Computer Graphics

– Parallel Programming with Cuda –

Hendrik Lensch

Overview

- **So far:**
    - Introduction to Cuda
    - GPGPU via Cuda (general purpose computing on the GPU)
    - Block matrix-matrix multiplication

- **Today:**
    - Some parallel programming principles
    - Parallel Vector Reduction
    - Parallel Prefix Sum Calculation

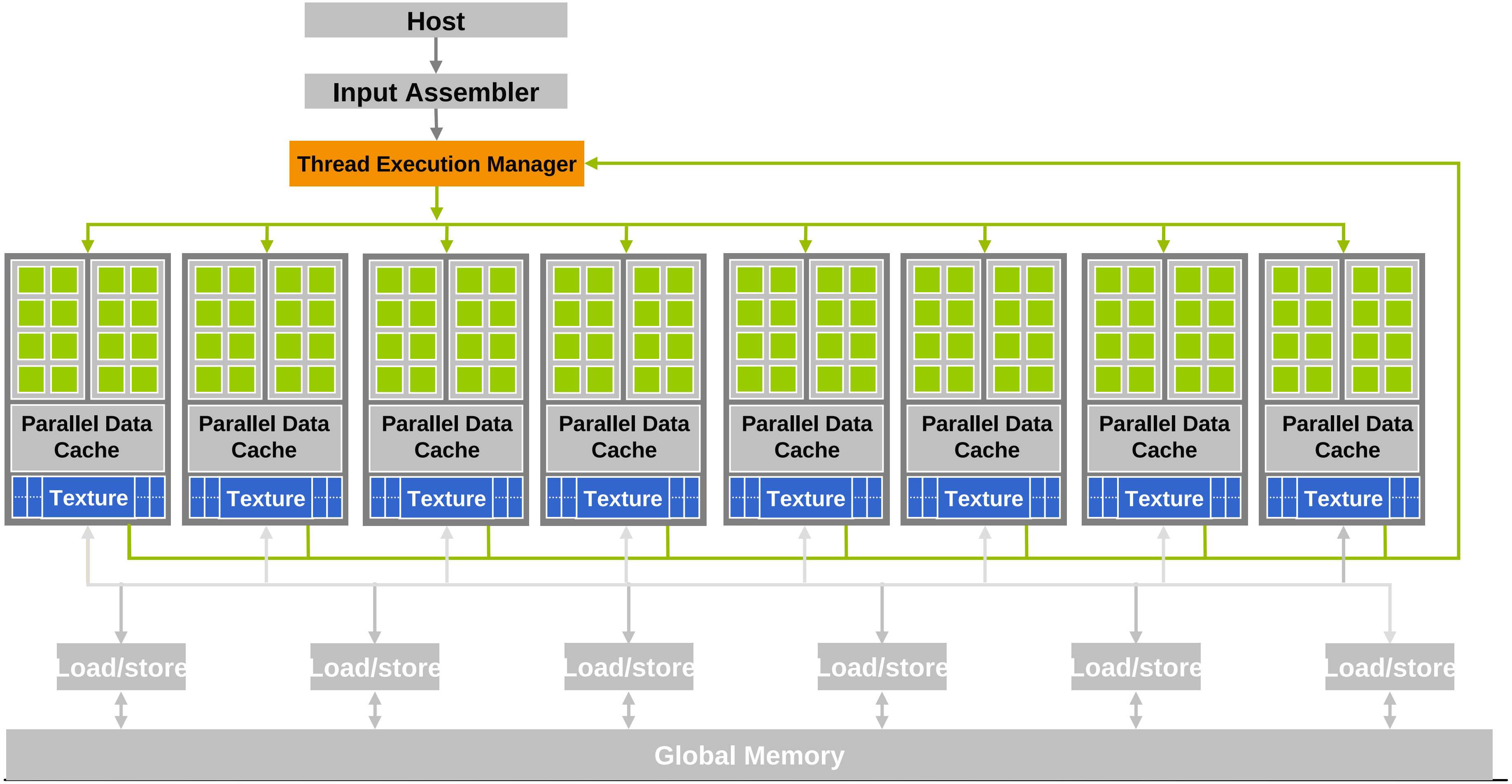
- **Next:**
    - No lectures on Monday
    - Input/Output devices

Resources

- **Where to find Cuda and the documentation?**
  - http://www.nvidia.com/object/cuda_home.html
- **Lecture on parallel programming on the GPU by David Kirk and Wen-mei W. Hwu (most of the following slides are copied from this course)**
  - http://courses.ece.uiuc.edu/ece498/al1/Syllabus.html
- **On the Parallel Prefix Sum (Scan) algorithm**
  - http://developer.download.nvidia.com/compute/cuda/sdk/website/projects/scan/doc/scan.pdf

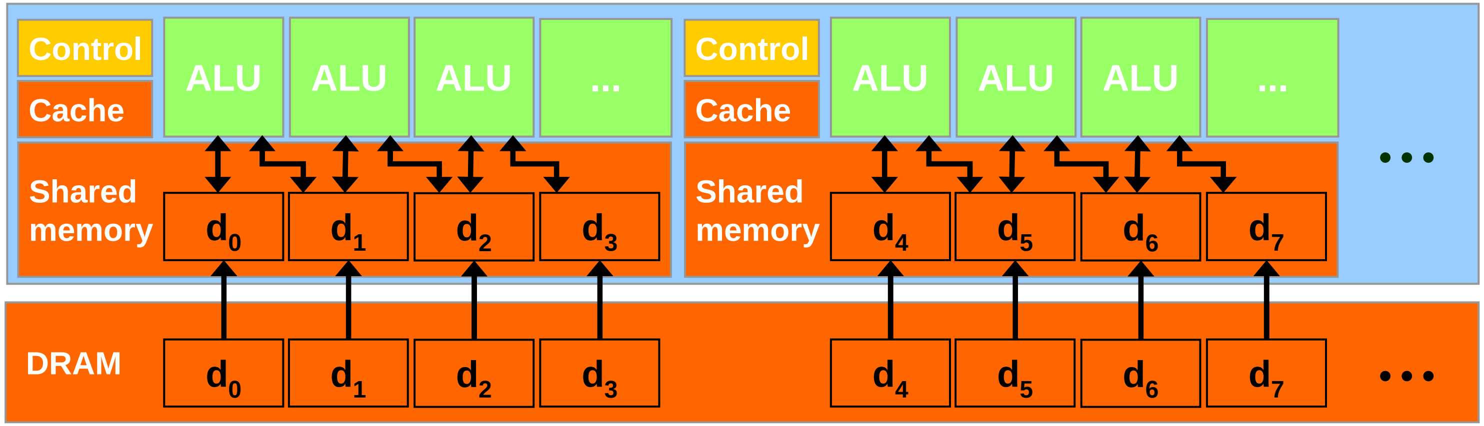
GeForce 8800

16 highly threaded SM's, >128 FPU's, 367 GFLOPS, 768 MB DRAM, 86.4 GB/S Mem BW, 4GB/S BW to CPU



CUDA Highlights: On-Chip Shared Memory

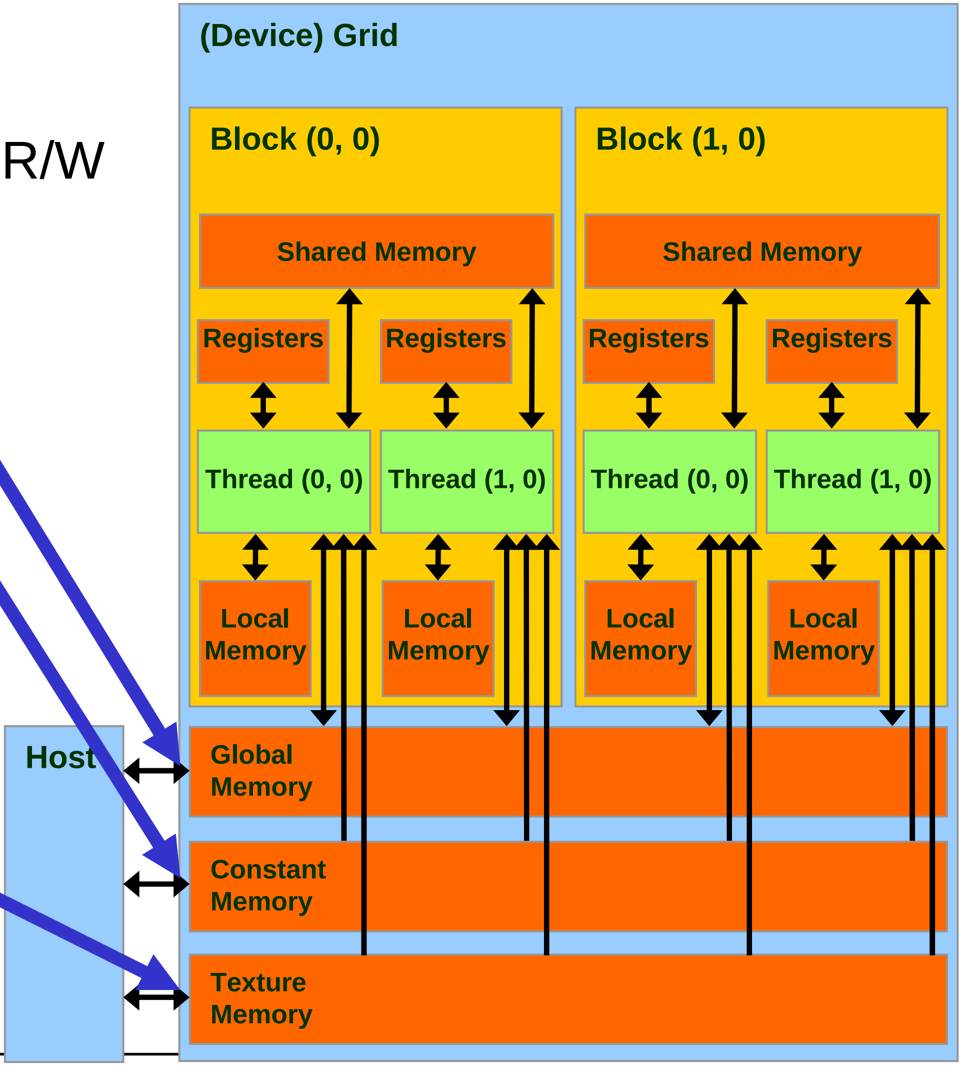
- **CUDA enables access to a parallel on-chip shared memory for efficient inter-thread data sharing**



