CUDA架构

CUDA是一种新的操作GPU计算的硬件和软件架构，它将GPU视作一个数据并行计算设备，而且无需把这些计算映射到图形API。

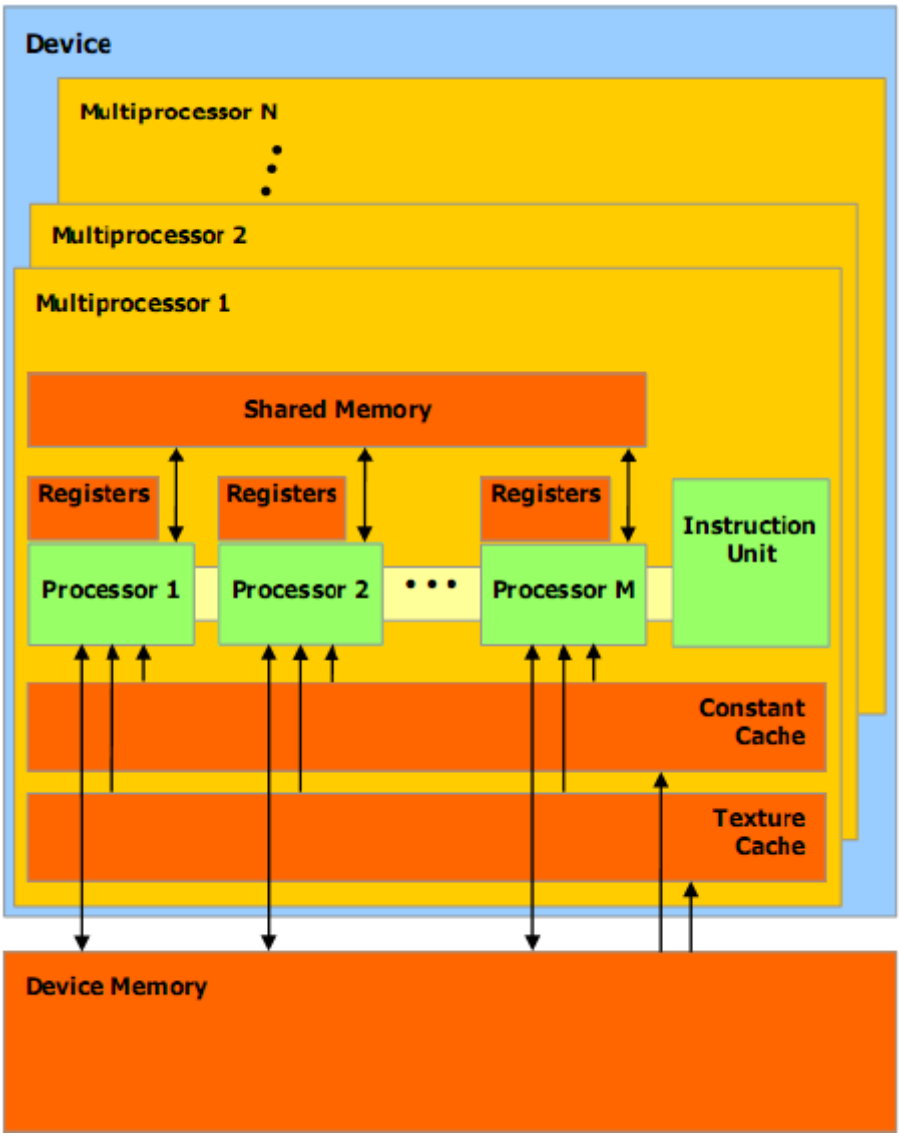
1.1 GPU困境

虽然GPU通过图形应用程序的算法存在如下几个特征：算法密集、高度并行、控制简单、分多个阶段执行以及前馈(Feed
Forward)流水线等，能够在高度密集型的并行计算上获得较高的性能和速度，但在2007年以前GPU要实现这样的应用还是存在许多困难的：

1. GPU只能通过一个图形的API来编程，这不仅加重了学习负担更造成那些非图像应用程序处理这些 API 的额外开销。
2. 由于DRAM内存带宽，一些程序会遇到瓶颈。
3. 无法在 DRAM 上进行通用写操作。
   所以NVIDIA于2006年11月在G80系列中引入的Tesla统一图形和计算架构扩展了GPU，使其超越了图形领域。通过扩展处理器和存储器分区的数量，其强大的多线程处理器阵列已经成为高效的统一计算平台，同时适用于图形和通用并行计算应用程序。从G80系列开始NVIDIA加入了对CUDA的支持。

