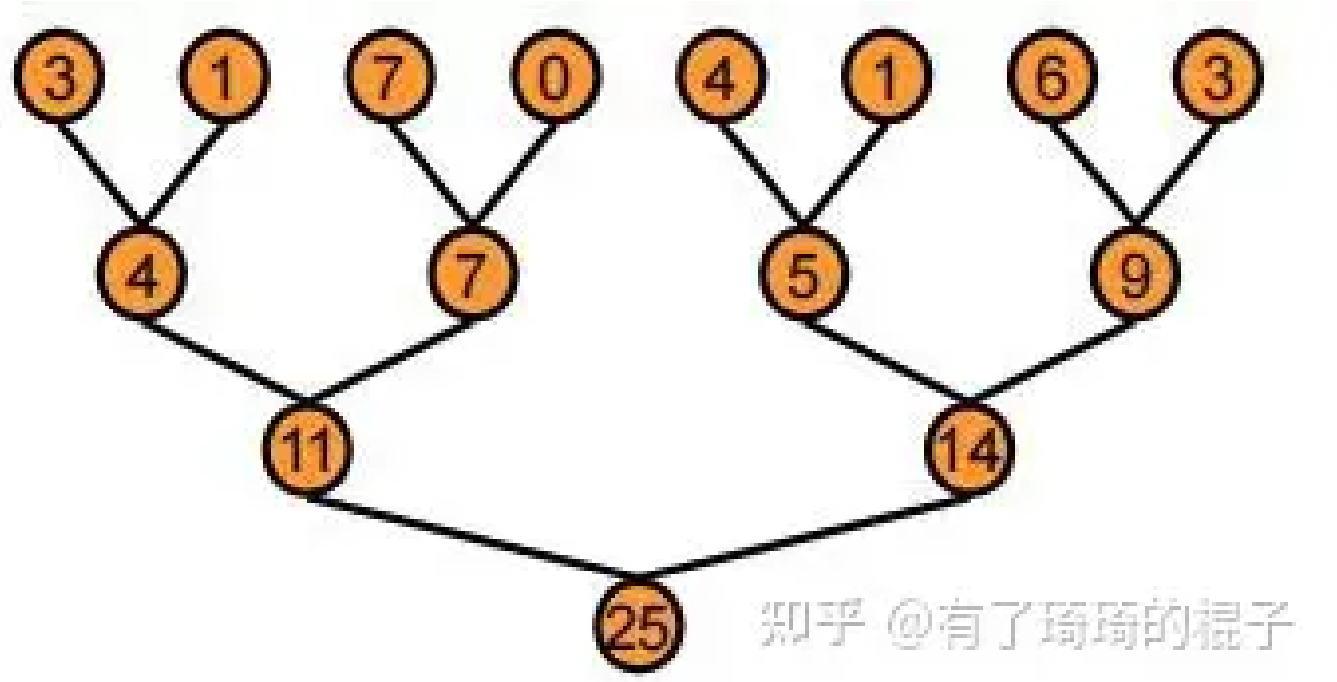
本篇文章主要是介绍**如何对GPU中的reduce算法进行优化**。目前针对reduce的优化，Nvidia的官方文档reduce优化已经说得比较详细，但是过于精简，很多东西一笔而过。对于初入该领域的新人而言，理解起来还是较为费劲。因而在官方文档的基础，进行更深入地说明和讲解，尽可能地让每一个读者通过此文都能彻底地了解reduce的优化技术。

前言

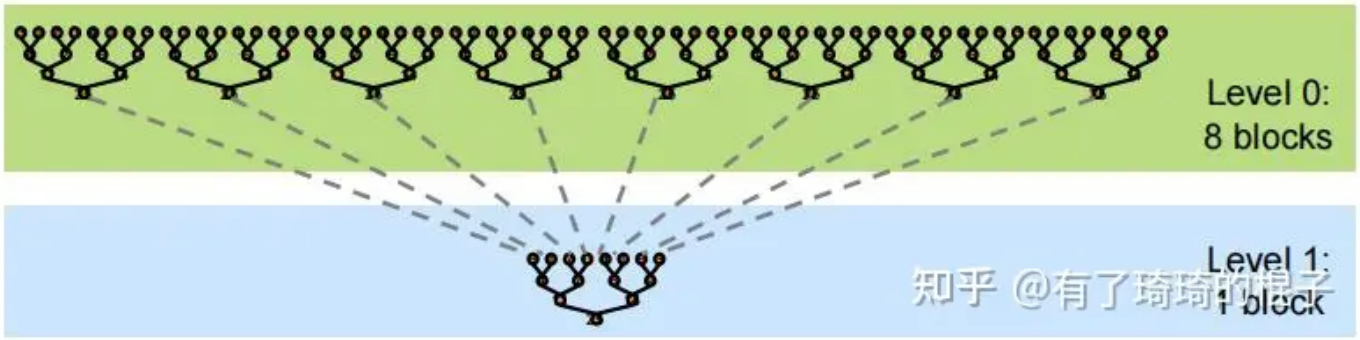
首先需要对reduce算法进行介绍。reduce算法本质上就是计算
。下面本文将详细说明如何在GPU中实现reduce算法并进行深入地优化。

并行算法设计

在GPU中，reduce采用了一种树形的计算方式。如下图所示。



从上至下，将数据不断地累加，直到得出最后的结果，即25。但由于GPU没有针对global数据的同步操作，只能针对block的数据进行同步。所以，一般而言将reduce分为两个阶段，其示意图如下：



我们仔细来看看这个事，假设给定一个长度为N的数组，需要计算该数组的所有元素之和。首先需要将数组分为m个小份。而后，在第一阶段中，开启m个block计算出m个小份的reduce值。最后，在第二阶段中，使用一个block将m个小份再次进行reduce，得到最终的结果。由于第二阶段本质上是可以调用第一个阶段的kernel，所以不做单独说明，本文只是探索**第一阶段**的优化技巧。

所以kernel接口为：

```
__global__ void reduce(T *input, T* output)
```

其中，input代表输入的数组，即一个长度为N的数组，output代表输出数组，即第一阶段的结果，即长度为M的数组。随后要开始激动人心的coding阶段，但在CUDA编程中，我们首先需要设置三个参数:

BlockNum：即开启的block数量，即上面所说的M，代表需要将数组切分为几份。
Thread_per_block:每个block中开启的线程数，一般而言，取128，256，512，1024这几个参数会比较多。
Num_per_block:每个block需要进行reduce操作的长度。

