
Massively Parallel Computing with Cuda

- Control Flow -

Hendrik Lensch
Robert Strzodka

Today

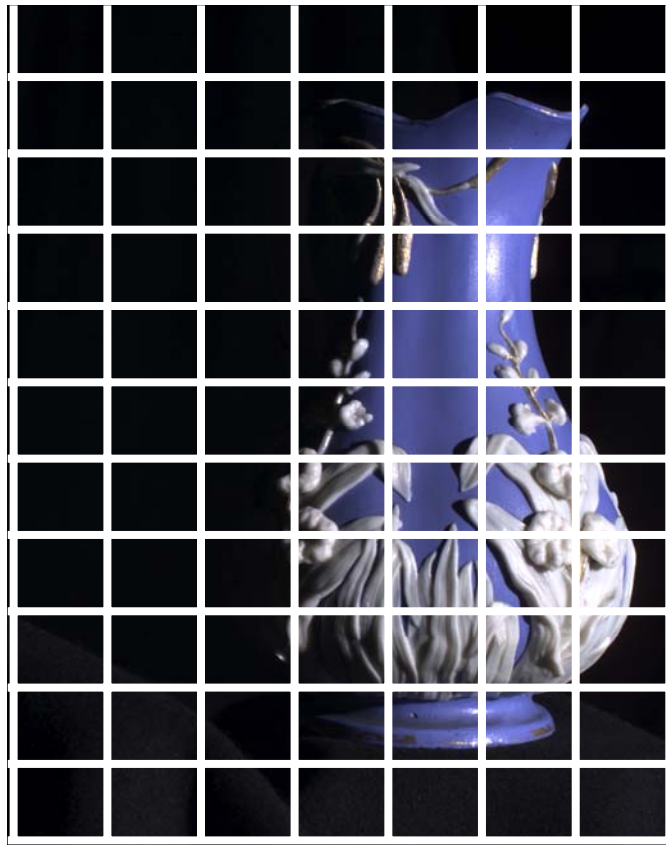
- **Branching**
 - Non-power-of Two
 - Divergence
 - Granularity
- **Algorithms**
 - Reduction Version 2
 - PrefixSums

Programming Models

Data-parallel
Thread-parallel

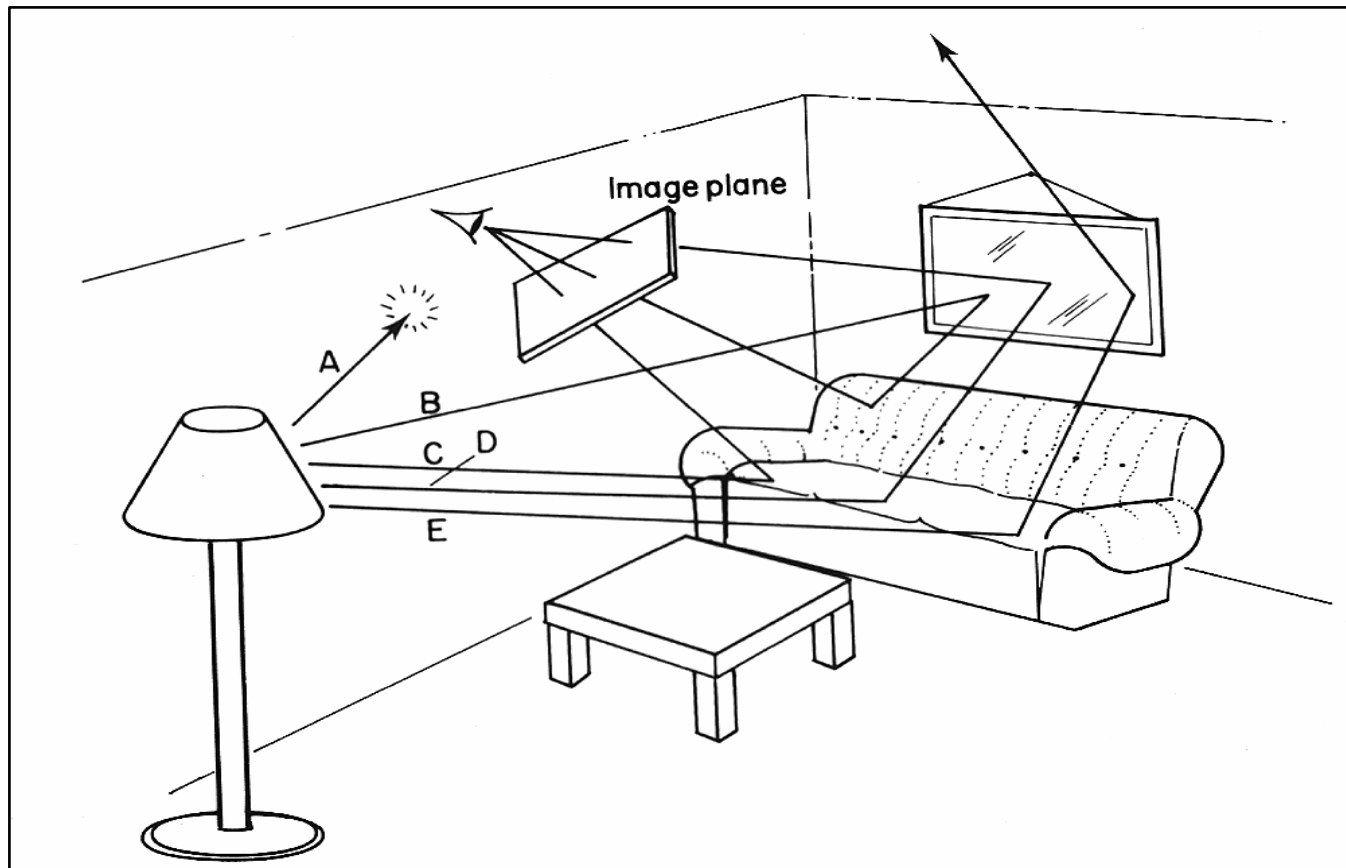
Data-Parallel Approach

- **Each thread is responsible for one data item.**
- **In principle all threads execute the same code.**
- **Example: gamma correction or filtering of an image**



Task-Parallel Programming

- **Possibly independent threads**
- **Each thread could execute his own code / branch**
- **Example: Ray-Tracing, one task per ray**



Task-Parallel Programming

- **Possibly independent threads**
- **Each thread could execute his own code / branch**
- **However, all threads within a warp execute one common instruction at a time**
- **Best performance if all threads within a warp execute the same branch**

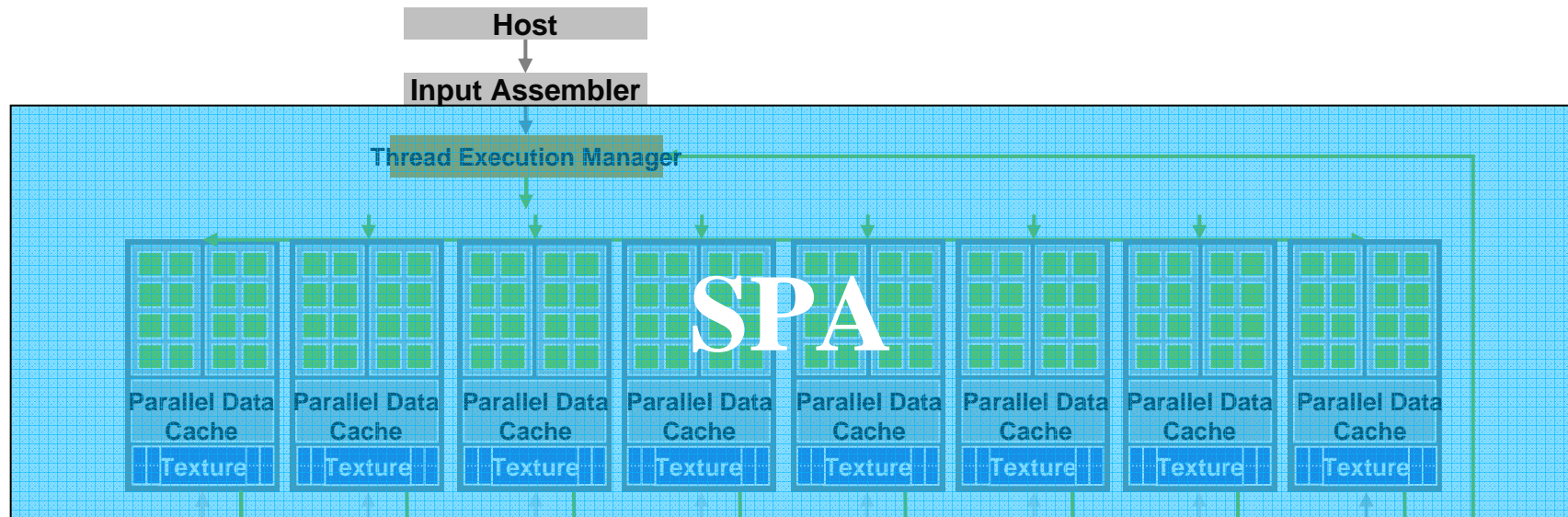
- **For performance reasons**
avoid branching within a warp!

- **Align diverging branches with warp boundaries**

Control Flow with Cuda

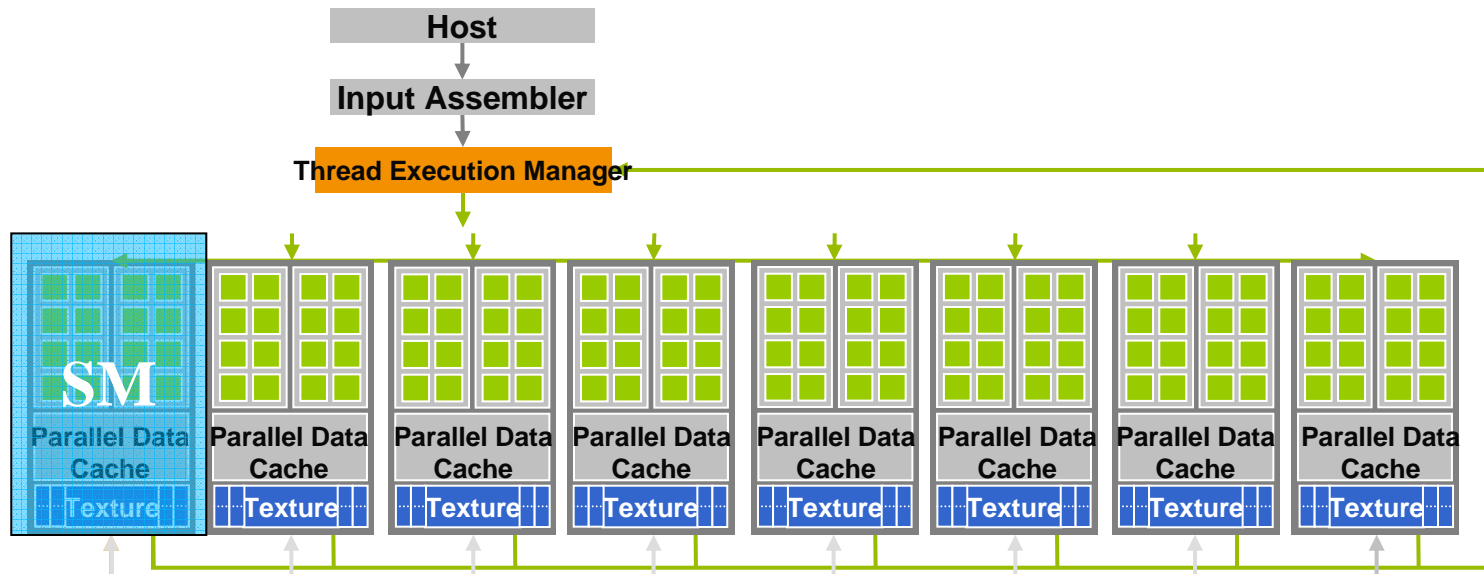
CUDA Processor Terminology

- **SPA**
 - Streaming Processor Array (variable across GeForce 8-series, 8 in GeForce8800)
- **SM**
 - Streaming Multiprocessor (8 SP)
 - Multi-threaded processor core
 - Fundamental processing unit for CUDA thread block
- **SP**
 - Streaming Processor
 - Scalar ALU for a single CUDA thread



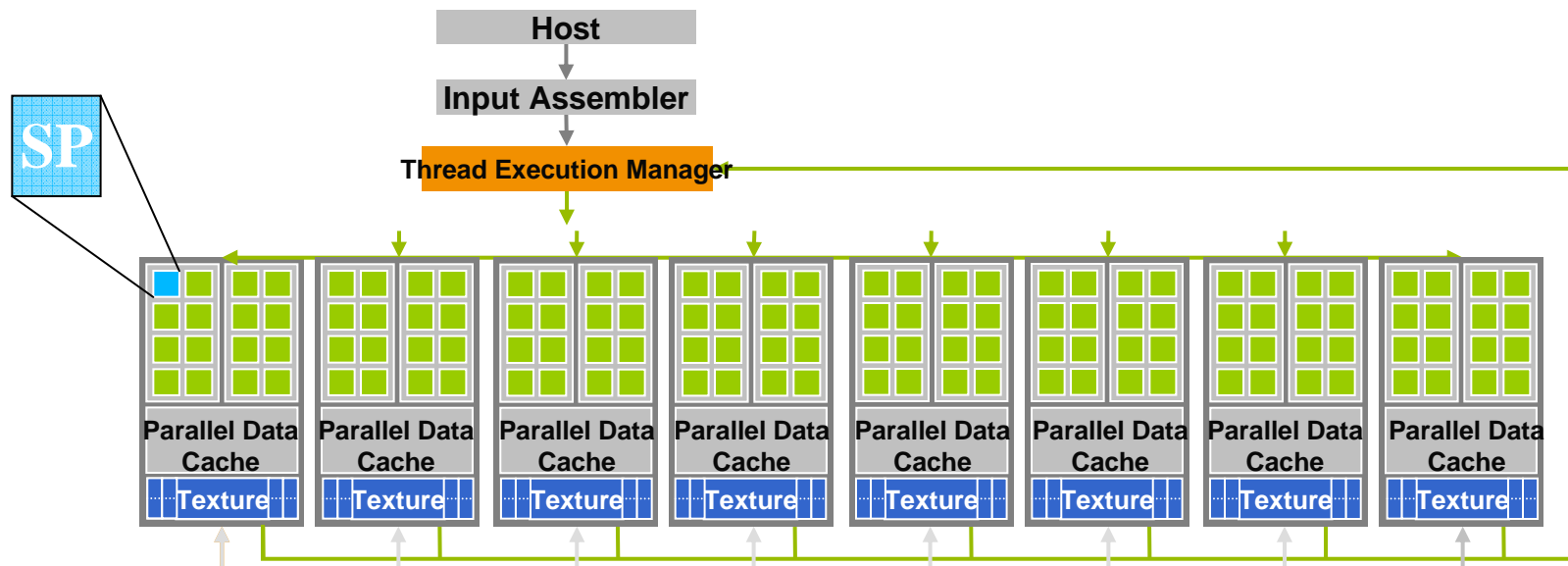
CUDA Processor Terminology

- **SPA**
 - Streaming Processor Array (variable across GeForce 8-series, 8 in GeForce8800)
- **SM**
 - Streaming Multiprocessor (8 SP)
 - Multi-threaded processor core
 - Fundamental processing unit for CUDA thread block
- **SP**
 - Streaming Processor
 - Scalar ALU for a single CUDA thread



