



中山大學  
SUN YAT-SEN UNIVERSITY



国家超级计算广州中心  
NATIONAL SUPERCOMPUTER CENTER IN GUANGZHOU

# 多核程序设计与实践

## CUDA性能优化

陶钧

[taoj23@mail.sysu.edu.cn](mailto:taoj23@mail.sysu.edu.cn)

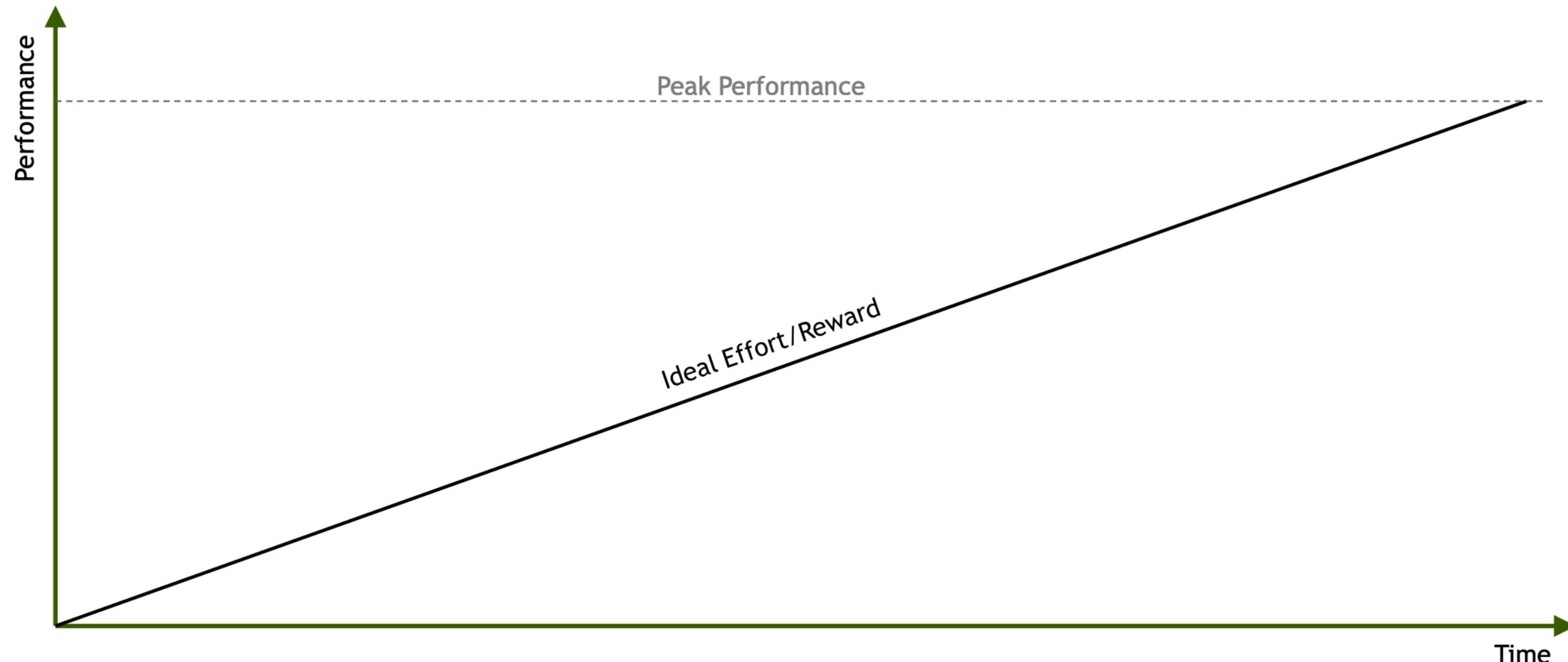
中山大学 数据科学与计算机学院  
国家超级计算广州中心

- 引言
- 优化全局内存访问
- 其他优化方法
- 优化实例

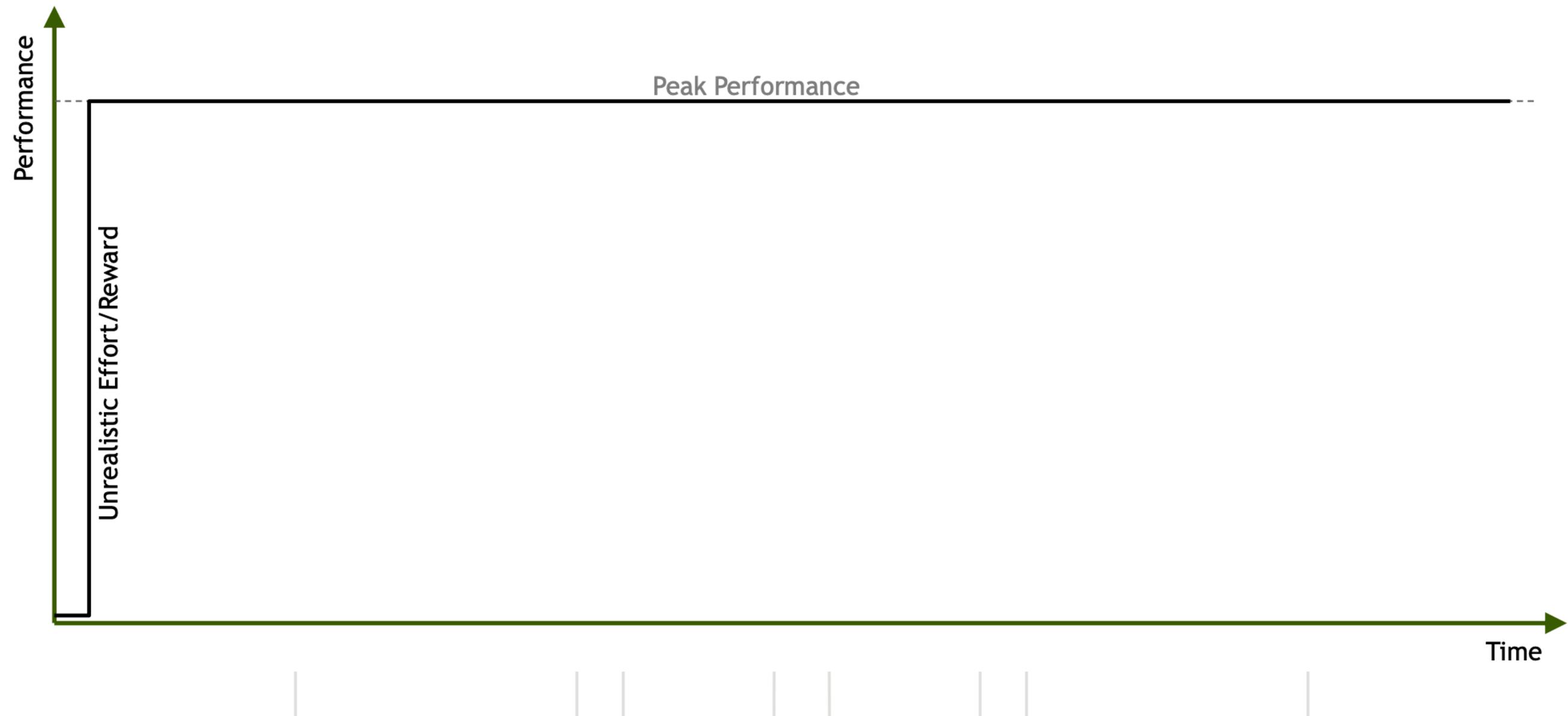


## ● 优化过程中应有的心态

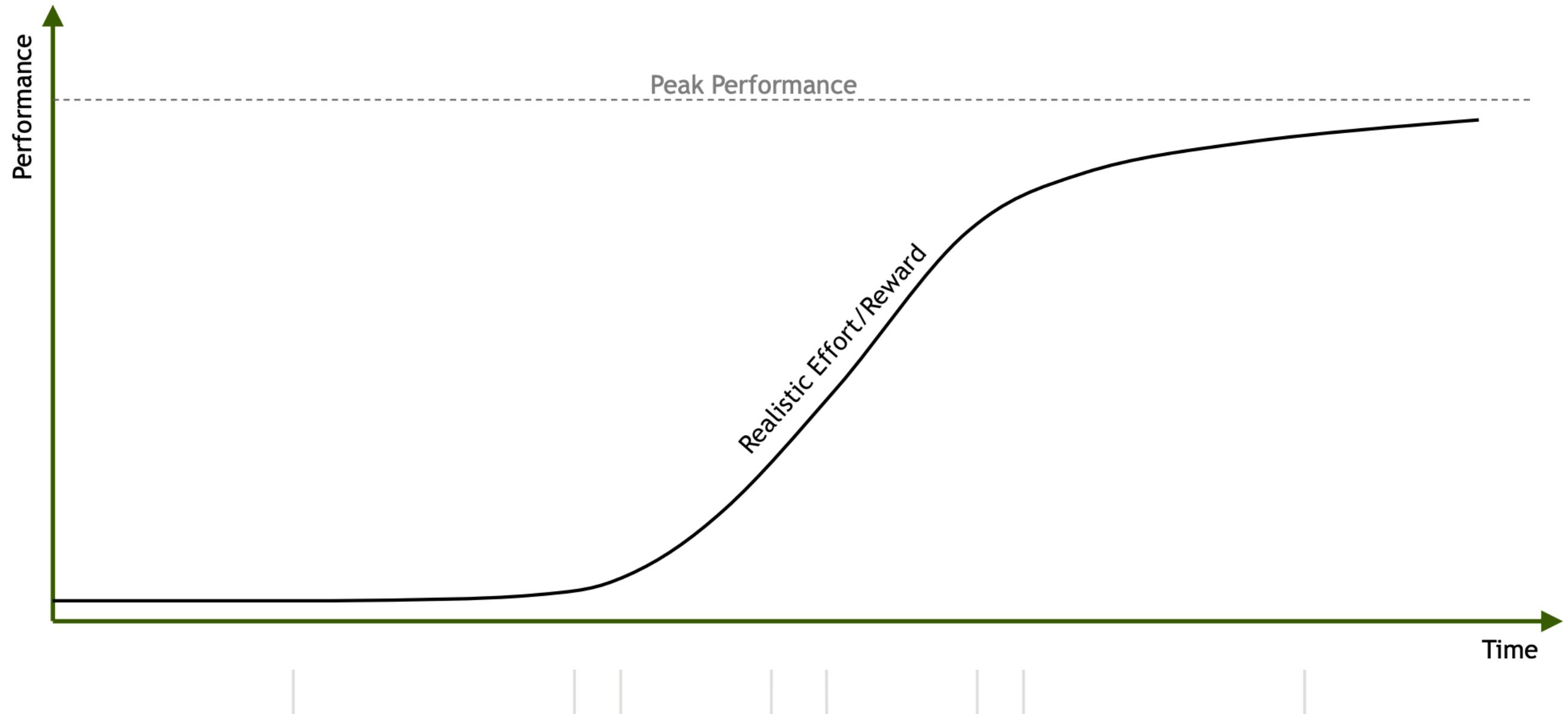
- 内容引自<http://on-demand.gputechconf.com/gtc/2017/presentation/s7122-stephen-jones-cuda-optimization-tips-tricks-and-techniques.pdf>



