King Abdullah University of Science and Technology KAUST

CS 380 - GPU and GPGPU Programming
Lecture 25: Parallel Scan Bank Conflicts; Shuffle Instructions

Markus Hadwiger, KAUST

Reading Assignment #14 (until Dec 7)

Read (required):

- Warp Shuffle Functions
  - CUDA Programming Guide 11.1, Appendix B.21
- CUDA Cooperative Groups (Volta + Turing)
  - **https://devblogs.nvidia.com/cooperative-groups/**
  - CUDA Programming Guide 11.1, Appendix C
- Programming Tensor Cores
  - **https://devblogs.nvidia.com/programming-tensor-cores-cuda-9/**
  - CUDA Programming Guide 11.1, Appendix B.23

Read (optional):

- CUDA Warp-Level Primitives
  - **https://developer.nvidia.com/blog/using-cuda-warp-level-primitives/**
- Warp-aggregated atomics
  - **https://devblogs.nvidia.com/cuda-pro-tip-optimized-filtering-warp-aggregated-atomics/**

Quiz #4: Dec 9

Organization

- First 30 min of lecture
- No material (book, notes, ...) allowed

Content of questions

- Lectures (both actual lectures and slides)
- Reading assigments
- Programming assignments (algorithms, methods)
- Solve short practical examples

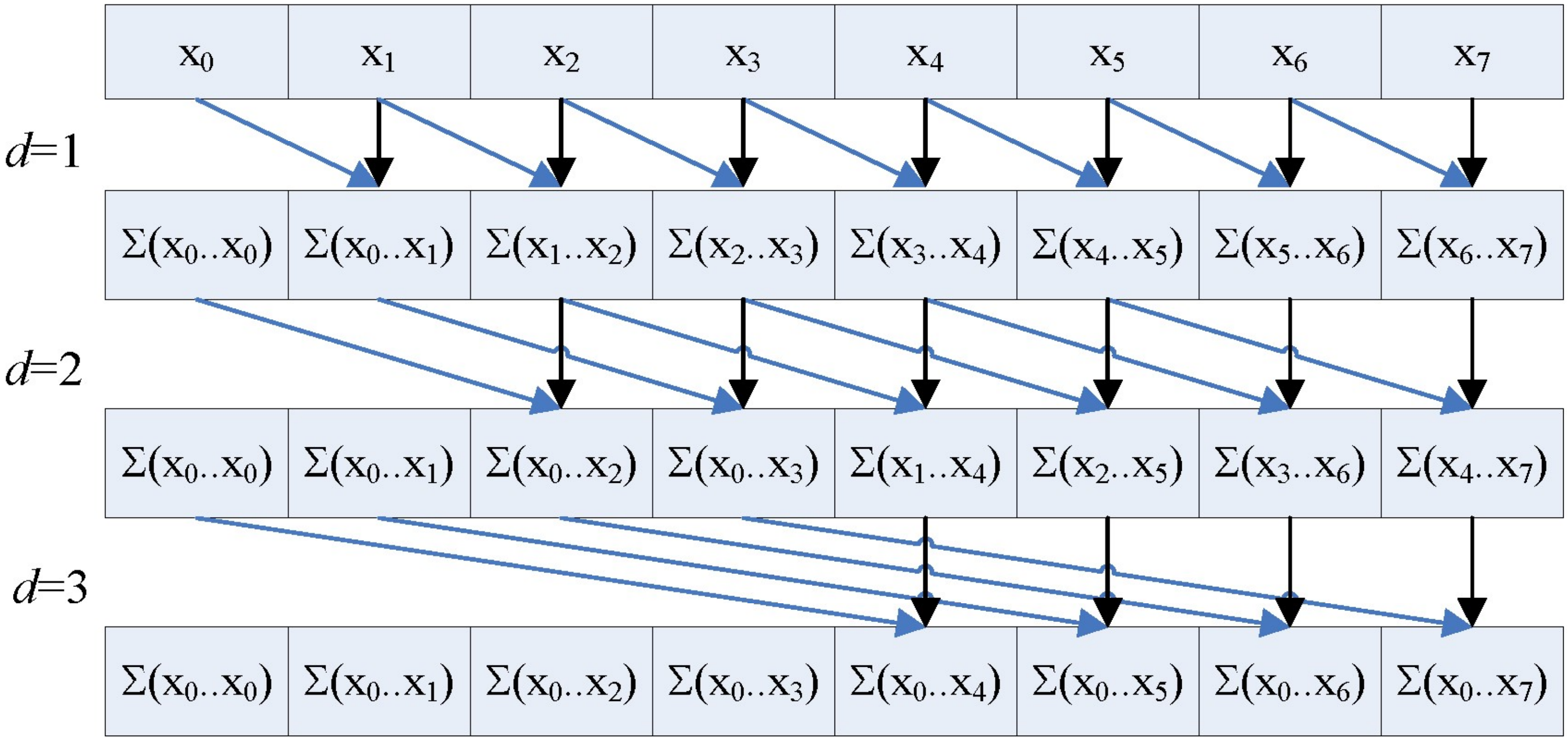
Semester Project Presentation Event(s)

Sunday/Monday, Dec 13/14

tbd

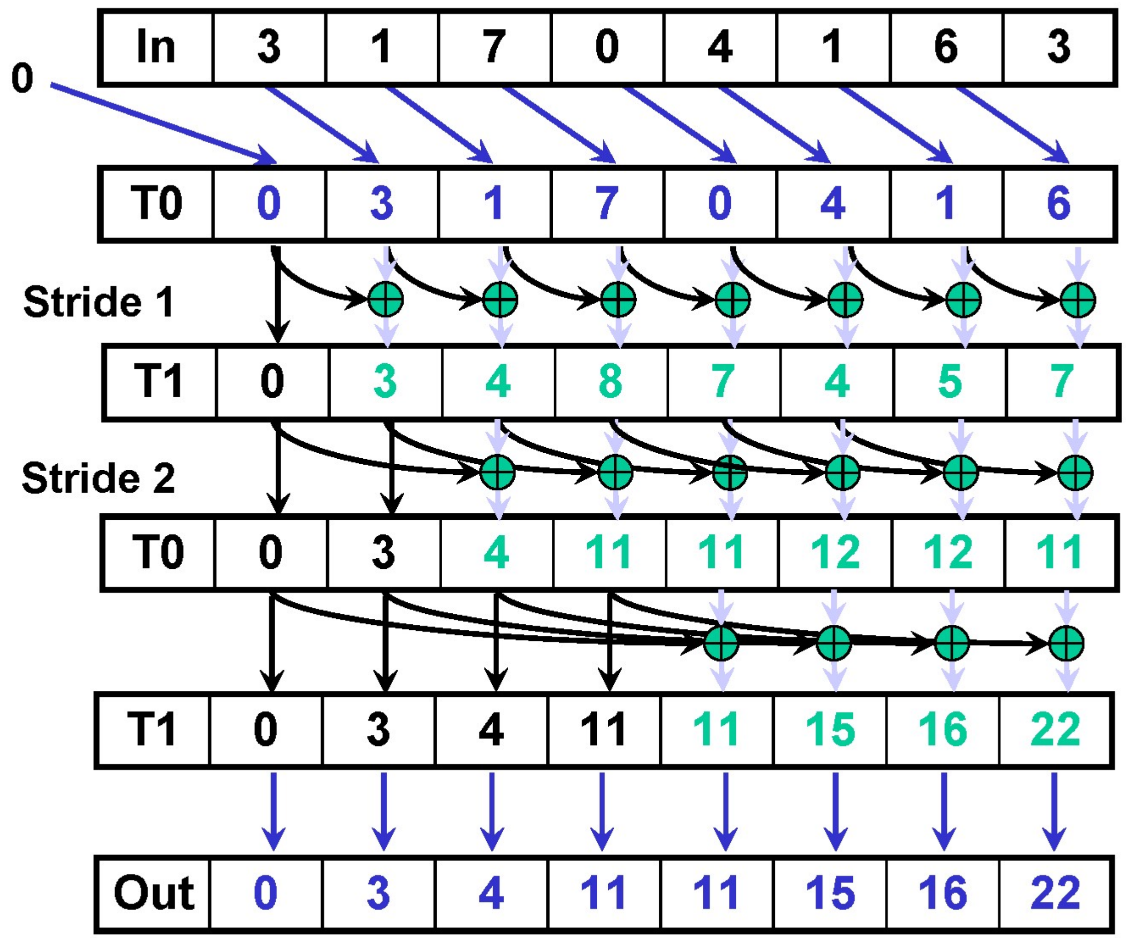
$O(n \log n)$ Scan

Courtesy John Owens



- Step efficient ($\log n$ steps)
- Not work efficient ($n \log n$ work)
- Requires barriers at each step (WAR dependencies)

