- ▶ 例如，在计算能力为2.x的GPU中，每个 SM 有 32768 个寄存器，最多可支持 1536 个活动线程
 - ✎ 如果单个线程使用 21 个寄存器，则最多可有1536 个活动线程 ($1536 \times 21 < 32768$)
 - ✎ 如果单个线程使用 22 个寄存器，那么最多只能有1489个线程。如果线程块大小为 512，那么只能有2个活动线程块，即 1024 个活动线程

硬件限制

计算能力	1.1	1.2, 1.3	2.x	3.0	3.5	5.0	7.0
每个SM中可使用的32-bit寄存器数	8K	16K	32K	64K	64K	64K	64K
每个线程可使用的32-bit寄存器的上限	128	128	63	63	255	255	255
SM每个SM可使用的共享内存上限	16KB	16KB	48KB	48KB	48KB	64KB	96KB
每个线程块可使用的共享内存上限	16KB	16KB	48KB	48KB	48KB	48KB	48KB

▶ GPU 由多种硬件资源组成，每种资源的容量各不相同

✎ 算术单元：

☞ 吞吐量：单位时间内可进行多少次计算？

☞ 延迟：单次计算需要多长时间？

✎ 内存系统：

☞ 带宽/吞吐量：单位时间可传输多少字节？

☞ 延迟：单次数据访问需要多长时间？

▶ CUDA 内核的速度很大程度上取决于 GPU 设备的资源限制

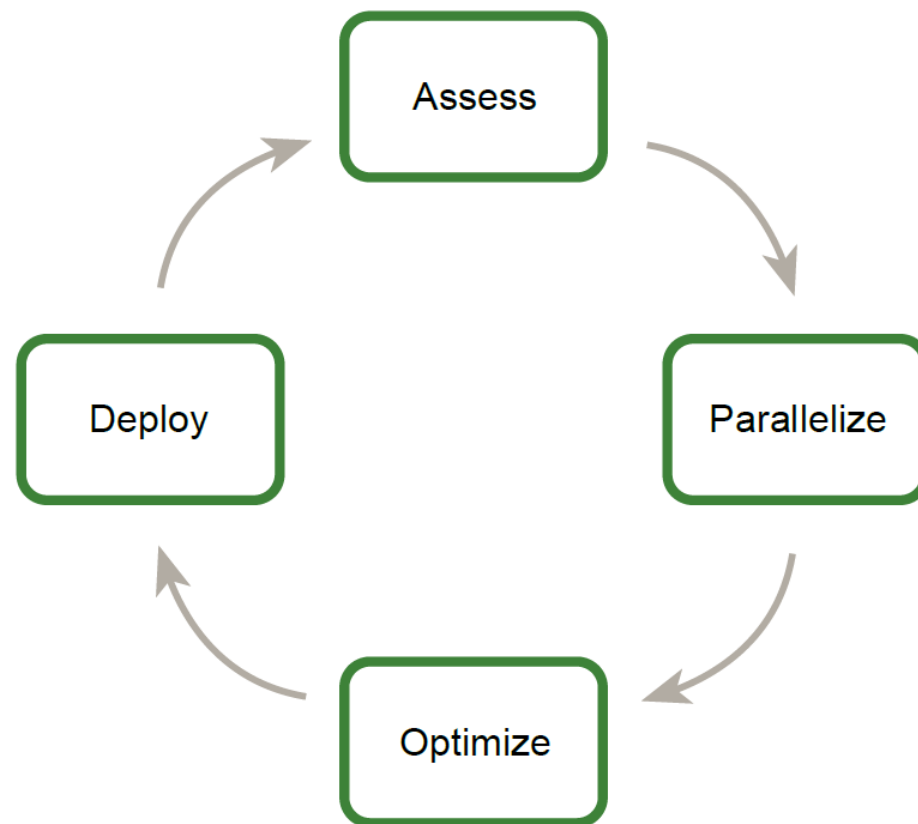
▶ 了解目标程序在GPU 中的主要资源限制类型后才能开发出高效的 CUDA 程序

✎ 例如，在第二部分中，“全局内存”可能是瓶颈，可使用“共享内存”改善

▶ APOD: 建模，并行，优化和部署(Assess, Parallelize, Optimize, and Deploy)

APOD 设计周期

- ▶ APOD: 分析, 并行, 优化和部署(Assess, Parallelize, Optimize, and Deploy)



I. 分析

▶ 第一步是对应用程序进行分析，找出热点，即大部分执行时间都来自哪些代码的执行

✎ 通过理论分析：对应用程序的每个主要步骤进行时间复杂性分析

✎ 通过性能分析实验：性能分析程序是测量函数调用或指令耗时的软件工具。例如

✎ 面向Linux开发的GNU gprof

✎ https://www.cs.utah.edu/dept/old/texinfo/as/gprof_toc.html

✎ 面向CUDA 开发：

✎ 面向Linux操作系统的nvprof

✎ 面向Windows操作系统的Nvidia Visual Profiler

✎ Ref: <http://docs.nvidia.com/cuda/profiler-users-guide>



\$ nvprof ./matrix

==19300== NVPROF is profiling process 19300, command: ./matrix

CUDA initialized.

GPU done!

Elapsed Time by event: 16.180511 ms

GPU (shared memory) done!

Elapsed Time by event: 6.920672 ms

Elapsed Time by CPU: 1458.828125 ms

==19300== Profiling application: ./matrix

==19300== Profiling result:

Time(%)	Time	Calls	Avg	Min	Max	Name
59.04%	13.146ms	1	13.146ms	13.146ms	13.146ms	MatrixMulKernel(float*, float*, float*, int)
20.27%	4.5129ms	1	4.5129ms	4.5129ms	4.5129ms	SharedMatrixMulKernel(float*, float*, float*, int)
11.90%	2.6503ms	4	662.59us	659.75us	669.44us	[CUDA memcpy HtoD]
8.79%	1.9560ms	2	978.02us	749.31us	1.2067ms	[CUDA memcpy DtoH]

==19300== API calls:

Time(%)	Time	Calls	Avg	Min	Max	Name
77.83%	228.31ms	2	114.16ms	1.7350us	228.31ms	cudaEventCreate
13.99%	41.047ms	1	41.047ms	41.047ms	41.047ms	cudaDeviceReset
7.85%	23.016ms	6	3.8360ms	588.40us	14.740ms	cudaMemcpy
0.10%	302.22us	166	1.8200us	164ns	61.213us	cuDeviceGetAttribute
0.09%	257.85us	3	85.949us	63.839us	128.96us	cudaMalloc
0.06%	186.60us	3	62.198us	46.782us	92.858us	cudaFree
0.02%	52.845us	2	26.422us	26.248us	26.597us	cudaLaunch
0.02%	46.251us	2	23.125us	22.843us	23.408us	cuDeviceTotalMem
0.01%	34.460us	2	17.230us	16.290us	18.170us	cuDeviceGetName
0.01%	33.645us	6	5.6070us	1.6740us	14.791us	cudaEventRecord

.....

II. 并行化

► 确定热点后，尝试将其并行化

✎ 可以使用现有的并行库，如 cuBLAS、cuFFT 等。

✎ 或者，也可以自行设计并行算法

III. 优化

- ▶ 如何在特定硬件上实现并行算法，以达到最佳性能？
- ▶ 程序优化是一项具有挑战性的任务
 - ✎ 充分了解应用程序
 - ✎ 充分了解目标硬件
 - ✎ 进行多轮 APOD
 - ✎ 优化后性能可提高 10 倍

▶ 内核(kernel)优化

- ✎ 内核配置: # 线程块数, # 线程数/线程块
- ✎ 控制流
- ✎ 全局内存访问
- ✎ 共享内存访问
- ✎ 指令优化

▶ 优化 CPU 与 GPU 的交互

- ✎ 最大化 PCI-e 吞吐量
- ✎ 内核执行与内存数据拷贝重叠

内核配置:

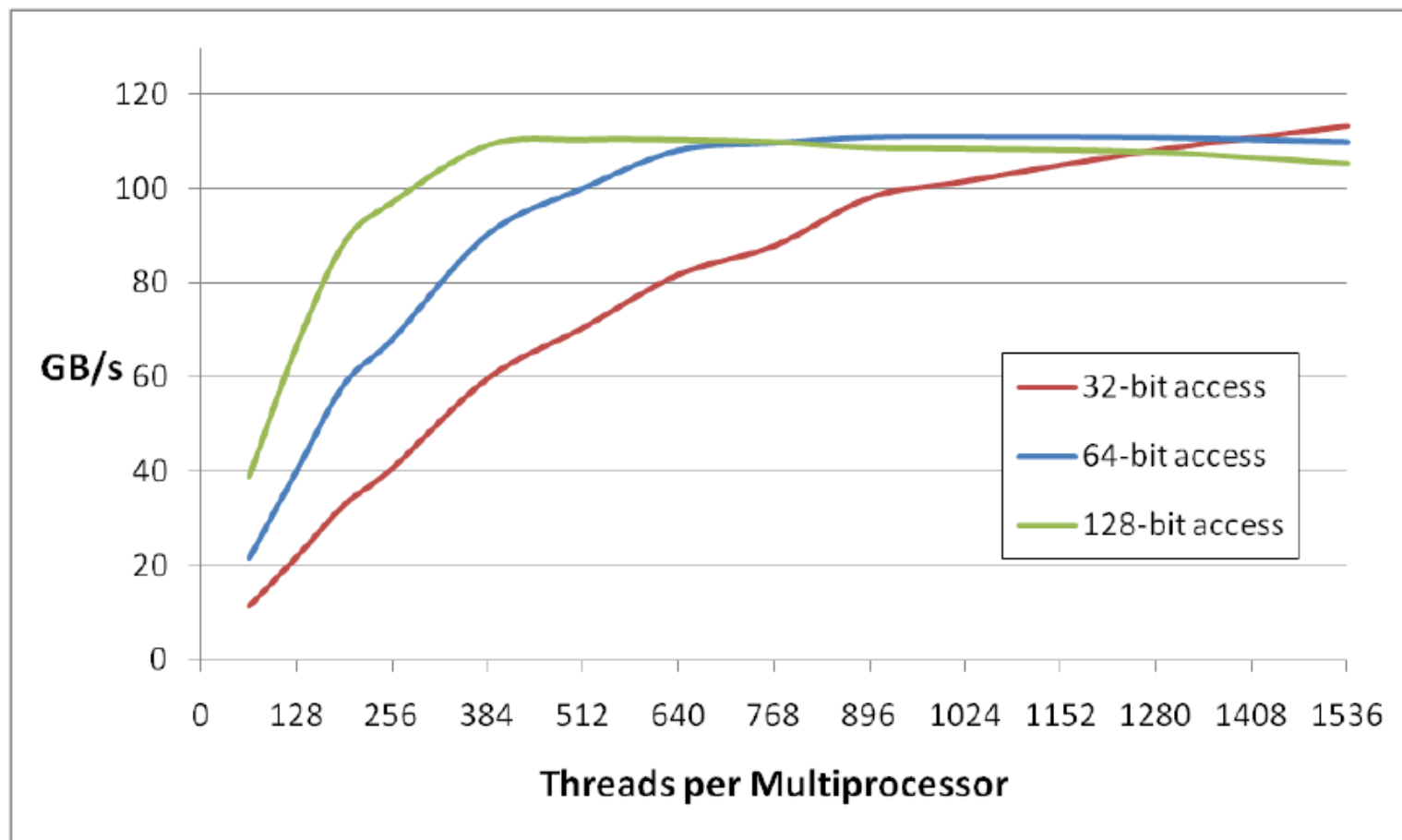


- ▶ 要启动多少线程块/线程?
- ▶ 理解的关键点:
 - ✎ 指令按发射
 - ✎ 当其中一个操作数尚未准备好时，线程会暂停
 - ✎ 通过切换线程能够隐藏延迟
 - ☞ 全局内存延迟：几百个时钟周期
 - ☞ 算术计算延迟：18-22 个周期
- ▶ 结论:
 - ✎ 每个 SM 需要足够多Warps隐藏延迟