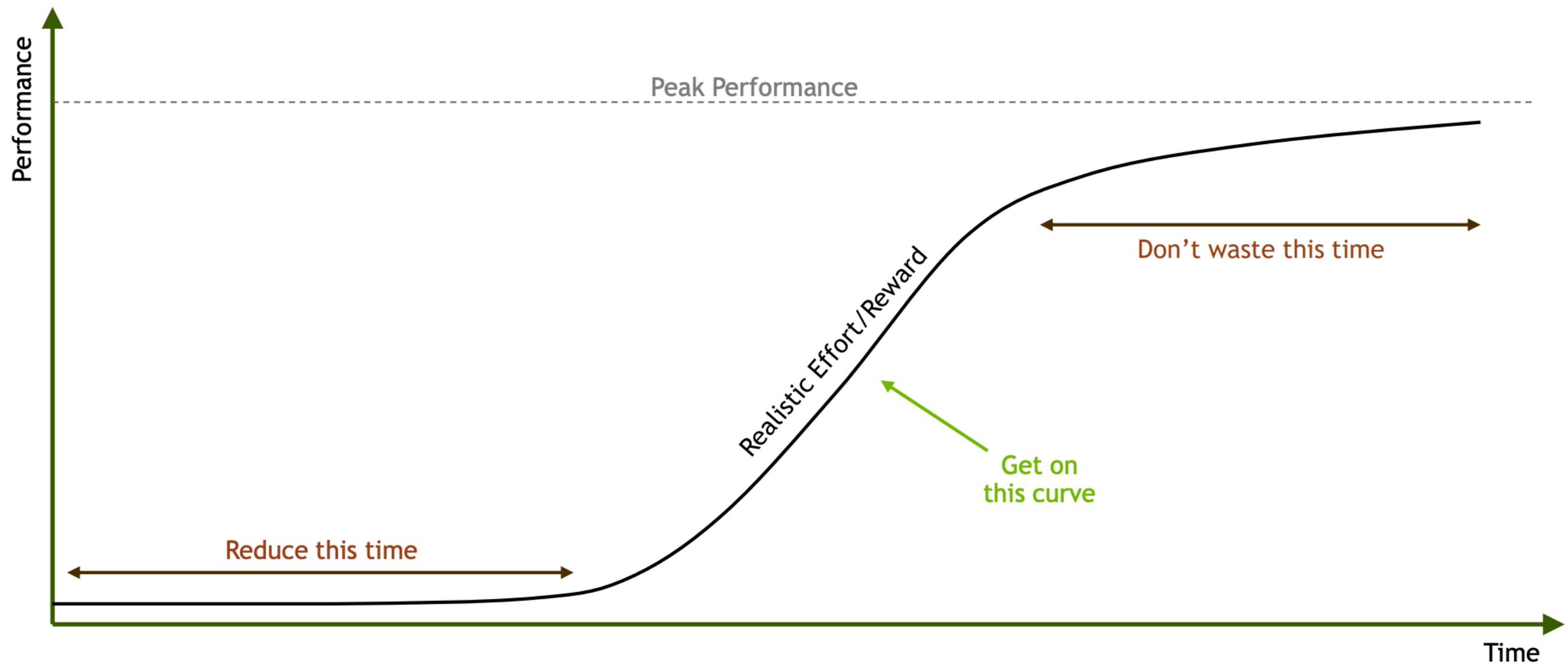
## ● 优化过程中应有的心态



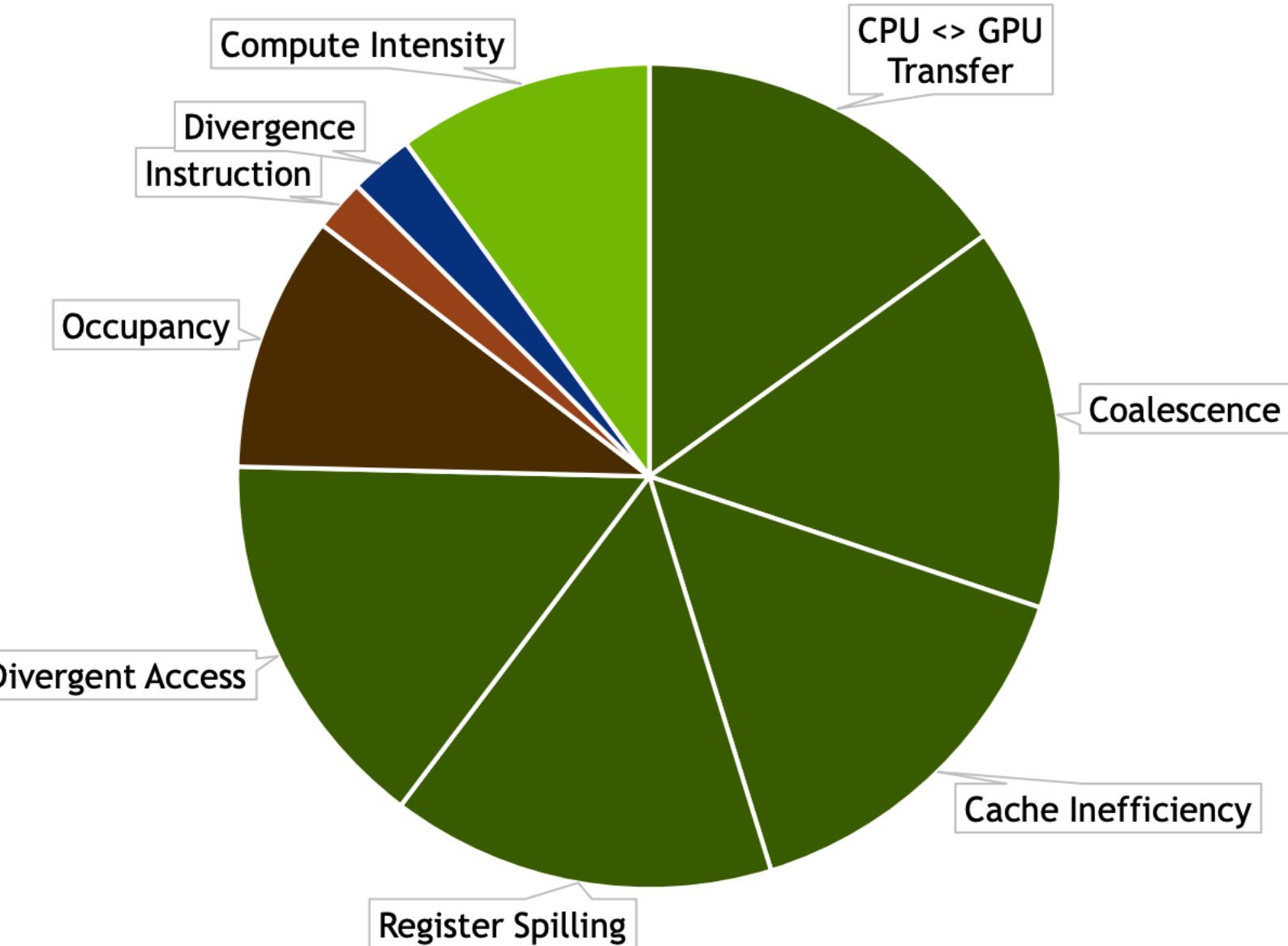
- 优化过程中应有的心态



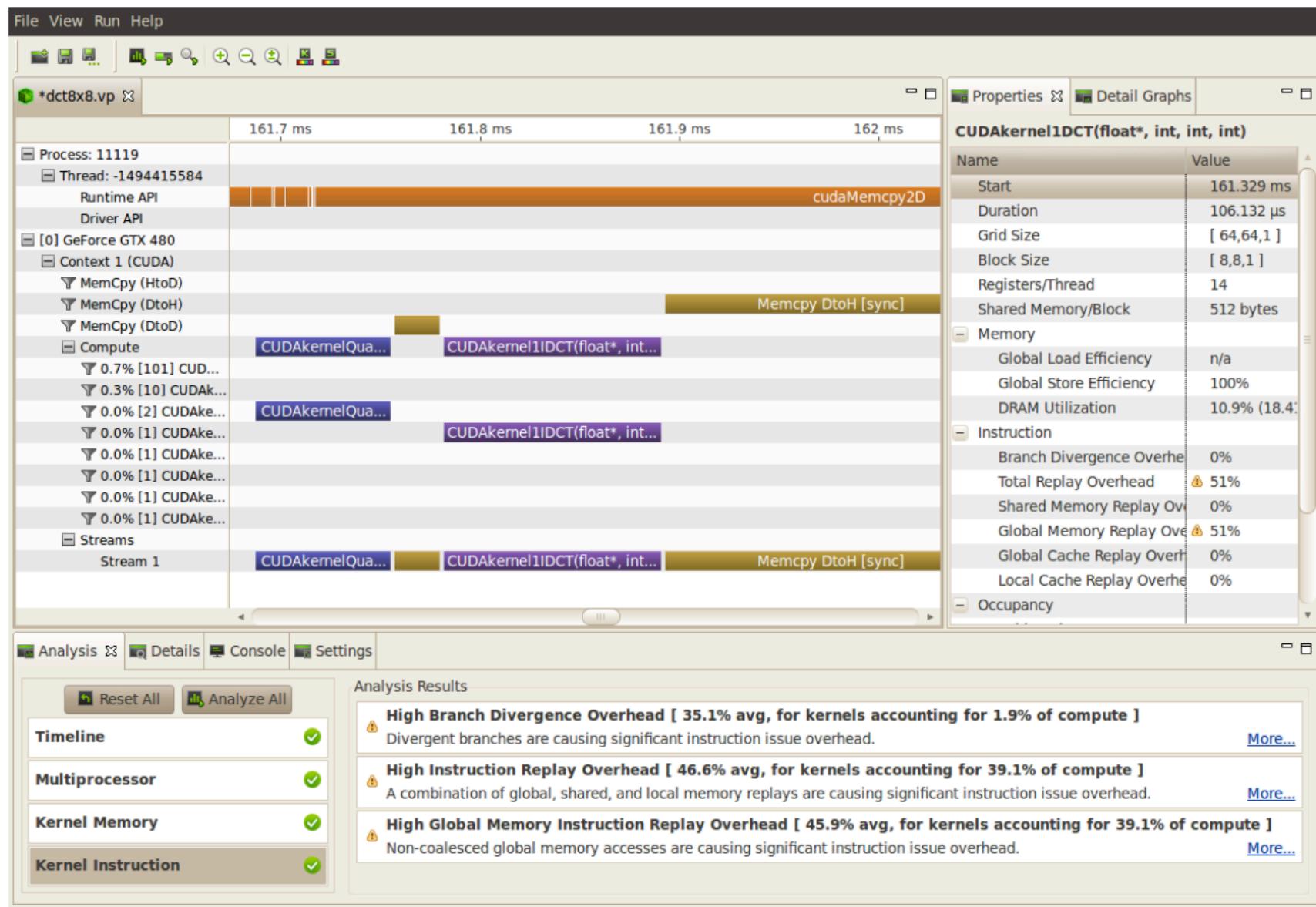
## ● 优化过程中应有的心态



## ● CUDA程序中的一般性能瓶颈



# ● 从哪儿入手？：Visual Profiler!



- 引言
- 优化全局内存访问
- 其他优化方法
- 优化实例



## ● CUDA是异构计算 (CPU+GPU) (讲义4)

- 针对计算任务选用合适的硬件
- CPU使用单个（或少数）线程完成控制流复杂的计算任务
  - 快速串行执行
  - 大缓存掩盖存储器延迟
- GPU使用大量线程完成控制流简单的计算任务
  - 线程高度轻量级（低创建、切换开销、单个线程速度较CPU慢）
  - 高带宽存储用于完成大量并发访问
- 需避免过于频繁地在CPU与GPU之间切换
  - 避免大量在CPU与GPU之间的内存拷贝



- 选择合适的内存（讲义5、6）
- 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/一级缓存
  - 重复计算有时优于存储器访问
- 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

## ● 线程束执行模式对性能的影响（讲义6）

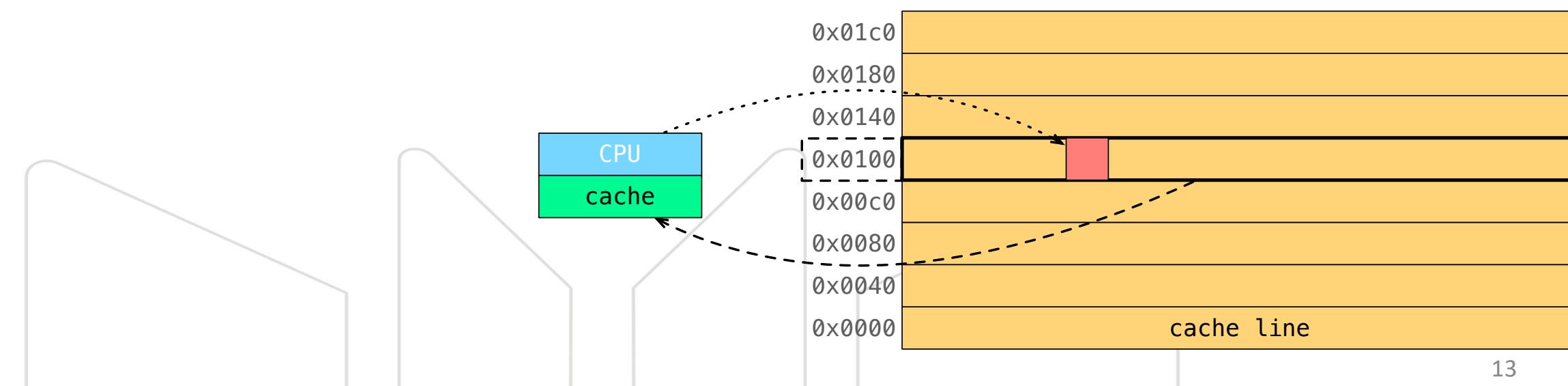
- SIMD：在控制流出现分支分化时，不同分支将串行执行
- 减少分支分化的影响
  - 减少判断语句
    - 尤其是减少基于threadIdx的判断语句
      - » 判断语句不必然导致分支分化
    - 使用条件语句代替条件赋值
  - 平衡分支执行时间
    - 避免出现执行时间过长的分支

## ● 原子操作对性能的影响（讲义6）

- 避免大量线程同时使用原子操作更新同一地址上的数据
  - 使用两级原子操作（线程块→全局）

## ● 对齐与合并访问: CPU

- 从CPU开始说起
- **cache line**: CPU每次从内存中缓存的数据大小
  - 通常为64B (字节)
  - 从64B的倍数起
- 每次访问内存时, 将加载包含该内存地址的一个cache line
  - 此后对同一cache line的访问将通过缓存进行



## ● 对齐与合并访问：CPU

- cache line意味着访问连续内存空间的重要性
  - C/C++中，数组为row-major（每行在内存中连续）
- 试比较以下代码：

```
#define N 50000
int mat[N][N];

for(int i=0; i<N; ++i){
    for (int j=0; j<N; ++j){
        sum += mat[i][j];
    }
}
```

```
#define N 50000
int mat[N][N];

for(int i=0; i<N; ++i){
    for (int j=0; j<N; ++j){
        sum += mat[j][i];
    }
}
```

## ● 对齐与合并访问：CPU

- cache line意味着访问连续内存空间的重要性
  - C/C++中，数组为row-major（每行在内存中连续）
- 试比较以下代码：

```
#define N 50000
int mat[N][N];

for(int i=0; i<N; ++i){
    for (int j=0; j<N; ++j){
        sum += mat[i][j];
    }
}
```

运行时间：5.414秒

```
#define N 50000
int mat[N][N];

for(int i=0; i<N; ++i){
    for (int j=0; j<N; ++j){
        sum += mat[j][i];
    }
}
```

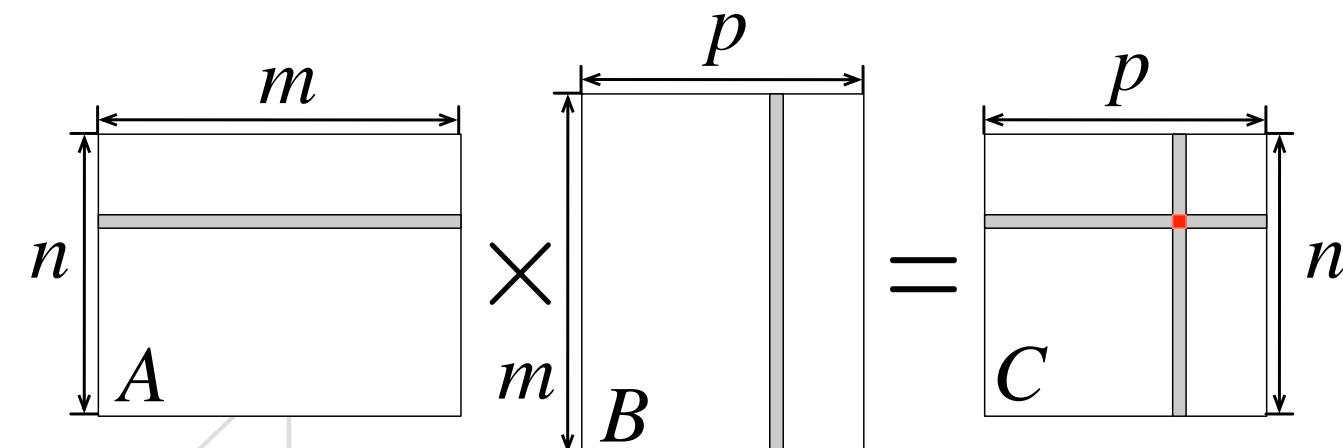
运行时间：35.708秒

## ● 对齐与合并访问：CPU

- cache line意味着访问连续内存空间的重要性
  - C/C++中，数组为row-major（每行在内存中连续）
- 在矩阵乘法中，常见写法容易不自觉地使用column-major的访问模式

```
#define N 3000
int A[N][N], B[N][N], C[N][N];

for(int i=0; i<N; ++i)
    for(int j = 0; j<N; ++j)
        for(int k = 0; k < N; ++k)
            C[i][j] += A[i][k]*B[k][j];
```



## ● 对齐与合并访问：CPU

- cache line意味着访问连续内存空间的重要性
  - C/C++中，数组为row-major（每行在内存中连续）
- 在矩阵乘法中，常见写法容易不自觉地使用column-major的访问模式

```
#define N 3000
int A[N][N], B[N][N], C[N][N];

for(int i=0; i<N; ++i)
    for(int j = 0; j<N; ++j)
        for(int k = 0; k < N; ++k)
            C[i][j] += A[i][k]*B[k][j];
```

运行时间：231.348秒

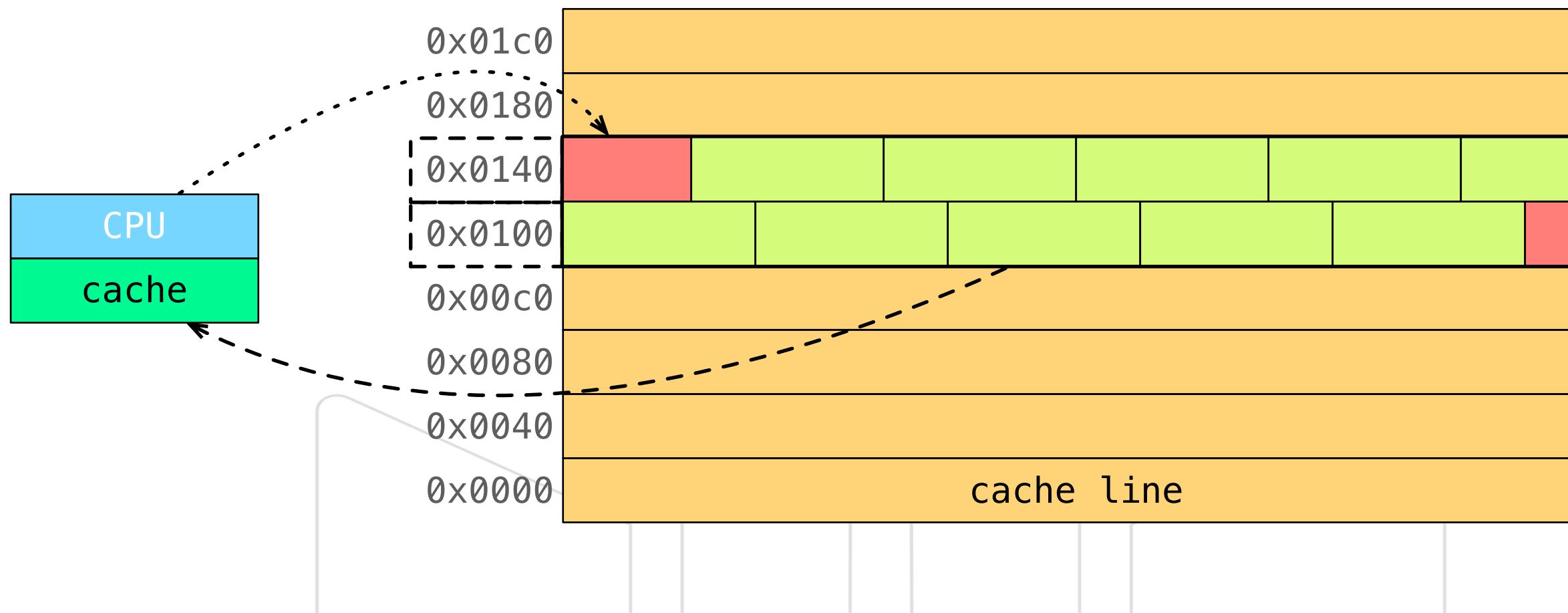
```
#define N 3000
int A[N][N], B[N][N], C[N][N];

for(int i=0; i<N; ++i)
    for(int k = 0; k<N; ++k)
        for(int j = 0; j<N; ++j)
            C[i][j] += A[i][k]*B[k][j];
```

运行时间：72.557秒

## ● 对齐与合并访问：CPU

- 非对齐存储可能导致对一个数据的读取需要载入两个cache lines
  - 使用`_align_(n)`强制对齐
  - n必须为2的幂次方



- 对齐与合并访问：全局内存读取

- 三种缓存路径
  - 一级和二级缓存
  - 常量缓存
  - 只读缓存
- 默认为一级和二级缓存
  - 是否通过一级缓存加载取决于设备的计算能力及编译器选项
- 常量缓存和只读缓存需要在程序中显示说明（讲义5）
  - `__constant__`、`__ldg()`、`const __restrict__`



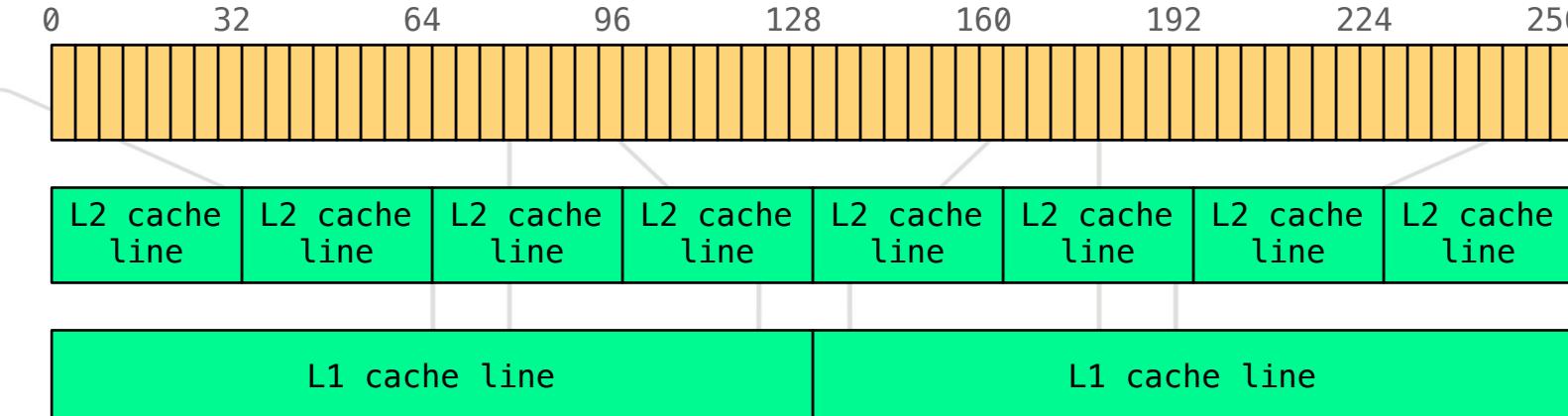
## ● 对齐与合并访问：全局内存读取

### — 一级缓存

- 128B的cache line宽度
- 通常用于缓存寄存器溢出到本地内存中的数据
- 可以用来进行全局内存加载（如，通过只读缓存）
- 某些硬件上可以启用一级缓存用于所有全局内存加载
  - 仍然需要先通过二级缓存
  - 需要确定设备支持启用一级缓存加载全局内存

