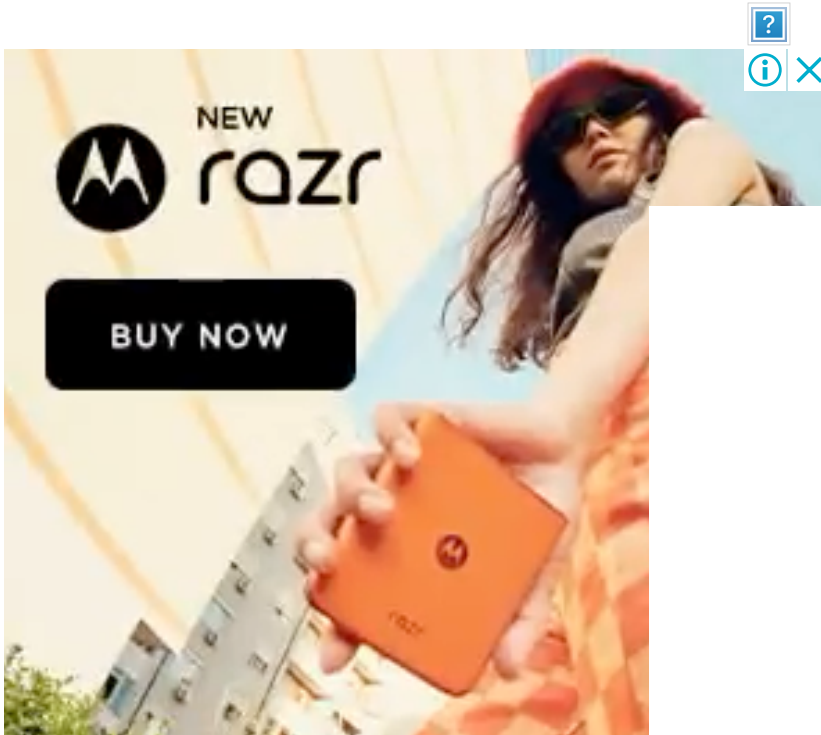




谭升的博客

人工智能基础



【CUDA 基础】5.6 线程束洗牌指令

📅 2018-06-06 | 📁 [CUDA](#) | [Freshman](#) | 💬 0 | 👁

Abstract: 本文介绍线程束洗牌指令的用法

Keywords: 线程束洗牌指令

线程束洗牌指令

前面介绍了共享内存，常量内存，只读内存的使用，今天我们来研究一个比较特殊的机制，名字也很特殊，叫做线程束洗牌指令。

支持线程束洗牌指令的设备最低也要3.0以上，

洗牌指令，shuffle instruction作用在线程束内，允许两个线程见相互访问对方的寄存器。这就给线程束内

的线程相互交换信息提供了了一种新的渠道，我们知道，核函数内部的变量都在寄存器中，一个线程束可以看做是32个内核并行执行，换句话说这32个核函数中寄存器变量在硬件上其实都是邻居，这样就为相互访问提供了物理基础，线程束内线程相互访问数据不通过共享内存或者全局内存，使得通信效率高很多，线程束洗牌指令传递数据，延迟极低，且不消耗内存

线程束洗牌指令是线程束内线程通讯的极佳方式。

我们先提出一个叫做束内线程的概念，英文名lane，简单的说，就是一个线程束内的索引，所以束内线程的ID在【0, 31】内，且唯一，唯一是指线程束内唯一，一个线程块可能有很多个束内线程的索引，就像一个网格中有很多相同的threadIdx.x一样，同时还有一个线程束的ID，可以通过以下方式计算线程在当前线程块内的束内索引，和线程束ID：

```
1 unsigned int LaneID=threadIdx.x%32;
2 unsigned int warpID=threadIdx.x/32;
```

根据上面的计算公式，一个线程块内的threadIdx.x=1,33,65等对应的laneID都是1

线程束洗牌指令的不同形式

线程束洗牌指令有两组：一组用于整形变量，另一种用于浮点型变量。一共有四种形式的洗牌指令。

在线程束内交换整形变量，其基本函数如下：

```
1 int __shfl(int var,int srcLane,int width=warpSize);
```

