

CUDA内存管理与同步

汤善江 副教授

天津大学智能与计算学部

tashj@tju.edu.cn

<http://cic.tju.edu.cn/faculty/tangshanjiang/>

Outline

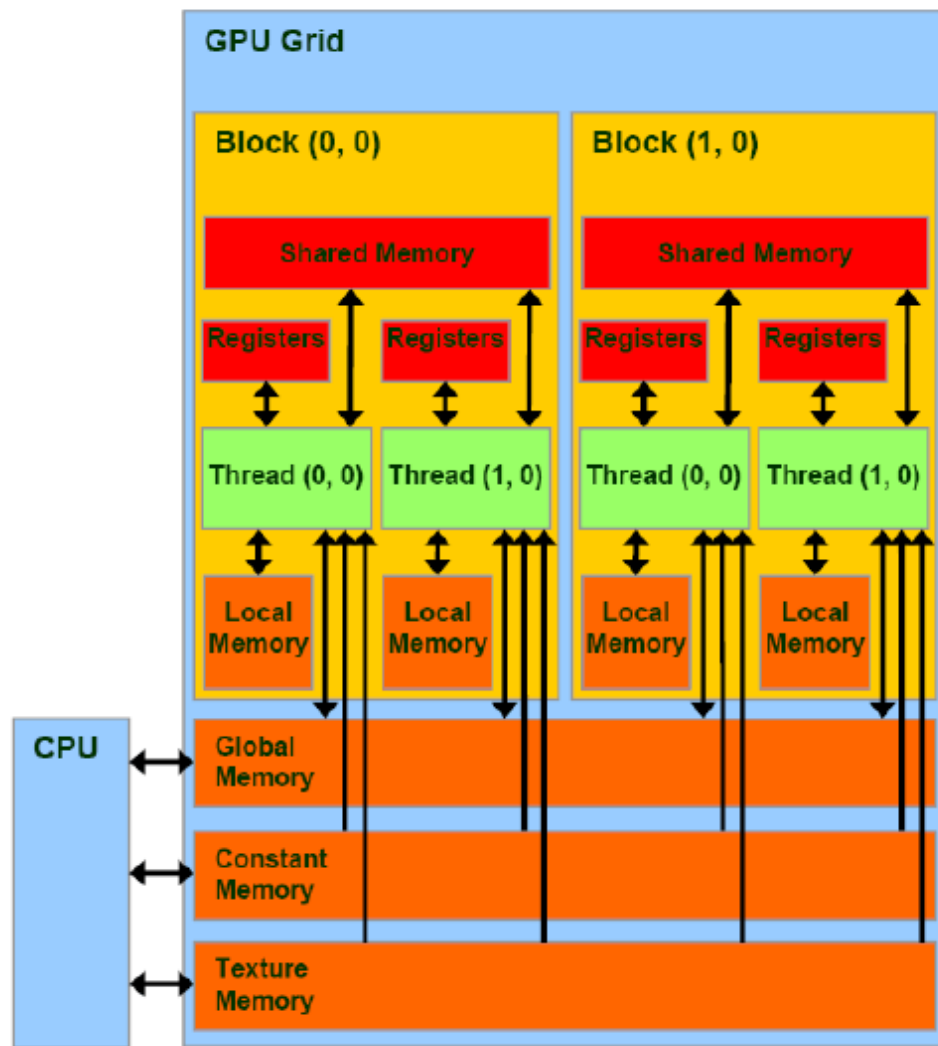
- CUDA内存管理
 - CUDA内存模型
 - CUDA变量存储
 - CUDA内存分配
- CUDA线程同步
 - CUDA线程同步
 - CUDA原子操作

Outline

- **CUDA内存管理**
 - **CUDA内存模型**
 - CUDA变量存储
 - CUDA内存分配
- **CUDA线程同步**
 - CUDA线程同步
 - CUDA原子操作

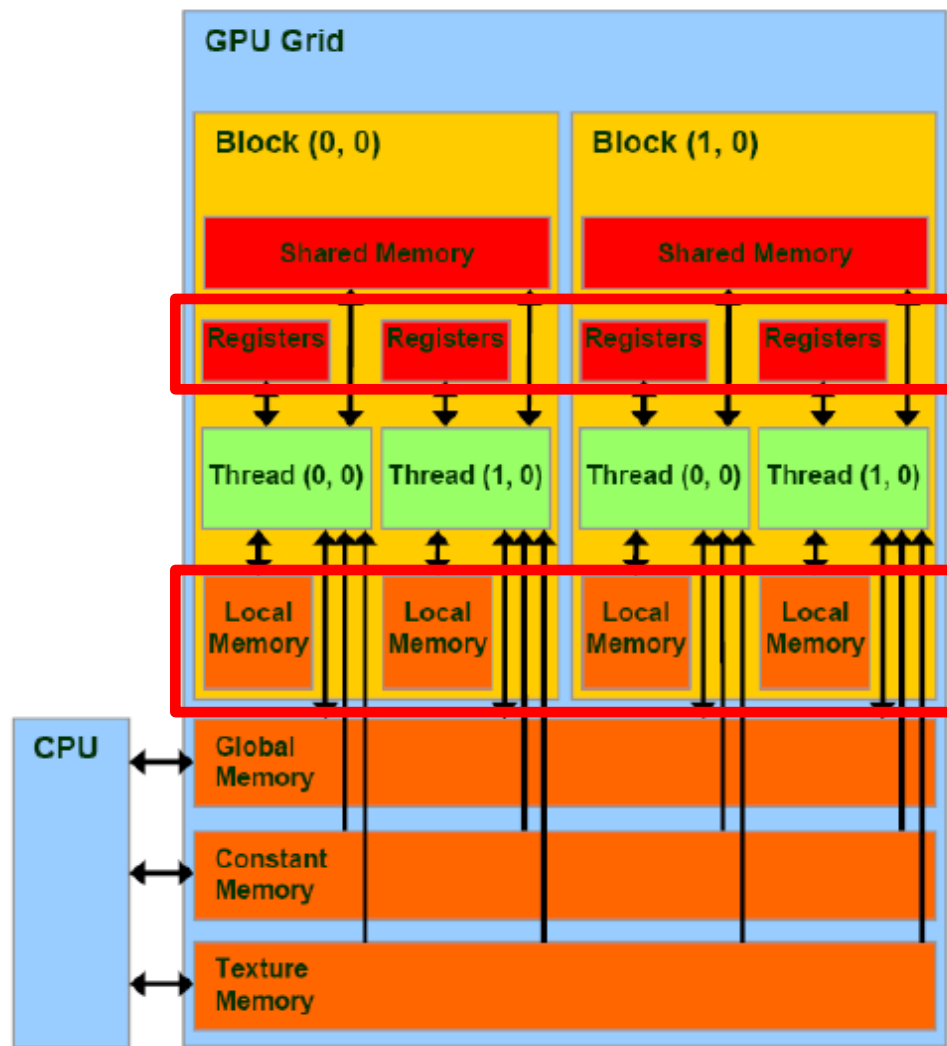
存储器模型

- Register
- Local
- shared
- Global
- Constant
- Texture
- Host memory
- Pinned host memory



