Introduction to Parallel & Distributed Computing

GPU Programming with HIP (3)

Atomics & Parallel Patterns: Reductions, Histograms and Prefix Scans

Lecture 9, Spring 2024

Instructor: 罗国杰

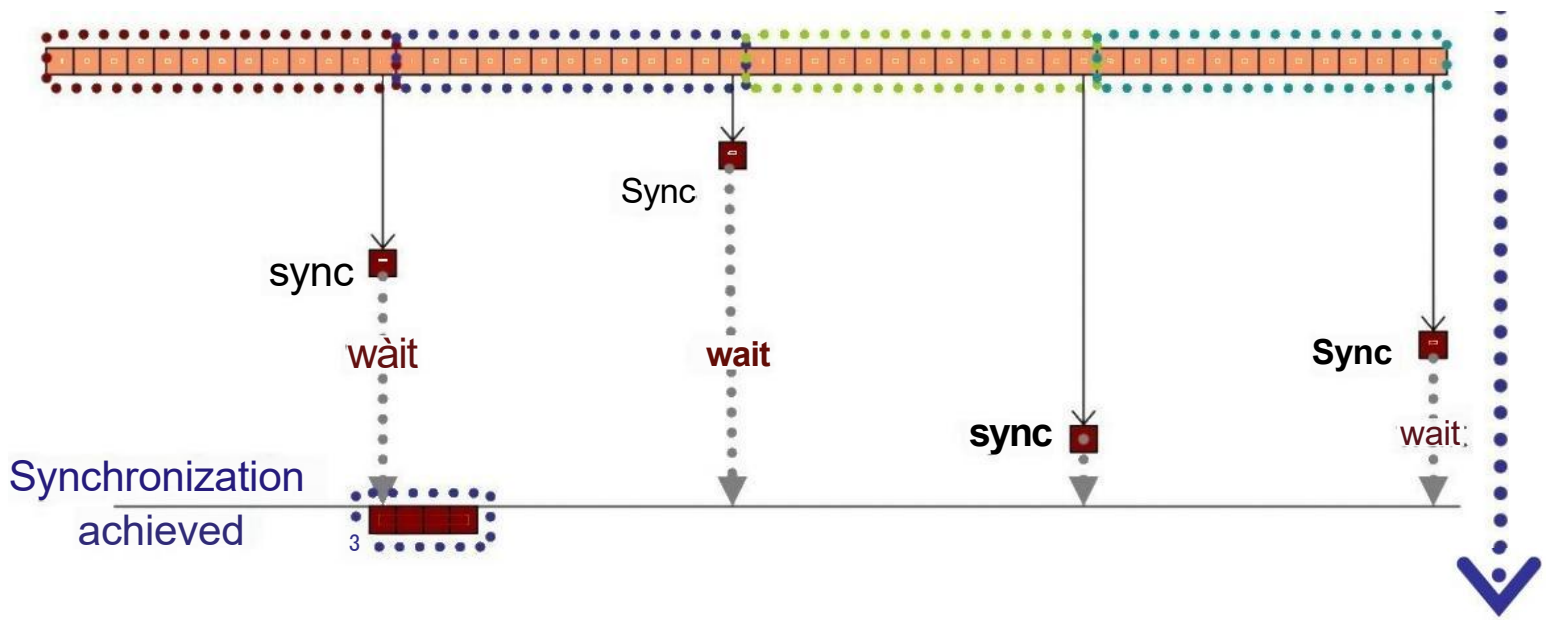
gluo@pku.edu.cn

Adapted from UW CSEP 590B, Dr. Hari Sadasivan
https://courses.cs.washington.edu/courses/csep590b/

Global Synchronization

Synchronization of the blocks is needed:

- Multiple kernels
- Atomic operations in global memory



adapted from UW CSEP 590, Dr. Hari Sadasivan

Atomic Operations (I)

- HIP provides atomic instructions on shared memory and global memory
  - They perform read-modify-write operations atomically
  - Multiple instructions are serialized in undefined order

- Arithmetic functions
  - Add, Sub, Max, Min, Exch, Inc, Dec, CAS

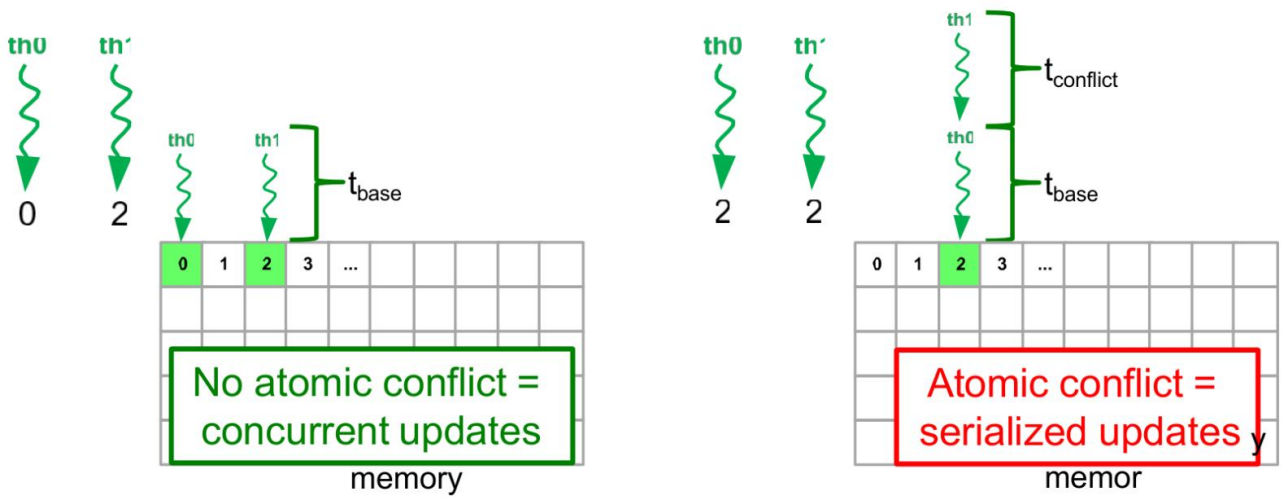
- Bitwise functions
  - And, Or, Xor

- Data types: int, uint, ull, float (half,single,double)*

adapted from UW CSEP 590, Dr. Hari Sadasivan

Atomic Operations (II)

Atomic operations serialize the execution if there are atomic conflicts



adapted from UW CSEP 590, Dr. Hari Sadasivan

Uses of Atomic Operations

- **Computation**
  - Atomics on an array that will be the output of the kernel
  - Example: Histogram, reduction

- **Synchronization**
  - Atomics on memory locations that are used for synchronization or coordination
  - Example: Counters, locks, flags…

- Use them to prevent data races when more than one thread need to update the same memory location

Atomic Operations (III)

- Atomic hardware is optimized for integers.
- FP atomics rely on compare-and-swap (CAS) loop which is slow
- Some HIP devices support fast atomics on FP using hardware Read-Modify-Write (RMW) instruction.

  (-munsafe-fp-atomics)

  • May produce incorrect results

Reduction

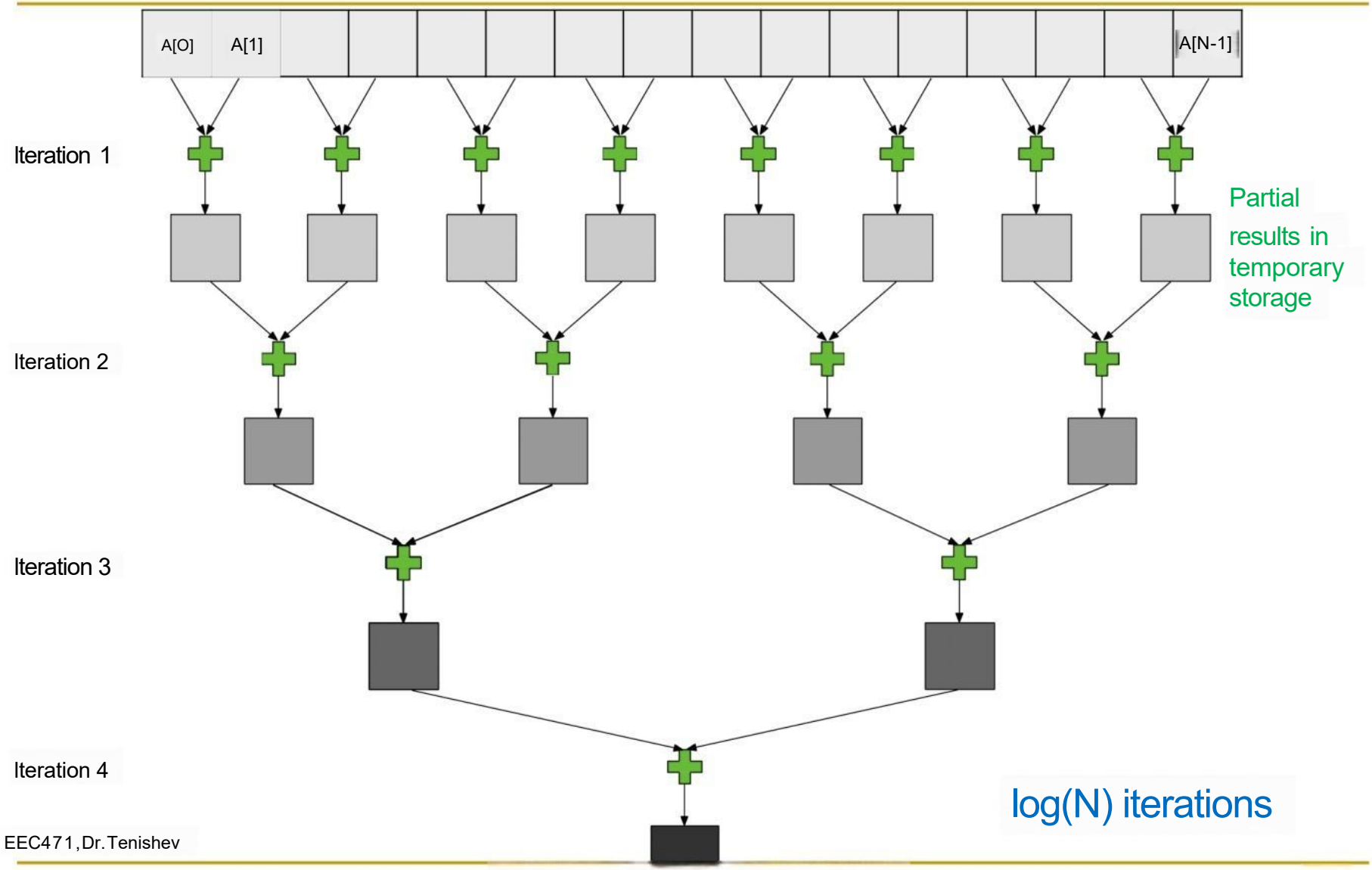
- Reduce a scalar array into a single element

- Reduction operator must be:
  - Associative: (A o B) o C = A o (B o C)
  - Commutative: A o B = B o A
  - Has a null/identity operator: A o 0 = A
  - Examples: Add, Multiply, Min, Max

- atomicAdd is sequential.
  - Use tree-based reduction within each block and share data between threads
  - Do atomicAdds at the very end using 1 thread per block
  - OR Launch a second reduction kernel to reduce this partially reduced list

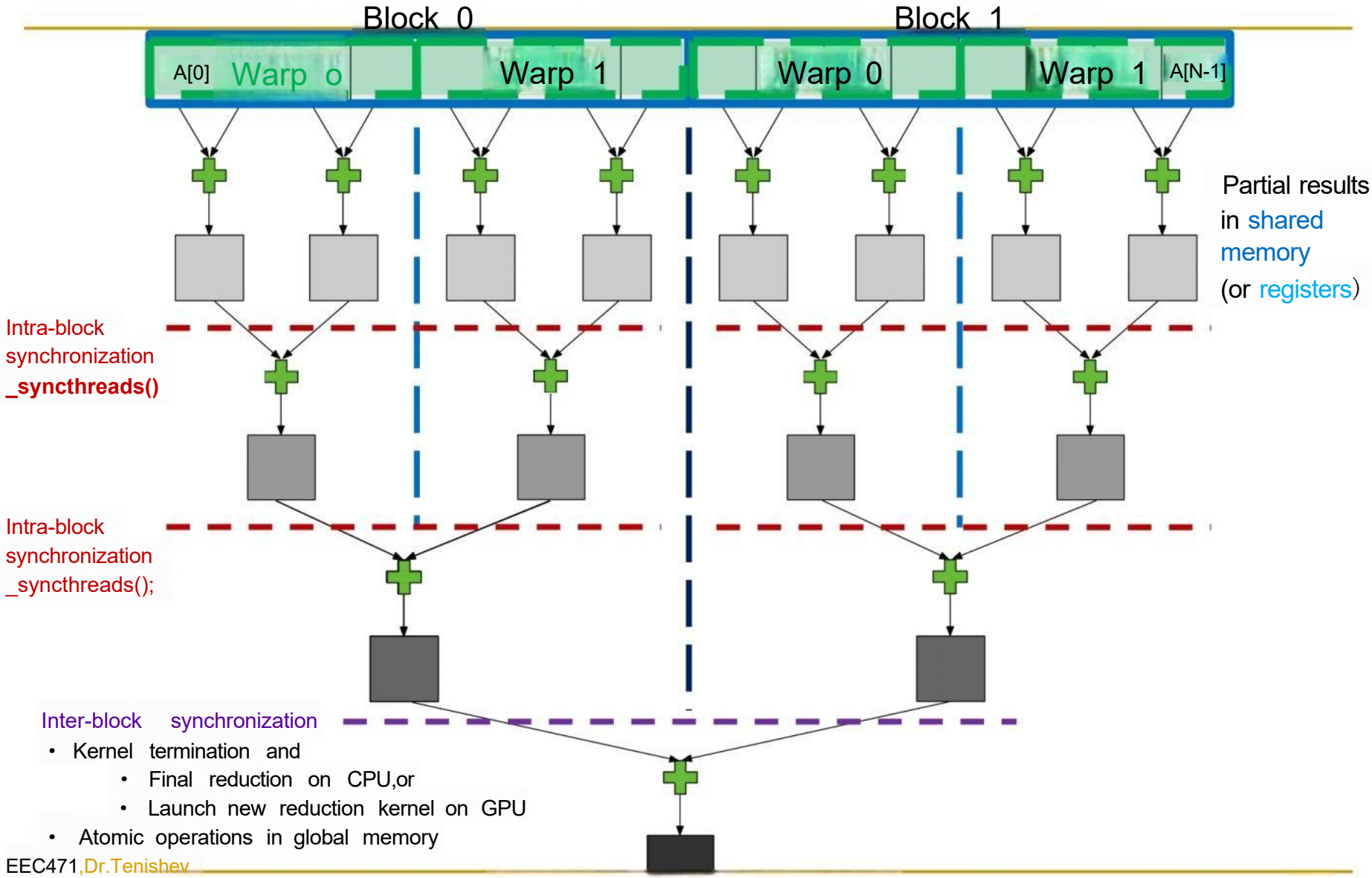
Courtesy: ROCm Training from Prof. David Kaeli

