



中山大學
SUN YAT-SEN UNIVERSITY



多核程序设计与实践

CUDA内存结构（续）

线程执行模式与原子操作

陶钧

taoj23@mail.sysu.edu.cn

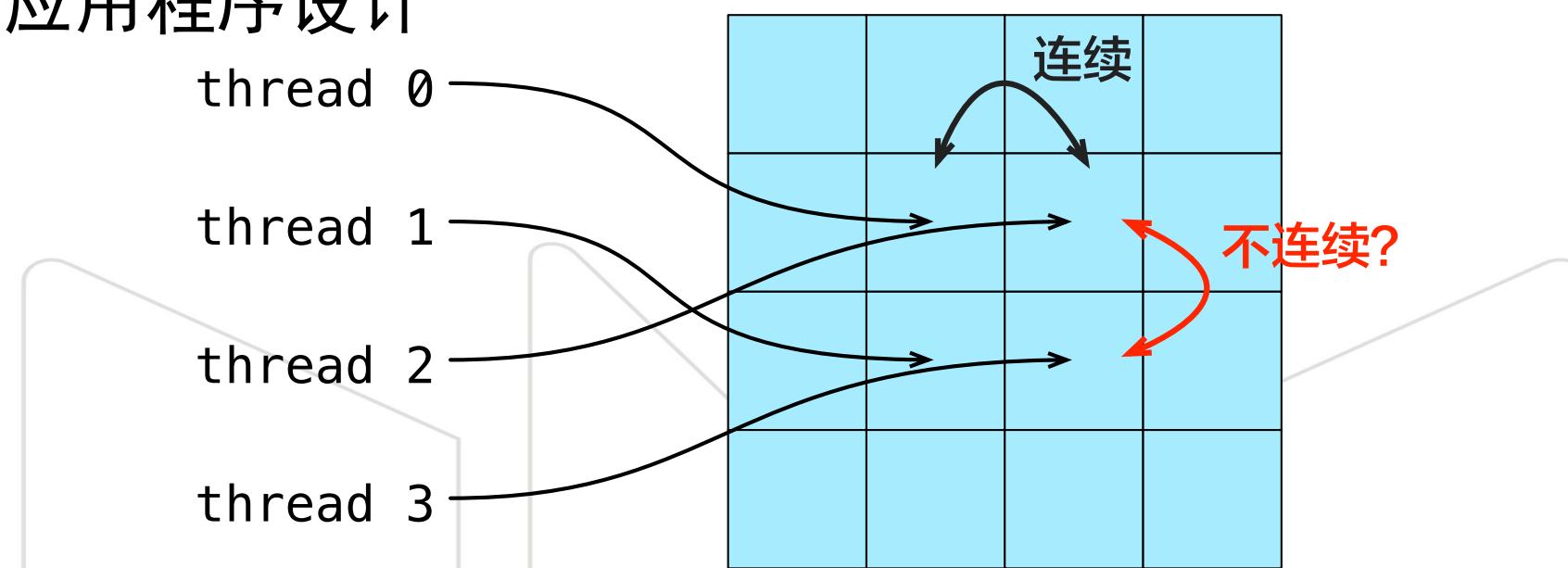
中山大学 数据科学与计算机学院
国家超级计算广州中心

- 内存结构（续）
- 线程执行模型
- 原子操作与同步



● 特性

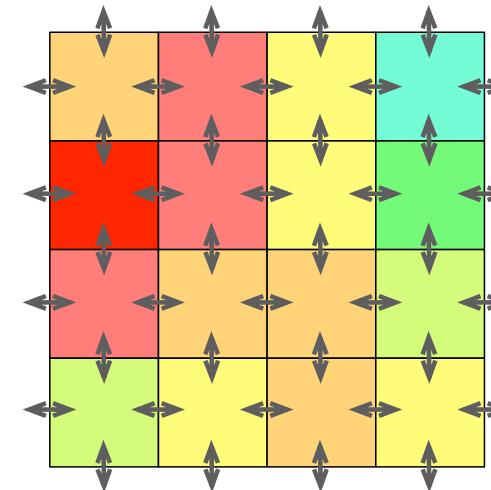
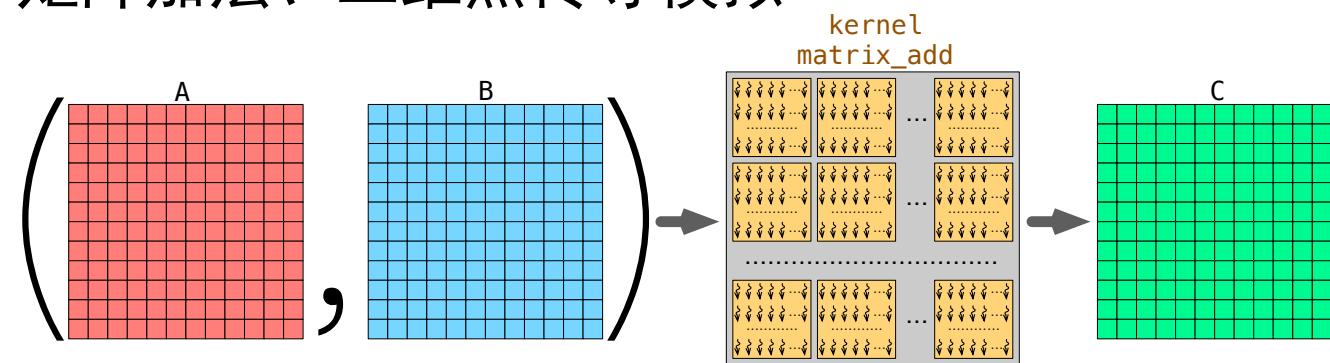
- 与只读内存共享物理空间（Kepler以后）
- 每个SM具有数个texture fetch units
- 高维纹理
 - 物理内存一维连续
 - 缓存数据方式具有高维空间局部性
 - 为图形应用程序设计



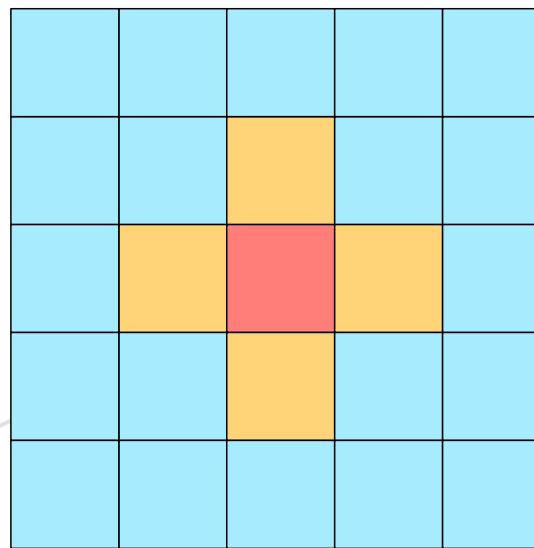
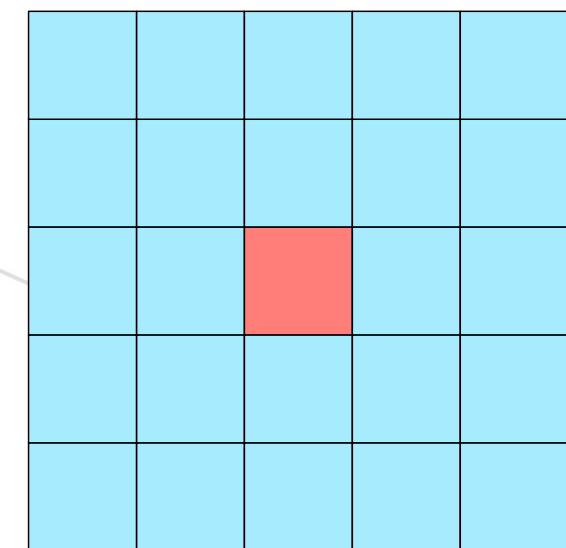
● 适用计算模型

- 线程束访问内存的模式具有空间局部性
- 二维纹理内存举例
 - 矩阵加法、二维热传导模拟

计算模型



内存访问模型



● 绑定纹理内存使用

– 声明全局纹理引用

- `texture<Datatype, Type, ReadMode> tex;`

– 调用过程

- 分配内存空间→绑定纹理→核函数调用→解除绑定→释放内存
 - `cudaBindTexture()`、`cudaBindTexture2D()`
 - `cudaUnbindTexture()`
- 与全局内存对比：分配内存空间→核函数调用→释放内存
- 与CPU上二维数组对比：`int **A` 为 `int *A_data` 提供二维引用

```
int *A_data = new int[n*m];
int **A = new int*[n];

for (int i=0; i<n; ++i)
    A[i] = &A_data[i*m];
```

● 纹理内存使用举例

```
#define N 1024
texture<float, 1, cudaReadModeElementType> tex;

__global__ void kernel() {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    float x = tex1Dfetch(tex, i);
}

int main() {
    float *buffer;
    cudaMalloc(&buffer, N*sizeof(float));
    cudaBindTexture(0, tex, buffer, N*sizeof(float));
    kernel <<<grid, block >>>();
    cudaUnbindTexture(tex);
    cudaFree(buffer);
}
```