→ **Big memory bandwidth savings**

Global, Constant, and Texture Memories (Long Latency Accesses)

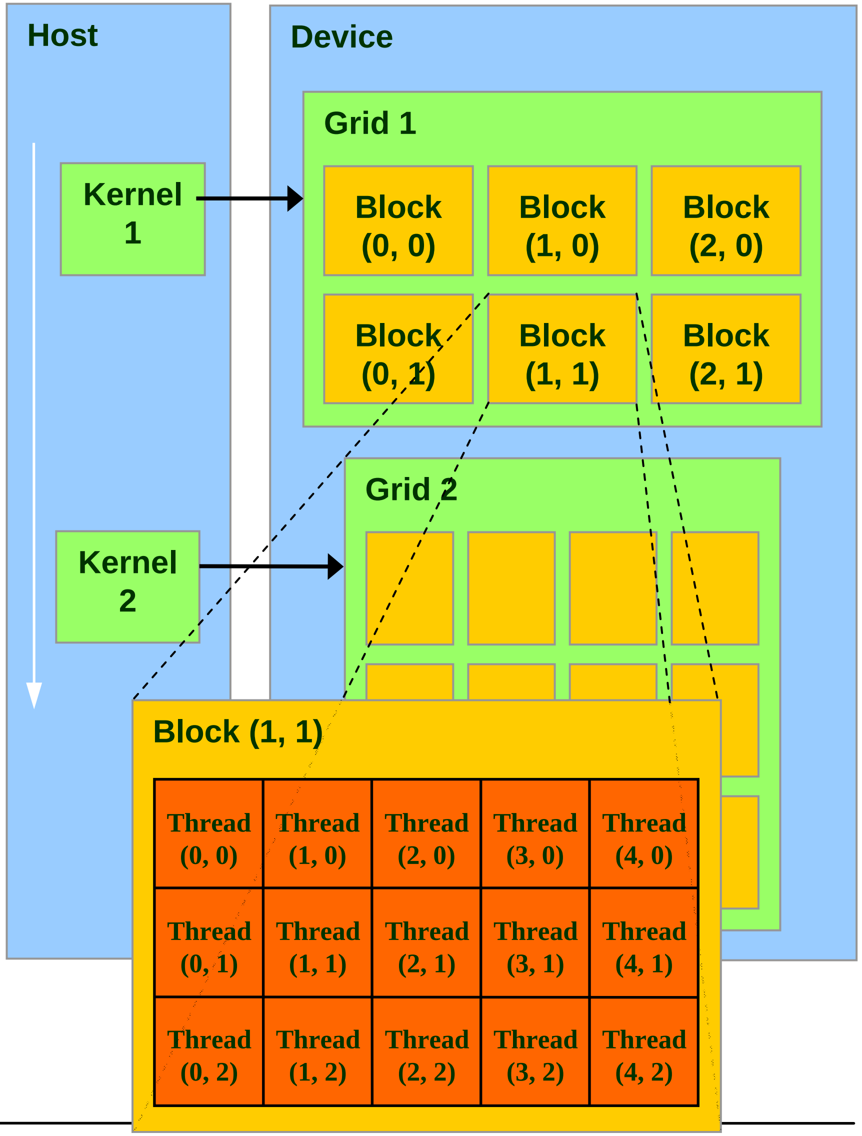
- **Global memory**
  - Main means of communicating R/W Data between host and device
  - Contents visible to all threads
- **Texture and Constant Memories**
  - Constants initialized by host
  - Contents visible to all threads



Courtesy: NDVIA

Thread Batching: Grids and Blocks

- **A kernel is executed as a grid of thread blocks**
  - All threads share data memory space
- **A thread block is a batch of threads that can cooperate with each other by:**
  - Synchronizing their execution
    - For hazard-free shared memory accesses
  - Efficiently sharing data through a low latency shared memory
- **Two threads from two different blocks cannot cooperate**



Courtesy: NDVIA

Quick Terminology Review

- ***Thread*: concurrent code and associated state executed on the CUDA device (in parallel with other threads)**
  - The unit of parallelism in CUDA

- ***Warp*: a group of threads executed *physically* in parallel in G80**

- ***Block*: a group of threads that are executed together and form the unit of resource assignment**

- ***Grid*: a group of thread blocks that must all complete before the next phase of the program can begin**

How Thread Blocks are Partitioned

- **Thread blocks are partitioned into warps**
  - Thread IDs within a warp are consecutive and increasing
  - Warp 0 starts with Thread ID 0

- **Partitioning is always the same**
  - Thus you can use this knowledge in control flow
  - However, the exact size of warps may change from generation to generation
  - (Covered next)

- **However, DO NOT rely on any ordering between warps**
  - If there are any dependencies between threads, you must `__syncthreads()` to get correct results

Control Flow Instructions

- **Main performance concern with branching is divergence**
  - Threads within a single warp take different paths
  - Different execution paths are serialized in G80
    - The control paths taken by the threads in a warp are traversed one at a time until there is no more.
- **A common case: avoid divergence when branch condition is a function of thread ID**
  - Example with divergence:
    - `If (threadIdx.x > 2) { }`
    - This creates two different control paths for threads in a block
    - Branch granularity < warp size; threads 0 and 1 follow different path than the rest of the threads in the first warp
  - Example without divergence:
    - `If (threadIdx.x / WARP_SIZE > 2) { }`
    - Also creates two different control paths for threads in a block
    - Branch granularity is a whole multiple of warp size; all threads in any given warp follow the same path

Shared Memory Bank Conflicts

- **Shared memory is as fast as registers if there are no bank conflicts**

- **The fast case:**
  - If all threads of a half-warp access different banks, there is no bank conflict
  - If all threads of a half-warp access the identical address, there is no bank conflict (broadcast)
- **The slow case:**
  - Bank Conflict: multiple threads in the same half-warp access the same bank
  - Must serialize the accesses
  - Cost = max # of simultaneous accesses to a single bank

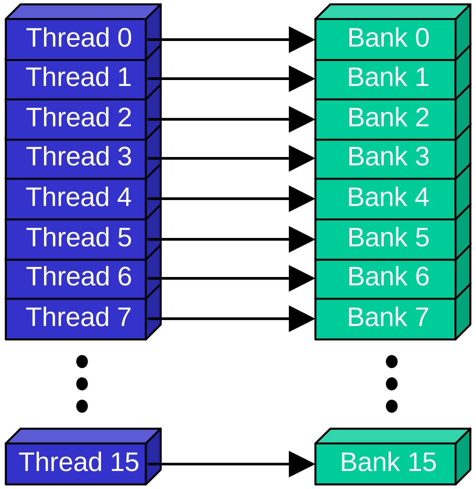
Linear Addressing

- **Given:**

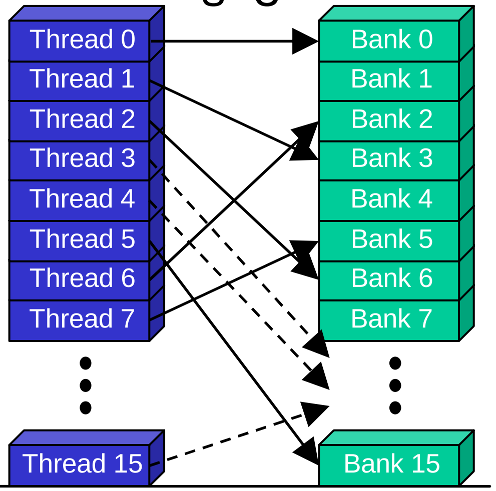
```
__shared__ float shared[256];
float foo =
  shared[baseIndex + s * threadIdx.x];
```

