

Atomic Operation & Optimizing Parallelism

In This Lecture

- **CUDA Atomic Operation**
- **Histogram Algorithm**
- **Parallel Histogram Computation**
- **Optimizing Parallelism**
- **Example: Reduction Algorithm**

Atomic Operation



Atomic operations

- The basics atomic operation in hardware is something like a **read-modify-write** operation performed by **a single hardware instruction** on a memory location address
- Read the old value, calculate a new value, and write the new value to the old value location
- The hardware ensures that no other threads can perform another read-modify-write operation on the same location until the current atomic operation is complete
- Any other threads that attempt to perform an atomic operation on the same location will typically be held in a queue
- All threads perform their atomic operations serially on the same location

Atomic Operations in CUDA

- Performed by calling functions that are translated into a single instruction (a.k.a. intrinsic functions or intrinsics)
- Operation on a single 32-bit or 64-bit word residing in global or shared memory.
- Atomic functions can only be used in device functions
- Atomic add, sub, inc, dec, min, max, exch (exchange), CAS (compare and swap), and, or, xor

Some examples

- Atomic Sub
 - `int atomicSub(int* address, int val);`
 - reads the 32-bit word old located at the address address in global or shared memory, computes (old - val), and stores the result back to memory at the same address. These three operations are performed in one atomic transaction. The function returns old.
- Atomic Exchange
 - `int atomicExch(int* address, int val);`
 - reads the 32-bit or 64-bit word old located at the address address in global or shared memory and stores val back to memory at the same address. These two operations are performed in one atomic transaction. The function returns old.
- Atomic Max&Min
 - `int atomicMax(int* address, int val);`
 - `int atomicMin(int* address, int val);`
 - reads the 32-bit or 64-bit word old located at the address address in global or shared memory, computes the maximum(or minimum) of old and val, and stores the result back to memory at the same address. These three operations are performed in one atomic transaction. The function returns old.

Some examples

- Atomic Increase&Decrease

- `int atomicInc(int* address, int val);`
 - reads the 32-bit word `old` located at the address `address` in global or shared memory, computes $((old \geq val) ? 0 : (old+1))$, and stores the result back to memory at the same address. These three operations are performed in one atomic transaction. The function returns `old`.
- `int atomicDec(int* address, int val);`
 - reads the 32-bit word `old` located at the address `address` in global or shared memory, computes $((old == 0) \vee (old > val)) ? val : (old-1)$, and stores the result back to memory at the same address. These three operations are performed in one atomic transaction. The function returns `old`.

- Atomic Bitwise Functions

- `int atomicAnd(int* address, int val);`
- `int atomicOr(int* address, int val);`
- `int atomicXor(int* address, int val);`

Examples: atomicAdd

- Atomic Add
 - `int atomicAdd(int* address, int val);`
 - reads the 32-bit word old from the location pointed to by address in global or shared memory, computes (old + val), and stores the result back to memory at the same address. The function returns old.
- Unsigned 32-bit integer atomic add
 - `unsigned int atomicAdd(unsigned int* address, unsigned int val);`
- Unsigned 64-bit integer atomic add
 - `unsigned long long int atomicAdd(unsigned long long int* address, unsigned long long int val);`
- Single-precision floating-point atomic add (capability > 2.0)
 - `float atomicAdd(float* address, float val);`
- Double precision floating-point atomic add (capability > 6.0)
 - `double atomicAdd(double* address, double val);`

Examples: atomicCAS

- `int atomicCAS(int* address, int compare, int val);`
- `unsigned int atomicCAS(unsigned int* address,
 unsigned int compare,
 unsigned int val);`
- `unsigned long long int atomicCAS
 (unsigned long long int* address,
 unsigned long long int compare,
 unsigned long long int val);`

Reads the 32-bit or 64-bit word `old` located at the address in global or shared memory, and computes `(old == compare ? val : old)`, and stores the result back to memory at the same address. These three operations are performed in one atomic transaction. The function returns `old` (Compare And Swap).

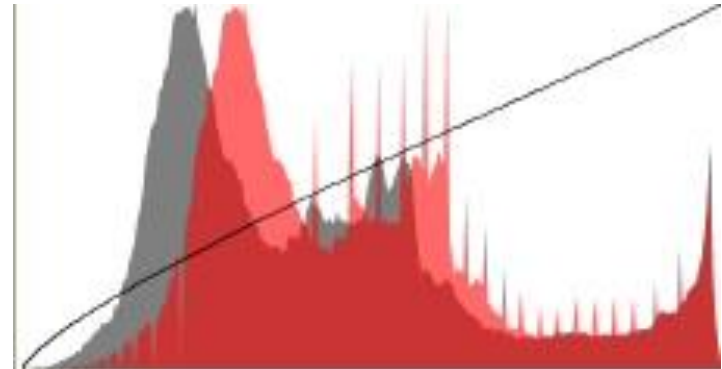
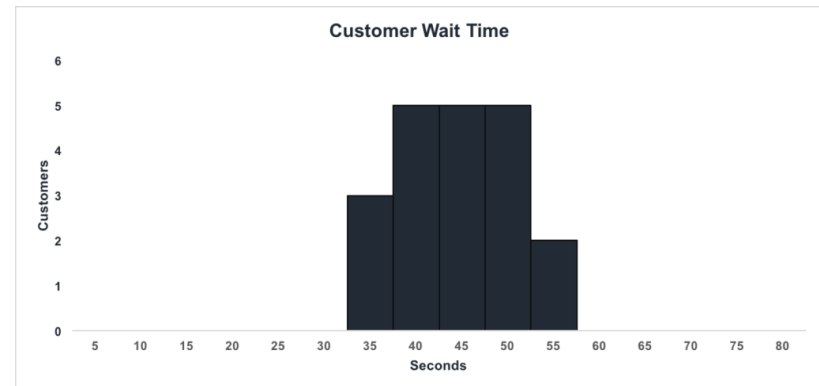
Histogram



Histogram

- Histogram is a representation of the distribution of numerical data.

Customer Wait Time in Seconds (n=20)	
43.1	42.2
35.6	45.5
37.6	30.3
36.5	31.4
45.3	35.6
43.5	45.2
40.3	54.1
50.2	45.6
47.3	36.5
31.2	43.1



Simple Example

- Count each 1 digit numbers from following array.

2	4	3	1	4	4	5	0	7	9	2	3	7	1	7	8	3	7	4	2
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Simple Example with CPU

- Count each 1 digit numbers from following array.

2	4	3	1	4	4	5	0	7	9	2	3	7	1	7	8	3	7	4	2
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

CPU CODE

```
void histogram(int* inputArray, int counts[10], int size){  
    for(int i = 0; i < size; i++){  
        int tmp = inputArray[i];  
        counts[tmp]++;  
    }  
}
```

Simple Example with CUDA

- Count each 1 digit numbers from following array.

2	4	3	1	4	4	5	0	7	9	2	3	7	1	7	8	3	7	4	2
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

CUDA CODE

```
__global__ void histogram(int* inputArray, int* counts, int size){  
    int i = blockIdx.x*blockDim.x + threadIdx.x;  
  
    if(i < size){  
        int tmp = inputArray[i];  
  
        counts[tmp]++;  
    }  
}
```

Simple Example with CUDA

- Count each 1 digit numbers from following array.