1.2 芯片结构

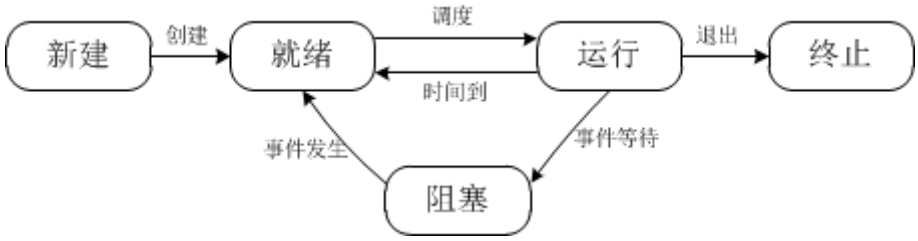
具有Tesla架构的GPU是具有芯片共享存储器的一组SIMT（单指令多线程）多处理器。它以一个可伸缩的**多线程流处理器**（Streaming Multiprocessors，SMs）阵列为中心实现了MIMD（多指令多数据）的异步并行机制，其中每个多处理器包含多个**标量处理器**（Scalar Processor，SP），为了管理运行各种不同程序的数百个线程，SIMT架构的多处理器会将各线程映射到一个标量处理器核心，各标量线程使用自己的指令地址和寄存器状态独立执行。



