GPGPU—Parallel Reduction

0.What & Why & How

Reduce：给一组数据，一个满足结合律的二元操作符⊕，那么reduce可以表示为：

$$\sum_i a_i = a_0 \oplus a_1 \oplus a_2 \oplus a_3 \oplus a_4 \oplus a_5 \oplus a_6 \oplus a_7$$

举个例子，在单线程下计算一个Reduce：1+2+3+4 = 10，只要顺序执行就可以了。但是有两个线程要怎么并行处理呢？

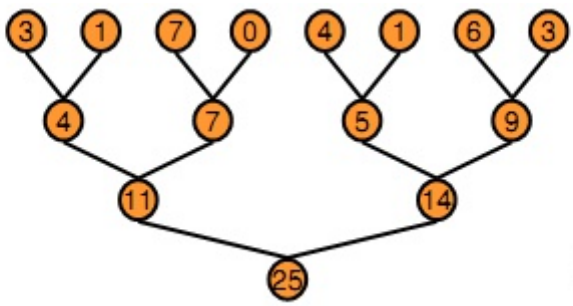
可以使用两个线程，第一个线程计算1+2，第二个线程计算3+4，两个线程一起执行，最后得到的两个结果再由线程1计算，这样并行计算要比顺序计算快很多。

看一张图：

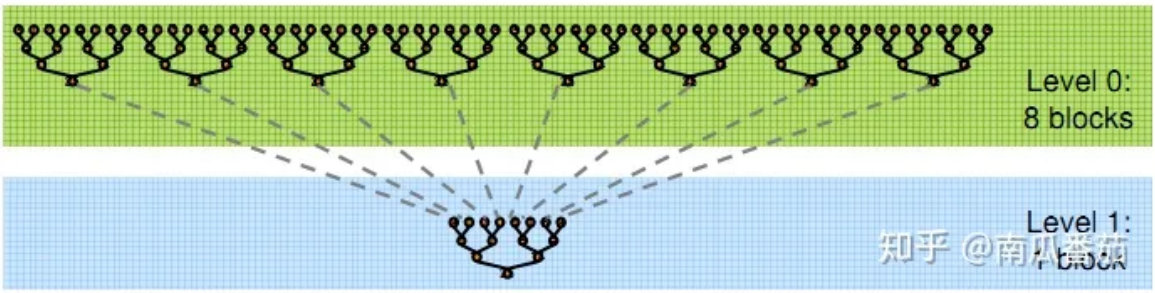
```
Input:      1   7   4   0   9   4   8   8   2   4   5   5   1   7   1   1   5   2   7   6

Reduce progress by pass:
    1:      1   7   4   0   9   4   8   8   2   4   +
            5   5   1   7   1   1   5   2   7   6
            6  12   5   7  10   5  13  10   9  10

    2:      6  12   5   7  10   +
            5  13  10   9  10
           11  25  15  16  20


    3:     11  25  15   +
               16  20
           11  41  35

    4:     11  41   +
               35
           11  76

    4:     11   +
           76
           87
```


