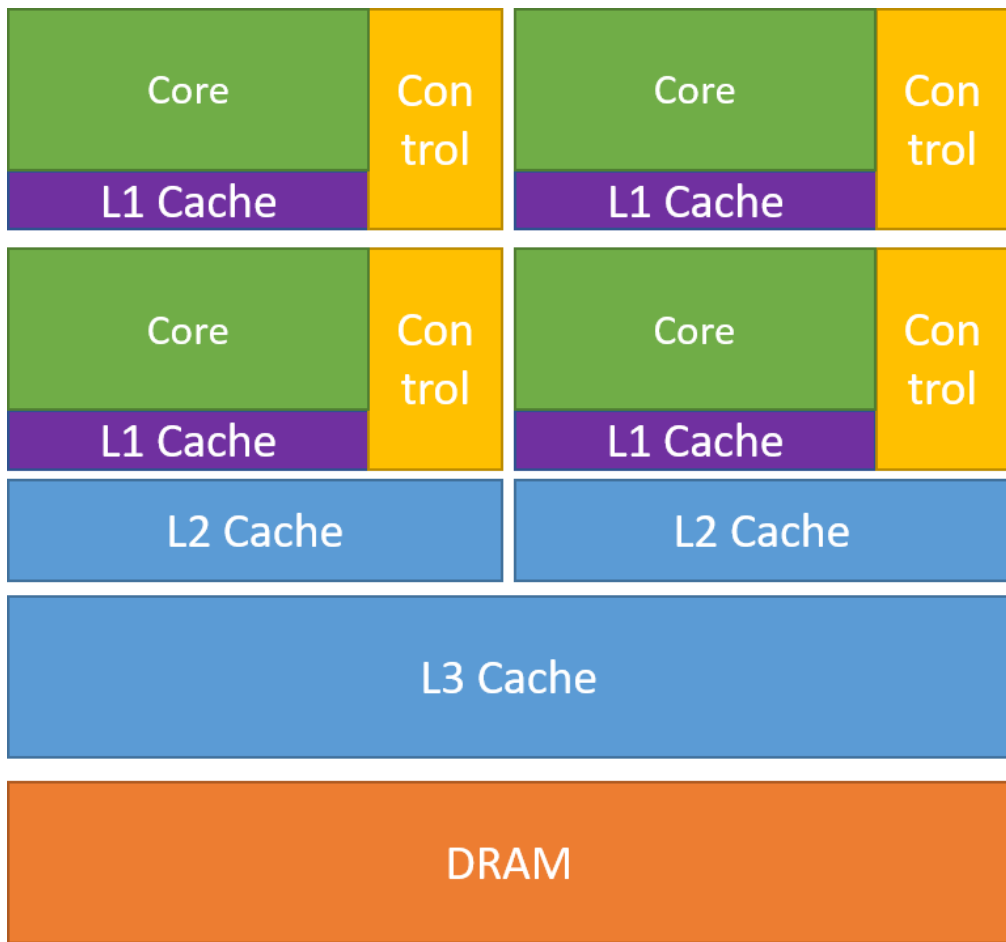


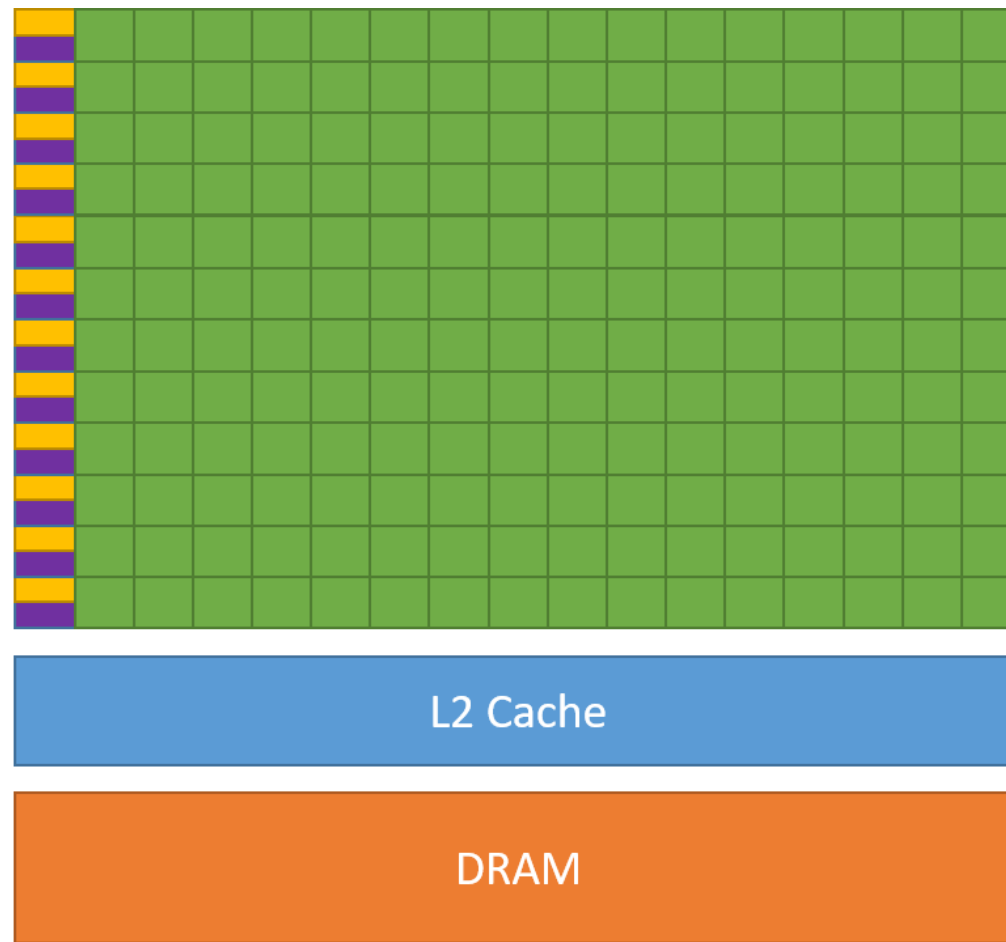
CUDA C++编程



CPU和GPU



CPU



GPU

C Program Sequential Execution

Serial code

Parallel kernel
Kernel0<<<>>>()

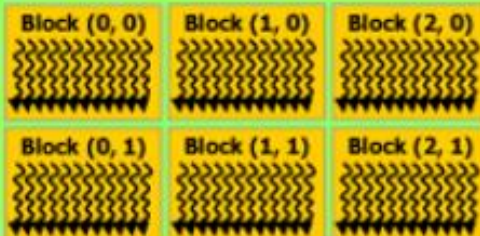
Serial code

Parallel kernel
Kernel1<<<>>>()

Host

Device

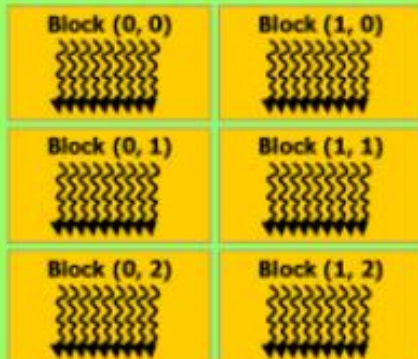
Grid 0



Host

Device

Grid 1







CPU+GPU异构计算

CPU在host端调用CUDA kernel函数后立即返回，继续执行后续的host code，并在某个同步点等待。而GPU会同时并行的执行CUDA code。同步点等待成功之后，继续向下执行



