

CS 380 - GPU and GPGPU Programming

Lecture 21: GPU Parallel Reduction

Markus Hadwiger, KAUST

Reading Assignment #8 (until Oct 28)



Read (required):

- Programming Massively Parallel Processors book, 4th edition
Chapter 10: Reduction
- Optimizing Parallel Reduction in CUDA, Mark Harris,

<https://developer.download.nvidia.com/assets/cuda/files/reduction.pdf>

Read (optional):

- Faster Parallel Reductions on Kepler, Justin Luitjens

<https://devblogs.nvidia.com/parallelforall/faster-parallel-reductions-kepler/>



Next Lectures

Lecture 22: Mon, Oct 28

Lecture 23: Tue, Oct 29 (make-up lecture; 14:30 – 15:45)

Lecture 24: Thu, Oct 31

Lecture 25: Mon, Nov 4

Lecture 26: Tue, Nov 5 (make-up lecture; 14:30 – 15:45)

No lecture on Thu, Nov 7 !

GPU Reduction

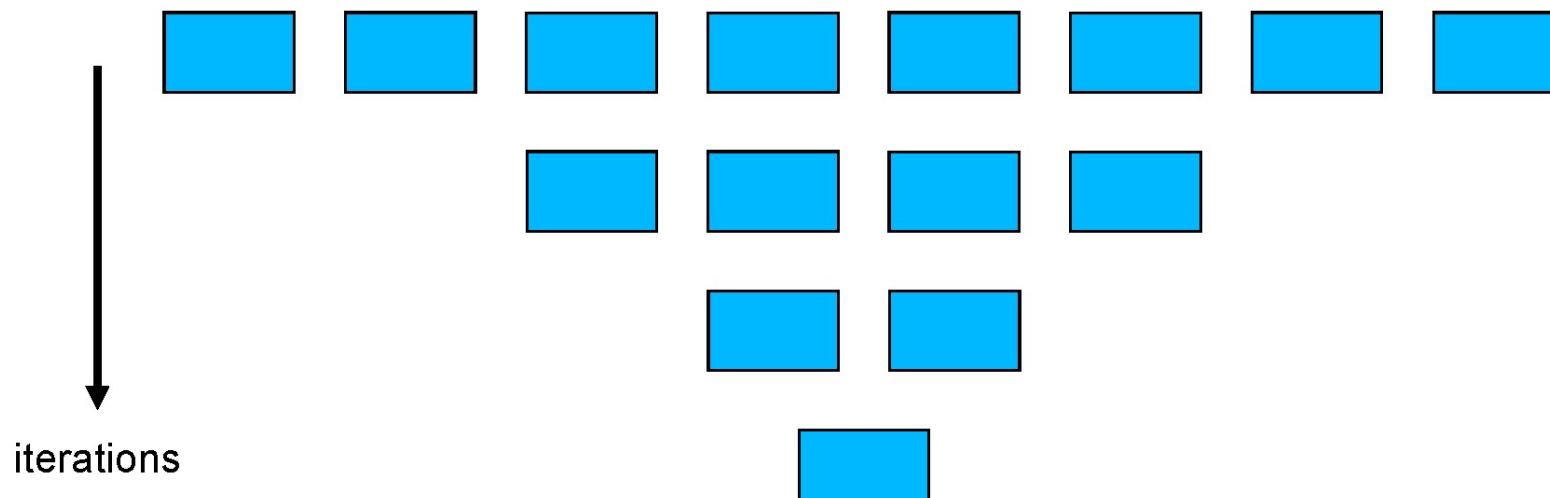
- Parallel reduction is a basic parallel programming primitive; see reduction operation on a stream, e.g., in Brook for GPUs

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



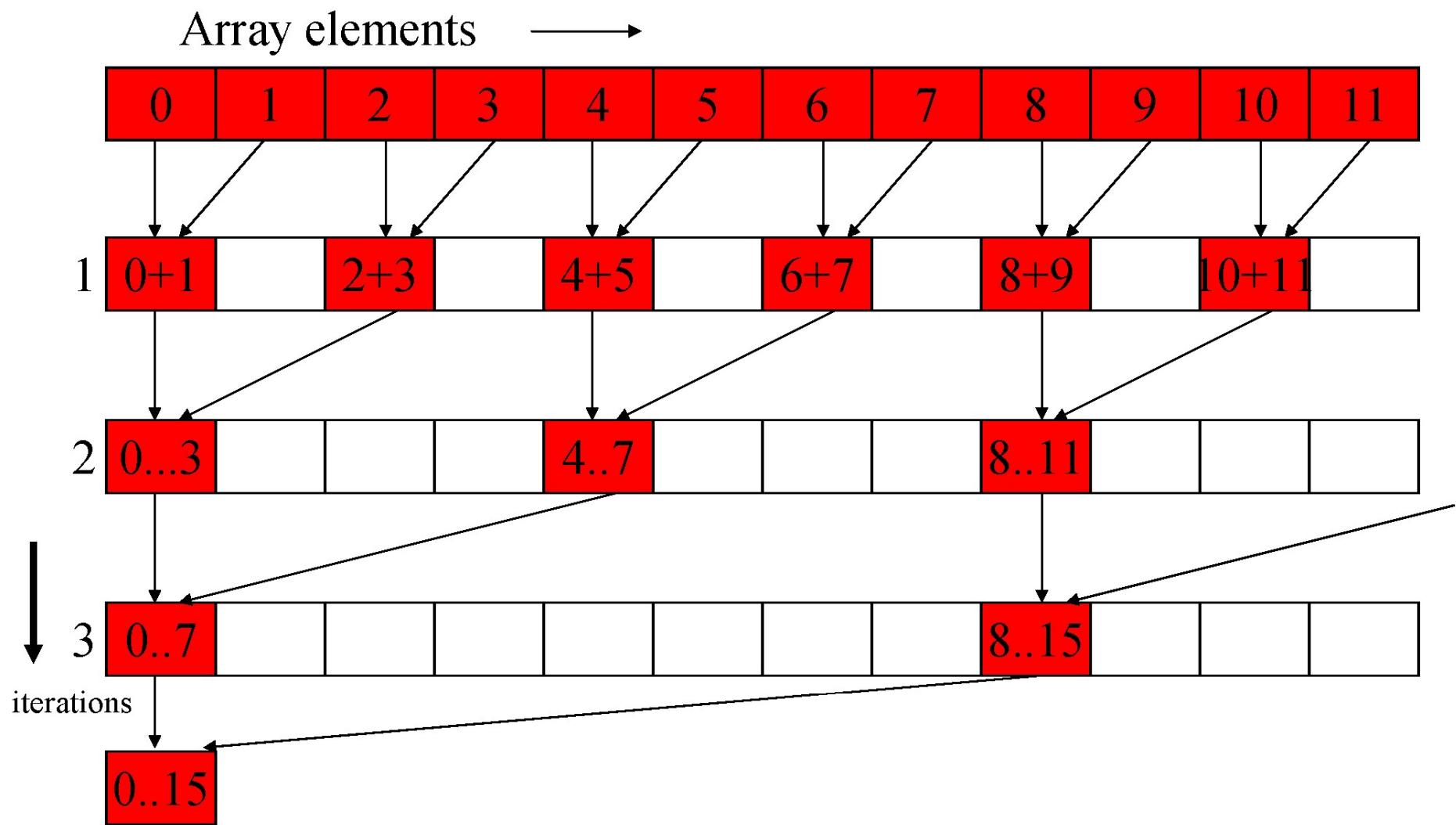
Helpful fact for counting nodes of full binary trees:
If there are N leaf nodes, there will be N-1 non-leaf nodes