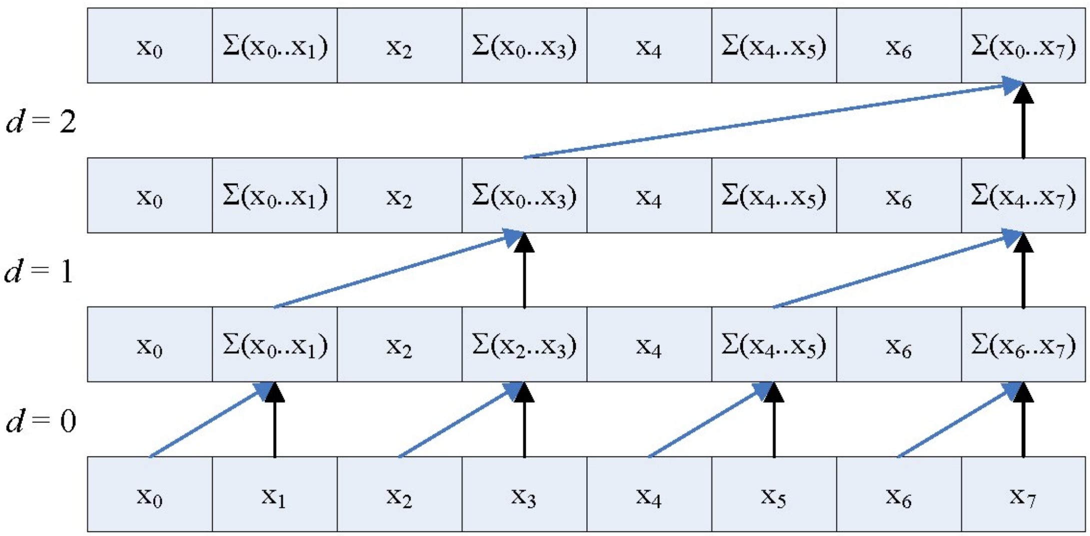
A First-Attempt Parallel Scan Algorithm

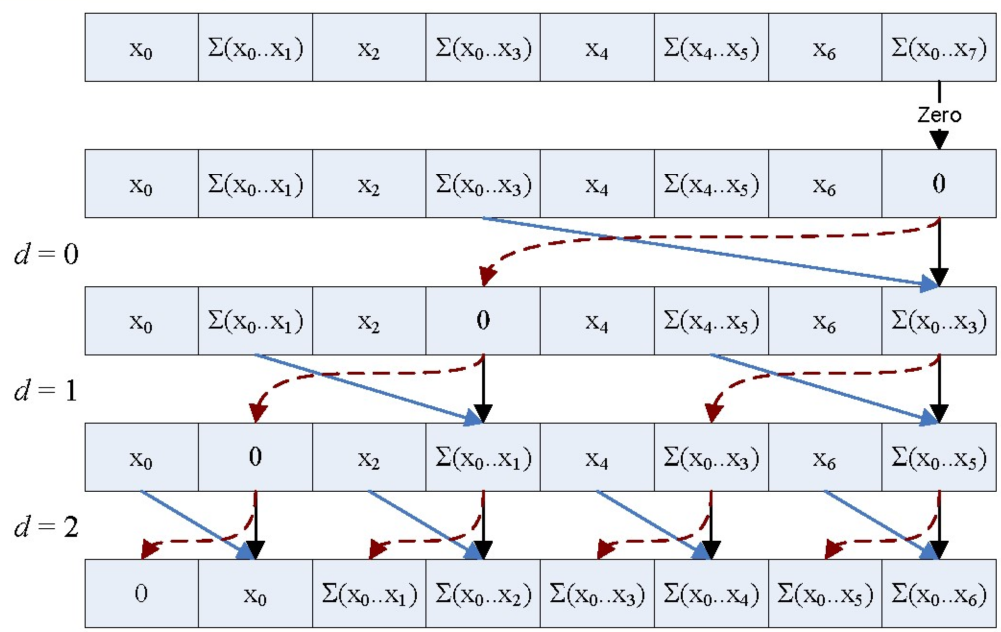


1. Read input from device memory to shared memory. Set first element to zero and shift others right by one.
2. Iterate log(n) times: Threads *stride* to *n:* Add pairs of elements *stride* elements apart. Double *stride* at each iteration. (note must double buffer shared mem arrays)
3. Write output to device memory.

$O(n)$ Scan [Blelloch]