这个指令必须好好研究一下，因为这里的输入非常之乱，谁乱？var乱，一个int值，这个变量很明显是当前线程中的一个变量，作为输入，其传递给函数的并不是这个变量存储的值，而是这个变量名，换句话说，当前线程中有var这个变量，比如我们说1号线程的var值是1，那么2号线程中的var值不一定是1，所以，这个__shfl返回的就是var值，哪个线程var值呢？srcLane这个线程的，srcLane并不是当前线程的束内线程，而是结合with计算出来的相对线程位置，比如我想得到3号线程内存的var值，而且width=16，那么就是，0~15的束内线程接收0+3位置处的var值，也就是3号束内线程的var值，16~32的束内线程接收16+3=19位置处的var变量。

这个是非常重要的，虽然有些困难，但是却相当灵活。width的默认参数是32.srcLane我们后面简单的叫他束内线程，注意我们必须心理明白只有width是默认值的时候，他才是真正的束内线程。

图示如下

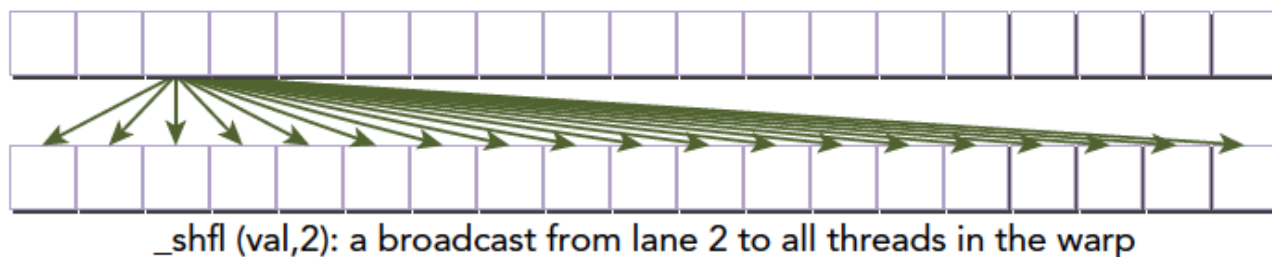
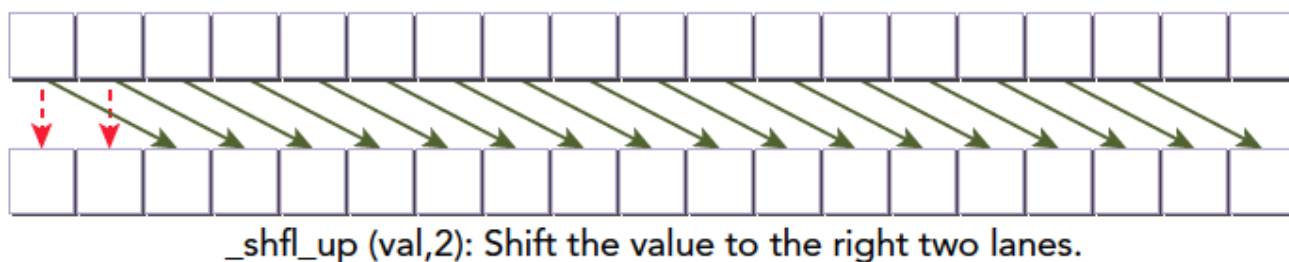


FIGURE 5-20

接着是另一个指令，其主要特点是从与调用线程相关的线程中复制数据。

```
1 int __shfl_up(int var,unsigned int delta,int with=warpSize);
```

这个函数的作用是调用线程得到当前束内线程编号减去delta的编号的线程内的var值，with和__shfl中都一样，默认是32，作用效果如下：



如果是with其他值，我们可以根据前面的讲解，把线程束再分成若干个大小为with的块，进行上图的操作。

最左边两个元素没有前面的delta号线程，所以不做任何操作，保持原值。

同样下一个指令是上面的反转版本：

```
1 int __shfl_down(int var,unsigned int delta,int with=warpSize);
```