Datatype:
char, short, int, long,
longlong, float, double
及其向量形式：
uchar4, float3等

● 纹理内存使用举例

```
#define N 1024
texture<float, 1, cudaReadModeElementType> tex;

__global__ void kernel() {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    float x = tex1Dfetch(tex, i);
}

int main() {
    float *buffer;
    cudaMalloc(&buffer, N*sizeof(float));
    cudaBindTexture(0, tex, buffer, N*sizeof(float));
    kernel <<<grid, block >>>();
    cudaUnbindTexture(tex);
    cudaFree(buffer);
}
```

Type:

cudaTextureType1D (1)
cudaTextureType2D (2)
cudaTextureType3D (3)
cudaTextureType1DLayered (4)
cudaTextureType2DLayered (5)
cudaTextureTypeCubemap (6)
cudaTextureTypeCubemapLayered (7)

● 纹理内存使用举例

```
#define N 1024
texture<float, 1, cudaReadModeElementType> tex;

__global__ void kernel() {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    float x = tex1Dfetch(tex, i);
}

int main() {
    float *buffer;
    cudaMalloc(&buffer, N*sizeof(float));
    cudaBindTexture(0, tex, buffer, N*sizeof(float));
    kernel <<<grid, block >>>();
    cudaUnbindTexture(tex);
    cudaFree(buffer);
}
```

ReadMode:
cudaReadModeElementType
cudaReadModeNormalizedFloat

● 二维纹理使用举例

```
texture<float, 2, cudaReadModeElementType> tex;

__global__ void kernel() {
    int x = blockIdx.x * blockDim.x + threadIdx.x;
    int y = blockIdx.y * blockDim.y + threadIdx.y;
    float v = tex2D (tex, x, y); ←
}

int main() {
    float *buffer;
    cudaMalloc(&buffer, W*H*sizeof(float));
    cudaChannelFormatDesc desc = cudaCreateChannelDesc<float>(); ←
    cudaBindTexture2D(0, tex, buffer,
                      desc, w, h,
                      w*sizeof(float)); ←
    kernel << <grid, block >> >();
    cudaUnbindTexture(tex);
    cudaFree(buffer);
}
```

- w*sizeof(float):
- 最后一个参数为pitch而非分配空间大小
 - pitch指明的是内存对齐的方式，目的是避免存储体冲突

● 使用无绑定纹理

- 分配内存
- 描述通道信息
- 描述纹理信息
- 创建纹理对象
- 核函数调用
- 销毁纹理对象
- 释放内存

```
float *buffer;
cudaMalloc(&buffer, N*sizeof(float));

cudaResourceDesc resDesc;
memset(&resDesc, 0, sizeof(resDesc));
resDesc.resType = cudaResourceTypeLinear;
resDesc.res.linear.devPtr = buffer;
resDesc.res.linear.desc.f = cudaChannelFormatKindFloat;
resDesc.res.linear.desc.x = 32;
resDesc.res.linear.sizeInBytes = N*sizeof(float);

cudaTextureDesc texDesc;
memset(&texDesc, 0, sizeof(texDesc));
texDesc.readMode = cudaReadModeElementType;

cudaTextureObject_t tex;
cudaCreateTextureObject(&tex, &resDesc, &texDesc, NULL);
kernel << <grid, block >> >(tex);
cudaDestroyTextureObject(tex);
cudaFree(buffer);
```

● address mode与filter mode

– address mode: 指明当坐标越界时如何处理

- cudaAddressModeClamp
 - 越界时等于对应边界上的值
 - $\text{tex}[k] = \text{tex}[0] \text{ (if } k < 0\text{)}$
 - $\text{tex}[k] = \text{tex}[n-1] \text{ (if } k \geq n\text{)}$
- cudaAddressModeBorder
 - 越界时值为常数
 - $\text{tex}[k] = 0 \text{ (if } k < 0 \text{ or } k \geq n\text{)}$
- cudaAddressModeWrap
 - 以n为周期重复
 - $\text{tex}[k] = \text{tex}[k \% n] \text{ (if } k < 0 \text{ or } k \geq n\text{)}$
- cudaAddressModeMirror
 - 沿边界镜像，以 $2n-2$ 为周期重复
 - $\text{tex}[l] = \text{tex}[k] \text{ (if } (l+k) \% (2n-2) == 0\text{)}$

● address mode与filter mode

– filter mode: 指明插值方式

- cudaFilterModeLinear: 返回相邻格点的线性插值
- cudaFilterModePoint: 返回所处格点的值

– address mode与filter mode的指明方式

- 绑定纹理

```
texture<float, 1, cudaMemcpyElementType> tex;  
tex.addressMode = cudaAddressModeClamp;  
tex.filterMode = cudaFilterModeLinear;
```

- 非绑定纹理

```
cudaTextureObject_t tex;  
cudaCreateTextureObject(&tex, &resDesc, &texDesc, NULL);  
tex.addressMode = cudaAddressModeClamp;  
tex.filterMode = cudaFilterModeLinear;
```

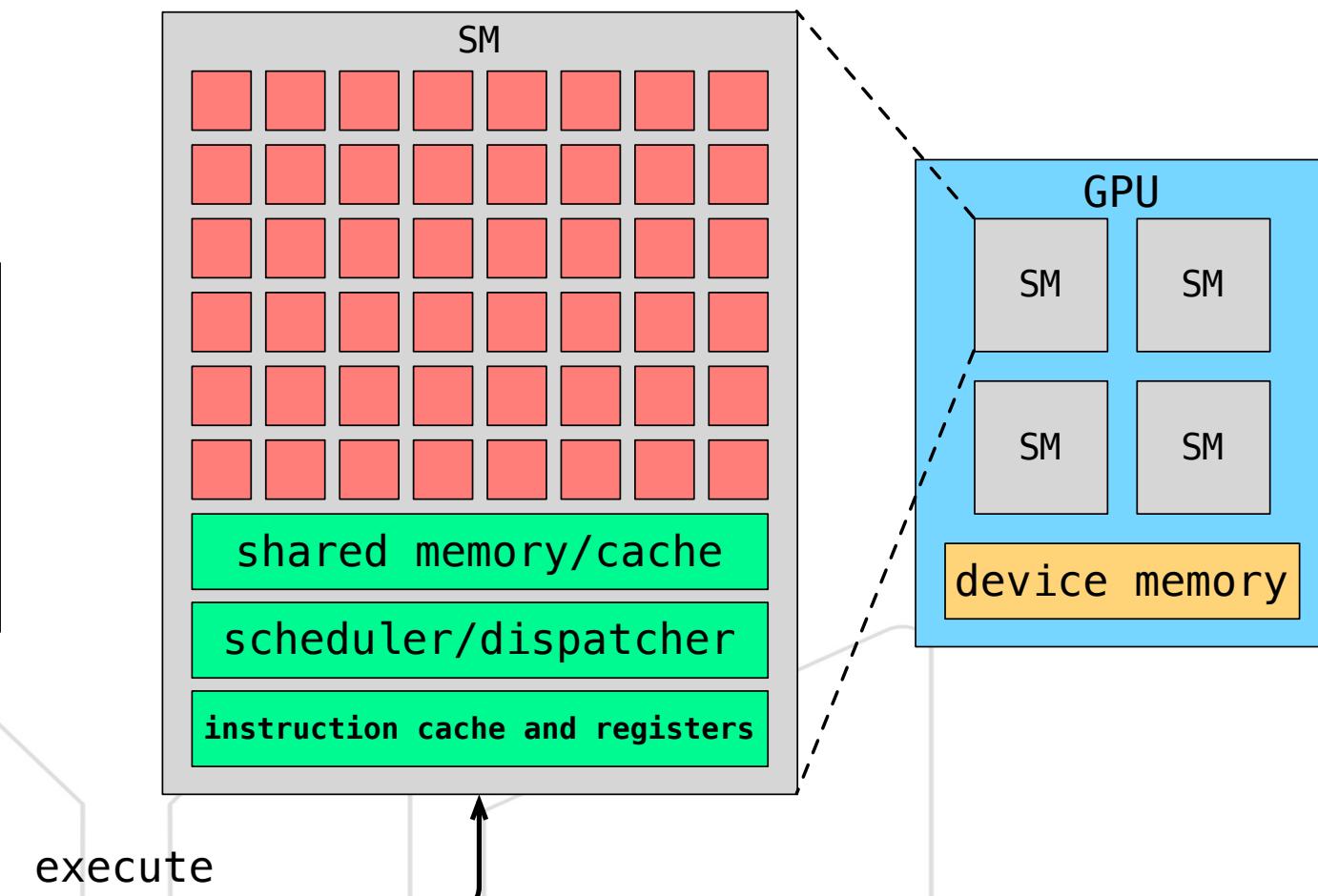
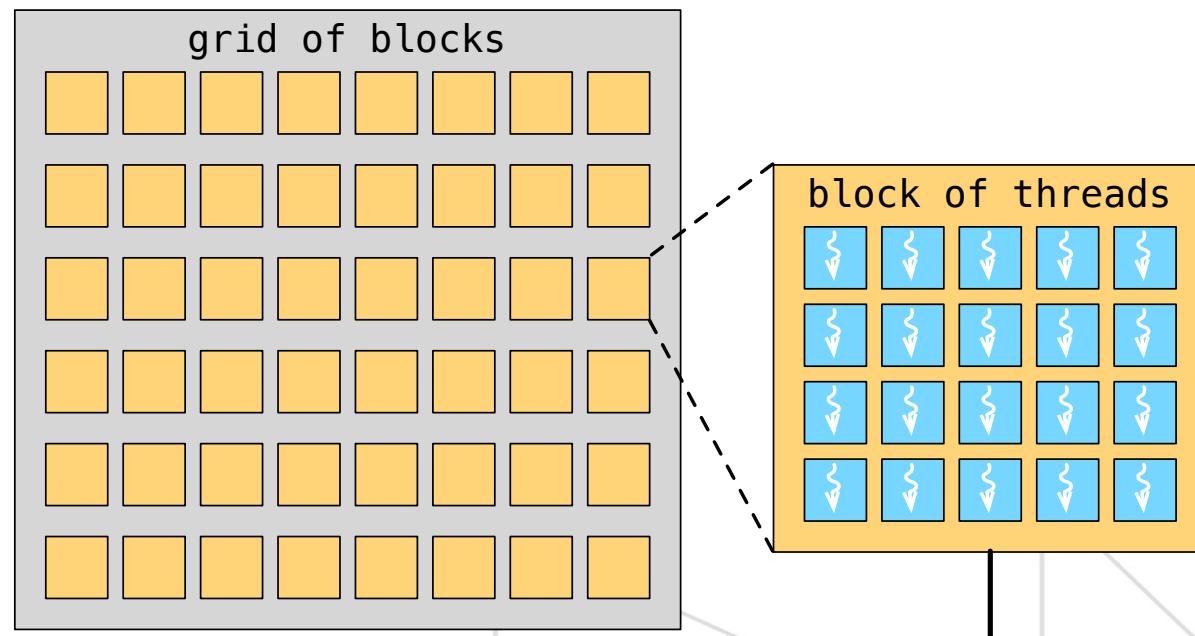
- 全局内存
 - 动态、静态全局内存、统一内存寻址、零拷贝内存（扩展阅读）
 - 二级缓存
- 常量内存
 - 片上常量缓存、适合统一读取，如广播访问
- 只读内存
 - 片上缓存、适合分散读取
- 纹理内存
 - 片上缓存、高维空间局部性
- 共享内存
 - 片上、可编程、适合分散读取、存储体冲突
- 内存选择：取决于程序对数据的访问模式

