Parallel Architectures
Parallel Algorithms
CUDA

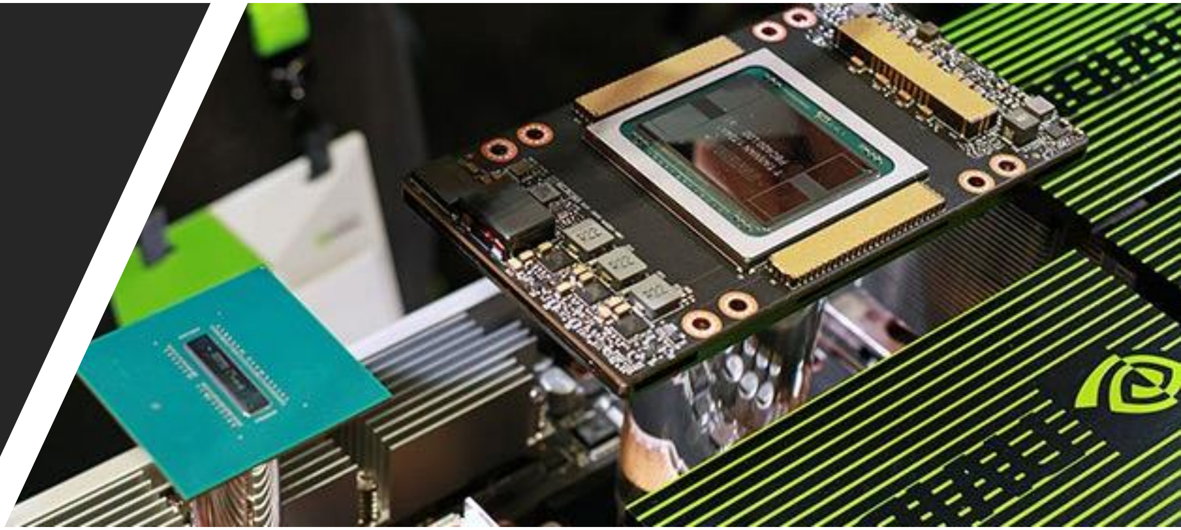
Chris Rossbach

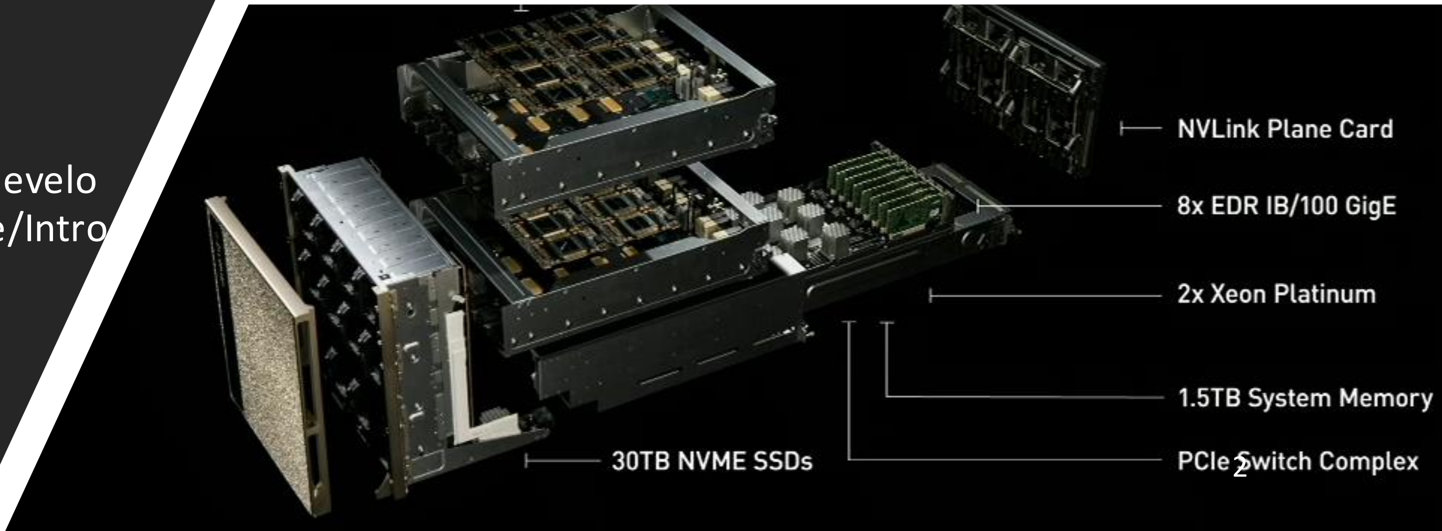
cs378h

Outline for Today

- Questions?
- Administrivia
  - pedagogical-* machines should be available
- Agenda
  - Parallel Algorithms
  - CUDA