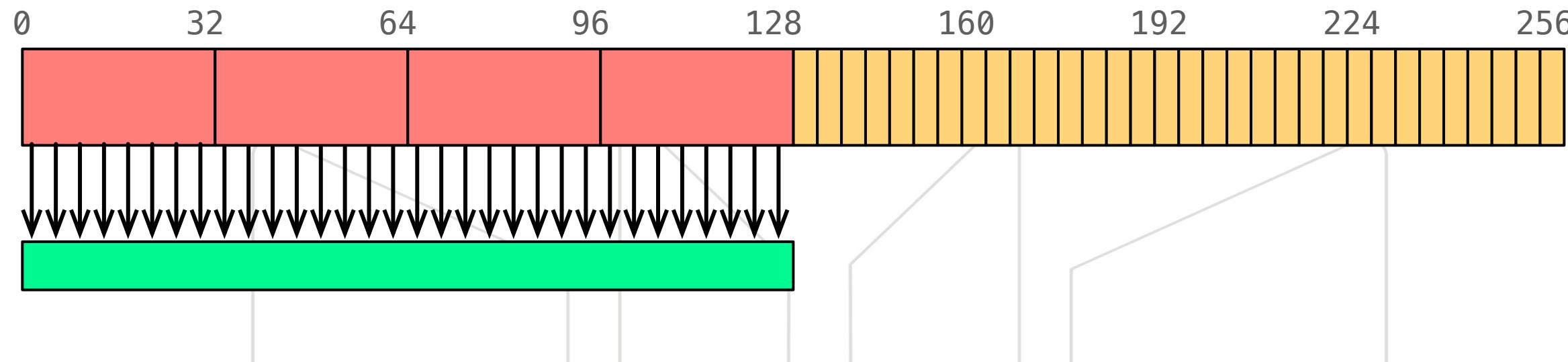
### — 二级缓存

- 32B的cache line宽度
- 所有全局内存访问都需要通过二级缓存



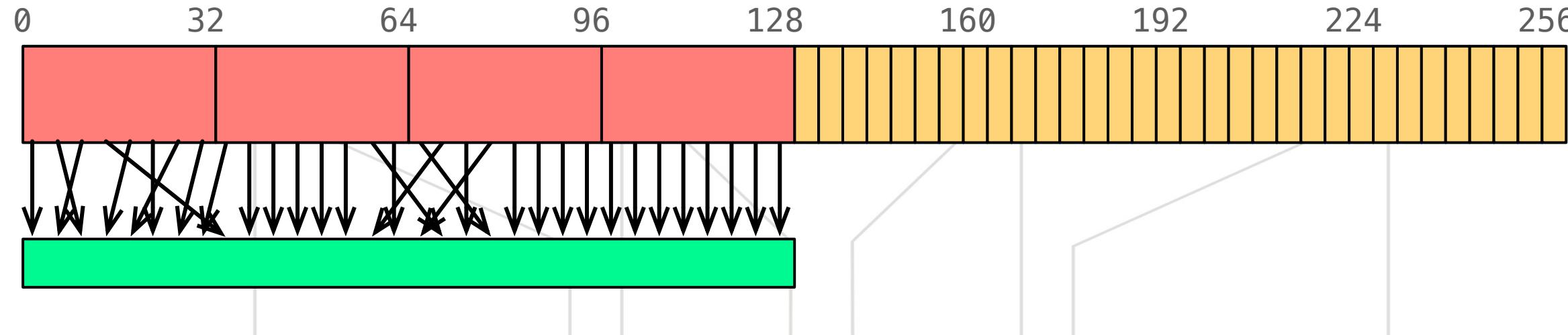
- 通过二级缓存的合并访问

- 线程束对全局内存的访问将被合并为**内存事务**进行
- 理想情况：对齐与合并内存访问
  - 线程束所有内存请求引用地址连续
    - 假设每个线程请求4字节的数据（整型、单精度浮点型等），完成加载操作只需4个32字节的内存事务
    - 总线利用率100%



## 通过二级缓存的合并访问

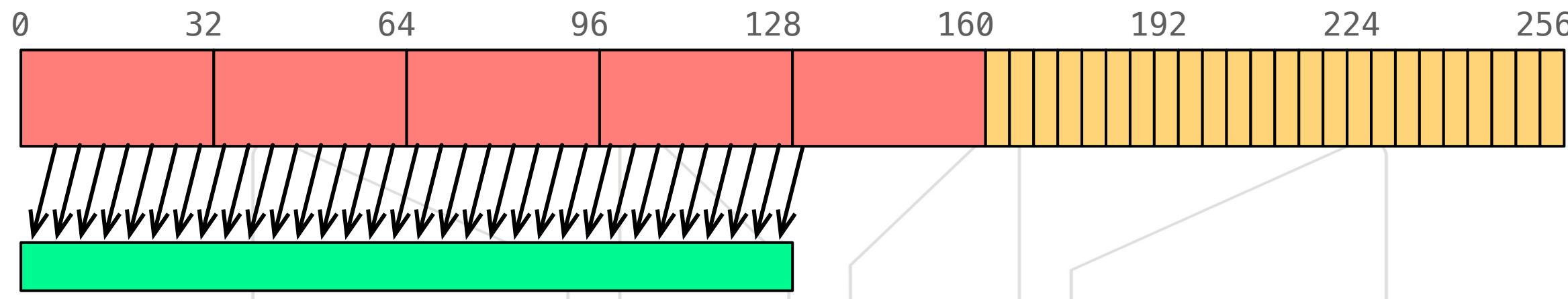
- 线程束对全局内存的访问将被合并为**内存事务**进行
- 对齐访问，但引用地址不连续
  - 假设每个线程请求4字节的数据（整型、单精度浮点型等）
  - 只要线程引用地址均落在同样的128字节段内，并且无重复（跨过32字节段边界）
  - 完成加载操作仍然只需4个32字节的内存事务
  - 总线利用率仍然为100%



## 通过二级缓存的合并访问

- 线程束对全局内存的访问将被合并为**内存事务**进行
- 非对齐访问，引用地址连续
  - 假设每个线程请求4字节的数据（整型、单精度浮点型等）
  - 线程引用地址均落在5个32字节
  - 完成加载操作需要5个32字节的内存事务
  - 总线利用率为80%

$$\text{利用率} = \frac{\text{请求的全局内存加载吞吐量}}{\text{所需的全局内存加载吞吐量}}$$

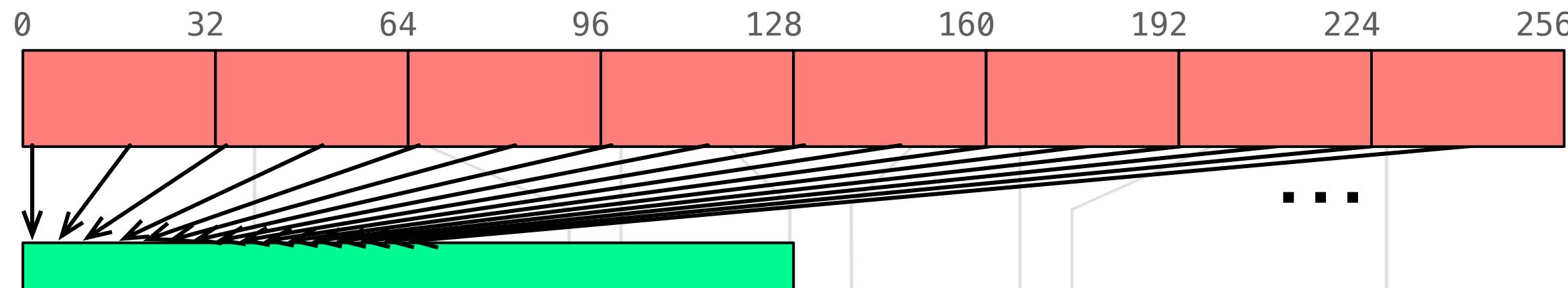


- 通过二级缓存的合并访问

- 线程束对全局内存的访问将被合并为**内存事务**进行
- 固定步长访问

```
__global__ void kernel(int* data){  
    int tid = blockIdx.x * blockDim.x + threadIdx.x;  
    do_something(data[tid*stride]);  
}
```

- $stride=4$ 时，完成加载需16次内存事务，总线利用率为1/16
- 最差情况需32次内存事务才能完成加载！



## 通过二级缓存的合并访问

### – Array of Structures (AoS) vs Structure of Arrays (SoA)

- AoS: 数组中每个元素为一个结构体
  - 如, 三维点集

```
typedef struct {
    float x, y, z;
} point;

__global__ void kernel(point* data) {
    int tid = blockIdx.x*blockDim.x+threadIdx.x;

    float x = data[tid].x;
    float y = data[tid].y;
    float z = data[tid].z;

    do_something(x, y, z);
}
```



- 通过二级缓存的合并访问

- Array of Structures (AoS) vs Structure of Arrays (SoA)

- AoS: 数组中每个元素为一个结构体
      - 如, 三维点集

使用

```
typedef struct __align__(16) {  
    float x, y, z;  
} point;
```

```
typedef struct {  
    float x, y, z;  
} point;  
  
__global__ void kernel(point* data) {  
    int tid = blockIdx.x*blockDim.x+threadIdx.x;  
  
    float x = data[tid].x;  
    float y = data[tid].y;  
    float z = data[tid].z;  
  
    do_something(x, y, z);  
}
```

单个元素占用12个字节, 容易导致非对齐访问

等价于stride=3的固定步长访问!



## 通过二级缓存的合并访问

### – Array of Structures (AoS) vs Structure of Arrays (SoA)

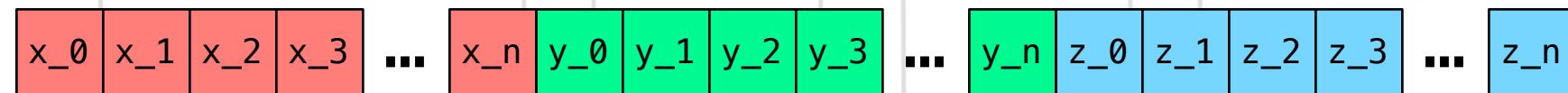
- SoA: 结构体每个成员为一个数组
  - 例如, 同样用于表示三维点集时

```
typedef struct {
    float x[N], y[N], z[N];
} points;

__global__ void kernel(points data){
    int tid = blockIdx.x*blockDim.x+threadIdx.x;

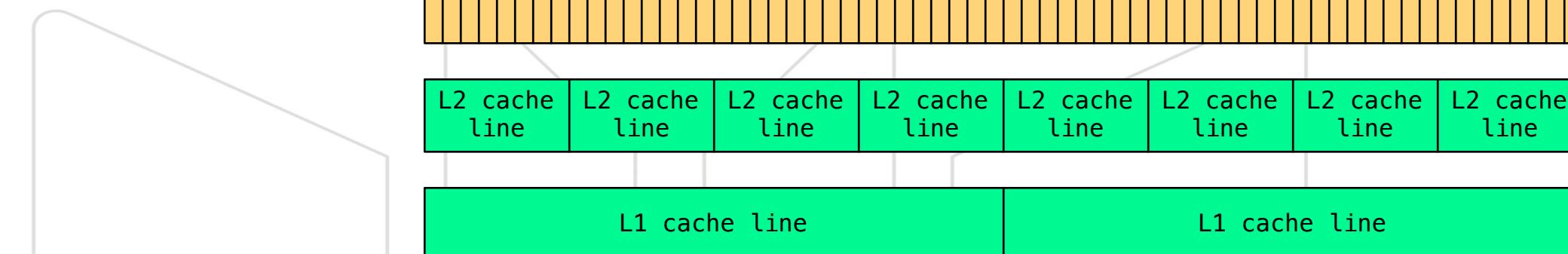
    float x = data.x[tid];
    float y = data.y[tid];
    float z = data.z[tid];

    do_something(x, y, z);
}
```



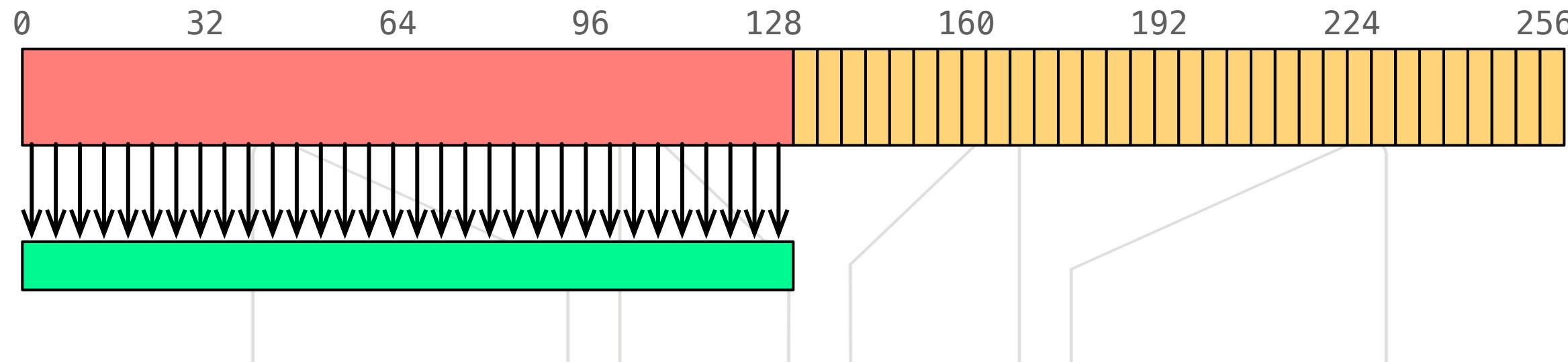
## 通过一级缓存的合并访问

- 一级缓存有着更长的cache line
  - 好处：可能合并相邻线程束的访问
  - 坏处：可能载入不必要的信息，非合并访问的性能可能更差
- 使用一级缓存进行全局加载
  - 启用：在编译时使用-Xptxas -dlcm=ca flag
  - 禁用：在编译时使用-Xptxas -dlcm=cg flag
  - 通过检查属性globalL1CacheSupported与localL1CacheSupported，可知设备是否支持此功能



- 通过一级缓存的合并访问

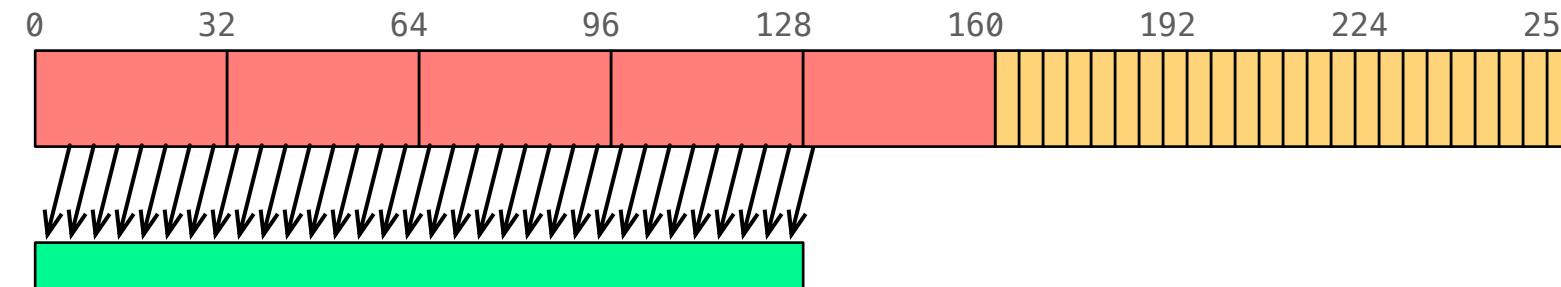
- 与二级缓存的合并访问相似
- 理想情况：对齐与合并内存访问
  - 线程束所有内存请求引用地址连续
    - 假设每个线程请求4字节的数据（整型、单精度浮点型等），完成加载操作只需1个128字节的内存事务
    - 总线利用率100%



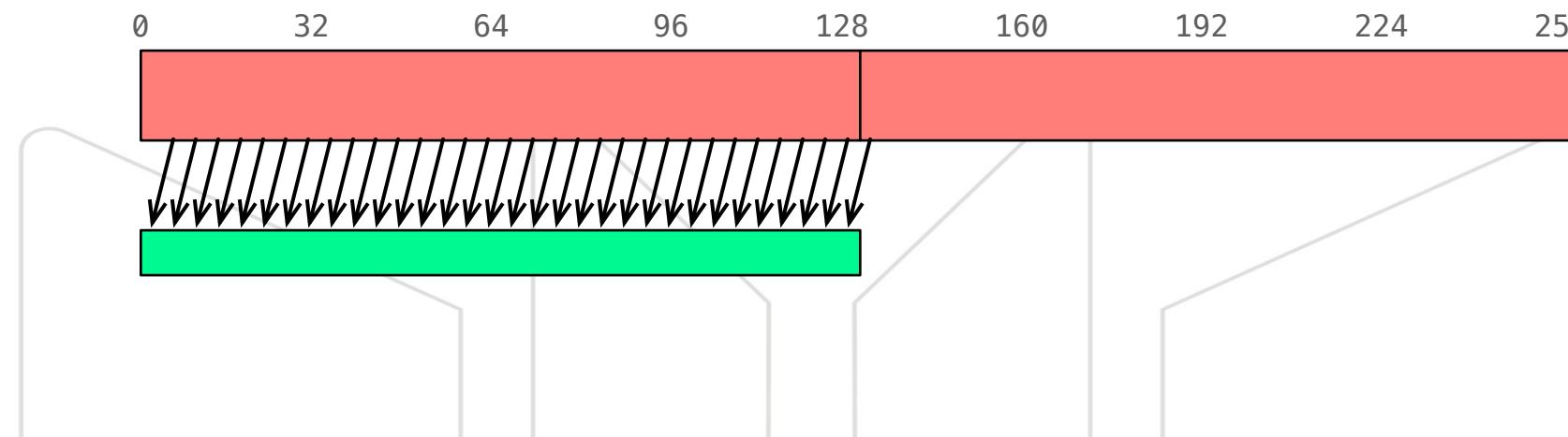
- 通过一级缓存的合并访问

- 与二级缓存的合并访问相似
- 非对齐访问时利用率可能更低！

- 使用二级缓存访问时效率为80% (4/5)



