King Abdullah University of Science and Technology KAUST

CS 380 - GPU and GPGPU Programming
Lecture 26: GPU Reduction; GPU Prefix Sum (Pt. 1)

Markus Hadwiger, KAUST

Reading Assignment #14 (until Dec 4)

Read (required):

- Warp Shuffle Functions
  - CUDA Programming Guide 11.8, Appendix B.22
- CUDA Cooperative Groups
  - CUDA Programming Guide 11.8, Appendix C
  - `https://developer.nvidia.com/blog/cooperative-groups/`
- Programming Tensor Cores
  - CUDA Programming Guide 11.8, Appendix B.24 (Warp matrix functions)
  - `https://developer.nvidia.com/blog/programming-tensor-cores-cuda-9/`

Read (optional):

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

Next Lectures

Lecture 27:	Wed, Nov 30 (last regular lecture)
Quiz #3:	Wed, Dec 7 (regular time)
Presentations:	Mon, Dec 12 16:00?

Quiz #3: Dec 7

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

GPU Reduction

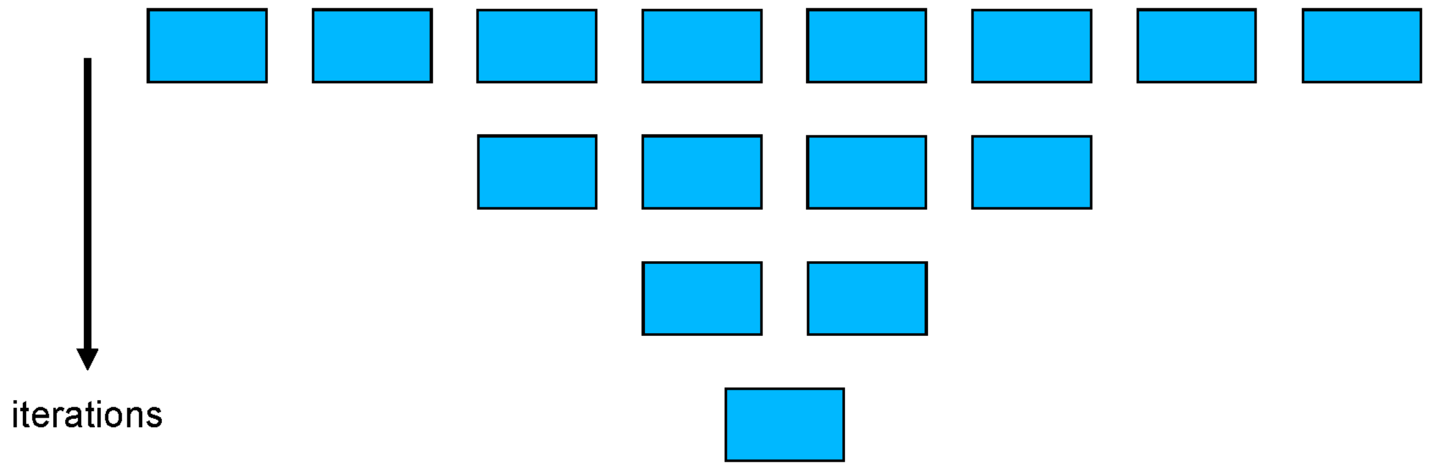
- Parallel reduction is a basic parallel programming primitive; see reduction operation on a stream, e.g., in Brook for GPUs

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**



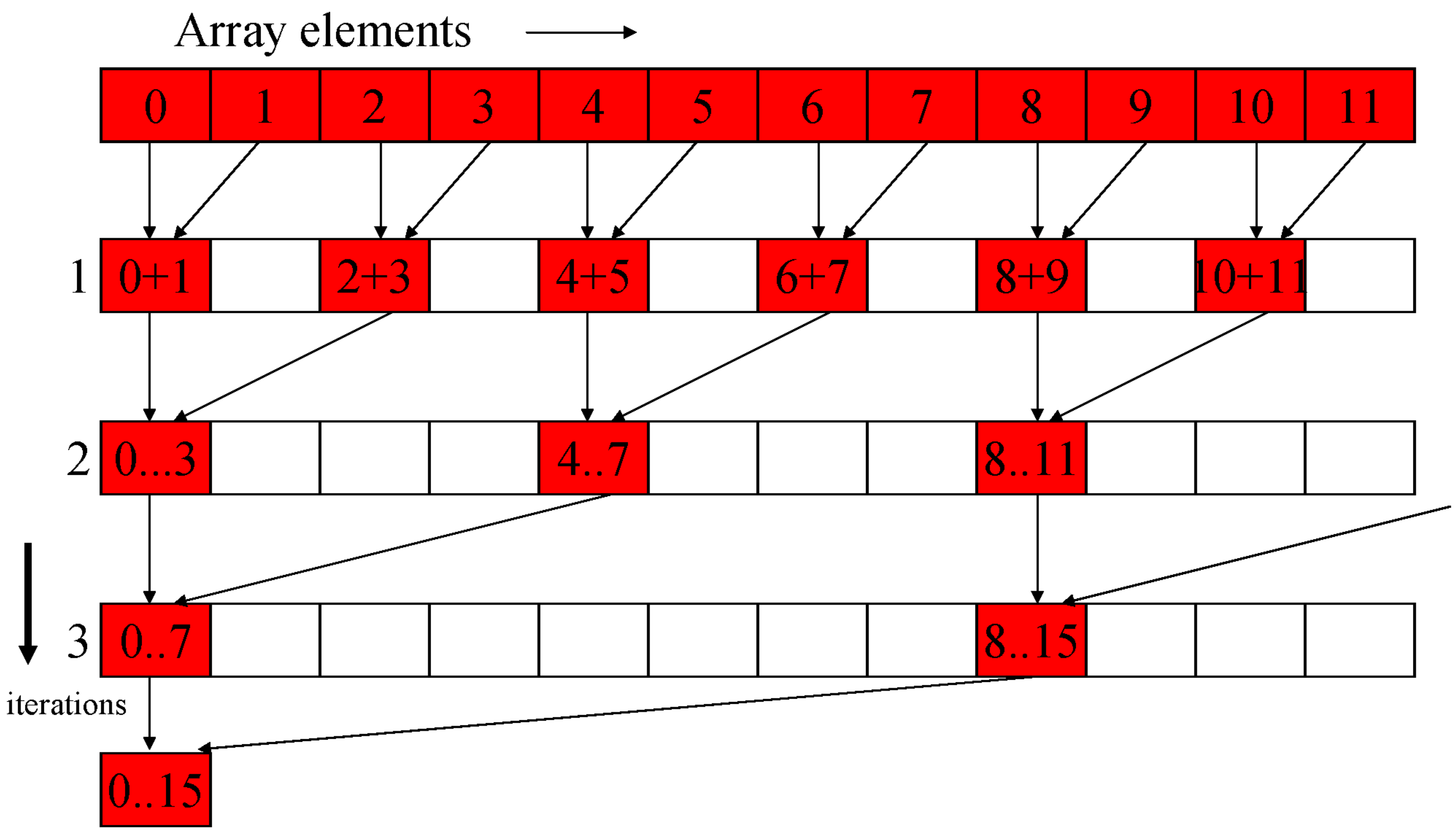
Helpful fact for counting nodes of full binary trees:
If there are N leaf nodes, there will be N-1 non-leaf nodes

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

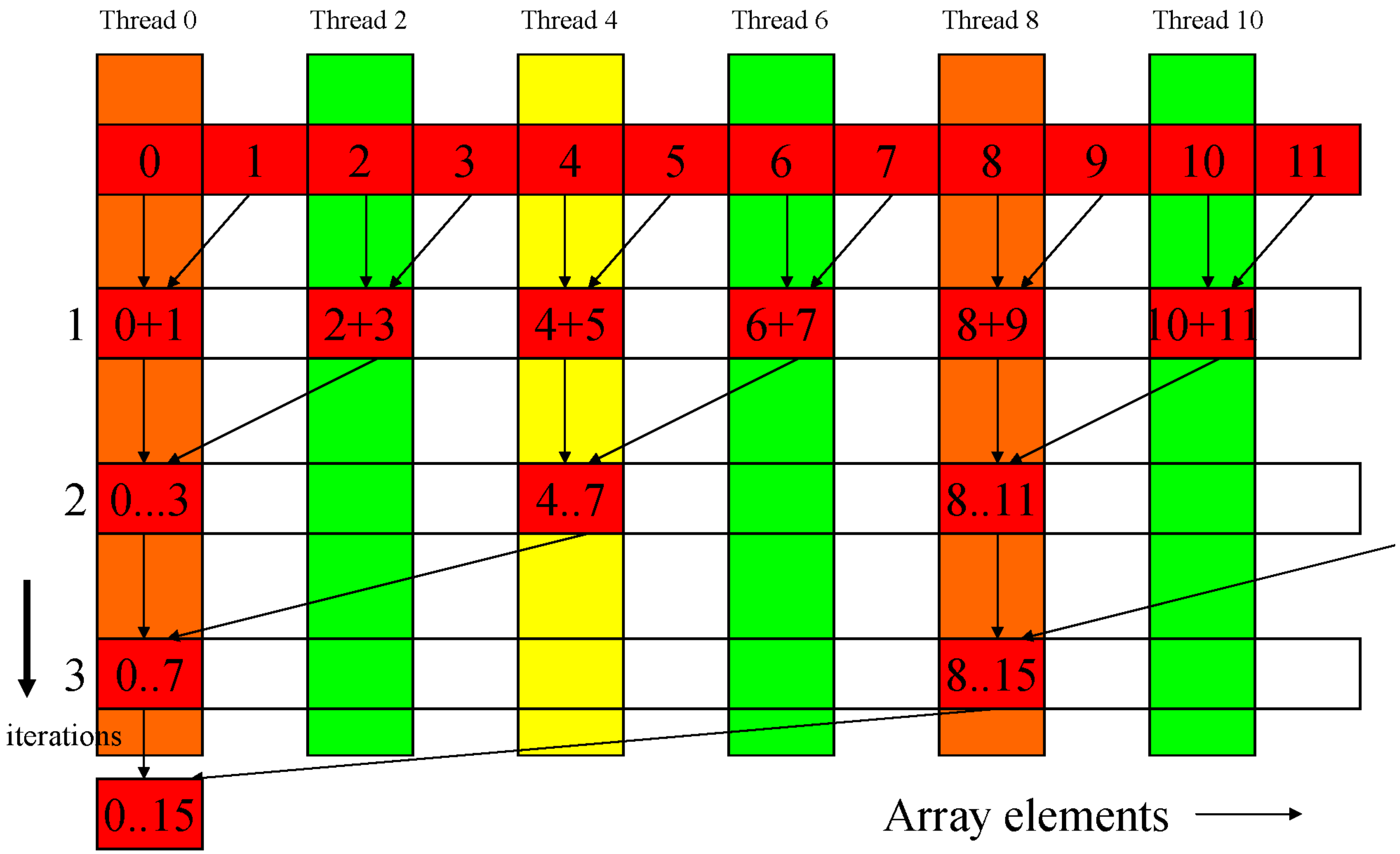
- **Assume we have already loaded array into**

```
__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 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

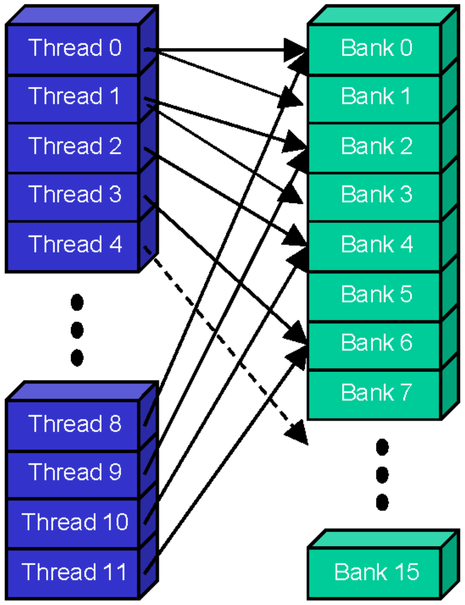
BAD: Bank conflicts due to stride

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];
```

