CUDA编程模型----多种CUDA存储单元详解

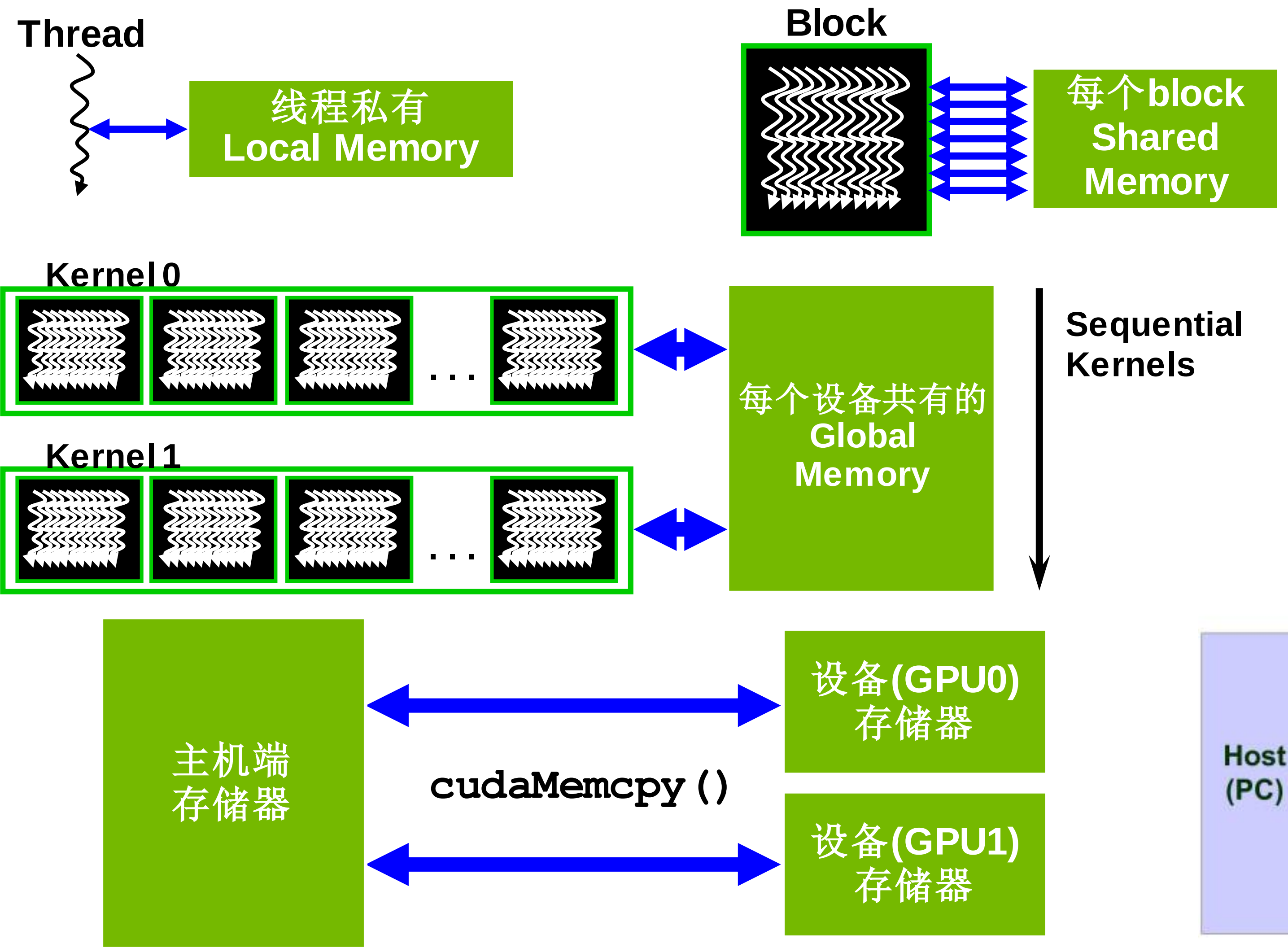
NVIDIA企业级开发者社区 何琨

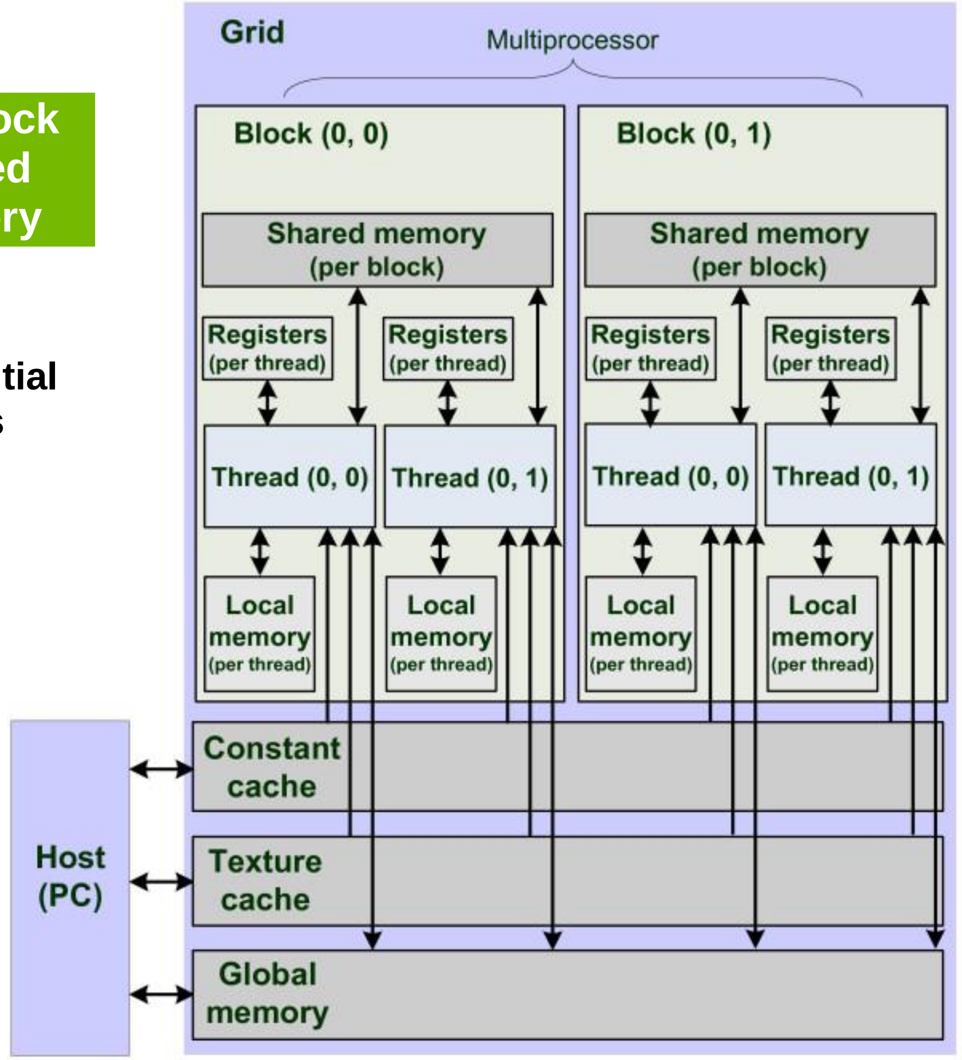
多种CUDA存储单元详解

AGENDA

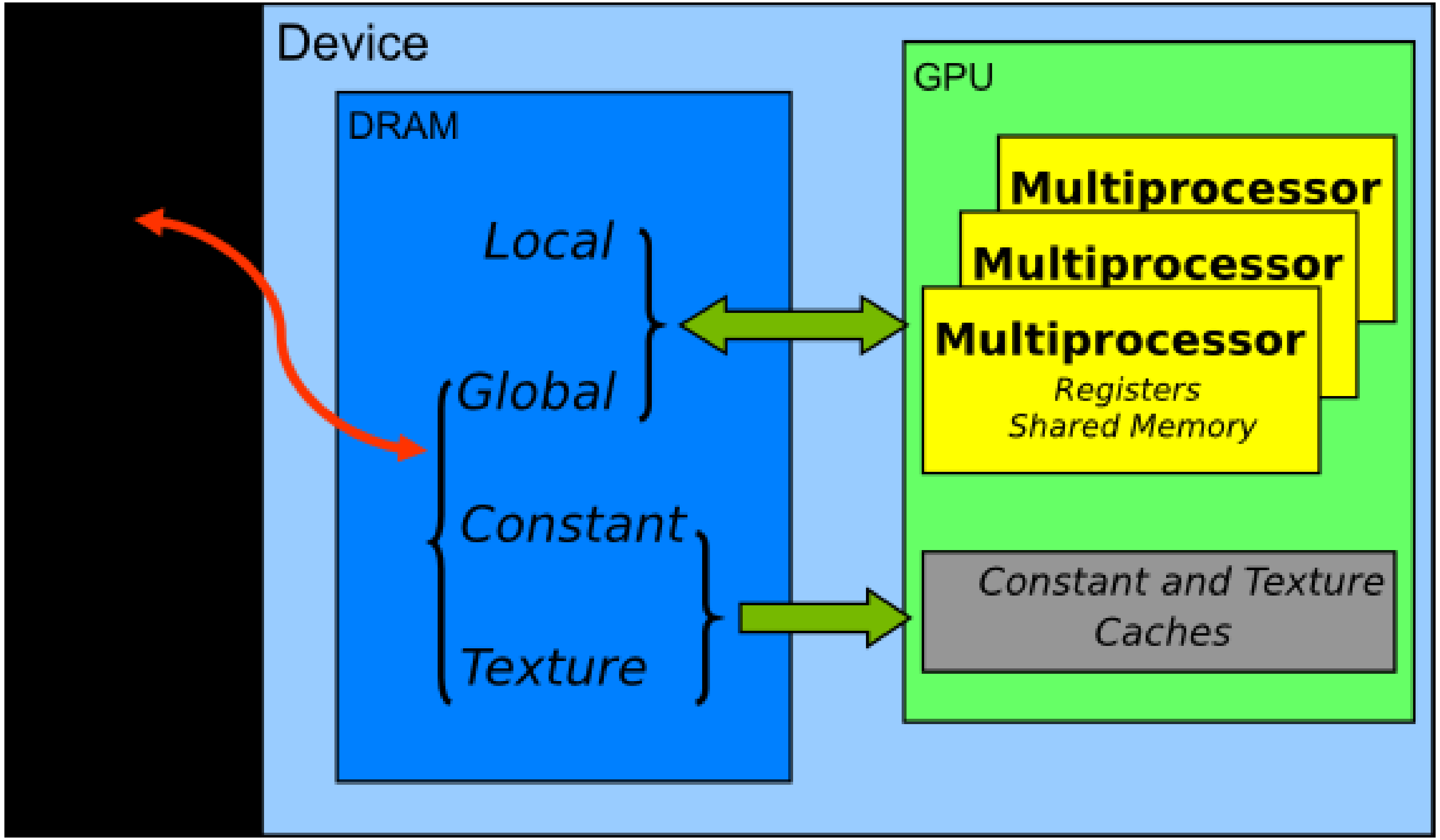
- CUDA中的存储单元种类
- CUDA中的各种存储单元的使用方法
- CUDA中的各种存储单元的适用条件

多种CUDA存储单元详解





多种CUDA存储单元详解



多种CUDA存储单元详解

Registers:

寄存器是GPU最快的memory，kernel中没有什么特殊声明的自动变量都是放在寄存器中的。当数组的索引是constant类型且在编译期能被确定的话，就是内置类型，数组也是放在寄存器中。

- 寄存器变量是每个线程私有的，一旦thread执行结束，寄存器变量就会失效。
- 寄存器是稀有资源。(省着点用，能让更多的block驻留在SM中，增加Occupancy)
- `--maxrregcount` 可以设置大小
- 不同设备架构，数量不同

