CS 380 - GPU and GPGPU Programming
Lecture 22: GPU Parallel Reduction

Markus Hadwiger, KAUST

Reading Assignment #13 (until Nov 29)

Read (required):

- Programming Massively Parallel Processors book, 3rd edition
  Chapter 9 (Parallel patterns – parallel histogram computation)
- Programming Massively Parallel Processors book, 3rd edition
  Chapter 13 (CUDA dynamic parallelism)

Read (optional):

- Prefix Sums and Their Applications, Guy Blelloch

`https://www.cs.cmu.edu/~guyb/papers/Ble93.pdf`

GPU Reduction

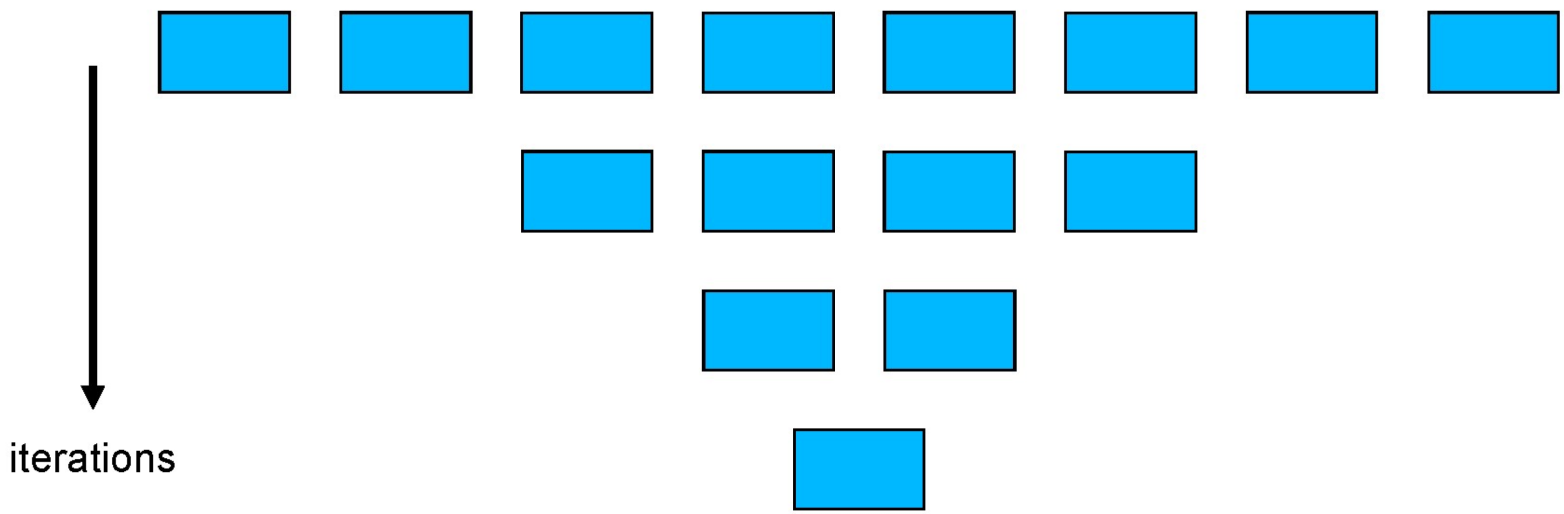
Parallel reduction is a basic parallel programming primitive;
see reduction operation on a stream, e.g., in Brook for GPUs

Example: Parallel Reduction

- **Given an array of values, “reduce” them to a single value in parallel**
- **Examples**
  - sum reduction: sum of all values in the array
  - Max reduction: maximum of all values in the array
- **Typical parallel implementation:**
  - Recursively halve # threads, add two values per thread
  - Takes log(n) steps for n elements, requires n/2 threads