- 内存结构（续）
- 线程执行模型
- 原子操作与同步



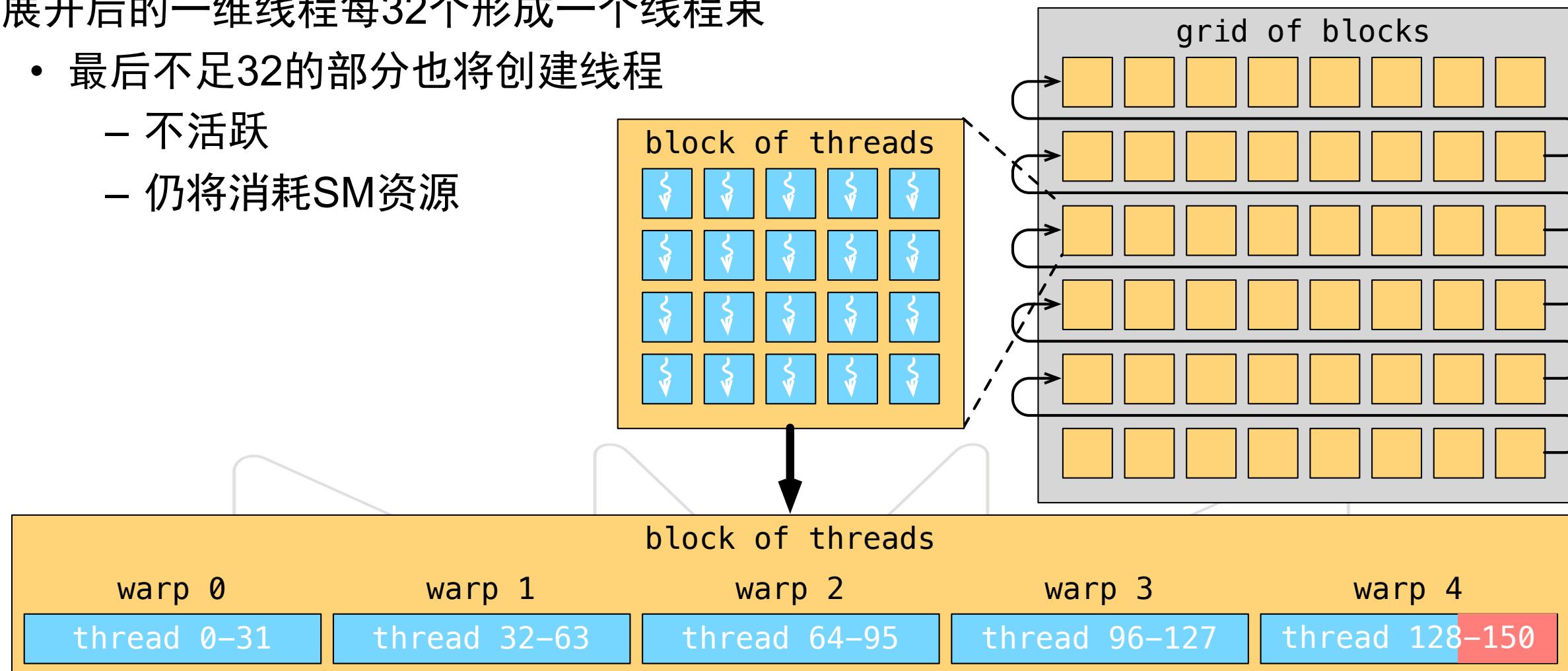
● 逻辑视图

- 每个线程块由一个SM执行
- 由硬件调度
- 无法控制线程块的执行顺序



● 硬件视图

- 所有线程块在硬件上都是一维
- 三维线程将沿 $x \rightarrow y \rightarrow z$ 顺序展开到一维
 - `threadIdx.z * blockDim.y * blockDim.x + threadIdx.y * blockDim.x + threadIdx.x`
- 展开后的一维线程每32个形成一个线程束
 - 最后不足32的部分也将创建线程
 - 不活跃
 - 仍将消耗SM资源

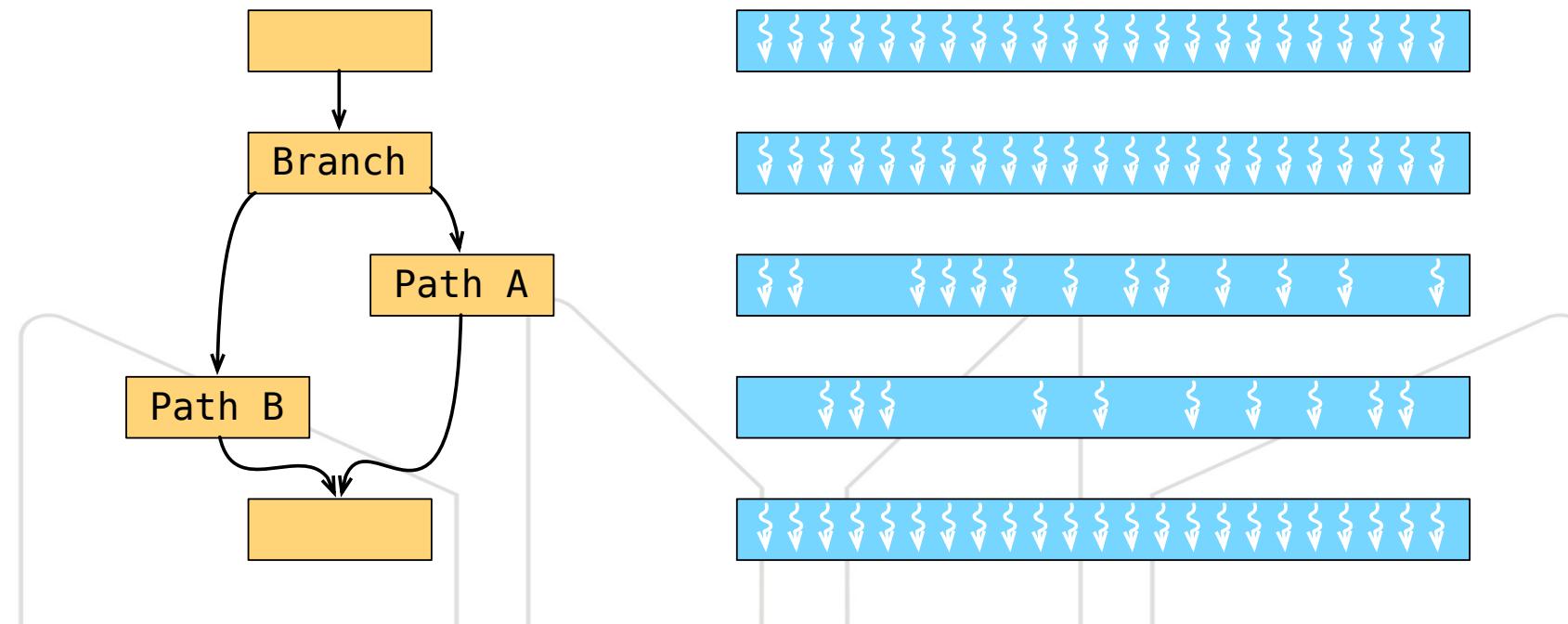


● 线程束调度

- 线程束切换开销为0
 - SM保存每个线程束的执行上下文
 - 在整个线程束的生存周期中保存于芯片内
 - 上下文切换没有损失
 - 可切换同一SM上不同线程块的线程束
- SM中常驻线程块数量受可用资源限制
 - 资源：程序计数器、寄存器、共享内存
 - **活跃线程束**：具备计算资源（寄存器、共享内存等）的线程束
 - Kepler上最大为64
 - **选定的线程束**：被调度到执行单元的线程束（Kepler上最大为4）
 - **符合条件的线程束**：准备执行但尚未执行
 - **阻塞的线程束**：没做好执行准备（参数未就绪、无可用CUDA核心）