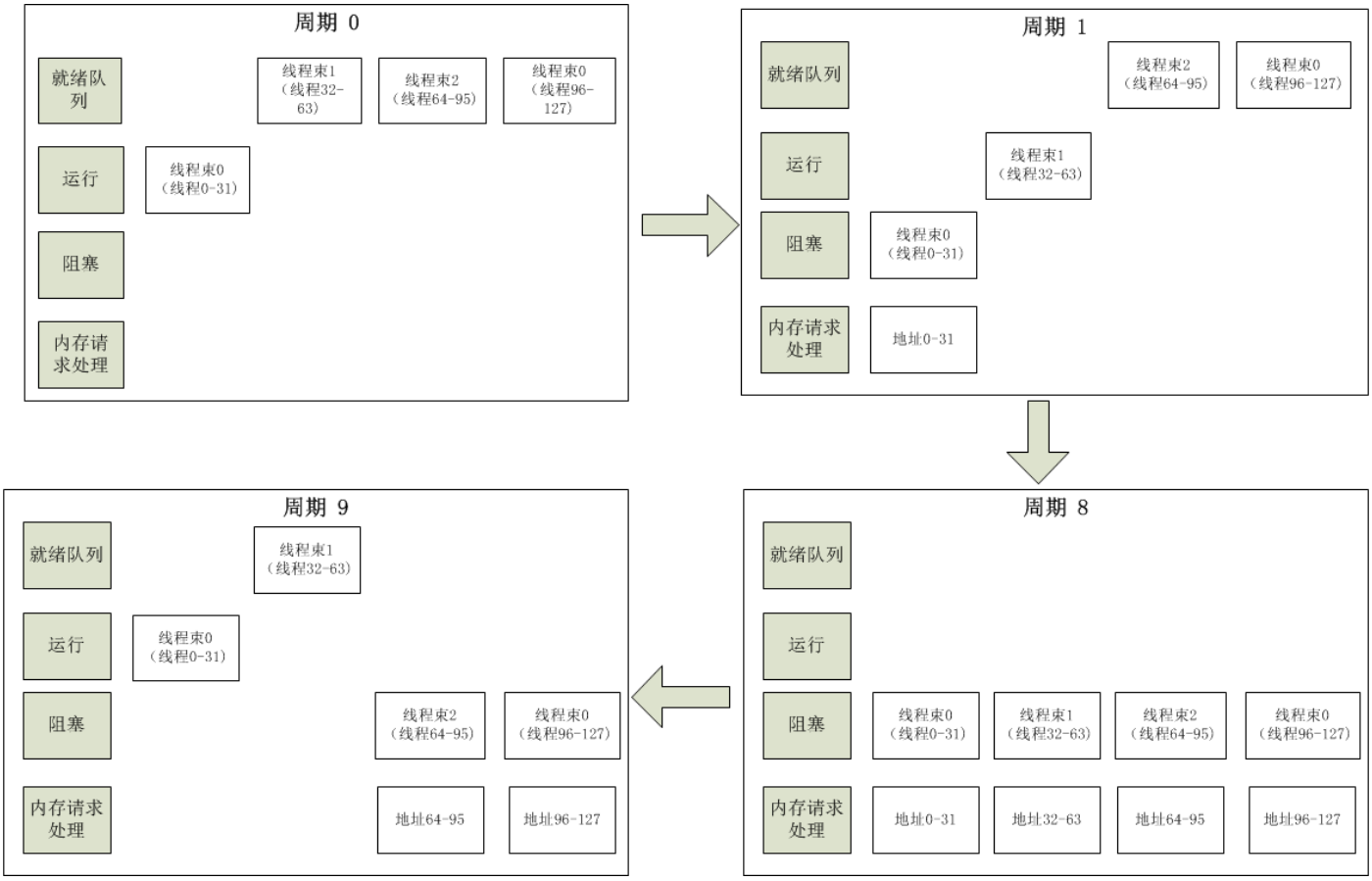
GPU的共享存储器的SIMT多处理器模型.png

如上图所示，每个多处理器(Multiprocessor)都有一个属于以下四种类型之一的芯片存储器：

- 每个处理器上有一组本地 32 位寄存器（**Registers**）；
- 并行数据缓存或共享存储器(**Shared Memory**)，由所有标量处理器核心共享，共享存储器空间就位于此处；
- 只读固定缓存(**Constant Cache**)，由所有标量处理器核心共享，可加速从固定存储器空间进行的读取操作（这是设备存储器的一个只读区域）；
- 一个只读纹理缓存(**Texture Cache**)，由所有标量处理器核心共享，加速从纹理存储器空间进行的读取操作（这是设备存储器的一个只读区域），每个多处理器都会通过实现不同寻址模型和数据过滤的纹理单元访问纹理缓存。
  多处理器 SIMT单元以32个并行线程为一组来创建、管理、调度和执行线程，这样的线程组称为 **warp 块(束)**，即以线程束为调度单位，但只有所有32个线程都在诸如内存读取这样的操作时，它们就会被挂起，如图7所示的状态变化。当主机CPU上的CUDA程序调用内核网格时，网格的块将被枚举并分发到具有可用执行容量的多处理器；SIMT单元会选择一个已准备好执行的 warp 块，并将下一条指令发送到该 warp块的活动线程。一个线程块的线程在一个多处理器上并发执行，在线程块终止时，将在空闲多处理器上启动新块。