Typical Parallel Programming Pattern

- **log(n) steps**



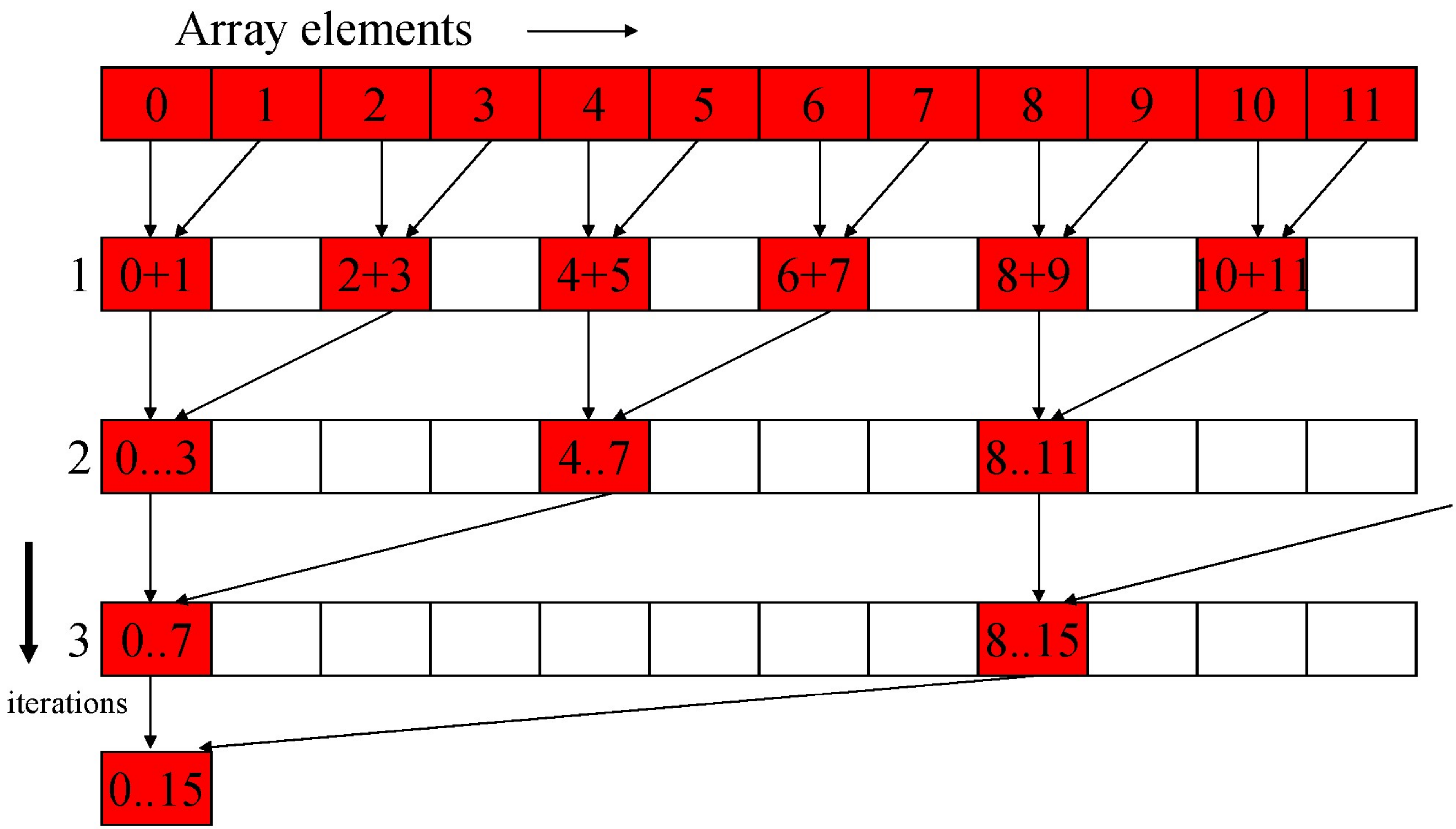
Helpful fact for counting nodes of full binary trees:
If there are N leaf nodes, there will be N-1 non-leaf nodes

Reduction – Version1

A Vector Reduction Example

- **Assume an in-place reduction using shared memory**
  - The original vector is in device global memory
  - The shared memory used to hold a partial sum vector
  - Each iteration brings the partial sum vector closer to the final sum
  - The final solution will be in element 0