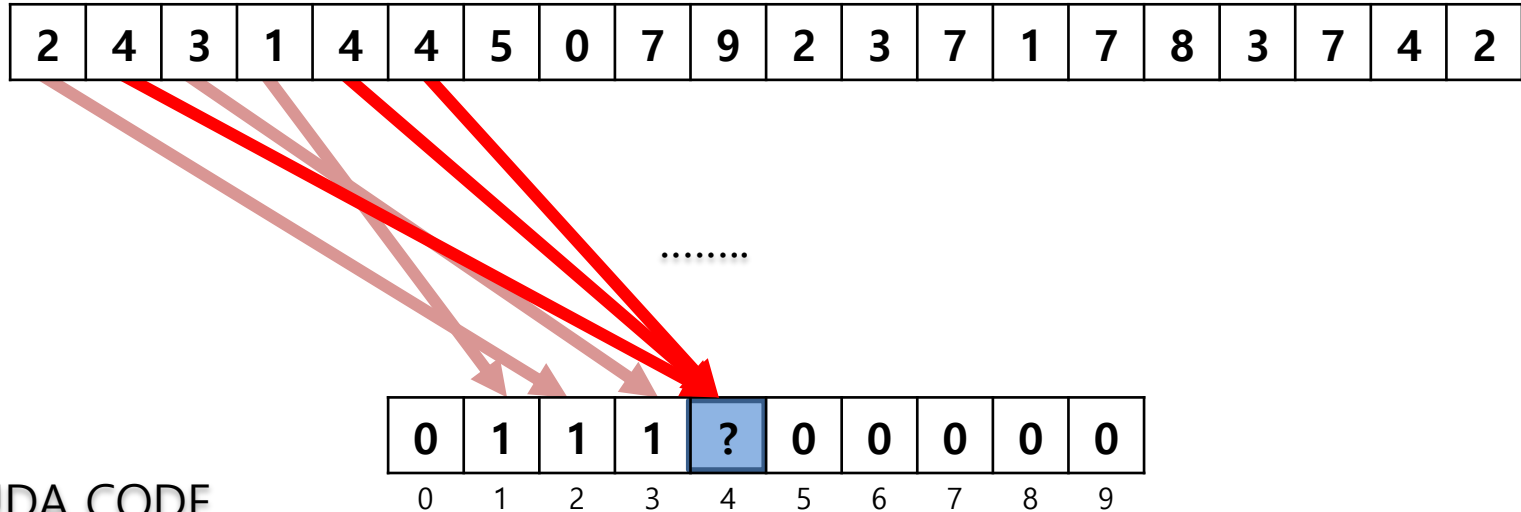
2	4	3	1	4	4	5	0	7	9	2	3	7	1	7	8	3	7	4	2
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

CUDA CODE

```
__global__ void histogram(int* inputArray, int* counts, int size){  
    int i = blockIdx.x*blockDim.x + threadIdx.x;  
    if(i < size){  
        int tmp = inputArray[i];  
        counts[tmp]++;  
    }  
}
```

**Wrong
Way!**

Problem Analysis



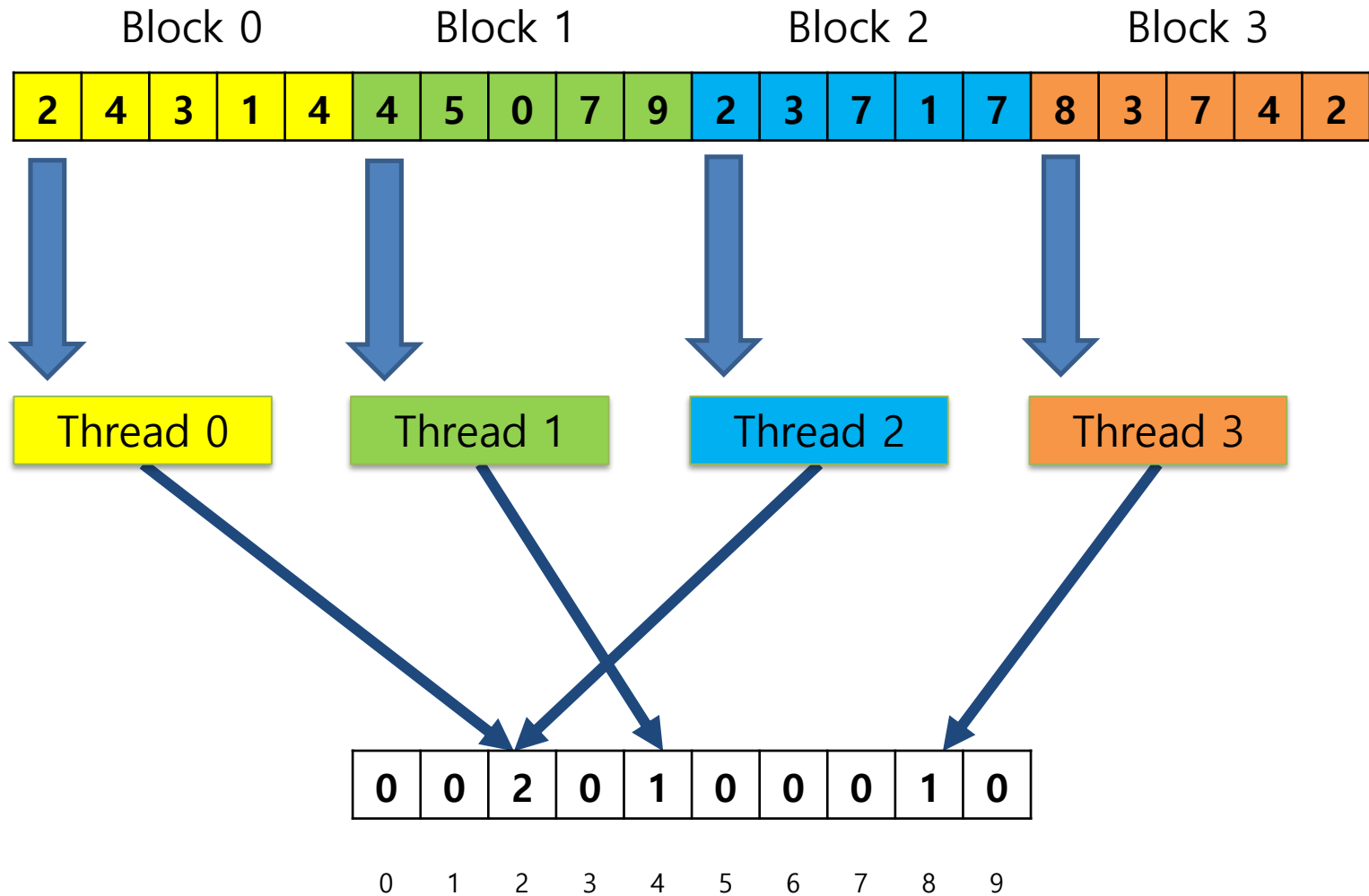
CUDA CODE

```
__global__ void histogram(int* inputArray, int* counts, int size){  
    int i = blockIdx.x*blockDim.x + threadIdx.x;  
  
    if(i < size){  
        int tmp = inputArray[i];  
        counts[tmp]++;  
    }  
}
```

Different threads try to update the same address simultaneously.
Atomic Operation can Solve the Conflict.

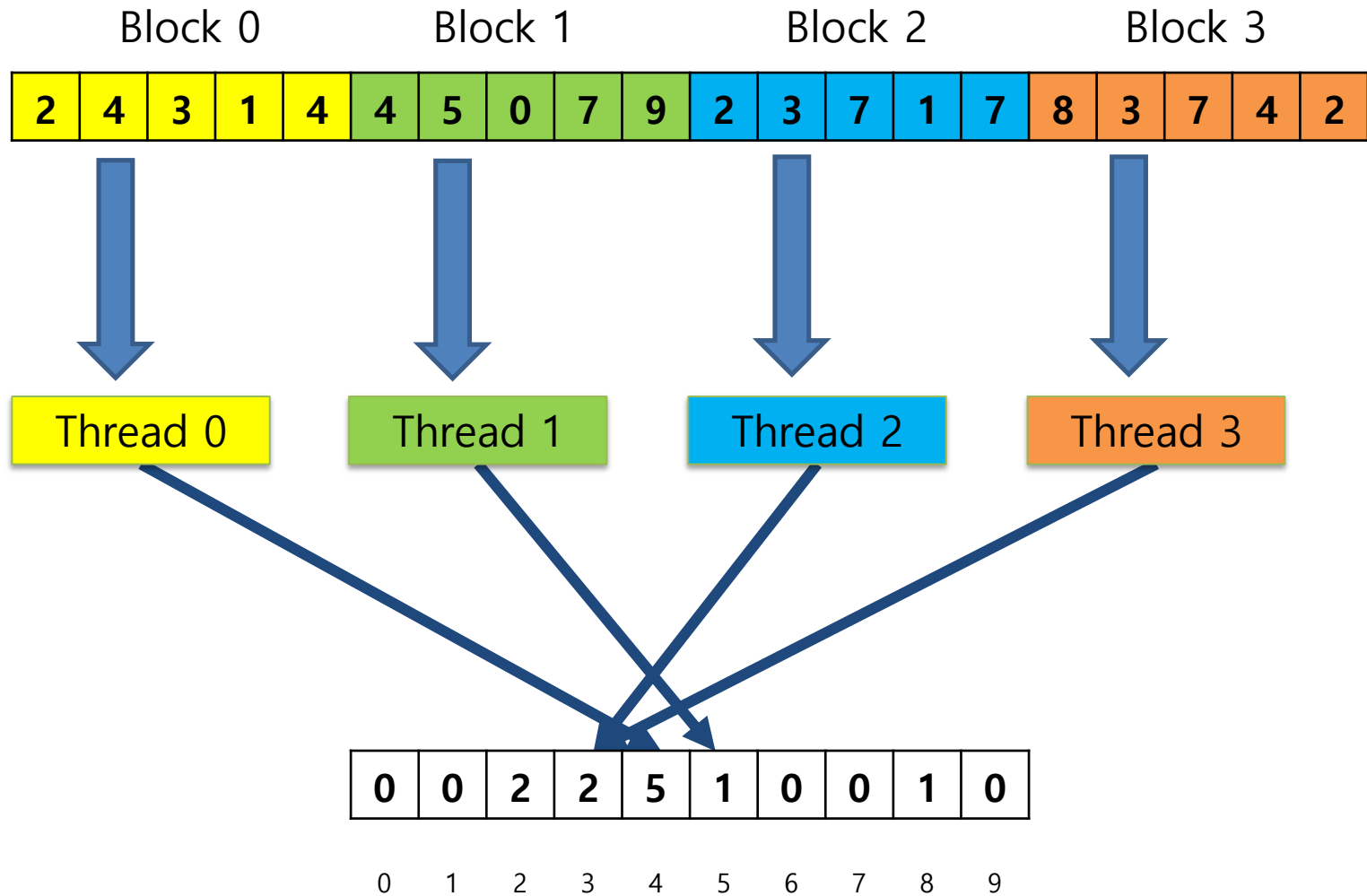
Design1:

Sectioned Partitioning (Iteration #1)



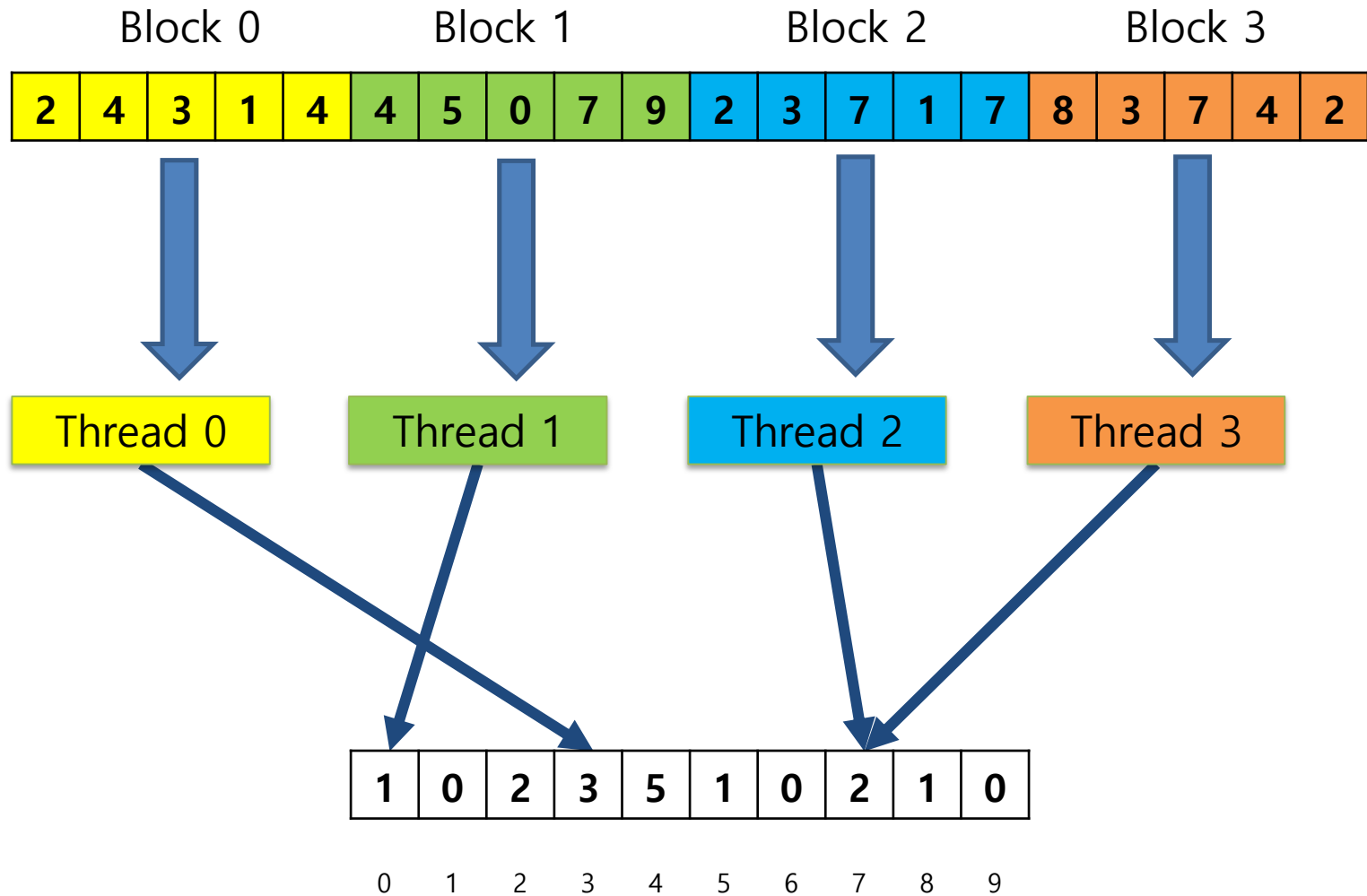
Design1:

Sectioned Partitioning (Iteration #2)



Design1:

Sectioned Partitioning (Iteration #3)



Design1:

Example Code

```
__global__ void histogram_kernel(const int *input, unsigned int *bins,
                                unsigned int num_elements, unsigned int num_bins) {

    unsigned int tid = blockIdx.x * blockDim.x + threadIdx.x;
    int numThreads = (blockDim.x * gridDim.x);
    int num_elementsPerThread = (int)ceil((float)num_elements/numThreads);

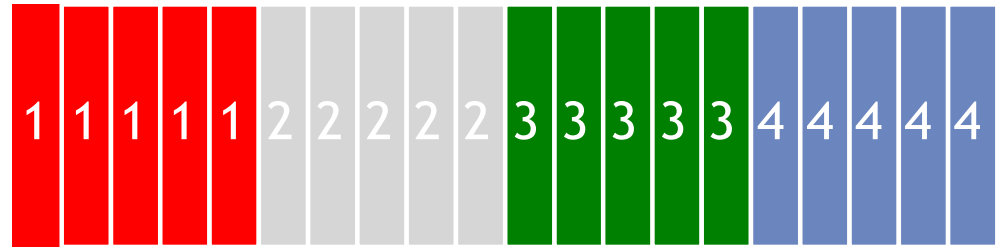
    // Allocate sections to compute for each thread
    for(unsigned int i= tid*num_elementsPerThread; i < (tid+1)*num_elementsPerThread; i++){
        if(i<num_elements){
            int binIdx = input[i];
            atomicAdd(&(bins[binIdx]), 1);
        }
    }
}
```

Improving memory access

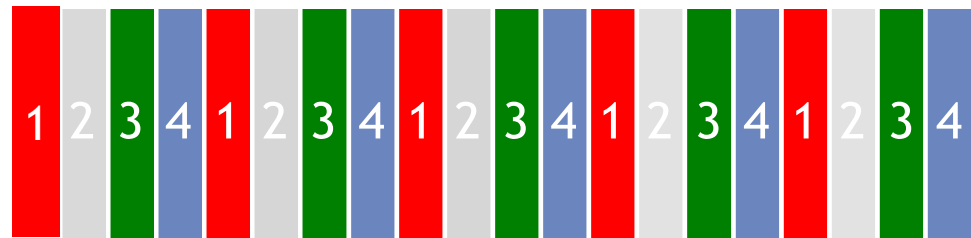
- A simple parallel histogram algorithm partitions the input into sections
- Each section is given to each thread, which iterates through it
- This makes sense in CPU code, where we have few threads, each of which can efficiently use the cache lines when accessing memory
- This access is less convenient in GPUs

Input Partitioning Affects Memory Access Efficiency

- **Sectioned partitioning results in poor memory access efficiency**
 - Adjacent threads do not access adjacent memory locations
 - Accesses are not coalesced
 - DRAM bandwidth is poorly utilized