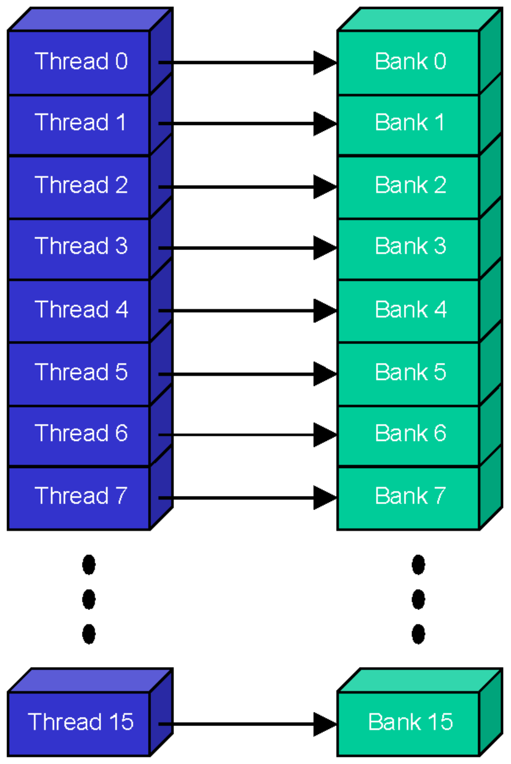


- **This makes sense for traditional CPU threads, locality in cache line usage and reduced sharing traffic.**
  - Not in shared memory usage where there is no cache line effects but banking effects

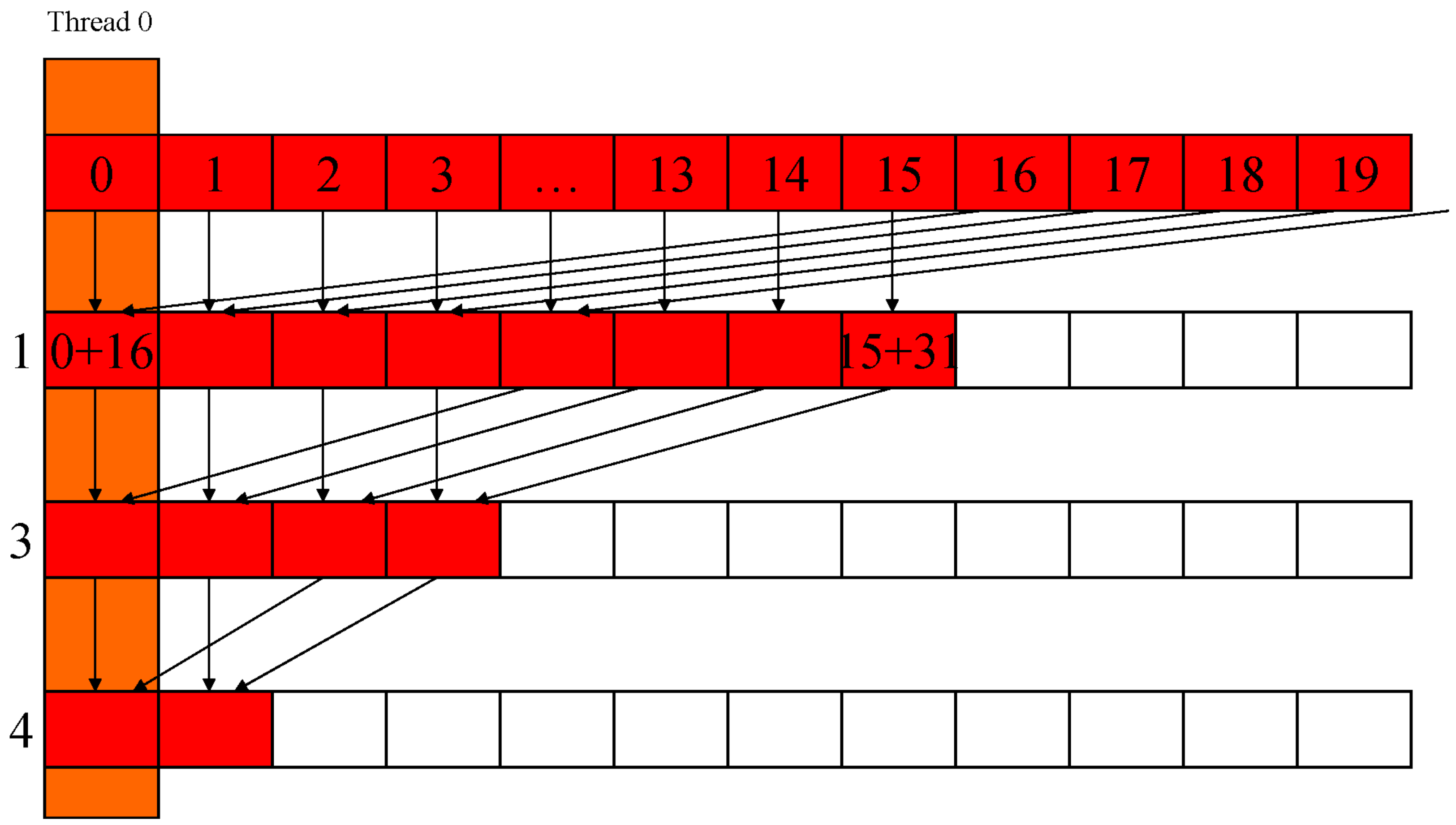
A Better Array Access Pattern

- 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**

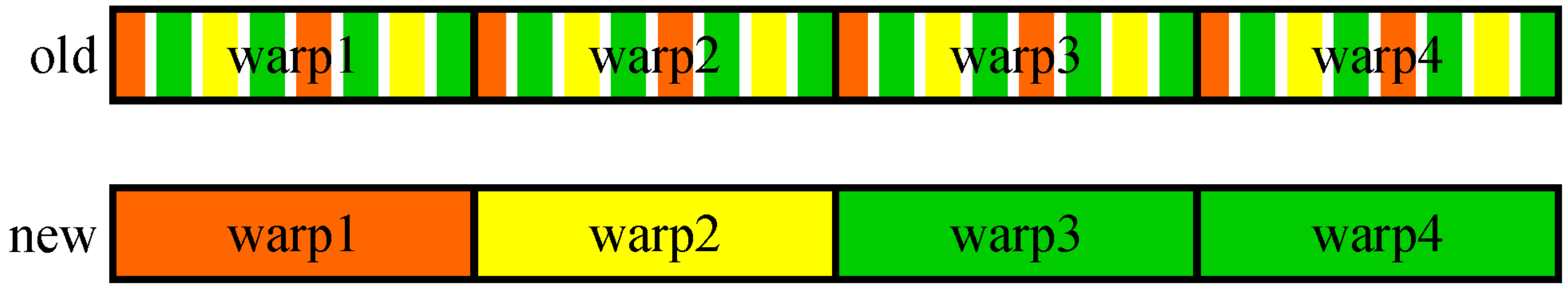
```
__shared__ float partialSum[];

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
    __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.

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

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();
```

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) {
      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];
}
```

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

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];
}
```

