【CUDA 基础】5.1 CUDA共享内存概述

2018-06-01 | CUDA , Freshman | 0 |

Abstract: 本文为CUDA内存的概述，介绍共享内存的模型，分配，访问，配置，同步等内容

Keywords: 模型，分配，访问，配置，同步

CUDA共享内存概述

这里首先要进一步说明一下，前面我们在说缓存的时候说其是可编程的，这是不准确的，应该说是可以控制的，而我们今天要说的共享内存才是真正意义上的可编程的。

废话不多说了，一套CUDA内容写到现在，一大半已经进行完了，希望我们在一个系列完成后都能有所成长，而不是纯粹的阅读或者码字。

GPU内存按照类型（物理上的位置）可以分为

- 板载内存
- 片上内存

全局内存是较大的板载内存，延迟高，共享内存是片上的较小的内存，延迟低，带宽高。前面我我们讲过工厂的例子，全局内存就是原料工厂，要用车来运输原料，共享内存是工厂内存临时存放原料的房间，取原料路程短速度快。
共享内存是一种可编程的缓存，共享内存通常的用途有：

1. 块内线程通信的通道
2. 用于全局内存数据的可编程管理的缓存
3. 告诉暂存存储器，用于转换数据来优化全局内存访问模式

本章我们研究两个例子：

1. 归约核函数
2. 矩阵转置核函数

共享内存

共享内存（shared memory，SMEM）是GPU的一个关键部分，物理层面，每个SM都有一个小的内存池，这个线程池被次SM上执行的线程块中的所有线程所共享。共享内存使同一个线程块中可以相互协同，便于片上的内存可以被最大化的利用，降低回到全局内存读取的延迟。
共享内存是被我们用代码控制的，这也是是他称为我们手中最灵活的优化武器。
结合我们前面学习的一级缓存，二级缓存，今天的共享内存，以及后面的只读和常量缓存，他们的关系如下图：

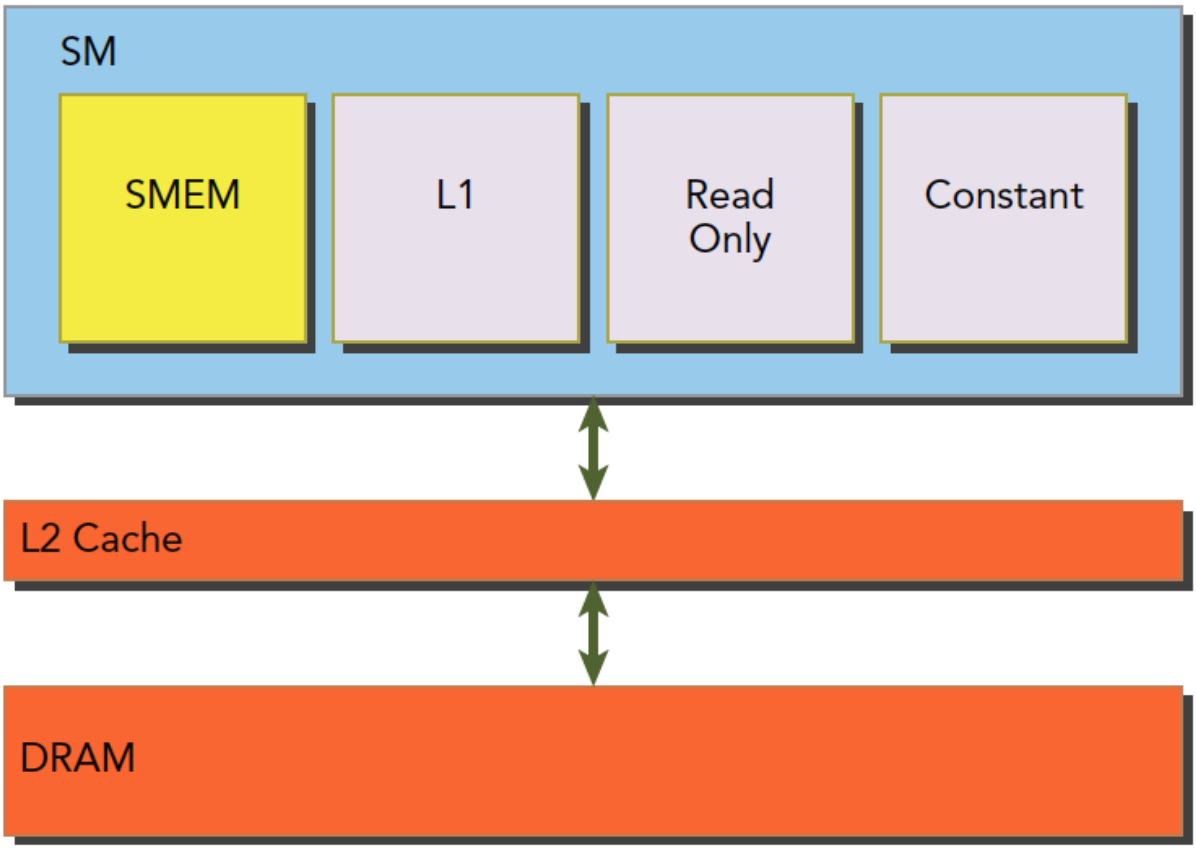


FIGURE 5-1

SM上有共享内存，L1一级缓存，ReadOnly 只读缓存，Constant常量缓存。所有从Dram全局内存中过来的数据都要经过二级缓存，相比之下，更接近SM计算核心的SMEM，L1，ReadOnly，Constant拥有更快的读取速度，SMEM和L1相比于L2延迟低大概20~30倍，带宽大约是10倍。

下面我们了解下共享内存的生命周期和读取性质。

共享内存是在他所属的线程块被执行时建立，线程块执行完毕后共享内存释放，线程块和他的共享内存有相同的生命周期。

对于每个线程对共享内存的访问请求

1. 最好的情况是当前线程束中的每个线程都访问一个不冲突的共享内存，具体是什么样的我们后面再说，这种情况，大家互不干扰，一个事务完成整个线程束的访问，效率最高
2. 当有访问冲突的时候，具体怎么冲突也要后面详细说，这时候一个线程束32个线程，需要32个事务。
3. 如果线程束内32个线程访问同一个地址，那么一个线程访问完后以广播的形式告诉大家

