【CUDA 基础】5.3 减少全局内存访问

2018-06-04 | CUDA , Freshman | 0 |

Abstract: 本文介绍使用共享内存进行归约，并比较全局内存归约与共享内存归约之间的性能差距

Keywords: 共享内存，归约

减少全局内存访问

逻辑是非常重要的，一旦你学会了逻辑，很多假的东西你可以轻松的识别出来，这会使你更加强大而不会被任何人或者组织洗脑。

废话少说，开始今天的博客。

使用共享内存的主要原因就是减少对全局内存的访问，来减少不必要的延迟，第三章我们学过了归约，可

以参考：

- https://face2ai.com/CUDA-F-3-4-避免分支分化/
- https://face2ai.com/CUDA-F-3-5-展开循环/

这两篇博客包含我们前面使用全局内存进行归约的各种技术，我们几天也要用其中一部分代码作为比较，来体现我们共享内存的优势。
我们要集中解决下面两个问题：

1. 如何重新安排数据访问模式以避免线程束分化
2. 如何展开循环以保证有足够的操作使指令和内存带宽饱和

本文我们通过对比研究前面的部分代码，来分析为何要使用共享内存，以及如何使用共享内存。

使用共享内存的并行归约

我们首先来回忆全局内存下的，完全展开的归约计算：

```
__global__ void reduceGmem(int * g_idata,int * g_odata,unsigned int n)
{
        //set thread ID
        unsigned int tid = threadIdx.x;
        unsigned int idx = blockDim.x*blockIdx.x+threadIdx.x;
        //boundary check
        if (tid >= n) return;
        //convert global data pointer to the
        int *idata = g_idata + blockIdx.x*blockDim.x;

        //in-place reduction in global memory
        if(blockDim.x>=1024 && tid <512)
                idata[tid]+=idata[tid+512];
        __syncthreads();
        if(blockDim.x>=512 && tid <256)
                idata[tid]+=idata[tid+256];
        __syncthreads();
        if(blockDim.x>=256 && tid <128)
                idata[tid]+=idata[tid+128];
        __syncthreads();
        if(blockDim.x>=128 && tid <64)
                idata[tid]+=idata[tid+64];
        __syncthreads();
```

```
        //write result for this block to global mem
        if(tid<32)
        {
                volatile int *vmem = idata;
                vmem[tid]+=vmem[tid+32];
                vmem[tid]+=vmem[tid+16];
                vmem[tid]+=vmem[tid+8];
                vmem[tid]+=vmem[tid+4];
                vmem[tid]+=vmem[tid+2];
                vmem[tid]+=vmem[tid+1];

        }

        if (tid == 0)
                g_odata[blockIdx.x] = idata[0];

}
```

下面这步是计算当前线程的索引位置：

```
unsigned int idx = blockDim.x*blockIdx.x+threadIdx.x;
```

当前线程块对应的数据块首地址

```
int *idata = g_idata + blockIdx.x*blockDim.x;
```

然后是展开循环的部分，tid是当前线程块中线程的标号，主要区别于全局编号idx：

```
 if(blockDim.x>=1024 && tid <512)
     idata[tid]+=idata[tid+512];
 __syncthreads();
 if(blockDim.x>=512 && tid <256)
     idata[tid]+=idata[tid+256];
 __syncthreads();
 if(blockDim.x>=256 && tid <128)
     idata[tid]+=idata[tid+128];
 __syncthreads();
 if(blockDim.x>=128 && tid <64)
     idata[tid]+=idata[tid+64];
 __syncthreads();
```

