



中山大學
SUN YAT-SEN UNIVERSITY



多核程序设计与实践

CUDA性能优化

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

- 优化全局内存访问
- 其他优化方法



● 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）

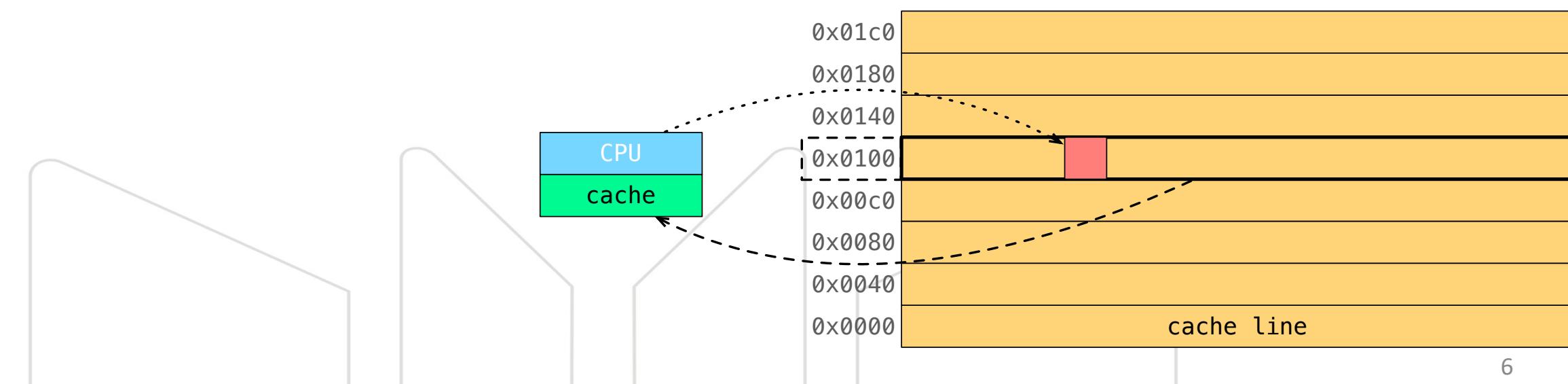
- SIMD：在控制流出现分支分化时，不同分支将串行执行
- 减少分支分化的影响
 - 减少判断语句
 - 尤其是减少基于threadIdx的判断语句
 - » 判断语句不必然导致分支分化
 - 使用条件语句代替条件赋值
 - 平衡分支执行时间
 - 避免出现执行时间过长的分支