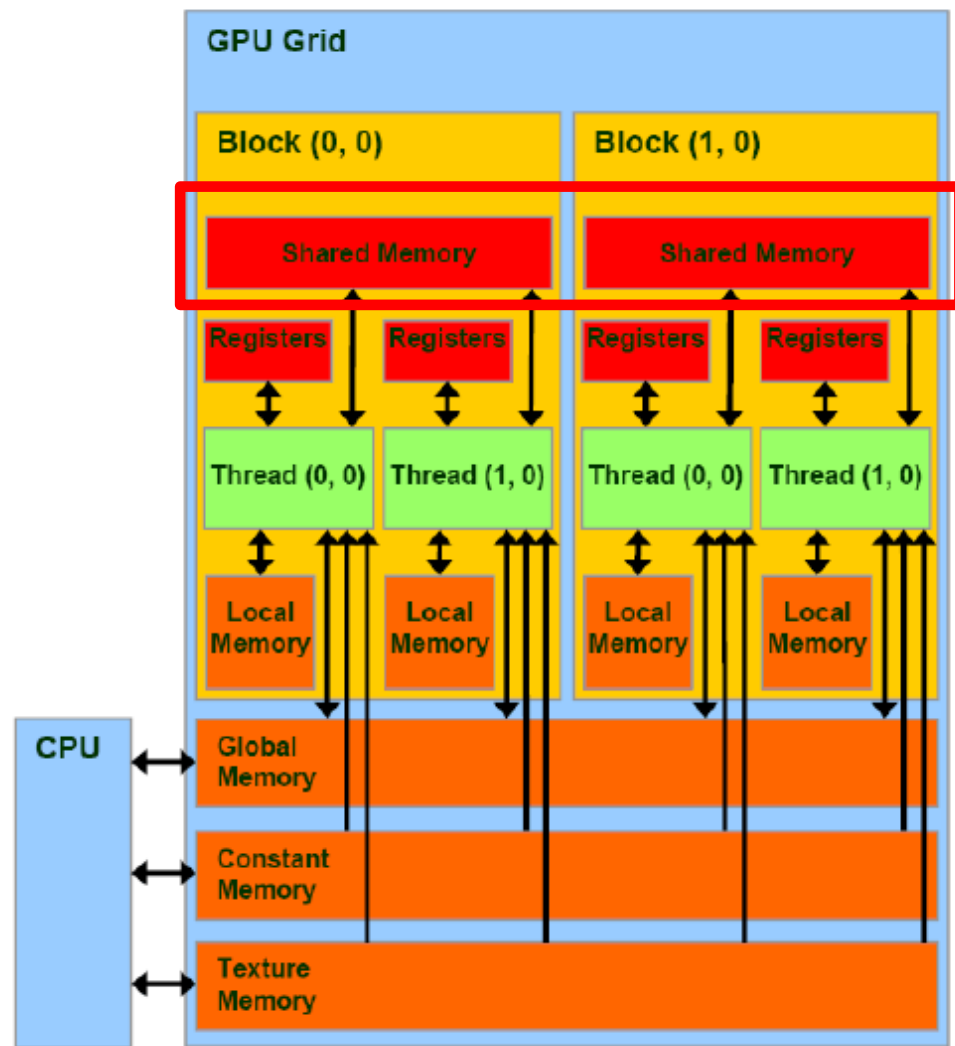
寄存器与local memory

- 对每个线程来说，寄存器都是线程私有的--这与CPU中一样。如果寄存器被消耗完，数据将被存储在本地存储器(local memory)。Local memory对每个线程也是私有的，但是local memory中的数据是被保存在显存中，而不是片内的寄存器或者缓存中，速度很慢。线程的输入和中间输出变量将被保存在寄存器或者本地存储器中。



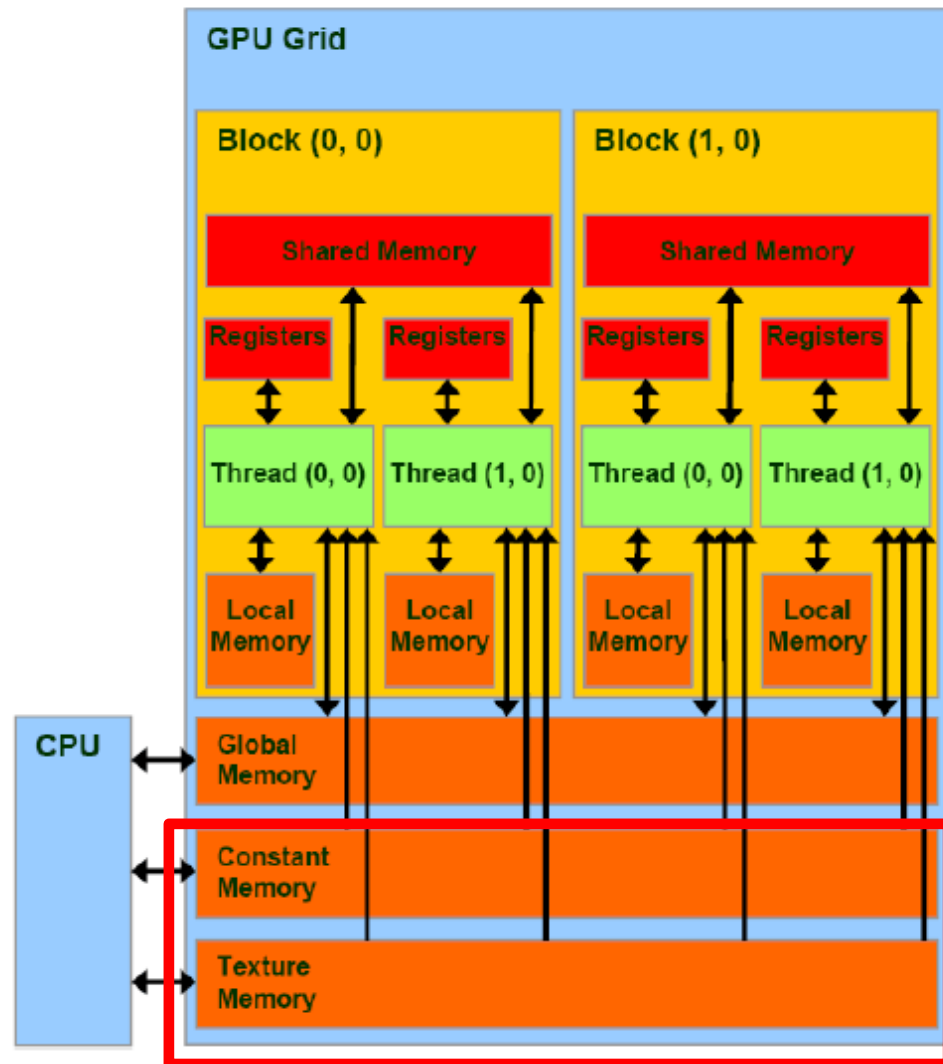
Shared memory

- 用于线程间通信的共享存储器。共享存储器是一块可以被同一block中的所有thread访问的可读写存储器
- 访问共享存储器几乎和访问寄存器一样快，是实现线程间通信的延迟最小的方法。
- 共享存储器可以实现许多不同的功能，如用于保存共用的计数器(例如计算循环次数)或者block的公用结果(例如计算512个数的平均值，并用于以后的计算)。