全局内存带宽



Tesla C2050, 开启ECC后的理论带宽: ~120 GB/s



建议



▶ 需要足够的Warps保持 GPU 忙碌

- ✎ 线程块数量应远远大于 SM 数量

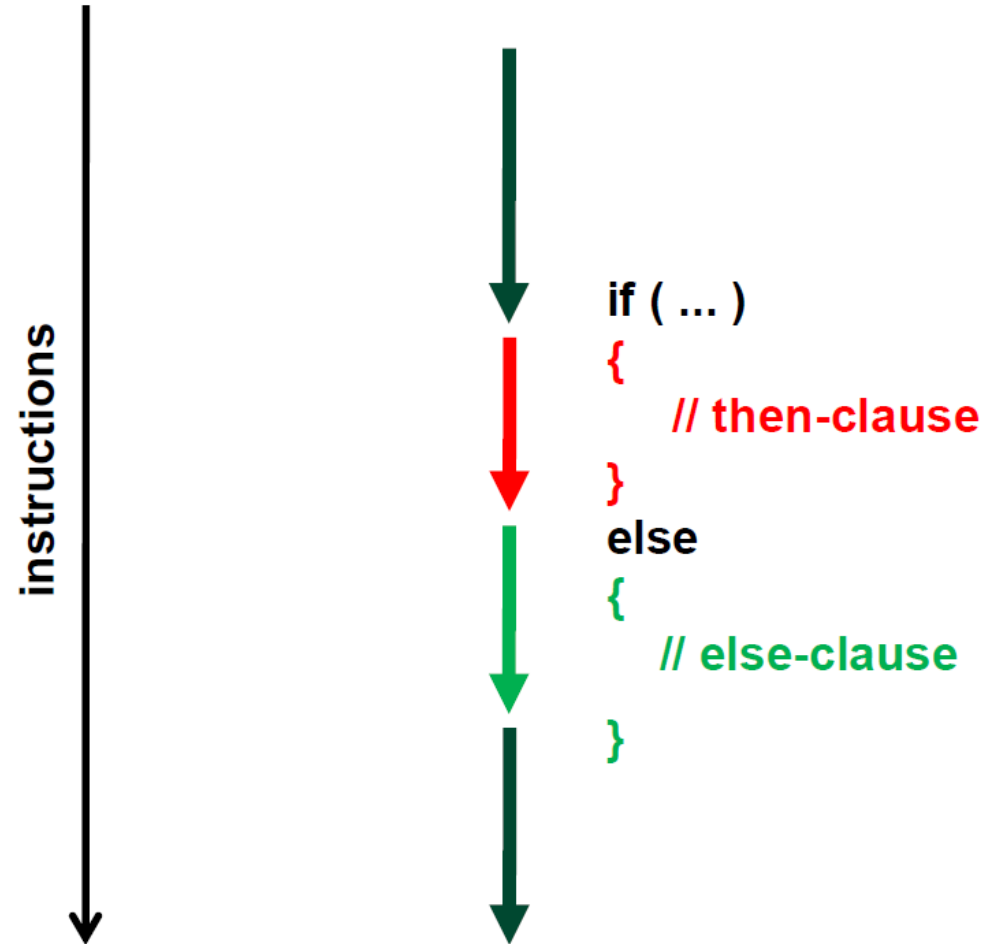
- ✎ 通常情况下，每个 SM 至少需要有 16 个活动Warps

▶ 线程块配置

- ✎ 每个线程块的线程数应该是Warp大小（32）的倍数

- ✎ 通常每块 128-256 个线程即可

控制流



► 分支的主要性能问题是分歧(divergence)