- **This is only bank-conflict-free if s shares no common factors with the number of banks**
  - 16 on G80, so s must be odd

s=1



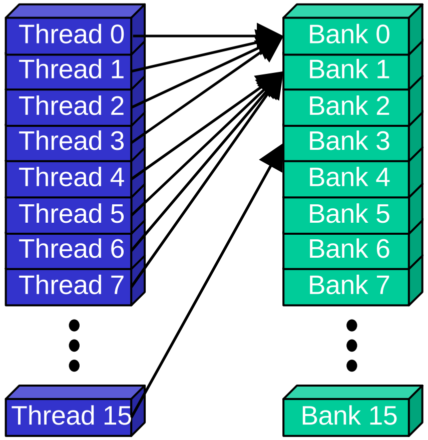
s=3



Data Types and Bank Conflicts

- **This has no conflicts if type of shared is 32-bits:**

```
foo = shared[baseIndex + threadIdx.x]
```

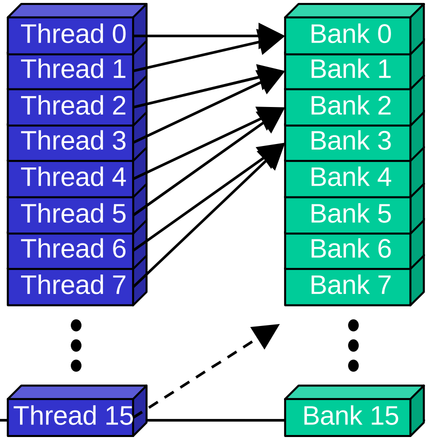


- **But not if the data type is smaller**
  - 4-way bank conflicts:

```
__shared__ char shared[];
foo = shared[baseIndex + threadIdx.x];
```

  - 2-way bank conflicts:

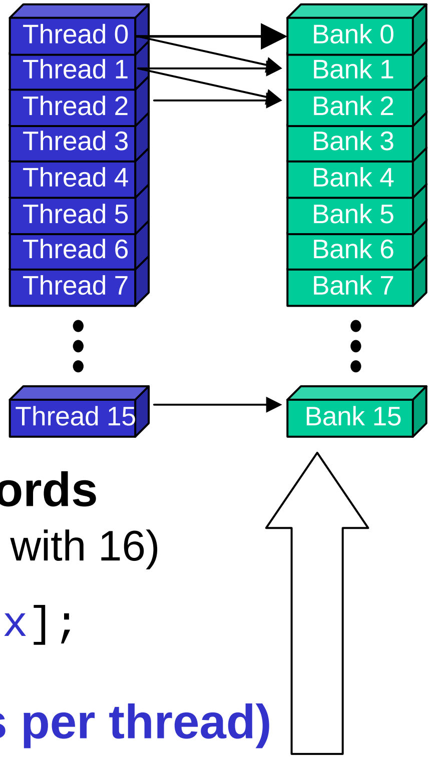
```
__shared__ short shared[];
foo = shared[baseIndex + threadIdx.x];
```



Structs and Bank Conflicts

- **Struct assignments compile into as many memory accesses as there are struct members:**

```
struct vector { float x, y, z; };
struct myType {
    float f;
    int c;
};
__shared__ struct vector vectors[64];
__shared__ struct myType myTypes[64];
```



- **This has no bank conflicts for vector; struct size is 3 words**
  - 3 accesses per thread, contiguous banks (no common factor with 16)

```
struct vector v = vectors[baseIndex + threadIdx.x];
```

- **This has 2-way bank conflicts for my Type; (2 accesses per thread)**

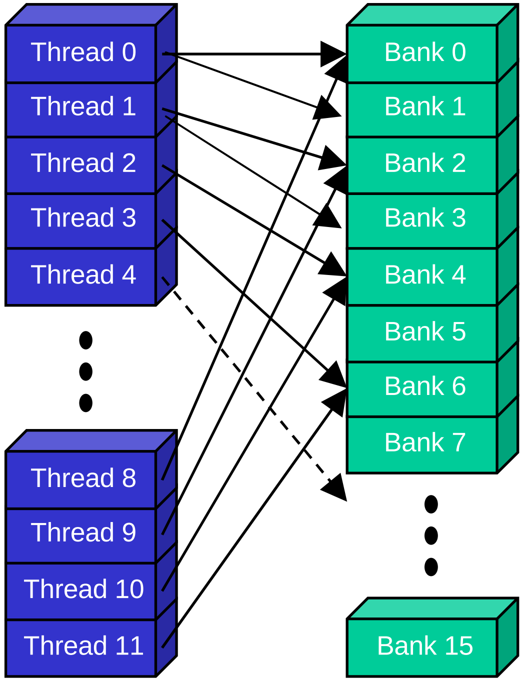
```
struct myType m = myTypes[baseIndex + threadIdx.x];
```

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];
```

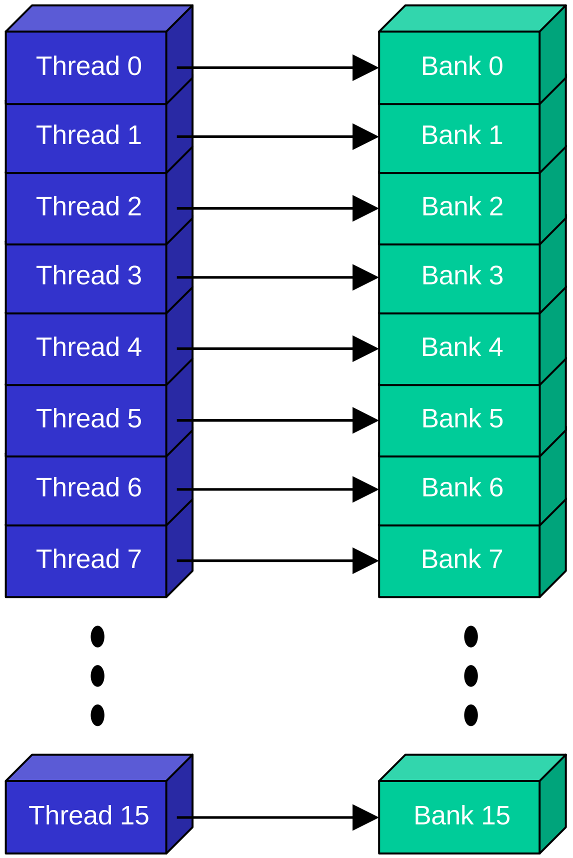
- **This makes sense for traditional CPU threads, locality in cache line usage and reduced sharing traffice.**
  - Not in shared memory usage where there is no cache line effects but banking effects



A Better Array Access Pattern

- **Each thread loads one element in every consecutive group of bockDim elements.**

