Parallel Algorithms

Chris Rossbach and Calvin Lin

cs380p

Outline

Over the next few classes:

Background from many areas

- Architecture
  - Vector processors
  - Hardware multi-threading
- Graphics
  - Graphics pipeline
  - Graphics programming models
- Algorithms
  - parallel architectures → parallel algorithms

Programming GPUs

- CUDA
- Basics: getting something working
- Advanced: making it perform

Outline

Over the next few classes:

Background from many areas

- Architecture
  - Vector processors
  - Hardware multi-threading
- Graphics
  - Graphics pipeline
  - Graphics programming models

This lecture

Programming GPUs

- CUDA
- Basics: getting something working
- Advanced: making it perform

Review


PCI Express 3.0 Host Interface
GigaThread Engine
GPC
TPC
SM
Memory Controller
HBM2
L2 Cache
GAMERS NEXUS
High-Speed Hub
NVLink

Review

Each SM has multiple vector units (4)

32 lanes wide → warp size

Review

Each SM has multiple vector units (4)

32 lanes wide → warp size

Vector units use ***hardware multi-threading***

Review

Each SM has multiple vector units (4)

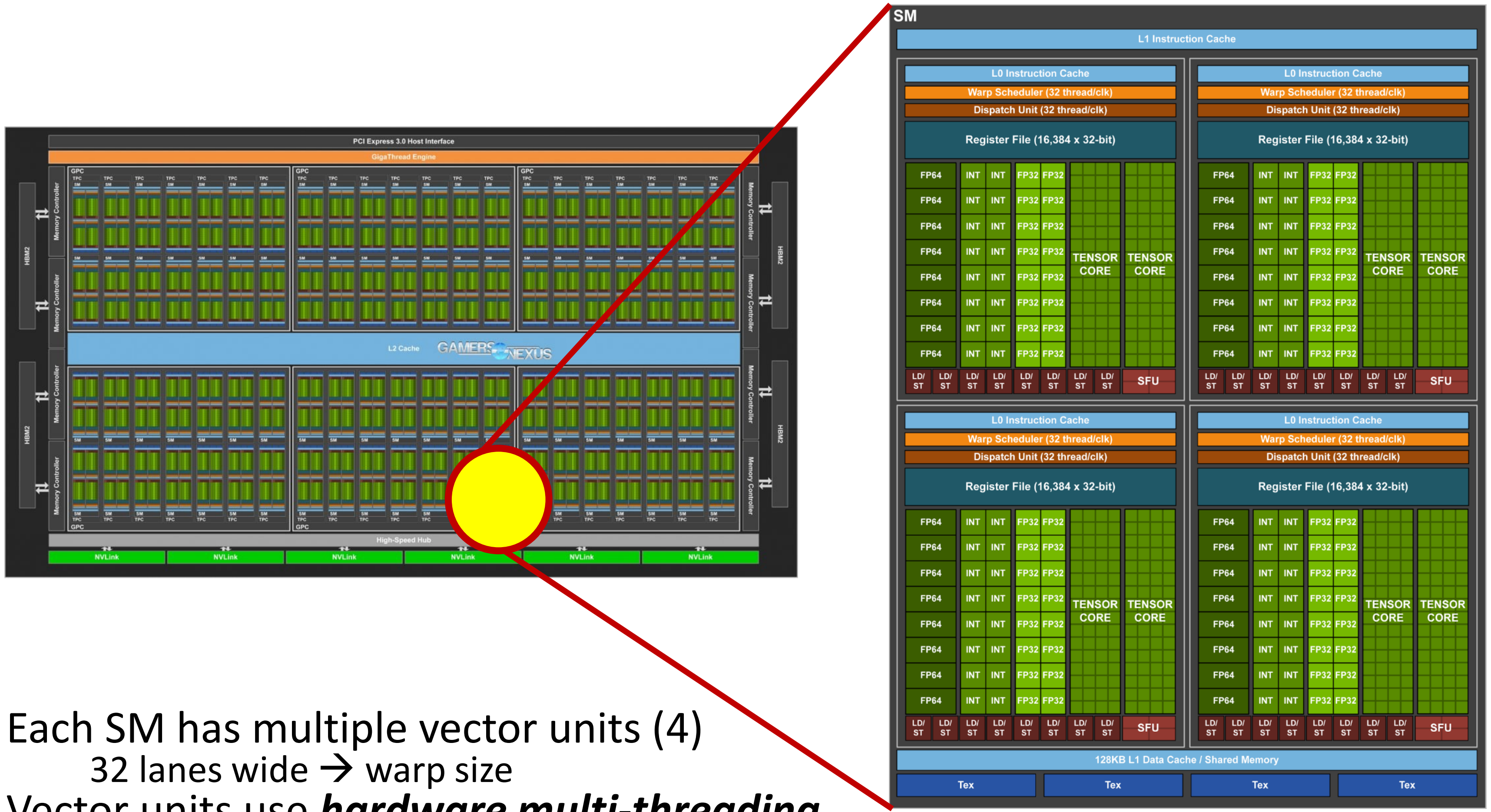
    32 lanes wide → warp size

Vector units use ***hardware multi-threading***

Execution → a grid of thread blocks (TBs)

    Each TB has some number of threads

Review



Each SM has multiple vector units (4)

    32 lanes wide → warp size

Vector units use ***hardware multi-threading***

Execution → a grid of thread blocks (TBs)

    Each TB has some number of threads

Review

Thread block scheduler

Each SM has multiple vector units (4)

32 lanes wide → warp size

Vector units use ***hardware multi-threading***

Execution → a grid of thread blocks (TBs)

Each TB has some number of threads

Review

Thread block scheduler

warp (thread) scheduler

Each SM has multiple vector units (4)

- 32 lanes wide → warp size

Vector units use ***hardware multi-threading***

Execution → a grid of thread blocks (TBs)

- Each TB has some number of threads

Programming Model

“kernels” == “shader programs”

1000s of HW-scheduled threads per kernel

Threads grouped into independent blocks.

- Threads in a block can synchronize (barrier)
- This is the *only* synchronization

“Grid” == “launch” == “invocation” of a kernel

- a group of blocks (or warps)

Programming Model

“kernels” == “shader programs”

1000s of HW-scheduled threads per kernel

Threads grouped into independent blocks.

- Threads in a block can synchronize (barrier)
- This is the *only* synchronization

“Grid” == “launch” == “invocation” of a kernel

- a group of blocks (or warps)

Need codes that are 1000s-X parallel….

Parallel Algorithms

Sequential algorithms often do not permit easy parallelization

- Does not mean there work has no parallelism
- A different approach can yield parallelism
- but often changes the algorithm
- Parallelizing != just adding locks to a sequential algorithm

Parallel Algorithms

Sequential algorithms often do not permit easy parallelization

- Does not mean there work has no parallelism
- A different approach can yield parallelism
- but often changes the algorithm
- Parallelizing != just adding locks to a sequential algorithm

If you can express your algorithm using these patterns, an apparently fundamentally sequential algorithm can be made parallel

Parallel Algorithms

Parallel Algorithms

Key idea:

Parallel Algorithms

Key idea:

Express sequential algorithms as combinations of parallel patterns

Parallel Algorithms

Key idea:

Express sequential algorithms as combinations of parallel patterns

Examples:

Parallel Algorithms

Key idea:

Express sequential algorithms as combinations of parallel patterns

Examples:

Map

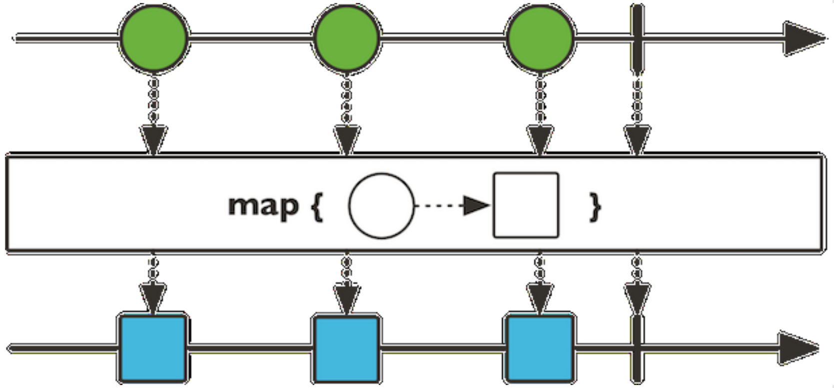
Parallel Algorithms

Key idea:

Express sequential algorithms as combinations of parallel patterns

Examples:

Map



Parallel Algorithms

Key idea:

Express sequential algorithms as combinations of parallel patterns

Examples:

Map

Parallel Algorithms

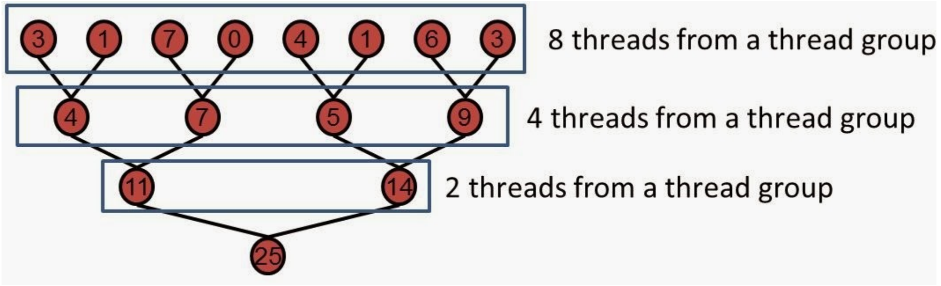
Key idea:

Express sequential algorithms as combinations of parallel patterns

Examples:

Map

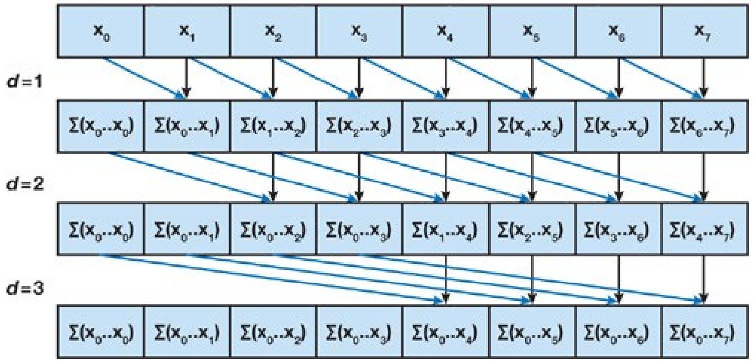
Reductions



Parallel Algorithms

Key idea:
Express sequential algorithms as combinations of parallel patterns

Examples:
Map
Reductions
Scans



Parallel Algorithms

Key idea:

Express sequential algorithms as combinations of parallel patterns

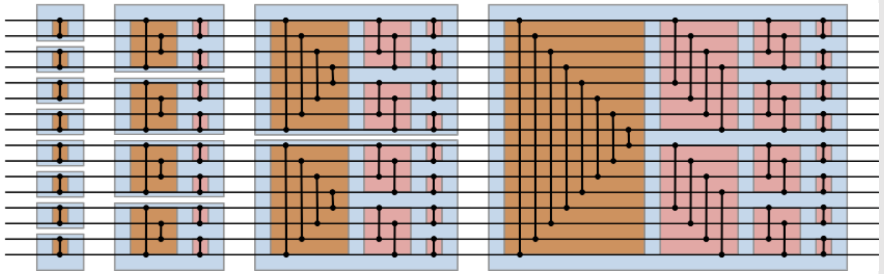
Examples:

Map

Reductions

Scans

Re-orderings (scatter/gather/sort)



Map

Inputs

- Array A
- Function f(x)

map(A, f) → apply f(x) on all elements in A

Parallelism trivially exposed

- f(x) can be applied in parallel to all elements, in principle

Map

Inputs

- Array A
- Function f(x)

map(A, f) → apply f(x) on all elements in A

Parallelism trivially exposed

- f(x) can be applied in parallel to all elements, in principle

```
for(i=0; i<numPoints; i++) {
   labels[i] = findNearestCenter(points[i]);
}
```

```
map(points, findNearestCenter)
```

Scatter and Gather

Scatter and Gather

Gather:

Read multiple items to single /packed location

Scatter and Gather

Gather:

Read multiple items to single /packed location

Scatter:

Write single/packed data item to multiple locations

Scatter and Gather

Gather:

Read multiple items to single /packed location

Scatter:

Write single/packed data item to multiple locations

Inputs: x, y, indices, N

Scatter and Gather

Gather:

Read multiple items to single /packed location

Scatter:

Write single/packed data item to multiple locations

Inputs: x, y, indices, N

```
for (i=0; i<N; ++i)
  x[i] = y[idx[i]];
```

```
gather(x, y, idx)
```

```
for (i=0; i<N; ++i)
  y[idx[i]] = x[i];
```

```
scatter(x, y, idx)
```

Scatter and Gather

Gather:

Read multiple items to single /packed location

Scatter:

Write single/packed data item to multiple locations

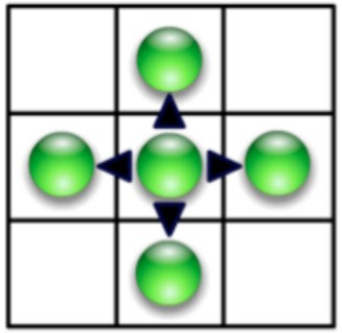
Inputs: x, y, indices, N

```
for (i=0; i<N; ++i)
  x[i] = y[idx[i]];
```

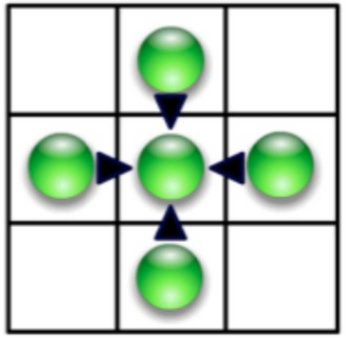
```
gather(x, y, idx)
```

```
for (i=0; i<N; ++i)
  y[idx[i]] = x[i];
```

```
scatter(x, y, idx)
```



Scatter



Gather

Reduce

Input

- Associative operator op
- Ordered set s = [a, b, c, … z]

Reduce(op, s) returns

a op b op c … op z

Reduce

Input

- Associative operator op
- Ordered set s = [a, b, c, … z]

Reduce(op, s) returns

a op b op c … op z

```
for(i=0; i<N; ++i) {
   accum += (point[i]*point[i])
}
```

```
accum = reduce(*, point)
```

Reduce

Input

Associative operator op

Ordered set s = [a, b, c, … z]

Reduce(op, s) returns

a op b op c … op z

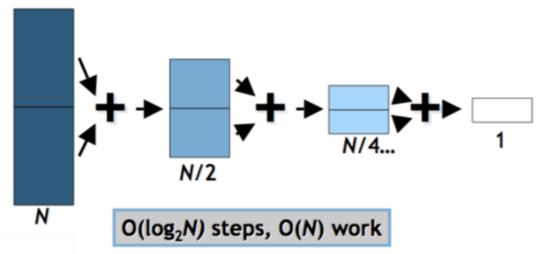
```
for(i=0; i<N; ++i) {
   accum += (point[i]*point[i])
}
```

→

```
accum = reduce(*, point)
```

Why must op be associative?

Reduce



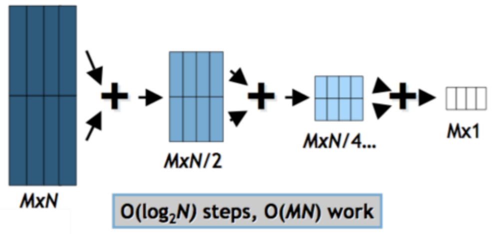
Input

Associative operator op

Ordered set s = [a, b, c, … z]

Reduce(op, s) returns

a op b op c … op z




```
for(i=0; i<N; ++i) {
    accum += (point[i]*point[i])
}
```

```
accum = reduce(*, point)
```

Why must op be associative?

Scan (Prefix Sum)

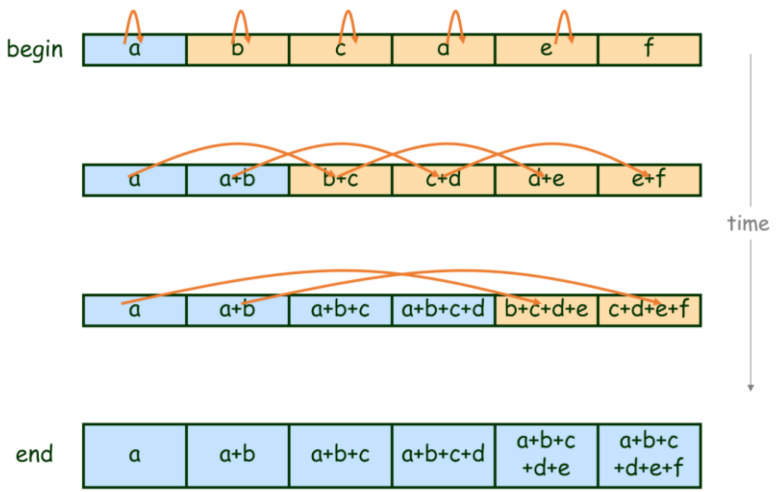
Input

- Associative operator op
- Ordered set s = [a, b, c, … z]
- Identity I

scan(op, s) = [I, a, (a op b), (a op b op c) …]

