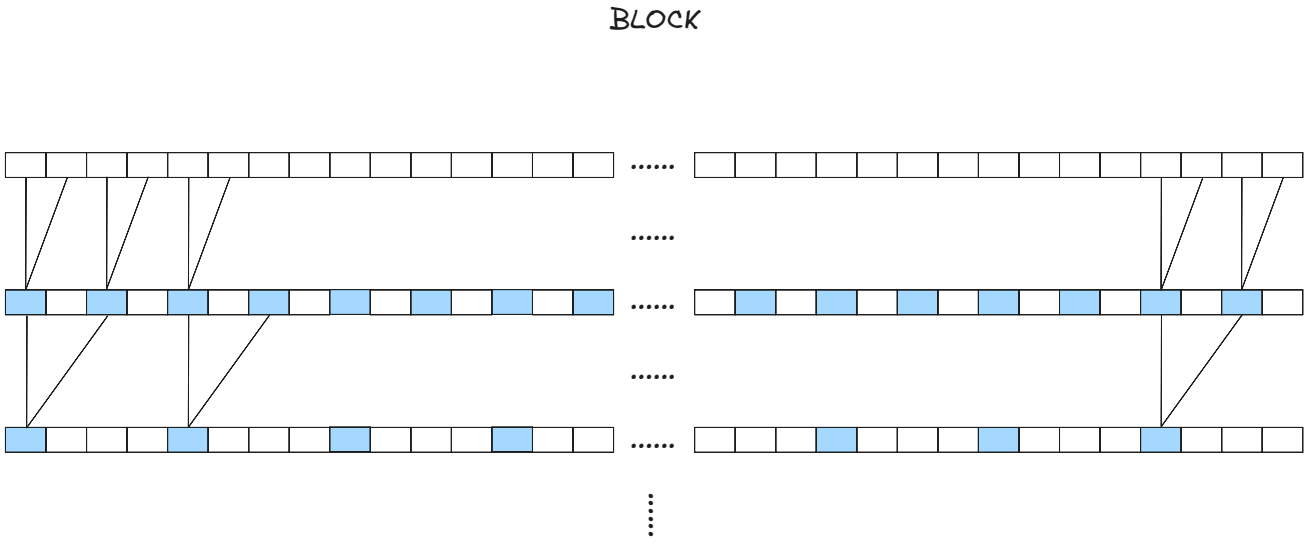
Reduce

Intro. reduce算子

reduce_v0_global_mem

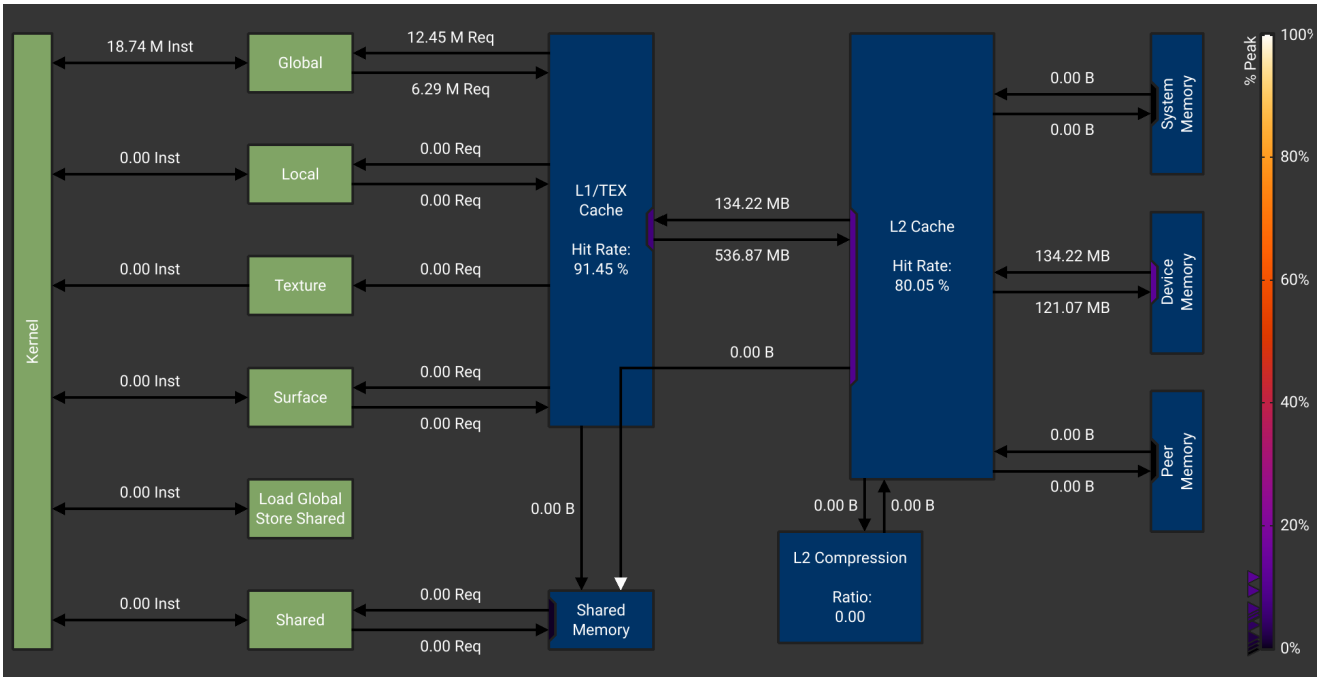
在v0版本中，kernel直接访问DRAM，计算并写回。具体流程是我们只操作偶数线程，让其读取计算相邻的两个数据并写回到偶数线程对应的点。经过循环最后每个block剩下一个线程0和block中的结果。




```
__global__ void reduce0(float *d_input, float *d_output)
{
    unsigned int tid = threadIdx.x;
    unsigned int global_tid = blockIdx.x * blockDim.x + tid;

    for (unsigned int i = 1; i < blockDim.x; i *= 2)
    {
        if (tid % (i * 2) == 0)
        {
            d_input[global_tid] += d_input[global_tid + i];
            __syncthreads();
        }
    }

    if (tid == 0)
    {
        d_output[blockIdx.x] = d_input[global_tid];
```

```
    }
}
```

由于我们频繁的访问这个矩阵，所以更明智的做法是使用shared_mem。另外在kernel内做除法也是不明智的，因为除法的实现可能是用过迭代实现的，会占用很多运算资源，我们会在后面的某一步把（乘）除法换成位运算。



可以看到此时L1到L2的写入比反方向读取要多了4倍，原因可能为：一个warp有32个线程操作32个数据的reduce。由于每一个req都会直接操作8个float，所以直到仅剩4个线程工作时每次写入也是写入32个float。再次之后会有写入16个float和8个float两次计算，再加上最后写回到DRAM共计$1+1+1+0.5+0.25+0.25=4$倍读取。