作用和参数和up一模一样，图示如下：

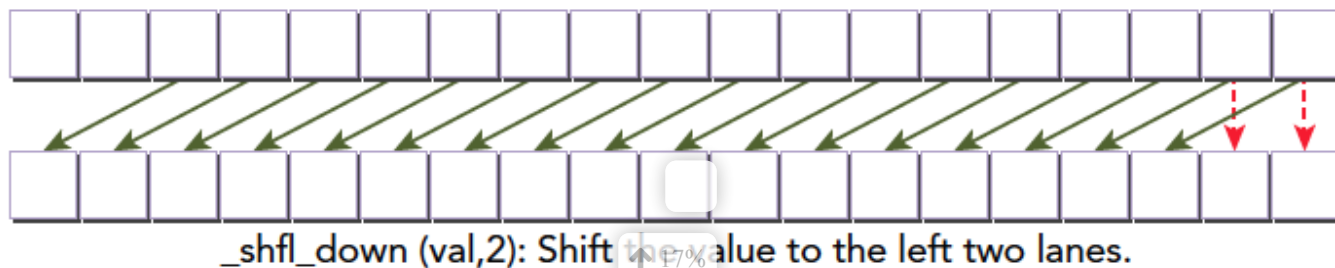


FIGURE 5-22

最后一个洗牌指令比较夸张，也是很灵活的一个指令

```
1  int __shfl_xor(int var,int laneMask,int with=warpSize);
```

xor是异或操作，这个指令如果学过硬件或者c语言学的比较扎实的人可能知道，这是电路里面最最最重要的操作，没有之一，什么是异或？逻辑中我们假设只有0，1两种信号，用“^”表示异或：

```
1  0^0=0;
2  1^0=1;
3  0^1=1;
4  1^1=0;
```

二元操作，只要两个不同就会得到真，否则为假

那么我们的__shfl_xor操作就是包含抑或操作在内的洗牌指令，怎么算呢？

如果我们输入的laneMask是1，其对应的二进制是 000...001 ,当前线程的索引是0~31之间的一个数，那么我们用laneMask与当前线程索引进行抑或操作得到的就是目标线程的编号了，这里laneMask是1，那么我们把1与0~31分别抑或就会得到：

```
1  000001^000000=000001;
2  000001^000001=000000;
3  000001^000010=000011;
4  000001^000011=000010;
5  000001^000100=000101;
6  000001^000101=000100;
7  .
8  .
9  .
10 000001^011110=011111;
11 000001^011111=011110;
```

这就是当前线程的束内线程编号和目标线程束内线程编号之间的对应关系，画成图会非常犀利：

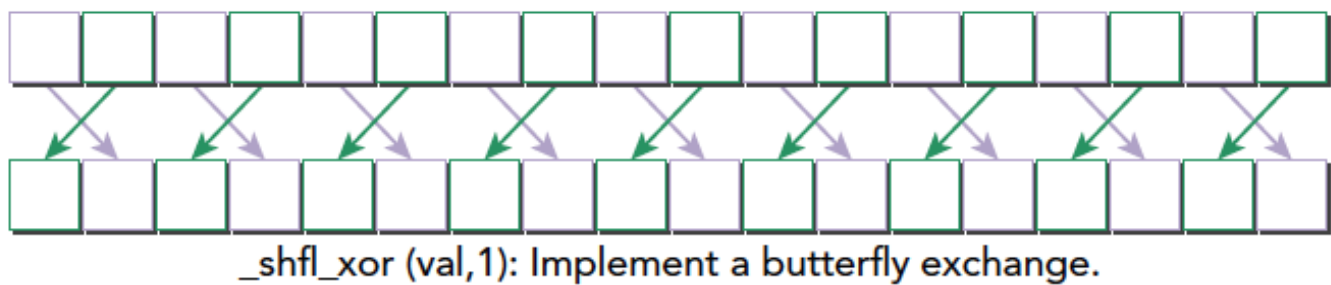


FIGURE 5-23

这就是4个线程束洗牌指令对整形的操作了。对应的浮点型不需要该函数名，而是只要把var改成float就行了，函数就会自动重载了。

线程束内的共享内存数据

接下来我们用代码实现以下，看看每一个指令的作用效果，洗牌指令可以用于下面三种整数变量类型中：

- 标量变量
- 数组
- 向量型变量

跨线程束值的广播

这个就是 `__shfl` 函数作用结果了，代码如下

```
1  __global__ void test_shfl_broadcast(int *in,int*out,int const srcLane)
2  {
3      int value=in[threadIdx.x];
4      value=__shfl(value,srcLane,BDIM);
5      out[threadIdx.x]=value;
6
7  }
```