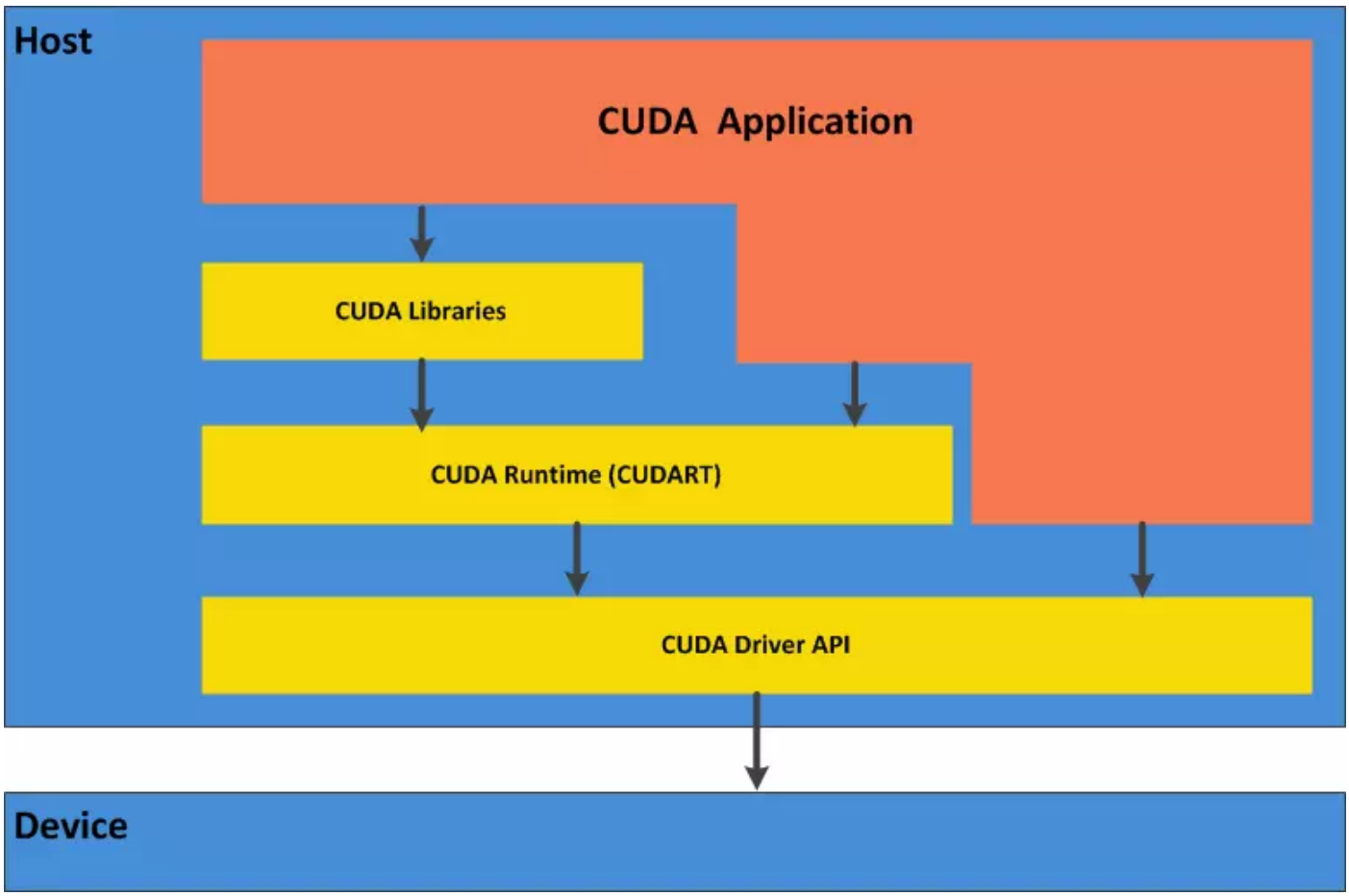


CPU五种状态的转换.png



线程束调度变化.png

CUDA是一种新的操作GPU计算的硬件和软件架构，它将GPU视作一个数据并行计算设备，而且无需把这些计算映射到图形API。操作系统的多任务机制可以同时管理CUDA访问GPU和图形程序的运行库，其计算特性支持利用CUDA直观地编写GPU核心程序。目前Tesla架构具有在笔记本电脑、台式机、工作站和服务器上的广泛可用性，配以C/C++语言的编程环境和CUDA软件，使这种架构得以成为最优秀的超级计算平台。