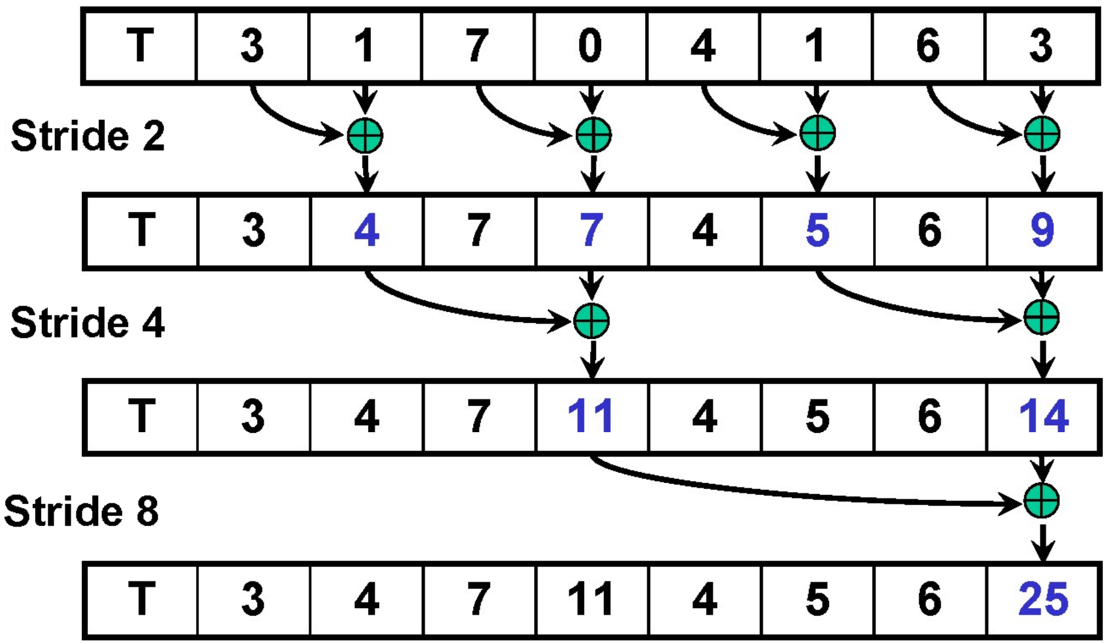
Courtesy John Owens





- Work efficient ($O(n)$ work)
- Bank conflicts, and lots of 'em

Build the Sum Tree



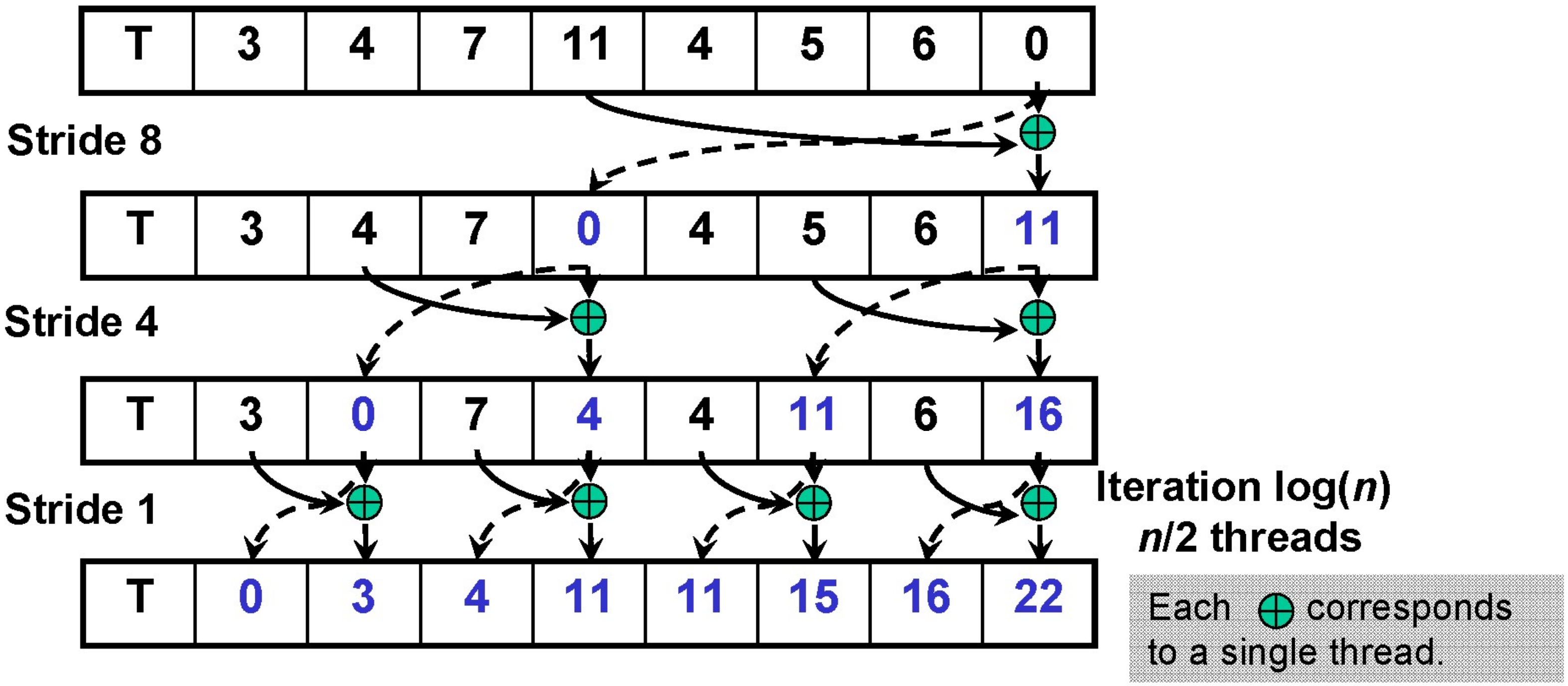
Iteration log(*n*), 1 thread

Each ⊕ corresponds to a single thread.

Iterate log(n) times. Each thread adds value *stride* / 2 elements away to its own value.

Note that this algorithm operates in-place: no need for double buffering

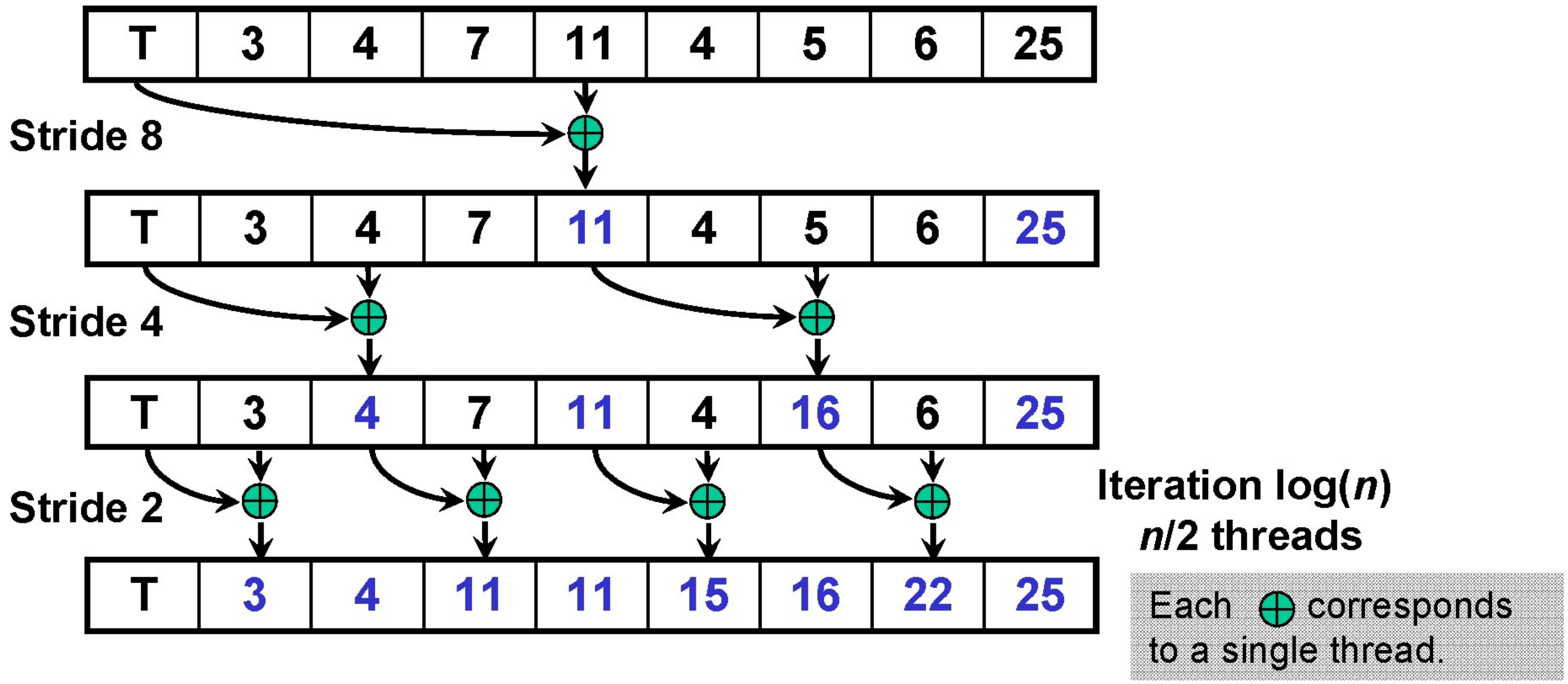
Build Scan From Partial Sums



Done! We now have a completed scan that we can write out to device memory.

Total steps: 2 * log(*n*).
Total work: 2 * (*n*-1) adds = *O*(*n*) **Work Efficient!**

Build Scan From Partial Sums



Done! We now have a completed scan that we can write out to device memory.

Total steps: 2 * log(*n*).
Total work: < 2 * (*n*-1) adds = *O*(*n*) **Work Efficient!**

Bank Conflicts in Scan
- Non-power-of-two -

Initial Bank Conflicts on Load

- **Each thread loads two shared mem data elements**
- **Tempting to interleave the loads**

```
temp[2*thid]   = g_idata[2*thid];
temp[2*thid+1] = g_idata[2*thid+1];
```

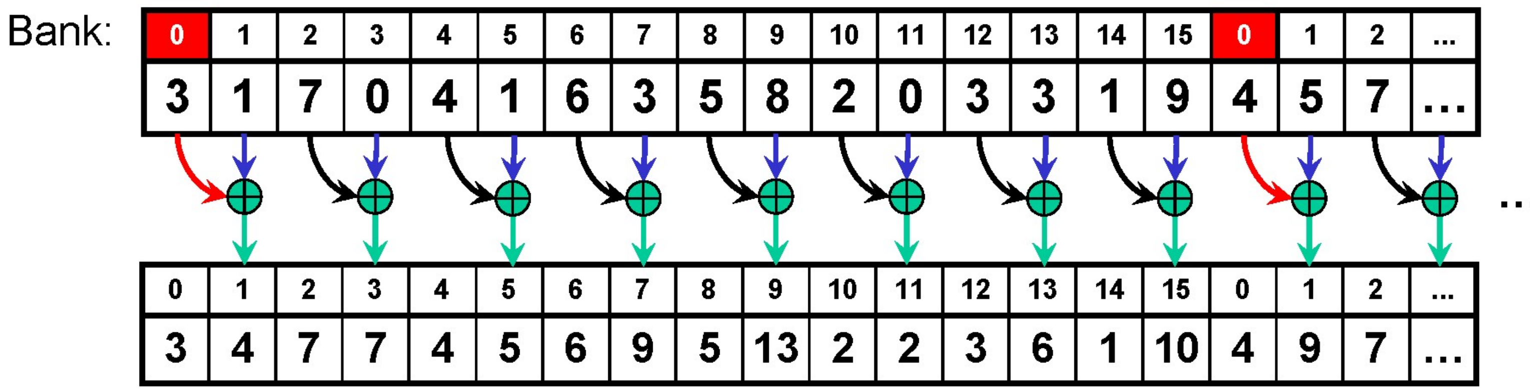
- **Threads:(0,1,2,…,8,9,10,…)→banks:(0,2,4,…,0,2,4,…)**