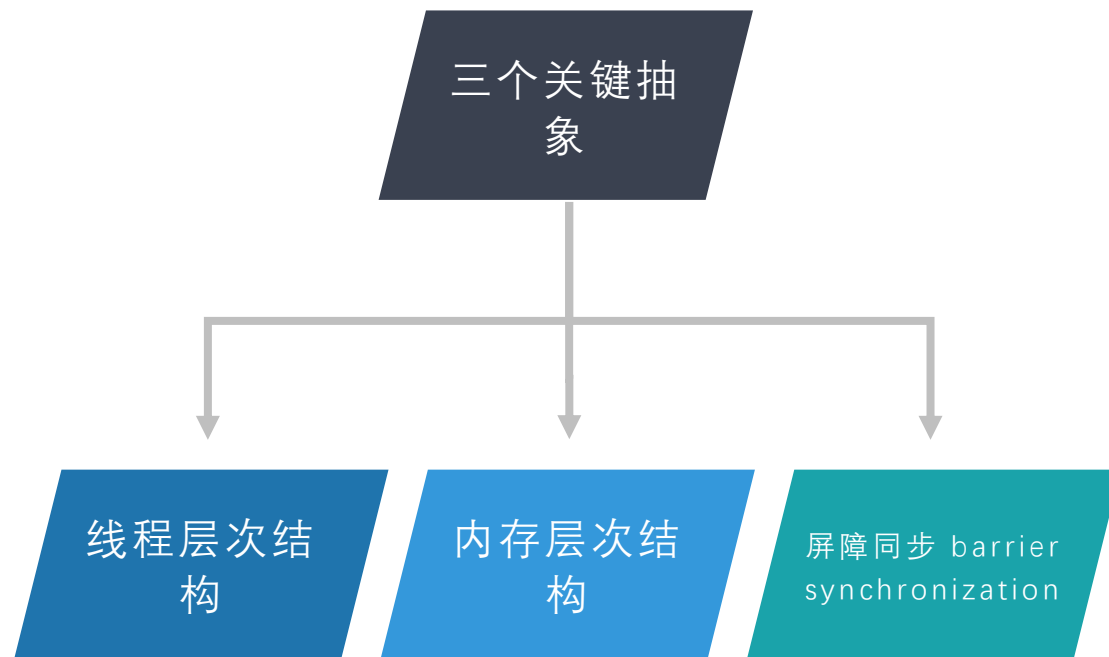
- CUDA可以用在Tesla、GeForce、Tegra等多种型号的GPU上，并且支持多种编程语言。
- NVIDIA基于CUDA进一步提供了非常多的库和中间件，比如：
 - 矩阵运算库cuBLAS
 - 快速傅里叶变换cuFFT
 - 求解线性代数问题的CULA
 - 用于深度学习的cuDNN和TensorRT
 - 类似C++ STL的cuda模板库Thrust
 - 包含一系列图像信号处理功能的NPP

GPU Computing Applications						
Libraries and Middleware						
cuDNN TensorRT	cuFFT cuBLAS cuRAND cuSPARSE	CULA MAGMA	Thrust NPP	VSIPL SVM OpenCurrent	PhysX OptiX iRay	MATLAB Mathematica
Programming Languages						
C	C++	Fortran	Java Python Wrappers	DirectCompute	Directives (e.g. OpenACC)	
CUDA-Enabled NVIDIA GPUs						
NVIDIA Ampere Architecture (compute capabilities 8.x)					Tesla A Series	
NVIDIA Turing Architecture (compute capabilities 7.x)		GeForce 2000 Series	Quadro RTX Series		Tesla T Series	
NVIDIA Volta Architecture (compute capabilities 7.x)	DRIVE/JETSON AGX Xavier		Quadro GV Series		Tesla V Series	
NVIDIA Pascal Architecture (compute capabilities 6.x)	Tegra X2	GeForce 1000 Series	Quadro P Series		Tesla P Series	
	 Embedded	 Consumer Desktop/Laptop	 Professional Workstation	 Data Center		



CUDA编程模型

CUDA编程模型

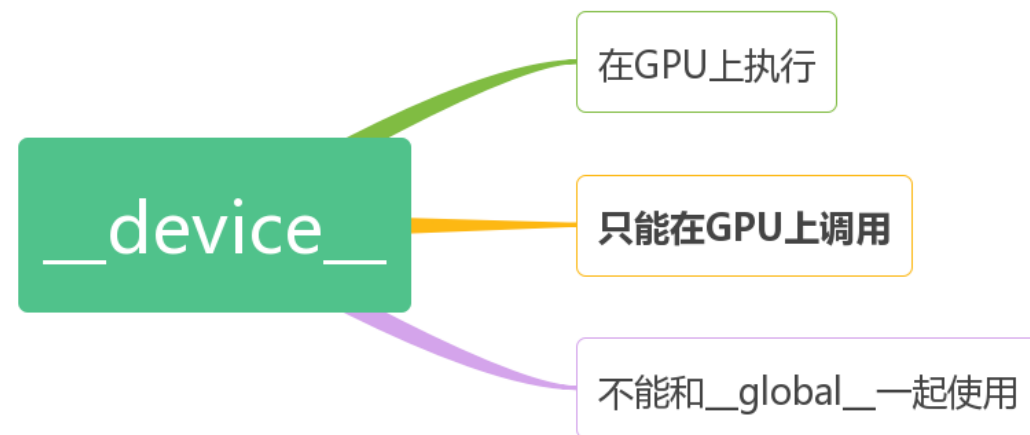


这些抽象提供了细粒度的数据并行和线程并行，嵌套在粗粒度的数据并行和任务并行中。它们指导我们将问题划分为粗略的子问题，这些子问题可以由线程块独立地并行解决，而每个子问题则划分为更细的部分，可以由块内的所有线程合作并行解决



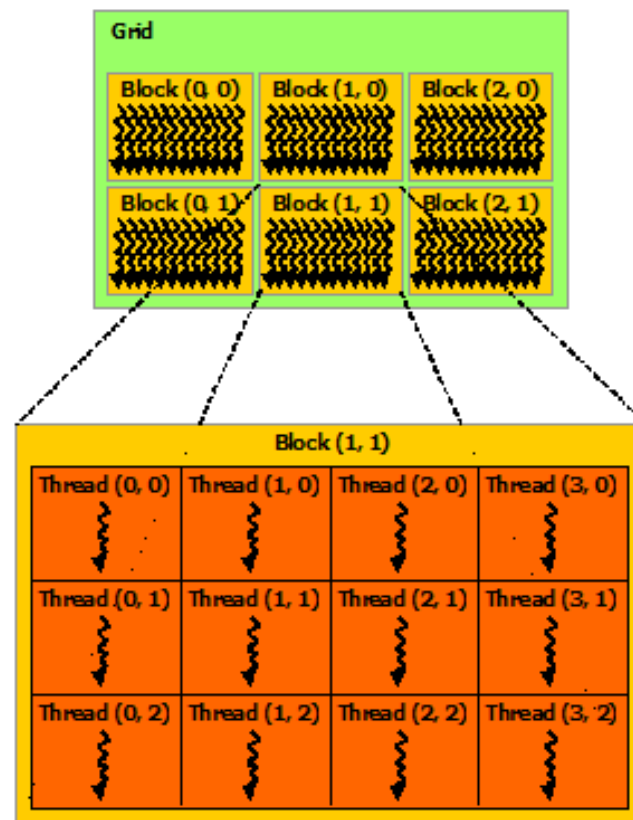
```
__global__ void func_name(func_args);  
__device__ T func_name(func_args);
```