多种CUDA存储单元详解

Shared Memory：

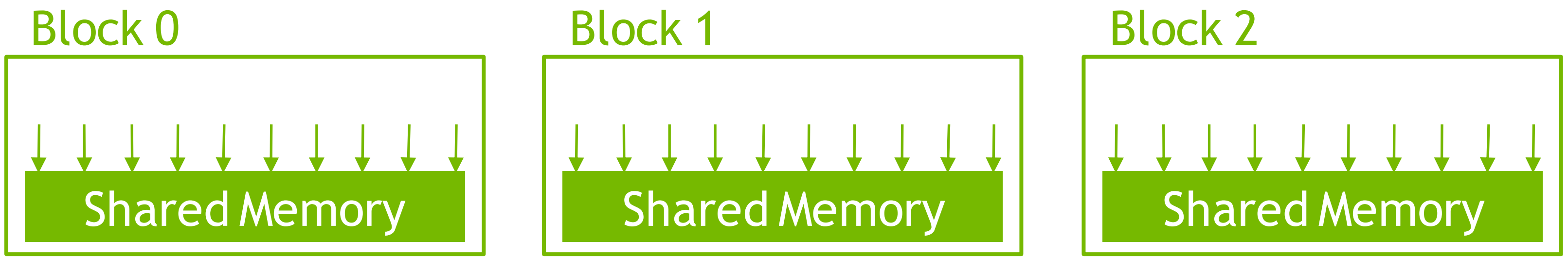
The only two types of memory that actually reside on the GPU chip are register and shared memory.

所以，Shared Memory是目前最快的可以让多个线程沟通的地方。

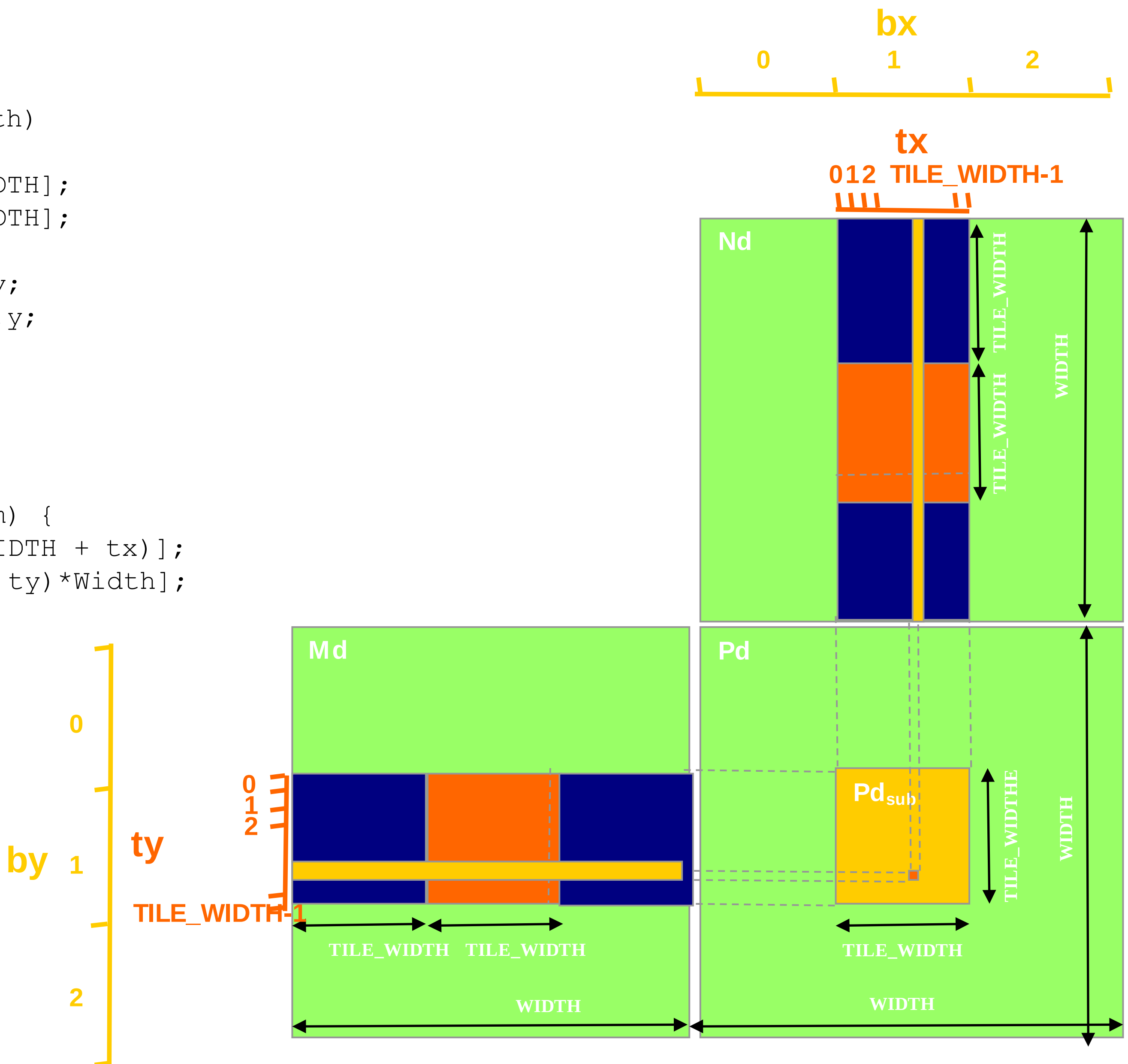
那么，就有可能会出现同时有很多线程访问Shared Memory上的数据。

为了克服这个同时访问的瓶颈，Shared Memory被分成32个逻辑块（banks）

Successive sections of memory are assigned to successive banks



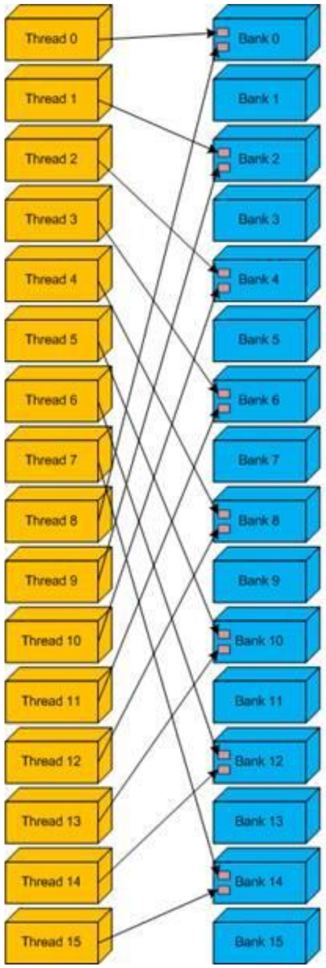
```
__global__ void MatrixMulKernel(
  float* Md, float* Nd, float* Pd, int Width)
{
  __shared__ float Mds[TILE_WIDTH][TILE_WIDTH];
  __shared__ float Nds[TILE_WIDTH][TILE_WIDTH];

  int bx = blockIdx.x;  int by = blockIdx.y;
  int tx = threadIdx.x; int ty = threadIdx.y;

  int Row = by * TILE_WIDTH + ty;
  int Col = bx * TILE_WIDTH + tx;

  float Pvalue = 0;
  for (int m = 0; m < Width/TILE_WIDTH; ++m) {
    Mds[ty][tx] = Md[Row*Width + (m*TILE_WIDTH + tx)];
    Nds[ty][tx] = Nd[Col + (m*TILE_WIDTH + ty)*Width];
    __syncthreads();

    for (int k = 0; k < TILE_WIDTH; ++k)
      Pvalue += Mds[ty][k] * Nds[k][tx];
    __synchthreads();
  }
  Pd[Row*Width+Col] = Pvalue;
}
```

