CUDA ON ARM PLATFORM—多种CUDA存储单元详解

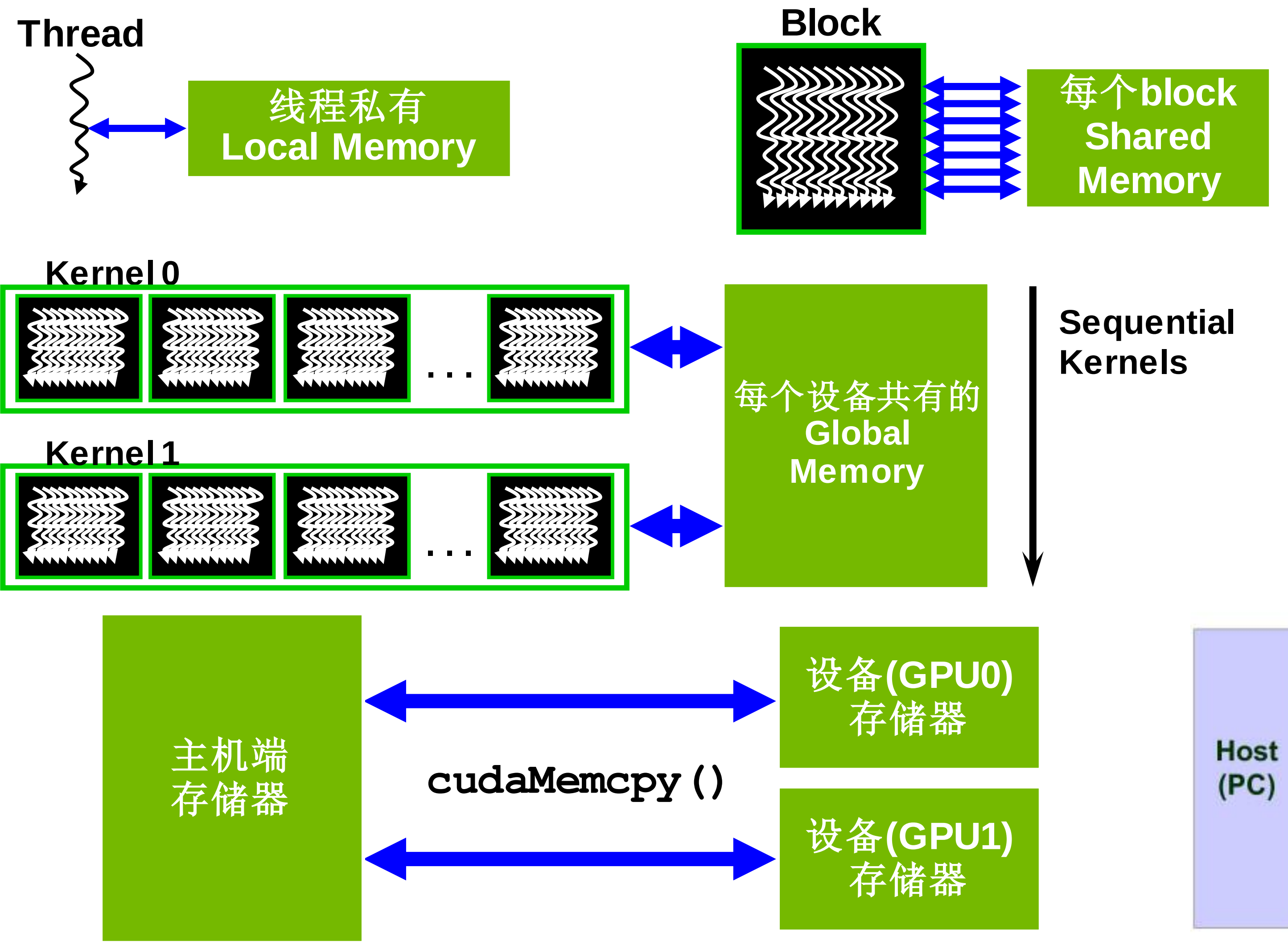
NVIDIA企业级开发者社区 何琨

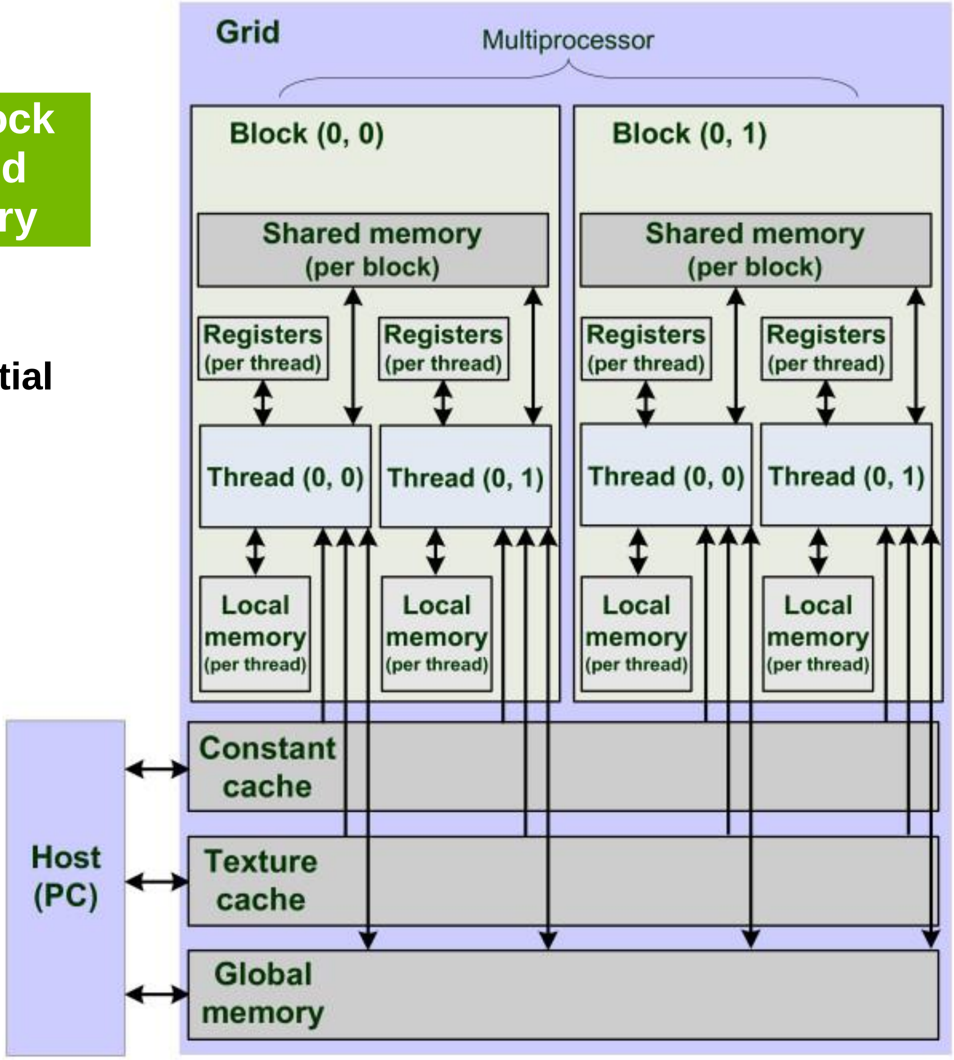
多种CUDA存储单元详解

AGENDA

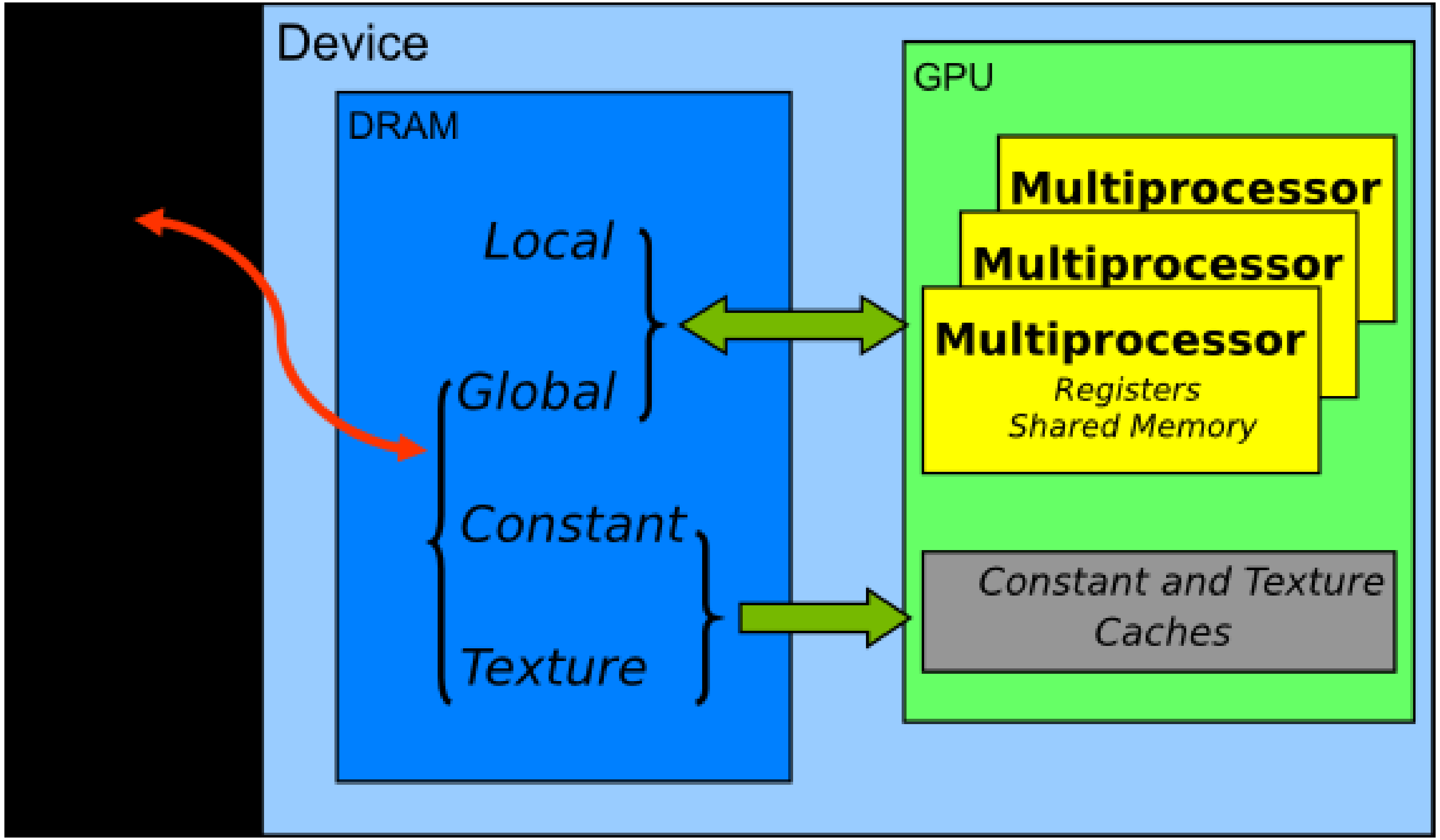
- CUDA中的存储单元种类
- CUDA中的各种存储单元的使用方法
- CUDA中的各种存储单元的适用条件

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:

用__shared__修饰符修饰的变量存放在shared memory：

- On-chip
- 拥有高的多bandwidth和低很多的latency。
- 同一个Block中的线程共享一块Shared Memory。
- __syncthreads()同步。
- 比较小，要节省着使用，不然会限制活动warp的数量

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

Constant Memory:

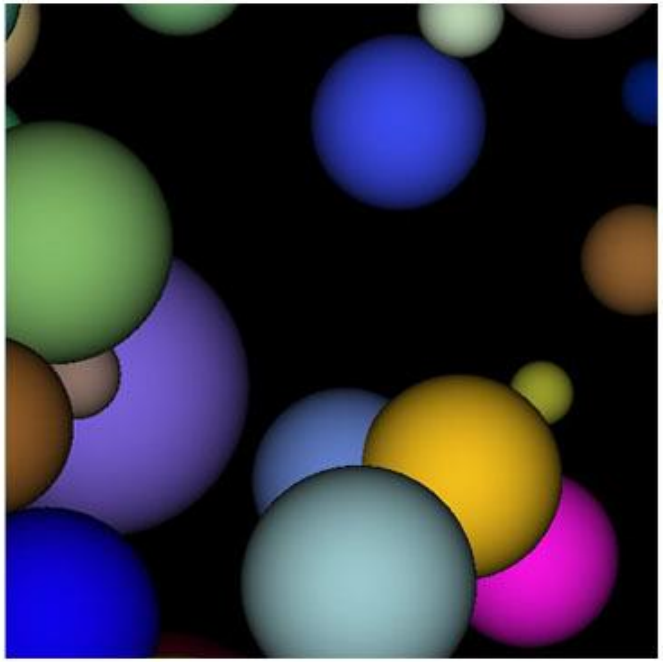
固定内存空间驻留在设备内存中，并缓存在固定缓存中（constant cache）：

- 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：

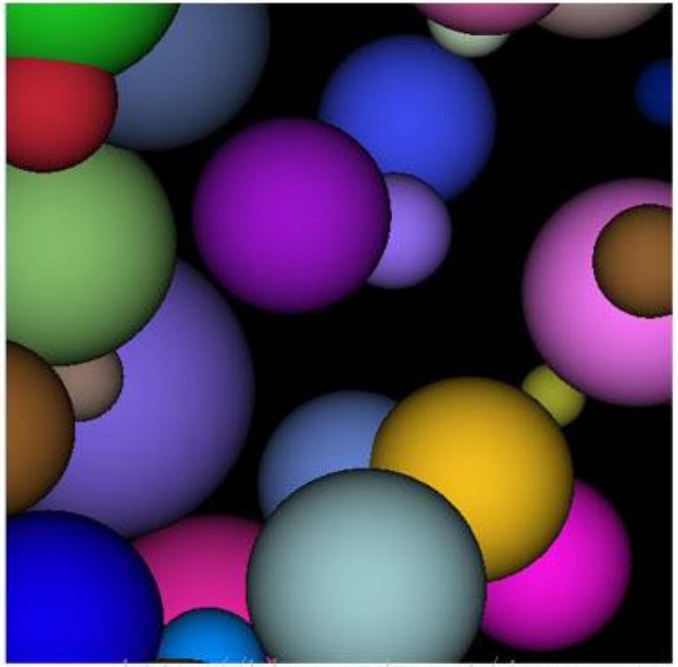
  cudaError_t cudaMemcpyToSymbol(const void* symbol, const void* src,size_t count);
- 当一个warp中所有thread都从同一个Memory地址读取数据时，constant Memory表现会非常好，会触发广播机制。

常量内存

光线跟踪



a. Scene with 50 Spheres



b. Scene with 50 Spheres

常量内存

hit方法，计算光线是否与球面相交，若相交则返回光线到命中球面处的距离

```
struct Sphere
{
        float r, g, b;
        float radius;
        float x, y, z;

        __device__ float hit(float ox, float oy, float *n)
        {
                float dx = ox - x;
                float dy = oy - y;

                if (dx*dx + dy*dy < radius*radius)
                {
                        float dz = sqrt(radius*radius - dx*dx - dy*dy);
                        *n = dz / sqrt(radius*radius);
                        return dz+z;
                }

                return -INF;
        }
};
```