● 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];
    }
}
```

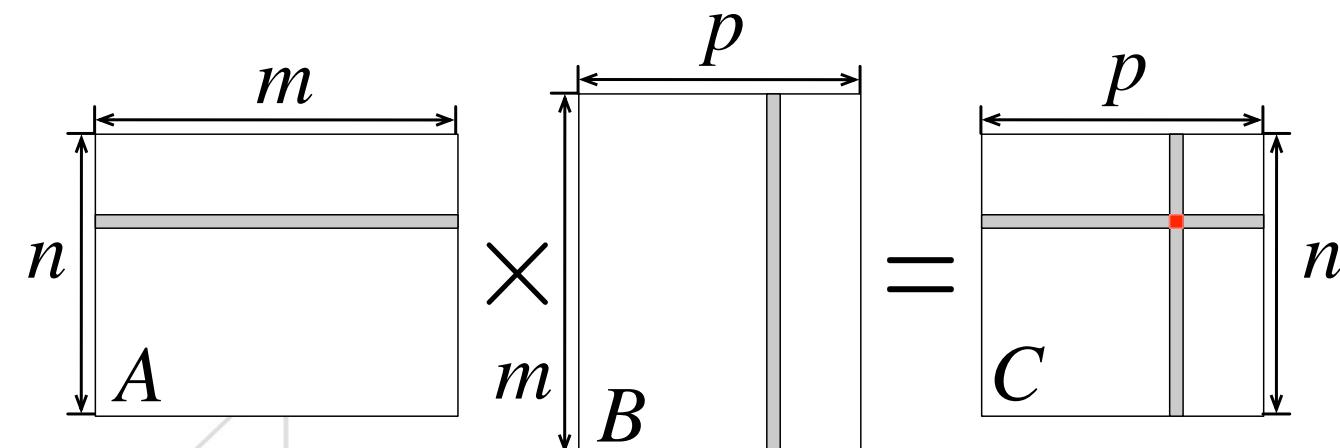
运行时间：35.708秒

● 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

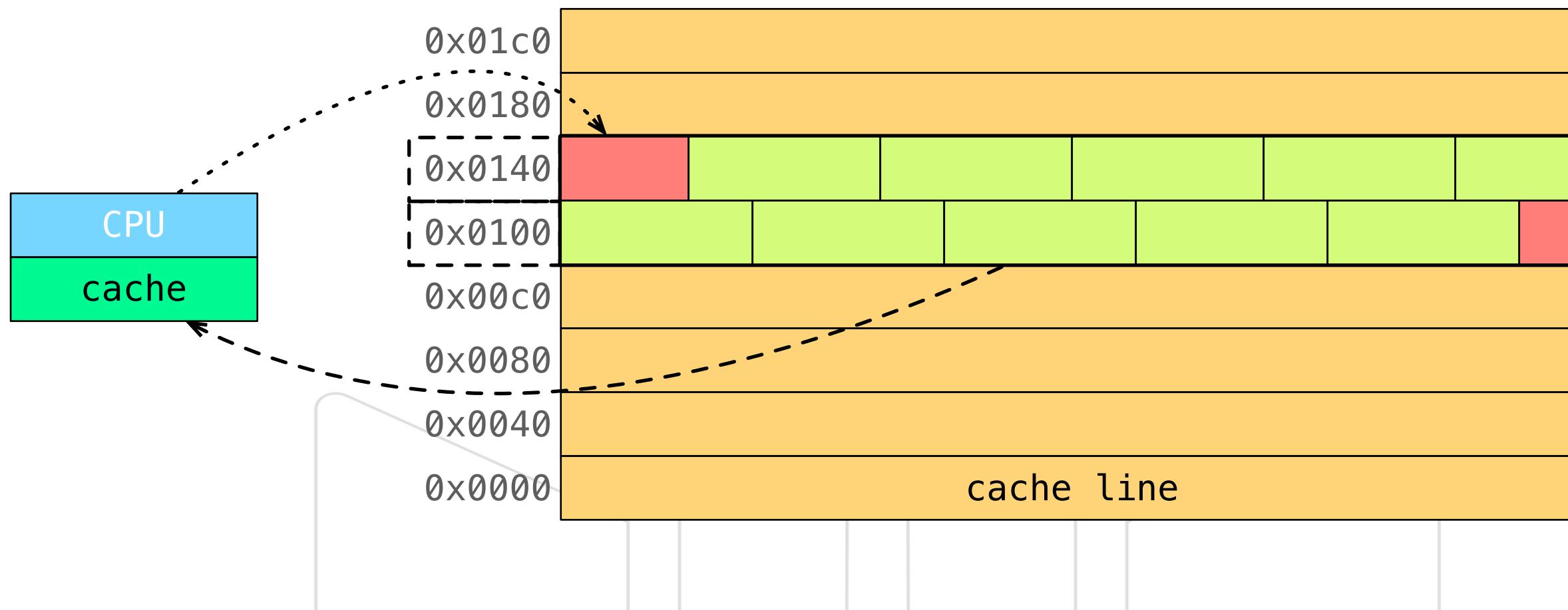
```
#define N 3000
int A[N][N], B[N][N], C[N][N];