Vector Reduction



A Simple Implementation

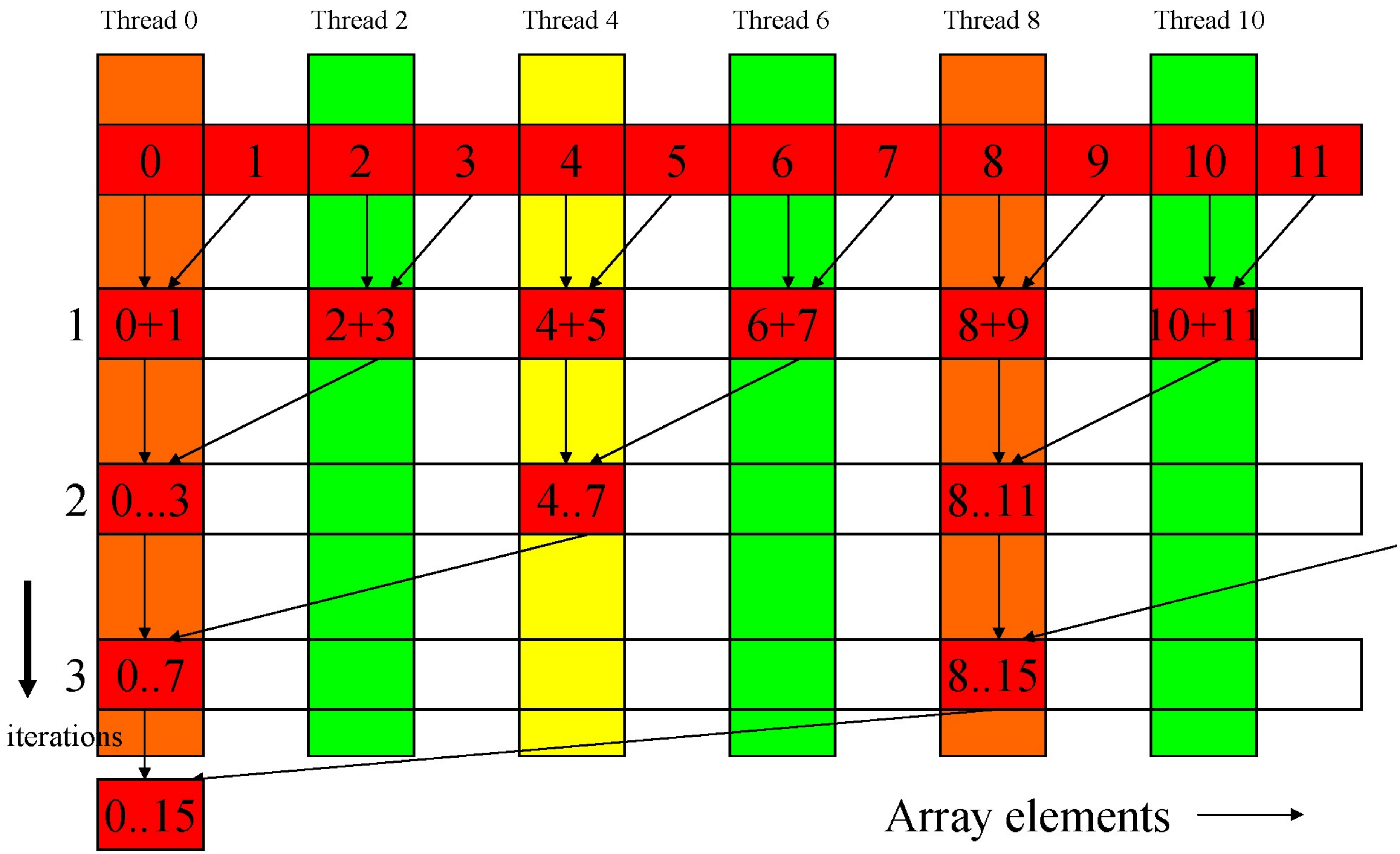
- **Assume we have already loaded array into**

```
__shared__ float partialSum[];

unsigned int t = threadIdx.x;

// loop log(n) times
for (unsigned int stride = 1;
     stride < blockDim.x;  stride *= 2)
{
  // make sure the sum of the previous iteration
  // is available
  __syncthreads();

  if (t % (2*stride) == 0)
     partialSum[t] += partialSum[t+stride];
}
```

Reduction #1: Interleaved Addressing

```
__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];
}
```

Vector Reduction with Branch Divergence



Some Observations

- **In each iterations, two control flow paths will be sequentially traversed for each warp**
  - Threads that perform addition and threads that do not
  - Threads that do not perform addition may cost extra cycles depending on the implementation of divergence