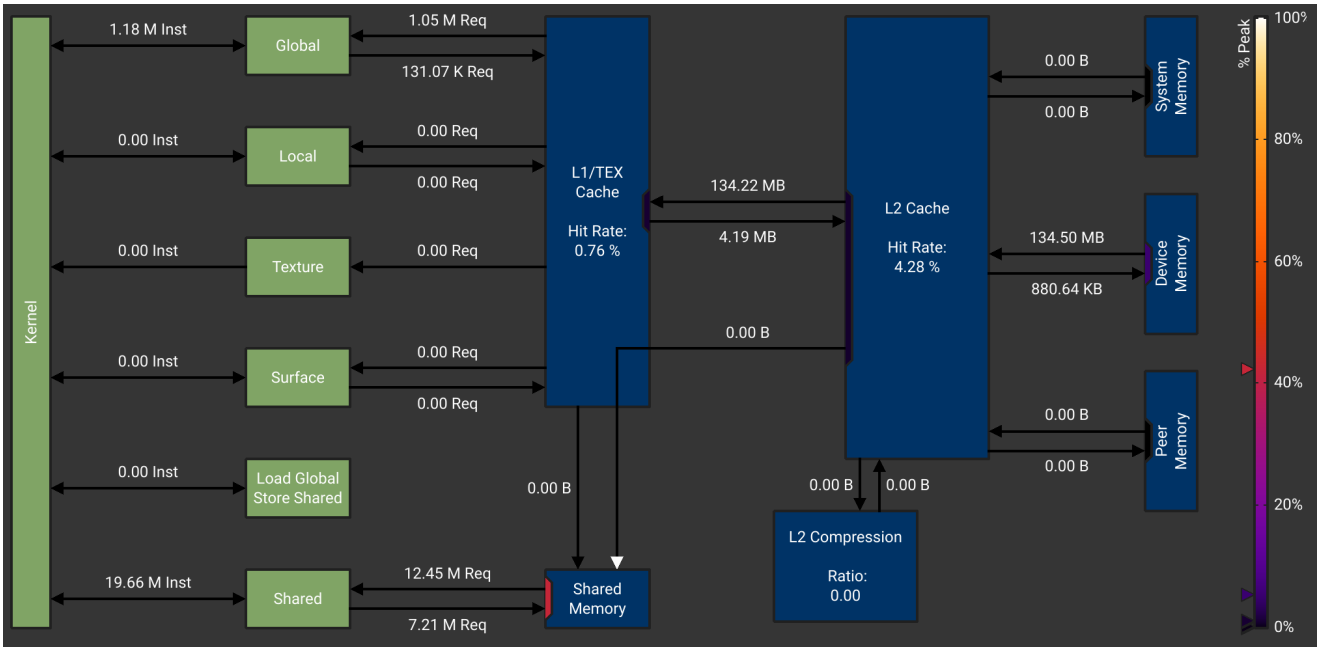
此时的吞吐量是：$30,12.61\%$。

reduce_v1_shared_mem

这一次我们用shared_mem。

```
__global__ void reduce1(float *d_input, float *d_output)
{
    __shared__ float shared_mem[THREAD_PER_BLOCK];
    unsigned int tid = threadIdx.x;
    unsigned int global_tid = blockIdx.x * blockDim.x + tid;

    shared_mem[tid] = d_input[global_tid];
    __syncthreads();

    for (unsigned int i = 1; i < blockDim.x; i *= 2)
    {
```

```
        if (tid % (i * 2) == 0)
        {
            shared_mem[tid] += shared_mem[tid + i];
            __syncthreads();
        }
    }

    if (tid == 0)
    {
        d_output[blockIdx.x] = shared_mem[tid];
    }
}
```



可以注意到L1到L2的写入被减少到了读取的$1/32$这和程序中我们使用了shared mem且只有最后一次同步有写回到DRAM的操作相对应。
此时的吞吐量为：$14,65.71\%$。

为什么反而吞吐量降低了?

L1 cache hit： 在reduce0中我们有极高的L1和L2 cache hit rate，这意味着在第一次加载完数据后L1可以完全装下几乎所有的数据并复用另外根据SM结构可知，shared mem的物理实体实际就是L1 cache。实际上reduce0并没有经常访问DRAM，在加载数据时从shared mem或L1的吞吐量所差无几，而且第一次的加载是使用shared mem时也会执行的操作。
__syncthreads()： 线程间的同步开销会拖慢程序效率，降低吞吐量。
Bank conflict： bank conflict会导致并行的读取变成串行，在reduce4中被解决。

reduce_v2_no_divergence_branch

在这一步中我们要解决线程调度的问题。首先分析在reduce1中的线程分支问题。

Warp State Statistics

Analysis of the states in which all warps spent cycles during the kernel execution. The warp states describe a warp's readiness or inability to issue its next instruction. The warp cycles per instruction define the latency between two consecutive instructions. The higher the value, the more warp parallelism is required to hide this latency. For each warp state, the chart shows the average number of cycles spent in that state per issued instruction. Stalls are not always impacting the overall performance nor are they completely avoidable. Only focus on stall reasons if the schedulers fail to issue every cycle. When executing a kernel with mixed library and user code, these metrics show the combined values.