```
shared[tid] = global[tid];
shared[tid + blockDim.x] =
  global[tid + blockDim.x];
```



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

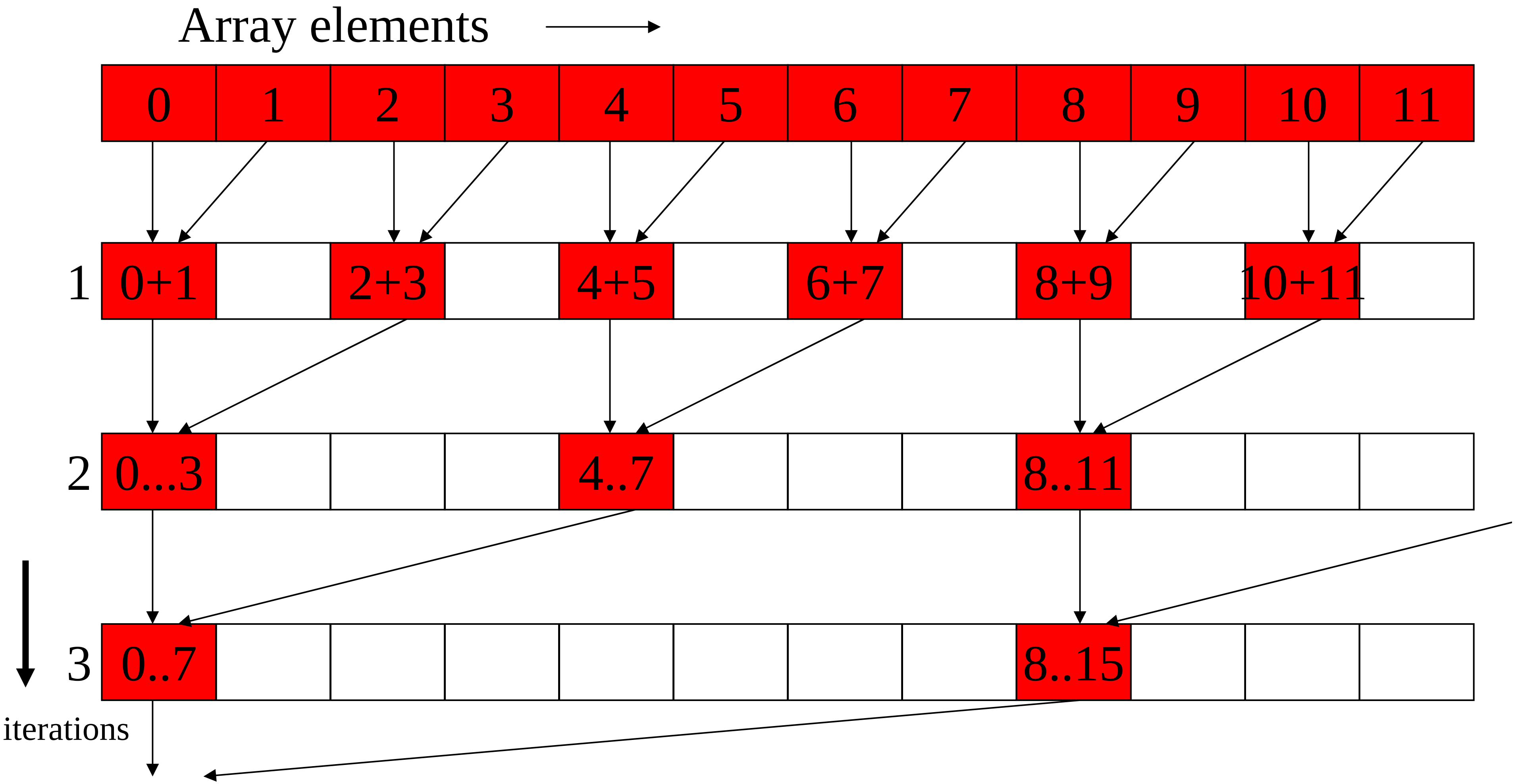
- **Assume an in-place reduction using shared memory**
  - The original vector is in device global memory
  - The shared memory used to hold a partial sum vector
  - Each iteration brings the partial sum vector closer to the final sum
  - The final solution will be in element 0

A Simple Implementation

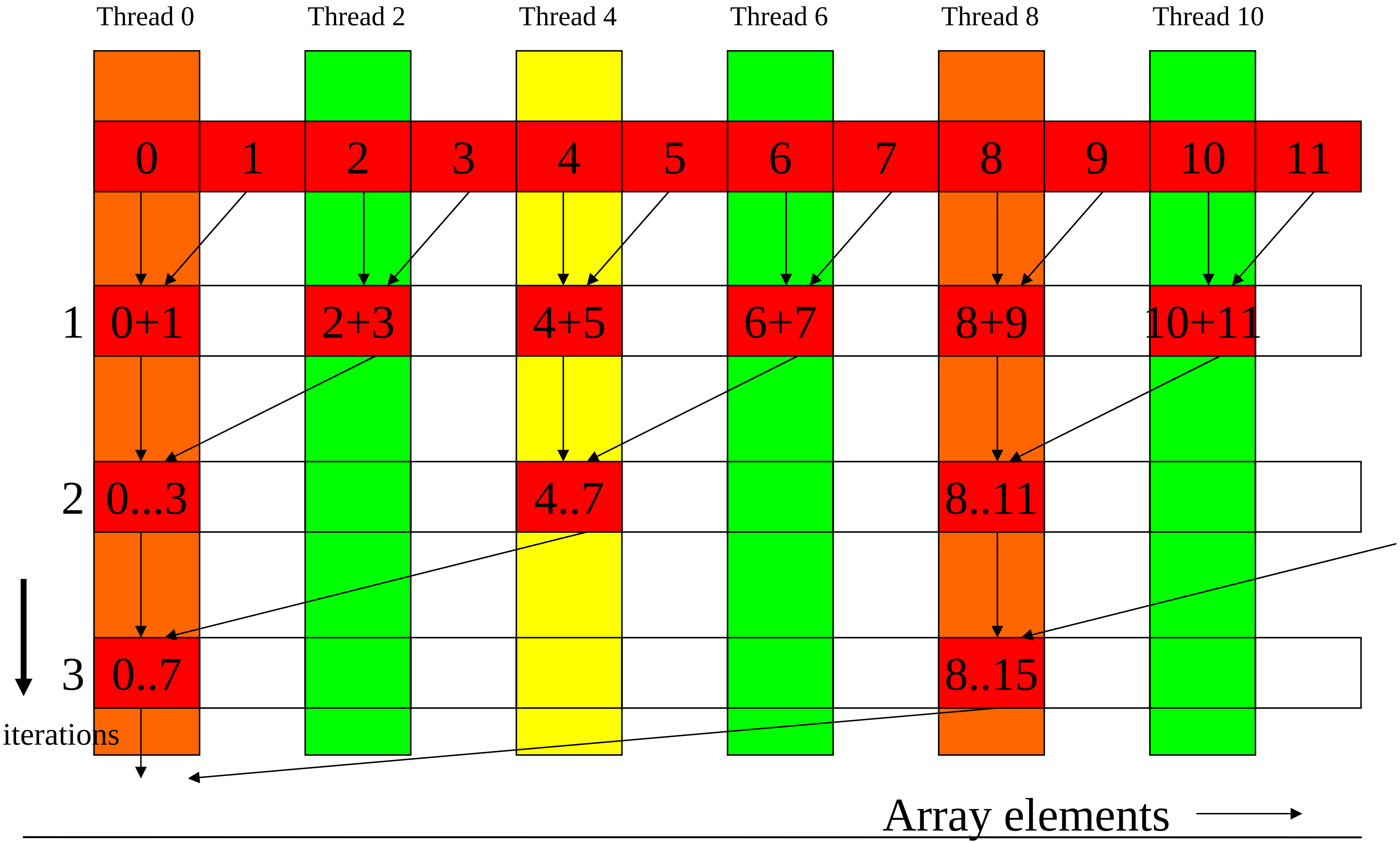
- **Assume we have already loaded array into**
  - `__shared__ float partialSum[]`

