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内存结构

陶钧

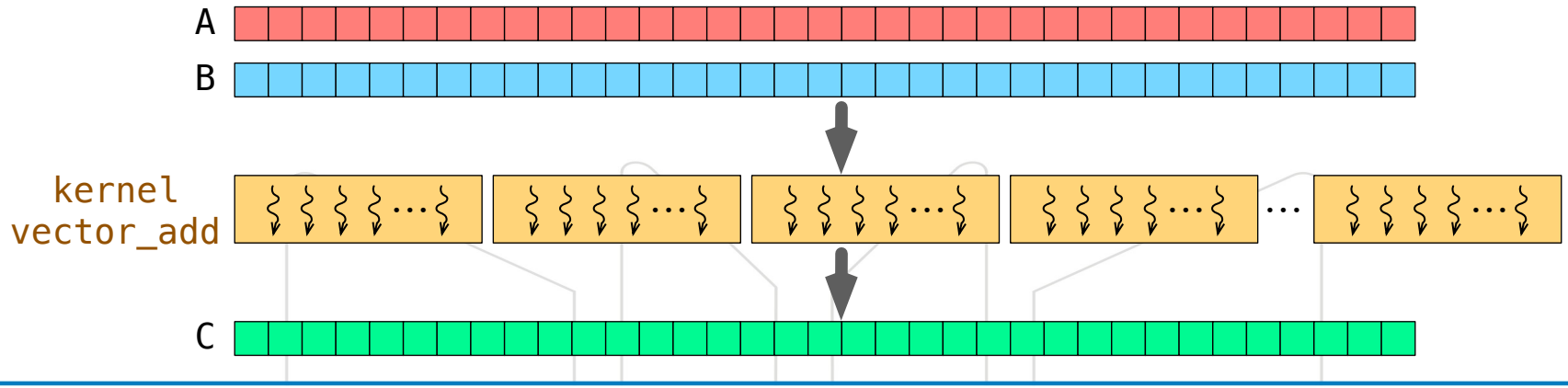
taoj23@mail.sysu.edu.cn

中山大学 数据科学与计算机学院

国家超级计算广州中心

上周回顾

- CUDA架构特征
  - 异构、SIMD与SPMD特性、大量超轻量级线程、高吞吐量
- CUDA编程结构
  - __host__, __global__ (kernel) , __device__
  - 网格（grid）与块（block）
- CUDA编程举例
  - 向量相加

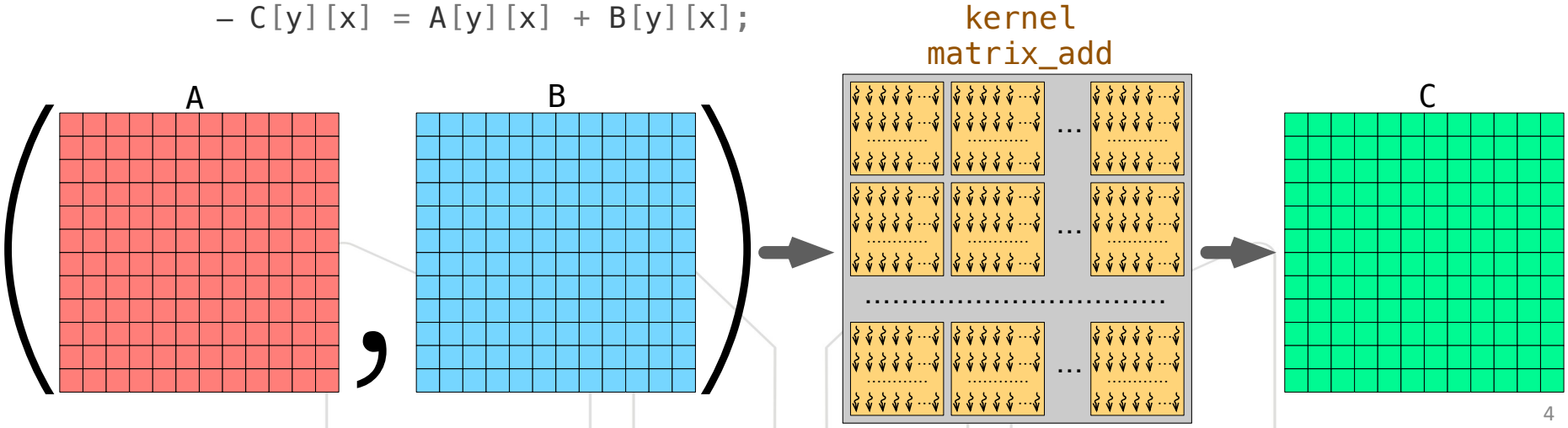


课程提纲

- 线程组织与内存结构
- CUDA内存模型
- 全局内存
- 常量内存
- 只读/纹理内存
- 共享内存

线程组织与内存结构

- 例子：矩阵相加
  - 直接将一维gird与block扩展至二维
    - 每个线程负责一个位置上的数字相加
    - x、y方向上分别求全局坐标
      - `int x = blockDim.x * blockIdx.x + threadIdx.x;`
      - `int y = blockDim.y * blockIdx.y + threadIdx.y;`
      - `C[y][x] = A[y][x] + B[y][x];`



线程组织与内存结构

- 例子：矩阵相加
  - 直接将一维gird与block扩展至二维

```
__global__ void matrix_add(int **A, int **B, int **C, int n, int m){
    int x = blockDim.x * blockIdx.x + threadIdx.x;
    int y = blockDim.y * blockIdx.y + threadIdx.y;

    if (y<n && x<m ){
        C[y][x] = A[y][x] + B[y][x];
    }
}

dim3 block(w, h, 1);
dim3 grid(divup(m, w), divup(n, h), 1);
matrix_add<<< grid, block >>>(A, B, C);
```

线程组织与内存结构

- 例子：矩阵相加
  - 直接将一维gird与block扩展至二维
    - 存在问题：需要二级指针结构创建二维数组
    - CPU

```
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];
```

    - GPU

```
__global__ void init_matrix(int **A, int *A_data, int n, int m){
    int tid = blockDim.x * blockIdx.x + threadIdx.x;
    if (tid<n){
        A[tid] = &A_data[tid*m];
    }
}

int *A_data, **A;
cudaMalloc((void*)&A_data, sizeof(int)*n*m);
cudaMalloc((void*)&A, sizeof(int*)*n);
```

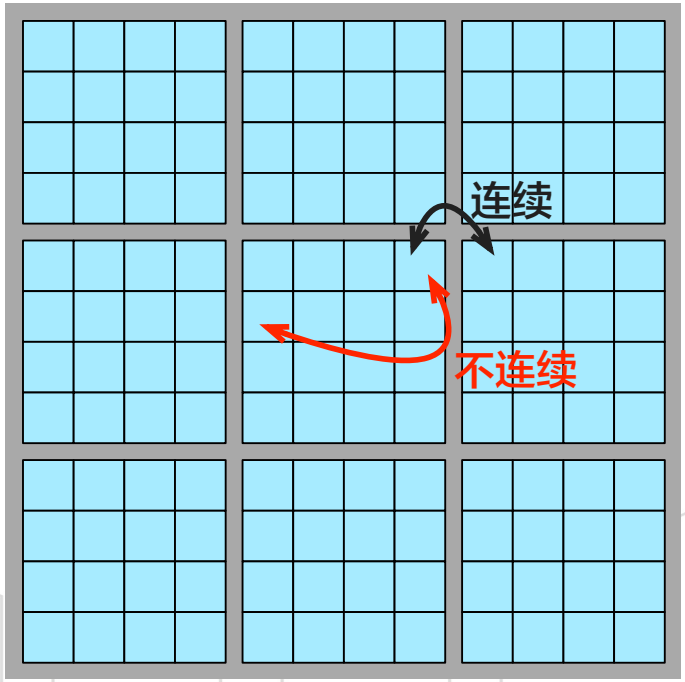
线程组织与内存结构

- 例子：矩阵相加
  - 直接将一维gird与block扩展至二维
    - 存在问题：需要二级指针结构创建二维数组
    - 解决方案：使用一维数组存放矩阵
      - 即使在前一个版本中，数据在内存中也是一维连续（int *A_data）
      - int **A只提供访问数据的方式（ A[y][x]增加内存访问次数！）

```
__global__ void matrix_add(int *A, int *B, int *C, int n, int m){
    int x = blockDim.x * blockIdx.x + threadIdx.x;
    int y = blockDim.y * blockIdx.y + threadIdx.y;

    if (y<n && x<m ){
        C[y*m+x] = A[y*m+x] + B[y*m+x];
    }
}

dim3 block(w, h, 1);
dim3 grid(divup(m, w), divup(n, h), 1);
matrix_add<<< grid, block >>>(A, B, C);
```