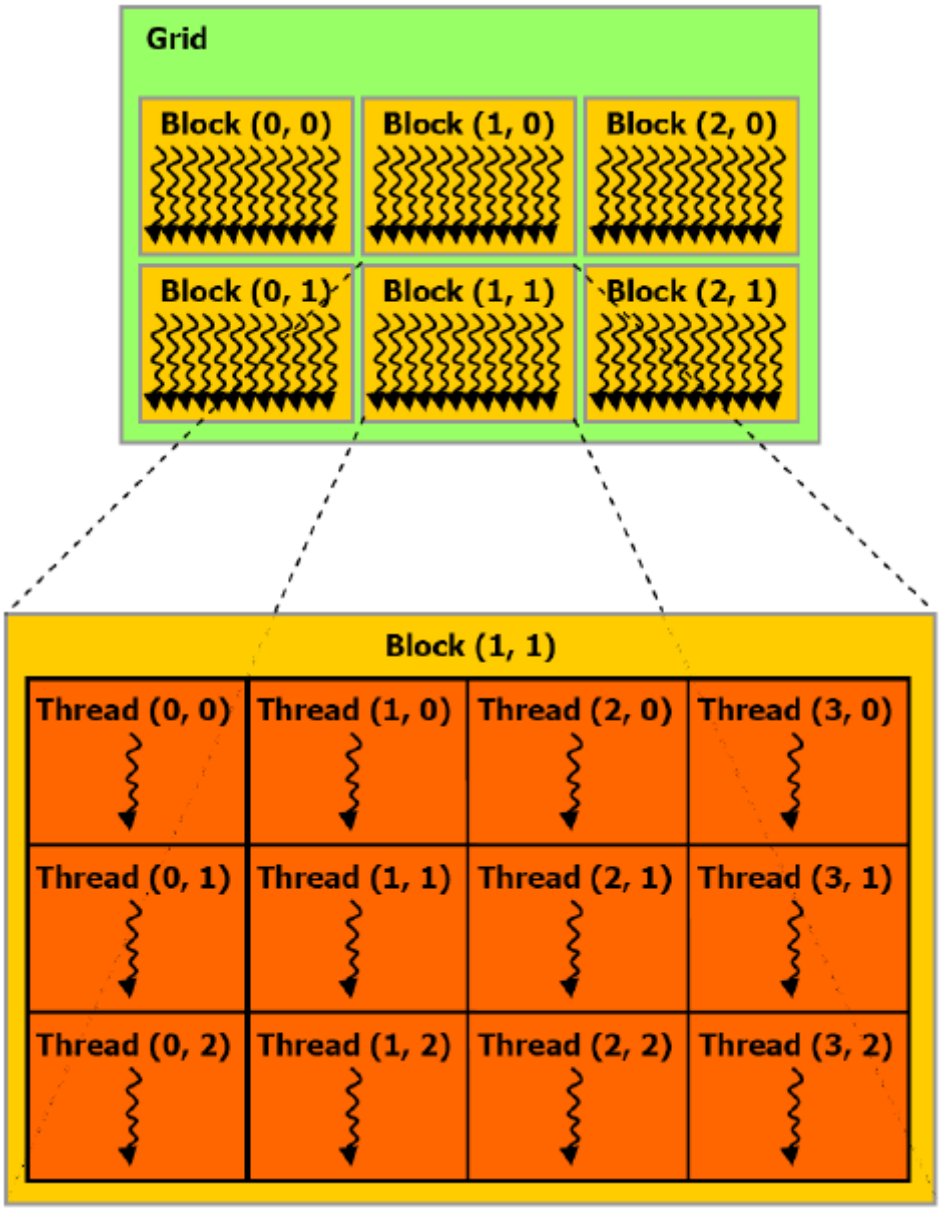
CUDA软件层次结构.png

CUDA在软件方面组成有：一个CUDA库、一个应用程序编程接口（API）及其运行库(Runtime)、两个较高级别的通用数学库，即CUFFT和CUBLAS。CUDA改进了DRAM的读写灵活性，使得GPU与CPU的机制相吻合。另一方面，CUDA提供了片上（on-chip）共享内存，使得线程之间可以共享数据。应用程序可以利用共享内存来减少DRAM的数据传送，更少的依赖DRAM的内存带宽。

CUDA程序构架分为两部分：Host和Device。一般而言，Host指的是CPU，Device指的是GPU。在CUDA程序构架中，主程序还是由CPU 来执行，而当遇到数据并行处理的部分，CUDA 就会将程序编译成 GPU能执行的程序，并传送到GPU。而这个程序在CUDA里称做核（kernel）。CUDA允许程序员定义称为核的C语言函数，从而扩展了C语言，在调用此类函数时，它将由N个不同的CUDA线程并行执行N次，这与普通的C语言函数只执行一次的方式不同。执行核的每个线程都会被分配一个独特的线程ID，可通过内置的threadIdx变量在内核中访问此ID。在 CUDA 程序中，主程序在调用任何 GPU内核之前，必须对核进行执行配置，即确定线程块数和每个线程块中的线程数以及共享内存大小。

3.1 线程层次结构

在GPU中要执行的线程，根据最有效的数据共享来创建块(Block)，其类型有一维、二维或三维。在同一个块内的线程可彼此协作，通过一些共享存储器来共享数据，并同步其执行来协调存储器访问。**一个块中的所有线程都必须位于同一个处理器核心中**。因而，一个处理器核心的有限存储器资源制约了每个块的线程数量。在早起的NVIDIA 架构中，一个线程块最多可以包含 512个线程，而在后期出现的一些设备中则最多可支持1024个线程。一般 GPGPU程序线程数目是很多的，所以不能把所有的线程都塞到同一个块里。但一个内核可由多个大小相同的线程块同时执行，因而线程总数应等于每个块的线程数乘以块的数量。这些同样维度和大小的块将组织为一个一维或二维线程块网格(Grid)。具体框架如图9所示。



线程块网格.png

核函数只能在主机端调用，其调用形式为：**Kernel<<<Dg,Db, Ns, S>>>(paramlist)**

- Dg：用于定义整个grid的维度和尺寸，即一个grid有多少个**block**。为dim3类型。Dim3 Dg(Dg.x, Dg.y,
  1)表示grid中每行有Dg.x个block，每列有Dg.y个block，第三维恒为1(目前一个核函数只有一个grid)。整个grid中共有Dg.x*Dg.y个block，其中Dg.x和Dg.y最大值为65535。
- Db：用于定义一个block的维度和尺寸，即一个block有多少个**thread**。为dim3类型。Dim3 Db(Db.x, Db.y,
  Db.z)表示整个block中每行有Db.x个thread，每列有Db.y个thread，高度为Db.z。Db.x和Db.y最大值为512，Db.z最大值为62。
  一个block中共有Db.x*Db.y*Db.z个thread。计算能力为1.0,1.1的硬件该乘积的最大值为768，计算能力为1.2,1.3的硬件支持的最大值为1024。
- Ns：是一个可选参数，用于设置每个block除了静态分配的shared Memory以外，最多能动态分配的shared
  memory大小，单位为byte。不需要动态分配时该值为0或省略不写。