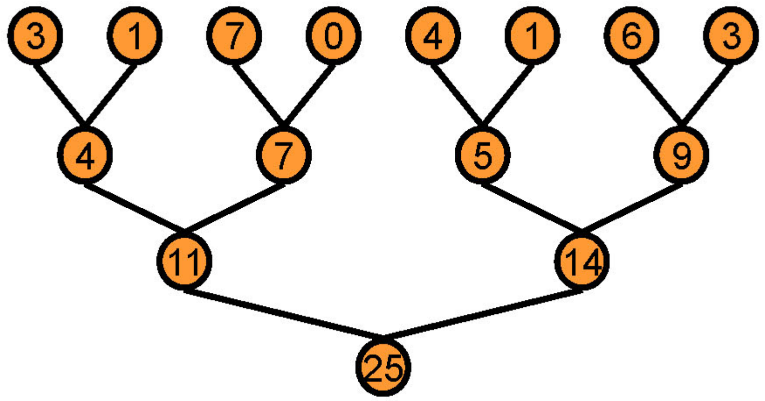
Invoking Template Kernels

- 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?**

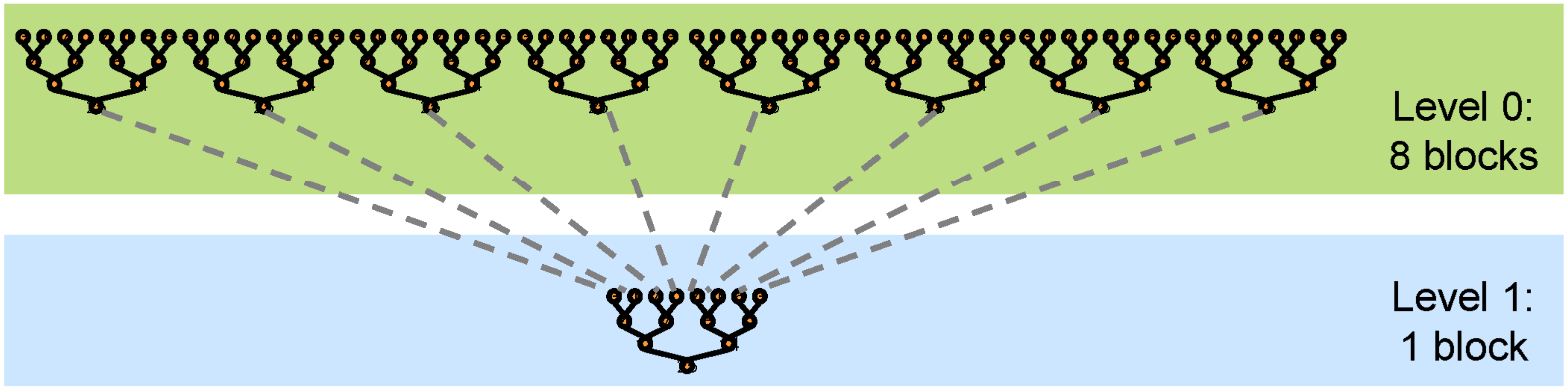
Problem: Global Synchronization

- 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

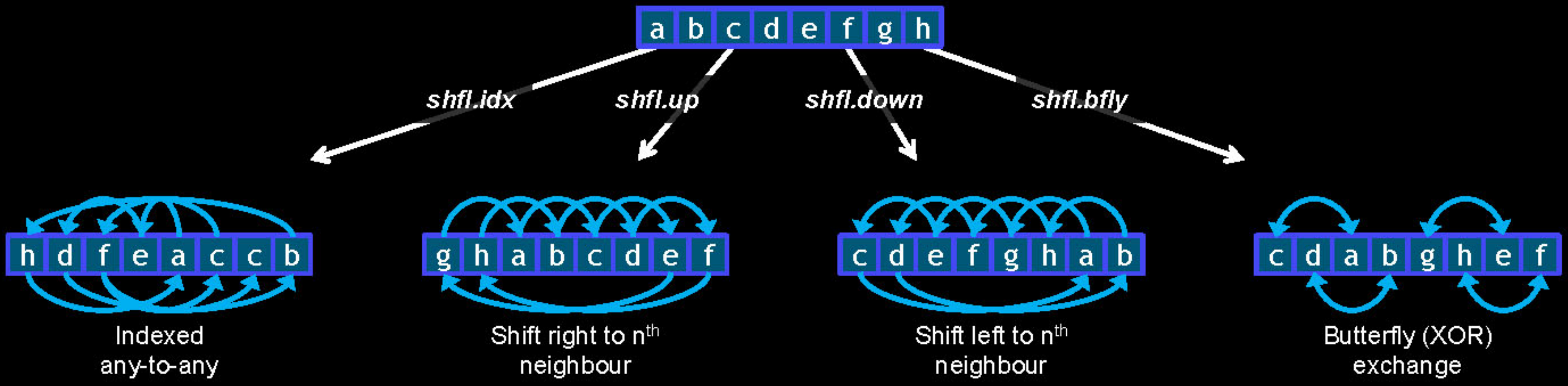
```
v += shmem[tid+16]; __syncwarp();
shmem[tid] = v;     __syncwarp();
v += shmem[tid+8];  __syncwarp();
shmem[tid] = v;     __syncwarp();
v += shmem[tid+4];  __syncwarp();
shmem[tid] = v;     __syncwarp();
v += shmem[tid+2];  __syncwarp();
shmem[tid] = v;     __syncwarp();
v += shmem[tid+1];  __syncwarp();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

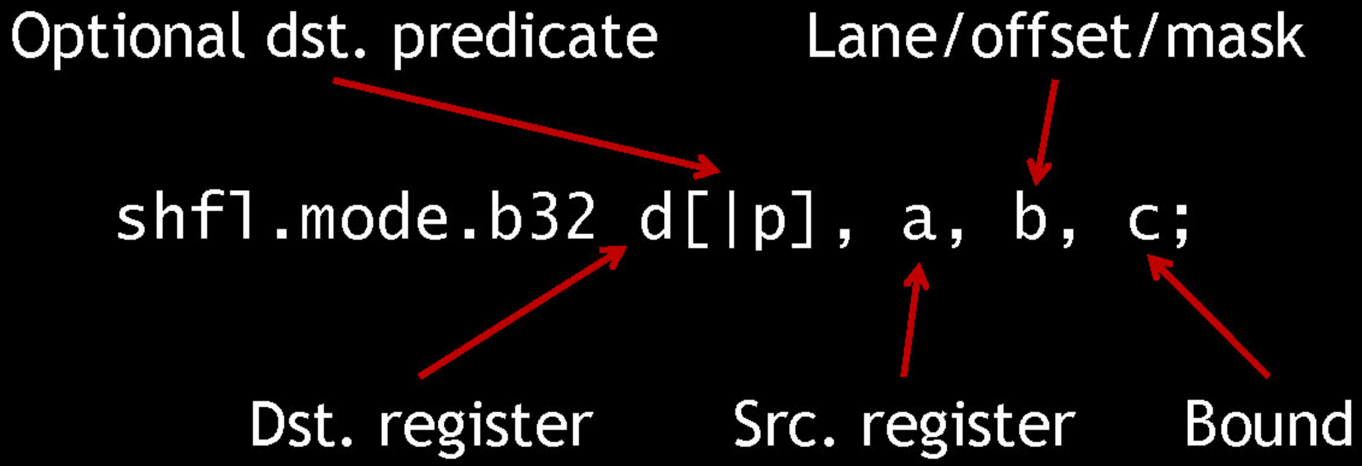
- 4 variants (idx, up, down, bfly):



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

GPU TECHNOLOGY CONFERENCE

Instruction (PTX)