其中，BlockNum* Num_per_block=N
三个参数的示意图如下：

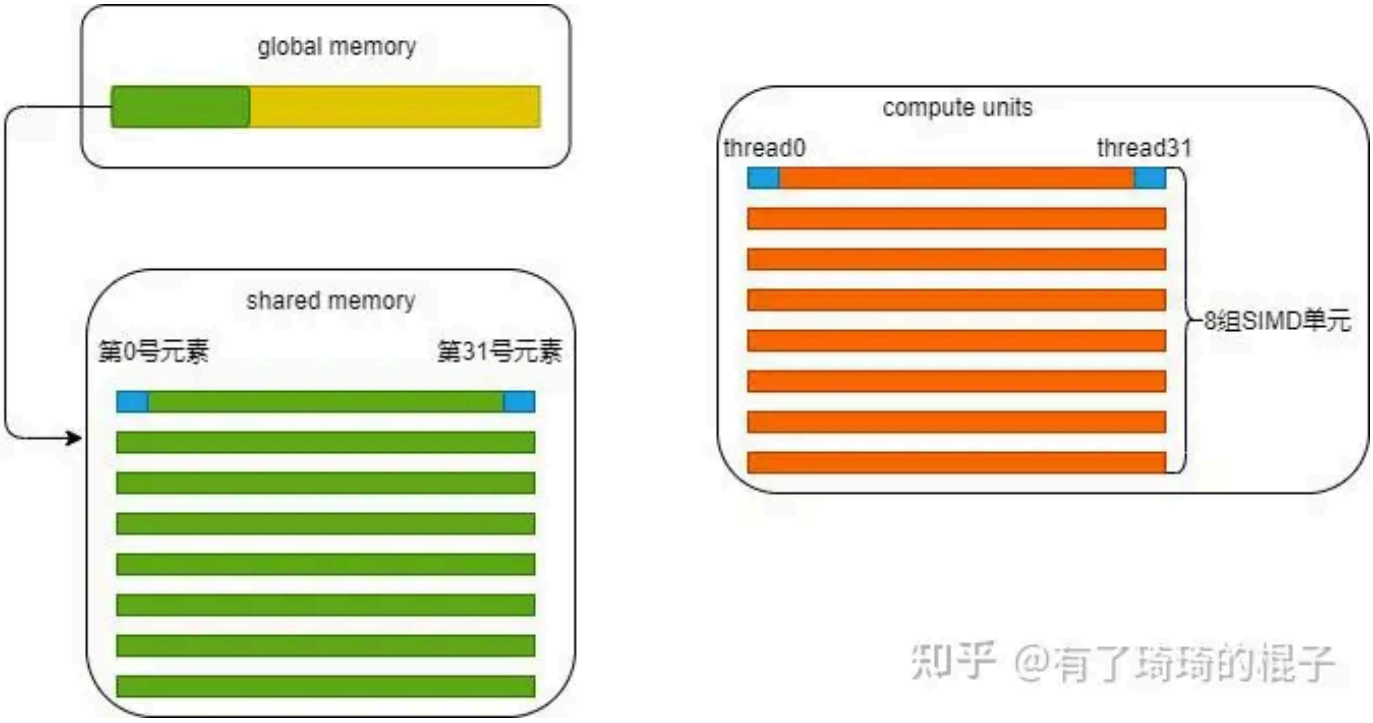
reduce baseline算法介绍

Baseline算法比较简单，分为三个步骤。第一个步骤是将数据load至shared memory中，第二个步骤是在shared memory中对数据进行reduce操作，第三个步骤是将最后的结果写回global memory中。代码如下：

```
__global__ void reduce0(float *d_in,float *d_out){
    __shared__ float sdata[THREAD_PER_BLOCK];

    //each thread loads one element from global memory to shared mem
    unsigned int i=blockIdx.x*blockDim.x+threadIdx.x;
    unsigned int tid=threadIdx.x;
    sdata[tid]=d_in[i];
    __syncthreads();

    // do reduction in shared mem
    for(unsigned int s=1; s<blockDim.x; s*=2){
        if(tid%(2*s) == 0){
            sdata[tid]+=sdata[tid+s];
        }
        __syncthreads();
    }

    // write result for this block to global mem
    if(tid==0)d_out[blockIdx.x]=sdata[tid];
}
```

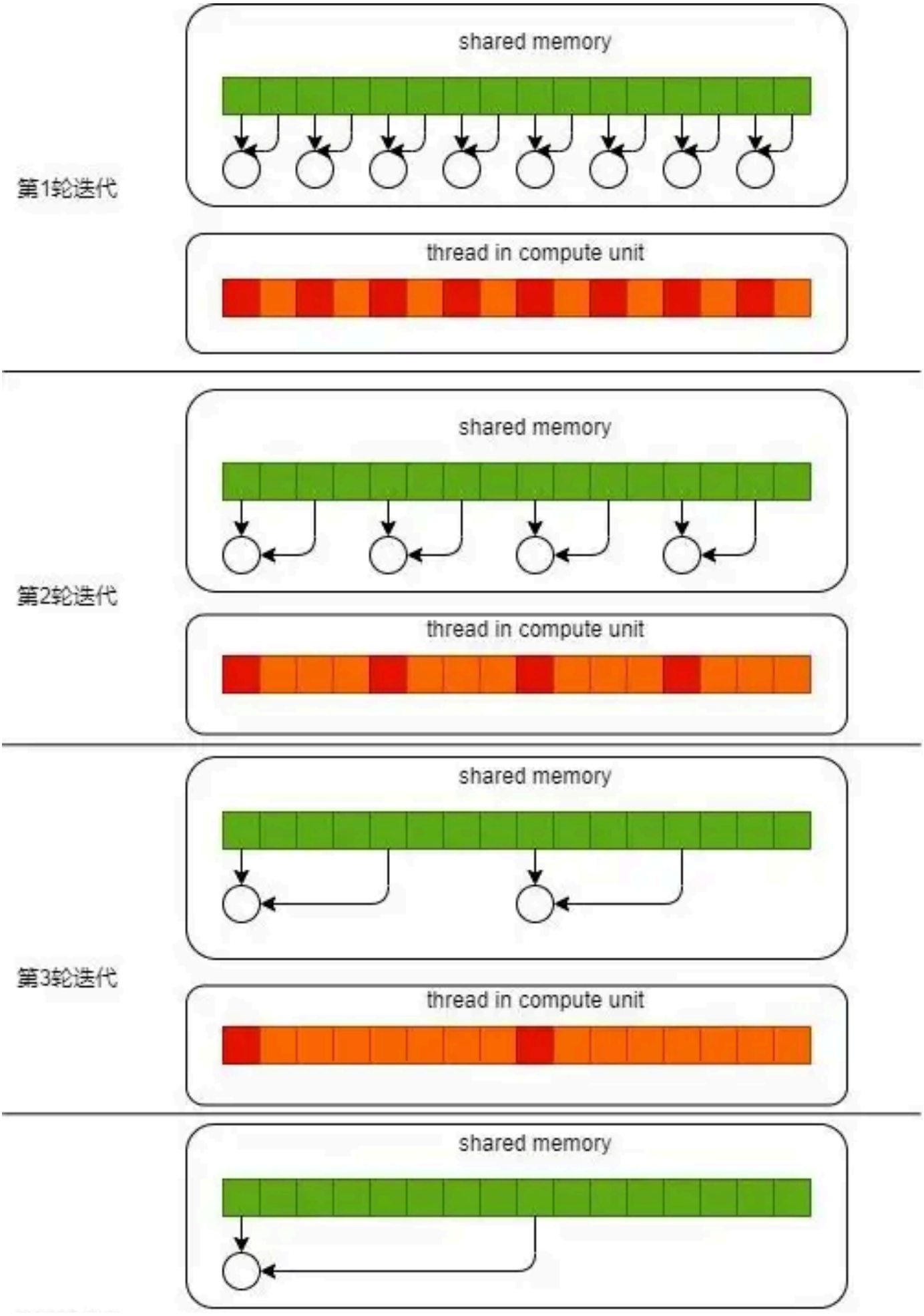
在进行优化之前，我们需要再来好好地梳理一下这个baseline代码。优化的本质是通过软件榨干硬件资源，所以必须清楚地了解代码在硬件上的执行过程才能更好地进行优化。因此，本节将花较多的篇幅说明**代码和硬件的对应关系**，为后续的优化打好基础。