一共20个输入数据，十个线程同步处理。

在每个线程块中，使用基于树结构的方法来执行

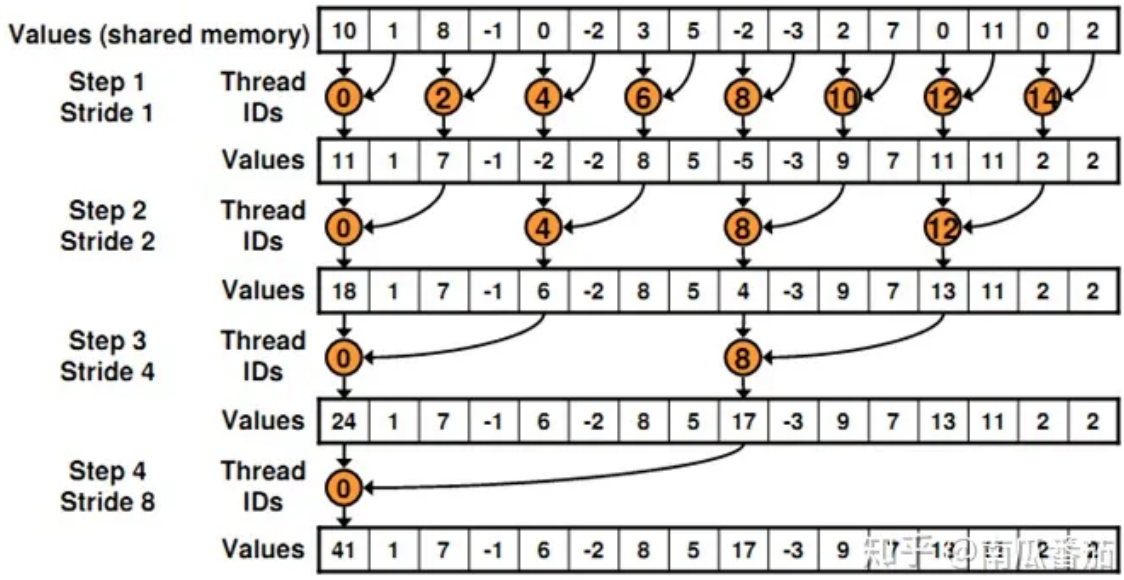


每个线程块中都是对部分数据的reduce，多个线程块之间如何通信呢？使用kernel递归，如下图：



通过调用多次kernel来分解计算，来避免全局同步

Reduction #1: Interleaved Addressing



伪代码如下：

```
#pragma kernel ParallelReduction
#define THREADTOTAL 64
//一个Group之间的线程共享的内存
groupshared float sharedMem[THREADTOTAL];
StructuredBuffer<float> Source;
RWStructuredBuffer<float> Result;
//SV_DispatchThreadID 为当前线程所在的线程组在Dispatch中位置*一个线程组的大小+当前线程在线程组中的位置
//SV_GroupIndex 为当前Group按照行展开后，当前线程在一维数组中的位置
//SV_GroupID 为当前Group在Dispatch中的位置
[numthreads(THREADTOTAL,1,1)]
void ParallelReduction(uint3 Gid : SV_GroupID,uint3 DTid : SV_DispatchThreadID,uint GI : SV_GroupIndex)
{
    sharedMem[GI] =Source[DTid.x]; // store in shared memory
    //该方法会强制所有组内线程同步，
    GroupMemoryBarrierWithGroupSync(); // wait until everything is transfered from device memory to shared memory

    for (uint s = 1; s < THREADTOTAL; s *= 2)  // stride: 1, 2, 4, 8, 16, 32, 64, 128
    {
        if (GI%(2*s) == 0)
            sharedMem[index] += sharedMem[index + s];
```

```
        GroupMemoryBarrierWithGroupSync();
    }
    // Have the first thread write out to the output
    if (GI == 0){
        // write out the result for each thread group
                Result[Gid.x] = sharedMem[0];
    }
}
```

举个例子，我们需要计算256个数据的和。每个Group是[64,1,1],发射4个Dispatch[4，1，1]。

按照上述算法，四个线程组。

1. 第一个线程组

   Gid = (0,0,0), DTid = (0,0,0)*(64,1,1)+(x,0,0),GI = x。X是一个线程组中的第几个线程，从0开始。

   sharedMem 中的数据是 所有数据中的前64个。

   等到前64个数据全部Load完毕，开始循环计算。第0个线程会计算0+1，第二个计算1+2一直到把这64个数的和计算出来。但是这样中间的奇数线程会空闲起来。

2. 第二三四组线程和第一组一样

但是这样有一个问题，即GPU一个Warp中的所有线程是SIMT工作方式，即同一个Group中的线程执行同一个指令，取不同的数据。刚才的算法奇数线程会空闲下来，同时奇数线程和偶数线程进入了不同的分支，在SIMT结构下，两个分支都会执行。这就是**warp divergence**。同时%操作消耗很大。

Reduction #2: Interleaved Addressing

解决上述问题。

```
//将循环改成
for(unsigned int s = 1; s< THREADTOTAL; s*=2)
{
    int index = 2*s*GI;
    if(index < THREADTOTAL)
    {
        sharedMem[index] += sharedMem[index + s];
    }
    GroupMemoryBarrierWithGroupSync();
}
```

