CUDA 笔记

郑华

2018 年 5 月 13 日

目录

第一章　入门

1.1　参考

http://blog.chinaunix.net/uid-20620288-id-4705367.html

http://bbs.csdn.net/topics/390798229

1.2　CUDA 简介

CUDA，Compute Unified Device Architecture 的简称，是由NVIDIA 公司创立的基于他们公司生产的图形处理器GPUs（Graphics Processing Units, 可以通俗的理解为显卡）的一个并行计算平台和编程模型。通过CUDA，GPUs 可以很方便地被用来进行通用计算（有点像在CPU 中进行的数值计算等等）。在没有CUDA 之前，GPUs 一般只用来进行图形渲染（如通过OpenGL，DirectX）。

开发人员可以通过调用CUDA 的API，来进行并行编程，达到高性能计算目的。NVIDIA 公司为了吸引更多的开发人员，对CUDA 进行了编程语言扩展，如CUDA C/C++，CUDA Fortran 语言。注意CUDA C/C++ 可以看作一个新的编程语言，因为NVIDIA 配置了相应的编译器nvcc

1.3　基本概念

1.3.1　主机

将CPU 及系统的内存（内存条）称为主机

1.3.2 设备

将GPU 及GPU 本身的显示内存称为设备, 设备中分为大核和小核，流多处理器，大核是也，小核则是线程并行单元流处理器。

1.3.3 线程

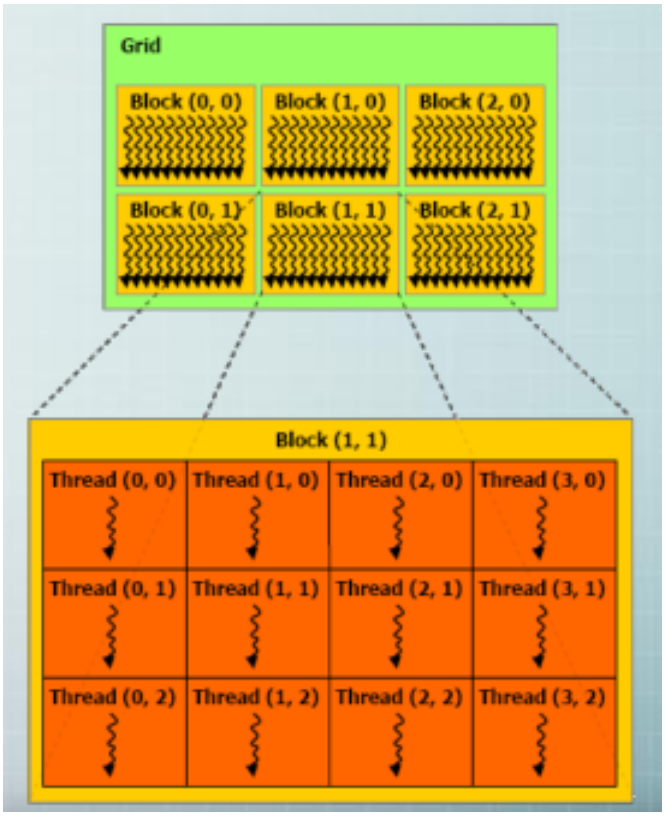
一般通过 GPU 的一个核进行处理。(可以表示成一维，二维，三维)

1.3.4 线程块

1. 由多个线程组成
2. 各block 是并行执行的，block 间无法通信，也没有执行顺序
3. 注意线程块的数量限制为不超过65535（硬件限制）。

1.3.5 线程格

由多个线程块组成.