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__`



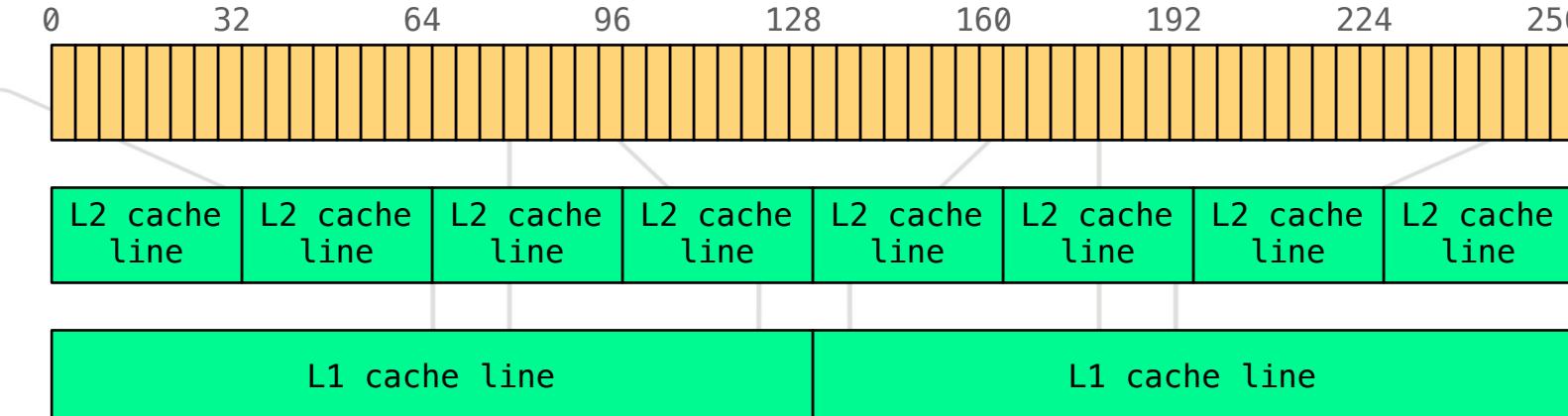
● 对齐与合并访问：全局内存读取

— 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

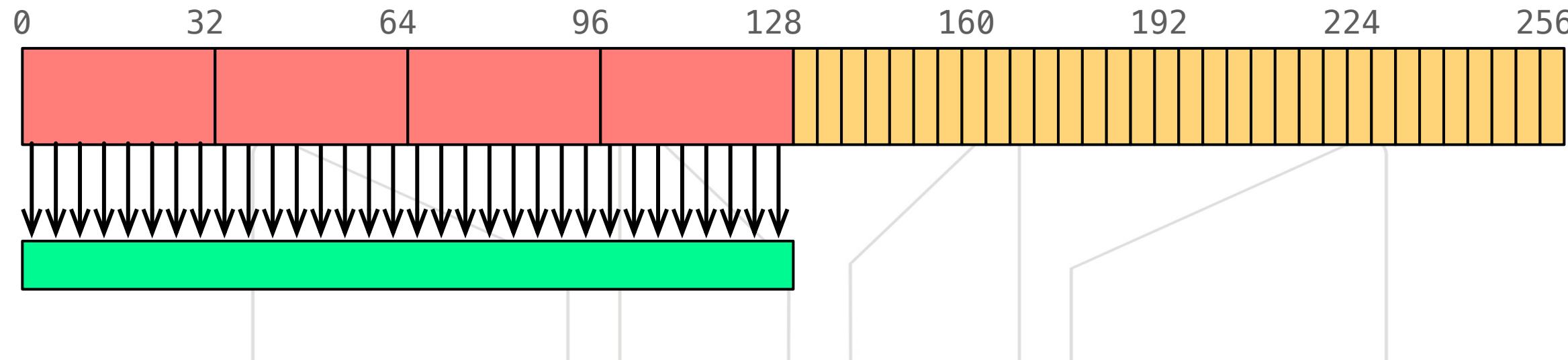
— 二级缓存

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



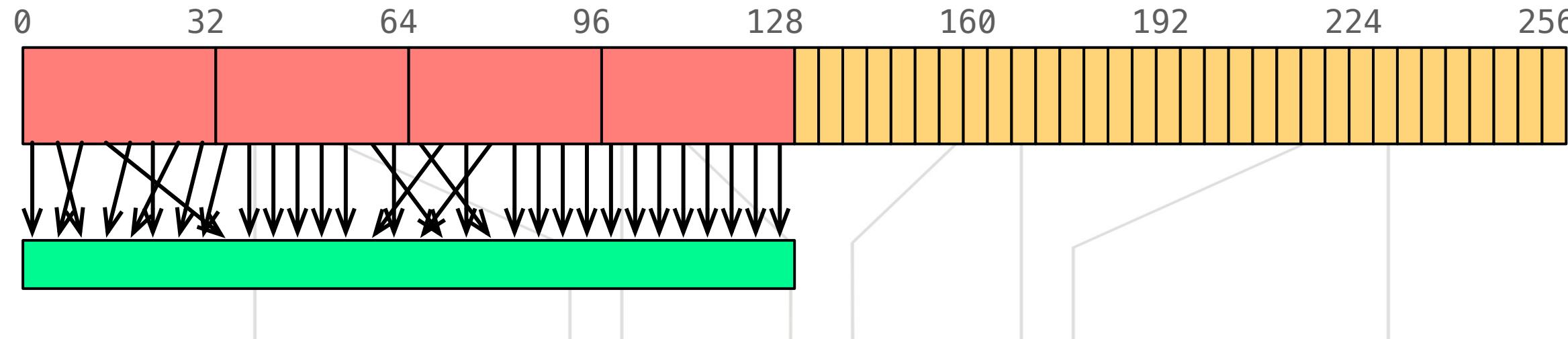
- 通过二级缓存的合并访问

- 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

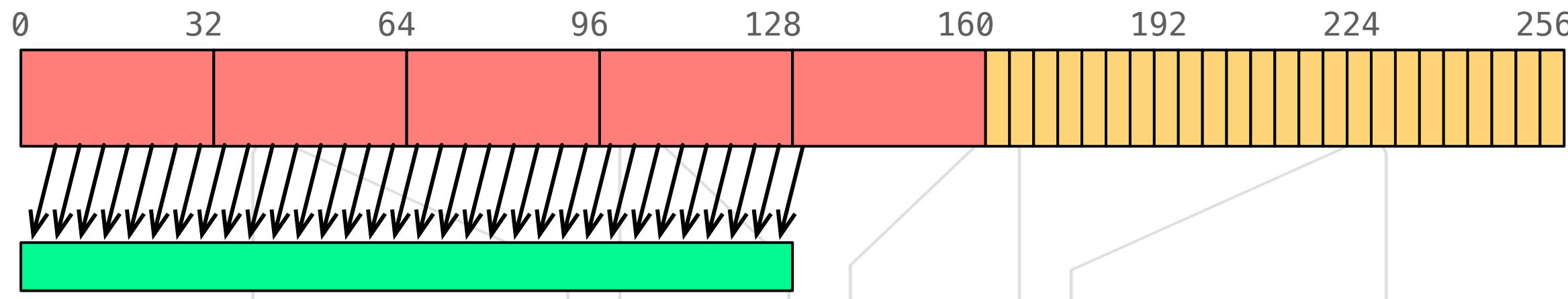
- 线程束对全局内存的访问将被合并为**内存事务**进行
- 对齐访问，但引用地址不连续
 - 假设每个线程请求4字节的数据（整型、单精度浮点型等）
 - 只要线程引用地址均落在同样的128字节段内，并且无重复（跨过32字节段边界）
 - 完成加载操作仍然只需4个32字节的内存事务
 - 总线利用率仍然为100%