- CUDA编程模型允许程序员自定义一种被称为CUDA kernel的函数，被调用时由N个不CUDA线程并行执行N次，而不是像普通C++函数那样只执行一次。
- CUDA kernel使用__global__或者__device__关键字来声明





- 每个执行内核的线程都有一个唯一的线程ID，可以在kernel函数内通过内置变量访问，比如**threadIdx**
- threadIdx是一个3分量的向量，因此可以使用一维、二维或三维的线程索引来识别线程，形成一个一维、二维或三维的线程块，称为thread block。每个block包含的线程数不是无限多的。
- 线程块被组织成一个一维、二维或三维的线程块网格。网格中的线程块的数量通常由正在处理的数据的大小决定，这通常超过了系统中的处理器数量。
- 网格中的每个块可以通过一个一维、二维或三维的唯一索引来识别，在内核中可以通过内置的**blockIdx**变量访问。
- 线程块的尺寸可以在内核中通过内置的**blockDim**变量访问
- 线程网格的尺寸可以在内核中通过内置的**gridDim**变量访问





```
// Kernel definition
__global__ void MatAdd(float A[N][N], float B[N][N], float C[N][N])
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    if (i < N && j < N) C[i][j] = A[i][j] + B[i][j];
}
int main() {
    // Kernel invocation
    dim3 threadsPerBlock(16, 16);
    dim3 numBlocks(N / threadsPerBlock.x, N / threadsPerBlock.y);
    MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
}
```



- GPU的核心是一组可扩展的流式多处理器（Streaming Multiprocessors, SM）。每个GPU有多个SM，每个SM可以支持数百个线程并行操作。kernel启动时，会分配多个block在多个SM上运行，每个SM处理多个块，每个块上的thread并发执行。SM越多，处理速度越快。
- 一个block的最大线程数是由SM硬件层面上决定的。一个block中的所有线程必须在一个SM上进行操作，必须共享所有该内核的资源。故目前来说**一个block块最多1024个线程**。
- CUDA在运行时会对同一个block中的线程进行分组，**每组32个线程，被称为warps**。**同一warps中的所有线程执行相同指令**。
- block会以连续的方式划分warp。例如，如果一个block由64个thread，则分为2组warp。0-31为warp0，32-63为warp1。如果block不是32的倍数，则多余的thread独立分成一组warp。例如block有65个thread，则最后一个thread单独为一个warp，那么此时这个warp中的其他thread处于非活动状态。
- SM一次只会并行执行多个block中的一个warp，当某个块中的warp在存取数据时，会切换到同一个块中其他warp执行
- 在处理完某个block中的所有thread后，SM会找还没有处理的block进行处理。每个SM处理block的个数与硬件有关。假设16个SM，64个block，每个SM处理3个block，则一开始会有48个block被占用，16个block空闲，等某个SM处理完一个block后，才选择某个空闲的block进行处理，知道所有block处理完成



- warp发散

- 假设在kernel中有一些控制语句，则warp会出现发散行为，即同一warp中执行不同的指令。
- 假设kernel中出现如下控制语句，并且一个warp出现两个分支，16个线程执行true条件，另外16个线程执行false条件。这就是warp发散行为。
- 当warp中出现发散行为时，warp会串行执行每个分支路径，并禁用其他非活动时的路径中的线程。这会造成性能显著下降。

```
if (cond) {  
    ...  
} else {  
    ...  
}
```

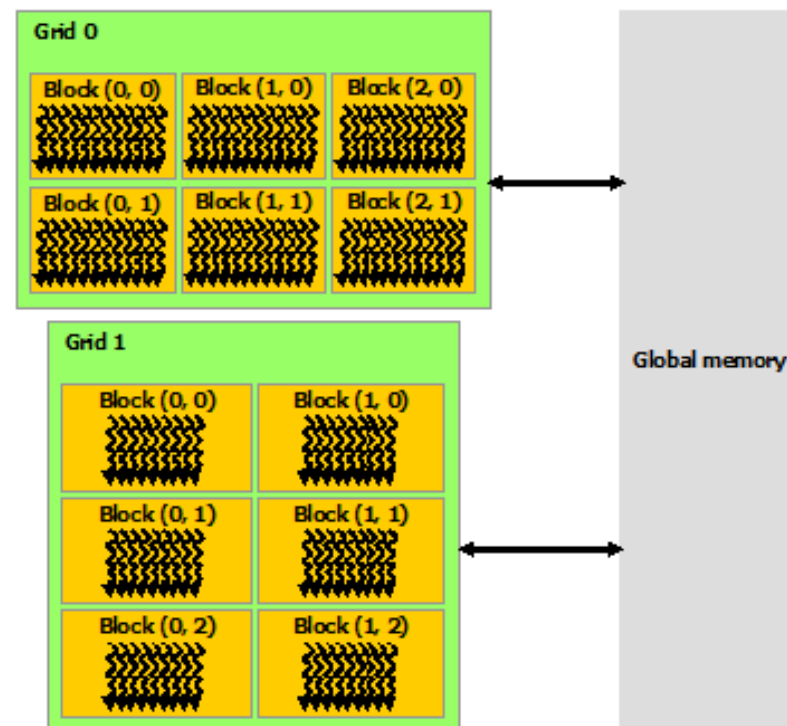
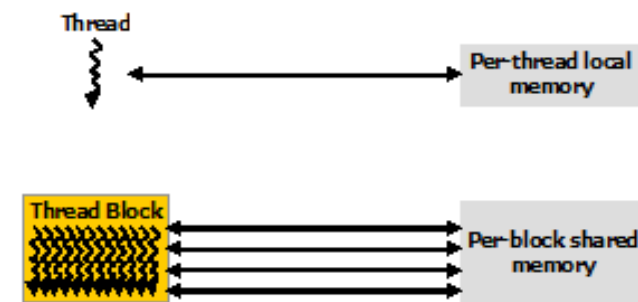


•线程同步

- 一个块内的线程可以通过一些共享内存来共享数据，并通过同步它们的执行来协调内存访问。
- 更准确地说，我们可以通过调用__syncthreads()来指定内核中的同步点
- __syncthreads()作为一个屏障，块中的所有线程必须在这个障碍处等待，然后才允许继续进行



- CUDA线程在执行过程中可以从多个内存空间访问数据。
- 每个线程都有私有的**本地内存**。
- 每个线程块都有**共享内存**，对该块的所有线程都是可见的，并且与该块具有相同的生命周期。
- 所有线程都可以访问相同的**全局内存**
- 还有两个额外的只读内存空间可供所有线程访问：**常量和纹理内存空间**
- 根据访问速度从快到慢排序，本地内存 > 共享内存 > 常量内存 = 纹理内存 > 全局内存





