Lec7: Reductions

Textbook chapter 10

Introduction

- A **reduction** is a very common computation primitive—deriving a single value out of an array of values using **binary operator.**
- E.g.: reduction(+): sum of array: s = a0+a1 + .. an
  reduction(*): prod of array: s = a0*a1*...*an
  reduction(max): s = max(a0, max(a1, ..., an)))..) = max(a0,...,an)
- The binary operator is **commutative and associative**
- A very important parallel primitive as well as its computational structure can be parallelized

General Reduction

```
acc = IDENTITY;
for (i = 0; i < n; i++)
    acc = op(acc, a[i]);
```

- For add reduction,
  IDENTITY = 0
  op = +
- For max reduction,
  IDENTITY = -infinity
  op = max(.,.)
- For prod reduction,
  IDENTITY = 1
  op = *
- …

Reduction tree

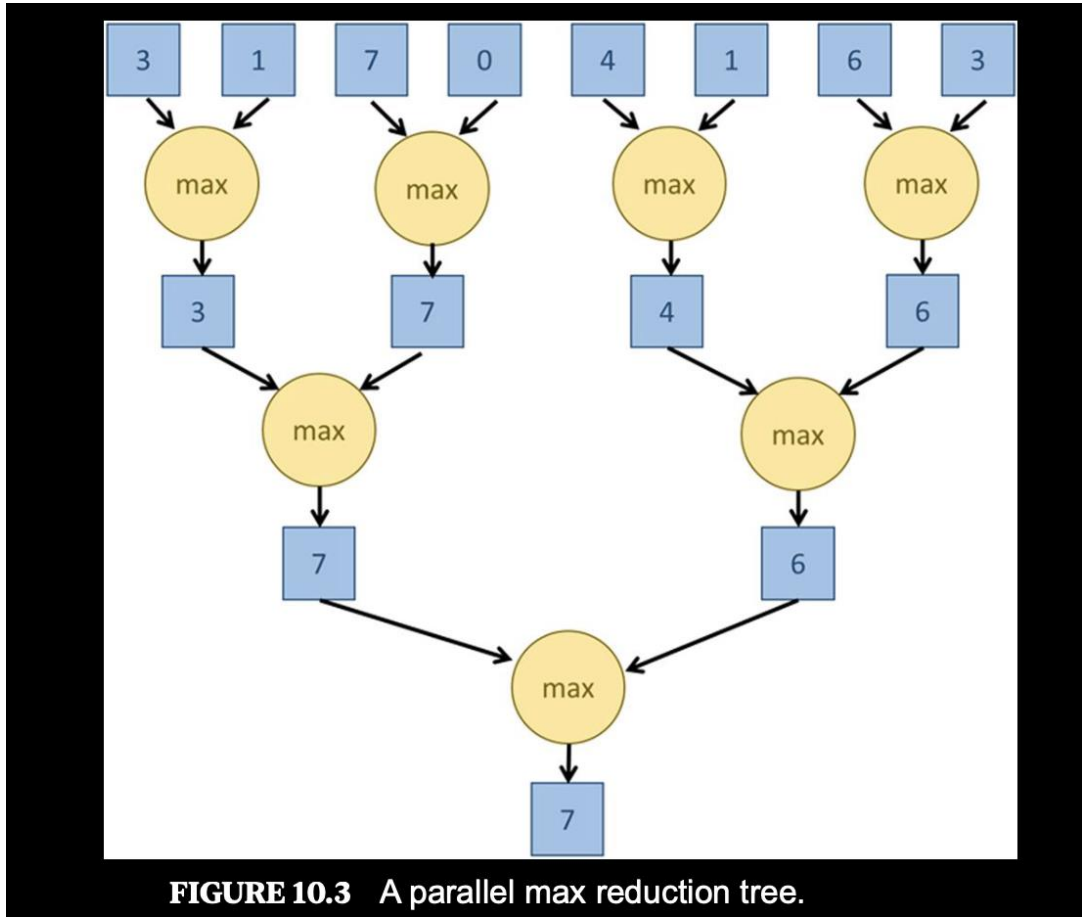

FIGURE 10.3 A parallel max reduction tree.