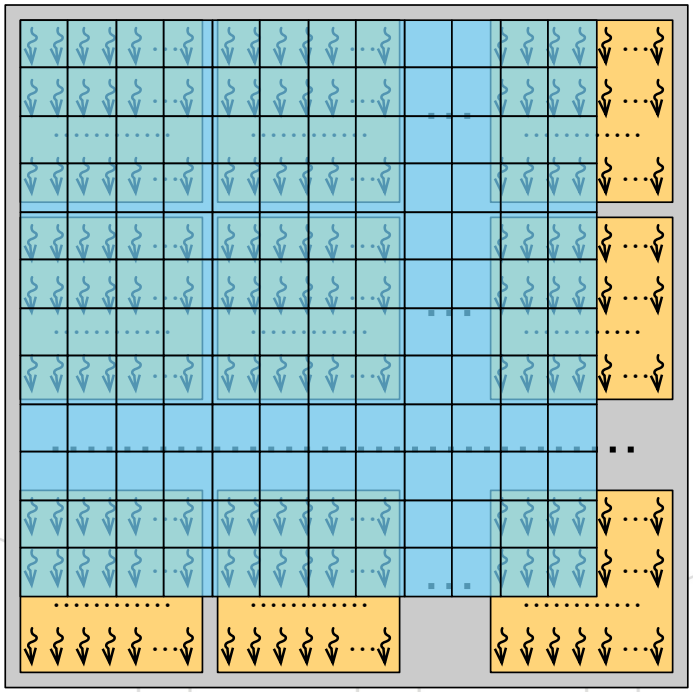
线程组织与内存结构

- 例子：矩阵相加
  - 直接将一维gird与block扩展至二维
    - 存在问题：block中线程访问的内存空间不连续
      - 内存访问的局部性差



线程组织与内存结构

- 例子：矩阵相加
  - 直接将一维gird与block扩展至二维
    - 存在问题：在x、y维度上都可能出现余数
      - 线程利用率低



线程组织与内存结构

- 例子：矩阵相加
  - 直接将一维gird与block扩展至二维
    - 存在问题：block中线程访问的内存空间不连续
      在x、y维度上都可能出现余数
    - 解决方案：使用一维grid与block
      - 与vector_add(A, B, C, n*m)等价!

```
__global__ void matrix_add(int *A, int *B, int *C, int n, int m){
    int tid = blockDim.x * blockIdx.x + threadIdx.x;

    if ( tid<n*m ){
        C[tid] = A[tid] + B[tid];
    }
}

matrix_add<<< divup(n*m, block_size), block_size >>>(A, B, C);
```

线程组织与内存结构

- 例子：矩阵相加
  - 线程组织方式和内存中数据的组织方式可以互相独立
    - 二维grid与block也可以用于一维数据，反之亦然
  - 相同的组织方式在编程中显得更直观
    - 但效率不一定最优
    - 效率归根结底由硬件结构决定

课程提纲

- 线程组织与内存结构
- CUDA内存模型
- 全局内存
- 常量内存
- 只读/纹理内存
- 共享内存

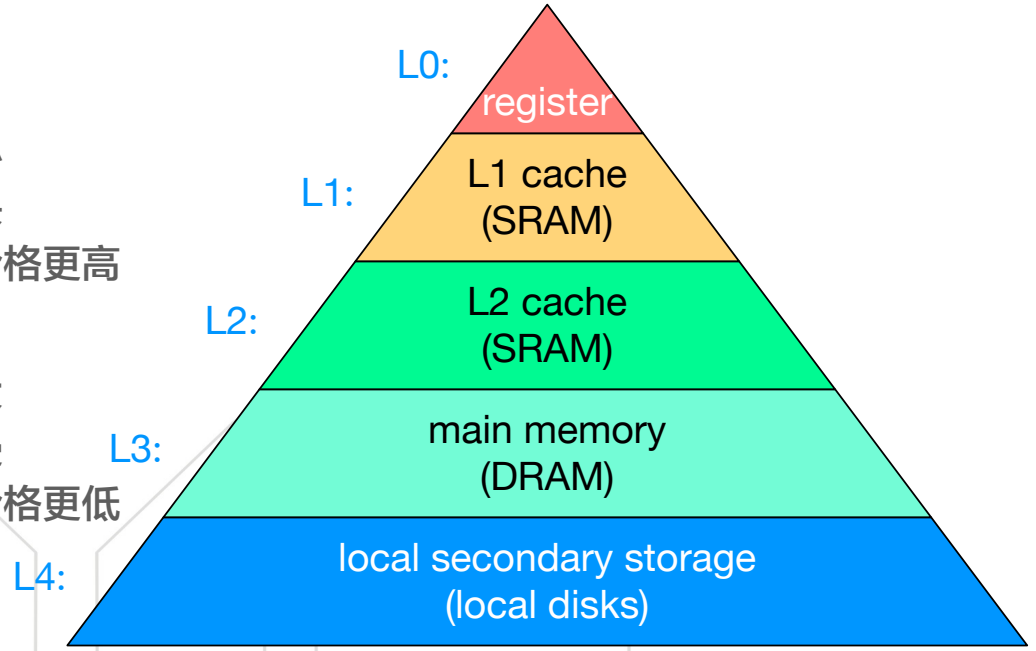
内存层次结构

- 应用程序往往遵循局部性原则
  - 访问数据/代码的模式并不是完全任意的
  - 时间局部性：数据在较短时间内很可能被重复访问
  - 空间局部性：临近位置很可能被访问
  - 多级内存层次结构
    - 优化性能、降低延迟
    - 可编程的存储器
    - 不可编程的存储器
      - L1、L2