这里面的过程就不用说了，注意var参数对应value就是我们要找的目标，srcLane这里是2，所以，我们取得了2号束内线程的value值给了当前线程，于是所有束内线程的value都是2了：

计算结果：

```
Tony — tony@tony-Lenovo: ~/Project/CUDA_Freshman/build/28_shfl_test — ssh tony@192.168.3.19 — 104x18
tony@tony-Lenovo:~/Project/CUDA_Freshman/build/28_shfl_test$ ./shfl_test 0
strating...
Using device 0: GeForce GTX 1050 Ti
test_shfl_broadcast
input:    0    1    2    3    4    5    6    7    8    9   10   11   12   13   14   15

output:    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2
tony@tony-Lenovo:~/Project/CUDA_Freshman/build/28_shfl_test$
```

线程束内上移

这里使用__shfl_up指令进行上移。代码如下

```
1  __global__ void test_shfl_up(int *in,int*out,int const delta)
2  {
3      int value=in[threadIdx.x];
4      value=__shfl_up(value,delta,BDIM);
5      out[threadIdx.x]=value;
6
7  }
```

运行结果：

```
Tony — tony@tony-Lenovo: ~/Project/CUDA_Freshman/build/28_shfl_test — ssh tony@192.168.3.19 — 104x18
tony@tony-Lenovo:~/Project/CUDA_Freshman/build/28_shfl_test$ ./shfl_test 1
strating...
Using device 0: GeForce GTX 1050 Ti
test_shfl_up
input:    0    1    2    3    4    5    6    7    8    9   10   11   12   13   14   15

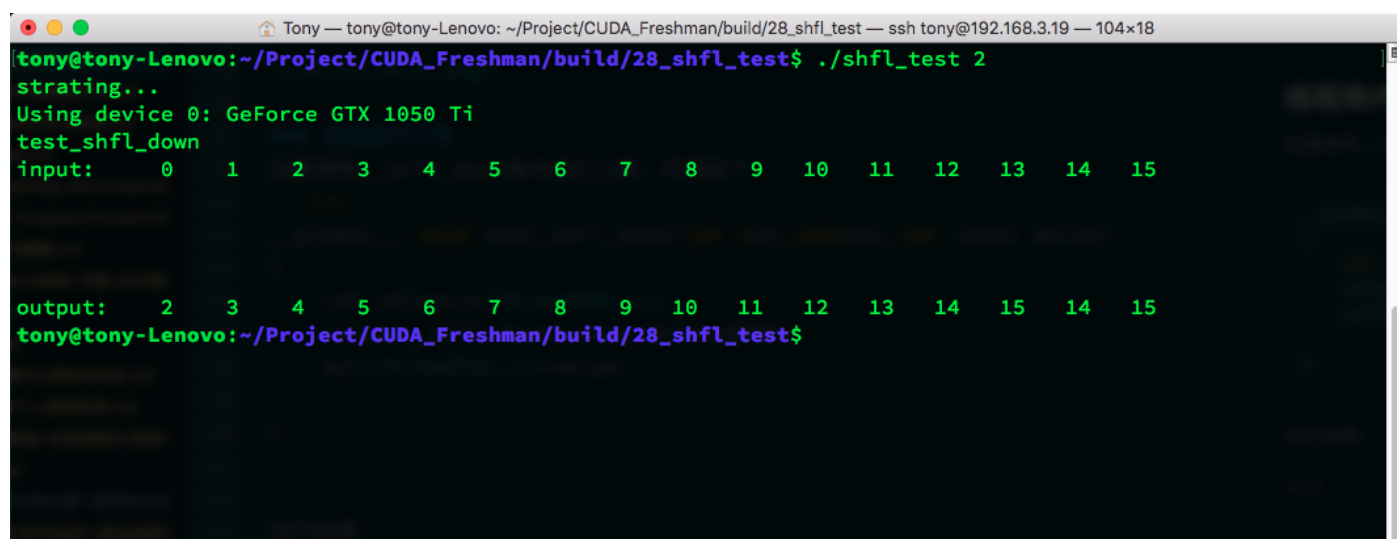
output:    0    1    0    1    2    3    4    5    6    7    8    9   10   11   12   13
tony@tony-Lenovo:~/Project/CUDA_Freshman/build/28_shfl_test$
```