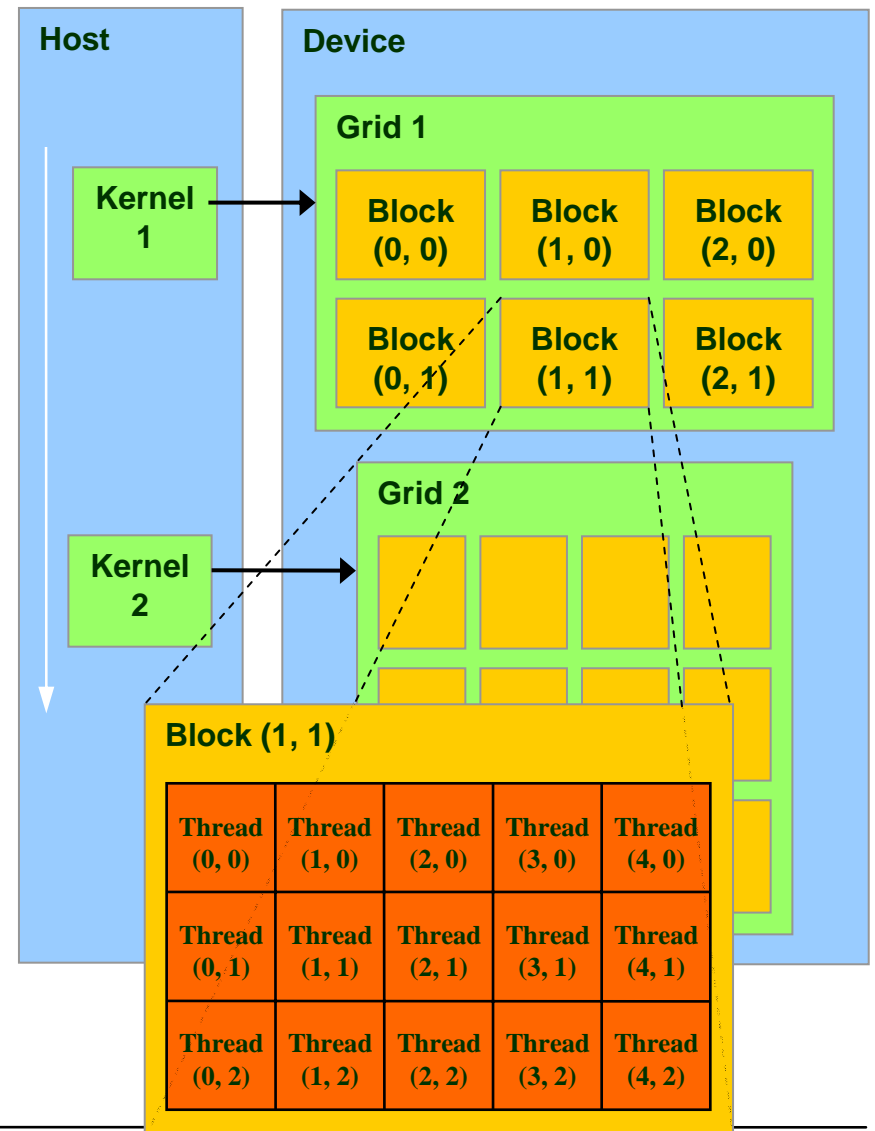
CUDA Processor Terminology

- **SPA**
 - Streaming Processor Array (variable across GeForce 8-series, 8 in GeForce8800)
- **SM**
 - Streaming Multiprocessor (8 SP)
 - Multi-threaded processor core
 - Fundamental processing unit for CUDA thread block
- **SP**
 - Streaming Processor
 - Scalar ALU for a single CUDA thread

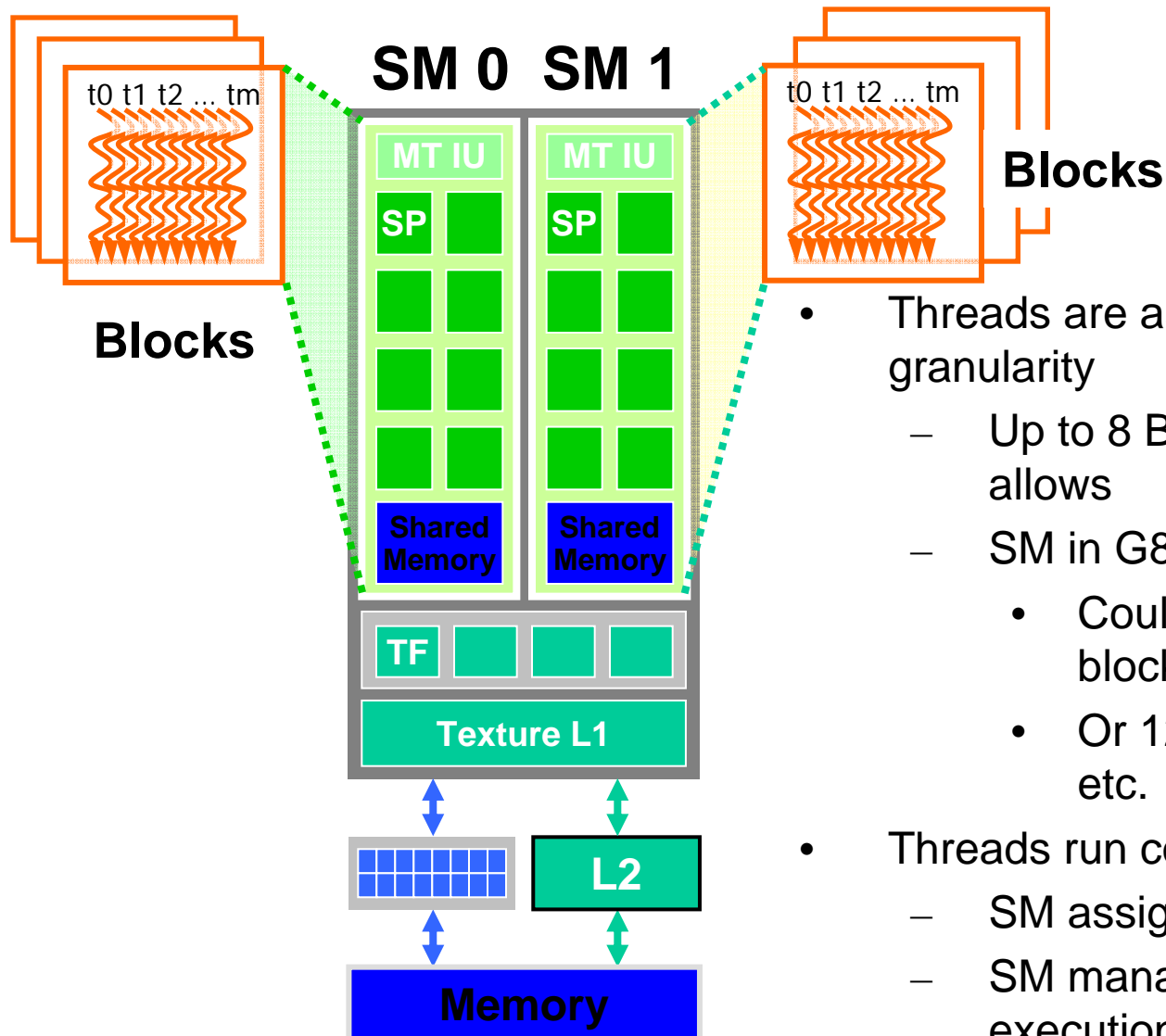


Thread Life Cycle in HW

- **Grid is launched on the SPA**
- **Thread Blocks are serially distributed to all the SM's**
 - Potentially >1 Thread Block per SM
- **Each SM launches Warps of Threads**
 - *2 levels of parallelism*
- **SM schedules and executes Warps that are ready to run**
- **As Warps and Thread Blocks complete, resources are freed**
 - SPA can distribute more Thread Blocks



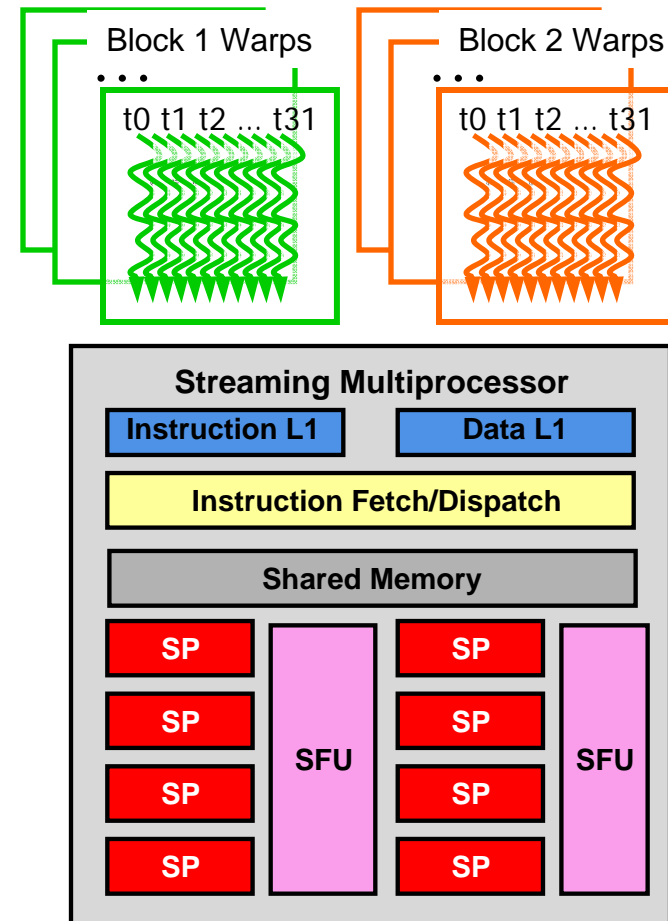
SM Executes Blocks



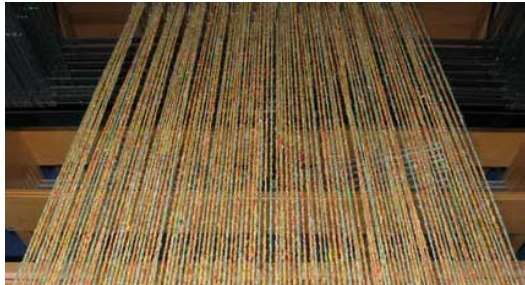
- Threads are assigned to SMs in Block granularity
 - Up to 8 Blocks to each SM as resource allows
 - SM in G80 can take up to 768 threads
 - Could be 256 (threads/block) * 3 blocks
 - Or 128 (threads/block) * 6 blocks, etc.
- Threads run concurrently
 - SM assigns/maintains thread id #s
 - SM manages/schedules thread execution

Thread Scheduling/Execution

- Each Thread Blocks is divided in 32-thread Warps
 - This is an implementation decision, not part of the CUDA programming model
- Warps are scheduling units in SM
- If 3 blocks are assigned to an SM and each Block has 256 threads, how many Warps are there in an SM?
 - Each Block is divided into $256/32 = 8$ Warps
 - There are $8 * 3 = 24$ Warps
 - At any point in time, only one of the 24 Warps will be selected for instruction fetch and execution.

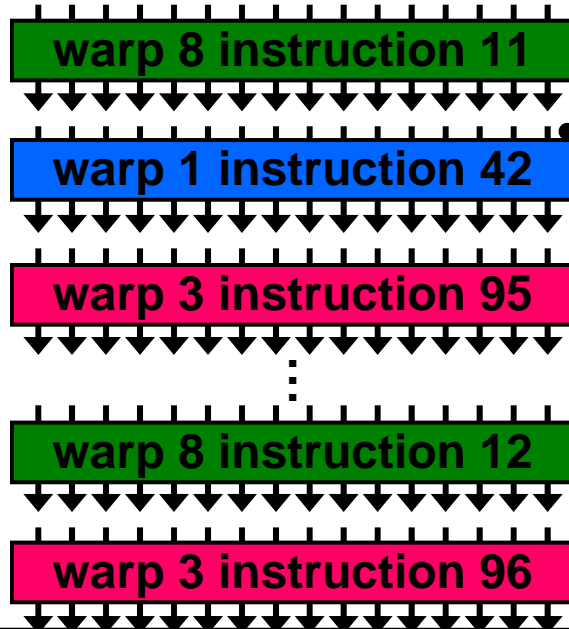


SM Warp Scheduling



SM multithreaded
Warp scheduler

time



- SM hardware implements zero-overhead Warp scheduling

- Warps whose next instruction has its operands ready for consumption are eligible for execution
- Eligible Warps are selected for execution on a prioritized scheduling policy
- All threads in a Warp execute the same instruction when selected

4 clock cycles needed to dispatch the same instruction for all threads in a Warp in G80

- If one global memory access is needed for every 4 instructions
- A minimal of 13 Warps are needed to fully tolerate 200-cycle memory latency

Warp

- **Group of (32) threads that are created, managed and executed on a multiprocessor**
- **Size can be queried: `warpSize`**
- **Example code**