- 使用一级缓存访问时效率仅为50% (1/2)



- CPU一级缓存和GPU一级缓存的差异
  - CPU一级缓存优化时间和空间局部性
  - GPU一级缓存只优化空间局部性
    - 频繁访问一个一级缓存中的内存位置不会增加数据留在缓存中的概率
- 全局内存写入
  - 写入只能通过二级缓存进行
    - 在32个字节段的粒度上执行
    - 内存事务可被分为一段（32B）、二段（64B）或四段（128B）执行
    - 执行一个二段事务效率优于两个一段事务

- 优化全局内存访问
- 其他优化方法
- 优化实例



## ● 占用率 (Occupancy)

– 表明SM的使用状况：

$$\text{占用率} = \frac{\text{活跃线程束数量}}{\text{最大活跃线程束数量}}$$

– 占用率受以下因素影响

- 寄存器数量、共享内存使用量、block中的线程数限制

– 增加占用率的影响

- 好处：掩盖存储器延迟

– 线程等待全局内存载入数据时，切换到其他warp是隐藏延时和保持GPU运转的有效办法

- 缺点：为增加占用率减小共享内存、寄存器可能导致缓存效率降低

- 应在占用率及分配资源间寻找平衡

## ● 占用率 (Occupancy)

### – 优化占用率的原则

- 基本的方式：每个SM至少有一个线程块可以执行
- 更优的方式：每个SM有多个线程块可以执行
- 每个线程块占用SM资源不宜过高（最好是一半以下）
  - 寄存器、共享内存等
  - 允许线程块在SM上并发运行
    - » 减小所有线程共同等待 `syncthreads()` 的概率
- 线程块内线程数量应该是warp size (32) 的整数倍
  - 避免出现under-populated warps

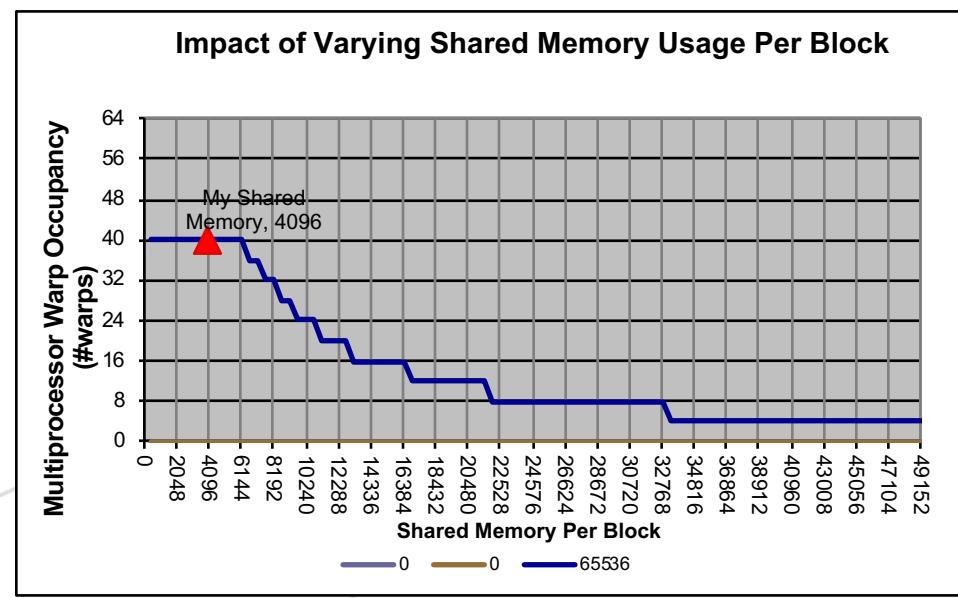
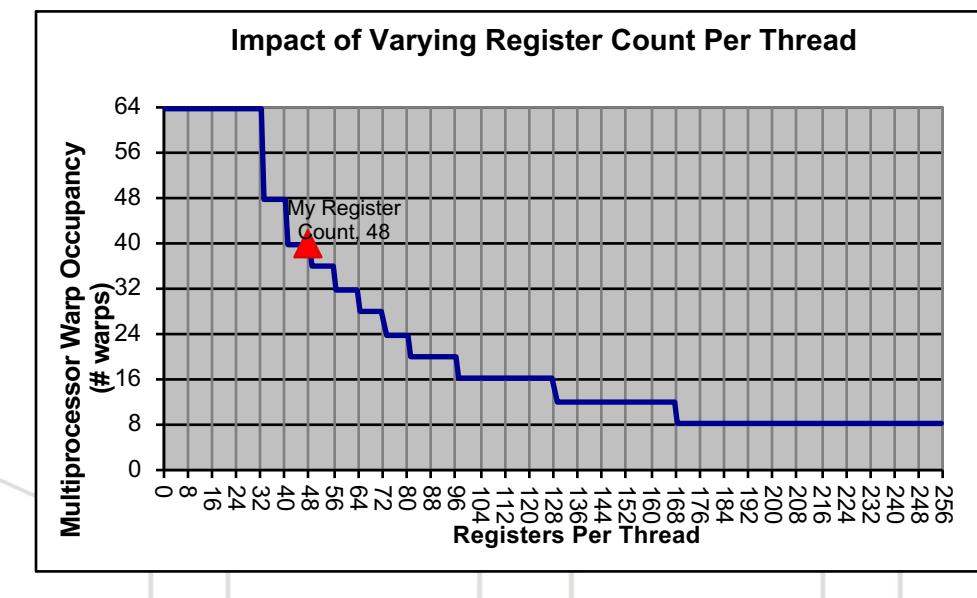
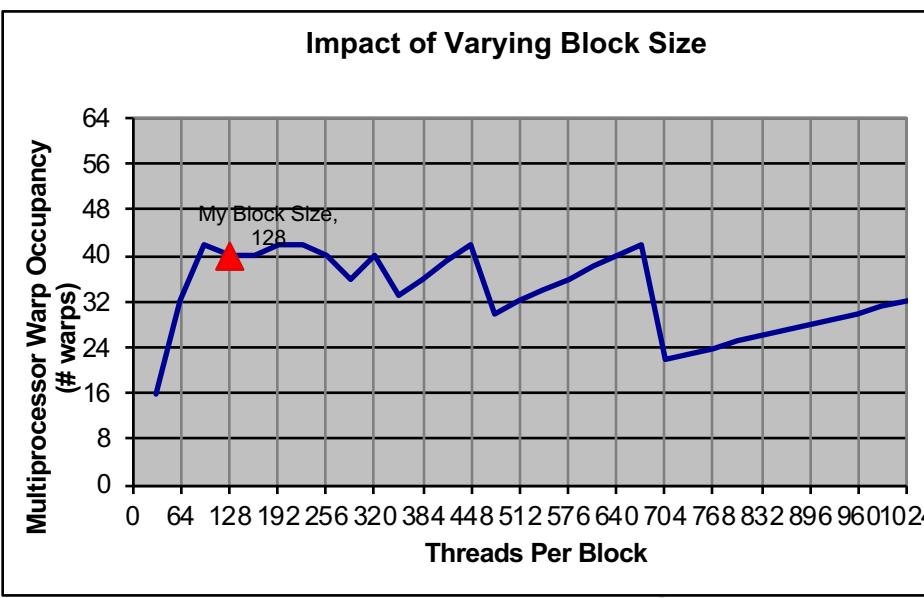


## ● 计算占用率

– NVIDIA官方计算器（支持至计算能力6.2）

- [http://developer.download.nvidia.com/compute/cuda/CUDA\\_Occupancy\\_calculator.xls](http://developer.download.nvidia.com/compute/cuda/CUDA_Occupancy_calculator.xls)

- 输入：计算能力、共享内存大小、全局内存缓存模式（L1/L2），每个 block 中的线程数、每个线程的寄存器数目、每个线程块的共享内存大小



- 自动决定线程块大小 (CUDA 6.5以后)

- 使用函数：

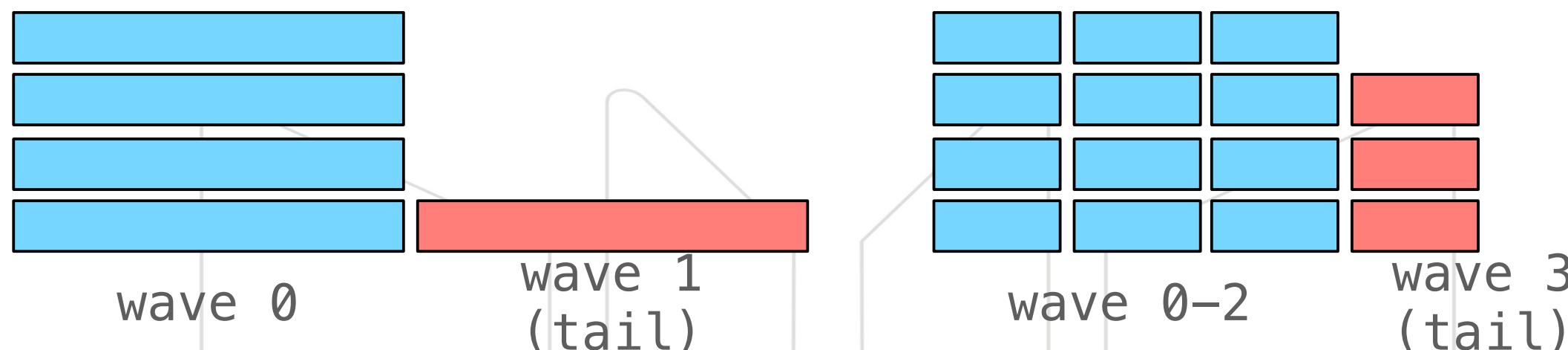
```
__host__ cudaError_t cudaOccupancyMaxPotentialBlockSize ( int* minGridSize,  
int* blockSize, T func, size_t dynamicSMemSize, int blockSizeLimit) [inline]
```

- `minGridSize`: 返回达到最优占用率所需的最小网格大小
- `blockSize`: 返回达到最优占用率的线程块大小
- `func`: 优化目标的核函数
- `dynamicSMemSize`: 动态内存大小
- `blockSizeLimit`: 核函数能支持的最大线程块大小



## ● Waves and Tails

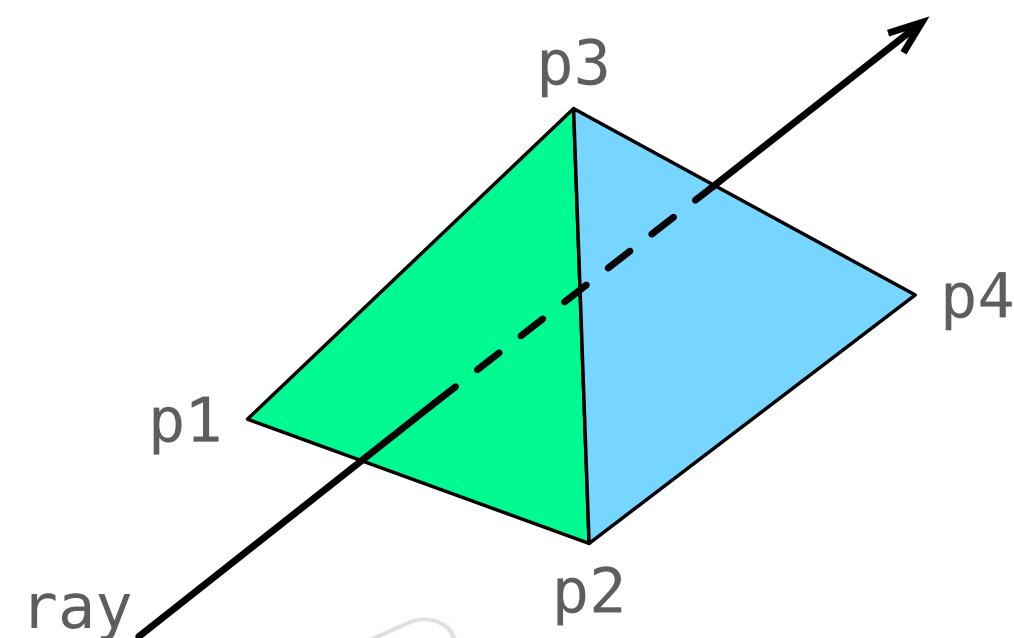
- <http://on-demand.gputechconf.com/gtc/2012/presentations/S0514-GTC2012-GPU-Performance-Analysis.pdf>
- Wave: 在设备上同时并行的线程块
- Tail: 处理无法被整除的工作量中余下部分的线程块
- 意义: 过大的线程块可能无法充分利用硬件资源
  - 但过小的线程块可能限制利用率
    - SM上同时活跃的线程块数量有限 (Kepler为16, Maxwell为32)



## ● 提取共同任务

- 若不同分支中完成相同任务，尝试将其从分支中抽离出来
- 例子：

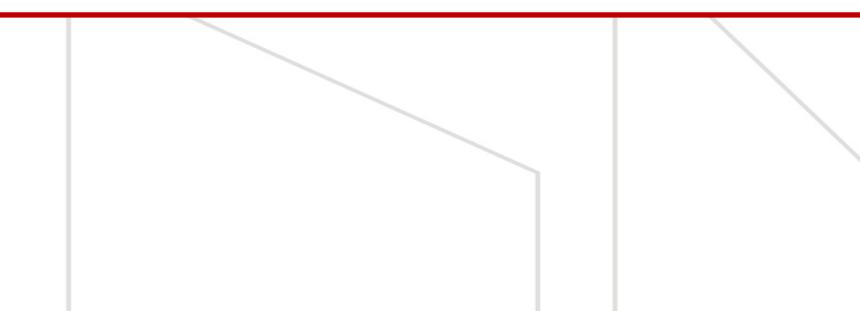
```
__global__ void kernel(vector* rays, quad* quads){  
    int tid = blockIdx.x*blockDim.x+threadIdx.x;  
    vector ray = rays[tid];  
    quad q = quads[tid]; // p1, p2, p3, p4  
  
    if (ray_hit_triangle_1){  
        do_something(q.p1, q.p2, q.p3);  
    } else {  
        do_something(q.p2, q.p3, q.p4);  
    }  
}
```



## ● 提取共同任务

- 若不同分支中完成相同任务，尝试将其从分支中抽离出来
- 例子：

```
__global__ void kernel(vector* rays, quad* quads){  
    int tid = blockIdx.x*blockDim.x+threadIdx.x;  
    vector ray = rays[tid];  
    quad q = quads[tid]; // p1, p2, p3, p4  
  
    if (ray_hit_triangle_1){  
        do_something(q.p1, q.p2, q.p3);  
    } else {  
        do_something(q.p2, q.p3, q.p4);  
    }  
}
```



```
__global__ void kernel(vector* rays, quad* quads){  
    int tid = blockIdx.x*blockDim.x+threadIdx.x;  
    vector ray = rays[tid];  
    quad q = quads[tid]; // p1, p2, p3, p4  
  
    point p;  
    if (ray_hit_triangle_1){  
        p = q.p1;  
    } else {  
        p = q.p4;  
    }  
    do_something(p, q.p2, q.p3);  
}
```

- 展开循环 (loop unrolling)

- 使用编译器优化

- gcc -funroll-loops
    - 某些编译器的-O3级别优化同样会展开循环

- 手动展开

```
for(int i=0; i<n; ++i){  
    do_something(i);  
}
```

```
for(int i=0; i<n; ++i){  
    do_something(i); ++i;  
    do_something(i); ++i;  
    do_something(i); ++i;  
    do_something(i);
```

- 展开循环 (loop unrolling)

- 使用编译器优化

- gcc -funroll-loops
    - 某些编译器的-O3级别优化同样会展开循环

- 手动展开 (并行归约的最后6次迭代)

```
for (int stride=blockDim.x/2; stride>0; stride>>=1){  
    if (tid<stride){  
        sdata[tid] += sdata[tid+stride];  
    }  
    __syncthreads();  
}
```

```
if (tid<32){  
    volatile int* vdata = sdata;  
    vdata[tid] += vdata[tid+32];  
    vdata[tid] += vdata[tid+16];  
    vdata[tid] += vdata[tid+8];  
    vdata[tid] += vdata[tid+4];  
    vdata[tid] += vdata[tid+2];  
    vdata[tid] += vdata[tid+1];  
}
```

## ● 串行程序优化

- Michael E. Lee, Optimization of Computer Programs in C
  - [http://icps.u-strasbg.fr/~bastoul/local\\_copies/lee.html](http://icps.u-strasbg.fr/~bastoul/local_copies/lee.html)
- Lucas A. Wilson, Serial & Vector Optimization
  - [https://portal.tacc.utexas.edu/documents/13601/1041435/06-Serial\\_and\\_Vector\\_Optimization.pdf/4eef1e1c-7592-4ac4-8608-1f0662553a88](https://portal.tacc.utexas.edu/documents/13601/1041435/06-Serial_and_Vector_Optimization.pdf/4eef1e1c-7592-4ac4-8608-1f0662553a88)



- 优化全局内存访问
- 其他优化方法
- 优化实例



## ● 问题定义

- 将输入矩阵 i 行 j 列上的元素置于输出矩阵的 j 行 i 列
- CPU 代码：

```
void transpose(int* out, int* in, int n, int m){  
    for(int y = 0; y < n; ++y){  
        for(int x = 0; x < m; x++){  
            out[x*n+y] = in[y*m+x];  
        }  
    }  
}
```

- 存在问题？

- 回顾：访问连续内存空间的重要性

- CPU上的cache line及GPU上的合并对齐访问

- 数组为row-major（每行在内存中连续）
- Column-major访问将极大地影响效率

```
#define N 50000
int mat[N][N];

for(int i=0; i<N; ++i){
    for (int j=0; j<N; ++j){
        sum += mat[i][j];
    }
}
```

运行时间：5.414秒

```
#define N 50000
int mat[N][N];

for(int i=0; i<N; ++i){
    for (int j=0; j<N; ++j){
        sum += mat[j][i];
    }
}
```