1.3.6 线结束

在CUDA 架构中，线程束是指一个包含 32 个线程的集合，这个线程集合被“编织在一起”并且“步调一致”的形式执行。在程序中的每一行，线程束中的每个线程都将在不同数据上执行相同的命令。

1.3.7 函数修饰符

1. `__global__` 表明被修饰的函数**在设备上执行，但在主机上调用**
2. `__device__` 表明被修饰的函数**在设备上执行**,但只能在其他`__device__` 函数或者`__global__` 函数中调用

1.3.8 核函数

1. 在 GPU 上执行的函数通常称为核函数
2. 一般通过标识符`__global__` 修饰，调用通过`<<<参数1,参数2>>>`，用于说明内核函数中的线程数量，以及线程是如何组织的。
3. 以线程格（`Grid`）的形式组织，每个线程格由若干个线程块（`block`）组成，而每个线程块又由若干个线程（`thread`）组成
4. 是以`block` 为单位执行的
5. 另能在主机端代码中调用
6. 调用时必须声明内核函数的执行参数
7. 在编程时，必须先为 kernel 函数中用到的数组或变量分配好足够的空间，再调用 kernel 函数，否则在 GPU 计算时会发生错误，例如越界或报错，甚至导致蓝屏和死机

```
/*
* @file_name HelloWorld.cu  后缀名称.cu
*/

#include <stdio.h>
#include <cuda_runtime.h>  //头文件

//核函数声明，前面的关键字__global__
__global__ void kernel( void ) {}
```

```
int main( void ) {
    //核函数的调用，注意<<<1,1>>>，第一个1，代表线程格里只有一个线程块；第二个1，代表一个
        线程块里只有一个线程。
    kernel<<<1,1>>>();
    printf( "Hello, World!\n" );
    return 0;
}
```

1.3.9 dim3 结构类型

1. dim3 是基于uint3 定义的矢量类型，相当于由 3 个unsigned int 型组成的结构体。uint3 类型有三个数据成员unsigned int x; unsigned int y; unsigned int z;
2. 可使用于一维、二维或三维的索引来标识线程，构成一维、二维或三维线程块。
3. dim3 结构类型变量用在核函数调用的<<<,>>> 中
4. 相关的几个内置变量
   - threadIdx,顾名思义获取线程thread 的ID 索引;如果线程是一维的那么就取 threadIdx.x，二维的还可以多取到一个值 threadIdx.y，以此类推到三维 threadIdx.z
   - blockIdx，线程块的ID 索引；同样有 blockIdx.x，blockIdx.y，blockIdx.z
   - blockDim，线程块的维度，同样有 blockDim.x，blockDim.y，blockDim.z
   - gridDim，线程格的维度，同样有 gridDim.x，gridDim.y，gridDim.z
5. 对于一维的block，线程的 threadID = threadIdx.x
6. 对于大小为(blockDim.x, blockDim.y)的 二维 block,线程的 threadID = threadIdx.x + threadIdx.y*blockDim.x
7. 对于大小为(blockDim.x, blockDim.y, blockDim.z)的 三维 block,线程的 threadID = threadIdx.x+threadIdx.y*blockDim.x+threadIdx.z*blockDim.x*blockDim.y
8. 对于计算线程索引偏移增量为已启动线程的总数。如 stride = blockDim.x * gridDim.x; threadId += stride

第二章　进阶

2.1　常用的 GPU 函数

2.1.1　cudaMalloc()

函数原型　`cudaError_t cudaMalloc (void **devPtr, size_t size)`

函数用处　与 C 语言中的`malloc` 函数一样，只是此函数在 GPU 的内存你分配内存

注意事项

1. 可以将`cudaMalloc()` 分配的指针传递给在设备上执行的函数
2. 可以在设备代码中使用`cudaMalloc()` 分配的指针进行设备内存读写操作
3. 可以将`cudaMalloc()` 分配的指针传递给在主机上执行的函数
4. 不可以在主机代码中使用`cudaMalloc()` 分配的指针进行主机内存读写操作（即不能进行解引用）。

2.1.2　cudaMemcpy()

函数原型　`cudaError_t cudaMemcpy (void *dst, const void *src, size_t count, cudaMemcpyKind kind)`