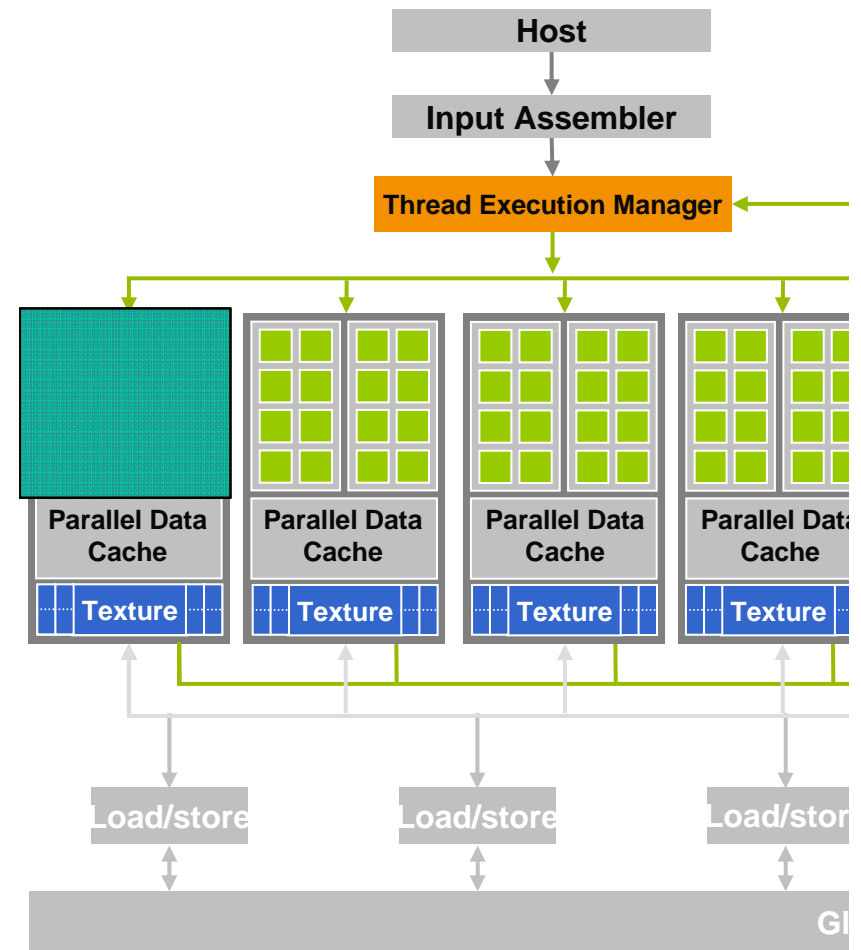
inst 1

inst 2

inst 3

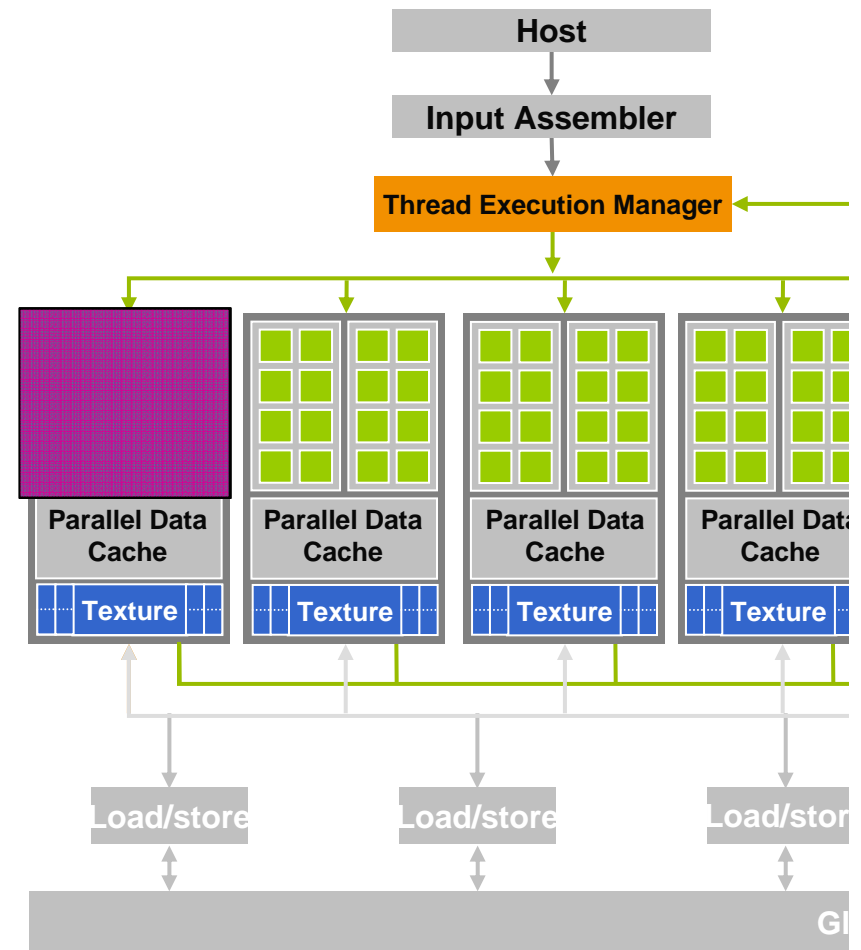
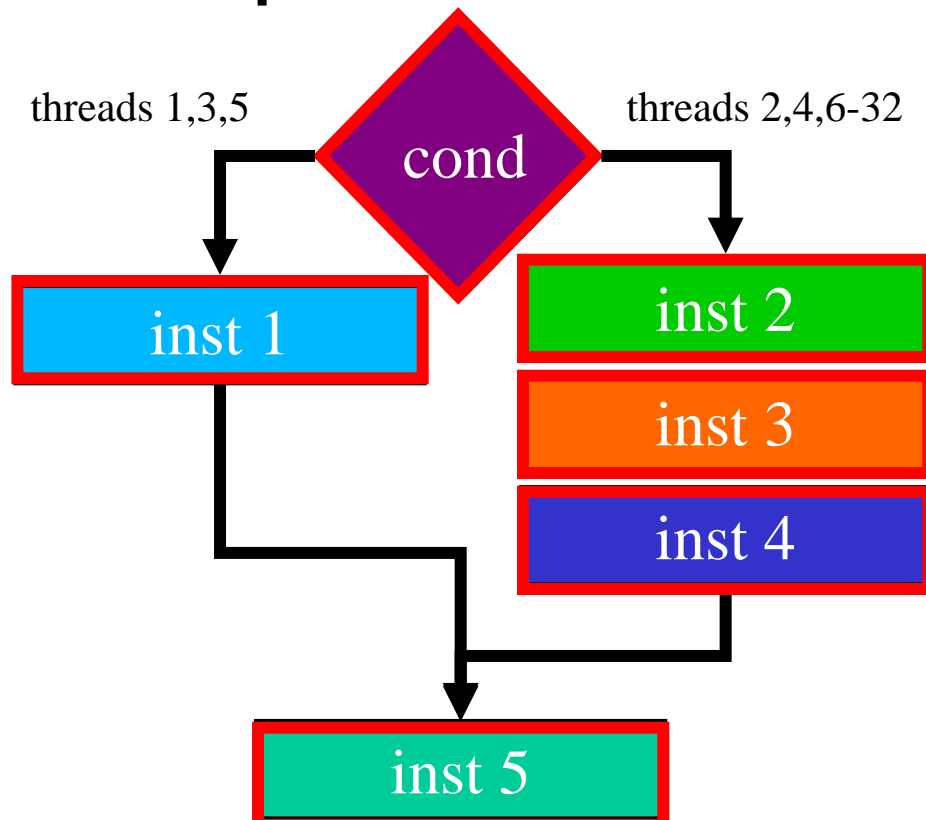
inst 4

inst 5



Branch Divergence within Warp

- Each warp will execute a single instruction at each time
- Diverging branches will be serialized
- Example code



Reduction – Version1

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
- **Typically parallel implementation:**
 - Recursively halve # threads, add two values per thread
 - Takes $\log(n)$ steps for n elements, requires $n/2$ threads

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

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
```

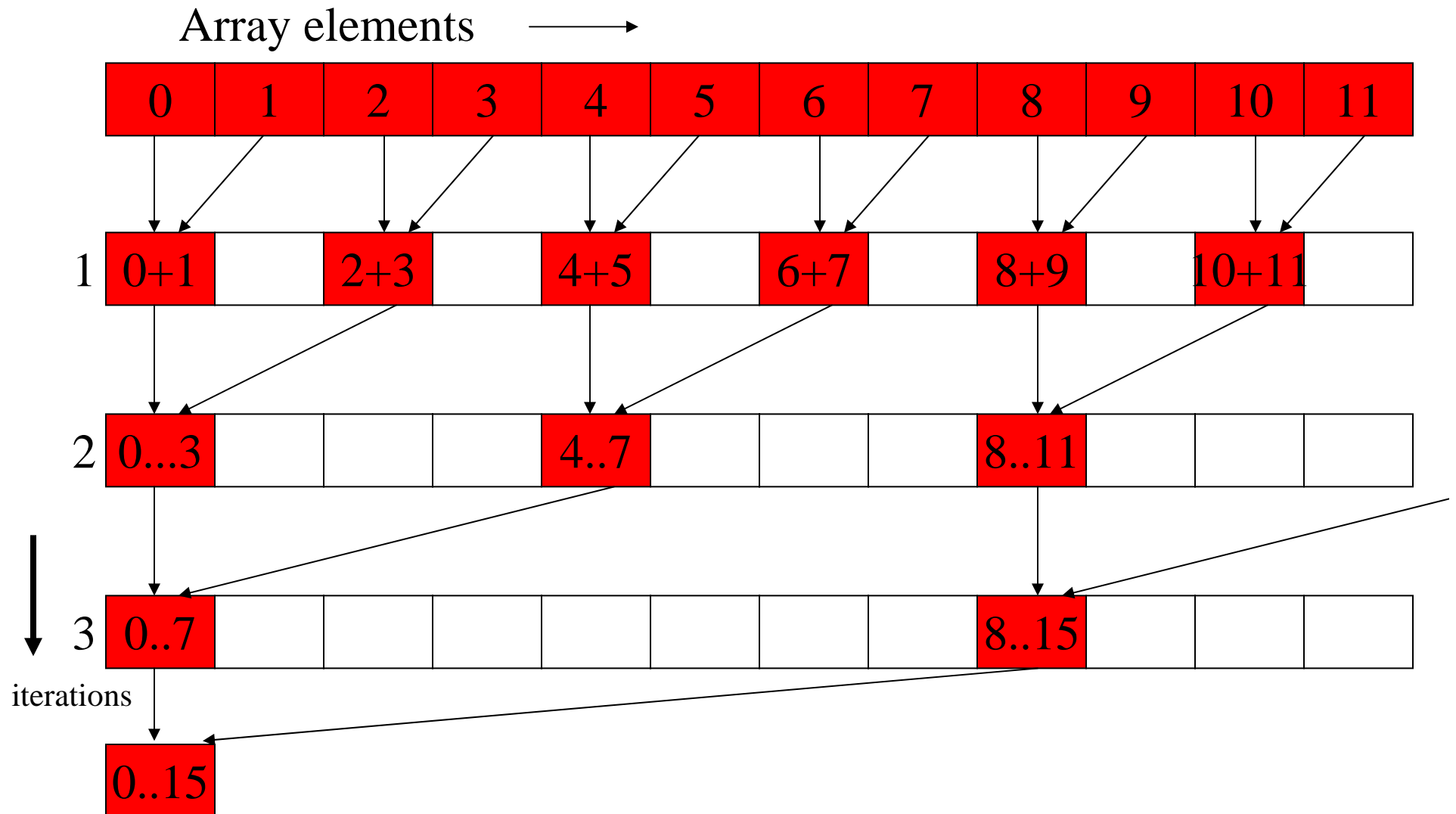
```
    __syncthreads();
```

```
    if (t % (2*stride) == 0)
```

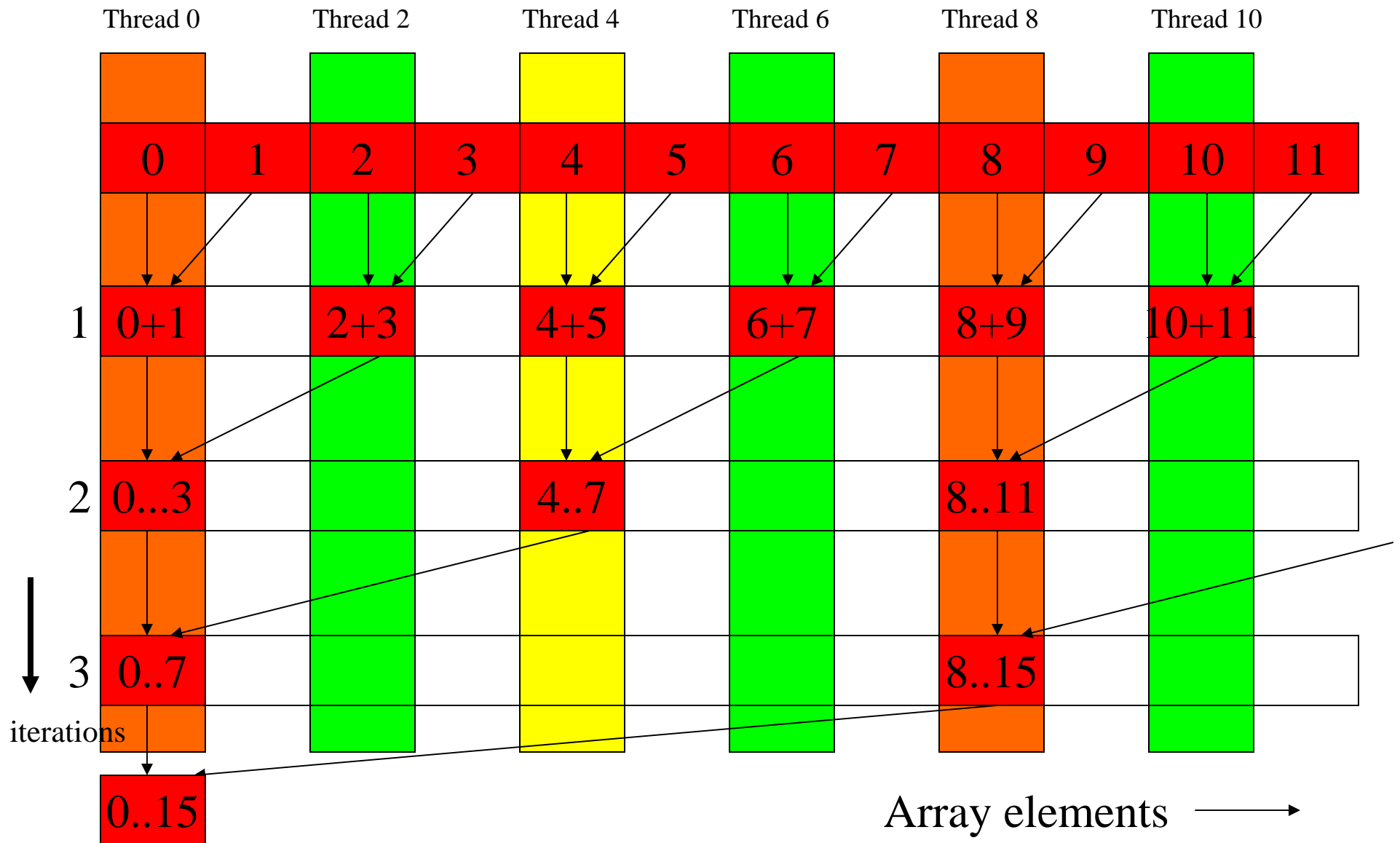
```
        partialSum[t] += partialSum[t+stride];
```

```
}
```

Vector Reduction



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

Shortcomings 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

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)
```

```
{
```