这一步把是当前线程块中的所有数据归约到前64个元素中，接着使用如下代码，将最后64个元素归约成一个

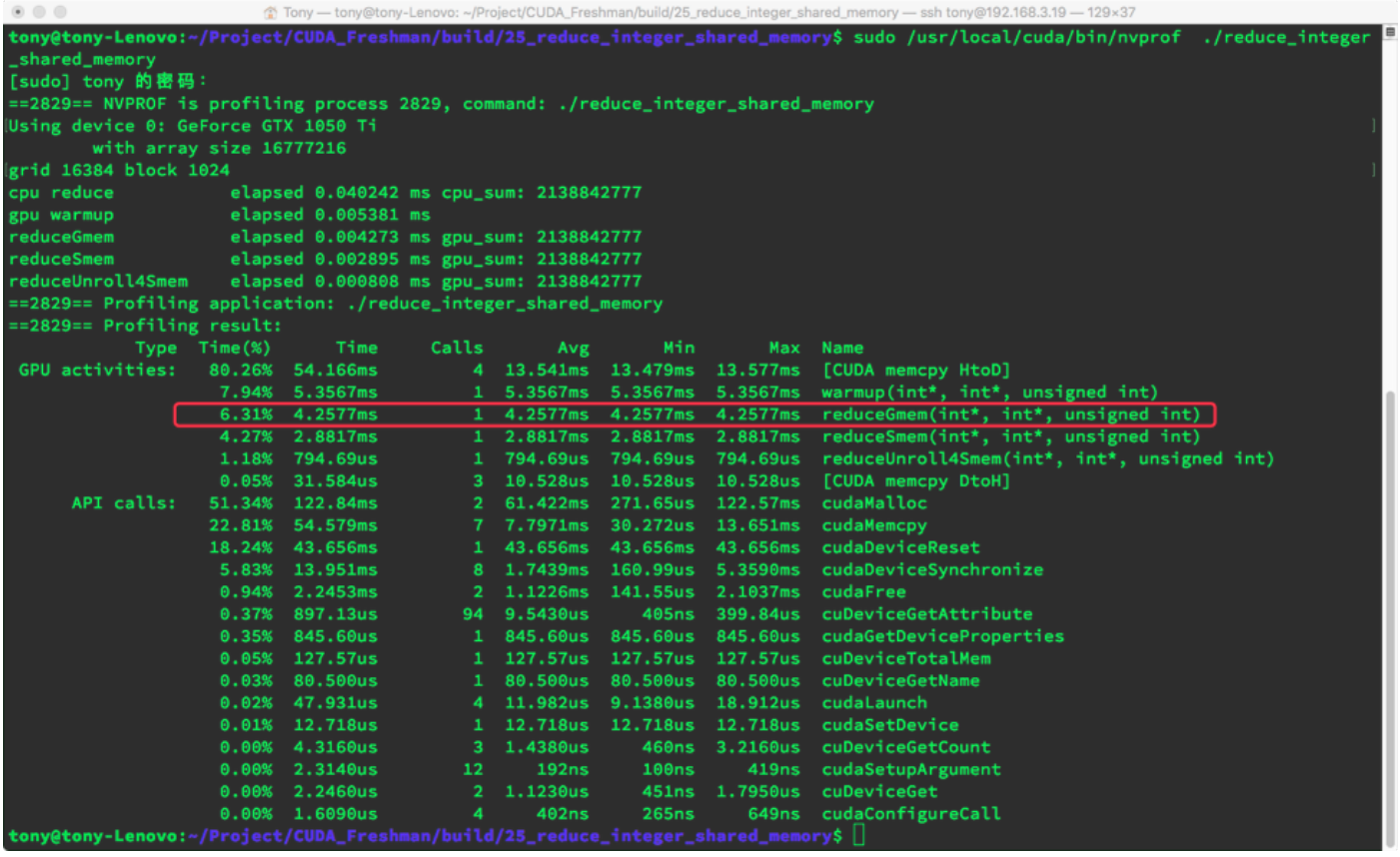
```
if(tid<32)
{
    volatile int *vmem = idata;
    vmem[tid]+=vmem[tid+32];
    vmem[tid]+=vmem[tid+16];
    vmem[tid]+=vmem[tid+8];
    vmem[tid]+=vmem[tid+4];
    vmem[tid]+=vmem[tid+2];
    vmem[tid]+=vmem[tid+1];
}
```

注意这里声明了一个volatile变量，如果我们不这么做，编译器不能保证这些数据读写操作按照代码中的顺序执行（参考5.1中关于编译器数据传输部分的说明），所以必须要这么做。