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%

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

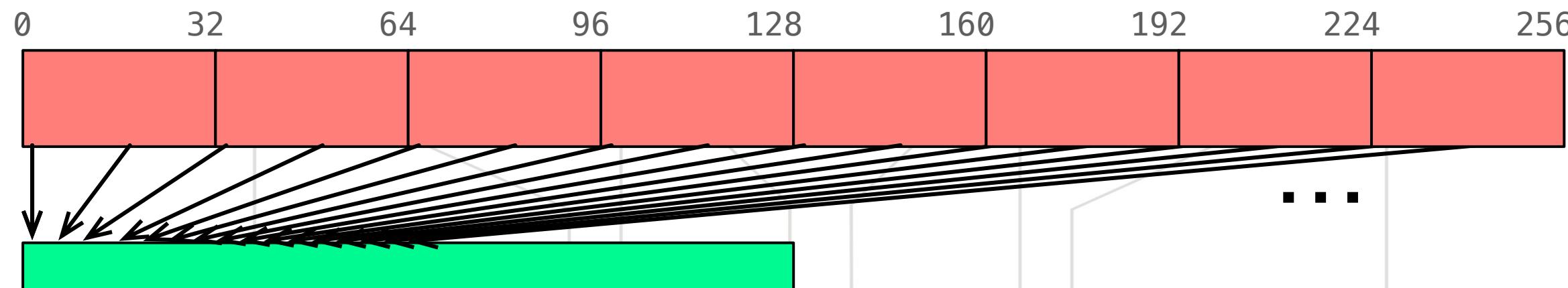


- 通过二级缓存的合并访问

- 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]);  
}
```

- $\text{stride}=4$ 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)

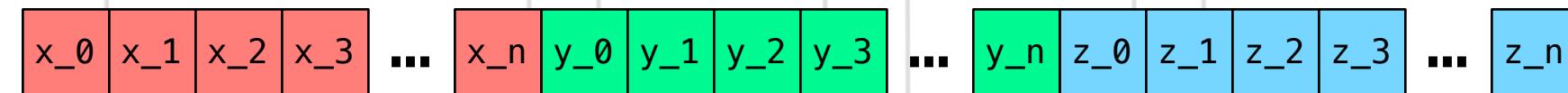
- SoA: 结构体每个成员为一个数组
 - 例如, 同样用于表示三维点集时

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

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

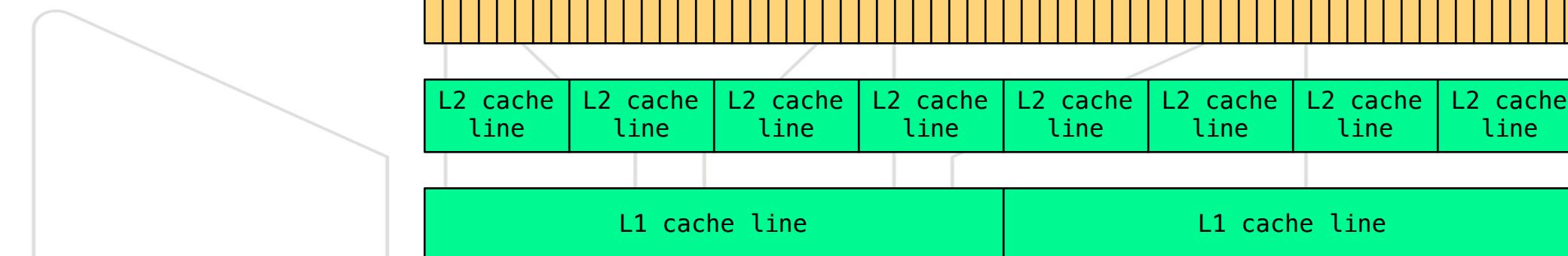
