# Технології графічного процесінгу

## (Масивно-паралельні обчислення на графічних прискорювачах
## …
## Massively Parallel Computing on Graphic Processing Units – GPUs)

# Lecture 4. CUDA – Parallel Patterns

Yuri G. Gordienko
(NTUU-KPI, 2021)

(on the basis of materials by NVIDIA, R.Franklin, S.Sengupta, J.Dean, W.Hwu, D.Kirk)

# What is Beyond these Trees? **Forest!**

- Before, we've concerned ourselves with low-level details of kernel programming
  - Cores
  - Threads
  - Grids
  - `__shared__` memory management
  - Resource allocation

- The huge number of details and small parts

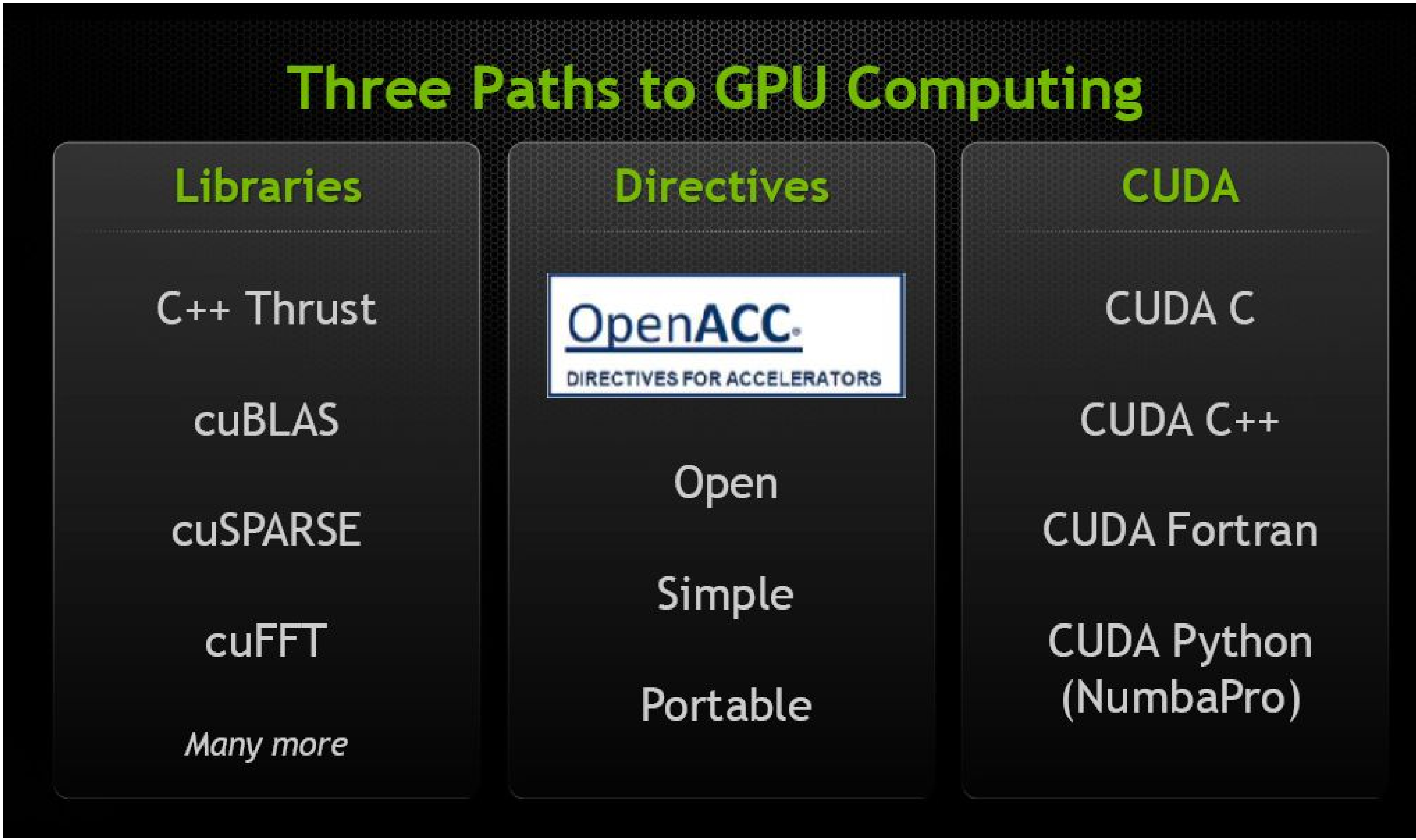
- Hard to see the forest for the trees

# Compute Capability

- The **compute capability** of a device describes its architecture, e.g.
  - Number of registers
  - Sizes of memories
  - Features & capabilities

| Compute Capability | Selected Features (see CUDA C Programming Guide for complete list) | GPU models |
|---|---|---|
| 1.0 | Fundamental CUDA support | Tesla C870 |
| 1.3 | Double precision, improved memory accesses, atomics | Tesla 10-series |
| 2.0 | Caches, fused multiply-add, 3D grids, surfaces, ECC, P2P, concurrent kernels/copies, function pointers, recursion | Tesla 20-series |
| **2.1** | **-** | **GTX 560** |
| **3.5** | **Warp shuffle functions, funnel shift, dynamic parallelism** | **Tesla K40** |

# Current trends in GPU programming



# Parallel Patterns: Introduction

# Parallel Patterns

- Think at a higher level than individual CUDA kernels
- Specify **what** to compute, not **how** to compute it
- Let programmer worry about algorithm
- Defer pattern implementation to someone else

# Parallel Computing Scenarios

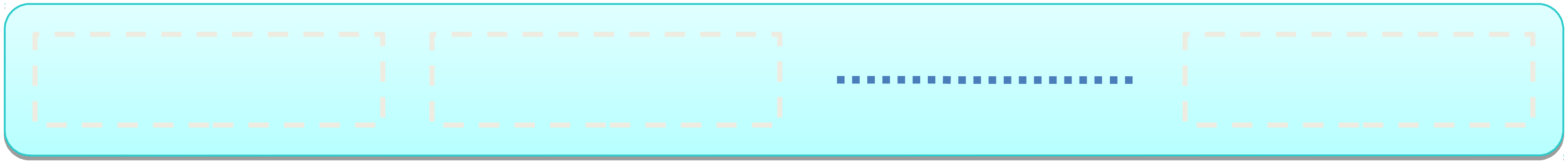
- Many parallel threads need to generate a single result
  - **Reduce**
- Many parallel threads need to partition data
  - **Split**
- Many parallel threads produce variable output / thread
  - **Compact / Expand**