● 线程束执行

- 每一个线程束以SIMD方式在SM上执行
 - 线程束内同时执行同样语句
 - 线程束外的视角看来为SIMT
- 分支分化 (branch divergence)
 - 线程束出现不同的控制流



● 线程束执行

– 以下代码哪个会出现分支分化?

```
__global__ void kernel_a(){
    if (blockIdx.x % 2) {
        do_something();
    } else {
        do_something_else();
    }
}
```

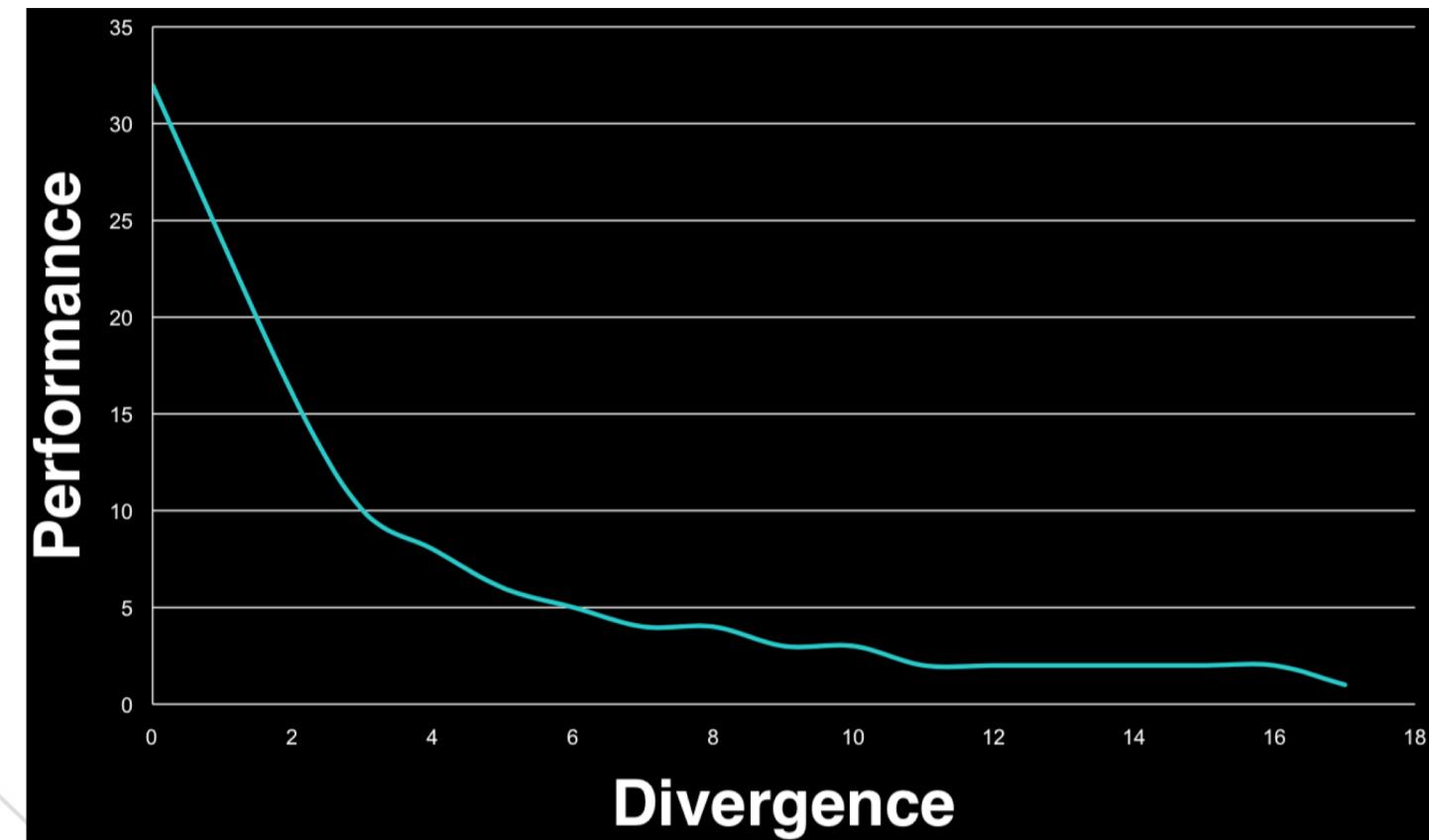
```
__global__ void kernel_b(){
    if (threadIdx.x % 2) {
        do_something();
    } else {
        do_something_else();
    }
}
```



● 线程束执行

– 分支数与效率

```
__global__ void kernel(){
    switch(threadIdx.x % N) {
        case (0):
            do_case_0();
            break;
        case (1):
            do_case_1();
            break;
        ...
    }
}
```



图片来自NVIDIA

● 线程束执行

– busy waiting vs signal

- busy waiting: 如, 使用while循环不断检查条件是否满足
- signal: 当条件满足由系统发送指令
- 现实生活中都很常见: 如等飞机
- busy waiting 的问题是?

```
__global__ void kernel_a(){
    if (threadIdx.x!=0){
        while (not_ready){
            //do nothing but wait
        }
    } else {
        set_ready();
    }
}
```

```
__global__ void kernel_b(){
    if(threadIdx.x==0){
        set_ready();
    }
    __syncthreads();
}
```

● 线程束执行

- 条件语句 vs 条件赋值
 - C代码

```
if (i<n){  
    a = 1;  
} else {  
    a = 2;  
}
```

```
a = (i<n)?1:2;
```



● 线程束执行

– 条件语句 vs 条件赋值

- C代码

```
if (i<n){  
    a = 1;  
} else {  
    a = 2;  
}
```

```
a = (i<n)?1:2;
```

- PTX (Parallel Thread eXecution) ISA 指令级源码

```
mov.s32 a, 0; //a=0  
setp.lt.s32 p, i, n; //p=(i<n)  
@p    mov.s32 a, 1; //a=1  
@!p   mov.s32 a, 2; //a=2
```

```
mov.s32 a, 0; //a=0  
setp.lt.s32 p, i, n; //p=(i<n)  
selp a, 1, 2, p; //a=(p)?1:2
```

● 线程束执行

- 减少分支分化的影响
 - 减少判断语句
 - 尤其是减少基于threadIdx的判断语句
 - » 判断语句不必然导致分支分化
 - 使用条件语句代替条件赋值

```
if (threadIdx.x < RADIUS) {  
    smem[sid-RADIUS] = in[tid-RADIUS];  
    smem[sid+BDIM] = in[tid+BDIM];  
}
```

```
if (threadIdx.x < RADIUS) {  
    smem[sid-RADIUS] = in[tid-RADIUS];  
} else if (threadIdx.x > BDIM-RADIUS) {  
    smem[sid+RADIUS] = in[tid+RADIUS];  
}
```

- 平衡分支执行时间
 - 避免出现执行时间过长的分支

- 内存结构（续）
- 线程执行模型
- 原子操作与同步



● 原子指令

- 执行过程不能分解为更小的部分：不被中断
- 避免竞争条件出现（回顾讲义3：OpenMp）
- 竞争条件（race condition）
 - 程序运行结果依赖于不可控的执行顺序

● OpenMP计算直方图例子中

- 使用临界区或原子指令避免对histogram的更新出现竞争条件

```
#pragma omp parallel for
for(int i=0; i<1000; ++i){
    int value = rand()%20;

    #pragma omp critical
    {
        histogram[value]++;
    }
}
```

```
#pragma omp parallel for
for(int i=0; i<1000; ++i){
    int value = rand()%20;
    #pragma omp atomic
    histogram[value]++;
}
```

● CUDA原子操作

- 加减: `atomicAdd()`, `atomicSub()`, `atomicInc()`, `atomicDec()`
- 比较与交换: `atomicMin()`, `atomicMax()`, `atomicExch()`, `atomicCAS()`
- 位运算: `atomicAnd()`, `atomicOr()`, `atomicXor()`
- 支持的原子操作依据GPU架构而异
 - 计算能力1.0的设备不支持全局内存上的原子操作
 - 无法同步所有线程