✎ 单个warp内的各线程执行路径不同

✎ 不同的执行路径被序执行

☞ Warp中的线程走过的控制路径会被逐次遍历，直到没有控制路径为止

► 常见情况：当分支条件是线程 ID 的函数时，要避免分歧

✎ 有分歧的例子：

☞ `If (threadIdx.x > 2) { }`

☞ 上述语句为线程块中的线程创建了两种不同的控制路径

📖 分支粒度 < warp大小；以第一个warp为例，线程 0、1 和 2 的执行路径与Warp中的其他线程不同

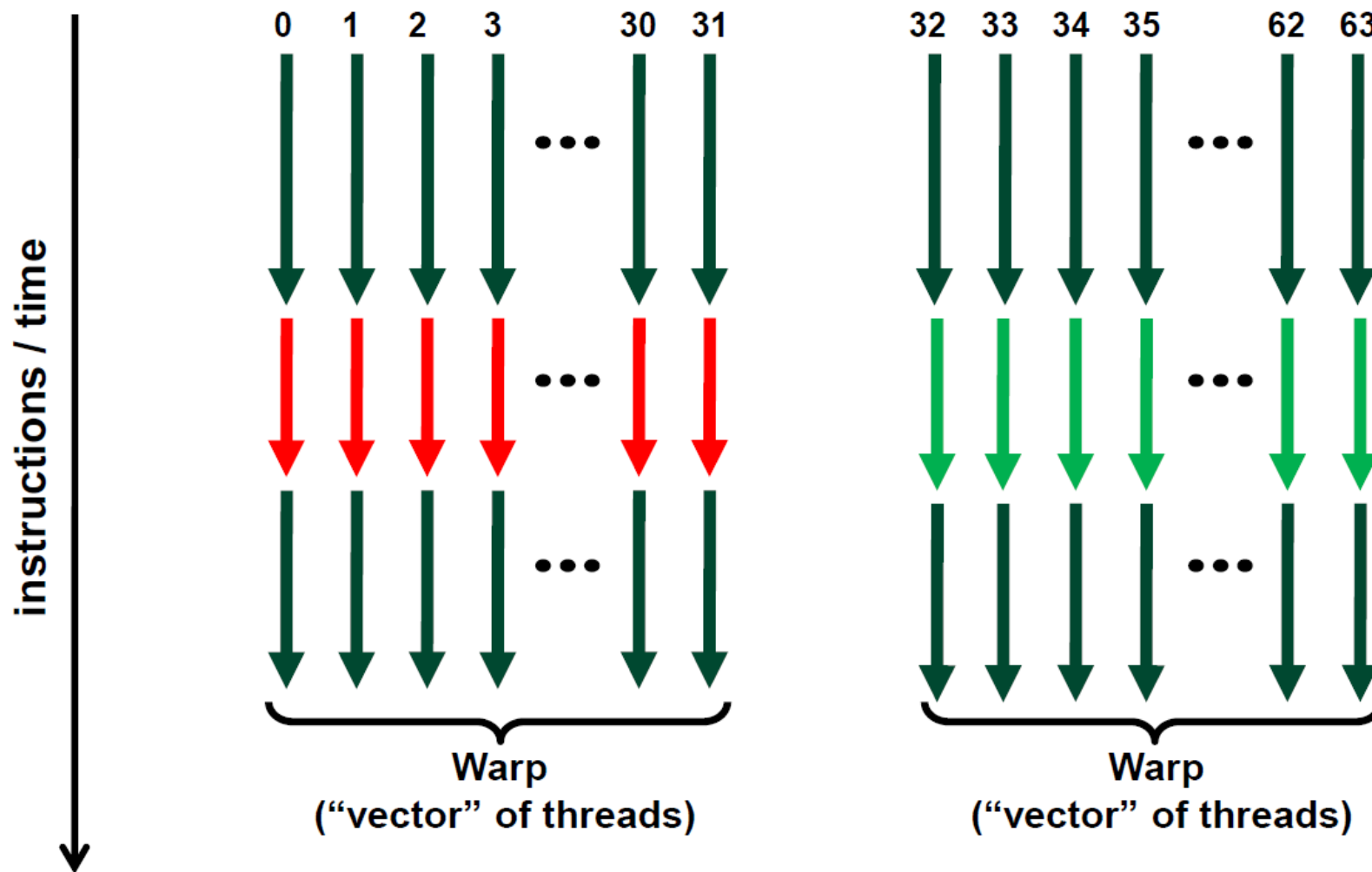
✎ 无分歧的例子：

☞ `If (threadIdx.x / WARP_SIZE > 2) { }`

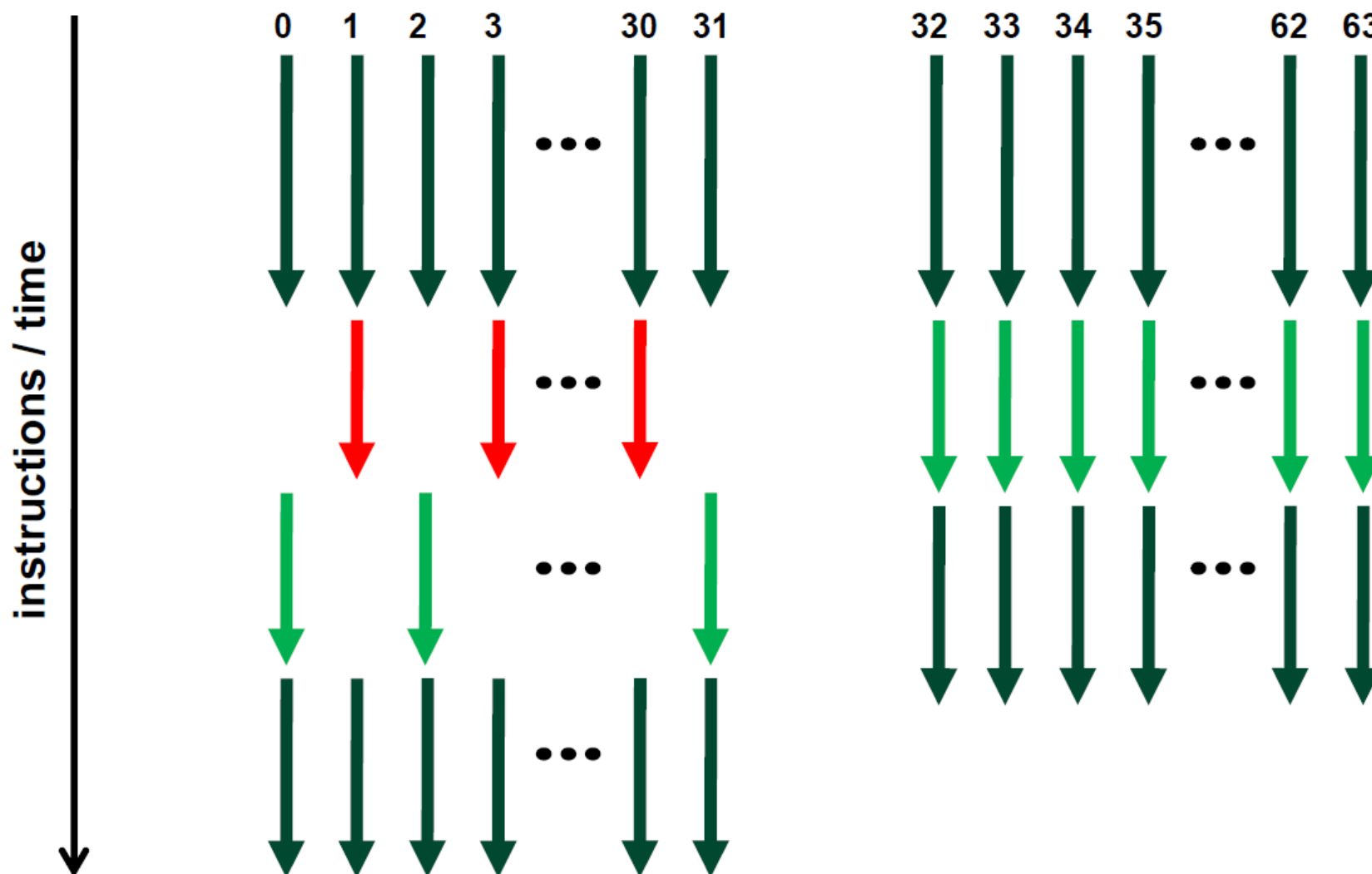
☞ 上述代码为线程块中的线程创建了两种不同的控制路径

☞ 分支粒度为warp大小的整数倍；任何给定warp中的所有线程都遵循相同的路径

Warp内的执行是连续的



Warp中的执行分歧



▶ 给定数值数组，并行将其“归约(reduce)”为单一数值

▶ 示例

✎ 求和归约：求数组中所有值的总和

✎ 最大化归约：求数组中所有数值的最大值

▶ 典型并行实现：

✎ 递归将 # 个线程减半，每个线程对两个值求和

✎ 处理 n 个元素需要 $\log(n)$ 步，共需 $n/2$ 个线程

矢量归约示例



▶ 假设使用共享内存进行就地归约

✎ 原始矢量位于设备全局内存

✎ 共享内存用于保存部分和矢量

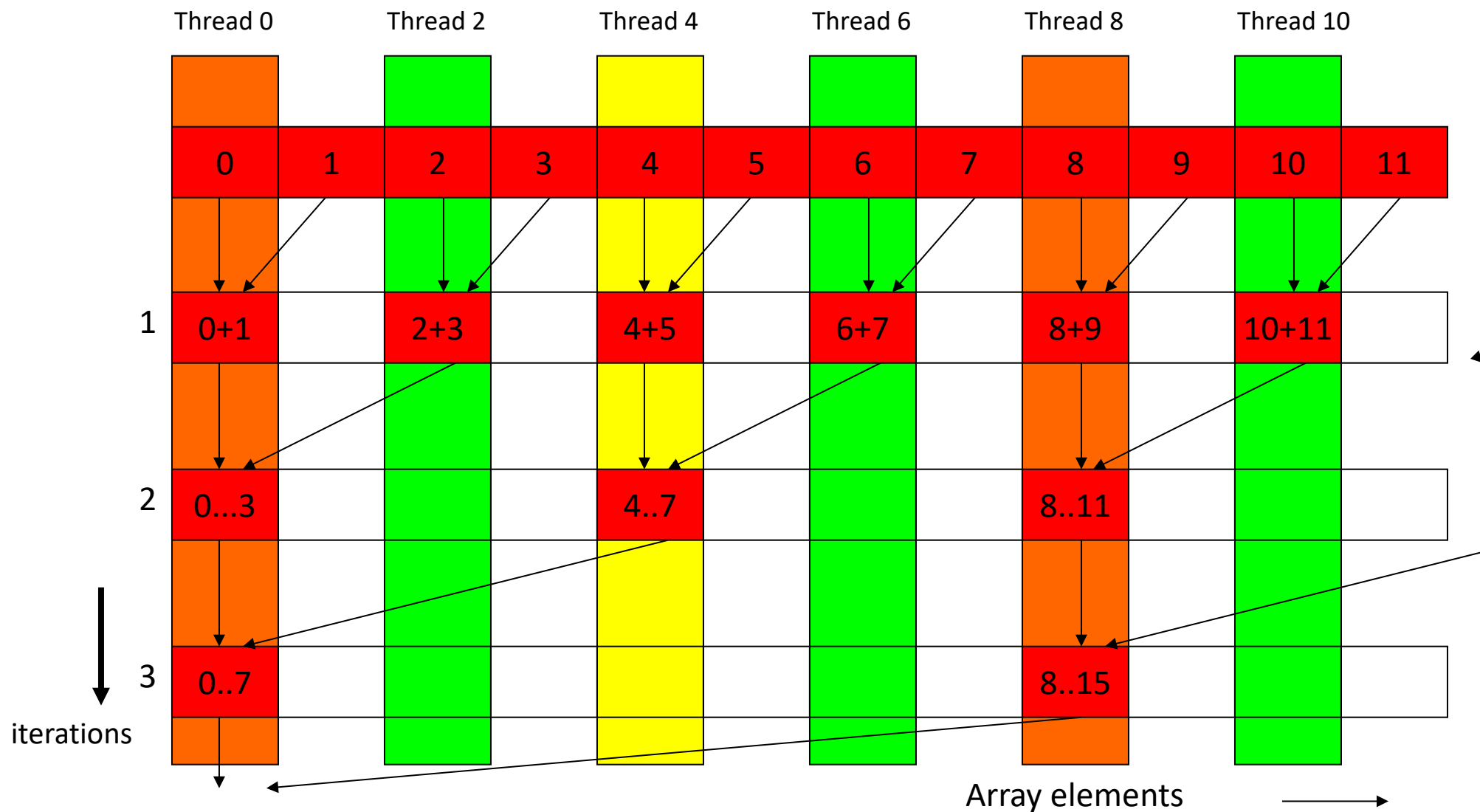
✎ 每次迭代都会使部分和矢量更接近最终结果

✎ 最终结果是元素 0 的值

- ▶ 假设已将数组加载到共享内存中

```
1 __shared__ float partialSum[];  
2  
3 unsigned int t = threadIdx.x;  
4 for (unsigned int stride = 1; stride < blockDim.x; stride *= 2)  
5 {  
6     __syncthreads();  
7     if (t % (2*stride) == 0)  
8         partialSum[t] += partialSum[t+stride];  
9 }
```

利用分支分歧进行矢量归约



- ▶ 在每次迭代中，每个 warp 将依次遍历两条控制流路径
 - ✎ 执行加法运算的线程和不执行加法运算的线程
 - ✎ 取决于分歧的执行情况, 不执行加法运算的线程可能会耗费额外的周期
- ▶ 任何时候执行的线程都不会超过一半
 - ✎ 所有奇数索引线程从一开始就被禁用！
 - ✎ 随着时间的推移，所有warp中，平均只有不到 1/4 的线程被激活
 - ✎ 第 5 次迭代后，每个线程块中都存在完全被禁用的warp，资源利用率很低，但不会存在分歧
 - ☞ 持续一段时间后，最多再进行 4 次迭代（ $512/32=16=2^4$ ），每次迭代只会激活一个线程，直到所有warp运行结束

实现中的不足



- ▶ 假设已经将数组加载到共享内存中

```
1 __shared__ float partialSum[];  
2  
3 unsigned int t = threadIdx.x;  
4 for (unsigned int stride = 1; stride < blockDim.x; stride *= 2)  
5 {  
6     __syncthreads();  
7     if (t % (2*stride) == 0)  
8         partialSum[t] += partialSum[t+stride];  
9 }
```