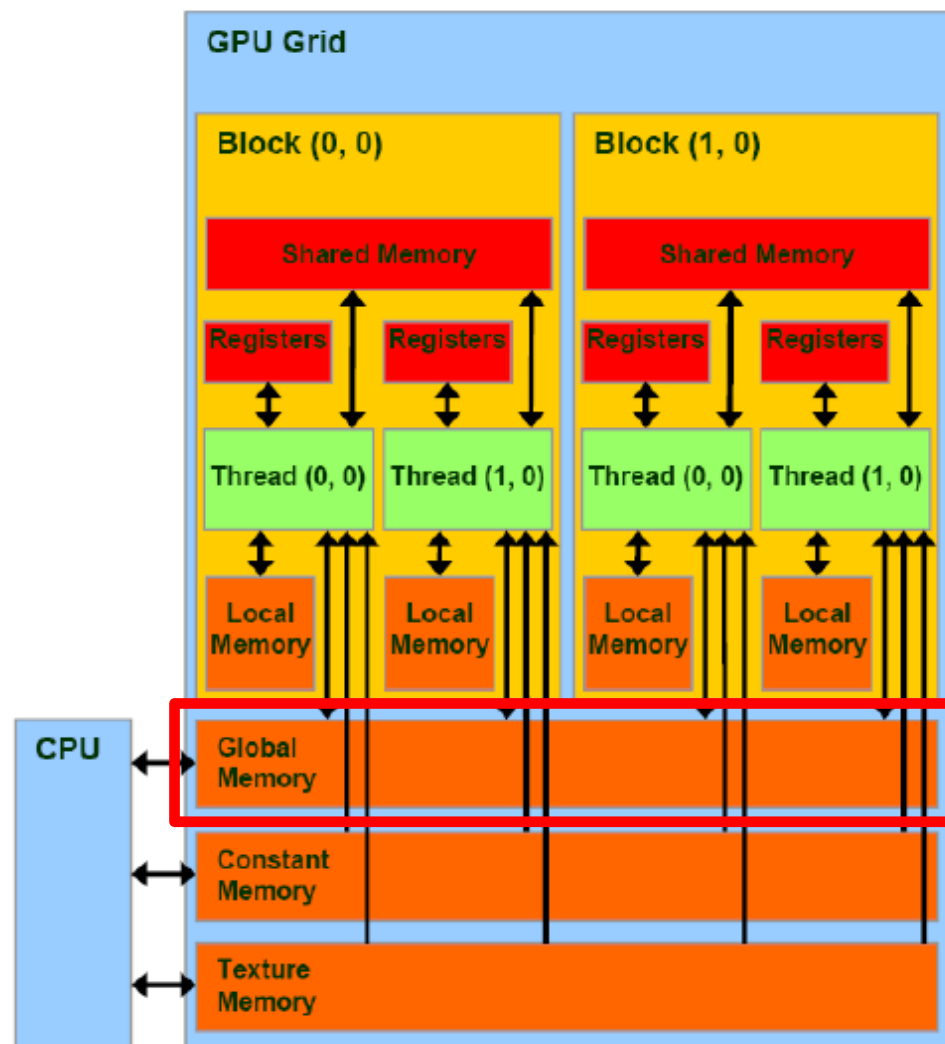
constant memory, texture memory

- 利用GPU用于图形计算的专用单元发展而来的高速只读缓存
- 速度与命中率有关，不命中时将进行对显存的访问
- 常数存储器空间较小(只有64k)，支持随机访问。从host端只写，从device端只读
- 纹理存储器尺寸则大得多，并且支持二维寻址。(一个数据的“上下左右”的数据都能被读入缓存) 适合实现图像处理算法和查找表



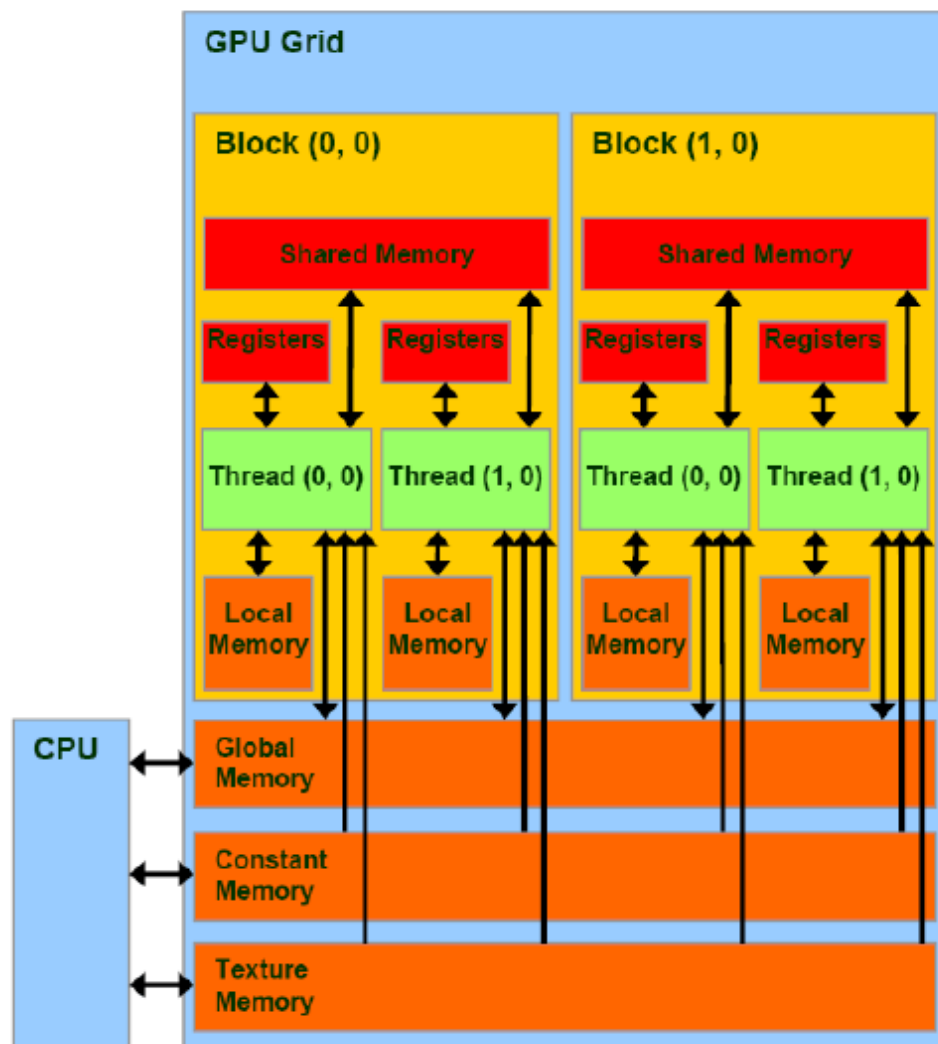
全局存储器(Global Memory)

- 使用的是普通的显存，无缓存，可读写，速度慢
- 整个网格中的任意线程都能读写全局存储器的任意位置，并且既可以从CPU访问，也可以从GPU访问。



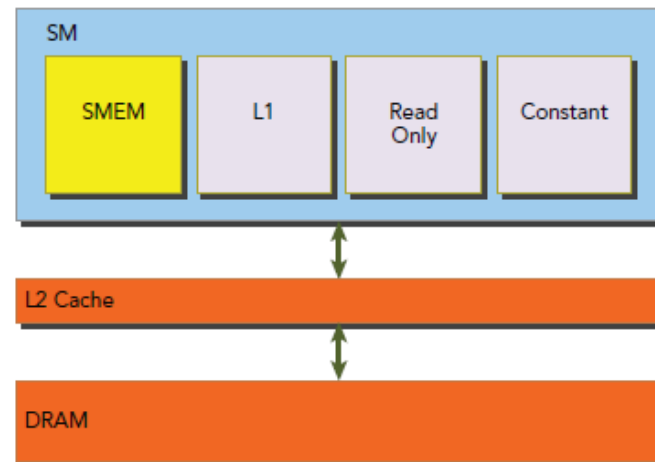
各种存储器的大小

- 每个SM中有64K (GT200) 或者32K(G8x, G9x)寄存器, 寄存器的最小单位是32bit的register file
- 每个SM中有16K shared memory
- 一共可以声明64K的constant memory, 但每个SM的cache序列只有8K
- 可以声明很大的texture memory, 但是实际上的texture cache序列为每SM 6-8K