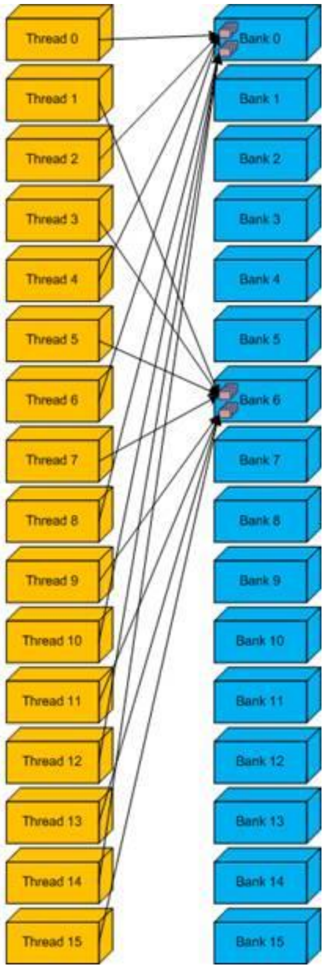


Shared memory

Bank conflict

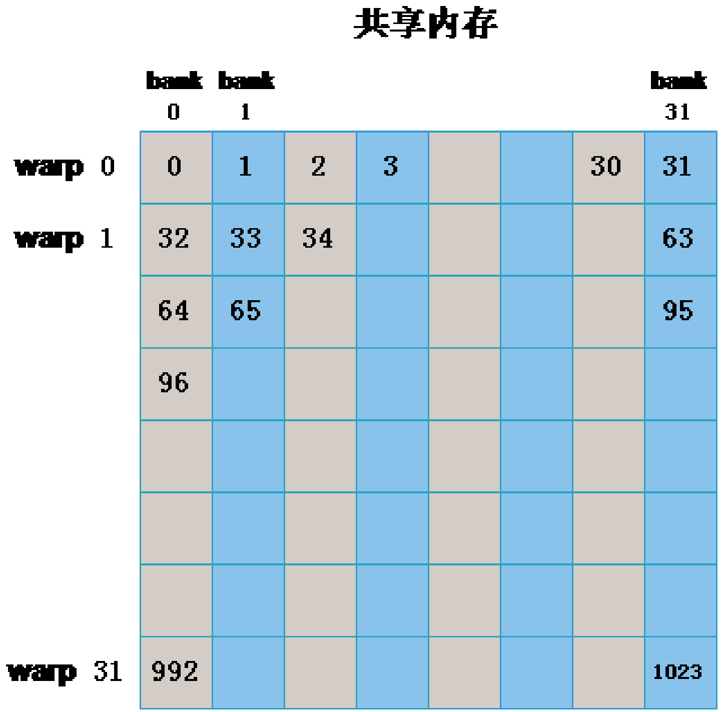
1. 同常量内存一样，当一个 warp 中的所有线程访问同一地址的共享内存时，会触发一个广播(broadcast)机制到 warp 中所有线程，这是最高效的。
2. 如果同一个 half-warp/warp 中的线程访问同一个 bank 中的不同地址时将发生 bank conflict。
3. 每个 bank 除了能广播(broadcast)还可以多播(mutilcast)(计算能力 >= 2.0)，也就是说，如果一个 warp 中的多个线程访问同一个 bank 的同一个地址时(其他线程也没有访问同一个bank 的不同地址)不会发生 bank conflict。
4. 即使同一个 warp 中的线程 随机的访问不同的 bank，只要没有访问同一个 bank 的不同地址就不会发生 bank conflict。





Shared memory

如何避免 Bank conflict

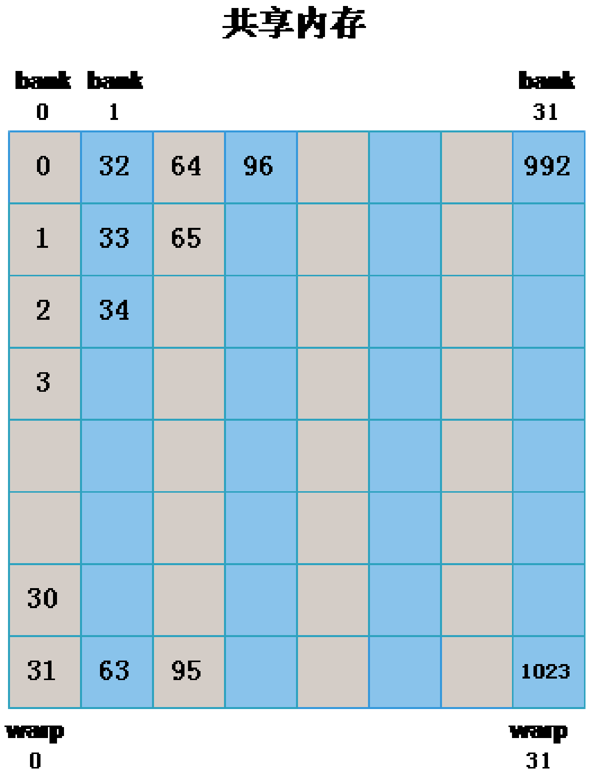


此时是没有 bank conflict

```
int x_id = blockDim.x * blockIdx.x + threadIdx.x; //列坐标
int y_id = blockDim.y * blockIdx.y + threadIdx.y; //行坐标
int index = y_id * col + x_id;

__shared__ float sData[BLOCKSIZE][BLOCKSIZE];

if (x_id < col && y_id < row)
{
    sData[threadIdx.y][threadIdx.x] = matrix[index];
    __syncthreads();
    matrixTest[index] = sData[threadIdx.y][threadIdx.x];
}
```