Scan is the workhorse of parallel algorithms:

- Sort, histograms, sparse matrix, string compare, …



Example: Parallel GroupBy

Group a collection by key

Lambda function maps elements → key

Example: Parallel GroupBy

Group a collection by key

Lambda function maps elements → key

```
var res = ints.GroupBy(x => x);
```

Example: Parallel GroupBy

Group a collection by key

Lambda function maps elements → key

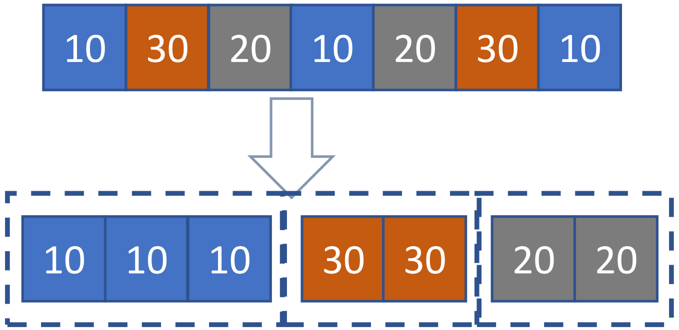
```
var res = ints.GroupBy(x => x);
```

Example: Parallel GroupBy

Group a collection by key

Lambda function maps elements → key

```
var res = ints.GroupBy(x => x);
```

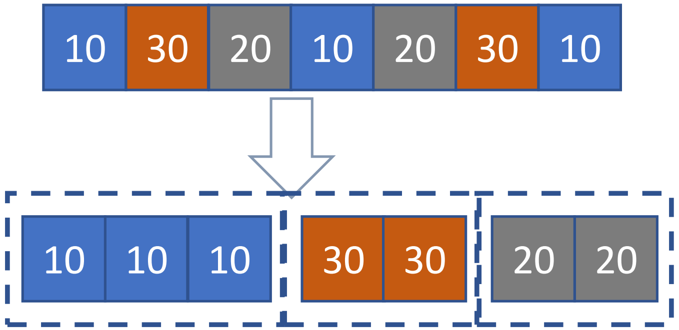


Example: Parallel GroupBy

Group a collection by key

Lambda function maps elements → key

```
var res = ints.GroupBy(x => x);
```



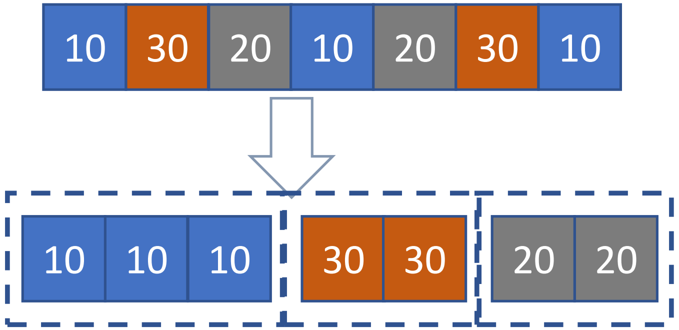

```
foreach(T elem in ints)
{
   key   = KeyLambda(elem);

   group = GetGroup(key);

   group.Add(elem);
}
```

Example: Parallel GroupBy

Group a collection by key

Lambda function maps elements → key

```
var res = ints.GroupBy(x => x);
```




```
foreach(T elem in PF(ints))
{
   key   = KeyLambda(elem);

   group = GetGroup(key)

   group.Add(elem);
}
```

Parallel GroupBy

ints	10	30	20	10	20	30	10
res	10	10	10	30	30	20	20

Parallel GroupBy

Process each input element in parallel

grouping ~ shuffling

input item → output offset such that groups are contiguous

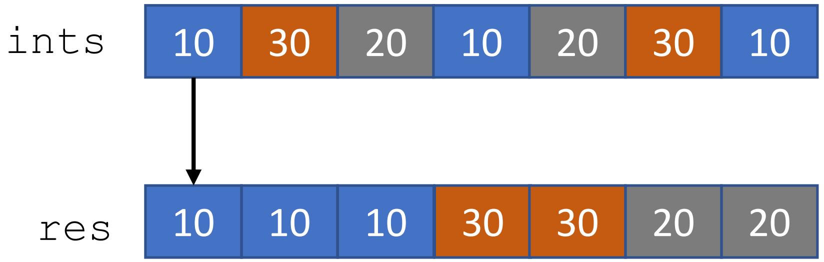
`ints`	10	30	20	10	20	30	10
`res`	10	10	10	30	30	20	20