# Parallel Patterns:
# CUDA Memory Pattern

# CUDA Memory Pattern: Blocking

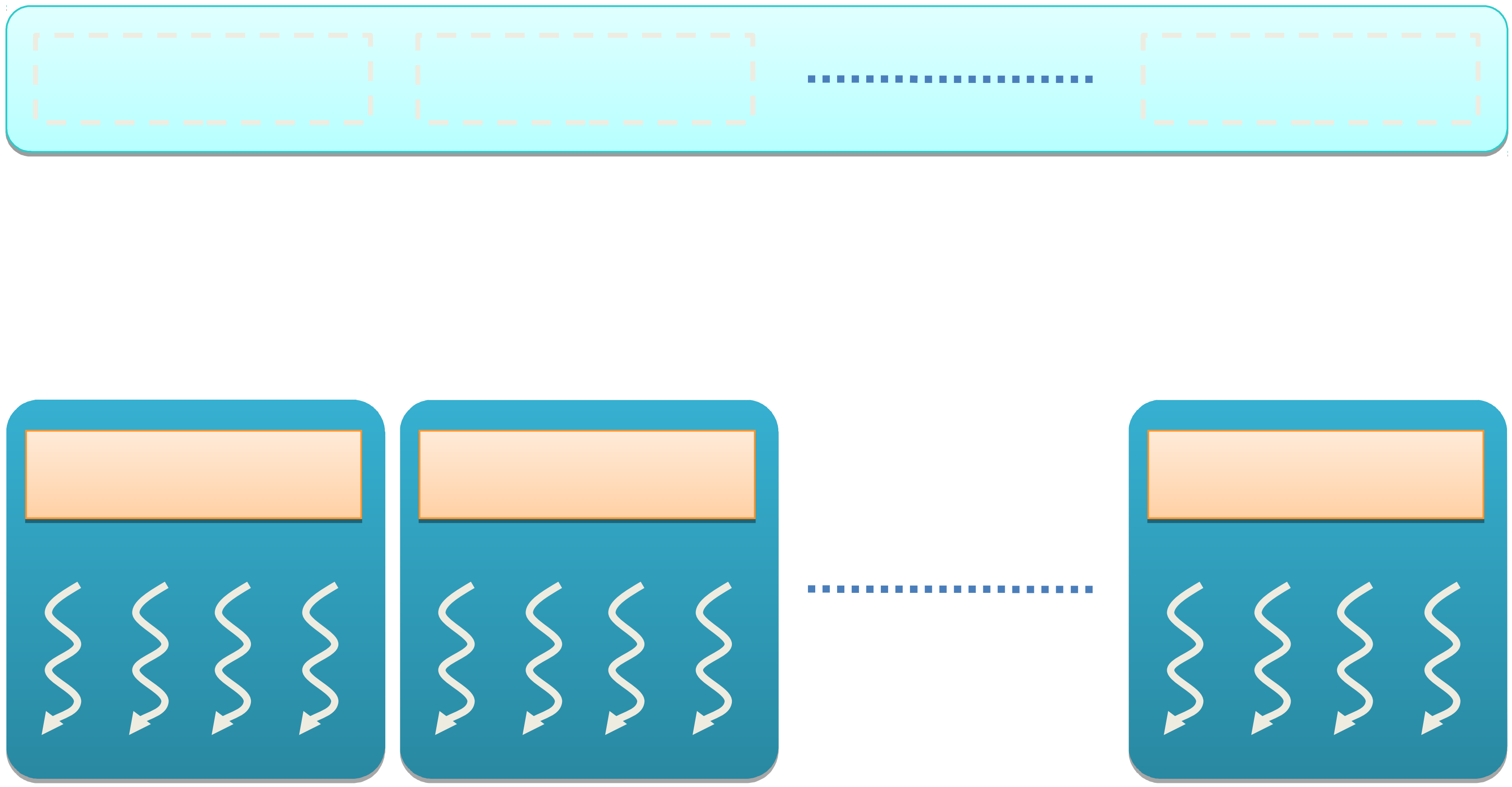
- Partition data to operate in well-sized blocks
  - Small enough to be staged in shared memory
  - Assign each data partition to a thread block
  - No different from cache blocking!
- Provides several performance benefits
  - Have enough blocks to keep processors busy
  - Working in shared memory cuts memory latency dramatically
  - Likely to have coherent access patterns on load/store to shared memory

# Globally Shared Pattern: Blocking



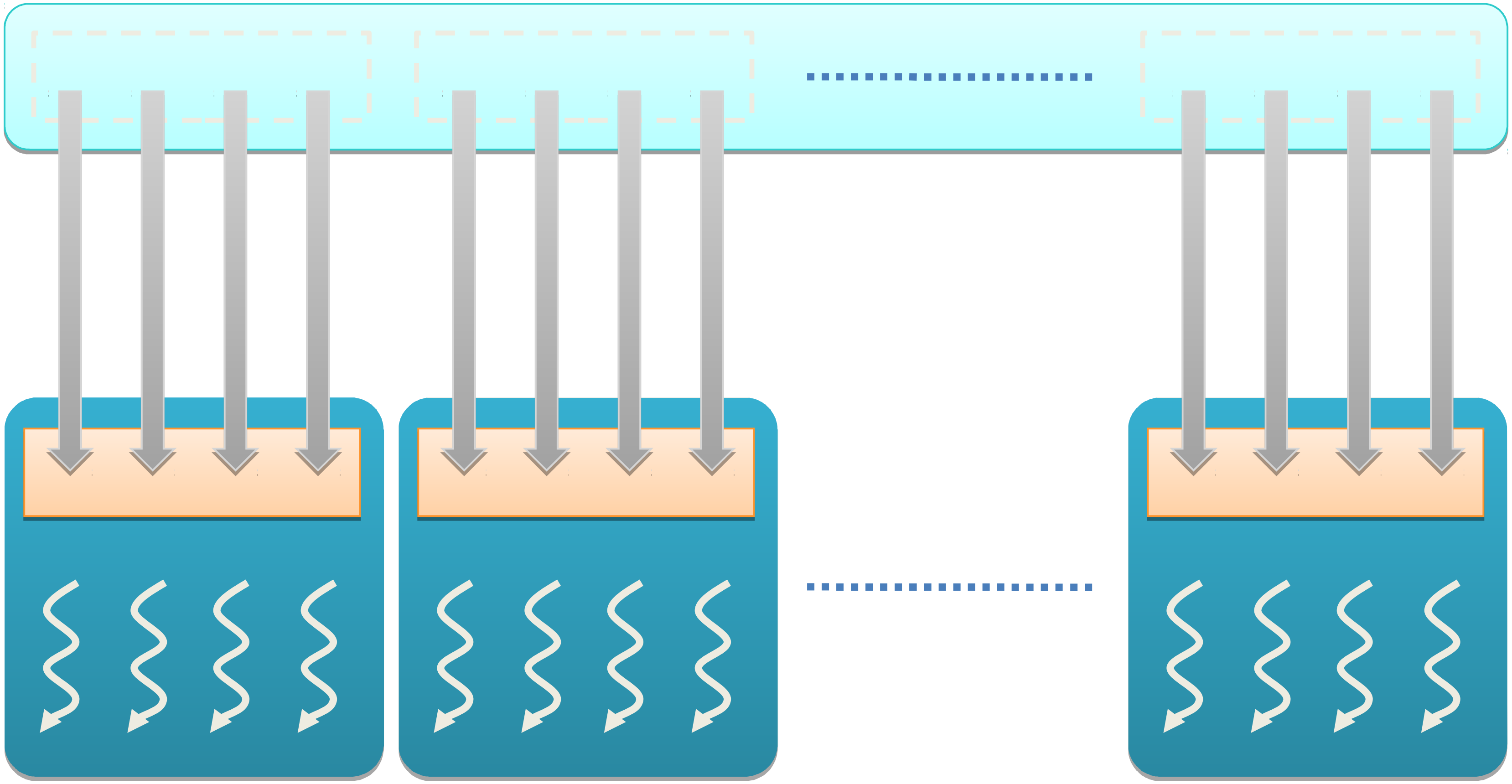
- **Partition** data into **subsets** that fit into **shared memory**