● CUDA原子操作

– 基础操作: **atomicCAS()**

- 其他所有原子操作均可由**atomicCAS()**实现
- **CAS: compare and swap**
 - 读取目标位置(address) 并于预期值(old_val) 进行比较
 - » 相等则将new_val写入目标位置
 - » 不等则不发生变化
 - 返回目标位置中原值: 可用来检查CAS操作是否成功

```
int compare_and_swap(int* address, int old_val, int new_val){  
    int old_reg_val = * address;  
  
    if(old_reg_val == old_val) {  
        * address = new_val;  
    }  
  
    return old_reg_val;  
}
```

● CUDA原子操作

– 基础操作: **atomicCAS()**

- 例: 使用**atomicCAS()**实现**atomicAdd()**

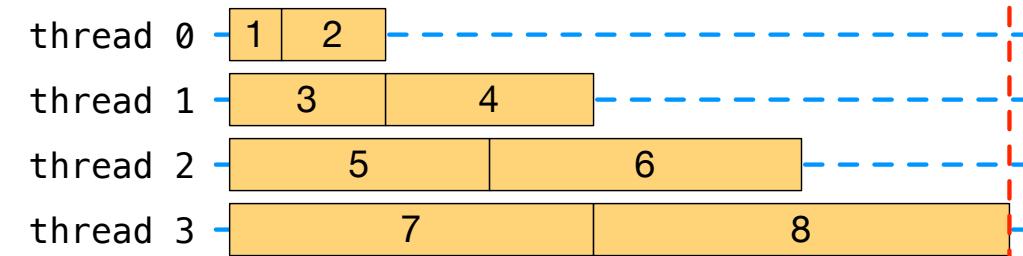
```
__device__ int my_atomicAdd(int* address, int incr){  
    int expected = *address;  
    int old_val = atomicCAS(address, expected, expected+incr);  
  
    while (old_val != expected) {  
        expected = old_val;  
        old_val = atomicCAS(address, expected, expected+incr);  
    }  
  
    return old_val;  
}
```



- CUDA原子操作应用举例：动态工作队列

- 使用计数器动态获取工作
 - 增加负载平衡？

```
__global__ void work_queue(int* work_q, int* q_counter,
                           int* output, int queue_max)
{
    int local_counter = 0;
    do {
        int qid = atomicInc(q_counter, queue_max);
        int result = work(work_q[qid]);
        output[qid] = result;
        ++local_counter;
    } while (qid!=0 && local_counter!=N);
}
```



注意 **atomicInc** 中第二个参数不是增加量：
(可由 **atomicAdd** 替代)

atomicInc (address, val):
 $((\ast\text{address}) \geq \text{val}) ? 0 : (\ast\text{address} + 1)$

- CUDA原子操作应用举例：计算直方图

- 最直接的思路：

- 使用**atomicAdd**给histogram中相应值每次增加1

```
__global__ void histogram(int *histogram, int* val){  
    int tid = blockIdx.x*blockDim.x + threadIdx.x;  
    int val_i = val[tid];  
    atomicAdd(&histogram[val_i], 1);  
}
```

- 问题？



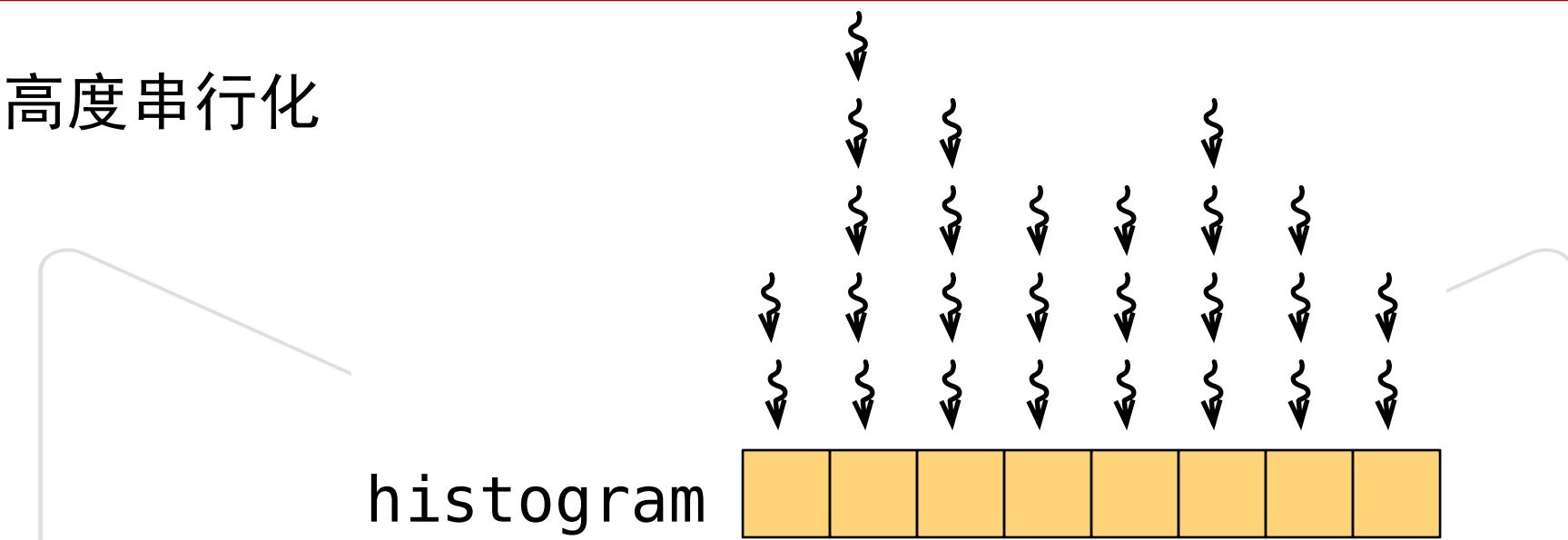
- CUDA原子操作应用举例：计算直方图

- 最直接的思路：

- 使用**atomicAdd**给histogram中相应值每次增加1

```
__global__ void histogram(int *histogram, int* val){  
    int tid = blockIdx.x*blockDim.x + threadIdx.x;  
    int val_i = val[tid];  
    atomicAdd(&histogram[val_i], 1);  
}
```