运行时间：35.708秒

## ● 问题定义

- 将输入矩阵 i 行 j 列上的元素置于输出矩阵的 j 行 i 列
- CPU 代码：

```
void transpose(int* out, int* in, int n, int m){  
    for(int y = 0; y < n; ++y){  
        for(int x = 0; x < m; x++){  
            out[x*n+y] = in[y*m+x];  
        }  
    }  
}
```

访问内存空间不连续：  
无法通过交换内外层循环  
优化访问模式！

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

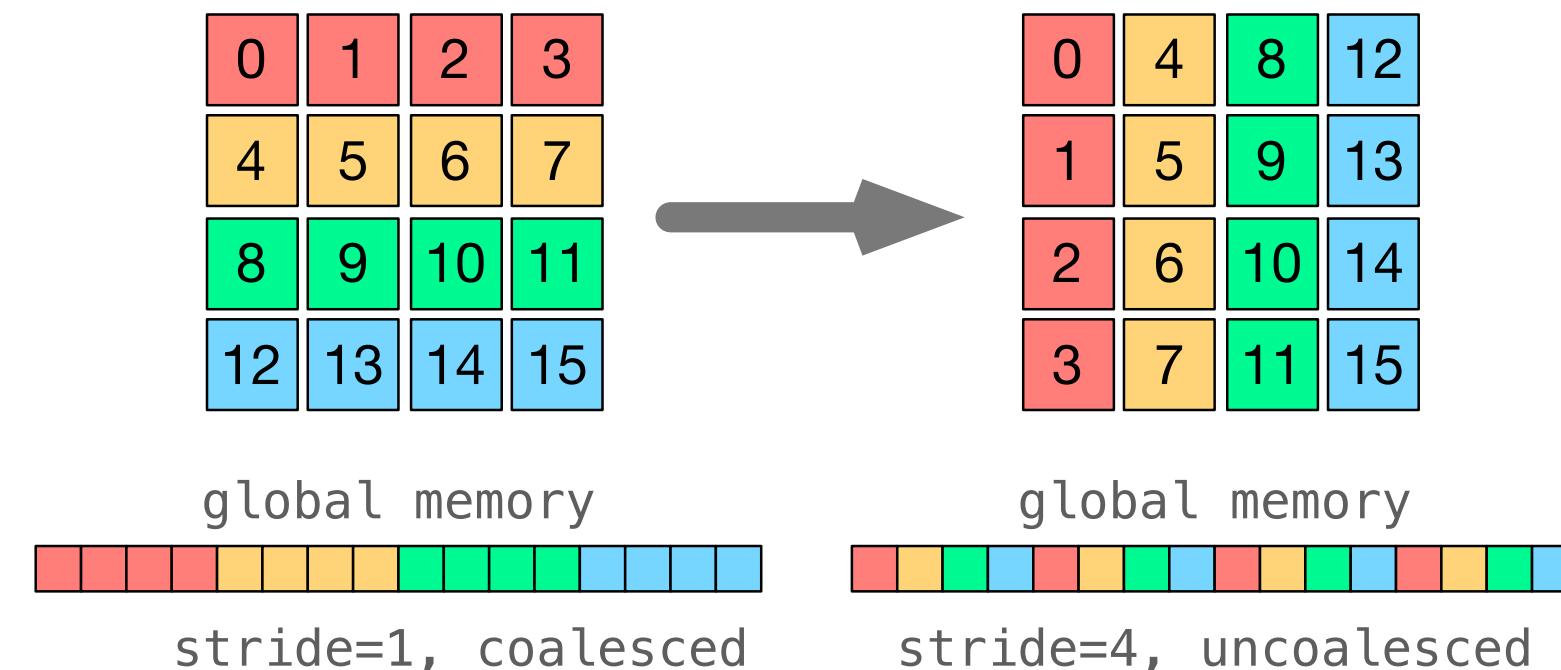
0	4	8	12
1	5	9	13
2	6	10	14
3	7	11	15

## ● GPU转置

### - 直接思路：

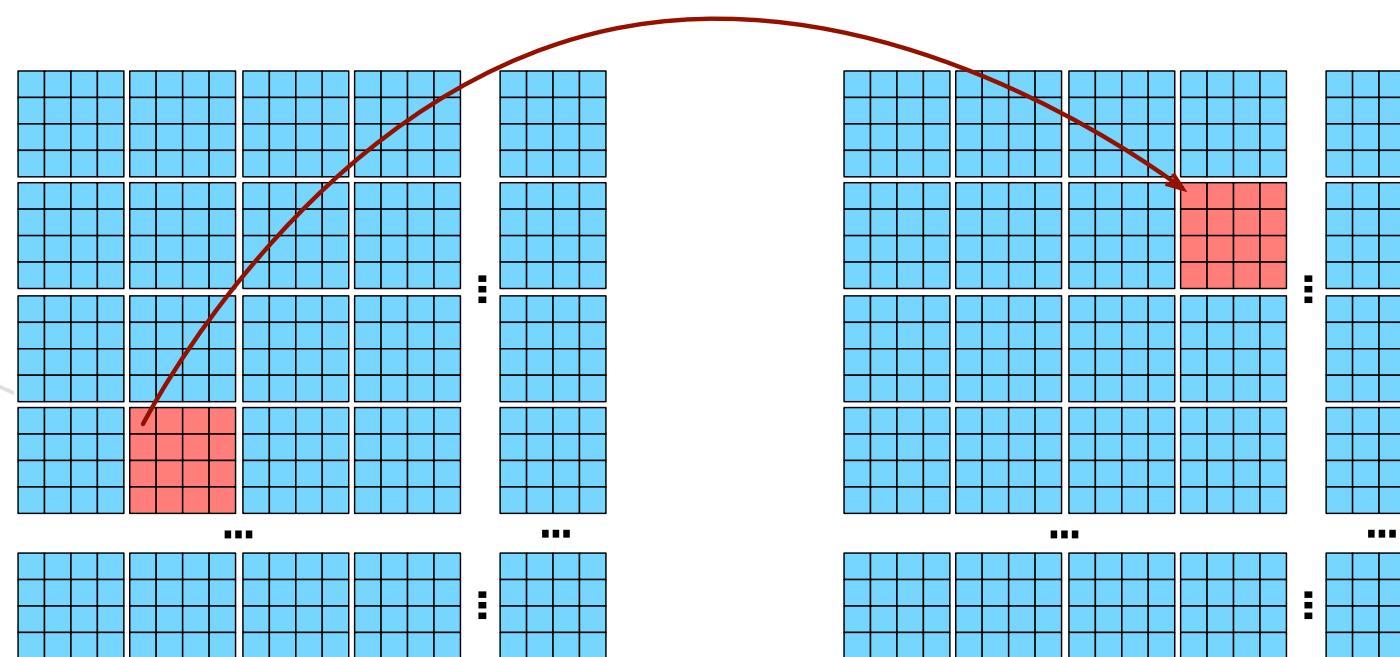
- 存在非合并存储！（无法通过交换x, y）解决

```
__global__ void transpose(int * out, int* in, int n, int m){  
    int x = blockIdx.x*blockDim.x+threadIdx.x;  
    int y = blockIdx.y*blockDim.y+threadIdx.y;  
  
    out[x*n+y] = in[y*m+x];  
}
```



- 使用共享内存的CUDA矩阵转置

- 将矩阵划分为数据块
- 每个线程块处理一个数据块的转置
- 从全局内存中读入第  $(i, j)$  个数据块至共享内存
- 从共享内存中写出至第  $(j, i)$  个数据块



## ● 使用共享内存的CUDA矩阵转置

### - 数据块内部转置

- 元素 $(i, j) \rightarrow (j, i)$

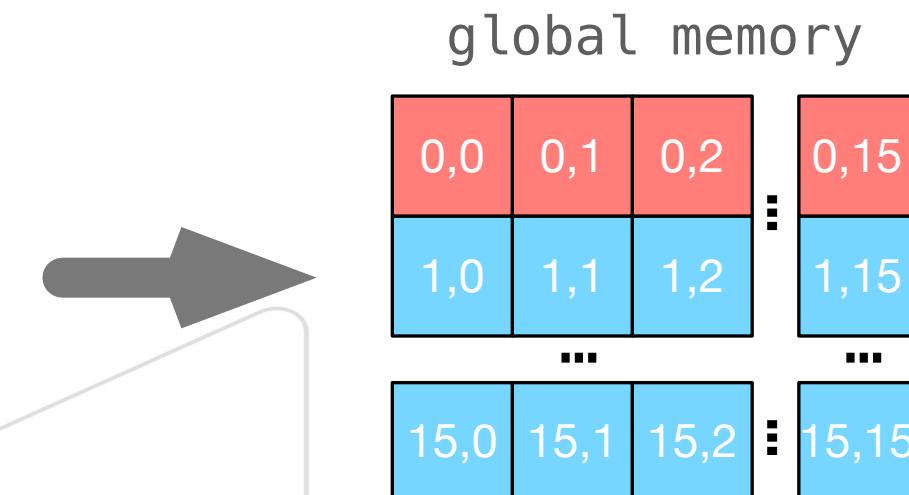
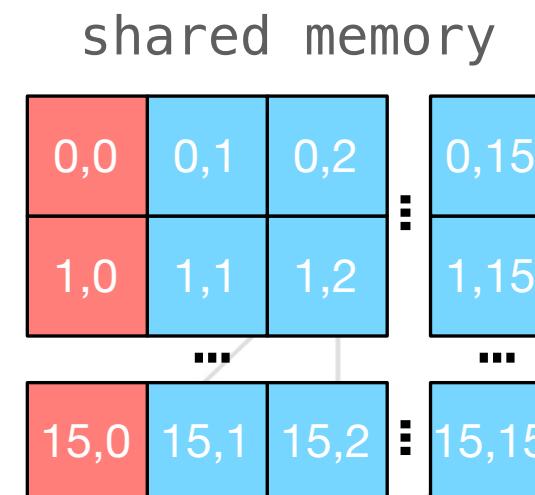
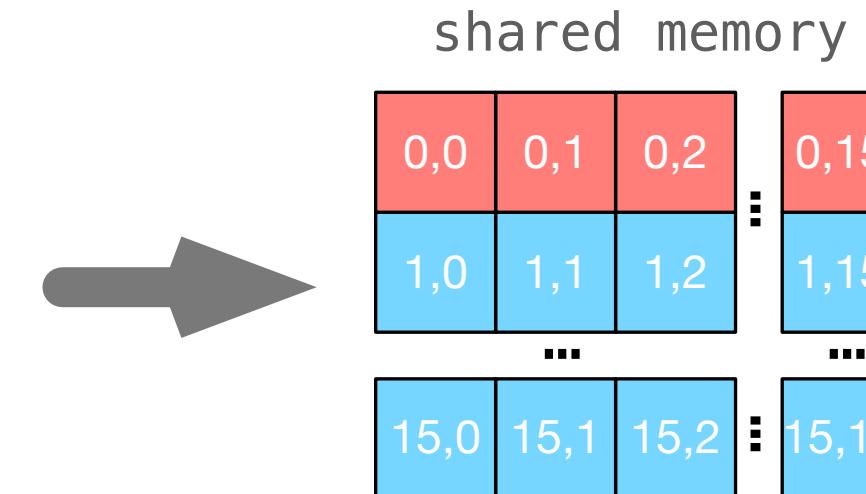
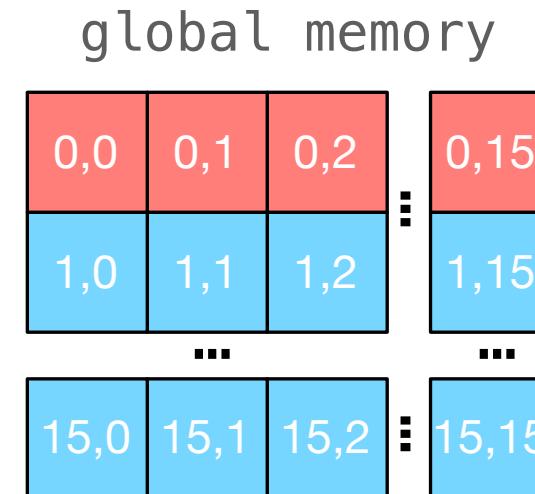
### - 读写全局内存均为合并存储

- 线程块宽度为16倍数

### - 共享内存访问模式

- 不受合并存储影响

- 存储体冲突？

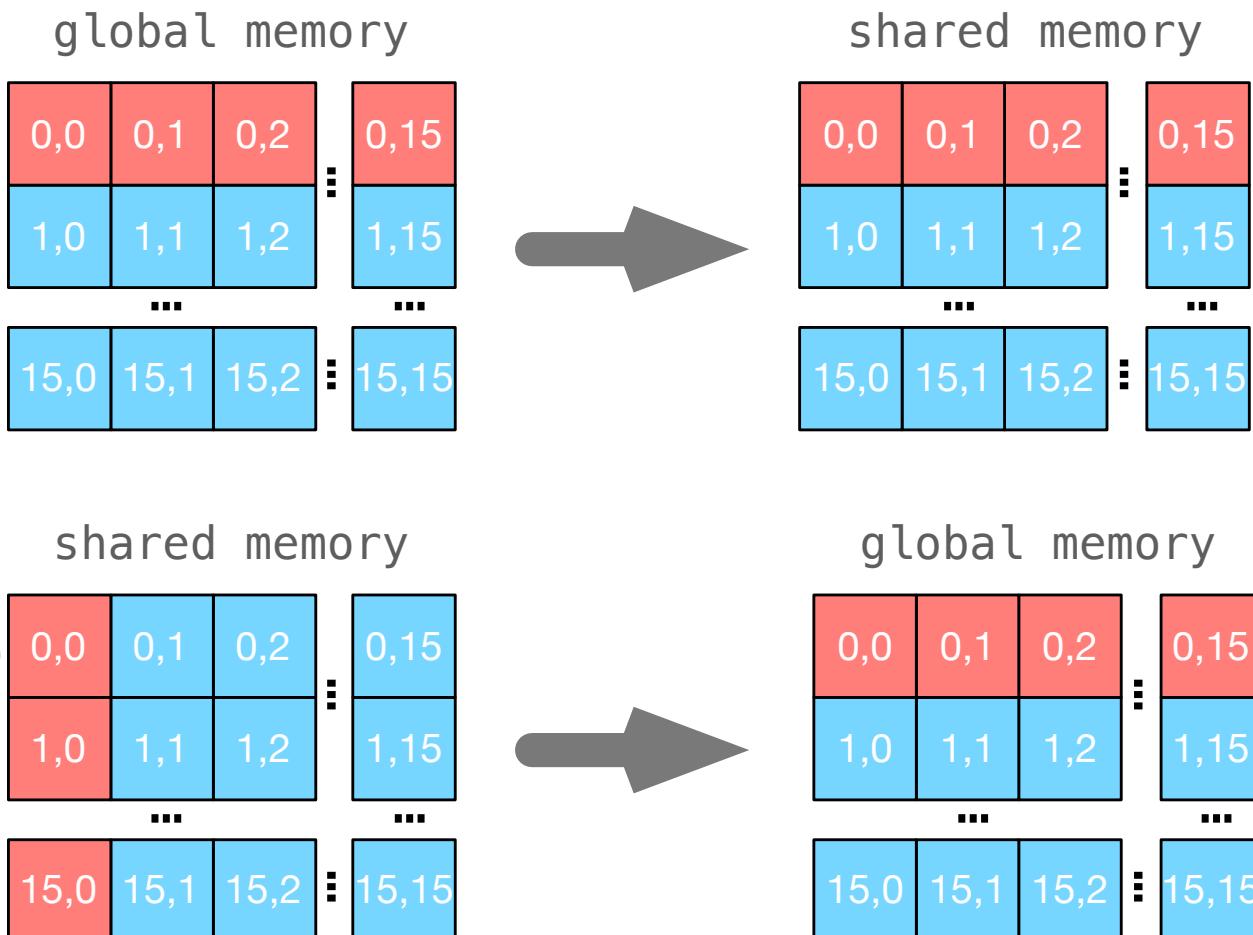


## ● 使用共享内存的CUDA矩阵转置

```
__global__ void transpose(int* out, int* in, int n, int m){  
    __shared__ int smem[BDIM*BDIM];  
  
    int bx = blockIdx.x*blockDim.x;  
    int by = blockIdx.y*blockDim.y;  
    int tx = threadIdx.x;  
    int ty = threadIdx.y;  
  
    smem[ty*BDIM+tx] = in[(by+ty)*m+bx+tx];  
    __syncthreads();  
  
    out[(bx+ty)*n+by+tx]=smem[tx*BDIM+ty];  
}
```

是否可以使用：

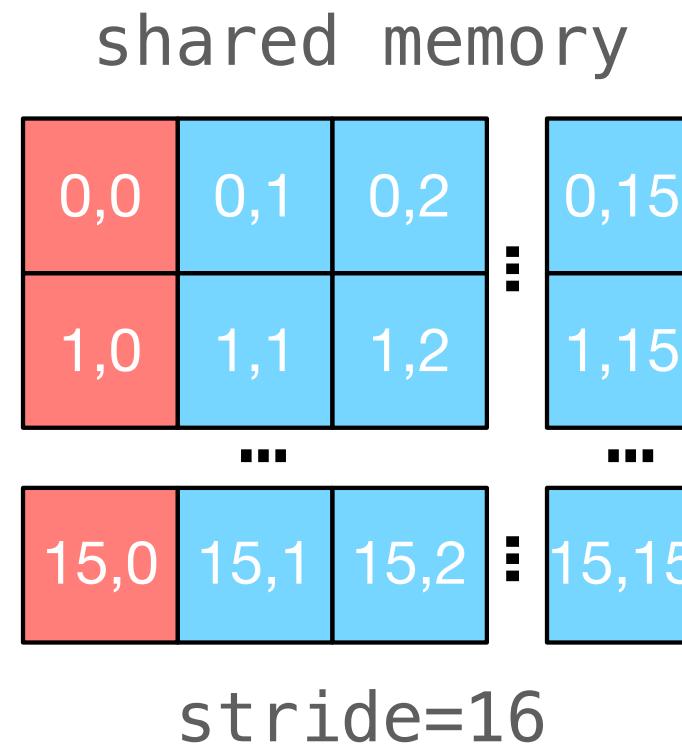
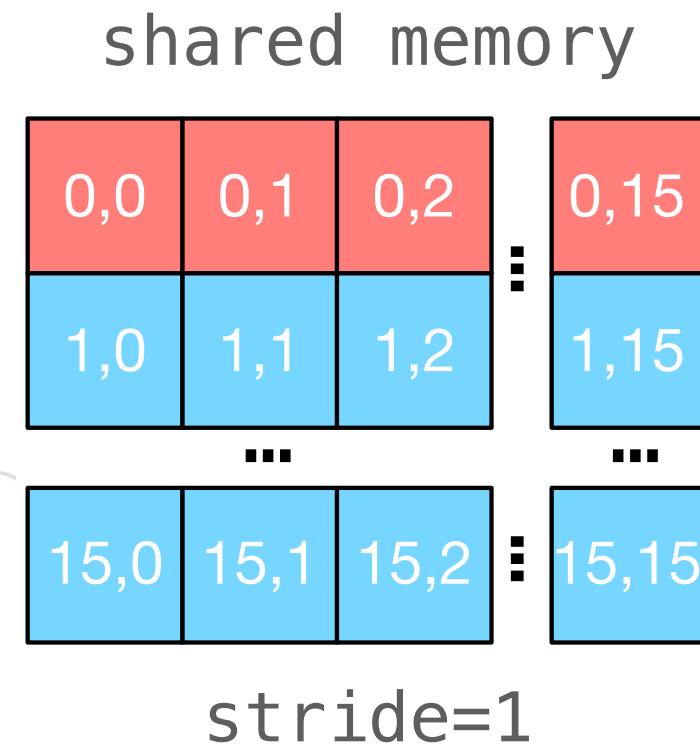
out[(bx+tx)\*n+by+ty]=smem[ty\*BDIM+tx]; ?