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

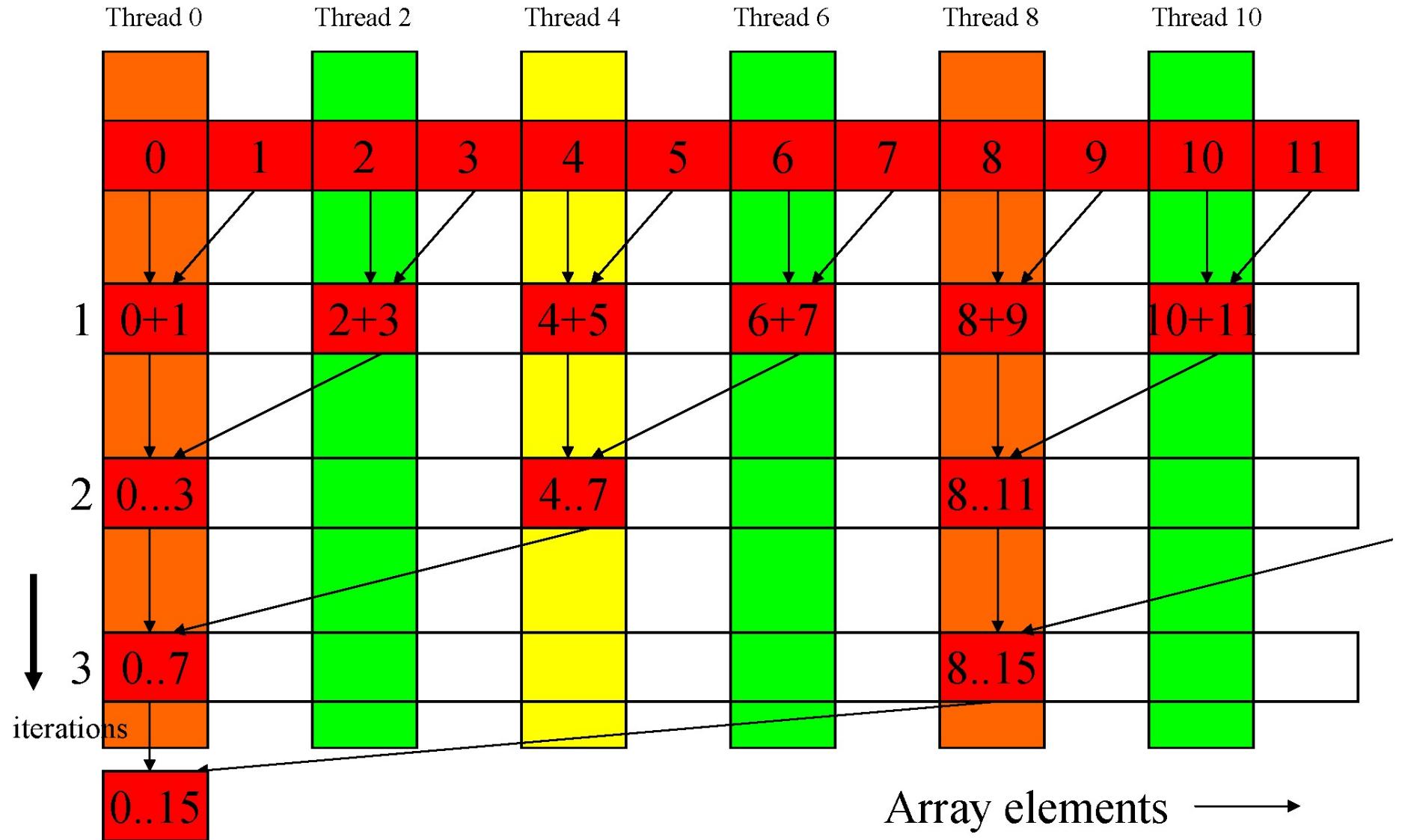
```
__global__ void reduce0(int *g_idata, int *g_odata) {
extern __shared__ int sdata[];

// 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 $\frac{1}{4}$ of the threads will be activated for all warps over time.
 - After the 5th 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

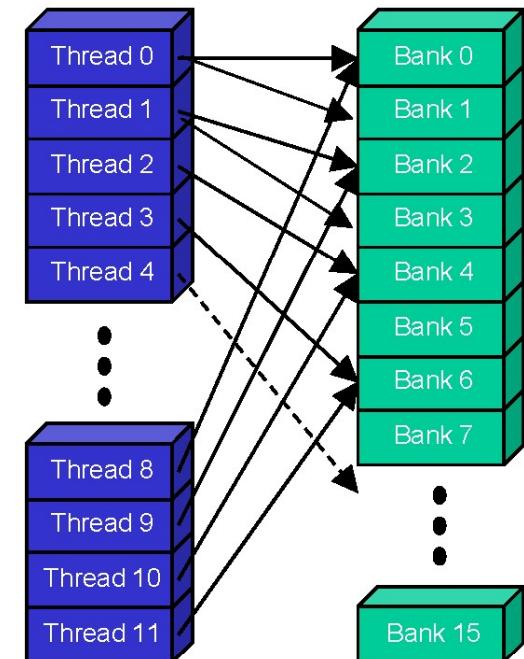
1D

- **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];
```

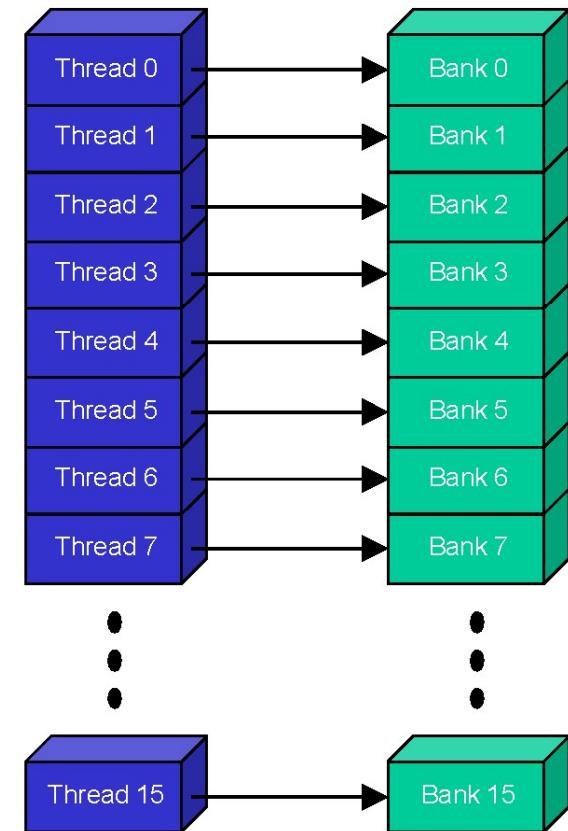
- **This makes sense for traditional CPU threads, locality in cache line usage and reduced sharing traffic.**
 - Not in shared memory usage where there is no cache line effects but banking effects



