King Abdullah University of Science and Technology

KAUST

# CS 380 - GPU and GPGPU Programming
# Lecture 18: Shuffle Instructions, Pt. 2; GPU Parallel Prefix Sum

Markus Hadwiger, KAUST

# Reading Assignment #11 (until Nov 17)

Read (required):

- Interpolation for Polygon Texture Mapping and Shading,
  Paul Heckbert and Henry Moreton

  `https://www.ri.cmu.edu/publications/interpolation-for-polygon-texture-mapping-and-shading/`

- Homogeneous Coordinates

  `https://en.wikipedia.org/wiki/Homogeneous_coordinates`

Read (optional; highly recommended!):

- MIP-Map Level Selection for Texture Mapping

  `https://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=765326`

# Next Lectures

Lecture 18: Mon, Nov 10

Lecture 19: Tue, Nov 11 (make-up lecture; 14:30 – 16:00, room 3131)

Lecture 20: Thu, Nov 13

Lecture 21: Mon, Nov 17 (Quiz #2)

Lecture 22: Tue, Nov 18 (make-up lecture ; 14:30 – 16:00, room 3131)

Lecture 23: Thu, Nov 20

# Quiz #2: Oct 17

Organization

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

Content of questions

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

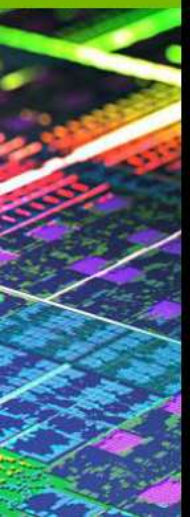

# Shuffle: Tips and Tricks

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`

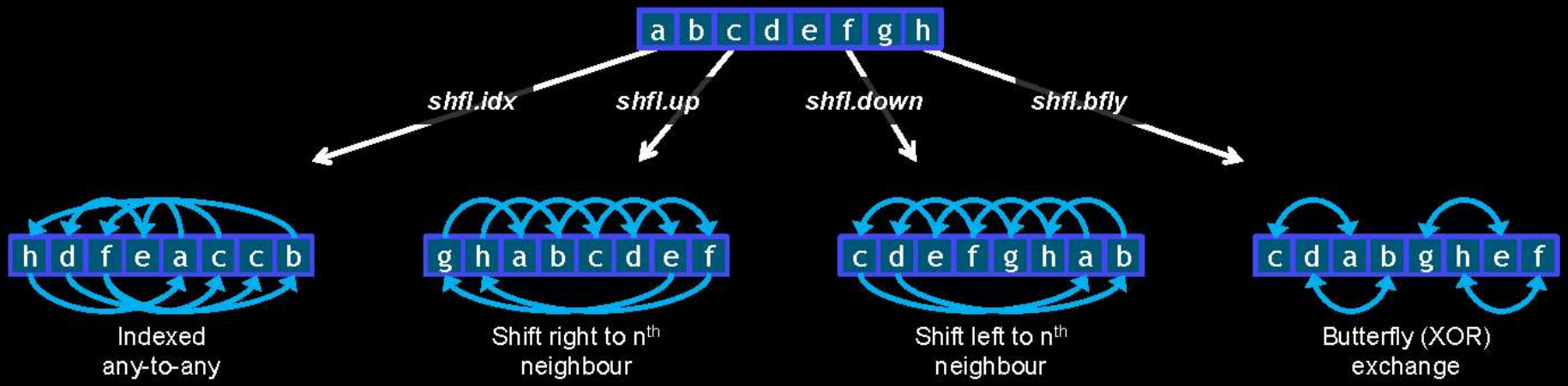


# Shuffle (SHFL)

- 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

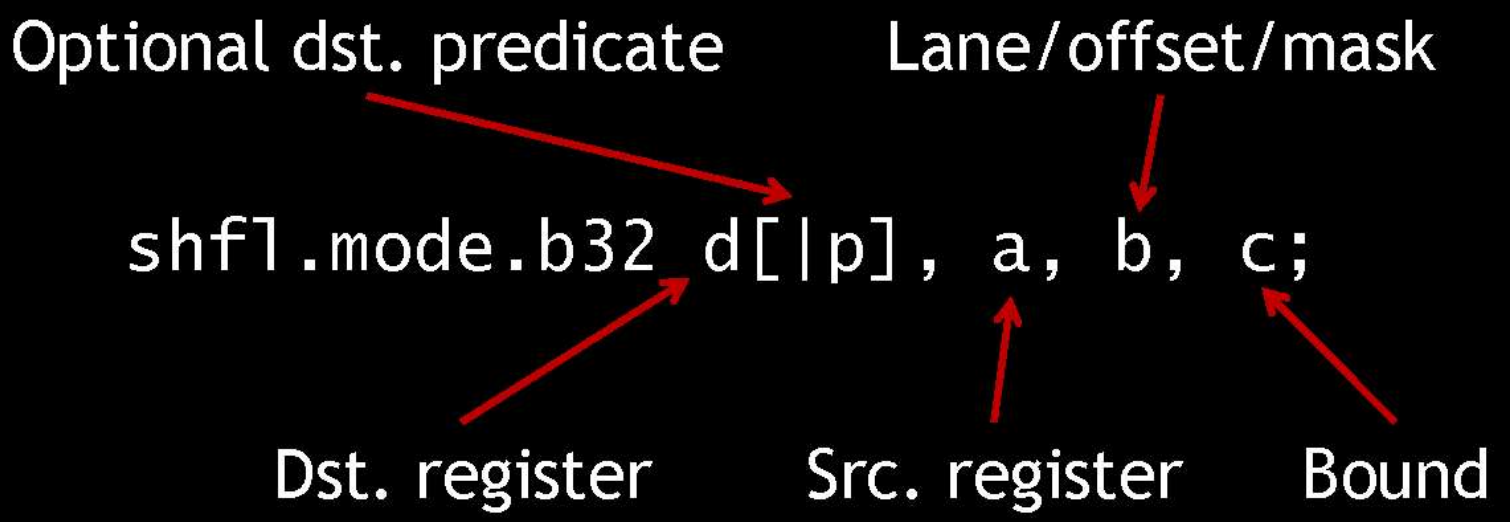
# Variants

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




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

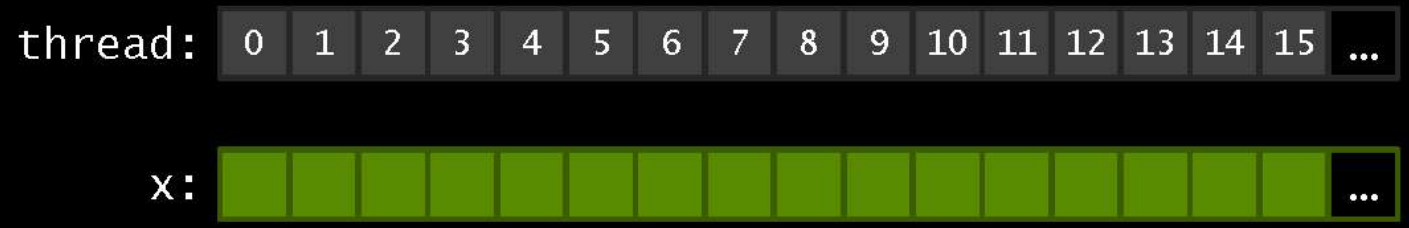
# Instruction (PTX)



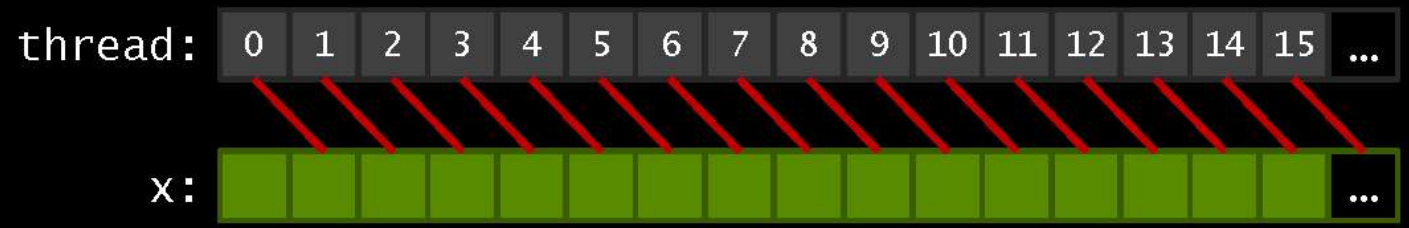

Now: Use _sync variants / shuffle in cooperative thread groups!