Warp Cycles Per Issued Instruction [cycle]	15.58	Avg. Active Threads Per Warp	7.80
Warp Cycles Per Executed Instruction [cycle]	15.67	Avg. Not Predicated Off Threads Per Warp	6.83

Thread Divergence
Est. Speedup: 56.67%

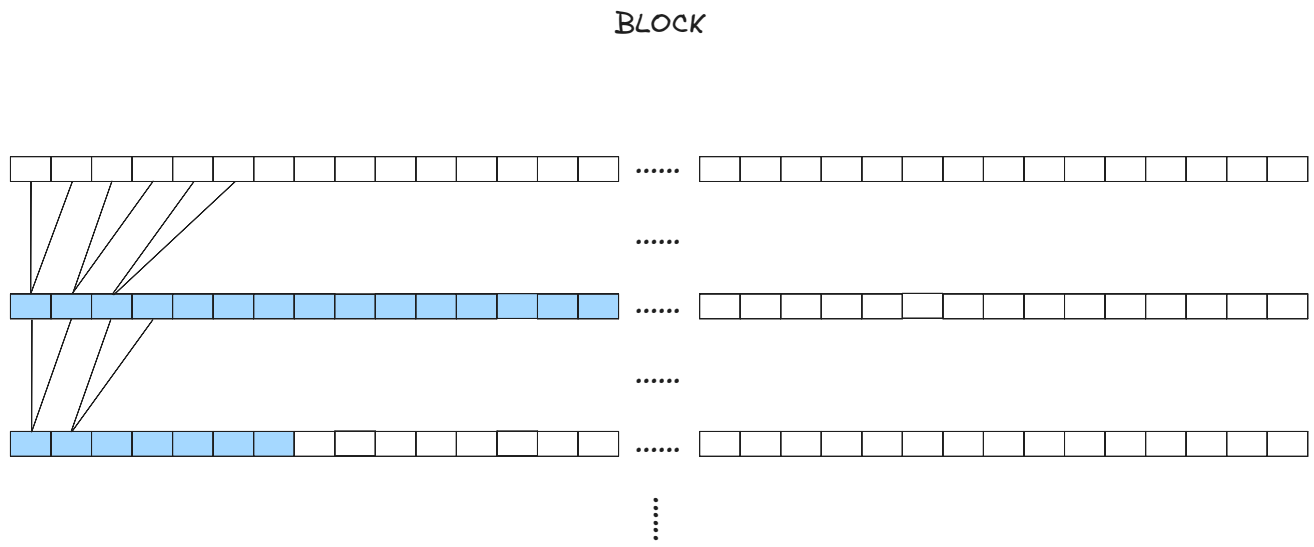
Instructions are executed in warps, which are groups of 32 threads. Optimal instruction throughput is achieved if all 32 threads of a warp execute the same instruction. The chosen launch configuration, early thread completion, and divergent flow control can significantly lower the number of active threads in a warp per cycle. This kernel achieves an average of 7.8 threads being active per cycle. This is further reduced to 6.8 threads per warp due to predication. The compiler may use predication to avoid an actual branch. Instead, all instructions are scheduled, but a per-thread condition code or predicate controls which threads execute the instructions. Try to avoid different execution paths within a warp when possible.

The following table lists the metrics that are key performance indicators:

Metric Name	Value	Guidance
smsp__thread_inst_executed_pred_on_per_inst_executed.ratio	6.82609	Increase the number of predicated-on threads per instruction towards 32
smsp__thread_inst_executed_per_inst_executed.ratio	7.79767	Increase the number of threads per instruction towards 32

注意到平均每个warp中即使被优化（编译器或架构？predicated-execution?）只有7.8个线程在工作。这是因为在SIMT中如果一个warp中存在分支，warp中thread不再并行，会变成串行。也就是说在warp中判断tid奇偶性这一步导致了分歧和一半（每次循环都会）的线程怠速，然后再其他线程执行完后再合并。