- **No more than half of threads will be executing at any time**
  - All odd index threads are disabled right from the beginning!
  - On average, less than ¼ of the threads will be activated for all warps over time.
  - After the 5^{th} iteration, entire warps in each block will be disabled, poor resource utilization but no divergence.
    - This can go on for a while, up to 4 more iterations ($512/32=16= 2^4$), where each iteration only has one thread activated until all warps retire

Short comings of the implementation

- **Assume we have already loaded array into**

```
__shared__ float partialSum[];

unsigned int t = threadIdx.x;
for (unsigned int stride = 1;
      stride < blockDim.x;  stride *= 2)
{
  __syncthreads();

  if (t % (2*stride) == 0)
     partialSum[t] += partialSum[t+stride];
}
```

BAD: Divergence due to interleaved branch decisions

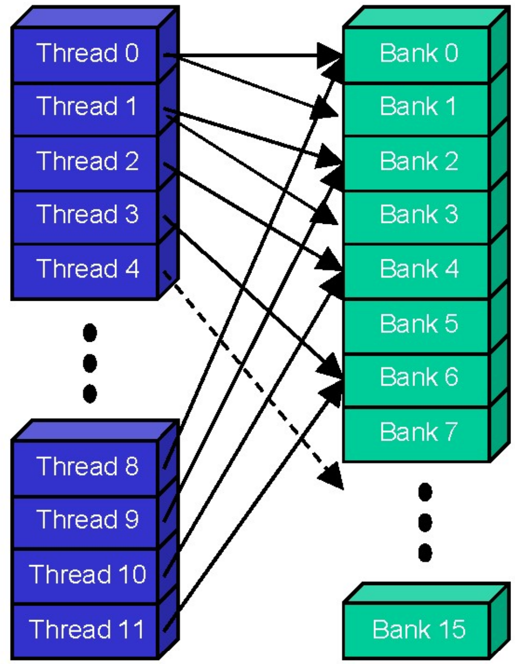
BAD: Bank conflicts due to stride

Reduction – Version2

Common Array Bank Conflict Patterns 1D

- **Each thread loads 2 elements into shared mem:**
  - 2-way-interleaved loads result in 2-way bank conflicts:

```
int tid = threadIdx.x;
shared[2*tid] = global[2*tid];
shared[2*tid+1] = global[2*tid+1];
```

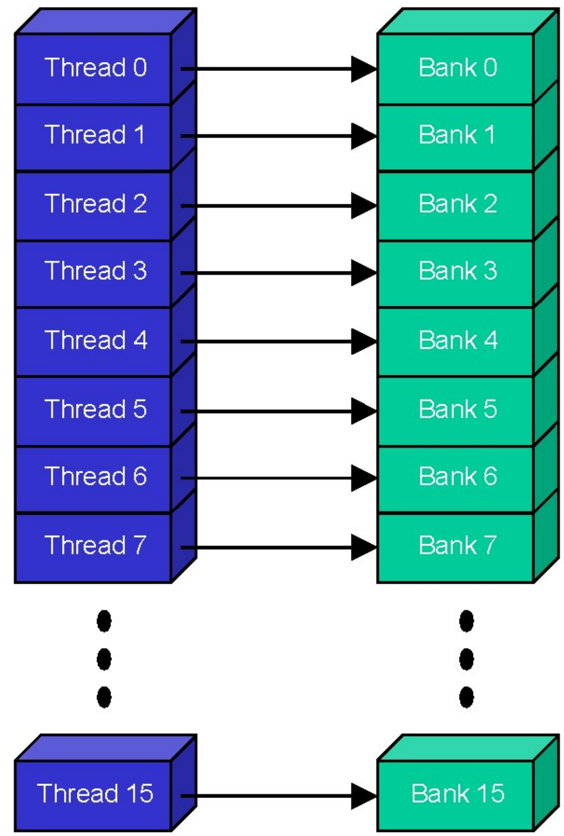


- **This makes sense for traditional CPU threads, locality in cache line usage and reduced sharing traffic.**
  - Not in shared memory usage where there is no cache line effects but banking effects