BAD:交错分支决策
导致的分歧

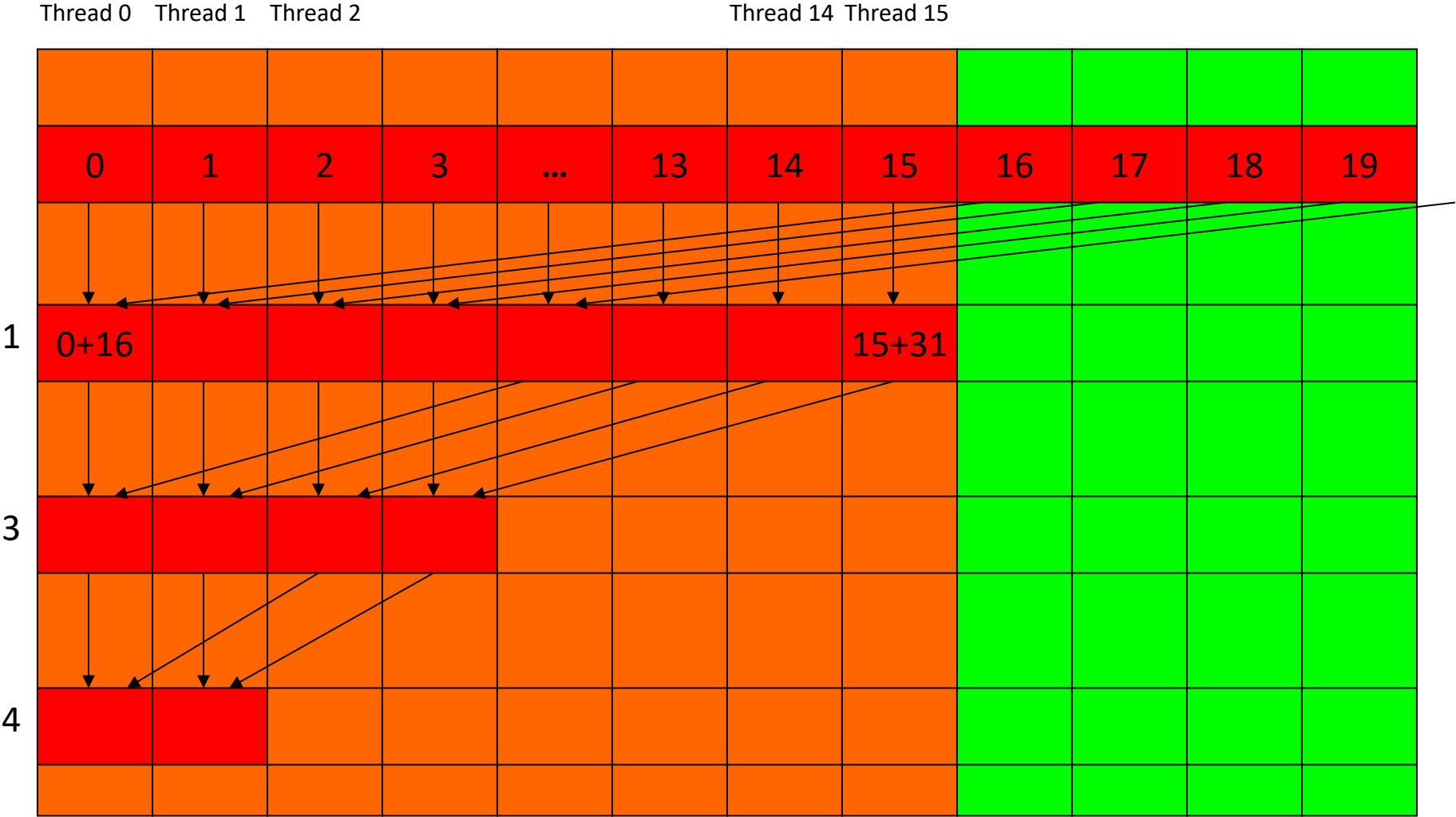
更好地实现方法



- ▶ 假设已经将数组加载到共享内存中

```
1 __shared__ float partialSum[];
2
3 unsigned int t = threadIdx.x;
4 for (unsigned int stride = blockDim.x; stride > 1; stride >> 1)
5 {
6     __syncthreads();
7     if (t < stride)
8         partialSum[t] += partialSum[t+stride];
9 }
```

部分和数<16 之前无分歧



CUDA 设备内存空间：回顾

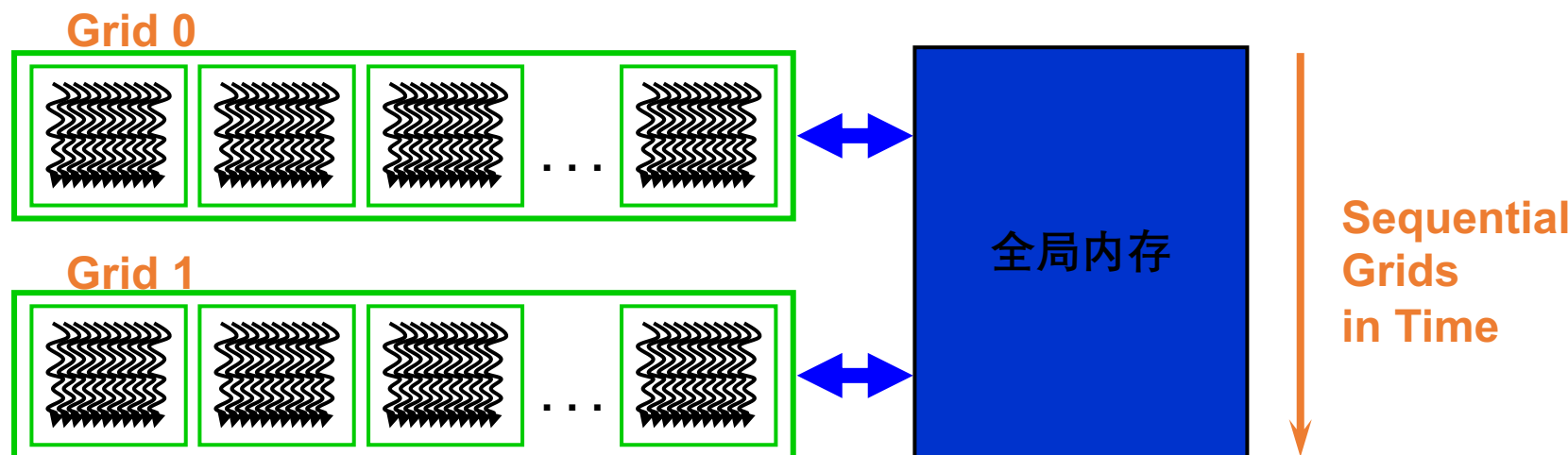
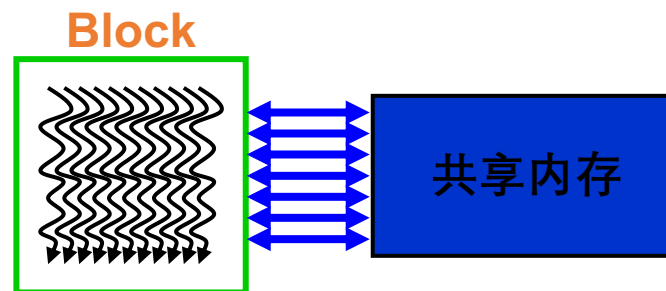
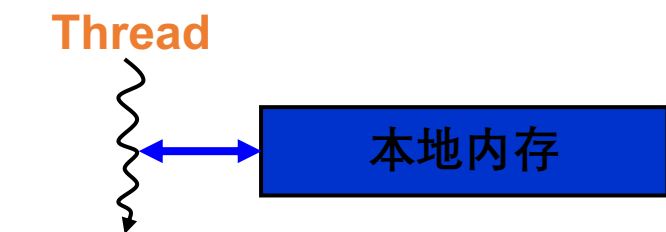
▶ 每个线程都可以：

- ✍ 读/写线程寄存器
- ✍ 读/写线程本地内存
- ✍ 读/写所属线程块共享内存
- ✍ 读/写所属网格全局内存
- ✍ 读所属网格常量内存
- ✍ 读所属网格纹理内存

声明	存储
<code>int v</code>	寄存器
<code>int vArray[10]</code>	本地内存
<code>__shared__ int sharedV</code>	共享内存
<code>__device__ int globalV; or cudaMalloc()</code>	全局内存
<code>__constant__ int constantV[10]; and cudaMemcpyToSymbol(...);</code>	常量内存
<code>cudaBindTexture2D struct textureReference</code>	纹理内存

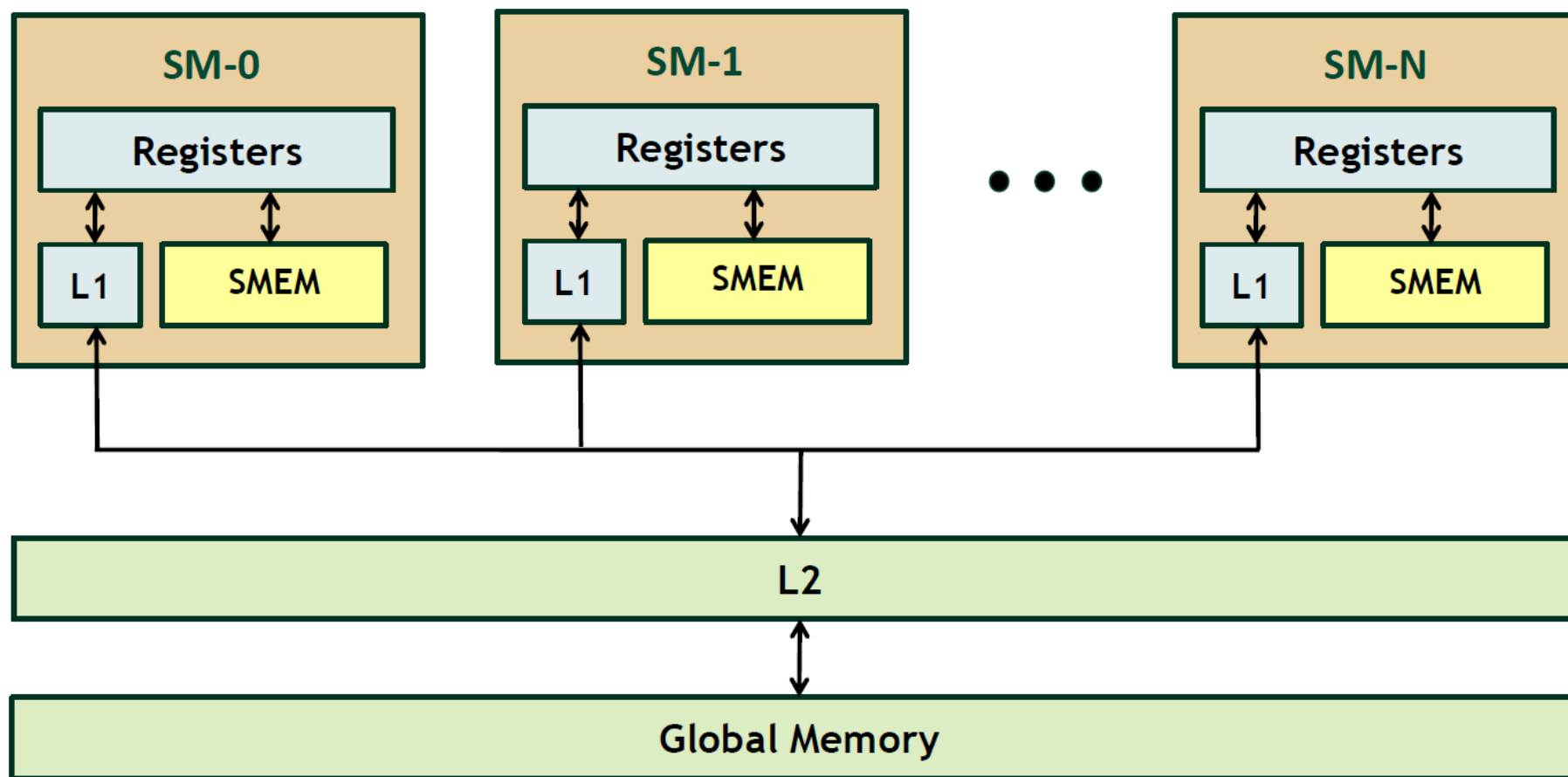
Memory	Location on/off chip	Cached	Access	Scope	Lifetime
Register	On	n/a	R/W	1 thread	Thread
Local	Off	Yes ^{††}	R/W	1 thread	Thread
Shared	On	n/a	R/W	All threads in block	Block
Global	Off	[†]	R/W	All threads + host	Host allocation
Constant	Off	Yes	R	All threads + host	Host allocation
Texture	Off	Yes	R	All threads + host	Host allocation
[†] Cached in L1 and L2 by default on devices of compute capability 6.0 and 7.x; cached only in L2 by default on devices of lower compute capabilities, though some allow opt-in to caching in L1 as well via compilation flags.					
^{††} Cached in L1 and L2 by default except on devices of compute capability 5.x; devices of compute capability 5.x cache locals only in L2.					