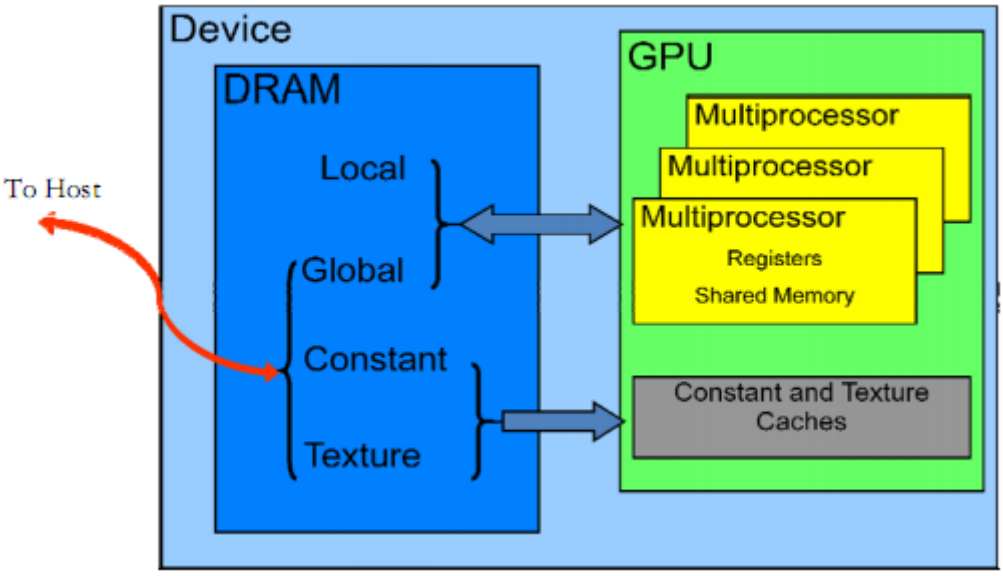
- S：是一个cudaStream_t类型的可选参数，初始值为零，表示该核函数处在哪个流之中。
  如下是一个CUDA简单的求和程序：

```
#define N 10
int main(void ){
    int a[N],b[N],c[N];
    int *dv_a,*dev_b,*dev_c;
     //创建GPU内存
    HANDLE_ERROR( cudaMalloc( (void**)&dev_a, N*sizeof(int) ) );
    HANDLE_ERROR( cudaMalloc( (void**)&dev_b, N*sizeof(int) ) );
    HANDLE_ERROR( cudaMalloc( (void**)&dev_c, N*sizeof(int) ) );

    for(int i=0; i<N; i++){
            a[i]=-i;
            b[i]=i*i;
     }

     //从主机空间的数据复制到设备上
     HANDLE_ERROR( cudaMemcpy (dev_a, a, N*sizeof(int), cudaMemcpyHostToDevice ) );
     HANDLE_ERROR( cudaMemcpy (dev_b, b, N*sizeof(int), cudaMemcpyHostToDevice ) );
     HANDLE_ERROR( cudaMemcpy (dev_c, c, N*sizeof(int), cudaMemcpyHostToDevice ) );

     add<<N, 1>> (dev_a, dev_b, dev_c); //调用处理函数，并声明N个线程block，每个block有1个线程。

     HANDLE_ERROR( cudaMemcpy( c, dev_c, N*sizeof(int) cudaMemcpyDeviceHost ) );
     for(int i=0; i<N; i++)
            printf("%d+%d =%d\n",a[i],b[i],c[i]);

     cudaFree(dev_a);
     cudaFree(dev_b);
     cudaFree(dev_c);

     return 0;
}
_global_ void add( int *a, int *b, int *c){
            int tid = blockIdx.x; //blockId是CUDA一个关键字
            if(tid<N)
                    c[tid] = a[tid] + b[tid];
}
```