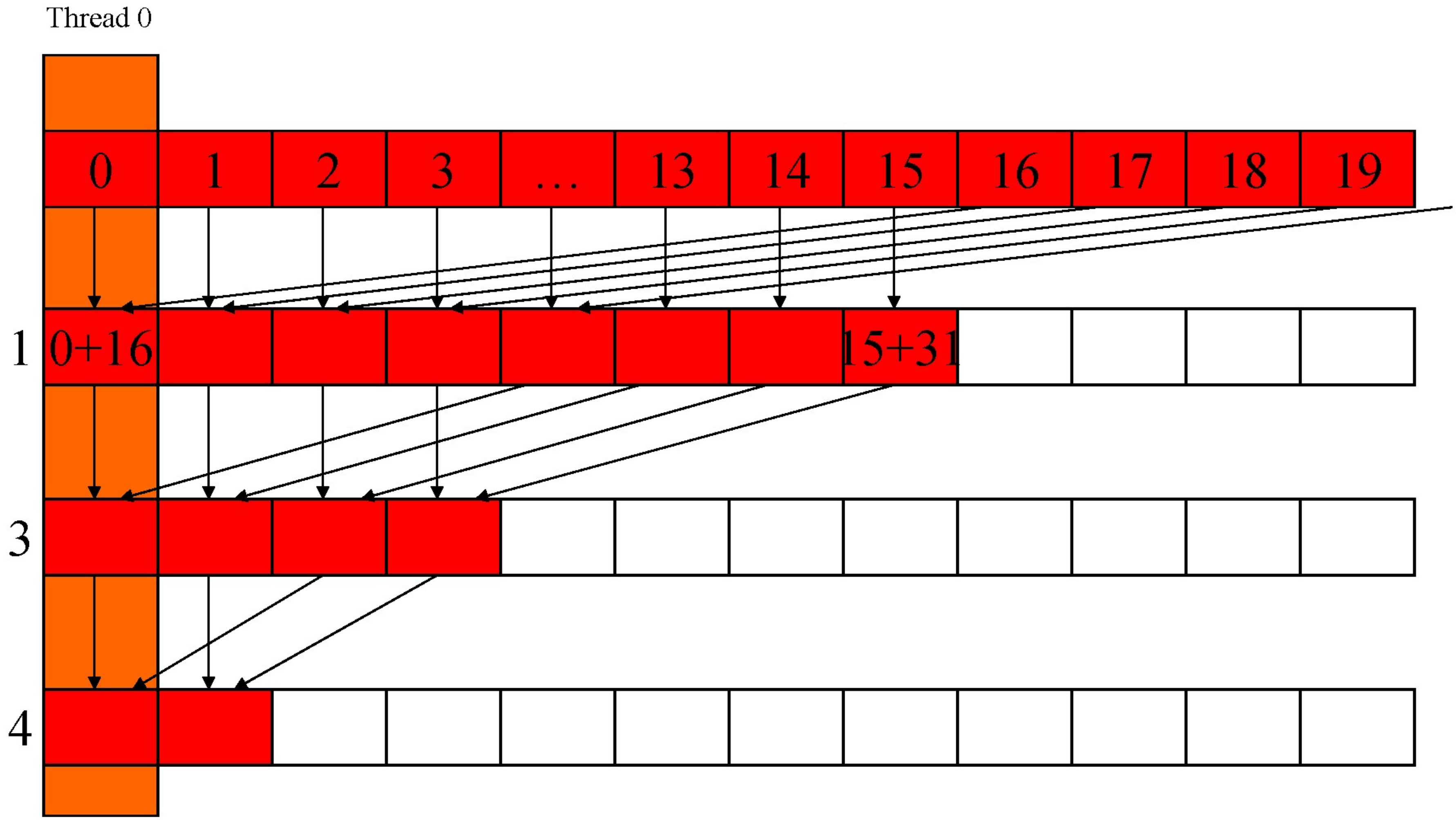
A Better Array Access Pattern

- Each thread loads one element in every consecutive group of `blockDim` elements.

```
shared[tid] = global[tid];
shared[tid + blockDim.x] =
  global[tid + blockDim.x];
```