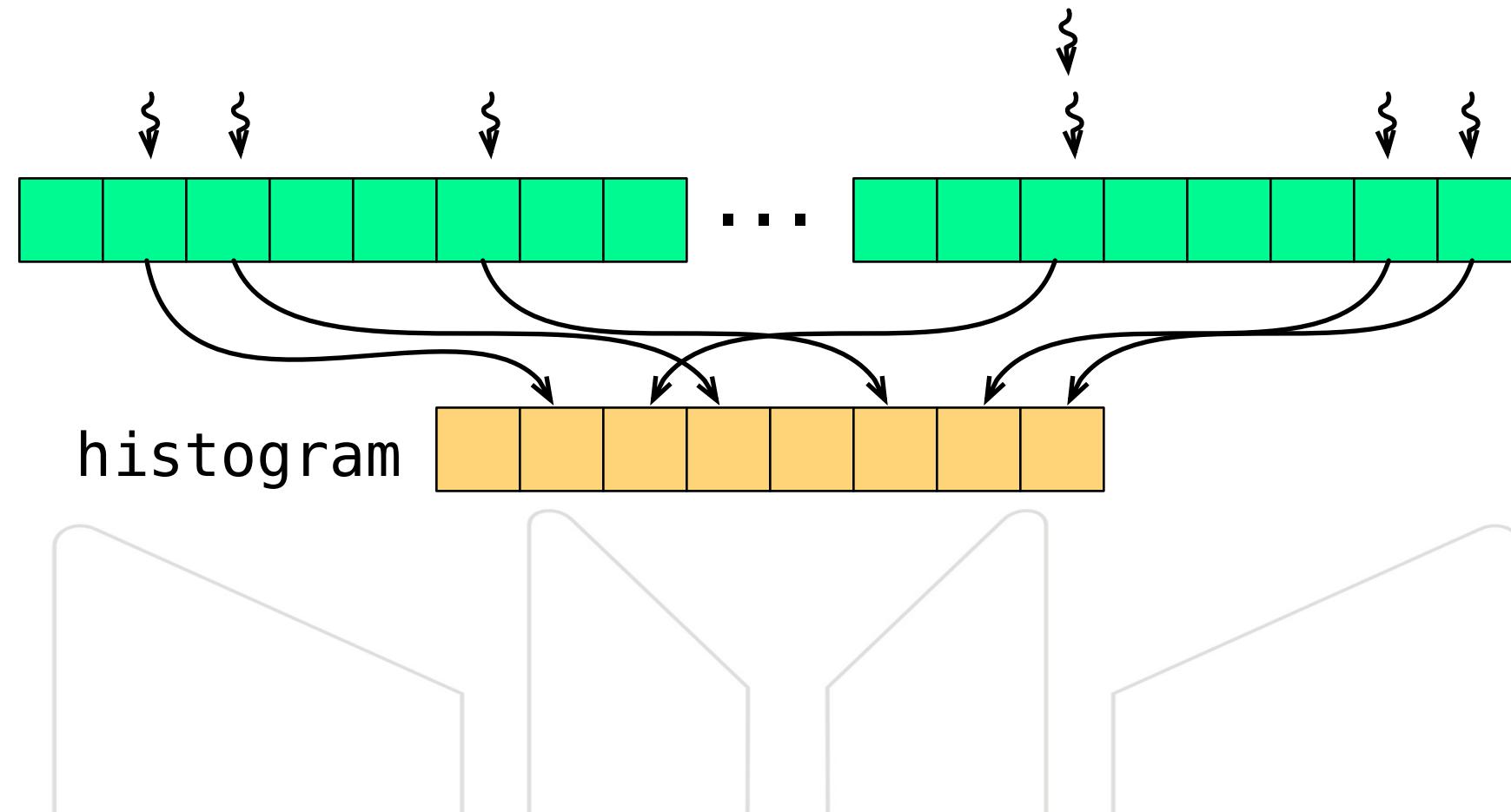
- 问题：高度串行化



- CUDA原子操作应用举例：计算直方图

- 改进方法：使用共享内存

- 每个线程块先更新本地直方图
 - 所有线程更新结束后，一次写入全局直方图



- CUDA原子操作应用举例：计算直方图

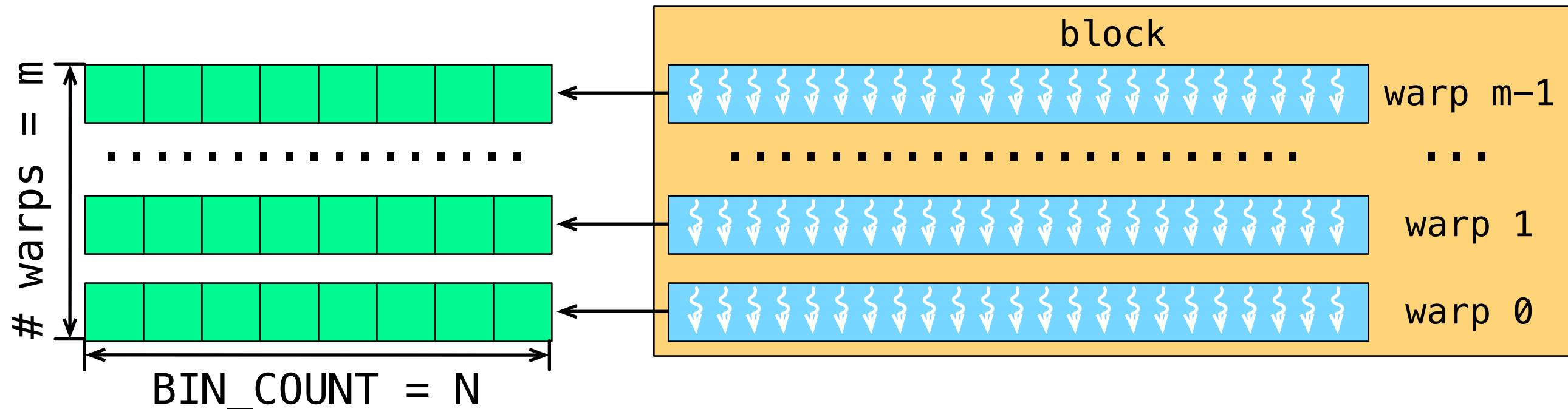
- 改进方法：使用共享内存

```
__global__ void histogram(int *histogram, int* val){  
    int tid = blockIdx.x*blockDim.x + threadIdx.x;  
    __shared__ int local_hist[N];  
  
    if (tid < N) {  
        local_hist[tid] = 0;  
    }  
    __syncthreads();  
  
    int val_i = val[tid];  
    atomicAdd(&local_hist[val_i], 1);  
    __syncthreads();  
  
    if (tid < N && local_hist[tid]!=0) {  
        atomicAdd(&histogram[tid], local_hist[tid]);  
    }  
}
```

● CUDA原子操作应用举例：计算直方图

– 是否能不使用原子操作？

- 利用线程束隐含的同步（早期算法）
- 分配大小为 $m \times N$ 的共享内存
 - 其中为 m 为一个线程块中线程束的个数
 - 每个线程束写入独立的local copy