如此一来，第一个线程计算 0+1，第二线程计算2+3，第三个计算4+5，依次类推，每个线程都参与了运算。

并且取消了**%**运算。同时几乎所有的线程都走了同一个分支，缓解了warp divergence。

现在还有一个问题需要解决，即线程之间Bank冲突的问题。

Bank

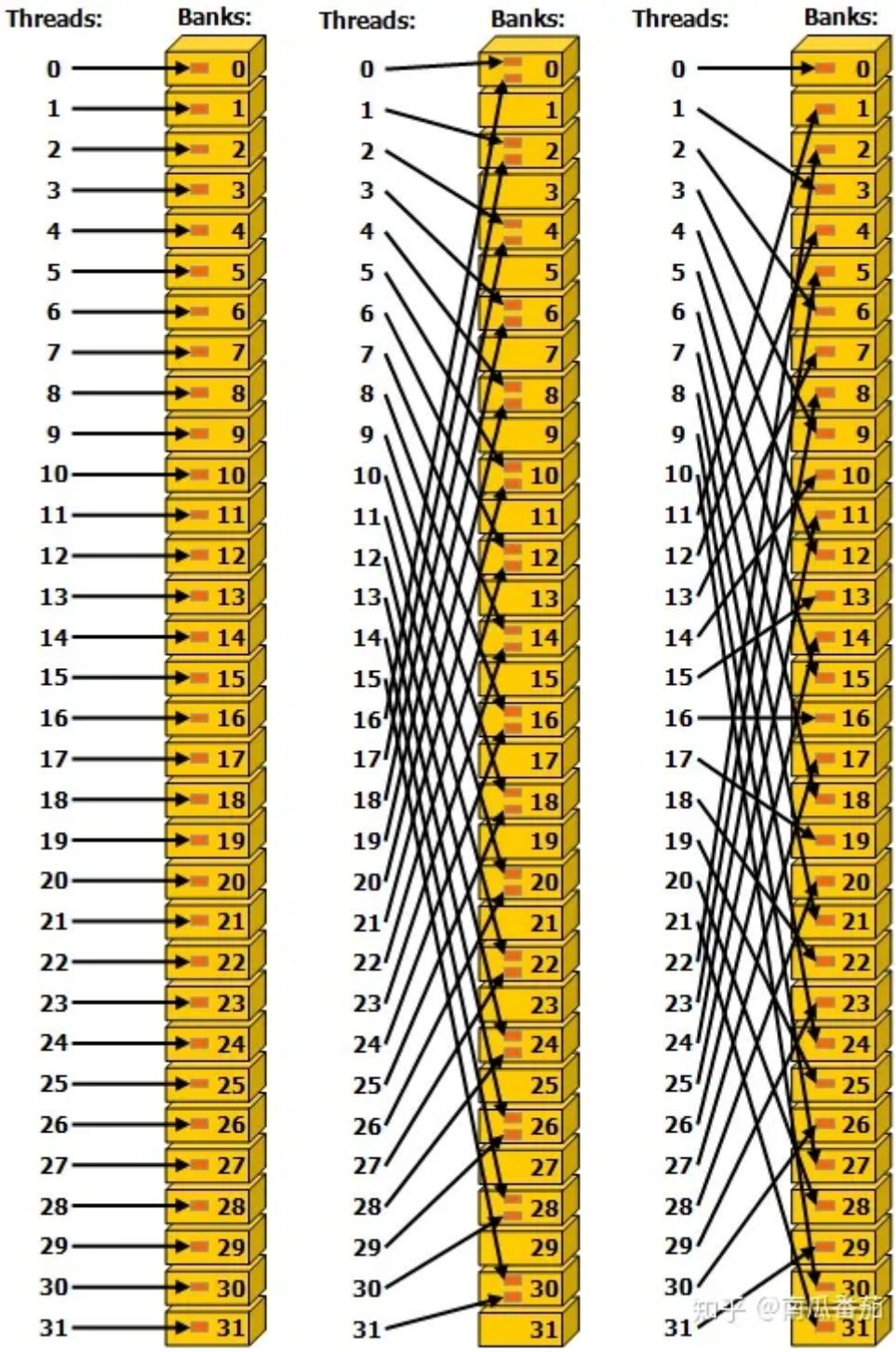
Shared Memoyr在SM中，对同一个Block内的所有线程共享。但是一个warp能处理的线程为32个，为了高效率的利用共享内存，GPU会把共享内存划分为一个一个的bank，每个Bank内再划分不同的地址，在线程访问时，0-31共享32个Bank。但是当一个Block内的线程超过32个时，Bank的访问就会从0开始循环。