是否能显式控制哪些数据存放在存储器中

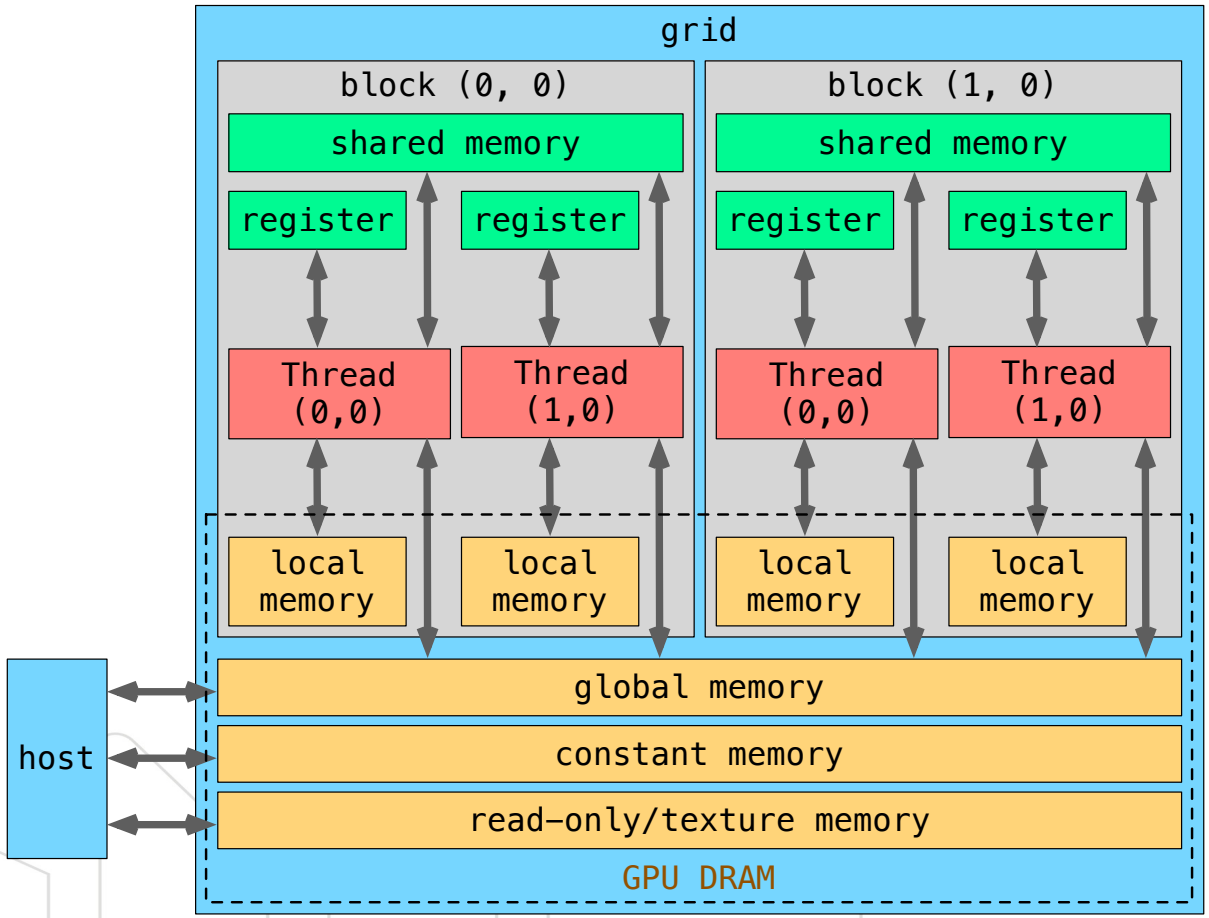
空间更小
速度更快
每字节价格更高

空间更大
速度更慢
每字节价格更低



CUDA内存模型

- 每个线程
  - 寄存器
  - 本地内存
- 线程块
  - 共享内存
- 所有线程
  - 全局内存
  - 常量内存
  - 纹理内存



CUDA内存模型

- CUDA变量与类型修饰符
  - 没有修饰符的变量将被置于寄存器
    - 超过寄存器限制的变量将被置于本地内存
    - 极大地降低程序效率！
  - 没有修饰符的数组将被置于寄存器/本地内存

变量声明	存储器	作用域	生存周期
`int var;`	寄存器	线程	线程
`int array_var[100];`	寄存器/本地	线程	线程
`__shared__ int shared_var;`	共享	线程块	线程块
`__device__ int global_var;`	全局	全局	应用程序
`__constant__ int constant_var;`	常量	全局	应用程序

CUDA内存模型

- 本地内存
  - 每个线程独立读写
  - 并非物理存在
    - 与全局内存在同一块存储区域
    - 计算能力2.0以上的CPU中，存储在SM的一级缓存以及设备的二级缓存
  - 可能存放到本地内存的变量：
    - 编译时使用未知索引引用的本地数组
    - 可能占用大量寄存器空间的本地数组
    - 不满足寄存器限定的变量

```
__global__ void local_memory
(int *input)
{
    int a;
    int b;
    int index;

    int array1[4];
    int array2[4];
    int array3[100];

    index = input[threadIdx.x];
    a = array1[0];
    b = array2[index];
}
```

CUDA内存模型

- CUDA变量与类型修饰符
  - 存储器位置与访问开销

变量声明	存储器	片上/片外	缓存	访问开销
`int var;`	寄存器	片上	NA	1
`int array_var[100];`	本地	片外	是*	1 （L1缓存） 200-800
`__shared__ int shared_var;`	共享	片上	NA	~1
`__device__ int global_var;`	全局	片外	是*	200-800
`__constant__ int constant_var;`	常量	片外	是	~1（缓存）

*只在计算能力2.0的设备上进行缓存

CUDA内存模型

- GPU DRAM访问模式
  - 全局内存
    - 通过二级缓存或配置共享内存
  - 常量内存
    - 通过二级缓存或线程块的常量缓存（需Kepler以后的设备）
  - 只读/纹理内存
    - 通过二级缓存或线程块的只读缓存（需Kepler以后的设备）

CUDA内存模型

- 了解你的硬件！
  - 不同系列的GPU之间参数及设计都可能有变动
    - 如，Fermin与Kepler中共享内存与一级缓存共用同一块存储区域，而Maxwell与Pascal中共享内存使用独立缓存而只读/纹理缓存与一级缓存共用同一块存储区域

Technical specifications	Compute capability (version)																
	1.0	1.1	1.2	1.3	2.x	3.0	3.2	3.5	3.7	5.0	5.2	5.3	6.0	6.1	6.2	7.0 (7.2?)	7.5
Maximum number of 32-bit registers per thread	124				63		255										
Maximum amount of shared memory per multiprocessor	16 KB				48 KB				112 KB	64 KB	96 KB	64 KB		96 KB	64 KB	96 KB (of 128)	64 KB (of 96)
Maximum amount of shared memory per thread block	48 KB															48/96 KB	64 KB
Number of shared memory banks	16				32												
Amount of local memory per thread	16 KB				512 KB												
Constant memory size	64 KB																
Cache working set per multiprocessor for constant memory	8 KB												4 KB	8 KB			
Cache working set per multiprocessor for texture memory	6 – 8 KB				12 KB		12 – 48 KB			24 KB	48 KB	N/A	24 KB	48 KB	24 KB	32 – 128 KB	32 – 64 KB

图片截取自https://en.wikipedia.org/wiki/CUDA

CUDA内存模型

- 了解你的硬件！
  - 不同系列的GPU之间参数及设计都可能有变动
    - 使用**cudaGetDeviceProperties**查询
      - 如 prop.sharedMemPerBlock

```
int nDevices;
cudaGetDeviceCount(&nDevices);

for (int i = 0; i < nDevices; i++) {
    cudaDeviceProp prop;
    cudaGetDeviceProperties(&prop, i);
    printf(" Device Number: %d\n", i);
    ...
}
```

```
deviceQuery.exe Starting...

 CUDA Device Query (Runtime API) version (CUDART static linking)

Detected 1 CUDA Capable device(s)

Device 0: "GeForce GTX 680"
  CUDA Driver Version / Runtime Version          6.0 / 6.0
  CUDA Capability Major/Minor version number:    3.0
  Total amount of global memory:                 2048 MBytes (2147483648 bytes)
  ( 8) Multiprocessors, (192) CUDA Cores/MP:     1536 CUDA Cores
  GPU Clock rate:                                1059 MHz (1.06 GHz)
  Memory Clock rate:                             3004 Mhz
  Memory Bus Width:                              256-bit
  L2 Cache Size:                                 524288 bytes
  Maximum Texture Dimension Size (x,y,z)         1D=(65536), 2D=(65536, 65536), 3D=(4096, 4096, 4096)
  Maximum Layered 1D Texture Size, (num) layers  1D=(16384), 2048 layers
  Maximum Layered 2D Texture Size, (num) layers  2D=(16384, 16384), 2048 layers
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
  Total number of registers available per block: 65536
  Warp size:                                     32
  Maximum number of threads per multiprocessor:  2048
  Maximum number of threads per block:           1024
  Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
  Max dimension size of a grid size    (x,y,z): (2147483647, 65535, 65535)
  Maximum memory pitch:                          2147483647 bytes
  Texture alignment:                             512 bytes
  Concurrent copy and kernel execution:          Yes with 1 copy engine(s)
  Run time limit on kernels:                     Yes
  Integrated GPU sharing Host Memory:            No
  Support host page-locked memory mapping:       Yes
  Alignment requirement for Surfaces:            Yes
  Device has ECC support:                        Disabled
  CUDA Device Driver Mode (TCC or WDDM):         WDDM (Windows Display Driver Model)
  Device supports Unified Addressing (UVA):      No
  Device PCI Bus ID / PCI location ID:           1 / 0
  Compute Mode:
     < Default (multiple host threads can use ::cudaSetDevice() with device simultaneously) >

deviceQuery, CUDA Driver = CUDART, CUDA Driver Version = 6.0, CUDA Runtime Version = 6.0, NumDevs = 1, Device0 = GeForce GTX 680
Result = PASS
```