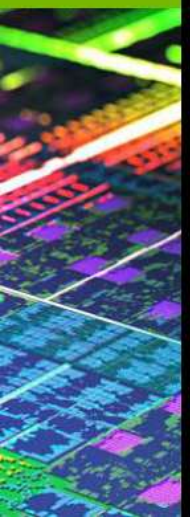
# Performance Experiment

- One element per thread




- Each thread takes its right neighbor





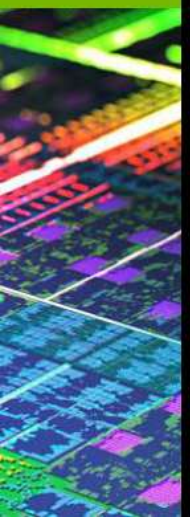
# Performance Experiment

- 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();
```

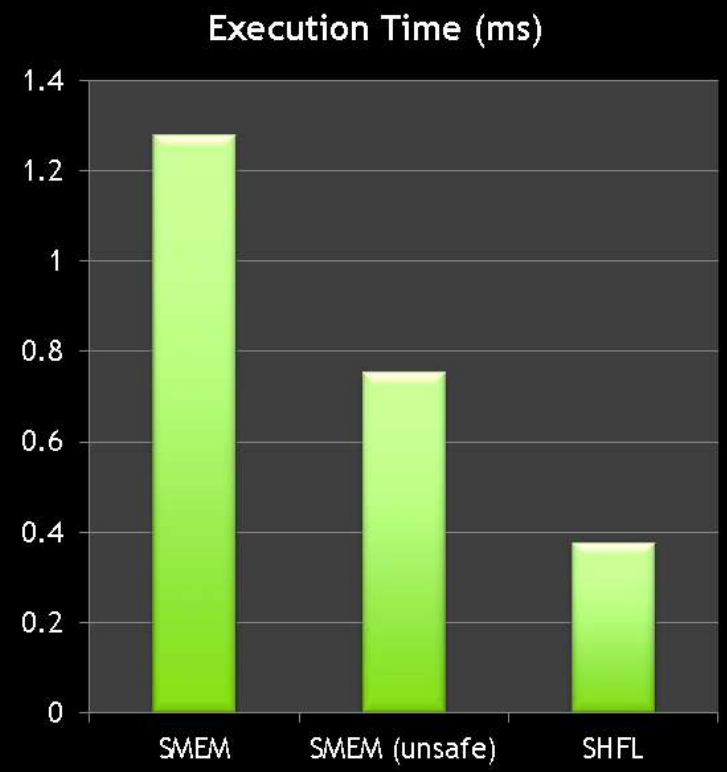
- Shuffle (SHFL)

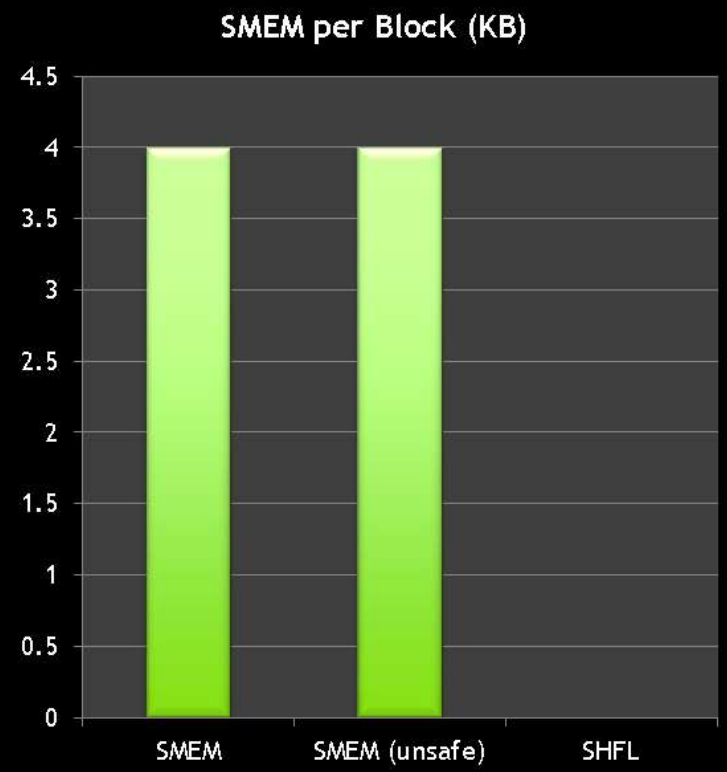
```
x = __shfl(x, (laneid+1) % 32);
```

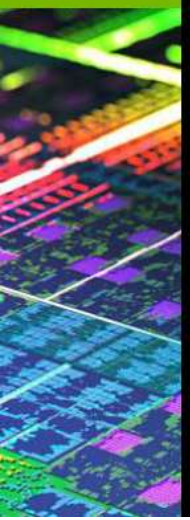
- Shared memory without __syncthreads + volatile (*unsafe*)

```
__shared__ volatile T *smem = ...;

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

# Performance Experiment (fp32)

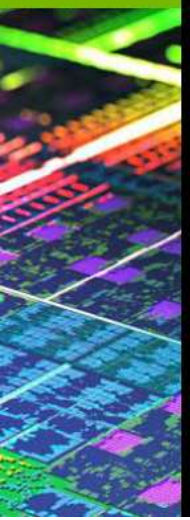




# Performance Experiment

- 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

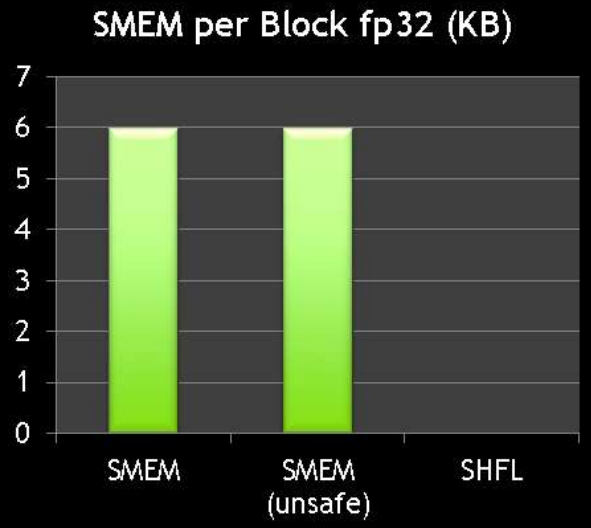
```
// All threads evaluate a predicate.
int predicate = ...;

// 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

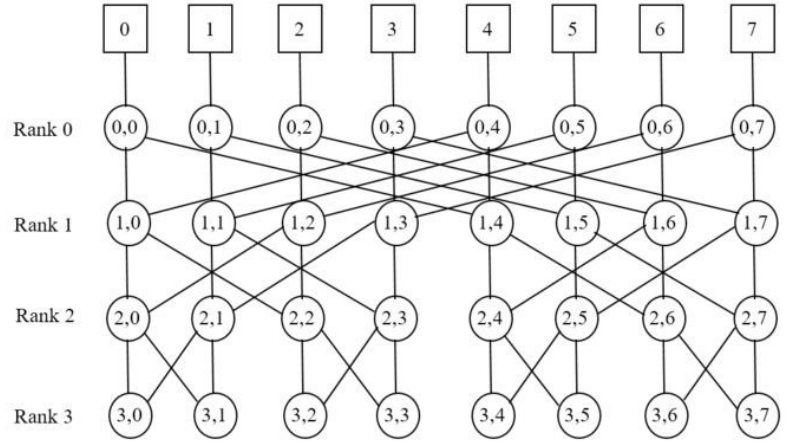
# Reduce

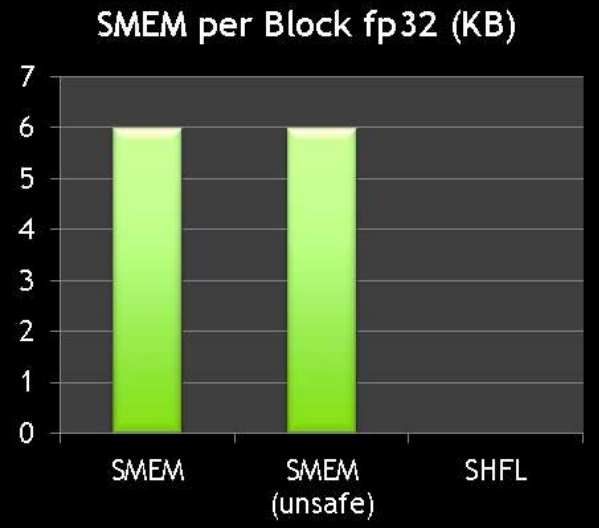
- Code