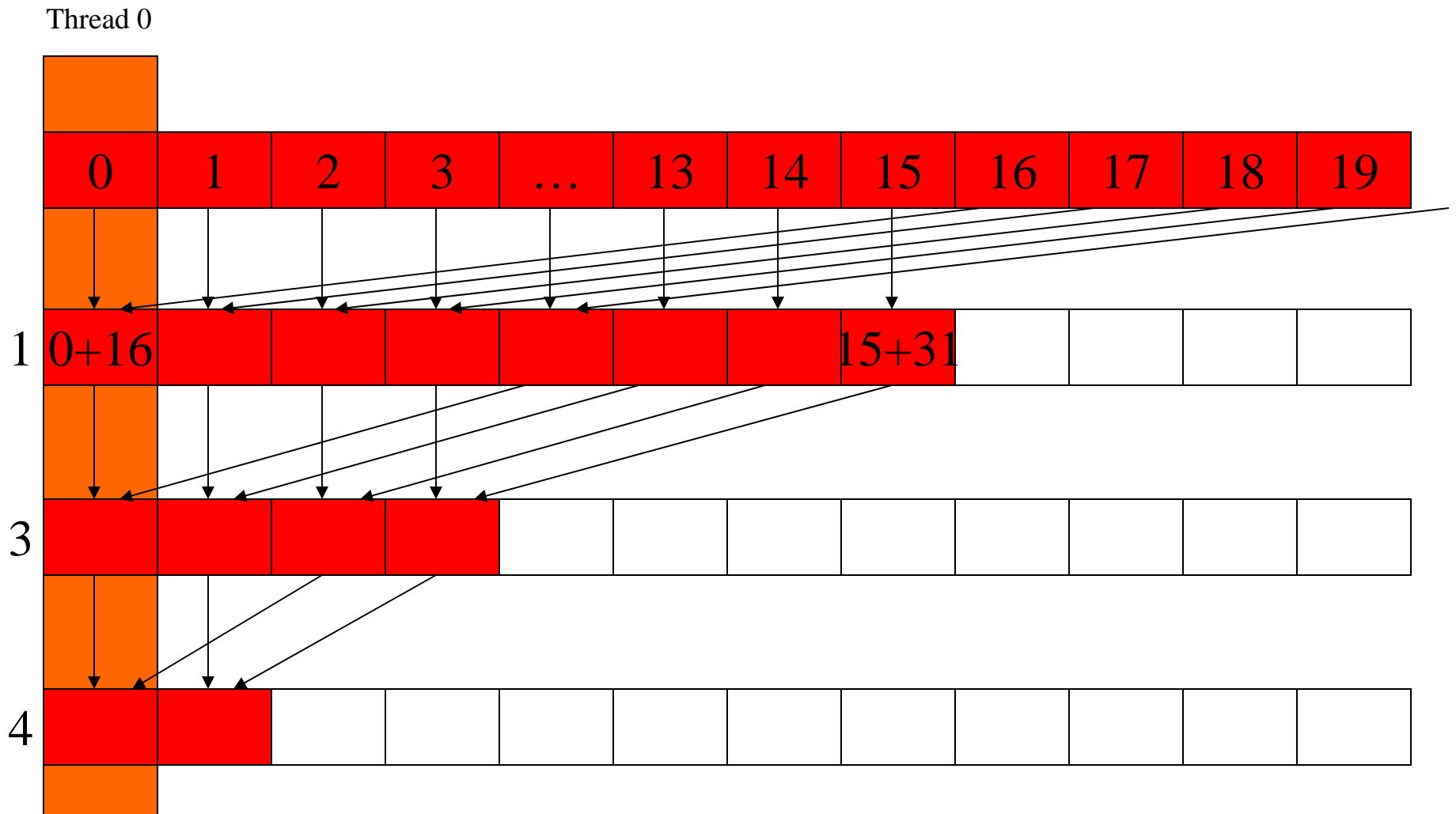
```
    __syncthreads();
```

```
    if (t < stride)
```

```
        partialSum[t] += partialSum[t+stride];
```

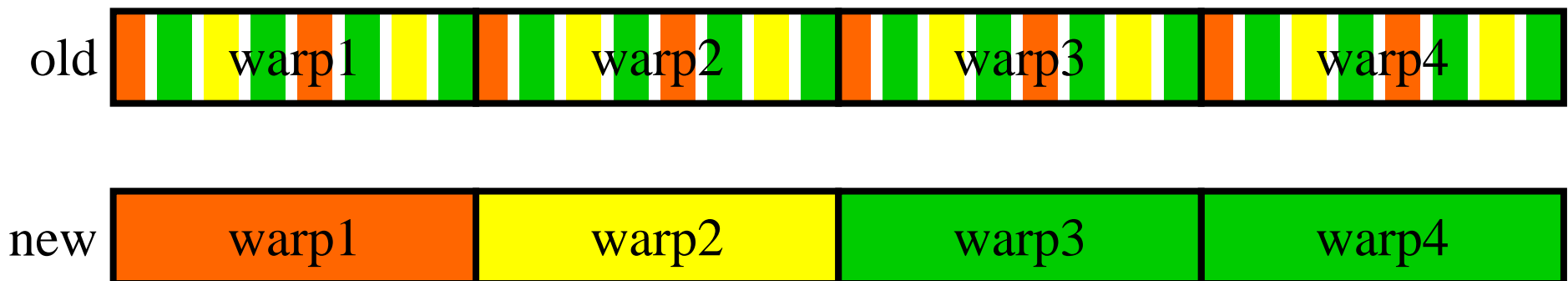
```
}
```

No Divergence until < 16 sub-sums



Observations About the New 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

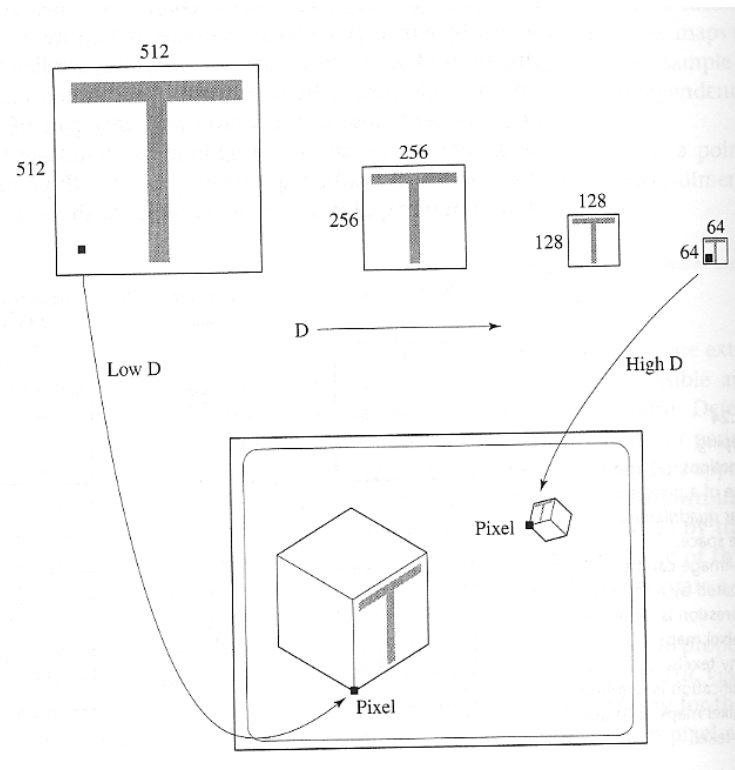
```
unsigned int tid = threadIdx.x;
for (unsigned int d = n>>1; d > 0; d /= 2)
    __syncthreads();
    if (tid < d)
        shared[tid] += shared[tid + d];
}
__syncthreads();
if (tid <= 32) { // unroll last 5 iterations
    shared[tid] += shared[tid + 1];
    shared[tid] += shared[tid + 2];
    shared[tid] += shared[tid + 3];
    shared[tid] += shared[tid + 4];
    shared[tid] += shared[tid + 5];
}
```

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

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

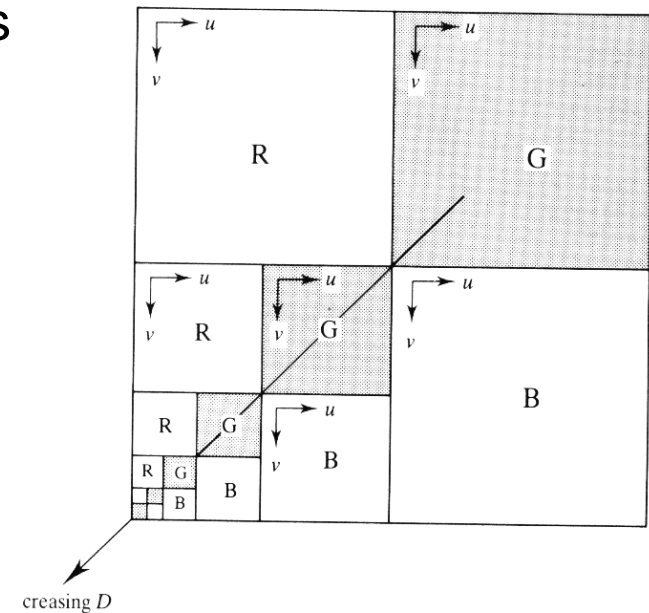
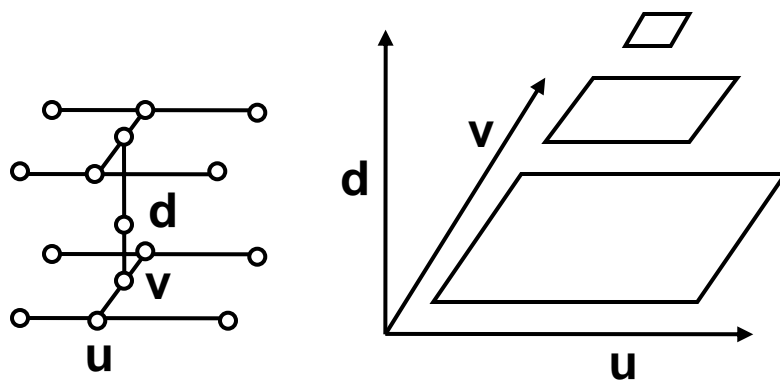
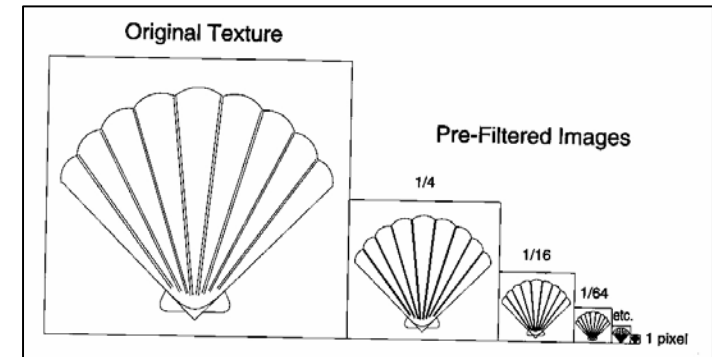
Application: MipMap Construction

- **Texture available in multiple resolutions**
 - Pre-processing step
- **Rendering: select appropriate texture resolution**
 - Selection is usually per pixel !!
 - Texel size(n) < extent of pixel footprint < texel size(n+1)



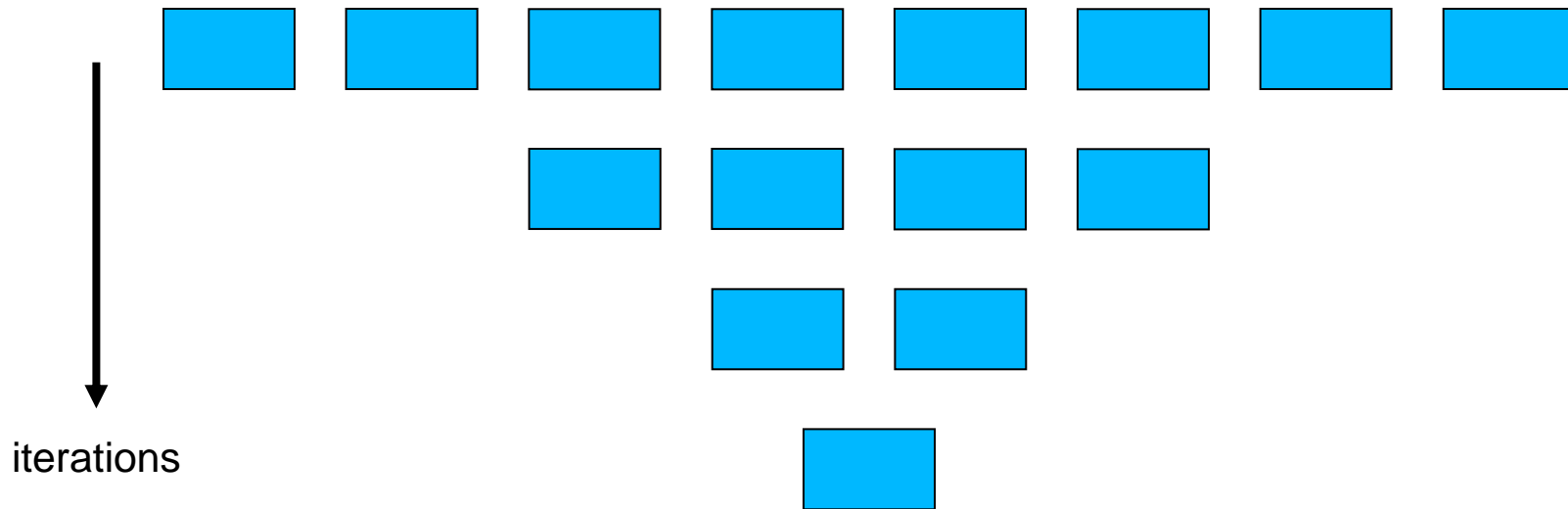
Application: MipMapping

- **Multum In Parvo (MIP): much in little**
- **Hierarchical resolution pyramid**
 - Repeated averaging over 2x2 texels
 - *This is vector reduction!*
- **Rectangular arrangement (RGB)**
- **Reconstruction**
 - Tri-linear interpolation of 8 nearest texels



Typical Parallel Programming Pattern

- $\log(n)$ steps



Scan / Prefix Sum – Algorithm Effects on Parallelism and Memory Conflicts

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

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

and returns the ordered set

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

- **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:**

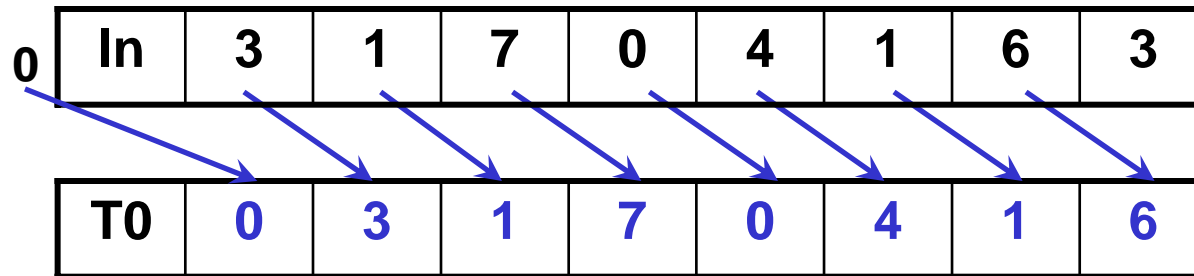
- radix sort
- quicksort
- String comparison
- Lexical analysis
- Stream compaction
- Polynomial evaluation
- Solving recurrences
- Tree operations
- Range Histograms
- Etc.