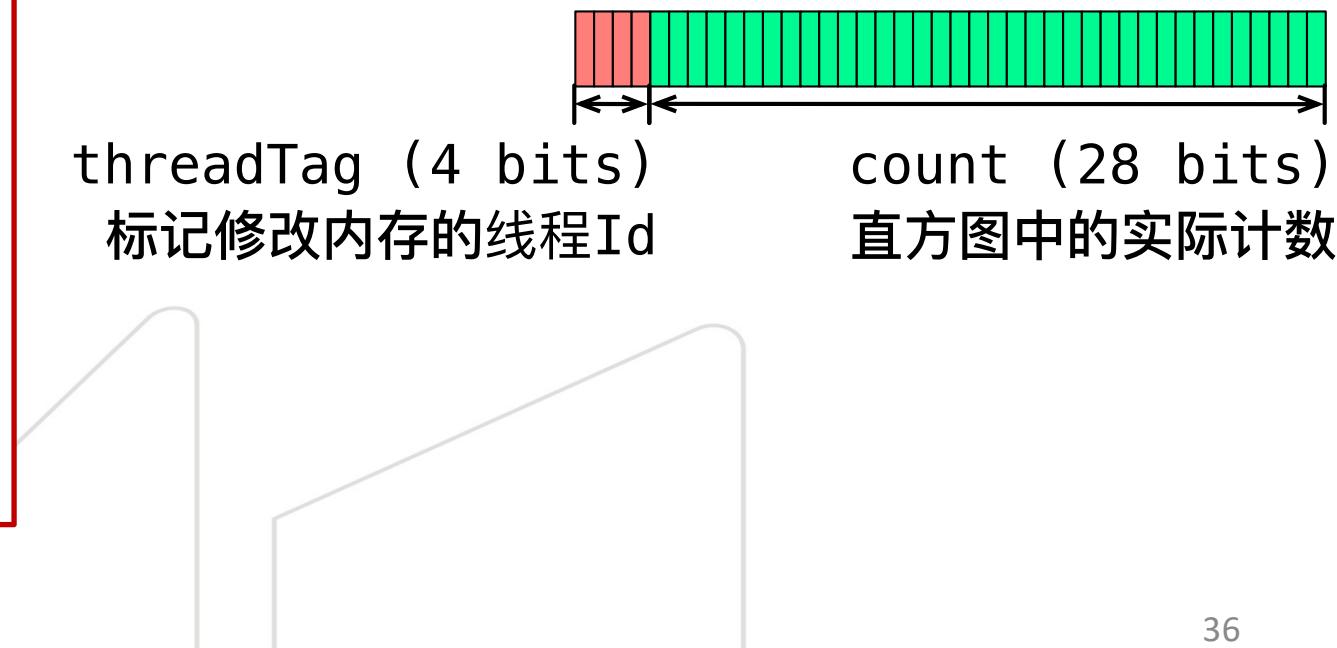


• CUDA原子操作应用举例：计算直方图

- 不使用原子操作：利用线程束隐含的同步（早期算法）
 - 如何保证线程束之间的线程不产生竞争状态？
 - 线程实际处于同步状态，需要知道的只是最后写入的线程

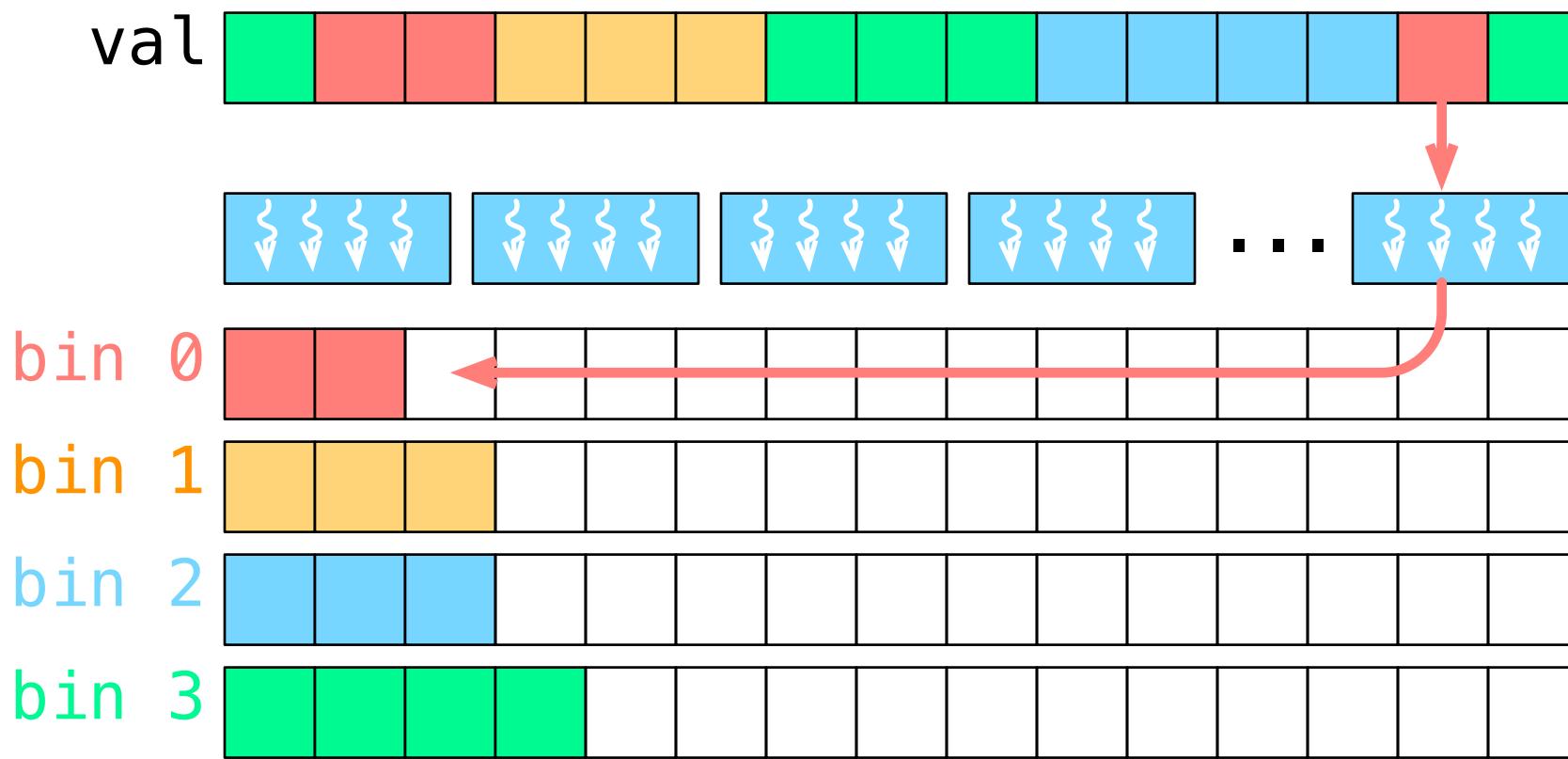
```
__device__ void addData(
    volatile unsigned int *s_WarpHist,
    unsigned int data,
    unsigned int threadTag)
{
    unsigned int count;
    do{
        count = s_WarpHist[data] & 0x07FFFFFFU;
        count = threadTag | (count + 1);
        s_WarpHist[data] = count;
    } while (s_WarpHist[data] != count);
}
```

`volatile unsigned int *s_WarpHist`
: `volatile` 是否可以省略?



- CUDA原子操作应用举例：元素归类

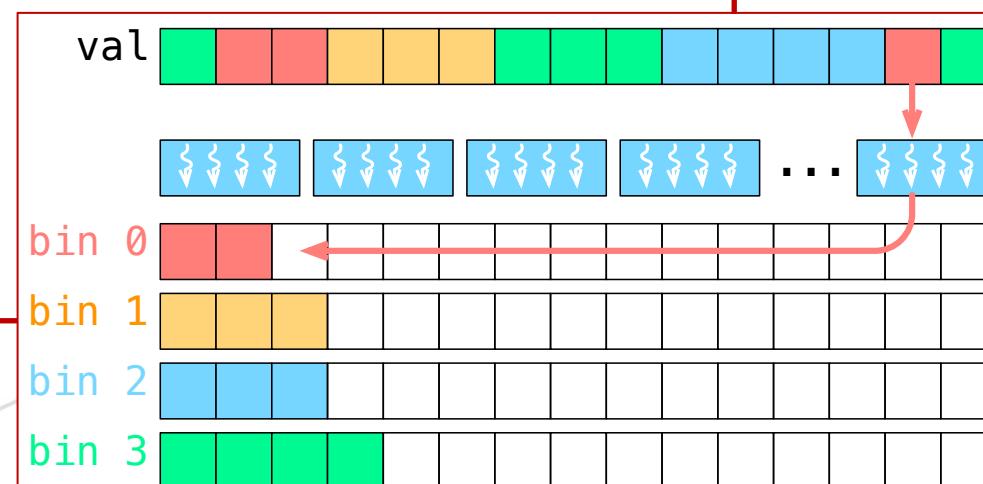
- 在计算直方图的基础上，记录每个bin中的元素
- 直接思路：使用N个数组，每次将元素置于相应数组末端
 - N为bin的数目



● CUDA原子操作应用举例：元素归类

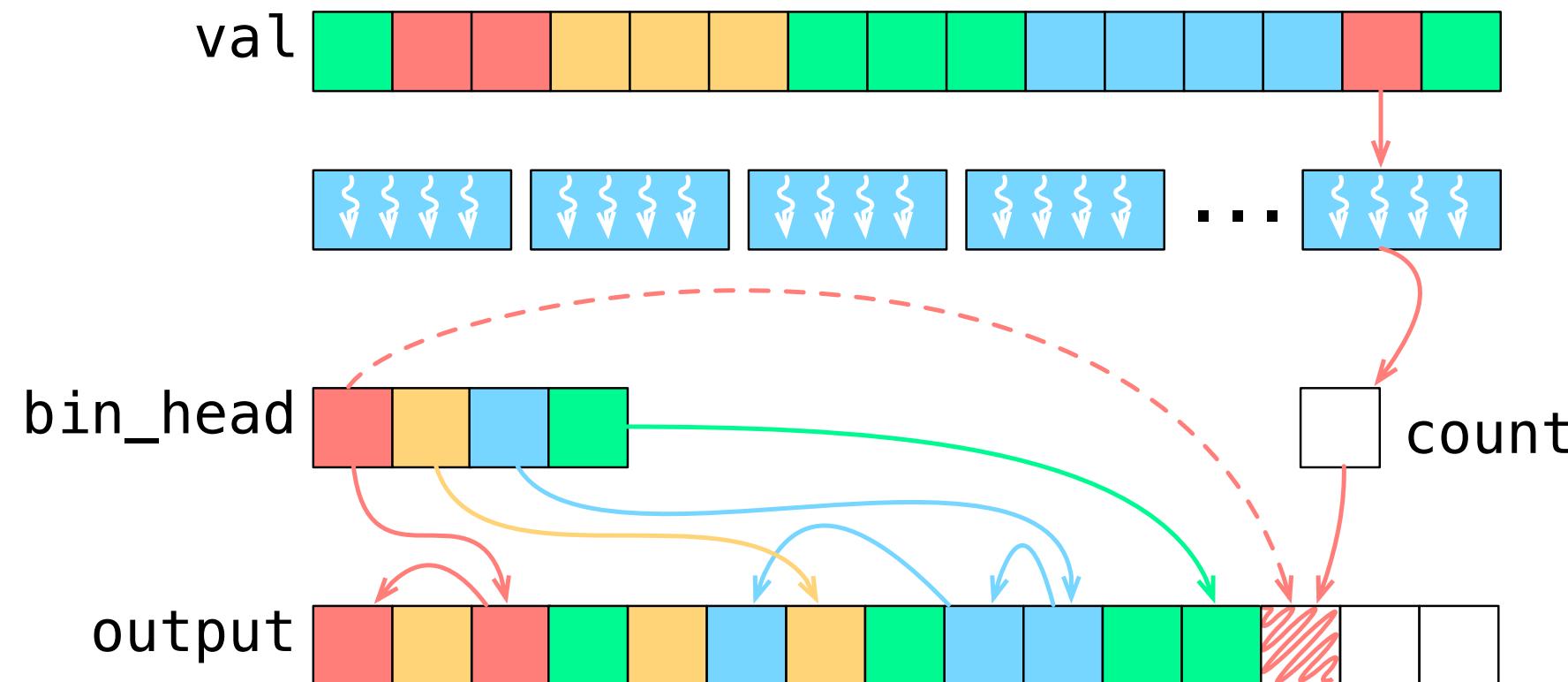
- 在计算直方图的基础上，记录每个bin中的元素
- 直接思路：使用N个数组，每次将元素置于相应数组末端
 - N为bin的数目

```
__global__ void classify(int *output, int* bin_size, int* val, int m){  
    int tid = blockIdx.x*blockDim.x + threadIdx.x;  
  
    int val_i = val[tid];  
    int bin = compute_bin(val_i);  
  
    int pos = atomicInc(&bin_size[bin], m);  
    output[bin*m+pos] = val_i  
}
```