```
// 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

butterfly pattern



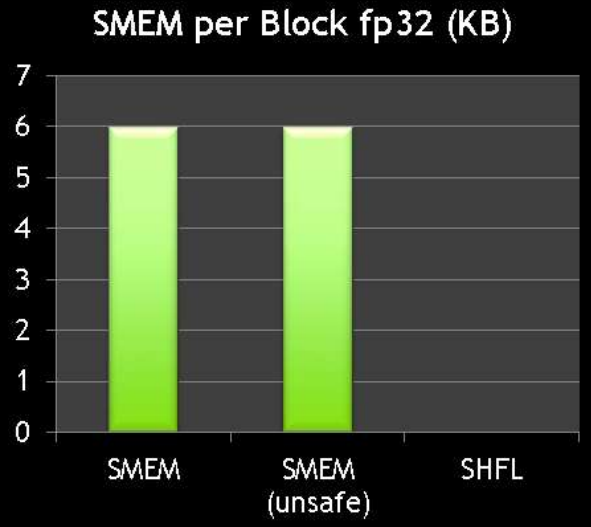



# Scan

- 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

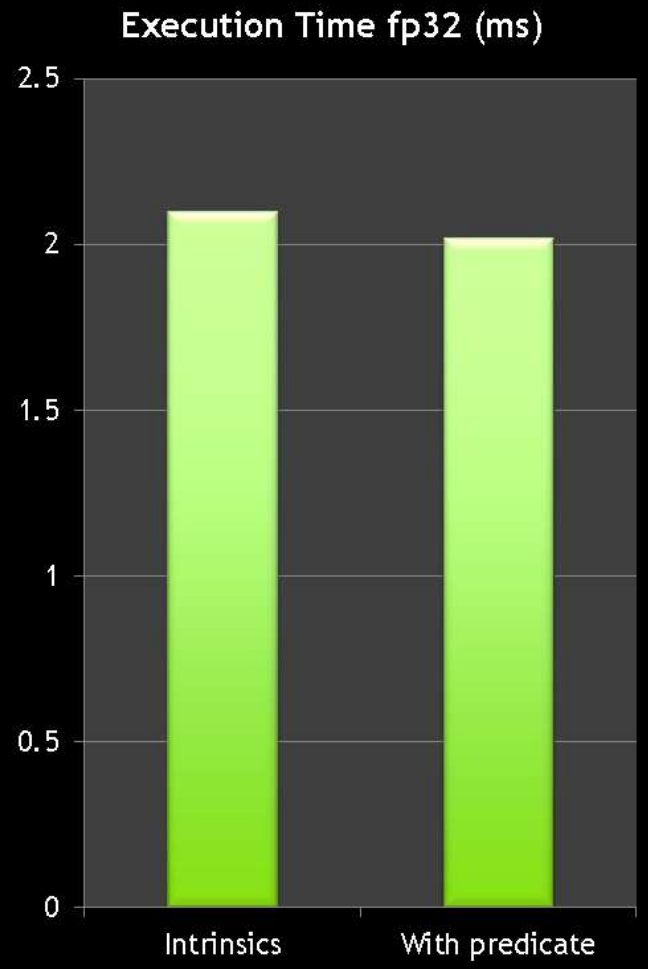



# Scan

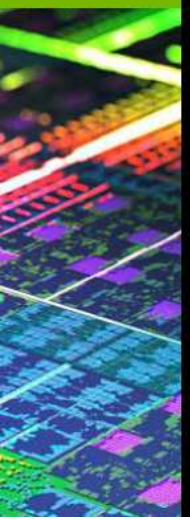
- Use the predicate from SHFL

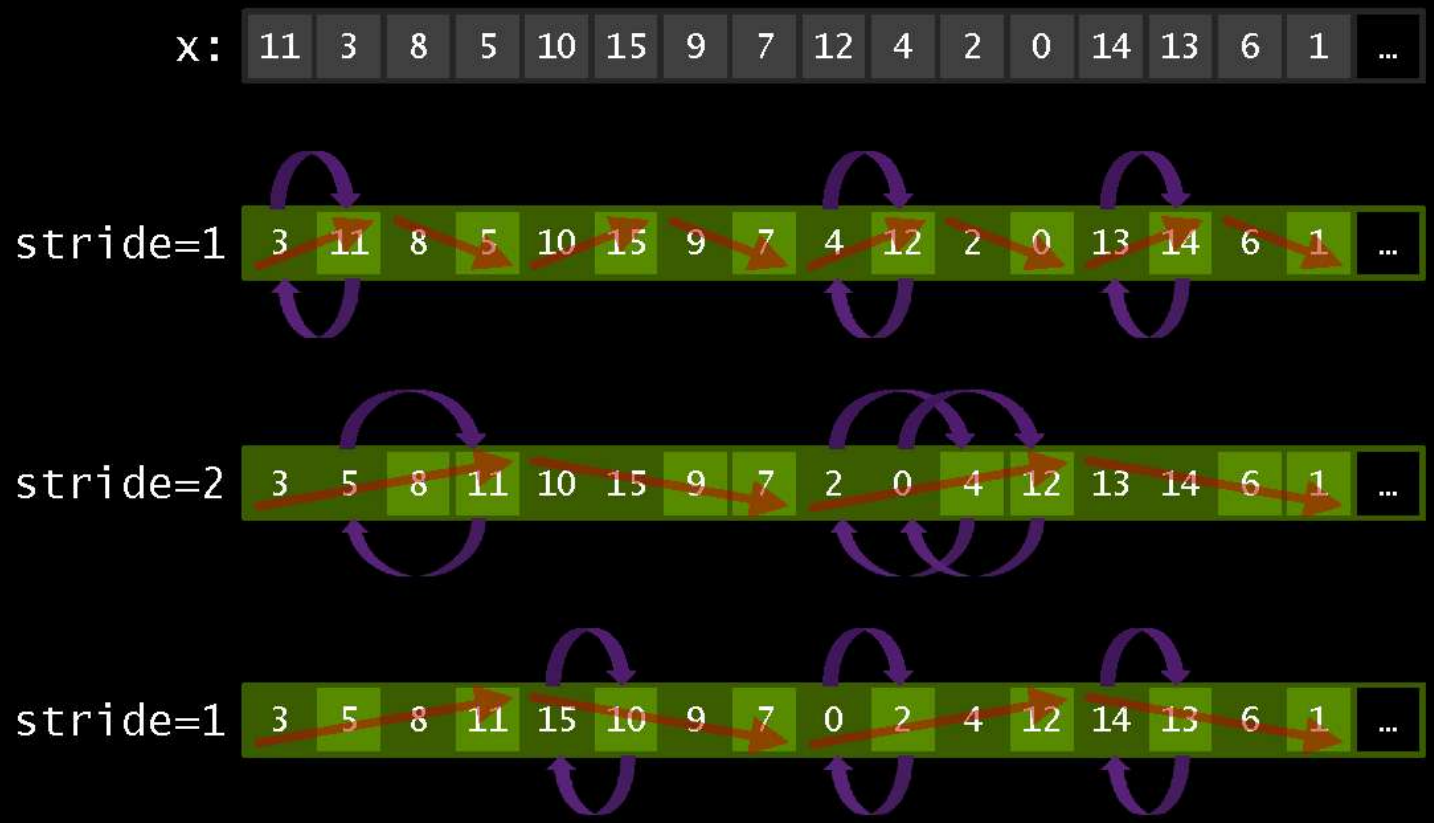
```
#pragma unroll
for( int offset = 1 ; offset < 32 ; offset <<= 1 )
{
    asm volatile( "{"
          "  .reg .f32 r0;"
          "  .reg .pred p;"
          "  shfl.up.b32 r0|p, %0, %1, 0x0;"
          "  @p add.f32 r0, r0, %0;"
          "  mov.f32 %0, r0;"
          "}" : "+f"(x) : "r"(offset));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

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

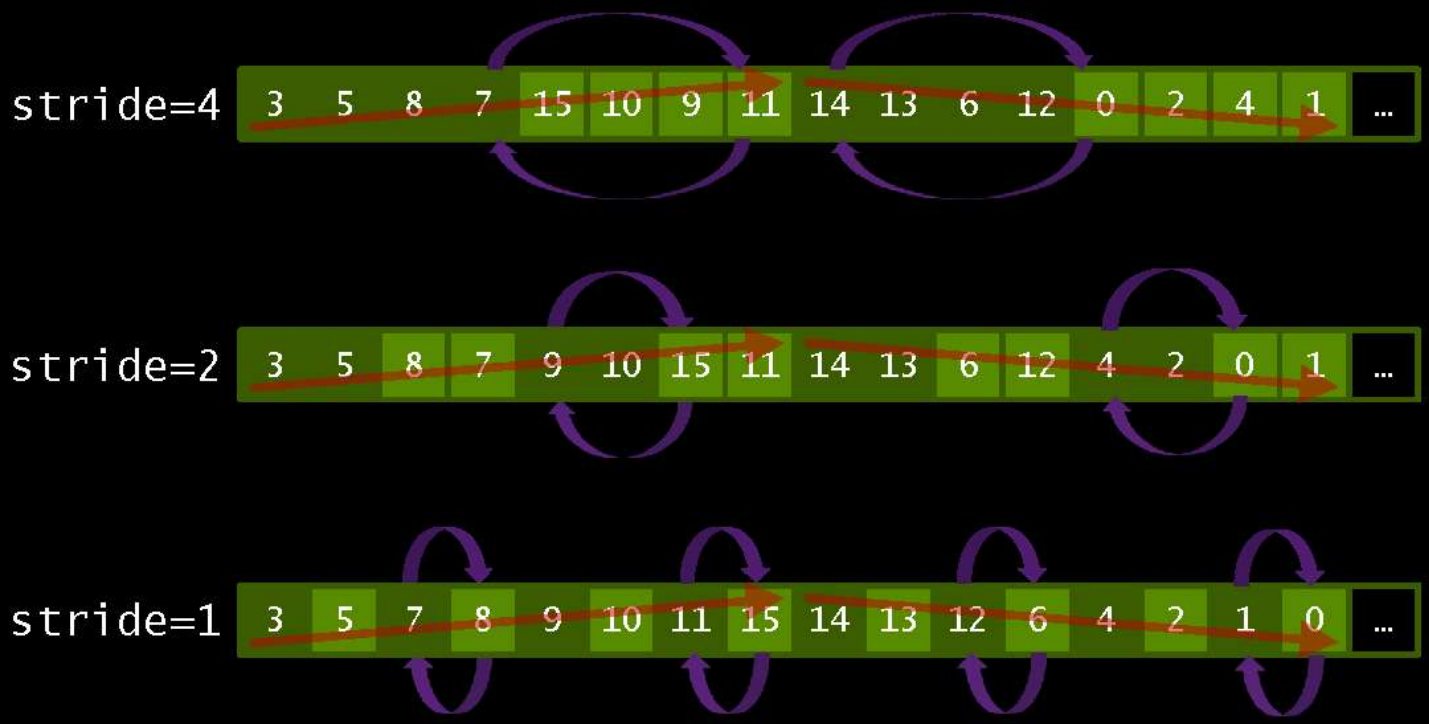


# Bitonic Sort



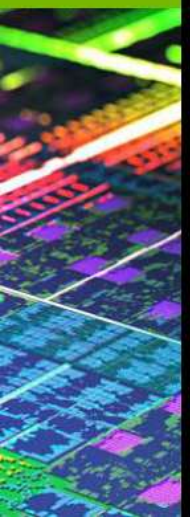


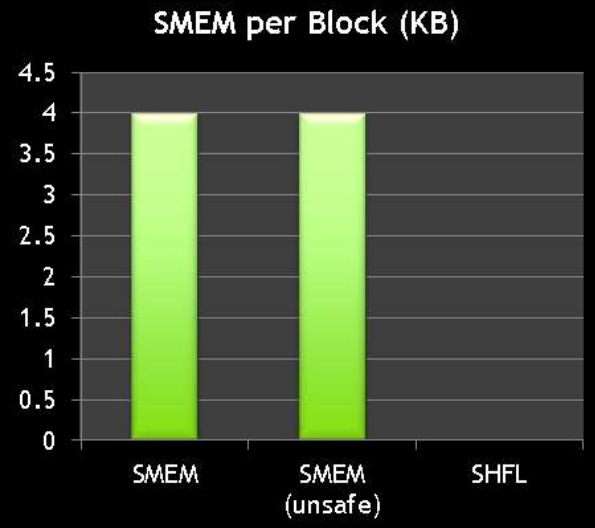
# Bitonic Sort



# Bitonic Sort

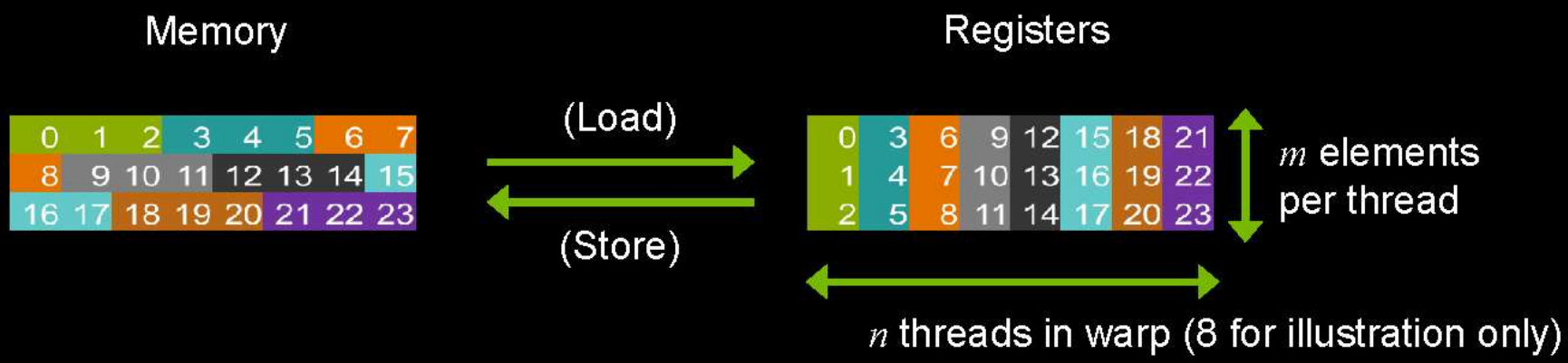
```
int swap(int x, int mask, int dir)
{
     int y = __shfl_xor(x, mask);
     return x < y == dir ? y : x;
}