图片来自NVIDIA

课程提纲

- 线程组织与内存结构
- CUDA内存模型
- 全局内存
- 常量内存
- 只读/纹理内存
- 共享内存

全局内存

- 动态与静态全局内存
  - 动态全局内存管理
    - **cudaMalloc()**, **cudaMemcpy()**, **cudaFree()**
  - 静态全局内存
    - 通过__device__修饰符声明
    - 使用**cudaMemcpyToSymbol()**与**cudaMemcpyFromSymbol()**在主机端与设备端之间拷贝

```
#define N 1024
__device__ int d_a[N];

__global__ void kernel(){
    int tid = ...
    d_a[tid] ...
}

int main(){
    int size = sizeof(int)*N;
    int *h_a = (int*)malloc(size);
    init_data(h_a);

    cudaMemcpyToSymbol(d_a, h_a, size);
    kernel<<<...>>>();
    cudaMemcpyFromSymbol(h_a, d_a, size);

    free(h_a);
}
```

全局内存

- 动态与静态全局内存
  - 与C中静态/动态数组的关系类似
    - `int a[N];`
    - `int *a = (int*)malloc(sizeof(int)*N);`

动态：

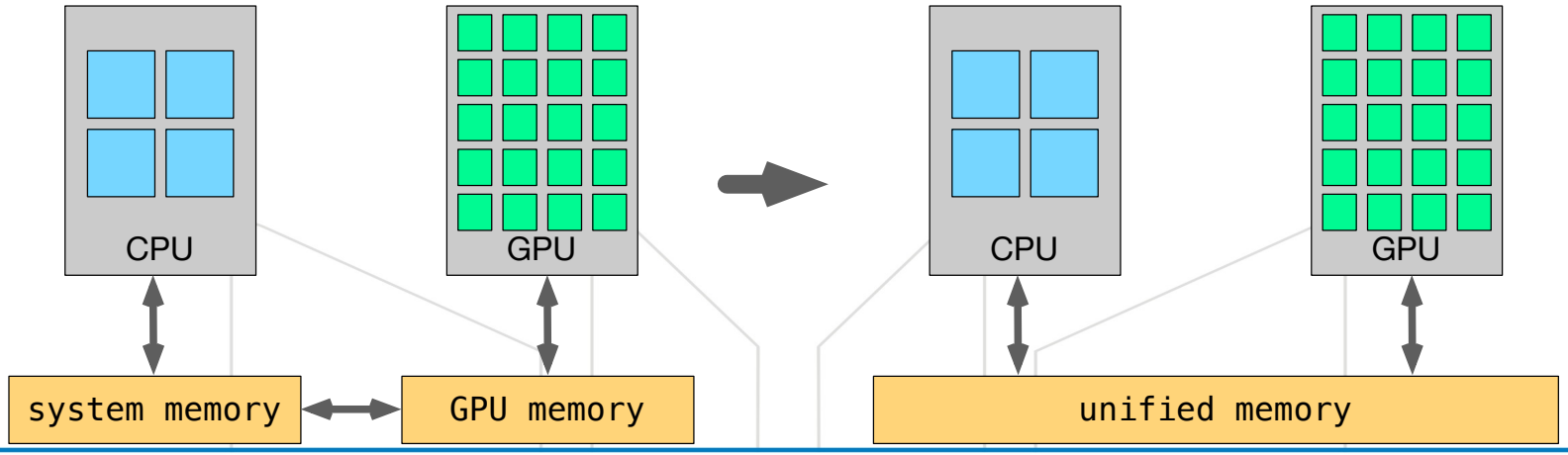
```
int *h_a = (int*)malloc(size);
init_data(h_a);

int *d_a;
cudaMalloc((void*)&d_a, size);

cudaMemcpy(d_a, h_a, size, cudaMemcpyHostToDevice);
kernel<<<...>>>(d_a);
cudaMemcpy(h_a, d_a, size, cudaMemcpyDeviceToHost);

cudaFree(d_a);
free(h_a);
```

静态：

```
#define N 1024
__device__ int d_a[N];

__global__ void kernel(){
    int tid = ...
    d_a[tid] ...
}

int main(){
    int size = sizeof(int)*N;
    int *h_a = (int*)malloc(size);
    init_data(h_a);

    cudaMemcpyToSymbol(d_a, h_a, size);
    kernel<<<...>>>();
    cudaMemcpyFromSymbol(h_a, d_a, size);

    free(h_a);
}
```

全局内存

- 统一内存寻址（unified memory）
  - 使用相同的内存地址（指针）在主机和设备上进行访问
    - 统一内存中创建托管内存池
    - 底层系统在统一内存空间中自动在主机和设备间进行传输
    - 简化程序员视角中的内存管理
  - 需要CUDA 6.0+与计算能力 3.0+（Kepler及以上架构）



全局内存

- 统一内存寻址（unified memory）
  - 例子（来自NVIDIA）

C源码：

```
void sortfile(FILE* fp, int N){
    char* data;
    data = (char*)malloc(N);

    fread(data, 1, N, fp);

    qsort(data, N, 1, compare);

    use_data(data);

    free(data);
}
```

CUDA源码：

```
void sortfile(FILE* fp, int N){
    char* data;
    cudaMallocManaged(&data, N);

    fread(data, 1, N, fp);

    qsort<<<...>>>(data, N, 1, compare);
    cudaDeviceSynchronize();

    use_data(data);

    cudaFree(data);
}
```

全局内存

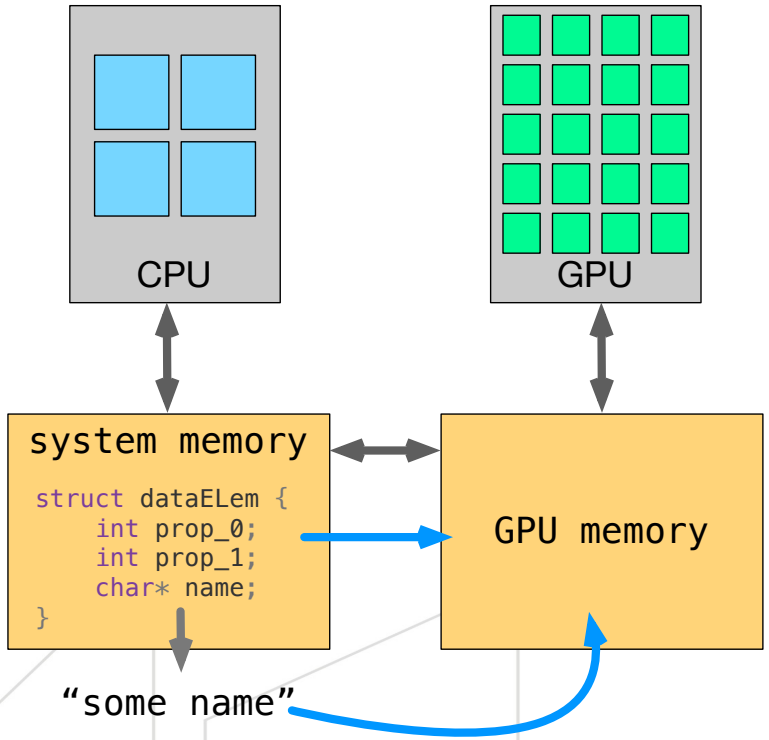
- 统一内存寻址（unified memory）
  - 优点
    - 更容易将C代码移植到CUDA
    - 更方便数据管理
      - 不需要显式控制数据在主机与设备端的传输
      - 与操作系统管理虚拟内存方式相似
  - 缺点
    - 内存只是“虚拟统一”
      - GPU上内存依然独立于主机内存外
      - 依然需要通过PCIe或NVLINK传输数据
    - 需要适当的页面调度及同步保证内存中数据正确
    - 与显示控制内存相比效率往往更低
      - 在课程中我们依然手动管理内存！