Tree-Based Reduction



adapted from UW CSEP 590, Dr. Hari Sadasivan

Tree-Based Reduction on GPU

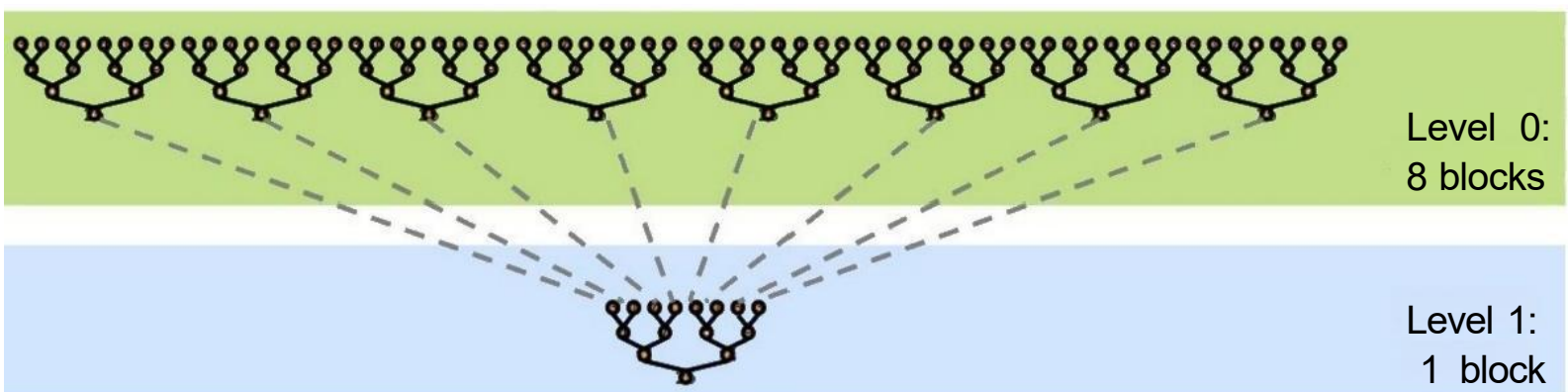


adapted from UW CSEP 590, Dr. Hari Sadasivan

Reduction: Big Picture

The code for all levels is the same

The same kernel code can be called multiple times



adapted from UW CSEP 590, Dr. Hari Sadasivan

Optimization Goal

Two components:

- Compute Bandwidth: GFLOPs
- Memory Bandwidth: GB/s

Reductions typically have low arithmetic intensity

- Memory bound

Naive reduction: Strategy

- Load data:

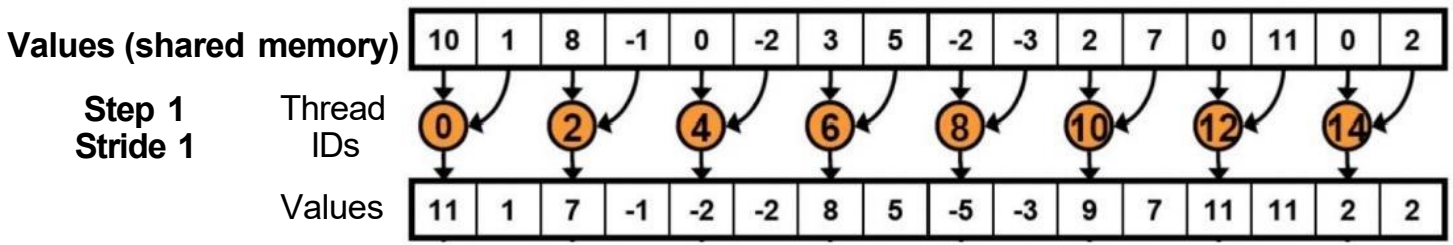
  - Each thread loads one element from global memory to shared memory

- Actual Reduction: Proceed in log (N) steps

  - A thread reduces two elements
    - The first two elements in the first thread
    - The next two will be by the next thread
    - And so on
  - At the end of each step:
    - Deactivate half of the threads
  - Terminate: when one thread left

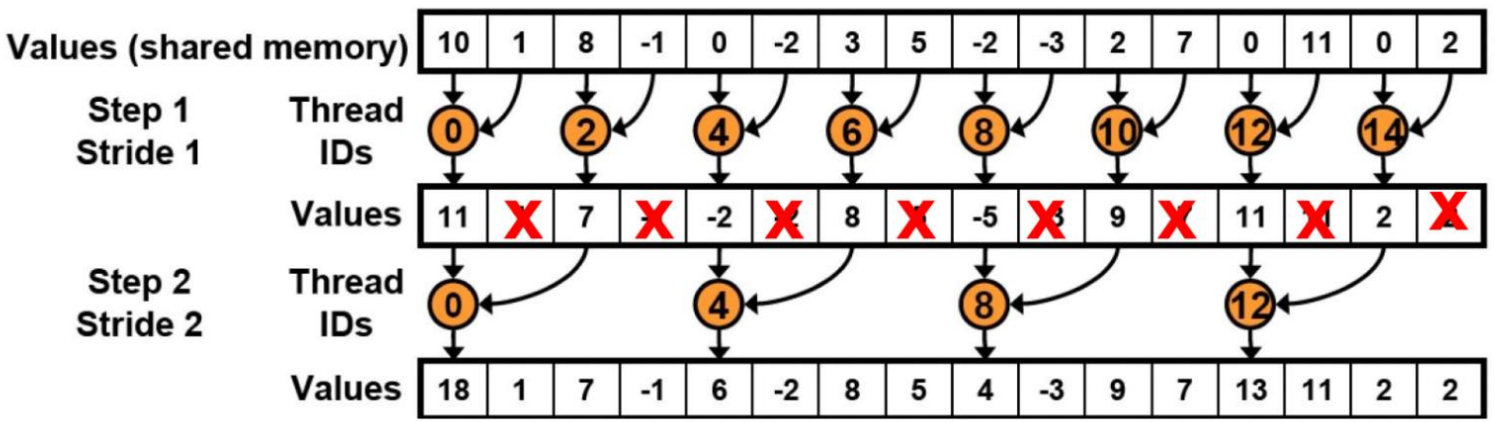
  Write back to global memory

Step 1: Interleaved Addressing



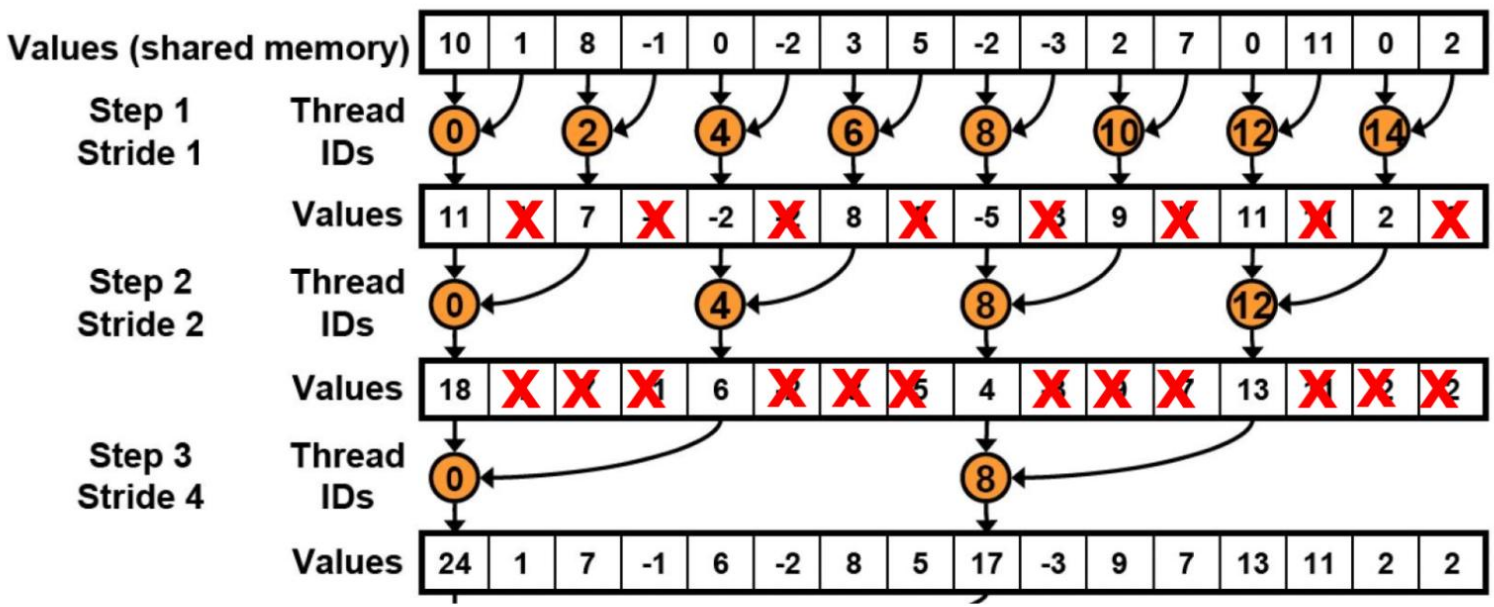
Picture courtesy: EECS471, UM

Step 1: Interleaved Addressing



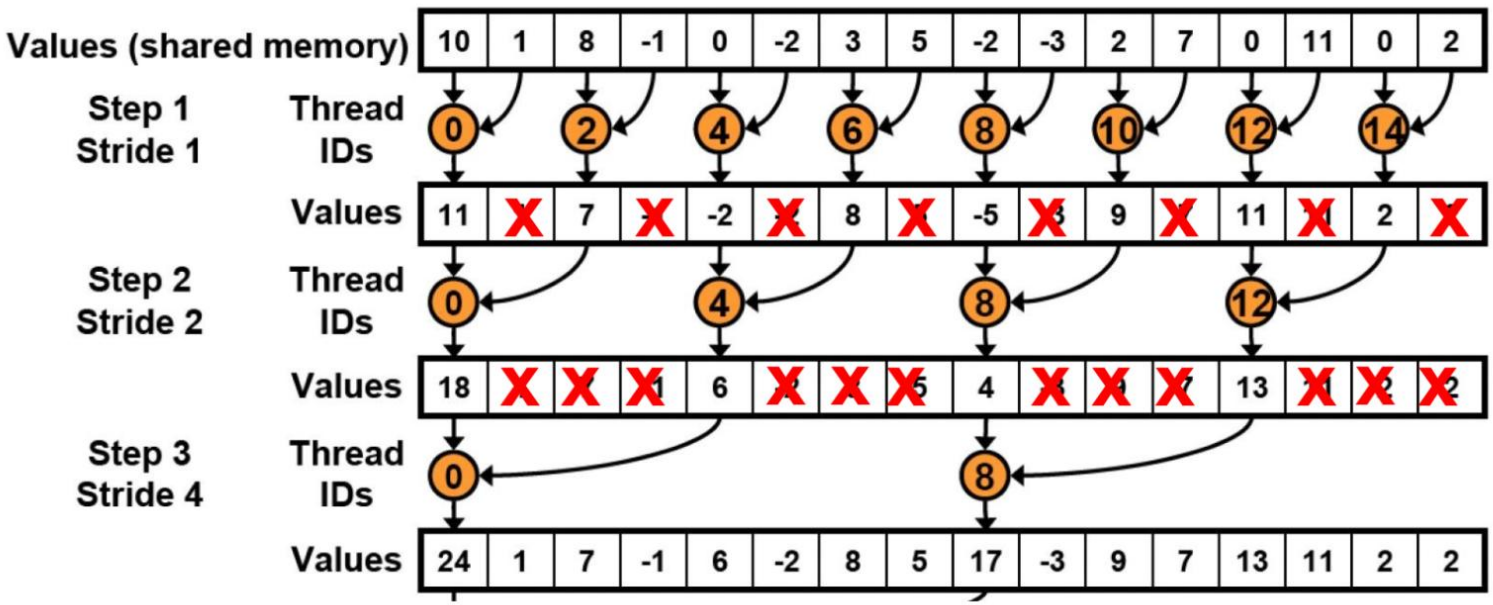
Picture courtesy: EECS471,UM

Step 1: Interleaved Addressing



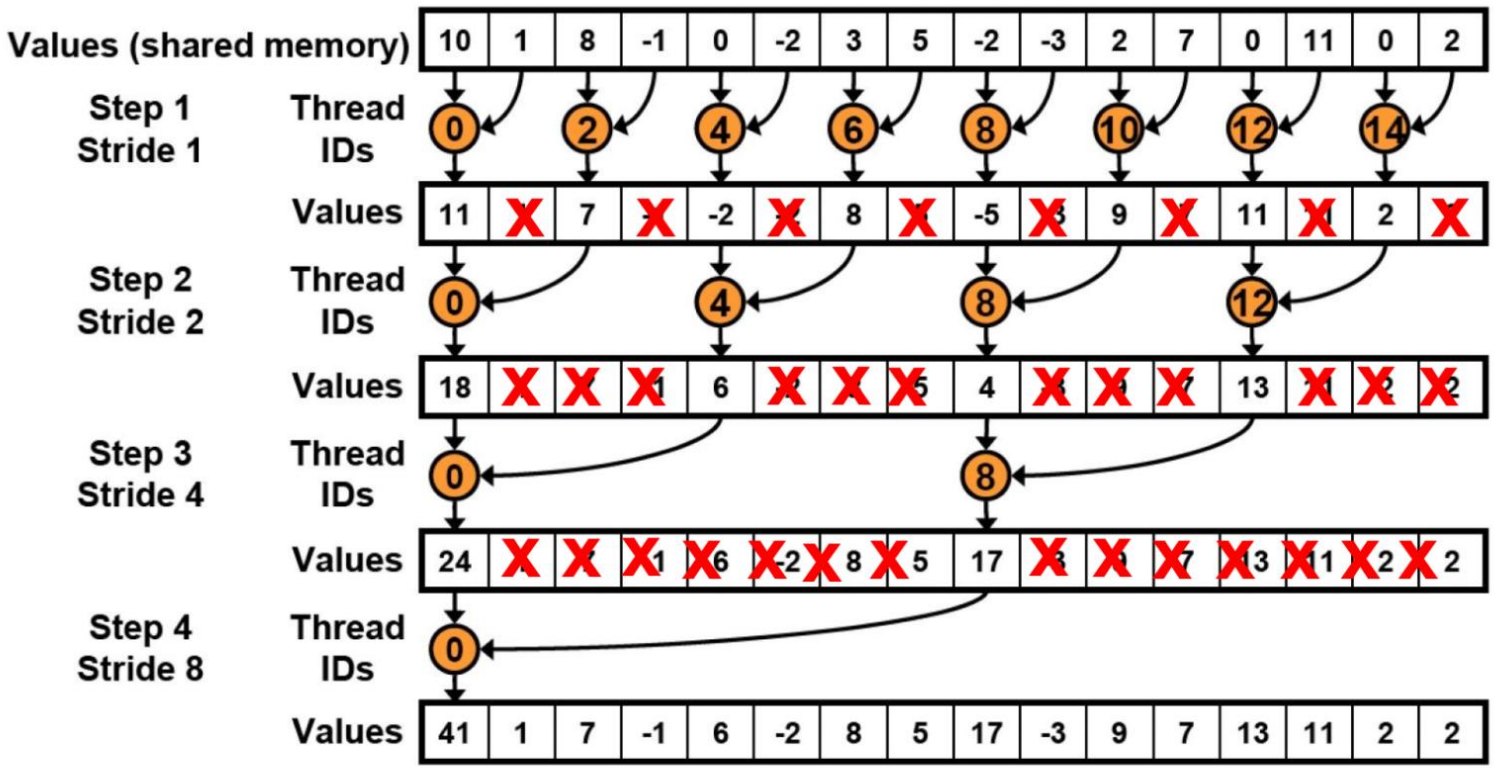
Picture courtesy: EECS471,UM

Step 1: Interleaved Addressing



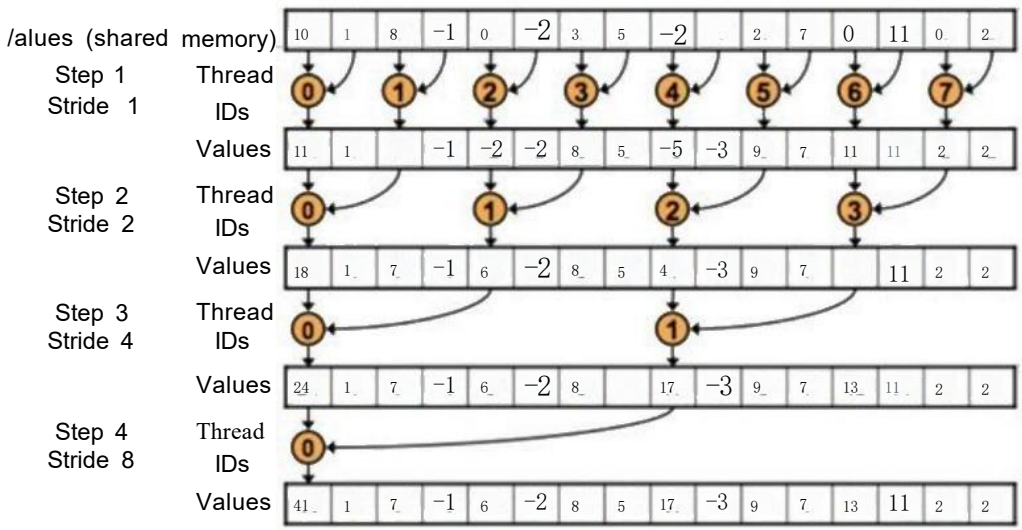
Picture courtesy: EECS471, UM

Step 1: Interleaved Addressing



Picture courtesy: EECS471, UM

Step 1: Interleaved Addressing



Picture courtesy: EECS471, UM

Step 1: Interleaved Addressing

```
6   __global__ void reduce(int *g_idata, int *g_odata, int len)
7   {
8   //Method1: simple reduction using intervleaved addresing======
9     __shared__ int sdata[BLOCK_SIZE];
10  int tid = threadIdx.x;
11  int i = blockIdx.x * blockDim.x + threadIdx.x;
12
13
14  sdata[tid] = g_idata[i];
15    __syncthreads();
16
17  // do reduction in shared mem
18  for (int s = 1; s < blockDim.x; s *= 2)
19  {
20    if (tid % (2 * s) == 0)
21    {
22      sdata[tid] += sdata[tid + s];
23    }
24    __syncthreads();
25  }
```