# Locally Shared Pattern : Blocking



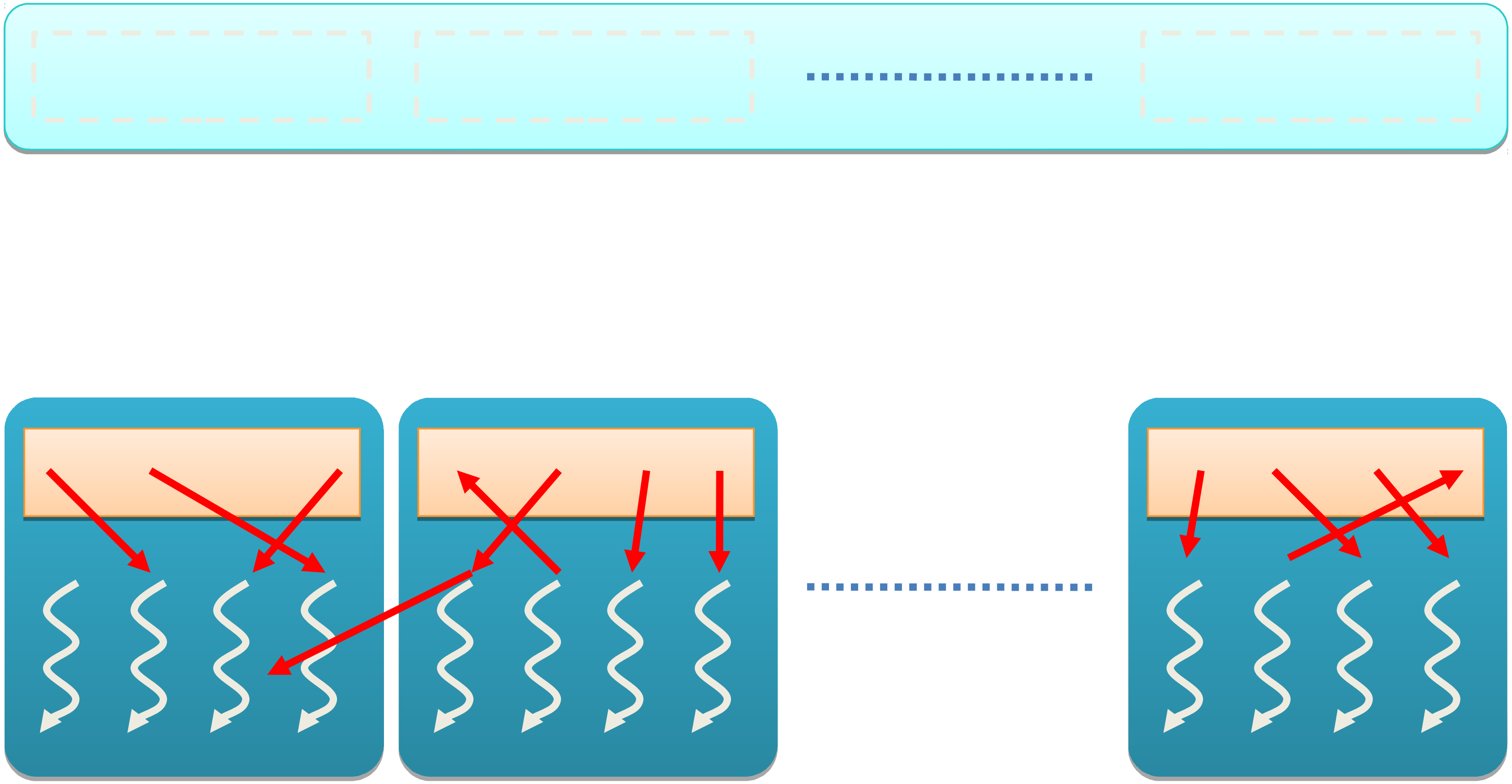
- Handle each data subset with one **thread block**

# Locally Shared Pattern : Blocking



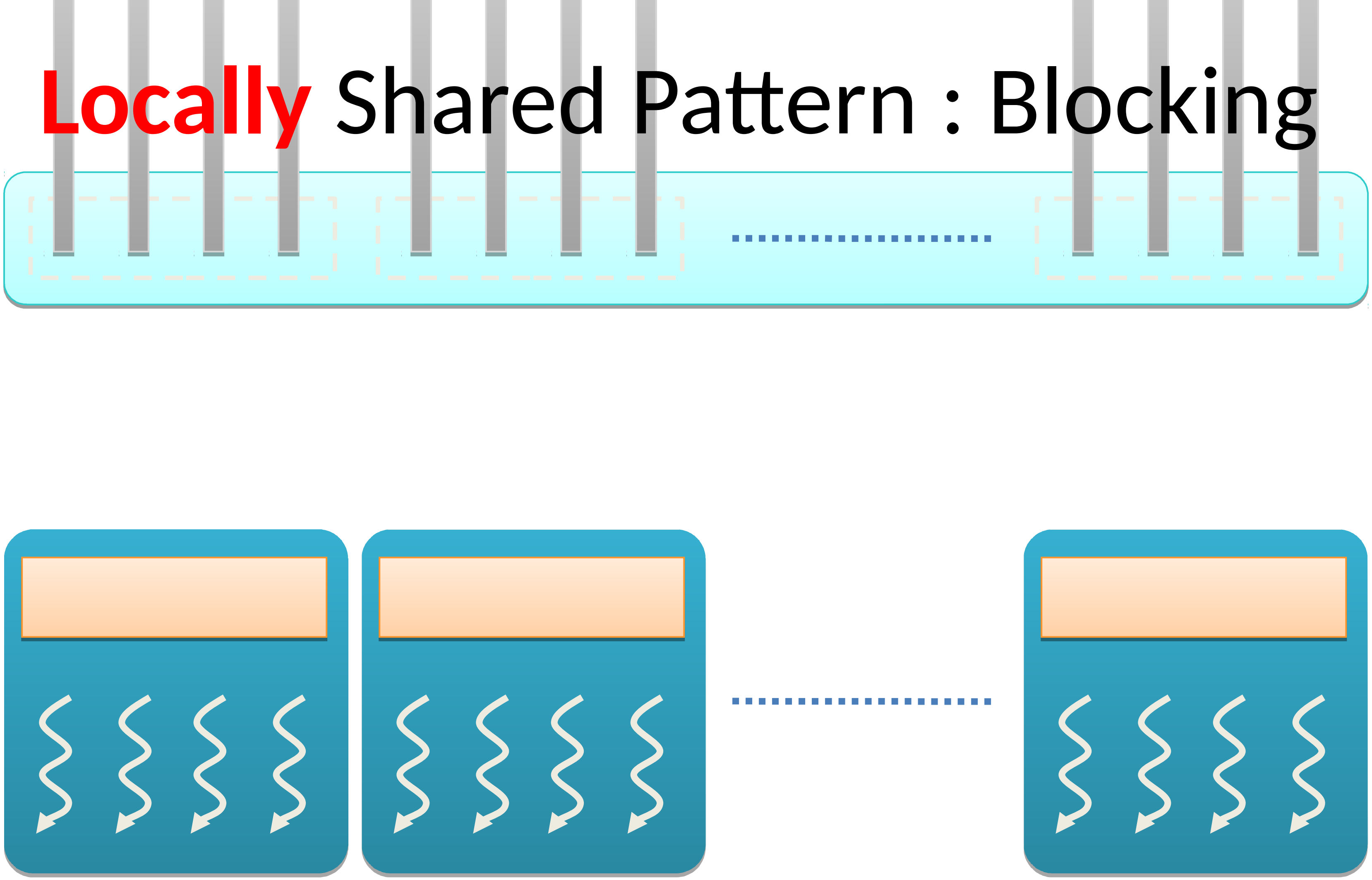
- Load the subset from global memory to shared memory, **using multiple threads to exploit memory-level parallelism**

# Locally Shared Pattern : Blocking



- Perform the computation on the subset from **shared memory**

# Locally Shared Pattern : Blocking



- Copy the result from **shared memory** back to global memory

# Locally Shared Pattern : Blocking

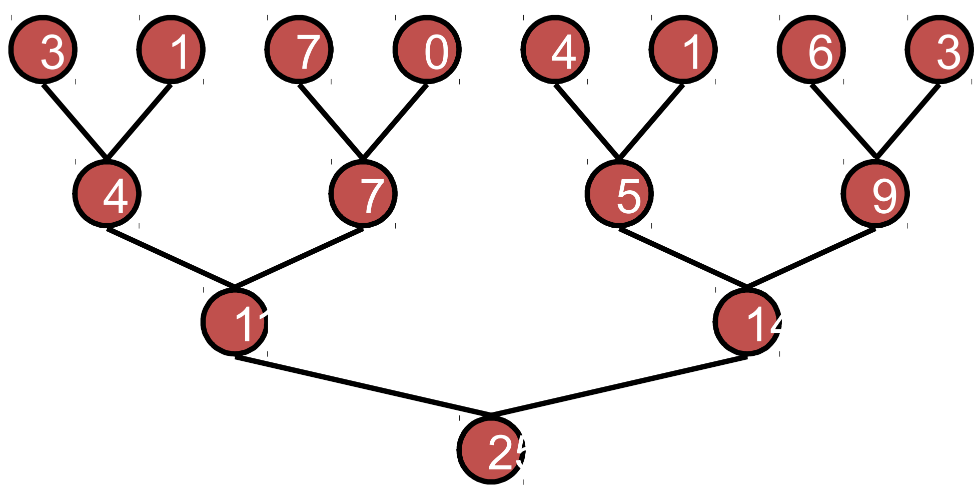
- All CUDA kernels are built this way
  - Blocking may not matter for a particular problem, but you're still forced to think about it
  - Not all kernels require `__shared__` memory
  - All kernels do require registers
- All of the parallel patterns we'll discuss have CUDA implementations that exploit blocking in some fashion