CUDA PROGRAMMING INTERFACE

CUDA Runtime API

提供了在主机上执行的C和C++函数，用于分配和释放设备内存，在主机内存和设备内存之间传输数据，管理具有多个设备的系统等

1

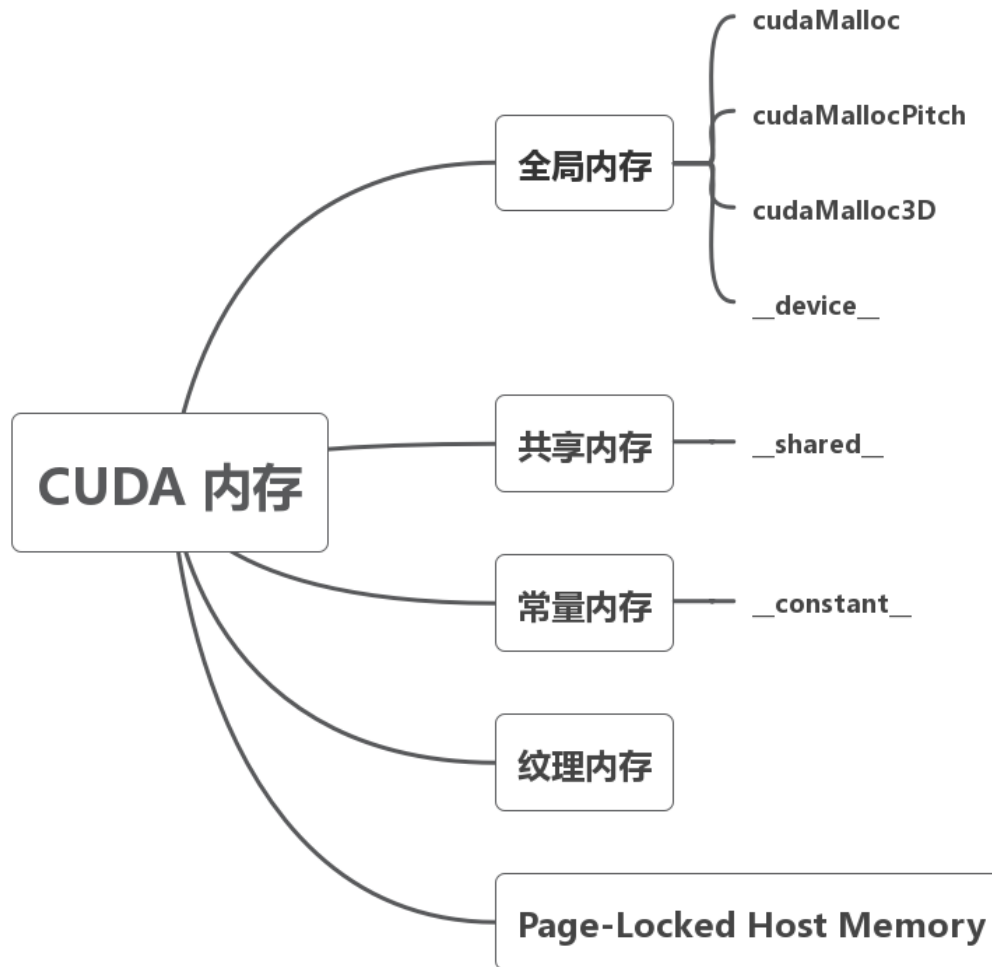
CUDA Driver API

- CUDA driver API暴露了低级别的概念，如CUDA上下文（设备的主机进程的类似物）和CUDA modules（设备的动态加载库的类似物）
- 大多数应用程序不使用驱动API，而且在使用runtime时，上下文和模块管理是隐含的。
- 由于运行时与驱动API具有互操作性，大多数需要某些驱动API功能的应用程序可以默认使用运行时API，只在需要时使用驱动API

2



- CUDA runtime没有明确的初始化函数；它在第一次调用运行时函数） 时进行初始化。
- 在对运行时函数调用进行计时，以及在解释首次调用运行时的错误代码时，需要记住这一点。
- 在初始化过程中，运行时为系统中的每个设备创建一个CUDA上下文。该上下文是该设备的主要上下文，它在应用程序的所有主机线程之间共享
- 当主机线程调用**cudaDeviceReset()**时，这将破坏主机线程当前操作的设备的主要上下文。任何以该设备为当前设备的主机线程的下次运行时函数调用将为该设备创建一个新的主上下文。





- GPU的全局内存之所以是全局的，只是因为GPU和CPU都可以对其进行写操作。
- 任何设备都可以通过PCI-E总线对其进行访问
- GPU之间不通过CPU，直接将数据从一块GPU卡传输到另一块GPU卡(需要平台支持)
- CPU主机端可以通过以下三种方式对GPU上的内存进行访问：
 - 显式地阻塞传输
 - 显式地非阻塞传输
 - 隐式地使用零内存复制
- 通常的执行模型是CPU将一个数据块传输到GPU上，GPU内核对其进行处理，然后再由CPU将数据块传输回主机端内存中。
- 更高级的模型是使用流，使数据传输和内核执行部分重叠。

- 最常用的全局内存分配方式:
 - 使用cudaMalloc分配内存
 - 使用cudaFree释放内存
 - 使用cudaMemcpy或者 cudaMemcpyAsync在主机和设备之间传输数据

```
// Device code
__global__ void VecAdd(float* A, float* B, float* C, int N) {
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    if (i < N) C[i] = A[i] + B[i];
}

int main() {
    int N = ...;
    size_t size = N * sizeof(float);
    // Allocate input vectors h_A and h_B in host memory
    float* h_A = (float*)malloc(size);
    float* h_B = (float*)malloc(size);
    // Initialize input vectors
    // Allocate vectors in device memory
    float* d_A;
    cudaMalloc(&d_A, size);
    float* d_B;
    cudaMalloc(&d_B, size);
    float* d_C;
    cudaMalloc(&d_C, size);
    // Copy vectors from host memory to device memory
    cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);
    // Invoke kernel
    int threadsPerBlock = 256;
    int blocksPerGrid = (N + threadsPerBlock - 1) / threadsPerBlock;
    VecAdd<<<blocksPerGrid, threadsPerBlock>>>>(d_A, d_B, d_C, N);
    // Copy result from device memory to host memory
    cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);
    // Free device memory
    cudaFree(d_A);
    cudaFree(d_B);
    cudaFree(d_C);
    // Free host memory
    free(h_A);
    free(h_B);
}
```



```
cudaError_t cudaMallocPitch(void** devPtr, size_t* pitch, size_t widthInBytes, size_t height);  
cudaError_t cudaMemcpy2D(void* dst, size_t dpitch, const void* src, size_t spitch, size_t width, size_t height,  
                        enum cudaMemcpyKind kind);
```

- **cudaMallocPitch**用于分配二维数组，因为它确保分配的内存被适当填充以满足对齐要求，以此来确保在访问行地址或执行拷贝时的最佳性能。
- 分配的二维数组使用**cudaFree**释放，使用**cudaMemcpy2D**拷贝数据
- 返回的间距（或**stride**）指定了一行数组所占的字节数
- 如果给定一个T类型数组元素的行和列，可按如下方法计算地址：

```
T* pElement = (T*)((char*)BaseAddress + Row * pitch) + Column;
```



```
__global__ void myKernel(float *devPtr, int height, int width, int pitch) {
    int row, col;
    float *rowHead;
    for (row = 0; row < height; row++) {
        rowHead = (float *)((char *)devPtr + row * pitch);
        for (col = 0; col < width; col++) {
            rowHead[col]++;
        }
    }
}

int main() {
    size_t width = 6;
    size_t height = 5;
    float *h_data, *d_data;
    size_t pitch;
    h_data = (float *)malloc(sizeof(float) * width * height);
    for (int i = 0; i < width * height; i++) h_data[i] = (float)i;
    cudaMallocPitch((void **)&d_data, &pitch, sizeof(float) * width, height);
    cudaMemcpy2D(d_data, pitch, h_data, sizeof(float) * width,
                 sizeof(float) * width, height, cudaMemcpyHostToDevice);
    myKernel<<<1, 1>>>(d_data, height, width, pitch);
    cudaDeviceSynchronize();
    cudaMemcpy2D(h_data, sizeof(float) * width, d_data, pitch,
                 sizeof(float) * width, height, cudaMemcpyDeviceToHost);
    free(h_data);
    cudaFree(d_data);
    return 0;
}
```