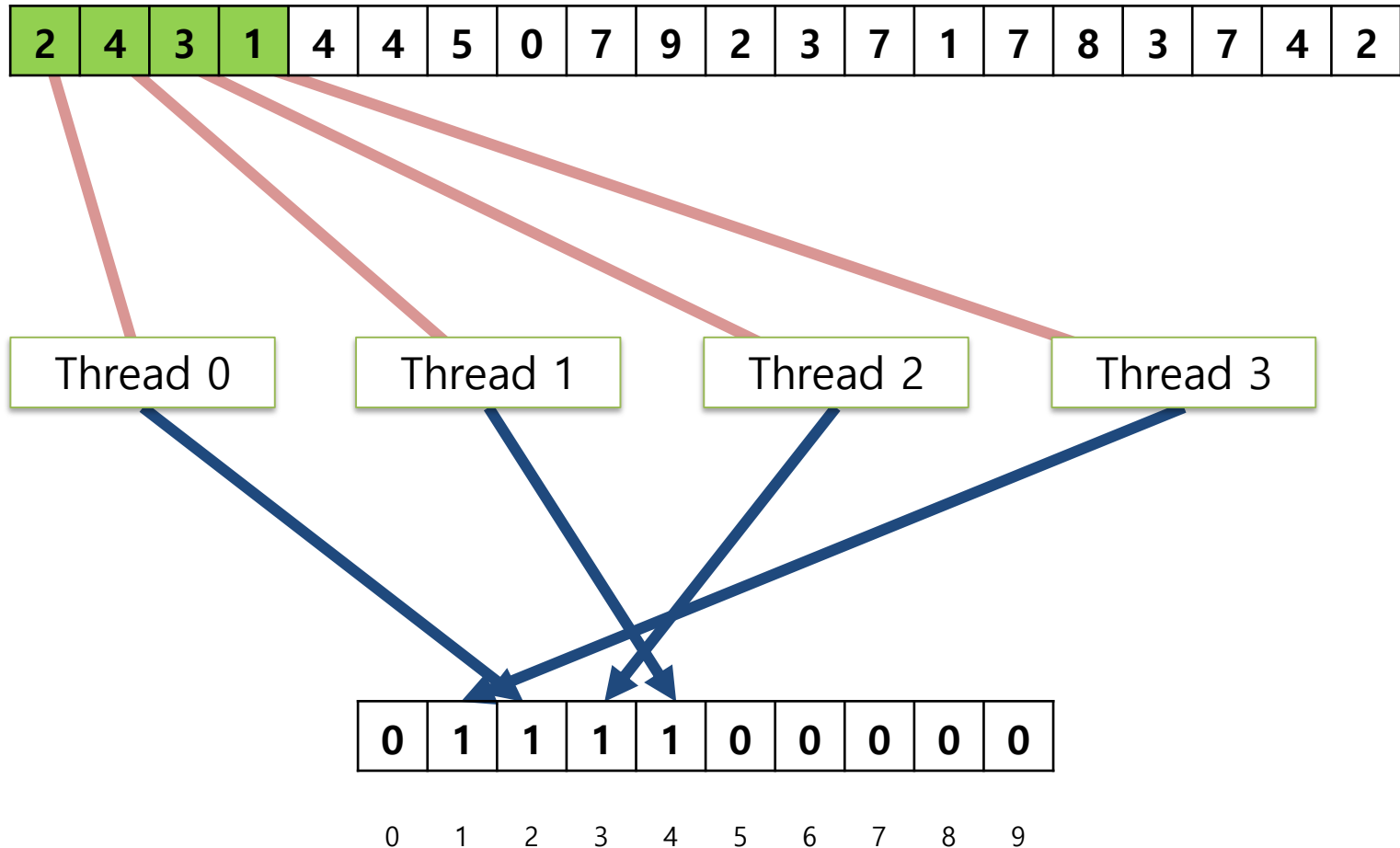


- **Change to interleaved partitioning**
 - All threads process neighboring elements
 - They all move to the next section and repeat
 - The memory accesses are coalesced



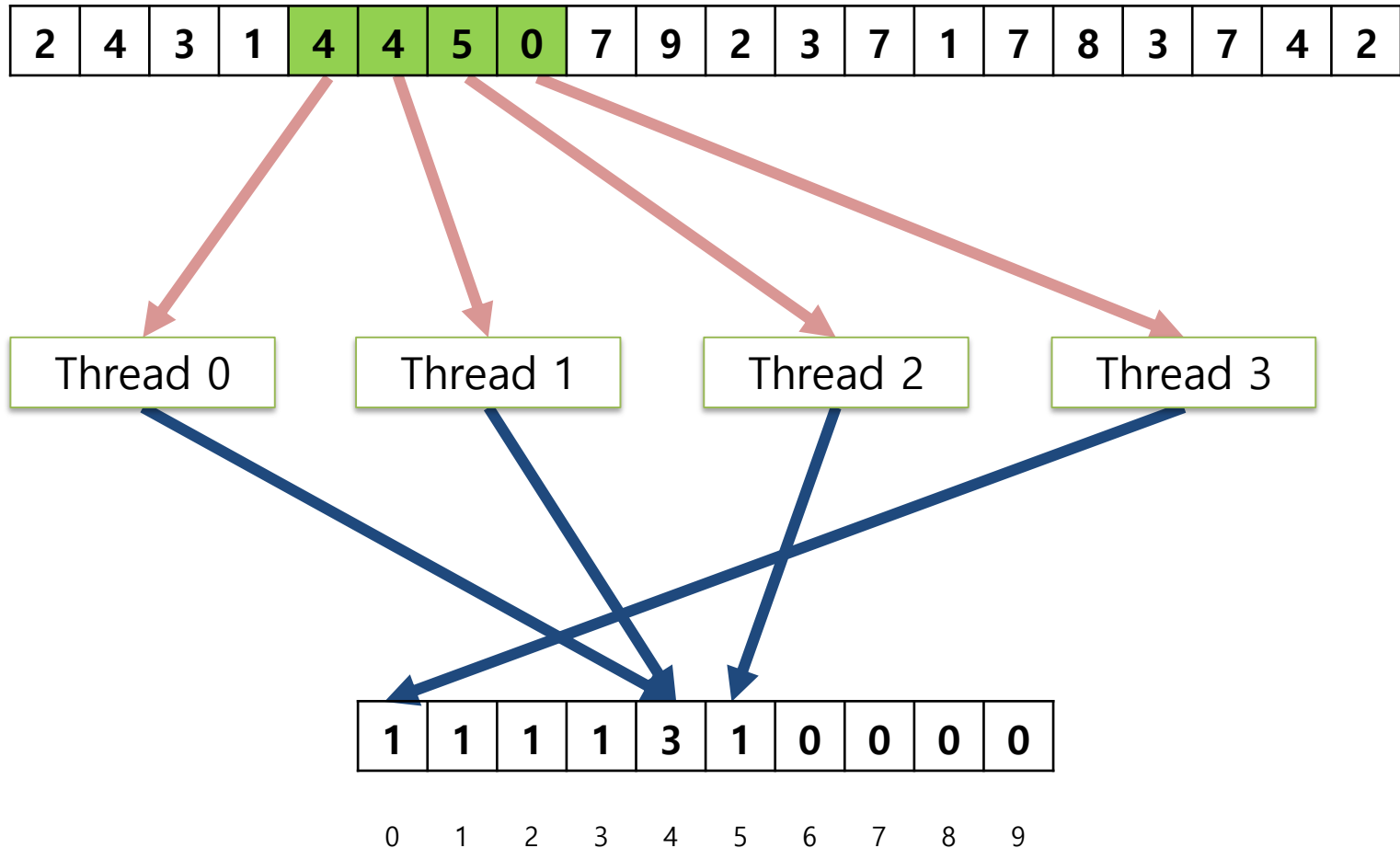
Design 2: Interleaved Partitioning (Iteration 1)