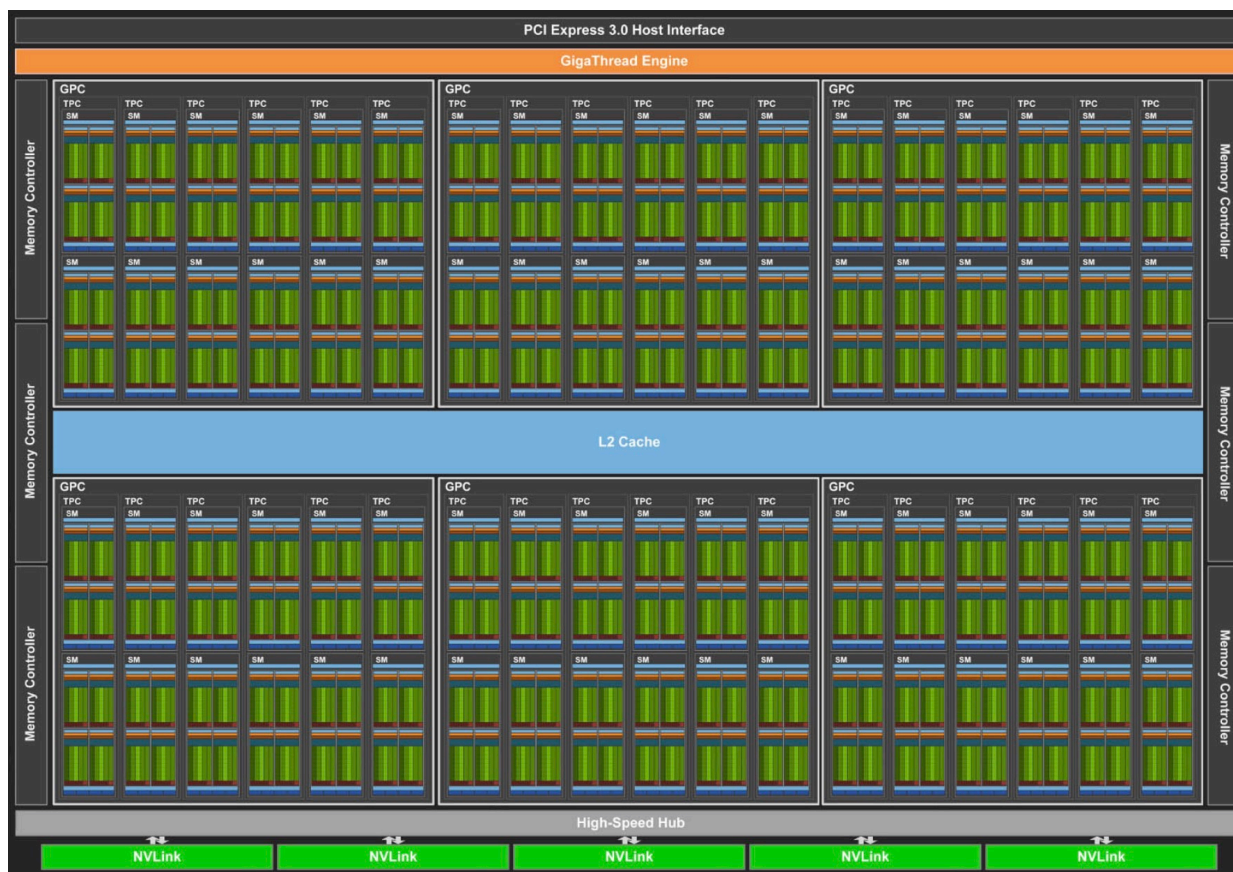


Tesla GPU 内存大小

| Tesla Product | Tesla K40 | Tesla M40 | Tesla P100 | Tesla V100 |
|---------------------------------|---------------------|---------------------|---------------------|--------------------------|
| GPU | GK180 (Kepler) | GM200 (Maxwell) | GP100 (Pascal) | GV100 (Volta) |
| SMs | 15 | 24 | 56 | 80 |
| TPCs | 15 | 24 | 28 | 40 |
| FP32 Cores / SM | 192 | 128 | 64 | 64 |
| FP32 Cores / GPU | 2880 | 3072 | 3584 | 5120 |
| FP64 Cores / SM | 64 | 4 | 32 | 32 |
| FP64 Cores / GPU | 960 | 96 | 1792 | 2560 |
| Tensor Cores / SM | NA | NA | NA | 8 |
| Tensor Cores / GPU | NA | NA | NA | 640 |
| GPU Boost Clock | 810/875 MHz | 1114 MHz | 1480 MHz | 1530 MHz |
| Peak FP32 TFLOPS ¹ | 5 | 6.8 | 10.6 | 15.7 |
| Peak FP64 TFLOPS ¹ | 1.7 | .21 | 5.3 | 7.8 |
| Peak Tensor TFLOPS ¹ | NA | NA | NA | 125 |
| Texture Units | 240 | 192 | 224 | 320 |
| Memory Interface | 384-bit GDDR5 | 384-bit GDDR5 | 4096-bit HBM2 | 4096-bit HBM2 |
| Memory Size | Up to 12 GB | Up to 24 GB | 16 GB | 16 GB |
| L2 Cache Size | 1536 KB | 3072 KB | 4096 KB | 6144 KB |
| Shared Memory Size / SM | 16 KB/32 KB/48 KB | 96 KB | 64 KB | Configurable up to 96 KB |
| Register File Size / SM | 256 KB | 256 KB | 256 KB | 256KB |
| Register File Size / GPU | 3840 KB | 6144 KB | 14336 KB | 20480 KB |
| TDP | 235 Watts | 250 Watts | 300 Watts | 300 Watts |
| Transistors | 7.1 billion | 8 billion | 15.3 billion | 21.1 billion |
| GPU Die Size | 551 mm ² | 601 mm ² | 610 mm ² | 815 mm ² |
| Manufacturing Process | 28 nm | 28 nm | 16 nm FinFET+ | 12 nm FFN |



Tesla GPU 内存大小

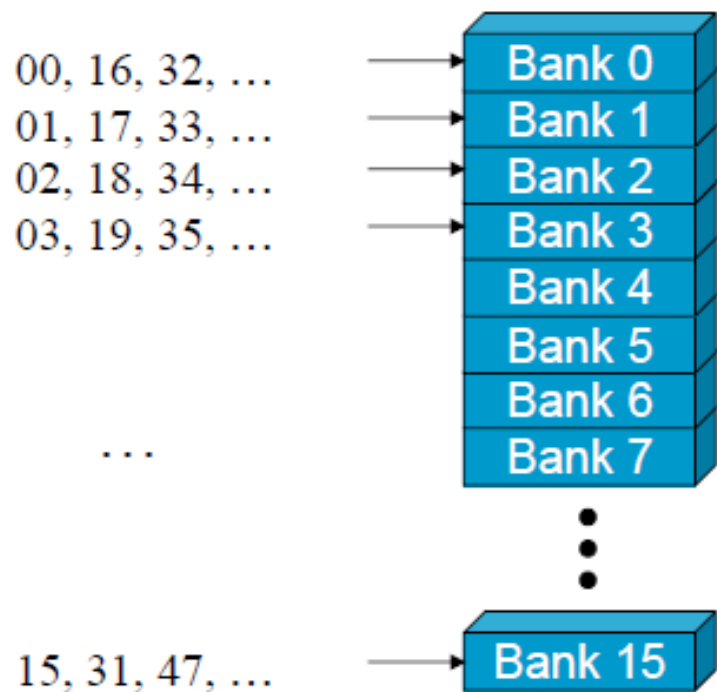


使用存储器时可能出现的问题

- 致命问题：无法产生正确结果
 - 多个block访问global同一块，以及block内thread间线程通信时的数据一致性问题
- 效率问题：大大增加访存延迟
 - Shared bank conflict问题
 - Global 合并访问问题