Parallel GroupBy

Process each input element in parallel

grouping ~ shuffling

input item → output offset such that groups are contiguous

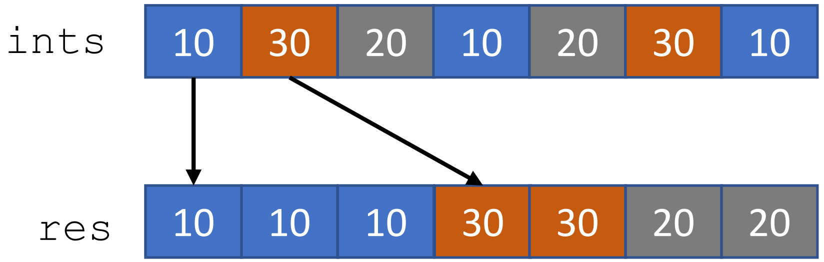


Parallel GroupBy

Process each input element in parallel

grouping ~ shuffling

input item → output offset such that groups are contiguous

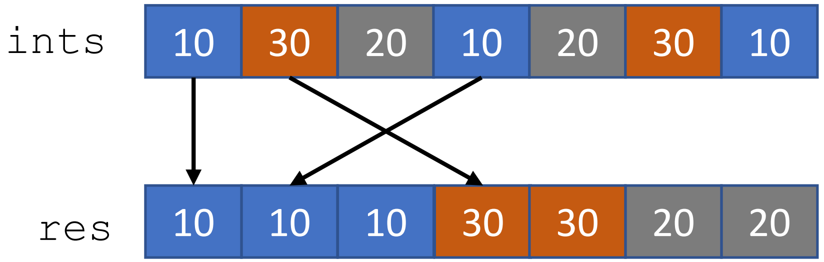


Parallel GroupBy

Process each input element in parallel

grouping ~ shuffling

input item → output offset such that groups are contiguous

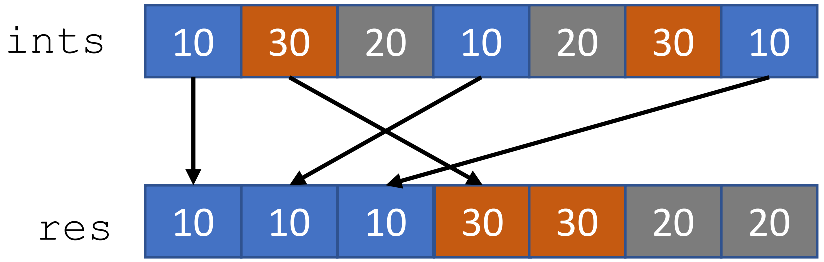


Parallel GroupBy

Process each input element in parallel

grouping ~ shuffling

input item → output offset such that groups are contiguous

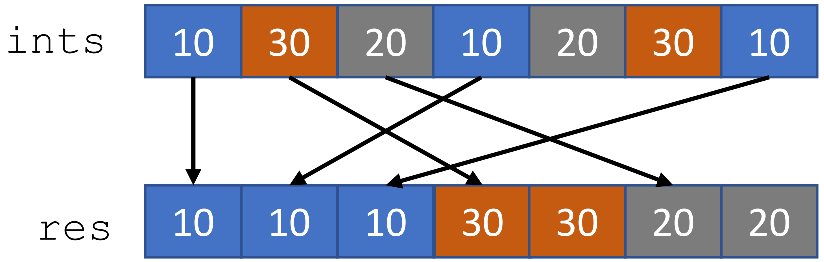


Parallel GroupBy

Process each input element in parallel

grouping ~ shuffling

input item → output offset such that groups are contiguous

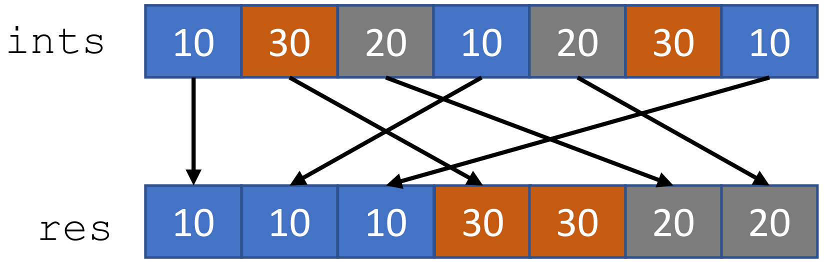


Parallel GroupBy

Process each input element in parallel

grouping ~ shuffling

input item → output offset such that groups are contiguous

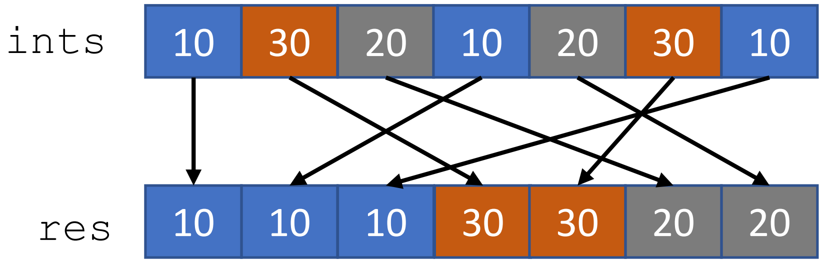


Parallel GroupBy

Process each input element in parallel

grouping ~ shuffling

input item → output offset such that groups are contiguous

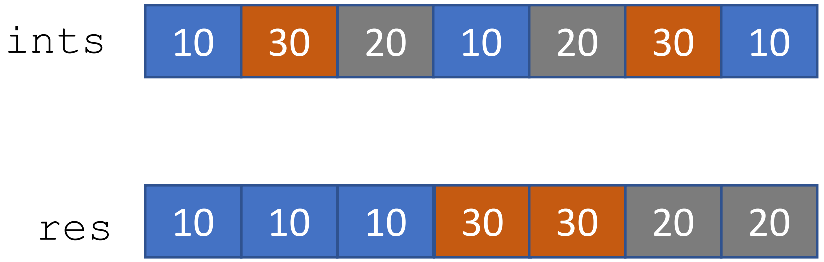


Parallel GroupBy

Process each input element in parallel

grouping ~ shuffling

input item → output offset such that groups are contiguous

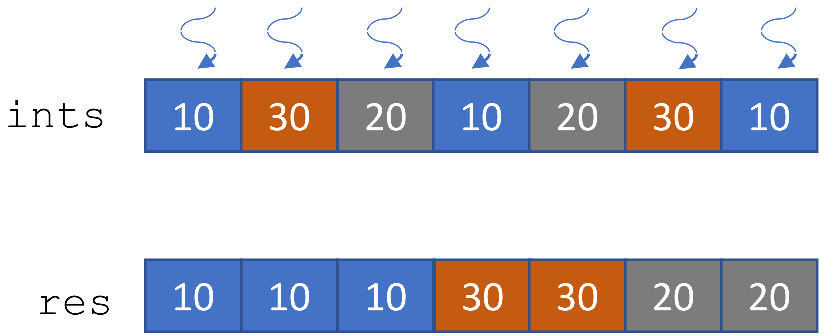


Parallel GroupBy

Process each input element in parallel

grouping ~ shuffling

input item → output offset such that groups are contiguous

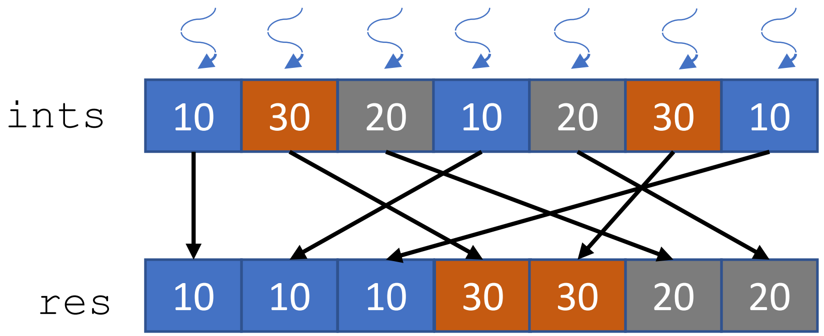


Parallel GroupBy

Process each input element in parallel

grouping ~ shuffling

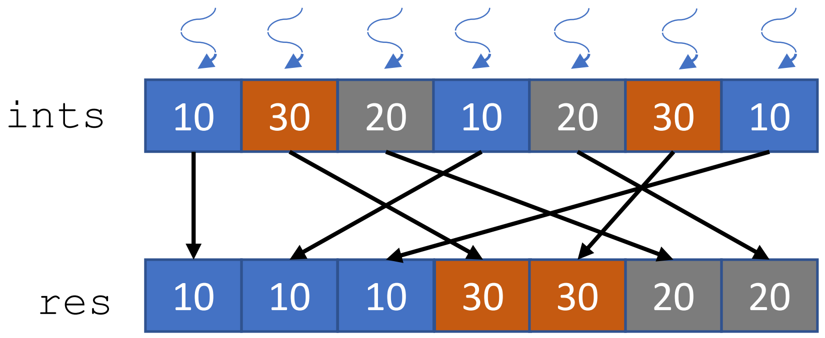
input item → output offset such that groups are contiguous



Parallel GroupBy

Process each input element in parallel

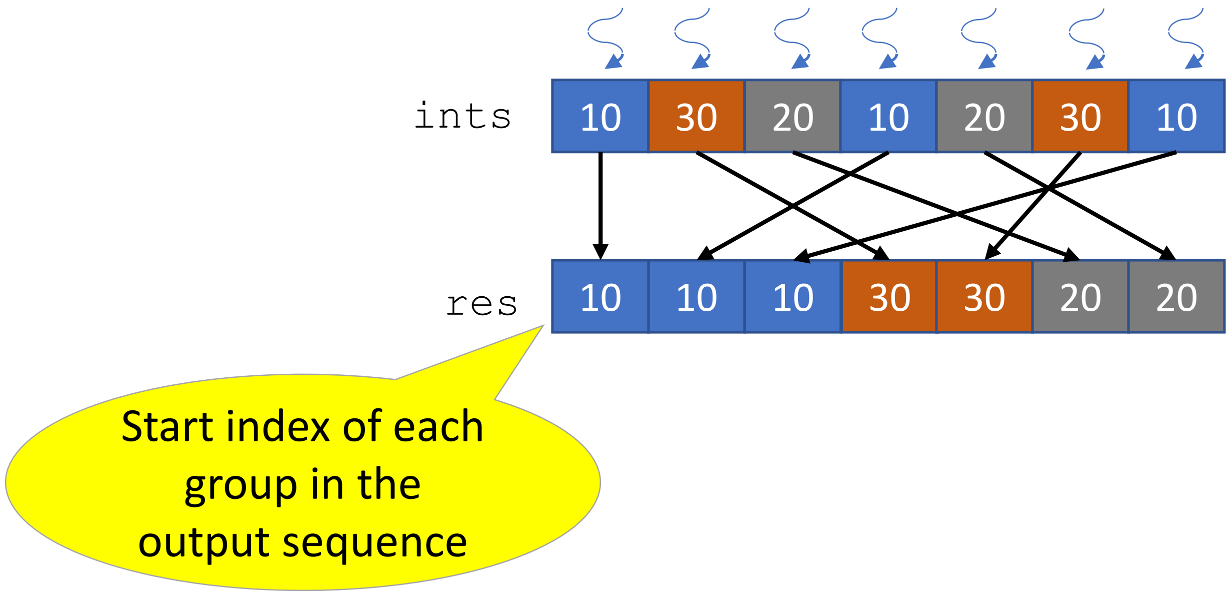
grouping ~ shuffling
input item → output offset such that groups are contiguous
output offset = group offset + item number
… but how to get the group offset, item number?



Parallel GroupBy

Process each input element in parallel

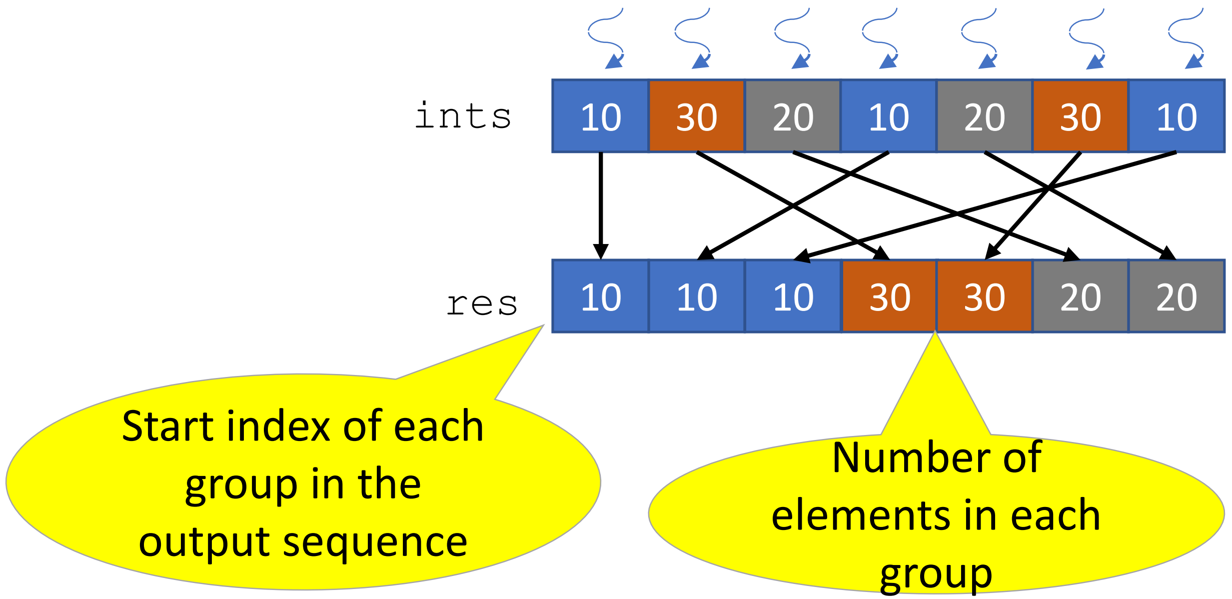
grouping ~ shuffling
input item → output offset such that groups are contiguous
output offset = group offset + item number
… but how to get the group offset, item number?