```
shfl.mode.b32 d[|p], a, b, c;
```

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

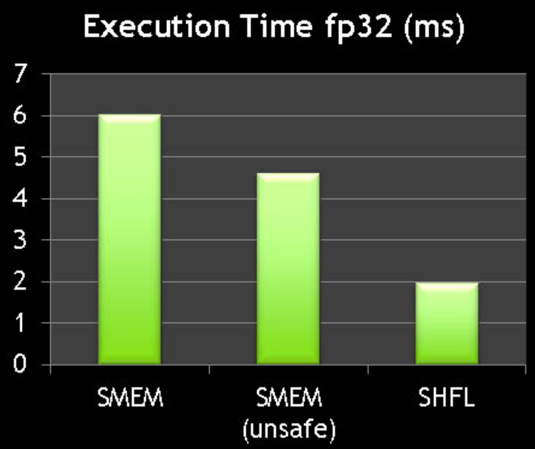
GPU TECHNOLOGY CONFERENCE

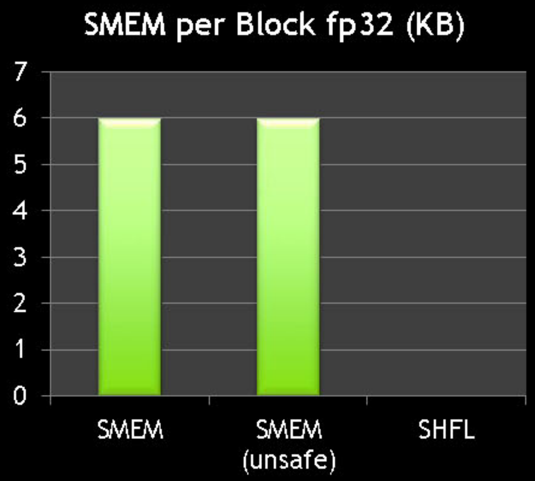
Reduce

- 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 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 $\oplus$ 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 $\oplus$ 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**

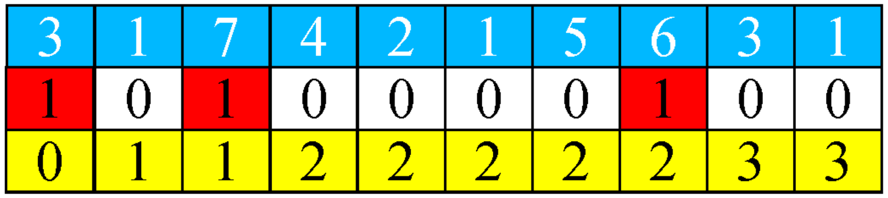
3	1	7	4	2	1	5	6	3	1
1	0	1	0	0	0	0	1	0	0

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

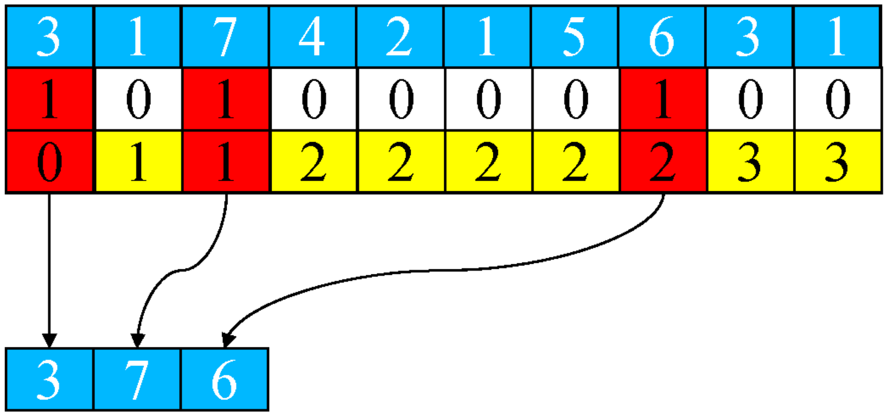
3	7	6

Using Prefix Sum

- **Calculate prefix sum on index array**



- **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:**

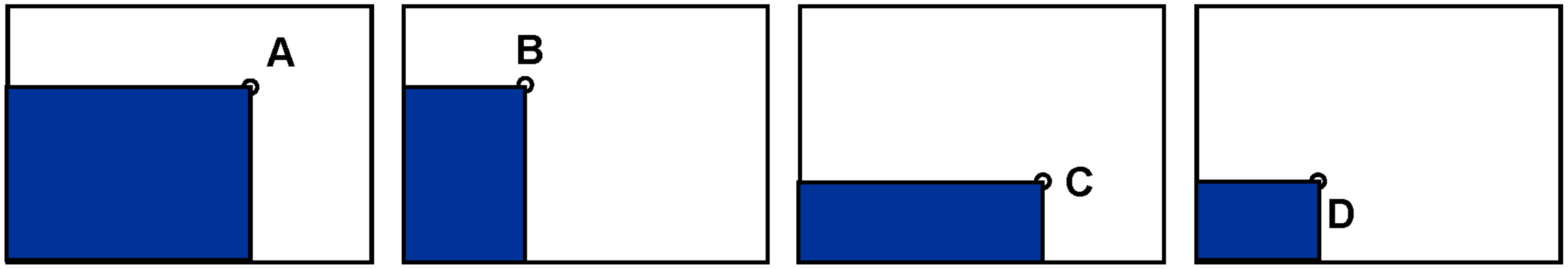
```
hrange = pref[B] - pref[A];
```

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

Prefix Sum Application
- Summed Area Tables -

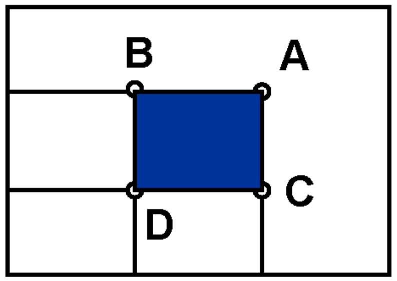
Summed Area Tables

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



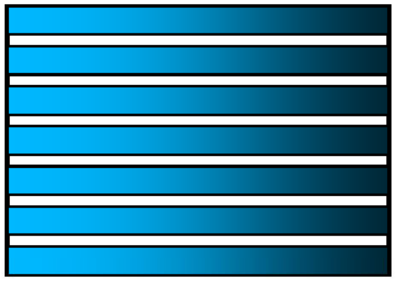
- Many bits per texel (sum !)
- Evaluation of 2D integrals in constant time!

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

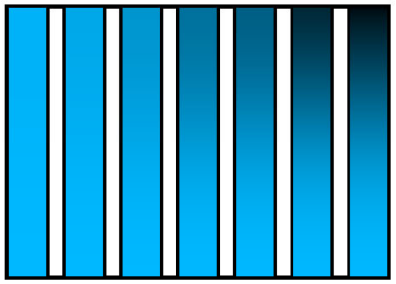