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)
```



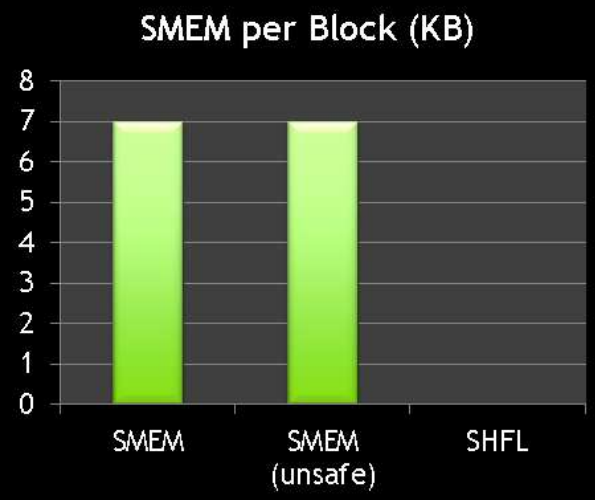
# Transpose

- 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



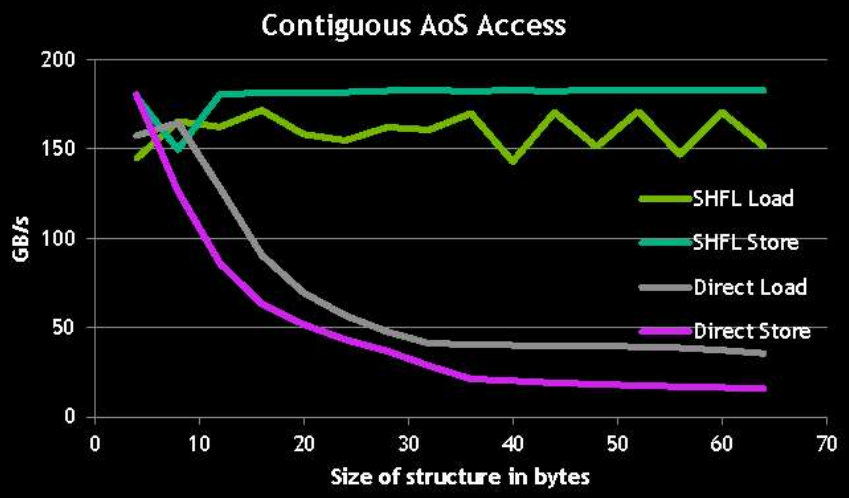
# Transpose

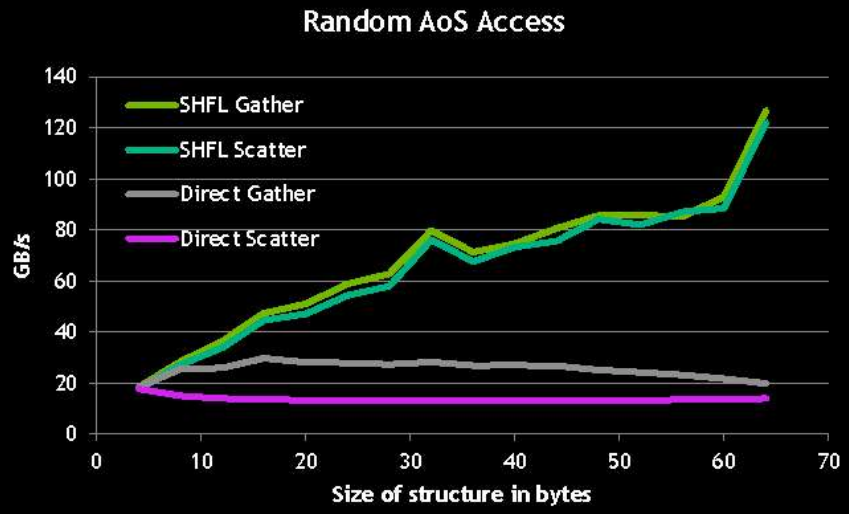
- You can use SMEM to implement this transpose, or you can use SHFL
- Code:
  http://github.com/bryancatanzaro/trove
- Performance
  - Launch 104 blocks of 256 threads
  - Run the transpose 4096 times

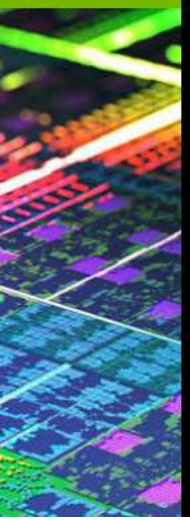



# Array of Structures Access via Transpose

- 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

# GPU Parallel Prefix Sum

- Basic parallel programming primitive;
  parallelize inherently sequential operations

# Parallel Prefix Sum (Scan)

- **Definition:**

  **The all-prefix-sums operation takes a binary associative operator ⊕ with identity *I*, and an array of n elements**

  $$[a_0, a_1, \ldots, a_{n-1}]$$

  **and returns the ordered set**

  $$[I, a_0, (a_0 \oplus a_1), \ldots, (a_0 \oplus a_1 \oplus \ldots \oplus a_{n-2})].$$

- **Example:**
  **if ⊕ is addition, then scan on the set**

  **[3 1 7 0 4 1 6 3]**

  **returns the set**

  **[0 3 4 11 11 15 16 22]**

Exclusive scan: last input element is not included in the result

***(From Blelloch, 1990, "Prefix Sums and Their Applications)***

# Applications of Scan

- **Scan is a simple and useful parallel building block**
  - Convert recurrences from sequential :

```
for(j=1;j<n;j++)
   out[j] = out[j-1] + f(j);
```

  - into parallel:

```
forall(j) { temp[j] = f(j) };
scan(out, temp);
```

- **Useful for many parallel algorithms:**
  - radix sort
  - quicksort
  - String comparison
  - Lexical analysis
  - Stream compaction
  - Polynomial evaluation
  - Solving recurrences
  - Tree operations
  - Range Histograms
  - Etc.

# Scan on the CPU