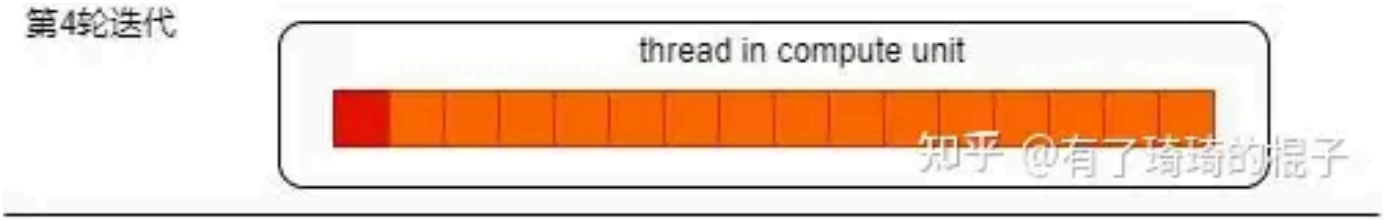
在**第一个步骤**中，我们让Num_per_block与Thread_per_block一致，每个block设定为256个线程，一个block负责256个数据的reduce工作。假设需要处理32M的数据，则有128K个block。tid代表线程号，i代表在原始数组中的索引号。第tid号线程将第i号的数据从global中取出，放到shared memory的第tid元素中。比如在第0号block中，0号线程将0号元素取出，放到shared memory的第0号位置。示意图见：



从硬件角度来分析一下代码。为了执行代码，GPU需要分配两种资源，一个是**存储资源**，一个是**计算资源。存储资源**包括在global memory中分配的一块32M　sizeof(float)的空间以及在shared memory中分配的256　sizeof(float)的空间。需要注意的是，**shared memory存在bank冲突的问题，因而需要格外小心。 计算资源**其实是根据thread数量来确定的，一个block中分配256个thread线程，32个线程为一组，绑定在一个SIMD单元。所以256个线程可以简单地理解为分配了8组SIMD单元。（但实际的硬件资源分配不是这样，因为一个SM的计算资源有限，不可能真的给每一个block都分配这么多的SIMD单元。）总而言之，在第一个阶段，就是tid号线程将i号数据从global memory中取出，再放进shared memory中，严谨一点的话，中间是走一遍寄存器再到shared memory中的。

到了**第二个阶段**，block中需要计算的256个元素已经全部被存储在了shared memory中，此时需要对其进行reduce操作。这个过程需要进行多轮迭代，在第一轮迭代中，如果tid%2 ==0, 则第tid号线程将shared memory中第tid号位置的值和第tid+1号的值进行相加，而后放在第tid号位置。在第二轮迭代中，如果tid%4==0,则第tid号线程将shared memory中第tid号位置的值和第tid+2号的值进行相加，而后放在第tid号位置。不断迭代，则所有元素都将被累加到第0号位置。其示意图如下。其中，红色的线程代表符合if条件的线程，只有它们有任务，需要干活。


shared memory
第1轮迭代
thread in compute unit
shared memory
第2轮迭代
thread in compute unit
shared memory
第3轮迭代
thread in compute unit
shared memory



在**第三个阶段**中，block负责的256个元素之和都放置在shared memory的0号位置，此时，只需要将0号位置的元素写回即可。

实验结果

-	耗时(ns)	带宽(GB/s)	加速比
reduce0	804,187	159.2	-

在本次实验中，采用的GPU是V100。耗时结果是使用nsight测出来的。在V100 中global的带宽是900GB/s。可以看出，其带宽利用率较差，reduce0版本存在着较大的改进空间。

优化技巧1：解决warp divergence

现有问题

目前reduce0存在的最大问题就是**warp divergent**的问题。对于一个block而言，它所有的thread都是执行同一条指令。如果存在if-else这样的分支情况的话，thread会执行所有的分支。只是不满足条件的分支，所产生的结果不会记录下来。可以在上图中看到，在每一轮迭代中都会产生两个分支，分别是红色和橙色的分支。这严重影响了代码执行的效率。

解决方式

解决的方式也比较明了，就是尽可能地让所有线程走到同一个分支里面。
代码示意如下：

```
__global__ void reduce1(float *d_in,float *d_out){
    __shared__ float sdata[THREAD_PER_BLOCK];

    //each thread loads one element from global memory to shared mem
    unsigned int i=blockIdx.x*blockDim.x+threadIdx.x;
```

```
    unsigned int tid=threadIdx.x;
    sdata[tid]=d_in[i];
    __syncthreads();

    // do reduction in shared mem
    for(unsigned int s=1; s<blockDim.x; s*=2){
        int index = 2*s*tid;
        if(index < blockDim.x){
            sdata[index]+=sdata[index+s];
        }
        __syncthreads();
    }

    // write result for this block to global mem
    if(tid==0)d_out[blockIdx.x]=sdata[tid];
}
```

虽然代码依旧存在着if语句，但是却与reduce0代码有所不同。我们继续假定block中存在256个thread，即拥有256/32=8个warp。当进行**第1次迭代**时，0-3号warp的index<blockDim.x， 4-7号warp的index>=blockDim.x。对于每个warp而言，都只是进入到一个分支内，所以并不会存在warp divergence的情况。当进行**第2次迭代**时，0、1号两个warp进入计算分支。当进行**第3次迭代**时，只有0号warp进入计算分支。当进行**第4次迭代**时，只有0号warp的前16个线程进入分支。此时开始产生warp divergence。通过这种方式，我们消除了前3次迭代的warp divergence。