后面的全章内容都是基本围绕如何避免访问冲突，高效的是有共享内存来展开的。

注意我们刚才说的共享内存的生命周期是和其所属的线程块相同的，这个共享内存是编程模型层面上的。

物理层面上，一个SM上的所有的正在执行的线程块共同使用物理的共享内存，所以共享内存也成为了活跃线程块的限制，共享内存越大，或者块使用的共享内存越小，那么线程块级别的并行度就越高。
共享内存，高端有限资源，合理使用！

接着说说可编程，矩阵乘法的串行形式，最简单的方式是三层循环，通过调整循环可以获得更好的缓存命中率，这个题在找工作笔试的时候有，当时在大学笔试工作的时候，我会很傻x的在笔试习题上写上注释，可以通过调整循环顺序提高缓存命中率，但是现在想一下，CPU的缓存是不可控制的，你只能调整自己的程序来适应它。
GPU高端的一点，就是你不止有一个缓存可以编程控制，而是有好几个。

共享内存分配

分配和定义共享内存的方法有多种，动态的声明，静态的声明都是可以的。可以在核函数内，也可以在核函数外（也就是本地的和全局的，这里是说变量的作用域，在一个文件中），CUDA支持1，2，3维的共享内存声明，当然多了不知道支不支持，可能新版本支持，但是要去查查手册，一般情况下我们就假装最多只有三维。
声明共享内存通过关键字：

```
__shared__
```

声明一个二维浮点数共享内存数组的方法是：

```
__shared__ float a[size_x][size_y];
```

这里的size_x,size_y和声明c++数组一样，要是一个编译时确定的数字，不能是变量。
如果想动态声明一个共享内存数组，可以使用extern关键字，并在核函数启动时添加第三个参数。
声明：

```
extern __shared__ int tile[];
```

在执行上面这个声明的核函数时，使用下面这种配置：

```
kernel<<<grid,block,isize*sizeof(int)>>>(...);
```

isize就是共享内存要存储的数组的大小。比如一个十个元素的int数组，isize就是10.

注意，动态声明只支持一维数组。

共享内存存储体和访问模式

声明和定义是代码层面上的产生了共享内存，接下来我们看看共享内存是怎么存储以及是如何访问的。上一章我们研究了全局内存，并且明确的学习了带宽和延迟是如何对核函数造成性能影响的。共享内存是用来隐藏全局内存延迟以及提高带宽性能的主要武器之一。掌握武器的办法就是了解武器的工作原理和各个部件的特性。

内存存储体

共享内存是一个一维的地址空间，注意这句话的意思是，共享内存的地址是一维的，也就是和所有我们前面提到过的内存一样，都是线性的，二维三维更多维的地址都要转换成一维的来对应物理上的内存地址。

共享内存有个特殊的形式是，分为32个同样大小的内存模型，称为存储体，可以同时访问。32个存储体的目的是对应一个线程束中有32个线程，这些线程在访问共享内存的时候，如果都访问不同存储体（无冲突），那么一个事务就能够完成，否则（有冲突）需要多个内存事务了，这样带宽利用率降低。

是否有冲突，以及冲突如何发生我们下面介绍。

存储体冲突

当多个线程要访问一个存储体的时候，冲突就发生了，注意这里是说访问同一个存储体，而不是同一个地址，访问同一个地址不存在冲突（广播形式）。当发生冲突就会有等待和更多的事务产生，这是严重影响效率的。

线程束访问共享内存的时候有下面3种经典模式：

1. 并行访问，多地址访问多存储体
2. 串行访问，多地址访问同一存储体
3. 广播访问，单一地址读取单一存储体

并行访问是最常见，也是效率较高的一种，但是也可以分为完全无冲突，和小部分冲突的情况，完全无冲突是理想模式，线程束中所有线程通过一个内存事务完成自己的需求，互不干扰，效率最高，当有小部分冲突的时候，大部分不冲突的部分可以通过一个内存事务完成，冲突的被分割成另外的不冲突的事务被执行，这样效率稍低。

上面的小部分冲突变成完全冲突就是串行模式了，这是最糟糕的形式，所有线程访问同一个存储体，注意不是同一个地址，是同一个存储体，一个存储体有很多地址。这时就是串行访问。

广播访问是所有线程访问一个地址，这时候，一个内存事务执行完毕后，一个线程得到了这个地址的数据，他会通过广播的形式告诉其他所有线程，虽然这个延迟相比于完全的并行访问并不慢，但是他只读取了一个数据，带宽利用率很差。