- Any reduction can be parallelized visualized as a tree
- Time (rounds) flow from top to bottom.
- The ops at the same round can be done in parallel.
- Order of evaluation is different from the sequential for loop, but it's OK because op is associative
- #rounds = O(log n) vs sequential O(n)
- The O(log n) is actually optimal

Simple Reduction Kernel

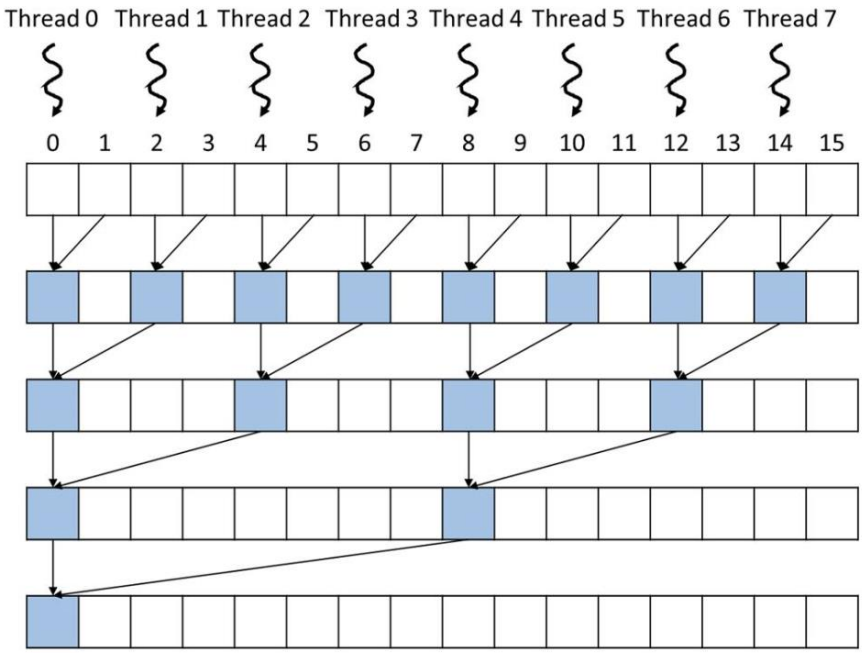

FIGURE 10.7 The assignment of threads ("owners") to the `input` array locations and progress of execution over time for the `SimpleSumReudctionKernel` in Fig. 10.6. The time progresses from top to bottom, and each level corresponds to one iteration of the for-loop.

- Start with 1 Thread block, with a fixed maximum array size

Single Thread Block Reduction

```
// compute the sum of input[0:n]; n <=256
__global__ void add_reduction1(float *input, float *output, int n)
{
    for (int stride = 1; stride <= blockDim.x; stride *= 2) {
        if (threadIdx.x % (2*stride) == 0 && threadIdx.x + stride < n) {
            input[threadIdx.x] += input[threadIdx.x + stride];
        }
        __syncthreads();
    }
    if (threadIdx.x == 0) {
        *output = input[0];
    }
}
```