Summed Area Table with Prefix Sums

- 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.

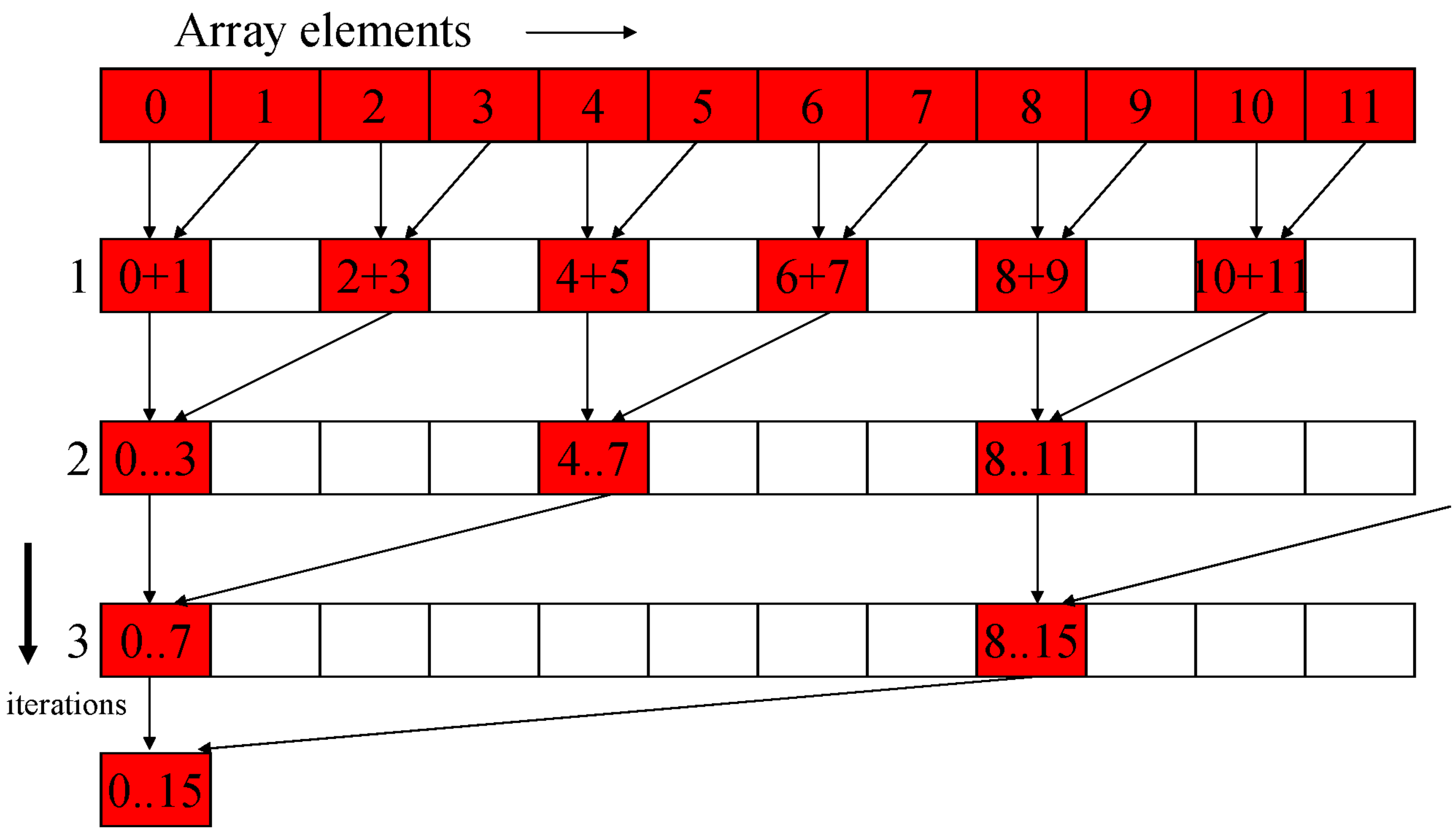
We call an algorithm work-efficient (or just efficient) if it performs the same amount of work, to within a constant factor, as the fastest known sequential algorithm.

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.

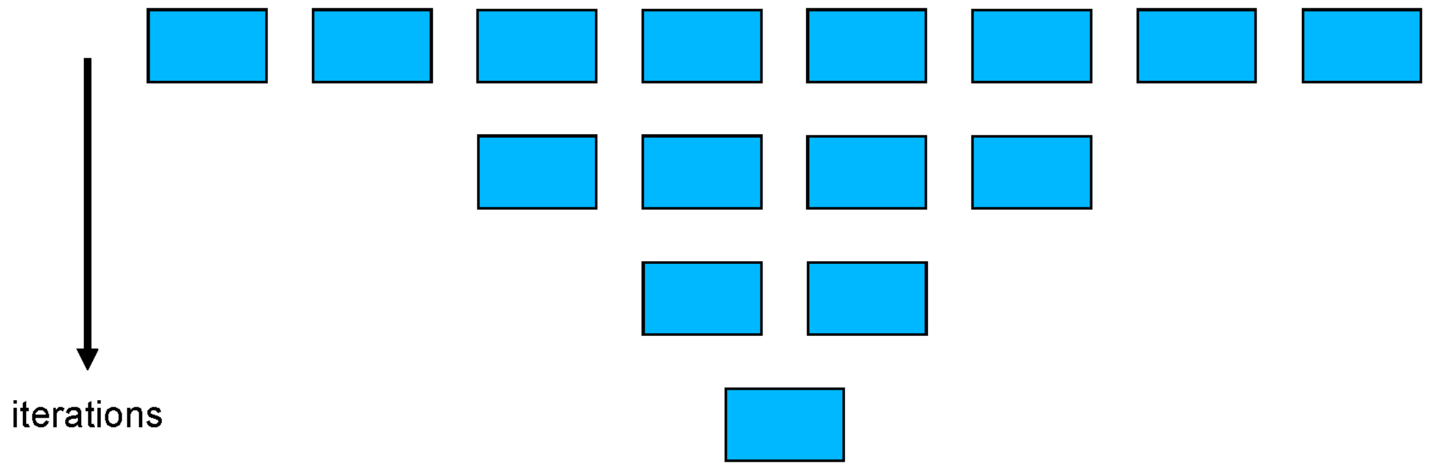
The first algorithm is better than the second - even though it is slower - because its work, or cost, is smaller. Of course, given two parallel algorithms that perform the same amount of work, the faster one is generally better.

Vector Reduction



Typical Parallel Programming Pattern

- **log(n) steps**

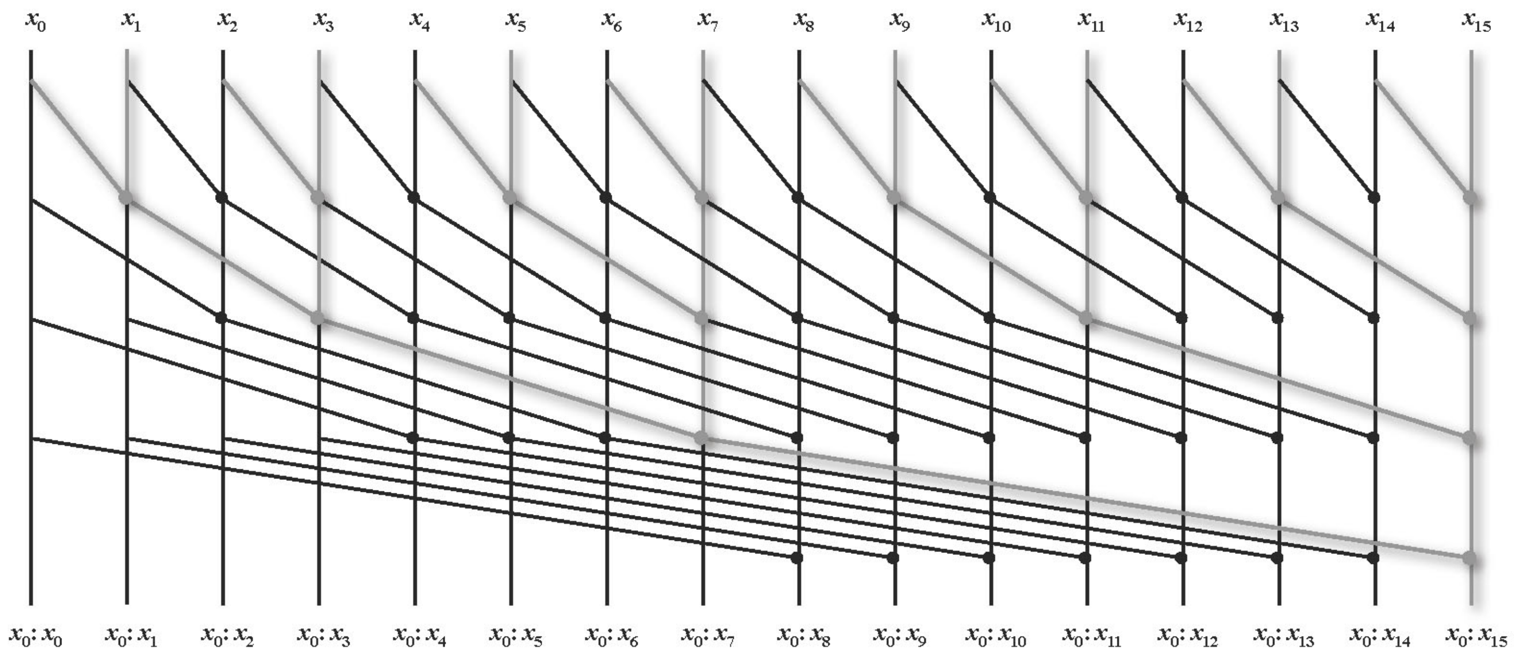


Helpful fact for counting nodes of full binary trees:
If there are N leaf nodes, there will be N-1 non-leaf nodes

Courtesy John Owens

Kogge-Stone Scan

Circuit family

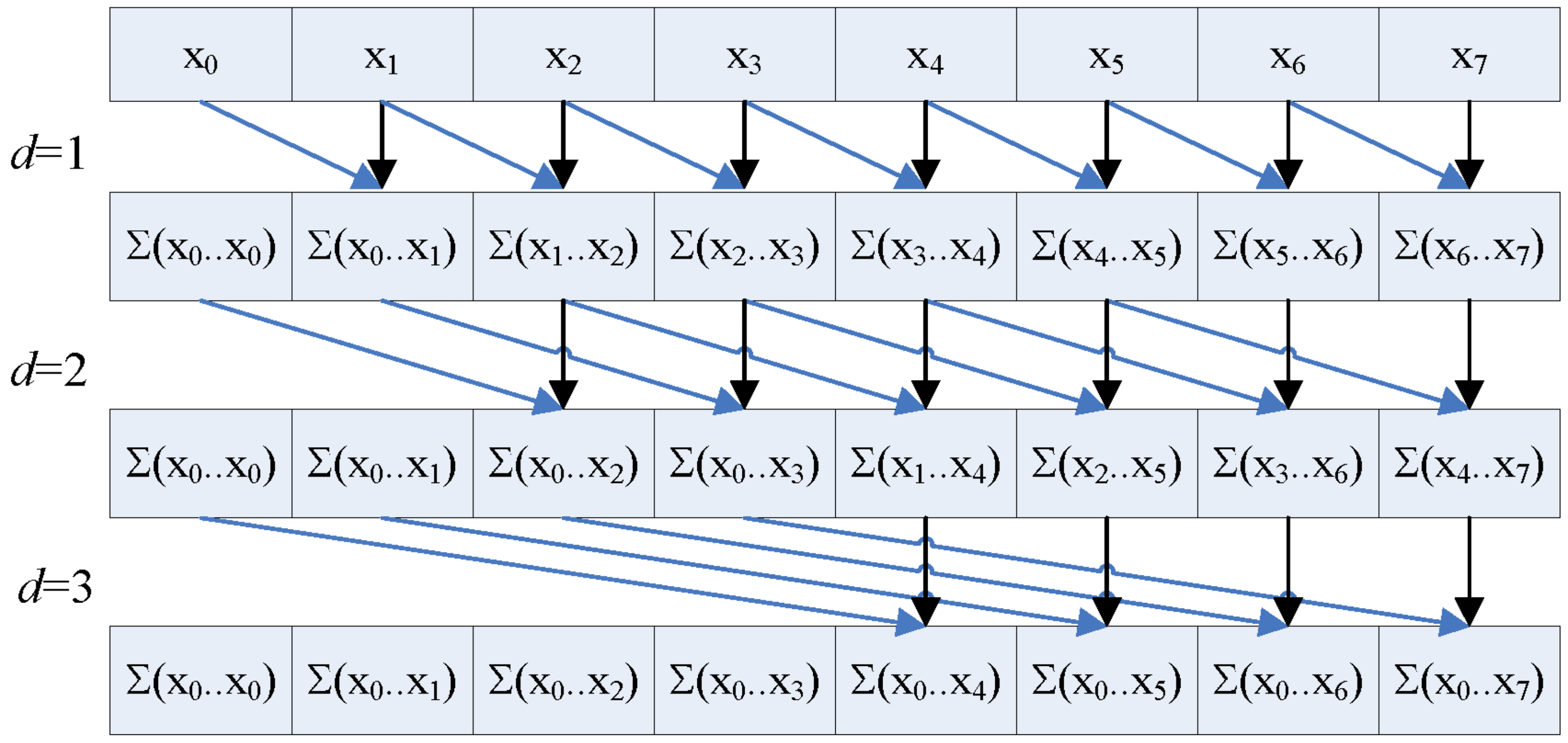


A Parallel Algorithm for the Efficient Solution of a General Class of Recurrence Equations, Kogge and Stone, 1973

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

$O(n \log n)$ Scan

Courtesy John Owens



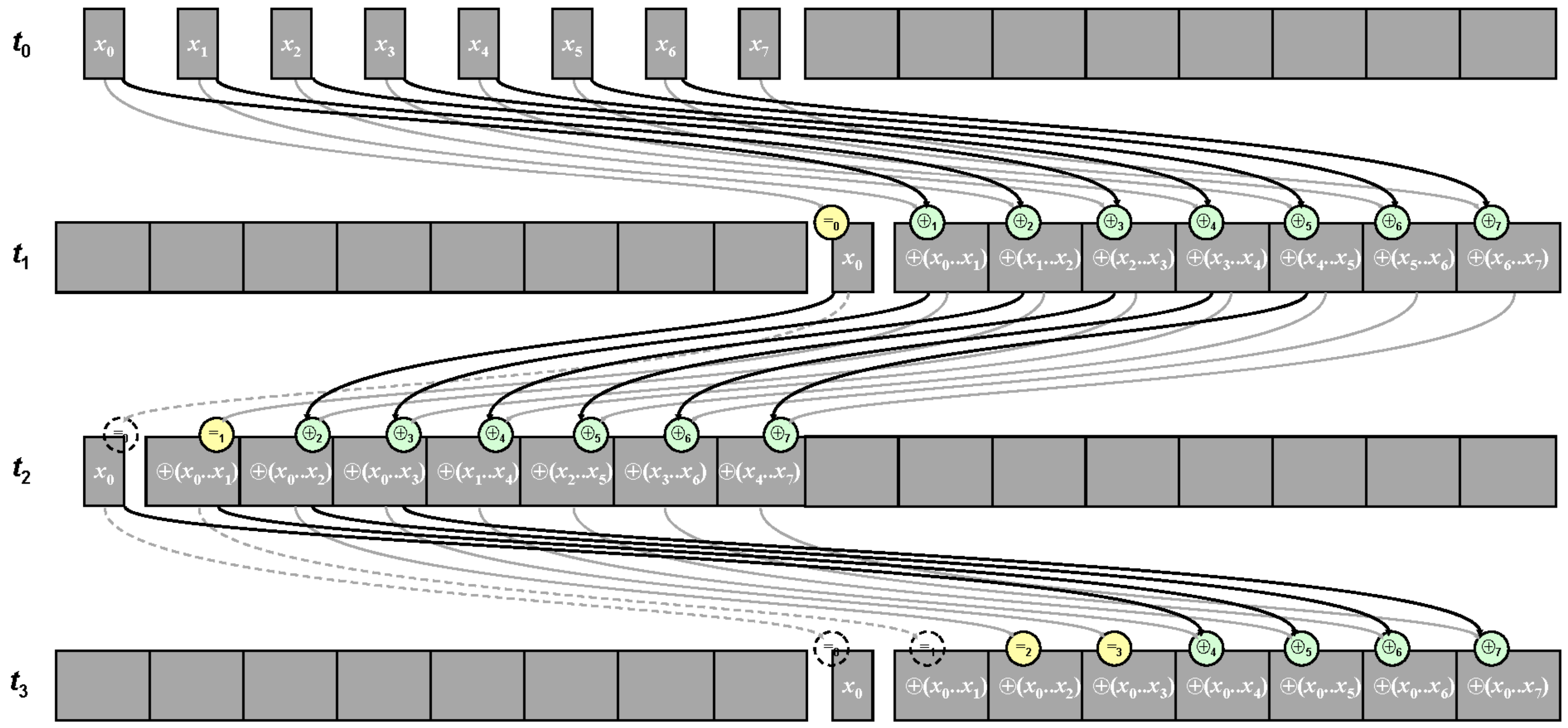
x_0 x_1 x_2 x_3 x_4 x_5 x_6 x_7