- Acknowledgements: http://developer.download.nvidia.com/compute/developertrainingmaterials/presentations/cuda_language/Introduction_to_CUDA_C.pptx

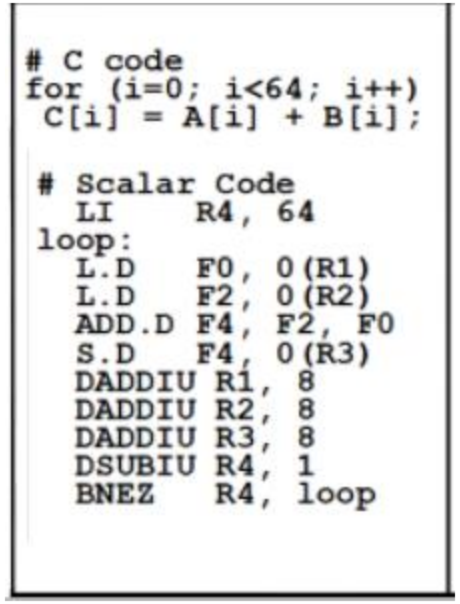




Faux Quiz Questions

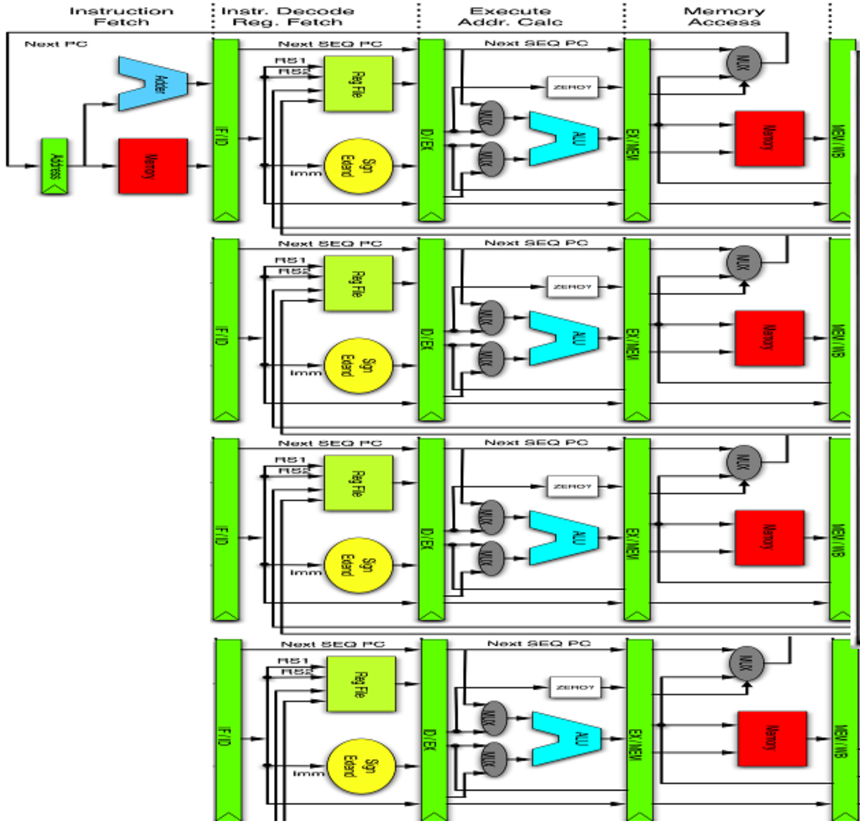
- What is a reduction? A prefix sum? Why are they hard to parallelize and what basic techniques can be used to parallelize them?
- Define flow dependence, output dependence, and anti-dependence: give an example of each. Why/how do compilers use them to detect loop-independent vs loop-carried dependences?
- What is the difference between a thread-block and a warp?
- How/Why must programmers copy data back and forth to a GPU?
- What is “shared memory” in CUDA? Describe a setting in which it might be useful.
- CUDA kernels have implicit barrier synchronization. Why is __syncthreads() necessary in light of this fact?
- How might one implement locks on a GPU?
- What ordering guarantees does a GPU provide across different hardware threads’ access to a single memory location? To two disjoint locations?
- When is it safe for one GPU thread to wait (e.g. by spinning) for another?