```
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 with Bank Conflicts



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

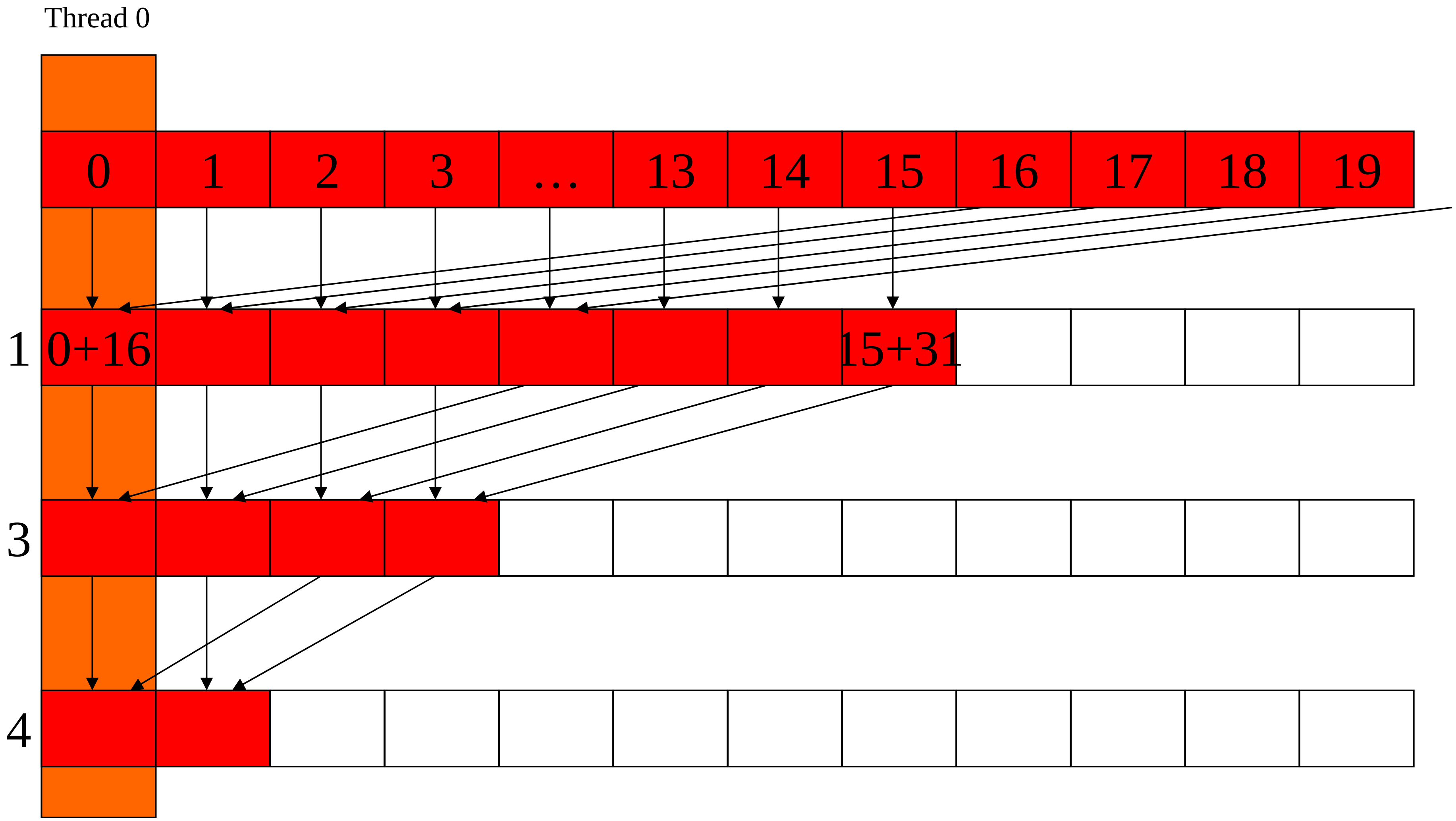
BAD: Bank conflicts due to stride

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

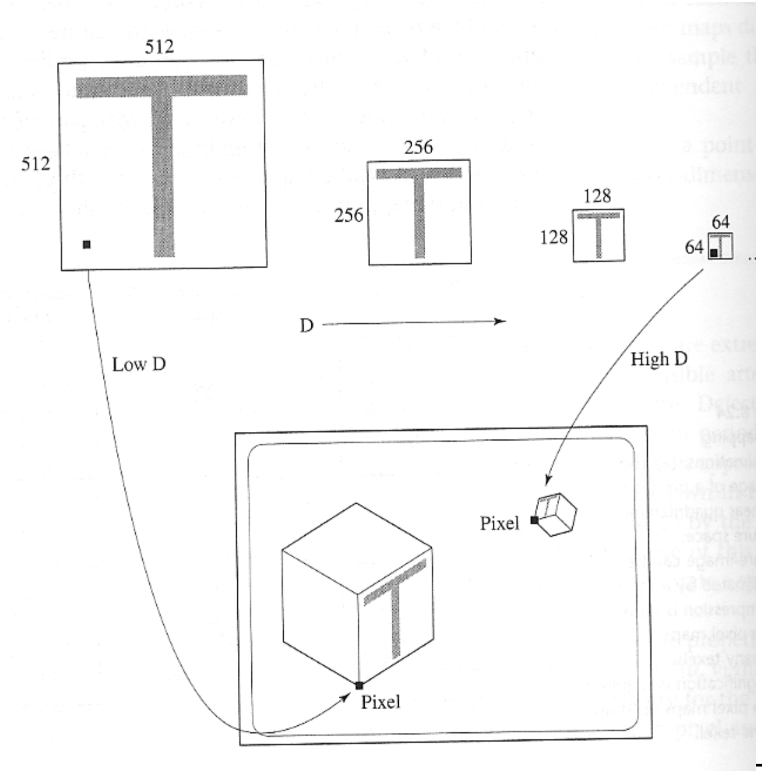


Observations About the New Implementation

- **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 warps in each block
  - Better resource utilization, will likely retire warps and thus blocks faster
- **Recall, no bank conflicts either**

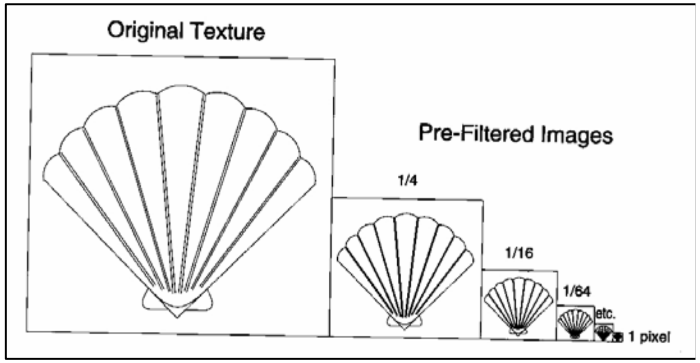
Application: MipMap Construction

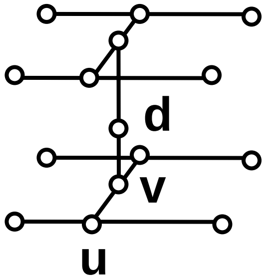
- **Texture available in multiple resolutions**
  - Pre-processing step
- **Rendering: select appropriate texture resolution**
  - Selection is usually per pixel !!
  - Texel size(n) < extent of pixel footprint < texel size(n+1)

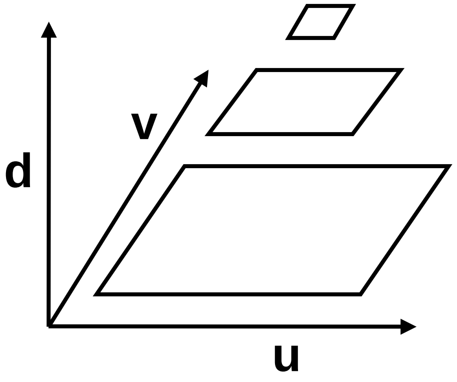


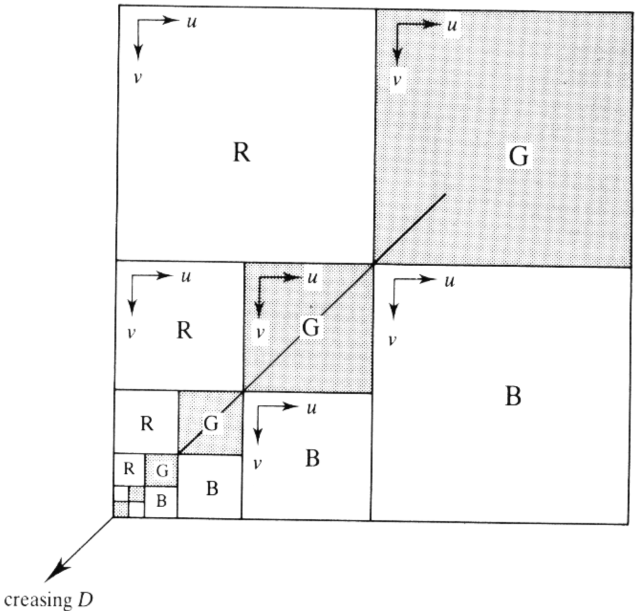
Application: MipMapping II

- **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









Scan – 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

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

  and returns the ordered set

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

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

- **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]