d=1
$\Sigma(x_0..x_0)$ $\Sigma(x_0..x_1)$ $\Sigma(x_1..x_2)$ $\Sigma(x_2..x_3)$ $\Sigma(x_3..x_4)$ $\Sigma(x_4..x_5)$ $\Sigma(x_5..x_6)$ $\Sigma(x_6..x_7)$
d=2
$\Sigma(x_0..x_0)$ $\Sigma(x_0..x_1)$ $\Sigma(x_0..x_2)$ $\Sigma(x_0..x_3)$ $\Sigma(x_1..x_4)$ $\Sigma(x_2..x_5)$ $\Sigma(x_3..x_6)$ $\Sigma(x_4..x_7)$
d=3
$\Sigma(x_0..x_0)$ $\Sigma(x_0..x_1)$ $\Sigma(x_0..x_2)$ $\Sigma(x_0..x_3)$ $\Sigma(x_0..x_4)$ $\Sigma(x_0..x_5)$ $\Sigma(x_0..x_6)$ $\Sigma(x_0..x_7)$

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

Courtesy John Owens

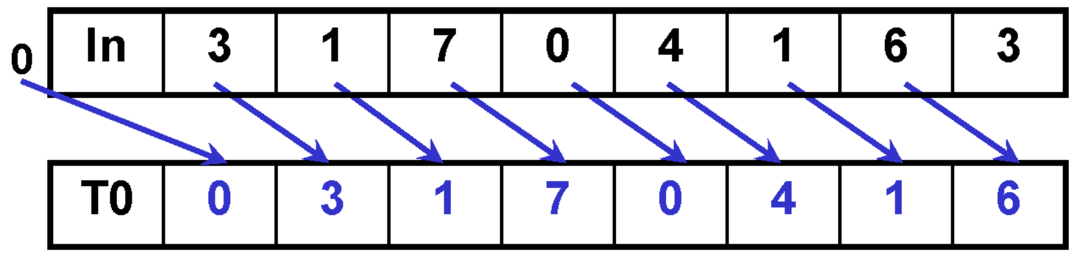
Hillis-Steele Scan Implementation

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



t_0
x_0 x_1 x_2 x_3 x_4 x_5 x_6 x_7
t_1
$=_0$ $\oplus_1$ $\oplus_2$ $\oplus_3$ $\oplus_4$ $\oplus_5$ $\oplus_6$ $\oplus_7$
x_0 $\oplus(x_0..x_1)$ $\oplus(x_1..x_2)$ $\oplus(x_2..x_3)$ $\oplus(x_3..x_4)$ $\oplus(x_4..x_5)$ $\oplus(x_5..x_6)$ $\oplus(x_6..x_7)$
t_2
$=_0$ $=_1$ $\oplus_2$ $\oplus_3$ $\oplus_4$ $\oplus_5$ $\oplus_6$ $\oplus_7$
x_0 $\oplus(x_0..x_1)$ $\oplus(x_0..x_2)$ $\oplus(x_0..x_3)$ $\oplus(x_1..x_4)$ $\oplus(x_2..x_5)$ $\oplus(x_3..x_6)$ $\oplus(x_4..x_7)$
t_3
$=_0$ $=_1$ $=_2$ $=_3$ $\oplus_4$ $\oplus_5$ $\oplus_6$ $\oplus_7$