# Parallel Computing Scenarios: Reduction

# Reduction

- **Reduce** vector to a single value
  - Via an associative operator (+, *, min/max, AND/OR, …)
  - CPU: sequential implementation

    ```
    for(int i = 0, i < n, ++i) ...
    ```
  - GPU: “tree”-based implementation

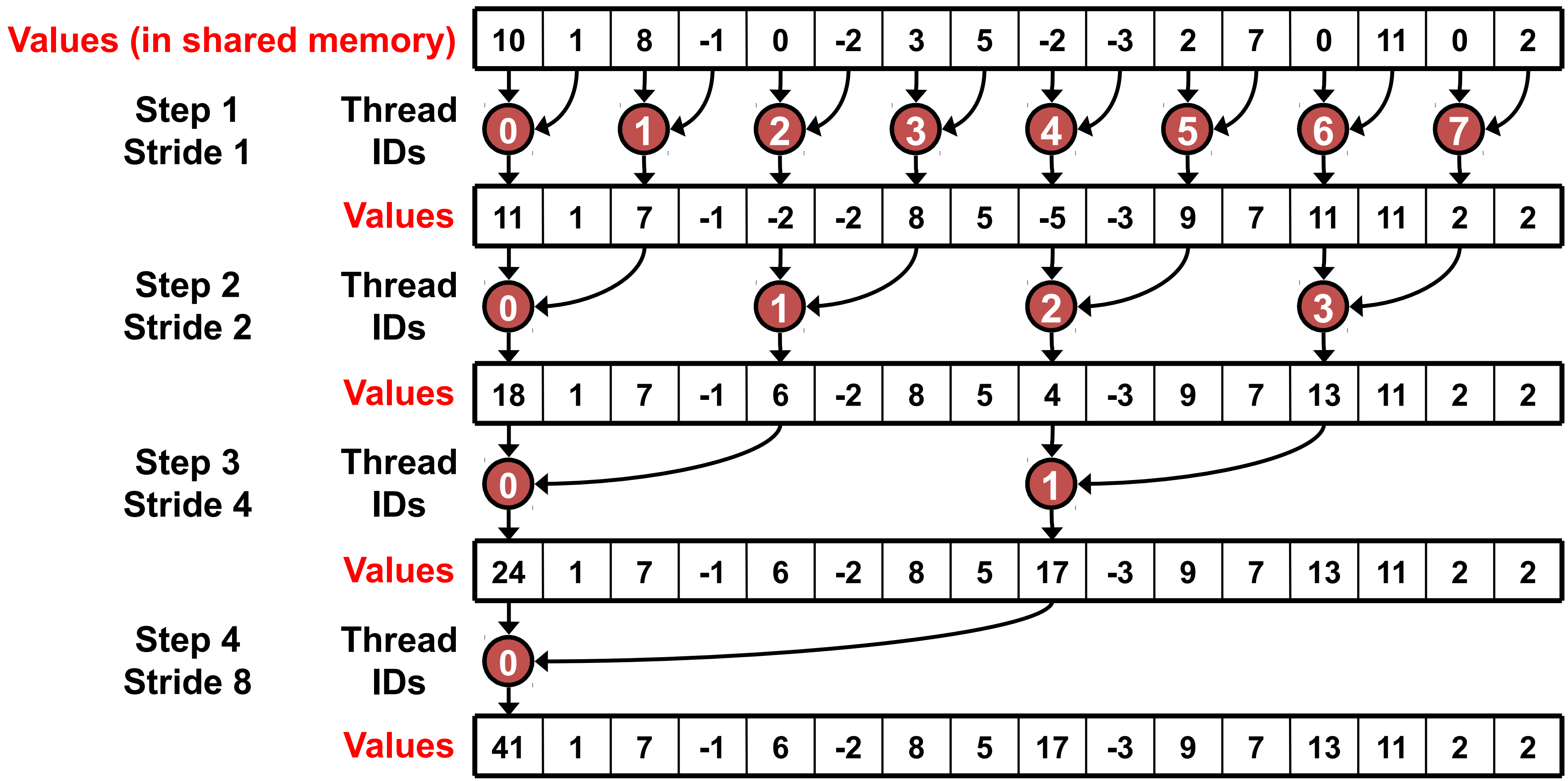


# Serial Reduction

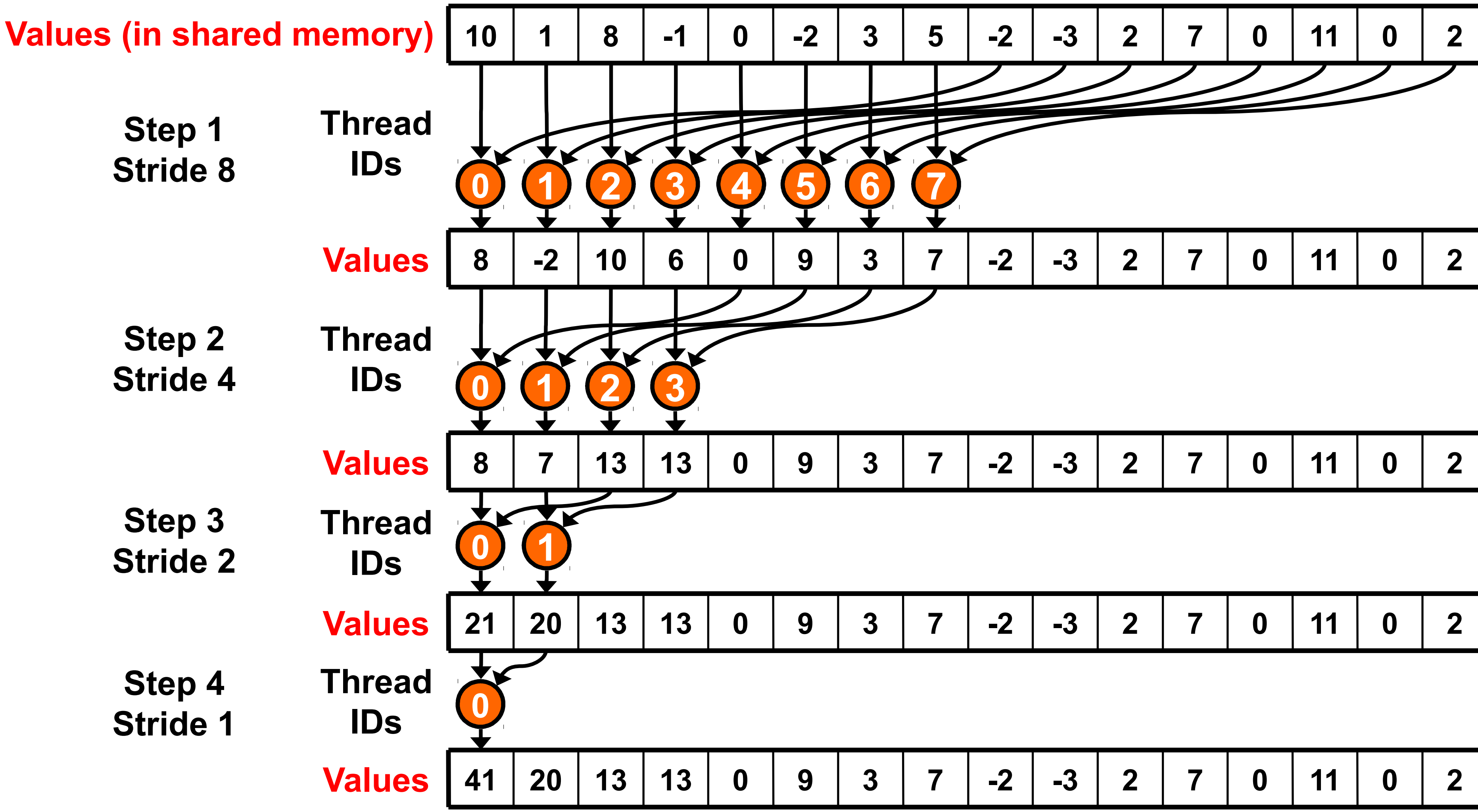
```
// reduction via serial iteration
float sum(float *data, int n)
{
  float result = 0;
  for(int i = 0; i < n; ++i)
  {
    result += data[i];
  }

  return result;
}
```