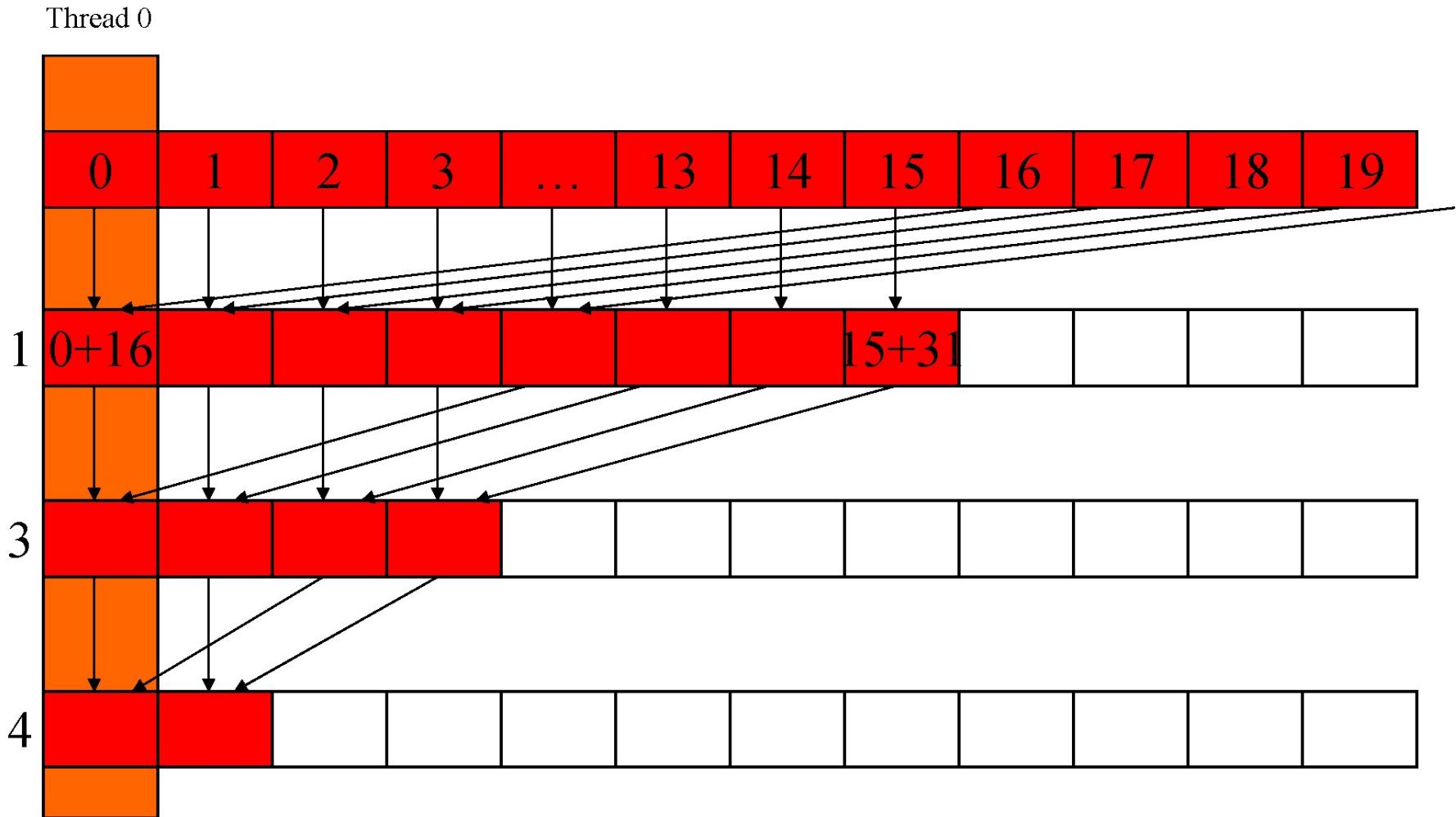
A Better Array Access Pattern

- **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[];
```

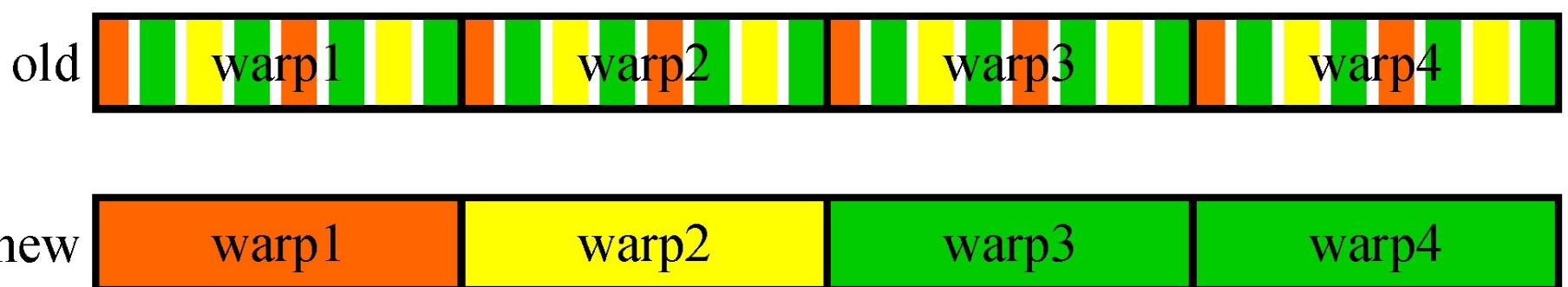
```
unsigned int t = threadIdx.x;
for (unsigned int stride = blockDim.x;
     stride > 1;  stride >>=1)
{
    __syncthreads();
    if (t < stride)
        partialSum[t] += partialSum[t+stride];
}
```

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

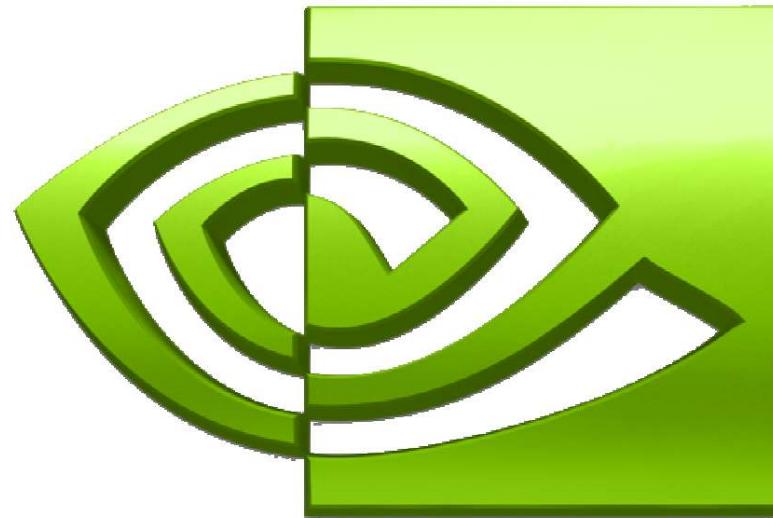
```
unsigned int tid = threadIdx.x;
for (unsigned int d = n>>1; d <= 32; d += 1) {
    __syncthreads();
    if (tid < d)
        shared[tid] += shared[tid + d];
    __syncthreads();
    if (tid <= 32) { // unroll last 5 iterations
        shared[tid] += shared[tid + 1];
        shared[tid] += shared[tid + 2];
        shared[tid] += shared[tid + 3];
        shared[tid] += shared[tid + 4];
        shared[tid] += shared[tid + 5];
        shared[tid] += shared[tid + 6];
    }
}
```

This would not work properly
is warp size decreases; need
`__syncthreads()` between each
statement!

However, having
`__syncthreads()` in if
statement is problematic.

now: `__syncwarp()`
or better: Cooperative Groups

Look at CUDA SDK reduction example and slides!



nVIDIA®

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

```

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();                                out-of-bounds check missing, see SDK code

    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) {be careful that shared variables are declared volatile! see SDK code
        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];
}

```



Final Optimized Kernel



```
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

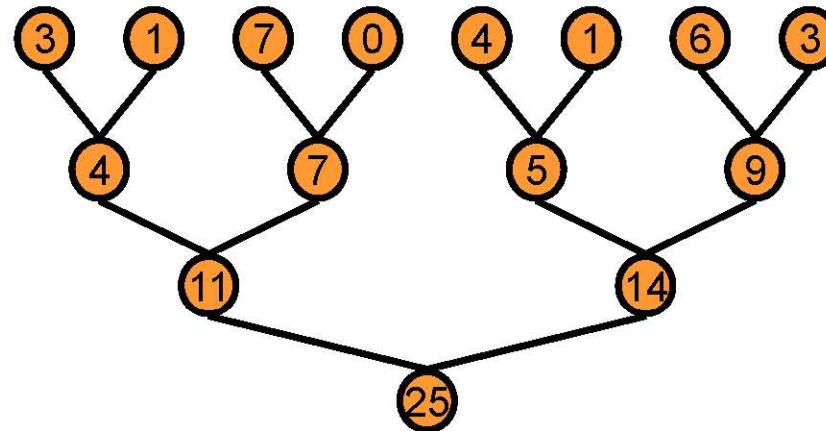
- Don't we still need block size at compile time?