A better implementation



A better implementation

- Assume we have already loaded array into

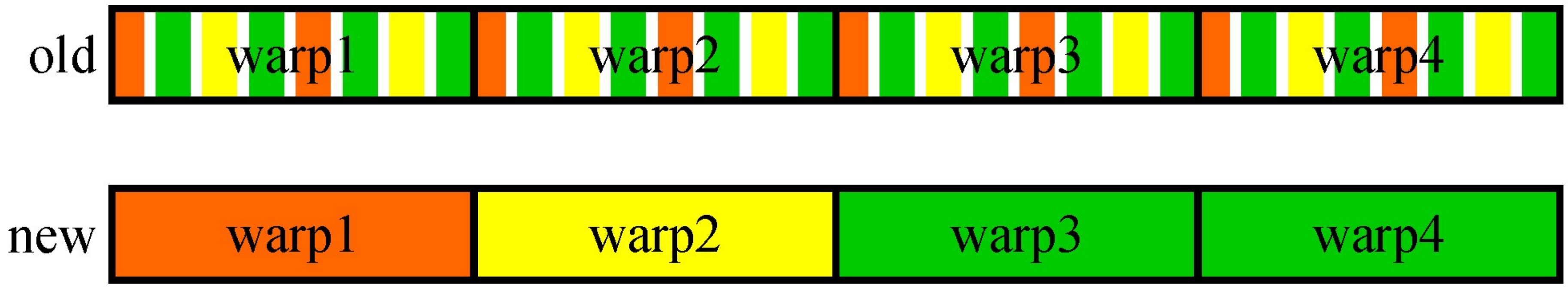
```
__shared__ float partialSum[];

unsigned int t = threadIdx.x;
for (unsigned int stride = blockDim.x;
     stride > 1;  stride >>=1)
{
  __syncthreads();
  if (t < stride)
     partialSum[t] += partialSum[t+stride];
}
```

if you want to fully retire warps, this should actually be:

```
if ( t < stride ) {
   partialSum[ t ] += partialSum[ t + stride ];
} else {
   break;
}
```

A better implementation

- **Only the last 5 iterations will have divergence**
- **Entire warps will be shut down as iterations progress**
  - For a 512-thread block, 4 iterations to shut down all but one warp in each block
  - Better resource utilization, will likely retire warps and thus blocks faster
- **Recall, no bank conflicts either**



Implicit Synchronization in a Warp

- **For last 6 loops only one warp active (i.e. tid's 0..31)**
  - Shared reads & writes SIMD synchronous within a warp
  - So skip `__syncthreads()` and unroll last 5 iterations

```
unsigned int tid = threadIdx.x
for (unsigned int d = n>>1; d
    __syncthreads();
    if (tid < d)
        shared[tid] += shared[
}
__syncthreads();
if (tid <= 32) {   // unroll last
    shared[tid] += shared[tid +
    shared[tid] += shared[tid +
    shared[tid] += shared[tid +
    shared[tid] += shared[tid +
    shared[tid] += shared[tid + 2];
    shared[tid] += shared[tid + 1];
}
```

This would not work properly is warp size decreases; need __synchthreads() between each statement!
However, having ___synchthreads() in if statement is problematic.

Look at CUDA SDK reduction example and slides!

Optimizing Parallel Reduction in CUDA

Mark Harris
NVIDIA Developer Technology

Parallel Reduction

- Common and important data parallel primitive
- Easy to implement in CUDA
  - Harder to get it right
- Serves as a great optimization example
  - We'll walk step by step through 7 different versions
  - Demonstrates several important optimization strategies

Final Optimized Kernel

```
template <unsigned int blockSize>
__global__ void reduce6(int *g_idata, int *g_odata, unsigned int n)
{
   extern __shared__ int sdata[];

   unsigned int tid = threadIdx.x;
   unsigned int i = blockIdx.x*(blockSize*2) + tid;
   unsigned int gridSize = blockSize*2*gridDim.x;
   sdata[tid] = 0;

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

   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(); }

   if (tid < 32) {
      if (blockSize >=  64) sdata[tid] += sdata[tid + 32];
      if (blockSize >=  32) sdata[tid] += sdata[tid + 16];
      if (blockSize >=  16) sdata[tid] += sdata[tid +  8];
      if (blockSize >=    8) sdata[tid] += sdata[tid +  4];
      if (blockSize >=    4) sdata[tid] += sdata[tid +  2];
      if (blockSize >=    2) sdata[tid] += sdata[tid +  1];
   }

   if (tid == 0) g_odata[blockIdx.x] = sdata[0];
}
```

