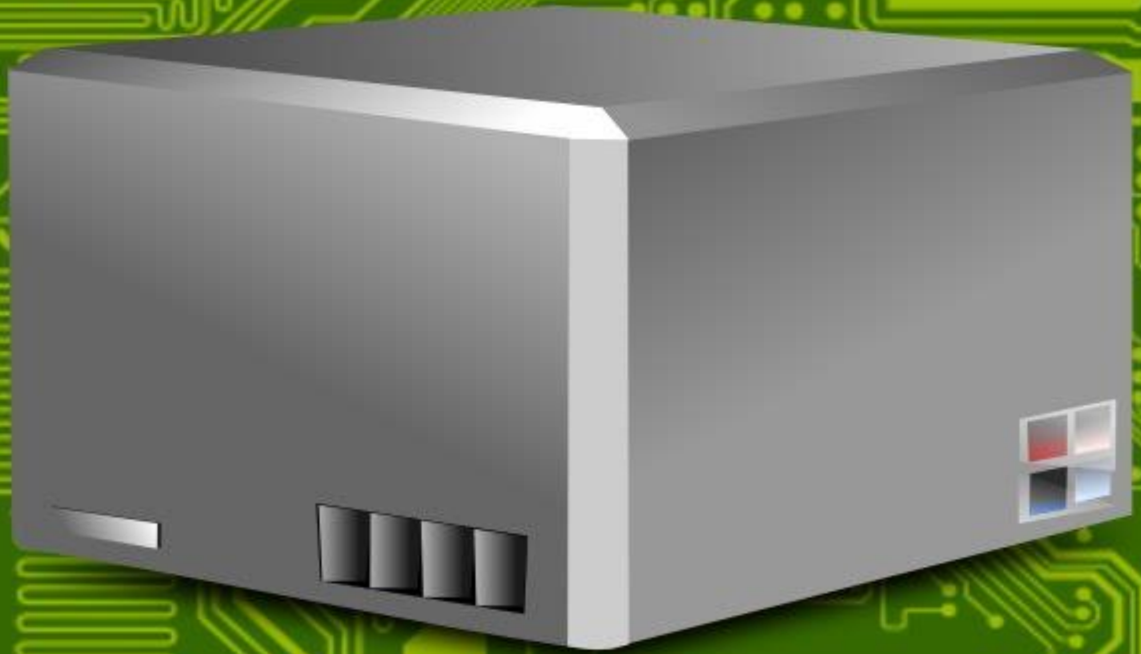
GPU Computing



规约算法

Hui Liu
Email: hui.sc.liu@gmail.com

Optimization Goal

- Get maximum GPU performance

- Two components:
  - – Compute Bandwidth: GFLOPs
  - – Memory Bandwidth: GB/s

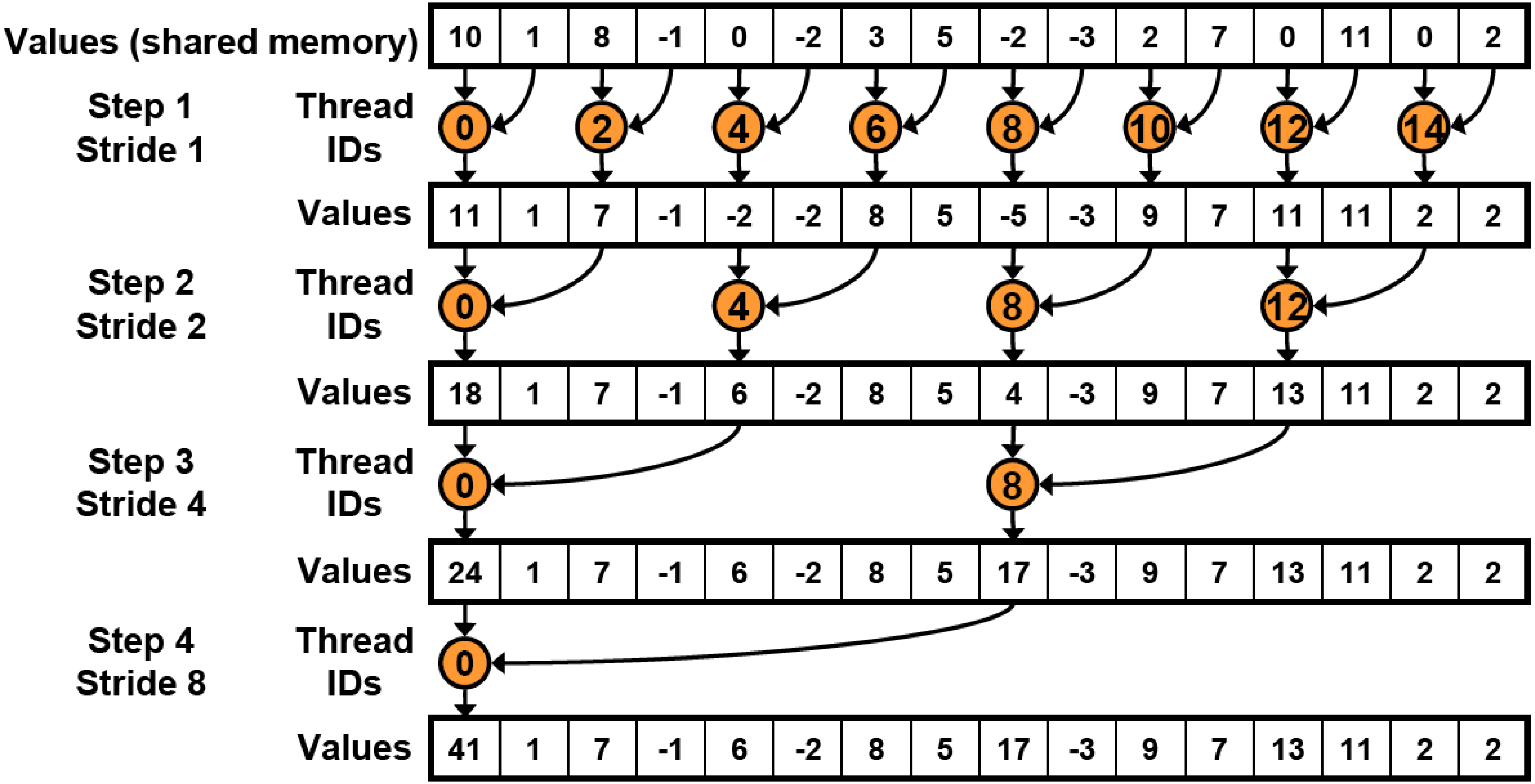
Optimization Goals Cont.

- Reductions typically have low arithmetic intensity
  - FLOPs/element loaded from memory
- So, bandwidth will be the limiter
- For the **ASUS ENGTX280 OC**
  - 512-bit interface, 1.14GHz DDR3
  - 512 / 8 x 1.14 x 2 = **145.92 GB/s**

Reduction #1: Strategy

- Load data:
  - Each thread loads one element from global memory to shared memory
- Actual Reduction: Proceed in log*N* steps
  - A thread reduces two elements
    - The first two elements by the first thread
    - The next two by the next thread
    - And so, on
  - At the end of each step:
    - Deactivate half of the threads
  - Terminate: when one thread left
- Write back to global memory

Reduction Steps



Values (shared memory)		10	1	8	-1	0	-2	3	5	-2	-3	2	7	0	11	0	2
Step 1 Stride 1	Thread IDs	0		2		4		6		8		10		12		14	
	Values	11	1	7	-1	-2	-2	8	5	-5	-3	9	7	11	11	2	2
Step 2 Stride 2	Thread IDs	0				4				8				12			
	Values	18	1	7	-1	6	-2	8	5	4	-3	9	7	13	11	2	2
Step 3 Stride 4	Thread IDs	0								8							
	Values	24	1	7	-1	6	-2	8	5	17	-3	9	7	13	11	2	2
Step 4 Stride 8	Thread IDs	0															
	Values	41	1	7	-1	6	-2	8	5	17	-3	9	7	13	11	2	2