- 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



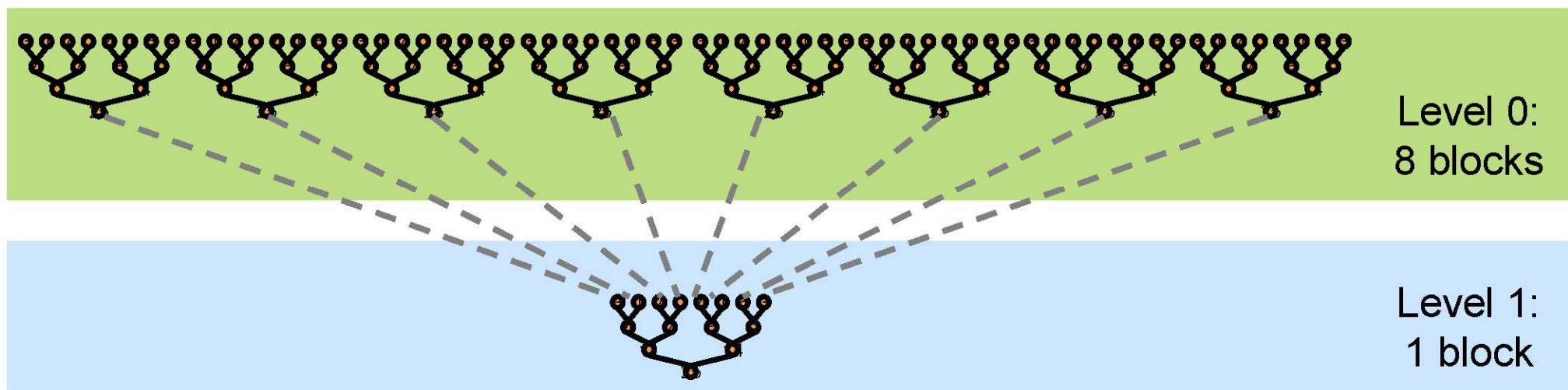
- Need to be able to use multiple thread blocks
 - To process very large arrays
 - To keep all multiprocessors on the GPU busy
 - Each thread block reduces a portion of the array
- But how do we communicate partial results between thread blocks?

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! `__syncwarp()` or Cooperative Groups

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

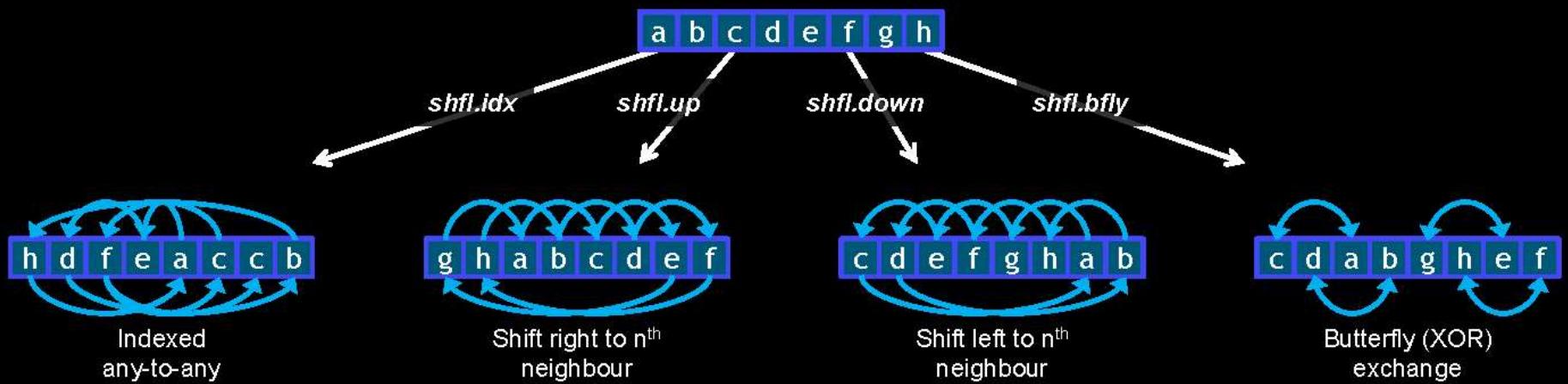
```
v += shmem[tid+16]; __syncwarp();
shmem[tid] = v;    __syncwarp();
v += shmem[tid+8]; __syncwarp();
shmem[tid] = v;    __syncwarp();
v += shmem[tid+4]; __syncwarp();
shmem[tid] = v;    __syncwarp();
v += shmem[tid+2]; __syncwarp();
shmem[tid] = v;    __syncwarp();
v += shmem[tid+1]; __syncwarp();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

Shuffle (SHFL)

- Instruction to exchange data in a warp
- Threads can “read” other threads’ registers
- No shared memory is needed
- It is available starting from SM 3.0

Variants

- 4 variants (idx, up, down, bfly):



Now: Use `_sync` variants / shuffle in cooperative thread groups!

Instruction (PTX)

Optional dst. predicate Lane/offset/mask
shfl.mode.b32 d[|p], a, b, c;
 ↑
 Dst. register Src. register Bound

Now: Use _sync variants / shuffle in cooperative thread groups!

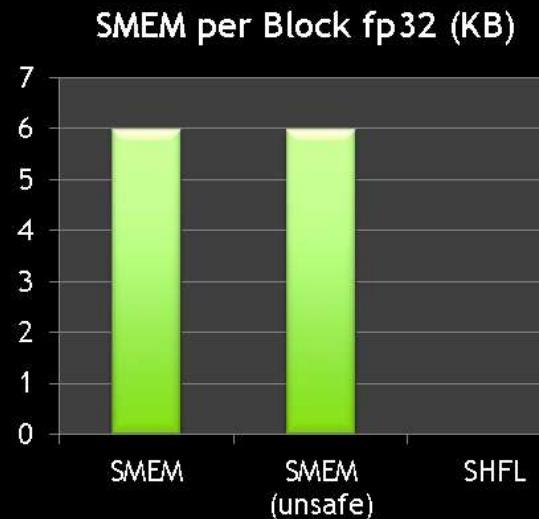
Reduce

■ 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





GPU TECHNOLOGY
CONFERENCE

Shuffle: Tips and Tricks

Julien Demouth, NVIDIA

Glossary

Safer with cooperative thread groups!

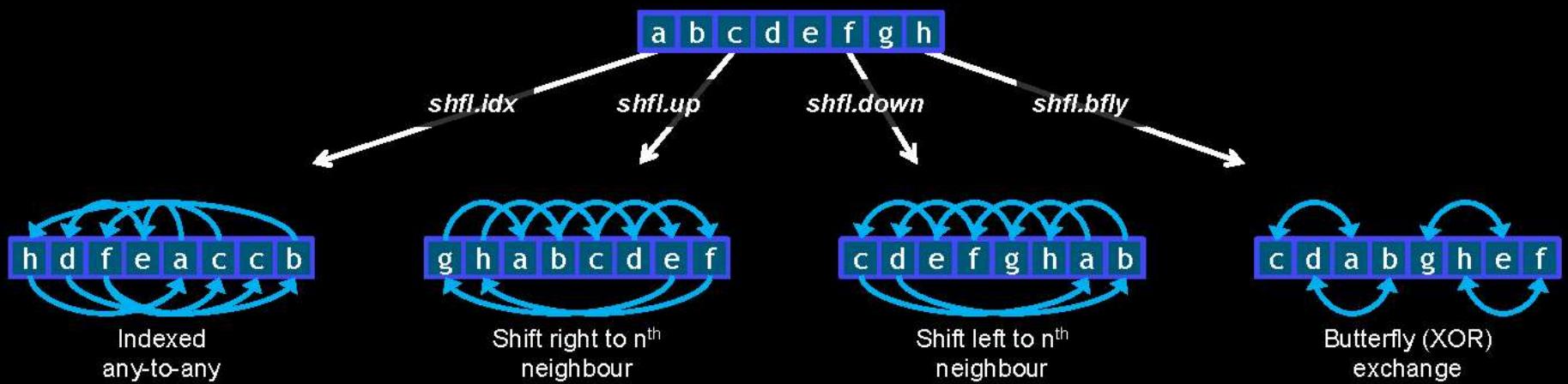
- Warp
 - ~~Implicitly synchronized~~ group of threads (32 on current HW)
- Warp ID (warpid)
 - Identifier of the warp in a block: `threadIdx.x / 32`
- Lane ID (laneid)
 - Coordinate of the thread in a warp: `threadIdx.x % 32`
 - Special register (available from PTX): `%laneid`

Shuffle (SHFL)

- Instruction to exchange data in a warp
- Threads can “read” other threads’ registers
- No shared memory is needed
- It is available starting from SM 3.0

Variants

- 4 variants (idx, up, down, bfly):



Now: Use `_sync` variants / shuffle in cooperative thread groups!

Instruction (PTX)

Optional dst. predicate Lane/offset/mask
shfl.mode.b32 d[|p], a, b, c;
 ↑
 Dst. register Src. register Bound

Now: Use _sync variants / shuffle in cooperative thread groups!

Implement SHFL for 64b Numbers