out-of-bounds check missing, see SDK code

be careful that shared variables are declared *volatile*! see SDK code

```
template <unsigned int blockSize>
__device__ void warpReduce(volatile int *sdata, unsigned int tid) {
    if (blockSize >=  64) sdata[tid] += sdata[tid + 32];
    if (blockSize >=  32) sdata[tid] += sdata[tid + 16];
    if (blockSize >=  16) sdata[tid] += sdata[tid +  8];
    if (blockSize >=   8) sdata[tid] += sdata[tid +  4];
    if (blockSize >=   4) sdata[tid] += sdata[tid +  2];
    if (blockSize >=   2) sdata[tid] += sdata[tid +  1];
}
template <unsigned int blockSize>
__global__ void reduce6(int *g_idata, int *g_odata, unsigned int n) {
    extern __shared__ int sdata[];
    unsigned int tid = threadIdx.x;
    unsigned int i = blockIdx.x*(blockSize*2) + tid;
    unsigned int gridSize = blockSize*2*gridDim.x;
    sdata[tid] = 0;

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

    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(); }

    if (tid < 32) warpReduce(sdata, tid);
    if (tid == 0) g_odata[blockIdx.x] = sdata[0];
}
```

Final Optimized Kernel

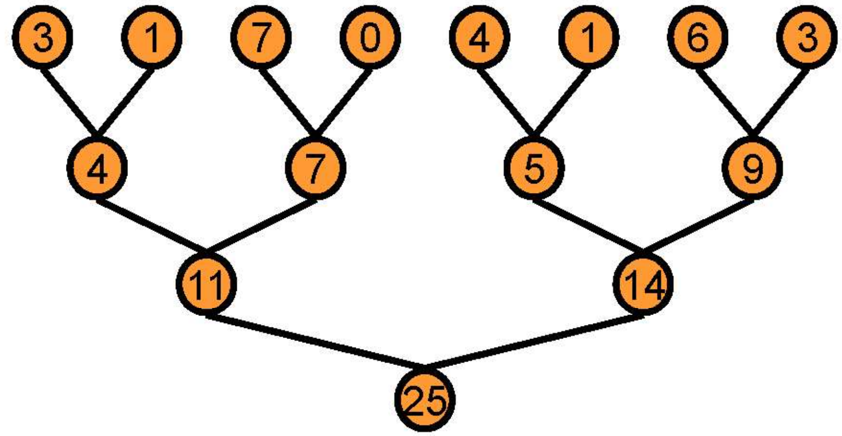
Invoking Template Kernels

- **Don't we still need block size at compile time?**
  - **Nope, just a switch statement for 10 possible block sizes:**

```
switch (threads)
    {
    case 512:
      reduce5<512><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;
    case 256:
      reduce5<256><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;
    case 128:
      reduce5<128><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;
    case 64:
      reduce5< 64><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;
    case 32:
      reduce5< 32><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;
    case 16:
      reduce5< 16><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;
    case  8:
      reduce5<  8><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;
    case  4:
      reduce5<  4><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;
    case  2:
      reduce5<  2><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;
    case  1:
      reduce5<  1><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;
    }
```

Parallel Reduction

- Tree-based approach used within each thread block



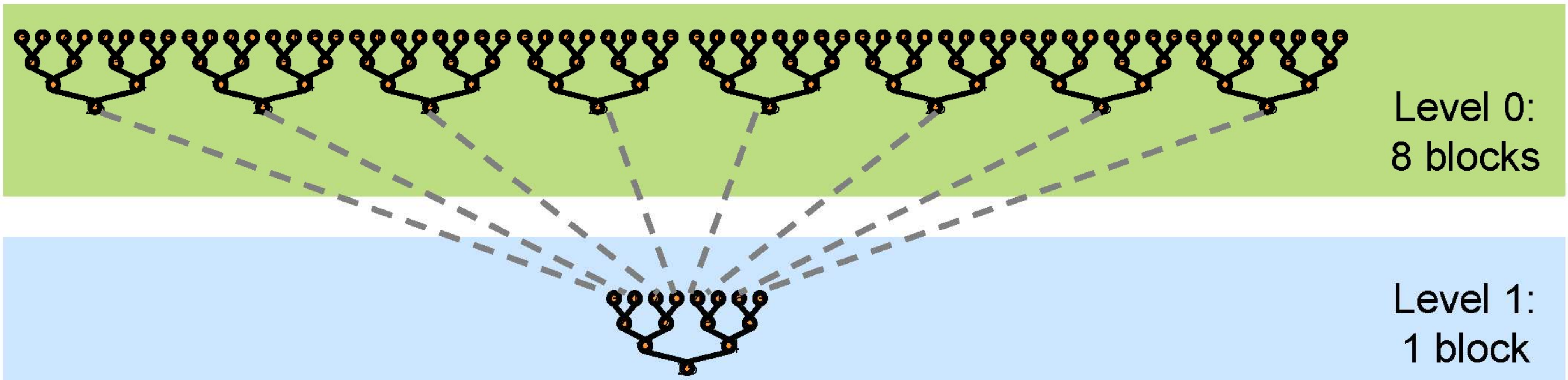
- Need to be able to use multiple thread blocks
  - To process very large arrays
  - To keep all multiprocessors on the GPU busy
  - Each thread block reduces a portion of the array
- But how do we communicate partial results between thread blocks?