## ● 使用共享内存的CUDA矩阵转置

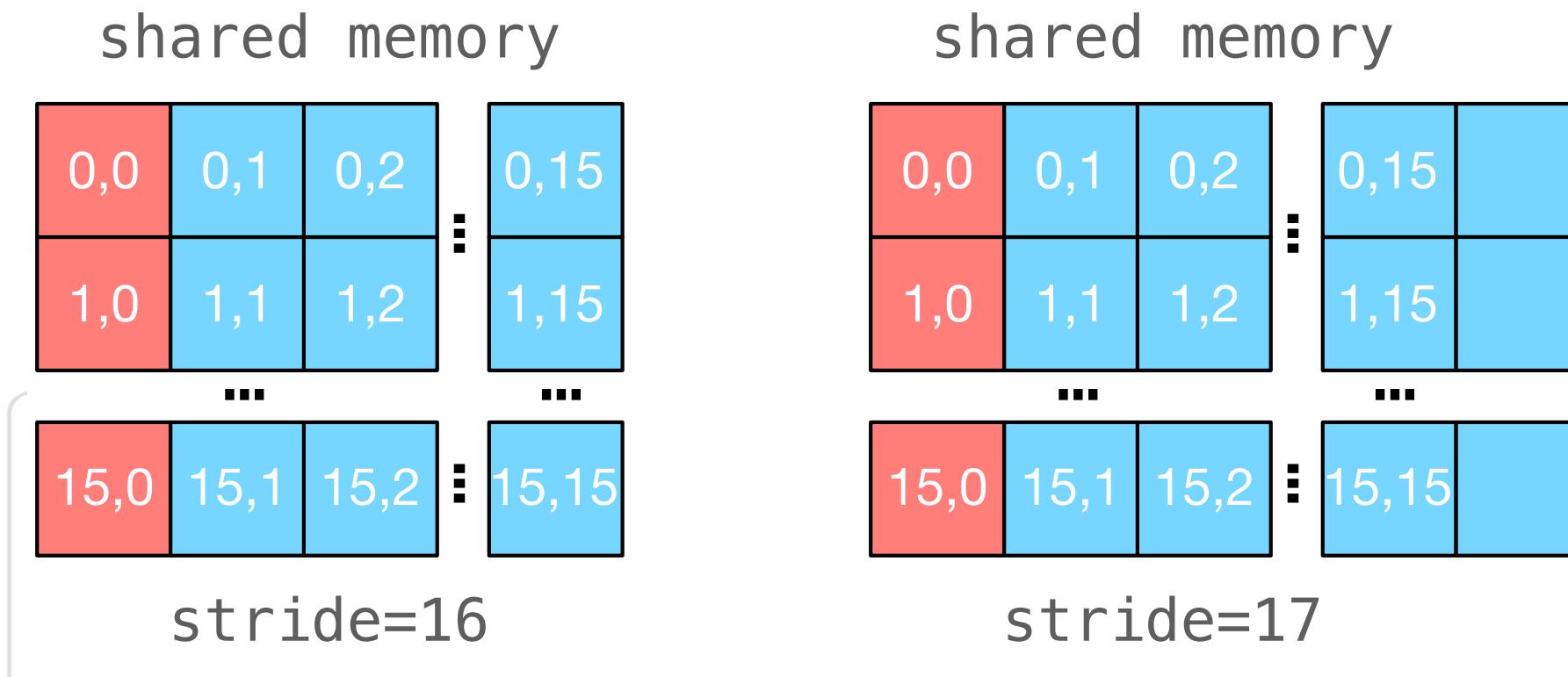
### – 共享内存访问：存储体冲突

- 从全局内存拷贝至共享内存过程中无冲突
- 从共享内存拷贝至全局内存有冲突！
  - 16-way conflicts：所有线程只对两个存储体进行操作

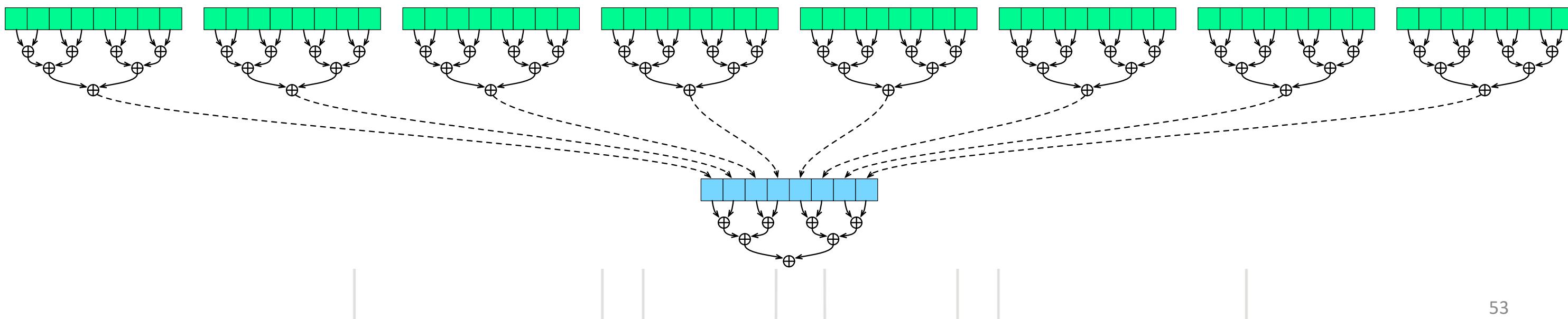


## ● 使用共享内存的CUDA矩阵转置

- 共享内存访问：存储体冲突
  - 解决从共享内存拷贝至全局内存的存储体冲突
    - 给共享内存分配一列空白数据
    - Stride=17：无存储体冲突！



- M. Harris, Optimizing Parallel Reduction in CUDA
  - <https://developer.download.nvidia.cn/assets/cuda/files/reduction.pdf>
  - 对4M个元素进行归约
  - 使用7个版本代码
  - 展示每一步所带来的性能提升
- 回顾：并行归约



## ● GPU峰值性能

- 首先，我们需要有一个正确的目标
  - GFLOP/s: compute-bound 应用
    - 例如：矩阵分解，卷积等
  - 带宽：memory-bound 应用
    - 例如：数据库，图遍历等
  - 归约的计算密度很低
    - 数组中的每个元素只需要参与一次运算
    - 因此主要目标应放在争取峰值带宽上！
  - 实验使用G80 GPU
    - 带宽峰值为86.4 GB/s

## ● 版本1：交错地址读取

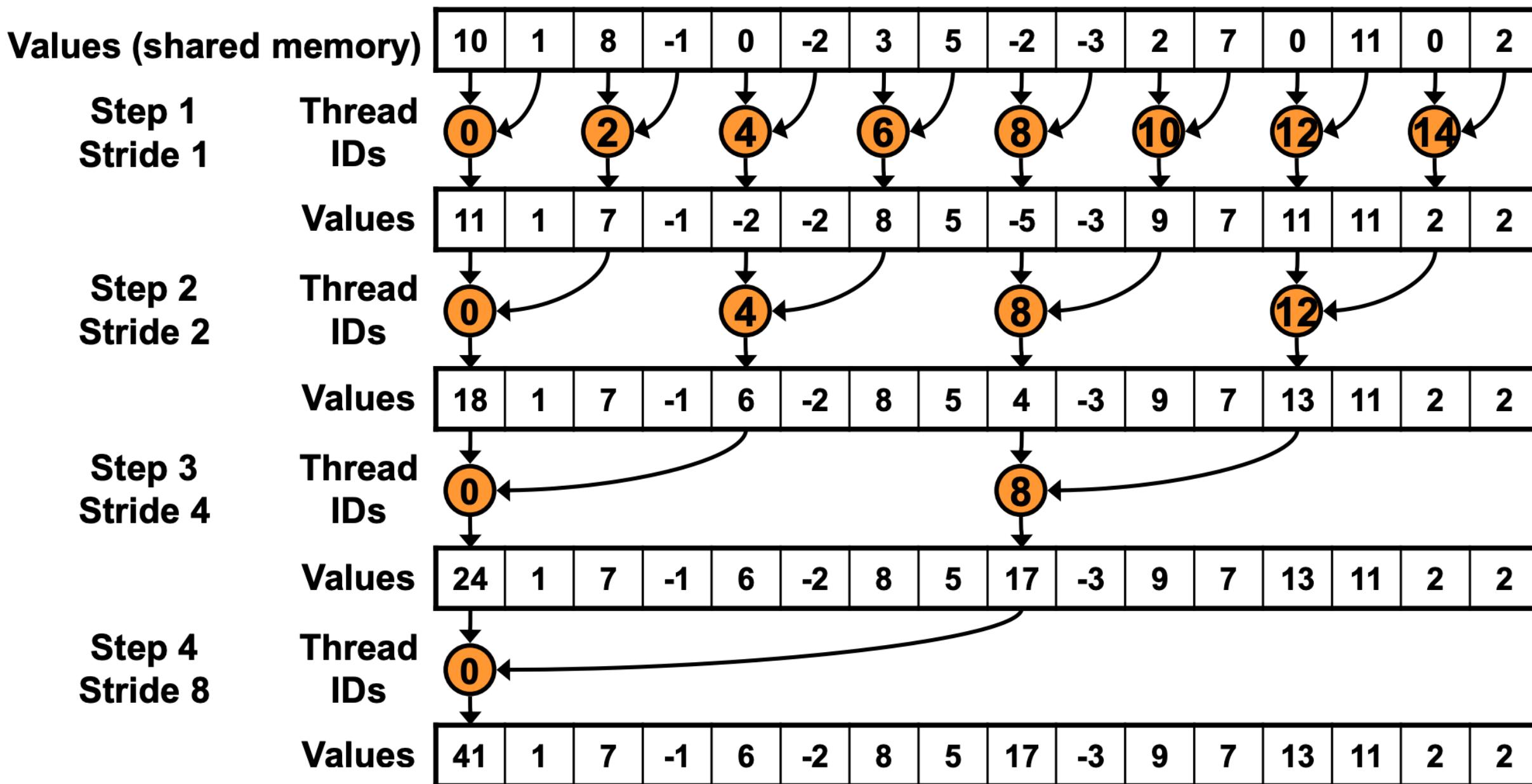
```
__global__ void reduce0(int *g_idata, int *g_odata) {
    extern __shared__ int sdata[];

    // each thread loads one element from global to shared mem
    unsigned int tid = threadIdx.x;
    unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
    sdata[tid] = g_idata[i];
    __syncthreads();

    // do reduction in shared mem
    for(unsigned int s=1; s < blockDim.x; s *= 2) {
        if (tid % (2*s) == 0) {
            sdata[tid] += sdata[tid + s];
        }
        __syncthreads();
    }

    // write result for this block to global mem
    if (tid == 0) g_odata[blockIdx.x] = sdata[0];
}
```

## ● 版本1：交错地址读取



## ● 版本1：交错地址读取

```
__global__ void reduce0(int *g_idata, int *g_odata) {
    extern __shared__ int sdata[];

    // each thread loads one element from global to shared mem
    unsigned int tid = threadIdx.x;
    unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
    sdata[tid] = g_idata[i];
    __syncthreads();

    // do reduction in shared mem
    for(unsigned int s=1; s < blockDim.x; s *= 2) {
        if (tid % (2*s) == 0) {
            sdata[tid] += sdata[tid + s];
        }
        __syncthreads();
    }

    // write result for this block to global mem
    if (tid == 0) g_odata[blockIdx.x] = sdata[0];
}
```

存在问题：  
高度分化的控制流  
求模运算 (%) 缓慢

	Time ( $2^{22}$ ints)	Bandwidth
<b>Kernel 1:</b> interleaved addressing with divergent branching	<b>8.054 ms</b>	<b>2.083 GB/s</b>

Note: Block Size = 128 threads for all tests



- 版本2：交错地址读取
  - 去除内层判断中的分支分化

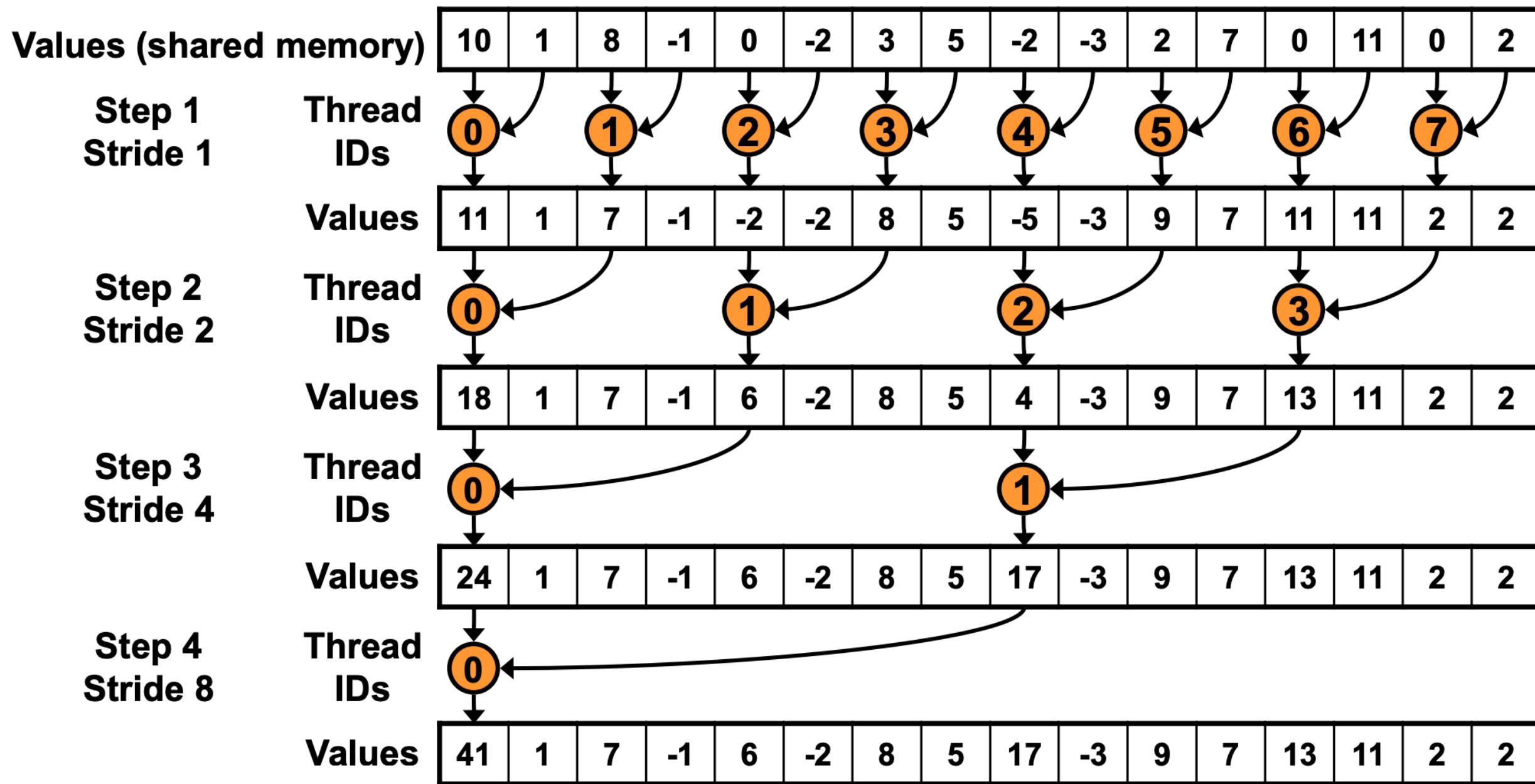
```
for(unsigned int s=1; s < blockDim.x; s *= 2) {  
    if (tid % (2*s) == 0) {  
        sdata[tid] += sdata[tid + s];  
    }  
    __syncthreads();  
}
```

- 增加访问内存时的步长

```
for (unsigned int s=1; s < blockDim.x; s *= 2) {  
    int index = 2 * s * tid;  
  
    if (index < blockDim.x) {  
        sdata[index] += sdata[index + s];  
    }  
    __syncthreads();  
}
```