Shared memory

如何避免 Bank conflict



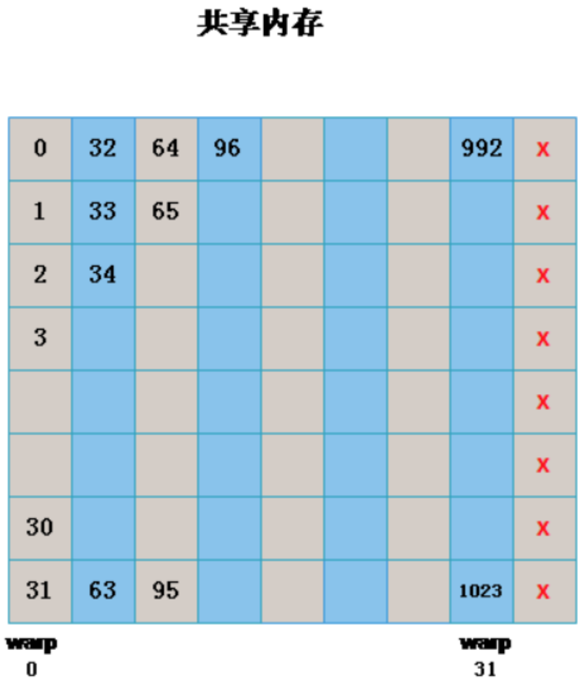
此时是有 bank conflict

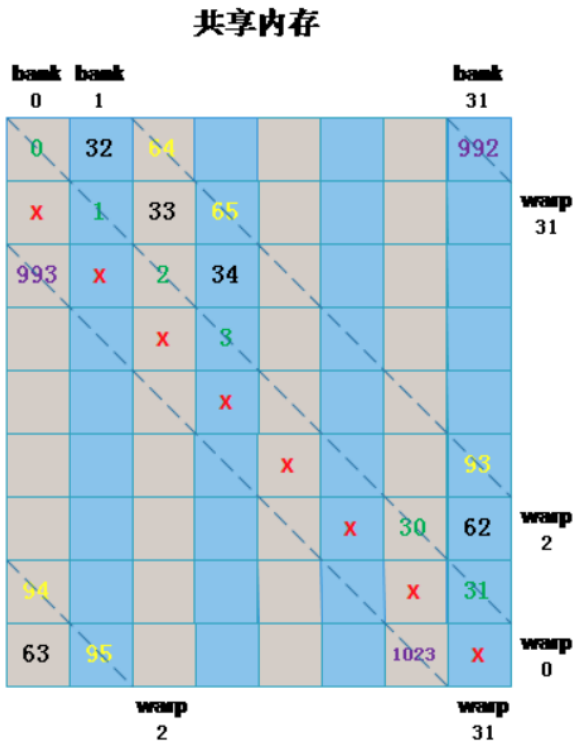
```
int x_id = blockDim.x * blockIdx.x + threadIdx.x; //列坐标
int y_id = blockDim.y * blockIdx.y + threadIdx.y; //行坐标
int index = y_id * col + x_id;

__shared__ float sData[BLOCKSIZE][BLOCKSIZE];

if (x_id < col && y_id < row)
{
    sData[threadIdx.x][threadIdx.y] = matrix[index];
    __syncthreads();
    matrixTest[index] = sData[threadIdx.x][threadIdx.y];
}
```

Shared memory

如何避免 Bank conflict

Memory Padding





Shared memory

如何避免 Bank conflict

Memory Padding

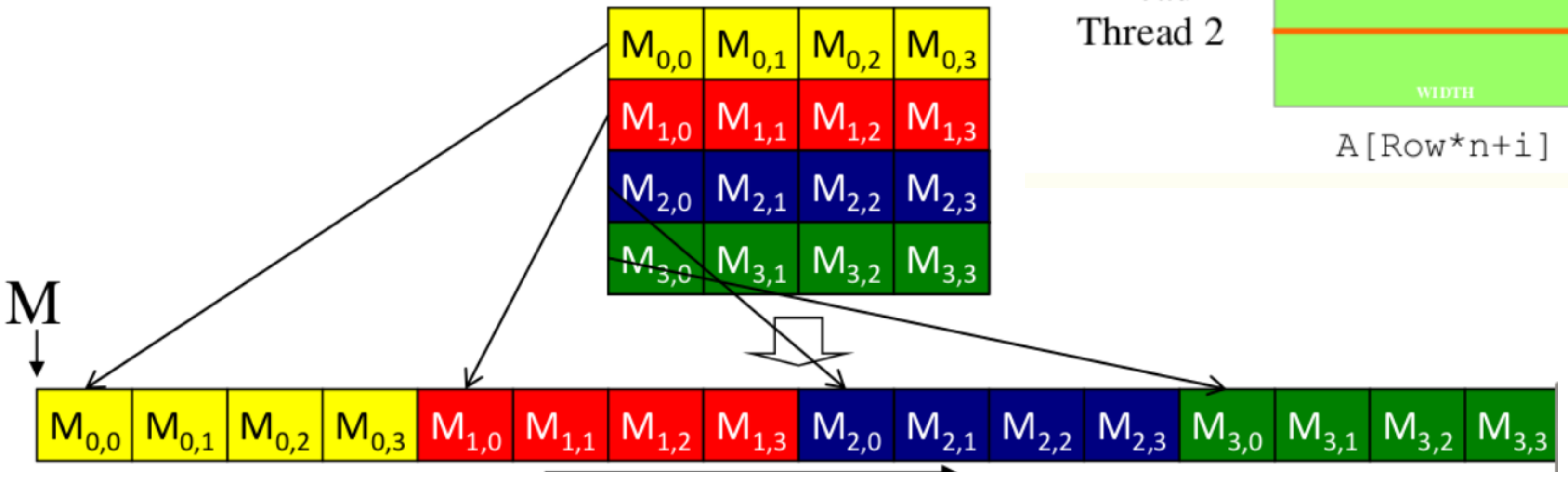
```
int x_id = blockDim.x * blockIdx.x + threadIdx.x; //列坐标
int y_id = blockDim.y * blockIdx.y + threadIdx.y; //行坐标
int index = y_id * col + x_id;

__shared__ float sData[BLOCKSIZE][BLOCKSIZE+1];

if (x_id < col && y_id < row)
{
    sData[threadIdx.x][threadIdx.y] = matrix[index];
    __syncthreads();
    matrixTest[index] = sData[threadIdx.x][threadIdx.y];
}
```