Problem: Global Synchronization

- If we could synchronize across all thread blocks, could easily reduce very large arrays, right?
  - Global sync after each block produces its result
  - Once all blocks reach sync, continue recursively
- But CUDA has no global synchronization. Why?
  - Expensive to build in hardware for GPUs with high processor count
  - Would force programmer to run fewer blocks (no more than # multiprocessors * # resident blocks / multiprocessor) to avoid deadlock, which may reduce overall efficiency
- Solution: decompose into multiple kernels
  - Kernel launch serves as a global synchronization point
  - Kernel launch has negligible HW overhead, low SW overhead

Solution: Kernel Decomposition

- **Avoid global sync by decomposing computation into multiple kernel invocations**



- **In the case of reductions, code for all levels is the same**
  - Recursive kernel invocation

Performance for 4M element reduction

	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!

And More...

1. On Volta and newer (Ampere, ...),
   reduction in shared memory must use
   warp synchronization!

2. Last optimization step for parallel reduction:

Do not use shared memory for last 5 steps, but use

warp shuffle instructions

EXAMPLE: REDUCTION VIA SHARED MEMORY

__syncwarp

Re-converge threads and perform memory fence

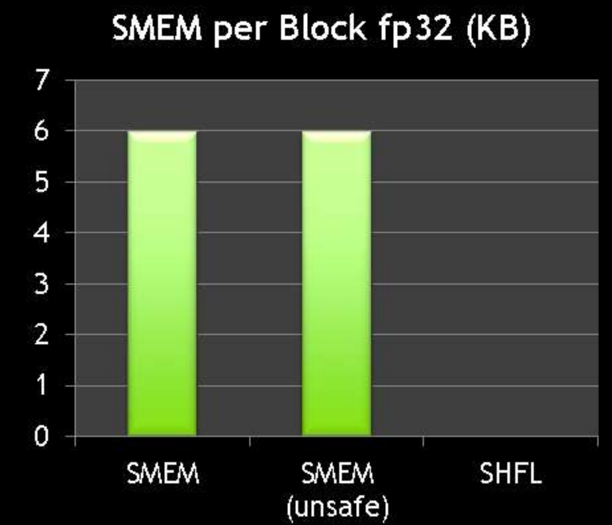
```
v += shmem[tid+16]; __syncwarp();
shmem[tid] = v;     __syncwarp();
v += shmem[tid+8];  __syncwarp();
shmem[tid] = v;     __syncwarp();
v += shmem[tid+4];  __syncwarp();
shmem[tid] = v;     __syncwarp();
v += shmem[tid+2];  __syncwarp();
shmem[tid] = v;     __syncwarp();
v += shmem[tid+1];  __syncwarp();
shmem[tid] = v;
```

Reduce

- Code

```
// Threads want to reduce the value in x.

float x = …;

#pragma unroll
for(int mask = WARP_SIZE / 2 ; mask > 0 ; mask >>= 1)
    x += __shfl_xor(x, mask);

// The x variable of laneid 0 contains the reduction.
```

- Performance
  - Launch 26 blocks of 1024 threads
  - Run the reduction 4096 times



Thank you.

- Hendrik Lensch, Robert Strzodka
- Mark Harris
- NVIDIA