线程束内下移

这里使用__shfl_down指令进行上移。代码如下

```
1  __global__ void test_shfl_down(int *in,int*out,int const delta)
2  {
3      int value=in[threadIdx.x];
4      value=__shfl_down(value,delta,BDIM);
5      out[threadIdx.x]=value;
6
7  }
```

运行结果：



```
Tony — tony@tony-Lenovo: ~/Project/CUDA_Freshman/build/28_shfl_test — ssh tony@192.168.3.19 — 104x18
tony@tony-Lenovo:~/Project/CUDA_Freshman/build/28_shfl_test$ ./shfl_test 2
strating...
Using device 0: GeForce GTX 1050 Ti
test_shfl_down
input:    0    1    2    3    4    5    6    7    8    9   10   11   12   13   14   15

output:   2    3    4    5    6    7    8    9   10   11   12   13   14   15   14   15
tony@tony-Lenovo:~/Project/CUDA_Freshman/build/28_shfl_test$
```

线程束内环绕移动

然后是循环移动，我们修改__shfl中的参数，把静态的目标改成一个动态的目标，如下：

```
1  __global__ void test_shfl_wrap(int *in,int*out,int const offset)
2  {
3      int value=in[threadIdx.x];
4      value=__shfl(value,threadIdx.x+offset,BDIM);
5      out[threadIdx.x]=value;
6
7  }
```

当offset=2的时候，得到结果：

```
Tony — tony@tony-Lenovo: ~/Project/CUDA_Freshman/build/28_shfl_test — ssh tony@192.168.3.19 — 104x18
tony@tony-Lenovo:~/Project/CUDA_Freshman/build/28_shfl_test$ ./shfl_test 3
strating...
Using device 0: GeForce GTX 1050 Ti
test_shfl_wrap
input:      0      1      2      3      4      5      6      7      8      9     10     11     12     13     14     15

output:      2      3      4      5      6      7      8      9     10     11     12     13     14     15      0      1
tony@tony-Lenovo:~/Project/CUDA_Freshman/build/28_shfl_test$
```

前14个元素的值是可以预料到的，但是14号，15号并没有像shfl_down那样保持不变，而是获得了0号和1号的值，那么我们有必要相信，shfl中计算目标线程编号的那步有取余操作，对with取余，我们真正得到的数据来自

```
1  srcLane=srcLane%width;
```

这样就说的过去了，同理我们通过将srcLane设置成-2的话就能得到对应的向上的环绕移动。

跨线程束的蝴蝶交换

接着我们看看__shfl_xor像我说的这个操作非常之灵活，可以组合出任何你想要的到的变换，我们先来个简单的就是我们上面讲原理的时候得到的结论：

```
1  __global__ void test_shfl_xor(int *in,int*out,int const mask)
2  {
3      int value=in[threadIdx.x];
4      value=__shfl_xor(value,mask,BDIM);
5      out[threadIdx.x]=value;
6  }
```

mask我们设置成1，然后就能得到下面的结果：


```
Tony — tony@tony-Lenovo: ~/Project/CUDA_Freshman/build/28_shfl_test — ssh tony@192.168.3.19 — 104x18
tony@tony-Lenovo:~/Project/CUDA_Freshman/build/28_shfl_test$ ./shfl_test 4
strating...
Using device 0: GeForce GTX 1050 Ti
test_shfl_xor
input:  0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
output: 1  0  3  2  5  4  7  6  9  8 11 10 13 12 15 14
tony@tony-Lenovo:~/Project/CUDA_Freshman/build/28_shfl_test$
```