Minimizing Control Divergence

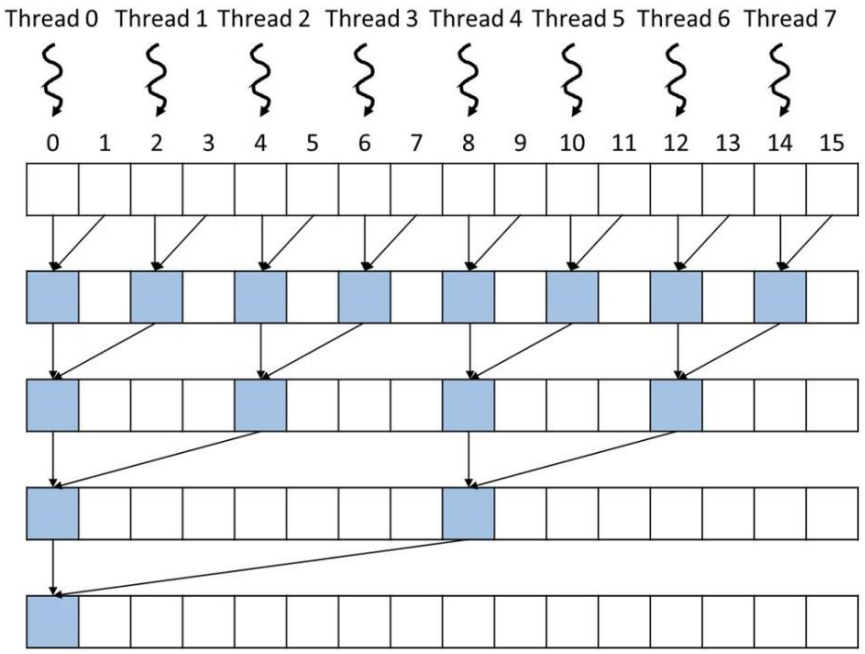

FIGURE 10.7 The assignment of threads ("owners") to the `input` array locations and progress of execution over time for the `SimpleSumReudctionKernel` in Fig. 10.6. The time progresses from top to bottom, and each level corresponds to one iteration of the for-loop.

- Round by round, less and less threads are active
- Utilization of the SIMD lanes?
- Round 1: 50%
  Round 2: 25%
  Round 3: 12.5%
- What's the utilization of the SIMD lanes for n=256, blockDim.x=128?
- Utilization := total active threads/total SIMD lanes:
  About 255 active useful lane op
  Total lane op: 4*5*32 + (2+1)*32=736
  Utilization = 255/736 = 35%

Reducing Control Divergence

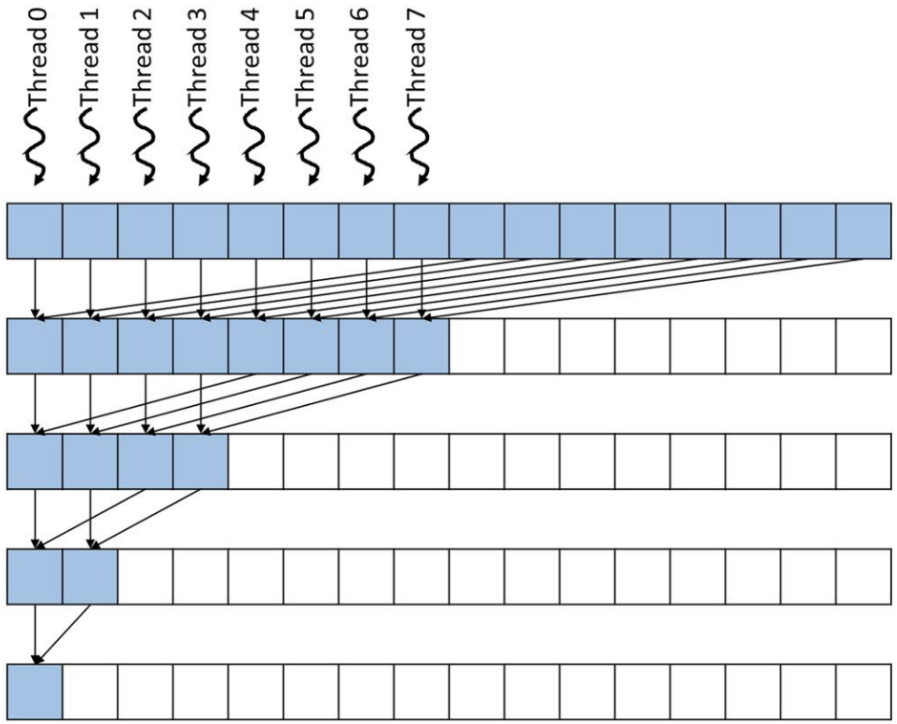

FIGURE 10.8 A better assignment of threads to input array locations for reduced control divergence.