然后我们执行以下这段代码，虽然前面执行过了，我们还是执行以下，观察下结果：

完整的可执行代码依旧在GitHub上可以找到

github:https://github.com/Tony-Tan/CUDA_Freshman点个星星也不会很累，拜托啦😆



这里我们假装看不到别的，只看我们的核函数，可见器质性时间是4.25ms左右

然后我们对上面的代码进行改写，改写成共享内存的版本，来看代码：

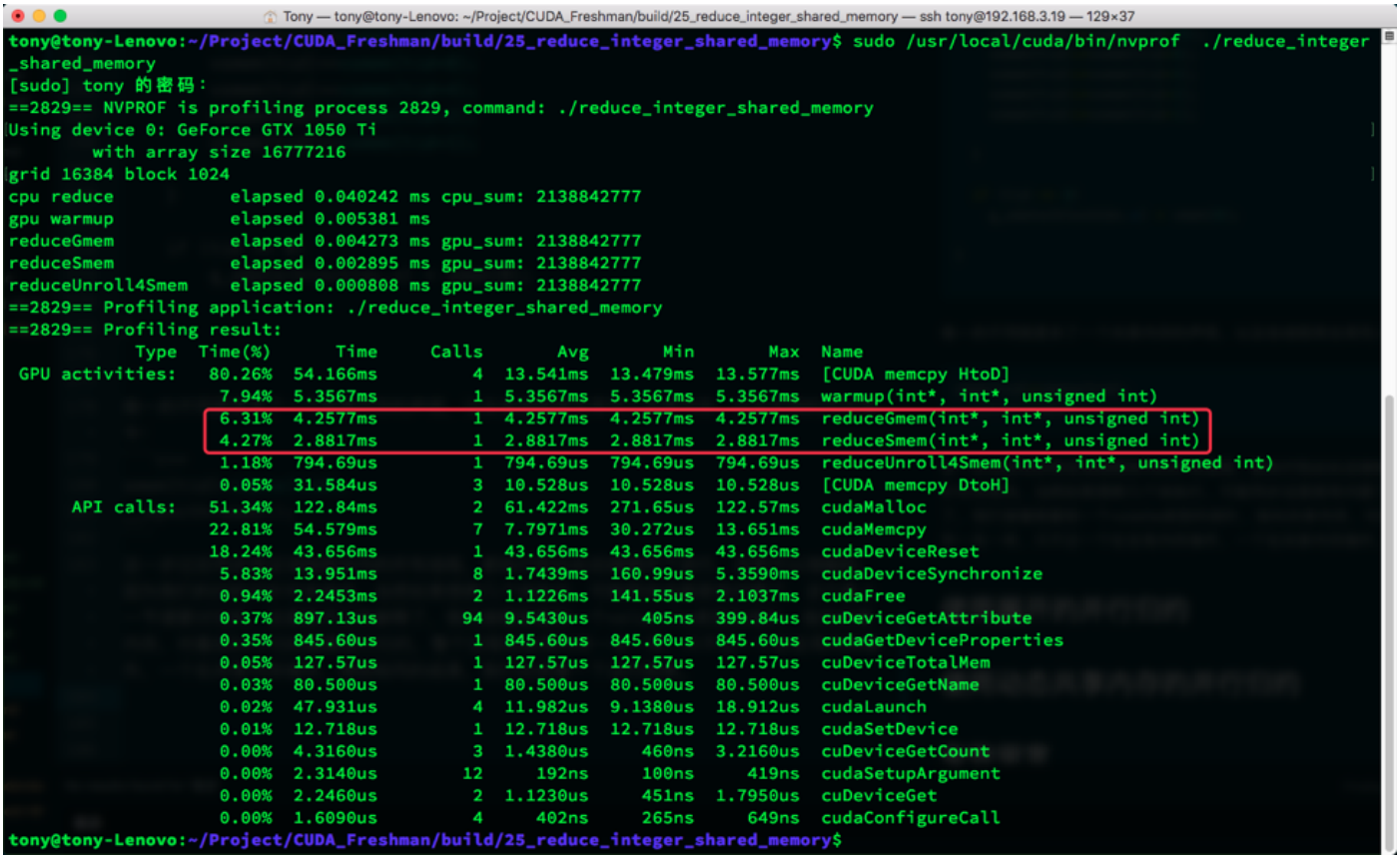
```
__global__ void reduceSmem(int * g_idata,int * g_odata,unsigned int n)
{
        //set thread ID
    __shared__ int smem[DIM];
        unsigned int tid = threadIdx.x;
        //unsigned int idx = blockDim.x*blockIdx.x+threadIdx.x;
        //boundary check
        if (tid >= n) return;
        //convert global data pointer to the
        int *idata = g_idata + blockIdx.x*blockDim.x;

    smem[tid]=idata[tid];
        __syncthreads();
        //in-place reduction in global memory
        if(blockDim.x>=1024 && tid <512)
                smem[tid]+=smem[tid+512];
        __syncthreads();
        if(blockDim.x>=512 && tid <256)
                smem[tid]+=smem[tid+256];
        __syncthreads();
        if(blockDim.x>=256 && tid <128)
                smem[tid]+=smem[tid+128];
        __syncthreads();
        if(blockDim.x>=128 && tid <64)
                smem[tid]+=smem[tid+64];
        __syncthreads();
        //write result for this block to global mem
        if(tid<32)
        {
                volatile int *vsmem = smem;
                vsmem[tid]+=vsmem[tid+32];
                vsmem[tid]+=vsmem[tid+16];
                vsmem[tid]+=vsmem[tid+8];
                vsmem[tid]+=vsmem[tid+4];
                vsmem[tid]+=vsmem[tid+2];
                vsmem[tid]+=vsmem[tid+1];

        }

        if (tid == 0)
```

```
41                  g_odata[blockIdx.x] = smem[0];
42
43  }
```