函数用处　与 c 语言中的`memcpy` 函数一样，只是此函数可以在主机内存和`GPU` 内存之间互相拷贝数据

函数参数 cudaMemcpyKind kind 表示数据拷贝方向,如果 kind 赋值为cudaMemcpyDeviceToHost 表示数据从设备内存拷贝到主机内存

其他

1. 与 C 中的memcpy() 一样，以同步方式执行，即当函数返回时，复制操作就已经完成了，并且在输出缓冲区中包含了复制进去的内容
2. 相应的有个异步方式执行的函数cudaMemcpyAsync()，这个函数详解请看下面的流一节有关内容

2.1.3 cudaFree()

函数原型 cudaError_t cudaFree (void* devPtr)

函数作用 与 c 语言中的free() 函数一样，只是此函数释放的是cudaMalloc() 分配的内存

2.1.4 示例

```
#include <stdio.h>
#include <cuda_runtime.h>
__global__ void add( int a, int b, int *c ) {
    *c = a + b;
}
int main( void ) {
    int c;
    int *dev_c;
    //cudaMalloc()
    cudaMalloc( (void**)&dev_c, sizeof(int) );
    //核函数执行
    add<<<1,1>>>( 2, 7, dev_c );
    //cudaMemcpy()
    cudaMemcpy( &c, dev_c, sizeof(int),cudaMemcpyDeviceToHost ) ;
    printf( "2 + 7 = %d\n", c );
    //cudaFree()
    cudaFree( dev_c );

    return 0;
}
```

2.2 GPU 内存分类

2.2.1 全局内存

通俗意义上的 **设备内存**

2.2.2 共享内存

位置 设备内存

形式 关键字__shared__ 添加到变量声明中。如__shared__ float cache[10]

目的 对于GPU 上启动的每个线程块，CUDA C 编译器都将创建该共享变量的一个副本。**线程块中的**每个线程都共享这块内存，**但线程却无法看到也不能修改其他线程块的变量副本**。这样使得一个线程块中的多个线程能够在计算上通信和协作

2.2.3 常量内存

位置 设备内存

形式 关键字__constant__ 添加到变量声明中。如__constant__ float s[10];

目的 为了提升性能。常量内存采取了不同于标准全局内存的处理方式。在某些情况下，用常量内存替换全局内存能有效地减少内存带宽

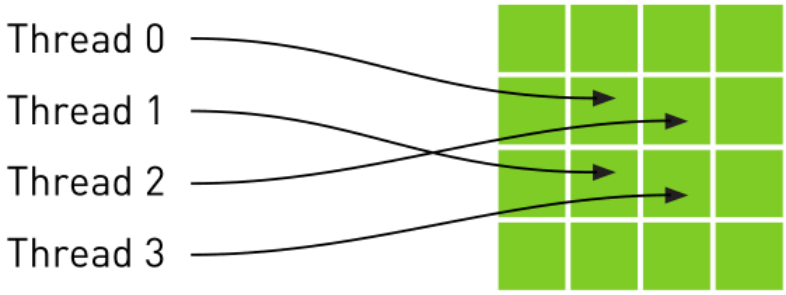
特点 常量内存用于保存在核函数执行期间不会发生变化的数据。**变量的访问限制为只读**。NVIDIA 硬件提供了64KB 的常量内存。不再需要cudaMalloc()或者cudaFree(), 而是在编译时，静态地分配空间

要求 当我们需要拷贝数据到常量内存中应该使用cudaMemcpyToSymbol(),而cudaMemcpy()会复制到全局内存

2.2.4 纹理内存

位置 设备内存

目的 能够减少对内存的请求并提供高效的内存带宽。是专门为那些在内存访问模式中存在大量空间局部性的图形应用程序设计，意味着一个线程读取的位置可能与邻近线程读取的位置“非常接近”。如下图：



2.2.5 固定内存

位置 主机内存

概念 也称为页锁定内存或者不可分页内存，操作系统将不会对这块内存分页并交换到磁盘上，从而确保了该内存始终驻留在物理内存中。因此操作系统能够安全地使某个应用程序访问该内存的物理地址，因为这块内存将不会破坏或者重新定位。