实验结果

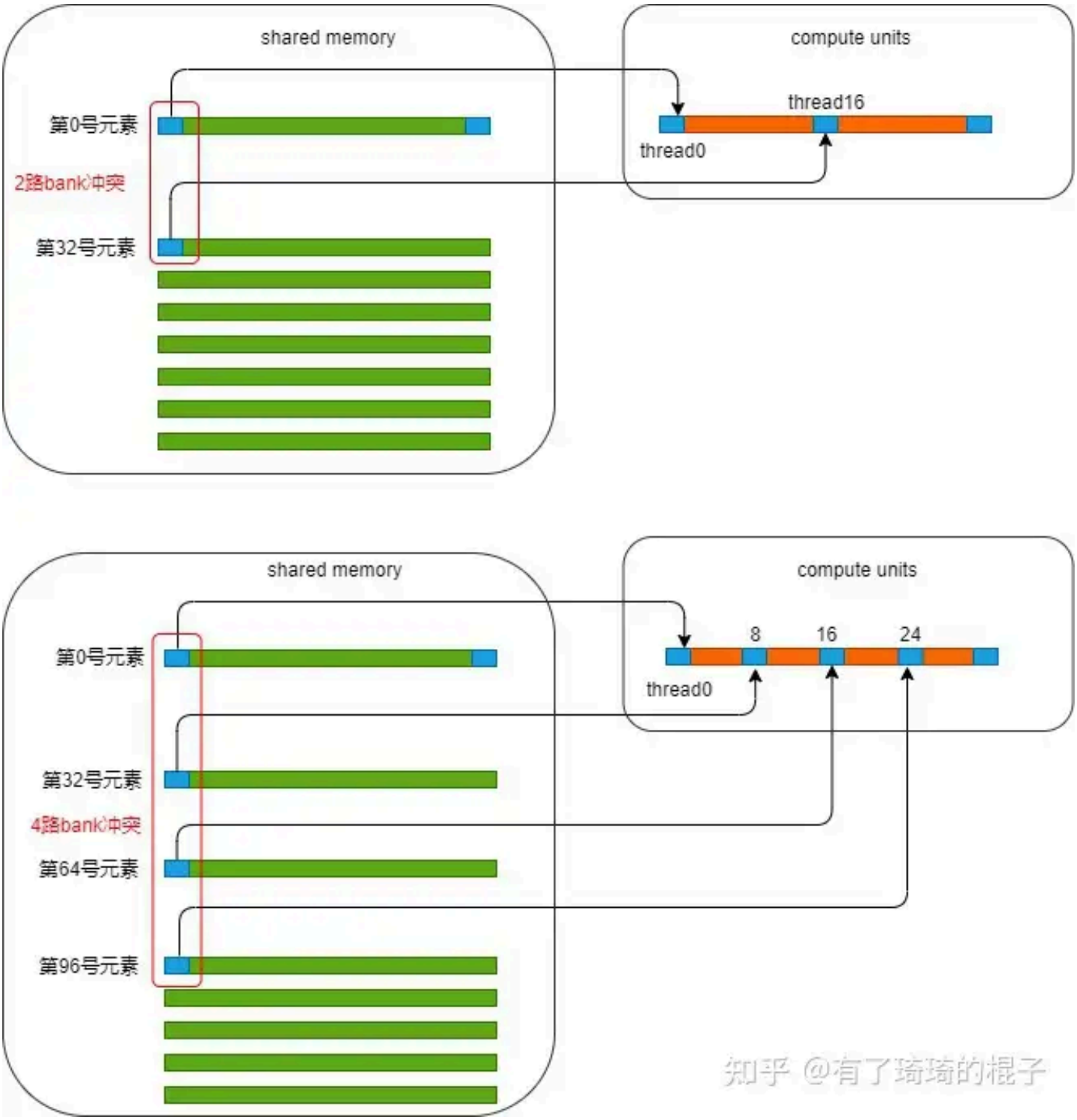
-	耗时(ns)	带宽(GB/s)	加速比
reduce0	804,187	159.2	-
reduce1	506,365	252.8	1.59x

优化技巧2：解决bank冲突

现有问题

reduce1的最大问题是**bank冲突**。我们把目光聚焦在这个for循环中。并且只聚焦在**0号warp**。在**第一次迭代**中，0号线程需要去load shared memory的0号地址以及1号地址的数，然后写回到0号地址。而

此时，这个warp中的16号线程，需要去load shared memory中的32号地址和33号地址。可以发现，0号地址跟32号地址产生了**2路的bank冲突**。在**第2次迭代**中，0号线程需要去load shared memory中的0号地址和2号地址。这个warp中的8号线程需要load shared memory中的32号地址以及34号地址，16号线程需要load shared memory中的64号地址和68号地址，24号线程需要load shared memory中的96号地址和100号地址。又因为0、32、64、96号地址对应着同一个bank，所以此时产生了**4路的bank冲突**。现在，可以继续算下去，8路bank冲突，16路bank冲突。由于bank冲突，所以reduce1性能受限。下图说明了在load第一个数据时所产生的bank冲突。