```
__device__ __inline__ double shfl(double x, int lane)
{
    // split the double number into 2 32b registers.
    int lo, hi;
    asm volatile( "mov.b32 {%0,%1}, %2;" : "=r"(lo), "=r"(hi) : "d"(x));

    // Shuffle the two 32b registers.
    lo = __shfl(lo, lane);
    hi = __shfl(hi, lane);

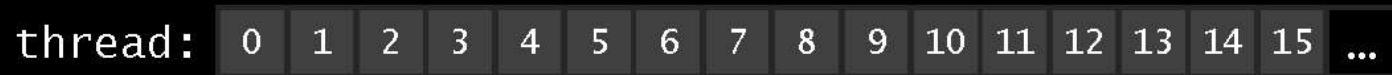
    // Recreate the 64b number.
    asm volatile( "mov.b64 %0, {%1,%2};" : "=d(x)" : "r"(lo), "r"(hi));

    return x;
}
```

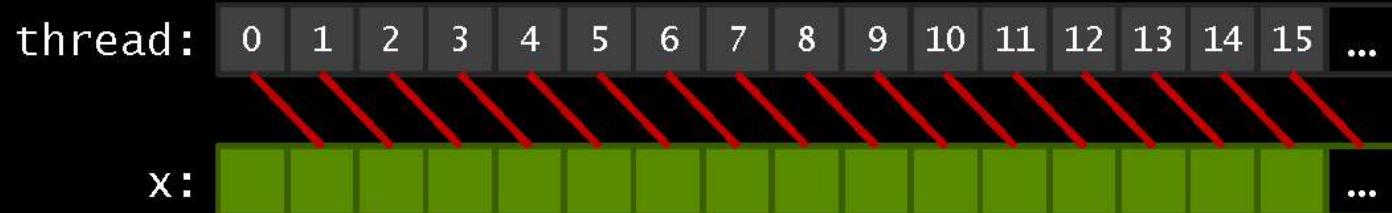
- Generic SHFL: <https://github.com/BryanCatanzaro/generics>

Performance Experiment

- One element per thread



- Each thread takes its right neighbor



Performance Experiment

- We run the following test on a K20

```
T x = input[tidx];
for(int i = 0 ; i < 4096 ; ++i)
    x = get_right_neighbor(x);
output[tidx] = x;
```

- We launch 26 blocks of 1024 threads
 - On K20, we have 13 SMs
 - We need 2048 threads per SM to have 100% of occupancy
- We time different variants of that kernel

Performance Experiment

- Shared memory (SMEM)

```
smem[threadIdx.x] = smem[32*warpid + ((laneid+1) % 32)];  
__syncthreads();
```

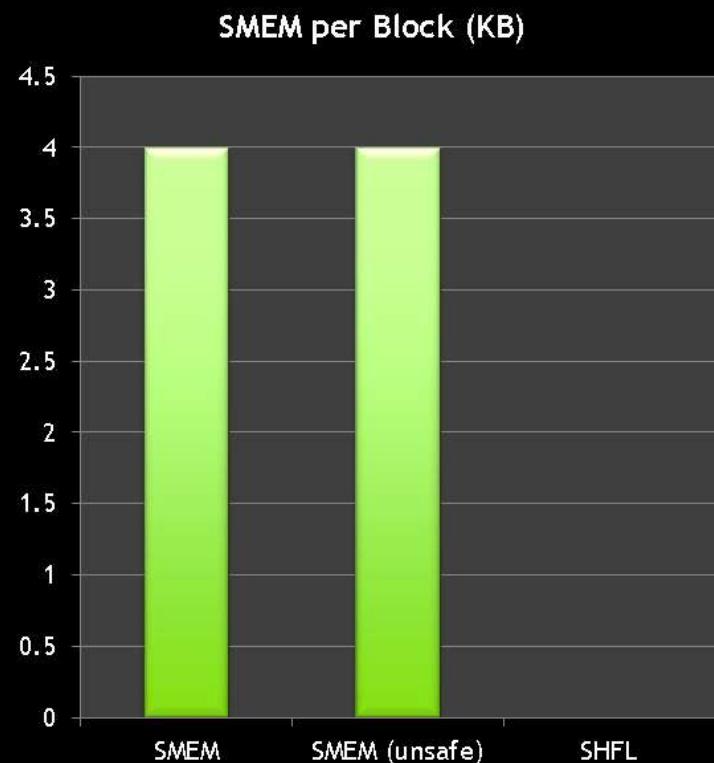
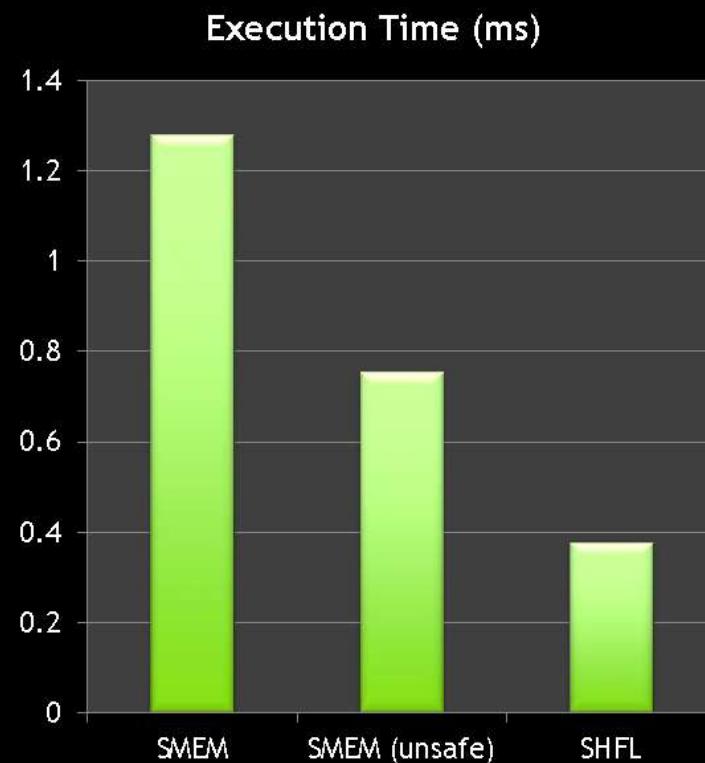
- Shuffle (SHFL)