- Summing 1G elements on MI250 takes 11.79ms
- Divergent branching

adapted from UW CSEP 590, Dr. Hari Sadasivan

Step 1: Problem: Inactive threads & branching

Pipeline Stats

	Avg	Min	Max	Unit
VALU Active Threads	45.85	45.85	45.85	Threads

Instruction Mix

Profile & analyze using Omniperf:

· omniperf profile -n reduce_X -- ./reduce_X

· omniperf analyze -p workloads/reduce_X/mi200/ --gui -- random-port

Step 2: Avoid divergent branching and %

Replace

```
19    // do reduction in shared mem
20    for (int s = 1; s < blockDim.x; s *= 2)
21    {
22      if (tid % (2 * s) == 0)
23      {
24        sdata[tid] += sdata[tid + s];
25      }
26      __syncthreads();
27    }
```

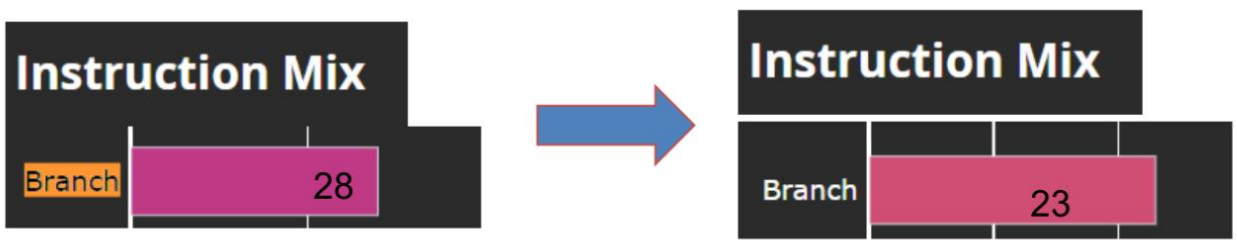
With:

```
19    // do reduction in shared mem
20    for (int s = 1; s < blockDim.x; s=s<<1)
21    {
22      int index= s*tid<<1;
23      if (index< blockDim.x)
24      {
25        sdata[index] += sdata[index + s];
26      }
27      __syncthreads();
28    }
```

adapted from UW CSEP 590, Dr. Hari Sadasivan

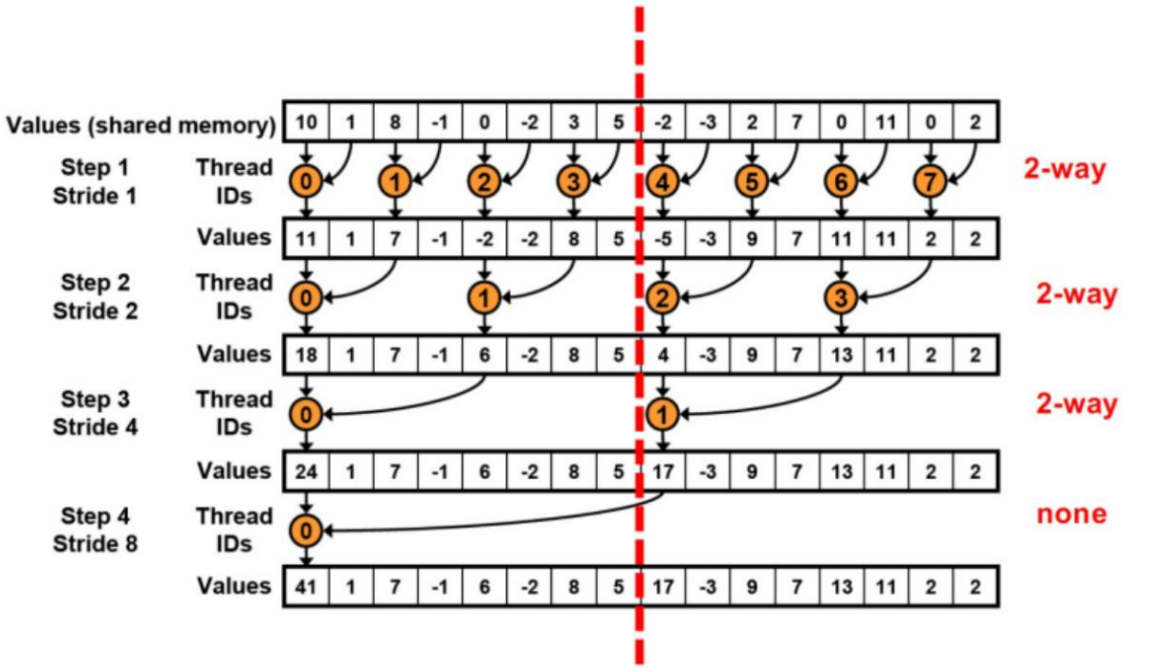
Step 2: More active threads & less branching

Pipeline Stats	Avg	Min	Max	Unit
VALU Active Threads	60.23	60.23	60.23	Threads



adapted from UW CSEP 590, Dr. Hari Sadasivan

Step 2: Avoid divergent branching and %



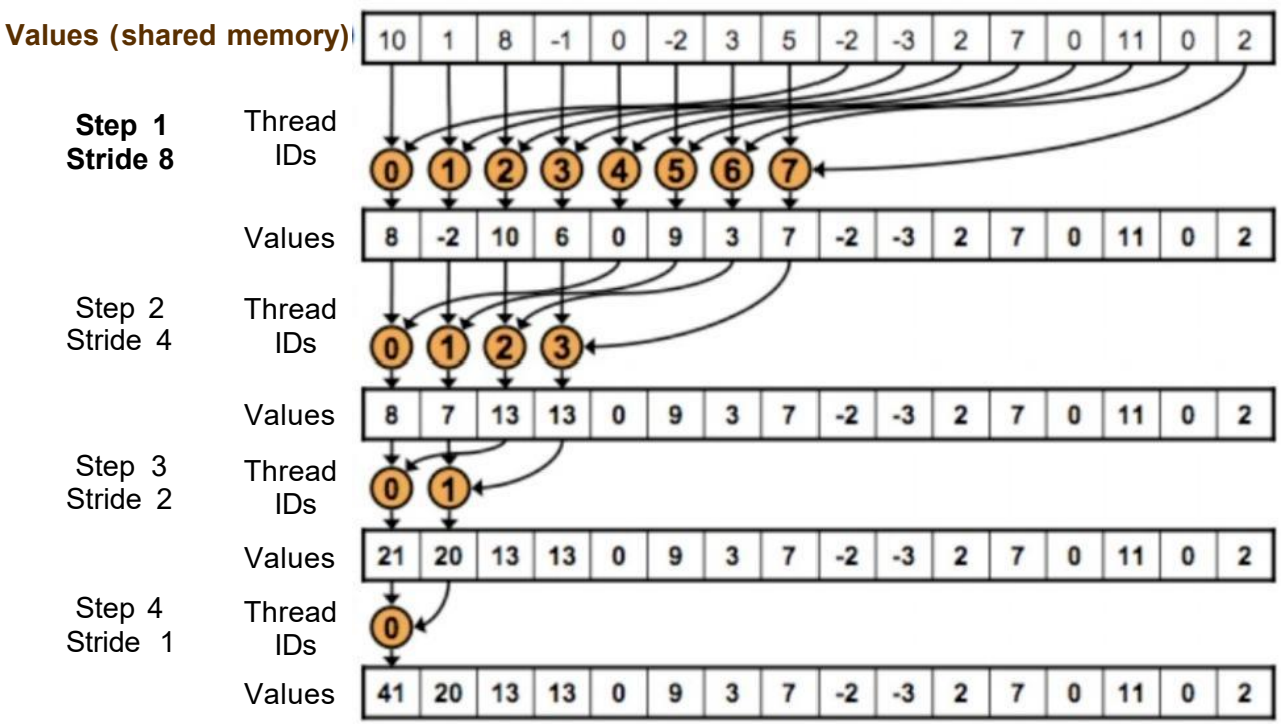
Picture courtesy: EECS471, UM

Step 2: Problem: LDS bank conflicts

12.2 LDS Stats

Bank Conflict	28.13	28.13	28.13	Cycles per wave

Step 3: Sequential addressing



Picture courtesy: EECS471, UM

Step 3: No LDS bank conflicts

12.2 LDS Stats

Bank Conflict	0.00	0.00	0.00	Cycles per wave

Step 3: Sequential addressing

Replace

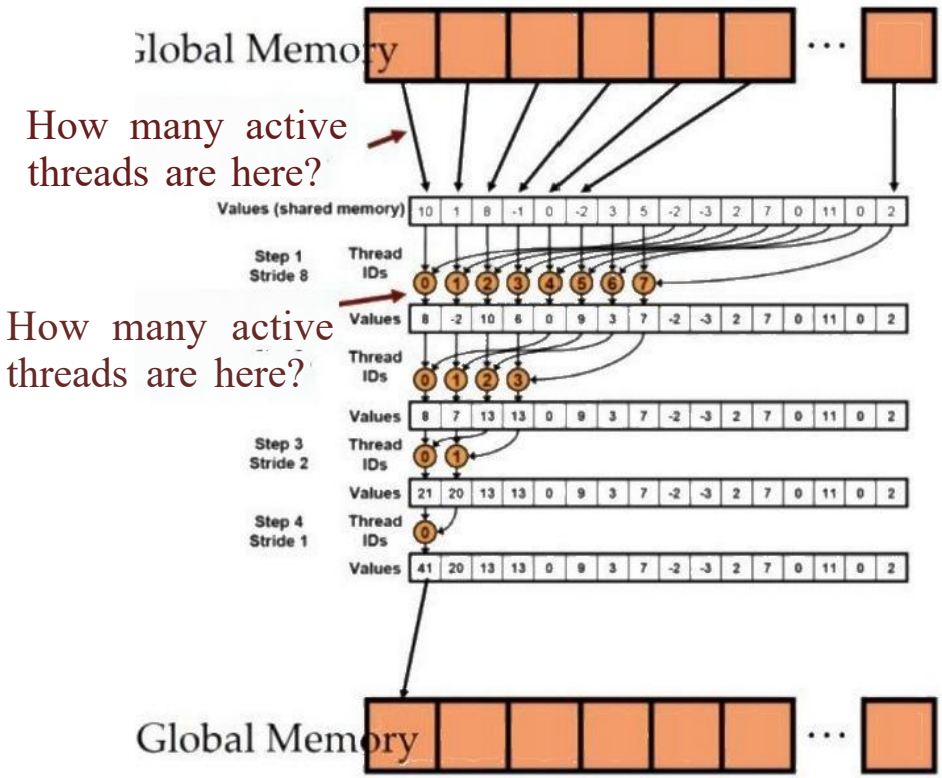
```
19  // do reduction in shared mem
20  for (int s = 1; s < blockDim.x; s=s<<1)
21  {
22    int index= s*tid<<1;
23    if (index< blockDim.x)
24    {
25      sdata[index] += sdata[index + s];
26    }
27    __syncthreads();
28  }
```

With:

```
19  // do reduction in shared mem
20  for (int s = blockDim.x/2; s >0; s>>=1)
21  {
22    if (tid<s)
23    {
24      sdata[tid] += sdata[tid + s];
25    }
26    __syncthreads();
27  }
```

adapted from UW CSEP 590, Dr. Hari Sadasivan

Increasing #inactive threads after each iteratior



Picture courtesy: EECS471, UM

Step 4: First add during load

Original: Each thread reads one element

```
10  int tid = threadIdx.x;
11  int i = blockIdx.x * blockDim.x + threadIdx.x;
12
13  sdata[tid] = g_idata[i];
14  __syncthreads();
```

Original: Halve the number of blocks & insert two loads and add into the first step

```
10  int tid = threadIdx.x;
11  int i = blockIdx.x * blockDim.x*2 + threadIdx.x;
12
13  sdata[tid] = g_data[i]+g_data[i+blockDim.x];
14  __syncthreads();
```

Use Step-4 or higher for HW3

Step 4: Increased VALU utilization

Metric	Value	Unit	Peak	PoP
VALU FLOPs	175.59	Gflop	22630.40	0.78

Metric	Value	Unit	Peak	PoP
VALU FLOPs	258.02	Gflop	22630.40	1.14

adapted from UW CSEP 590, Dr. Hari Sadasivan

Step 5: Unroll the last iteration of the loop

* Unroll iteration of the loop
  * As instruction proceeds, #active threads decreases
  * We do not want all wavefronts executing the last iteration of the loop
  * When we have less than 64 threads
    * We don't need __syncthreads()
    * Don't need if statements

Step 5: Unroll the last iteration of the loop

```
__shared__ volatile int sdata[BLOCK_SIZE];
```

```
19  // do reduction in shared mem
20  for (int s = blockDim.x/2; s >64; s>>=1)
21  {
22    if (tid<s)
23    {
24      sdata[tid] += sdata[tid + s];
25    }
26    __syncthreads();
27  }
28  //Last loop iteration is unrolled
29  if(tid<64){
30    sdata[tid]+=sdata[tid+64];
31    sdata[tid]+=sdata[tid+32];
32    sdata[tid]+=sdata[tid+16];
33    sdata[tid]+=sdata[tid+8];
34    sdata[tid]+=sdata[tid+4];
35    sdata[tid]+=sdata[tid+2];
36    sdata[tid]+=sdata[tid+1];
37  }
```

(Optional) Step 6: Complete Unrolling

Complete Unrolling

If we knew the # iterations at compile time we could completely unroll reduction

Use templates

Some of the branching will be evaluated at compile time
Block size will be our template parameter