    float x = data.x[tid];
    float y = data.y[tid];
    float z = data.z[tid];

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



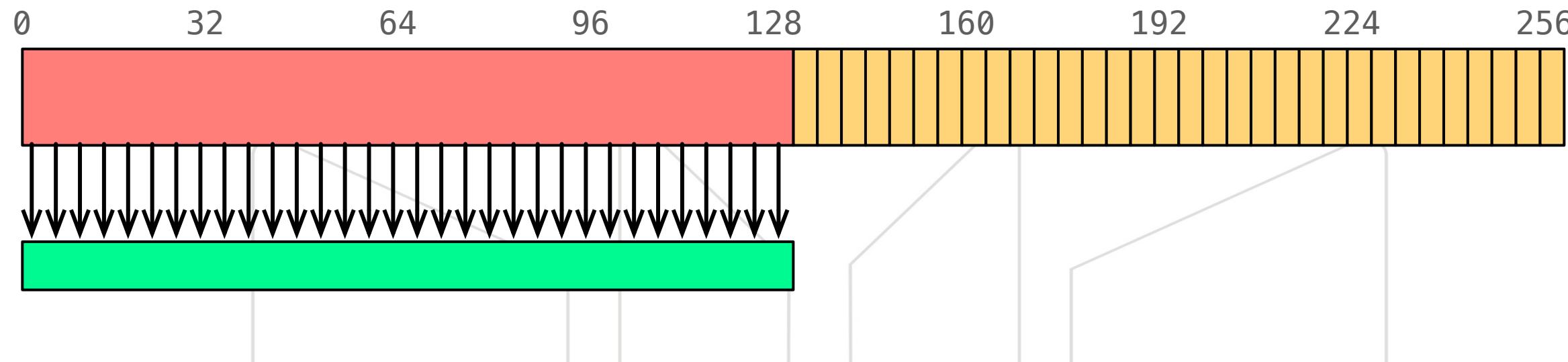
通过一级缓存的合并访问

- 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

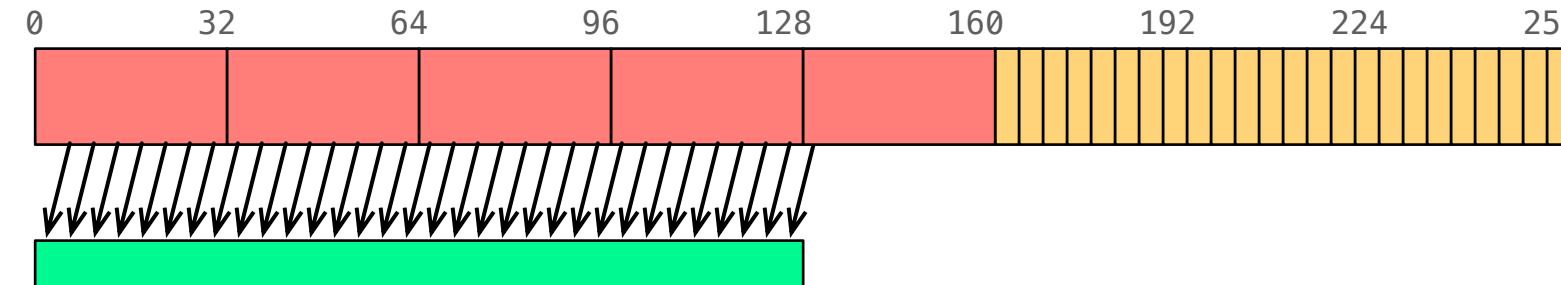
- 与二级缓存的合并访问相似
- 理想情况：对齐与合并内存访问
 - 线程束所有内存请求引用地址连续
 - 假设每个线程请求4字节的数据（整型、单精度浮点型等），完成加载操作只需1个128字节的内存事务
 - 总线利用率100%



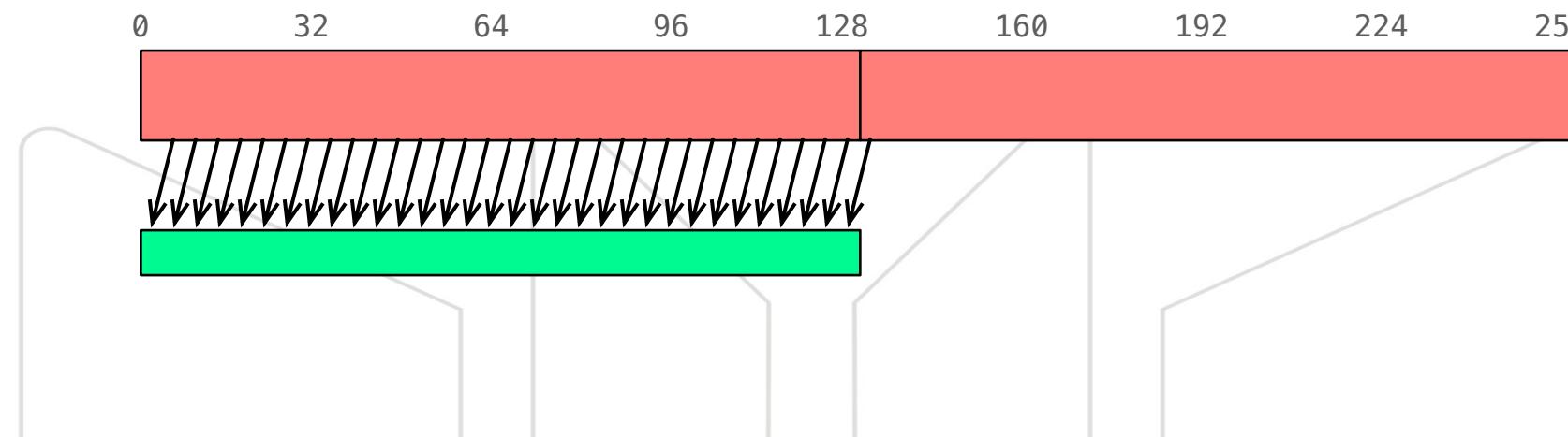
- 通过一级缓存的合并访问

- 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