```
x = __shfl(x, (laneid+1) % 32);
```

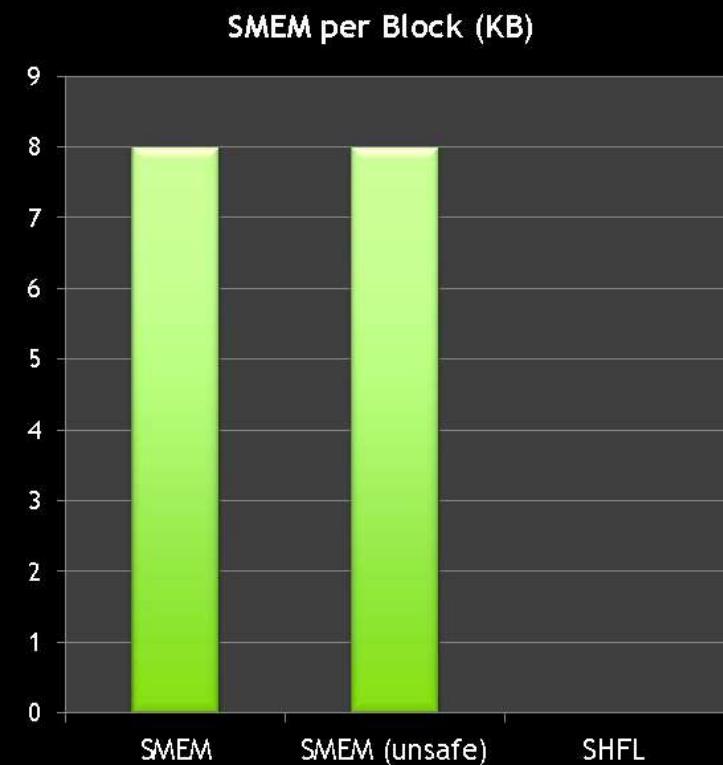
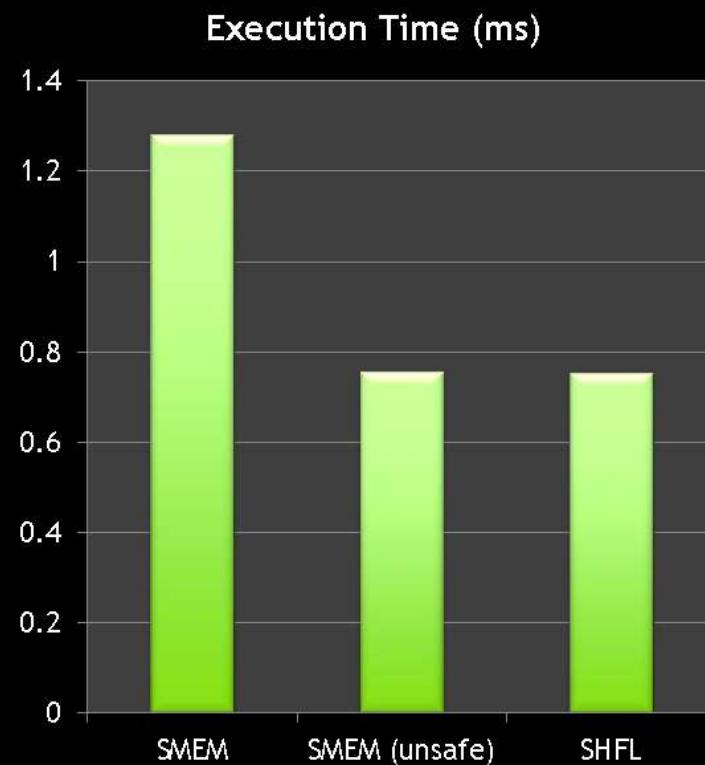
- Shared memory without `__syncthreads` + volatile (*unsafe*)

```
__shared__ volatile T *smem = ...;  
smem[threadIdx.x] = smem[32*warpid + ((laneid+1) % 32)];
```

Performance Experiment (fp32)



Performance Experiment (fp64)



Performance Experiment

- Always faster than shared memory
- Much safer than using no `__syncthreads` (and volatile)
 - And never slower
- Does not require shared memory
 - Useful when occupancy is limited by SMEM usage

Broadcast

Now: Use cooperative thread groups!

- All threads read from a single lane

```
x = __shfl(x, 0); // All the threads read x from laneid 0.
```

- More complex example

```
// All threads evaluate a predicate.  
int predicate = ...;  
  
// All threads vote.  
unsigned vote = __ballot(predicate);  
  
// All threads get x from the "last" lane which evaluated the predicate to true.  
if(vote)  
    x = __shfl(x, __bfind(vote));  
  
// __bind(unsigned i): Find the most significant bit in a 32/64 number (PTX).  
__bfind(&b, i) { asm volatile("bfind.u32 %0, %1;" : "=r"(b) : "r"(i)); }
```

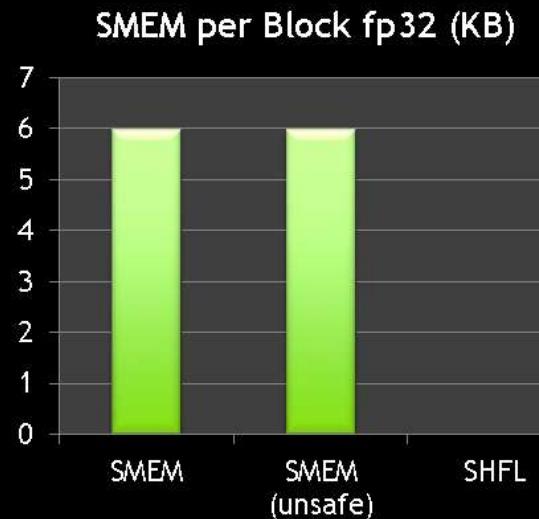
Reduce

■ 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



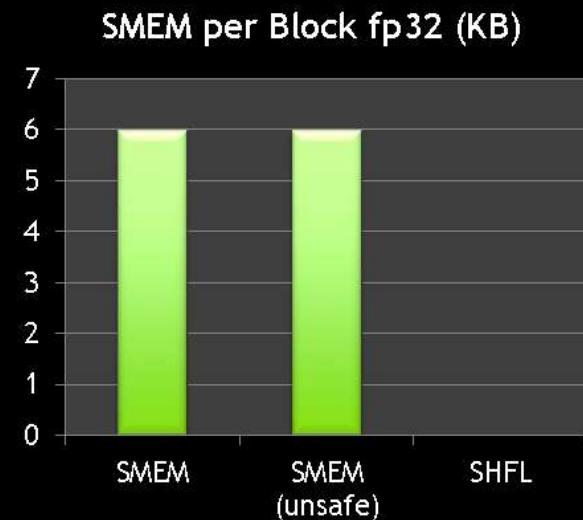
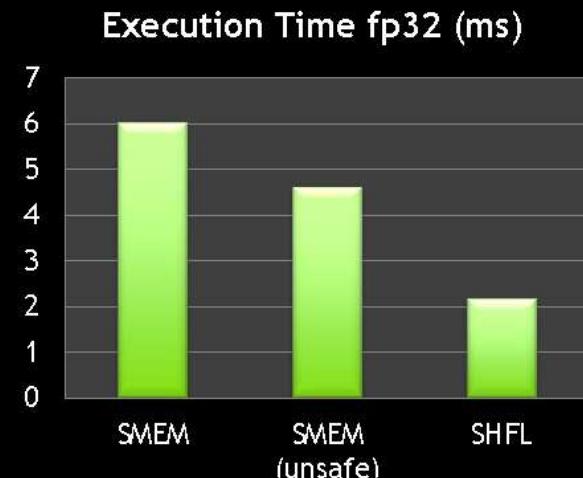
Scan

- Code