- Now the first few rounds, utilization is 100%
- Utilization only degrades in later rounds.
- What's the new utilization?
  # active threads still 255
  # lanes used:
  (4+2+1 + 5)*32 = 384
  around half of divergent kernel

Minimal-divergent Reduction

```
// blockDim.x = 128.
__global__ void add_reduction2(float *input, float *output, int n)
{
    for (int section=blockDim.x; section >0; section /= 2) {
        if (threadIdx.x < section/2) {
            input[threadIdx.x] += input[threadIdx.x + section];
        }
        __syncthreads();
    }
    if (threadIdx.x == 0) {
        *output = input[0];
    }
}
```

Minimizing global memory access explicitly

- The kernel writes intermediary results into global memory, and then read them in subsequent rounds
- A better idea is to explicitly store the intermediate data in shared memory.

```
// blockDim.x = 128
__global__ void add_reduction3(float *input, float *output, int n)
{
    __shared__ float partial[blockDim.x];
    partial[threadIdx.x] = input[threadIdx.x] + input[threadIdx.x + blockDim.x];
    for (int section=blockDim.x/2; section > 0; section /= 2) {
        __syncthreads();
        if (threadIdx.x < section) {
            partial[threadIdx.x] += partial[threadIdx.x + section];
        }
    }
    if (threadIdx.x == 0) {
        *output = partial[0];
    }
}
```

Arbitrary input length?

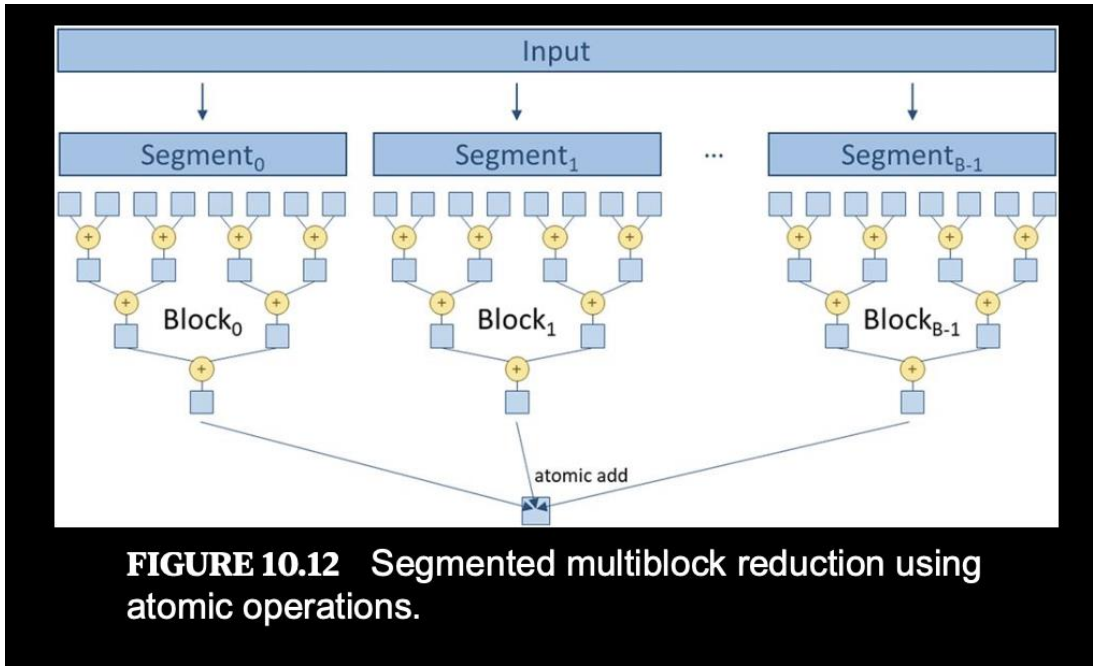

FIGURE 10.12 Segmented multiblock reduction using atomic operations.