唯一的不同就是多了一个共享内存的声明，以及各线程将全局写入共享内存，以及后面的同步指令：

```
1   smem[tid]=idata[tid];
2   __syncthreads();
```

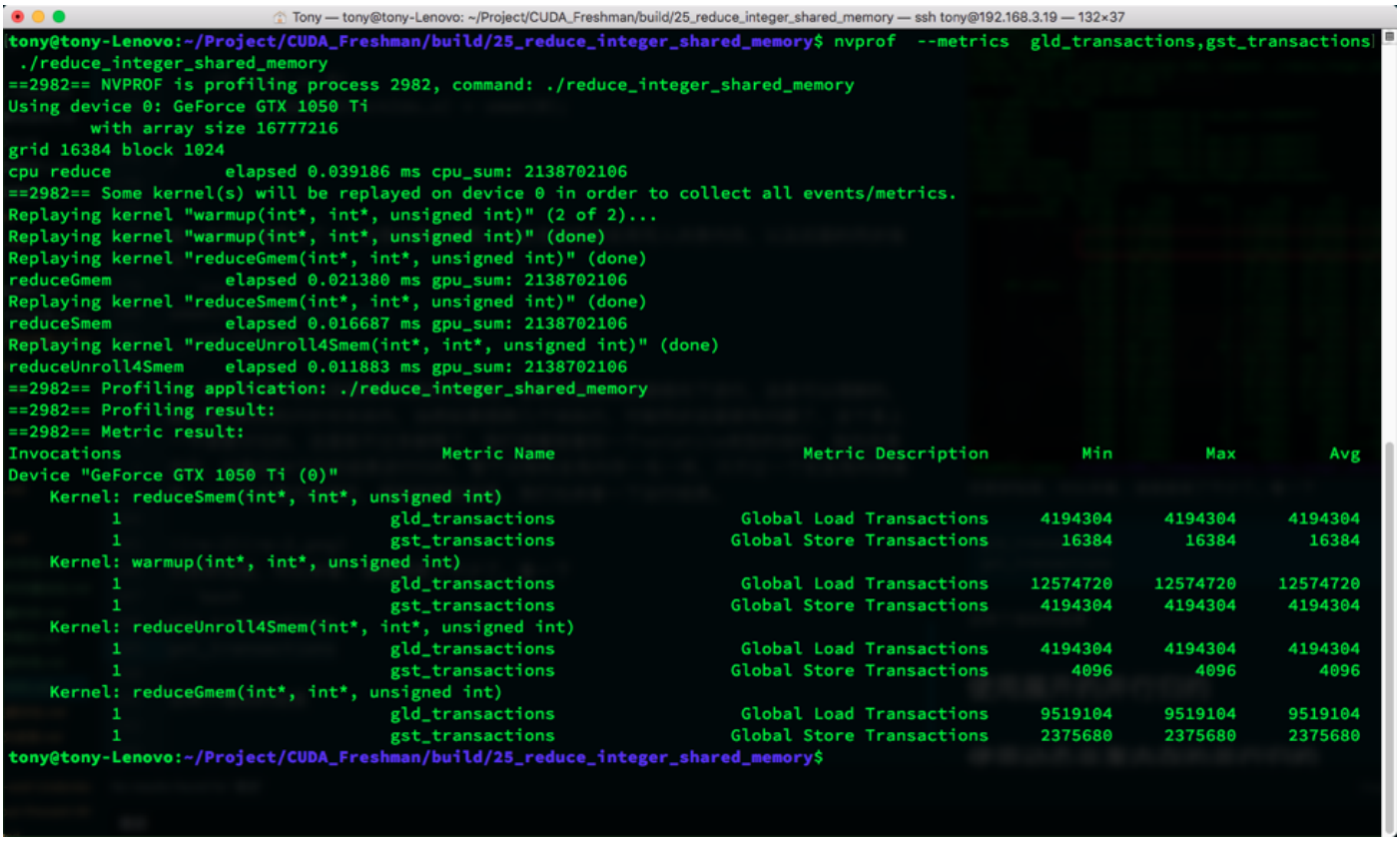
这一步过后同步保证该线程块内的所有线程，都执行到此处后继续向下进行，这是可以理解的，因为我们的归约只针对本块内，当然如果想跨几个块执行，可能同步这里就有问题了，这个是上一节课要讨论的，这里就不过多解释了，我们接着就看到一个volatile类型的指针，指向共享内存，对最后64个归约结果进行归约，整个过程和全局内存一毛一样，只不过一个在全局内存操作，一个在共享内存操作，得到相同的结果，我们也来看一下运行结果。