CUDA求和程序.png

3.2 存储器层次结构

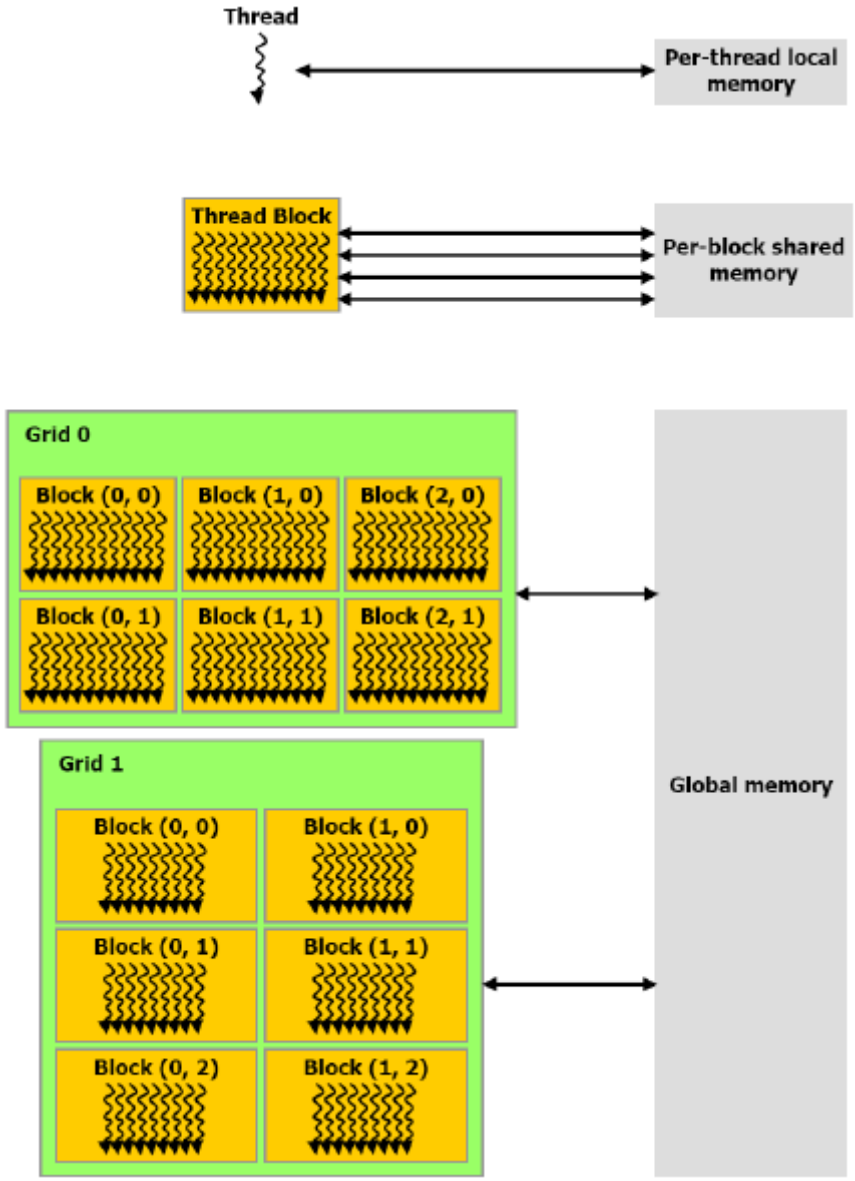
CUDA设备拥有多个独立的存储空间，其中包括：全局存储器、本地存储器、共享存储器、常量存储器、纹理存储器和寄存器，如图
11所示。



CUDA设备上的存储器.png

CUDA线程可在执行过程中访问多个存储器空间的数据，如图 12所示其中：

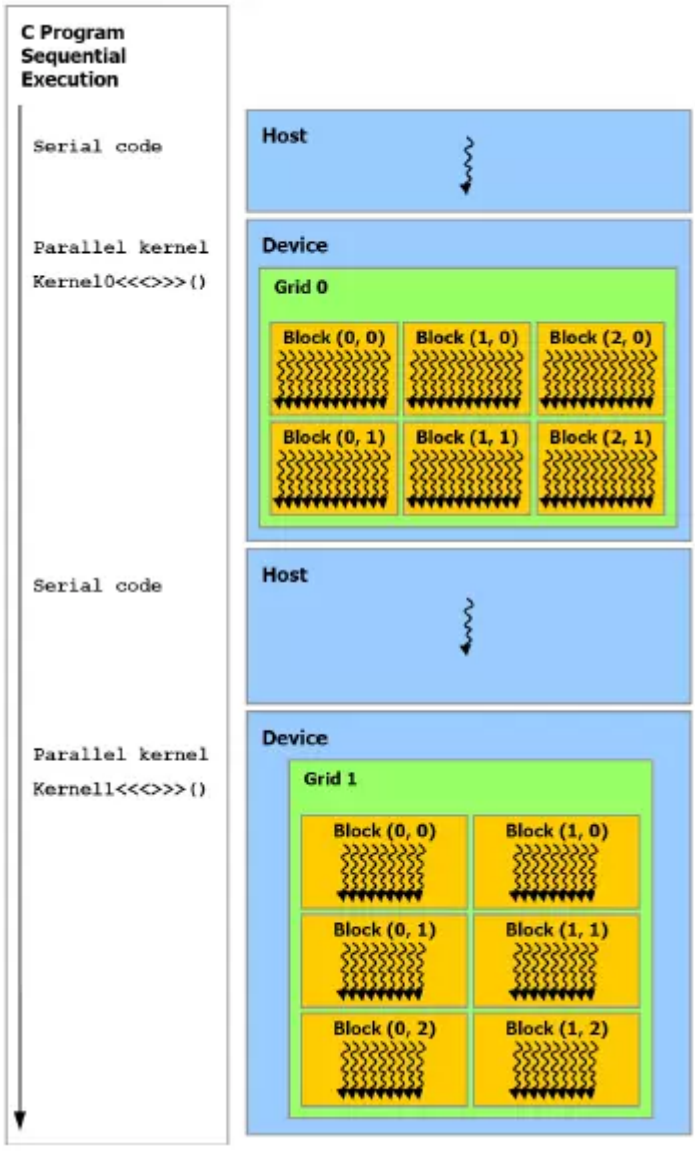
- 每个线程都有一个私有的**本地存储器**。
- 每个线程块都有一个**共享存储器**，该存储器对于块内的所有线程都是可见的，并且与块具有相同的生命周期。
- 所有线程都可访问相同的**全局存储器**。
- 此外还有两个只读的存储器空间，可由所有线程访问，这两个空间是**常量存储器**空间和**纹理存储器**空间。全局、固定和纹理存储器空间经过优化，适于不同的存储器用途。纹理存储器也为某些特殊的数据格式提供了不同的寻址模式以及数据过滤，方便
  Host对流数据的快速存取。