忍不住画了个叉，哈哈

这些都是预料之中的，接着我们看点高级的。也解释下为什么说可以操作数组，好吧我之前也蒙了。

跨线程束交换数组值

我们要交换数组了，假如线程内有数组，然后我们交换数组的位置，我们可以用下面代码实现一个简单小数组的例子：

```
1  __global__ void test_shfl_xor_array(int *in,int*out,int const mask)
2  {
3      //1.
4      int idx=threadIdx.x*SEGM;
5      //2.
6      int value[SEGM];
7      for(int i=0;i<SEGM;i++)
8          value[i]=in[idx+i];
9      //3.
10     value[0]=__shfl_xor(value[0],mask,BDIM);
11     value[1]=__shfl_xor(value[1],mask,BDIM);
12     value[2]=__shfl_xor(value[2],mask,BDIM);
13     value[3]=__shfl_xor(value[3],mask,BDIM);
14     //4.
15     for(int i=0;i<SEGM;i++)
16         out[idx+i]=value[i];
17
18 }
```

有逻辑的地方代码就会变得复杂，我们从头看，首先我们定义了一个宏SEGM为4，然后每个线程束包含一

个SEGM大小的数组，当然，这些数据数存在寄存器中的，如果数组过大可能会溢出到本地内存中，不用担心，也在片上，这个数组比较小，寄存器足够了。

我们看每一步都做了什么

1. 计算数组的起始地址，因为我们的输入数据是一维的，每个线程包含其中长度为SEGM的一段数据，所以，这个操作就是计算出当前线程对应的数组的起始位置
2. 声明数组，在寄存器中开辟地址，这句编译时就会给他们分配地址，然后从全局内存中读数据。
3. 计算当前线程中数组中的元素，与要交换的目标的线程的之间的抑或，此时mask为1，那么就相当于将多个寄存器变量进行跨线程束的蝴蝶交换
4. 将寄存器内的交换结果写会到全局内存

这个看起来有点复杂，但是其实就是把上面的蝴蝶交换重复执行。

```
Tony — tony@tony-Lenovo: ~/Project/CUDA_Freshman/build/28_shfl_test — ssh tony@192.168.3.19 — 104x18
tony@tony-Lenovo:~/Project/CUDA_Freshman/build/28_shfl_test$ ./shfl_test 6
strating...
Using device 0: GeForce GTX 1050 Ti
test_shfl_swap
input:  0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15

output:  7  1  2  3  4  5  6  0 15  9 10 11 12 13 14  8
tony@tony-Lenovo:~/Project/CUDA_Freshman/build/28_shfl_test$
```

大蝴蝶~

跨线程束不是跨越线程束，而是横跨当前线程束的意思，这个标题有点让人迷惑。

跨线程束使用数组索引交换数值

接下来这个是个扩展了，交换了两个之间的一对值，并且这里是我们第一次写设备函数，也就是只能被核函数调用的函数：

```
1  __inline__ __device__
2  void swap(int *value,int laneIdx,int mask,int firstIdx,int secondIdx)
3  {
4      bool pred=((laneIdx%(2))==0);
5      if(pred)
```

```

6      {
7          int tmp=value[firstIdx];
8          value[firstIdx]=value[secondIdx];
9          value[secondIdx]=tmp;
10
11     }
12     value[secondIdx]=__shfl_xor(value[secondIdx],mask,BDIM);
13     if(pred)
14     {
15         int tmp=value[firstIdx];
16         value[firstIdx]=value[secondIdx];
17         value[secondIdx]=tmp;
18     }
19 }
20
21 __global__ void test_shfl_swap(int *in,int* out,int const mask,int firstIdx,int sec
22 {
23     //1.
24     int idx=threadIdx.x*SEGM;
25     int value[SEGM];
26     for(int i=0;i<SEGM;i++)
27         value[i]=in[idx+i];
28     //2.
29     swap(value,threadIdx.x,mask,firstIdx,secondIdx);
30     //3.
31     for(int i=0;i<SEGM;i++)
32         out[idx+i]=value[i];
33
34 }

```

这个过程有点复杂，这里面每一句指令的意思都很明确，而且与上面数组交换类似

1. 和上面数组交换类似，不赘述
2. 交换数组内的first和second，然后xor在second位置的元素，然后再次重新交换first和second的元素
3. 写入全局变量。