- **Better to load one element from each half of the array**

```
temp[thid]         = g_idata[thid];
temp[thid + (n/2)] = g_idata[thid + (n/2)];
```

Bank Conflicts in the tree algorithm

- **When we build the sums, each thread reads two shared memory locations and writes one:**
- **Th(0,8) access bank 0**



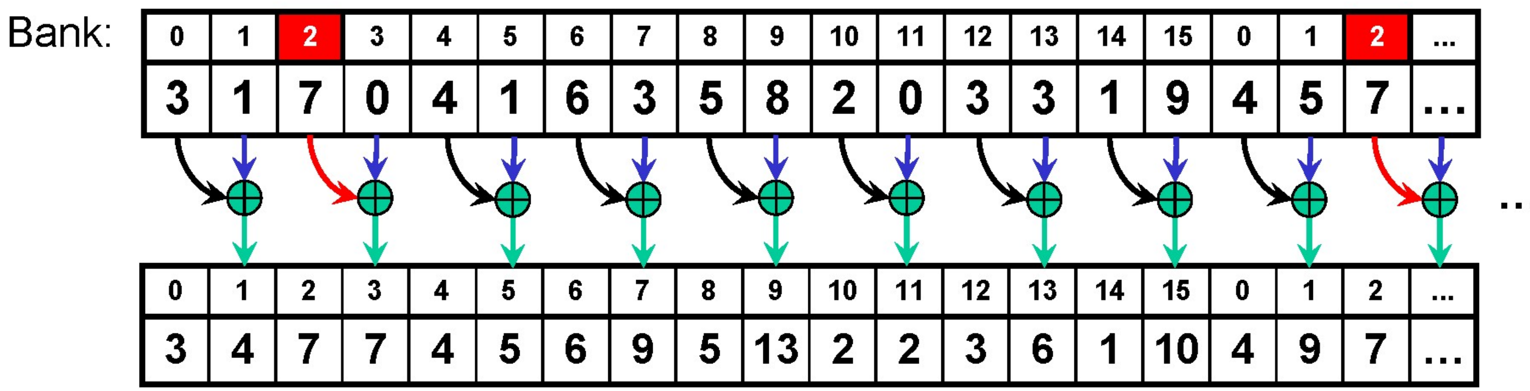
First iteration: 2 threads access each of 8 banks.

Each ⊕ corresponds to a single thread.

Like-colored arrows represent simultaneous memory accesses

Bank Conflicts in the tree algorithm

- **When we build the sums, each thread reads two shared memory locations and writes one:**
- **Th(1,9) access bank 2, etc.**



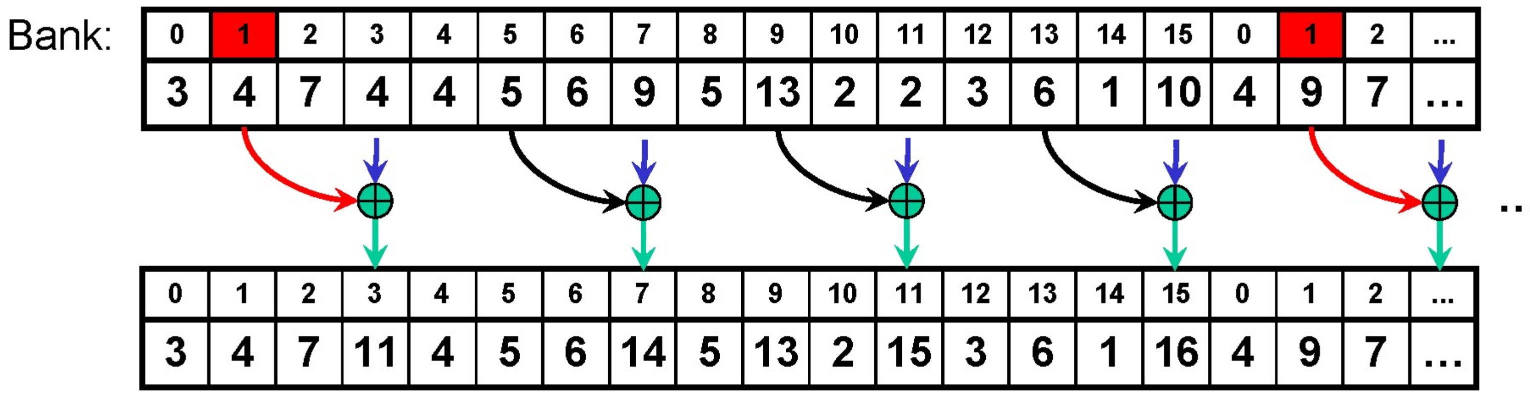
First iteration: 2 threads access each of 8 banks.

Each ⊕ corresponds to a single thread.

Like-colored arrows represent simultaneous memory accesses

Bank Conflicts in the tree algorithm

- **2^{nd} iteration: even worse!**
  - 4-way bank conflicts; for example:
    Th(0,4,8,12) access bank 1, Th(1,5,9,13) access Bank 5, etc.



2^{nd} iteration: 4 threads access each of 4 banks.

Each ⊕ corresponds to a single thread.

Like-colored arrows represent simultaneous memory accesses

Scan Bank Conflicts (1)

- **A full binary tree with 64 leaf nodes:**

Scale (s)	Thread addresses																															
1	0	2	4	6	8	10	12	14	16	18	20	22	24	26	28	30	32	34	36	38	40	42	44	46	48	50	52	54	56	58	60	62
2	0	4	8	12	16	20	24	28	32	36	40	44	48	52	56	60																
4	0	8	16	24	32	40	48	56																								
8	0	16	32	48																												
16	0	32																														
32	0																															
Conflicts	Banks																															
2-way	0	2	4	6	8	10	12	14	0	2	4	6	8	10	12	14	0	2	4	6	8	10	12	14	0	2	4	6	8	10	12	14
4-way	0	4	8	12	0	4	8	12	0	4	8	12	0	4	8	12																
4-way	0	8	0	8	0	8	0	8																								
4-way	0	0	0	0																												
2-way	0	0																														
None	0																															

- **Multiple 2-and 4-way bank conflicts**
- **Shared memory cost for whole tree**
  - 1 32-thread warp = 6 cycles per thread w/o conflicts
    - Counting 2 shared mem reads and one write (s[a] += s[b])
  - 6 * (2+4+4+4+2+1) = 102 cycles
  - 36 cycles if there were no bank conflicts (6 * 6)

Scan Bank Conflicts (2)

- **It's much worse with bigger trees!**
- **A full binary tree with 128 leaf nodes**
  - Only the last 6 iterations shown (root and 5 levels below)

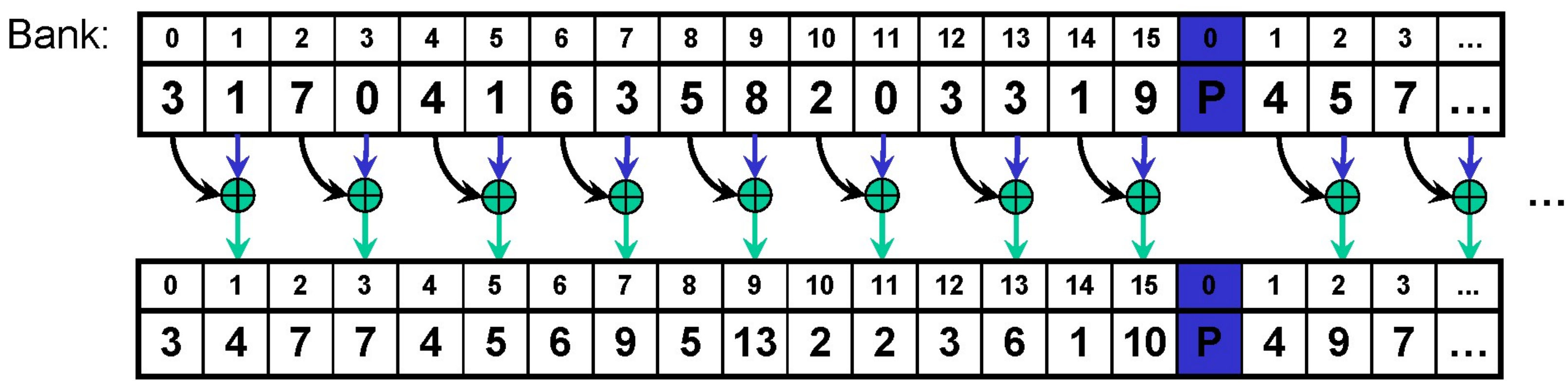
Scale (s)	Thread addresses																															
2	0	4	8	12	16	20	24	28	32	36	40	44	48	52	56	60	64	68	72	76	80	84	88	92	96	100	104	108	112	116	120	122
4	0	8	16	24	32	40	48	56	64	72	80	88	96	104	112	120																
8	0	16	32	48	64	80	96	112																								
16	0	32	64	96																												
32	0	64																														
64	0																															
Conflicts	Banks																															
4-way	0	4	8	12	0	4	8	12	0	4	8	12	0	4	8	12	0	4	8	12	0	4	8	12	0	4	8	12	0	4	8	10
8-way	0	8	0	8	0	8	0	8	0	8	0	8	0	8	0	8																
8-way	0	0	0	0	0	0	0	0																								
4-way	0	0	0	0																												
2-way	0	0																														
None	0																															