Step 6: Complete unrolling using templates

```
template <unsigned int blockSize>
__global__ void reduce6(int *g_data, int *g_odata)
{
   /* ... global memory access + first reduction is somewhere here ... */

   if (blockSize >= 1024) {
        if (tid < 512) { sdata[tid] += sdata[tid + 512]; } __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) {
        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];
   }
}
```

adapted from UW CSEP 590, Dr. Hari Sadasivan

Step 6: Complete unrolling using constexpr

```
18      // Unroll loops based on BLOCK_SIZE
19      if constexpr (BLOCK_SIZE >= 512)
20      {
21          if (tid < 256)
22              sdata[tid] += sdata[tid + 256];
23          __syncthreads();
24      }
25
26      if constexpr (BLOCK_SIZE >= 256)
27      {
28          if (tid < 128)
29              sdata[tid] += sdata[tid + 128];
30          __syncthreads();
31      }
32      if constexpr (BLOCK_SIZE >= 128)
33      {
34          if (tid < 64)
35              sdata[tid] += sdata[tid + 64];
36      }
37
38      if (tid < 64)
39      {
40
41          if constexpr (BLOCK_SIZE >= 64)
42              sdata[tid] += sdata[tid + 32];
43          if constexpr (BLOCK_SIZE >= 32)
44              sdata[tid] += sdata[tid + 16];
45          if constexpr (BLOCK_SIZE >= 16)
46              sdata[tid] += sdata[tid + 8];
47          if constexpr (BLOCK_SIZE >= 8)
48              sdata[tid] += sdata[tid + 4];
49          if constexpr (BLOCK_SIZE >= 4)
50              sdata[tid] += sdata[tid + 2];
51          if constexpr (BLOCK_SIZE >= 2)
52              sdata[tid] += sdata[tid + 1];
53      }
```

(Optional)Step 7: Algorithm cascading: Mix parallel & sequential execution

- Replace "load and reduce two elements"

```
// each thread loads two elements from global to shared mem
// end performs the first step of the reduction
unsigned int tid = threadIdx.x;
unsigned int i = blockIdx.x* blockDim.x * 2 + threadIdx.x;
sdata[tid] = g_idata[i] + g_idata[i + blockDim.x];
__syncthreads();
```

- With a loop to reduce as many elements as necessary

```
unsigned int tid = threadIdx.x;
unsigned int i = blockIdx.x * blockSize * 2 + threadIdx.x;
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();
```

gridSize steps to achieve coalescing

adapted from UW CSEP 590, Dr. Hari Sadasivan

Reduction is memory bound

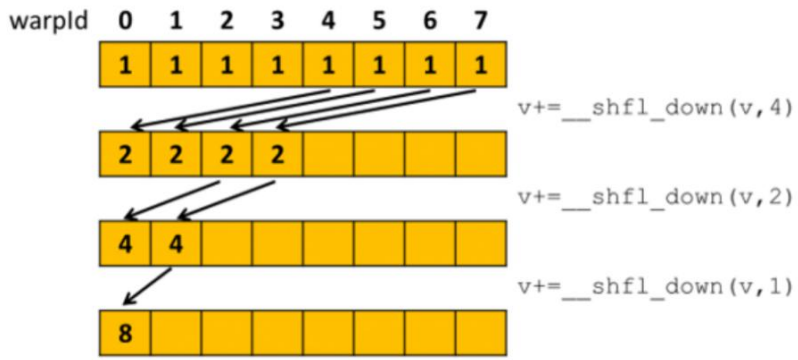
Arithmetic Intensity is 1

- AI measures FLOPs/Byte
  - O(N)ops on O(N)bytes of data

(Optional) Step 8: Using warp shuffles for inter-thread communication

__shfl_down:

```
/* warp-level sum using registers within a warp */
int offset;
for (offset = warpSize / 2; offset > 0; offset /= 2)
{
          data += __shfl_down(data, offset );
}
```



Warp Shuffle Functions

■ Built-in warp shuffle functions enable threads to share data with other threads in the same warp in lockstep.

o Faster than using shared memory and__syncthreads()to share across threads in the same block

■Variants:

- __shfl(var, srcLane, width=warpSize)

  Direct copy from indexed lane

- __shfl_up(var, delta, width=warpSize)

  Copy from a lane with lower ID relative to caller

- ___shfl_down(var, delta, width=warpSize)

  Copy from a lane with higher ID relative to caller

- __shfl_xor(var, laneMask, width=warpSize)

  Copy from a lane based on bitwise XOR of own lane ID

Slide credit: Izzat El Hajj

Reduction with Warp Shuffle

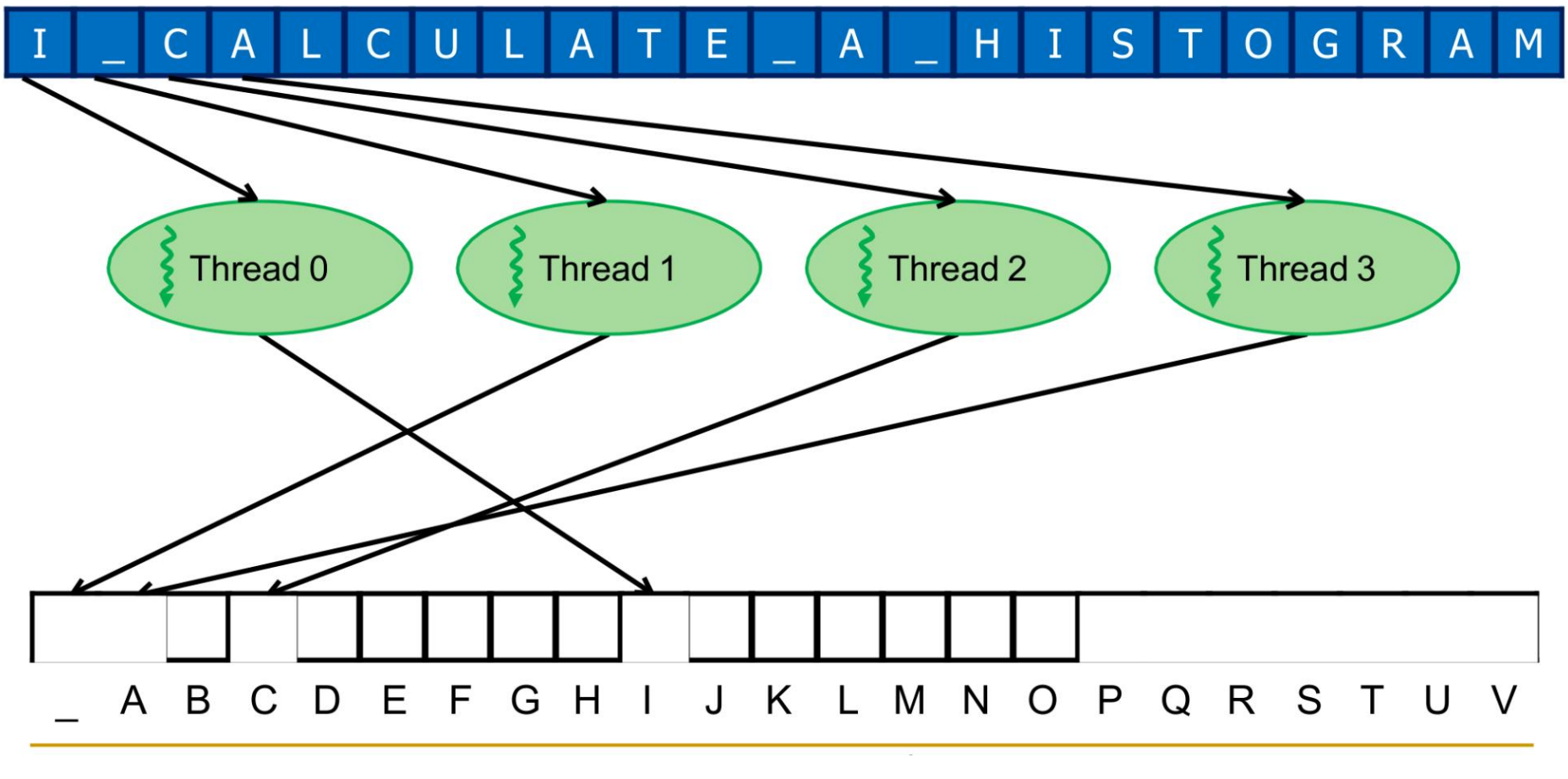
```
_global     void reduce_kernel(float* input, float* partialSums, unsigned int N) {

        unsigned int segment = 2*blockDim.x*blockIdx.x; unsigned int i
        = segment + threadIdx.x;

        // Load data to shared memory
        _shared_ float input_s[BLOCK_DIM];  input_s[threadIdx.x] = input[i] +
        input[i + BLOCK_DIM];
        _syncthreads();

        // Reduction tree in shared memory
        for(unsigned int stride = BLOCK_DIM/2; stride > WARP_SIZE; stride /= 2) { if(threadIdx.x < stride) {
                        input_s[threadIdx.x] += input_s[threadIdx.x + stride];
                }
                _syncthreads();
        }

        // Reduction tree with shuffle instructions float sum;
        if(threadIdx.x < WARP_SIZE) {
                sum = input_s[threadIdx.x] + input_s[threadIdx.x + WARP_SIZE];

                for(unsigned int stride = WARP_SIZE/2; stride > 0; stride /= 2) { sum +=_shfl_down(sum,
                        stride, 0xffffffff);
                }
        }
        // Store partial sum
        if(threadIdx.x == 0) {
                partialSums[blockIdx.x] = sum;
        }
}
```

Slide credit: Izzat El Hajj

Parallel Histogram Computation: Iteration 1

Adjacent threads read adjacent input characters

- Reads from the input array are coalesced



(Wrong) Parallel Histogram Kernel

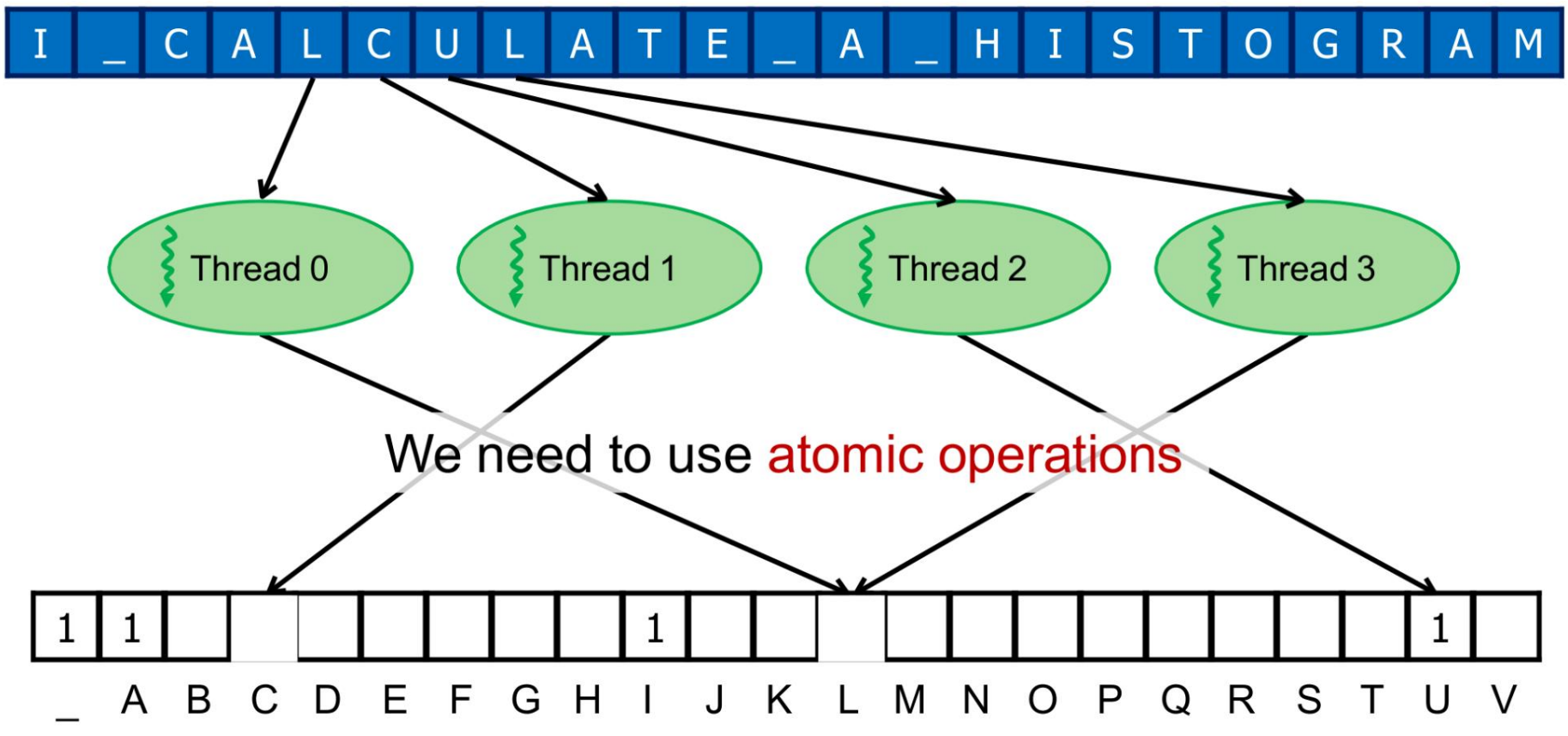
```
__global void histogram_kernel(unsigned int *histo,
                               unsigned int *input,
                               unsigned int input_size) {
  int i = blockIdx.x*blockDim.x + threadIdx.x; // Thread index
  int stride = blockDim.x*gridDim.x; // Total number of threads
  while (i<input_size) {
    unsigned int val = input[i];
    histo[val]+=1;
    i += stride;
  }
}
```