并行内存共享



- ▶ 本地内存：按线程
 - ✎ 线程私有
 - ✎ 自动变量、寄存器溢出
- ▶ 共享内存：按块
 - ✎ 同一线程块内的线程共享
 - ✎ 线程间通信
- ▶ 全局内存：按应用程序
 - ✎ 所有线程共享
 - ✎ 网格间通信

GPU 内存层次结构



▶ 按warp（32 个线程）进行内存操作

✎ 一条指令，32 次数据访问

✎ 如何满足这 32 次数据访问请求？

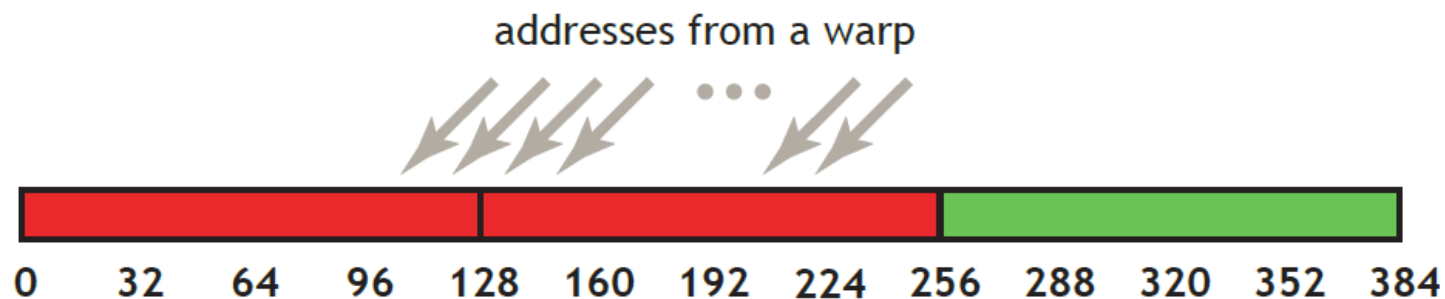
▶ 聚合：

✎ 当满足某些访问要求时，一个 warp 中（对于计算能力为 1.x 的设备为半个warp）线程的全局内存加载和存储可合并为一个事务

- ▶ 计算能力为2.x的设备的全局内存访问默认在 L1（128 字节大小的缓存行）中缓存
 - ✎ 基本的全局内存事务是读/写一个连续的 128 字节段
 - ✎ 来自 warp 的每个内存请求都被分解为缓存行请求
- ▶ 计算能力为 3.x 和 5.x 的设备的全局内存访问在 L2（32 字节段）中缓存
 - ✎ 基本的全局内存事务是读/写一个连续的 32 字节段