(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);
```

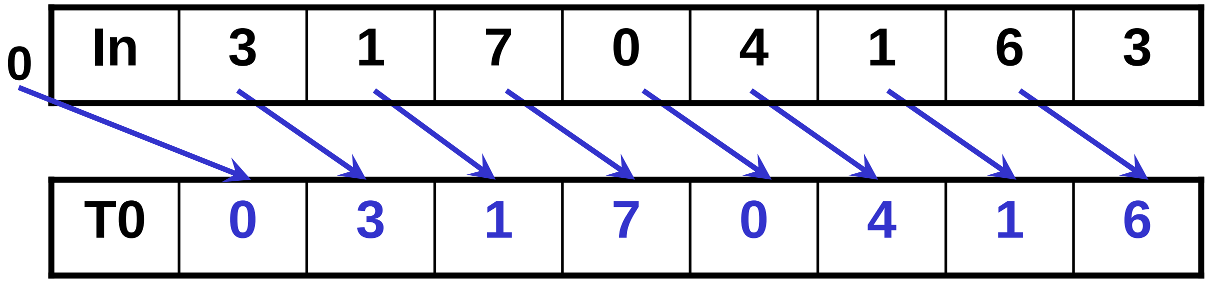
- **Useful for many parallel algorithms:**
  - radix sort
  - quicksort
  - String comparison
  - Lexical analysis
  - Stream compaction
  - Polynomial evaluation
  - Solving recurrences
  - Tree operations
  - Histograms
  - Etc.

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

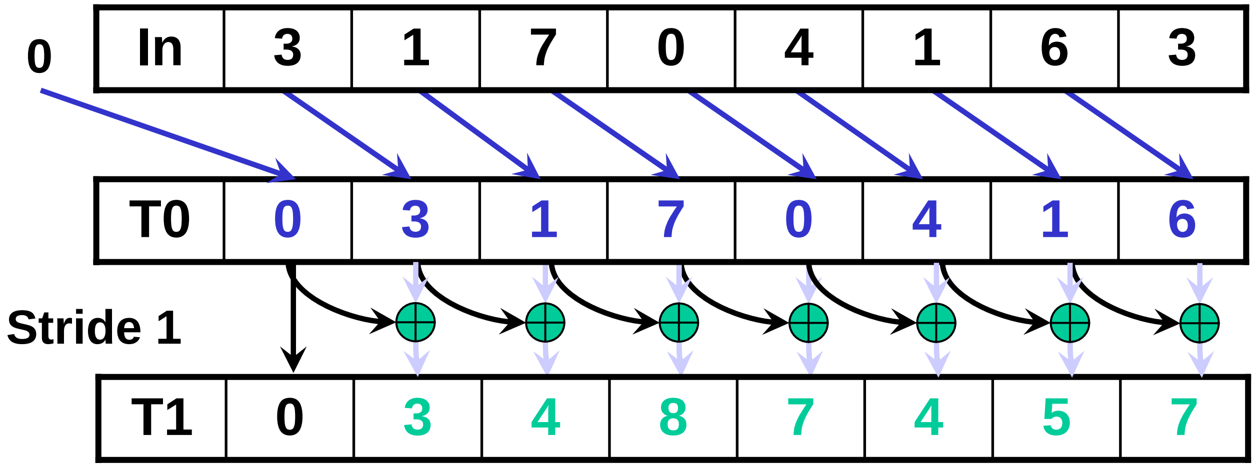


In	**3**	**1**	**7**	**0**	**4**	**1**	**6**	**3**
T0	**0**	**3**	**1**	**7**	**0**	**4**	**1**	**6**

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



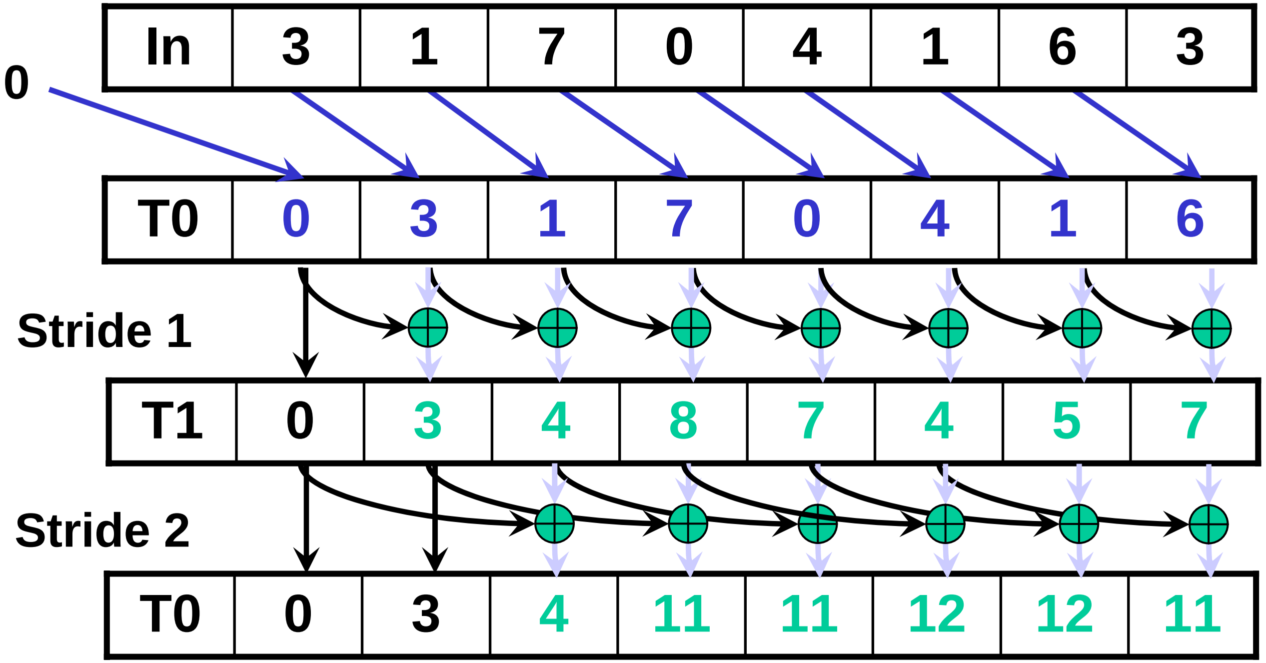
In	**3**	**1**	**7**	**0**	**4**	**1**	**6**	**3**
T0	**0**	**3**	**1**	**7**	**0**	**4**	**1**	**6**
T1	**0**	**3**	**4**	**8**	**7**	**4**	**5**	**7**

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)

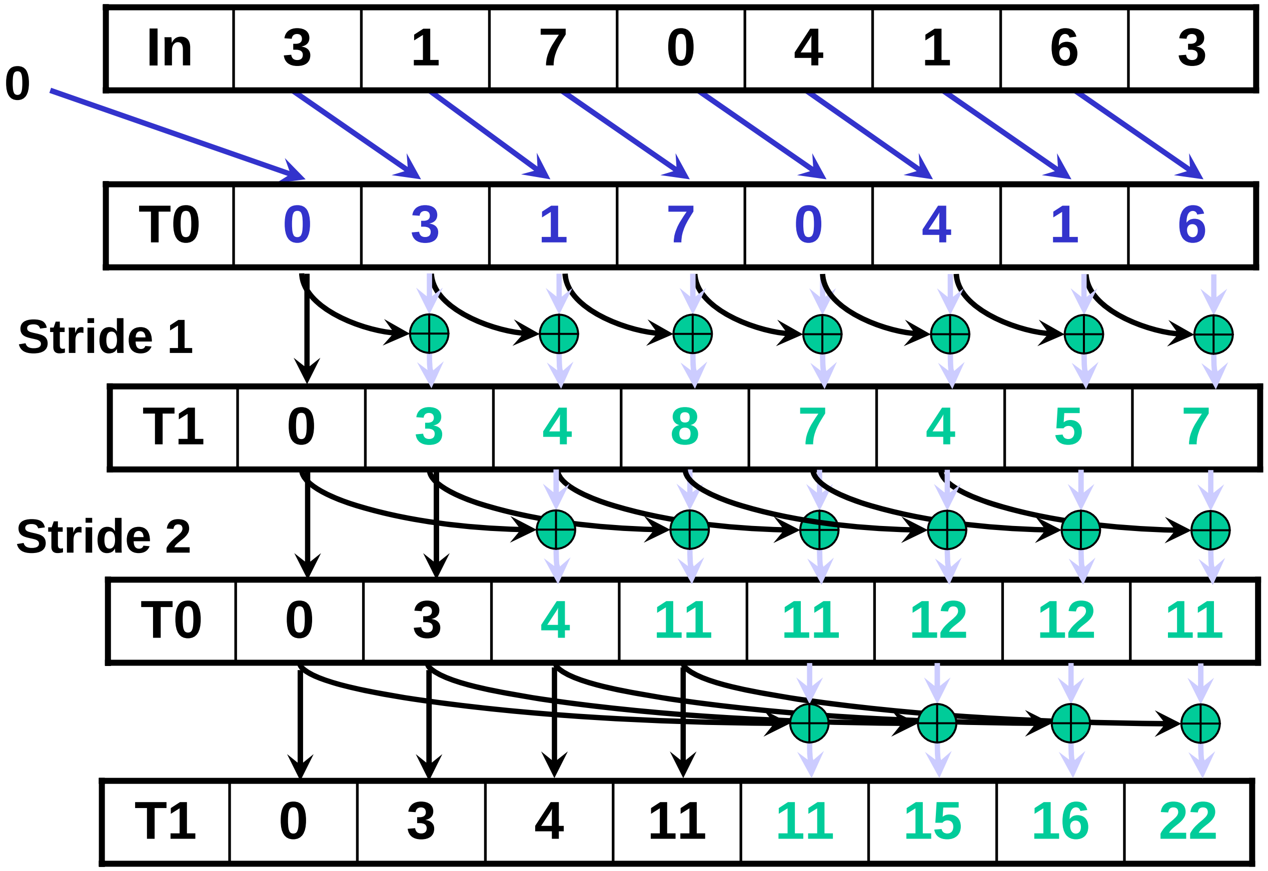
A First-Attempt Parallel Scan Algorithm



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

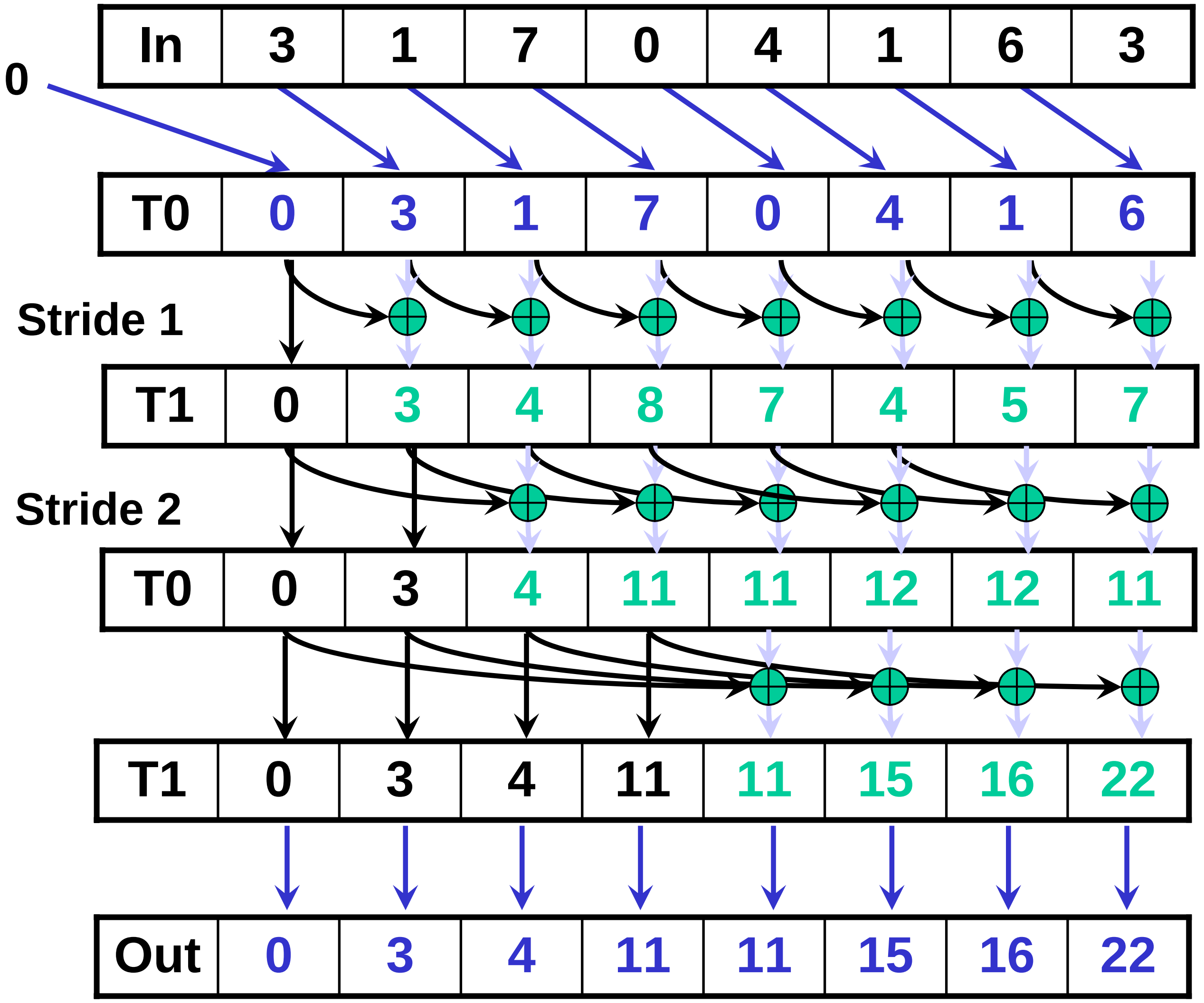


In	**3**	**1**	**7**	**0**	**4**	**1**	**6**	**3**
T0	**0**	**3**	**1**	**7**	**0**	**4**	**1**	**6**