```
#pragma unroll
for( int offset = 1 ; offset < 32 ; offset <= 1 )
{
    float y = __shfl_up(x, offset);
    if(laneid() >= offset)
        x += y;
}
```

- Performance

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



Scan

- Use the predicate from SHFL

```
#pragma unroll
for( int offset = 1 ; offset < 32 ; offset <= 1 )
{
    asm volatile( "{"
        "    .reg .f32 r0;" 
        "    .reg .pred p;" 
        "    shfl.up.b32 r0 |p, %0, %1, 0x0;" 
        "    @p add.f32 r0, r0, %0;" 
        "    mov.f32 %0, r0;" 
        "}" : "+f"(x) : "r"(offset));
}
```

- Use CUB:
<https://nvlabs.github.com/cub>

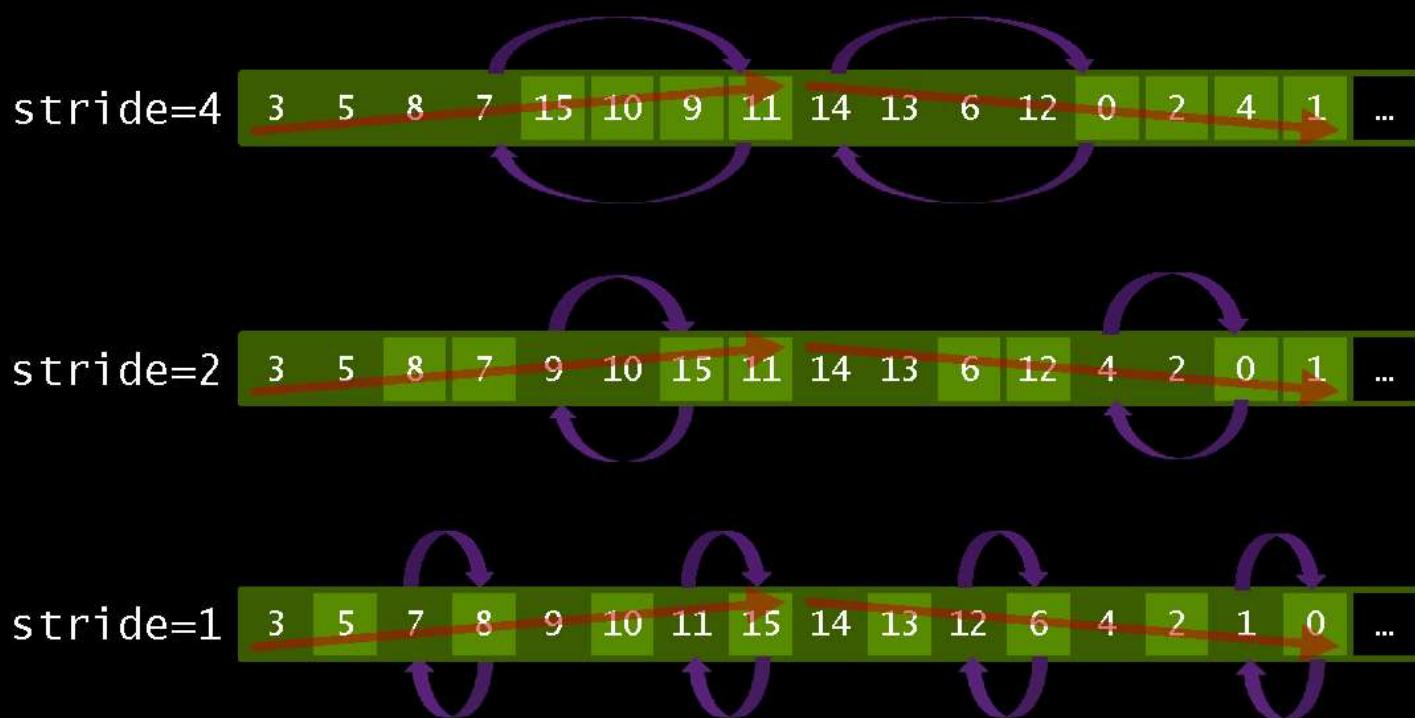


Bitonic Sort

x: 11 3 8 5 10 15 9 7 12 4 2 0 14 13 6 1 ...



Bitonic Sort



Bitonic Sort

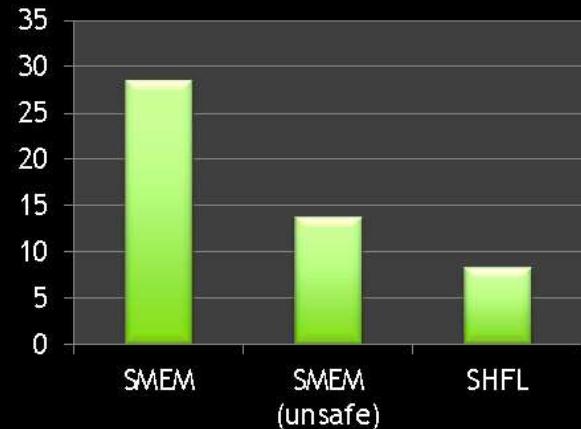
```
int swap(int x, int mask, int dir)
{
    int y = __shfl_xor(x, mask);
    return x < y == dir ? y : x;
}

x = swap(x, 0x01, bfe(laneid, 1) ^ bfe(laneid, 0)); // 2
x = swap(x, 0x02, bfe(laneid, 2) ^ bfe(laneid, 1)); // 4
x = swap(x, 0x01, bfe(laneid, 2) ^ bfe(laneid, 0));
x = swap(x, 0x04, bfe(laneid, 3) ^ bfe(laneid, 2)); // 8
x = swap(x, 0x02, bfe(laneid, 3) ^ bfe(laneid, 1));
x = swap(x, 0x01, bfe(laneid, 3) ^ bfe(laneid, 0));
x = swap(x, 0x08, bfe(laneid, 4) ^ bfe(laneid, 3)); // 16
x = swap(x, 0x04, bfe(laneid, 4) ^ bfe(laneid, 2));
x = swap(x, 0x02, bfe(laneid, 4) ^ bfe(laneid, 1));
x = swap(x, 0x01, bfe(laneid, 4) ^ bfe(laneid, 0));
x = swap(x, 0x10,
           bfe(laneid, 4)); // 32
x = swap(x, 0x08,
           bfe(laneid, 3));
x = swap(x, 0x04,
           bfe(laneid, 2));
x = swap(x, 0x02,
           bfe(laneid, 1));
x = swap(x, 0x01,
           bfe(laneid, 0));