Address:	0	1	2	3	4	…	31	32	33	34	35	…
Bank:	0	1	2	3	4	…	31	0	1	2	3	…

Bank Conflict

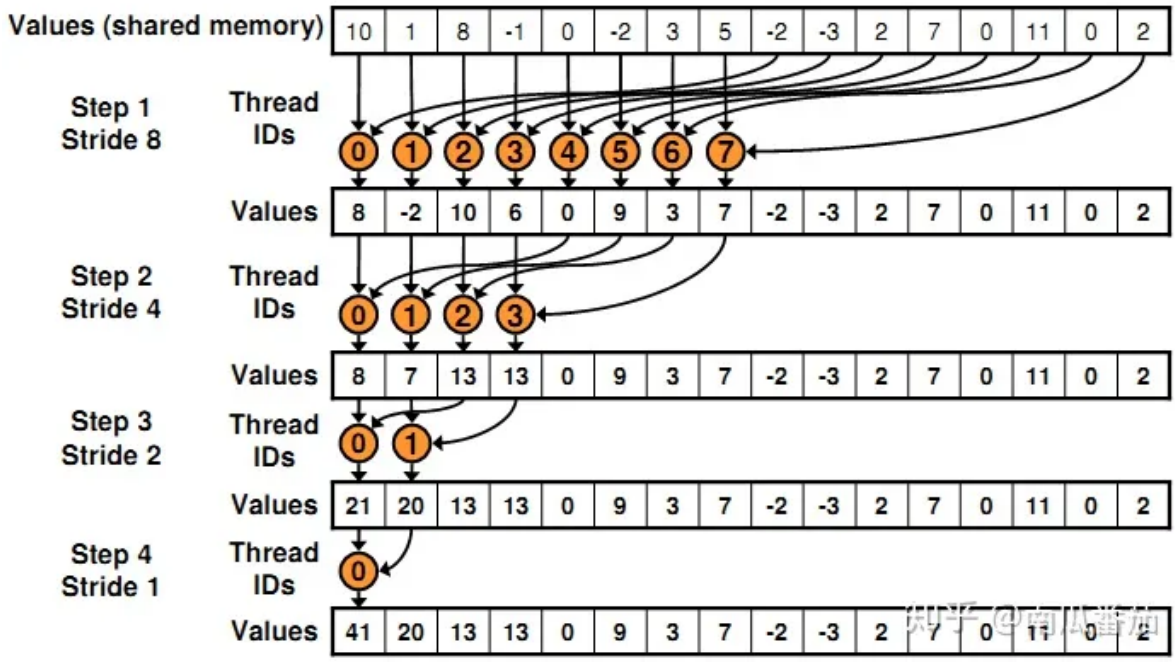
简单来说，bank conflict 指的是当**同一个warp**里的**多个线程**对**同一个bank**发出访问请求，只能进行串行访问，极大影响了并行效率。当bank conflict发生时，硬件将该访问请求拆分成多个conflict-free的请求，拆分出来的个数n，就说这个访问请求会引起 **n-way bank conflicts**。随后cuda引入**broadcast**机制，使得对**同一个bank**的**同一个地址**进行访问时，不会产生bank conflict, 一定程度上缓解了bank conflict，但对**同一个bank里**的**不同地址**进行访问时，bank conflict 问题仍然存在。

上面算法的Bank的访问与下图中间的流程一样



Reduction #3: Sequential Addressing

重新规划对shared memory中数据的访问逻辑如下：



代码如下：

```
for(int s = THREADTOTAL/2;s>0s>>=1)
{
    if(GI < s)
    {
        sharedMem[GI] += sharedMem[GI + s];
    }
    GroupMemoryBarrierWithGroupSync();
}
```

如此以来，线程1的数据访问从0，1变成了，0，32。

后续还可以再进行优化，如展开循环等，具体内容看文章把。

> 线程之间共享彼此的数据可以参考Reading Between The Threads: Shader Intrinsics.