- For coalescing and better memory access performance



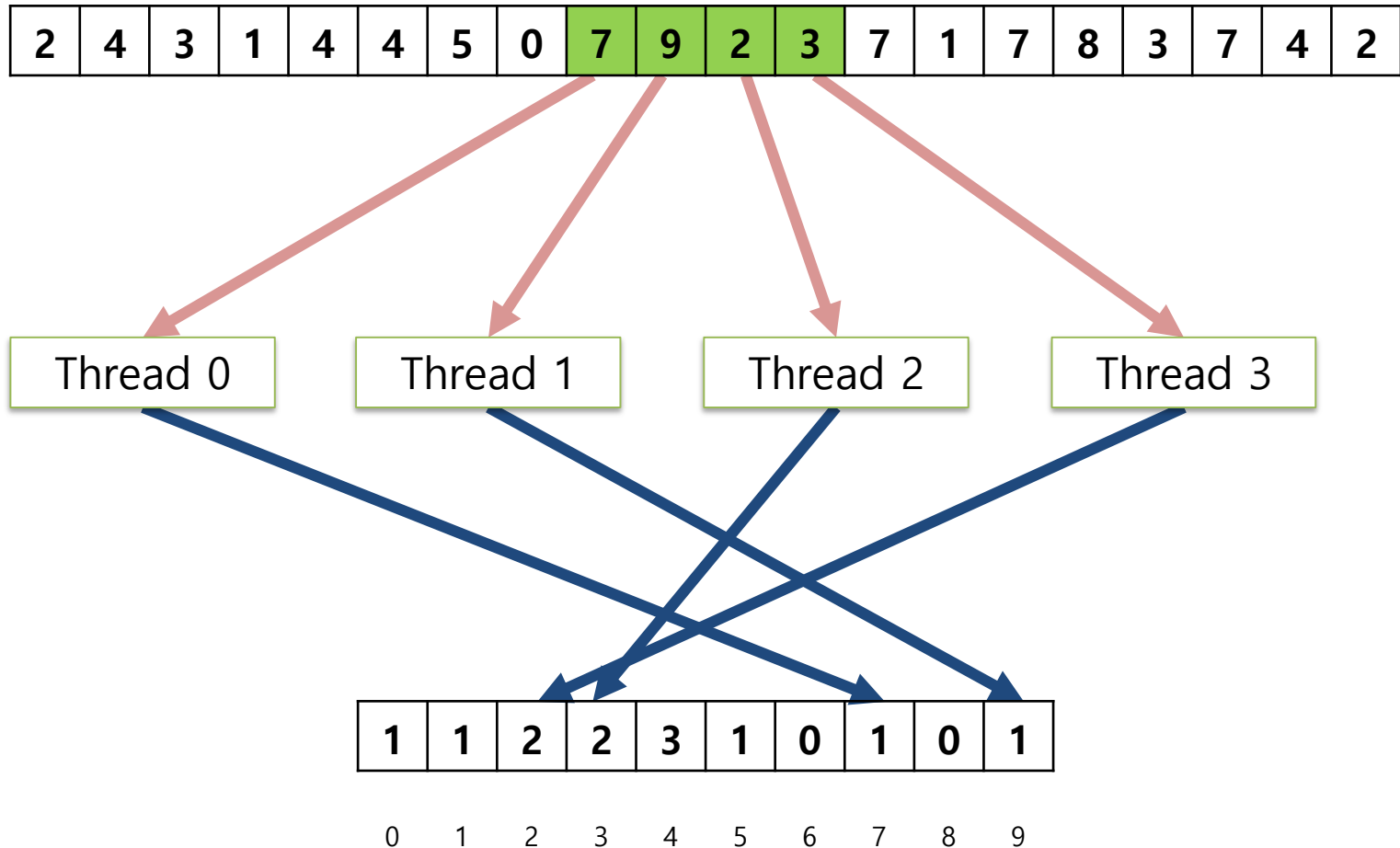
Design 2: Interleaved Partitioning (Iteration 2)

- For coalescing and better memory access performance



Design 2: Interleaved Partitioning (Iteration 3)

- For coalescing and better memory access performance



Design 2:

A stride algorithm Code Example

```
__global__ void histo_kernel(unsigned int *buffer,
                             long size, unsigned int *histo)
{
    int i = threadIdx.x + blockIdx.x * blockDim.x;

    // stride is total number of threads
    int stride = blockDim.x * gridDim.x;

    // All threads handle blockDim.x * gridDim.x
    // consecutive elements
    while (i < size) {
        atomicAdd( &(histo[buffer[i]]), 1);
        i += stride;
    }
}
```

Design 2:

A stride algorithm Code Example

Calculates a **stride** value, which is the total number threads launched during kernel invocation (**blockDim.x*gridDim.x**).

In the first iteration of the while loop, each thread index the input buffer using its global thread index: Thread 0 accesses element 0, Thread 1 accesses element 1, etc. Thus, all threads jointly process the first blockDim.x*gridDim.x elements of the input buffer.

```
int i = threadIdx.x + blockIdx.x * blockDim.x;
```

```
// stride is total number of threads
```

```
int stride = blockDim.x * gridDim.x;
```

```
// All threads handle blockDim.x * gridDim.x
```

```
// consecutive elements
```

```
while (i < size) {
```

```
    atomicAdd( &(histo[buffer[i]]), 1);
```

```
    i += stride;
```

```
}
```

```
}
```

Design 2:

A stride algorithm Code Example

```
__global__ void histo_kernel(unsigned int *buffer,  
                             long size, unsigned int *histo)  
{
```

```
    int i = threadIdx.x + blockIdx.x * blockDim.x;
```

The while loop controls the iterations for each thread. When the index of a thread exceeds the valid range of the input buffer (i is greater than or equal to size), the thread has completed processing its partition and will exit the loop.

```
    // consecutive elements
```

```
    while (i < size) {  
        atomicAdd( &(histo[buffer[i]]), 1);  
        i += stride;  
    }
```

```
}
```

Privatization

- The latency for accessing memory can be dramatically reduced by placing data in the shared memory.
- Shared memory is private to each SM and has very short access latency (a few cycles); this directly translates into increase throughput of atomic operations.
- The problem is that due to the private nature of shared memory, the updates by threads in one thread block is no longer visible to threads in other blocks.

Privatization




- The idea of **privatization** is to replicate highly contended data structures into private copies so that each thread (or each subset of threads) can access a private copy. The benefit is that the private copies can be accessed with much less contention and often at much lower latency.
- These private copies can dramatically increase the throughput for updating the data structures. The down side is that the private copies need to be merged into the original data structure after the computation completes. One must carefully balance between the level of contention and the merging cost.

Practice: Histogram

- **Copy Sample Skeleton Code**
 - `cp -r /home/share/20_Histogram ./[FolderName]`
 - `cd [FolderName]`
- **Notepad: histogram.cu 코드 Kernel function 완성**
- **Compile & run program**
 - `make`
 - `./EXE`

Design 3:

Example Code with Shared Memory

```
__global__ void histogram_kernel(int *input, int *bins,    int num_elements,int num_bins) {  
    // Todo: Call a __syncthreads() at proper position  
  
    int tid = blockIdx.x * blockDim.x + threadIdx.x;  
  
  
    // Privatized bins  
  
    extern __shared__ int bins_s[];  
    for (int binIdx = threadIdx.x; binIdx < num_bins; binIdx += blockDim.x) {  
        bins_s[binIdx] = 0;  
    }  
  
    __syncthreads();  
  
    // Todo: Compute Partition of histogram to Shared Memory  
    for (int i = tid; i < num_elements; i += blockDim.x * gridDim.x) {  
         // atomicAdd to shared memory  
    }  
  
     // wait for other threads  
    //TODO: Commit to global memory  
  
    for (int binIdx = threadIdx.x; binIdx < num_bins; binIdx += blockDim.x) {  
          
    }  
  
}
```


Design 3:

Example Code with Shared Memory

```
__global__ void histogram_kernel(int *input, int *bins,    int num_elements,int num_bins) {  
    // Todo: Call a __syncthreads() at proper position  
  
    int tid = blockIdx.x * blockDim.x + threadIdx.x;  
  
  
    // Privatized bins  
  
    extern __shared__ int bins_s[];  
    for (int binIdx = threadIdx.x; binIdx < num_bins; binIdx += blockDim.x) {  
        bins_s[binIdx] = 0;  
    }  
  
    __syncthreads();  
    // Todo: Compute Partition of histogram to Shared Memory  
    for (int i = tid; i < num_elements; i += blockDim.x * gridDim.x) {  
        atomicAdd(&(bins_s[(unsigned int)input[i]]), 1);  
    }  
  
    __syncthreads();  
    //TODO: Commit to global memory  
    for (int binIdx = threadIdx.x; binIdx < num_bins; binIdx += blockDim.x) {  
        atomicAdd(&(bins[binIdx]), bins_s[binIdx]);  
    }  
}
```

Design 3:

Example Code with Shared Memory

```
__global__ void histogram_kernel(int *input, int *bins,    int num_elements, int num_bins) {  
    // Todo: Call a __syncthreads() at proper position  
  
    int tid = blockIdx.x * blockDim.x + threadIdx.x;  
  
    // Privatized bins  
    extern __shared__ int bins_s[];  
  
    for (int binIdx = threadIdx.x; binIdx < num_bins; binIdx += blockDim.x) {
```

Dynamically allocated shared memory. To allocate it dynamically invoke the kernel with:

```
histogram_kernel<<<gridDim, blockDim, num_bins * sizeof(unsigned int)>>>  
(input, bins, num_elements, num_bins);
```

```
    __syncthreads();  
  
    //TODO: Commit to global memory  
  
    for (int binIdx = threadIdx.x; binIdx < num_bins; binIdx += blockDim.x) {  
        atomicAdd(&(bins[binIdx]), bins_s[binIdx]);  
    }  
}
```

Reduction



Problem Definition: Summing Array

- Sum all elements of the array.