存储器的应用层次.png

3.3 主机（Host）和设备（Device）

如图 13所示，CUDA假设线程可在物理上独立的设备上执行，此类设备作为运行C语言程序的主机的协处理器操作。内核在GPU上执行，而C语言程序的其他部分在CPU上执行（即串行代码在主机上执行，而并行代码在设备上执行）。此外，CUDA还假设主机和设备均维护自己的DRAM，分别称为主机存储器和设备存储器。因而，一个程序通过调用CUDA运行库来管理对内核可见的全局、固定和纹理存储器空间。这种管理包括设备存储器的分配和取消分配，还包括主机和设备存储器之间的数据传输。



CUDA异构编程模型.png

4. CUDA软硬件

4.1 CUDA术语

> 由于CUDA中存在许多概念和术语，诸如SM、block、SP等多个概念不容易理解，将其与CPU的一些概念进行比较，如下表所示。

CPU	GPU	层次
---	---	---
算术逻辑和控制单元	流处理器(SM)	硬件
算术单元	批量处理器(SP)	硬件
进程	Block	软件
线程	thread	软件

调度单位	Warp	软件

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

Detected 1 CUDA Capable device(s)

Device 0: "GeForce GT 630"
  CUDA Driver Version / Runtime Version          7.0 / 7.0
  CUDA Capability Major/Minor version number:    2.1
  Total amount of global memory:                 2047 MBytes (2146631680 bytes)
  ( 2) Multiprocessors, ( 48) CUDA Cores/MP:     96 CUDA Cores
  GPU Max Clock rate:                            1400 MHz (1.40 GHz)
  Memory Clock rate:                             667 Mhz
  Memory Bus Width:                              128-bit
  L2 Cache Size:                                 131072 bytes
  Maximum Texture Dimension Size (x,y,z)         1D=(65536), 2D=(65536, 65535), 3D=(2048, 2048, 2048)
  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: 32768
  Warp size:                                     32
  Maximum number of threads per multiprocessor:  1536
  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): (65535, 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:                     No
  Integrated GPU sharing Host Memory:            No
  Support host page-locked memory mapping:       Yes
  Alignment requirement for Surfaces:            Yes
  Device has ECC support:                        Disabled
  Device supports Unified Addressing (UVA):      Yes
  Device PCI Domain ID / Bus ID / location ID:   0 / 1 / 0
  Compute Mode:
     < Default (multiple host threads can use ::cudaSetDevice() with device simultaneously) >

deviceQuery, CUDA Driver = CUDART, CUDA Driver Version = 7.0, CUDA Runtime Version = 7.0, NumDevs = 1, Device0 = GeForce GT 630
Result = PASS
```

NVIDIA 630显卡CUDA信息.png

4.2 硬件利用率

当为一个GPU分配一个内核函数，我们关心的是如何才能充分利用GPU的计算能力，但由于不同的硬件有不同的计算能力，SM一次最多能容纳的线程数也不尽相同，SM一次最多能容纳的线程数量主要与底层硬件的计算能力有关，如下表显示了在不同的计算能力的设备上，每个线程块上开启不同数量的线程时设备的利用率。

计算能力 每个线 程块的线程数	1.0	1.1	1.2	1.3	2.0	2.1	3.0
---	---	---	---	---	---	---	---
64	67	67	50	50	33	33	50
96	100	100	75	75	50	50	75
128	100	100	100	100	67	67	100
192	100	100	94	94	100	100	94
256	100	100	100	100	100	100	100

……	……						