解决方法如下，直接使用**整个block前一半**的线程执行操作，这样可以利用更多连续的thread，也就更有可能使多个warp执行相同的操作而不发生分歧。




```
__global__ void reduce2(float *d_input, float *d_output)
{
    __shared__ float shared_mem[THREAD_PER_BLOCK];
    unsigned int tid = threadIdx.x;
    unsigned int global_tid = blockIdx.x * blockDim.x + tid;

    shared_mem[tid] = d_input[global_tid];
    __syncthreads();

    for (unsigned int i = 1; i < blockDim.x; i *= 2)
    {
        if (tid < blockDim.x / (i * 2))
        {
            shared_mem[2 * i * tid] += shared_mem[2 * i * tid + i];
```

```
            __syncthreads();
        }
    }

    if (tid == 0)
    {
        d_output[blockIdx.x] = shared_mem[tid];
    }
}
```

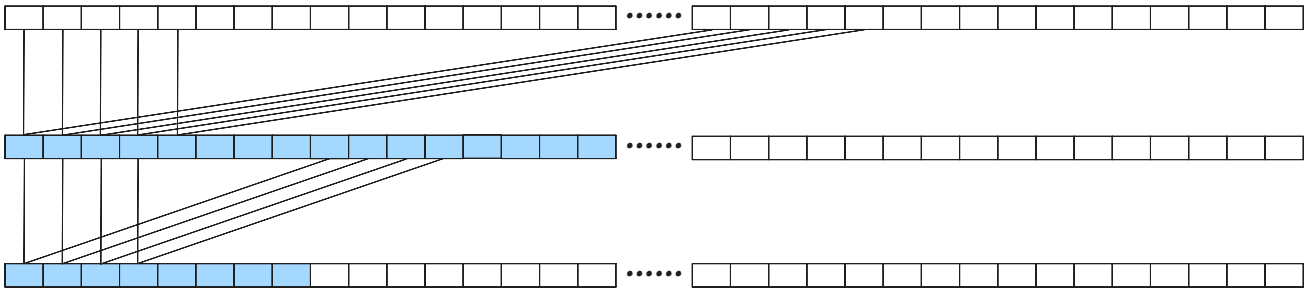
经过优化后可以得到平均每个warp有27.46个thread在工作，吞吐量是：$33,45.18\%$。

reduce_v3_no_bank_conflict

在reduce2的ncu性能分析中可以看出当前存在bank_conflicts。

	Instructions	Requests	Wavefronts	% Peak	Bank Conflicts
Shared Load	3276800	3276800	12555429	8.23	9175574
Shared Load Matrix	0	0			
Shared Store	2621440	2621440	7208960	1.18	4587520
Shared Store From Global Load	0	0	0	0	0
Shared Atomic	0	0	0	0	0
Other	-	-	37473513	28.10	79808
Total	5898240	5898240	57237902	37.51	13842902

bank_conflict是由于线程读取的数据的位置刚好是$2^i \times tid$和$2^i \times tid + 1$。而就导致在第一次迭代中0号thread和16号thread会同时访问bank0（32）和bank1（33）导致冲突，并且在多次迭代后会有更多路的冲突。解决方案是让同一个线程访问成32倍数的或尽可能远的数据，这样会让同bank下的数据（尽可能）总是被同一个thread访问。



```
__global__ void reduce3(float *d_input, float *d_output)
{
    __shared__ float shared_mem[THREAD_PER_BLOCK];
    unsigned int tid = threadIdx.x;
    unsigned int global_tid = blockIdx.x * blockDim.x + tid;

    shared_mem[tid] = d_input[global_tid];
    __syncthreads();
```

```
    for (unsigned int i = 1; i < blockDim.x; i *= 2)
    {
        if (tid < blockDim.x / (i * 2))
        {
            shared_mem[tid] += shared_mem[tid + blockDim.x / (i *
2)];
            __syncthreads();
        }
    }

    if (tid == 0)
    {
        d_output[blockIdx.x] = shared_mem[tid];
    }
}
```