Shared Memory Bank

- 为了获得高带宽，shared memory 被分为了16（或32）个等大小内存块(banks)，单位是32-bit。
- 相邻数据在不同bank中，对16（或32）余数相同的数据在同一bank
- 如果warp访问shared memory，对于每个bank只访问不多于一个内存地址，那么只需要一次内存传输就可以了，否则需要多次传输，因此会降低内存带宽的使用。



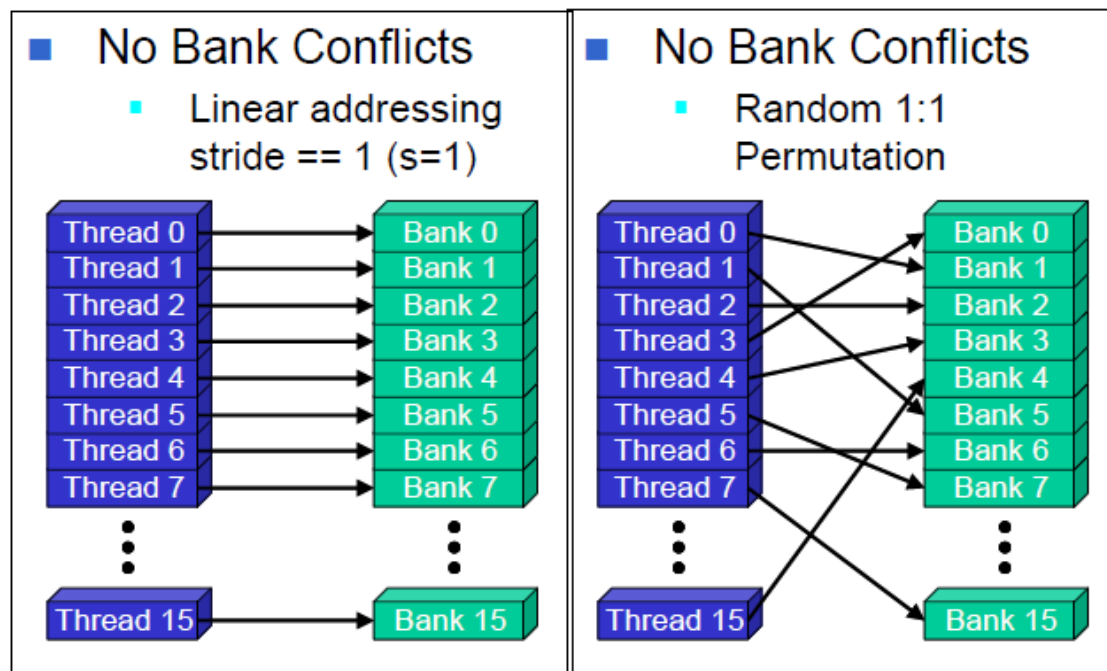
<http://blog.csdn.net/yu132563>

Shared Memory访问模式

- warp有三种典型的获取shared memory的模式：
 - Parallel access: 最通常的模式，多个地址分散在多个bank。这个模式一般暗示，一些（也可能是全部）地址请求能够被一次传输解决。理想情况是，获取无conflict的shared memory的时，每个地址都在落在不同的bank中。
 - Serial access: 最坏的模式，多个地址落在同一个bank。如果warp中的32个thread都访问了同一个bank中的不同位置，那就是32次单独的请求，而不是同时访问了。
 - Broadcast access: 一个地址读操作落在一个bank。也是只执行一次传输，然后传输结果会广播给所有发出请求的thread。这样的话就会导致带宽利用率低

Bank conflict

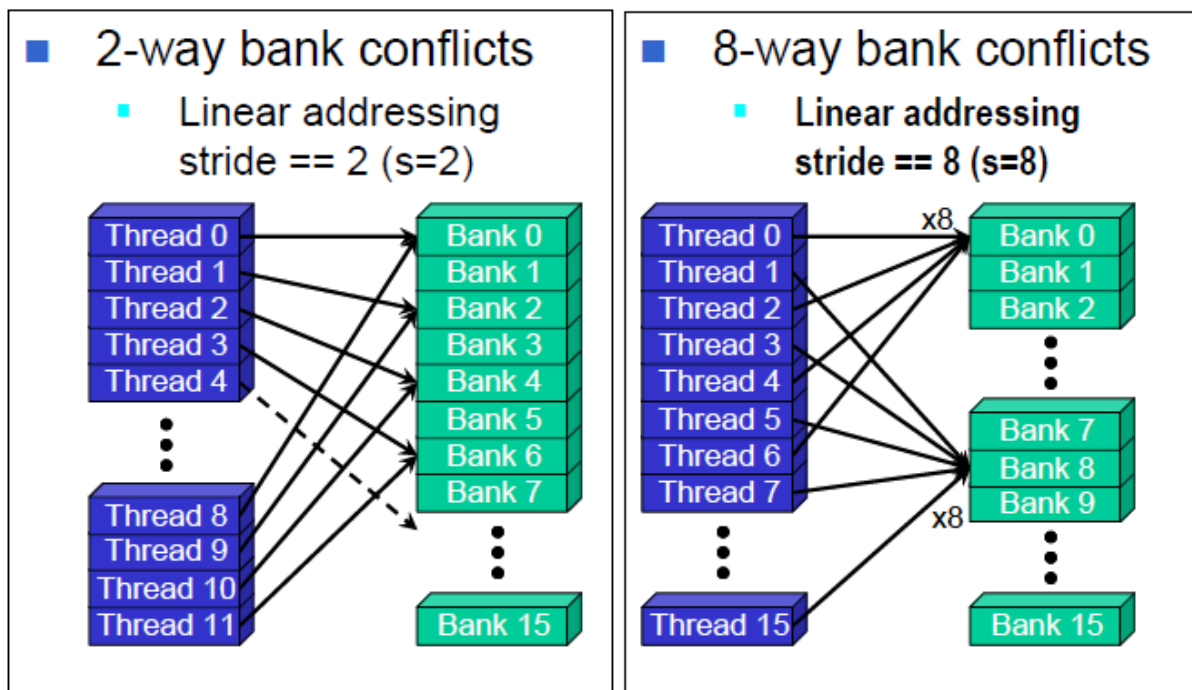
- 以G80为例，其share memory具有16个bank，在下面的图示中，只有S为奇数时，才不会存在bank conflicts



```
__shared__ float shared[256];  
float foo = shared[base + s * threadIdx.x];
```

Bank conflict

- 以G80为例，其share memory具有16个bank，在下面的图示中，只有S为奇数时，才不会存在bank conflicts



合并访问

- 数据从全局内存到SM (stream-multiprocessor) 的传输，会进行cache，如果cache命中了，下一次的访问的耗时将大大减少。
- 每个SM都具有单独的L1 cache，所有的SM共用一个L2 cache。对于L1 cache，每次按照128字节进行缓存；对于L2 cache，每次按照32字节进行缓存。
- 对于L1 cache，内存块大小支持32字节、64字节以及128字节，分别表示线程束中每个线程以一个字节 ($1*32=32$)、16位 ($2*32=64$)、32位 ($4*32=128$) 为单位读取数据。前提是，访问必须连续，并且访问的地址是以32字节对齐。

合并访问

- 例子：假设每个thread读取一个float变量，那么一个warp（32个thread）将会执行 $32 \times 4 = 128$ 字节的合并访存指令，通过一次访存操作完成所有thread的读取请求