cudaMalloc3D用于分配符合对其要求的三维数组

```
__global__ void myKernel(cudaPitchedPtr devPitchedPtr,
                        cudaExtent extent) {
    float *devPtr = (float *)devPitchedPtr.ptr;
    float *sliceHead, *rowHead;
    for (int z = 0; z < extent.depth; z++) {
        sliceHead =
            (float *)((char *)devPtr +
                    z * devPitchedPtr.pitch * extent.height);
        for (int y = 0; y < extent.height; y++) {
            rowHead = (float *)((char *)sliceHead +
                               y * devPitchedPtr.pitch);
            for (int x = 0; x < extent.width / sizeof(float);
                x++) {
                rowHead[x]++;
            }
        }
    }
}
```

```
int main() {
    size_t width = 2;
    size_t height = 3;
    size_t depth = 4;
    float *h_data;
    cudaPitchedPtr d_data;
    cudaExtent extent;
    cudaMemcpy3DParms cpyParm;
    h_data = (float *)malloc(sizeof(float) * width * height * depth);
    for (int i = 0; i < width * height * depth; i++) h_data[i] = (float)i;
    extent = make_cudaExtent(sizeof(float) * width, height, depth);
    cudaMalloc3D(&d_data, extent);
    cpyParm = {0};
    cpyParm.srcPtr = make_cudaPitchedPtr((void *)h_data, sizeof(float) *
width, width, height);
    cpyParm.dstPtr = d_data;
    cpyParm.extent = extent;
    cpyParm.kind = cudaMemcpyHostToDevice;
    cudaMemcpy3D(&cpyParm);
    myKernel<<<1, 1>>>(d_data, extent);
    cudaDeviceSynchronize();
    cpyParm = {0};
    cpyParm.srcPtr = d_data;
    cpyParm.dstPtr = make_cudaPitchedPtr((void *)h_data, sizeof(float) *
width, width, height);
    cpyParm.extent = extent;
    cpyParm.kind = cudaMemcpyDeviceToHost;
    cudaMemcpy3D(&cpyParm);
    free(h_data);
    cudaFree(d_data.ptr);
    return 0;
}
```



- 使用**__device__**修饰的变量有以下特征：
 - 停留在全局内存空间中
 - 和创建它的CUDA上下文拥有相同的生命周期。
 - 每个设备有一个不同的对象。
 - 可以从网格内的所有线程以及主机通过运行时库(cudaGetSymbolAddress/ cudaGetSymbolSize() / cudaMemcpyToSymbol() / cudaMemcpyFromSymbol) 访问

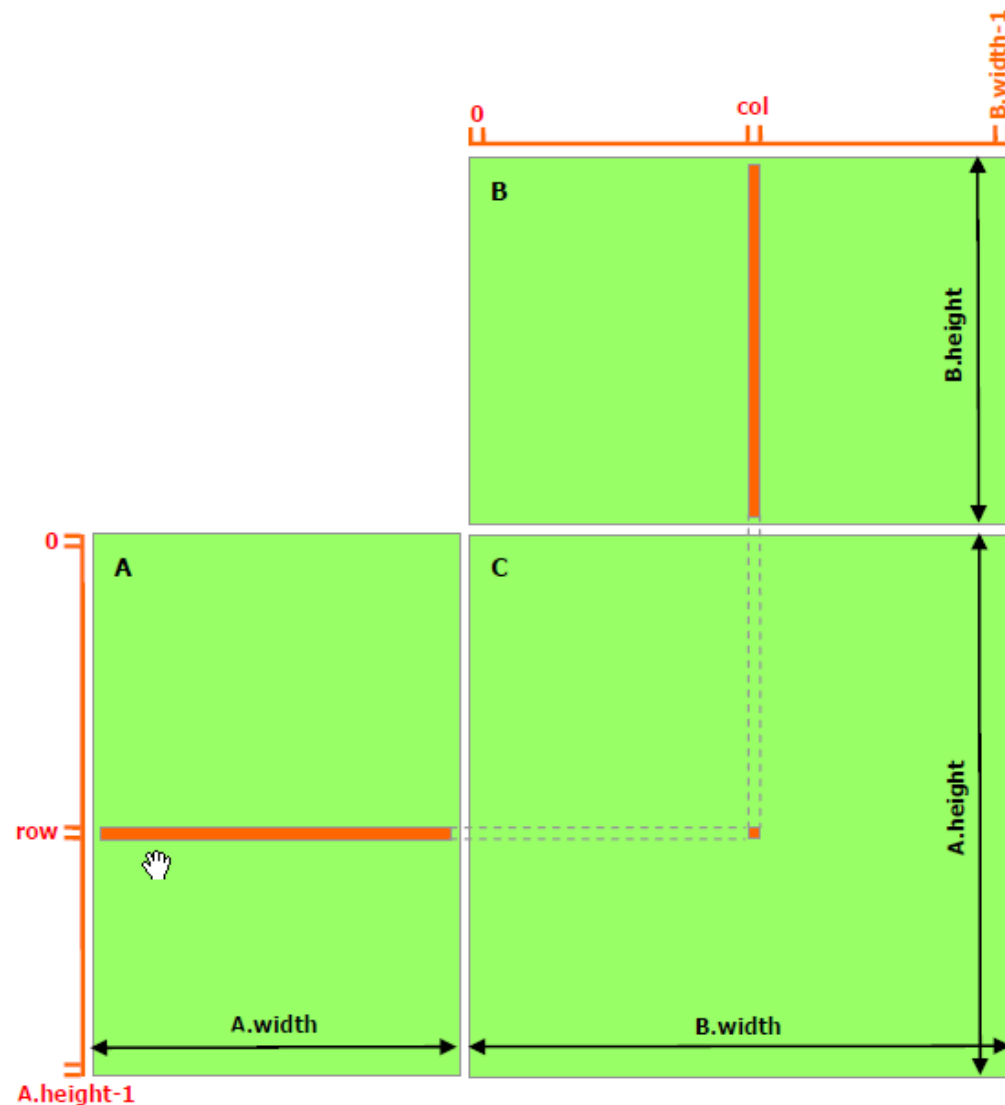
```
__device__ float devData;
__global__ void testStaticDev() {
    devData += 2.0f;
}
int main(int argc, char** argv) {
    float tmp = 2.0f;
    //从主机端向设备内存中复制数据
    cudaMemcpyToSymbol(devData, &tmp, sizeof(float));
    /*float * dptr = nullptr;
    cudaGetSymbolAddress((void*)&dptr, devData);
    cudaMemcpy(dptr, &tmp, sizeof(float), cudaMemcpyHostToDevice);*/
    testStaticDev<<<1, 1>>>();
    //从设备中复制到主机端
    cudaMemcpyFromSymbol(&tmp, devData, sizeof(float));
    printf("Data after changing in Device:%f \n", tmp);
    return 0;
}
```




- 共享内存是使用 `__shared__` 内存空间指定器分配的。
- 共享内存比全局内存要快得多.它可以作为scratchpad memory缓冲存储器（或软件管理的缓存）来使用，以尽量减少CUDA块的全局内存访问。
- 共享内存变量有以下特征：
 - 位于在一个线程块的共享内存空间中
 - 拥有该块的生命周期
 - 每个块有一个不同的对象
 - 只能由该块内的所有线程访问
 - 没有一个常数地址
- 当把共享内存中的变量声明为一个外部数组时，如 `extern __shared__ float shared[];` 数组的大小在cuda kernel启动时确定