目的 提高访问速度。由于 GPU 知道主机内存的物理地址，因此可以通过“直接内存访问DMA（Direct Memory Access）技术来在 GPU 和主机之间复制数据。由于DMA 在执行复制时无需 CPU 介入。因此 DMA 复制过程中使用固定内存是非常重要的

形式 通过cudaHostAlloc() 函数来分配；通过cudaFreeHost() 释放

2.2.6 原子性

概念 如果操作的执行过程不能分解为更小的部分，我们将满足这种条件限制的操作称为原子操作

形式 函数调用，如atomicAdd（addr,y）将生成一个原子的操作序列，这个操作序列包括读取地址addr 处的值，将 y 增加到这个值，以及将结果保存回地址addr。

http://www.opengpu.org/forum.php?mod=viewthread&tid=14715

2.3 使用事件测试性能

用途 为了测量GPU 在某个任务上花费的时间。CUDA 中的事件本质上是一个GPU 时间戳。由于事件是直接在GPU 上实现的。因此不适用于对同时包含设备代码和主机代码的混合代码设计

形式 首先创建一个事件，然后记录事件，再计算两个事件之差，最后销毁事件。如：

```
cudaEvent_t start, stop;
cudaEventCreate( &start );
cudaEventCreate( &stop );
cudaEventRecord( start, 0 );

//do something

cudaEventRecord( stop, 0 );
float   elapsedTime;
cudaEventElapsedTime( &elapsedTime,start, stop );
cudaEventDestroy( start );
cudaEventDestroy( stop );
```

2.4 线程并行

多线程我们应该都不陌生，在操作系统中，进程是资源分配的基本单元，而线程是CPU 时间调度的基本单元（这里假设只有 1 个CPU）。

将线程的概念引申到CUDA 程序设计中，我们可以认为线程就是执行CUDA 程序的最小单元，前面我们建立的工程代码中，有个核函数概念不知各位童鞋还记得没有，在GPU 上**每个线程**都会运行一次该核函数。