```
void scan( float* scanned, float* input, int length)
{
  scanned[0] = 0;
  for(int i = 1; i < length; ++i)
  {
    scanned[i] = input[i-1] + scanned[i-1];
  }
}
```

- **Just add each element to the sum of the elements before it**
- **Trivial, but sequential**
- **Exactly *n* adds: optimal in terms of work efficiency**

# Prefix Sum Application
# - Compaction -

# Parallel Data Compaction

- **Given an array of marked values**

| 3 | 1 | 7 | 4 | 2 | 1 | 5 | 6 | 3 | 1 |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |

- **Output the compacted list of marked values**

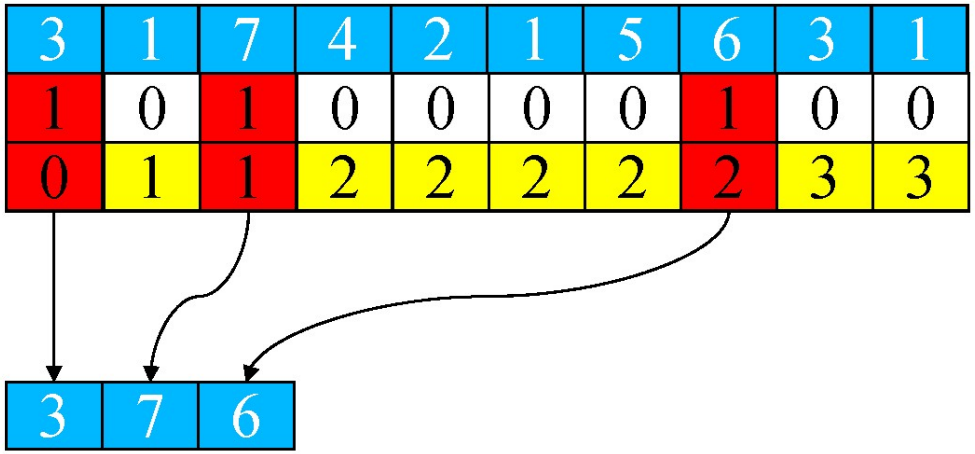
| 3 | 7 | 6 |
|---|---|---|

# Using Prefix Sum

- **Calculate prefix sum on index array**

| 3 | 1 | 7 | 4 | 2 | 1 | 5 | 6 | 3 | 1 |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 2 | 2 | 2 | 2 | 2 | 3 | 3 |

- **For each marked value lookup the destination index in the prefix sum**




- **Parallel write to separate destination elements**

# Prefix Sum Application
# - Range Histogram -

# Range Histogram

- **A histogram calculate the occurance of each value in an array.**

  **h[i] = |J|  J={j| v[j] = i}**

- **Range query: number over elements in interval [a,b].**

- **Slow answer:**