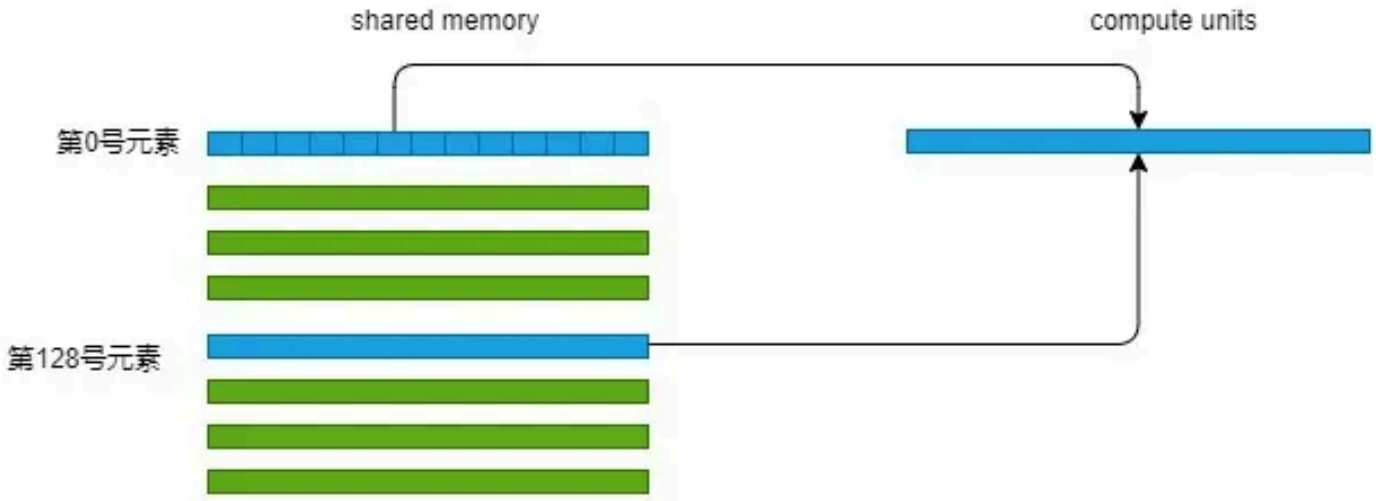
解决方式

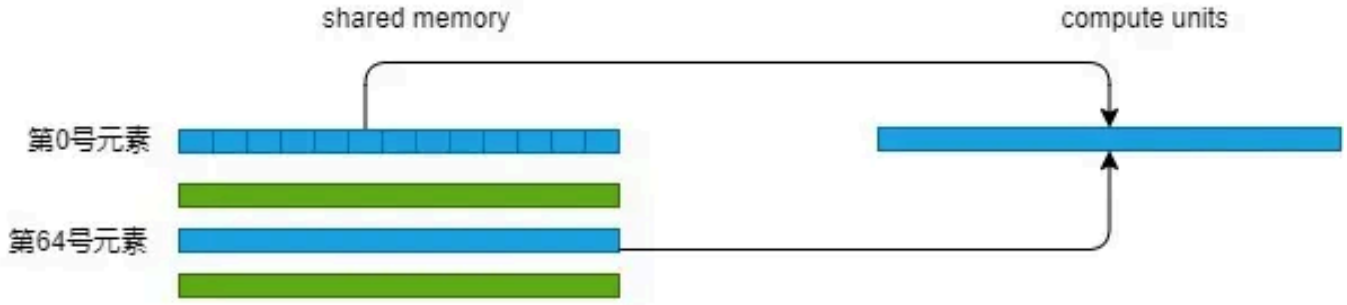
在reduce中，解决bank冲突的方式就是把for循环逆着来。原来stride从0到256，现在stride从128到0。其伪代码如下：

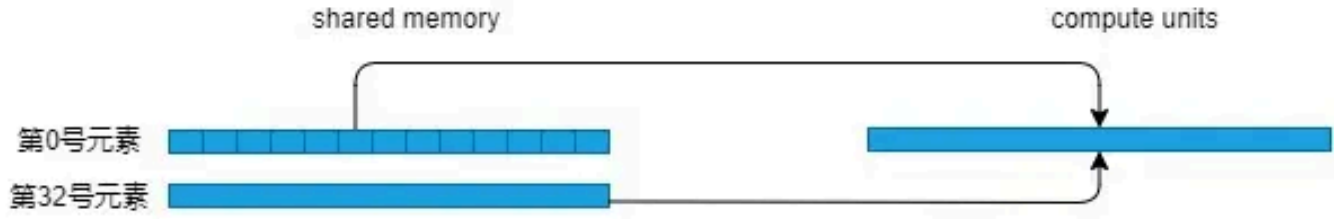
```
__global__ void reduce2(float *d_in,float *d_out){
    __shared__ float sdata[THREAD_PER_BLOCK];

    //each thread loads one element from global memory to shared mem
    unsigned int i=blockIdx.x*blockDim.x+threadIdx.x;
    unsigned int tid=threadIdx.x;
    sdata[tid]=d_in[i];
    __syncthreads();

    // do reduction in shared mem
    for(unsigned int s=blockDim.x/2; s>0; s>>=1){
        if(tid < s){
            sdata[tid]+=sdata[tid+s];
        }
        __syncthreads();
    }

    // write result for this block to global mem
    if(tid==0)d_out[blockIdx.x]=sdata[tid];
}
```

那为什么通过这么一个小小的改变就能消除bank冲突呢，我们继续进行分析。

把目光继续看到这个for循环中，并且只分析0号warp。0号线程需要load shared memory的0号元素以及128号元素。1号线程需要load shared memory中的1号元素和129号元素。这一轮迭代中，在读取第一个数时，warp中的32个线程刚好load 一行shared memory数据。再分析第2轮迭代，0号线程load 0号元素和64号元素，1号线程load 1号元素和65号元素。咦，也是这样，每次load shared memory的一行。再来分析第3轮迭代，0号线程load 0号元素和32号元素，接下来不写了，总之，一个warp load shared memory的一行。没有bank冲突。到了4轮迭代，0号线程load 0号元素和16号元素。那16号线程呢，16号线程啥也不干，因为s=16，16-31号线程啥也不干，跳过去了。示意图如下：


shared memory
compute units
第0号元素
第128号元素


shared memory
compute units
第0号元素
第64号元素


shared memory
compute units
第0号元素
第32号元素


shared memory
compute units
第0号元素
知乎 @有了琦琦的棍子

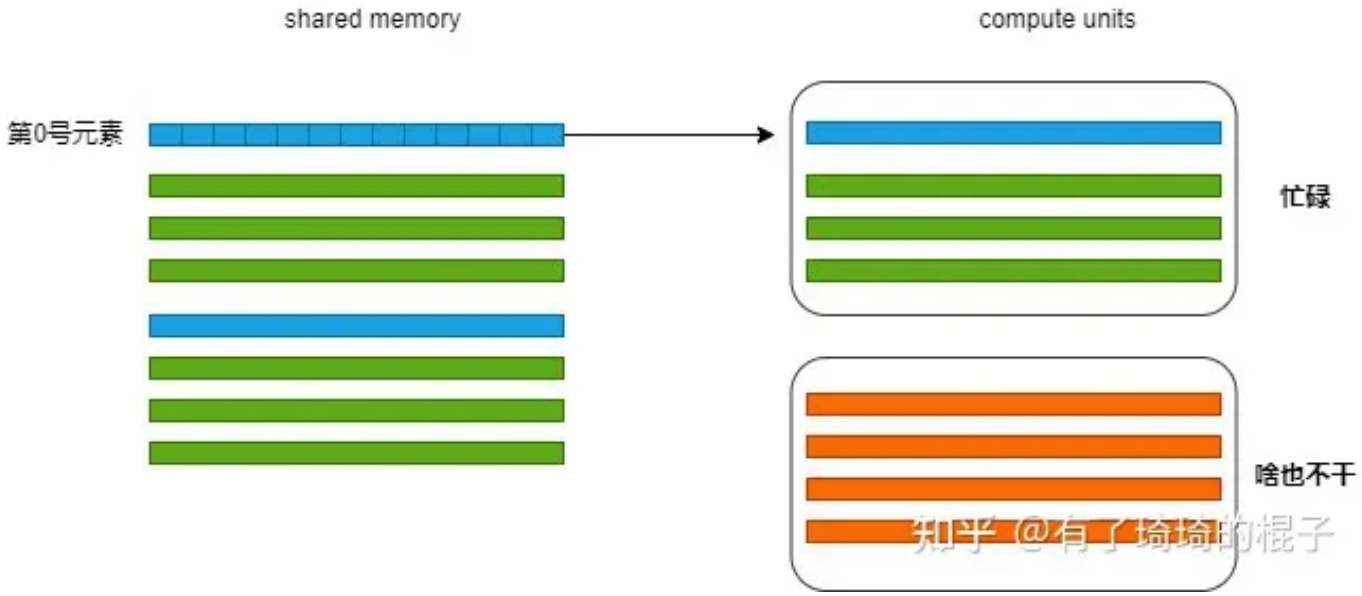
实验结果

-	耗时(ns)	带宽(GB/s)	加速比
reduce0	804,187	159.2	-
reduce1	506,365	252.8	1.59x
reduce2	397,886	321.7	1.27x