Parallel GroupBy

Process each input element in parallel

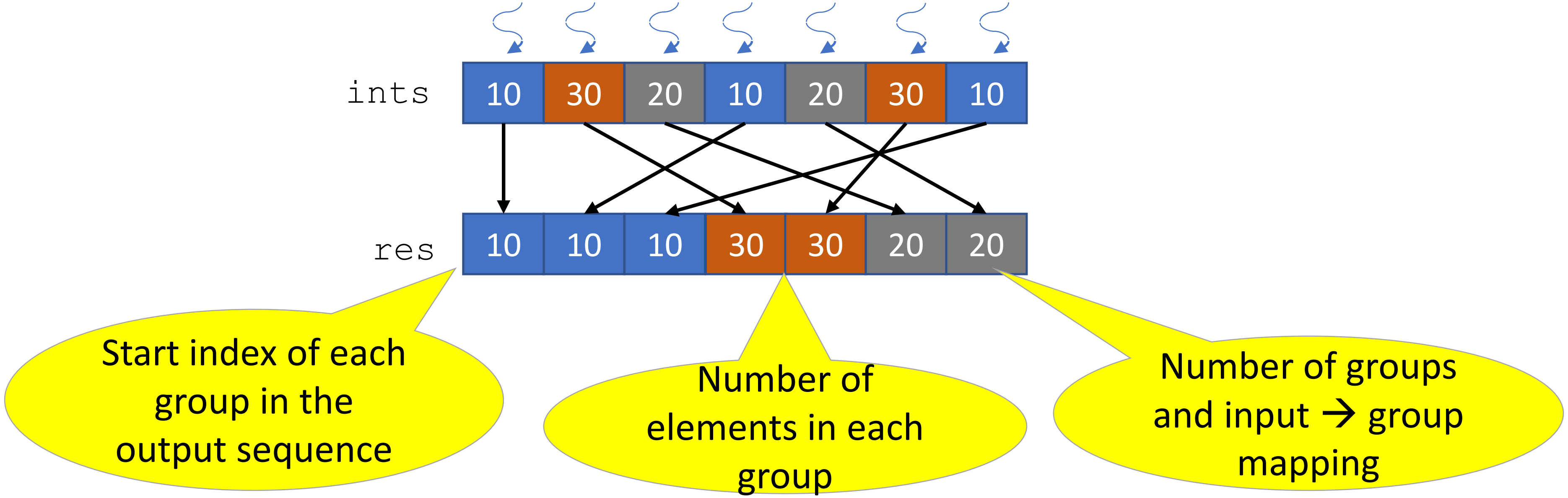
grouping ~ shuffling
input item → output offset such that groups are contiguous
output offset = group offset + item number
… but how to get the group offset, item number?



Parallel GroupBy

Process each input element in parallel

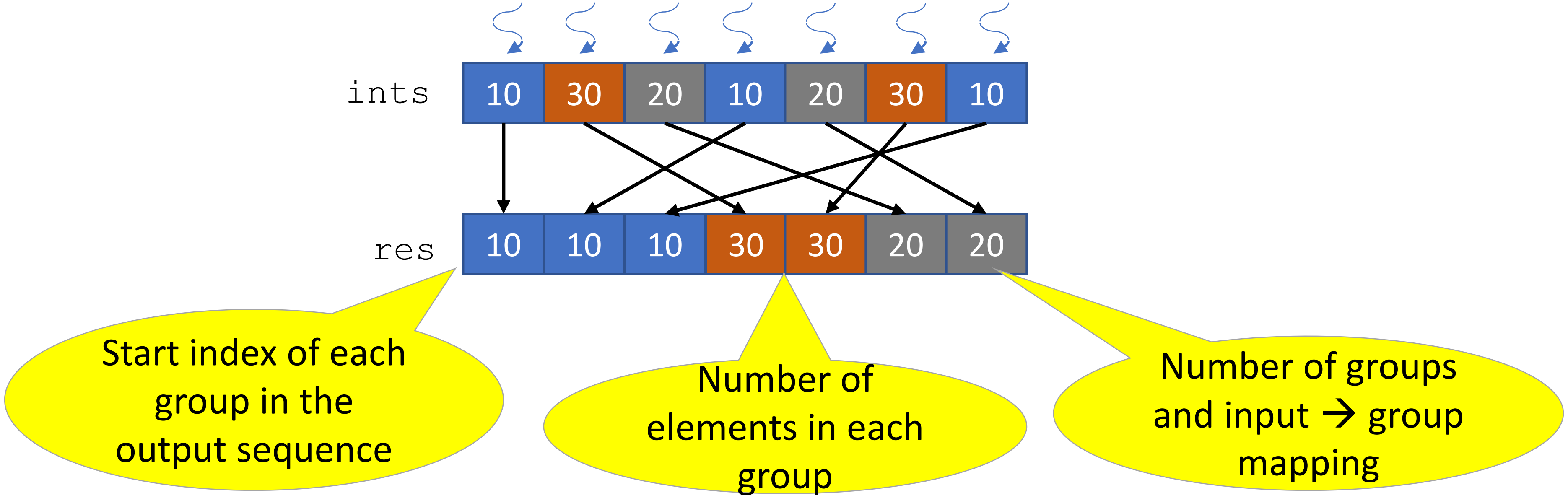
grouping ~ shuffling
input item → output offset such that groups are contiguous
output offset = group offset + item number
… but how to get the group offset, item number?



Parallel GroupBy

Process each input element in parallel

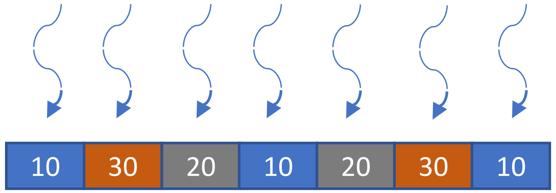
grouping ~ shuffling
input item → output offset such that groups are contiguous
output offset = group offset + item number
… but how to get the group offset, item number?