2的描述看起来简单，但是实际上比较麻烦，我们画个图来表示：

LandID	⁰	¹	²	³
	0	4	8	12
	1	5	9	13
	2	6	10	14
	3	7	11	15

交换first和second的元素

3	4	11	12
1	5	9	13
2	6	10	14
0	7	8	15

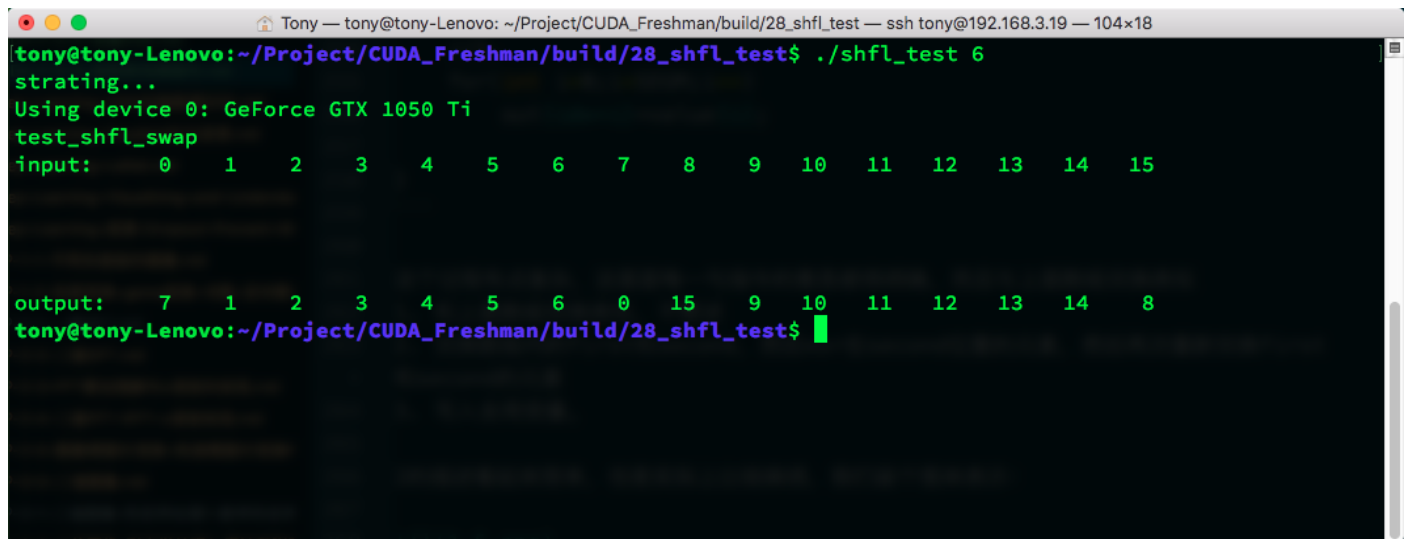
xor

3	4	11	12
1	5	9	13
2	6	10	14
7	0	15	8

交换first和second的元素

7	4	15	12
1	5	9	13
2	6	10	14
3	0	11	8

对照代码每一步变换的过程都画了绿线，所以看起来还是好理解的，运行结果：



```
Tony — tony@tony-Lenovo: ~/Project/CUDA_Freshman/build/28_shfl_test — ssh tony@192.168.3.19 — 104x18
tony@tony-Lenovo:~/Project/CUDA_Freshman/build/28_shfl_test$ ./shfl_test 6
strating...
Using device 0: GeForce GTX 1050 Ti
test_shfl_swap
input:      0   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15

output:     7   1   2   3   4   5   6   0  15   9  10  11  12  13  14   8
tony@tony-Lenovo:~/Project/CUDA_Freshman/build/28_shfl_test$
```

使用线程束洗牌指令的并行规约

前面我们已经很详细的介绍过归约算法了，从线程块之间到线程间的归约我们都进行了研究，包括使用共享内存以及各种展开方式，今天我们使用线程束洗牌指令完成归约，主要目标就是减少线程间数据传递的延迟，达到更快的效率：