全局内存

- 深度拷贝
  - 复杂数据结构需要多次拷贝
    - 如结构体

```
struct dataELem {
    int prop_0;
    int prop_1;
    char* name;
}
```

    - 需要两次拷贝
    - 一次拷贝结构体成员变量
    - 一次拷贝数组name
      - 还需要动态分配内存

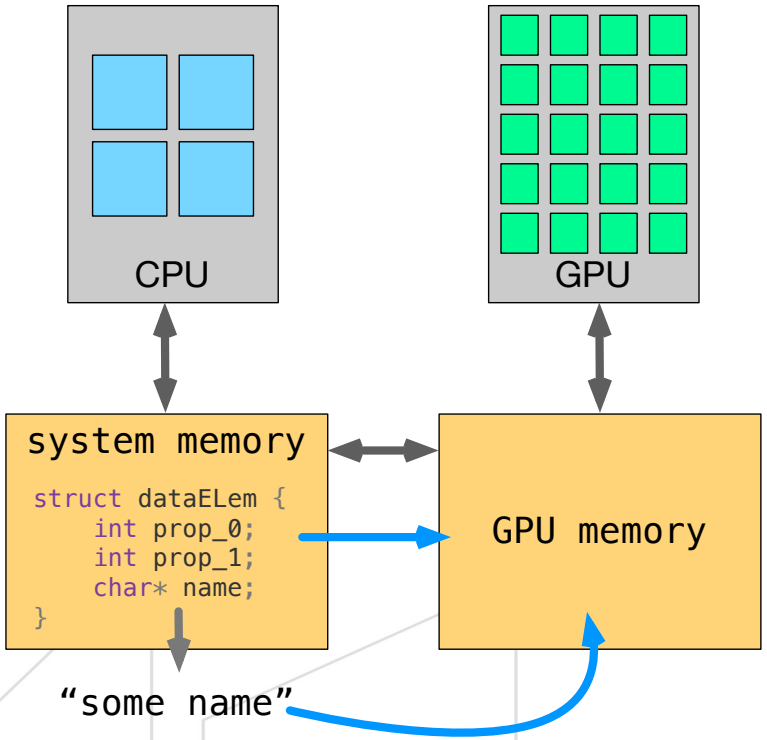


全局内存

- 深度拷贝
  - 使用统一内存寻址

```
class Managed {
public:
    void *operator new(size_t len) {
        void *ptr;
        cudaMallocManaged(&ptr, len);
        cudaDeviceSynchronize();
        return ptr;
    }

    void operator delete(void *ptr) {
        cudaDeviceSynchronize();
        cudaFree(ptr);
    }
};
```

代码来自NVIDIA



全局内存

- 深度拷贝
  - 个人习惯（仅供参考）
    - 在结构体中不使用指针，而使用一个index表明数据位置
      - 需要传输一个数组的dataElem时，只需两次拷贝
        - » 一次拷贝dataElem数组
        - » 一次拷贝数据（）all_names
      - 数据（name）在内存空间中连续
    - 使用统一内存寻址
      - 需要托管多个内存
        - » 不一定连续，可能需要多次传输

```
struct dataElem{
    int prop_0;
    int prop_1;
    char* name;
}
```

```
struct dataElem{
    int prop_0;
    int prop_1;
    int name_pos, name_len;
}

char* all_names;
```

课程提纲

- 线程组织与内存结构
- CUDA内存模型
- 全局内存
- 常量内存
- 只读/纹理内存
- 共享内存

常量内存

- 常量内存
  - 存储与GPU DRAM中（与全局内存一样）
  - 每个SM上有专用的片上缓存
  - 常量缓存中读取的延迟比常量内存中低的多
  - 在运行时设置
- 使用
  - 变量定义：使用__constant__修饰词
  - 值拷贝：使用**cudaMemcpyToSymbol**（与静态全局变量一致）
    - 用于少量只读数据
- 访问行为
  - 线程束中线程访问不同地址，则访问需要串行
    - 常量内存读取成本与线程束中线程读取唯一地址数量呈线性关系

常量内存

- 常量内存访问举例
  - 哪种访问更有效率？

```
__constant__ int const_var[16];

__global__ void kernel(){
    int i = blockIdx.x;
    int value = const_var[i%16];
}
```

```
__constant__ int const_var[16];

__global__ void kernel(){
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int value = const_var[i%16];
}
```

常量内存

- 常量内存访问举例
  - 哪种访问更有效率？

```
__constant__ int const_var[16];

__global__ void kernel(){
    int i = blockIdx.x;
    int value = const_var[i%16];
}
```

```
__constant__ int const_var[16];

__global__ void kernel(){
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int value = const_var[i%16];
}
```

- 常量内存的最佳访问模式
  - 基于blockIdx访问
  - 所有线程访问同一内存（广播访问）
- 无串行访问
  - 只需要一次内存读取
- 线程块中其他线程所需数据也同样会命中缓存

- 常量内存的最差访问模式
  - 基于threadIdx访问
  - 线程访问多个不同内存
- 需要串行访问
  - 需要16次内存读取
- 线程块中其他线程所需数据可能不会命中缓存

常量内存

- 常量内存 vs 宏定义
  - 宏定义由预处理器进行文字替换
    - 不占用寄存器
    - 存在于指令空间中
  - 何时使用常量内存/宏定义？
    - 宏定义中的值成为应用程序的一部分适用于编译后不再修改的值
    - 常量内存适用于在执行中可能更改的值（在GPU代码执行过程中不变）

课程提纲

- 线程组织与内存结构
- CUDA内存模型
- 全局内存
- 常量内存
- 只读/纹理内存
- 共享内存

只读/纹理内存

- 只读内存与纹理内存
  - Kepler架构中相互独立
  - 在此后架构中占用同一块内存空间（GPU DRAM）
- 特点
  - 数据均为只读
    - 不能在设备端代码中修改
  - 满足空间局部性的读取更有效率
    - 图形学代码访问纹理的特性
- 使用
  - 通过绑定到底层内存的纹理引用读取
  - 通过修饰词指示编译器使用只读内存

只读/纹理缓存

- 只读缓存
  - 使用内部函数__ldg()
    - __ldg()用于代替标准指针解引用，并且强制通过只读数据缓存加载

```
__global__ void kernel(int *buffer) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int x = __ldg(&buffer[i]);
}

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

只读/纹理缓存

- 只读缓存
  - 使用全局内存的限定指针
    - 使用**const __restrict__**表明数据应该通过只读缓存被访问

```
__global__ void kernel(const int* __restrict__ buffer){
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int x = buffer[i];
}

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

只读/纹理内存

- 常量缓存与只读缓存
  - 常量缓存与之都缓存都是只读的
  - 在SM上为相互独立的硬件
  - 常量缓存更适用于统一读取
    - 线程束中的每一个线程都访问相同的地址
  - 只读缓存更适用于分散读取
    - 线程束中的每一个线程访问不同地址

课程提纲

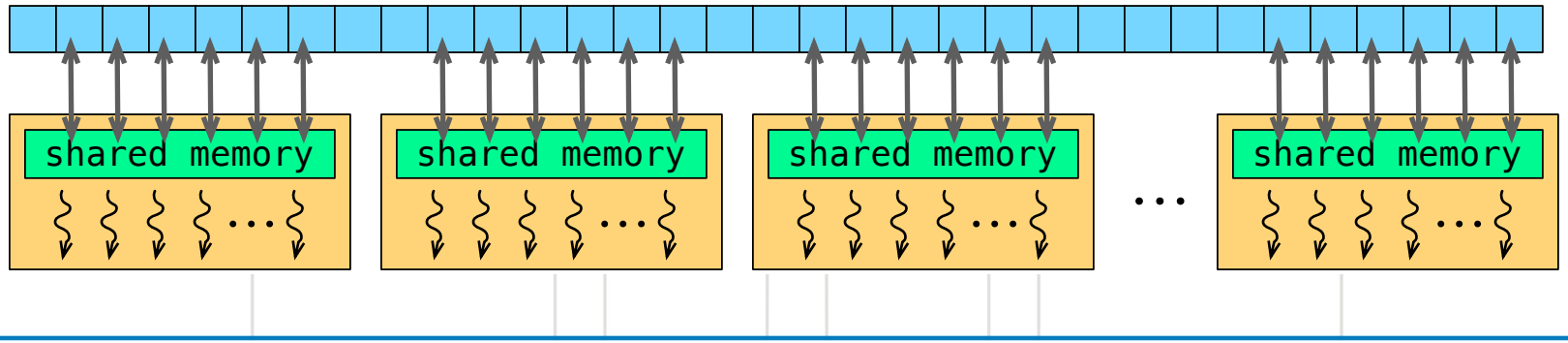
- 线程组织与内存结构
- CUDA内存模型
- 全局内存
- 常量内存
- 只读/纹理内存
- 共享内存