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**

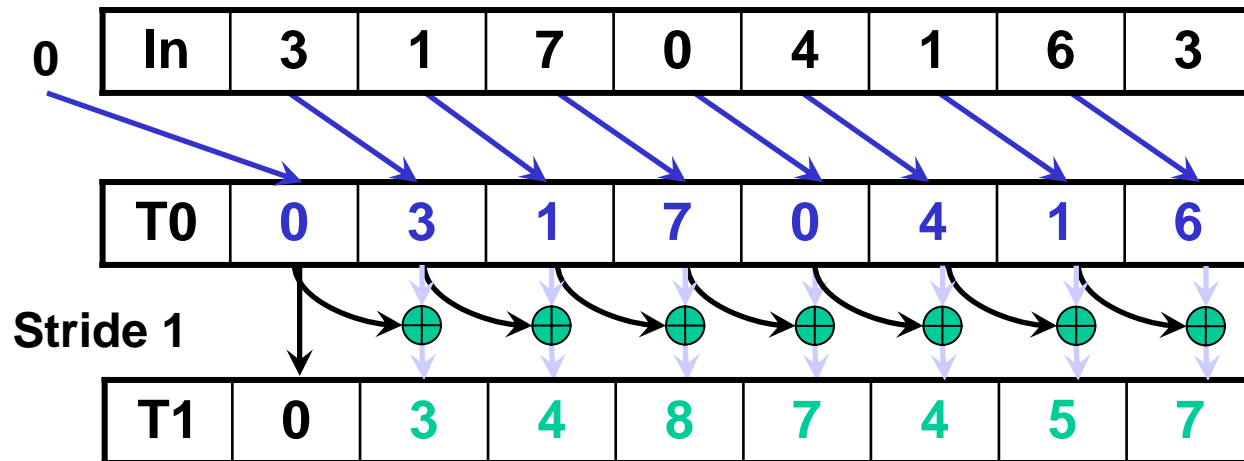
A First-Attempt Parallel Scan Algorithm



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

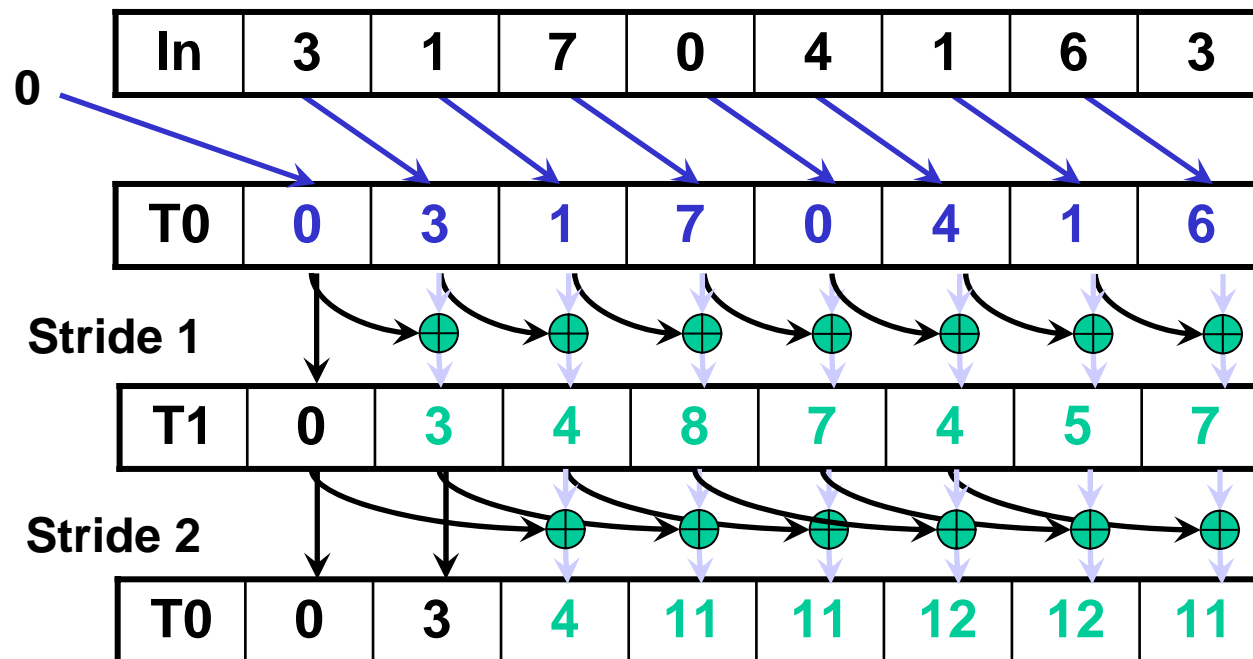


1. (previous slide)
2. Iterate $\log(n)$ times: Threads *stride* to n : Add pairs of elements *stride* elements apart. Double *stride* at each iteration. (note must double buffer shared mem arrays)

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)

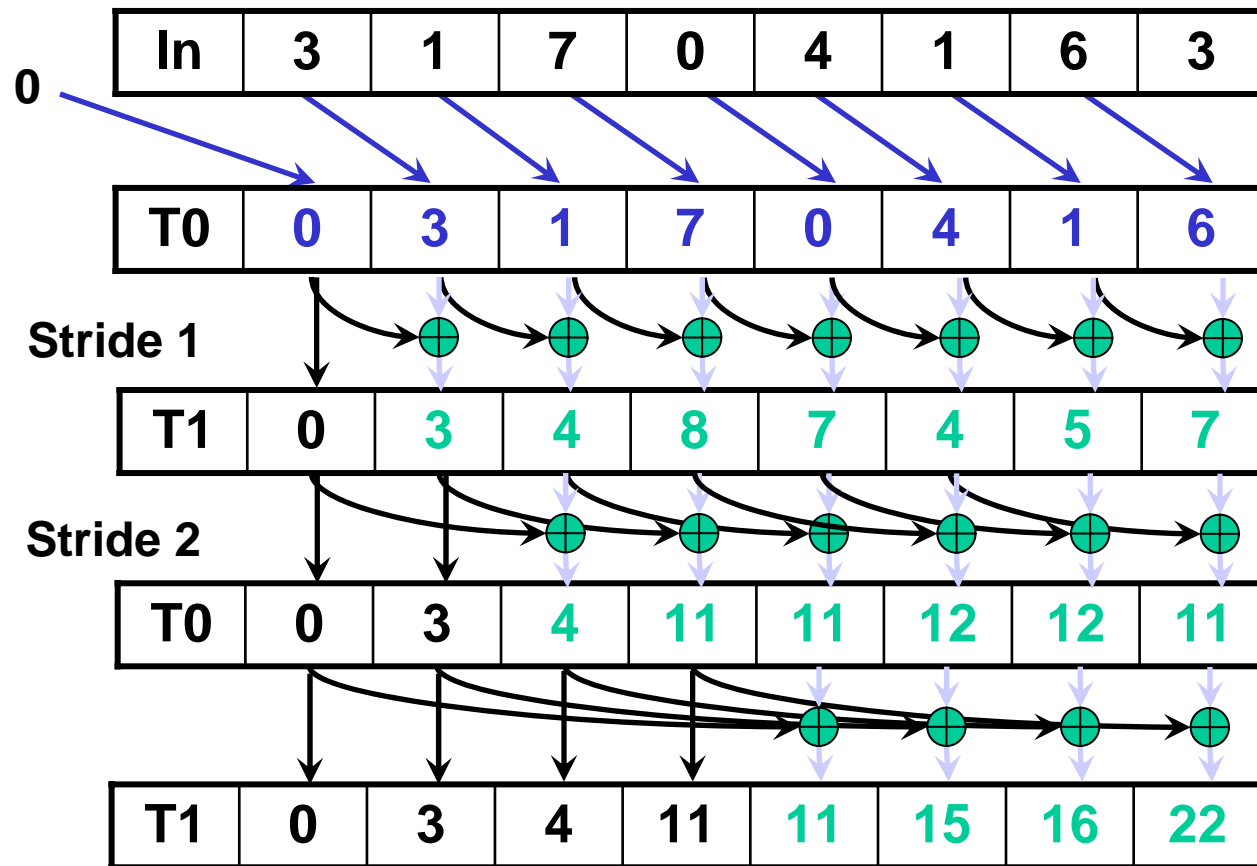
A First-Attempt Parallel Scan Algorithm



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)

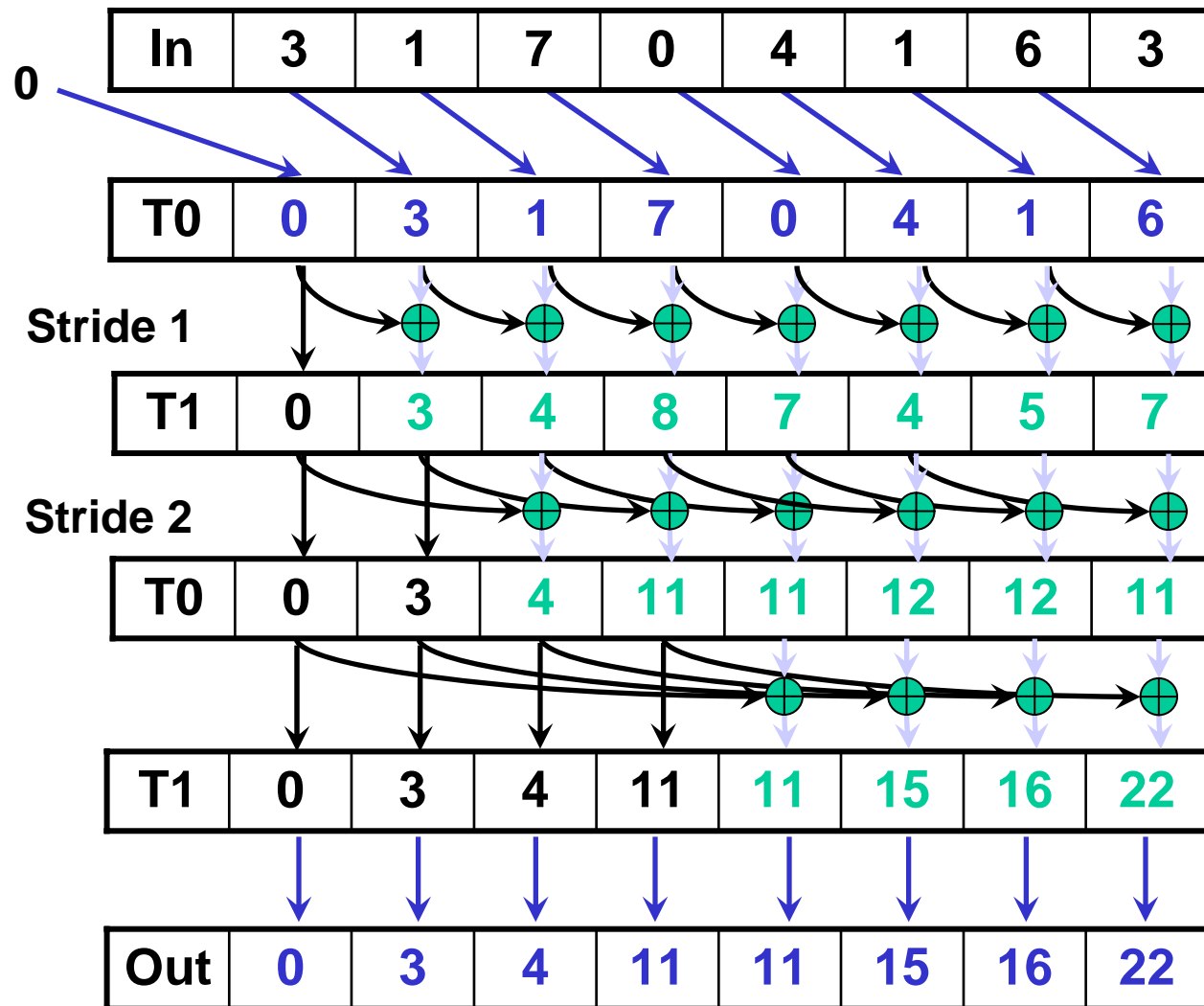
Iteration #2
Stride = 2

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)

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**
 - The steps do $(n/2 + n/2 - 1)$, $(n/4 + n/2 - 1)$, $(n/8 + n/2 - 1)$, .. $(1 + n/2 - 1)$ adds each
 - Total adds: $n * (\log(n) - 1) + 1 \rightarrow O(n * \log(n))$ 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

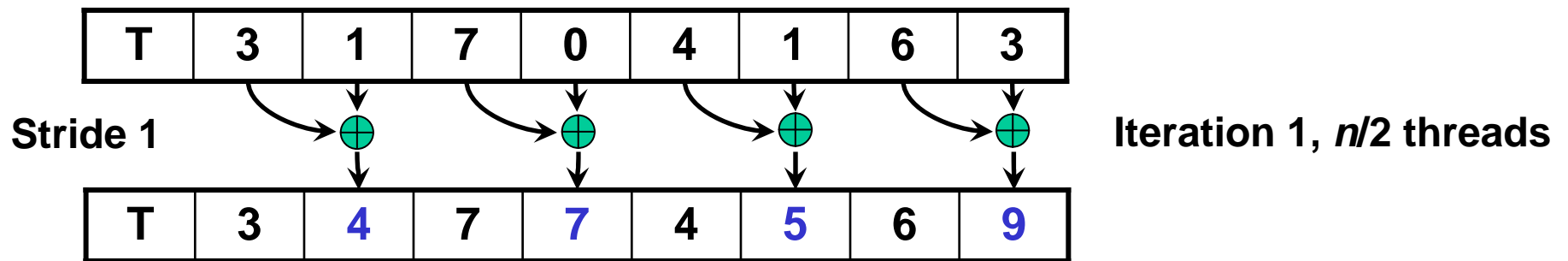
- **For improving efficiency**
- **A common parallel algorithm pattern:**
 - Build a balanced binary tree on the input data and sweep it to and from the root
 - Tree is not an actual data structure, but a concept to determine what each thread does at each step
- **For scan:**
 - Traverse down from leaves 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

Build the Sum Tree

T	3	1	7	0	4	1	6	3
---	---	---	---	---	---	---	---	---

Assume array is already in shared memory

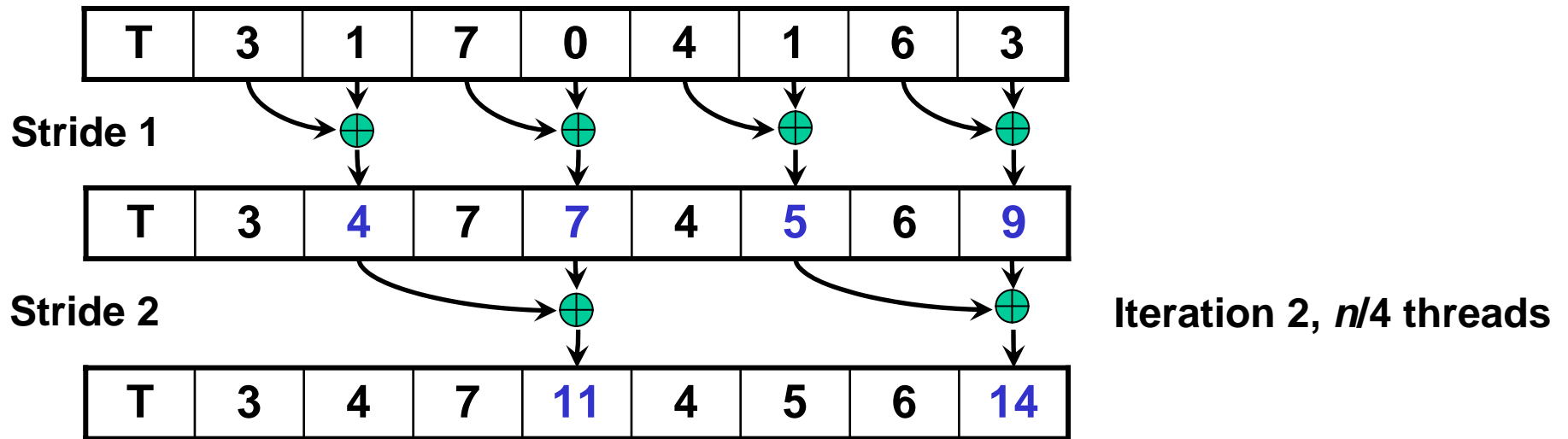
Build the Sum Tree



Each \oplus corresponds to a single thread.