最优访问模式（并行不冲突）：

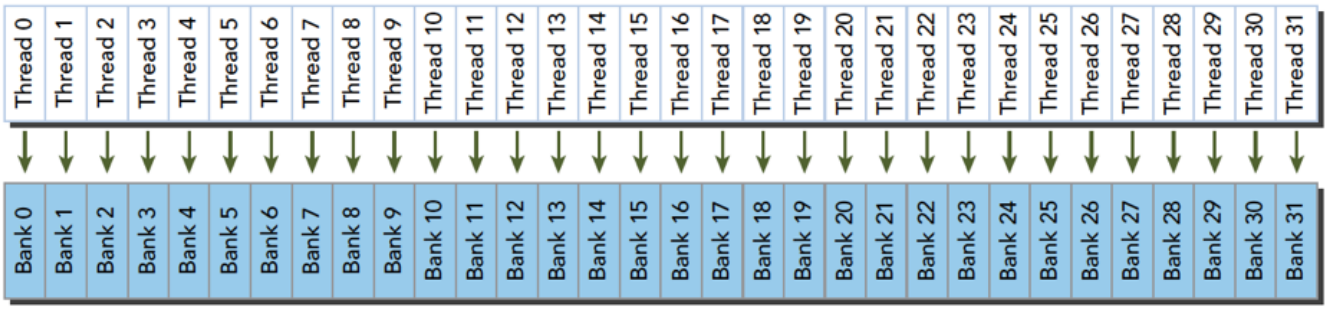


FIGURE 5-2

不规则的访问模式（并行不冲突）：

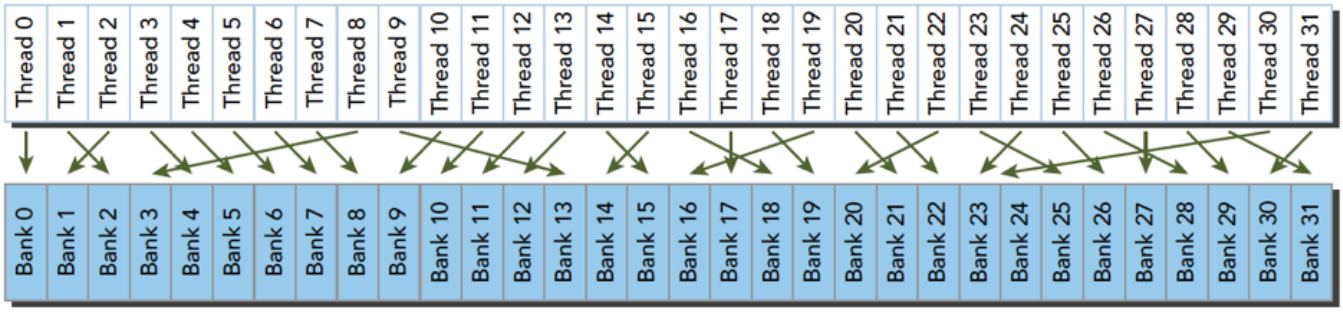


FIGURE 5-3

不规则的访问模式（并行可能冲突，也可能不冲突）

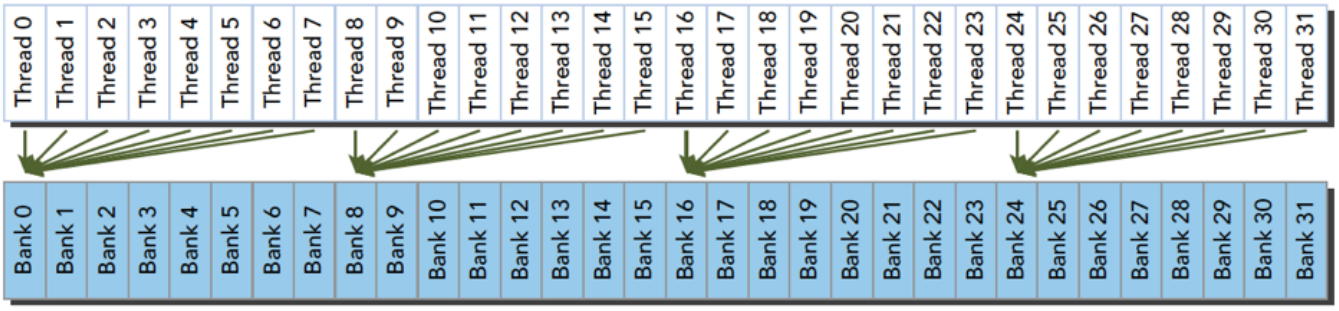


FIGURE 5-4

这时候又两种可能

1. 冲突：这时候就要等待了
2. 不冲突：访问同一个存储体的线程都要访问同一个地址，通过广播解决问题。

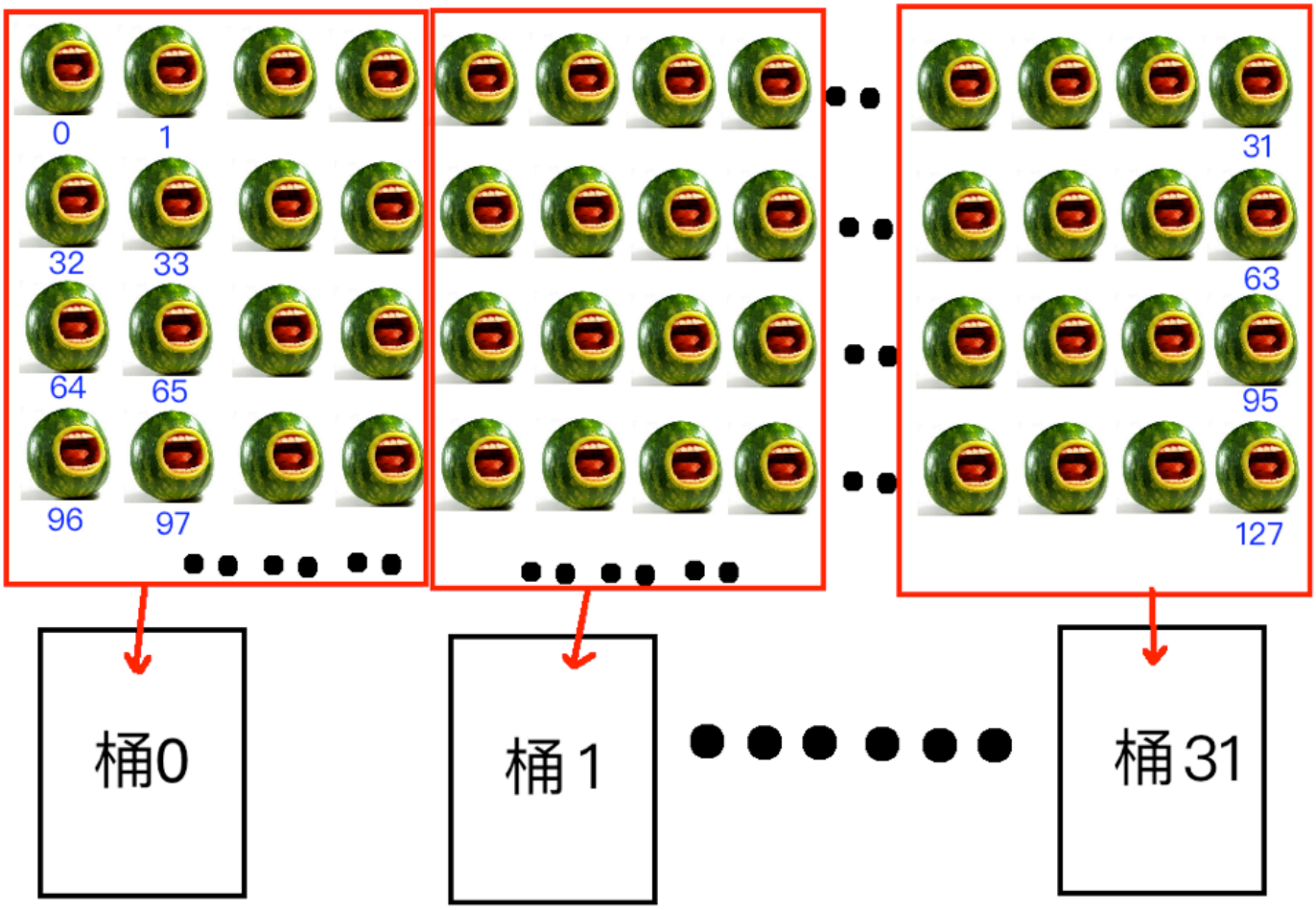
以上就是产生冲突的根本原因，我们通过调整数据，代码，算法，最好规避冲突，提高性能。