还是那张图，对比来看，速度提高了不少了。看一下

```
gld_transactions
gst_transactions
```

这两个指标的结果



可以看出使用共享内存的，比使用全局内存的高到不知道哪里去了

使用展开的并行归约

可能看到上面的截图你已经知道我接下来要并行4块了，对于前面说的，使用共享内存不能并行四块，是因为没办法同步读四个块，这里我们还是用老方法进行并行四个块，就是在写入共享内存之前进行归约，4个块变成一个，然后把这一个存入共享内存，进行常规的共享内存归约:

```
__global__ void reduceUnroll4Smem(int * g_idata,int * g_odata,unsigned int n)
{
        //set thread ID
    __shared__ int smem[DIM];
        unsigned int tid = threadIdx.x;
        unsigned int idx = blockDim.x*blockIdx.x*4+threadIdx.x;
        //boundary check
        if (tid >= n) return;
```

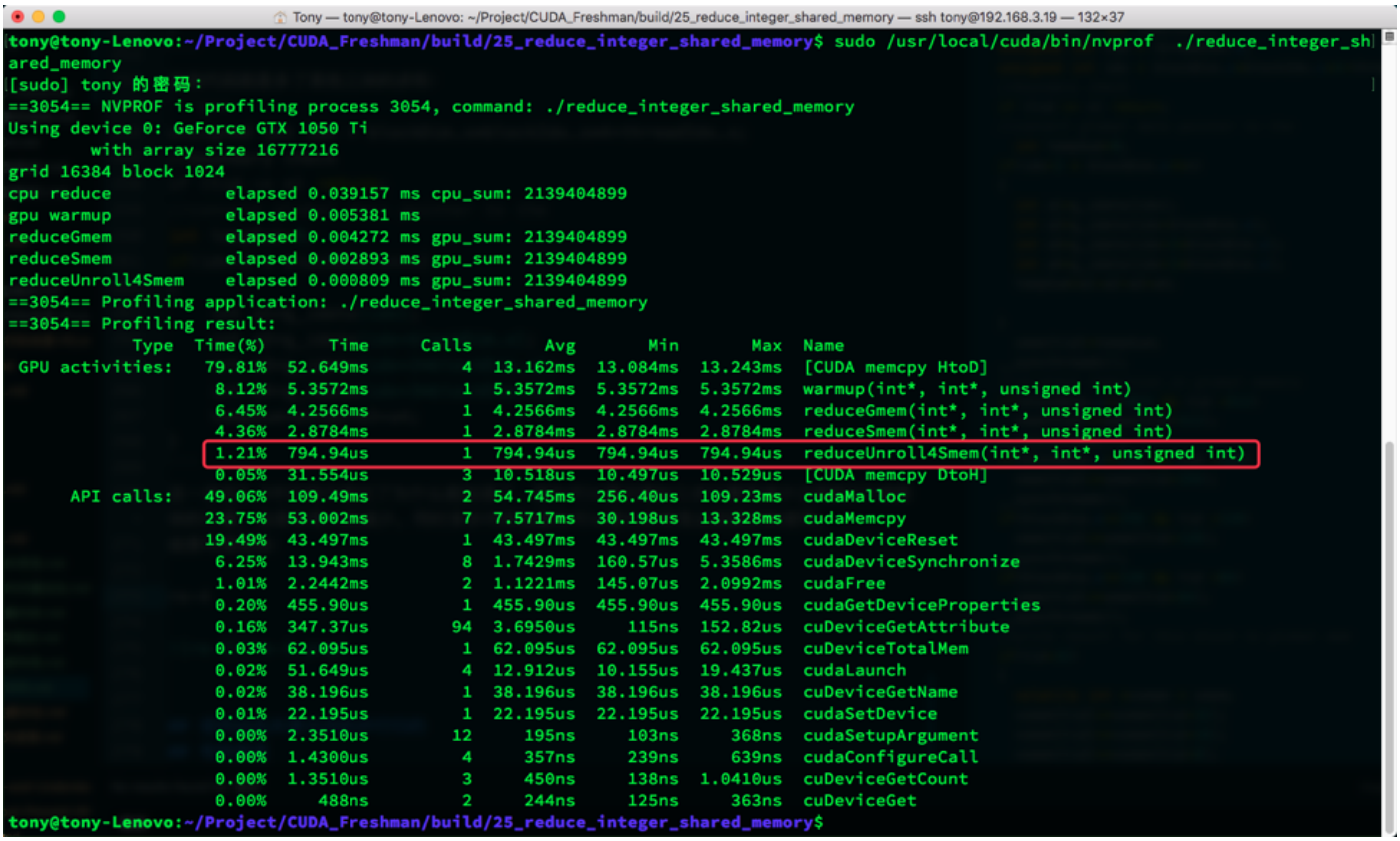
```
        //convert global data pointer to the
    int tempSum=0;
        if(idx+3 * blockDim.x<=n)
        {
                int a1=g_idata[idx];
                int a2=g_idata[idx+blockDim.x];
                int a3=g_idata[idx+2*blockDim.x];
                int a4=g_idata[idx+3*blockDim.x];
                tempSum=a1+a2+a3+a4;

        }
    smem[tid]=tempSum;
        __syncthreads();
        //in-place reduction in global memory
        if(blockDim.x>=1024 && tid <512)
                smem[tid]+=smem[tid+512];
        __syncthreads();
        if(blockDim.x>=512 && tid <256)
                smem[tid]+=smem[tid+256];
        __syncthreads();
        if(blockDim.x>=256 && tid <128)
                smem[tid]+=smem[tid+128];
        __syncthreads();
        if(blockDim.x>=128 && tid <64)
                smem[tid]+=smem[tid+64];
        __syncthreads();
        //write result for this block to global mem
        if(tid<32)
        {
                volatile int *vsmem = smem;
                vsmem[tid]+=vsmem[tid+32];
                vsmem[tid]+=vsmem[tid+16];
                vsmem[tid]+=vsmem[tid+8];
                vsmem[tid]+=vsmem[tid+4];
                vsmem[tid]+=vsmem[tid+2];
                vsmem[tid]+=vsmem[tid+1];

        }

        if (tid == 0)
                g_odata[blockIdx.x] = smem[0];

}
```