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

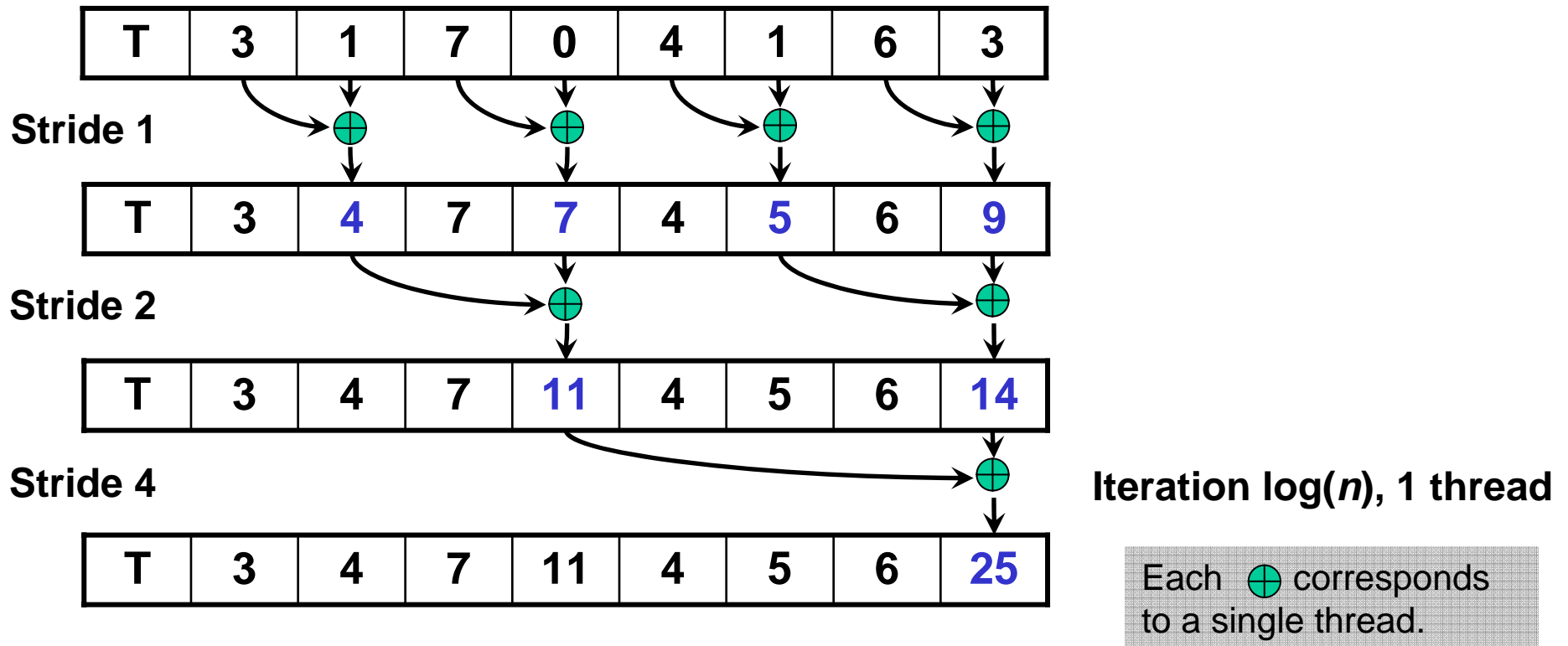
Build the Sum Tree



Each \oplus corresponds to a single thread.

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

Build the Sum Tree



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

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

Zero the Last Element

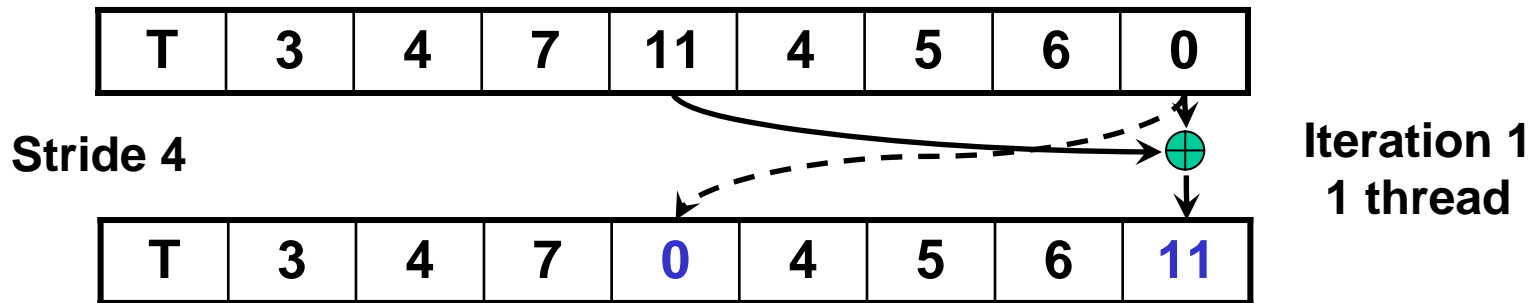
T	3	4	7	11	4	5	6	0
---	---	---	---	----	---	---	---	---


We now have an array of partial sums. Since this is an exclusive scan, set the last element to zero. It will propagate back to the first element.

Build Scan From Partial Sums

T	3	4	7	11	4	5	6	0
---	---	---	---	----	---	---	---	---

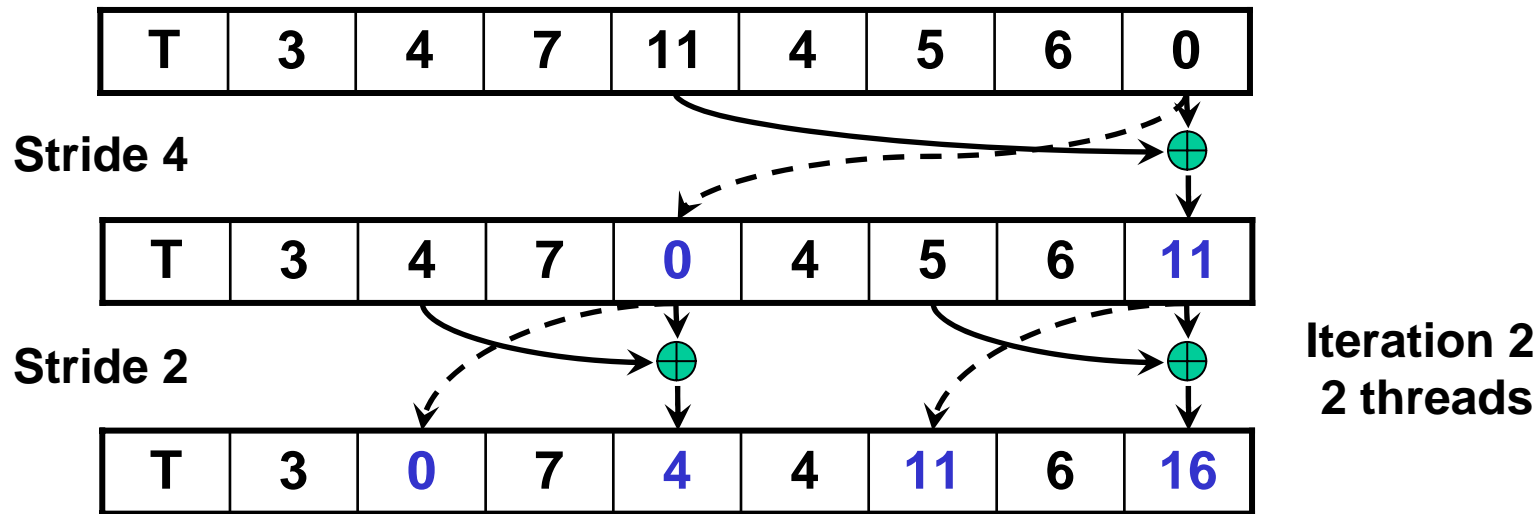
Build Scan From Partial Sums




Each  corresponds to a single thread.

Iterate $\log(n)$ times. Each thread adds value *stride* elements away to its own value, and sets the value *stride* elements away to its own *previous* value.

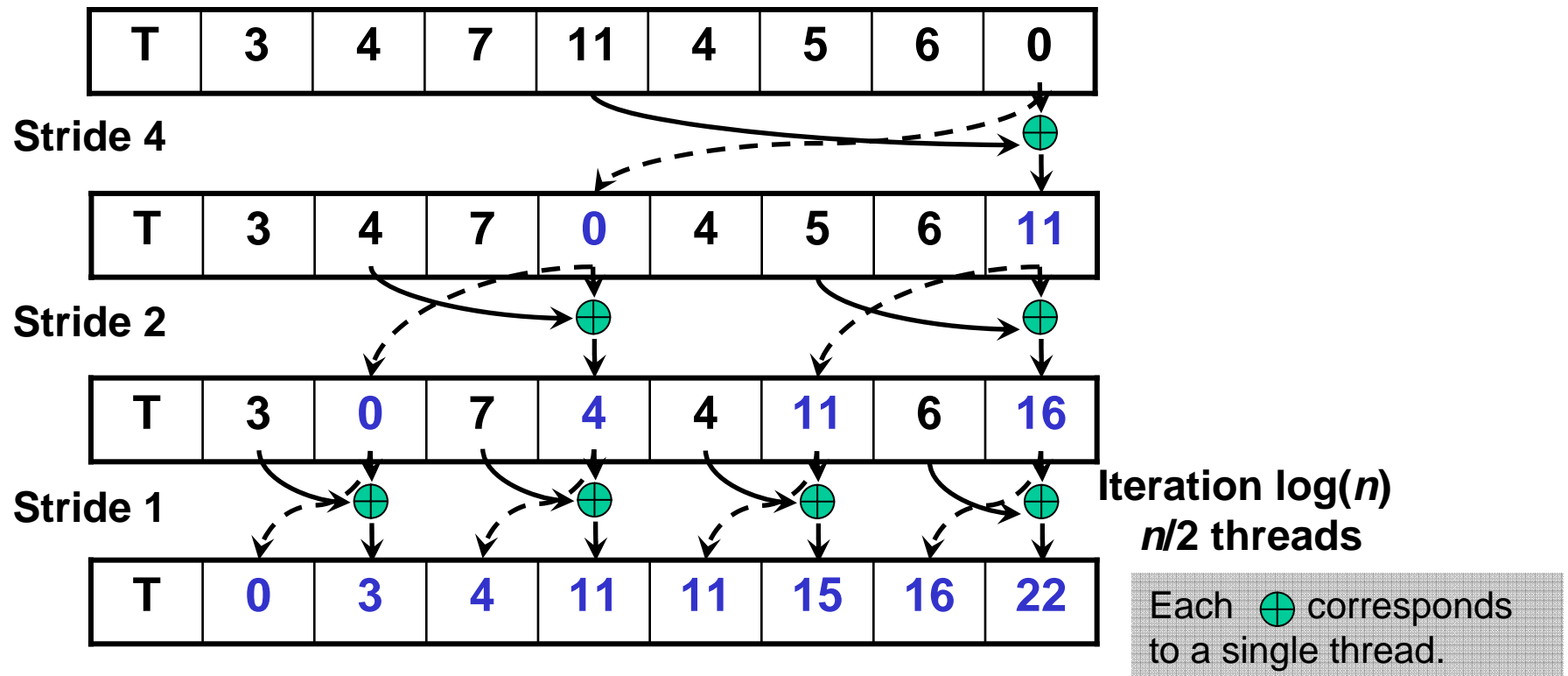
Build Scan From Partial Sums



Each  corresponds to a single thread.

Iterate $\log(n)$ times. Each thread adds value *stride* elements away to its own value, and sets the value *stride* elements away to its own *previous* value.

Build Scan From Partial Sums



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

Total steps: $2 * \log(n)$.

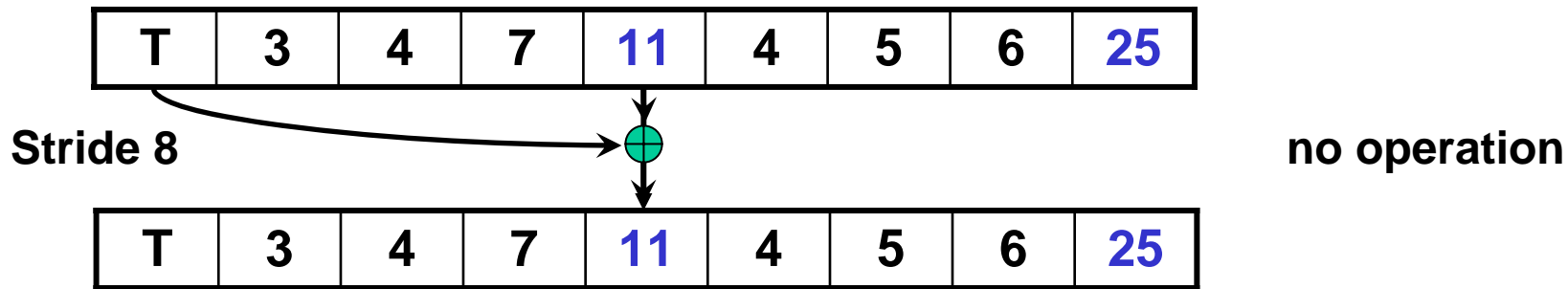
Total work: $2 * (n-1)$ adds = $O(n)$ **Work Efficient!**


Now the Inverse Direction

T	3	4	7	11	4	5	6	25
---	---	---	---	----	---	---	---	----

We now have an array of partial sums. Let's propagate the sums back.