访问模式

共享内存的存储体和地址有什么关系呢？这个关系决定了访问模式。内存存储体的宽度随设备计算能力不同而变化，有以下两种情况：

1. 2.x计算能力的设备，为4字节（32位）
2. 3.x计算能力的设备，为8字节（64位）

怎么理解宽度呢，我们假设我们这有三十二个水桶，每个水桶当做一个存储体，桶的口大小是固定的，假设我们用桶装西瓜，每个桶的口最多同时能拿出四个西瓜，能拿出四个西瓜，宽度就是4，能拿出八个就是八，这就是宽度的通俗解释。
然后我们将一排西瓜编号，从0开始，一直到n然后我们有三十二个编了号的桶（0~31号），摆成一排，然后往桶里同时装西瓜，因为一次只能装四个西瓜，那么我们把0~3号西瓜装到0号桶，4~7号习惯装入1号桶，以此类推，当装到第31号桶的时候，我们装 124~127号西瓜；然后我们每个桶里都有四个西瓜了，接着我们将128~131号西瓜装入0号桶，开始下一轮装西瓜。
这就是共享内存的存储体的访问模式，



那么我们已知西瓜的编号怎么知道西瓜在哪个桶里呢?

公式如下

$$桶号 = \frac{西瓜编号 \div 4}{32}\%32$$

当然这是我们的例子，转换成内存地址的话，西瓜编号对应的就是字节的地址，宽度就是4个西瓜，4字节（8字节），32个桶对应32个存储体：

$$存储体索引 = \frac{字节地址 \div 4}{存储体数}\%存储体数$$

上面公式中 $\%$ 表示取余。

我们来看个正规的图：

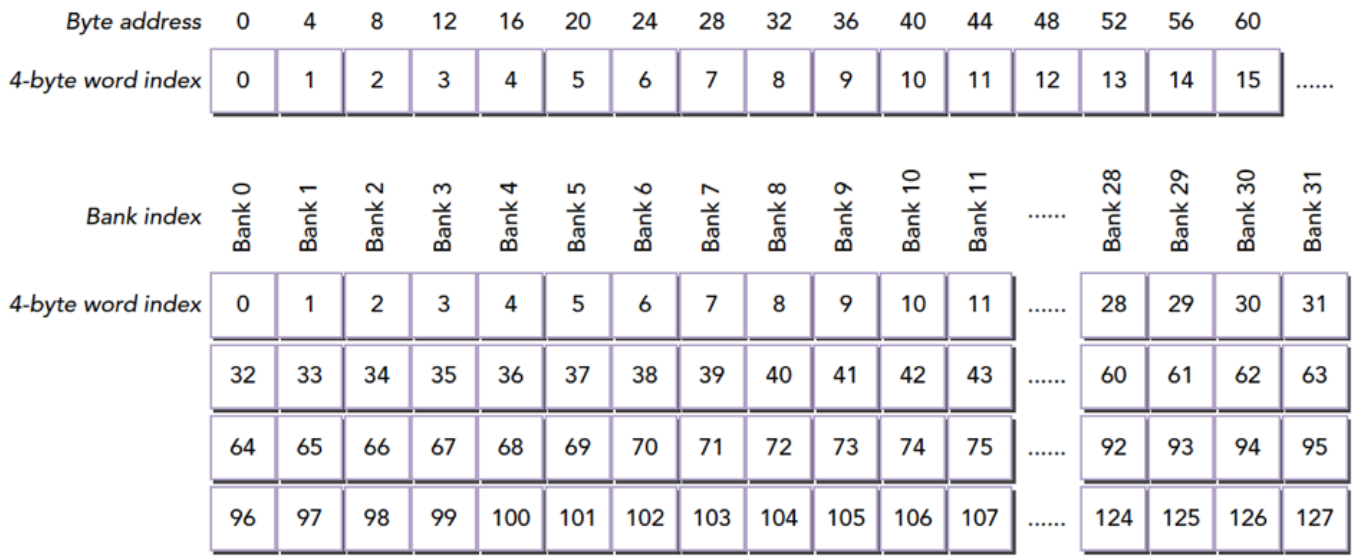


FIGURE 5-5

这个和我们的西瓜图内容一致，知识第一个byte address放成直线了，而我把西瓜放成了二维的，实际上一维的是准确的。

同一个线程束中的两个线程访问同一个地址不会发生冲突，一个线程读取后广播告诉有相同需求的线程。但是对于写入，这个就不确定了，结果不可预料。

上面我们介绍的存储体宽度是4的情况，宽度是8的情况同样，但是宽度变宽了，其带宽就有变宽了，比如，我们之前一次只能取四个西瓜，现在可以取八个西瓜了，这时候如果有两个线程访问同一个存储体，按照我们前面的解释，一种是访问同一个地址，这时候通过广播来解决冲突，还有一种冲突是需要用等待解决的，当桶变宽了，如果一个线程想要桶里左边的西瓜，而一个线程想要右边的西瓜，这时候是不冲突的，因为桶是够宽的。

或者我们可以理解为更宽的桶，在桶中间又进行了一次间隔，左右两边各一个空间，读取不影响，如果两个线程都要左边的西瓜则等待，如果一个要左边的一个要右边的，这时候可以同时进行不冲突。