T1	**0**	**3**	**4**	**8**	**7**	**4**	**5**	**7**
T0	**0**	**3**	**4**	**11**	**11**	**12**	**12**	**11**
T1	**0**	**3**	**4**	**11**	**11**	**15**	**16**	**22**

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



In	3	1	7	0	4	1	6	3
T0	0	3	1	7	0	4	1	6
T1	0	3	4	8	7	4	5	7
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**
  - 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 → 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

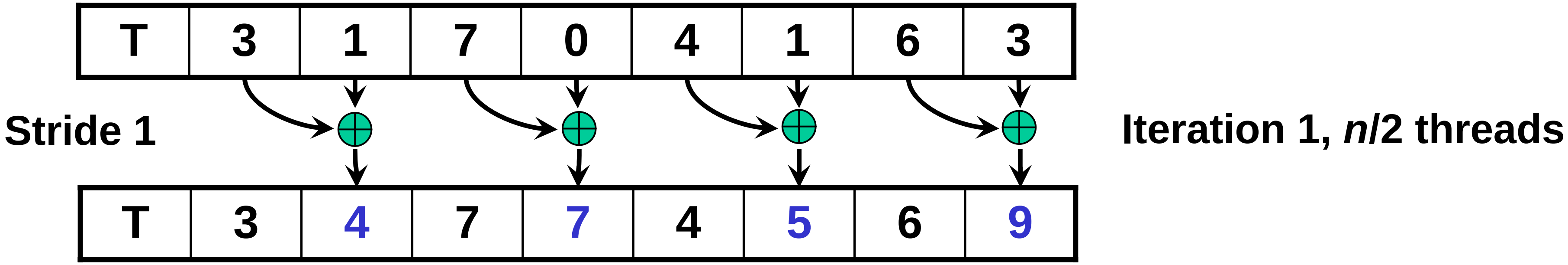
- **For scan:**
  - Traverse down from leaves to root building partial sums at internal nodes in the tree
    - Root holds sum of all leaves
  - Traverse back up the tree building the scan from the partial sums

Build the Sum Tree

T	**3**	**1**	**7**	**0**	**4**	**1**	**6**	**3**

Assume array is already in shared memory

Build the Sum Tree



T	3	1	7	0	4	1	6	3
T	3	4	7	7	4	5	6	9

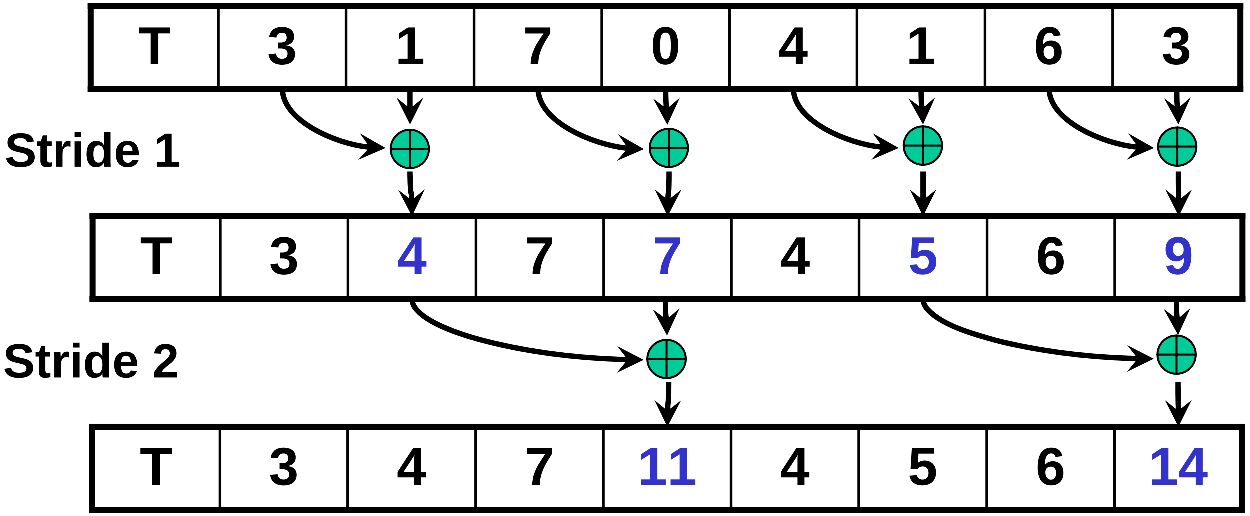
Stride 1

Iteration 1, *n*/2 threads

Each ⊕ corresponds to a single thread.

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

Build the Sum Tree



T	3	1	7	0	4	1	6	3

Stride 1