```
hrange = 0;
for (i = a; i<=b; ++i)
    hrange += h[i];
```

# Fast Range Histogram

- **Compute prefix sum of histogram**
- **Fast answer:**

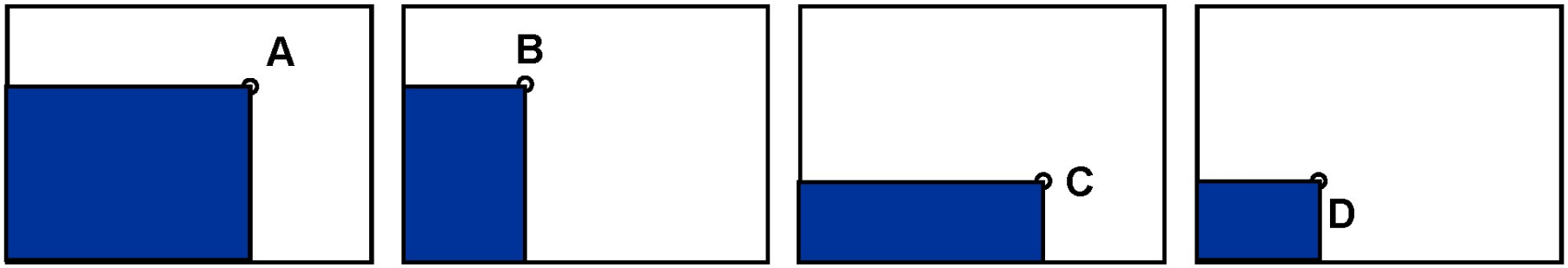
```
hrange = pref[B] - pref[A];
```

$$= \sum_{0}^{B} h[i] - \sum_{0}^{A} h[i] = \sum_{A}^{B} h[i]$$

# Prefix Sum Application
# - Summed Area Tables -

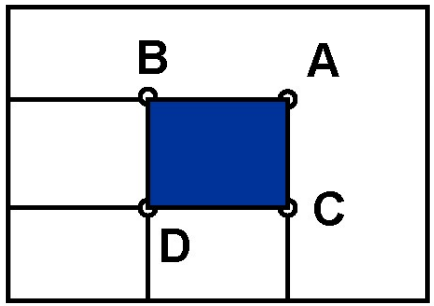
# Summed Area Tables

- **Per texel, store sum from (0, 0) to (u, v)**



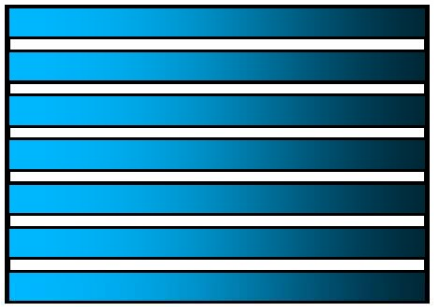

- **Many bits per texel (sum !)**
- **Evaluation of 2D integrals in constant time!**

$$\int_{Bx}^{Ax}\int_{Cy}^{Ay} I(x,y)\,dx\,dy = A - B - C + D$$

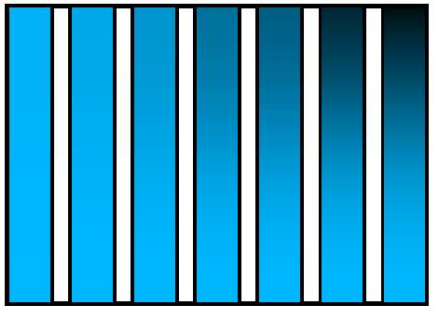


# Summed Area Table with Prefix Sums

- **One possible way:**
- **Compute prefix sum horizontally**



- **… then vertically on the result**



# Work Efficiency

Guy E. Blelloch and Bruce M. Maggs:
Parallel Algorithms, 2004 (https://www.cs.cmu.edu/~guyb/papers/BM04.pdf)

In designing a parallel algorithm, it is more important to make it efficient than to make it asymptotically fast. The efficiency of an algorithm is determined by the total number of operations, or work that it performs. On a sequential machine, an algorithm's work is the same as its time. On a parallel machine, the work is simply the processor-time product. Hence, an algorithm that takes time t on a P-processor machine performs work W = Pt. In either case, the work roughly captures the actual cost to perform the computation, assuming that the cost of a parallel machine is proportional to the number of processors in the machine.

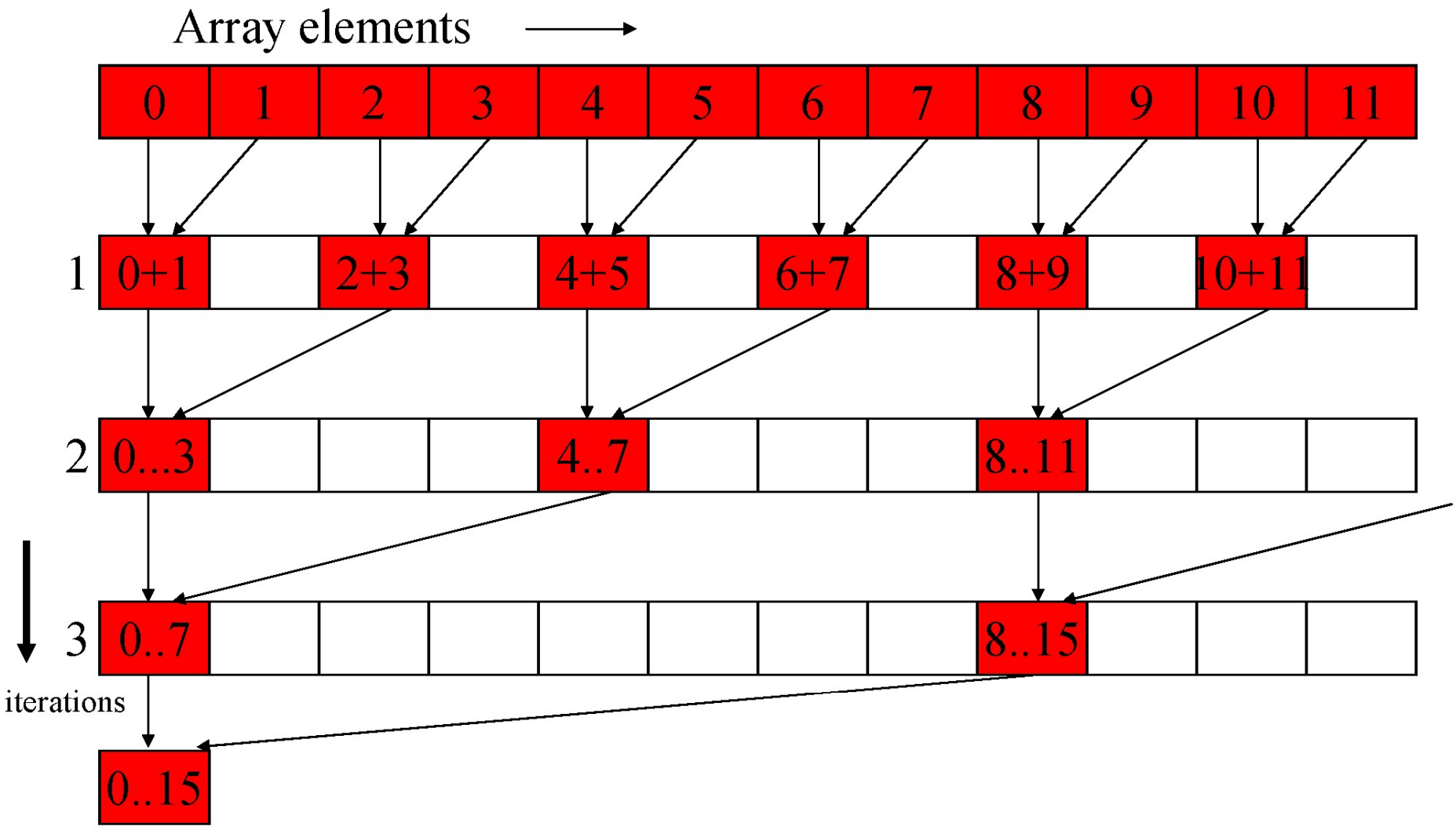