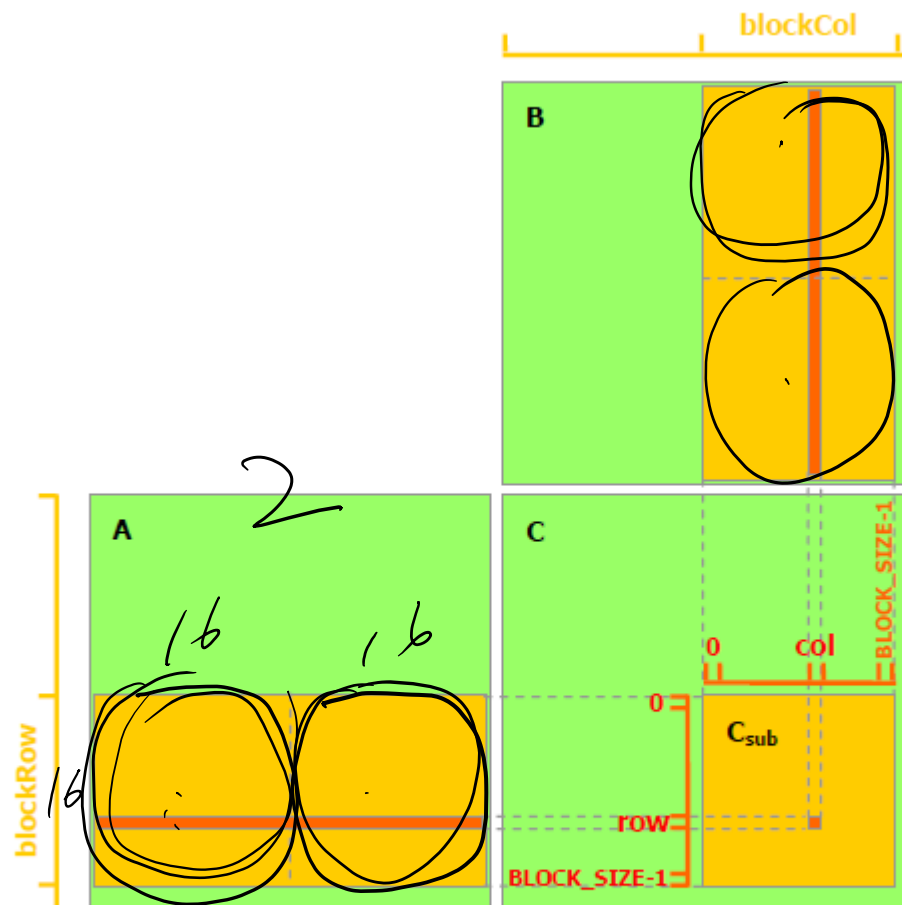


- 矩阵乘法: $C=A*B$ 。
- 不使用共享内存计算矩阵乘法，矩阵A的内容从全局内存中读取了B.width次，B被读取了A.height次





- 把矩阵C分成多个子矩阵Csub，每个线程块负责一个子矩阵的计算，线程块的每个线程则负责子矩阵中每个元素的计算.
- 每个Csub等于两个矩形矩阵的乘积：维度为 (A.width, block_size) 的A的子矩阵，其行指数与Csub相同；维度为 (block_size, A.width) 的B的子矩阵，其列指数与Csub相同。
- 运算时，A和B中子矩阵被分成尽可能多的block_size*block_size大小的正方形矩阵，并把他们加载到共享内存中。





- 运行时提供了允许使用锁页内存的函数（相对于由`malloc()`分配的普通可分页主机内存而言）：
 - `cudaHostAlloc()`和`cudaFreeHost()`分配和释放分页锁定的主机内存
 - `cudaHostRegister()`对`malloc()`分配的内存范围进行分页锁定
- 使用锁页内存有几个好处：
 - 在一些设备上，锁页内存和设备内存之间的拷贝可以与内核的执行同时进行，正如在异步并发执行中提到的
 - 在一些设备上，锁页内存可以被映射到设备的地址空间中，不需要从设备内存中复制它，这一点在Mapped Memory.中详细说明
 - 在有前端总线的系统上，如果主机内存被分配为锁页的，那么主机内存和设备内存之间的带宽会更高，如果再加上被分配为写组合内存，那么带宽甚至会更高。
- 但是需要注意锁页内存是一种稀缺资源，用多了会影响系统整体性能。



- `cudaHostAlloc`的函数原型如下：
 - `__host__ cudaError_t cudaHostAlloc (void** pHost, size_t size, unsigned int flags)`
- 最后一个参数可以为：
 - `cudaHostAllocDefault`: 默认参数，导致的行为使得`cudaHostAlloc`和`cudaMallocHost`变得一样.
 - `cudaHostAllocPortable`: 默认情况下锁页内存的好处只能在分配该内存的设备上起作用。如果要让所有设备都能使用该锁页内存，可以传递该标志，适用于有多个GPU的系统。
 - `cudaHostAllocWriteCombined`: 把内存分配为写组合的。写组合内存释放了主机的L1和L2缓存资源，使更多的缓存可用于应用程序的其他部分。此外，在PCI-E总线的传输过程中，可以提高传输性能达40%。从主机上读取写组合内存的速度非常慢，所以写组合内存一般应该被用于主机只写的内存
 - **`cudaHostAllocMapped`**: 把分配的地址映射到CUDA地址空间。此时，这块地址有两个地址：一个在主机内存中，由`cudaHostAlloc()`或`malloc()`返回，另一个在设备内存中，可以用`cudaHostGetDevicePointer()`检索，然后从内核中访问该块



- 从内核中直接访问主机内存并不能提供与设备内存相同的带宽，但确实有一些优势：
 - 不需要在设备内存中分配一个块，也不需要在这个块和主机内存中的块之间复制数据；数据传输是根据内核的需要隐式进行的
 - 不需要使用流来使数据传输与内核执行重叠；内核发起的数据传输自动与内核执行重叠。
- 然而，由于映射的页锁内存是在主机和设备之间共享的，应用程序必须使用流或事件来同步内存访问，以避免任何潜在的先读后写、先写后读或先写后写的危险性。