“不良”访问模式

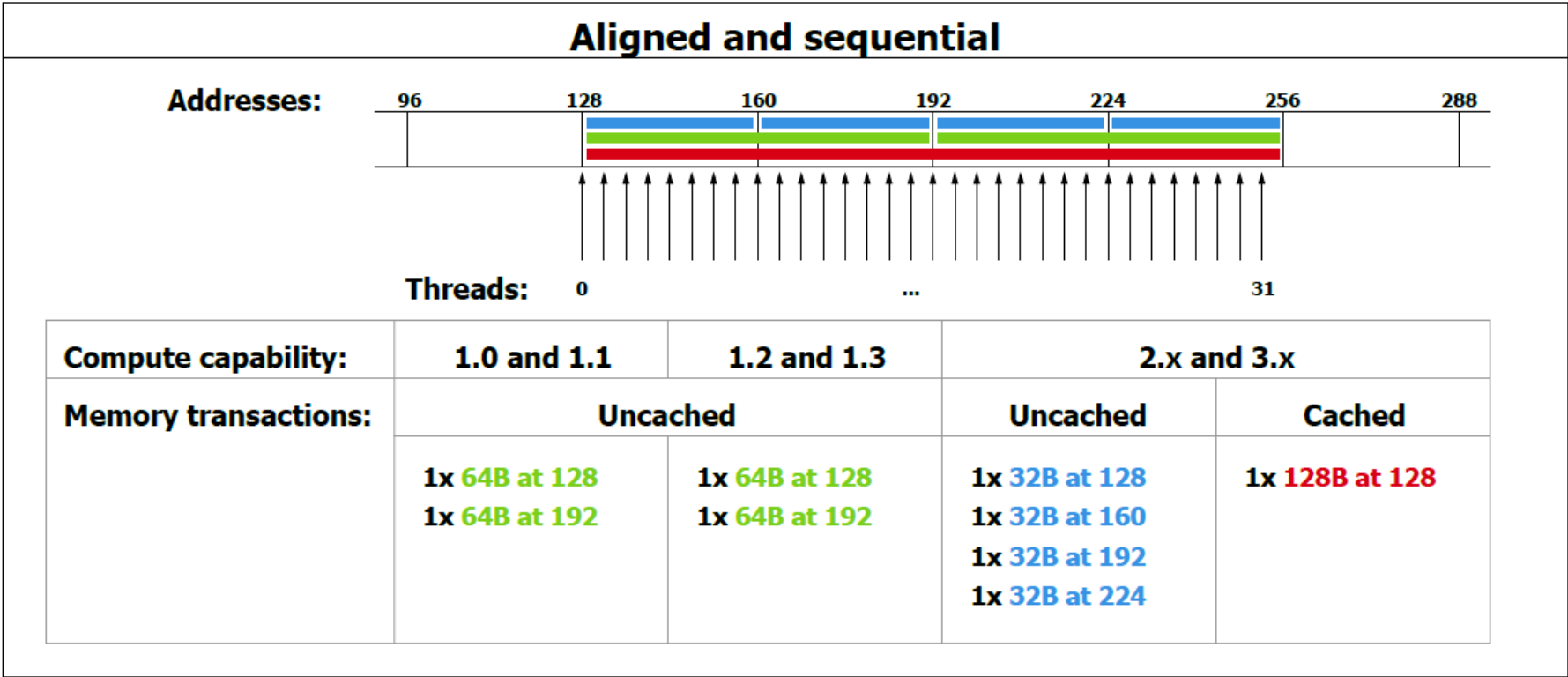
▶ 错位的数据访问



错位的数据按序访问会被拆分成两个128-byte的LI缓存行读取

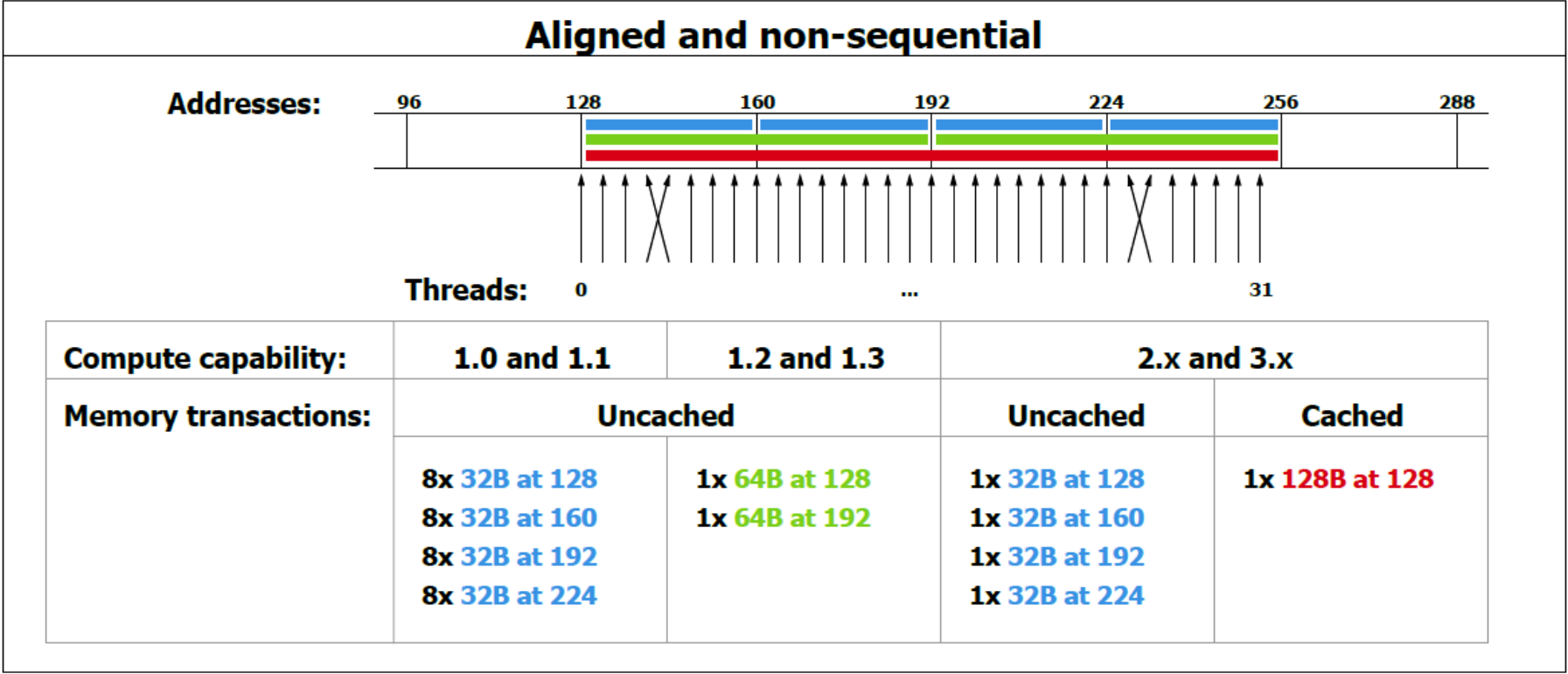
示例 1

4-Byte Word per Thread



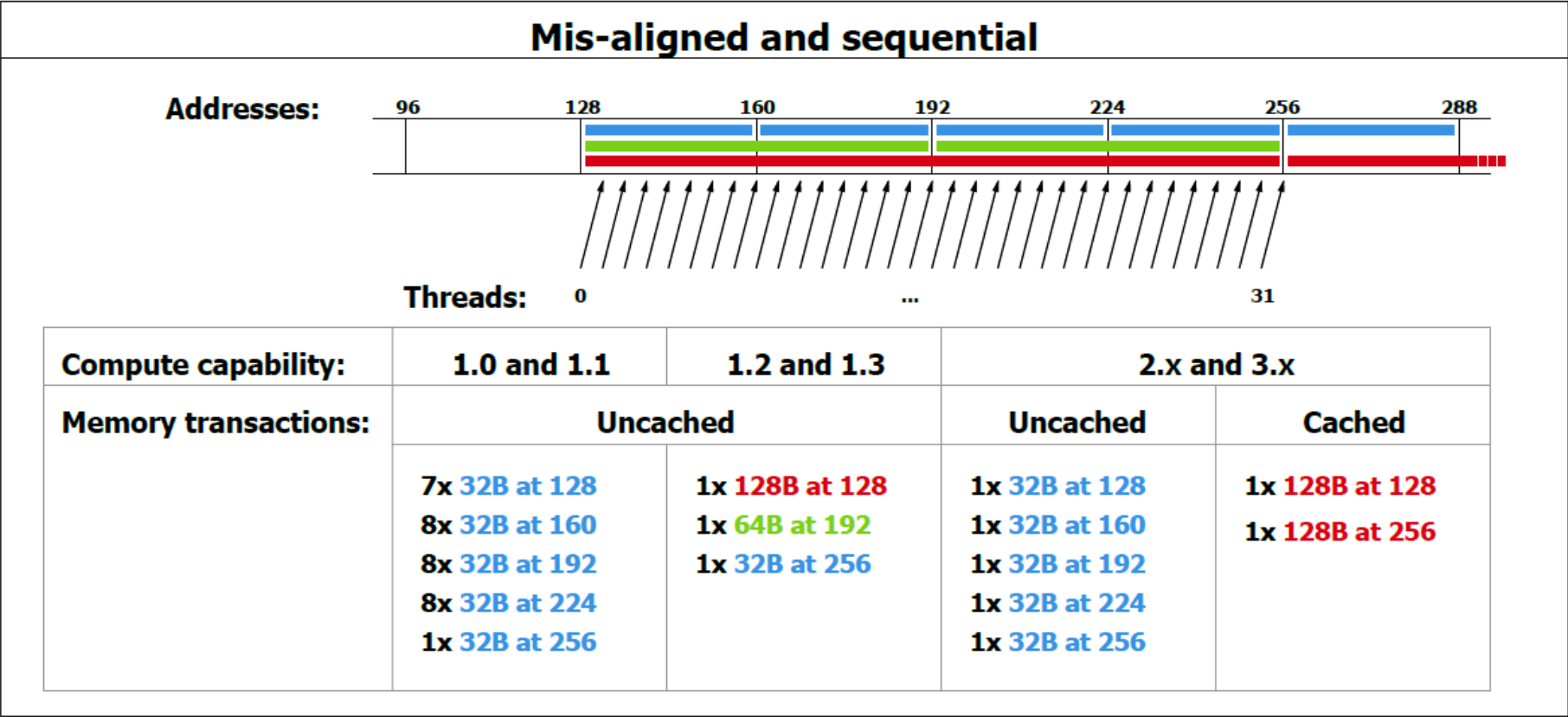
Cached: L1 (CC 2.x)
Uncached: no cache (CC 1.x), or L2 (CC 3.x or above)