- Break the input down into fixed-size segment, each one “reduced” by a thread block.
- Each block produces a single reduced result
- An extra step at the end to reduce the results from all thread blocks
  - Could be atomicAdds
  - Or another round of tree-reduction! (if many blocks are involved)

General Kernel

```
// array size n is not limited; multi-thread-block reduction
__global__ void add_reduction4(float *input, float *output, int n)
{
    __shared__ float partial[blockDim.x];
    partial[threadIdx.x] = input[threadIdx.x] + input[threadIdx.x + blockDim.x];
    for (int section=blockDim.x/2; section > 0; section /= 2) {
        __syncthreads();
        if (threadIdx.x < section) {
            partial[threadIdx.x] += partial[threadIdx.x + section];
        }
    }
    if (threadIdx.x == 0) {
        atomicAdd(output, partial[0]);
    }

}
```

Thread Coarsening

- For previous kernels, we decompose the workload at fine granularity; launching N/2 threads for length N input.
- Consider thread coarsening—
- Essentially, each (active) thread adds not only 2 numbers, but 4.
- Less threads N/2 -> N/4
- Less rounds: log_2(N) -> log_4(N)
- Each thread and each round is bigger

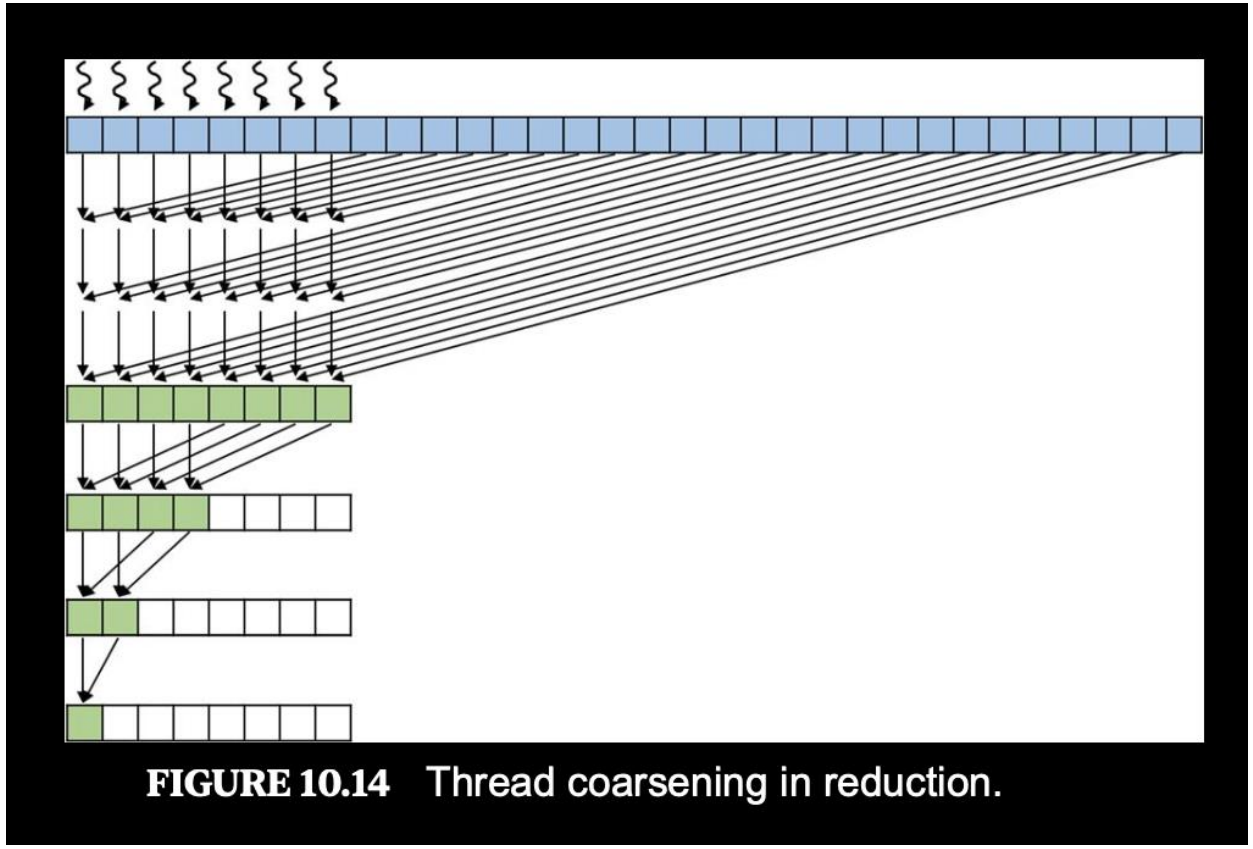

FIGURE 10.14 Thread coarsening in reduction.

Let’s write code: Optimization Goal

- What would be our goal for a certain sized array? Strive for peak performance of GPU. But by what metric?
  - GB/s?
  - GFLOP/s?
- Arithmetic intensity of Reduction:
  - 0.25 FLOP/B (assuming 4B element like float)
- Low arithmetic intensity (of problem! Not a particular impl)
  - Should be bottlenecked by memory bandwidth
  - Therefore GB/s is more appropriate metric
  - RTX 3080 has global memory bandwidth 760GB/s

Kernel1: naïve tree reduction: interleaving

```cpp
__global__ void reduce1(int *input, int *output) {
    extern __shared__ int sdata[];
    int i = blockIdx.x * blockDim.x + threadIdx.x;

    sdata[threadIdx.x] = input[i];
    __syncthreads();
    for (int s=1; s < blockDim.x; s *= 2) {
        if (threadIdx.x % (2*s) == 0) {
            sdata[threadIdx.x] += sdata[threadIdx.x + s];
        }