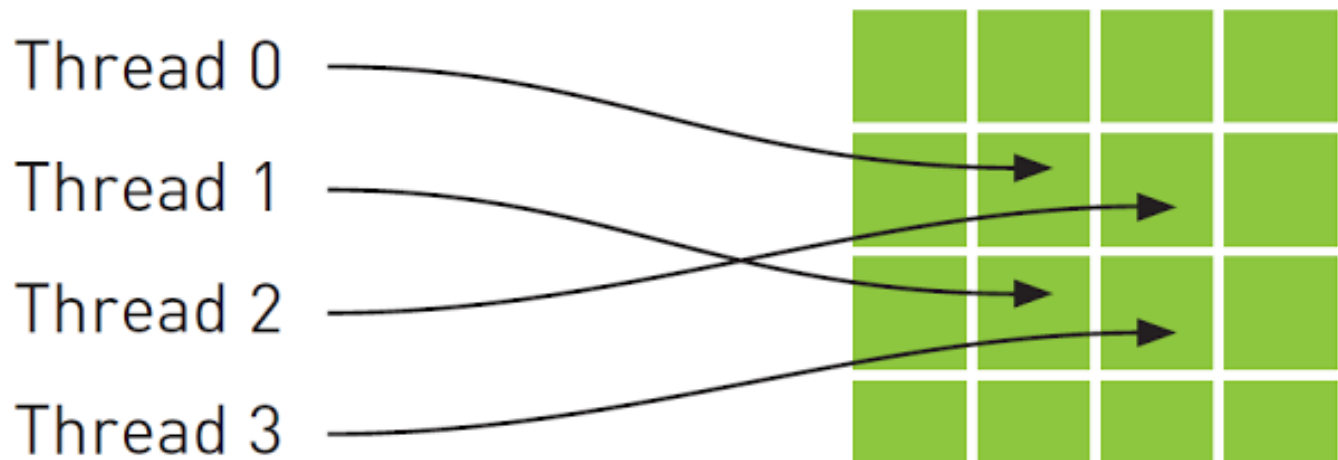


- `__constant__` 内存空间指定符，可以选择与 `__device__` 一起使用，声明一个变量：
 - 停留在恒定的内存空间中
 - 和创建它的CUDA上下文拥有相同的生命周期
 - 可以从网格内的所有线程以及主机通过运行时库(cudaGetSymbolAddress, cudaGetSymbolSize, cudaMemcpyToSymbol, cudaMemcpyFromSymbol) 访问
- 常量内存用于保存在核函数执行期间不会发生变化的数据，使用常量内存存在一些情况下，能有效减少内存带宽，降低GPU运算单元的空闲等待。
- 使用常量内存可以提升运算性能的原因如下：
 - 对常量内存的单次读操作可以广播到其他的“邻近(nearby)”线程，这将节约15次读取操作；
 - 高速缓存。常量内存的数据将缓存起来，因此对于相同地址的连续操作将不会产生额外的内存通信量
- 当处理常量内存时，NVIDIA硬件将把单次内存读取操作广播到每个半线程束(Half-Warp)。在半线程束中包含16个线程，即线程束中线程数量的一半。如果在半线程束中的每个线程从常量内存的相同地址上读取数据，那么GPU只会产生一次读取请求并在随后将数据广播到每个线程。如果从常量内存中读取大量数据，那么这种方式产生的内存流量只是使用全局内存时的1/16。

```
__constant__ float constData[256];  
float data[256];  
cudaMemcpyToSymbol(constData, data, sizeof(data));  
cudaMemcpyFromSymbol(data, constData, sizeof(data));
```



- 从硬件角度来说，纹理内存实际上是存储在全局内存上；但是，当某一个变量绑定纹理内存后，在运行时会将部分信息存储在纹理缓存中，以减少线程块对全局内存的读取，进而提高程序的运行速度。
- 如图所示生动的描述了Texture Memory的运行模式。观看下图可以发现Thread0~Thread3访问了一个图片上相邻区域的像素块。在常规模式下，分别需要在每个线程中访问全局内存来获取对应得像素值。不难发现存在许多重复访问，那么纹理内存就是将这些重复访问合并，减少不必要的访问步骤。





- 纹理内存的**核心是利用缓存来减少对全局内存的访问**，在CUDA中对其有一个专门的称呼：**纹理缓存**。可以从[文档](#)中查看不同计算能力GPU，纹理缓存大小
- 有两种不同的API来访问纹理和表面内存
 - 在所有设备上支持的**纹理引用API**。纹理引用的属性必须在代码编译前已知，作为全局变量使用，是不可更改的，并且不能作为参数使用
 - **纹理对象API**，只在计算能力为3.x及以上的设备上支持。纹理对象则可根据自己的需要在代码中灵活使用，可以作为核函数的参数传入，使用更加灵活



- 纹理对象或纹理引用指定了以下内容：
 - 纹理，也就是被取走的那块纹理内存。
 - 纹理对象是在运行时创建的，纹理是在创建纹理对象时指定的。
 - 纹理引用是在编译时创建的，纹理是在运行时指定的，通过运行时函数将纹理引用绑定到纹理上。
 - 纹理可以是线性内存或CUDA数组
 - 它的维度指定了纹理是使用一维、二维、三维数组。数组中的元素被称为texels，是纹理元素的简称。纹理的宽度、高度和深度是指阵列在每个维度上的大小。[Table 15](#)列出了取决于设备计算能力的最大纹理宽度、高度和深度。
 - texel的类型，**仅限于基本整数和单精度浮点类型，以及内置矢量类型中定义的任何1、2、4分量的矢量类型**，这些类型是由基本整数和单精度浮点类型派生的。
 - 读取模式，等于cudaReadModeNormalizedFloat或cudaReadModeElementType。
 - 如果是cudaReadModeNormalizedFloat，并且texel的类型是16位或8位的整数类型，那么纹理获取返回的值实际上是以浮点类型返回的，整数类型的全部范围被映射到[0.0, 1.0]的无符号整数类型和[-1.0, 1.0]的有符号整数类型；例如，值为0xff的无符号8位纹理元素读作1，
 - 如果是cudaReadModeElementType，则不进行转换。



•纹理对象或纹理引用指定了以下内容：

- 纹理坐标是否被归一化。默认情况下，纹理的引用是使用范围为 $[0, N-1]$ 的浮点坐标，其中 N 是纹理在坐标对应的尺寸。归一化的纹理坐标导致坐标被指定在 $[0.0, 1.0-1/N]$ 的范围内，而不是 $[0, N-1]$ 。
- 寻址模式。用超出范围的坐标调用B.8节的设备函数是有效的。寻址模式定义了在这种情况下会发生什么。
 - 默认的寻址模式`cudaAddressModeClamp`是将坐标夹在有效范围内。 $[0, N)$ 用于非标准化的坐标， $[0.0, 1.0)$ 用于标准化的坐标。如果指定了边界模式，那么纹理坐标超出范围的纹理获取将返回0
 - 对于归一化坐标，还可以使用`cudaAddressModeWrap`和`cudaAddressModeMirror`
 - `cudaAddressModeWrap`时，每个坐标 x 被转换为 $\text{frac}(x)=x - \text{floor}(x)$ ，其中 $\text{floor}(x)$ 是不大于 x 的最大整数
 - 当使用`cudaAddressModeMirror`时，如果 $\text{floor}(x)$ 是偶数，每个坐标 x 被转换为 $\text{frac}(x)$ ，如果 $\text{floor}(x)$ 是奇数，则为 $1-\text{frac}(x)$
 - 寻址模式被指定为一个大小为3的数组，其第一、第二和第三元素分别指定第一、第二和第三纹理坐标的寻址模式；寻址模式为`cudaAddressModeBorder`、`cudaAddressModeClamp`、`cudaAddressModeWrap`和`cudaAddressModeMirror`
- 滤波模式，它指定了获取纹理时返回的值是如何根据输入的纹理坐标计算的。线性纹理过滤只适用于被配置为返回浮点数据的纹理。它在相邻的texels之间进行低精度插值。当启用时，会读取纹理获取位置周围的纹理，并根据纹理坐标在纹理之间的位置对纹理获取的返回值进行插值。对一维纹理进行简单的线性内插，对二维纹理进行双线性内插，对三维纹理进行三线性内插。纹理获取给出了关于纹理获取的更多细节。**过滤模式等于`cudaFilterModePoint`或`cudaFilterModeLinear`。如果是`cudaFilterModePoint`，返回值是其纹理坐标最接近输入纹理坐标的文本。如果是`cudaFilterModeLinear`，返回值是纹理坐标最接近输入纹理坐标的两个（对于一维纹理）、四个（对于二维纹理）或八个（对于三维纹理）纹理的线性内插。**



- 纹理对象是使用 `cudaCreateTextureObject()` 从类型为 `struct cudaResourceDesc` 的资源描述中创建的，该描述指定了纹理，对应结构体如下：