adapted from UW CSEP 590, Dr. Hari Sadasivan

Parallel Histogram Computation: Iteration 2

All threads move to the next section of the input

- Each thread moves to element threadID + #threads

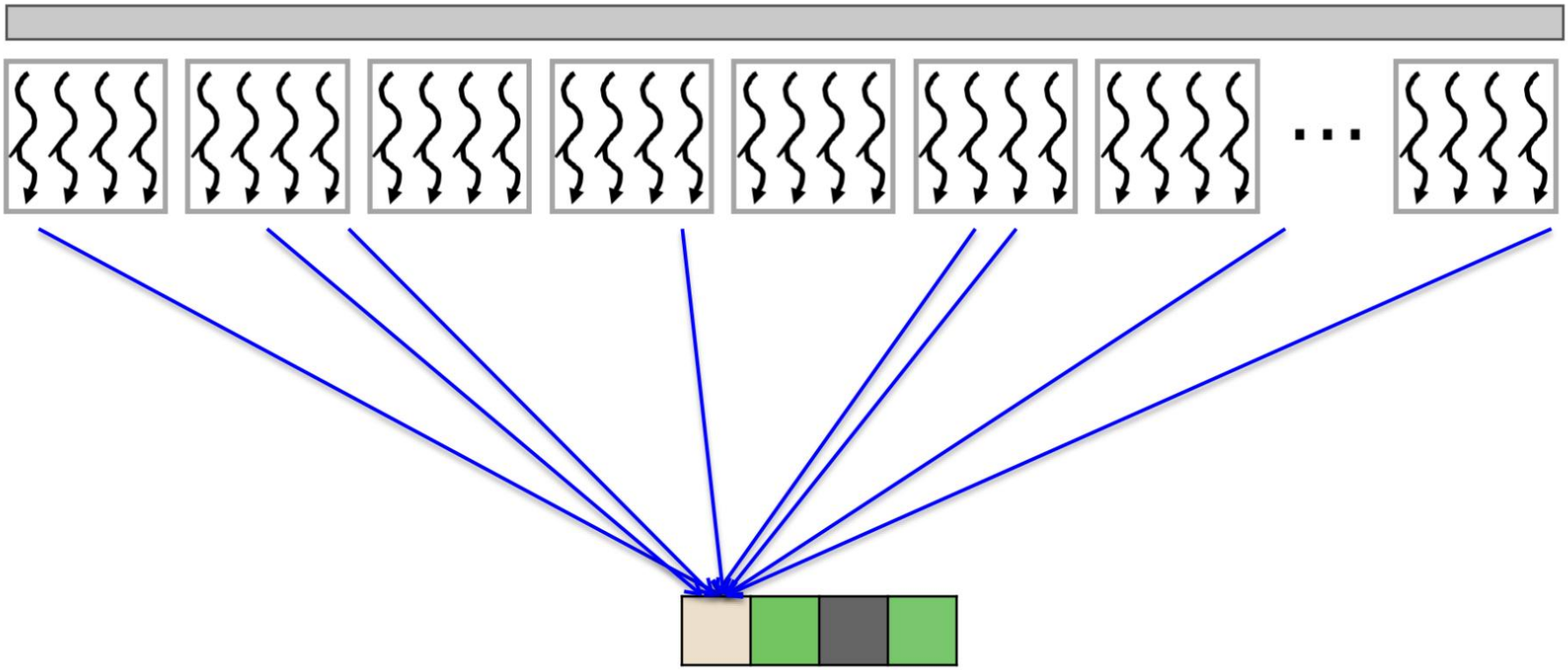


(Correct) Parallel Histogram Kernel

```
__global void histogram_kernel(unsigned int *histo,
                               unsigned int *input,
                               unsigned int input_size){
  int i = blockIdx.x*blockDim.x + threadIdx.x; // Thread index
  int stride = blockDim.x * gridDim.x; // Total number of threads
  while (i<input_size) {
    unsigned int val = input[i];
    atomicAdd(&histo[val],1);
    i += stride;
  }
}
```

adapted from UW CSEP 590, Dr. Hari Sadasivan

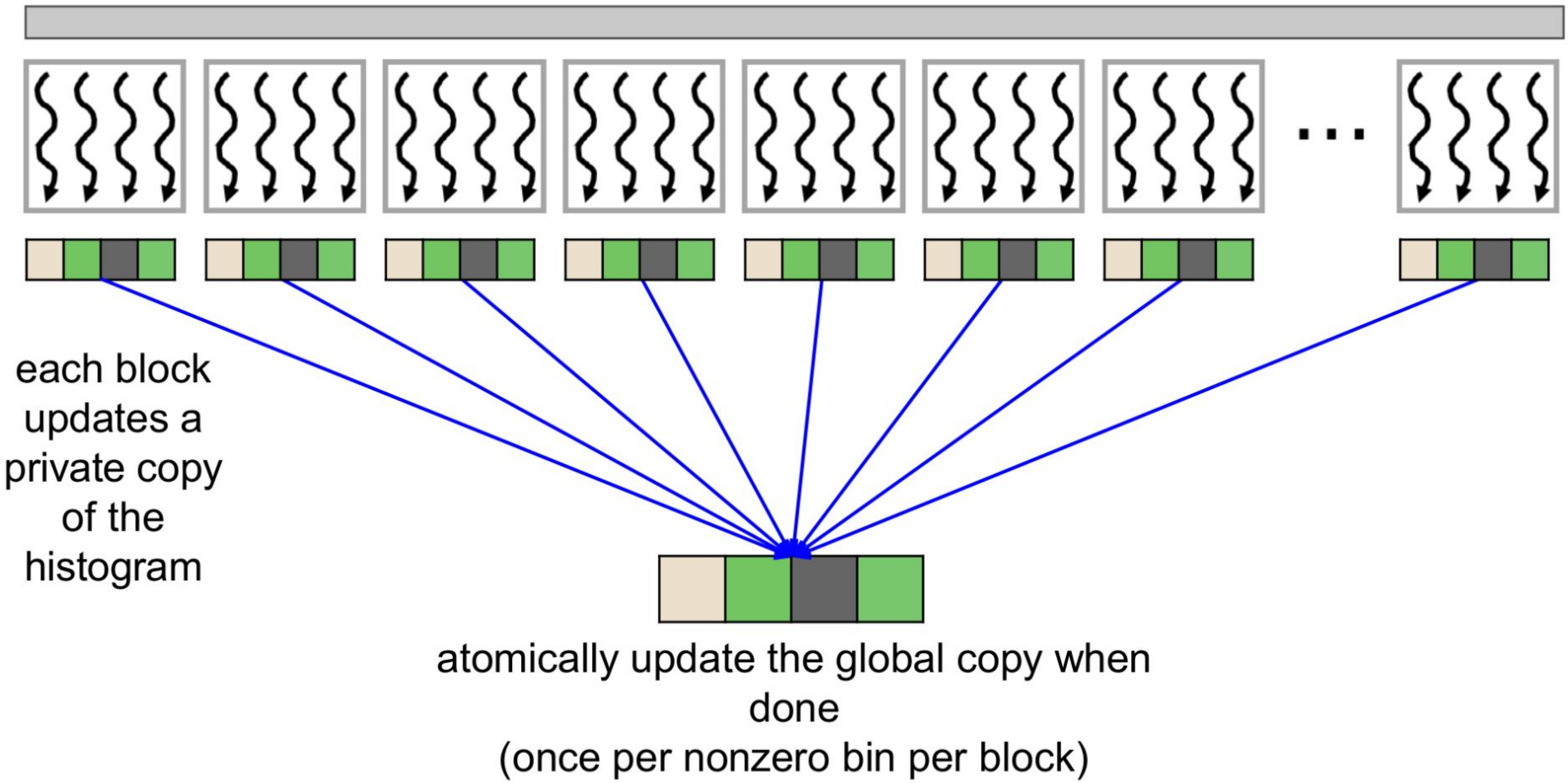
Global memory atomics have high latency



Atomic operations on global memory have high latency

- Need to wait for both read and write to complete
- Need to wait if there are other threads accessing the same location (high probability of contention)

Privatizing the Histogram



Privatization

Privatization is an optimization technique where multiple private copies of an output are maintained,then the global copy is updated on completion

- Operations on the output must be associative and commutative because the order of updates has changed

Advantages:

- Reduces contention on the global copy
- If the output is small enough,the private copy can be placed in shared memory reducing access latency

Slide credit: Izzat El Hajj

Privatization

Privatization is one of the most powerful and frequently used techniques for parallelizing applications

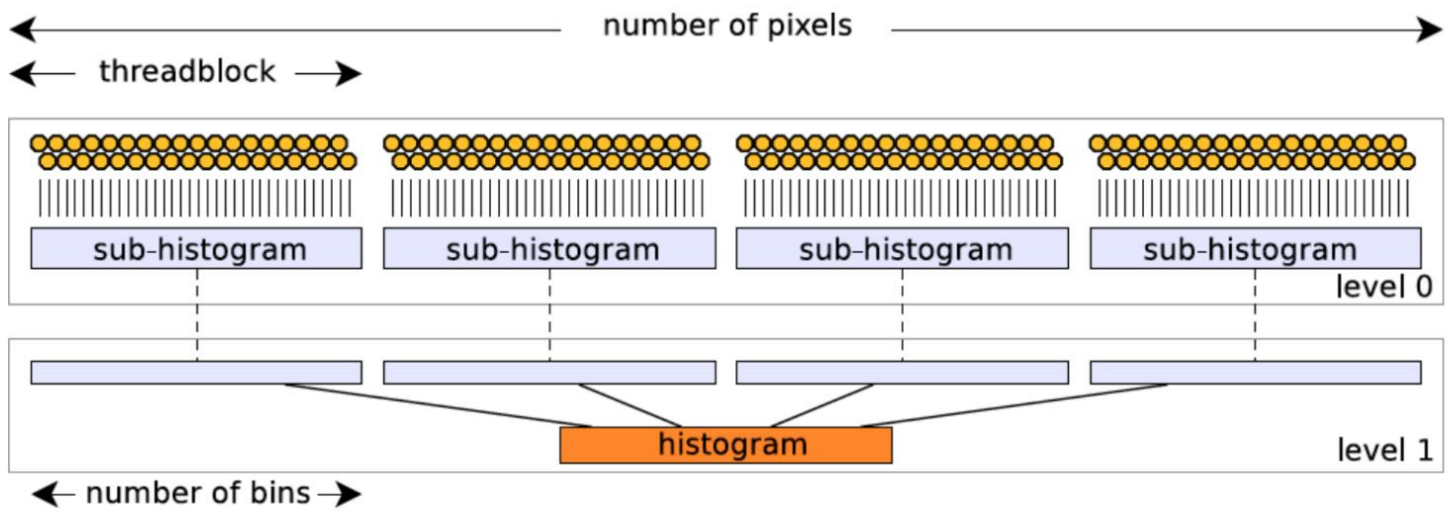
Each thread works on its private data;merge later

The operation needs to be

- Commutative: $x \times y = y \times x$
- Associative:$(x \times y) \times z = x \times (y \times z)$
- Histogram add operation is associative and commutative

Histogram: Parallel Strategy #1

- Input array in global memory
- Histogram array in shared memory
- One histogram per thread
- One column per pssile pixel value



adapted from UW CSEP 590, Dr. Hari Sadasivan

Sub-Histograms

- How big is each histogram (KB)?
  - Input value range: 0-255, 1 byte
  - Each histogram needs 256 entries
  - How many bytes per entry?
    - That's data dependent
  - Let's assume 32-bits or 4 bytes:256*4 = 1KB / histogram

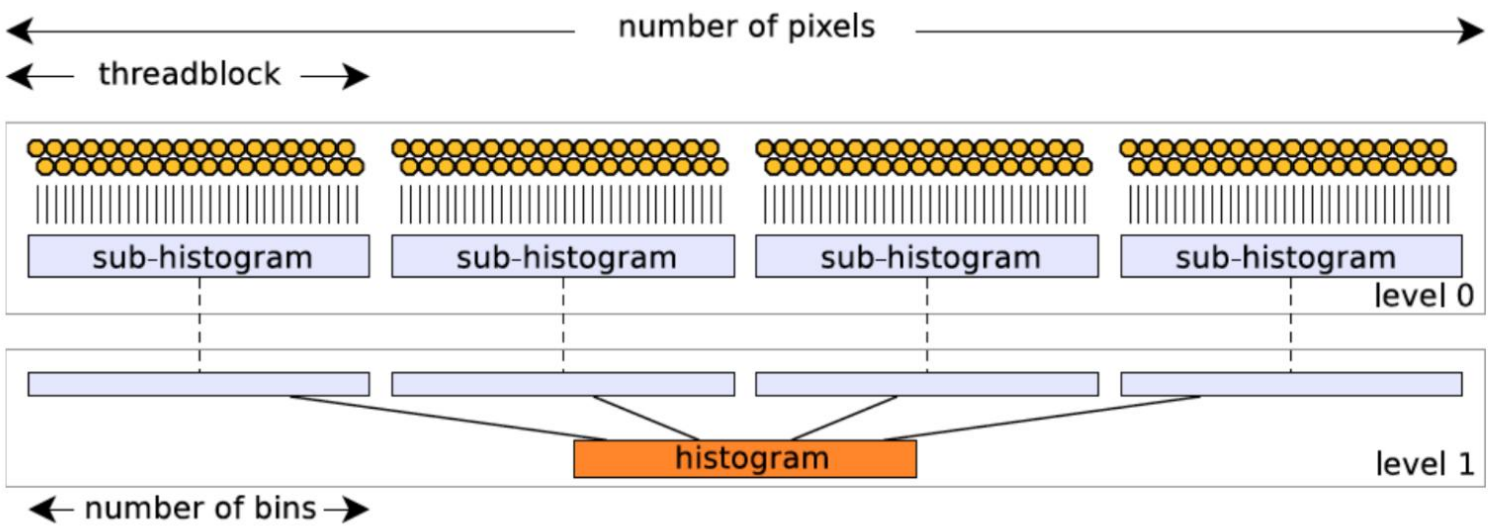
- How many sub-histograms can we fit in shared mem?
  - Shared mem: 64 KB
  - 64KB shared mem → only 1 wavefront (64KB in histograms) - Use only 1 wavefront per CU
  - Use only 1 out of 40 resident waves on CU
  - Unused hardware, low occupancy

Sub-Histograms

- Let's try one histogram per block
  - Many threads per block
  - Ordering problem persists but within a block
  - However, threads within a block can synchronize!!!
  - synchronization → no more race conditions

Histogram: Parallel Strategy #2

- Input array in global memory
- Histogram array in shared memory
- One histogram per threadblock; all threads in the block update it
- One column per possible pixel value
- Each block updates the global histogram at the end



adapted from UW CSEP 590, Dr. Hari Sadasivan

Algorithm Overview

·Step 1:

-Initialize partial histogram

-Each thread:

s_Hist[index]=0

index+=threads per block

-Until all bins are taken care of

·Step 2:

-Generate a partial histogram

-Each thread:

read data[index]

update s_Hist] ←conflicts possible (within block)

index+=Total number of threads

-Until all input/block is taken care of

- Step 3:
  - Update global histogram
  - Each thread
    - read s_hist[index]
    - update global histogram ← conflicts possible (across blocks)
    - index += threads per block
      - Until all bins are taken care of

adapted from UW CSEP 590, Dr. Hari Sadasivan

Parallel Histogram Kernel with Privatization (+Coarsening)

```
__global__ void histogram_kernel(unsigned int *histo,
                                 unsigned int *input, unsigned int input_size) {
  int tid = blockIdx.x * blockDim.x + threadIdx.x; // Thread index
  int stride = blockDim.x * gridDim.x; // Total number of threads
  __shared__ unsigned int histo_s[BINS];// Private per-block sub-histogram
  // Sub-histogram initialization
  for (int i=threadIdx.x; i<BINS; i + =blockDim.x) {
    histo_s[i]=0;
  }
  __syncthreads(); // Intra-block synchronization
  // Main loop to compute per-block sub-histograms
  for (int i=tid; i<input_size; i+=stride) {
    unsigned int val = inputfi]; // Global memory read (coalesced)
    atomicAdd(&histo_s[val],1);// Atomic addition in shared memory
  }
  __syncthreads(); // Intra-block synchronization
  // Merge per-block sub-histograms and write to global memory
  for(int i=threadIdx.x; I <BINS;I +=blockDim.x){
    //Atomic addition in global memory
    atomicAdd(histo+i, histo_s[]);
  }
}
```

Histogram Parallel Strategy #2: Review

- Are memory accesses coalesced?
  - Yes (for input data)
- Is there control divergence?
  - Negligible (only at the boundary check)
- Is there data reuse?
  - Input data:No
  - Output data:Yes

    Multiple threads in a block may update the same bin in the private histogram

    Place the private histogram in shared memory (if it fits)

    - Is there a price for parallelizing the work?
  - Atomic updates from the private copies to the global copy happen once per block
  - Use thread coarsening to reduce the number of blocks/updates

Checklist ofCommon Optimizations

Optimization	Benefit to Compute Cores	Benefit to Memory	Strategies
Maximizing occupancy	More work to hide pipeline latency	More parallel memory accesses to hide DRAM latency	Tuning usage of SM resources such as threads per block, shared memory per block, and registers per thread
Enabling coalesced global memory accesses	Fewer pipeline stalls waiting for global memory accesses	Less global memory traffic and better utilization of oursts/cache-lines	Transfer between global memory and shared memory in a coalesced manner and performing un-coalesced accesses in shared memory (e.g., corner turning) Rearranging the mapping of threads to data Rearranging the layout of the data
Minimizing control divergence	High SIMD efficiency minimizing idle cores during SIMD execution)		Rearranging the mapping of threads to work and/or data Rearranging the layout of the data
Tiling of reused data	Fewer pipeline stalls waiting for global memory accesses	Less global memory traffic	Placing data that is reused within a block in shared memory or registers so that it is transferred between global memory and the SM only once
Privatization	Fewer pipeline stalls waiting for atomic updates	Less contention and serialization of atomic updates	Applying partial updates to a private copy of the date then updating the universal copy when done
Thread coarsening	Less redundant work divergence, or synchronization	Less redundant global memory traffic	Assigning multiple units of parallelism to each thread in order to reduce the "price of parallelism"when it is incurred unnecessarily

Prefix Sum (Scan)

- Prefix sum or scan is an operation that takes an input array and an associative operator,
  - E.g.,addition,multiplication,maximum,minimum
- And returns an output array that is the result of recursively applying the associative operator on the elements of the input array

- Input array [xo,x_1,……,Xn-]
- Associative operator ①

- An output array [yo,y_1,…,yn-]where
  - Exclusive scan: $y_i = x_0 \oplus x_1 \oplus \ldots \oplus x_{i-1}$
  - Inclusive scan: $y_i = x_0 \oplus x_1 \oplus \ldots \oplus x_i$

Scan Applications

- Scan is a key parallel primitive that can
  - convert recurrences from sequential

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

  - into parallel

```
forall(i) {temp =f(i)}; scan(out,
temp);
```

Scan Applications

- Cut a bread to feed 10 people as per their preferences:
  - [3 5 2 7 28 4 3 0 8 1]
  - Method 1: Sequential-cut 3, then 5, then..
  - Method 2: Parallel-Use prefix-scan & calculate offsets

  [3, 8, 10, 17, 45, 49, 52, 52, 60, 61]

- Scan is a basic building block of many parallel algorithms
  - E.g.,stream compaction, partition, select, unique, radix sort, quicksort, string comparison, lexical analysis, polynomial evaluation, solving recurrences, tree operations, histograms, etc.