常量内存

将threadIdx映射到像素位置

让图像坐标偏移DIM/2，使z轴穿过图像中心

初始化背景颜色为黑色

如果比上一次命中距离更接近，我将这个距离保存为最接近距离，并且保存球面颜色值

更新距离

判断球面相交情况后，将当前颜色保存到输出图像中

```
__constant__ Sphere s[SPHERES];

/******************************************************************************/
//__global__ void rayTracing(unsigned char* ptr, Sphere* s)
__global__ void rayTracing(unsigned char* ptr)
{
        int x = threadIdx.x + blockIdx.x * blockDim.x;
        int y = threadIdx.y + blockIdx.y * blockDim.y;
        int offset = x + y  * blockDim.x * gridDim.x;
        float ox = (x - DIM/2);
        float oy = (y - DIM/2);

        float r=0, g=0, b=0;
        float maxz = -INF;
        for (int i=0; i<SPHERES; i++)
        {
                float n;
                float t = s[i].hit(ox, oy, &n);
                if (t>maxz)
                {
                        float fscale = n;
                        r = s[i].r * fscale;
                        g = s[i].g * fscale;
                        b = s[i].b * fscale;
                        maxz = t;
                }
        }

        ptr[offset*4 + 0] = (int)(r*255);
        ptr[offset*4 + 1] = (int)(g*255);
        ptr[offset*4 + 2] = (int)(b*255);
        ptr[offset*4 + 3] = 255;
}
```

常量内存

生成球面的中心坐标颜色和半径

通过球面数据生成bitmap

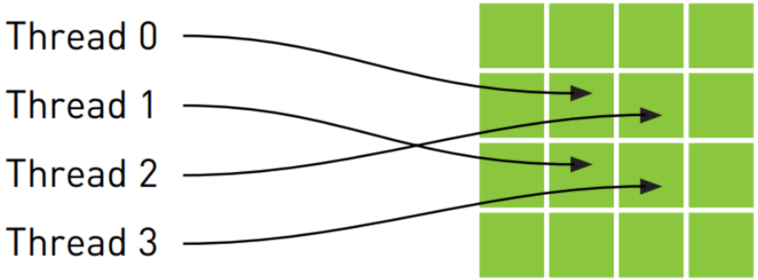
```
Sphere *temps = (Sphere*)malloc(sizeof(Sphere)*SPHERES);
for(int i=0; i<SPHERES; i++)
{
        temps[i].r = rnd(1.0f);
        temps[i].g = rnd(1.0f);
        temps[i].b = rnd(1.0f);
        temps[i].x = rnd(1000.0f) - 500;
        temps[i].y = rnd(1000.0f) - 500;
        temps[i].z = rnd(1000.0f) - 500;
        temps[i].radius = rnd(100.0f) + 20;
}

//      cutilSafeCall(cudaMemcpy(s, temps, sizeof(Sphere)*SPHERES, cudaMemcpyHostToDevice));
cutilSafeCall(cudaMemcpyToSymbol(s, temps, sizeof(Sphere)*SPHERES));
free(temps);

dim3 grids(DIM/16, DIM/16);
dim3 threads(16, 16);
//      rayTracing<<<grids, threads>>>(devBitmap, s);
rayTracing<<<grids, threads>>>(devBitmap);

cutilSafeCall(cudaMemcpy(bitmap.get_ptr(), devBitmap, bitmap.image_size(), cudaMemcpyDeviceToHost));

cutilSafeCall(cudaEventRecord(stop, 0));
cutilSafeCall(cudaEventSynchronize(stop));

float elapsedTime;
cutilSafeCall(cudaEventElapsedTime(&elapsedTime, start, stop));

printf("Processing time: %3.1f ms\n", elapsedTime);

bitmap.display_and_exit();

cudaFree(devBitmap);
```

多种CUDA存储单元详解

Texture Memory:

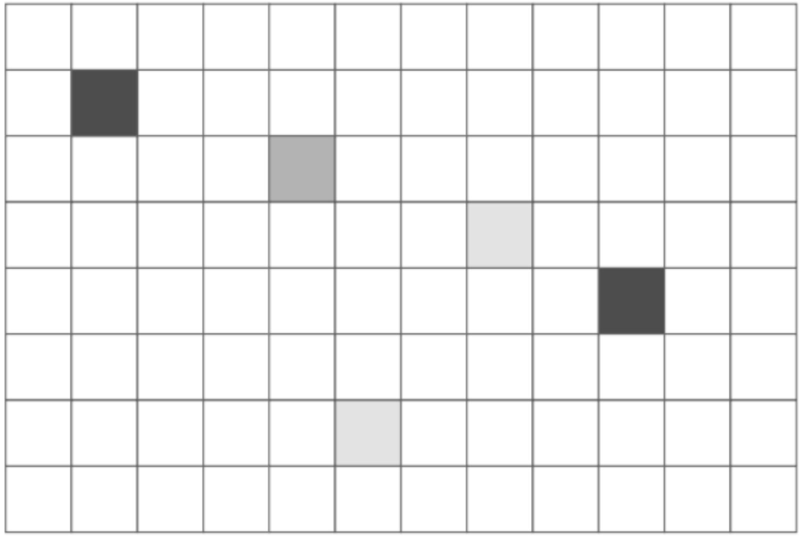
Texture Memory驻留在device Memory中，并且使用一个只读cache。Texture Mmeory是专门为那些在内存访问模式中存在大量空间局部性（Spatial Locality）的图形应用程序而设计的。意思是，在某个计算应用程序中，这意味着一个Thread读取的位置可能与邻近Thread读取的位置“非常接近”：

- 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



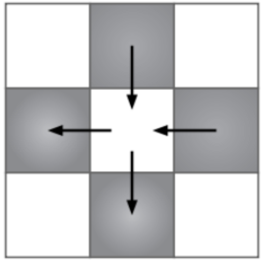


Figure 7.3 Heat dissipating from warm cells into cold cells

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

多种CUDA存储单元详解

Texture Memory:
实例：热传导模型

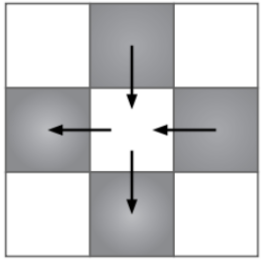


Figure 7.3 Heat dissipating from warm cells into cold cells

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

```
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

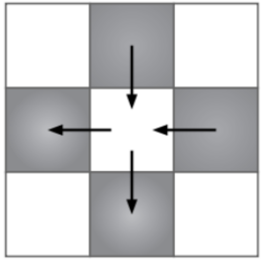


Figure 7.3 Heat dissipating from warm cells into cold cells

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

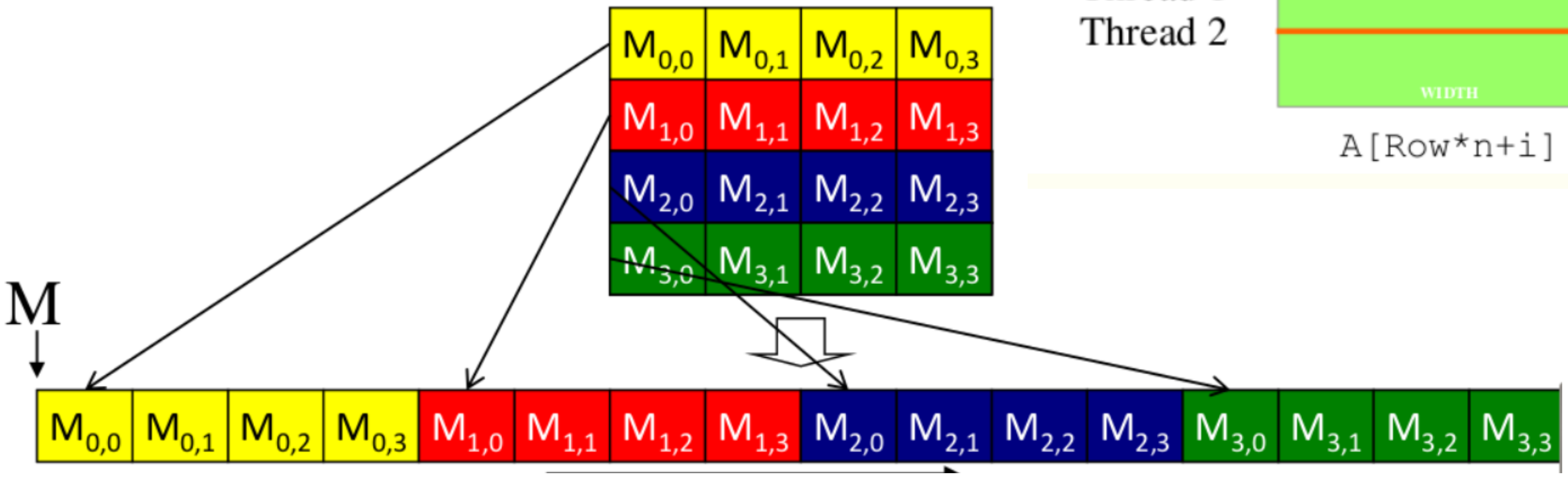
```
__global__ void blend_kernel( float *dst,
                              bool dstOut ) {
    // map from threadIdx/BlockIdx to pixel position
    int x = threadIdx.x + blockIdx.x * blockDim.x;
    int y = threadIdx.y + blockIdx.y * blockDim.y;
    int offset = x + y * blockDim.x * gridDim.x;