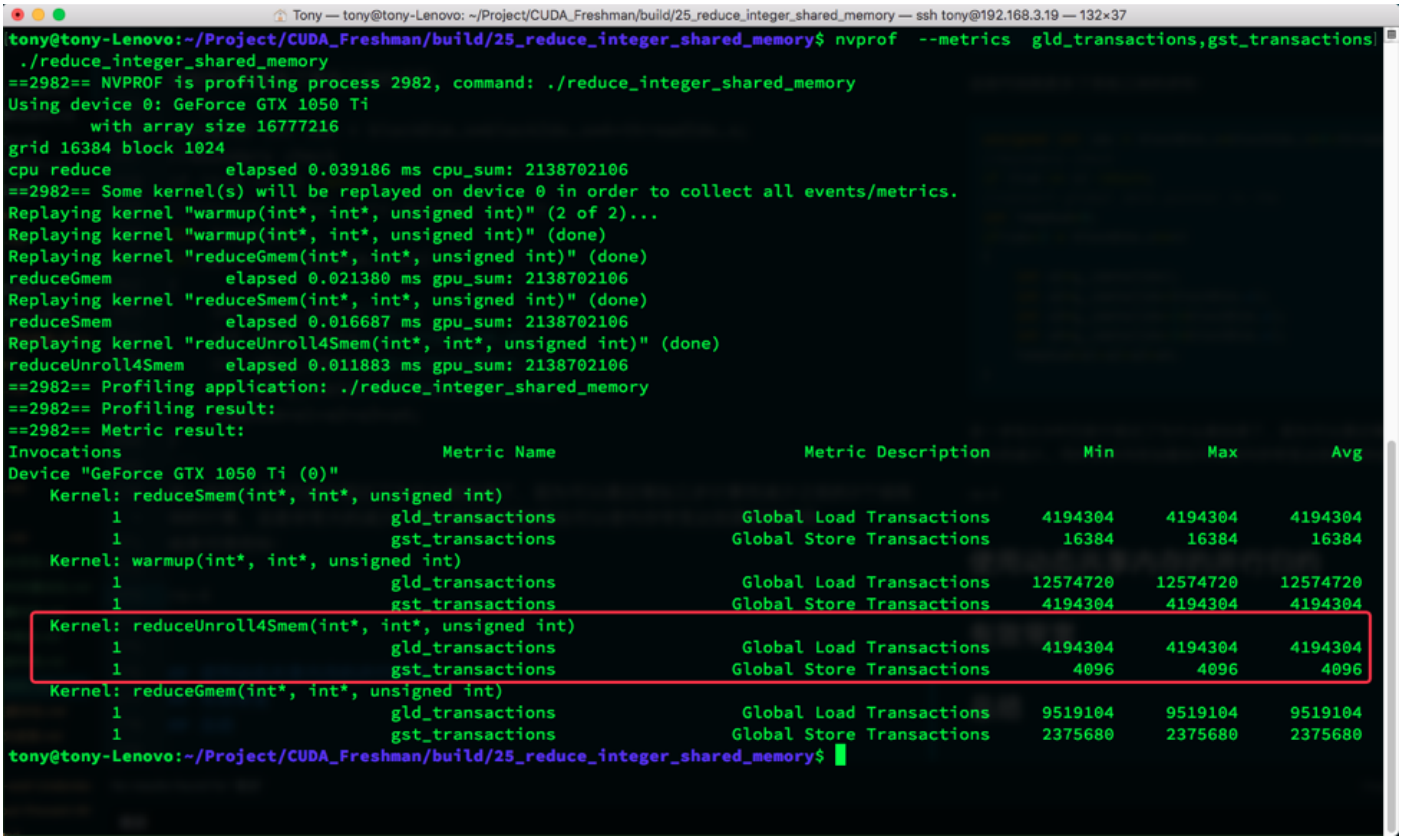
这段代码就是多了其他三块的求和：

```
unsigned int idx = blockDim.x*blockIdx.x*4+threadIdx.x;
//boundary check
if (tid >= n) return;
//convert global data pointer to the
int tempSum=0;
if(idx+3 * blockDim.x<=n)
{
    int a1=g_idata[idx];
    int a2=g_idata[idx+blockDim.x];
    int a3=g_idata[idx+2*blockDim.x];
    int a4=g_idata[idx+3*blockDim.x];
    tempSum=a1+a2+a3+a4;
}
```

这一步在3.5中已经介绍过了为什么能加速了，因为可以通过增加三步计算而减少之前的3个线程块的计算，这是非常大的减少。同时多步内存加载也可以使内存带宽达到更好的使用。