优化技巧3：解决idle线程

现有问题

reduce2最大的问题就是线程的浪费。可以看到我们启动了256个线程，但是在第1轮迭代时只有128个线程在干活，第2轮迭代只有64个线程在干活，每次干活的线程都会减少一半。第一轮迭代示意图如下，只有前128个线程在load数据。后128个线程啥也不干，光看着。



解决方式

对于HPC从业者而言，我们希望变成GPU的资本家，去尽可能地压榨GPU。但是呢，在这里，每一次迭代有一半的线程不干活。而且，128-255号线程最过分，它娘的，没有任何贡献，啥也不干。想来想去，能不能让它们干点活呢。想来想去，那这样吧，让它好歹做一次加法。除了去global memory中取数外，再做一次加法。当然为了实现这个，block数就得改一改了。Block数量减少，Num_per_block增加一倍。也就是说原来一个block只需要管256个数就行，现在得管512个数了。代码如下：

```
__global__ void reduce3(float *d_in,float *d_out){
    __shared__ float sdata[THREAD_PER_BLOCK];

    //each thread loads one element from global memory to shared mem
    unsigned int i=blockIdx.x*(blockDim.x*2)+threadIdx.x;
    unsigned int tid=threadIdx.x;
    sdata[tid]=d_in[i] + d_in[i+blockDim.x];
    __syncthreads();

    // do reduction in shared mem
    for(unsigned int s=blockDim.x/2; s>0; s>>=1){
        if(tid < s){
            sdata[tid]+=sdata[tid+s];
        }
        __syncthreads();
    }

    // write result for this block to global mem
    if(tid==0)d_out[blockIdx.x]=sdata[tid];
}
```

通过这种方式，将一些idle的线程给利用起来了。

实验结果

-	耗时(ns)	带宽(GB/s)	加速比
reduce0	804,187	159.2	-
reduce1	506,365	252.8	1.59x
reduce2	397,886	321.7	1.27x
reduce3	213,182	600.4	1.86x

这里面多说一句，让idle线程利用起来的这个加速比其实有点出乎意料。

优化技巧4：展开最后一维减少同步

现有问题

对于reduce3来说，性能已经算是比较好了。但是依旧没有达到我们想要的效果。我们再来仔细地看看还有什么可以改进的地方。我们发现，当进行到最后几轮迭代时，此时的block中只有warp0在干活时，线程还在进行**同步**操作。这一条语句造成了极大的浪费。

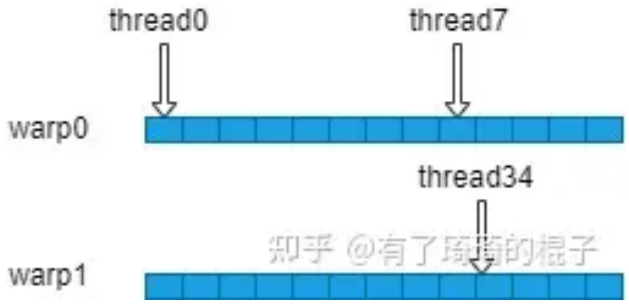
解决方式

由于一个warp中的32个线程其实是在一个SIMD单元上，这32个线程每次都是执行同一条指令，这天然地保持了同步状态，因而当s=32时，即只有一个SIMD单元在工作时，完全可以将__syncthreads()这条同步代码去掉。所以我们将最后一维进行展开以减少同步。伪代码如下：

```
__device__ void warpReduce(volatile float* cache,int tid){
    cache[tid]+=cache[tid+32];
    cache[tid]+=cache[tid+16];
    cache[tid]+=cache[tid+8];
    cache[tid]+=cache[tid+4];
    cache[tid]+=cache[tid+2];
    cache[tid]+=cache[tid+1];
}
```

```
__global__ void reduce4(float *d_in,float *d_out){
    __shared__ float sdata[THREAD_PER_BLOCK];

    //each thread loads one element from global memory to shared mem
    unsigned int i=blockIdx.x*(blockDim.x*2)+threadIdx.x;
    unsigned int tid=threadIdx.x;
    sdata[tid]=d_in[i] + d_in[i+blockDim.x];
    __syncthreads();

    // do reduction in shared mem
    for(unsigned int s=blockDim.x/2; s>32; s>>=1){
        if(tid < s){
            sdata[tid]+=sdata[tid+s];
        }
        __syncthreads();
    }

    // write result for this block to global mem
    if(tid<32)warpReduce(sdata,tid);
    if(tid==0)d_out[blockIdx.x]=sdata[tid];
}
```

可以通过下面的示意图更好地了解，（图画的有点丑，比例也不太对，大家将就着看看。）warp0会被绑定在一个SIMD单元上，上面有thread0-thread31。warp1会被绑在另外一个SIMD单元上，上面有thread32-thread63。由于在一个SIMD单元上，然后不管啥时候thread0和thread7肯定是同一状态，不需要同步。而thread0和thread34就不能保证同步，必须用__syncthreads()来保证同步操作。



实验结果

-	耗时(ns)	带宽(GB/s)	加速比
reduce0	804,187	159.2	-
reduce1	506,365	252.8	1.59x
reduce2	397,886	321.7	1.27x
reduce3	213,182	600.4	1.86x
reduce4	169,215	756.4	1.26x

在做到这一步时，带宽已经到了756GB/s。这个时候就已经优化地差不多了。性能也很难做到有效提升了。

优化技巧5：完全展开减少计算

现有问题

其实到了这一步，reduce的效率已经足够高了。再进一步优化其实已经非常困难了。为了探索极致的性能表现，Mharris接下来给出的办法是**对for循环进行完全展开**。我觉得这里主要是减少for循环的开销。Mharris的实验表明这种方式有着1.41x的加速比。但是用的机器是G80，十几年前的卡。性能数据也比较老了，至于能不能真的有这么好的加速比，我们拭目以待。

解决方法

我们将整个for循环进行展开，非常暴力，代码如下：

```
template <unsigned int blockSize>
__device__ void warpReduce(volatile float* cache,int tid){
    if(blockSize >= 64)cache[tid]+=cache[tid+32];
```

```
    if(blockSize >= 32)cache[tid]+=cache[tid+16];
    if(blockSize >= 16)cache[tid]+=cache[tid+8];
    if(blockSize >= 8)cache[tid]+=cache[tid+4];
    if(blockSize >= 4)cache[tid]+=cache[tid+2];
    if(blockSize >= 2)cache[tid]+=cache[tid+1];
}

template <unsigned int blockSize>
__global__ void reduce5(float *d_in,float *d_out){
    __shared__ float sdata[THREAD_PER_BLOCK];

    //each thread loads one element from global memory to shared mem
    unsigned int i=blockIdx.x*(blockDim.x*2)+threadIdx.x;
    unsigned int tid=threadIdx.x;
    sdata[tid]=d_in[i] + d_in[i+blockDim.x];
    __syncthreads();

    // do reduction in shared mem
    if(blockSize>=512){
        if(tid<256){
            sdata[tid]+=sdata[tid+256];
        }
        __syncthreads();
    }
    if(blockSize>=256){
        if(tid<128){
            sdata[tid]+=sdata[tid+128];
        }
        __syncthreads();
    }
    if(blockSize>=128){
        if(tid<64){
            sdata[tid]+=sdata[tid+64];
        }
        __syncthreads();
    }

    // write result for this block to global mem
    if(tid<32)warpReduce<blockSize>(sdata,tid);
    if(tid==0)d_out[blockIdx.x]=sdata[tid];
}
```