- 扫描算法（续）
- 优化全局内存访问
- 其他优化方法



- 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

- 缺点: 为增加占用率减小共享内存可能导致缓存效率降低

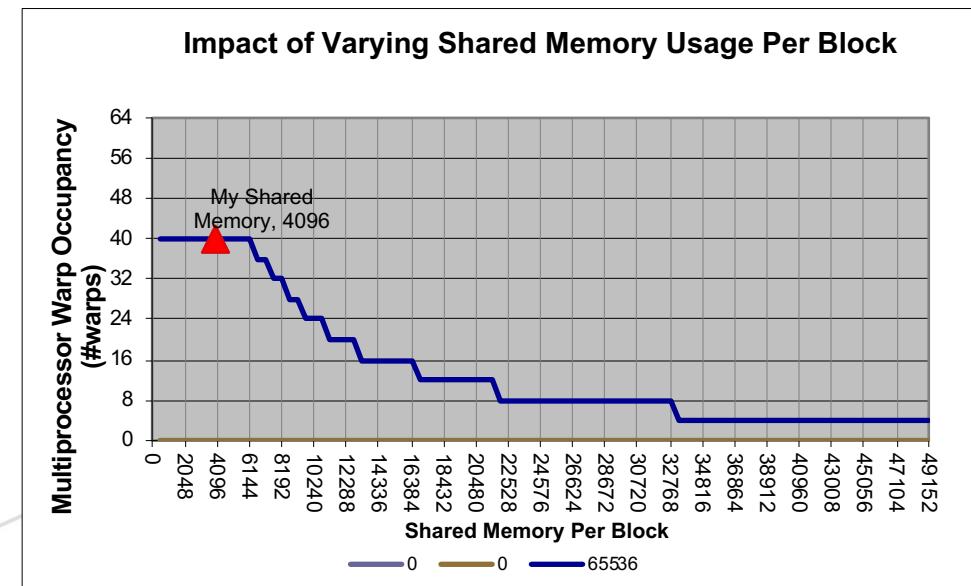
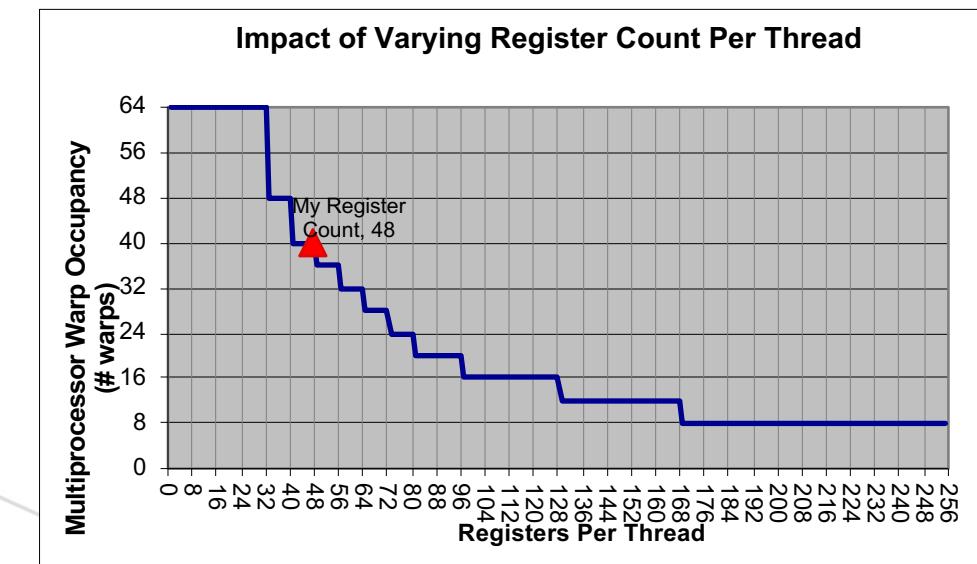
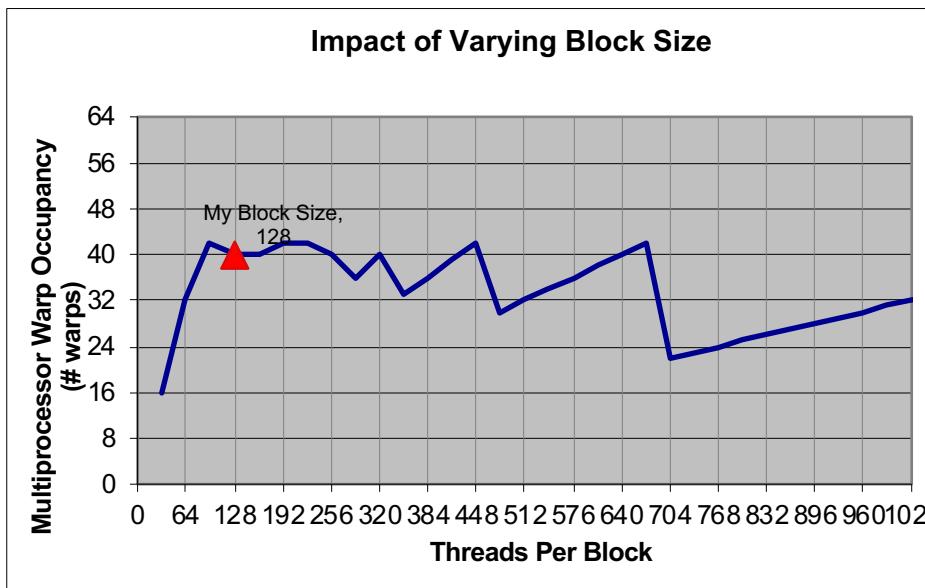
- 应在占用率及分配资源间寻找平衡

● 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

- 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：

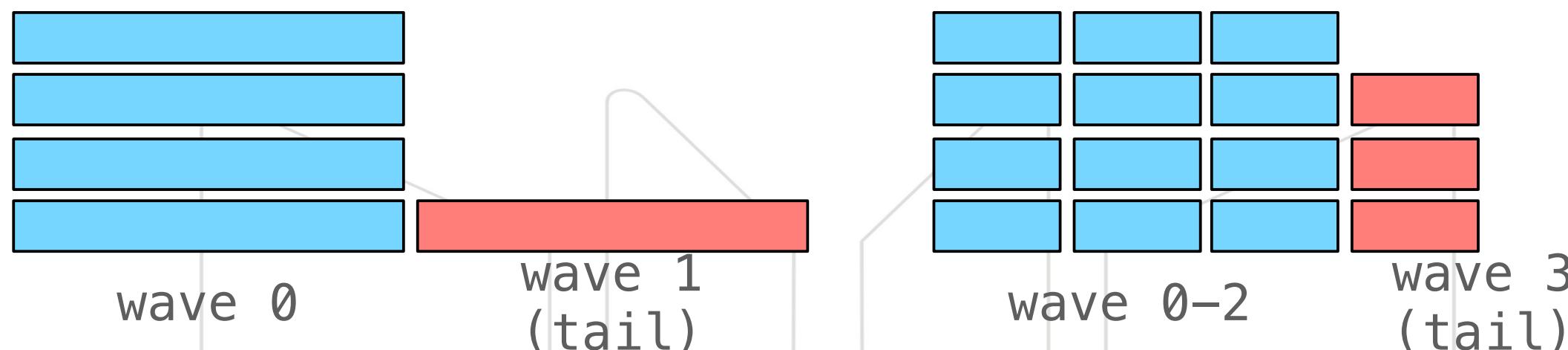
```
__host__ cudaError_t cudaOccupancyMaxPotentialBlockSize ( int* minGridSize,  
int* blockSize, T func, size_t dynamicSMemSize, int blockSizeLimit) [inline]