x_0 $\oplus(x_0..x_1)$ $\oplus(x_0..x_2)$ $\oplus(x_0..x_3)$ $\oplus(x_0..x_4)$ $\oplus(x_0..x_5)$ $\oplus(x_0..x_6)$ $\oplus(x_0..x_7)$

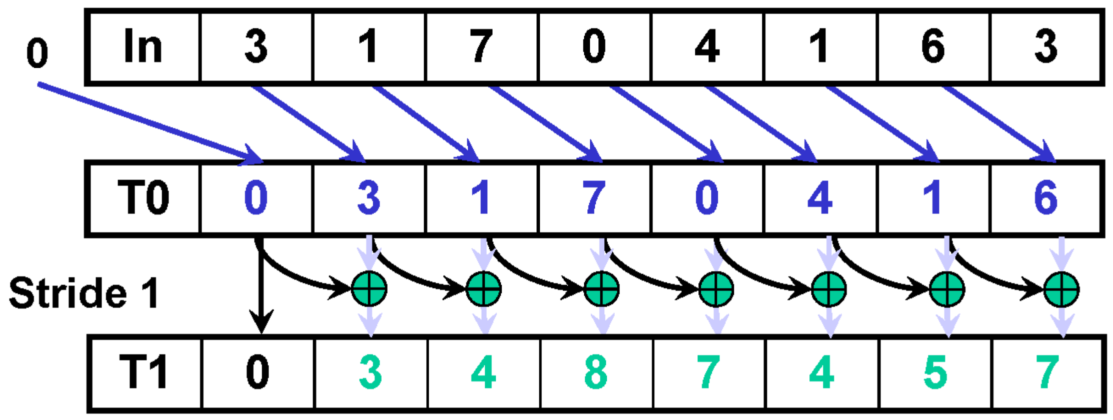
A First-Attempt Parallel Scan Algorithm



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

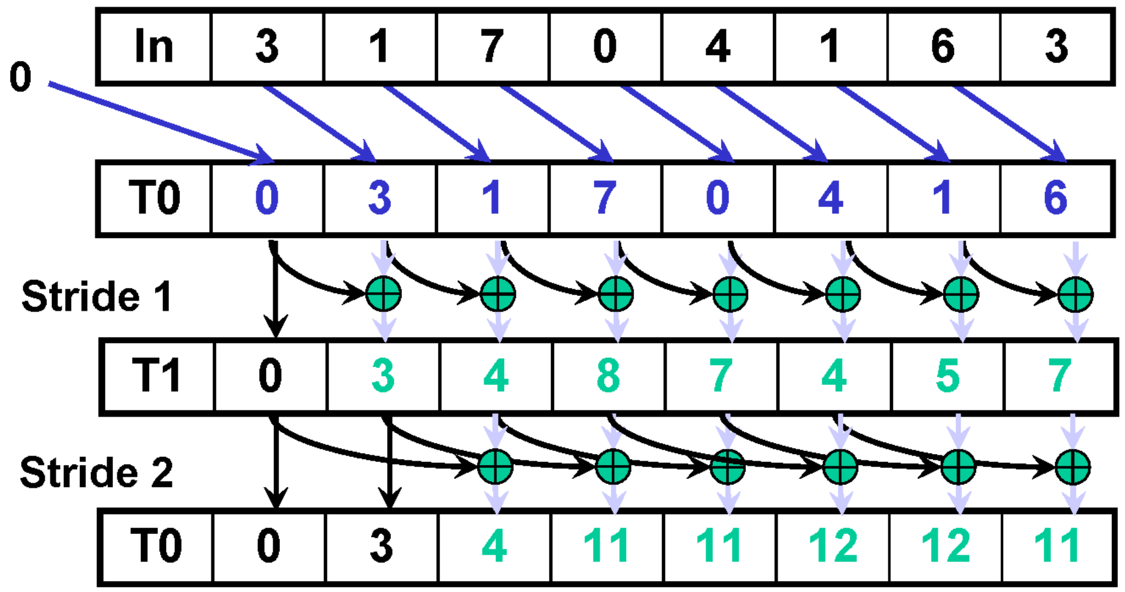


1. (previous slide)
2. Iterate log(n) times: Threads *stride* to *n:* Add pairs of elements *stride* elements apart. Double *stride* at each iteration. (note must double buffer shared mem arrays)

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)

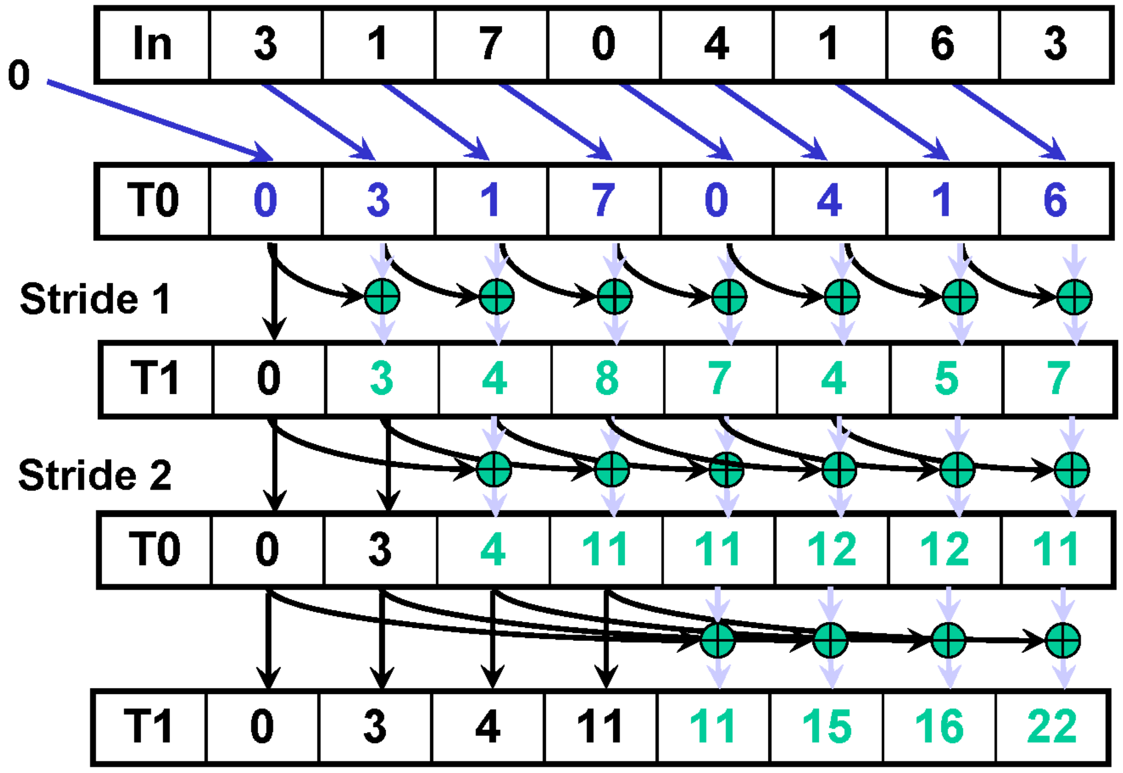
A First-Attempt Parallel Scan Algorithm



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)

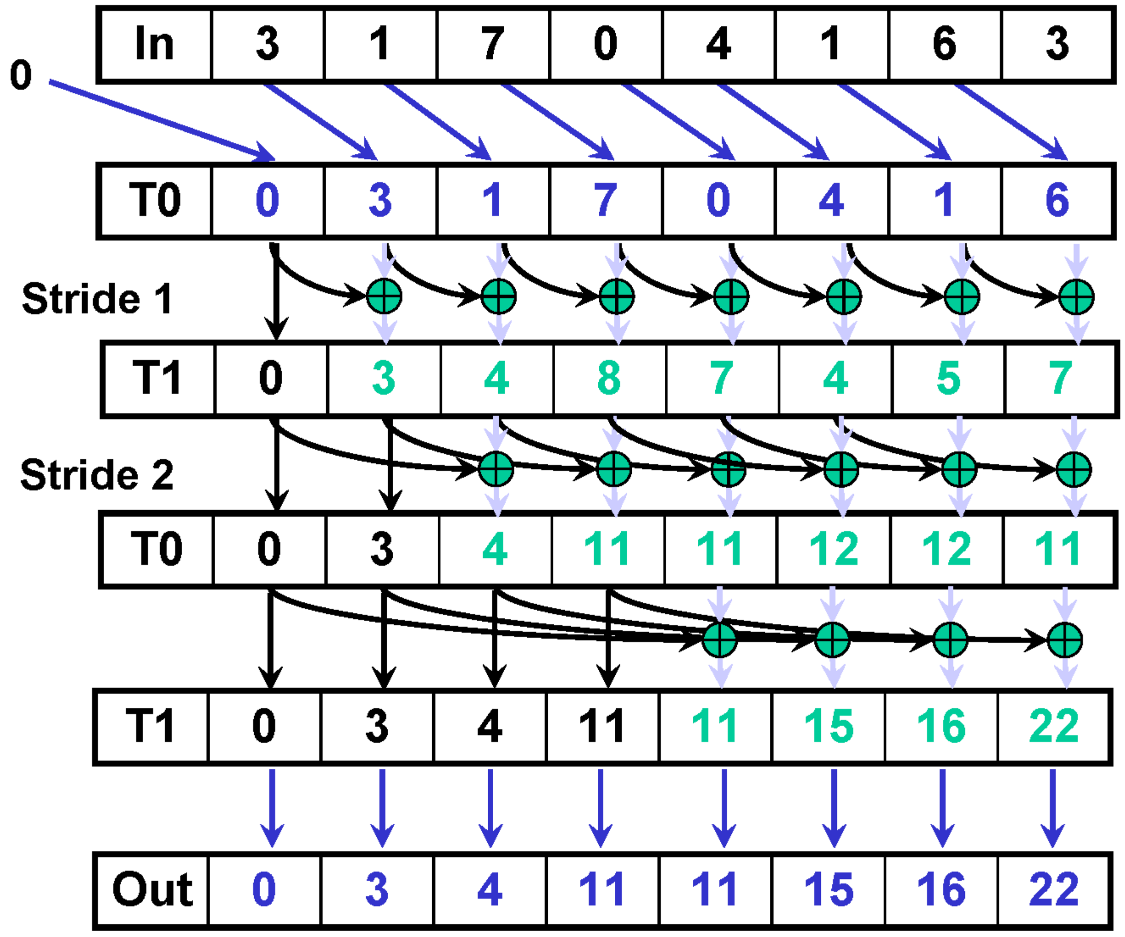
A First-Attempt Parallel Scan Algorithm



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



0
In | 3 | 1 | 7 | 0 | 4 | 1 | 6 | 3
T0 | 0 | 3 | 1 | 7 | 0 | 4 | 1 | 6
Stride 1
T1 | 0 | 3 | 4 | 8 | 7 | 4 | 5 | 7
Stride 2
T0 | 0 | 3 | 4 | 11 | 11 | 12 | 12 | 11
T1 | 0 | 3 | 4 | 11 | 11 | 15 | 16 | 22
Out | 0 | 3 | 4 | 11 | 11 | 15 | 16 | 22

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

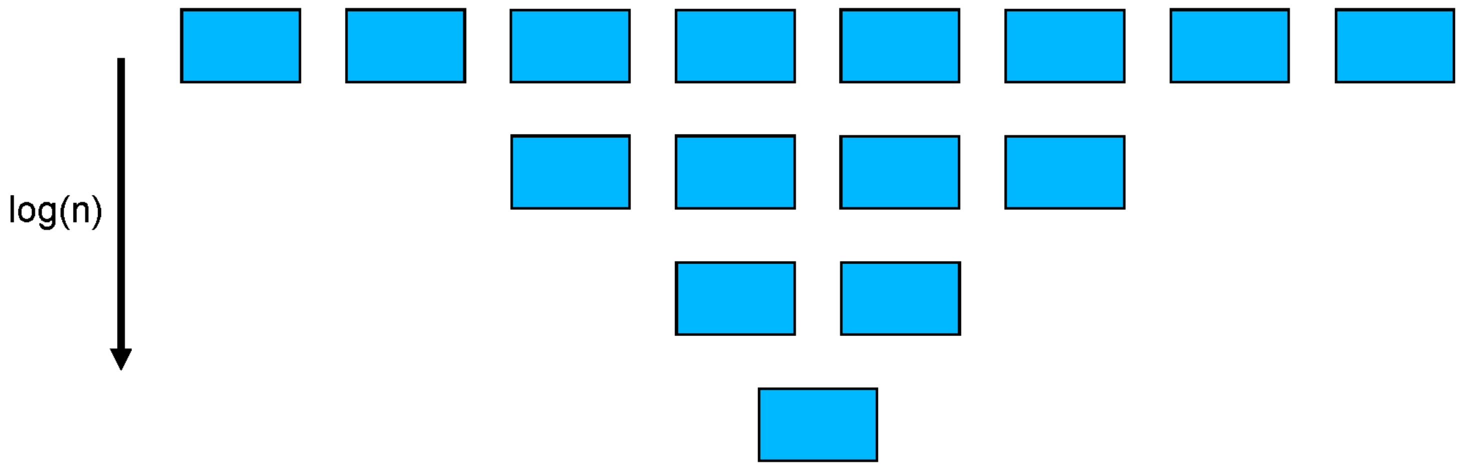
- **The first-attempt Scan executes log(n) parallel iterations**
  - Total adds: n * (log(n) – 1) + 1 → O(n*log(n)) work