把桶换成存储体就是

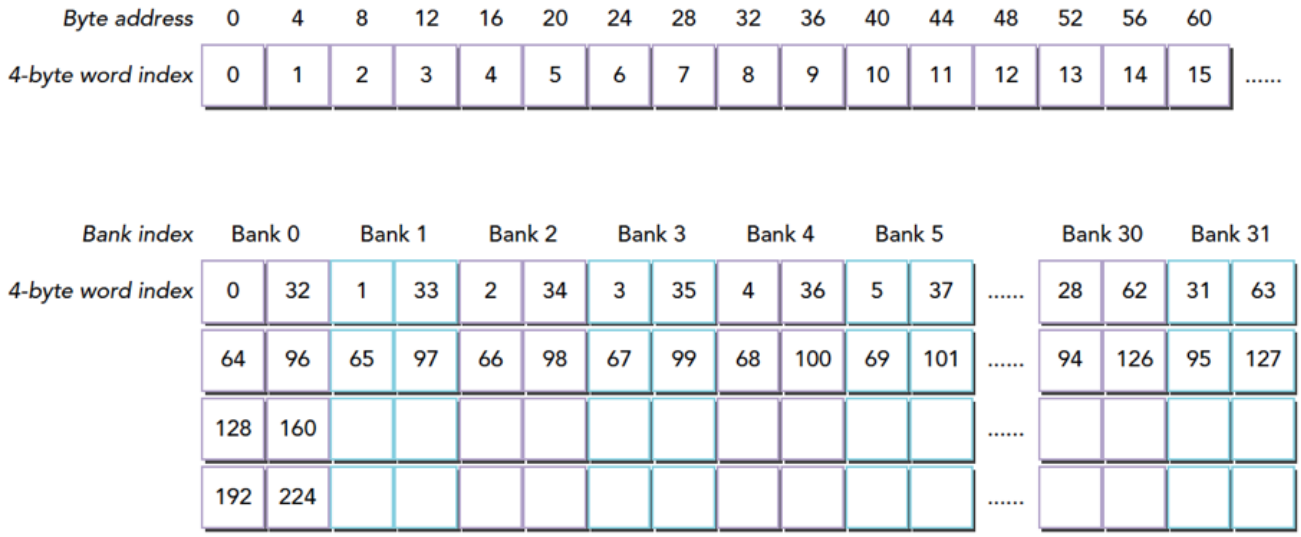


FIGURE 5-6

下图显示64位宽的存储体无冲突访问的一种情况，每个bank被划分成了两部分

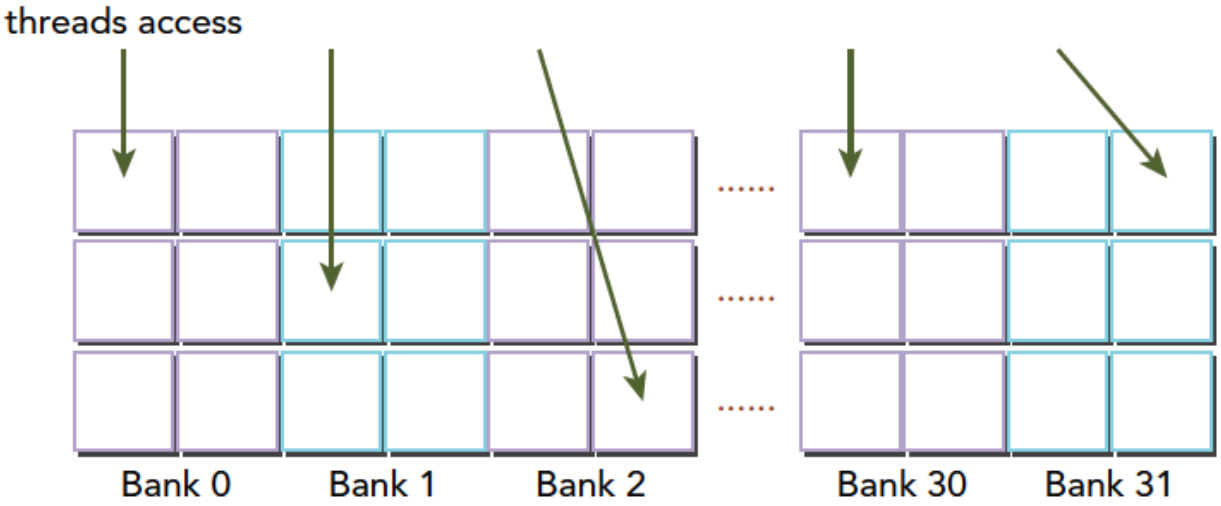


FIGURE 5-7

下图是另一种无冲突方式：

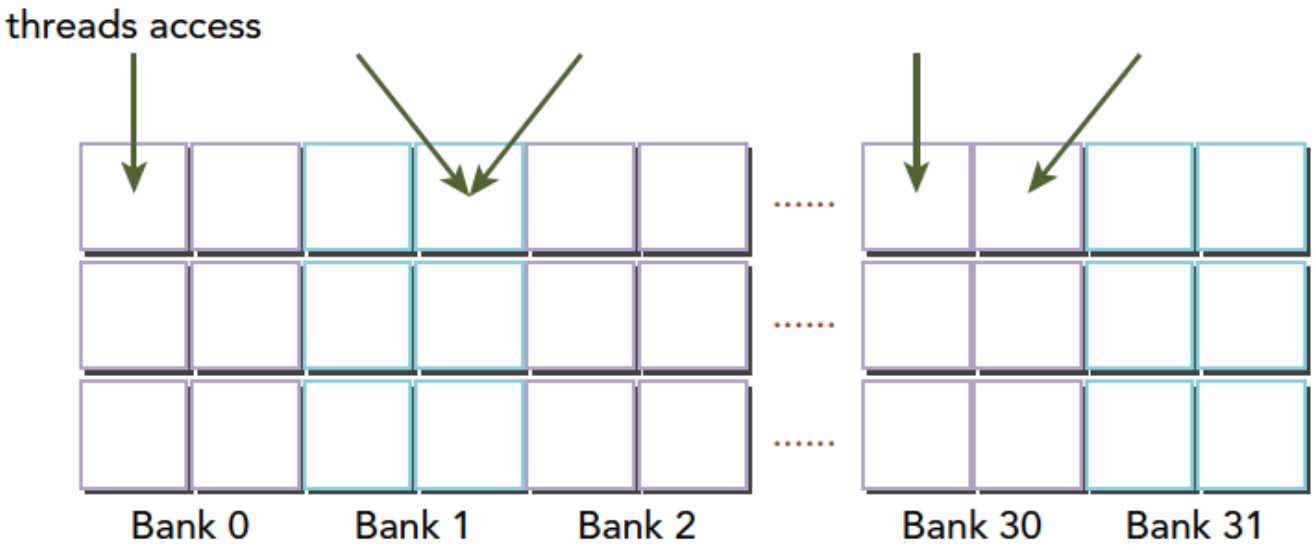