实验结果

-	耗时(ns)	带宽(GB/s)	加速比
reduce0	804,187	159.2	-
reduce1	506,365	252.8	1.59x
reduce2	397,886	321.7	1.27x
reduce3	213,182	600.4	1.86x
reduce4	169,215	756.4	1.26x
reduce5	166,752	767.6	1.01x

可以看到，还是有所收益，但是并没有那么地显著。这主要是因为GPU硬件架构的不断发展，以及NV在编译器上面也做了较多的工作。

优化技巧6：合理设置block数量

现有问题

当走到这一步的时候，能调的东西已经基本上调完了。我们再把眼光放在block和thread的设置上。之前默认了Num_per_block=Thread_per_block。也就是说，一个block开启256个线程时，这个block负责256个元素的reduce操作。那可不可以让一个block多管点数。这样的话，开启的block数量少一些。以此**对block设置进行调整**，获得最优block取值，这样或许能够带来一些性能收益?

解决方式

这样需要再思考一下block的取值。对于GPU而言，block的取值到底是多更好，还是少更好。如此对CUDA编程熟悉的同学，肯定会毫不犹豫地说：“那肯定是多更好啦。Block数量多，block可以进行快速地切换，去掩盖访存的延时。”这个问题按下不表，我们看看Mharris是怎么说的。

如果一个线程被分配更多的work时，可能会更好地覆盖延时。这一点比较好理解。如果线程有更多的work时，对于编译器而言，就可能有更多的机会对相关指令进行重排，从而去覆盖访存时的巨大延时。虽然这句话并没有很好地说明在某种程度上而言，block少一些会更好。但是，有一点不可否认,**block需要进行合理地设置。**唠唠叨叨说了很多，现在把代码贴一下：

```
template <unsigned int blockSize>
__global__ void reduce6(float *d_in,float *d_out){
    __shared__ float sdata[THREAD_PER_BLOCK];

    //each thread loads one element from global memory to shared mem
    unsigned int i=blockIdx.x*(blockDim.x*2)+threadIdx.x;
    unsigned int tid=threadIdx.x;
    unsigned int gridSize = blockSize * 2 * gridDim.x;
    sdata[tid] = 0;

    while(i<n){
        sdata[tid] +=d_in[i]+d_in[i+blockSize];
        i+=gridSize;
    }
    __syncthreads();

    // do reduction in shared mem
    if(blockSize>=512){
        if(tid<256){
            sdata[tid]+=sdata[tid+256];
        }
        __syncthreads();
    }
    if(blockSize>=256){
        if(tid<128){
            sdata[tid]+=sdata[tid+128];
        }
        __syncthreads();
    }
    if(blockSize>=128){
        if(tid<64){
            sdata[tid]+=sdata[tid+64];
        }
        __syncthreads();
    }

    // write result for this block to global mem
    if(tid<32)warpReduce<blockSize>(sdata,tid);
    if(tid==0)d_out[blockIdx.x]=sdata[tid];
}
```

实验结果

-	耗时(ns)	带宽(GB/s)	加速比
reduce0	804,187	159.2	-
reduce1	506,365	252.8	1.59x
reduce2	397,886	321.7	1.27x
reduce3	213,182	600.4	1.86x
reduce4	169,215	756.4	1.26x
reduce5	166,752	767.6	1.01x
reduce6	166,656	768.1	1.00x

对于block的取值，我进行了微调，大概取2048会比512，1024，4096的效果要好。理论上来说，这个值取SM数量的倍数会比较合理。但是V100的SM是80，取一个完美的倍数还是比较困难。目前达到了768GB/s。

优化技巧7：使用shuffle指令

现有问题

其实，对于Mharris的讲义。reduce优化就到此结束了。但是NV后来出了Shuffle指令，对于reduce优化有着非常好的效果。目前绝大多数访存类算子，像是softmax，batch_norm，reduce等，都是用Shuffle实现。所以，在这里谈一下这么把shuffle指令用在reduce优化上。

Shuffle指令是一组针对warp的指令。Shuffle指令最重要的特性就是**warp内的寄存器可以相互访问**。在没有shuffle指令的时候，各个线程在进行通信时只能通过shared memory来访问彼此的寄存器。而采用了shuffle指令之后，warp内的线程可以直接对其他线程的寄存器进行访存。通过这种方式可以减少访存的延时。除此之外，带来的最大好处就是可编程性提高了，在某些场景下，就不用shared memory了。毕竟，开发者要自己去控制 shared memory还是挺麻烦的一个事。

伪代码如下：

```
template <unsigned int blockSize>
__device__ __forceinline__ float warpReduceSum(float sum){
    if(blockSize >= 32)sum += __shfl_down_sync(0xffffffff,sum,16);
    if(blockSize >= 16)sum += __shfl_down_sync(0xffffffff,sum,8);
    if(blockSize >= 8)sum += __shfl_down_sync(0xffffffff,sum,4);
    if(blockSize >= 4)sum += __shfl_down_sync(0xffffffff,sum,2);
    if(blockSize >= 2)sum += __shfl_down_sync(0xffffffff,sum,1);
    return sum;
}

template <unsigned int blockSize>
__global__ void reduce7(float *d_in,float *d_out, unsigned int n){
    float sum = 0;

    //each thread loads one element from global memory to shared mem
    unsigned int i=blockIdx.x*(blockDim.x*2)+threadIdx.x;
    unsigned int tid=threadIdx.x;
    unsigned int gridSize = blockSize * 2 * gridDim.x;

    while(i<n){
        sum +=d_in[i]+d_in[i+blockSize];
        i+=gridSize;
    }

    // shared mem for partial sums(one per warp in the block
    static __shared__ float warpLevelSums[WARP_SIZE];
    const int laneId = threadIdx.x % WARP_SIZE;
    const int warpId = threadIdx.x / WARP_SIZE;

    sum = warpReduceSum<blockSize>(sum);

    if(laneId == 0)warpLevelSums[warpId]=sum;
    __syncthreads();

    sum = (threadIdx.x < blockDim.x / WARP_SIZE)? warpLevelSums[laneId]:0;
    // Final reduce using first warp
    if(warpId == 0)sum = warpReduceSum<blockSize/WARP_SIZE>(sum);
    // write result for this block to global mem
    if(tid==0)d_out[blockIdx.x]=sum;
}
```