- **Cost for whole tree:**
  - 12*2 + 6*(4+8+8+4+2+1) = 186 cycles
  - 48 cycles if there were no bank conflicts! 12*1 + (6*6)

Bank Conflicts in the tree algorithm

- **We can use padding to prevent bank conflicts**
  - Just add a word of padding every 16 words:
- **No more conflicts!**

32 for full warps!



Now, within a 16-thread half-warp, all threads access different banks.

32-thread full warp!

(Note that only arrows with the same color happen simultaneously.)

Use Padding to Reduce Conflicts

- **This is a simple modification to the last exercise**

- **After you compute a shared mem address like this:**

```
Address =     stride * thid;
```

- **Add padding like this:**

```
Address += (Address >> 4); // divide by NUM_BANKS
```

- **This removes most bank conflicts**
  - Not all, in the case of deep trees

Fixing Scan Bank Conflicts

- **Insert padding every NUM_BANKS elements**

```
const int LOG_NUM_BANKS = 4; // 16 banks
int tid = threadIdx.x;
int s = 1;
// Traversal from leaves up to root
for (d = n>>1; d > 0; d >>= 1)
{
   if (thid <= d)
   {
        int a = s*(2*tid); int b = s*(2*tid+1)
        a += (a >> LOG_NUM_BANKS); // insert pad word
        b += (b >> LOG_NUM_BANKS); // insert pad word
        shared[a] += shared[b];
   }
}
```

Fixing Scan Bank Conflicts

- **A full binary tree with 64 leaf nodes**

Leaf Nodes	Scale (s)	Thread addresses																														
64	1	0	2	4	6	8	10	12	14	17	19	21	23	25	27	29	31	34	36	38	40	42	44	46	48	51	53	55	57	59	61	63
	2	0	4	8	12	17	21	25	29	34	38	42	46	51	55	59	63															
	4	0	8	17	25	34	42	51	59																							
	8	0	17	34	51																											
	16	0	34																													
	32	0																														
	Conflicts	Banks																														
	None	0	2	4	6	8	10	12	14	1	3	5	7	9	11	13	15	2	4	6	8	10	12	14	0	3	5	7	9	11	13	15
	None	0	4	8	12	1	5	9	13	2	6	10	14	3	7	11	15															
	None	0	8	1	9	2	10	3	11																							
	None	0	1	2	3																											
	None	0	2																													
	None	0																														

= Padding inserted

- **No more bank conflicts!**
  - However, there are ~8 cycles overhead for addressing
    - For each s[a] += s[b] (8 cycles/iter. * 6 iter. = 48 extra cycles)
  - So just barely worth the overhead on a small tree
    - 84 cycles vs. 102 with conflicts vs. 36 optimal

Fixing Scan Bank Conflicts

- **A full binary tree with 128 leaf nodes**
  - Only the last 6 iterations shown (root and 5 levels below)

Scale (s)	Thread addresses																														
2	0	4	8	12	17	21	25	29	34	38	42	46	51	55	59	63	68	72	76	80	85	89	93	97	102	106	110	114	119	123	127	131
4	0	8	17	25	34	42	51	59	68	76	85	93	102	110	119	127																
8	0	17	34	51	68	85	102	119																								
16	0	34	68	102																												
32	0	68																														
64	0																															

= Padding inserted

Conflicts	Banks																														
None	0	4	8	12	1	5	9	13	2	6	10	14	3	7	11	15	4	8	12	0	5	9	13	1	6	10	14	2	7	11	15	3
None	0	8	1	9	2	10	3	11	4	12	5	13	6	14	7	15																
None	0	1	2	3	4	5	6	7																								
None	0	2	4	6																												
None	0	4																														
None	0																															

- **No more bank conflicts!**
  - Significant performance win:
    - 106 cycles vs. 186 with bank conflicts vs. 48 optimal

Fixing Scan Bank Conflicts

- **A full binary tree with 512 leaf nodes**
  - Only the last 6 iterations shown (root and 5 levels below)

Scale (s)	Thread addresses																															
8	0	17	34	51	68	85	102	119	136	153	170	187	204	221	238	255	272	289	306	323	340	357	374	391	408	425	442	459	476	493	510	527
16	0	34	68	102	136	170	204	238	272	306	340	374	408	442	476	510																
32	0	68	136	204	272	340	408	476																								
64	0	136	272	408																												
128	0	272																														
256	0																															

= Padding inserted

Conflicts	Banks																															
None	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
2-way	0	2	4	6	8	10	12	14	0	2	4	6	8	10	12	14																
2-way	0	4	8	12	0	4	8	12																								
2-way	0	8	0	8																												
2-way	0	0																														
None	0																															

- **Wait, we still have bank conflicts**
  - Method is not foolproof, but still much improved
  - 304 cycles vs. 570 with bank conflicts vs. 120 optimal
- **But it does not pay of to optimize for the rest. Address calculations are getting too expensive**

Summary

- **Parallel Programming requires careful planning**
  - of the branching behavior
  - of the memory access patterns
  - of the work efficiency

- **Vector Reduction**
  - branch efficient
  - bank efficient

- **Scan Algorithm**
  - based in Balanced Tree principle:
    bottom up, top down traversal

GPU TECHNOLOGY CONFERENCE

Shuffle: Tips and Tricks

Julien Demouth, NVIDIA

Glossary

Safer with cooperative thread groups!

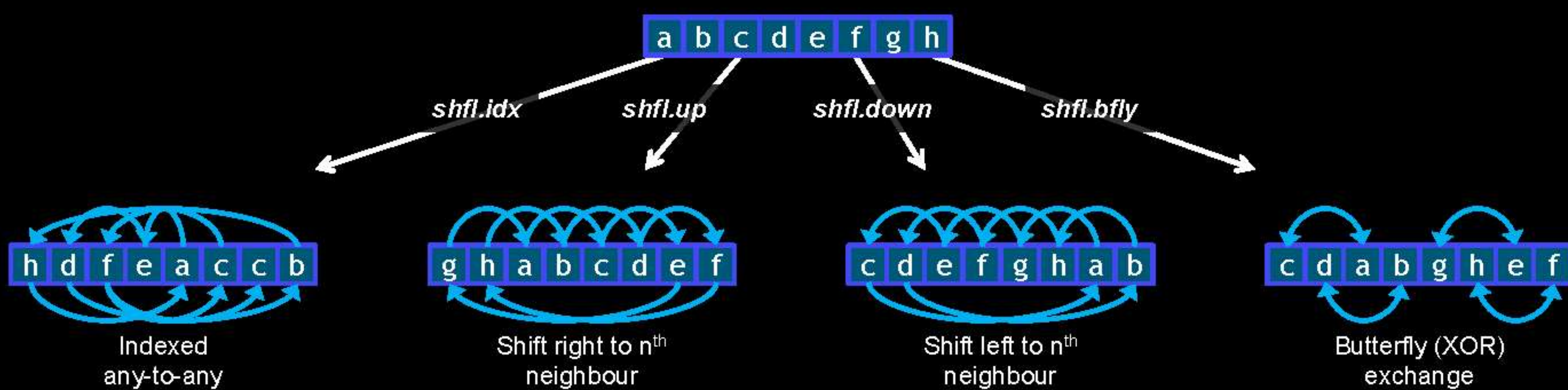
- Warp
  - ~~Implicitly synchronized~~ group of threads (32 on current HW)
- Warp ID (`warpid`)
  - Identifier of the warp in a block: `threadIdx.x / 32`
- Lane ID (`laneid`)
  - Coordinate of the thread in a warp: `threadIdx.x % 32`
  - Special register (available from PTX): `%laneid`