此时bank conflict已经被尽可能消除，吞吐量是：$33.79, 45.56\%$。

reduce_v4_add_during_load_ab

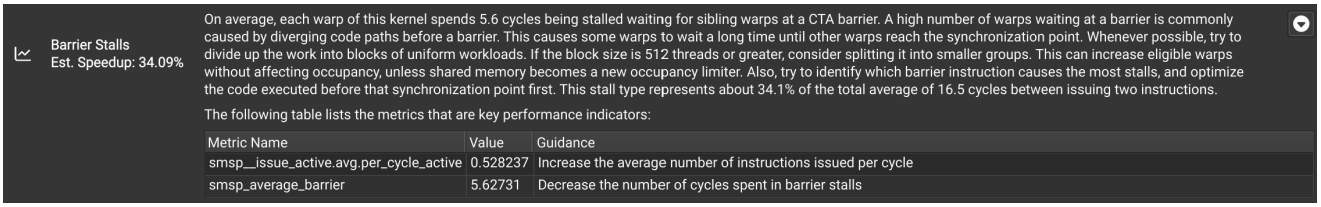
注意到目前位置，每次都有一半的thread在每次循环中处于限制（idle）状态，这造成了计算资源的浪费。所以可以通过让在一开始的时候就让一个线程负责两个数据的读取（并求和）。
这个目标可以通过减少每一个block的thread数或者减少block数量实现。
减少block数量或thread数量kernel都是一样的，只影响blockDim和GridDim：

```
__global__ void reduce4_a(float *d_input, float *d_output)
{
    __shared__ float shared_mem[THREAD_PER_BLOCK];
    unsigned int tid = threadIdx.x;
    unsigned int global_tid = 2 * blockIdx.x * blockDim.x + tid;

    shared_mem[tid] = d_input[global_tid] + d_input[global_tid +
blockDim.x];
    __syncthreads();

    for (unsigned int i = 1; i < blockDim.x; i *= 2)
    {
        if (tid < blockDim.x / (i * 2))
        {
            shared_mem[tid] += shared_mem[tid + blockDim.x / (i *
2)];
```

```
            __syncthreads();
        }
    }

    if (tid == 0)
    {
        d_output[blockIdx.x] = shared_mem[tid];
    }
}
```

减少block给出吞吐量：$68.11, 44.59\%$，减少thread数目给出：$55.5, 53.85\%$。猜测是因为block数量的增加会加大调度开销，而一个block内一个SM调度上线是1024或2048，由于我们的thread在不减少的情况下也是没有超出限制的，所以调度开销不会变化。

reduce_v5_unroll_last_loop



在这一步我们要解决多余的线程同步问题，在之前写的循环中，每次选取前一半的thread，然后同步这些thread再进入下一个循环。但是在最后只剩下32个thread时，这些thread全由一个warp调度，所以不存在执行不同步的问题，所以也不需要同步。我们可以把最后32个thread的循环展开，不做sync。

```
__device__ void warpReduce(volatile float *cache, unsigned 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 reduce5(float *d_input, float *d_output)
{
    __shared__ float shared_mem[THREAD_PER_BLOCK];
    unsigned int tid = threadIdx.x;
    unsigned int global_tid = 2 * blockIdx.x * blockDim.x + tid;
```

```
    shared_mem[tid] = d_input[global_tid] + d_input[global_tid +
blockDim.x];
    __syncthreads();

    // Division is not effective, using bit-ops. And we unfold the
last loop to reduce the sync time.

    for (unsigned int i = blockDim.x / 2; i > 32; i >>= 1)
    {
        if (tid < i)
        {
            shared_mem[tid] += shared_mem[tid + i];
            __syncthreads();
        }
    }

    if (tid < 32)
        warpReduce(shared_mem, tid);

    if (tid == 0)
        d_output[blockIdx.x] = shared_mem[tid];
}
```