实验结果

-	耗时(ns)	带宽(GB/s)	加速比
reduce0	804,187	159.2	-
reduce1	506,365	252.8	1.59x
reduce2	397,886	321.7	1.27x
reduce3	213,182	600.4	1.86x
reduce4	169,215	756.4	1.26x
reduce5	166,752	767.6	1.01x
reduce6	166,656	768.1	1.00x
reduce7	166,175	770.3	1.00x

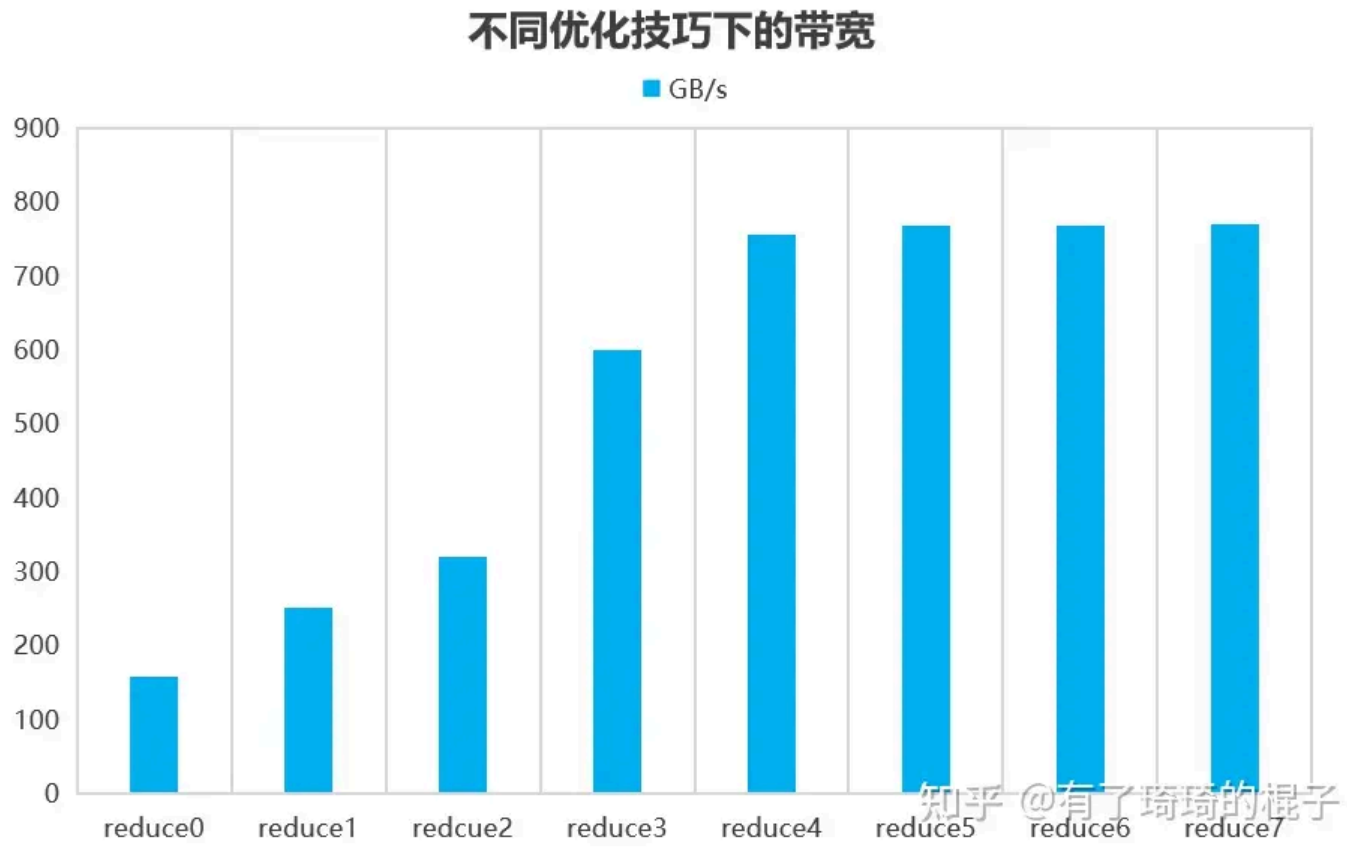
总结与思考

通过上次一系列的优化技巧，我们对reduce进行了不断地优化。最后效果是770.3GB/s。带宽利用达到85%。PS：具体的reduce性能数据还需要大量测试，但由于我比较懒。所以测试工作就到此为此，大家有兴趣可以自己再跑跑。代码在这里，还有点乱，没怎么整理，大家有疑问可以直接评论或者私信我。

reduce代码

github.com/Liu-xiandong/How_to_optimize_in_GPU

然后不同的优化技巧所带来的性能表现如下：



可以看到，其实在reduce4的时候就已经很难再提升了。而且这个数据跟NV在博客上的数据还是有比较大的出入，主要是对最后一维展开后，在V100上的带宽利用率就很难再提升了，而G80还有显著的提升。我觉得主要原因是硬件已经更新了好几代。

	Time (2^{22} ints)	Bandwidth	Step Speedup	Cumulative Speedup
Kernel 1: interleaved addressing with divergent branching	8.054 ms	2.083 GB/s		
Kernel 2: interleaved addressing with bank conflicts	3.456 ms	4.854 GB/s	2.33x	2.33x
Kernel 3: sequential addressing	1.722 ms	9.741 GB/s	2.01x	4.68x
Kernel 4: first add during global load	0.965 ms	17.377 GB/s	1.78x	8.34x
Kernel 5: unroll last warp	0.536 ms	31.289 GB/s	1.8x	15.01x
Kernel 6: completely unrolled	0.381 ms	43.996 GB/s	1.41x	21.16x
Kernel 7: multiple elements per thread	0.268 ms	62.671 GB/s	1.42x	30.04x

Kernel 7 on 32M elements: 73 GB/s!

总而言之，我们通过这一系列的优化已经可以把reduce优化到一个非常好的程度。我之前测过一次是160us，十分接近800GB/s。但不知道为啥就不能复现了。对于访存型的算子，在V100上能做到接近800GB/s的带宽就已经接近极限了。而目前能够做的优化，也都列了出来。如果还有更有意思的方式，也麻烦联系我一下。

最后，感谢大家看到这里。后续的话，打算开一个系列。最近有这么两个想说的主题：

GEMM（矩阵乘）的优化，这个主题的代码已经写完了。但是一直没有时间整理。而且关于汇编器来消除寄存器的bank冲突那里实在是太麻烦了，想做点实验需要耗费巨大的精力。
Spmv（稀疏矩阵乘）的优化，这个主题的代码写了第一版代码。后续还需要进行优化和整理。主要是针对CSR格式的稀疏矩阵调优。
GPU硬件架构。目前关于NVIDA的GPU架构，能够找到的东西并不是很多。打算从GPGPU-sim入手介绍。因为感觉GPGPU-sim应该是最贴近硬件细节的。目前体系结构的研究者主要也是用GPGPU-sim为基础进行模拟写一些demo。