但GPU 上的线程调度方式与CPU 有很大不同。CPU 上会有优先级分配，从高到低，同样优先级的可以采用时间片轮转法实现线程调度。GPU 上线程没有优先级概念，**所有线程机会均等**，线程状态只有等待资源和执行两种状态，如果资源未就绪，那么就等待；一旦就绪，立即执行。当GPU 资源很充裕时，所有线程都是并行执行的，这样加速效果很接近理论加速比；而GPU 资源少于总线程个数时，有一部分线程就会等待前面执行的线程释放资源，从而变为串行化执行。

```
cudaError_t addWithCuda(int *c, const int *a, const int *b, size_t size);
__global__ void addKernel(int *c, const int *a, const int *b)
{
    int i = threadIdx.x;
    c[i] = a[i] + b[i];
}
int main()
{
    const int arraySize = 5;
    const int a[arraySize] = { 1, 2, 3, 4, 5 };
    const int b[arraySize] = { 10, 20, 30, 40, 50 };
    int c[arraySize] = { 0 };
    // Add vectors in parallel.
    cudaError_t cudaStatus;
    int num = 0;
    cudaDeviceProp prop;
    cudaStatus = cudaGetDeviceCount(&num);
    for(int i = 0;i
    {
        cudaGetDeviceProperties(&prop,i);
    }
    cudaStatus = addWithCuda(c, a, b, arraySize);
    if (cudaStatus != cudaSuccess)
    {
        fprintf(stderr, "addWithCuda failed!");
        return 1;
    }

    printf("{1,2,3,4,5} + {10,20,30,40,50} = {%d,%d,%d,%d,%d}\n",c[0],c[1],c[2],c[3],c[4]);

    // cudaThreadExit must be called before exiting in order for profiling and
    // tracing tools such as Nsight and Visual Profiler to show complete traces.
    cudaStatus = cudaThreadExit();

    if (cudaStatus != cudaSuccess)
    {
        fprintf(stderr, "cudaThreadExit failed!");
        return 1;
    }
    return 0;
}

// 重点理解这个函数
```

```
cudaError_t addWithCuda(int *c, const int *a, const int *b, size_t size)
{
    int *dev_a = 0; //GPU设备端数据指针
    int *dev_b = 0;
    int *dev_c = 0;
    cudaError_t cudaStatus;    //状态指示

    // Choose which GPU to run on, change this on a multi-GPU system.
    cudaStatus = cudaSetDevice(0);  //选择运行平台
    if (cudaStatus != cudaSuccess)
    {
        fprintf(stderr, "cudaSetDevice failed!  Do you have a CUDA-capable GPU installed?");

        goto Error;
    }
    // 分配GPU设备端内存
    cudaStatus = cudaMalloc((void**)&dev_c, size * sizeof(int));
    if (cudaStatus != cudaSuccess)
    {
        fprintf(stderr, "cudaMalloc failed!");
        goto Error;
    }
    cudaStatus = cudaMalloc((void**)&dev_a, size * sizeof(int));
    if (cudaStatus != cudaSuccess)
    {
        fprintf(stderr, "cudaMalloc failed!");
        goto Error;
    }
    cudaStatus = cudaMalloc((void**)&dev_b, size * sizeof(int));
    if (cudaStatus != cudaSuccess)
    {
        fprintf(stderr, "cudaMalloc failed!");
        goto Error;
    }
    // 拷贝数据到GPU
    cudaStatus = cudaMemcpy(dev_a, a, size * sizeof(int), cudaMemcpyHostToDevice);
    if (cudaStatus != cudaSuccess)
    {
        fprintf(stderr, "cudaMemcpy failed!");
        goto Error;
    }
    cudaStatus = cudaMemcpy(dev_b, b, size * sizeof(int), cudaMemcpyHostToDevice);
    if (cudaStatus != cudaSuccess)
    {
```

```
        fprintf(stderr, "cudaMemcpy failed!");
        goto Error;
    }
    // 运行核函数
     addKernel<<<1, size>>>(dev_c, dev_a, dev_b);

    // cudaThreadSynchronize waits for the kernel to finish, and returns
    // any errors encountered during the launch.
    cudaStatus = cudaThreadSynchronize();   //同步线程

    if (cudaStatus != cudaSuccess)
    {
        fprintf(stderr, "cudaThreadSynchronize returned error code %
            d after launching addKernel!\n", cudaStatus);
        goto Error;
    }
    // Copy output vector from GPU buffer to host memory.
    cudaStatus = cudaMemcpy(c, dev_c, size * sizeof(int), cudaMemcpyDeviceToHost);      //拷
        贝结果回主机
    if (cudaStatus != cudaSuccess)
    {
        fprintf(stderr, "cudaMemcpy failed!");
        goto Error;
    }
    Error:
        cudaFree(dev_c);    //释放GPU设备端内存
        cudaFree(dev_a);
        cudaFree(dev_b);
    return cudaStatus;
}
```

<<<>>> 表示运行时配置符号，里面 1 表示只分配一个线程组（又称线程块、Block），size 表示每个线程组有size 个线程（Thread）。

本程序中size 根据前面传递参数个数应该为 5，所以运行的时候，核函数在 5 个 GPU 线程单元上分别运行了一次，总共运行了 5 次。

这 5 个线程是如何知道自己“身份”的？是靠threadIdx 这个内置变量，它是个 dim3 类型变量，接受<<<>>> 中第二个参数，它包含x,y,z 3 维坐标，而我们传入的参数只有一维，所以只有 x 值是有效的。通过核函数中int i = threadIdx.x; 这一句，每个线程可以获得自身的id 号，从而找到自己的任务去执行。