# Parallel Reduction – Interleaved



# Parallel Reduction – Contiguous



# CUDA Reduction

```
__global__ void block_sum(float *input,
                          float *results,
                          size_t n)
{
  extern __shared__ float sdata[];
  int i = ..., int tx = threadIdx.x;

  // load input into __shared__ memory
  float x = 0;
  if(i < n)
    x = input[i];
  sdata[tx] = x;
  __syncthreads();
```

# CUDA Reduction

```
// block-wide reduction in __shared__ mem
for(int offset = blockDim.x / 2;
    offset > 0;
    offset >>= 1)
{
  if(tx < offset)
  {
    // add a partial sum upstream to our own
    sdata[tx] += sdata[tx + offset];
  }
  __syncthreads();
}
```

# CUDA Reduction

```
  // finally, thread 0 writes the result
  if(threadIdx.x == 0)
  {
    // note that the result is per-block
    // not per-thread
    results[blockIdx.x] = sdata[0];
  }
}
```

# CUDA Reduction

```
// global sum via per-block reductions
float sum(float *d_input, size_t n)
{
  size_t block_size = ..., num_blocks = ...;

  // allocate per-block partial sums
  // plus a final total sum
  float *d_sums = 0;
  cudaMalloc((void**)&d_sums,
    sizeof(float) * (num_blocks + 1));
  ...
```

# CUDA Reduction

```
// reduce per-block partial sums
int smem_sz = block_size*sizeof(float);
block_sum<<<num_blocks,block_size,smem_sz>>>
  (d_input, d_sums, n);

// reduce partial sums to a total sum
block_sum<<<1,block_size,smem_sz>>>
  d_sums, d_sums + num_blocks, num_blocks);

// copy result to host
float result = 0;
cudaMemcpy(&result, d_sums+num_blocks, ...);
return result;

}
```

# Pitfalls!

- What happens if there are too many partial sums to fit into `__shared__` memory in the second stage?

- What happens if the temporary storage is too big?

- Give each thread more work in the first stage
  - Sum is associative & commutative
  - Order doesn't matter to the result
  - We can schedule the sum any way we want
    - serial accumulation before block-wide reduction

- Let's left these exercises for the self-guided work…

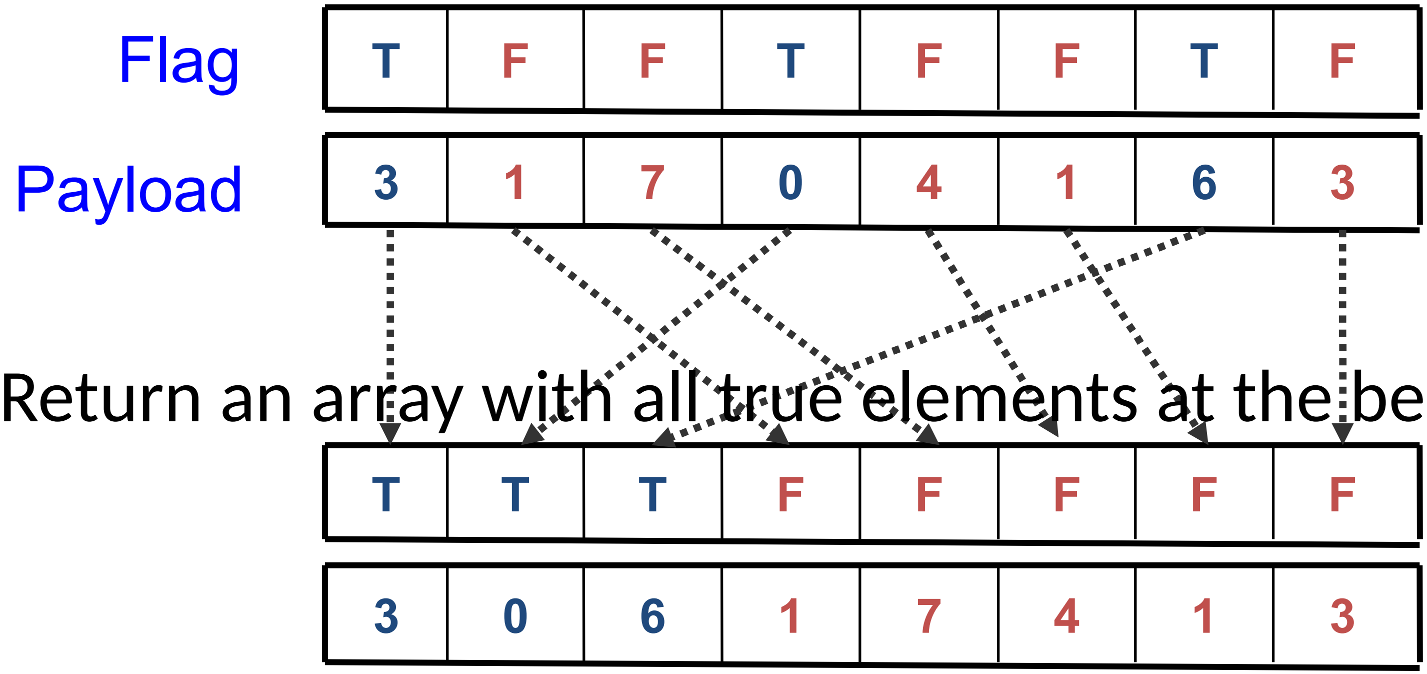
# Parallel Reduction Complexity

- Log($N$) parallel steps, each step $S$ does $N/2^S$ independent ops
  - Step Complexity is O(log $N$)
- For $N=2^D$, performs $\sum_{S\in[1..D]} 2^{D-S} = N-1$ operations
  - Work Complexity is O($N$) – It is work-efficient
  - i.e. does not perform more operations than a sequential algorithm
- With $P$ threads physically in parallel ($P$ processors), time complexity is O($N/P$ + log $N$)
  - Compare to O($N$) for sequential reduction

# Parallel Computing Scenarios:
# Split, Compact, Expand

# Split Operation

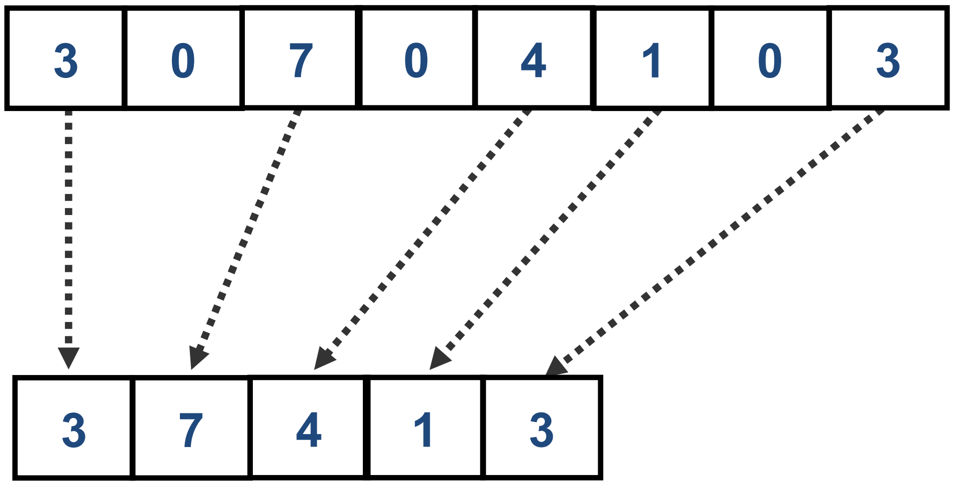
- Given:array of true and false elements (and payloads)