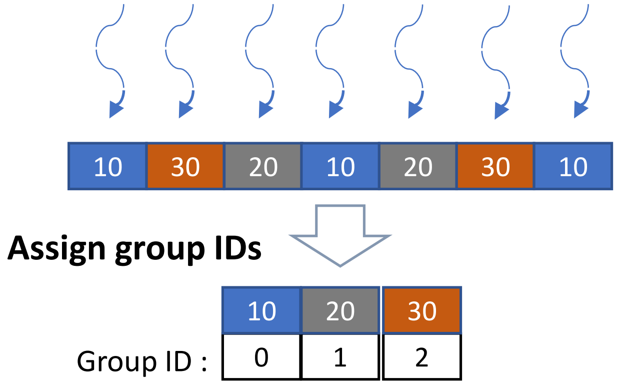
GroupBy with Parallel Primitives

10	30	20	10	20	30	10

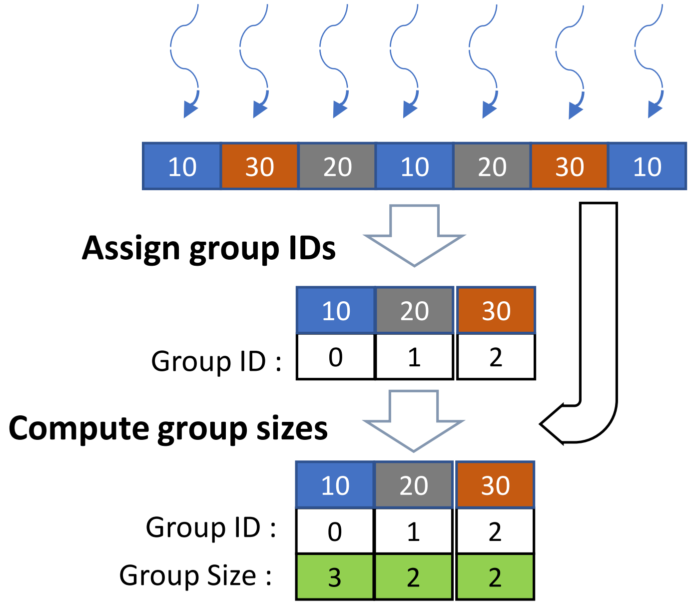
GroupBy with Parallel Primitives



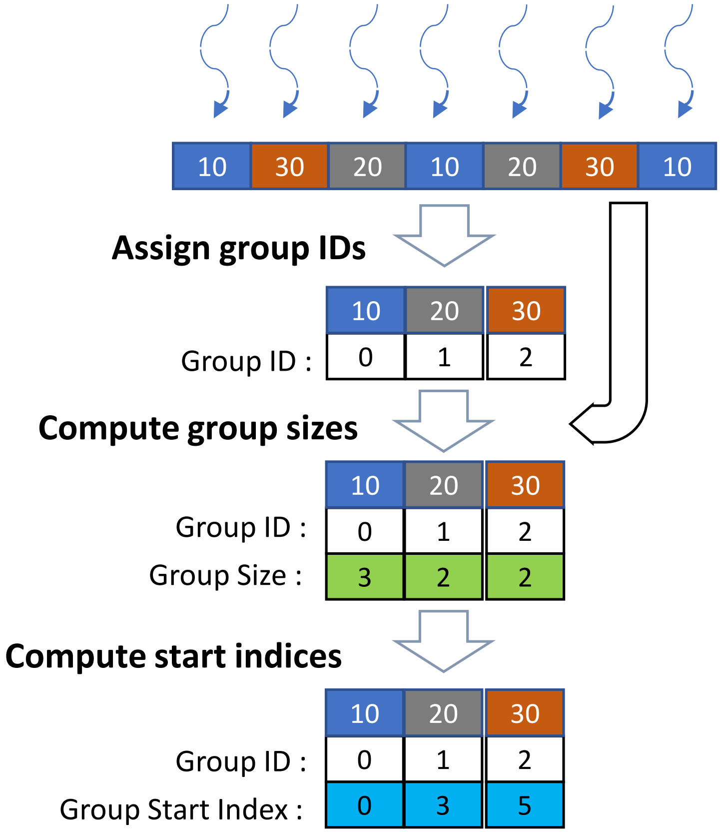
GroupBy with Parallel Primitives



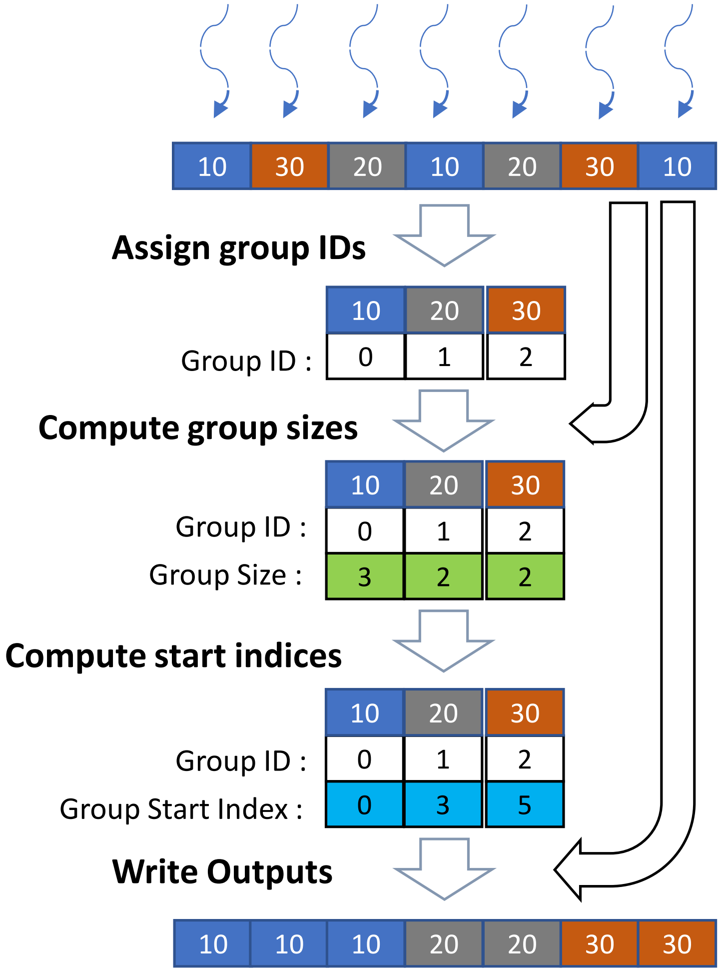
GroupBy with Parallel Primitives



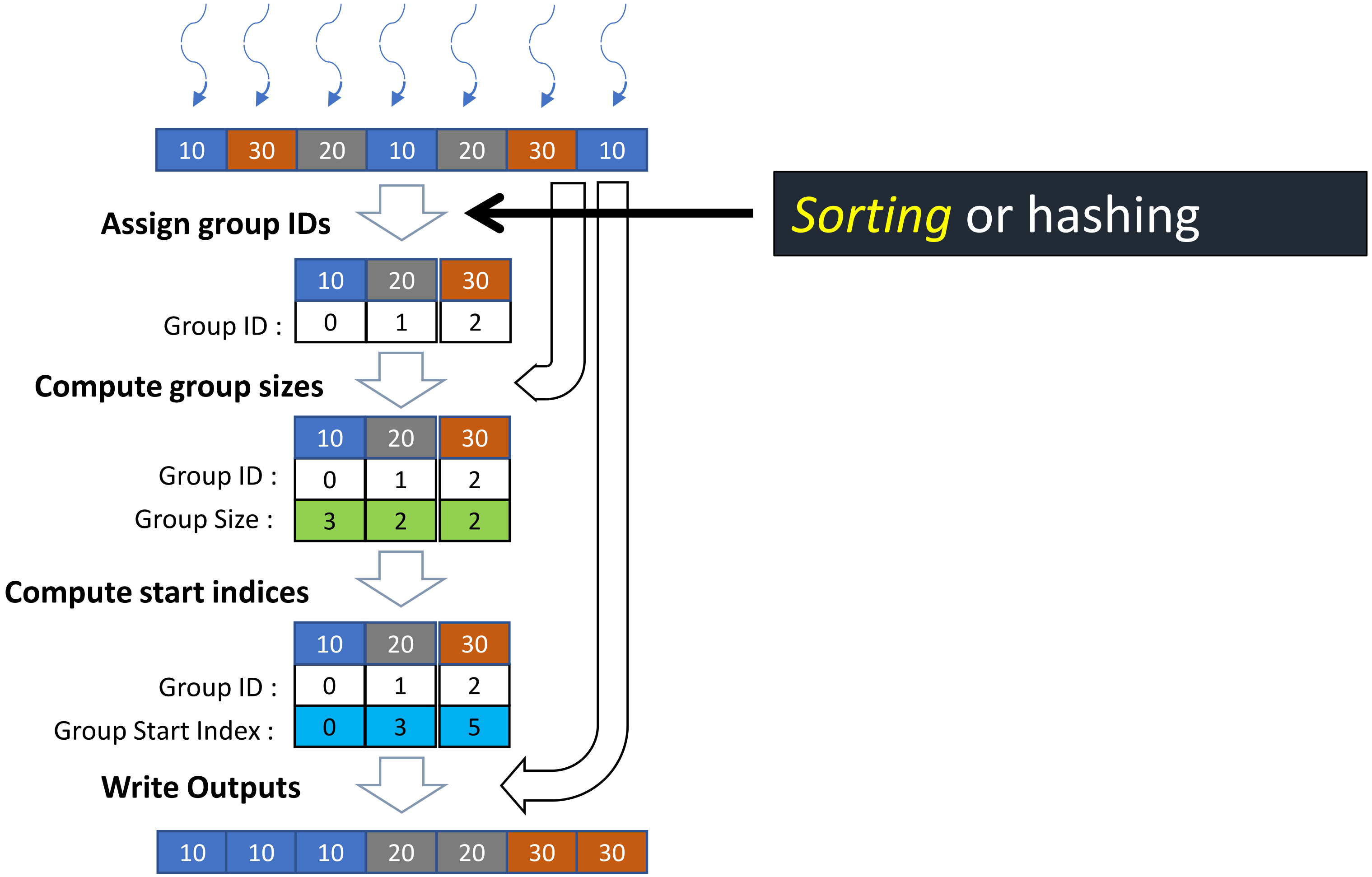
GroupBy with Parallel Primitives



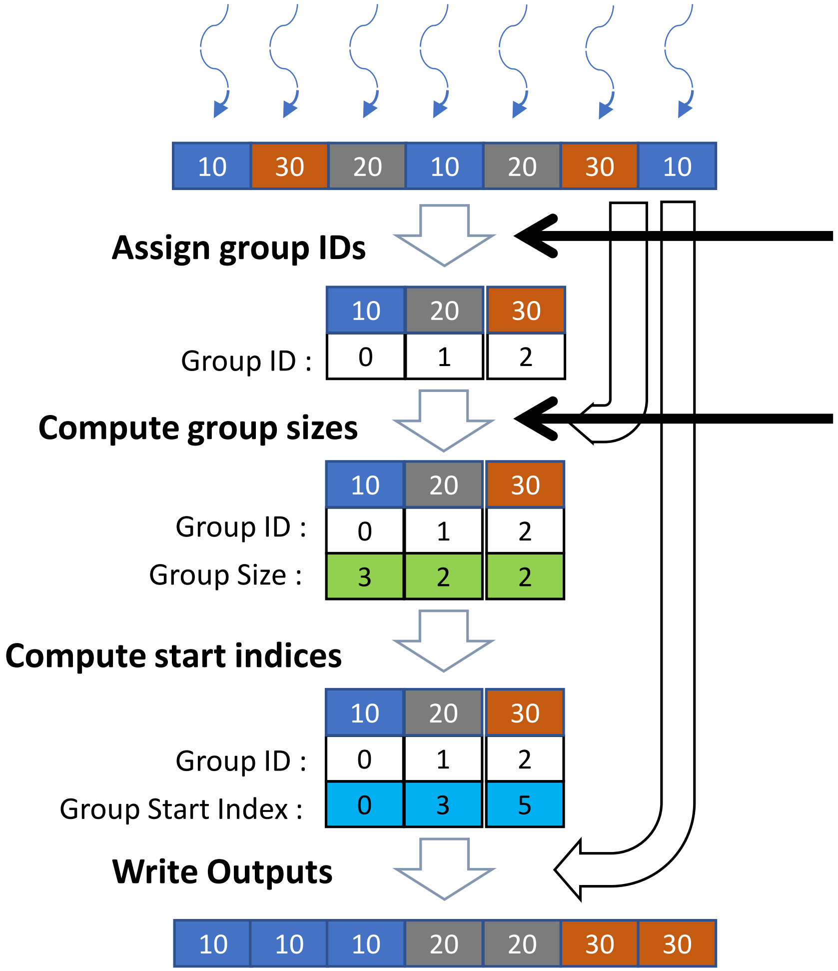
GroupBy with Parallel Primitives



GroupBy with Parallel Primitives



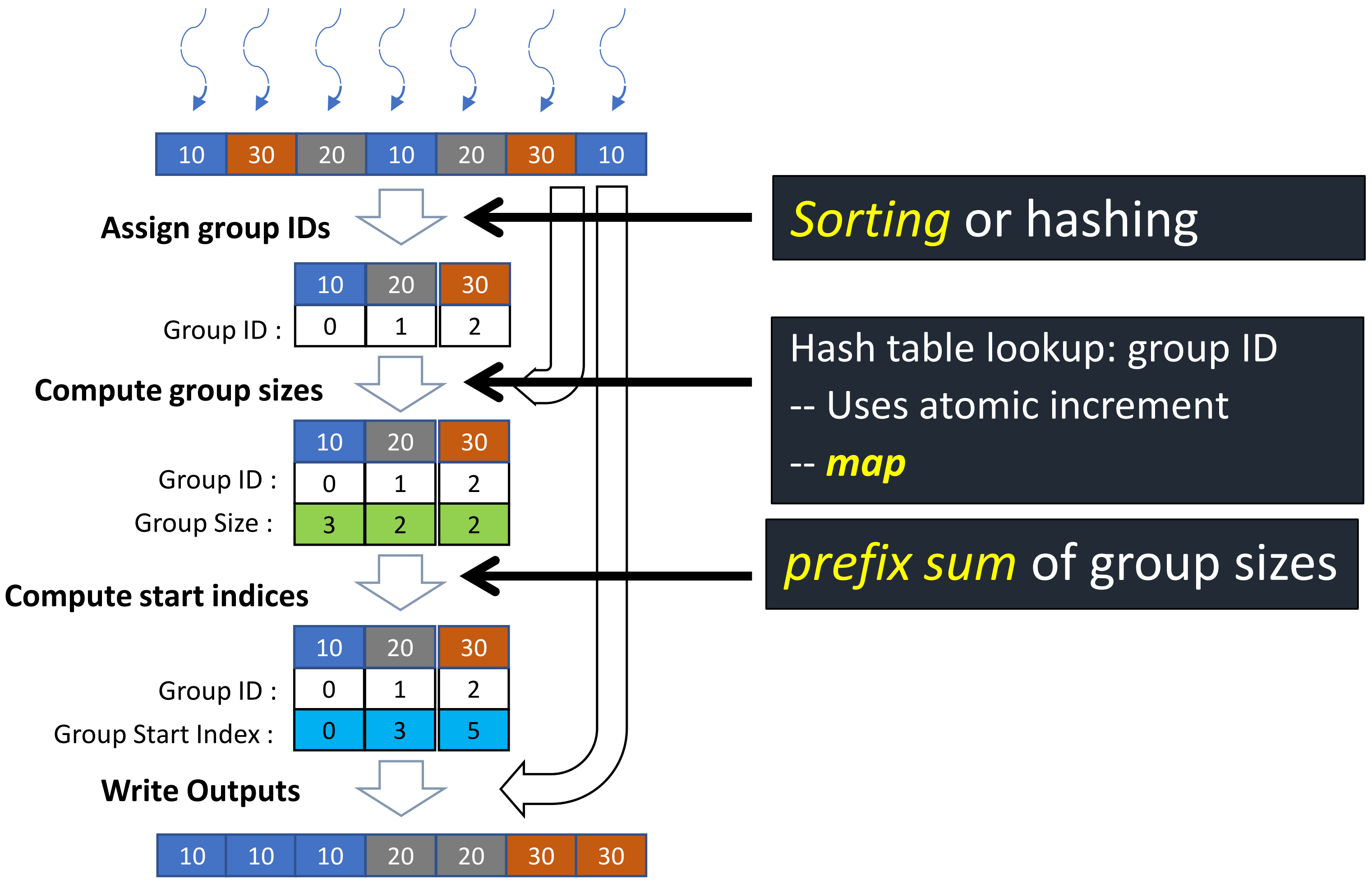
GroupBy with Parallel Primitives



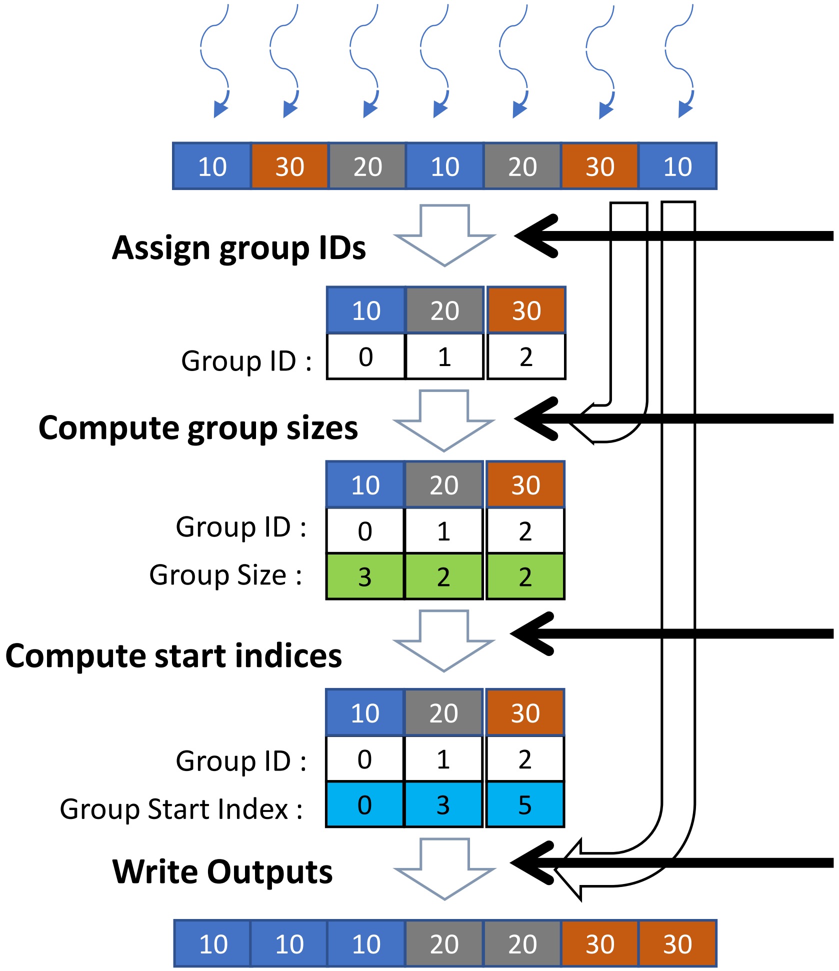
Sorting or hashing

Hash table lookup: group ID
-- Uses atomic increment
-- ***map***

GroupBy with Parallel Primitives



GroupBy with Parallel Primitives



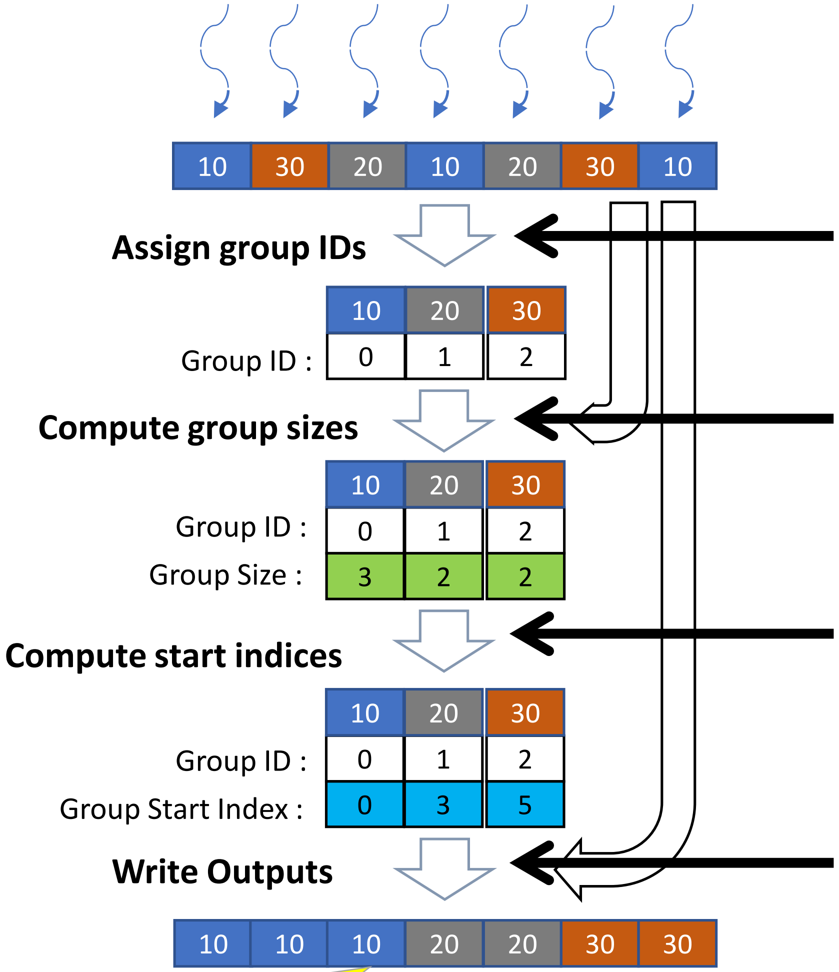
Sorting or hashing

Hash table lookup: group ID
-- Uses atomic increment
-- ***map***

prefix sum of group sizes

Write to output location
– Uses atomic increment
– ***Scatter gather***

GroupBy with Parallel Primitives