2.5 块并行

块并行相当于操作系统中多进程的情况，上节说到，CUDA 有线程组（线程块）的概念，将一组线程组织到一起，共同分配一部分资源，然后内部调度执行。**线程块与线程块之间，毫无瓜葛**。这有利于做更粗粒度的并行。我们将上一节的代码改为块并行版本如下：

```
__global__ void addKernel(int *c, const int *a, const int *b)
{
    int i = blockIdx.x;
    c[i] = a[i] + b[i];
}

addKernel<<>>(dev_c, dev_a, dev_b);
```

和上一节相比，只有这两行有改变，<<<>>> 里第一个参数改成了size，第二个改成了 1，表示我们分配size 个线程块，每个线程块仅包含 1 个线程，总共还是有 5 个线程。这 5 个线程相互独立，执行核函数得到相应的结果，与上一节不同的是，每个线程获取id 的方式变为int i = blockIdx.x；这是线程块ID。

2.5.1 线程并行月块并行的区别

线程并行是细粒度并行，调度效率高；块并行是粗粒度并行，每次调度都要重新分配资源，有时资源只有一份，那么所有线程块都只能排成一队，串行执行。

我们的任务有时可以采用分治法，将一个大问题分解为几个小规模问题，将这些小规模问题分别用一个线程块实现，线程块内可以采用细粒度的线程并行，而块之间为粗粒度并行，这样可以充分利用硬件资源，降低线程并行的计算复杂度。适当分解，降低规模，在一些矩阵乘法、向量内积计算应用中可以得到充分的展示

2.6 线程格并行-流

多个线程块组织成了一个Grid，称为线程格。一组线程并行处理可以组织为一个block，而一组block 并行处理可以组织为一个Grid，很自然地想到，Grid 只是一个网格，我们是否可以利用多个网格来完成并行处理呢？答案就是利用流

流可以实现在一个设备上**运行多个核函数**。前面的块并行也好，线程并行也好，运行的**核函数都是相同的**（代码一样，传递参数也一样）。而流并行，可以执行不同的核函数，也可以实现对同一个核函数传递不同的参数，实现任务级别的并行。

CUDA 中的流用cudaStream_t 类型实现, 用到的API 有以下几个：

cudaStreamCreate(cudaStream_t *s) 用于创建流，

cudaStreamDestroy(cudaStream_t s) 用于销毁流，

cudaStreamSynchronize() 用于单个流同步，

cudaDeviceSynchronize() 用于整个设备上的所有流同步，

cudaStreamQuery() 用于查询一个流的任务是否已经完成

```
cudaStream_t stream[5];
for(int i = 0;i<5;i++)
{
    cudaStreamCreate(&stream[i]);   //创建流
}

for(int i = 0;i<5;i++)
{
    // 线程块数 线程数 共享内存 流对象
    addKernel<<<1,1,0,stream[i]>>>(dev_c+i, dev_a+i, dev_b+i);    //执行流
}

for(int i = 0;i<5;i++)
{
    cudaStreamDestroy(stream[i]);   //销毁流
}
```

函数代码仍然和块并行的版本一样，只是在调用时做了改变，<<<>>> 中的参数多了两个，其中前两个和块并行、线程并行中的意义相同，仍然是**线程块数**（这里为 1）、每个线程块中**线程数**（这里也是 1）。第三个为 0 表示每个block 用到的**共享内存**大小，这个我们后面再讲；第四个为**流对象**，表示当前核函数在哪个流上运行。

我们创建了 5 个流，每个流上都装载了一个核函数，同时传递参数有些不同，也就是每个核函数作用的对象也不同。这样就实现了任务级别的并行，**当我们有几个互不相关的任务时**，可以写多个核函数，资源允许的情况下，我们将这些核函数装载到不同流上，然后执行，这样可以实现更粗粒度的并行