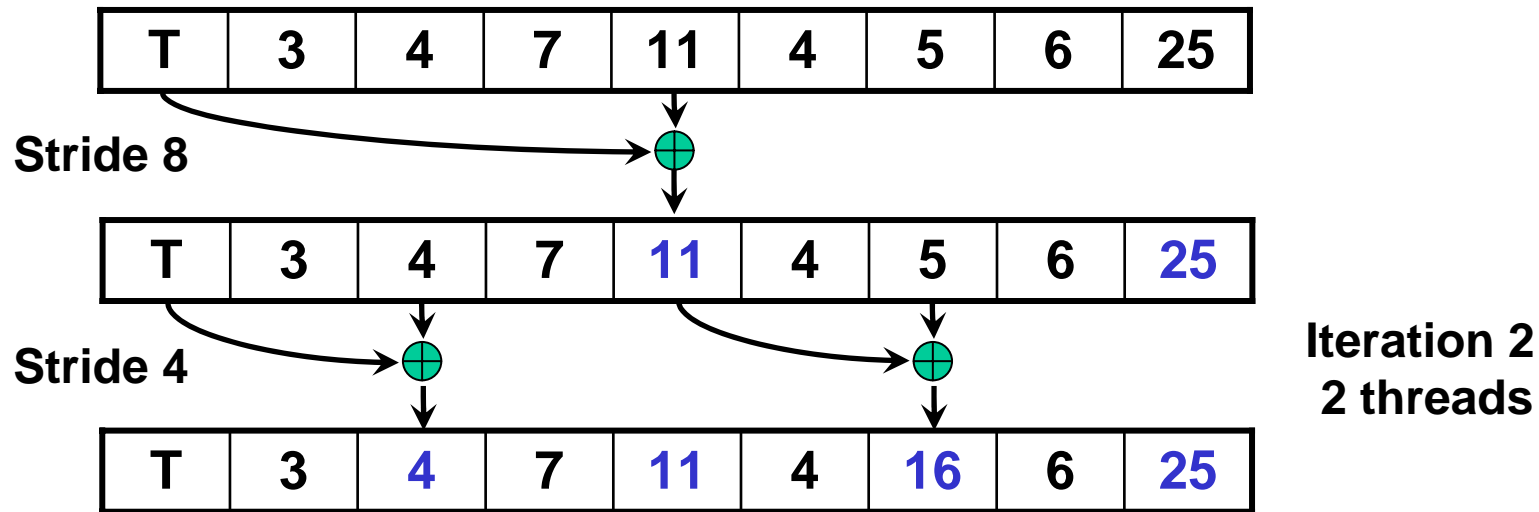
Build Scan From Partial Sums




Each  corresponds to a single thread.

Iterate $\log(n)$ times. Each thread adds value *stride* elements away to its own value. First element adds zero.

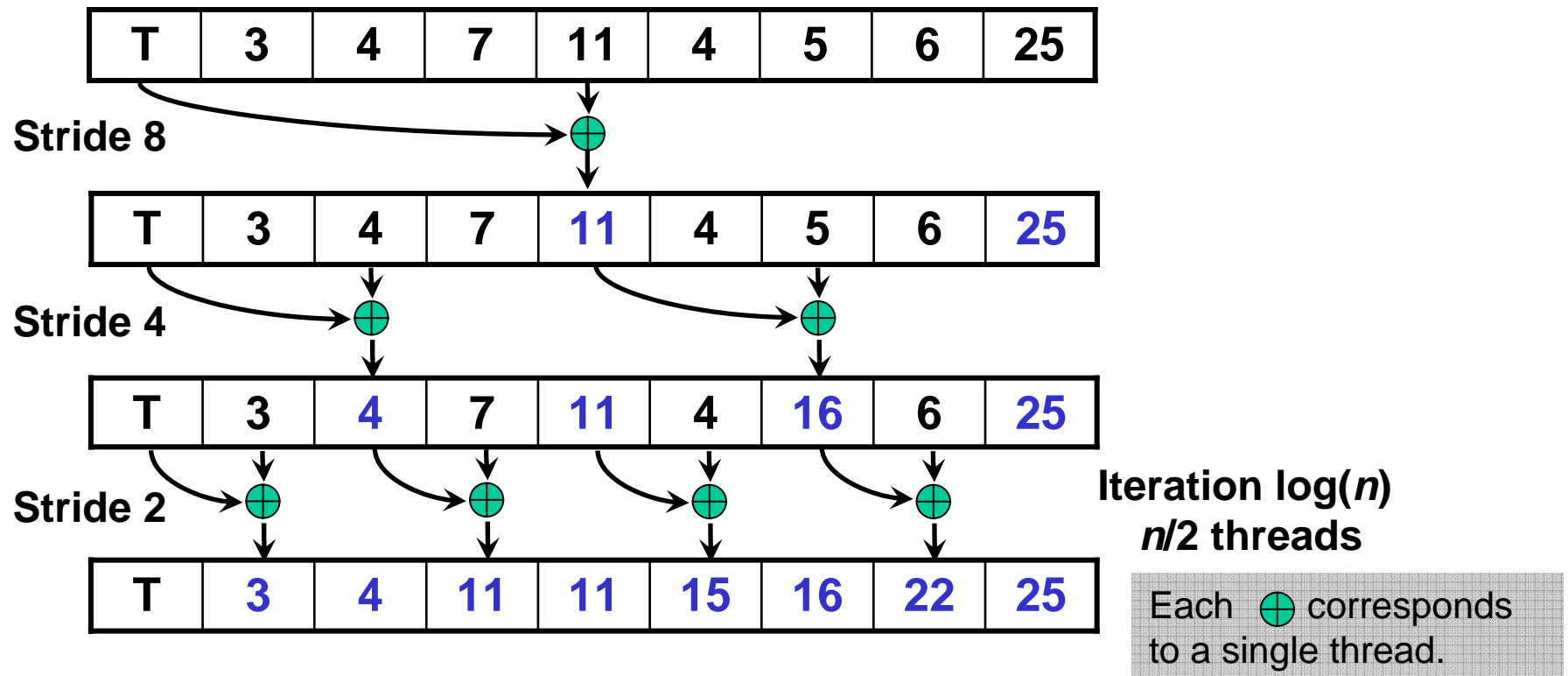
Build Scan From Partial Sums



Each  corresponds to a single thread.

Iterate $\log(n)$ times. Each thread adds value *stride* elements away to its own value. First element adds zero.

Build Scan From Partial Sums



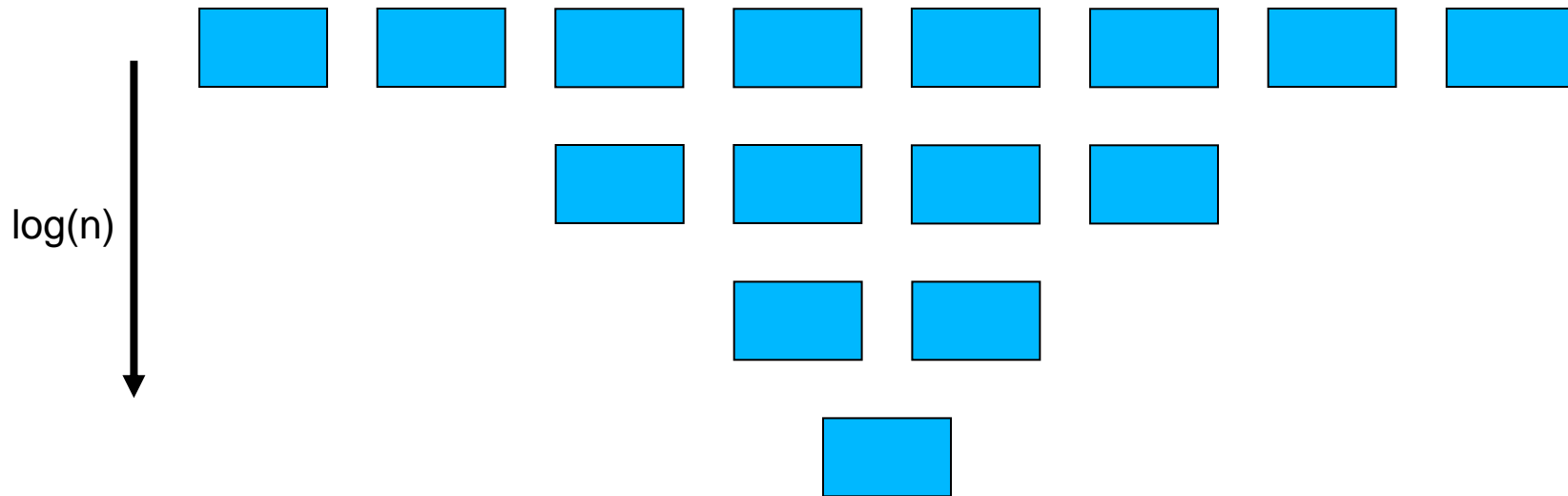
Done! We now have a completed scan that we can write out to device memory.

Total steps: $2 * \log(n)$.

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

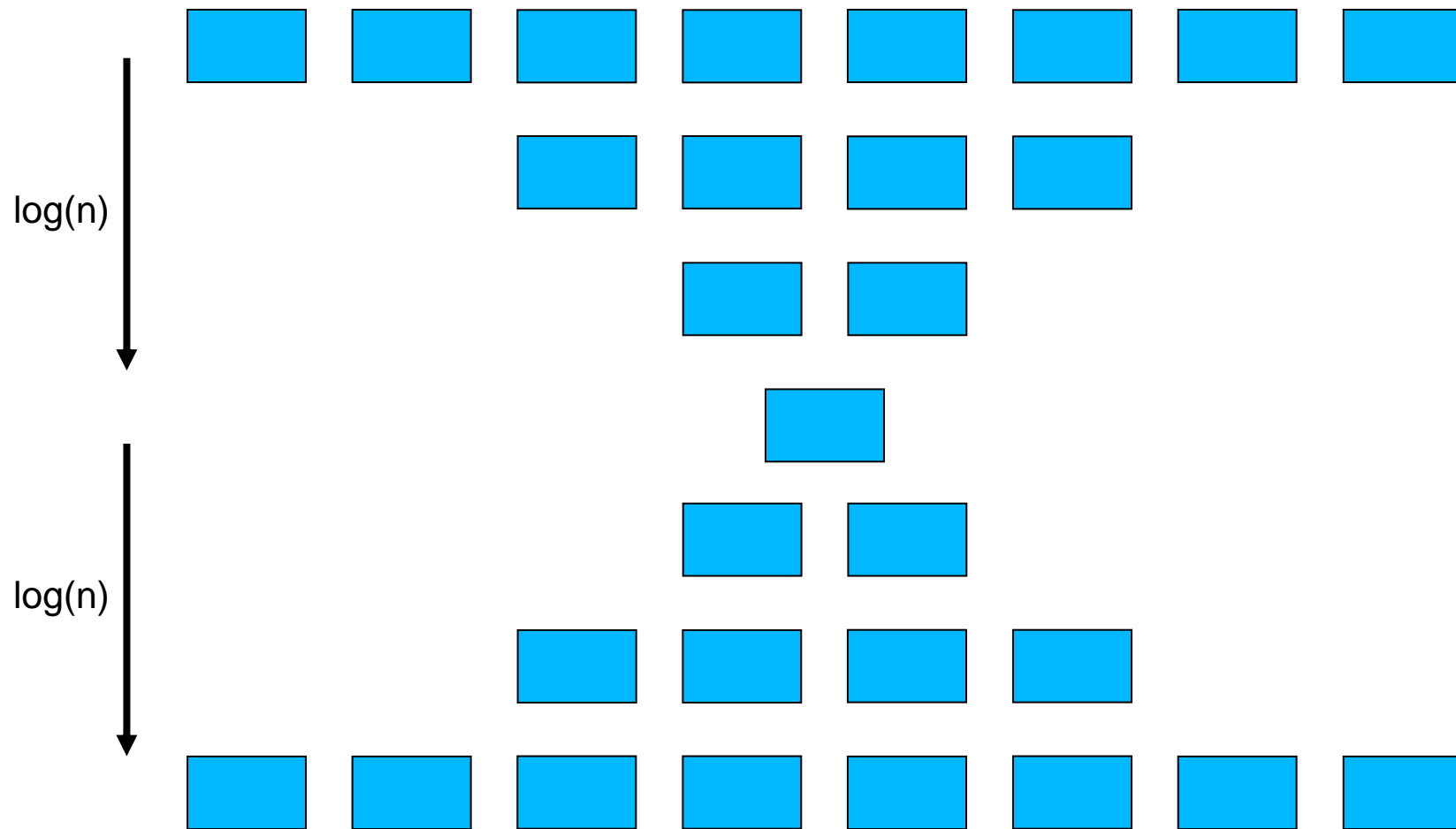
Typical Parallel Programming Pattern

- **2 log(n) steps**

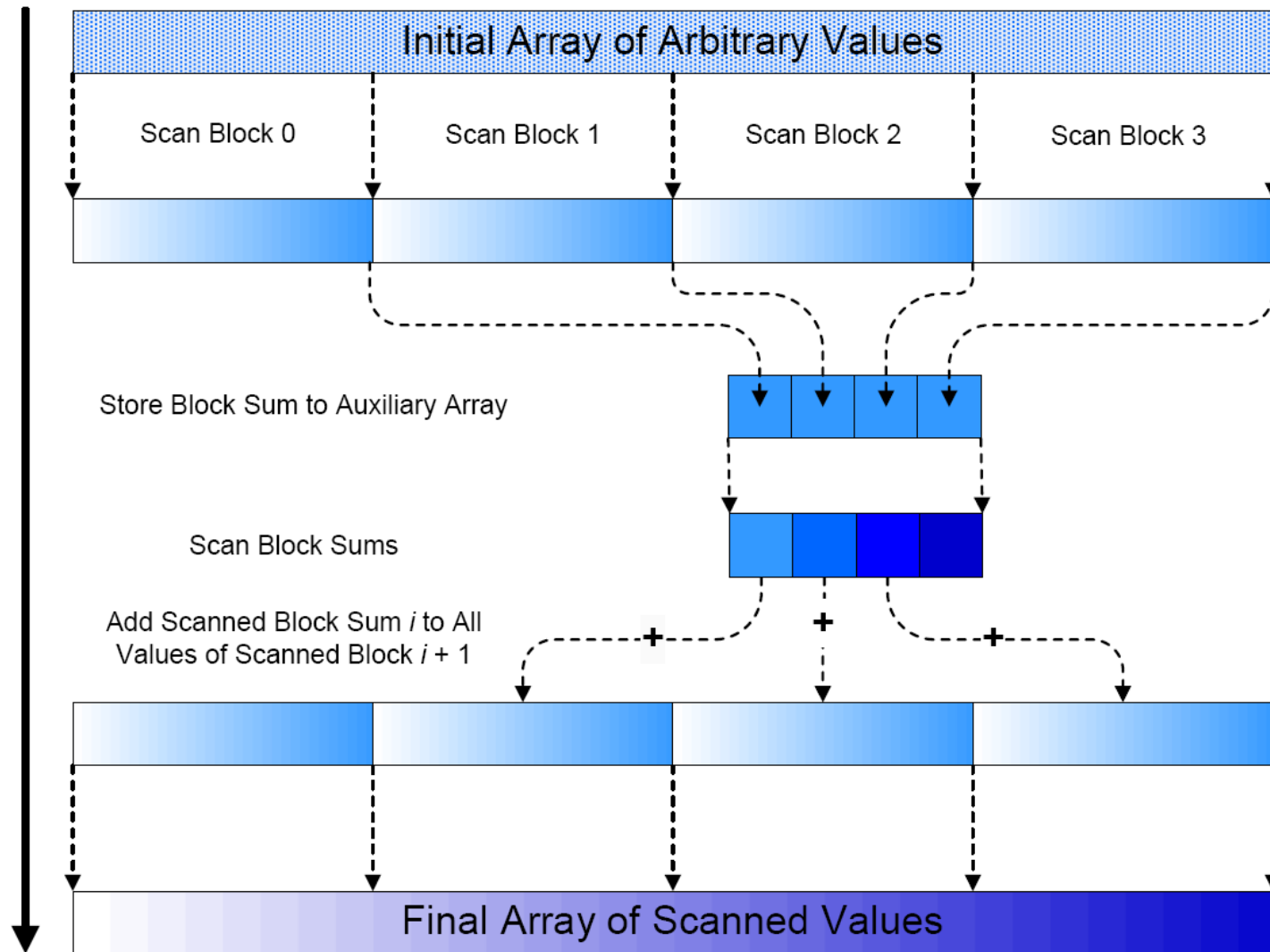


Typical Parallel Programming Pattern

- $2 \log(n)$ steps



Application to Large Arrays



[Mark Harris]