- **This scan algorithm is not very work efficient**
  - Sequential scan algorithm does *n* adds
  - A factor of log(n) hurts: 20x for 10^6 elements!
- **A parallel algorithm can be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

Balanced Trees

- **For improving efficiency**
- **A common parallel algorithm pattern:**
  - Build a balanced binary tree on the input data and sweep it to and from the root
  - Tree is not an actual data structure, but a concept to determine what each thread does at each step
- **For scan:**
  - Traverse down from leaves to root building partial sums at internal nodes in the tree
    - Root holds sum of all leaves
  - Traverse back up the tree building the scan from the partial sums

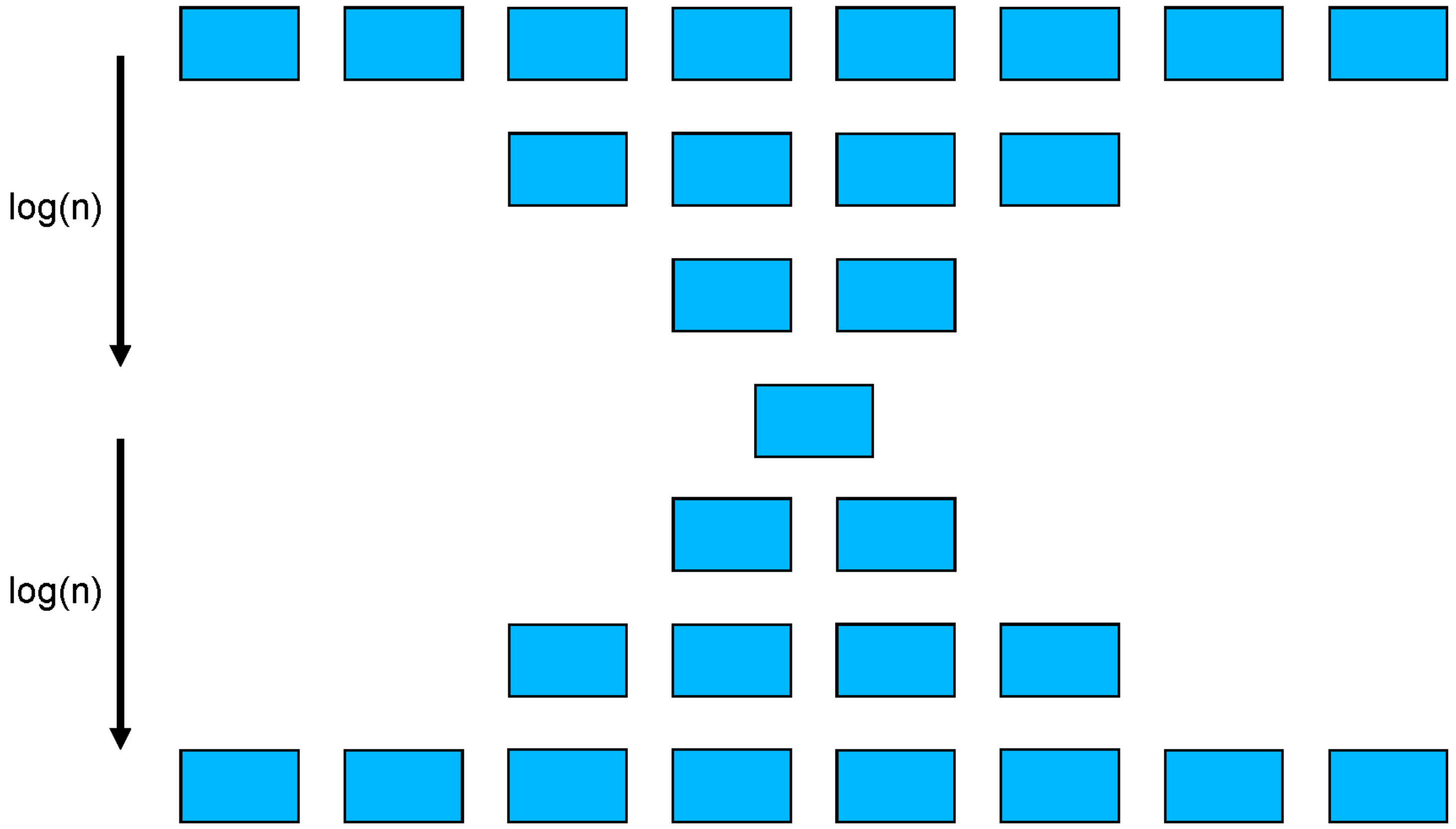
Typical Parallel Programming Pattern

- **2 log(n) steps**



Typical Parallel Programming Pattern

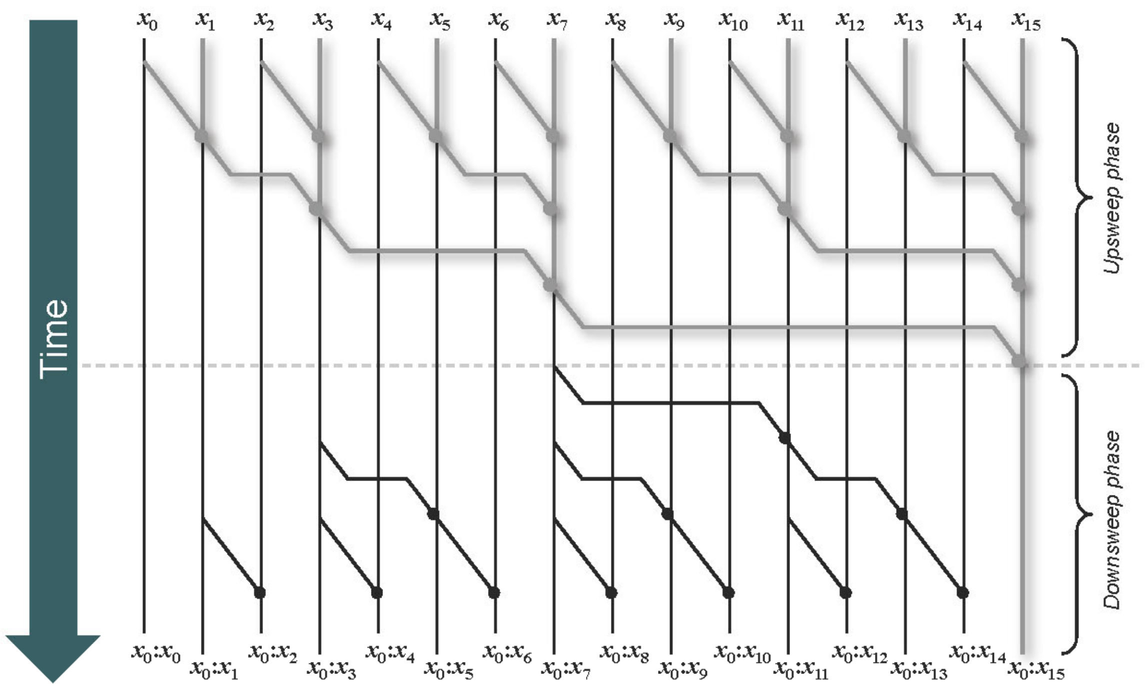
- **2 log(n) steps**



Courtesy John Owens

Brent Kung Scan

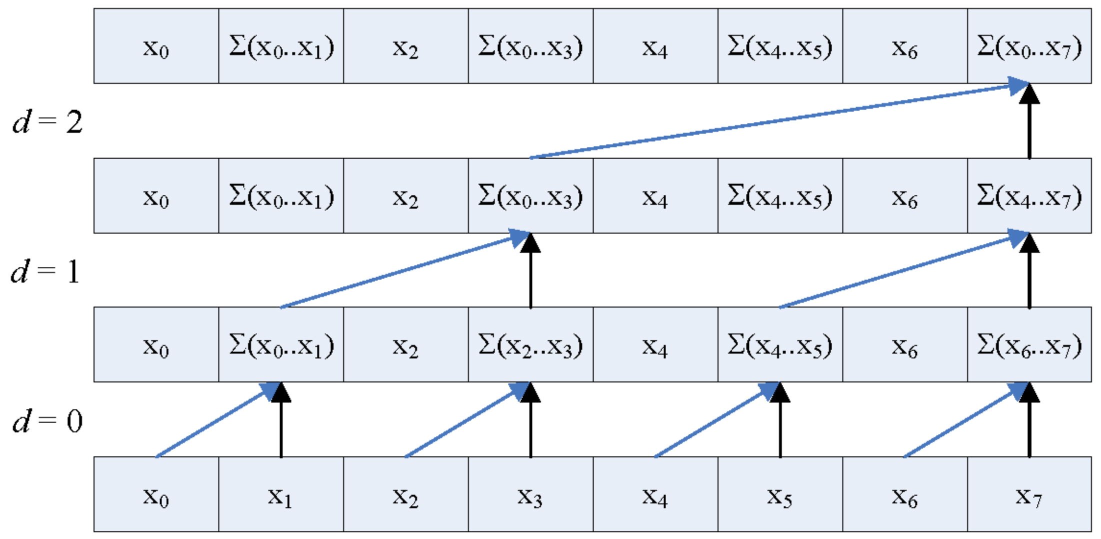
Circuit family

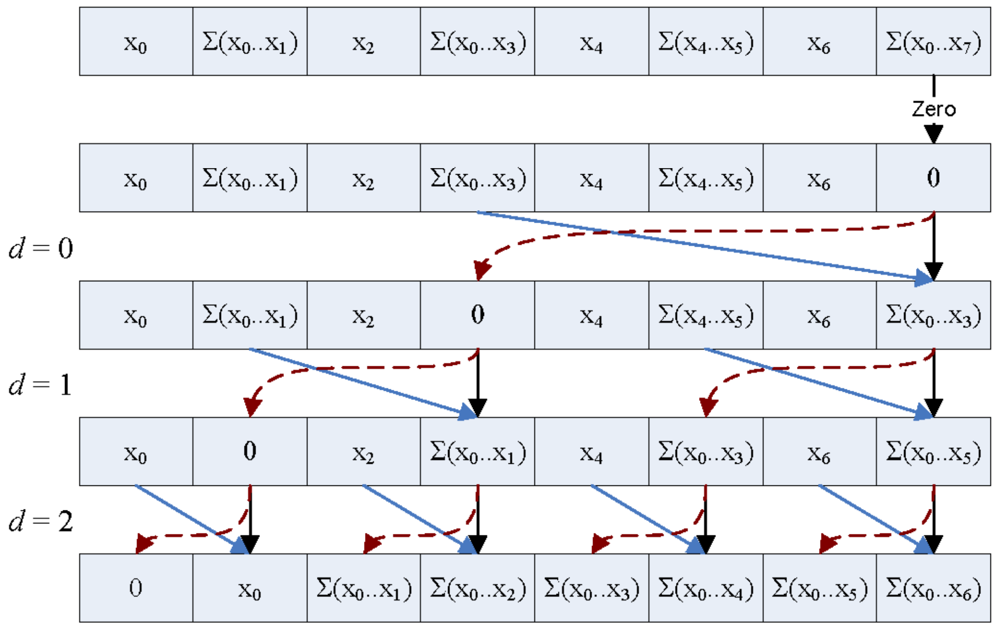


A Regular Layout for Parallel Adders, Brent and Kung, 1982

$O(n)$ Scan [Blelloch]

Courtesy John Owens





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

Thank you.