        __syncthreads();
    }
    if (threadIdx.x == 0) {
        output[blockIdx.x] = sdata[0];
    }
}
```

```cpp
// Multi-stage reduction using reduce1.
while (n > 1) {
    // Each block handles blockSize elements.
    int gridSize = (n + blockSize - 1) / blockSize;
    // Launch kernel with dynamic shared memory (blockSize * sizeof(int)).
    reduce1<<<gridSize, blockSize, blockSize * sizeof(int)>>>(d_in, d_out);
    CHECK_CUDA(cudaGetLastError());
    n = gridSize;
    // Swap pointers for next stage.
    int *temp = d_in;
    d_in = d_out;
    d_out = temp;
}
// Copy the final result from device to host.
int result_reduce1 = 0;
CHECK_CUDA(cudaMemcpy(&result_reduce1, d_in, sizeof(int), cudaMemcpyDeviceToHost));
```

```
numElements = 16777216
reduce1 result: 16777216, expected: 16777216
reduce1 execution time: 0.963488 ms, Bandwidth: 64.868479 GB/s
```

Kernel2: tree reduction: non-divergent

```
// non-divergent
__global__ void reduce2(int *input, int *output) {
    extern __shared__ int sdata[];
    int i = blockIdx.x * blockDim.x + threadIdx.x;

    sdata[threadIdx.x] = input[i];
    __syncthreads();
    for (int s=blockDim.x/2; s>=1; s /= 2) {
        if (threadIdx.x < s) {
            sdata[threadIdx.x] += sdata[threadIdx.x + s];
        }
        __syncthreads();
    }
    if (threadIdx.x == 0) {
        output[blockIdx.x] = sdata[0];
    }
}
```

- Minimal warp divergence
- Also shared memory access becomes bank conflict free

```
reduce2 result: 16777216, expected: 16777216
reduce2 execution time: 0.366880 ms, Bandwidth: 170.355430 GB/s
```

Kernel3: Reduce one round and half the threads

```cpp
// first add during load, reduce number of threads
__global__ void reduce3(int *input, int *output) {
    extern __shared__ int sdata[];
    int i = blockIdx.x * blockDim.x * 2 + threadIdx.x;

    sdata[threadIdx.x] = input[i] + input[i + blockDim.x];
    __syncthreads();
    for (int s=blockDim.x/2; s>=1; s /= 2) {
        if (threadIdx.x < s) {
            sdata[threadIdx.x] += sdata[threadIdx.x + s];
        }
        __syncthreads();
    }
    if (threadIdx.x == 0) {
        output[blockIdx.x] = sdata[0];
    }
}
```