我们主要考虑三个层面的归约：

- 线程束级归约
- 线程块级归约
- 网格级归约

一个线程块有过个线程束，每个执行自己的归约，每个线程束不适用共享内存，而是使用线程束洗牌指令，代码如下：

```
1  __inline__ __device__ int warpReduce(int localSum)
2  {
3      localSum += __shfl_xor(localSum, 16);
4      localSum += __shfl_xor(localSum, 8);
5      localSum += __shfl_xor(localSum, 4);
6      localSum += __shfl_xor(localSum, 2);
7      localSum += __shfl_xor(localSum, 1);
8      return localSum;
9  }
```

```

10  __global__ void reduceShfl(int * g_idata,int * g_odata,unsigned int n)
11  {
12      //set thread ID
13      __shared__ int smem[DIM];
14      unsigned int idx = blockDim.x*blockIdx.x+threadIdx.x;
15      //convert global data pointer to the
16      //1.
17      int mySum=g_idata[idx];
18      int laneIdx=threadIdx.x%warpSize;
19      int warpIdx=threadIdx.x/warpSize;
20      //2.
21      mySum=warpReduce(mySum);
22      //3.
23      if(laneIdx==0)
24          smem[warpIdx]=mySum;
25      __syncthreads();
26      //4.
27      mySum=(threadIdx.x<DIM)?smem[laneIdx]:0;
28      if(warpIdx==0)
29          mySum=warpReduce(mySum);
30      //5.
31      if(threadIdx.x==0)
32          g_odata[blockIdx.x]=mySum;
33
34  }

```

代码解释:

1. 从全局内存读取数据，计算线程束ID和当前线程的束内线程ID
2. 计算当前线程束内的归约结果，使用的xor，这里需要动手计算下每个线程和这些2的幂次计算的结果因为每个线程束只有32个线程，所以二进制最高位就是16，那么xor 16 是计算0+16，1+17，2+18，这些位置的和,计算完成后前16位是结果，16到31是一样结果，重复了一边，同理xor 8是计算0+8，1+9，2+10,...,前8位结果有效，后面是复制前面的答案，最后就得到当前线程束的归约结果。
3. 然后把线程束结果存储到共享内存中
4. 然后继续2中的过程计算3中得到的数据，完整的重复
- 5.将最后结果存入全局内存

```
Tony — tony@tony-Lenovo: ~/Project/CUDA_Freshman/build/29_reduce_shfl — ssh tony@192.168.3.19 — 104x18
tony@tony-Lenovo:~/Project/CUDA_Freshman/build/29_reduce_shfl$ ./reduce_shfl
Using device 0: GeForce GTX 1050 Ti
    with array size 16777216
grid 16384 block 1024
cpu reduce      elapsed 0.039166 ms cpu_sum: 2138687736
gpu warmup      elapsed 0.005259 ms
reduceGmem      elapsed 0.004199 ms gpu_sum: 2138687736
reduceSmem      elapsed 0.002601 ms gpu_sum: 2138687736
reduceShfl      elapsed 0.001507 ms gpu_sum: 2138687736
tony@tony-Lenovo:~/Project/CUDA_Freshman/build/29_reduce_shfl$
```

其他几个核函数前面文章都介绍过，我们通过实践可以看出使用线程束洗牌指令进行的归约效率最高。主要原因是使用寄存器进行数据交换而不需要任何位置的内存介入。

本文完整的代码在github:https://github.com/Tony-Tan/CUDA_Freshman (欢迎随手star 😊)

总结

本文介绍线程束洗牌指令的一些用法，其吸引人的地方就是不需要通过内存进行线程间数据交换，具有非常高的性能。

至此我们已经完成了第五章的学习，后面我们进入流和事件相关知识的学习。

本文作者：谭升

本文链接：<https://face2ai.com/CUDA-F-5-6-线程束洗牌指令/>

版权声明：本博客所有文章除特别声明外，均采用 [CC BY-NC-SA 4.0](#) 许可协议。转载请注明出处！

相关文章

- [【Julia】整型和浮点型数字](#)
- [【Julia】变量](#)
- [【Julia】开始使用Julia](#)

线程束洗牌指令

帮帮点一点 (不用付费哒~)