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

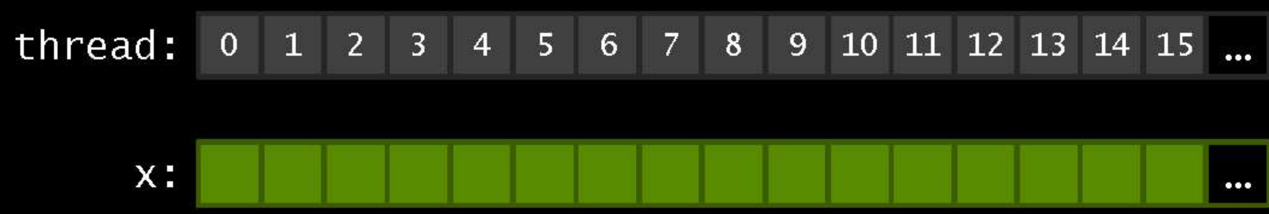
```
__device__ __inline__ double shfl(double x, int lane)
{
    // Split the double number into 2 32b registers.
    int lo, hi;
    asm volatile( “mov.b32 {%0,%1}, %2;” : “=r”(lo), “=r”(hi) : “d”(x));

    // 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

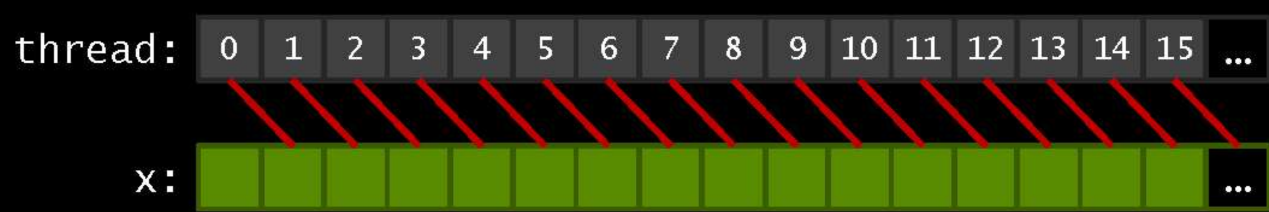
GPU TECHNOLOGY CONFERENCE

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();
```

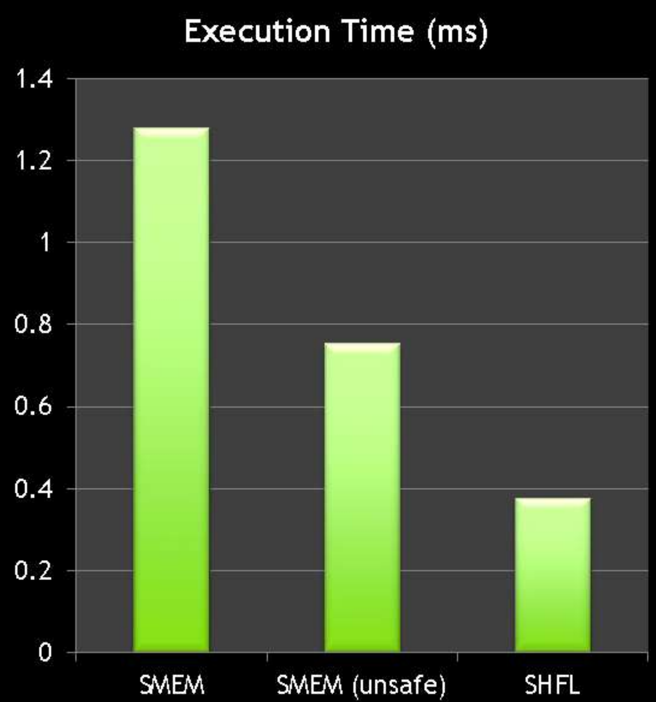
- Shuffle (SHFL)

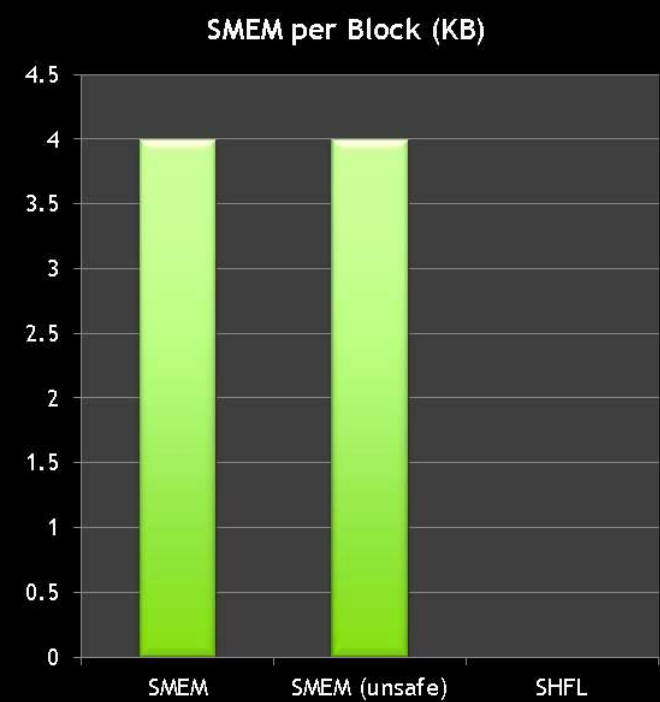
```
x = __shfl(x, (laneid+1) % 32);
```

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

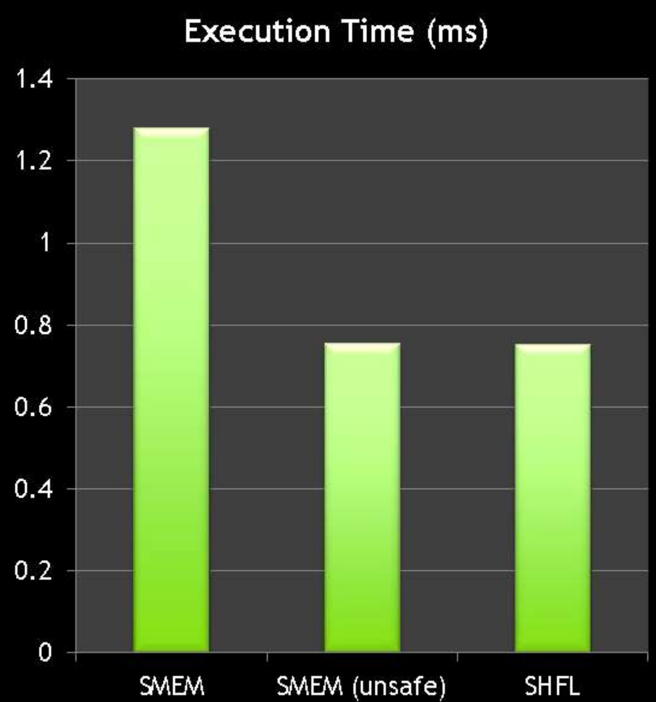
```
__shared__ volatile T *smem = ...;
smem[threadIdx.x] = smem[32*warpid + ((laneid+1) % 32)];
```

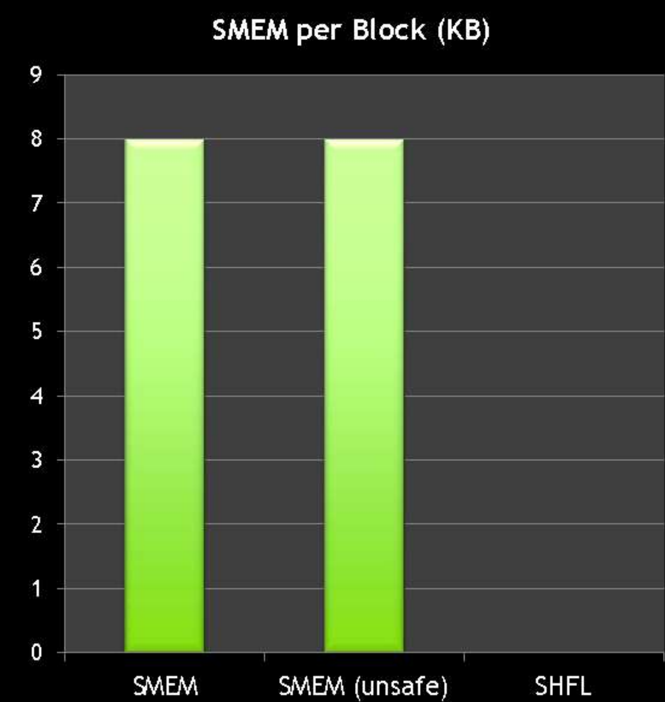
Performance Experiment (fp32)