共享内存

- 可编程的缓存
  - 可以显式控制载入/同步数据
  - 片上存储
  - 读写速度非常快
    - 带宽 > 1 TB/s
  - 基于线程块
    - 允许同一线程块中的线程共享部分数据
    - 无法同步不同线程块中的线程

共享内存

- 基于线程块的共享内存读写模型
  - 线程块往往只需要部分数据
  - 流程
    - 将全局数据切分成小块
    - 在核函数中将线程块所需的一个小块数据载入共享内存
    - 运行核函数进行计算
    - 核函数结束前将数据拷贝至全局内存



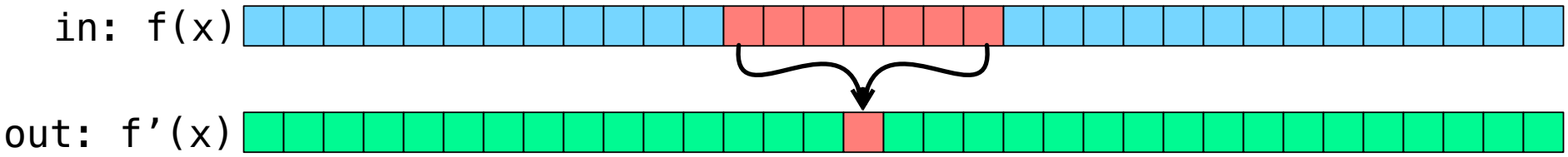
共享内存

- 读写模型：传输 -> 执行 -> 传输
  - 主机视角：
    - 传输：主机内存至显存
    - 执行：核函数
    - 传输：显存至主机内存
  - 设备视角：
    - 传输：设备内存至寄存器
    - 执行：指令
    - 传输：寄存器至设备内存
  - 程序块与共享内存视角：
    - 传输：共享内存至寄存器
    - 执行：指令
    - 传输：寄存器至共享内存

- 举例：估计一阶偏导
  - 六阶中心差分公式

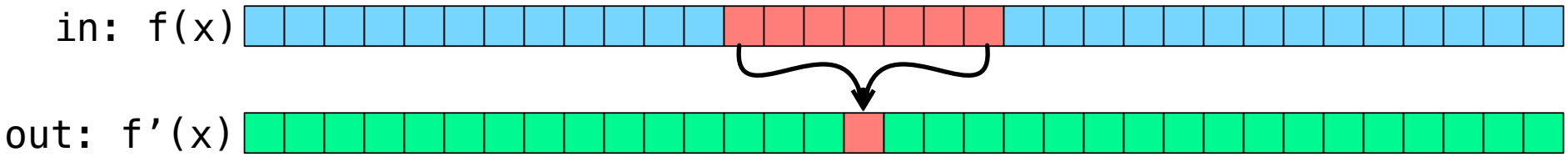
$$f'(x) \approx c_0(f(x+3h)-f(x-3h)) + c_1(f(x+2h)-f(x-2h)) + c_2(f(x+h)-f(x-h))$$

  - 计算输出f'(x)所需数据为输入f(x)中以x为中心的7个数



共享内存

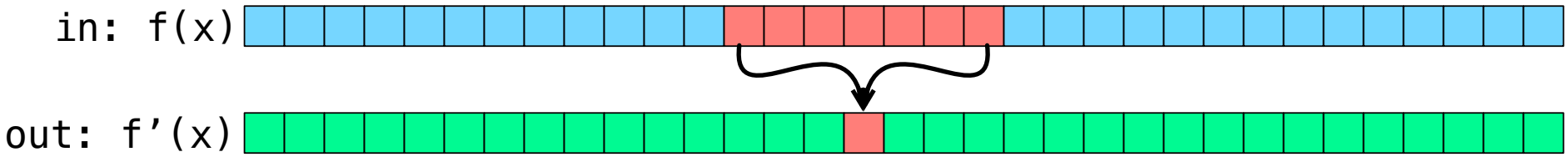
- 举例：估计一阶偏导




```
#define RADIUS 3
__constant__ float c[RADIUS+1];

__global__ void stencil(float *in, float *out){
    int tid = blockIdx.x*blockDim.x + threadIdx.x;

    float tmp = 0.0f;
    for(int i = 1; i <=RADIUS; ++i){
        tmp += c[i]*(in[tid+i]-in[tid-i]);
    }
    out[tid] = tmp;
}
```

共享内存

- 举例：估计一阶偏导




```
#define RADIUS 3
__constant__ float c[RADIUS+1];

__global__ void stencil(float *in, float *out){
    int tid = blockIdx.x*blockDim.x + threadIdx.x;

    float tmp = 0.0f;
    for(int i = 1; i <=RADIUS; ++i){
        tmp += c[i]*(in[tid+i]-in[tid-i]);
    }
    out[tid] = tmp;
}
```