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];
```

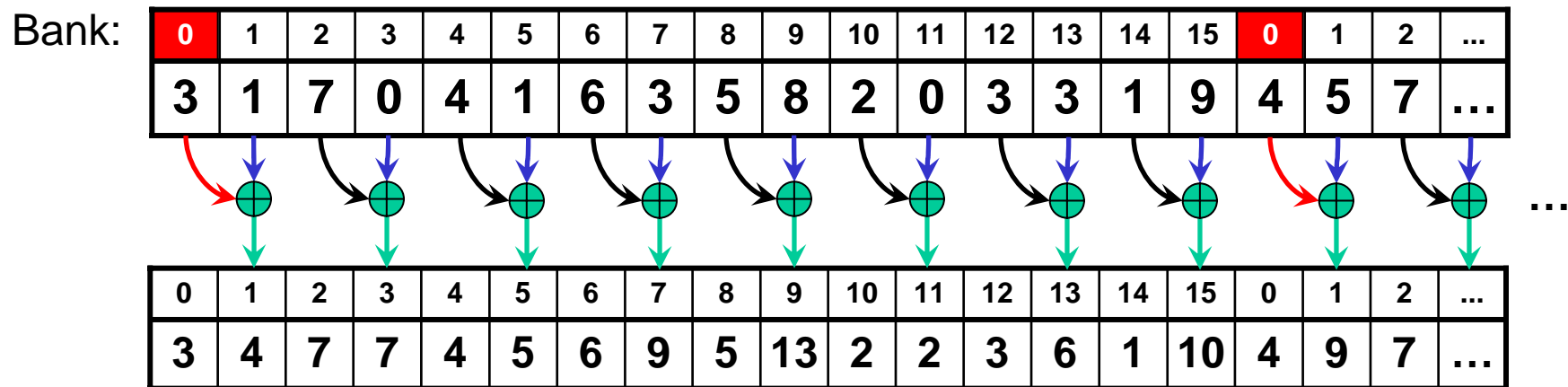
- **Threads:(0,1,2,...,8,9,10,...)→banks:(0,2,4,...,0,2,4,...)**

- **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



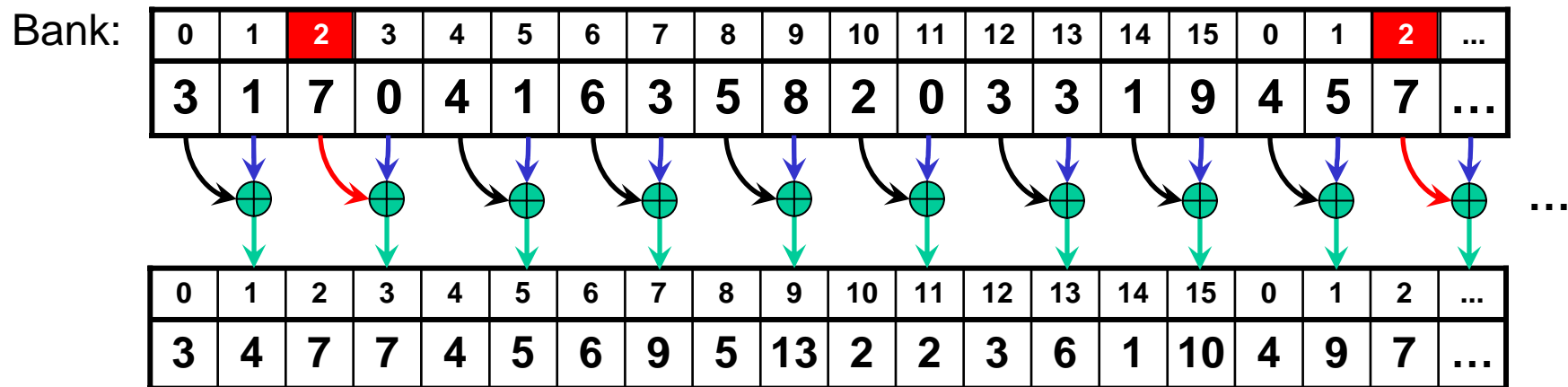
First iteration: 2 threads access each of 8 banks.

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.



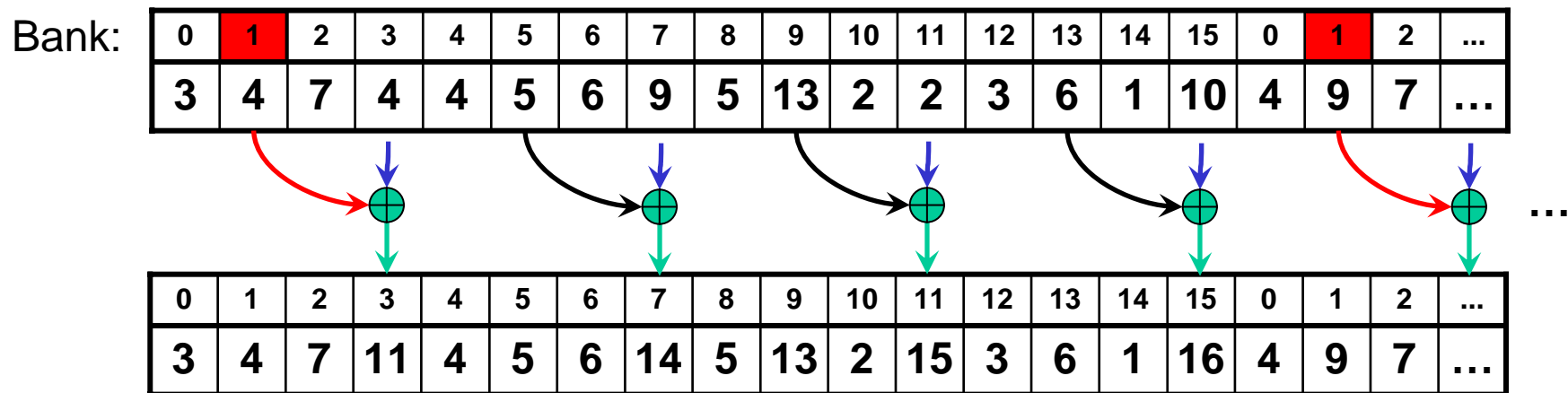
First iteration: 2 threads access each of 8 banks.

Each  corresponds to a single thread.


Like-colored arrows represent simultaneous memory accesses

Bank Conflicts in the tree algorithm

- **2nd 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.



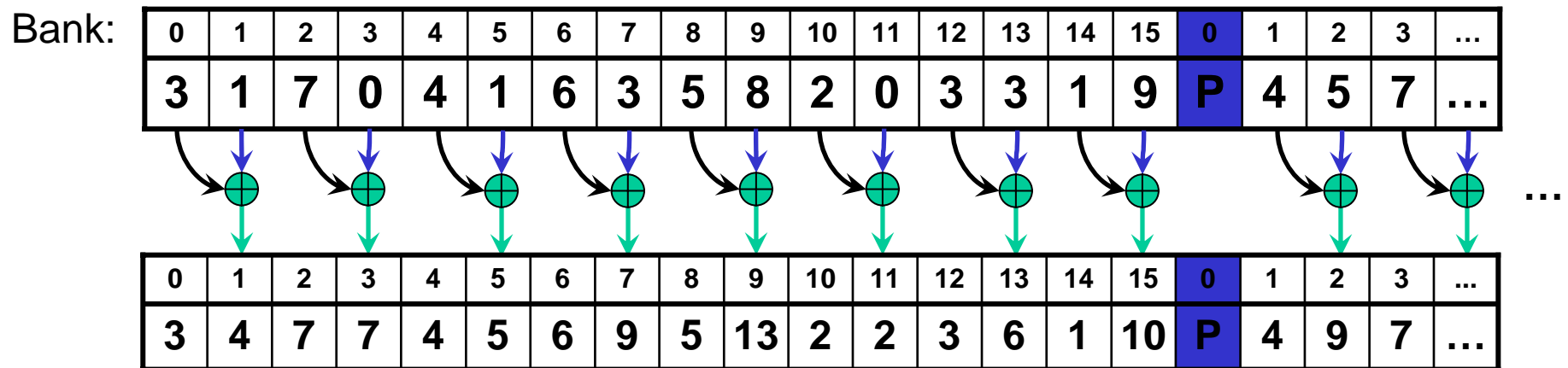
2nd iteration: 4 threads access each of 4 banks.

Each  corresponds to a single thread.

Like-colored arrows represent simultaneous memory accesses

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!**



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

(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 = 2 * 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

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:
 - $12 \cdot 2 + 6 \cdot (4 + 8 + 8 + 4 + 2 + 1) = 186$ cycles
 - 48 cycles if there were no bank conflicts! $12 \cdot 1 + (6 \cdot 6)$

Scan Bank Conflicts (3)

- **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	16	32	48	64	80	96	112	128	144	160	176	192	208	224	240	256	272	288	304	320	336	352	368	384	400	416	432	448	464	480	496		
16	0	32	64	96	128	160	192	224	256	288	320	352	384	416	448	480																		
32	0	64	128	192	256	320	384	448																										
64	0	128	256	384																														
128	0	256																																
256	0																																	
Conflicts	Banks																																	
16-way	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
16-way	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0																		
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:**
 - $48 \cdot 2 + 24 \cdot 4 + 12 \cdot 8 + 6 \cdot (16 + 16 + 8 + 4 + 2 + 1) = 570$ cycles
 - 120 cycles if there were no bank conflicts!

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	= Padding inserted																													
	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																															

- **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	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
None	0	8	17	25	34	42	51	59	68	76	85	93	102	110	119	127																
None	0	17	34	51	68	85	102	119																								
None	0	34	68	102																												
None	0	68																														
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	= Padding inserted																													
256	0																															
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**

Prefix Sum Application - Range Histogram -

Range Histogram

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

$$h[i] = |J| \quad J = \{j \mid v[j] = i\}$$

- 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:**

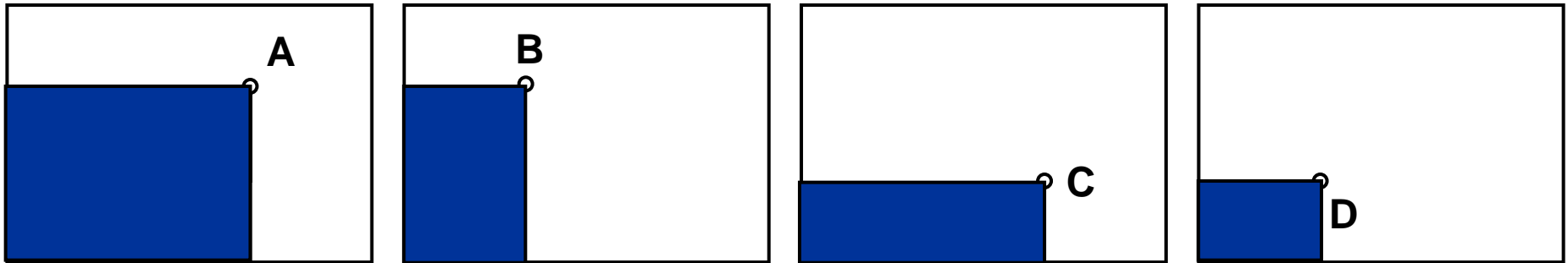
`hrange = pref[B] - pref[A];`

$$= \sum_0^B h[i] - \sum_0^A h[i] = \sum_A^B h[i]$$

Prefix Sum Application - Summed Area Tables -

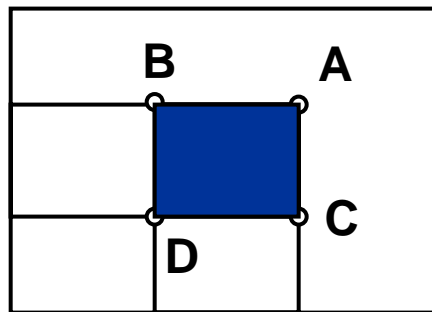
Summed Area Tables

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



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

$$\int_{BxCy}^{AxAy} I(x, y) dx dy = A - B - C + D$$

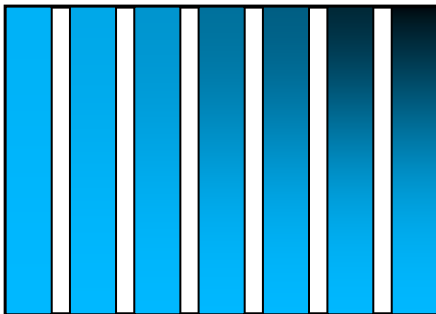


Summed Area Table with Prefix Sums

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



- **... then vertically on the result**



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**

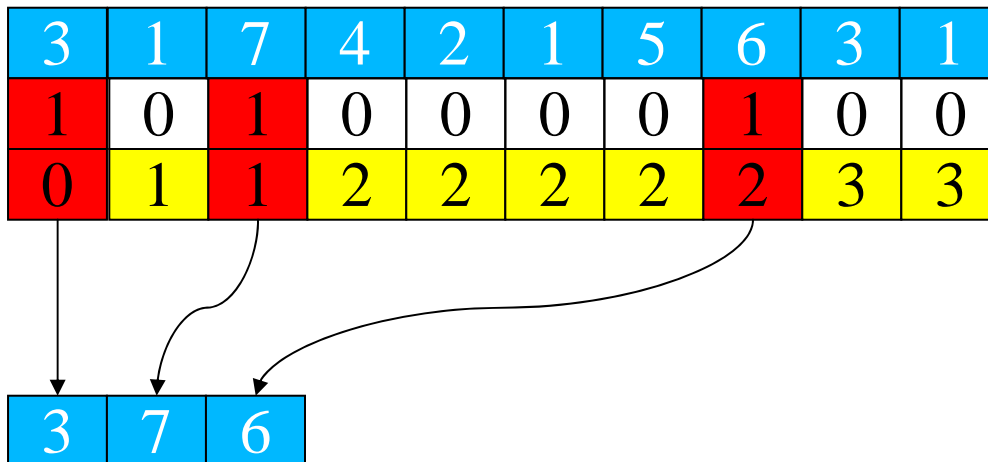
3	7	6
---	---	---

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

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