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

GPU TECHNOLOGY CONFERENCE

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

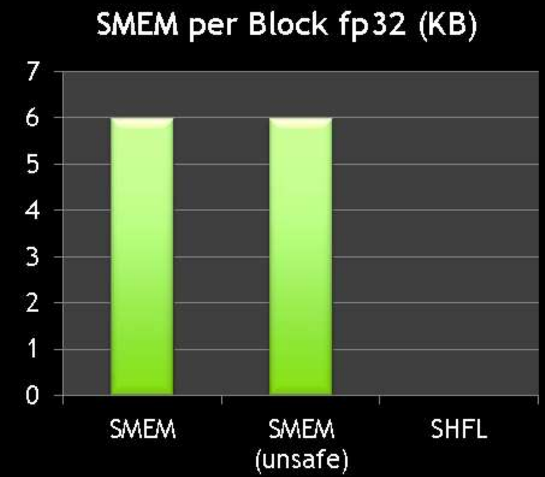
```
// All threads evaluate a predicate.
int predicate = ...;

// 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

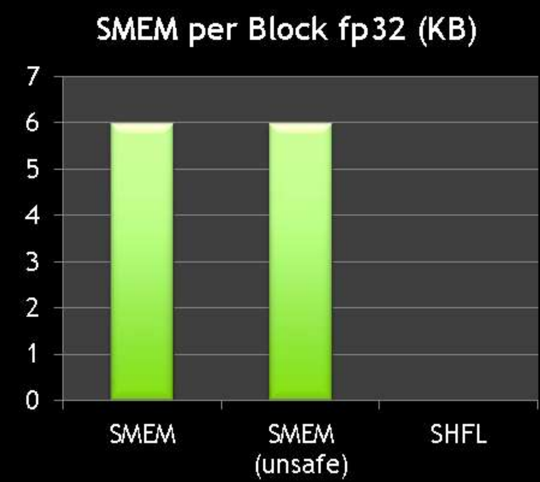


Scan

- 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

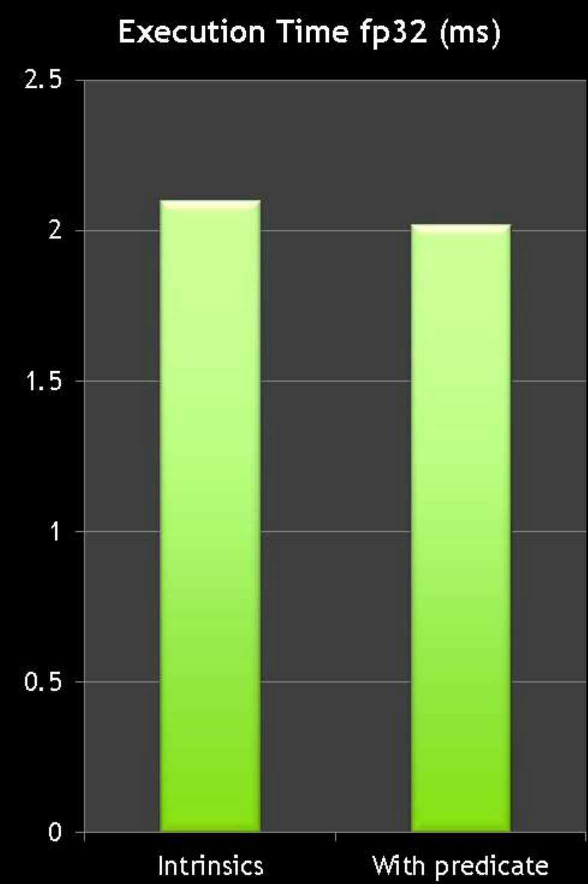


Scan

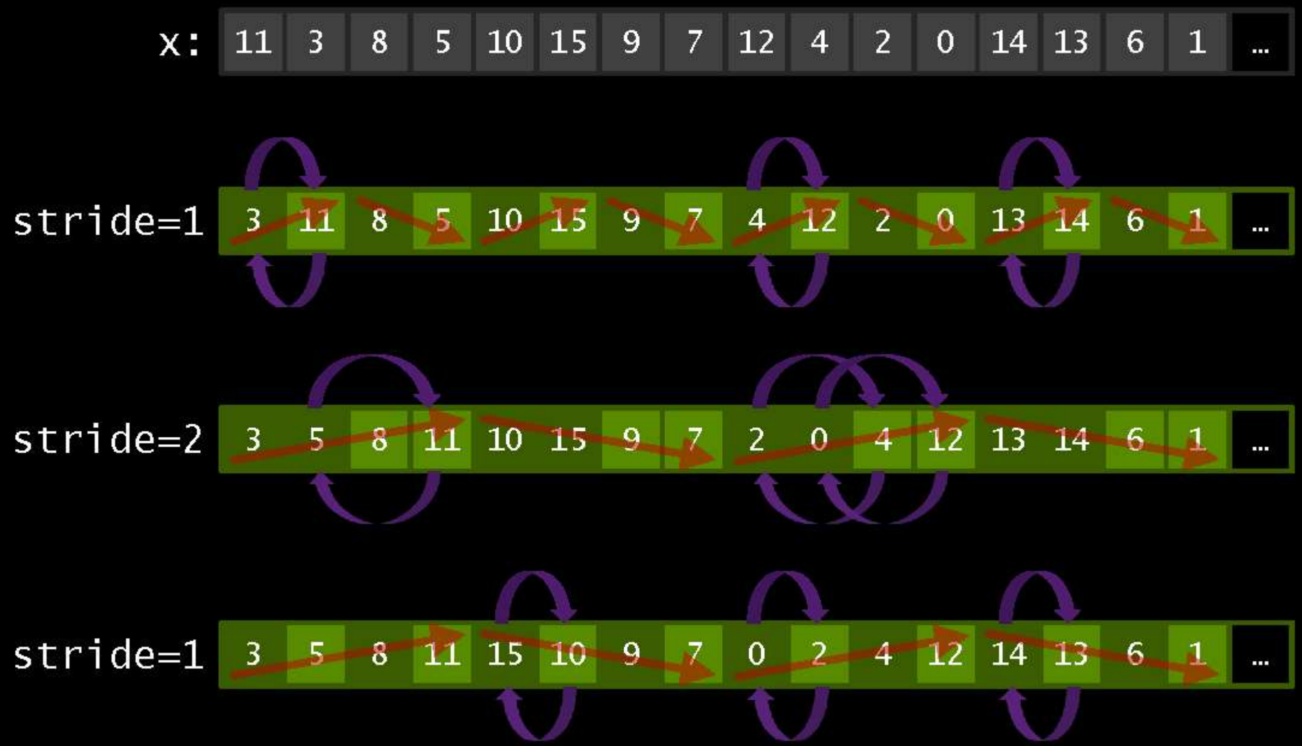
- Use the predicate from SHFL