2.7 线程通信

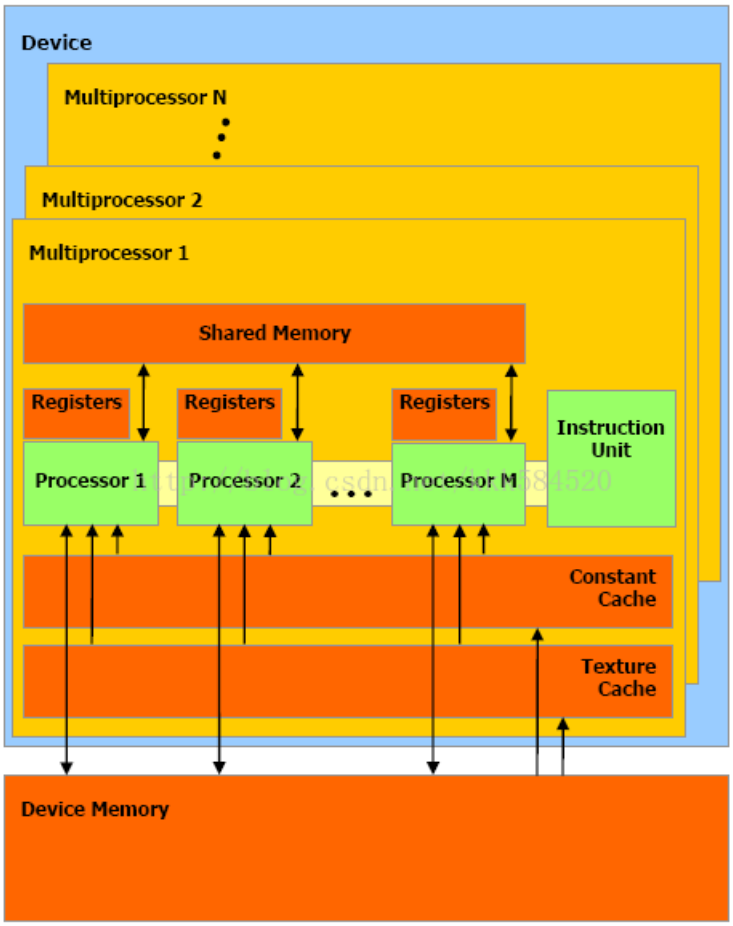
线程通信在 CUDA 中有三种实现方式：

共享存储器

线程同步

原子操作

最常用的是前两种方式，共享存储器，术语Shared Memory，是位于SM 中的特殊存储器。还记得 SM 吗，就是流多处理器，大核是也。一个SM 中不仅包含若干个SP（流处理器，小核），还包括一部分高速Cache，寄存器组，共享内存等，结构如图所示



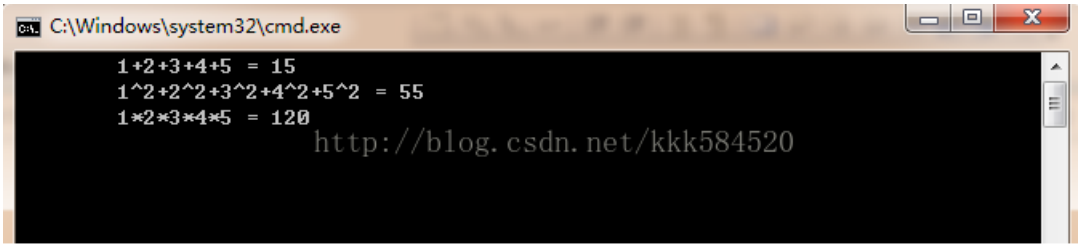
从图中可看出，一个SM 内有M 个SP，Shared Memory 由这M 个SP 共同占有。另外指令单元也被这M 个SP 共享，即SIMT 架构（单指令多线程架构），一个SM 中所有SP 在同一时间执行同一代码