FIGURE 5-8

一种冲突方式，两个线程访问同一个小桶：

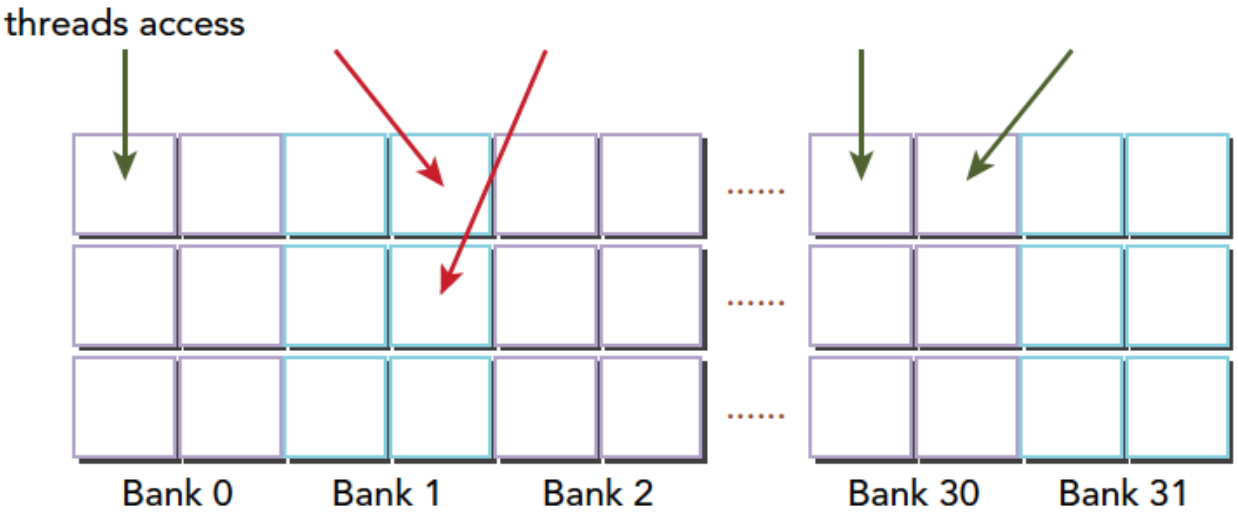


FIGURE 5-9

另一种冲突方式，三个线程访问同一个小桶

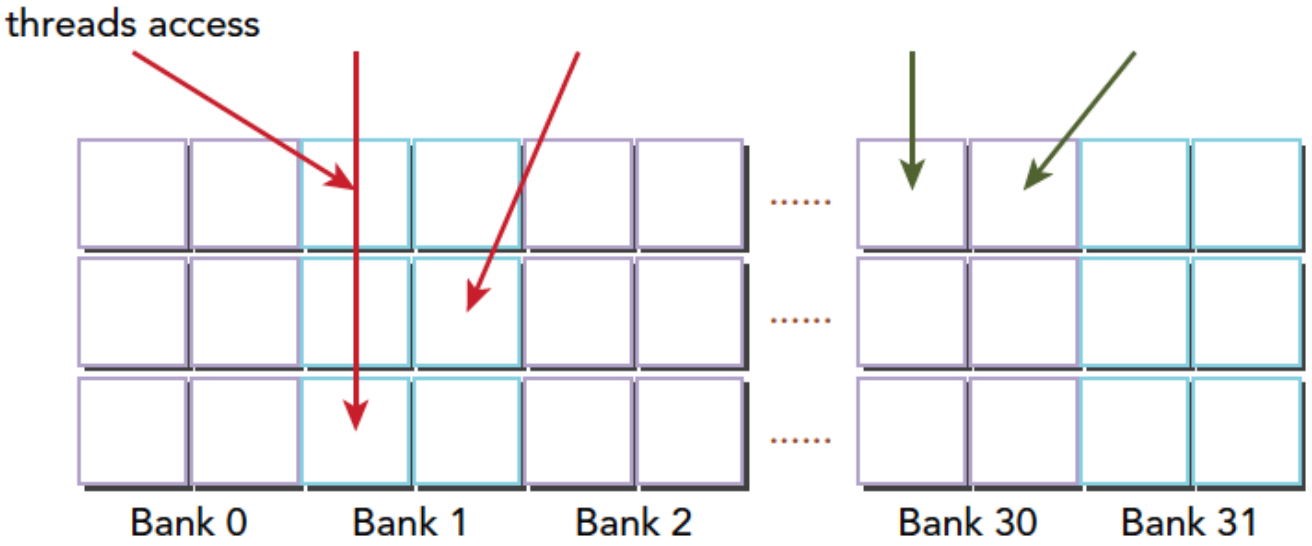
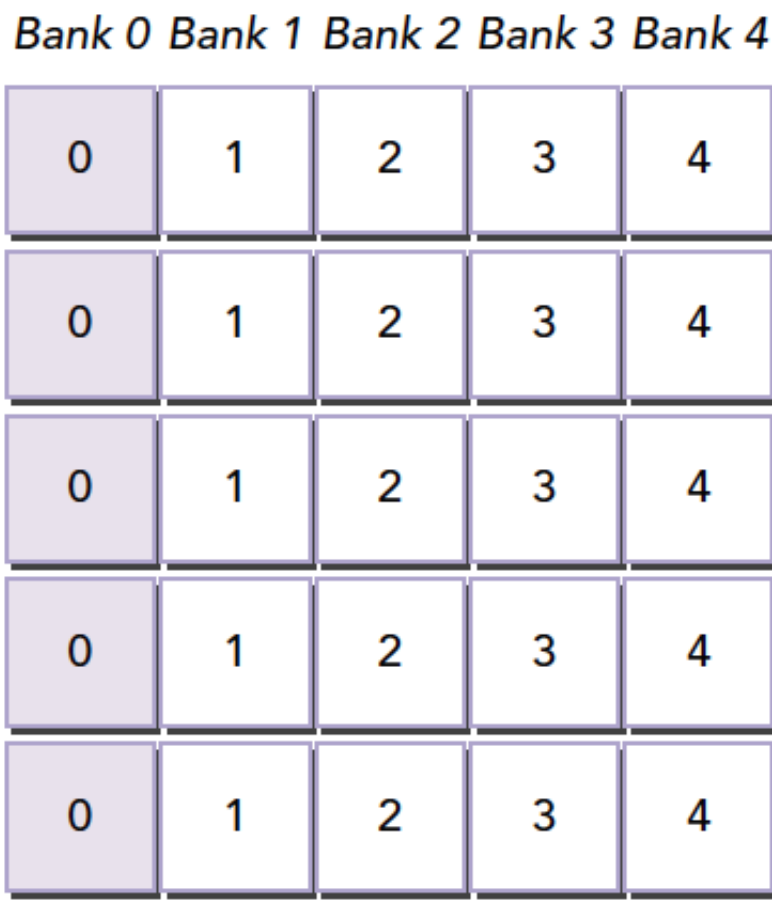