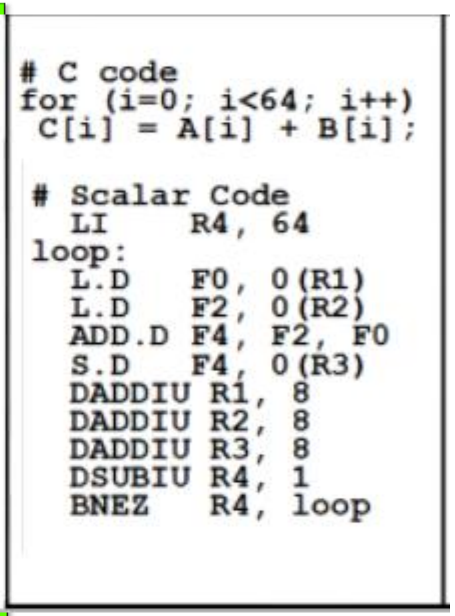
Review: what is a vector processor?



```
# C code
for (i=0; i<64; i++)
 C[i] = A[i] + B[i];

 # Scalar Code
   LI     R4, 64
 loop:
   L.D    F0, 0(R1)
   L.D    F2, 0(R2)
   ADD.D  F4, F2, F0
   S.D    F4, 0(R3)
   DADDIU R1, 8
   DADDIU R2, 8
   DADDIU R3, 8
   DSUBIU R4, 1
   BNEZ   R4, loop
```

Review: what is a vector processor?

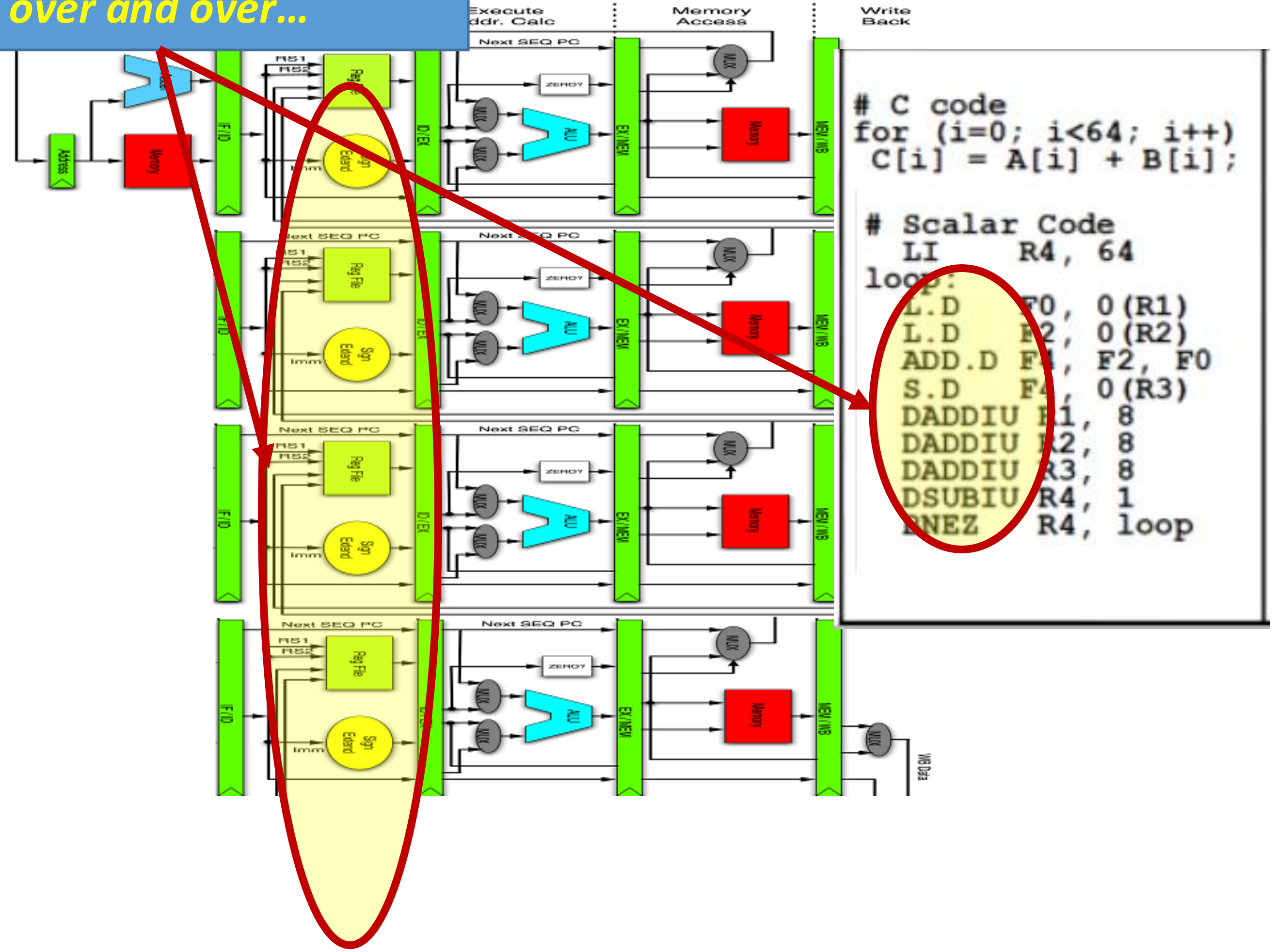




```
# C code
for (i=0; i<64; i++)
 C[i] = A[i] + B[i];

 # Scalar Code
   LI     R4, 64
 loop:
   L.D    F0, 0(R1)
   L.D    F2, 0(R2)
   ADD.D  F4, F2, F0
   S.D    F4, 0(R3)
   DADDIU R1, 8
   DADDIU R2, 8
   DADDIU R3, 8
   DSUBIU R4, 1
   BNEZ   R4, loop
```