示例 2



对于计算能力为1.0和1.1的设备，非顺序数据访问的性能非常差

示例 3

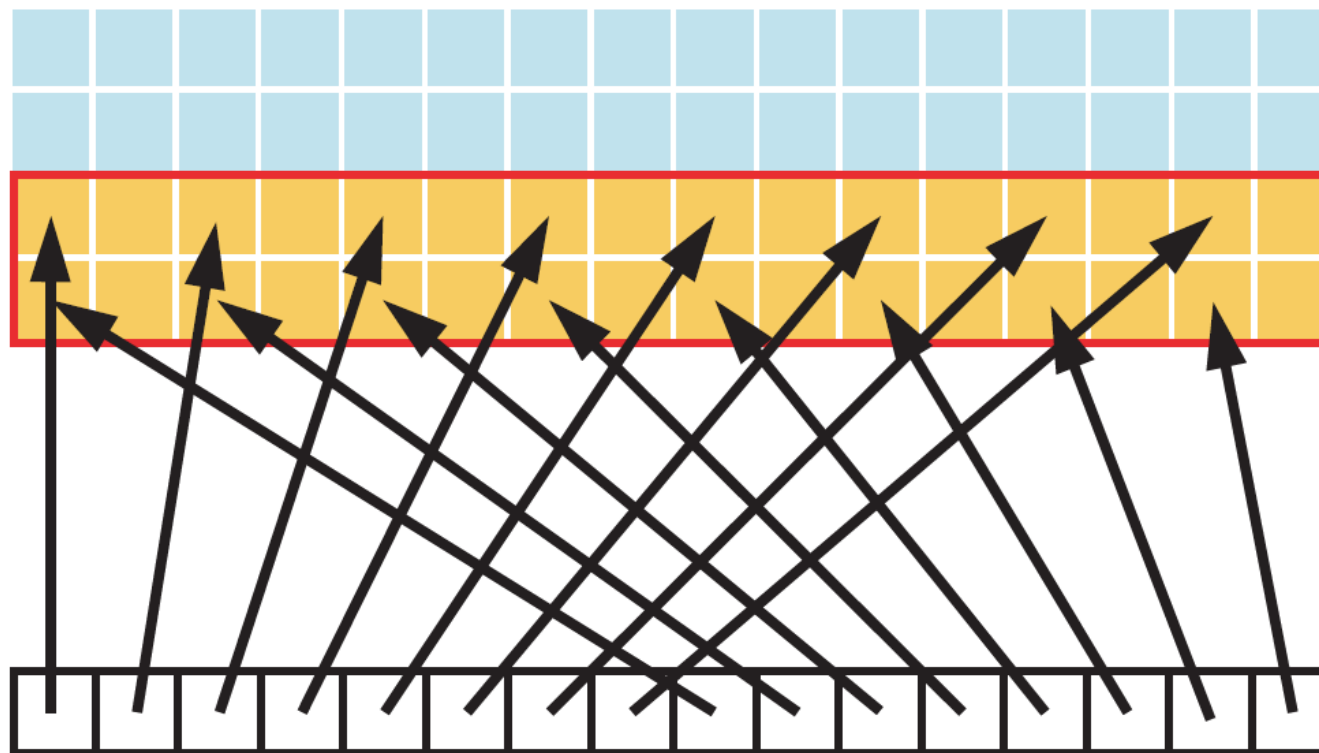


数据访问错位：起始地址不是 32 或 128 的倍数。缓存行中的某些字节无用。

示例 4：按步长访问



相邻线程访问内存的步长为 2



步长为 2 会导致加载/存储效率降低 50%，因为事务中有一半的元素没有被使用，浪费了带宽

▶ 共享内存位于芯片内

- ✎ 高带宽、低延迟

- ✎ 不如寄存器，但优于全局内存

▶ 用途：

- ✎ 线程块内的线程间通信

- ✎ 缓存数据以减少多余的全局内存访问

- ✎ 改进全局内存访问模式

▶ 硬件组织形式

- ✎ 划分为大小相等的内存模块，命名为bank

- ✎ 连续的 4 字节或 8 字节字属于不同的bank

共享内存的硬件组织形式



- ▶ 计算能力 1.x

- ✎ 16 个bank, 4 字节位宽

- ▶ 计算能力 2.x

- ✎ 32 个bank, 4 字节位宽

- ▶ 计算能力 3.x

- ✎ 32位模式: 32 个bank, 4 字节位宽

- ✎ 64位模式: 32 个bank, 8 字节位宽

- ▶ 计算能力 5.x

- ✎ 32 个bank, 4 字节位宽

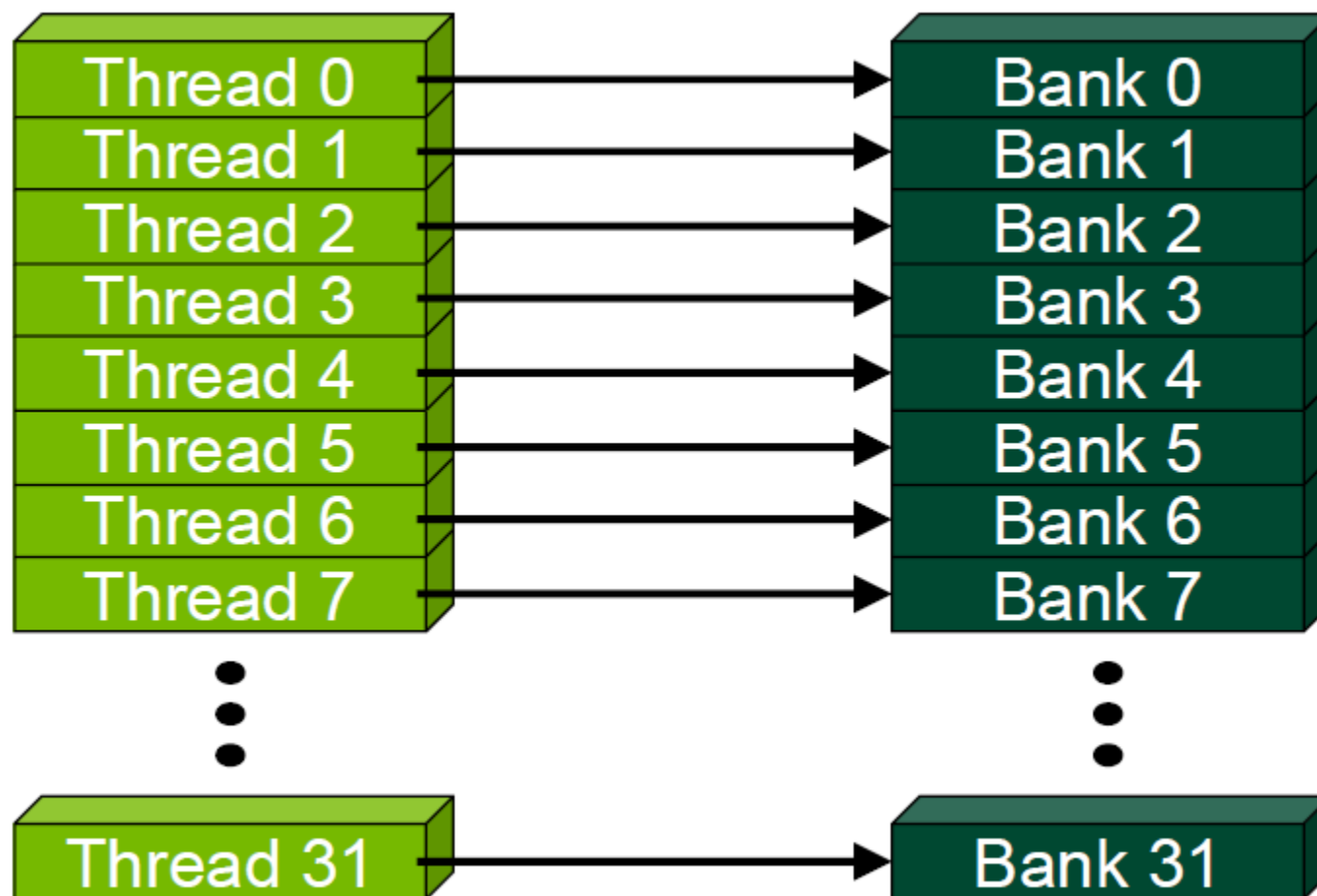
- ▶ 计算能力 7.x

- ✎ 32 个bank, 4 字节位宽

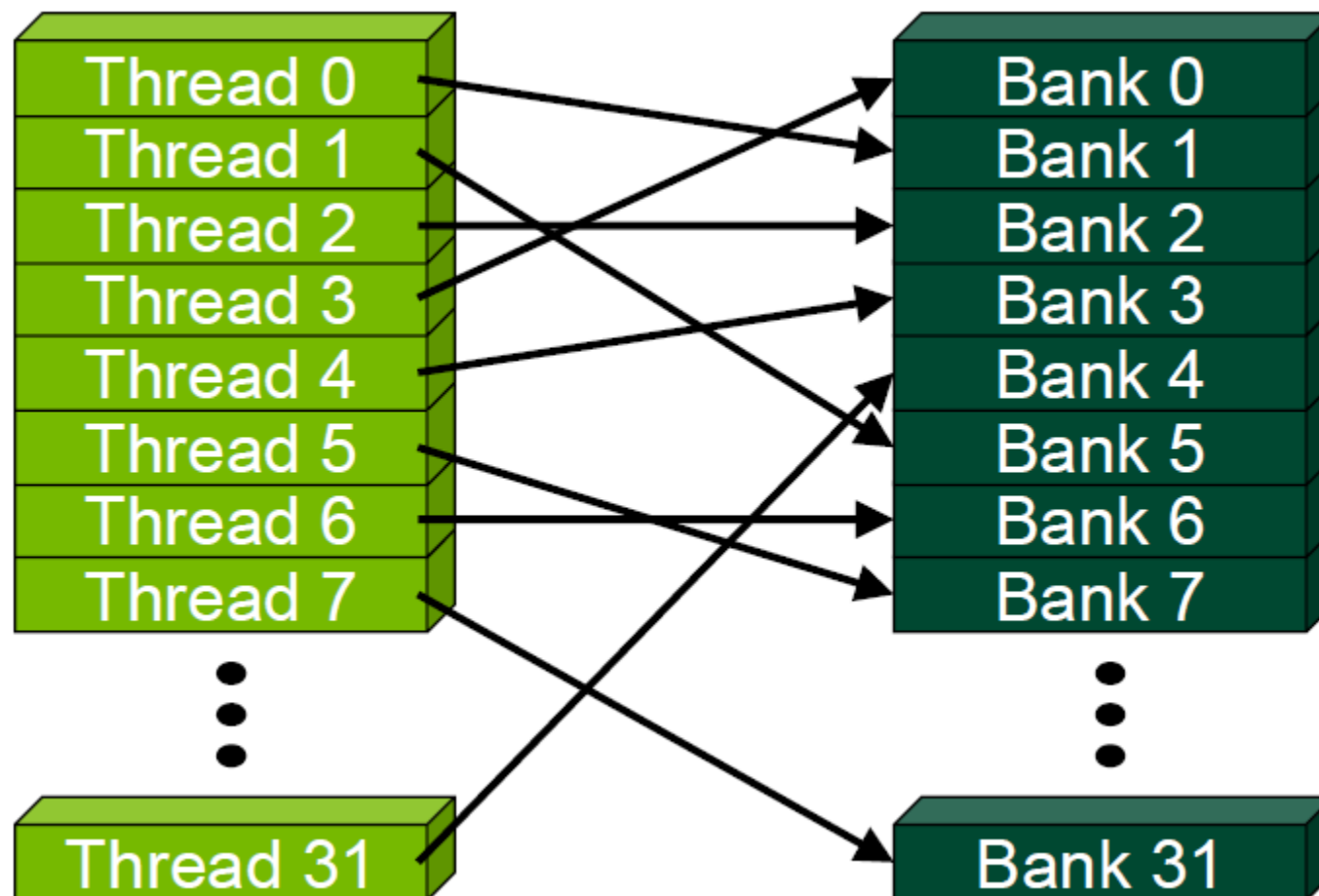
Bank访问冲突

- ▶ 共享内存访问按 32 个线程（warp）发射一次
- ▶ 串行化：bank冲突
 - ✎ 如果一个 warp 中有 n 个线程访问同一bank中的不同字，则 n 次共享内存访问将串行执行
 - ✎ 应尽可能避免bank冲突