- Return an array with all true elements at the beginning

# Variable Output Per Thread: Compact

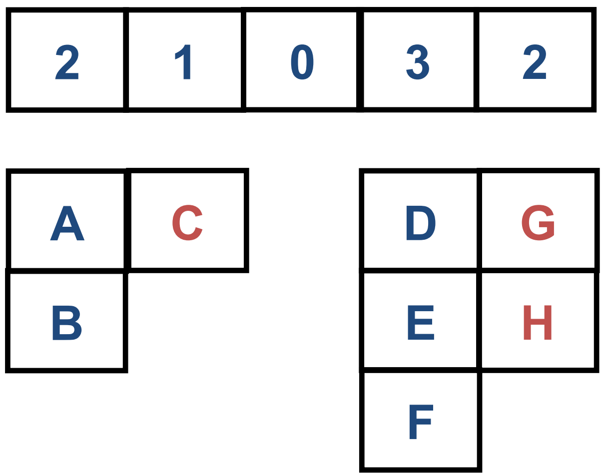
- Remove null elements




-

# Variable Output Per Thread: General Case

- Reserve Variable Storage Per Thread




-

# Split, Compact, Expand

- Each thread must answer a simple question:

  "Where do I write my output?"

- The answer depends on what other threads write!

- *Scan* provides an efficient parallel answer

# Parallel Computing Scenarios: Algorithm Example -> Scan

# Scan (or Parallel Prefix Sum)

- Given an array $A = [a_0, a_1, \ldots, a_{n-1}]$
  and a binary associative operator $\oplus$ with identity $I$,

$$\text{scan}(A) = [I, a_0, (a_0 \oplus a_1), \ldots, (a_0 \oplus a_1 \oplus \ldots \oplus a_{n-2})]$$

- Prefix sum: if $\oplus$ is addition, then scan on the series

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 3 | 1 | 7 | 0 | 4 | 1 | 6 | 3 |

returns the series

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0 | 3 | 4 | 11 | 11 | 15 | 16 | 22 |

# Applications of Scan

- Scan is a simple and useful parallel building block for many parallel algorithms:

  - **Radix sort**
  - **Quicksort (seg. scan)**
  - **String comparison**
  - **Lexical analysis**
  - **Stream compaction**
  - **Run-length encoding**
  - **Polynomial evaluation**
  - **Solving recurrences**
  - **Tree operations**
  - **Histograms**
  - **Allocation**
  - **Etc.**

- Fascinating, since scan is unnecessary in sequential computing!

# Serial Scan

```
int input[8] = {3, 1, 7, 0, 4, 1, 6, 3};
int result[8];
int running_sum = 0;
for(int i = 0; i < 8; ++i)
{
  result[i] = running_sum;
  running_sum += input[i];
}

// result = {0, 3, 4, 11, 11, 15, 16, 22}
```

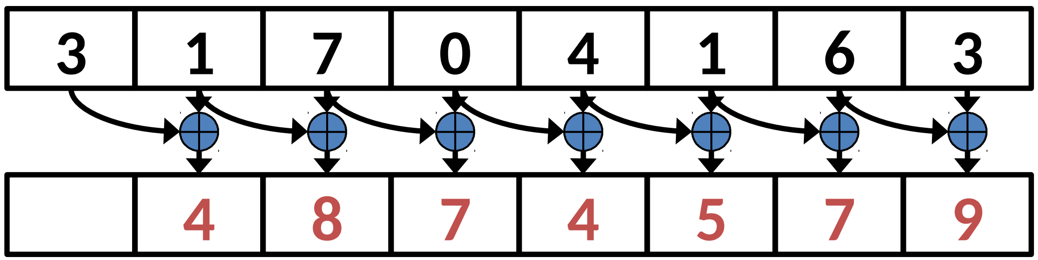
# A Scan Algorithm – Preview

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| **3** | **1** | **7** | **0** | **4** | **1** | **6** | **3** |

Assume array is already in shared memory

See Harris, M., S. Sengupta, and J.D. Owens. “Parallel Prefix Sum (Scan) in CUDA”, *GPU Gems 3*

# A Scan Algorithm – Preview

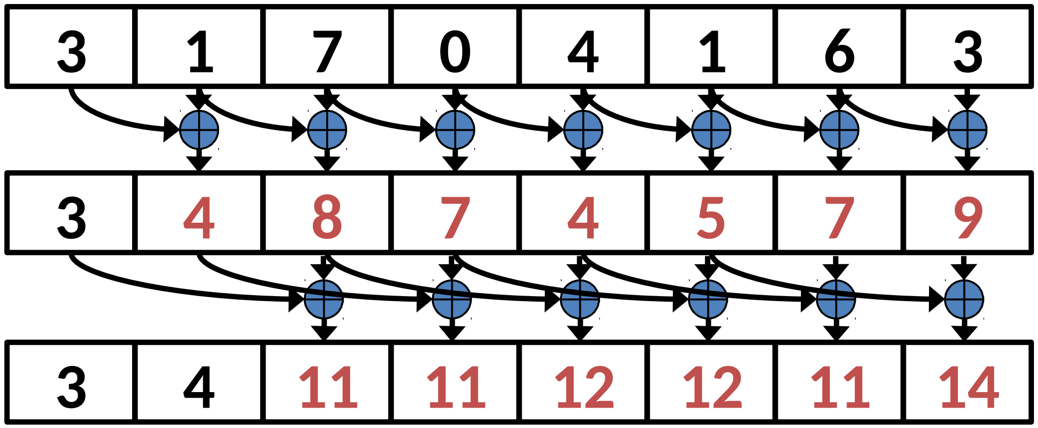



**Iteration 0, *n-1* threads**

Each $\oplus$ corresponds to a single thread.

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

# A Scan Algorithm – Preview

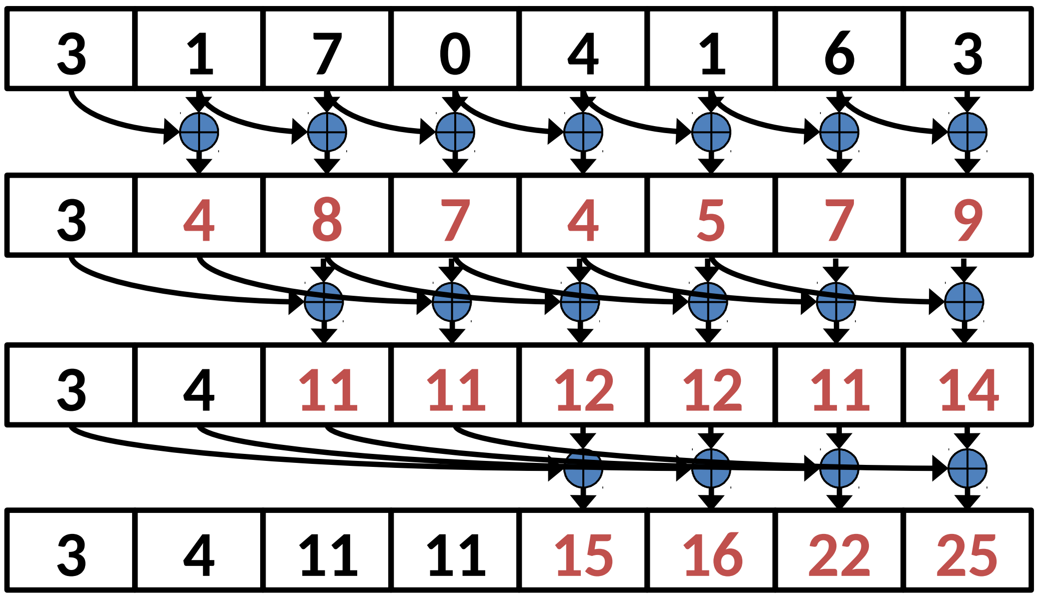



**Iteration 1, *n-2* threads**

Each ⊕ corresponds to a single thread.