是否能使用只读内存？

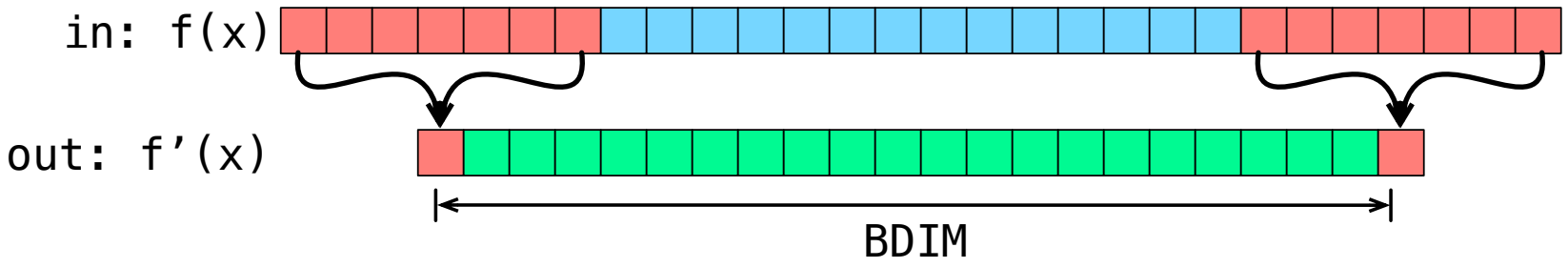
是否能使用out[tid]？

共享内存

- 举例：估计一阶偏导
  - 存在问题：大量重复全局内存访问
    - 每个block（大小为BDIM）所需全局内存访问次数为
      - BDIM*6

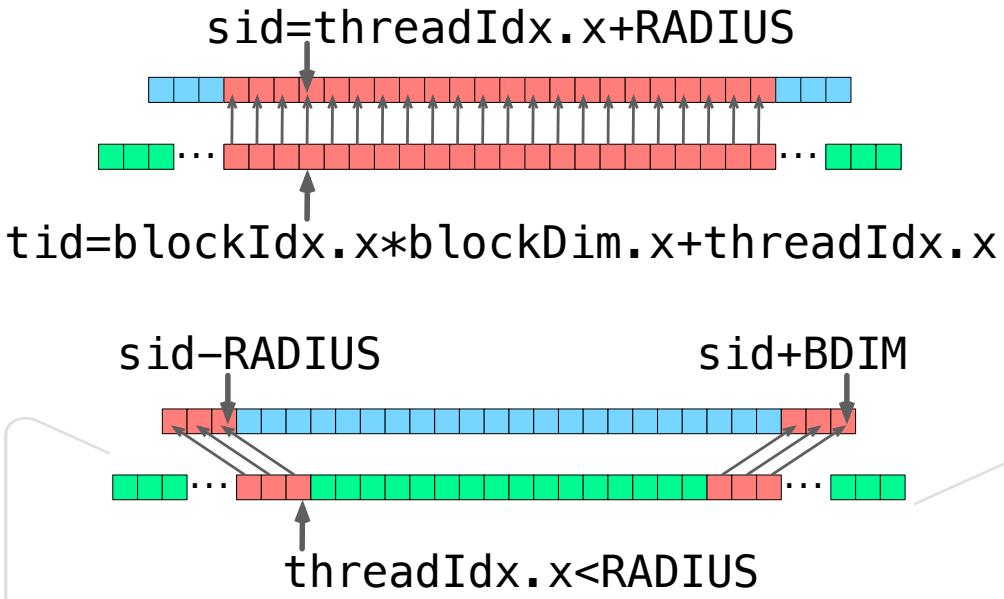
```
for(int i = 1; i <=RADIUS; ++i){
        tmp += c[i]*(in[tid+i]-in[tid-i]);
    }
```

    - 实际需要数据个数为BDIM+6



共享内存

- 举例：估计一阶偏导
  - 存在问题：大量重复全局内存访问
  - 解决方案：一次性将全局内存读入至共享内存



共享内存

- 举例：估计一阶偏导
  - 使用共享内存

```
__global__ void stencil(float *in, float *out){
    __shared__ float smem[BDIM+2*RADIUS];

    //thread index to global memory
    int tid = blockIdx.x*blockDim.x + threadIdx.x;
    //index to shared memory
    int sid = threadIdx.x + RADIUS;

    //copy to shared memory
    smem[sid] = in[tid];
    if (threadIdx.x < RADIUS) {
        smem[sid-RADIUS] = in[tid-RADIUS];
        smem[sid+BDIM] = in[tid+BDIM];
    }
    __syncthreads();

    float tmp = 0.0f;
    for(int i = 1; i <=RADIUS; ++i){
        tmp += c[i]*(smem[sid+i]-smem[sid-i]);
    }

    out[tid] = tmp;
}
```

共享内存

- 共享内存分配
  - 静态：编译时指定（常数、宏定义）

```
__global__ void kernel(int *in){
    __shared__ int smem[N];
    ...
}
```

  - 动态：运行时通过执行配置指定
    - 注意：__shared__ int *ptr是共享变量（指针）而非空间本身

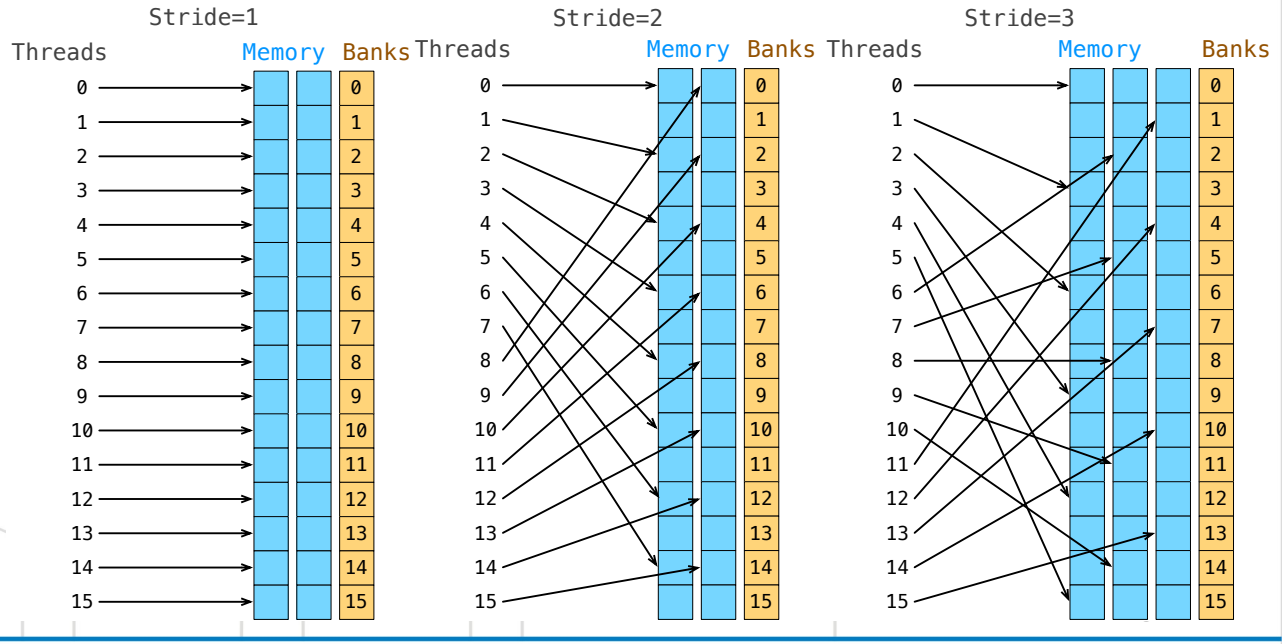
```
__global__ void kernel(int *in){
    extern __shared__ int smem[];
    ...
}

kernel<<<grid, block, smem_size>>>(in);
```

共享内存

- 存储体（bank）和访问模式
  - 共享内存被分为32个同样大小的内存模型（存储体）
    - 不同存储体可同时被访问
  - 访问模式
    - 并行访问：多个地址访问多个存储体
    - 串行访问：多个地址访问同一存储体
    - 广播访问：单一地址读取单一存储体
  - 存储体冲突（bank conflict）
    - 多个地址访问同一个存储体（串行）
    - 广播访问不引发存储体冲突
    - 只发生在同一个线程束的线程中
      - 32个存储体与32个线程

共享内存

- 存储体冲突
  - 每4字节为同一存储体
    - 示意图中简化为16个存储体
  - 规则访问
    - 步长为1
      - 访问整型、单精度浮点数
      - 无冲突
    - 步长为2
      - 访问双精度浮点数
      - 2-way 冲突
    - 步长为3
      - 如12字节的结构体
      - 无冲突



共享内存

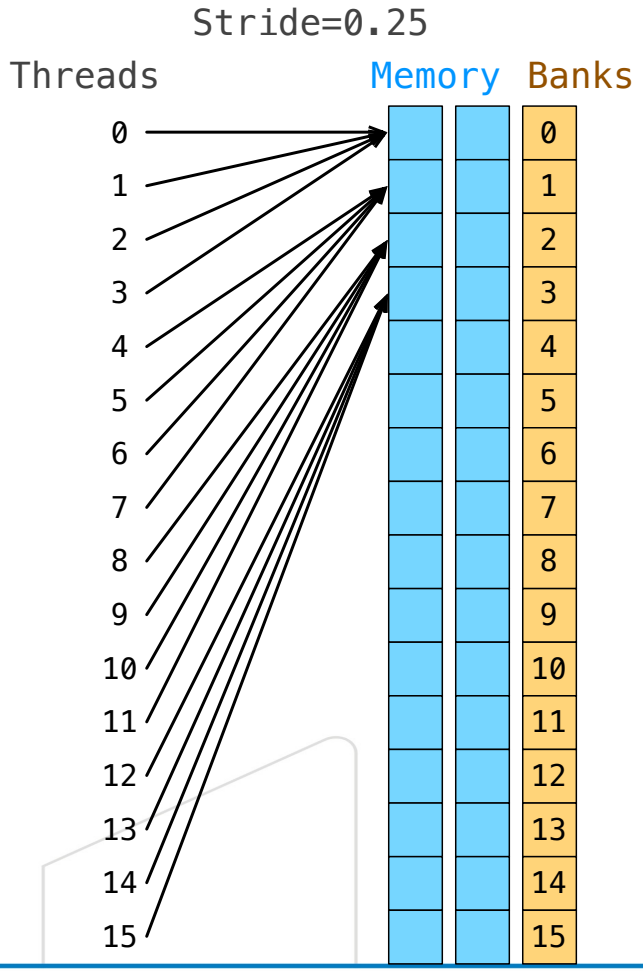
- 消除存储体冲突
  - 举例：
    - 步长？有无存储体冲突？

```
__shared__ char smem[BDIM];

smem[threadIdx.x] = some_value;

__syncthreads();
```