```

- `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

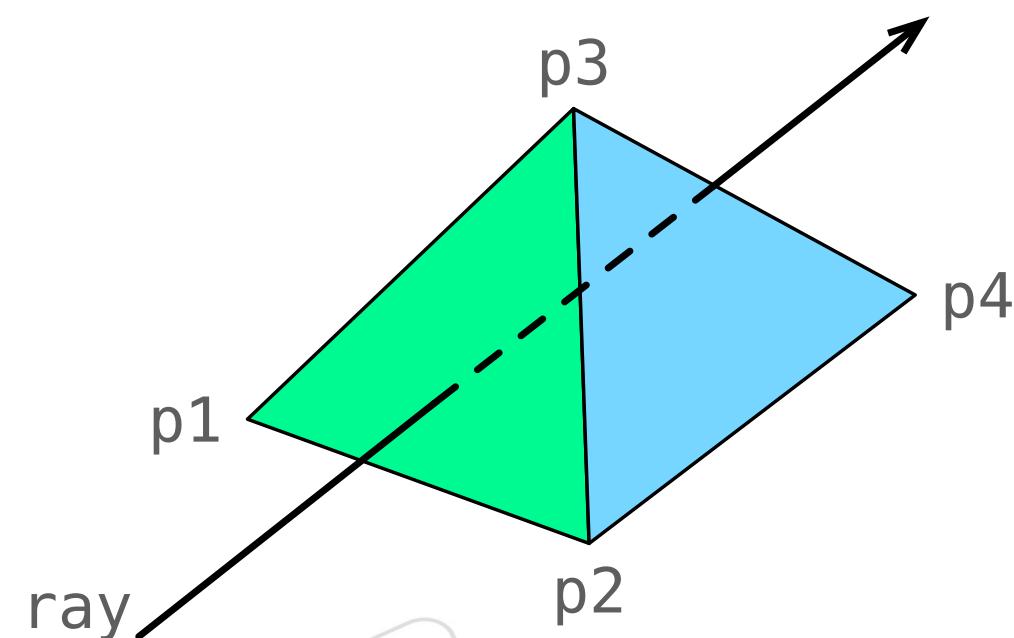
- <http://on-demand.gputechconf.com/gtc/2012/presentations/S0514-GTC2012-GPU-Performance-Analysis.pdf>
- Wave: 在设备上同时并行的线程块
- Tail: 处理无法被整除的工作量中余下部分的线程块
- 意义: 过大的线程块可能无法充分利用硬件资源
 - 但过小的线程块可能限制利用率
 - SM上同时活跃的线程块数量有限 (Kepler为16, Maxwell为32)



● 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：

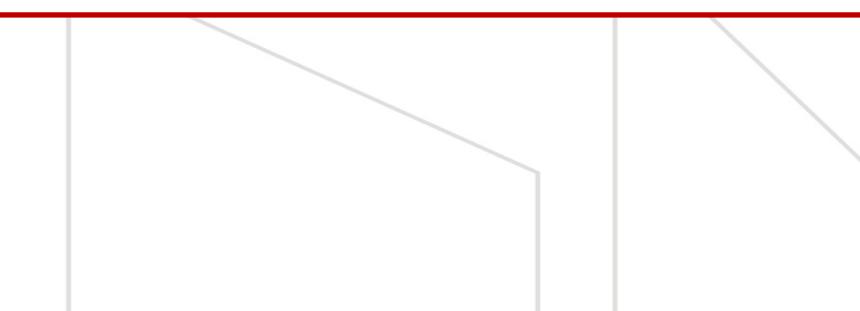
```
__global__ void kernel(vector* rays, quad* quads){  
    int tid = blockIdx.x*blockDim.x+threadIdx.x;  
    vector ray = rays[tid];  
    quad q = quads[tid]; // p1, p2, p3, p4  
  
    if (ray_hit_triangle_1){  
        do_something(q.p1, q.p2, q.p3);  
    } else {  
        do_something(q.p2,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



● 全局内存对齐与合并访问

- CUDA程序提升性能的关键
- 使程序在线程束内线程访问尽可能访问连续内存
 - 与共享内存的存储体冲突区分开
- 具体策略在实践中根据硬件决定
 - 是否支持一级缓存、合并内存事务的存取大小等

● 其他优化策略

- 优化占用率
- 选择合适的线程块大小
- 降低分支分流时分支中的工作量
- 选择性地应用串行程序的优化策略

Questions?