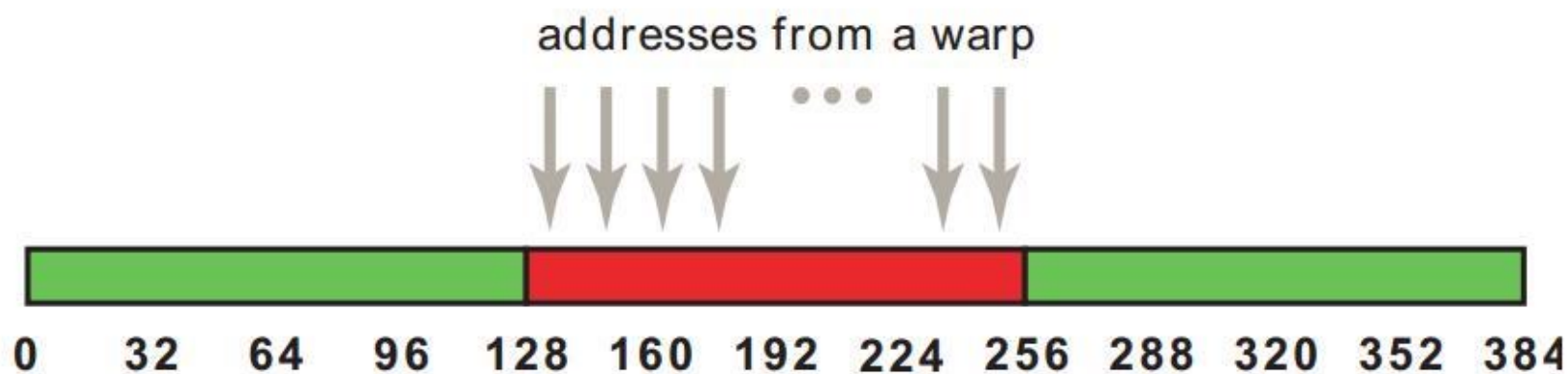
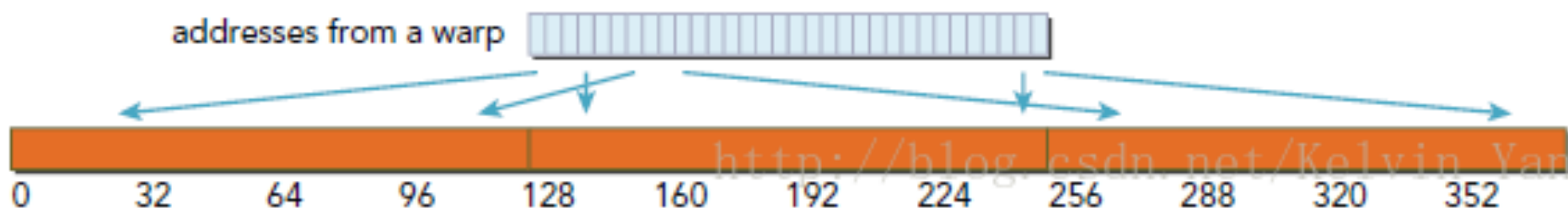


Figure 3 Coalesced access - all threads access one cache line Alvin_Yan

分散访问

- 例子：warp请求访问位于不同地址的数据，数据是非连续的，此时warp无法进行合并访问，每个thread访问一个float，一共需要执行32次访存指令



Outline

- **CUDA内存管理**

- CUDA内存模型
- **CUDA变量存储**
- CUDA内存分配

- **CUDA线程同步**

- CUDA线程同步
- CUDA原子操作

CUDA C语言

- 由Nvidia的CUDA编译器(nvcc)编译
- CUDA C不是C语言，而是对C语言进行扩展形成的变种。

CUDA对C的扩展：函数限定符

- 对函数有了限定符，用来规定函数是在host还是在device上执行，以及这个函数是从host调用还是从device调用。这些限定符是：__device__，__host__和__global__。

CUDA对C的扩展：函数限定符

- `__device__` 函数在device端执行，并且也只能从device端调用，即作为device端的子函数来使用
- `__global__` 函数即kernel函数，它在设备上执行，但是要从host端调用
- `__host__` 函数在host端执行，也只能从host端调用，与一般的C函数相同

CUDA对C的扩展：变量限定符

- 对变量类型的限定符，用来规定变量被存储在哪一种存储器上。
- 传统的在CPU上运行的程序中，编译器就能自动决定将变量存储在CPU的寄存器还是在计算机的内存中。
- 而在CUDA中，不仅要使用host端的内存，而且也要使用显卡上的显存和GPU上的几种寄存器和缓存。在CUDA编程模型中，一共抽象出来了多达8种不同的存储器！

CUDA对C的扩展：变量限定符

- `__device__`
- `__device__` 限定符声明的变量存在于device端，其他的变量限定符声明的变量虽然存在于不同的存储器里，但总体来说也都在device端。所以`__device__`限定符可以与其他限定符联用。当单独使用`__device__`限定符修饰变量时，这个变量存在于global memory中；
- 变量生命周期与整个程序一样长；
- 可以被grid中所有的线程都可以访问，也可以从host端通过运行时库中的函数访问。

CUDA对C的扩展：变量限定符

- `__constant__`
- `__constant__` 限定符，可以与 `__device__` 联用，即 `__device__ __constant__`，此时等同于单独使用 `__constant__`。使用 `__constant__` 限定符修饰的变量：
 - 存在于 constant memory 中，访问时速度一般比使用 global memory 略快；
 - 变量生命周期与整个程序一样长；
 - 可以被 grid 中所有的线程读，从 host 端通过运行时库中的函数写。

CUDA对C的扩展：变量限定符

- `__shared__`
- `__shared__` 限定符，可以与 `__device__` 联用，即 `__device__ __shared__`，此时等同于单独使用 `__shared__`。使用 `__shared__` 限定符修饰的变量：
 - 存在于block中的shared memory中；
 - 变量生命周期与block相同；
 - 只有同一block内的thread才能访问。

CUDA对C的扩展：kernel执行参数

- <<< >>>运算符，用来传递一些kernel执行参数
- Grid的大小和维度
- Block的大小和维度
- 外部声明的shared memory大小
- stream编号

CUDA对C的扩展：内建变量

- Dim3 ThreadIdx (三维)
- Dim3 ThreadDim (三维)
- Dim3 BlockIdx(二维)
- Dim3 BlockDim (二维)

执行参数与内建变量的作用

- 各个thread和block之间的唯一不同就是threadID和BlockID, 通过内建变量控制各个线程处理的指令和数据
- CPU运行核函数时的执行参数确定GPU在SPA上分配多少个block, 在SM上分配多少个thread

CUDA 中函数与变量类型汇总

函数类型

__device__

- 在GPU中执行
- 自GPU调用

__global__

- 在GPU中执行
- 自CPU调用

__host__

- 在CPU中执行/调用
- 默认函数类型

变量类型

__device__

- global memory space
- grid中所有线程可访问

__constant__

- constant memory space
- grid中所有线程可访问

__shared__

- space of a thread block
- 只能由block中的线程访问

Outline

- **CUDA内存管理**
 - CUDA内存模型
 - CUDA变量存储
 - **CUDA内存分配**
- CUDA线程共享与同步
 - CUDA线程同步
 - CUDA原子操作

内存分配

- 内存分为两种
 - 可分页内存 (Pageable Memory)
 - 标准C/C++函数操作, `malloc()` / `new()`
 - 分页锁定内存 (page-locked Memory)
 - `cudaMallocHost(void** ptr, size_t size)`
 - `cudaHostAlloc(void** pHost, size_t size, unsigned int flags)`
 - `cudaHostAllocPortable`: 多个 CPU 线程都可访问
 - `cudaHostAllocWriteCombined`: 数据传输快, 但只允许host 写 (读速度慢)
 - `cudaHostAllocMapped`: CPU/GPU 都可访问
 - 支持 DMA
 - 可分页内存转为分页锁定内存
 - `cudaHostRegister (void* ptr, size_t size, unsigned int flags)`

显存分配

- `cudaMalloc()`
 - 一维数组，数据连续存储
- `cudaMallocPitch()`
 - 二维数组，会自动对齐，可能会浪费部分内存空间
- `cudaMalloc3D()`
 - 三维数组