Definition: (Inclusive)Prefix-Sum (Scan)

- Definition: The all-prefix-sums operation takes a binary associative operator ④and an array of n elements

  [Xo,Xy,…,Xn-1],

and returns the array

[xo,(x₀①x₁),…,(x₀①x₁①…①Xn-)].

- Example:If ①is addition,then the all-prefix-sums

operation on the array [3 1 7 0 4 1 6 3],

would return [3 4 11 11 15 16 22 25]

An Inclusive Sequential Prefix-Sum

- Given a sequence: $[X_o, X_1, X_2, \ldots]$
- Calculate output: $[y_o, y_1, y_2, \ldots]$

Such that:

$Y_o = X_o$

$y_1 = X_o + X_1$

$Y_2 = X_o + X_1 + X2$

- A recursive definition: $y_i = y_{i-1} + x_i$

A Work-Efficient C Implementation

```
y[0] = x[0];
For (i=1; i<Max_i; i++)
        y[i] = y  [i-1] + x[i];
```

Computationally efficient:

N additions needed for N elements - O(N)!

A Naive Inclusive Parallel Scan

Assign one thread to calculate each y element.

- Have every thread add up all x elements needed for the y element

$Yo=Xo$

$y_1=Xo+X_1$

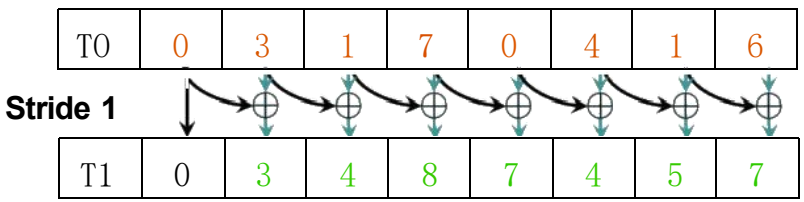
$Y_2=Xo+X_1+X2$

A Plausible Parallel Scan Algorithm

T0	0	3	1	7	0	4	1	6

Load input from global to shared memory array TO

A Plausible Parallel Scan Algorithm



1.(previous slide)

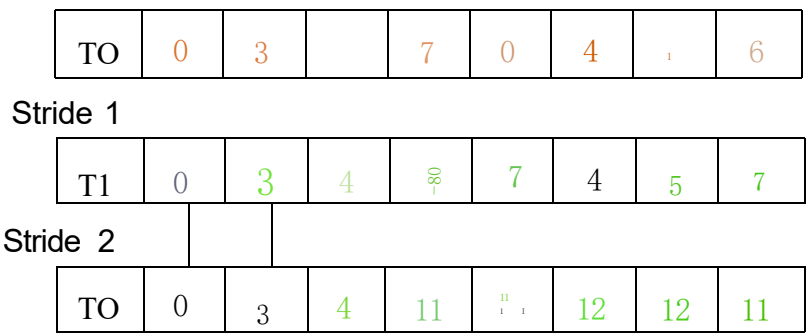
2.Iterate log(n)

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 TO and writes result into shared memory buffer T1 (ping-pong)

A Plausible Parallel Scan Algorithm

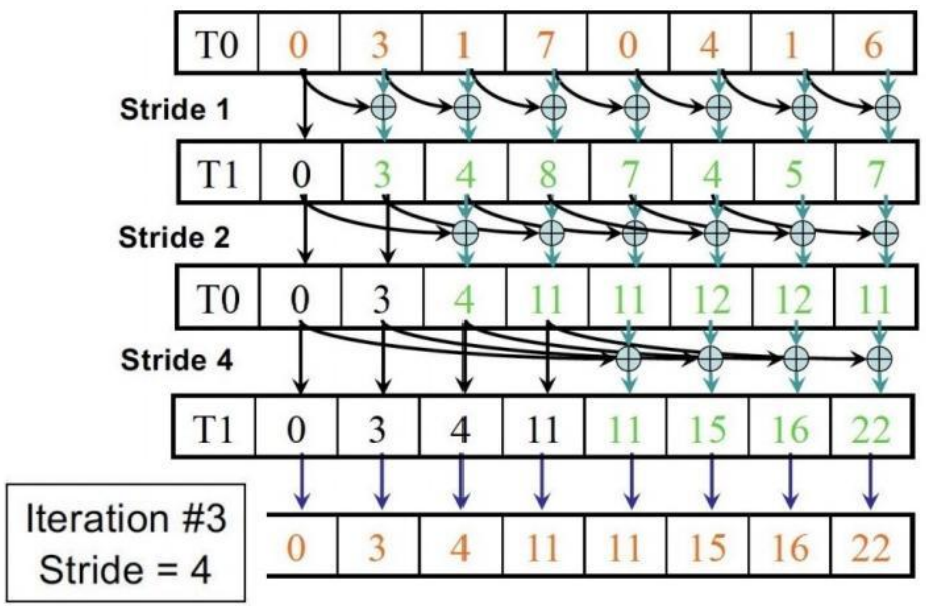


Lteration #2
Stride = 2

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)

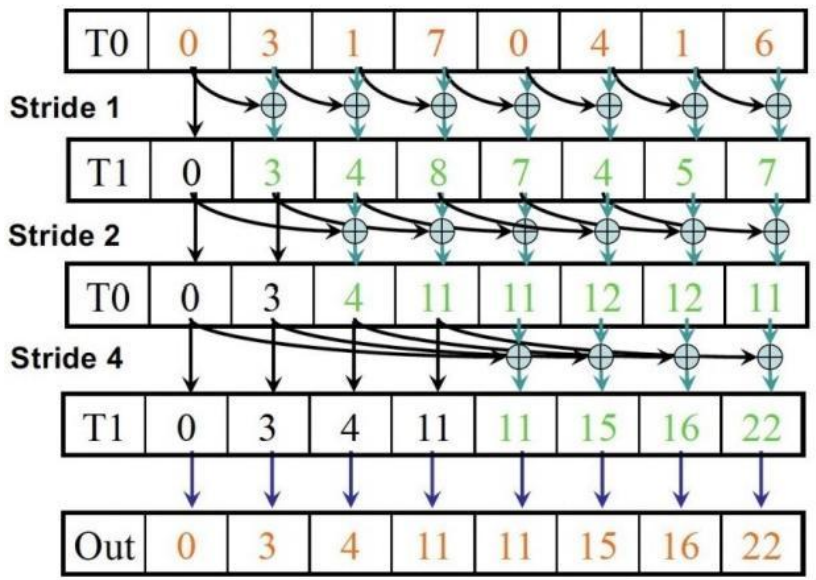
A Plausible Parallel Scan Algorithm



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)

A Plausible Parallel Scan Algorithm

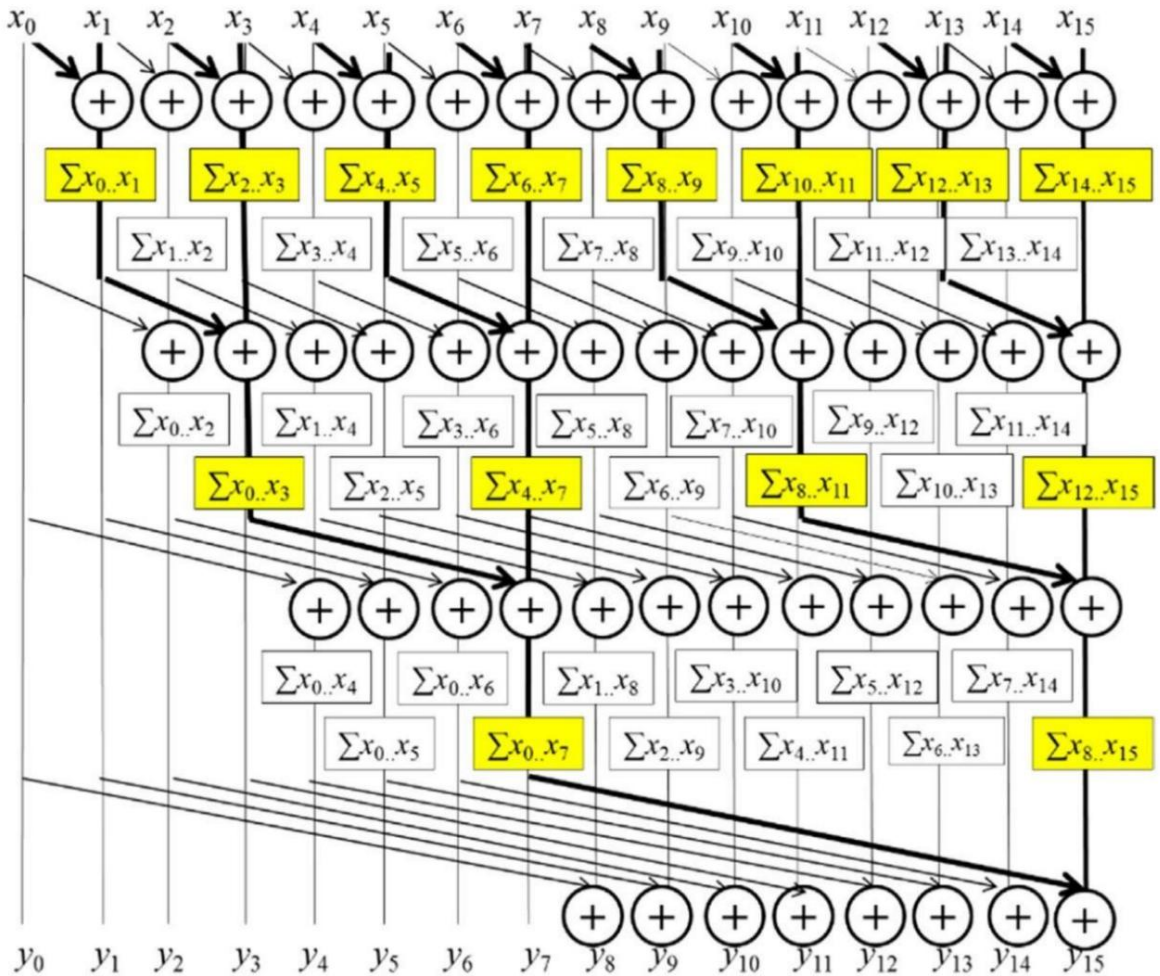


1.(previous slide)

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)

3.Write output to device memory.

Kogge-Stone algorithm



Kogge-Stone algorithm

```
01    __global__ void Kogge_Stone_scan_kernel(float *X, float *Y, unsigned int N){
02        __shared__ float XY[SECTION_SIZE];
03        unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
04        if(i < N) {
05            XY[threadIdx.x] = X[i];
06        } else {
07            XY[threadIdx.x] = 0.0f;
08        }
09        for(unsigned int stride = 1; stride < blockDim.x; stride *= 2) {
10            __syncthreads();
11            float temp;
12            if(threadIdx.x >= stride)
13                temp = XY[threadIdx.x] + XY[threadIdx.x-stride];
14            __ syncthreads();
15            if(threadIdx.x >= stride)
16                XY[threadIdx.x] = temp;
17        }
18        if(i < N) {
19            Y[i] = XY[threadIdx.x];
20        }
21    }
```

Work Efficiency Considerations

- The plausible parallel Scan executes log(n) parallel

  iterations

  -The steps do(n-1), (n-2), (n-4),…(n-n/2) adds

  -Total adds: ??? work

Work Efficiency Considerations

The plausible parallel Scan executes log(n)parallel iterations

-The steps do(n-1), (n-2), (n-4),..(n-n/2) adds

-Total adds: n x log(n) + (n-1)→O(n x log(n))work

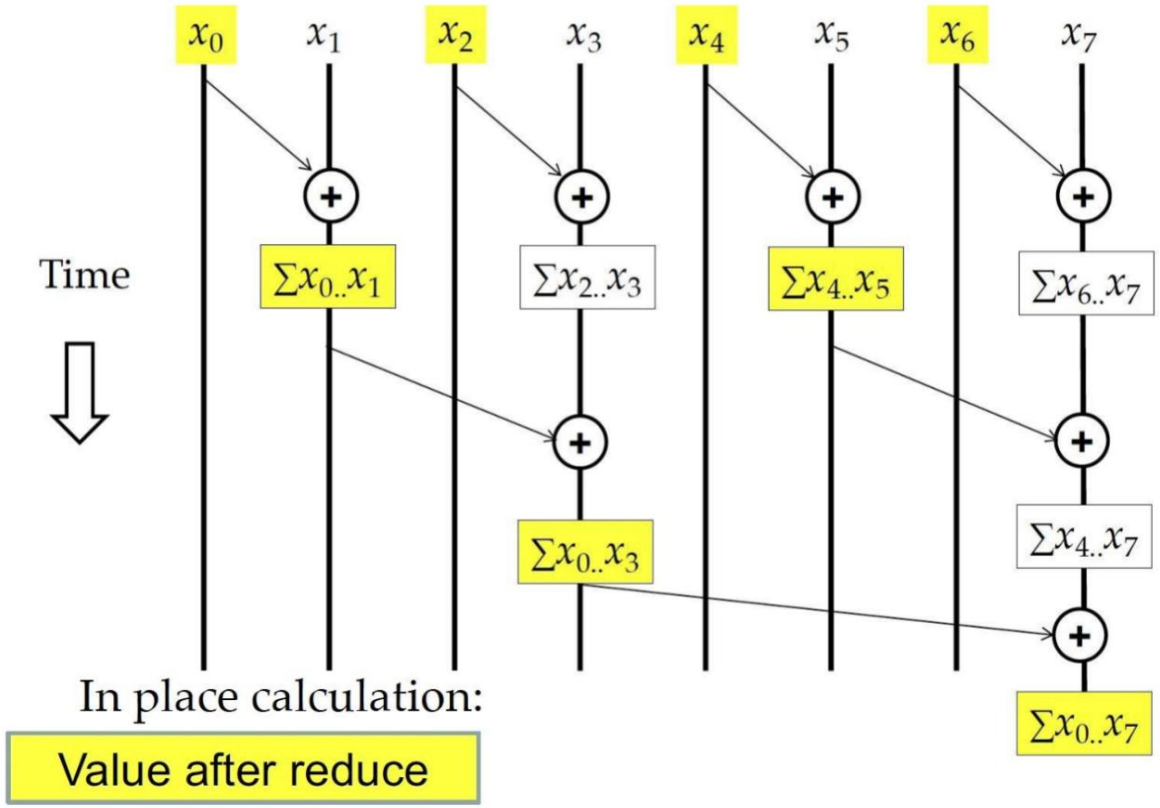
This scan algorithm is not very work efficient