Reduction #1 Code: Interleaved Accesses

```
__global__ void reduce0(int *g_idata, int *g_odata) {
    extern __shared__ int sdata[];

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

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

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

Performance for kernel #1

	Time (2^{22} ints)	Bandwidth
Kernel 1: interleaved addressing with divergent branching	**8.054 ms**	**2.083 GB/s**

Note: Block size = 128 for all experiments

Bandwidth calculation:

Each block processes 128 elements and does:

128 reads & 1 write

We only care about global memory

At each kernel/step:

N (element reads) + N / 128 (element writes)

Every kernel/step reduces input size by 128x

next set N = N / 128

So for:

N = 4194304

Accesses = 4227394

Each access is four bytes

Caveat:

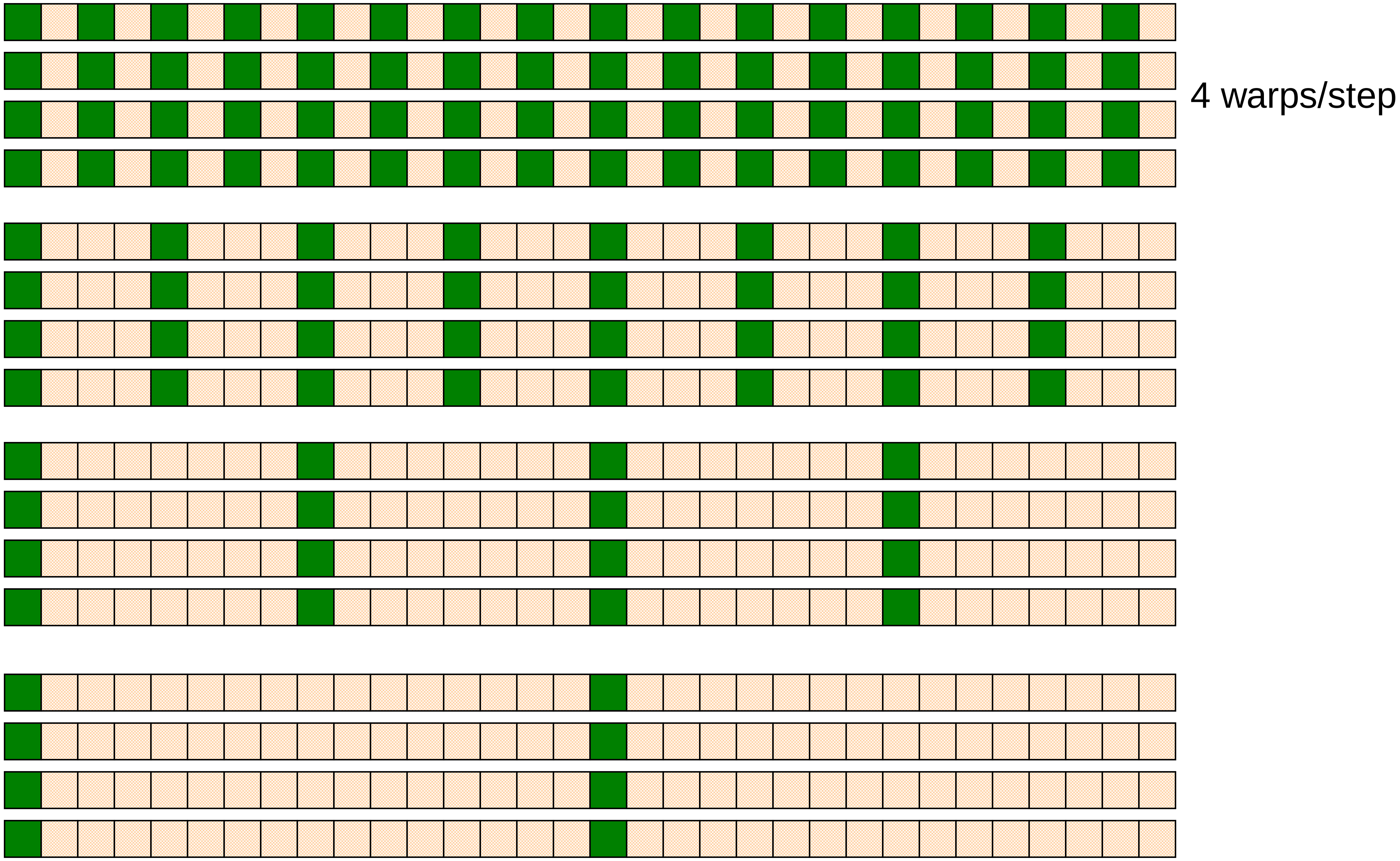
results are for a G80 processor

Reduction #1 code: Interleaved Accesses

```
__global__ void reduce0 (int *g_idata, int *g_odata) {
    extern __shared__ int sdata[];

    // each thread loads one element from global to shared mem
    unsigned int tid = threadIdx.x;
    unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
    sdata[tid] = g_idata[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) g_odata[blockIdx.x] = sdata[0];
}
```

Highly divergent code leads to very poor performance

Divergent Branching: Warp Control Flow



THANK YOU