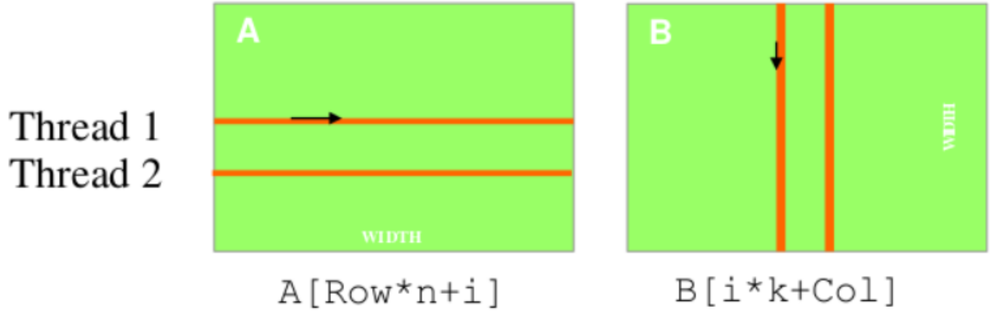
多种CUDA存储单元详解

Global Memory:

空间最大，latency最高，GPU最基础的memory：

- 驻留在Device memory中
- memory transaction对齐，合并访存



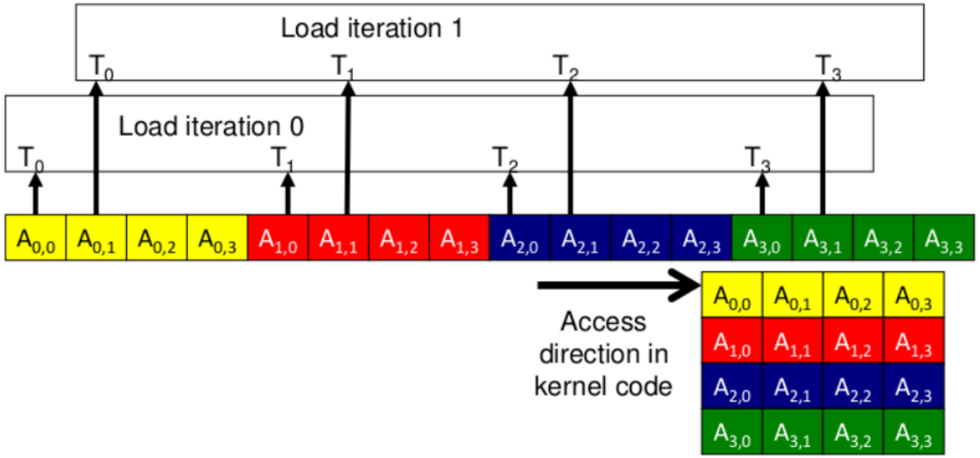


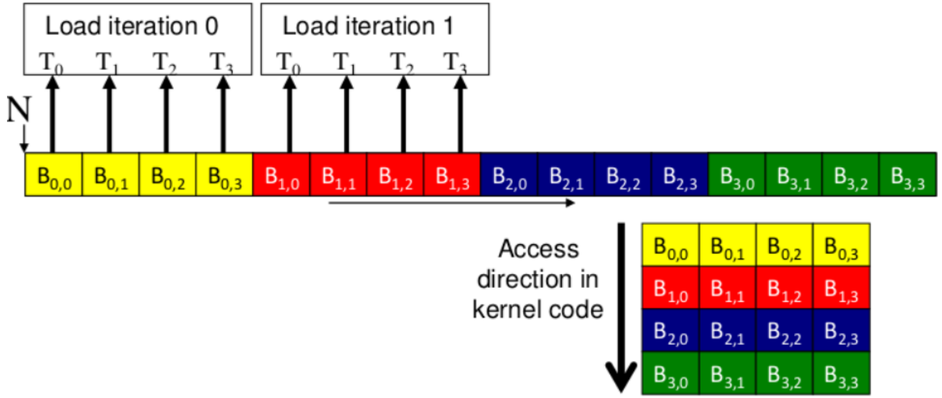
多种CUDA存储单元详解

Global Memory:

空间最大，latency最高，GPU最基础的memory：

- 驻留在Device memory中
- memory transaction对齐，合并访存





多种CUDA存储单元详解

Local Memory:

有时候， Registers 不够了，就会用Local Memory 来替代。但是，更多在以下情况，会使用Local Memory：

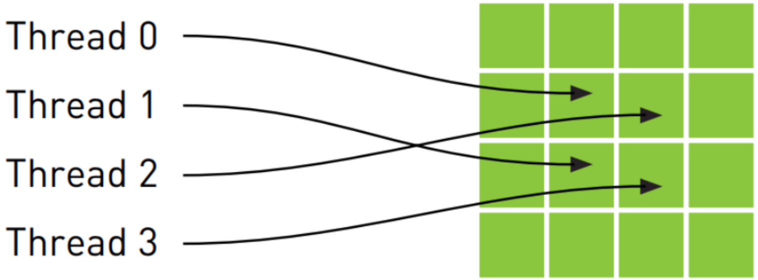
- 无法确定其索引是否为常量的数组。
- 会消耗太多寄存器空间的大型结构或数组。
- 如果内核使用了多于可用寄存器的任何变量(这也称为寄存器溢出)
- `--ptxas-options=-v`

多种CUDA存储单元详解

Texture Memory:

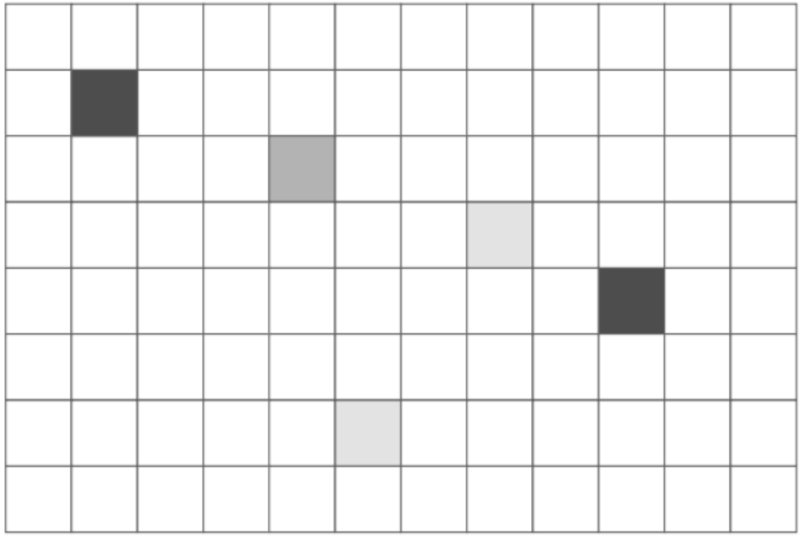
Texture Memory驻留在device Memory中，并且使用一个只读cache。Texture Mmeory是专门为那些在内存访问模式中存在大量空间局部性（Spatial Locality）的图形应用程序而设计的。意思是，在某个计算应用程序中，这意味着一个Thread读取的位置可能与邻近Thread读取的位置“非常接近”：

- Texture Memory实际上也是global Memory在一块，但是他有自己专有的只读cache。
- 纹理内存也是缓存在片上的，因此一些情况下相比从芯片外的DRAM上获取数据，纹理内存可以通过减少内存请求来提高带宽。
- 从数学的角度，下图中的4个地址并非连续的，在一般的CPU缓存中，这些地址将不会缓存。但由于GPU纹理缓存是专门为了加速这种访问模式而设计的，因此如果在这种情况中使用纹理内存而不是全局内存，那么将会获得性能的提升。



多种CUDA存储单元详解

Texture Memory:
实例：热传导模型



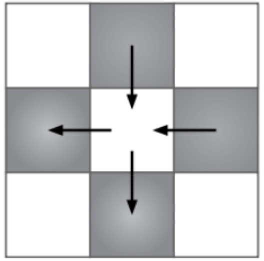


Figure 7.3 Heat dissipating from warm cells into cold cells

$$T_{NEW} = T_{OLD} + \sum_{NEIGHBORS} k \cdot (T_{NEIGHBOR} - T_{OLD})$$

多种CUDA存储单元详解

Texture Memory:
实例：热传导模型

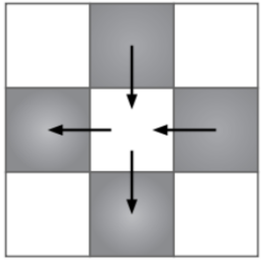


Figure 7.3 Heat dissipating from warm cells into cold cells

$$T_{NEW} = T_{OLD} + \sum_{NEIGHBORS} k \cdot (T_{NEIGHBOR} - T_{OLD})$$

```
HANDLE_ERROR( cudaMalloc( (void**)&data.output_bitmap,
                           imageSize ) );

// assume float == 4 chars in size (ie rgba)
HANDLE_ERROR( cudaMalloc( (void**)&data.dev_inSrc,
                          imageSize ) );
HANDLE_ERROR( cudaMalloc( (void**)&data.dev_outSrc,
                          imageSize ) );
HANDLE_ERROR( cudaMalloc( (void**)&data.dev_constSrc,
                          imageSize ) );

HANDLE_ERROR( cudaBindTexture( NULL, texConstSrc,
                               data.dev_constSrc,
                               imageSize ) );

HANDLE_ERROR( cudaBindTexture( NULL, texIn,
                               data.dev_inSrc,
                               imageSize ) );

HANDLE_ERROR( cudaBindTexture( NULL, texOut,
                               data.dev_outSrc,
                               imageSize ) );
```

多种CUDA存储单元详解

Texture Memory:
实例：热传导模型

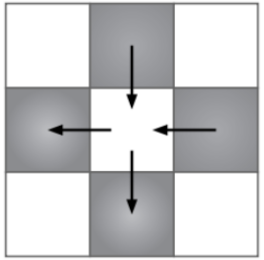


Figure 7.3 Heat dissipating from warm cells into cold cells

$$T_{NEW} = T_{OLD} + \sum_{NEIGHBORS} k \cdot (T_{NEIGHBOR} - T_{OLD})$$

```
__global__ void blend_kernel( float *dst,
                              bool dstOut ) {
    // map from threadIdx/BlockIdx to pixel position
    int x = threadIdx.x + blockIdx.x * blockDim.x;
    int y = threadIdx.y + blockIdx.y * blockDim.y;
    int offset = x + y * blockDim.x * gridDim.x;

    int left = offset - 1;
    int right = offset + 1;
    if (x == 0)   left++;
    if (x == DIM-1) right--;

    int top = offset - DIM;
    int bottom = offset + DIM;
    if (y == 0)   top += DIM;
    if (y == DIM-1) bottom -= DIM;

    float   t, l, c, r, b;
    if (dstOut) {
        t = tex1Dfetch(texIn,top);
        l = tex1Dfetch(texIn,left);
        c = tex1Dfetch(texIn,offset);
        r = tex1Dfetch(texIn,right);
        b = tex1Dfetch(texIn,bottom);

    } else {
        t = tex1Dfetch(texOut,top);
        l = tex1Dfetch(texOut,left);
        c = tex1Dfetch(texOut,offset);
        r = tex1Dfetch(texOut,right);
        b = tex1Dfetch(texOut,bottom);
    }
    dst[offset] = c + SPEED * (t + b + r + l - 4 * c);
}
```

多种CUDA存储单元详解

Constant Memory:

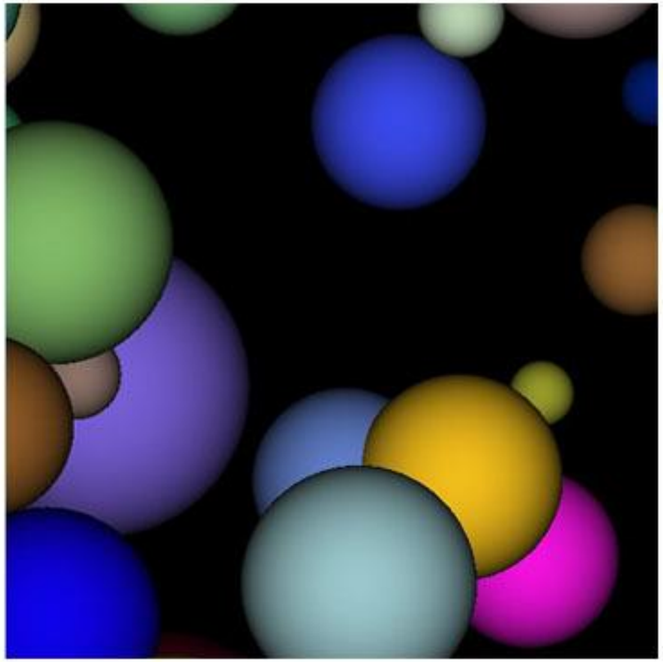
固定内存空间驻留在设备内存中，并缓存在固定缓存中（constant cache）：

- constant的范围是全局的，针对所有kernel。
- 在同一个编译单元，constant对所有kernel可见。
- kernel只能从constant Memory读取数据，因此其初始化必须在host端使用下面的function调用：

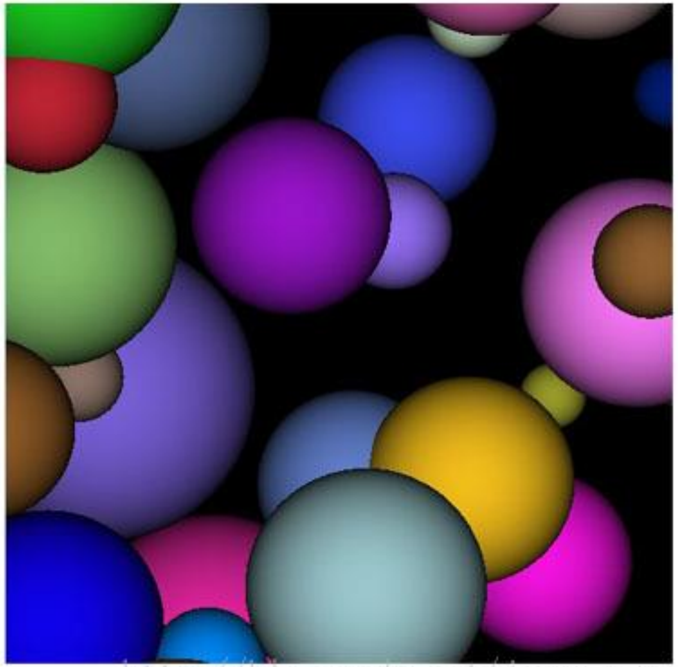
  cudaError_t cudaMemcpyToSymbol(const void* symbol, const void* src,size_t count);
- 当一个warp中所有thread都从同一个Memory地址读取数据时，constant Memory表现会非常好，会触发广播机制。

常量内存

光线跟踪



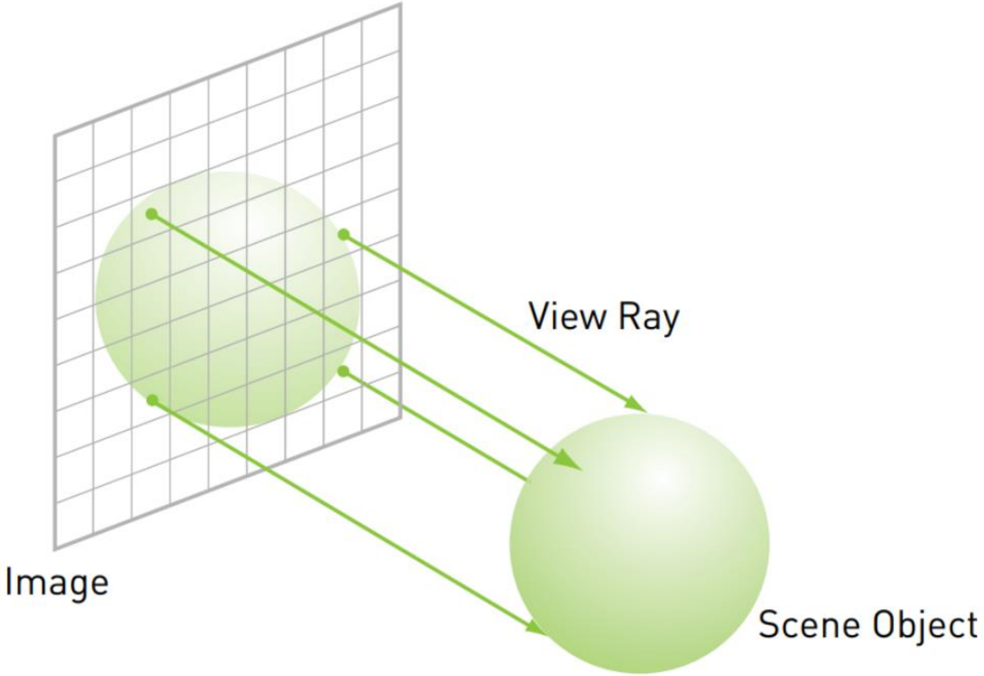
a. Scene with 50 Spheres



b. Scene with 50 Spheres

常量内存

光线跟踪



多种CUDA存储单元详解

Table 1. Salient Features of Device Memory

Memory	Location on/off chip	Cached	Access	Scope	Lifetime
Register	On	n/a	R/W	1 thread	Thread
Local	Off	Yes††	R/W	1 thread	Thread
Shared	On	n/a	R/W	All threads in block	Block
Global	Off	†	R/W	All threads + host	Host allocation
Constant	Off	Yes	R	All threads + host	Host allocation
Texture	Off	Yes	R	All threads + host	Host allocation

† Cached in L1 and L2 by default on devices of compute capability 6.0 and 7.x; cached only in L2 by default on devices of lower compute capabilities, though some allow opt-in to caching in L1 as well via compilation flags.

†† Cached in L1 and L2 by default except on devices of compute capability 5.x; devices of compute capability 5.x cache locals only in L2.

各种Memory要灵活运用，自定义的方法的上下限更高

更多资源：

https://developer.nvidia-china.com


NVIDIA