-Sequential scan algorithm does 10^6 dds

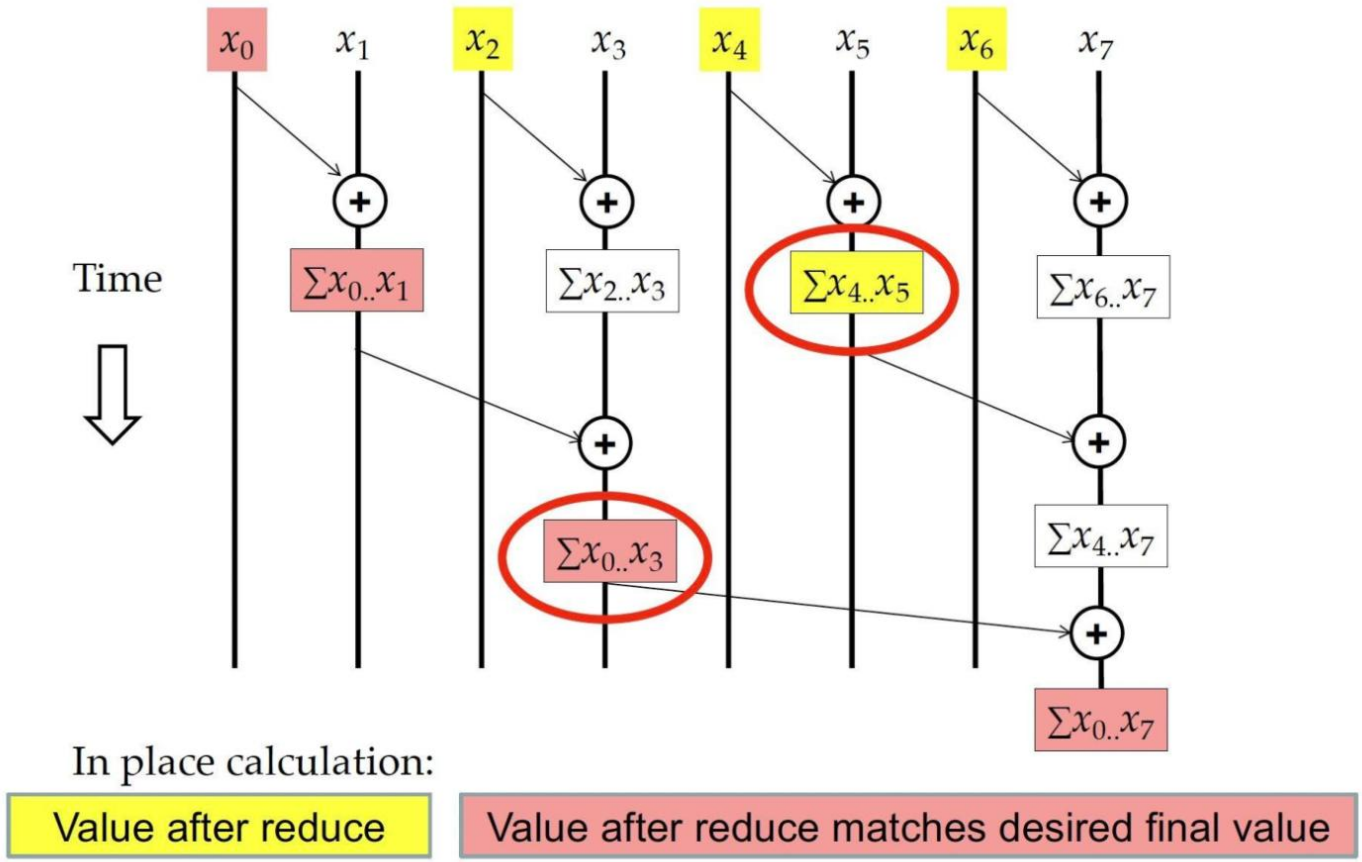
-A factor of log (n) hurts: 20x for 10 elements!

A parallel algorithm can be slow when execution resources are saturated due to low work efficiency

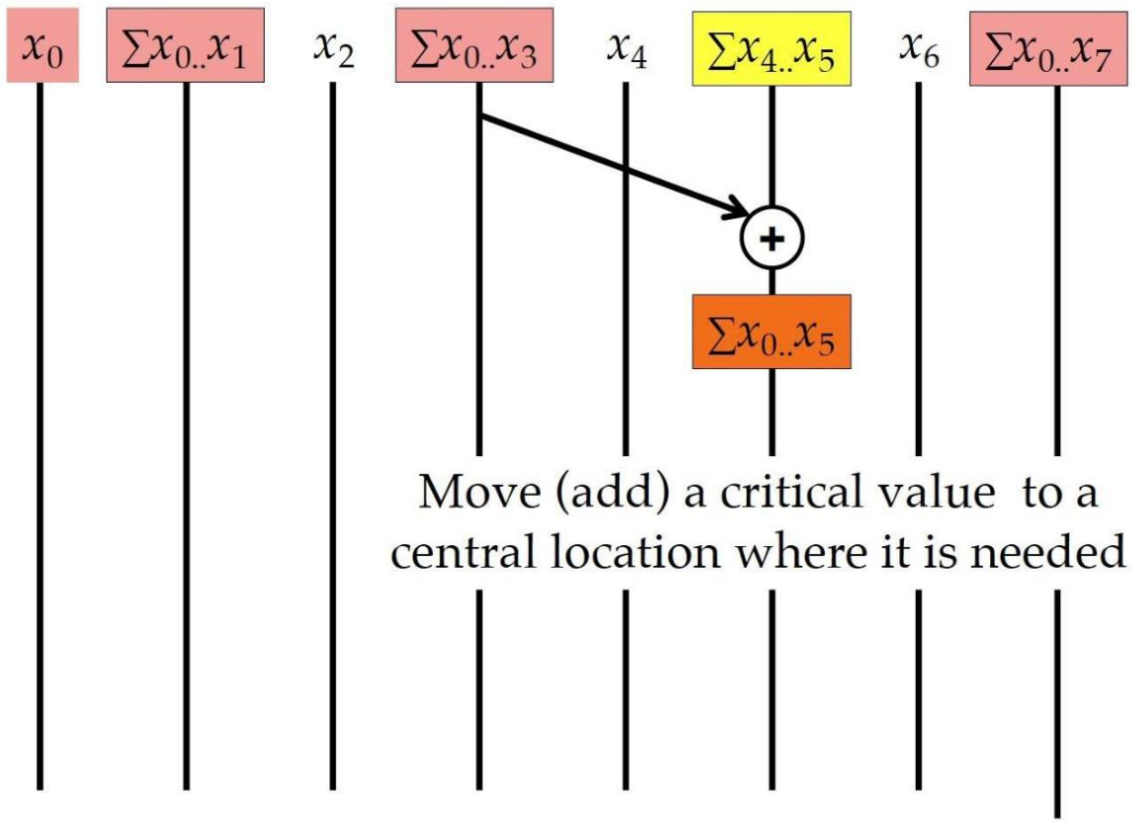
Another solution: Reduction Scan Step



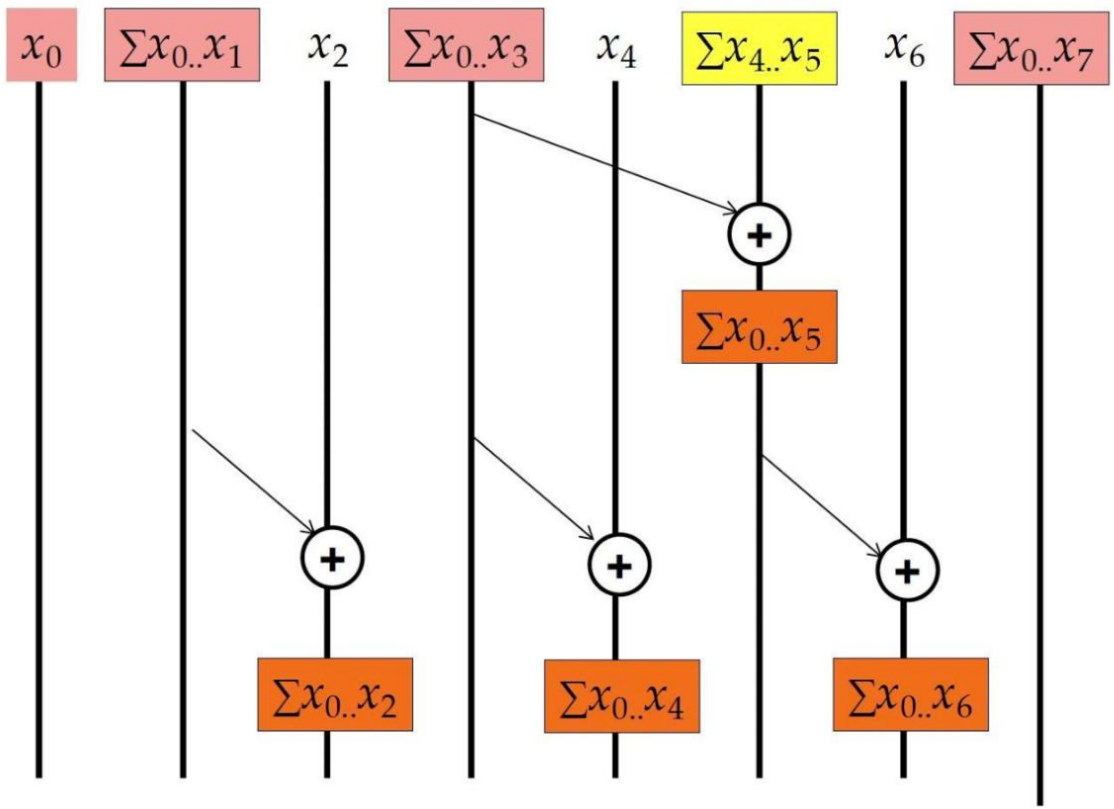
Reduction Scan Step



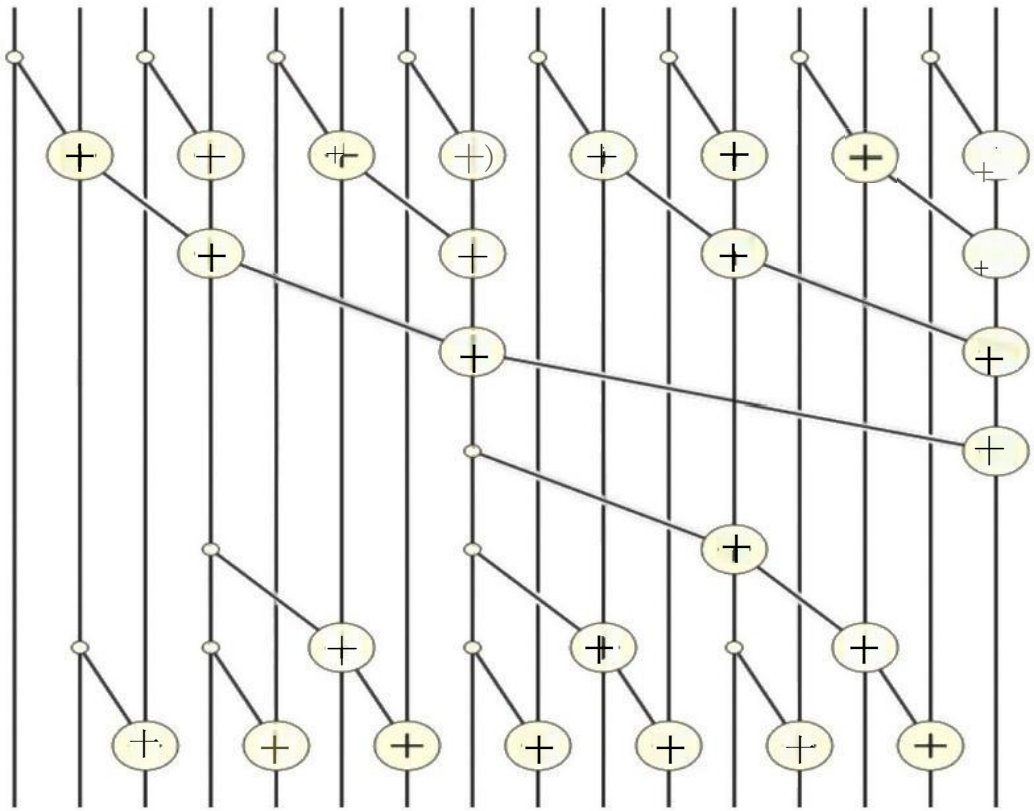
Inclusive Post Scan Step



Inclusive Post Scan Step



Putting it Together:Brent-Kung kernel

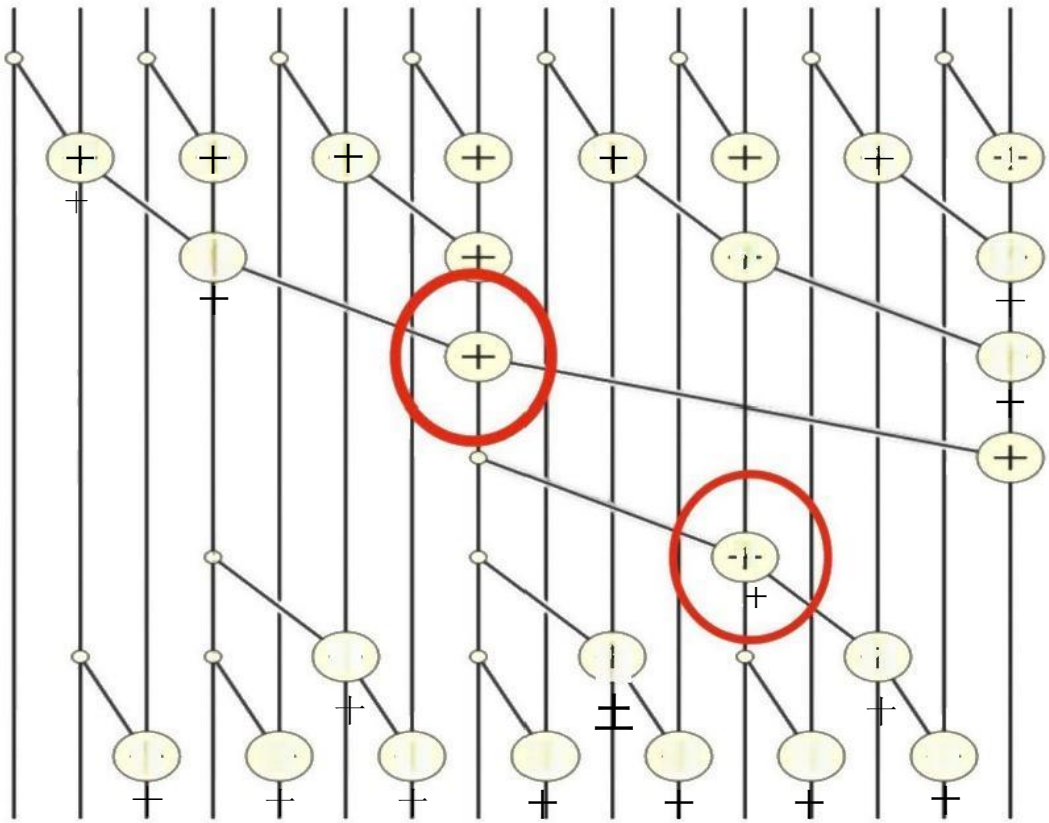


- Half #threads
- Twice #steps
- No double buffering

Reduction Step Kernel Code

```
// scan_array[BLOCK_SIZE] is in shared memory

int stride = 1;
while (stride < BLOCK_SIZE)
{
    int index = (threadIdx.x+1)*stride*2 - 1;
    if (index < BLOCK_SIZE)
        scan_array[index] += scan_array[index-stride];
    stride = stride*2;

    __syncthreads();
}
```