- `addressMode` 指定寻址模式
- `filterMode` 指定了过滤模式
- `readMode` 指定的是读取模式
- `normalizedCoords` 指定纹理坐标是否被归一化。

```
struct cudaTextureDesc {  
    enum cudaTextureAddressMode addressMode[3];  
    enum cudaTextureFilterMode filterMode;  
    enum cudaTextureReadMode readMode;  
    int sRGB;  
    int normalizedCoords;  
    unsigned int maxAnisotropy;  
    enum cudaTextureFilterMode mipmapFilterMode;  
    float mipmapLevelBias;  
    float minMipmapLevelClamp;  
    float maxMipmapLevelClamp;  
};
```

纹理对象

```
// Simple transformation kernel
__global__ void transformKernel(float* output,
    cudaTextureObject_t texObj, int width, int height, float
    theta) {
    // Calculate normalized texture coordinates
    unsigned int x = blockIdx.x * blockDim.x + threadIdx.x;
    unsigned int y = blockIdx.y * blockDim.y + threadIdx.y;
    float u = x / (float)width;
    float v = y / (float)height;
    // Transform coordinates
    u -= 0.5f;
    v -= 0.5f;
    float tu = u * cosf(theta) - v * sinf(theta) + 0.5f;
    float tv = v * cosf(theta) + u * sinf(theta) + 0.5f;
    // Read from texture and write to global memory
    output[y * width + x] = tex2D<float>(texObj, tu, tv);
}
```

```
int main() {
    cudaChannelFormatDesc channelDesc =
        cudaCreateChannelDesc(32, 0, 0, 0, cudaChannelFormatKindFloat);
    cudaArray* cuArray;
    cudaMallocArray(&cuArray, &channelDesc, width, height);
    cudaMemcpyToArray(cuArray, 0, 0, h_data, size, cudaMemcpyHostToDevice);
    // Specify texture
    struct cudaResourceDesc resDesc;
    memset(&resDesc, 0, sizeof(resDesc));
    resDesc.resType = cudaResourceTypeArray;
    resDesc.res.array.array = cuArray;
    // Specify texture object parameters
    struct cudaTextureDesc texDesc;
    memset(&texDesc, 0, sizeof(texDesc));
    texDesc.addressMode[0] = cudaAddressModeWrap;
    texDesc.addressMode[1] = cudaAddressModeWrap;
    texDesc.filterMode = cudaFilterModeLinear;
    texDesc.readMode = cudaReadModeElementType;
    texDesc.normalizedCoords = 1;
    // Create texture object
    cudaTextureObject_t texObj = 0;
    cudaCreateTextureObject(&texObj, &resDesc, &texDesc, NULL);
    // Allocate result of transformation in device memory
    float* output;
    cudaMalloc(&output, width * height * sizeof(float));
    // Invoke kernel
    dim3 dimBlock(16, 16);
    dim3 dimGrid((width + dimBlock.x - 1) / dimBlock.x,
        (height + dimBlock.y - 1) / dimBlock.y);
    transformKernel<<<dimGrid, dimBlock>>>(output, texObj, width, height, angle);
    // Destroy texture object
    cudaDestroyTextureObject(texObj);
    // Free device memory
    cudaFreeArray(cuArray);
    cudaFree(output);
    return 0;}
```



- 纹理引用的一些属性是不可改变的，必须在编译时知道；它们在声明纹理引用时被指定。纹理引用在文件范围内被声明为一个类型为**texture** 的变量。
 - `texture<DataType, Type, ReadMode> texRef;`
- 其中：
 - `DataType`指定了texel的类型
 - `Type`指定了纹理引用的类型，并且等于`cudaTextureType1D`、`cudaTextureType2D`或`cudaTextureType3D`，分别用于一维、二维或三维纹理，或者`cudaTextureType1DLayered`或`cudaTextureType2DLayered`，分别用于一维或二维分层纹理；`Type`是一个可选参数，默认为`cudaTextureType1D`
 - `ReadMode`指定读取模式；它是一个可选的参数，默认为`cudaReadModeElementType`
- 纹理引用只能被声明为一个静态的全局变量，不能作为参数传递给一个函数。



- 在内核使用纹理引用读取纹理内存之前，必须使用**cudaBindTexture()**或**cudaBindTexture2D()**将纹理参考绑定到线性内存上，或者使用**cudaBindTextureToArray()**将纹理参考绑定到CUDA数组上。**cudaUnbindTexture()**用于解除纹理参考的绑定。一旦纹理引用被解除绑定，它就可以安全地被重新绑定到另一个数组中，即使使用先前绑定的纹理的内核还没有完成。建议使用**cudaMallocPitch()**在线性内存中分配二维纹理，并使用**cudaMallocPitch()**返回的间距作为**cudaBindTexture2D()**的输入参数
- 下面的代码样本将一个二维纹理参考绑定到devPtr所指向的线性内存中：

```
texture<float, cudaTextureType2D, cudaReadModeElementType> texRef;  
cudaChannelFormatDesc channelDesc = cudaCreateChannelDesc<float>();  
size_t offset;  
cudaBindTexture2D(&offset, texRef, devPtr, channelDesc, width, height, pitch);
```

- 下面的代码示例将一个2D纹理引用与CUDA数组cuArray绑定

```
texture<float, cudaTextureType2D, cudaReadModeElementType> texRef;  
cudaBindTextureToArray(texRef, cuArray);
```




```
// 2D float texture
texture<float, cudaTextureType2D, cudaReadModeElementType>
texRef;
// Simple transformation kernel
__global__ void transformKernel(float* output, int width,
int height, float theta) {
    // Calculate normalized texture coordinates
    unsigned int x = blockIdx.x * blockDim.x + threadIdx.x;
    unsigned int y = blockIdx.y * blockDim.y + threadIdx.y;
    float u = x / (float)width;
    float v = y / (float)height;
    // Transform coordinates
    u -= 0.5f;
    v -= 0.5f;
    float tu = u * cosf(theta) - v * sinf(theta) + 0.5f;
    float tv = v * cosf(theta) + u * sinf(theta) + 0.5f;
    // Read from texture and write to global memory
    output[y * width + x] = tex2D(texRef, tu, tv);
}
```

```
int main() {
    cudaChannelFormatDesc channelDesc =
        cudaCreateChannelDesc(32, 0, 0, 0, cudaChannelFormatKindFloat);
    cudaArray* cuArray;
    cudaMallocArray(&cuArray, &channelDesc, width, height);
    // Copy to device memory some data located at address h_data
    cudaMemcpyToArray(cuArray, 0, 0, h_data, size, cudaMemcpyHostToDevice);
    // Set texture reference parameters
    texRef.addressMode[0] = cudaAddressModeWrap;
    texRef.addressMode[1] = cudaAddressModeWrap;
    texRef.filterMode = cudaFilterModeLinear;
    texRef.normalized = true;
    // Bind the array to the texture reference
    cudaBindTextureToArray(texRef, cuArray, channelDesc);
    // Allocate result of transformation in device memory
    float* output;
    cudaMalloc(&output, width * height * sizeof(float));
    // Invoke kernel
    dim3 dimBlock(16, 16);
    dim3 dimGrid((width + dimBlock.x - 1) / dimBlock.x,
        (height + dimBlock.y - 1) / dimBlock.y);
    transformKernel<<<dimGrid, dimBlock>>>(output, width, height, angle);
    // Free device memory
    cudaFreeArray(cuArray); cudaFree(output);
    return 0;
}
```