FIGURE 5-10

内存填充

存储体冲突会严重影响共享内存的效率，那么当我们遇到严重冲突的情况下，可以使用填充的办法让数据错位，来降低冲突。

假如我们当前存储体内的数据罗列如下，这里假设共4个存储体，实际是32个



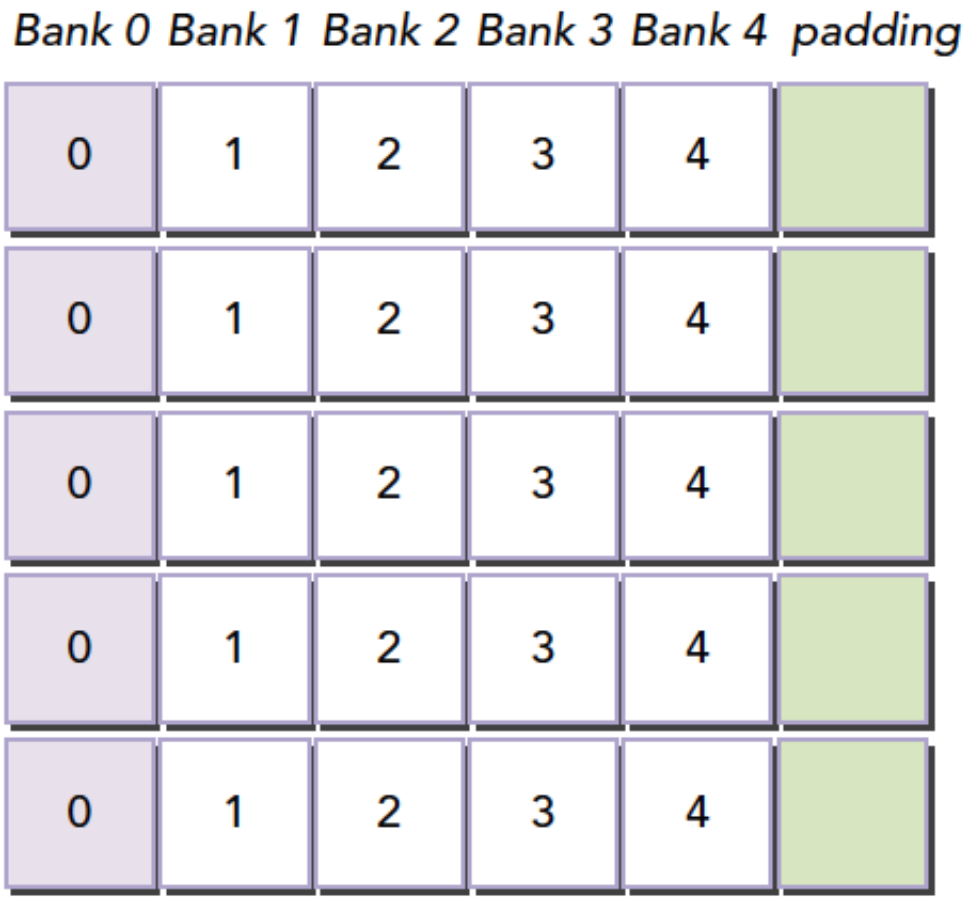
当我们的线程束访问bank0中的不同数据的时候就会发生一个5线程的冲突，这时候我们假如我们分配内存时候的声明是：

```
__shared__ int a[5][4];
```

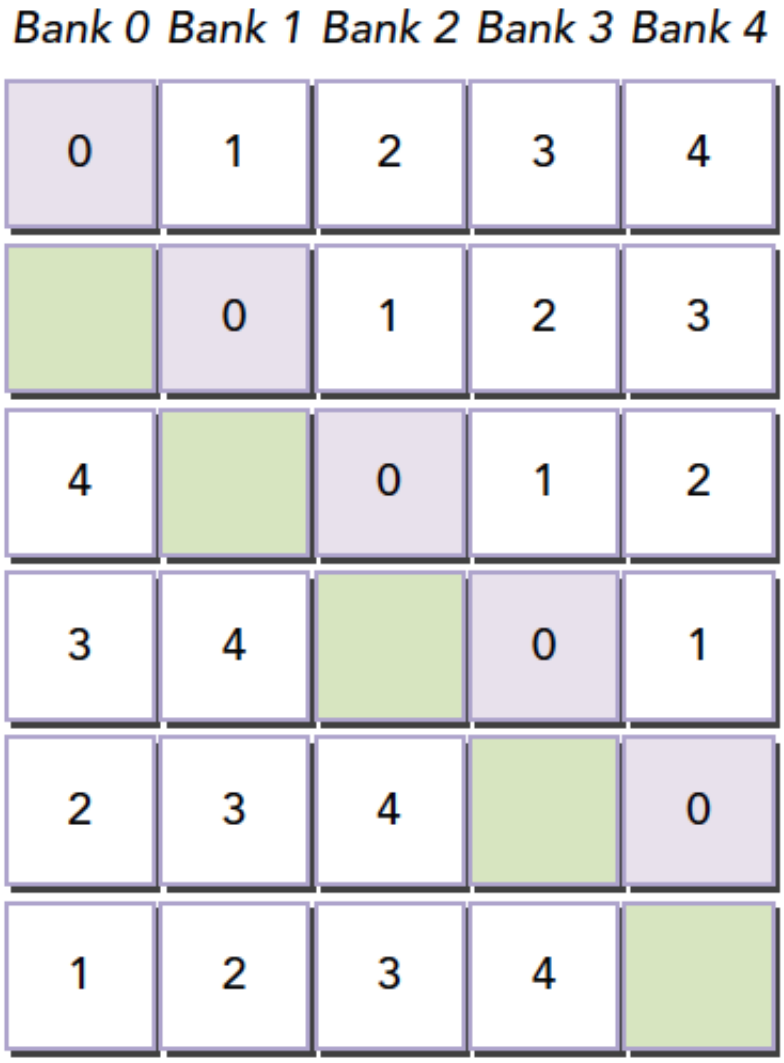
这时候我们的就会得到上面的图中的这种内存布局，但是当我们声明的时候改成

```
__shared__ int a[5][5];
```

就会产生这个效果，在编程时候加入一行填充物



然后编译器会将这个二维数组重新分配到存储体，因为存储体一共就4个，我们每一行有5个元素，所以有一个元素进入存储体的下一行，这样，所有元素都错开了，就不会出现冲突了。



这个例子可能有点难懂，比如，我们还是举例子装西瓜，当我们装到第31个的时候，我们不想把32号33号34号35号放到0号桶里，怎么办，叫四个冬瓜过来占位置，那么其就应该放在31号后面，加4列冬瓜。