## ● 版本2：交错地址读取



# 优化举例：并行归约

	Time ( $2^{22}$ ints)	Bandwidth	Step Speedup	Cumulative Speedup
<b>Kernel 1:</b> interleaved addressing with divergent branching	<b>8.054 ms</b>	<b>2.083 GB/s</b>		
<b>Kernel 2:</b> interleaved addressing with bank conflicts	<b>3.456 ms</b>	<b>4.854 GB/s</b>	<b>2.33x</b>	<b>2.33x</b>



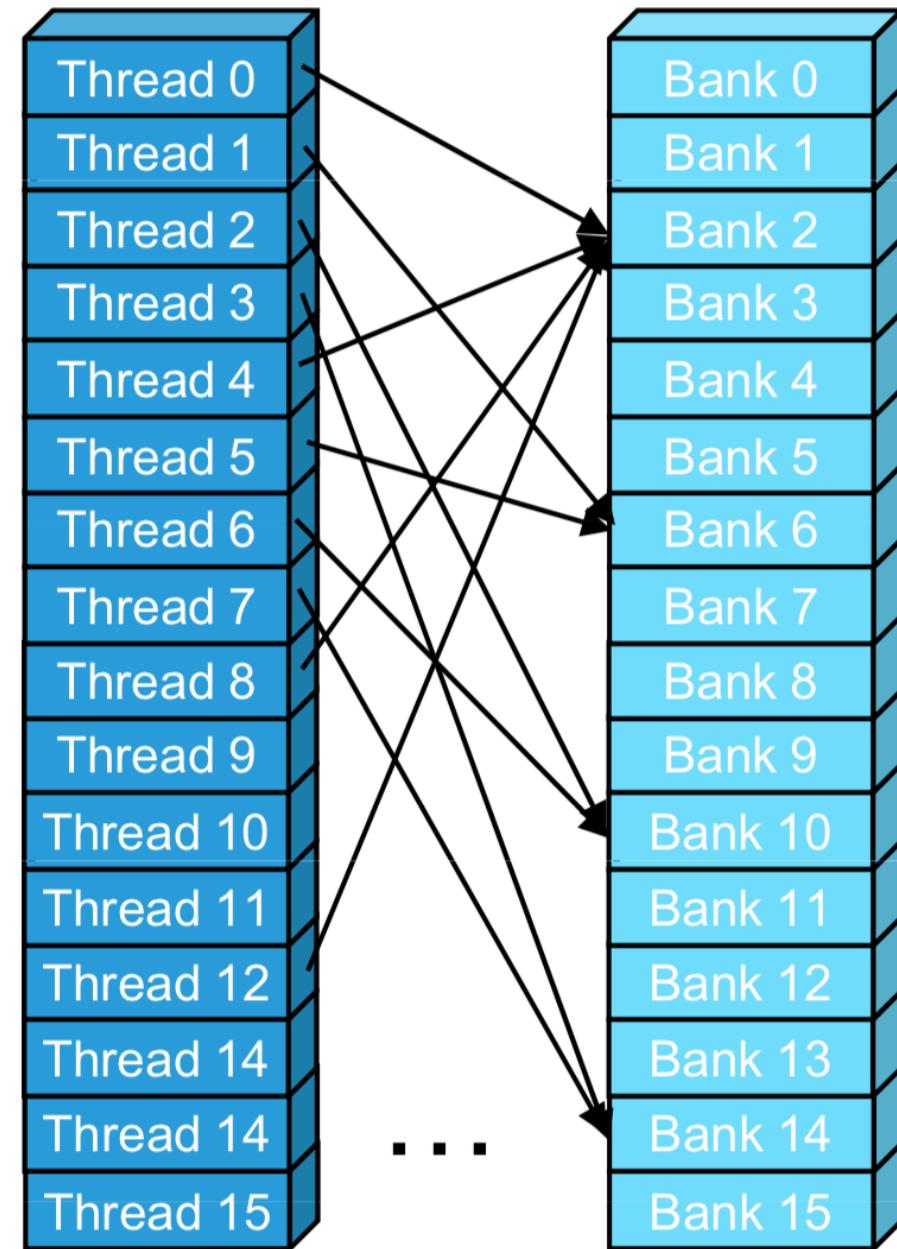
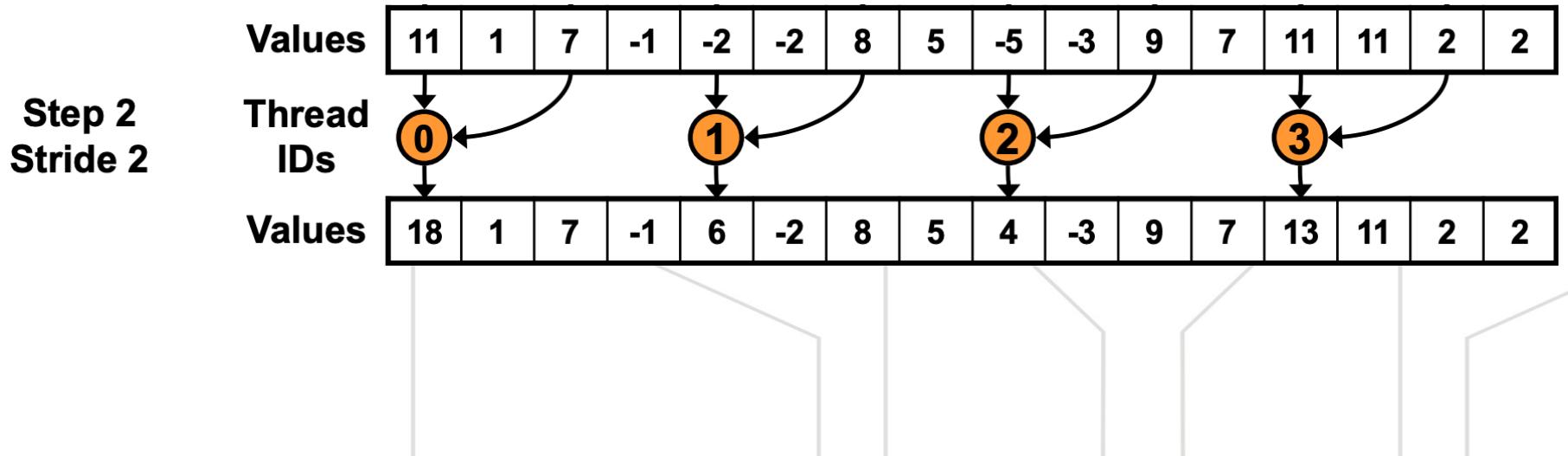
- 版本2：交错地址读取

- 存在问题：存储体冲突！

```

for (unsigned int s=1; s < blockDim.x; s *= 2) {
    int index = 2 * s * tid;

    if (index < blockDim.x) {
        sdata[index] += sdata[index + s];
    }
    __syncthreads();
}
  
```



- 版本3：连续地址读取

- 消除存储体冲突

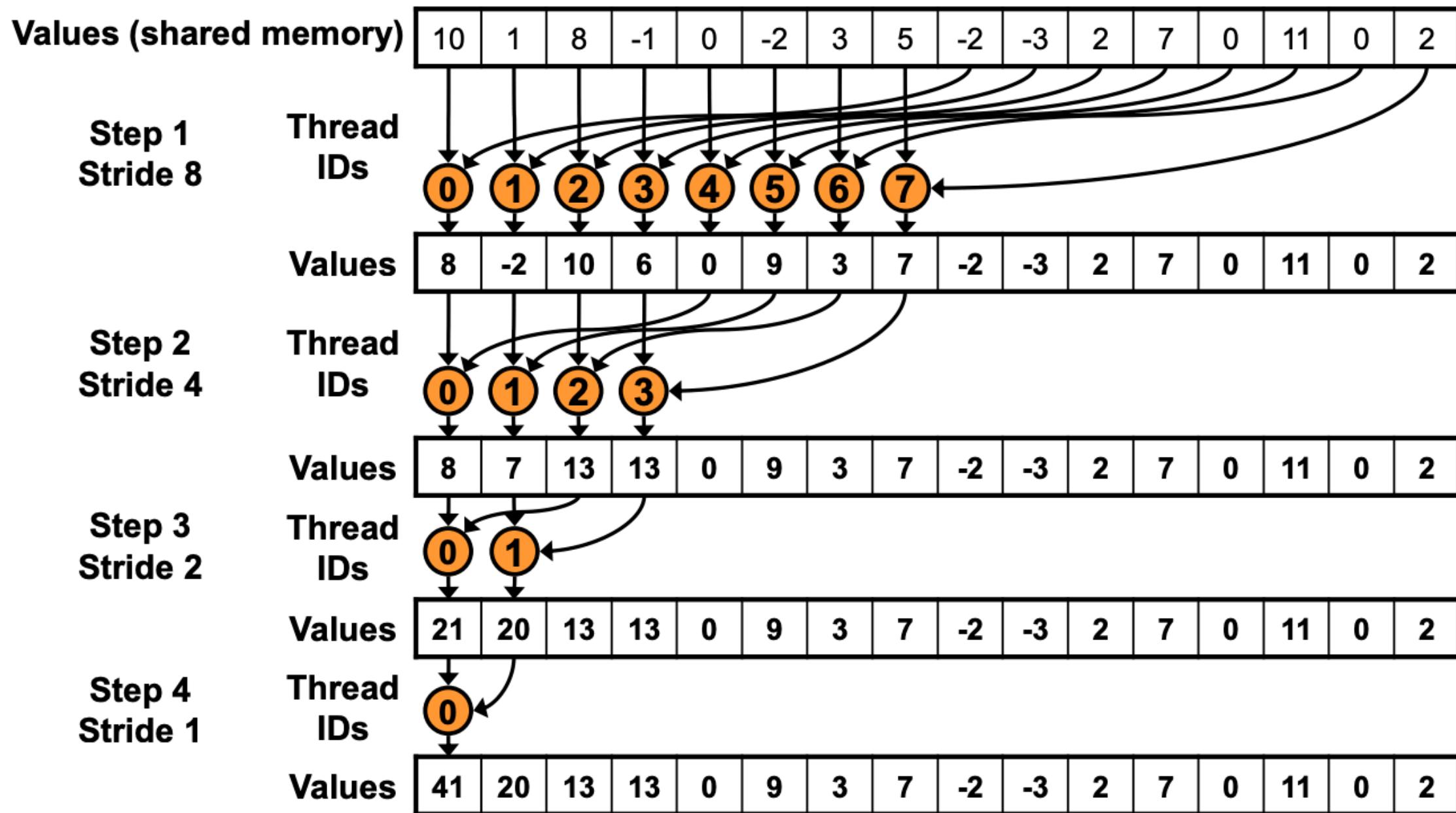
```
for (unsigned int s=1; s < blockDim.x; s *= 2) {  
    int index = 2 * s * tid;  
  
    if (index < blockDim.x) {  
        sdata[index] += sdata[index + s];  
    }  
    __syncthreads();  
}
```

- 基于线程编号读取数据

```
for (unsigned int s=blockDim.x/2; s>0; s>>=1) {  
    if (tid < s) {  
        sdata[tid] += sdata[tid + s];  
    }  
    __syncthreads();  
}
```



## ● 版本3：连续地址读取

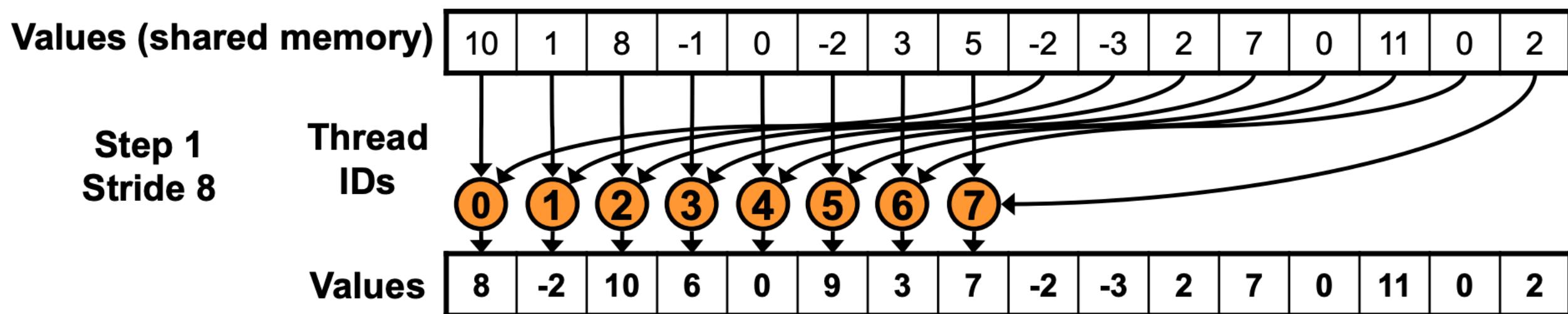


	Time ( $2^{22}$ ints)	Bandwidth	Step Speedup	Cumulative Speedup
<b>Kernel 1:</b> interleaved addressing with divergent branching	<b>8.054 ms</b>	<b>2.083 GB/s</b>		
<b>Kernel 2:</b> interleaved addressing with bank conflicts	<b>3.456 ms</b>	<b>4.854 GB/s</b>	<b>2.33x</b>	<b>2.33x</b>
<b>Kernel 3:</b> sequential addressing	<b>1.722 ms</b>	<b>9.741 GB/s</b>	<b>2.01x</b>	<b>4.68x</b>



- 版本3：连续地址读取
  - 存在问题：空闲线程多！
    - 第一次循环时有一半线程空闲

```
for (unsigned int s=blockDim.x/2; s>0; s>>=1) {  
    if (tid < s) {  
        sdata[tid] += sdata[tid + s];  
    }  
    __syncthreads();  
}
```



- 版本4：每个线程加载两个数据执行一次加法

- 此前每个线程载入一个数据

```
// each thread loads one element from global to shared mem
unsigned int tid = threadIdx.x;
unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
sdata[tid] = g_idata[i];
__syncthreads();
```

- 每个线程载入两个数据同时执行一次加法

```
// perform first level of reduction,
// reading from global memory, writing to shared memory
unsigned int tid = threadIdx.x;
unsigned int i = blockIdx.x*(blockDim.x*2) + threadIdx.x;
sdata[tid] = g_idata[i] + g_idata[i+blockDim.x];
__syncthreads();
```

	Time (2 <sup>22</sup> ints)	Bandwidth	Step Speedup	Cumulative Speedup
<b>Kernel 1:</b> interleaved addressing with divergent branching	<b>8.054 ms</b>	<b>2.083 GB/s</b>		
<b>Kernel 2:</b> interleaved addressing with bank conflicts	<b>3.456 ms</b>	<b>4.854 GB/s</b>	<b>2.33x</b>	<b>2.33x</b>
<b>Kernel 3:</b> sequential addressing	<b>1.722 ms</b>	<b>9.741 GB/s</b>	<b>2.01x</b>	<b>4.68x</b>
<b>Kernel 4:</b> first add during global load	<b>0.965 ms</b>	<b>17.377 GB/s</b>	<b>1.78x</b>	<b>8.34x</b>



## ● 性能瓶颈

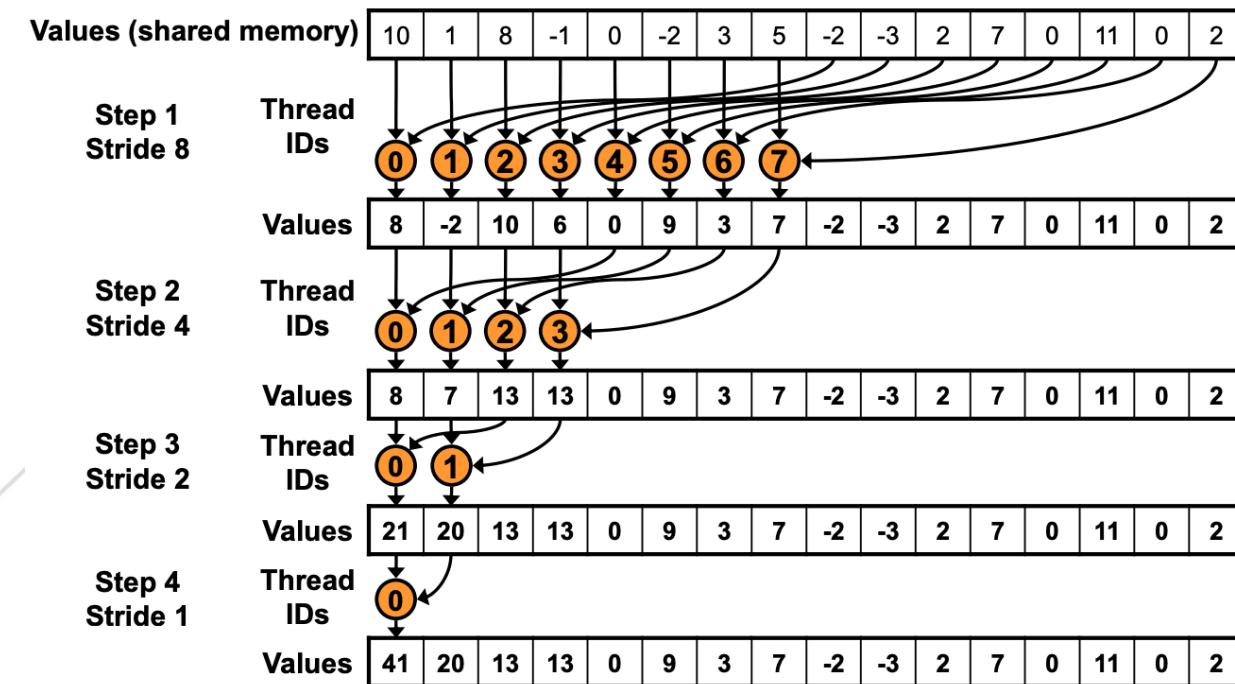
- 17GB/s的带宽仍然与理论峰值86.4GB/s相去甚远
  - 这与并行归约（求和）极低的运算密度有很大的差距
- 当前仍然存在相当大的**辅助指令开销**
  - 非内存读写或进行运算的指令
  - 地址运算、循环开销
- 解决方案：展开循环