Host - Device 数据交换

- `cudaMemcpy()`

- 在存储器直接传输数据

- 四个参数

- 目的对象数组指针

- 源对象数组指针

- 数组尺寸

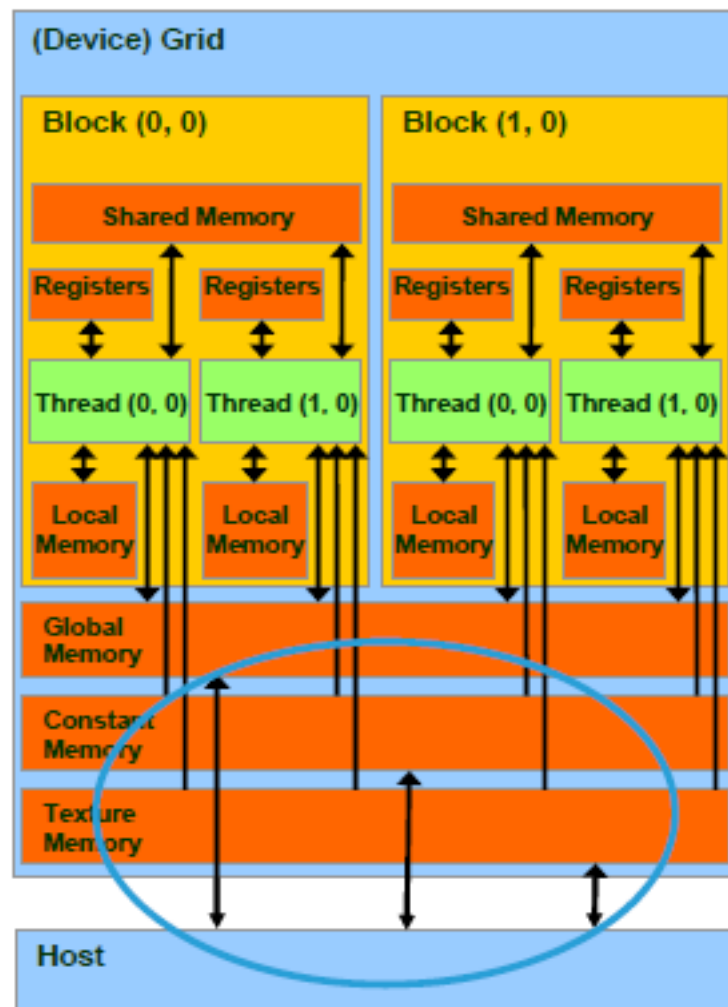
- 传输方向

- Host到Host

- Host到Device

- Device到Host

- Device到Device



Host - Device 数据交换

- 代码实例
 - M.elements: CPU 主存
 - Md: GPU 显存
 - 符号常数: cudaMemcpyHostToDevice 和 cudaMemcpyDeviceToHost

```
cudaMemcpy(Md, M.elements, size, cudaMemcpyHostToDevice);  
cudaMemcpy(M.elements, Md, size, cudaMemcpyDeviceToHost);
```

数据拷贝

```
int main() {  
    int N = 256 * 1024;  
    float* h_a = malloc(sizeof(float) * N);  
    //Similarly for h_b, h_c. Initialize h_a, h_b  
  
    float *d_a, *d_b, *d_c;  
    cudaMalloc(&d_a, sizeof(float) * N);  
    //Similarly for d_b, d_c  
  
    cudaMemcpy(d_a, h_a, sizeof(float) * N, cudaMemcpyHostToDevice);  
    //Similarly for d_b  
  
    //Run N/256 blocks of 256 threads each  
    vecAdd<<<N/256, 256>>>(d_a, d_b, d_c);  
  
    cudaMemcpy(h_c, d_c, sizeof(float) * N, cudaMemcpyDeviceToHost);  
}
```

内存映射

- ZeroCopy

- host内存直接映射到GPU 内存空间
- 调用`cudaHostAlloc()` 分配时传入`cudaHostAllocMapped` 标签
- 使用`cudaHostRegister()` 注册时使用`cudaHostRegisterMapped` 标签
- device 可直接访问分页锁定内存，硬件在 kernel 访问显存时通过PCIE 总线传递，由硬件自动处理数据传输和计算的异步执行。

- 适用于计算密集型程序

- 数据访问量小
- 计算与数据传输重叠
- 不需要显式内存管理

统一寻址 (Unified Memory)

- Managed memory (CUDA6.0起开始支持)
 - 在 host 内存与device显存之间根据访问需要自动迁移数据，同时保证 host 和 device 都可访问，应用程序并不需要知道访问时数据所在位置
 - 需要进行显式同步（以保证前一步骤中数据更新操作全部完成）
- 优势
 - 编程简化
 - 数据访问性能提升
- 适用条件
 - SM 体系结构3.0以上（Kepler 架构，或更新的架构）
 - 64位进程，非嵌入式系统

统一寻址：host 与 device 使用同一指针

- 定义方式：cudaMallocManaged

```
__global__ void AplusB(int *ret, int a, int b) {  
    ret[threadIdx.x] = a + b + threadIdx.x;  
}  
  
int main() {  
    int *ret;  
    cudaMallocManaged(&ret, 1000 * sizeof(int));  
    AplusB<<< 1, 1000 >>>(ret, 10, 100);  
    cudaDeviceSynchronize(); //同步，等待所有kernel完成  
    for(int i=0; i<1000; i++)  
        printf("%d: A+B = %d\n", i, ret[i]);  
    cudaFree(ret);  
    return 0;  
}
```


统一寻址（全局变量）

```
__device__ __managed__ int ret[1000];  
__global__ void AplusB(int a, int b) {  
    ret[threadIdx.x] = a + b + threadIdx.x;  
}  
  
int main() {  
    AplusB<<< 1, 1000 >>>(10, 100);  
    cudaDeviceSynchronize(); //同步  
    for(int i=0; i<1000; i++)  
        printf("%d: A+B = %d\n", i, ret[i]);  
    return 0;  
}
```

Outline

- CUDA内存管理
 - CUDA内存模型
 - CUDA变量存储
 - CUDA内存分配
- **CUDA线程共享与同步**
 - **CUDA线程同步**
 - CUDA原子操作

Synchronization

- 为保证计算结果的正确性，有些计算过程需要对并发运行的线程进行同步
 - 如规约操作（e.g., sum, max），数据更新操作
 - CUDA中每一组Warp的32线程都是默认显示同步的
 - 在共享内存里头，CUDA还提供了__syncthreads()来保证一个block内的线程的同步。
- 在条件代码同步中，一定要确保所有线程都能达到__syncthreads()，否则就会出现死锁。

```
if(radius <= 10.0) {  
    area = PI * radius * radius;  
    __syncthreads();  
} else {  
    area = 0.0;  
}
```



Synchronization

- CUDA同步程序代码样例

```
// load in data to shared memory
```

```
...
```

```
...
```

```
...
```

```
// synchronisation to ensure this has finished  
__syncthreads();
```

```
// now do computation using shared data
```