- Fusing the load from global memory to shared memory with the first round
- Reduce ½ threads
- Reduce one round of shared memory read and write.

```
reduce3 result: 16777664, expected: 16777216
reduce3 execution time: 0.207744 ms, Bandwidth: 300.851044 GB/s
```

Kernel 4: Unrolling the loop (the last 6 iterations)

```cpp
// first add during load, reduce number of threads;
// add loop unrolling to reduce loop overhead and #instructions,
// and unblock Instruction Level Parallelism( ILP )
__global__ void reduce4(int *input, int *output) {
    extern __shared__ int sdata[];
    int i = blockIdx.x * blockDim.x * 2 + threadIdx.x;

    sdata[threadIdx.x] = input[i] + input[i + blockDim.x];
    __syncthreads();
    for (int s=blockDim.x/2; s>32; s /= 2) {
        if (threadIdx.x < s) {
            sdata[threadIdx.x] += sdata[threadIdx.x + s];
        }
        __syncthreads();
    }
    if (threadIdx.x < 32) {
        warpReduce(sdata, threadIdx.x);
    }
    if (threadIdx.x == 0) {
        output[blockIdx.x] = sdata[0];
    }
}
```

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

```
reduce4 result: 16777664, expected: 16777216
reduce4 execution time: 0.129408 ms, Bandwidth: 482.968588 GB/s
```

Reduction: CUB

```
// Determine temporary device storage requirements for CUB.
void *d_temp_storage = NULL;
size_t temp_storage_bytes = 0;
cub::DeviceReduce::Sum(d_temp_storage, temp_storage_bytes,
                       d_input, d_cub_result, numElements);
CHECK_CUDA(cudaMalloc(&d_temp_storage, temp_storage_bytes));

CHECK_CUDA(cudaEventRecord(start));
cub::DeviceReduce::Sum(d_temp_storage, temp_storage_bytes,
                       d_input, d_cub_result, numElements);
CHECK_CUDA(cudaEventRecord(stop));
CHECK_CUDA(cudaEventSynchronize(stop));
milliseconds = 0;
CHECK_CUDA(cudaEventElapsedTime(&milliseconds, start, stop));
```

- CUB is part of CUDA distribution
- It's a library of collective parallel primitives, of which "reduce" is one of them.
- Here we use the DeviceReduce version; there are also BlockReduce and WarpReduce which are block-wise and warp-wise reduction.

```
CUB result: 16777216, expected: 16777216
CUB execution time: 0.117120 ms, Bandwidth: 533.640721 GB/s
```

Sidenote: Block-wise reduction and AtomicAdd

```cpp
// first add during load, reduce number of threads;
// add loop unrolling to reduce loop overhead and #instructions,
// and unblock Instruction Level Parallelism( ILP )
__global__ void reduce6(int *input, int *output) {
    extern __shared__ int sdata[];
    int i = blockIdx.x * blockDim.x * 2 + threadIdx.x;

    sdata[threadIdx.x] = input[i] + input[i + blockDim.x];
    __syncthreads();
    for (int s=blockDim.x/2; s>32; s /= 2) {
        if (threadIdx.x < s) {
            sdata[threadIdx.x] += sdata[threadIdx.x + s];
        }
        __syncthreads();
    }
    if (threadIdx.x < 32) {
        warpReduce(sdata, threadIdx.x);
    }
    if (threadIdx.x == 0) {
        atomicAdd(output, sdata[0]);
    }
}
```

- No multi-level, multi-kernel, each one reducing #elements by 256x
- Instead just a block-wide reduction, and then atomicAdd all the results from all blocks
- Atomic Global memory access are serialized

```
reduce6 result: 16777216, expected: 16777216
reduce6 execution time: 0.143168 ms, Bandwidth: 436.550060 GB/s
```