## ● 版本5：展开最后一个线程束

- 随着归约的进行，活跃线程数量逐渐减少
  - 当线程数少于32时，只剩第一个线程束仍在工作
    - 其余线程束已完成计算
  - 线程束内指令自动同步
    - 不需要 `if (tid < s)`
    - 不需要 `_syncthreads();`
  - 展开最后一个线程束
    - 对应6次循环（从s=32开始）

```
for (unsigned int s=blockDim.x/2; s>0; s>>=1) {
    if (tid < s) {
        sdata[tid] += sdata[tid + s];
    }
    __syncthreads();
}
```



## ● 版本5：展开最后一个线程束

```
__device__ void warpReduce(volatile int* sdata, int tid) {  
    sdata[tid] += sdata[tid + 32];  
    sdata[tid] += sdata[tid + 16];  
    sdata[tid] += sdata[tid + 8];  
    sdata[tid] += sdata[tid + 4];  
    sdata[tid] += sdata[tid + 2];  
    sdata[tid] += sdata[tid + 1];  
}
```

可否去掉 volatile?

```
for (unsigned int s=blockDim.x/2; s>32; s>>=1) {  
    if (tid < s) {  
        sdata[tid] += sdata[tid + s];  
    }  
    __syncthreads();  
}  
  
if (tid<32)  
    warpReduce(sdata, tid);
```

这种写法减少了所有线程束中的无用功，而不仅仅是最后一个线程束！

	Time ( $2^{22}$ ints)	Bandwidth	Step Speedup	Cumulative Speedup
<b>Kernel 1:</b> interleaved addressing with divergent branching	<b>8.054 ms</b>	<b>2.083 GB/s</b>		
<b>Kernel 2:</b> interleaved addressing with bank conflicts	<b>3.456 ms</b>	<b>4.854 GB/s</b>	<b>2.33x</b>	<b>2.33x</b>
<b>Kernel 3:</b> sequential addressing	<b>1.722 ms</b>	<b>9.741 GB/s</b>	<b>2.01x</b>	<b>4.68x</b>
<b>Kernel 4:</b> first add during global load	<b>0.965 ms</b>	<b>17.377 GB/s</b>	<b>1.78x</b>	<b>8.34x</b>
<b>Kernel 5:</b> unroll last warp	<b>0.536 ms</b>	<b>31.289 GB/s</b>	<b>1.8x</b>	<b>15.01x</b>



## ● 版本6：完全展开循环

- 如果编译时就已知迭代次数，可以完全展开所有循环
  - 线程块至多包括1024个线程（Harris原文中为512个）
  - 线程块内线程数量通常为2的指数次幂
  - 已知迭代次数的情况下，很容易手动完全展开所有循环
- 但我们更需要的是通用代码，如何完全展开一个编译时未知迭代次数的循环？
  - 模板（template）：使核函数以不同配置方式被启动
  - CUDA的\_\_device\_\_及\_\_global\_\_函数都支持C++模板



## ● 版本6：完全展开循环

```
template <unsigned int blockSize>
__device__ void warpReduce(volatile int* sdata, int tid) {
    if (blockSize >= 64) sdata[tid] += sdata[tid + 32];
    if (blockSize >= 32) sdata[tid] += sdata[tid + 16];
    if (blockSize >= 16) sdata[tid] += sdata[tid + 8];
    if (blockSize >= 8) sdata[tid] += sdata[tid + 4];
    if (blockSize >= 4) sdata[tid] += sdata[tid + 2];
    if (blockSize >= 2) sdata[tid] += sdata[tid + 1];
}

if (blockSize >= 512) {
    if (tid < 256) { sdata[tid] += sdata[tid + 256]; } __syncthreads();
}
if (blockSize >= 256) {
    if (tid < 128) { sdata[tid] += sdata[tid + 128]; } __syncthreads();
}
if (blockSize >= 128) {
    if (tid < 64) { sdata[tid] += sdata[tid + 64]; } __syncthreads();
}

if (tid < 32) warpReduce<blockSize>(sdata, tid);
```

所有红色的语句均为编译时判断，而非运行时。

- 版本6：完全展开循环

- 函数调用

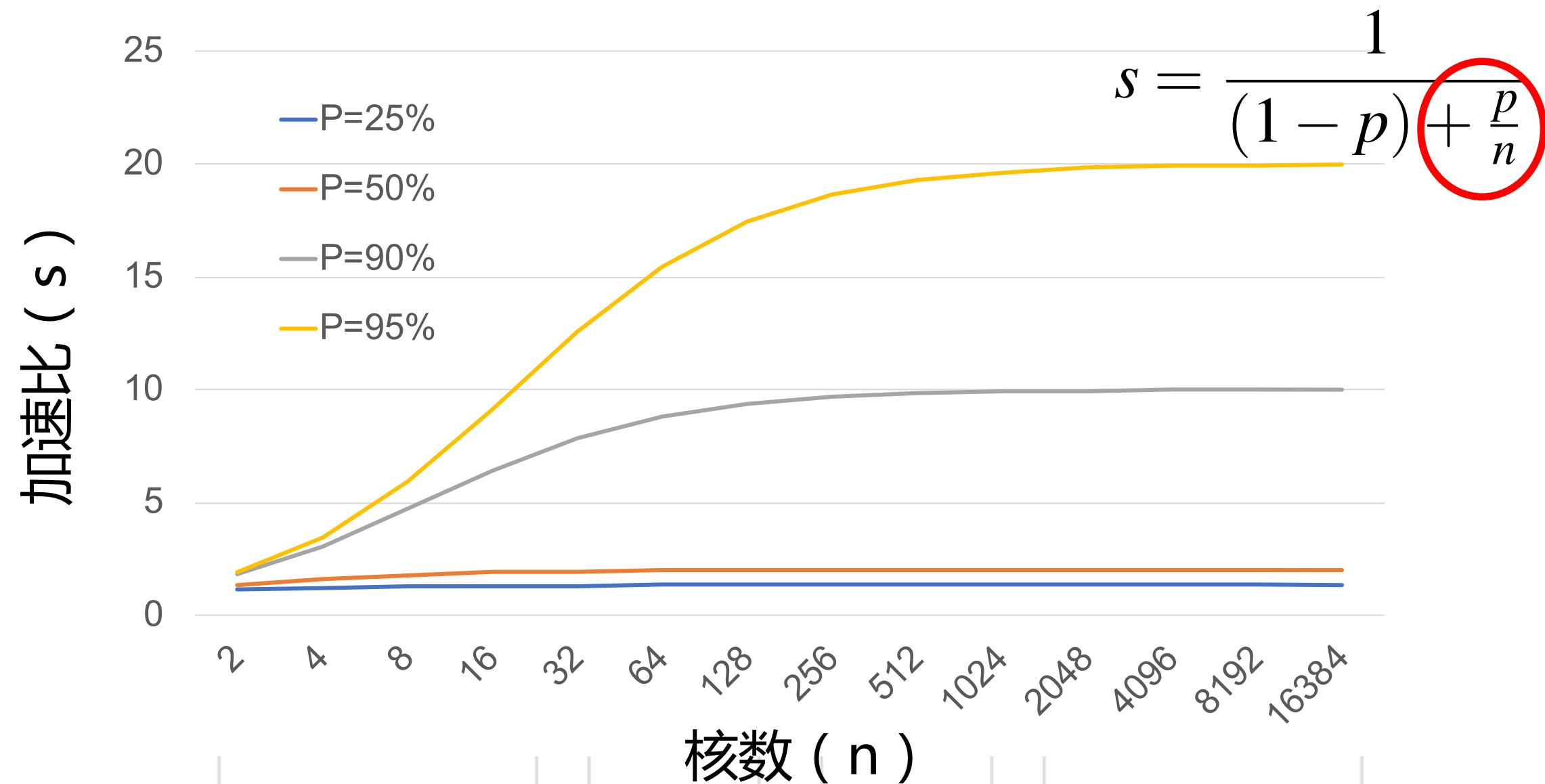
```
switch (threads){  
    case 512: reduce5<512><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;  
    case 256: reduce5<256><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;  
    case 128: reduce5<128><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;  
    case 64: reduce5< 64><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;  
    case 32: reduce5< 32><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;  
    case 16: reduce5< 16><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;  
    case 8: reduce5< 8><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;  
    case 4: reduce5< 4><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;  
    case 2: reduce5< 2><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;  
    case 1: reduce5< 1><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;  
}
```



	Time ( $2^{22}$ ints)	Bandwidth	Step Speedup	Cumulative Speedup
<b>Kernel 1:</b> interleaved addressing with divergent branching	<b>8.054 ms</b>	<b>2.083 GB/s</b>		
<b>Kernel 2:</b> interleaved addressing with bank conflicts	<b>3.456 ms</b>	<b>4.854 GB/s</b>	<b>2.33x</b>	<b>2.33x</b>
<b>Kernel 3:</b> sequential addressing	<b>1.722 ms</b>	<b>9.741 GB/s</b>	<b>2.01x</b>	<b>4.68x</b>
<b>Kernel 4:</b> first add during global load	<b>0.965 ms</b>	<b>17.377 GB/s</b>	<b>1.78x</b>	<b>8.34x</b>
<b>Kernel 5:</b> unroll last warp	<b>0.536 ms</b>	<b>31.289 GB/s</b>	<b>1.8x</b>	<b>15.01x</b>
<b>Kernel 6:</b> completely unrolled	<b>0.381 ms</b>	<b>43.996 GB/s</b>	<b>1.41x</b>	<b>21.16x</b>

## ● 回顾：Amdahl's Law

– 程序效率受限于其可并行的比例及可供使用的核数



## ● Brent's theorem

- 并行效率还受限于每步能并行的工作量
  - 如并行归约，总运算量为 $W(n)$ ，理论运行时间为 $O(\log n)$ 
    - 但在最后几步时能同时工作的线程数则远无法达到 $p$ （处理器数目）
- $W(n)$ 为并行算法A在运行时间 $T(n)$ 内执行的运算总量，则使用 $p$ 台处理器（线程）时，实际完成时间为 $t(n)=O(W(n)/p+T(n))$ 
  - $W_i$ 为在第*i*时刻完成的工作量

$$t(n) = \sum_{i=1}^{T(n)} \left\lceil \frac{W_i}{p} \right\rceil \leq \sum_{i=1}^{T(n)} \left( \left\lfloor \frac{W_i}{p} \right\rfloor + 1 \right) \leq \sum_{i=1}^{T(n)} \left( \frac{W_i}{p} + 1 \right) \leq \frac{W(n)}{p} + T(n)$$



## ● Brent's theorem

– 指导意义：增加每个线程的工作量

- Brent's theorem建议每个线程处理  $O(\log n)$  个数据
- 对并行归约而言：每个线程先对多个数据相加后，再写入共享内存
  - 串行加并行
  - Algorithm cascade
- Harris的实践经验
  - 增加每个线程的工作量通常能更好地掩盖延迟
  - 减少最后几层中核函数的调用开销（创建线程）



- 版本7：每个线程载入多个数据并先行求和

- 此前，每个线程载入两个数据并执行一次加法

```
unsigned int tid = threadIdx.x;
unsigned int i = blockIdx.x*(blockDim.x*2) +
threadIdx.x;
sdata[tid] = g_idata[i] +
g_idata[i+blockDim.x];
__syncthreads();
```

- 使用循环载入多个数据进行多次加法

```
unsigned int tid = threadIdx.x;
unsigned int i = blockIdx.x*(blockSize*2) + threadIdx.x;
unsigned int gridSize = blockSize*2*gridDim.x;
sdata[tid] = 0;
while (i < n) {
    sdata[tid] += g_idata[i] + g_idata[i+blockSize];
    i += gridSize;
}
__syncthreads();
```

- 版本7：每个线程载入多个数据并先行求和

- 此前，每个线程载入两个数据并执行一次加法

```
unsigned int tid = threadIdx.x;
unsigned int i = blockIdx.x*(blockDim.x*2) +
threadIdx.x;
sdata[tid] = g_idata[i] +
g_idata[i+blockDim.x];
__syncthreads();
```

- 使用循环载入多个数据进行多次加法

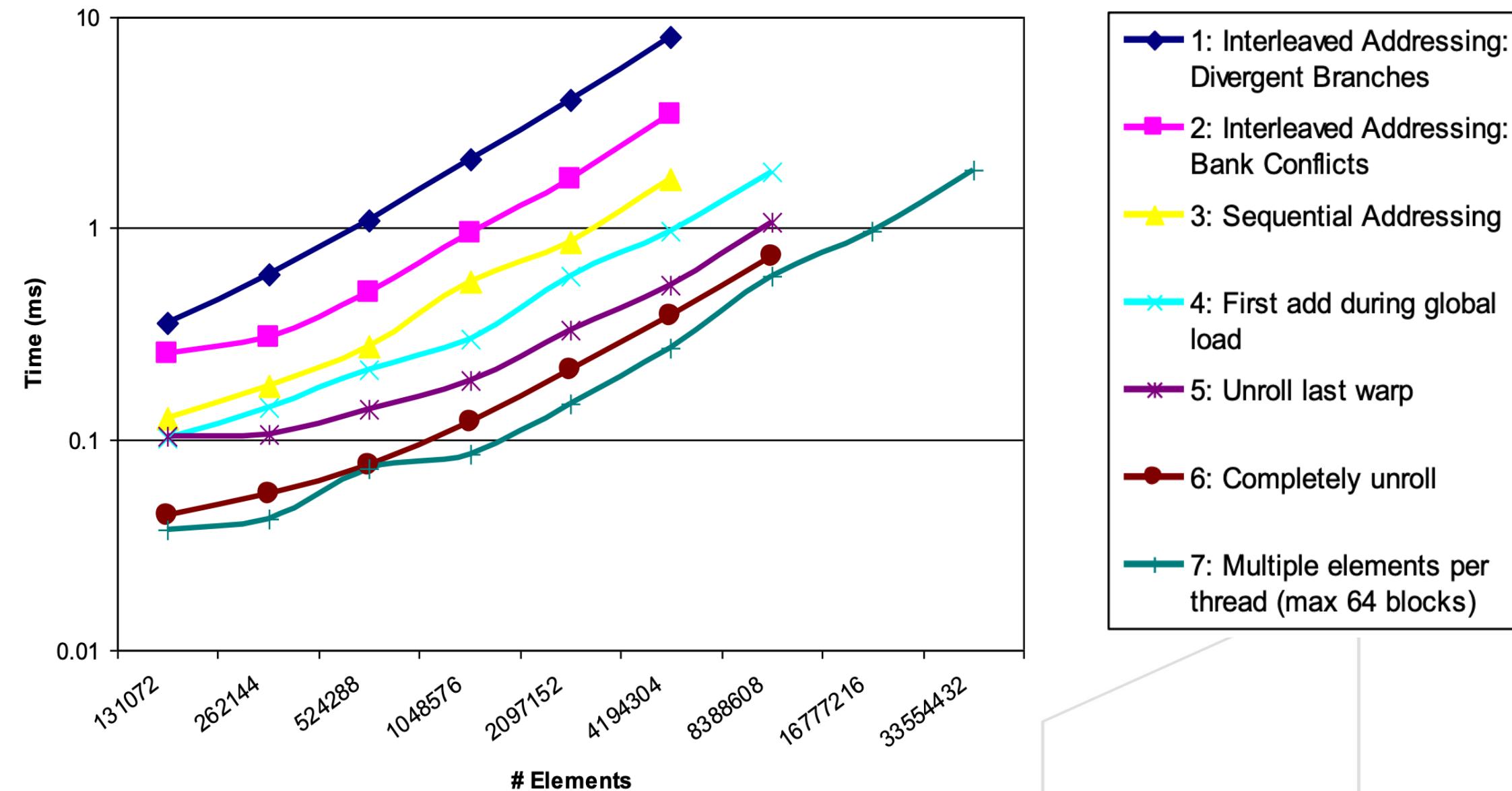
```
unsigned int tid = threadIdx.x;
unsigned int i = blockIdx.x*(blockSize*2) + threadIdx.x;
unsigned int gridSize = blockSize*2*gridDim.x;
sdata[tid] = 0;
while (i < n) {
    sdata[tid] += g_idata[i] + g_idata[i+blockSize];
    i += gridSize;
}
__syncthreads();
```

i += gridSize 意味每个线程处理的内存不连续：  
是否可以每个线程处理一段连续内存？

	Time ( $2^{22}$ ints)	Bandwidth	Step Speedup	Cumulative Speedup
<b>Kernel 1:</b> interleaved addressing with divergent branching	<b>8.054 ms</b>	<b>2.083 GB/s</b>		
<b>Kernel 2:</b> interleaved addressing with bank conflicts	<b>3.456 ms</b>	<b>4.854 GB/s</b>	<b>2.33x</b>	<b>2.33x</b>
<b>Kernel 3:</b> sequential addressing	<b>1.722 ms</b>	<b>9.741 GB/s</b>	<b>2.01x</b>	<b>4.68x</b>
<b>Kernel 4:</b> first add during global load	<b>0.965 ms</b>	<b>17.377 GB/s</b>	<b>1.78x</b>	<b>8.34x</b>
<b>Kernel 5:</b> unroll last warp	<b>0.536 ms</b>	<b>31.289 GB/s</b>	<b>1.8x</b>	<b>15.01x</b>
<b>Kernel 6:</b> completely unrolled	<b>0.381 ms</b>	<b>43.996 GB/s</b>	<b>1.41x</b>	<b>21.16x</b>
<b>Kernel 7:</b> multiple elements per thread	<b>0.268 ms</b>	<b>62.671 GB/s</b>	<b>1.42x</b>	<b>30.04x</b>

Kernel 7 on 32M elements: 73 GB/s!

## Performance Comparison



## ● 影响CUDA程序的关键因素

- 内存对齐与合并访问
- 分支分化
- 存储体冲突
- 掩盖演示

## ● 优化策略

- 设定合适的目标：内存、计算、指令开销
- 了解硬件：选用合适的策略/参数
- 优化占用率
- 选择性地应用串行程序的优化策略

# Questions?