We call an algorithm work-efficient (or just efficient) if it performs the same amount of work, to within a constant factor, as the fastest known sequential algorithm.

For example, a parallel algorithm that sorts n keys in O( sqrt(n) log(n) ) time using sqrt(n) processors is efficient since the work, O( n log(n) ), is as good as any (comparison-based) sequential algorithm.

However, a sorting algorithm that runs in O( log(n) ) time using n^2 processors is not efficient.

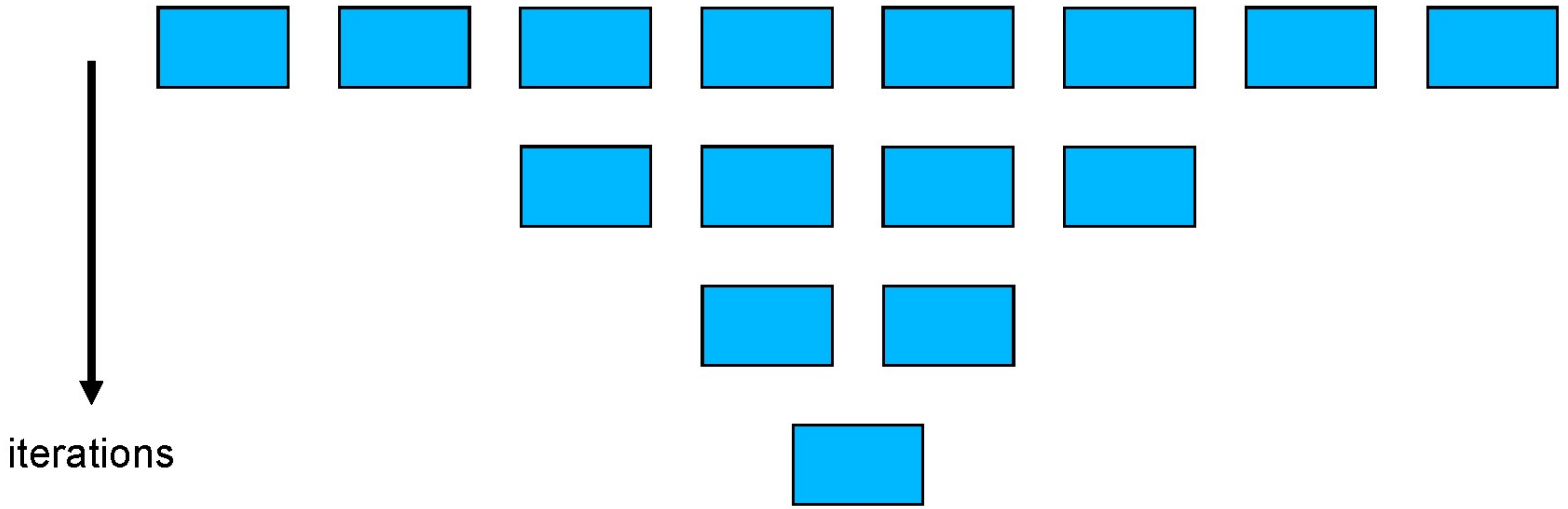
The first algorithm is better than the second - even though it is slower - because its work, or cost, is smaller. Of course, given two parallel algorithms that perform the same amount of work, the faster one is generally better.

# Vector Reduction



# Typical Parallel Programming Pattern

- **log(n) steps**

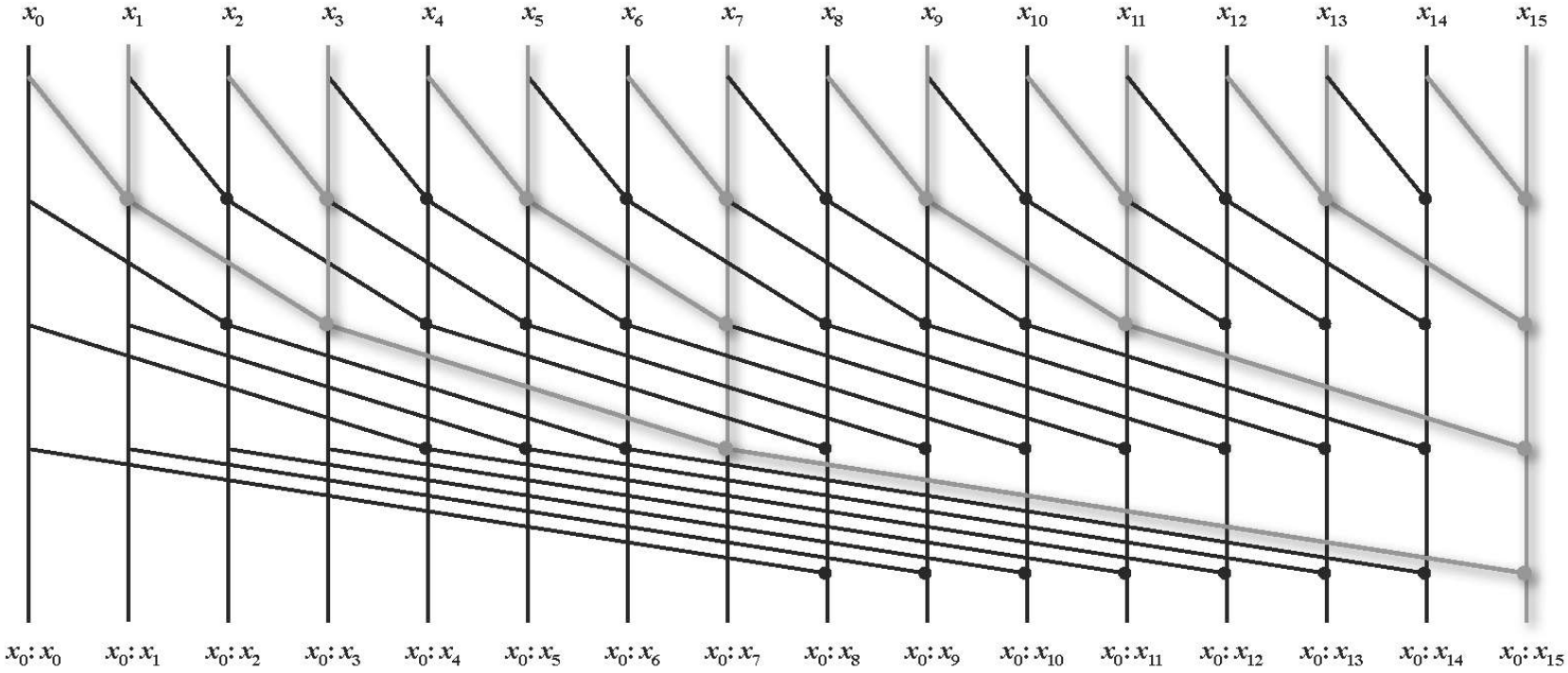



Helpful fact for counting nodes of full binary trees:
If there are N leaf nodes, there will be N-1 non-leaf nodes

Courtesy John Owens

# Kogge-Stone Scan

Circuit family

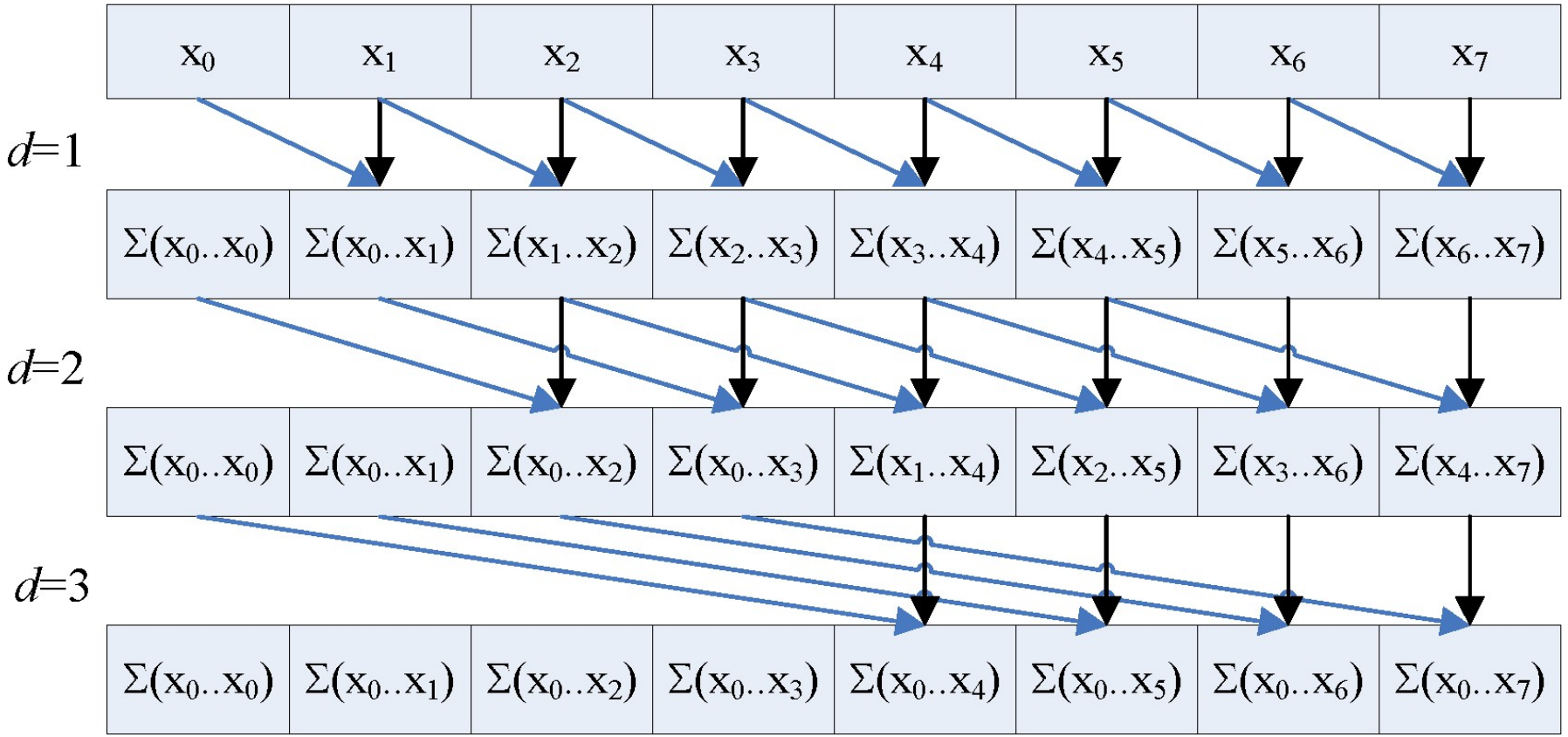



*A Parallel Algorithm for the Efficient Solution of a General Class of Recurrence Equations*, Kogge and Stone, 1973

See “carry lookahead” adders vs. “ripple carry” adders

# $O(n \log n)$ Scan

Courtesy John Owens

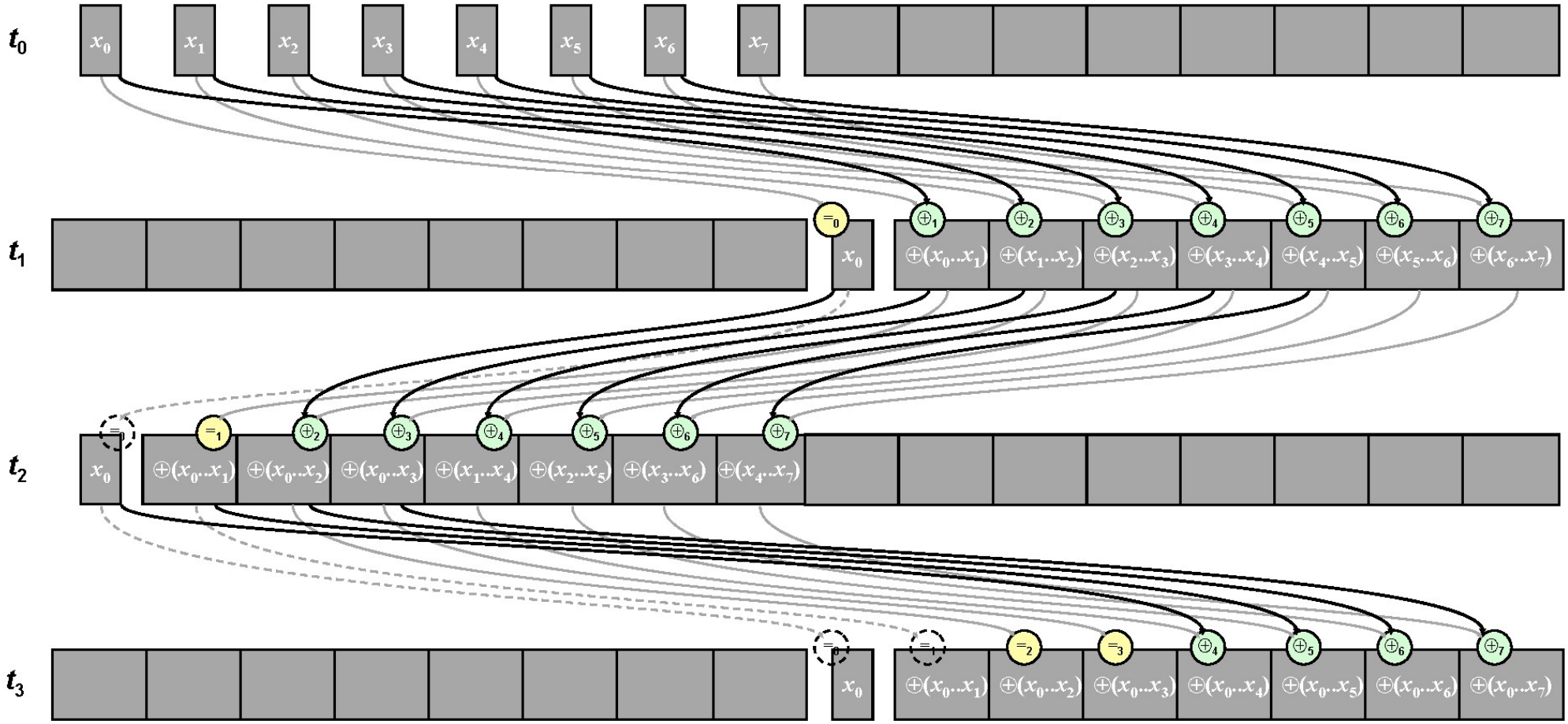



- Step efficient (log *n* steps)
- Not work efficient (*n* log *n* work)
- Requires barriers at each step (WAR dependencies)

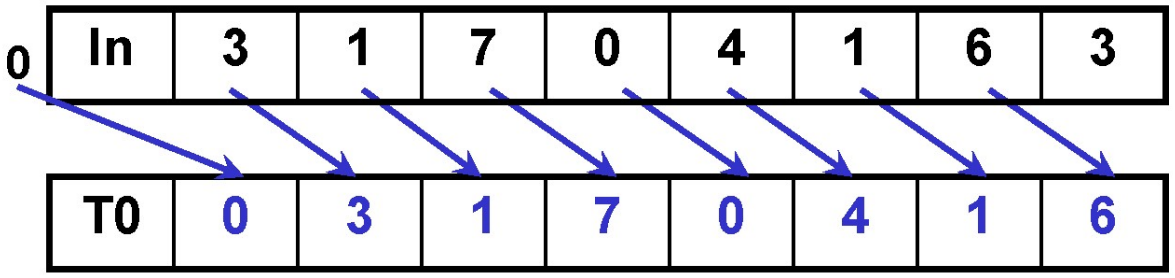
Courtesy John Owens

# Hillis-Steele Scan Implementation

No WAR conflicts, $O(2N)$ storage