```
#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));
}
```

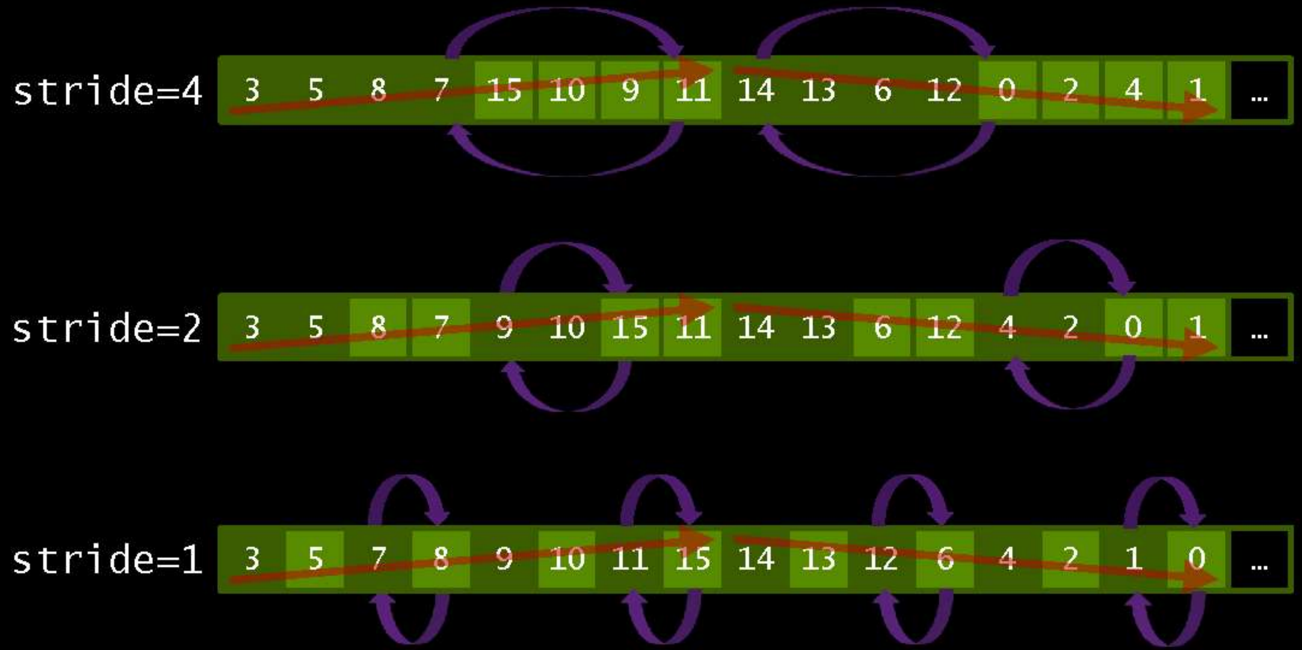
- Use CUB:
  https://nvlabs.github.com/cub



Bitonic Sort

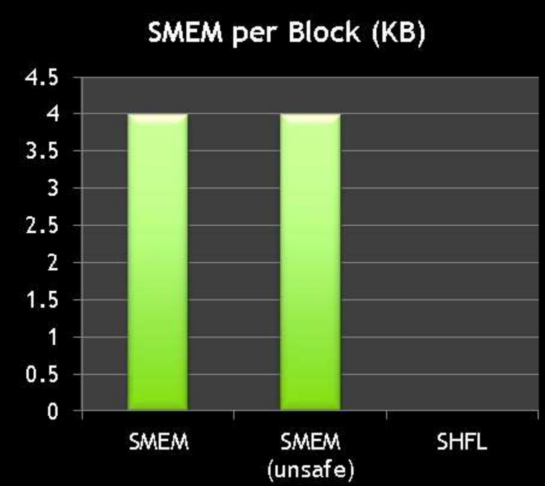


Bitonic Sort



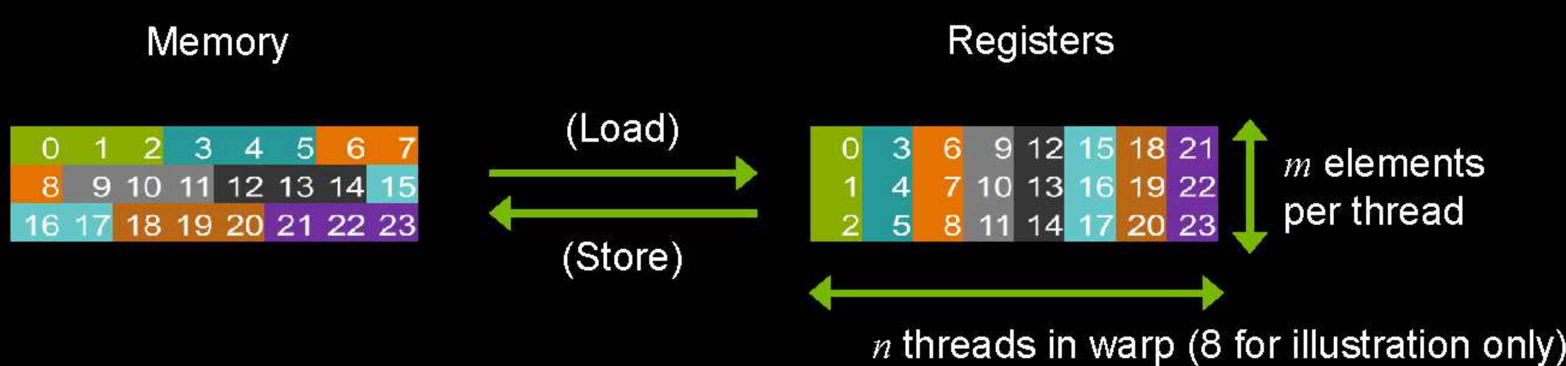
Bitonic Sort

```
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)
```



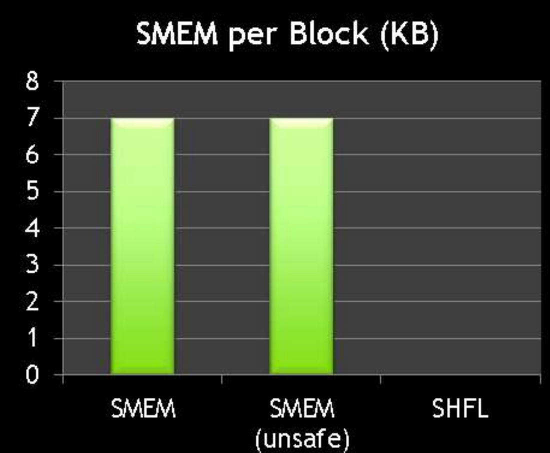
Transpose

- 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

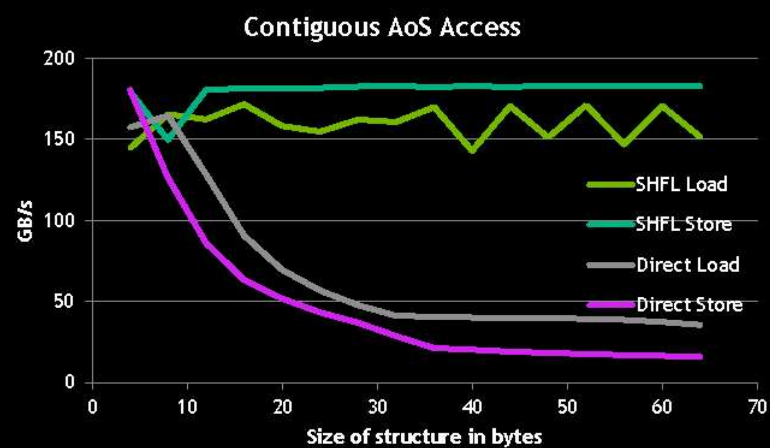
- You can use SMEM to implement this transpose, or you can use SHFL
- Code: http://github.com/bryancatanzaro/trove
- Performance
  - Launch 104 blocks of 256 threads
  - Run the transpose 4096 times

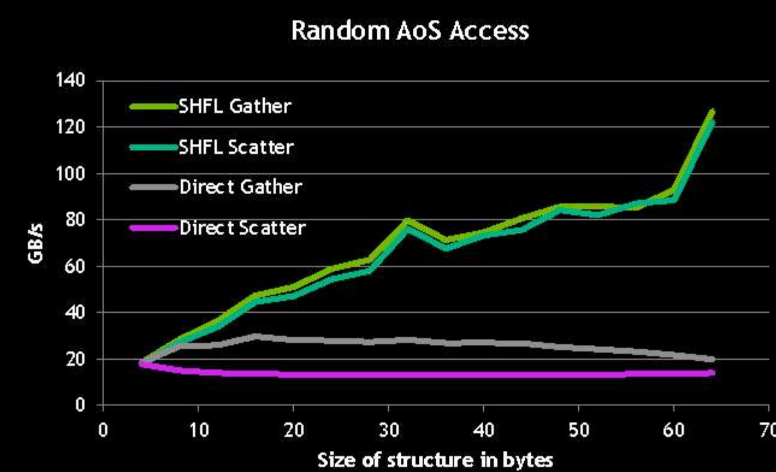


GPU TECHNOLOGY CONFERENCE

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 >= 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.

- Hendrik Lensch, Robert Strzodka
- NVIDIA