2	4	3	1	4	4	5	0	7	9	2	3	7	1	7	8	3	7	4	2
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Summing Array with CPU

- Sum all elements of the array.

2	4	3	1	4	4	5	0	7	9	2	3	7	1	7	8	3	7	4	2
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

CPU CODE

```
int sumArray(int* inputArray, int size){  
    int tmpSum = 0;  
    for(int i = 0; i < size; i++){  
        tmpSum += inputArray[i];  
    }  
    return tmpSum;  
}
```

Summing Array with GPU

- Sum all elements of the array.

2	4	3	1	4	4	5	0	7	9	2	3	7	1	7	8	3	7	4	2
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

CUDA CODE SIMPLE

```
__global__ void sumArray(int* inputArray, int* result, int size){  
    int i = blockIdx.x*blockDim.x + threadIdx.x;  
    if(i < size){  
        result[0] += inputArray[i];  
    }  
}
```

Summing Array with GPU

- Sum all elements of the array.

2	4	3	1	4	4	5	0	7	9	2	3	7	1	7	8	3	7	4	2
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

CUDA CODE SIMPLE

```
__global__ void sumA(int* inputArra, int* result, int size){  
    int i = blockIdx.x*blockDim.x + threadIdx.x;  
    if(i < size){  
        result[0] += inputArra[i];  
    }  
}
```

**Wrong
Way!**

Summing Array with GPU

- Sum all elements of the array.

2	4	3	1	4	4	5	0	7	9	2	3	7	1	7	8	3	7	4	2
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

CUDA CODE w/ Atomics

```
__global__ void sumArray(int* inputArray, int* result, int size){  
    int i = blockIdx.x*blockDim.x + threadIdx.x;  
  
    if(i < size){  
        atomicAdd(&result[0], inputArray[i]);  
    }  
}
```


Summing Array with GPU

- Sum all elements of the array.

2	4	3	1	4	4	5	0	7	9	2	3	7	1	7	8	3	7	4	2
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

CUDA CODE w/ Atomics

```
__global__ void sumArray(int* inputArray, int* result, int size){  
    int i = blockIdx.x*blockDim.x + threadIdx.x;  
  
    if(i < size){  
        atomicAdd(&result[0], inputArray[i]);  
    }  
}
```

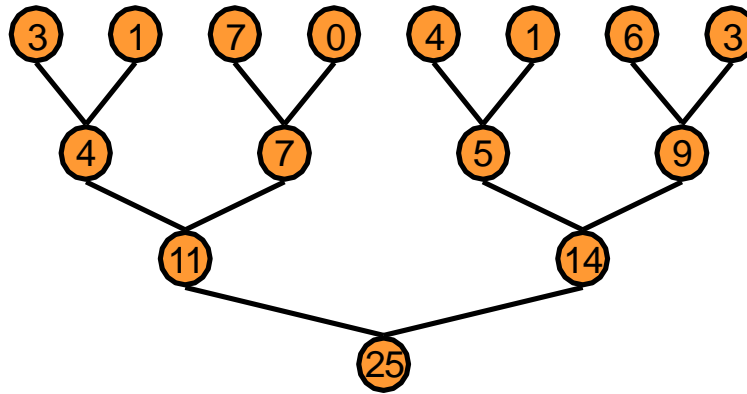
Serial Computation

Parallel Reduction

- **Common and important data parallel primitive**
- **Easy to implement in CUDA**
 - Harder to get it right
- **Serves as a great optimization example**
 - Demonstrates several important optimization strategies

Parallel Reduction

- **Tree-based approach used within each thread block**



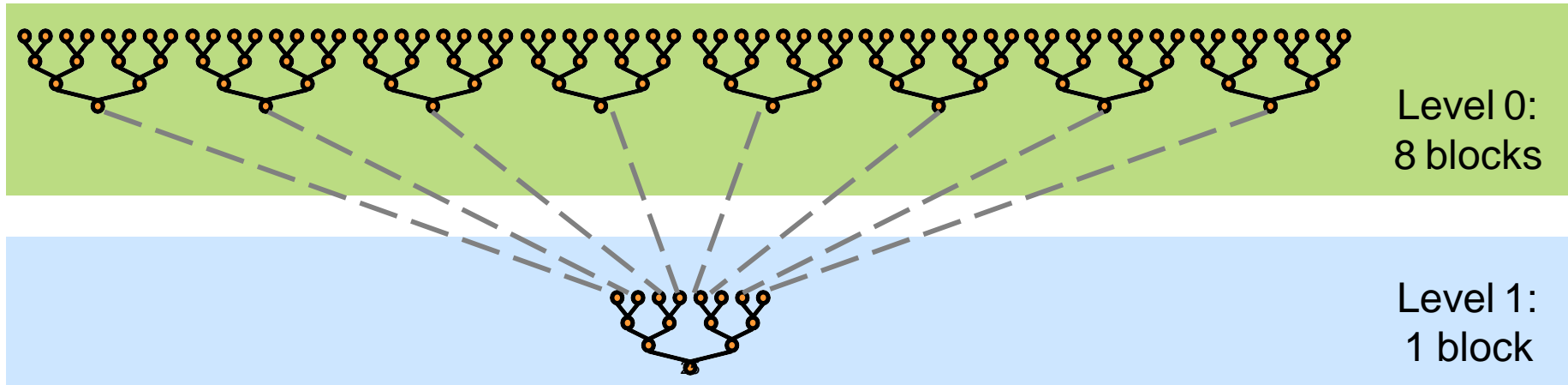
- **Need to use multiple thread blocks**
 - To process very large arrays
 - To keep all multiprocessors on the GPU busy
 - Each thread block reduces a portion of the array
- **But how do we communicate partial results between thread blocks?**

Problem: Global Synchronization

- **If we can synchronize across all thread blocks, we can easily reduce very large arrays**
 - Global sync after each block produces its result
once all blocks reach sync, continue recursively
- **But CUDA has no global synchronization. Why?**
 - Expensive to build in hardware for GPUs with high processor count
 - Would force programmer to run fewer blocks
 - no more than total available resident blocks
 $(\# \text{ of multiprocessors}) \times (\# \text{ of resident blocks per multiprocessor})$
to avoid deadlock, which may reduce overall efficiency
- **Solution: decompose into multiple kernels**
 - Kernel launch serves as a global synchronization point
 - Kernel launch has negligible HW overhead, low SW overhead

Solution: Kernel Decomposition

Global sync by decomposing computation into multiple kernel invocations



In the case of reductions, code for all levels is the same

Recursive kernel invocation

What is Our Optimization Goal?