Putting it Together: Brent-Kung kernel



Post Scan Step

```
int stride = BLOCK_SIZE >> 1;
while(stride > 0)
{
    int index = (threadIdx.x+1)*stride*2 - 1;
    if(index < BLOCK_SIZE) {
        scan_array[index+stride] += scan_array[index];
    }
    stride = stride >> 1;
    __syncthreads();
}
```

Brent-Kung kernel

```
01    __global__ void Brent_Kung_scan_kernel(float *X, float *Y, unsigned int N) {
02        __shared__ float XY[SECTION_SIZE];
03        unsigned int i = 2*blockIdx.x*blockDim.x + threadIdx.x;
04        if(i < N) XY[threadIdx.x] = X[i];
05        if(i + blockDim.x < N) XY[threadIdx.x + blockDim.x] = X[i + blockDim.x];
06        for(unsigned int stride = 1; stride <= blockDim.x; stride *= 2) {
07            __syncthreads();
08            unsigned int index = (threadIdx.x + 1)*2*stride - 1;
09            if(index < SECTION_SIZE) {
10                XY[index] += XY[index - stride];
11            }
12        }
13        for (int stride = SECTION_SIZE/4; stride > 0; stride /= 2) {
14            __syncthreads();
15            unsigned int index = (threadIdx.x + 1)*stride*2 - 1;
16            if(index + stride < SECTION_SIZE) {
17                XY[index + stride] += XY[index];
18            }
19        }
20        __syncthreads();
21        if (i < N) Y[i] = XY[threadIdx.x];
22        if (i + blockDim.x < N) Y[i + blockDim.x] = XY[threadIdx.x + blockDim.x];
23    }
```

Work Analysis

The parallel Inclusive Scan executes 2x log(n) parallel iterations

-log(n) in reduction and log(n) in post scan

-The iterations do ??? adds

Work Analysis

■ The parallel Inclusive Scan executes 2× log(n) parallel iterations

-log(n) in reduction and log(n) in post scan

-The iterations do n/2, n/4, …1,1,...., n/4, n/2 adds

-Total adds: 2× (n-1)→ O(n)work

■ The total number of adds is no more than twice of that done in the efficient sequential algorithm

-The benefit of parallelism can easily overcome the 2X work

Definition: (Exclusive) Prefix-Sum (Scan)

Definition: The all-prefix-sums operation takes a binary associative operator ① and an array of n elements

$[X_o, X_1, \ldots, X_{n-1}]$,

and returns the array

$[X_o, (x_o ① x_1), \ldots\ldots, (x_o ① x_1 ① \ldots ① X_{n-2})]$.

Example: If ① is addition, then the all-prefix-sums operation on the array [3 1 7 0 4 1 6 3],

would return [0 3 4 11 11 15 16 22]

Exclusive Scan

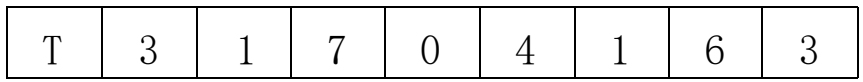
- Find the beginning address of allocated buffers
- Inclusive and Exclusive scans can be easily derived from each other;it is a matter of convenience

[3 1 7 0 4 1 6 3]

Exclusive: [0 3 4 11 11 15 16 22]

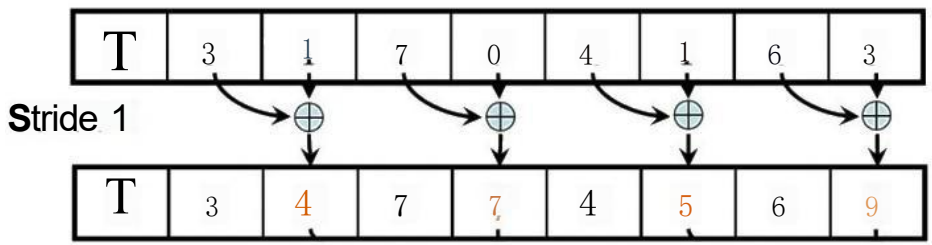
Inclusive: [3 4 11 11 15 16 22 25]

Exclusive Scan Example-Reduction Step



Assume array is already in shared memory

Exclusive Scan Example -Reduction Step

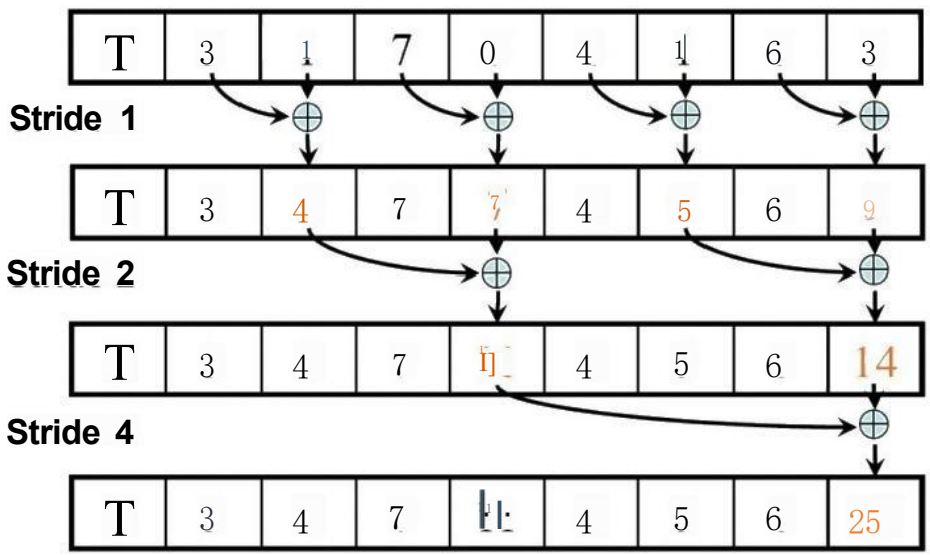


Iteration 1, n/2 threads

Each + corresponds to a single thread.

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

Exclusive Scan Example -Reduction Step



Iteration 2,nl4 threads

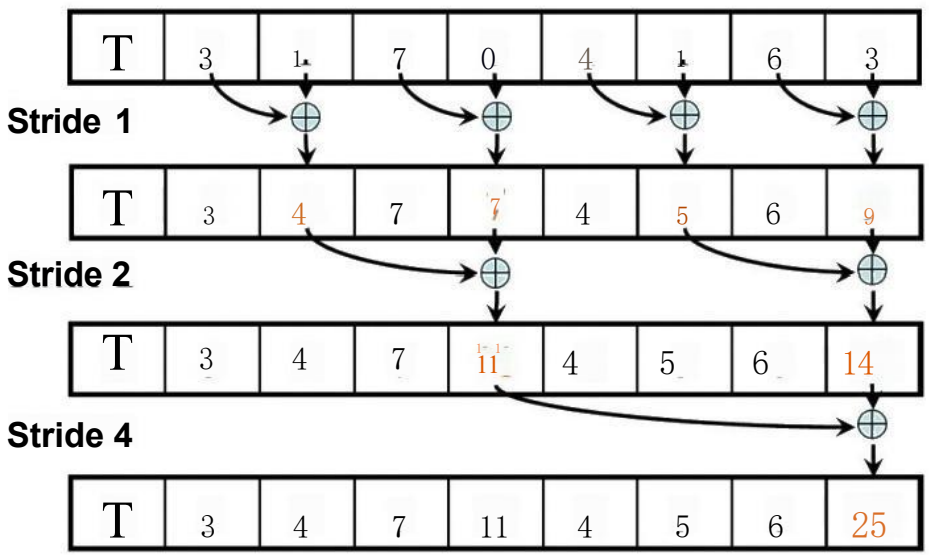
Iteration log(n),1 thread

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

Exclusive Scan Example -Reduction Step



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

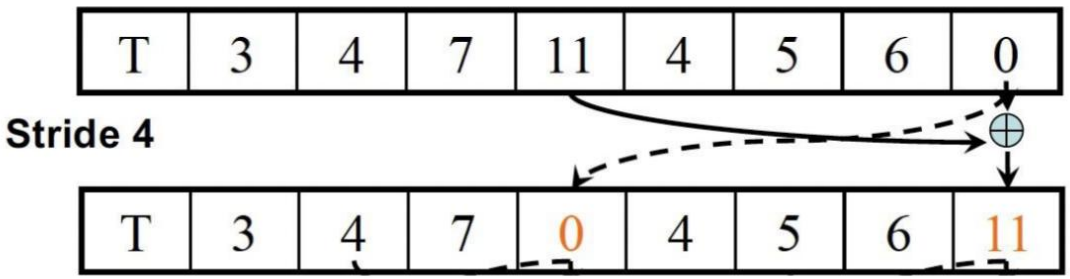
T	3	4	7	11	4	5	6	0

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.

Post Scan Step From Partial Sums

T	3	4	7	11	4	5	6	0

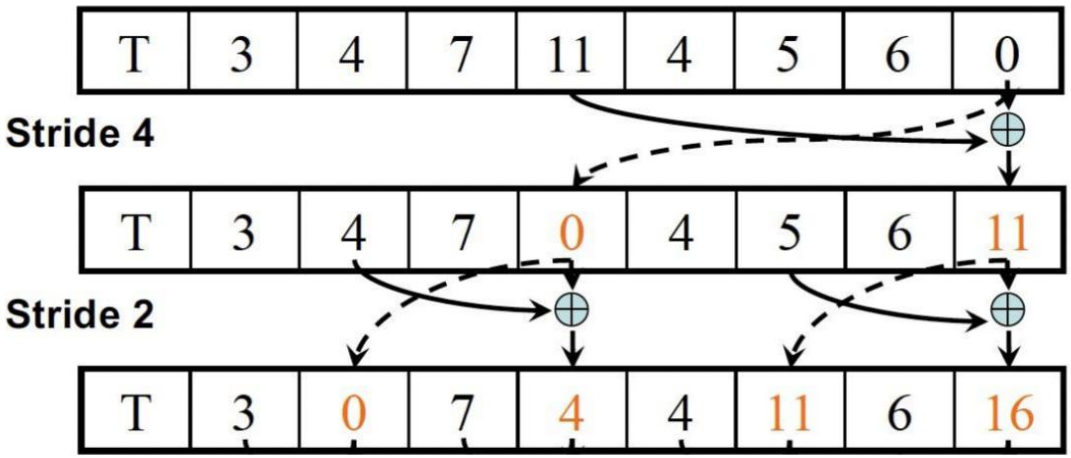
Post Scan Step From Partial Sums



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.

Post Scan Step From Partial Sums

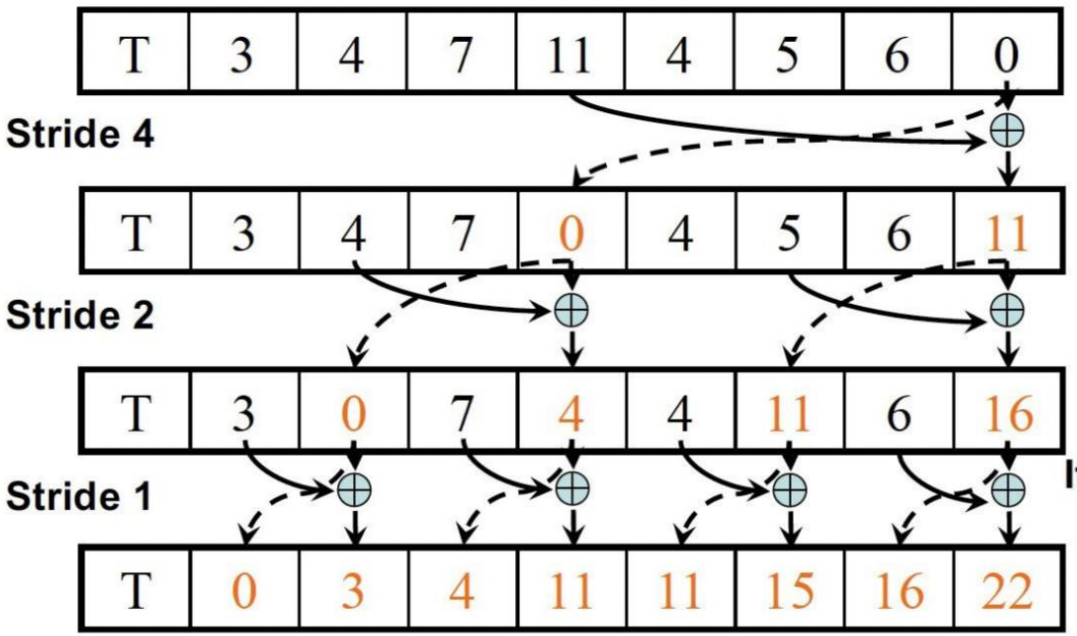


Iteration 2
2 threads

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.

Post Scan Step 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!**

Working on Arbitrary Input Length

Single-Kernel Parallel Scan:Potential for Deadlock

·However, this needs the sum of all preceding blocks to
  -add to the local scan values
  -replace initial value 0 at the start of the upward sweep

·Problem: blocks are not necessarily processed in order,so could end up in deadlock waiting for results from a block which doesn't get a chance to start.

·Could launch multiple kernels on multiple streams,enforce dependency order using events
  -Dependency encoding might be tricky to get right
  -Switching often to CPU degrades performance
  -Any other solutions?

·Solution: use atomic increments

Enforcing Block Processing Order

- Declare a global device variable

```
__device__int ___ my_block_count =0;
```

- At the beginning of the kernel code use

```
__shared __ unsigned int my blockId;
if(threadIdx.x==0)                    {

        my_blockId =atomicInc ( &my_block_count,
                                                UINT _MAX);

}
__syncthreads();
```

·This returns the old value of my_block_count and increments it, all in one operation. The UINT_MAX ensures atomicInt always increments the counter.

- This gives us a way of launching blocks in strict order.

Block-Ordered Global Scan

- For a single-kernel global scan,the kernel does the following:
  - -get in-order block ID
  - -do a downward pass
  - -wait until another global counter my_block_count2 shows that the preceding block has computed the sum of the blocks so far
  - -get the sum of blocks so far, increment the sum with the local partial sum, then increment my_block_count2
  - -do an upwards pass and store the results

Working on Arbitrary Length Input

1.Build on the scan kernel that handles up to 2xblockDim elements

2.Have each section of 2×blockDim elements assigned to each block

3. Have each block write the sum of its section into a Sum array indexed by blockIdx.x

4.Run parallel scan on the Sum array

-May need to breakdown Sum into multiple sections if it is too big for a block

5.Add the scanned Sum array values to the elements of the corresponding sections