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

# A Scan Algorithm – Preview




**Iteration *i*, *n*-2*^i* threads**

Each ⊕ corresponds to a single thread.

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

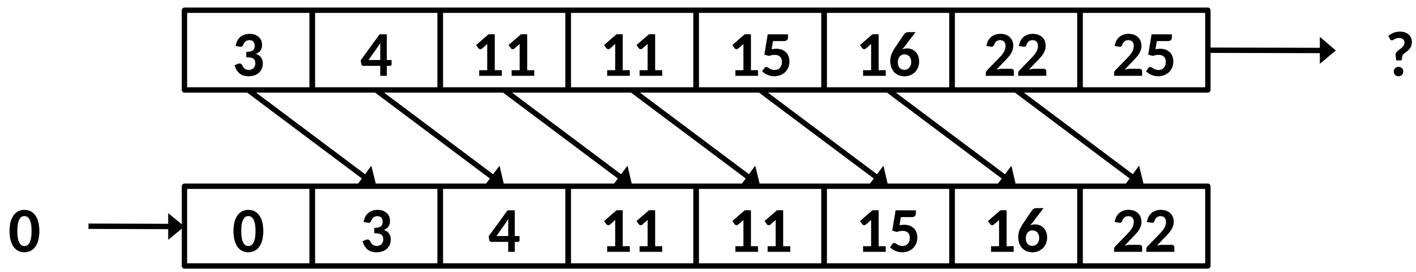
Note that this algorithm operates in-place: no need for double buffering

# A Scan Algorithm – Preview

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| **3** | **4** | **11** | **11** | **15** | **16** | **22** | **25** |

- We have an inclusive scan result

# A Scan Algorithm – Preview




- For an exclusive scan, right-shift through `__shared__` memory
- Note that the unused final element is also the sum of the entire array
  - Often called the "carry"
  - Scan & reduce in one pass

# CUDA Block-wise Inclusive Scan

```
__global__ void inclusive_scan(int *data)
{
  extern __shared__ int sdata[];

  unsigned int i = ...

  // load input into __shared__ memory
  int sum = input[i];
  sdata[threadIdx.x] = sum;
  __syncthreads();
  ...
```

# CUDA Block-wise Inclusive Scan

```
for(int o = 1; o < blockDim.x; o <<= 1)
{
  if(threadIdx.x >= o)
    sum += sdata[threadIdx.x - o];

  // wait on reads
  __syncthreads();

  // write my partial sum
  sdata[threadIdx.x] = sum;

  // wait on writes
  __syncthreads();
}
```

# CUDA Block-wise Inclusive Scan

```
  // we're done!
  // each thread writes out its result
  result[i] = sdata[threadIdx.x];
}
```

# Results are Local to Each Block

```
Block 0

Input:
 5  5  4  4  5  4  0  0  4  2  5  5  1  3  1  5
Result:
 5 10 14 18 23 27 27 27 31 33 38 43 44 47 48 53


Block 1

Input:
 1  2  3  0  3  0  2  3  4  4  3  2  2  5  5  0
Result:
 1  3  6  6  9  9 11 14 18 22 25 27 29 34 39 39
```

# Results are Local to Each Block

- Need to propagate results from each block to all subsequent blocks

- 2-phase scan
  1. Per-block scan & reduce
  2. Scan per-block sums

- Final update propagates phase 2 data and transforms to exclusive scan result

# Resume

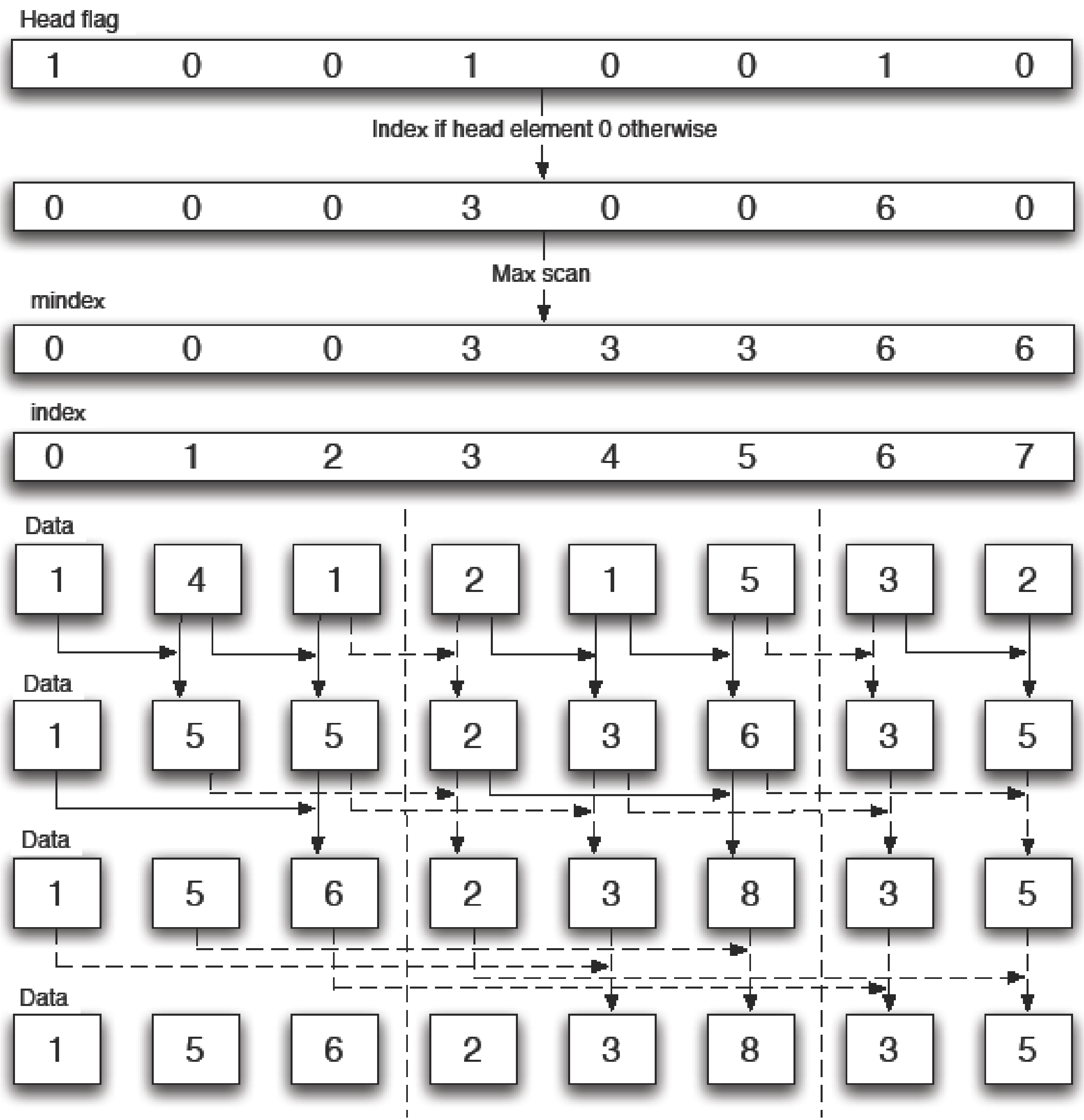
- Patterns like reduce, split, compact, scan, and others let us reason about data parallel problems abstractly
- Higher level patterns are built from more fundamental patterns
- Scan in particular is fundamental to parallel processing, but unnecessary in a serial world
- Get others to implement these for you!