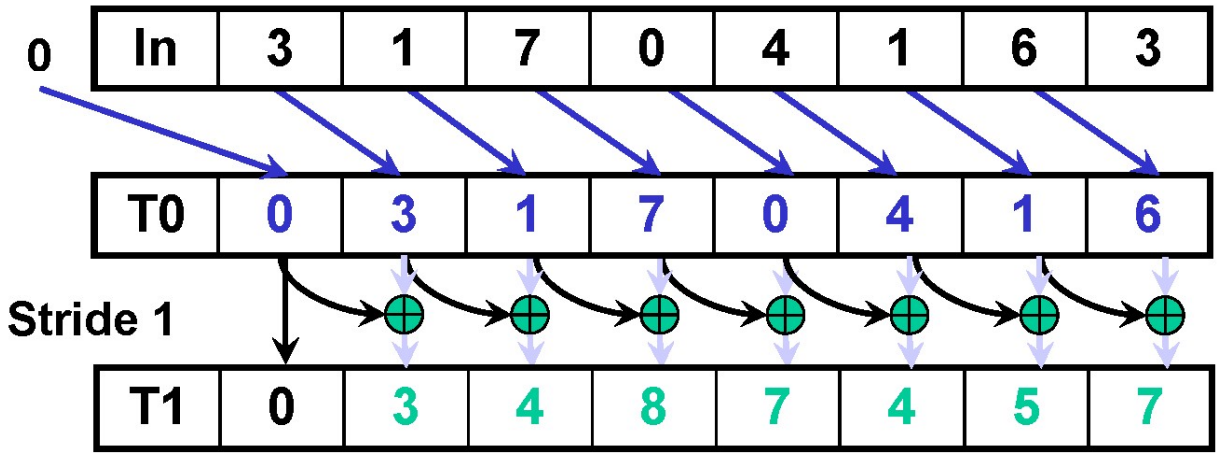
# A First-Attempt Parallel Scan Algorithm




Each thread reads one value from the input array in device memory into shared memory array T0. Thread 0 writes 0 into shared memory array.

1. Read input from device memory to shared memory. Set first element to zero and shift others right by one.

# A First-Attempt Parallel Scan Algorithm

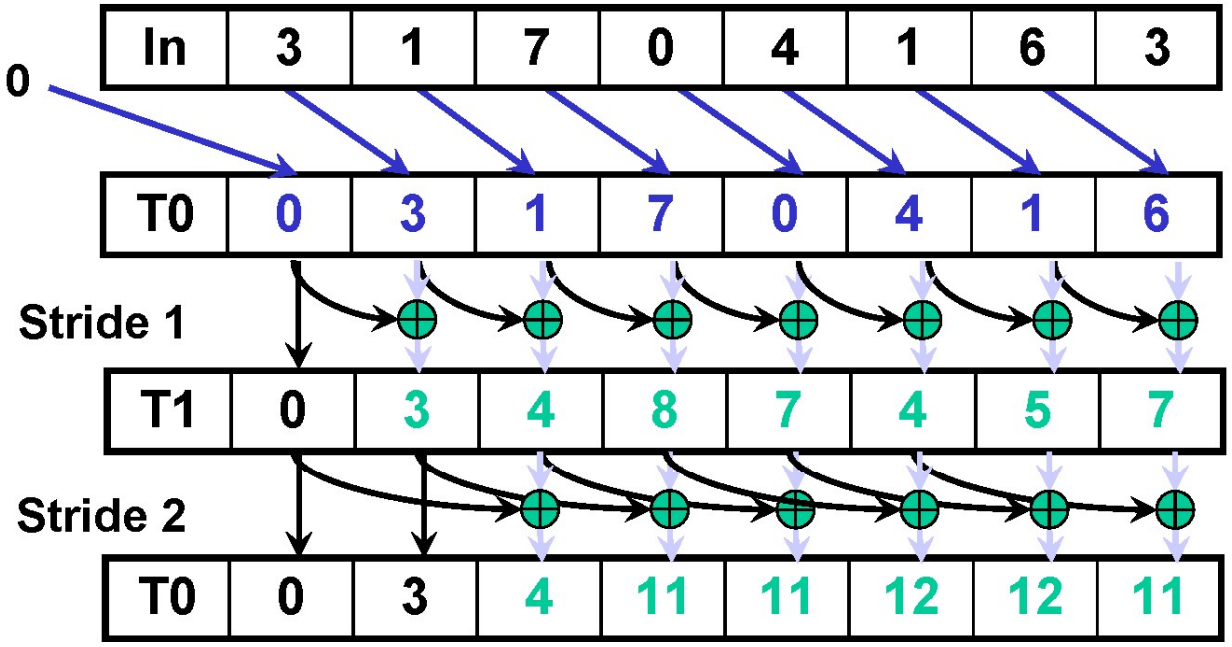



1. (previous slide)
2. Iterate log(n) times: Threads *stride* to *n:* Add pairs of elements *stride* elements apart. Double *stride* at each iteration. (note must double buffer shared mem arrays)

Iteration #1
Stride = 1

- Active threads: *stride* to *n*-1 (*n-stride* threads)
- Thread *j* adds elements *j* and *j-stride* from T0 and writes result into shared memory buffer T1 (ping-pong)

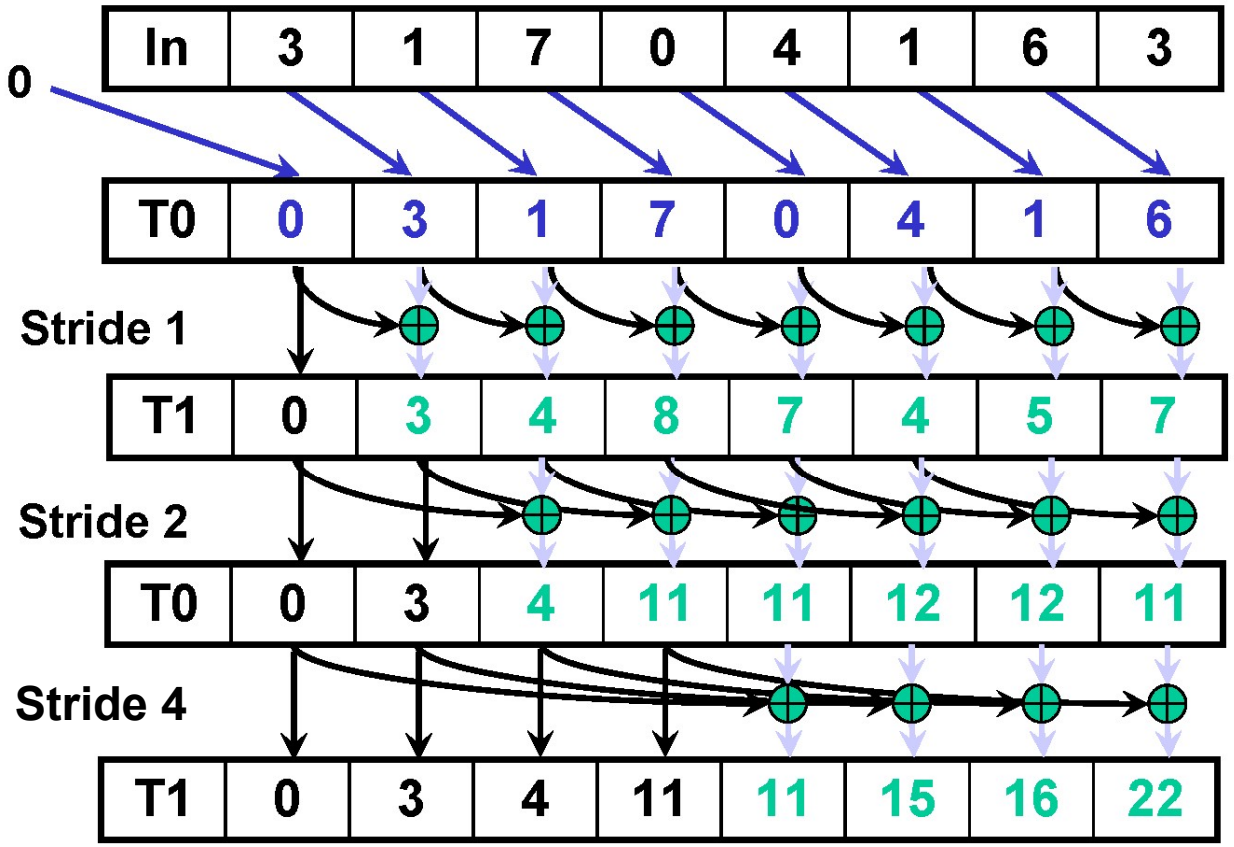
# A First-Attempt Parallel Scan Algorithm




Iteration #2
Stride = 2

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)

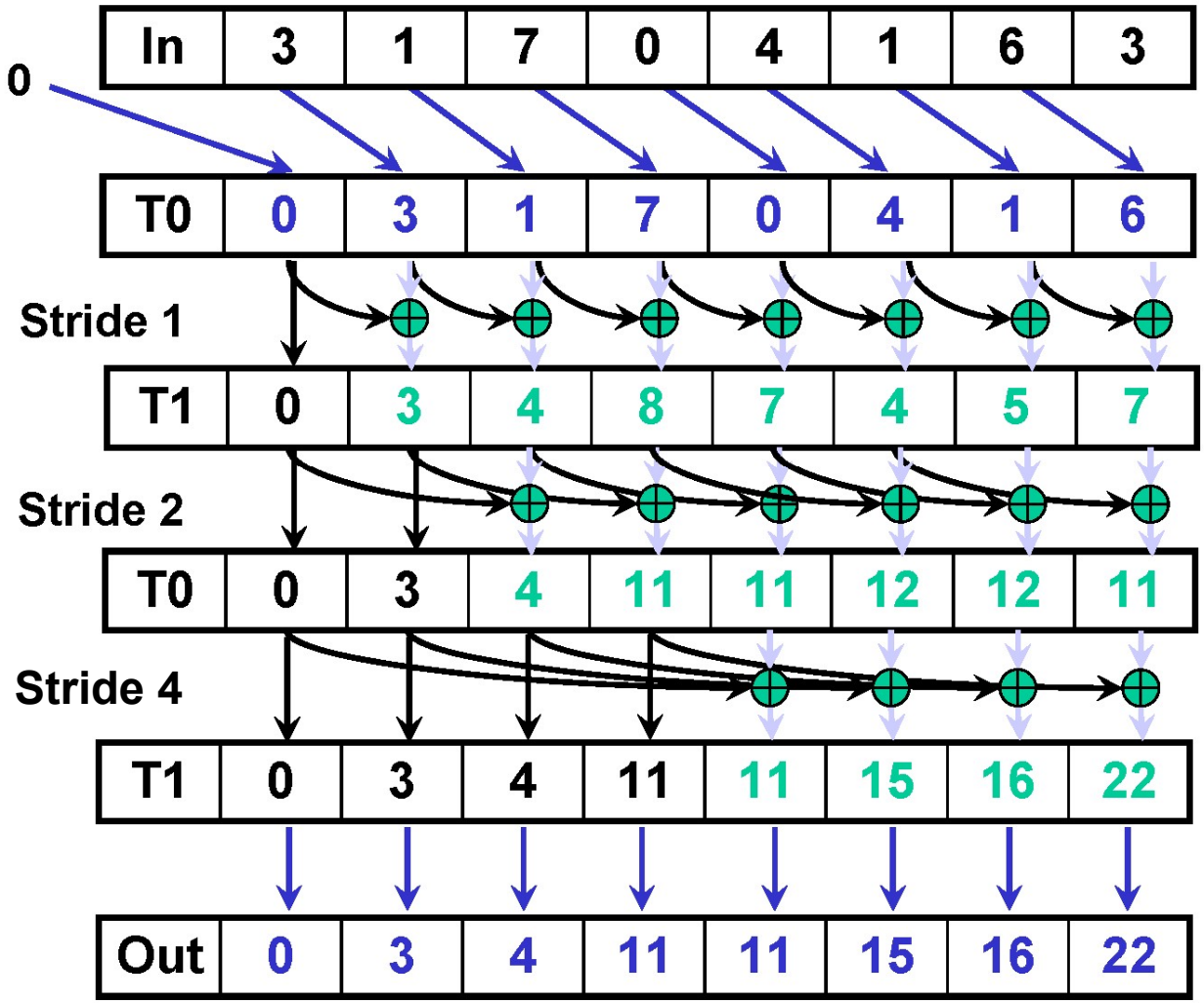
# A First-Attempt Parallel Scan Algorithm




Iteration #3
Stride = 4

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)

# 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.

# Work Efficiency Considerations

- **The first-attempt Scan executes log(n) parallel iterations**
  - Total adds: n * (log(n) – 1) + 1 → O(n*log(n)) work
- **This scan algorithm is not very work efficient**
  - Sequential scan algorithm does *n* adds
  - A factor of log(n) hurts: 20x for 10^6 elements!
- **A parallel algorithm can be slow when execution resources are saturated due to low work efficiency**

Thank you.