这里对`warpReduce`的输入加上volatile关键字是为了阻止编译器优化，否则编译器优化后可能会带来乱序执行或寄存器数据和计算并不同步等问题?
在ncu中可以看到此时duration从将近4s到1s。吞吐量已经达到了：$242.79, 94.93\%$
。

reduce_v6_completely_unroll

```
__global__ void reduce6(float *d_input, float *d_output)
{
    __shared__ float shared_mem[THREAD_PER_BLOCK];
    unsigned int tid = threadIdx.x;
    unsigned int global_tid = 2 * blockIdx.x * blockDim.x + tid;

    shared_mem[tid] = d_input[global_tid] + d_input[global_tid +
blockDim.x];
    __syncthreads();

    if(THREAD_PER_BLOCK >= 512){
        if(tid < 256) shared_mem[tid] += shared_mem[tid + 256];
        __syncthreads();
```

```
    }

    if(THREAD_PER_BLOCK >= 256){
        if(tid < 128) shared_mem[tid] += shared_mem[tid + 128];
        __syncthreads();
    }

    if(THREAD_PER_BLOCK >= 128){
        if(tid < 64) shared_mem[tid] += shared_mem[tid + 64];
        __syncthreads();
    }

    if (tid < 32) warpReduce(shared_mem, tid);

    if (tid == 0) d_output[blockIdx.x] = shared_mem[tid];
}
```

在这一步中我们展开所有的循环以消除循环的额外开销，但是带来的增益不是很大，有可能是因为已经优化到硬件性能上限了：$249.34, 97.49\%$。

reduce_v7_block_num

我们在每一个block上已经优化到了硬件性能的上限。所以下一步是优化block的形状。此时我们固定block_num为1024，平均每个线程处理128个元素。
这时duration已经到了500us。

reduce_v8_shuffle

```
__device__ __forceinline__ float warpReduce(float sum)
{
    // Here we use register of each warp to perform reduction. The
register is commutable for threads in one warp
    if (blockDim.x >= 32)
        sum += __shfl_down_sync(0xffffffff, sum, 16);
    if (blockDim.x >= 16)
        sum += __shfl_down_sync(0xffffffff, sum, 8);
    if (blockDim.x >= 8)
        sum += __shfl_down_sync(0xffffffff, sum, 4);
    if (blockDim.x >= 4)
        sum += __shfl_down_sync(0xffffffff, sum, 2);
    if (blockDim.x >= 2)
        sum += __shfl_down_sync(0xffffffff, sum, 1);
    return sum;
```

```
}

template <unsigned int NUM_PER_THREAD>
__global__ void reduce7(float *d_input, float *d_output)
{
    float sum = 0;
    unsigned int tid = threadIdx.x;
    unsigned int global_tid = blockIdx.x * (blockDim.x *
NUM_PER_THREAD) + tid;

#pragma unroll
    for (int i = 0; i < NUM_PER_THREAD; ++i)
    {
        sum += d_input[global_tid + i * blockDim.x];
    }
    __syncthreads();

    static __shared__ float warpLevelSums[WARP_SIZE]; // Why is 32?
due to thread limitation of a block or?
    const int laneId = tid % WARP_SIZE;
    const int warpId = tid / WARP_SIZE;

    sum = warpReduce(sum);

    // Store the first value of each warp register to block shared
mem
    if (laneId == 0)
        warpLevelSums[warpId] = sum;
    __syncthreads();

    // We choose several threads to load the result of each warp
reduction. N.B. typical pattern will not cause warp divergence.
    sum = (tid < THREAD_PER_BLOCK / WARP_SIZE) ?
warpLevelSums[laneId] : 0;
    // Final reduction
    if (warpId == 0)
        sum = warpReduce(sum);

    if (tid == 0)
        d_output[blockIdx.x] = sum;
}
```

在这一步我们使用寄存器而不是shared mem，直接在kernel内使用float变量就会将元素存入寄存器中，同一个warp的寄存器是可以互相访问的。我们通过寄存器的shuffle函数就可以完成warp内的reduce。随后将block内所有的warp的结果结合在一起，再次存入一个32宽度的寄存器中shuffle的到block的最终结果。此时性能没有提示，怀疑是已经达到了硬件带宽的瓶颈。