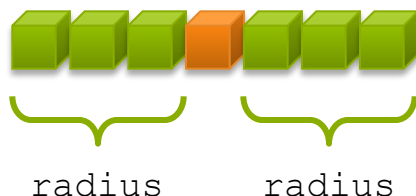
```
...
```

```
...
```

```
...
```

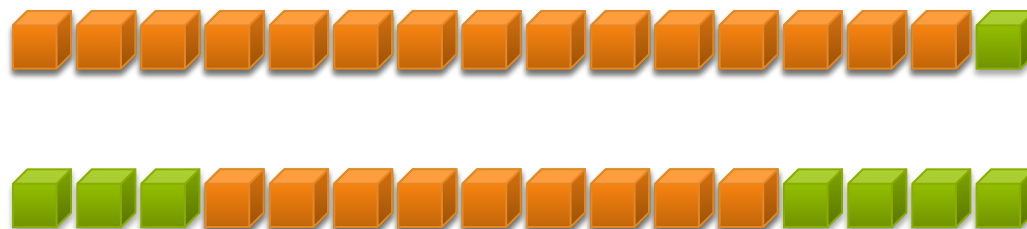
示例：1D Stencil

- 对1D数组进行1D stencil 处理（卷积）
 - 每个输出元素是输入元素一定“半径”内的数据和
- 若半径为3，则每个输出元素是7个输入元素的和：



在block内实现

- 每个thread处理一个输出元素
 - 每个block分配 blockDim.x 个元素（同thread数）
- 输入数据会被读取多次
 - 半径为3，每个输入元素被读取7次

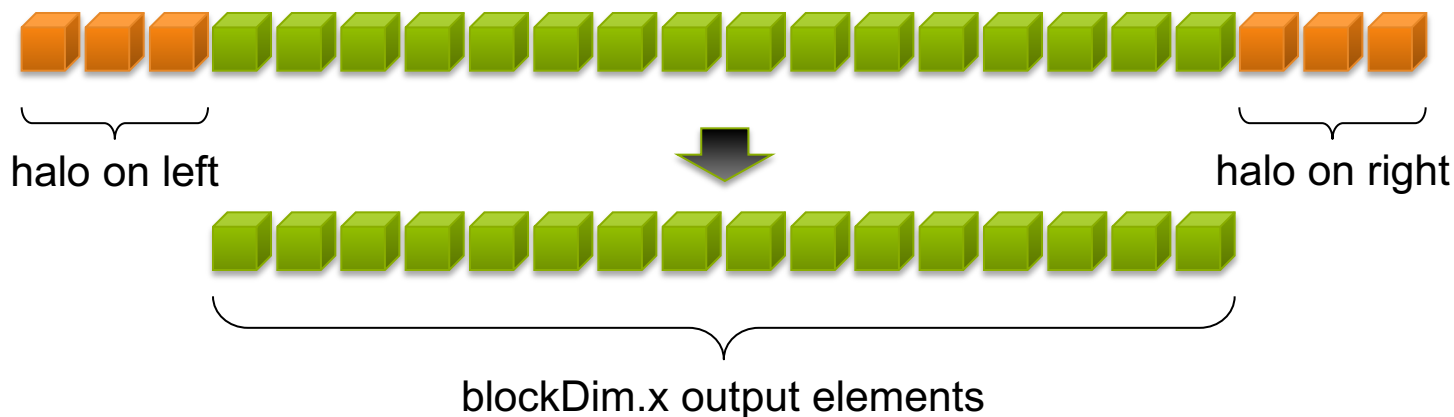


Threads之间共享数据

- 在block内部，threads 可以通过 **shared memory** 共享数据
- Shared memory特性：
 - 片内，高速，用户可管理
 - 使用 **__shared__** 声明，每个block单独分配
 - 其他block不可以读写

使用Shared Memory实现

- 在shared memory中缓存数据
 - 自global memory 读取 $(\text{blockDim.x} + 2 * \text{radius})$ 个输入元素到 shared memory
 - 计算 blockDim.x 个输出元素
 - 将 blockDim.x 个输出元素写回到global memory
 - 需要两个“半径”大小的 halo（边界数据）



Stencil Kernel

```
__global__ void stencil_1d(int *in, int *out) {
    __shared__ int temp[BLOCK_SIZE + 2 * RADIUS];
    int gindex = threadIdx.x + blockIdx.x * blockDim.x;
    int lindex = threadIdx.x + RADIUS;
```



// Read input elements into shared memory, 读数据

```
temp[lindex] = in[gindex];
```



```
if (threadIdx.x < RADIUS) {
```

```
    temp[lindex - RADIUS] = in[gindex - RADIUS];
```



```
    temp[lindex + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];
```

```
}
```



// Apply the stencil, 计算

```
int result = 0;
```

```
for (int offset = -RADIUS ; offset <= RADIUS ; offset++)
```

```
    result += temp[lindex + offset];
```


// Store the result, 存储结果


```
out[gindex] = result;
```

```
}
```

竞态条件

- 之前的 stencil 示例并不能保证结果正确
- 假设 thread 15 在 thread 0 完成数据缓存之前读取halo数据

```
temp[lindex] = in[gindex];           Store at temp[18] 
if (threadIdx.x < RADIUS) {
    temp[lindex - RADIUS] = in[gindex - RADIUS];      Skipped, threadIdx > RADIUS
    temp[lindex + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];
}

int result = 0;
for (int offset = -RADIUS ; offset <= RADIUS ; offset++)
    result += temp[lindex + 1];      Load from temp[19] 
```

__syncthreads()

- `void __syncthreads();`
- 同步同一个block之内的全部线程
 - 用于避免 RAW / WAR / WAW 风险
- 所有线程都必须都运行到barrier（路障）
 - 如果是条件代码，整个block中线程对应的条件必须一致

Stencil Kernel

```
__global__ void stencil_1d(int *in, int *out) {  
    __shared__ int temp[BLOCK_SIZE + 2 * RADIUS];  
    int gindex = threadIdx.x + blockIdx.x * blockDim.x;  
    int lindex = threadIdx.x + radius;  
  
    // Read input elements into shared memory, 读数据  
    temp[lindex] = in[gindex];  
    if (threadIdx.x < RADIUS) {  
        temp[lindex - RADIUS] = in[gindex - RADIUS];  
        temp[lindex + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];  
    }  
  
    // Synchronize (ensure all the data is available) 同步  
    __syncthreads();  
}
```

Stencil Kernel

```
// Apply the stencil  
int result = 0;  
for (int offset = -RADIUS ; offset <= RADIUS ; offset++)  
    result += temp[lindex + offset];  
  
// Store the result  
out[gindex] = result;  
}
```

Synchronization 函数

- 设备block内部同步函数

- `void __syncthreads();`
- `int __syncthreads_count(int predicate);`
- `int __syncthreads_and(int predicate);`
- `int __syncthreads_or(int predicate);`
- `void __syncwarp(unsigned mask=0xffffffff);`

- 设备block之间同步函数

- `__threadfence();`
- `__threadfence_block();`

- Host和Device之间同步

- `void cudaDeviceSynchronize();`
- 注意：这个函数用在host端

Outline

- CUDA内存管理
 - CUDA内存模型
 - CUDA变量存储
 - CUDA内存分配
- **CUDA线程共享与同步**
 - CUDA线程同步
 - **CUDA原子操作**

原子操作

- 当需要2个或者更多的线程更新同一个变量的时候，可采用原子操作进行更新同步
 - 保证任何时候只有一个线程进行变量值的更新操作
- CUDA提供了一系列原子操作
 - `type atomicAdd(type* address, type val);`
 - `type atomicSub(type* address, type val);`
 - `type atomicMin(type* address, type val);`
 - `type atomicMax(type* address, type val);`
 - `int atomicCAS(int * address, int compare, int val);`

例子：统计字符出现频率的直方图

```
__global__ void histo_kernel(unsigned char* buffer, long size, unsigned int* histo){
    __shared__ unsigned int temp[256];
    tmp[threadIdx.x] = 0;
    __syncThreads();

    int i = threadIdx.x + blockIdx.x * blockDim.x;
    int offset = blockDim.x * gridDim.x;
    while(i < size){

        atomicAdd( &temp[buffer[i]], 1);
        i += offset;
    }
    __syncthreads();
    atomicAdd( &(histo[threadIdx.x]), temp[threadIdx.x]);
}
```