Sorting or hashing

Hash table lookup: group ID
-- Uses atomic increment
-- ***map***

prefix sum of group sizes

Write to output location
- Uses atomic increment
- ***Scatter gather***

We'll revisit after more CUDA background…

Parallel Patterns

Parallel Patterns

Thrust:

Large set of algorithms

- ~75 functions
- ~125 variations

Flexible

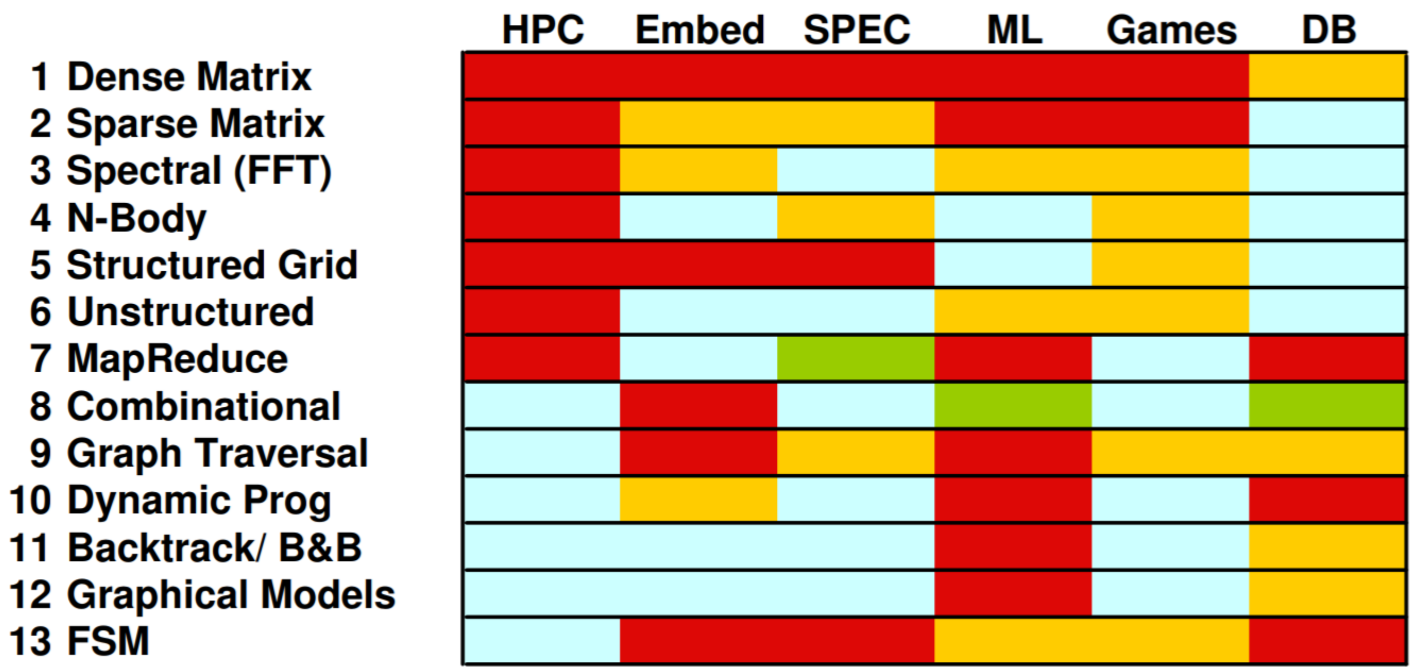
- User-defined types
- User-defined operators

Algorithm	Description
`reduce`	Sum of a sequence
`find`	First position of a value in a sequence
`mismatch`	First position where two sequences differ
`inner_product`	Dot product of two sequences
`equal`	Whether two sequences are equal
`min_element`	Position of the smallest value
`count`	Number of instances of a value
`is_sorted`	Whether sequence is in sorted order
`transform_reduce`	Sum of transformed sequence

Parallel Patterns

Parallel Patterns

Dwarf Popularity (Red Hot → Blue Cool)



Parallel Patterns

Parallel Patterns

TBB is a collection of components for parallel programming:

- Basic algorithms: `parallel_for`, `parallel_reduce`, `parallel_scan`
- Advanced algorithms: `parallel_while`, `parallel_do`, `parallel_pipeline`, `parallel_sort`
- Containers: `concurrent_queue`, `concurrent_priority_queue`, `concurrent_vector`, `concurrent_hash_map`
- Memory allocation: `scalable_malloc`, `scalable_free`, `scalable_realloc`, `scalable_calloc`, `scalable_allocator`, `cache_aligned_allocator`
- Mutual exclusion: `mutex`, `spin_mutex`, `queuing_mutex`, `spin_rw_mutex`, `queuing_rw_mutex`, `recursive_mutex`
- Atomic operations: `fetch_and_add`, `fetch_and_increment`, `fetch_and_decrement`, `compare_and_swap`, `fetch_and_store`
- Timing: portable fine grained global time stamp
- Task scheduler: direct access to control the creation and activation of tasks

Parallel Patterns

Summary

Re-expressing apparently sequential algorithms as combinations of parallel patterns is a common technique when targeting GPUs

Examples

- Reductions
- Scans
- Re-orderings (scatter/gather)
- Sort
- Map

What is the *right* set of parallel patterns to support?