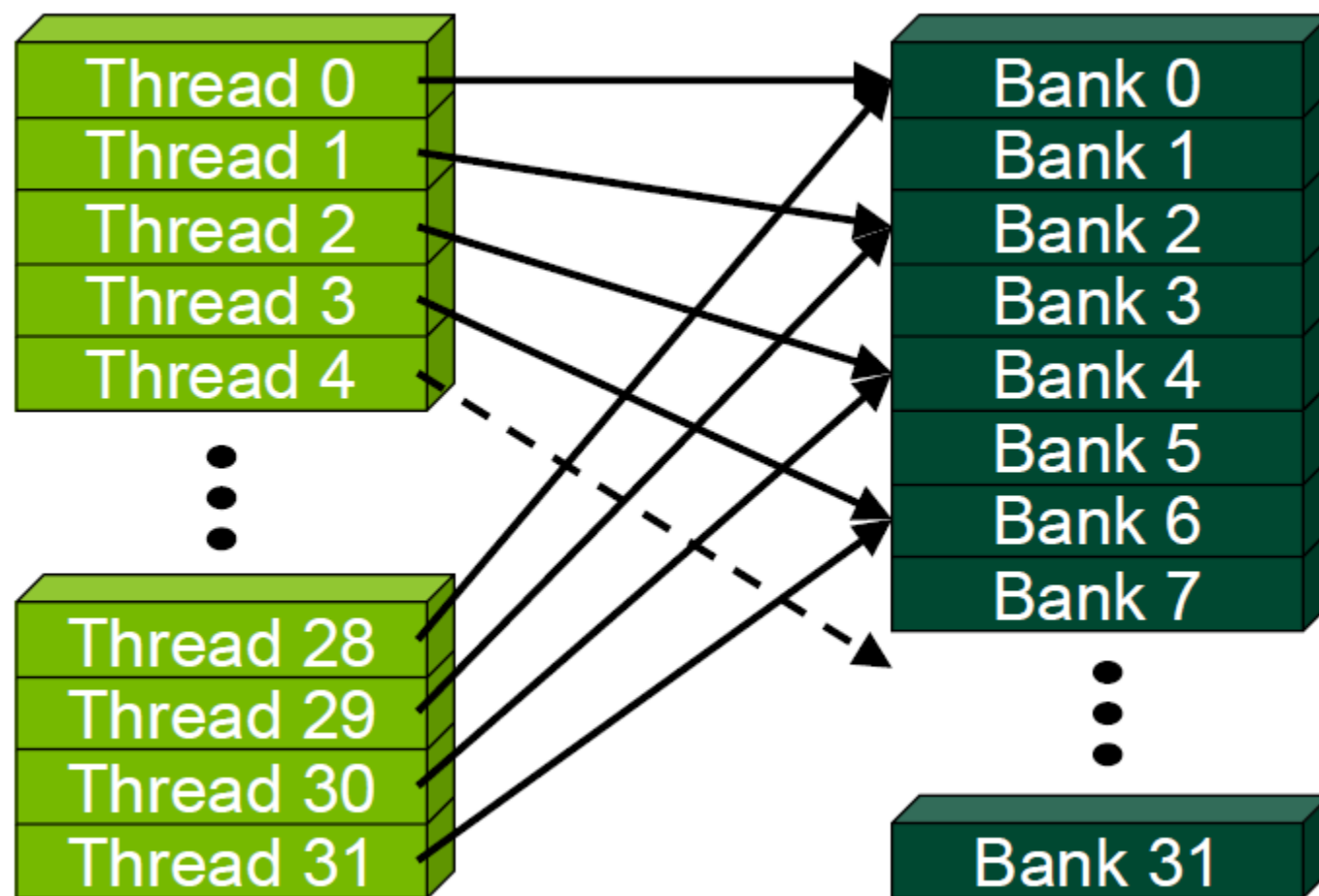
示例 1：无Bank冲突



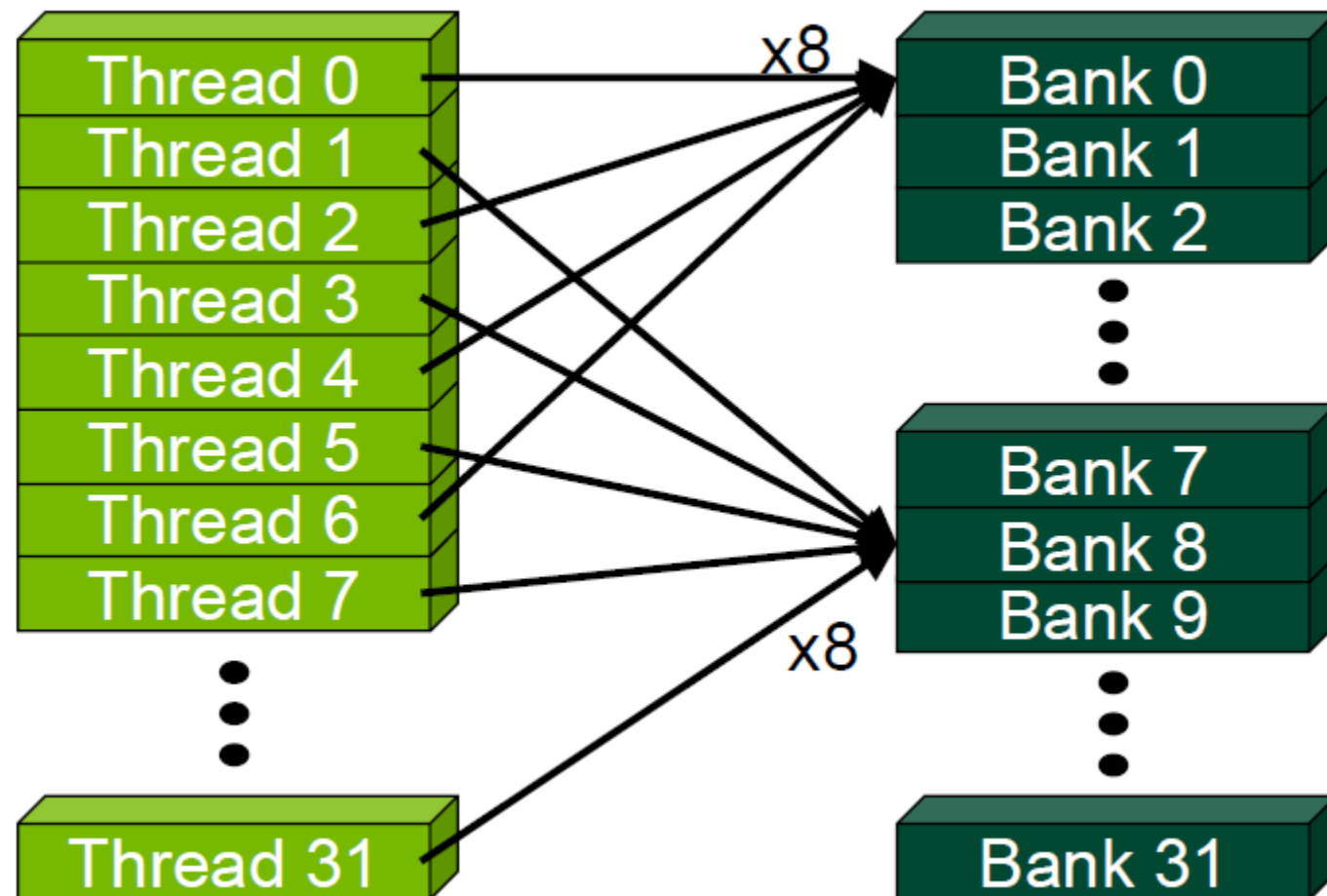
示例 2：无Bank冲突



示例 3：2-路Bank冲突

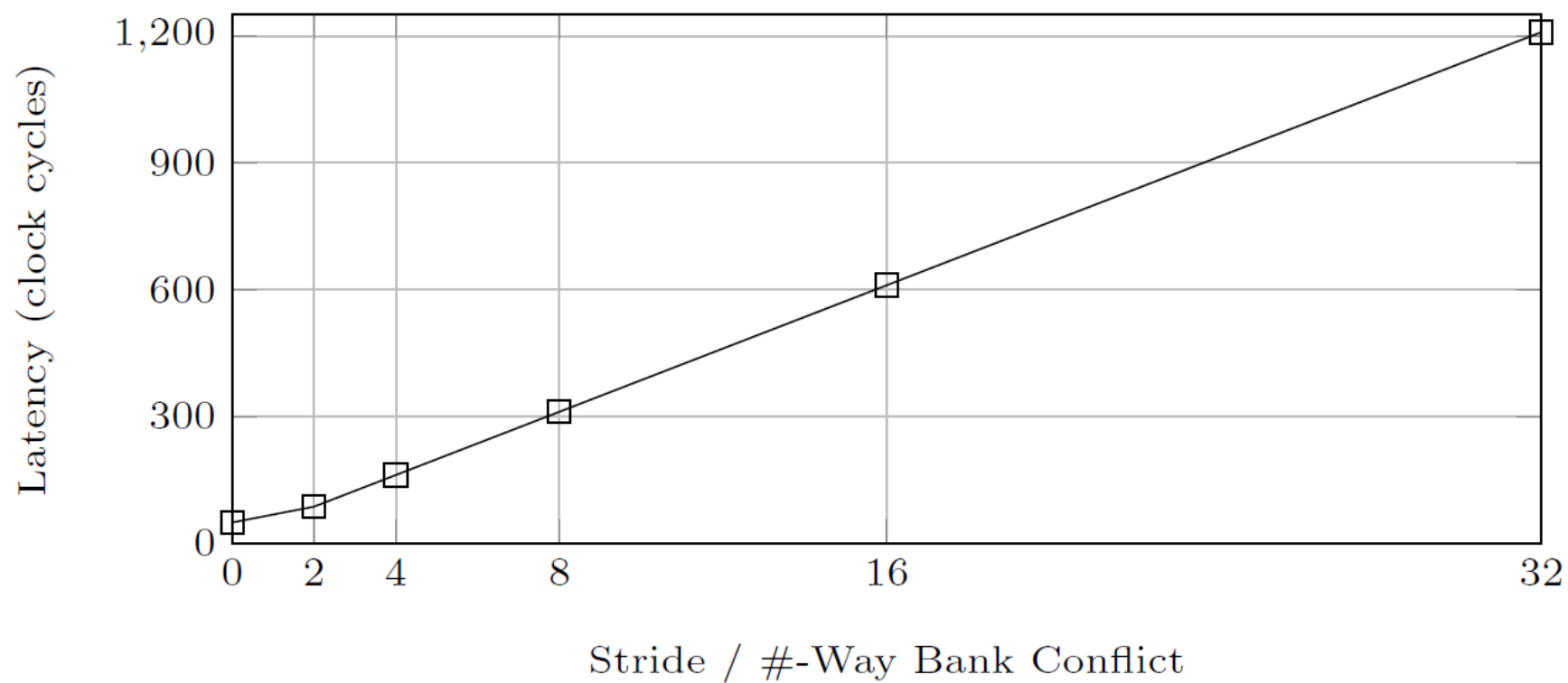


示例 4：8-路Bank冲突



Bank冲突的延迟

费米架构上的Bank冲突 (计算能力 2.x)

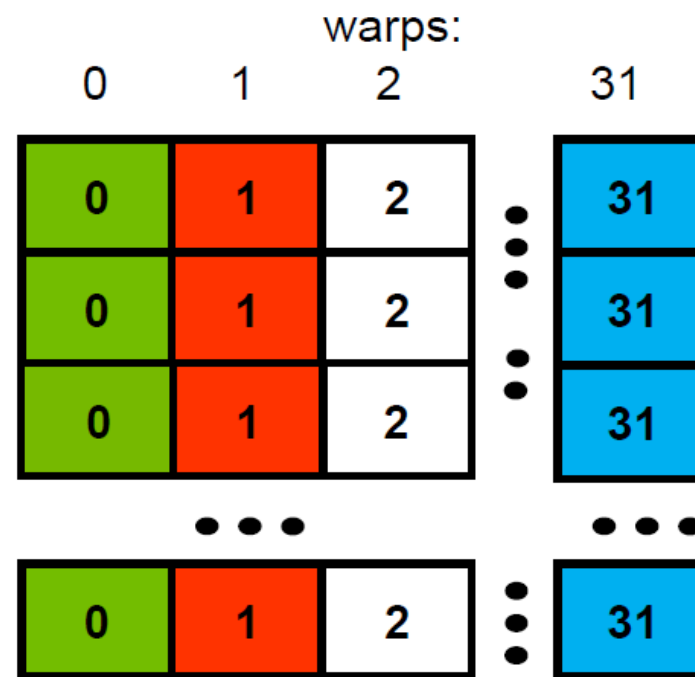


避免Bank冲突



- ▶ 32x32 共享内存数组
- ▶ 如果每个Warp访问一列
✎ 32-路bank冲突

Bank 0
Bank 1
...
Bank 31



避免Bank冲突

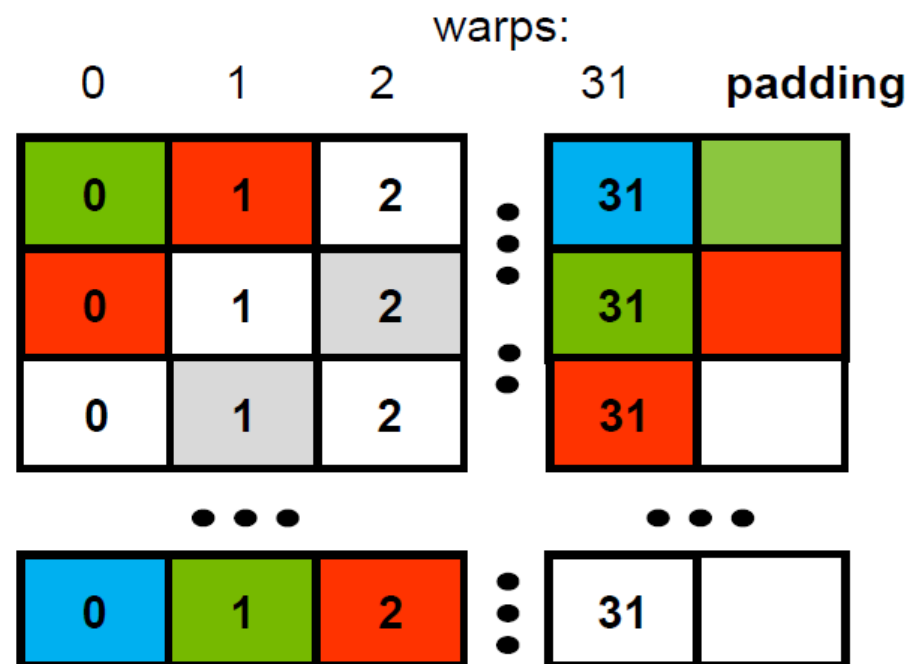


► 增加一列

✎ 32 x 33 共享内存数组

✎ 无bank冲突!

Bank 0
Bank 1
...
Bank 31



1. David B. Kirk and Wen-mei W. Hwu, Programming Massively Parallel Processors, 2nd Edition, Morgan Kaufmann, 2013.
2. CUDA C Programming Guide, Nvidia. <http://docs.nvidia.com/cuda/cuda-c-programming-guide/>
3. CUDA C BEST PRACTICES GUIDE, Nvidia. <http://docs.nvidia.com/cuda/cuda-c-best-practices-guide>
4. X. Mei and X.-W. Chu, “Dissecting GPU memory hierarchy through microbenchmarking,” IEEE TPDS 2017.