    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);
}
```

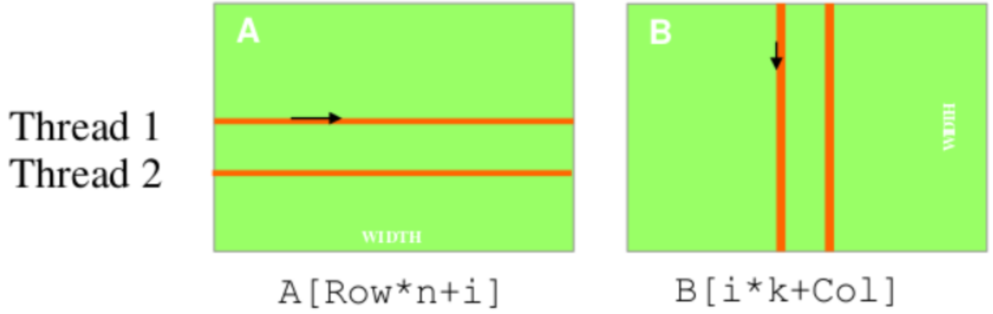
多种CUDA存储单元详解

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



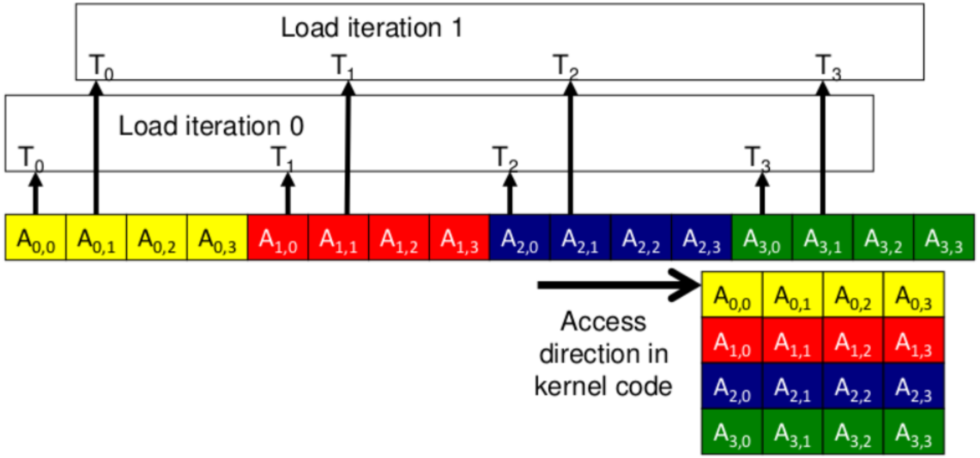


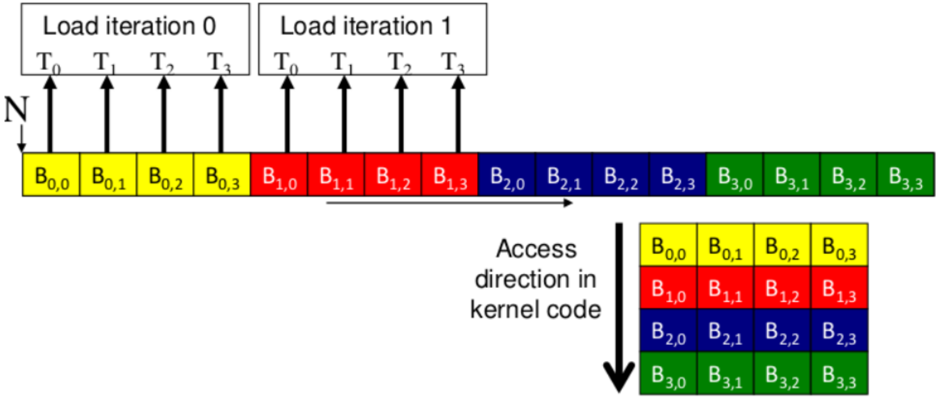
多种CUDA存储单元详解

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

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.