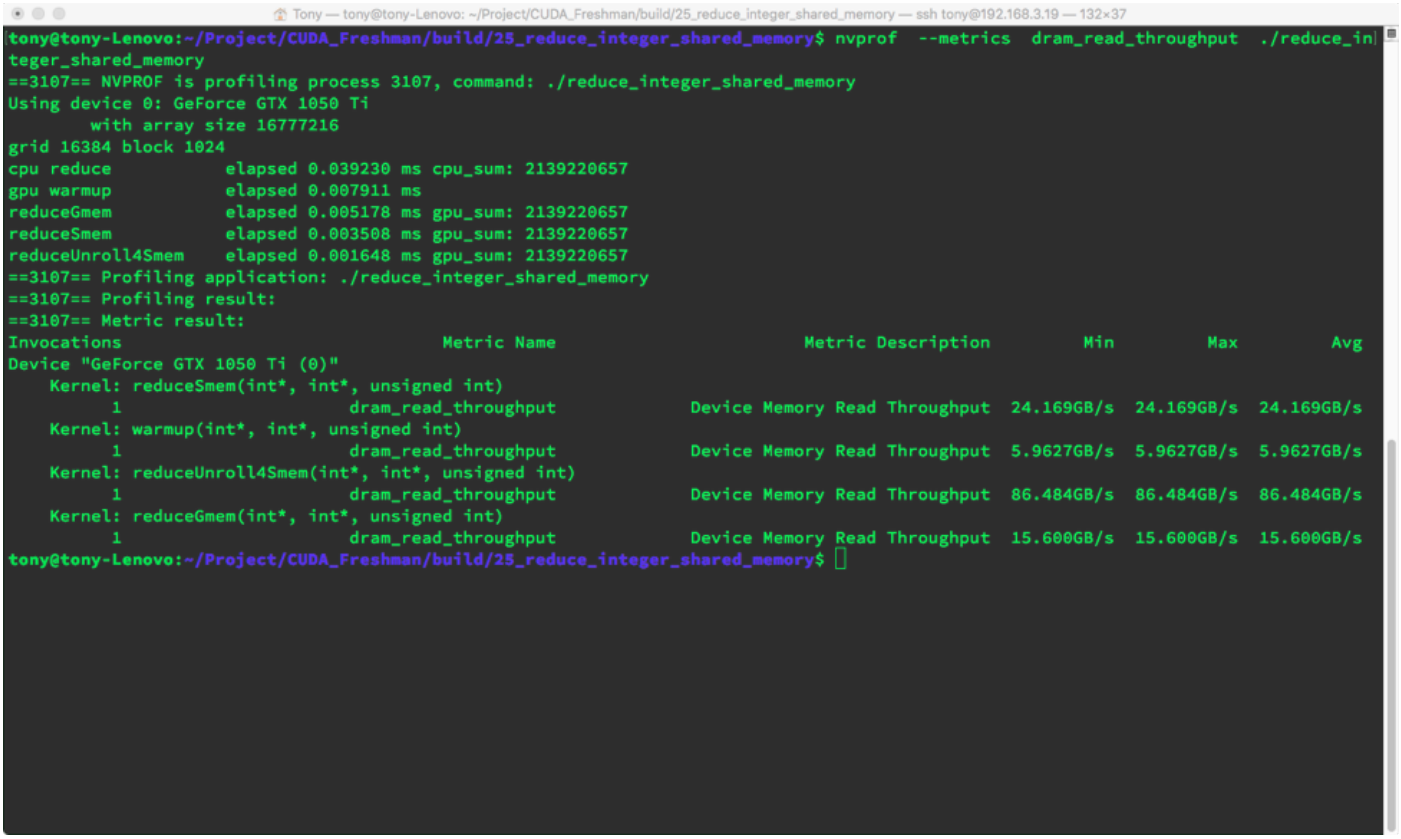
结果可想而知：





吞吐量指标：

```
nvprof --metrics dram_read_throughput ./reduce_integer_shared_memory
```



无论是指标还是运行速度，都有非常显著的提升。

我们这里总结下展开的优势：

- I/O得到了更多的并行，就是我上面说的更好的利用带宽，增加了吞吐量
- 全局内存存储事务减少到 $\frac{1}{4}$ ，这个主要针对最后一步，将结果存入全局内存
- 整体性能巨幅提升

使用动态共享内存的并行归约

然后我们看一下动态版本，其实动态版本没啥可看的，只是写法上有点不同，把宏改成核函数配置参数，注意其单位是字节就好，也就是不要忘了sizeof()就行了。

这里不啰嗦了。

有效带宽

对比一下数据，回顾一下我们的有效带宽，其计算公式(4.4中有详细介绍)：

$$\text{有效带宽} = \frac{(\text{读字节数} + \text{写字节数}) \times 10^{-9}}{\text{运行时间}} \tag{1}$$

可以研究三个核函数的有效带宽，这里就不再一个个计算了，因为这个在4.4中已经教大家做了，我们可以自己算一下，并得出结论

总结

本文主要高速大家如何使用共享内存加速归约，以及结合了共享内存的展开能更高的提高效率，注意线程块内的同步，这个是重要的。

相关文章

- 【CUDA 基础】3.5 展开循环
- 【CUDA 基础】5.0 共享内存和常量内存
- 【Julia】整型和浮点型数字
- 【Julia】变量
- 【Julia】开始使用Julia

归约 # 共享内存

【CUDA 基础】5.2 共享内存的数据布局

【CUDA 基础】5.4 合并的全局内存访问