共享内存在确定大小的时候，比如编译的时候，就已经被确定好每个地址在哪个存储体中了，想要改变分布，就在声明共享内存的时候调整就行，跟将要存储到共享内存中的数据没有关系。

注意：共享内存声明时，就决定了每个地址所在的存储体，想要调整每个地址对应的存储体，就要扩大声明的共享内存的大小，至于扩大多少，就要根据我们前面的公式好好计算了。这段是本文较难理解的一段。

访问模式配置

访问模式查询：可以通过以下语句，查询是4字节还是8字节：

```
cudaError_t cudaDeviceGetSharedMemConfig(cudaSharedMemConfig * pConfig);
```

返回的pConfig可以是下面的结果：

```
cudaSharedMemBankSizeFourByte
cudaSharedMemBankSizeEightByte
```

在可以配置的设备上，可以用下面函数来配置新的存储体大小：

```
cudaError_t cudaDeviceSetShareMemConfig(cudaSharedMemConfig config);
```

其中 config可以是：

```
cudaSharedMemBankSizeDefault
cudaSharedMemBankSizeFourByte
cudaSharedMemBankSizeEightByte
```

不同的核函数启动之间，更改共享内存的配置，可能需要一个隐式的设备同步点，更改共享内存存储体的大小不会增加共享内存的使用，也不会影响内核函数的占用率，但其对性能可能有重大的影响。大的存储体可能有更高的带宽，大可能导致更多的冲突，要根据具体情况进行分析。

配置共享内存

每个SM上有64KB的片上内存，共享内存和L1共享这64KB，并且可以配置。CUDA为配置一级缓存和共享内存提供以下两种方法：

1. 按设备进行配置
2. 按核函数进行配置

配置函数：

```
cudaError_t cudaDeviceSetCacheConfig(cudaFuncCache cacheConfig);
```

其中配置参数如下：

```
cudaFuncCachePreferNone: no preference(default)
cudaFuncCachePreferShared: prefer 48KB shared memory and 16 KB L1 cache
cudaFuncCachePreferL1: prefer 48KB L1 cache and 16 KB shared memory
```

```
cudaFuncCachePreferEqual: prefer 32KB L1 cache and 32 KB shared memory
```

那种更好全看核函数：

1. 共享内存使用较多，那么更多的共享内存更好
2. 更多的寄存器使用，L1更多更好。

另一个函数是通过不同核函数自动配置的。

```
cudaError_t cudaFuncSetCacheConfig(const void* func,enum cudaFuncCacheca cheConfig);
```

这里的func是核函数指针，当我们调用某个核函数时，次核函数已经配置了对应的L1和共享内存，那么其如果和当前配置不同，则会重新配置，否则直接执行。
一级缓存和共享内存都在同一个片上，但是行为大不相同，共享内存靠的的是存储体来管理数据，而L1则是通过缓存行进行访问。我们对共享内存有绝对的控制权，但是L1的删除工作是硬件完成的。
GPU缓存比CPU的更难理解，GPU使用启发式算法删除数据，由于GPU使用缓存的线程更多，所以数据删除更频繁而且不可预知。共享内存则可以很好的被控制，减少不必要的误删造成的低效，保证SM的局部性。

同步

同步是并行的重要机制，其主要目的就是防止冲突。同步基本方法：

1. 障碍
2. 内存栅栏

障碍是所有调用线程等待其余调用线程达到障碍点。
内存栅栏，所有调用线程必须等到全部内存修改对其余线程可见时才继续进行。
有点蒙圈？没事，我们下来了解下理解这两个概念的预备知识。

弱排序内存模型

CUDA采用宽松的内存模型，也就是内存访问不一定按照他们在程序中出现的位置进行的。宽松的内存模型，导致了更激进的编译器。
一下这一点非常重要：

> GPU线程在不同的内存，比如SMEM，全局内存，锁页内存或对等设备内存中，写入数据的顺序是不一定和这些数据在源代码中访问的顺序相同，当一个线程的写入顺序对其他线程可见的时候，他可能

> 和写操作被执行的实际顺序不一致。
> 指令之间相互独立，线程从不同内存中读取数据的顺序和读指令在程序中的顺序不一定相同。
> 换句话说，核函数内连续两个内存访问指令，如果独立，其不一定哪个先被执行。
> 在这种混乱的情况下，为了可控，必须使用同步技术，否则真就是一千只脱了缰的哈士奇，万马奔腾的场景了。

显示障碍

CUDA中，障碍点设置在核函数中，注意这个指令只能在核函数中调用，并只对同一线程块内线程有效。

```
void __syncthreads();
```

1. __syncthreads()作为一个障碍点，他保证在同一线程块内所有线程没到达此障碍点时，不能继续向下执行。
2. 同一线程块内此障碍点之前的所有全局内存，共享内存操作，对后面的线程都是可见的。
3. 这个也就能解决同一线程块内，内存竞争的问题，同步，保证先后顺序，不会混乱。
4. 避免死锁情况出现，比如下面这种情况，就会导致内核死锁：

```
if (threadID % 2 == 0) {
    __syncthreads();
} else {
    __syncthreads();
}
```

5. 只能解决一个块内的线程同步，想做块之间的，只能通过核函数的执行和结束来进行块之间的同步。（把要同步的地方作为核函数的结束，来隐式的同步线程块）

内存栅栏

内存栅栏能保证栅栏前的内核内存写操作对栅栏后的其他线程都是可见的，有以下三种栅栏：块，网格，系统。

1. 线程块内：

```
void __threadfence_block();
```

保证同一块中的其他线程对于栅栏前的内存写操作可见

2. 网格级内存栅栏

```
1   void __threadfence();
```

挂起调用线程，直到全局内存中所有写操作对相同的网格内的所有线程可见

3. 系统级栅栏，夸系统，包括主机和设备，

```
1   void __threadfence_system();
```

挂起调用线程，以保证该线程对全局内存，锁页主机内存和其他设备内存中的所有写操作对全部设备中的线程和主机线程可见。

Volatile修饰符

volatile声明一个变量，防止编译器优化，防止这个变量存入缓存，如果恰好此时被其他线程改写，那就会造成内存缓存不一致的错误，所以volatile声明的变量始终在全局内存中。

总结

本文有点长，但是作为概览，我们后面都要围绕这篇展开，没有什么代码，纯理论的东西，多读多理解，多查资料多翻书。

相关文章

- 【Julia】整型和浮点型数字
- 【Julia】变量
- 【Julia】开始使用Julia