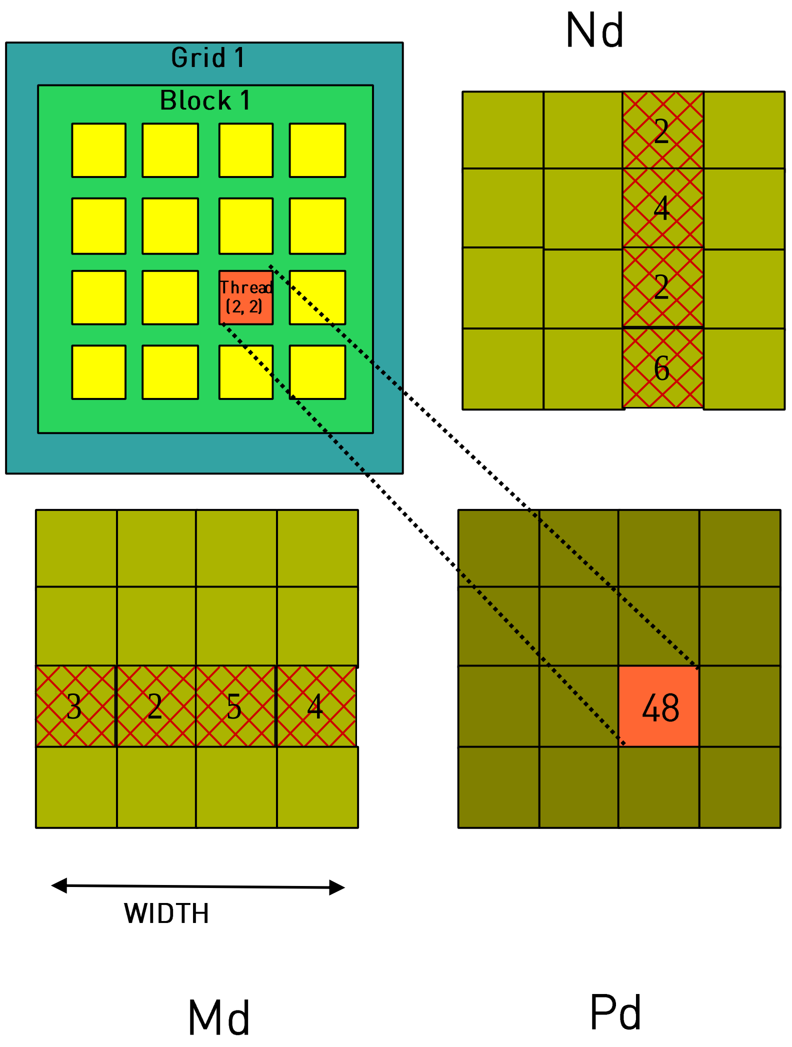
†† Cached in L1 and L2 by default except on devices of compute capability 5.x; devices of compute capability 5.x cache locals only in L2.

各种Memory要灵活运用，自定义的方法的上下限更高

如何灵活运用各种存储单元来优化程序

当我们在使用Global Memory来做矩阵相乘的优化时：

一共做了多少次memory的读写操作?



更多资源：

https://developer.nvidia-china.com

https://www.nvidia.cn/developer/community-training/

NVIDIA®