# Parallel Computing Scenarios: Algorithm Example -> Segmented Scan

# Segmented Scan

- What it is:
  - Scan + Barriers/Flags associated with certain positions in the input arrays
  - Operations don't propagate beyond barriers

- Do many scans at once, no matter their size


Head flag
1 0 0 1 0 0 1 0
Index if head element 0 otherwise
0 0 0 3 0 0 6 0
Max scan
mindex
0 0 0 3 3 3 6 6
index
0 1 2 3 4 5 6 7
Data
1 4 1 2 1 5 3 2
Data
1 5 5 2 3 6 3 5
Data
1 5 6 2 3 8 3 5
Data
1 5 6 2 3 8 3 5

**(C) Sengupta**

# Segmented Scan

```
__global__ void segscan(int * data, int * flags)
{
    __shared__ int s_data[BL_SIZE];
    __shared__ int s_flags[BL_SIZE];
    int idx = threadIdx.x + blockDim.x * blockIdx.x;
   // copy block of data into shared
   // memory
   s_data[idx] = …; s_flags[idx] = …;
   __syncthreads();
```

# Segmented Scan

```
  …
  // choose whether to propagate
  s_data[idx] = s_flags[idx] ?
  s_data[idx] :                         s_data[idx -
  1] + s_data[idx];

  // create merged flag
  s_flags[idx] =
     s_flags[idx - 1] | s_flags[idx];
  // repeat for different strides
}
```

# Segmented Scan

- Doing lots of reductions of unpredictable size at the same time is the most common use
- Think of doing sums/max/count/any over arbitrary sub-domains of your data

# Segmented Scan

- Common Usage Scenarios:
  - Determine which region/tree/group/object class an element belongs to and assign that as its new ID
  - Sort based on that ID
  - Operate on all of the regions/trees/groups/objects in parallel, no matter what their size or number

# Segmented Scan

- Also useful for implementing divide-and-conquer type algorithms
  - Quicksort and similar algorithms

# Parallel Computing Scenarios: Algorithm Example -> Sort

# Sort

- Useful for almost everything
- Optimized versions for the GPU already exist
- Sorted lists can be processed by segmented scan
- Sort data to restore memory and execution coherence

# Sort

- binning and sorting can often be used interchangeably

- Sort is standard, but can be suboptimal

- Binning is usually custom, has to be optimized, can be faster

# Sort

- Radixsort is faster than comparison-based sorts

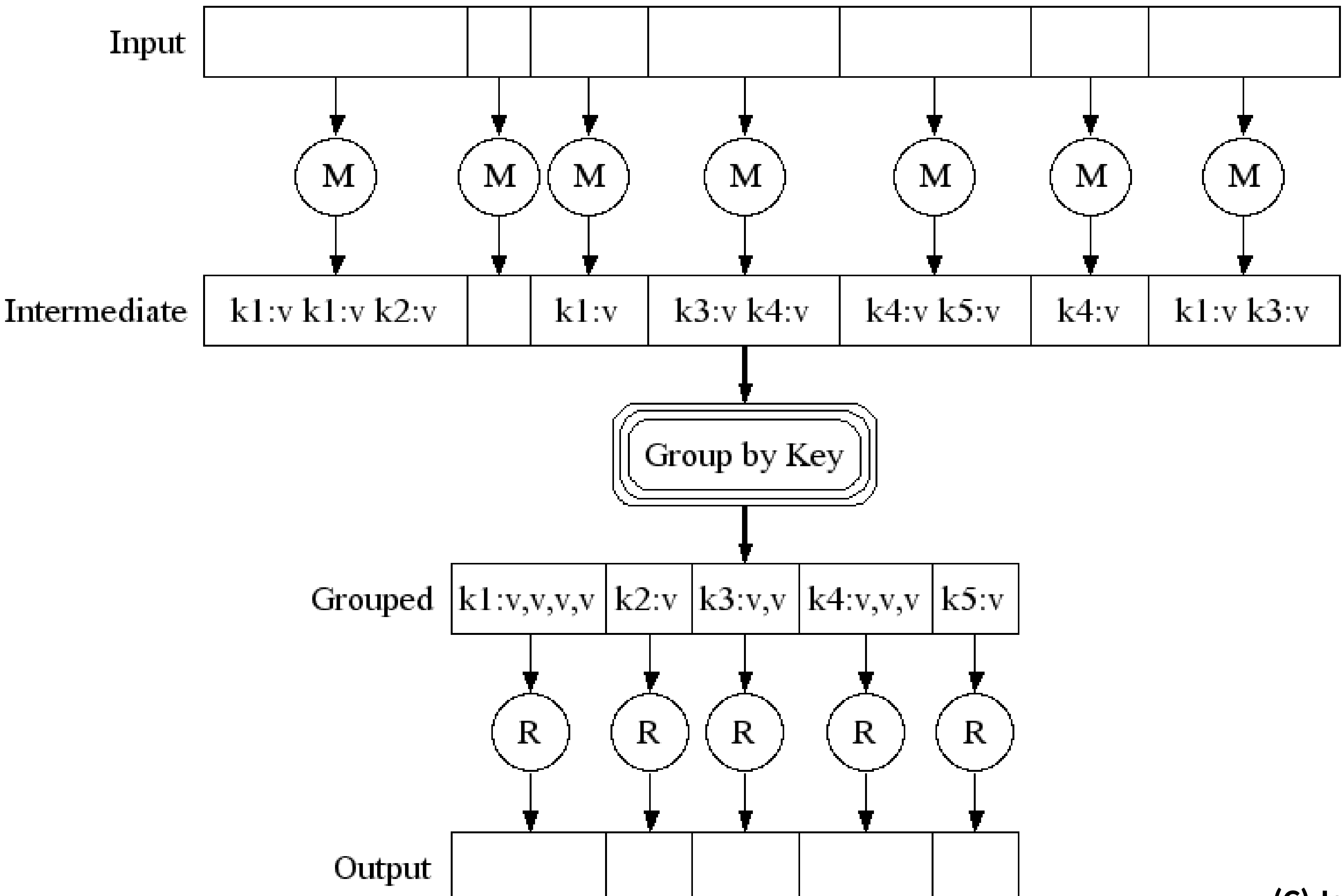
- If you can generate a fixed-size key for the attribute you want to sort on, you get better performance

# Parallel Computing Scenarios: Algorithm Example -> Map/Reduce

# Map/Reduce

- Old concept from functional progamming
- Repopularized by Google as parallel computing pattern
- Combination of sort and reduction (scan)

# Map/Reduce




(C) Jeff Dean

# Map/Reduce: Map

- Map a function over a domain
- Function is provided by the user
- Function can be anything which produces a (key, value) pair
  - Value can just be a pointer to arbitrary datastructure

# Map/Reduce: Sort

- All the (key,value) pairs are sorted based on their keys
- Happens implicitly
- Creates runs of (k,v) pairs with same key
- User usually has no control over sort function

# Map/Reduce: Reduce

- Reduce function is provided by the user
  - Can be simple **plus, product, max**,…
- Library makes sure that values from one key don't propagate to another (segscan)
- Final result is a list of keys and final values (or arbitrary datastructures)

# Parallel Computing Scenarios:
# Algorithm Example ->
# Kernel Fusion

# Kernel Fusion

- Combine kernels with simple **producer->consumer** dataflow

- Combine **generic data** movement kernel with **specific operator** function

- Save memory bandwidth **by not writing out** intermediate results **to global memory**

# Separate Kernels

```
__global__ void is_even(int * in, int * out)
{
    int i = …
   out[i] = ((in[i] % 2) == 0) ? 1: 0;
}

// separate scan-function
__global__ void scan(…)
{
   …
}
```

# Fused Kernel

```
__global__ void fused_even_scan(int * in, int * out, …)
{
  int i = …
  int flag = ((in[i] % 2) == 0) ? 1: 0;
  // your scan code here, using the flag directly
}
```

# Kernel Fusion

- Best when the pattern looks like

```
output[i] = g(f(input[i]));
```

- Any simple one-to-one mapping will work

# Fused Kernel

```
template <class F>
__global__ void opt_stencil(float * in, float * out, F f)
{ // your 2D stencil code here
   for(i,j)
   {
      partial = f(partial,in[…],i,j);
   }
   float result = partial;
}
```

# Fused Kernel

```
class boxfilter
{ private:
   table[3][3];
   boxfilter(float input[3][3])
   public:
   float operator()(float a, float b, int i, int j)
   {
      return a + b*table[i][j];
   }
}
```

# Fused Kernel

```
class maxfilter
{  public:
   float operator()(float a, float b, int i, int j)
   {
      return max(a,b);
   }
}
```