T	3	4	7	7	4	5	6	9

Stride 2

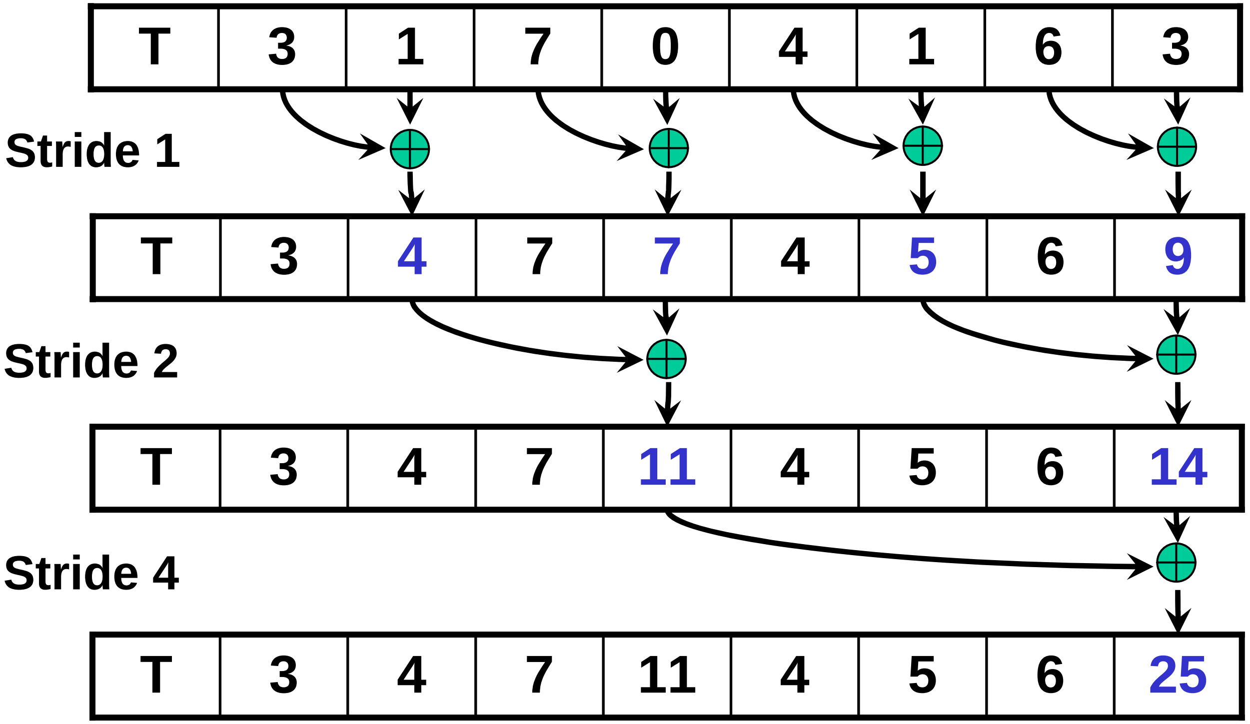
T	3	4	7	11	4	5	6	14

Iteration 2, *n*/4 threads

Each ⊕ corresponds to a single thread.

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

Build the Sum Tree



T	3	1	7	0	4	1	6	3
T	3	4	7	7	4	5	6	9
T	3	4	7	11	4	5	6	14
T	3	4	7	11	4	5	6	25

Iteration log(*n*), 1 thread

Each ⊕ corresponds to a single thread.

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

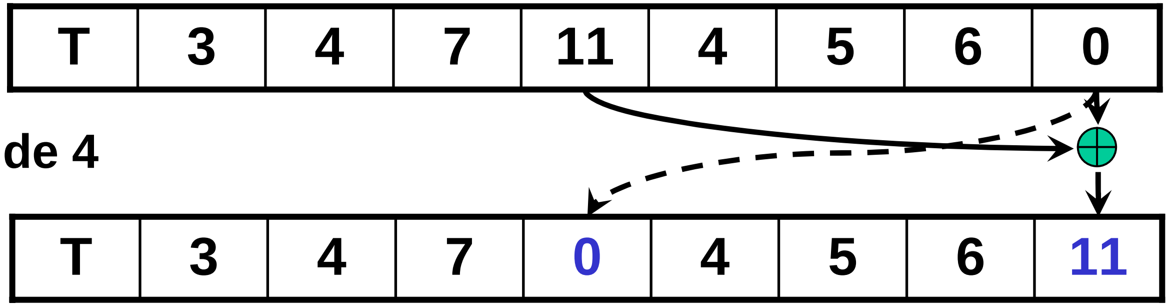
T	**3**	**4**	**7**	**11**	**4**	**5**	**6**	**0**

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



T	3	4	7	11	4	5	6	0

Stride 4

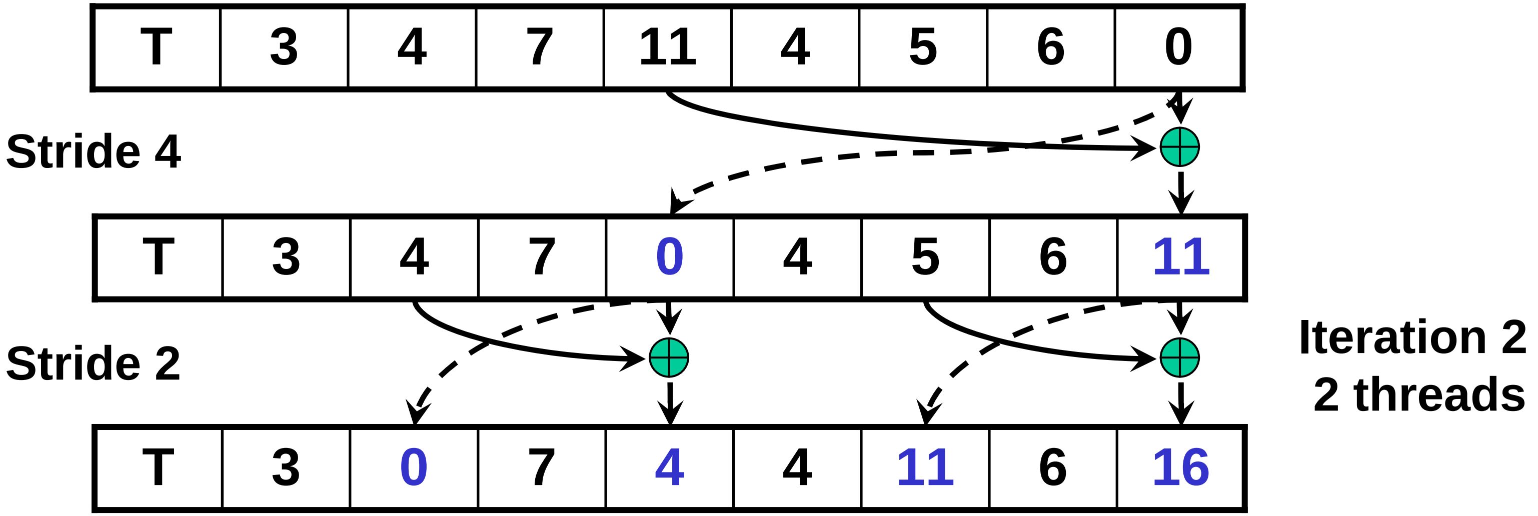
T	3	4	7	0	4	5	6	11

Iteration 1
1 thread

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



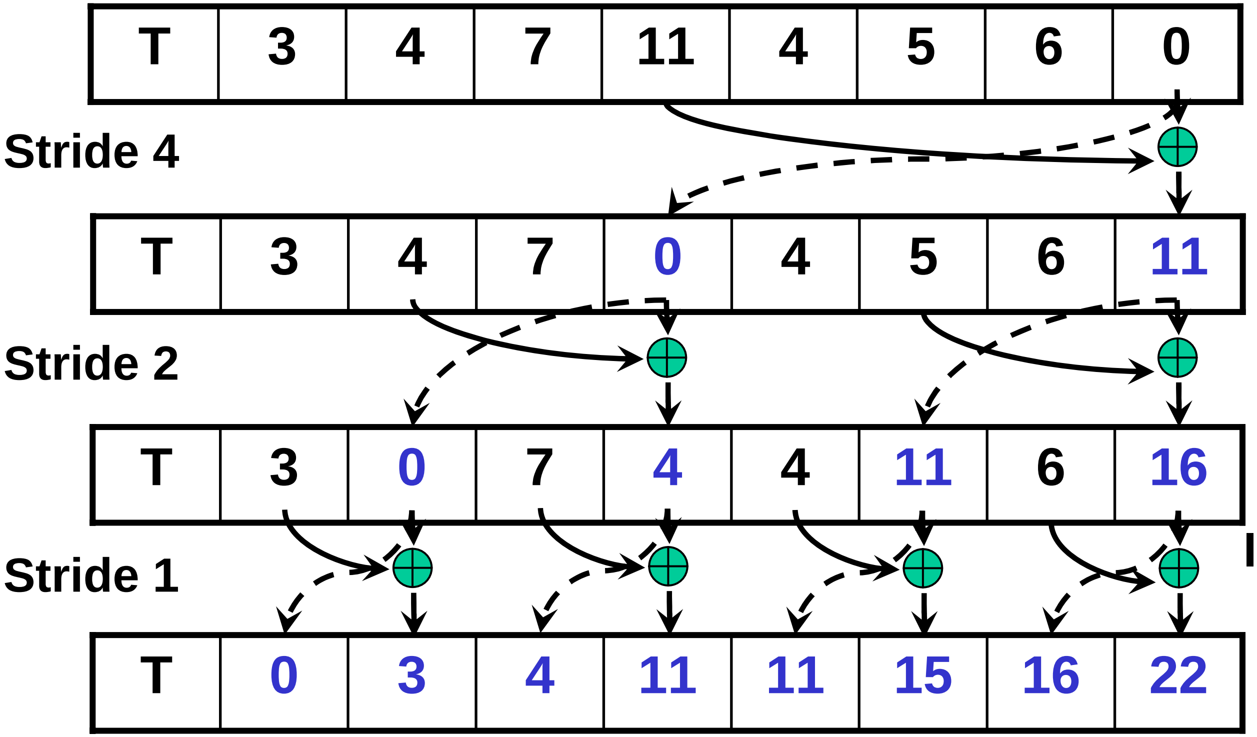
T	3	4	7	11	4	5	6	0
T	3	4	7	0	4	5	6	11
T	3	0	7	4	4	11	6	16

Iteration 2
2 threads

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



Iteration log(n)
n/2 threads

Each ⊕ corresponds to a single thread.

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!**

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