- **We should strive to reach GPU peak performance**
- **Choose the right metric:**
 - GFLOP/s: for compute-bound kernels
 - Bandwidth: for memory-bound kernels
- **Reductions have very low arithmetic intensity**
 - 1 flop per element loaded (bandwidth-optimal)
- **Therefore we should strive for peak bandwidth**
- **With a GPU of throughput**
 - 384-bit memory interface, 900 MHz DDR
 - $384 * 1800 / 8 = 86.4 \text{ GB/s}$

Reduction #1: Interleaved Addressing

```
__global__ void reduce0(int *g_idata, int *g_odata) {

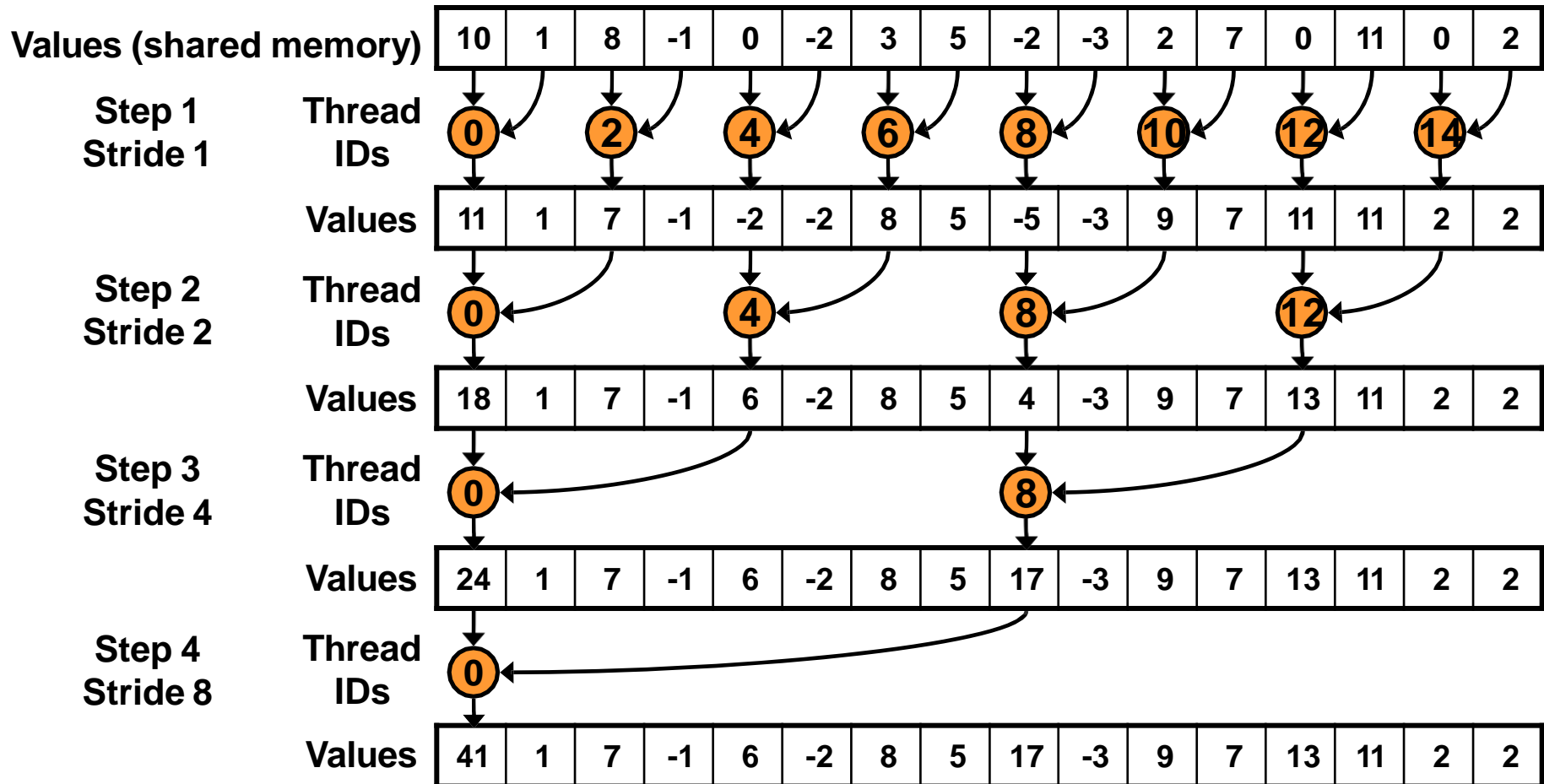
    extern __shared__ int sdata[];

    // each thread loads one element from global to shared mem
    unsigned int tid = threadIdx.x;
    unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
    sdata[tid] = g_idata[i];
    __syncthreads();

    // do reduction in shared mem
    for(unsigned int s=1; s < blockDim.x; s *= 2) {
        if (tid % (2*s) == 0) {
            sdata[tid] += sdata[tid + s];
        }
        __syncthreads();
    }

    // write result for this block to global mem
    if (tid == 0) g_odata[blockIdx.x] = sdata[0]
    ;
}
```

Parallel Reduction: Interleaved Addressing



Reduction #1: Interleaved Addressing

```
__global__ void reduce1(int *g_idata, int *g_odata) {

    extern __shared__ int sdata[];

    // each thread loads one element from global to shared mem
    unsigned int tid = threadIdx.x;
    unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
    sdata[tid] = g_idata[i];
    __syncthreads();

    // do reduction in shared mem
    for (unsigned int s=1; s < blockDim.x; s *= 2) {
        if (tid % (2*s) == 0) {
            sdata[tid] += sdata[tid + s];
        }
        __syncthreads();
    }

    // write result for this block to global mem
    if (tid == 0) g_odata[blockIdx.x] = sdata[0];
}
```

Problem: highly divergent warps are very inefficient, and % operator is very slow

Performance for 4M element reduction

	Time (2^{22} ints)		Bandwidth	
Kernel 1: interleaved addressing with divergent branching	8.054 ms		2.083 GB/s	

Note: Block Size = 128 threads for all tests

Reduction #2: Interleaved Addressing

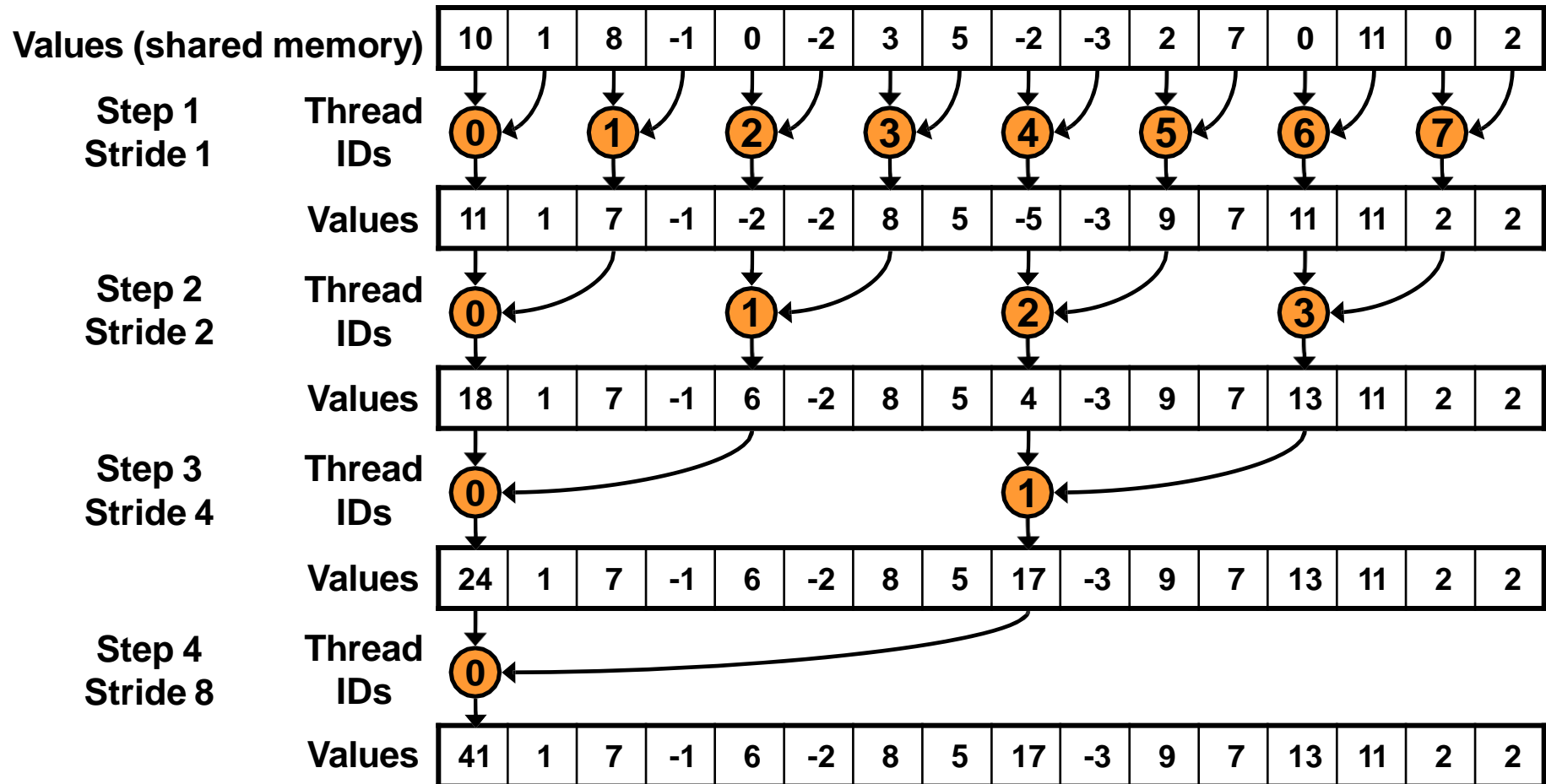
Just replace divergent branch in inner loop:

```
for (unsigned int s=1; s < blockDim.x; s *= 2) {  
    if (tid % (2*s) == 0) {  
        sdata[tid] += sdata[tid + s];  
    }  
    __syncthreads();  
}
```