- CUDA原子操作应用举例：元素归类

- 在计算直方图的基础上，记录每个bin中的元素
- 改进空间算法：使用链表



● CUDA原子操作应用举例：元素归类

- 在计算直方图的基础上，记录每个bin中的元素
- 改进空间算法：使用链表

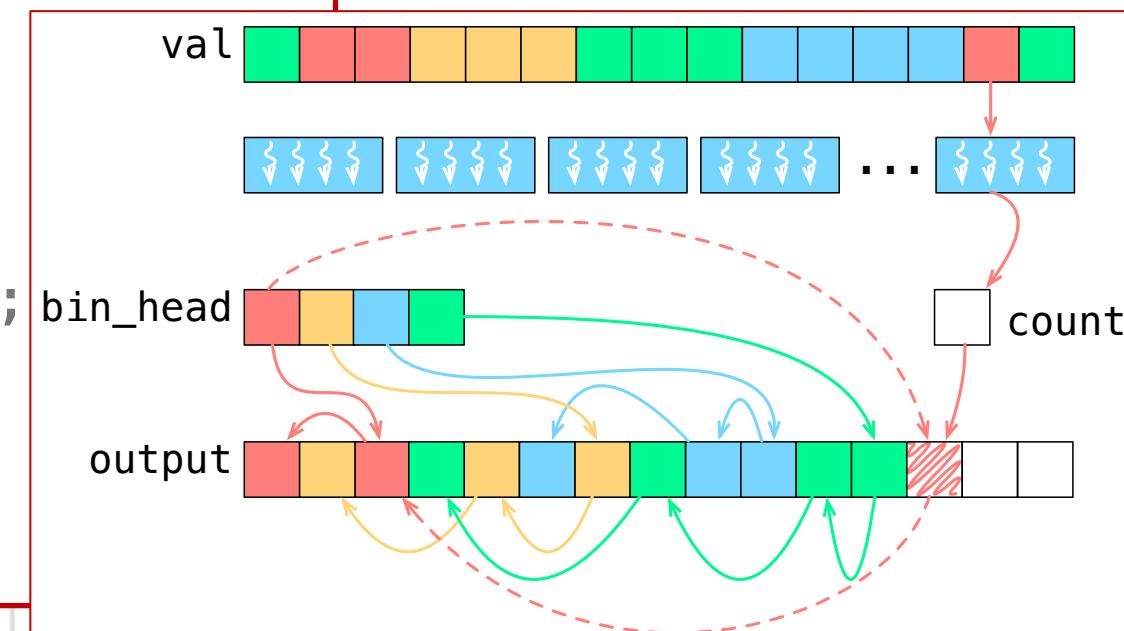
```
__global__ void classify(OutputType *output, int *count,
                        int* bin_head, int* val, int m){
    int tid = blockIdx.x*blockDim.x + threadIdx.x;

    int val_i = val[tid];
    int bin = compute_bin(val_i);

    int pos = atomicInc(count, m);
    int prev_pos = atomicExch(&bin_head[bin], pos);

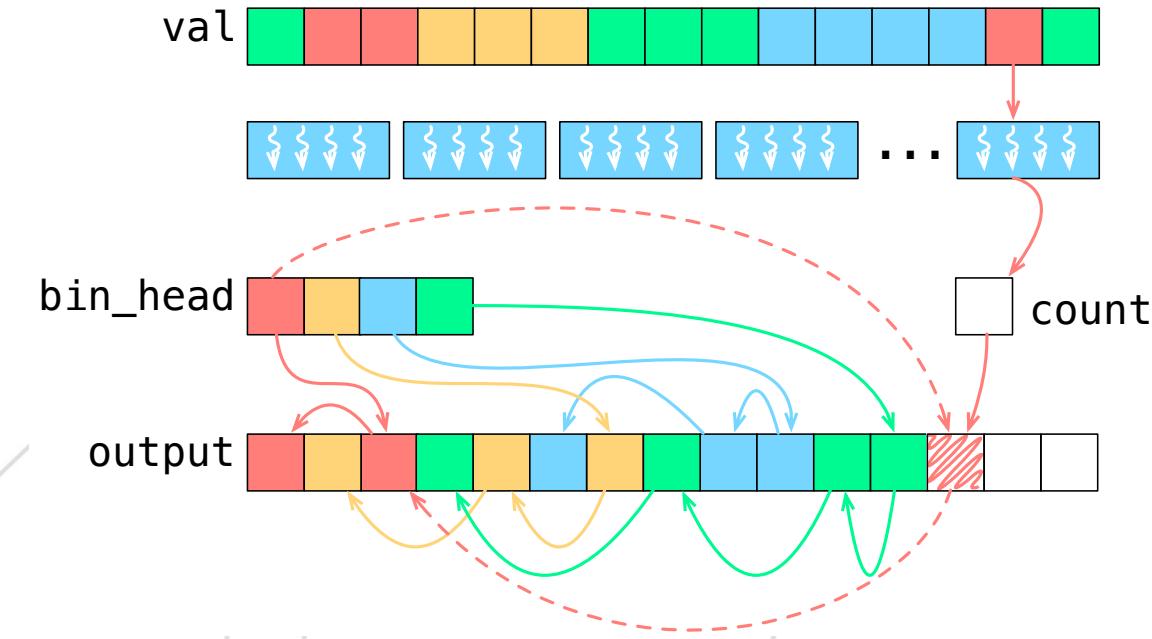
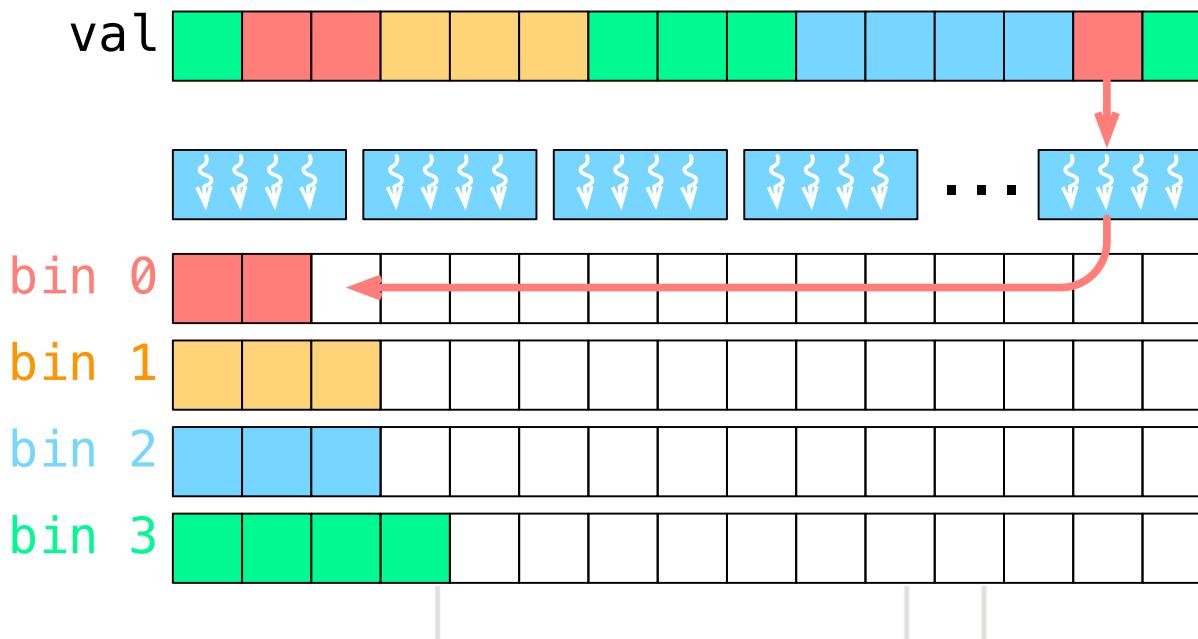
    OutputType ret = {val_i, prev_pos};
    output[bin*m+pos] = ret;
}
```

```
typedef struct {
    int val;
    int prev_ptr;
} OutputType;
```



● CUDA原子操作应用举例：元素归类

- 在计算直方图的基础上，记录每个bin中的元素
- 使用空间对比
 - 直接思路： $\text{bin_size}(N) + \text{output}(m \times N)$
 - 链表： $\text{bin_head}(N) + \text{output}(2 \times m) + \text{count}(1)$



- CUDA同步机制

- 核函数调用结束时同步
 - 在host线程中使用**cudaThreadSynchronize()**
 - 同步网格中所有线程
- 线程束中同步
 - 隐含的同步执行
- 使用原子操作?



● 使用原子操作实现原子锁

- 开锁状态: mutex = 0
- 锁状态: mutex = 1
- 使用while循环与atomicCAS()不断检查, 直到锁处于打开状态
- 需注意是否会导致死锁
 - 同一线程束内(最好)只使用一个线程执行开锁解锁操作
 - 使用额外操作保证线程束内原子操作/临界区状态

```
int *mutex;  
  
__device__ void lock(int* mutex){  
    while (atomicCAS(mutex, 0, 1)!=0);  
}  
  
__device__ void unlock(int* mutex){  
    atomicExch(mutex, 0);  
}
```

- 原子操作并不解决所有与执行顺序相关的问题

- 原子操作不指明执行顺序
 - 例：精度损失

$$1.23456 \times 10^{30} + 9.87654 \times 10^{-30} - 1.23456 \times 10^{30}$$
$$1.23456 \times 10^{30} - 1.23456 \times 10^{30} + 9.87654 \times 10^{-30}$$

- 在并行计算过程中需要更谨慎



● 线程执行模型

- 在硬件视角中，线程块始终为在一维上排列的一组线程
- 每32个线程形成一个线程束
- 线程束为同步执行
- 线程束在执行中遇到控制流将出现产生分支

● 原子操作与同步

- 基本原子操作
 - **atomic{Add, Sub, Inc, Dec, Min, Max, Exch, CAS, And, Or, Xor}**
- 提升效率：避免大量原子操作同时作用于同一数据
- 原子操作实现原子锁

Questions?