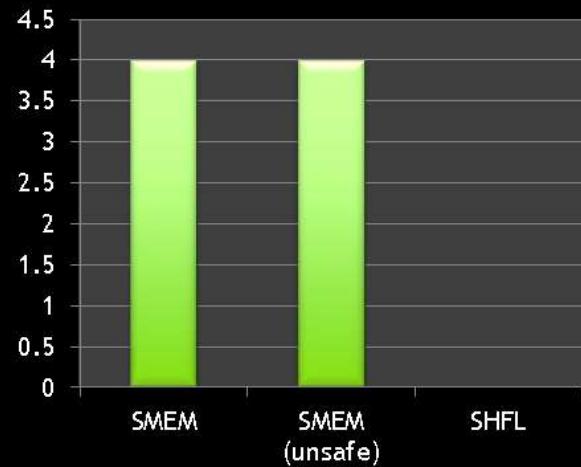
// int bfe(int i, int k): Extract k-th bit from i

// PTX: bfe dst, src, start, len (see p.81, ptx_isa_3.1)
```

Execution Time int32 (ms)

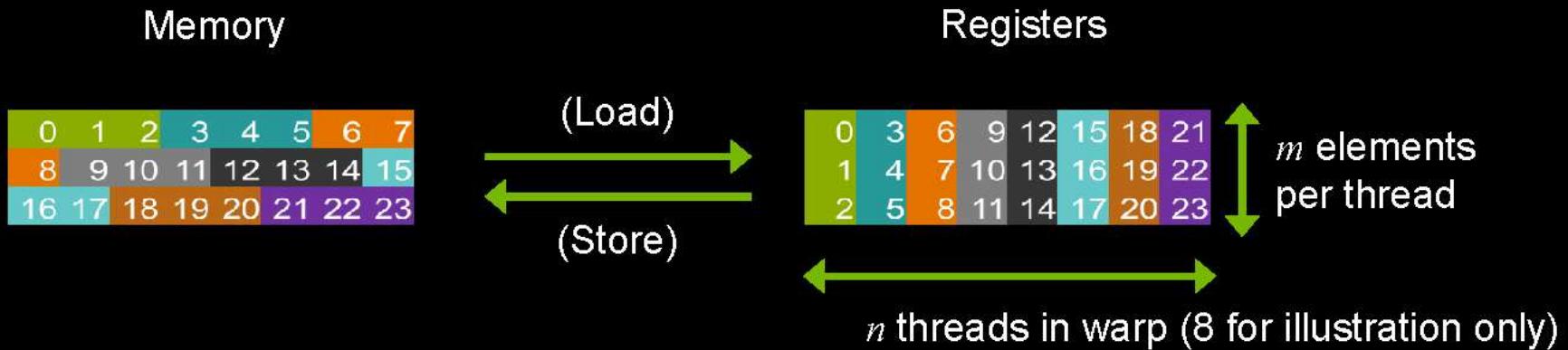


SMEM per Block (KB)



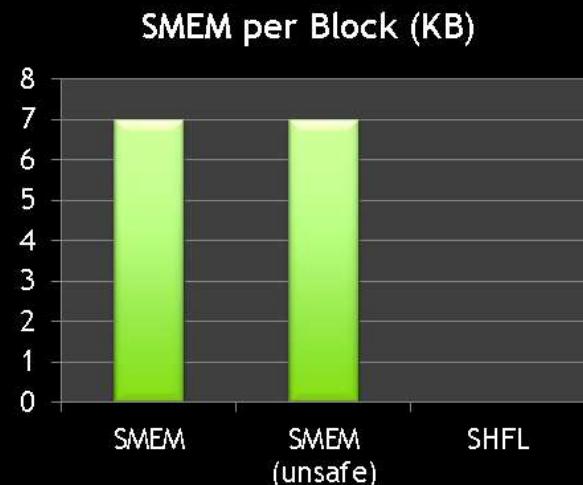
Transpose

- When threads load or store arrays of structures, transposes enable fully coalesced memory operations
- e.g. when loading, have the warp perform coalesced loads, then transpose to send the data to the appropriate thread



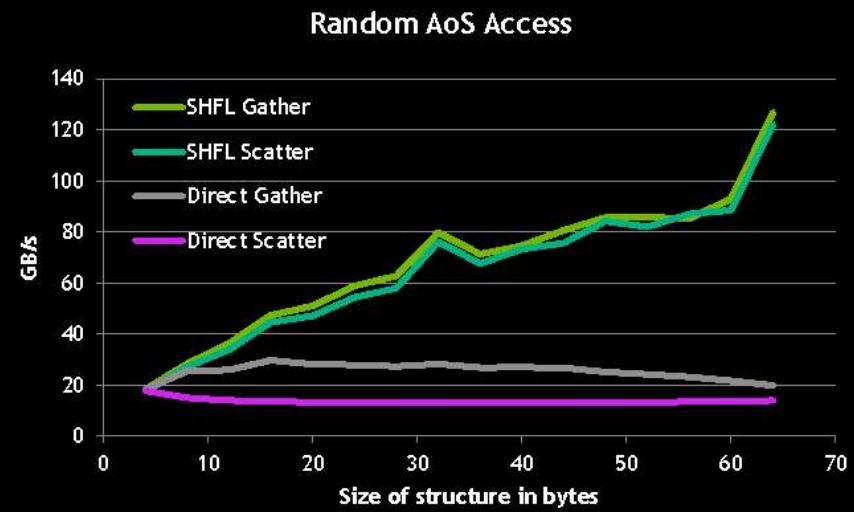
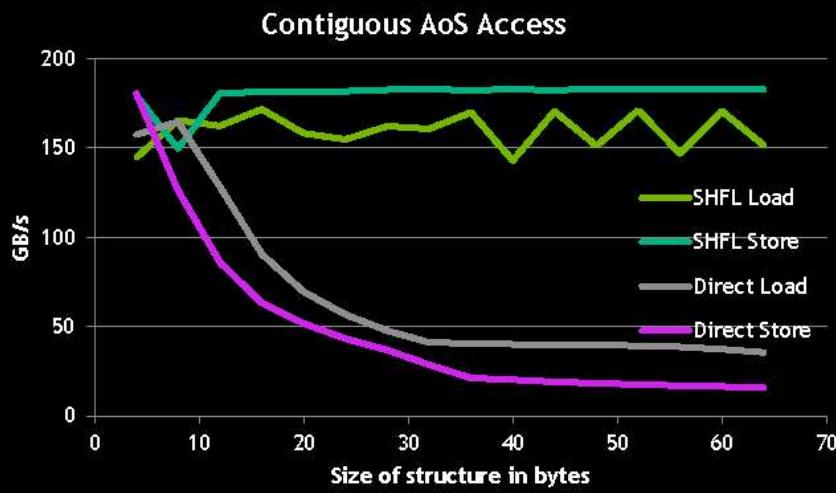
Transpose

- You can use SMEM to implement this transpose, or you can use SHFL
- Code:
<http://github.com/bryancatanzaro/trove>
- Performance
 - Launch 104 blocks of 256 threads
 - Run the transpose 4096 times



Array of Structures Access via Transpose

- Transpose speeds access to arrays of structures
- High-level interface: `coalesced_ptr<T>`
 - Just dereference like any pointer
 - Up to 6x faster than direct compiler generated access



Conclusion

- SHFL is available for SM \geq SM 3.0
- It is always faster than “safe” shared memory
- It is never slower than “unsafe” shared memory
- It can be used in many different algorithms

Thank you.