With strided index and non-divergent branch:

```
for (unsigned int s=1; s < blockDim.x; s *= 2) {  
    int index = 2 * s * tid;  
    if (index < blockDim.x) {  
        sdata[index] += sdata[index + s];  
    }  
    __syncthreads();  
}
```

Parallel Reduction: Interleaved Addressing

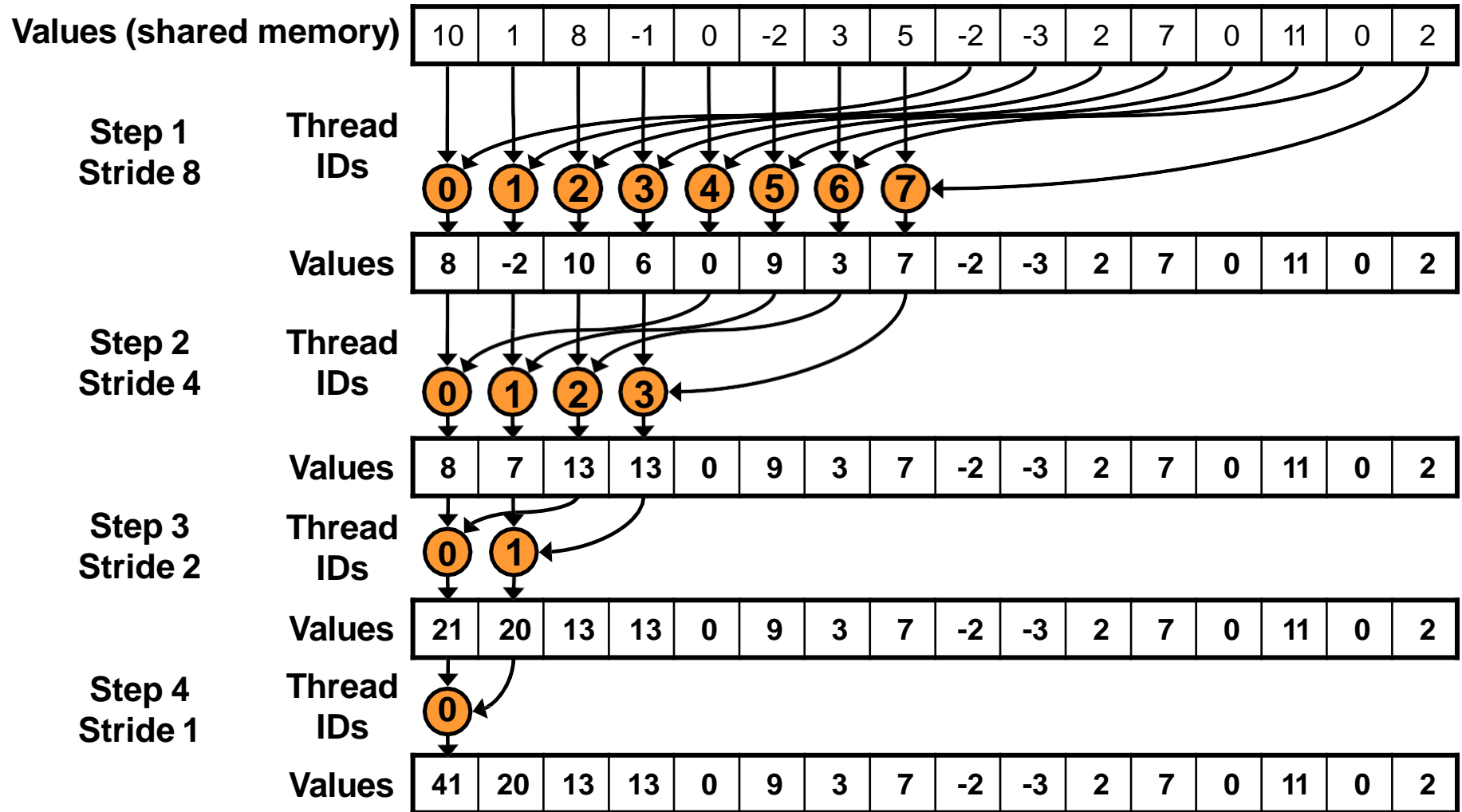


New Problem: Shared Memory Bank Conflicts

Performance for 4M element reduction

	Time (2^{22} ints)	Bandwidth	Step Speedup	Cumulative Speedup
Kernel 1: interleaved addressing with divergent branching	8.054 ms	2.083 GB/s		
Kernel 2: interleaved addressing with bank conflicts	3.456 ms	4.854 GB/s	2.33x	2.33x

Parallel Reduction: Sequential Addressing



Sequential addressing is conflict free

Practice: Reduction

- **Copy Sample Skeleton Code**
 - `cp -r /home/share/21_Reduction ./[FolderName]`
 - `cd [FolderName]`
- **Notepad: reduction.cu 코드 Kernel function 완성**
- **Compile & run program**
 - `make`
 - `./EXE`

Reduction #3: Kernel Function

```
__global__ void reduction(int *g_idata, int *g_odata, int n){
    // Todo: Call two __syncthreads() functions at proper position
    extern __shared__ int sdata[];

    // perform first level of reduction,
    // reading from global memory, writing to shared memory
    int tid = threadIdx.x;
    int i = blockIdx.x*blockDim.x + threadIdx.x;
    sdata[tid] = (i < n) ? g_idata[i] : 0;
    __syncthreads();

    // do reduction in shared mem
    for (int s=blockDim.x/2; s>0; s>>=1)
    {
        if (tid < s){
            sdata[tid] += sdata[tid + s];
        }
        __syncthreads();
    }
    //Todo: Write result for this block to global mem
    if(tid == 0) g_odata[blockIdx.x] = sdata[0];
}
```


Reduction #3: Launch Kernel Function

```
int main(){  
    .....  
    // Set Grid/Block Dimensions  
    int T = 1<<7;//2^7=128: T must be 2^n  
    int B = (int)ceil((float)ARRAY_SIZE/T);  
    int inputSize = ARRAY_SIZE;  
    // Launch Kernel  
    while(true){  
        reduction<<<B, T, T*sizeof(int)>>>(d_Array,d_Sum,inputSize);  
        if(B == 1) break; //last Step  
        inputSize = B;  
        B=(int)ceil((float)B/T);  
        /*pointer Swap*/  
        {  
            int* tmp = d_Sum;  
            d_Sum = d_Array;  
            d_Array = tmp;  
        }  
    }  
    .....  
}
```

Reduction #3: Sequential Addressing

Just replace strided indexing in inner loop:

```
for (unsigned int s=1; s < blockDim.x; s *= 2) {  
    int index = 2 * s * tid;  
    if (index < blockDim.x) {  
        sdata[index] += sdata[index + s];  
    }  
    __syncthreads();  
}
```

With reversed loop and threadID-based indexing:

```
for (unsigned int s=blockDim.x/2; s>0; s>>=1) {  
    if (tid < s) {  
        sdata[tid] += sdata[tid + s];  
    }  
    __syncthreads();  
}
```

Performance for 4M element reduction

	Time (2^{22} ints)	Bandwidth	Step Speedup	Cumulative Speedup
Kernel 1: interleaved addressing with divergent branching	8.054 ms	2.083 GB/s		
Kernel 2: interleaved addressing with bank conflicts	3.456 ms	4.854 GB/s	2.33x	2.33x
Kernel 3: sequential addressing	1.722 ms	9.741 GB/s	2.01x	4.68x