共享内存

- 消除存储体冲突
  - 举例：
    - 步长=0.25，有存储体冲突

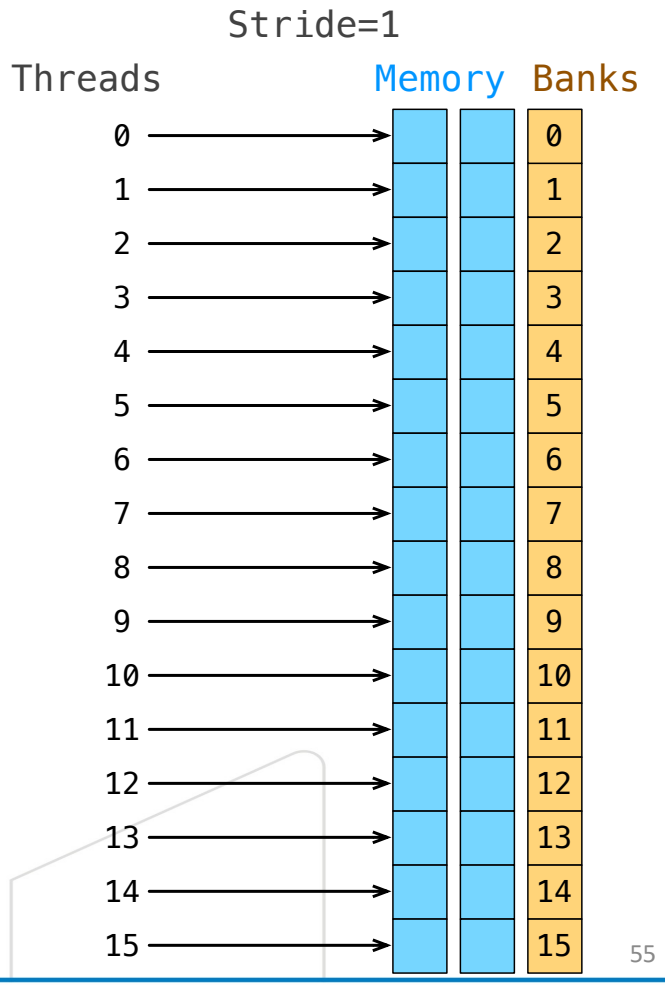
```
__shared__ char smem[BDIM];

smem[threadIdx.x] = some_value;

__syncthreads();
```



共享内存

- 消除存储体冲突
  - 举例：
    - 步长=0.25，有存储体冲突
    - 解决方案：增加步长

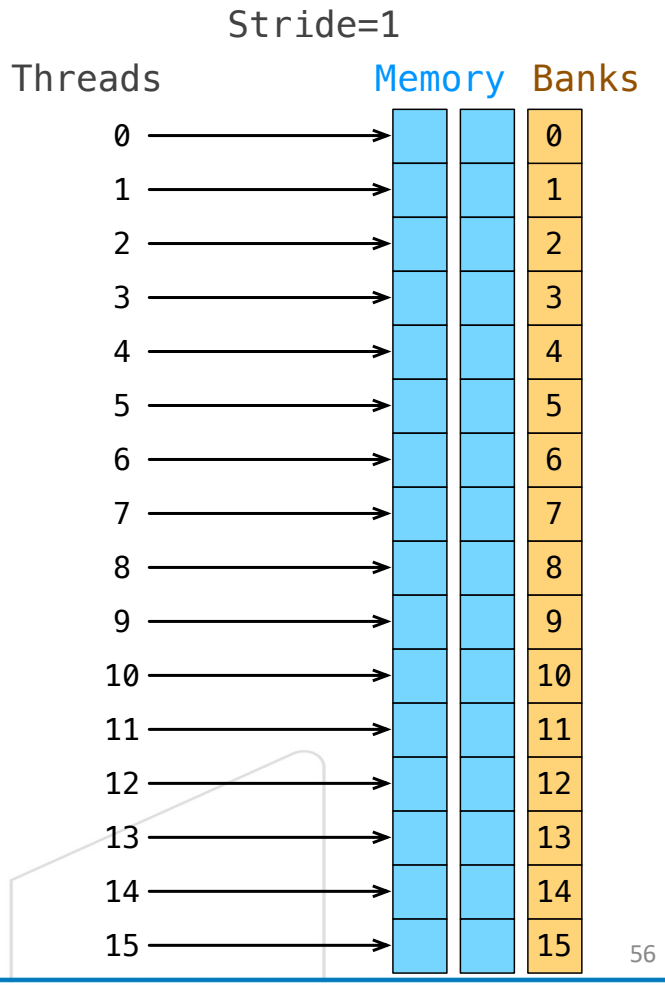
```
__shared__ char smem[BDIM*4];

smem[threadIdx.x*4] = some_value;

__syncthreads();
```



共享内存

- 消除存储体冲突
  - 举例：
    - 步长=0.25，有存储体冲突
    - 解决方案：增加步长
    - 问题：消耗内存增加

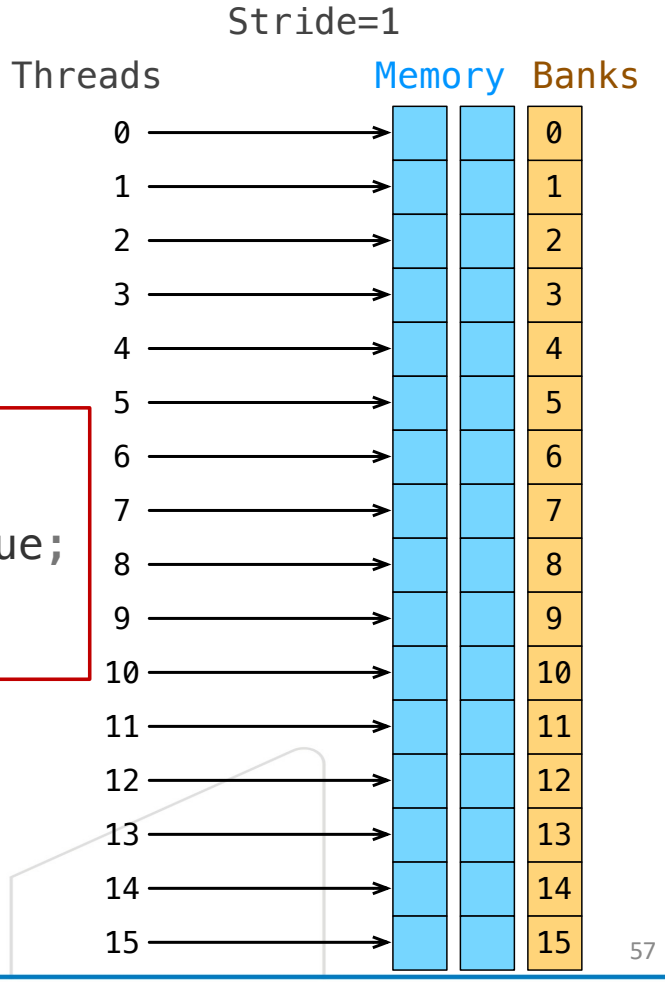
```
__shared__ char smem[BDIM*4];

smem[threadIdx.x*4] = some_value;

__syncthreads();
```



共享内存

- 消除存储体冲突
  - 举例：
    - 步长=1，无存储体冲突
    - 问题：消耗内存增加
    - 解决方案:设BDIM=64

```
__shared__ char smem[BDIM];

smem[threadIdx.x*4+threadIdx.x/16] = some_value;

__syncthreads();
```



小结

- 全局内存
  - 动态、静态全局内存、统一内存寻址
  - 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
  - 片上只读缓存、适合分散读取
- 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

Questions?