为了实现线程通信，仅仅靠共享内存还不够，需要有同步机制才能使线程之间实现有序处理。通常情况是这样：当线程 A 需要线程 B 计算的结果作为输入时，需要确保线程 B 已经将结果写入共享内存中，然后线程 A 再从共享内存中读出。同步必不可少，否则，线程A 可能读到的是无效的结果，造成计算错误。同步机制可以用CUDA 内置函数：__syncthreads()；当某个线程执行到该函数时，进入等待状态，直到同一线程块（Block）中所有线程都执行到这个函数为止，即一个__syncthreads() 相当于一个线程同步点，确保一个Block 中所有线程都达到同步，然后线程进入运行状态。

```
//Begin
if this is thread B
```

```
    write something to Shared Memory;
end if
__syncthreads();
if this is thread A
    read something from Shared Memory;
end if
//End
```

2.7.1 线程通信实例



很简单，就是分别求出 1 5 这 5 个数字的和，平方和，连乘积。相信学过 C 语言的童鞋都能用for 循环做出同上面一样的效果，但为了学习CUDA 共享内存和同步技术，我们还是要把简单的东西复杂化。

简要分析一下，上面例子的输入都是一样的，1,2,3,4,5 这 5 个数，但计算过程有些变化，而且每个输出和所有输入都相关，不是前几节例子中那样，一个输出只和一个输入有关。所以我们在利用CUDA 编程时，需要针对特殊问题做些让步，把一些步骤串行化实现。

输入数据原本位于主机内存，通过cudaMemcpy API 已经拷贝到GPU 显存（术语为全局存储器，Global Memory），每个线程运行时需要从Global Memory 读取输入数据，然后完成计算，最后将结果写回Global Memory。当我们计算需要多次相同输入数据时，大家可能想到，每次都分别去Global Memory 读数据好像有点浪费，如果数据很大，那么反复多次读数据会相当耗时间。索性我们把它从Global Memory 一次性读到 SM 内部，然后在内部进行处理，这样可以节省反复读取的时间。

有了这个思路，结合上节看到的 SM 结构图，看到有一片存储器叫做SharedMemory，它位于 SM 内部，处理时访问速度相当快（差不多每个时钟周期读一次），而全局存储器读一次需要耗费几十甚至上百个时钟周期。于是，我们就制定 A 计划如下：

线程块数：1，块号为 0；（只有一个线程块内的线程才能进行通信，所以我们只分配一个线程块，具体工作交给每个线程完成）

线程数：5，线程号分别为0~4；（线程并行，前面讲过）

共享存储器大小：5 个int 型变量大小（5 * sizeof(int) ）。

步骤一：读取输入数据。将Global Memory 中的 5 个整数读入共享存储器，位置一一对应，和线程号也一一对应，所以可以同时完成。

步骤二：线程同步，确保所有线程都完成了工作。

步骤三：指定线程，对共享存储器中的输入数据完成相应处理。

```
__global__ void addKernel(int *c, const int *a)
{
    int i = threadIdx.x;
    extern __shared__ int smem[];
    smem[i] = a[i];
    __syncthreads();
    if(i == 0)  // 0号线程做平方和
    {
        c[0] = 0;
        for(int d = 0; d < 5; d++)
        {
            c[0] += smem[d] * smem[d];
        }
    }
    if(i == 1)//1号线程做累加
    {
        c[1] = 0;
        for(int d = 0; d < 5; d++)
        {
            c[1] += smem[d];
        }
    }
    if(i == 2)  //2号线程做累乘
    {
        c[2] = 1;
        for(int d = 0; d < 5; d++)
        {
            c[2] *= smem[d];
        }
    }
}

addKernel<<<1, size, size * sizeof(int), 0>>>(dev_c, dev_a);
```

2.8 编译

设备程序需要由NVCC 进行编译，而主机程序只需要由主机编译器（如VS2008中的cl.exe，Linux上的GCC）。**主机程序**主要完成设备环境初始化，数据传输等必备过程，**设备程序**只负责计算。

2.8.1 编译注意事项

http://blog.csdn.net/shengwenj/article/details/48917203