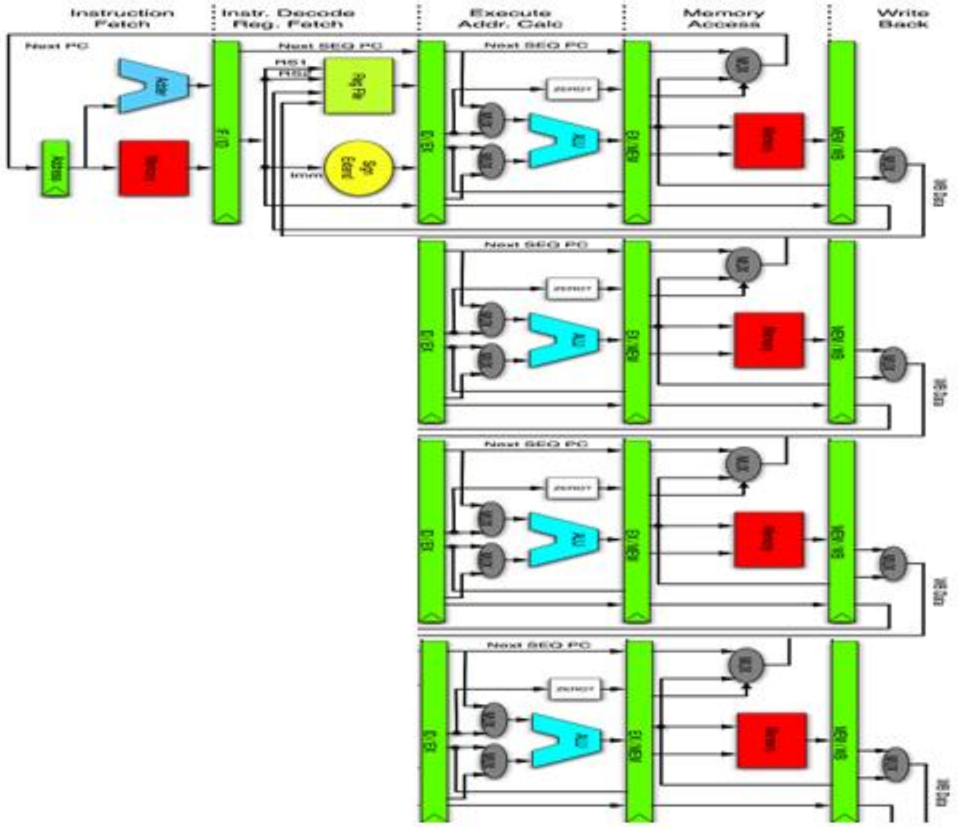
Review: what is a vector processor?

Dont decode same instruction over and over…

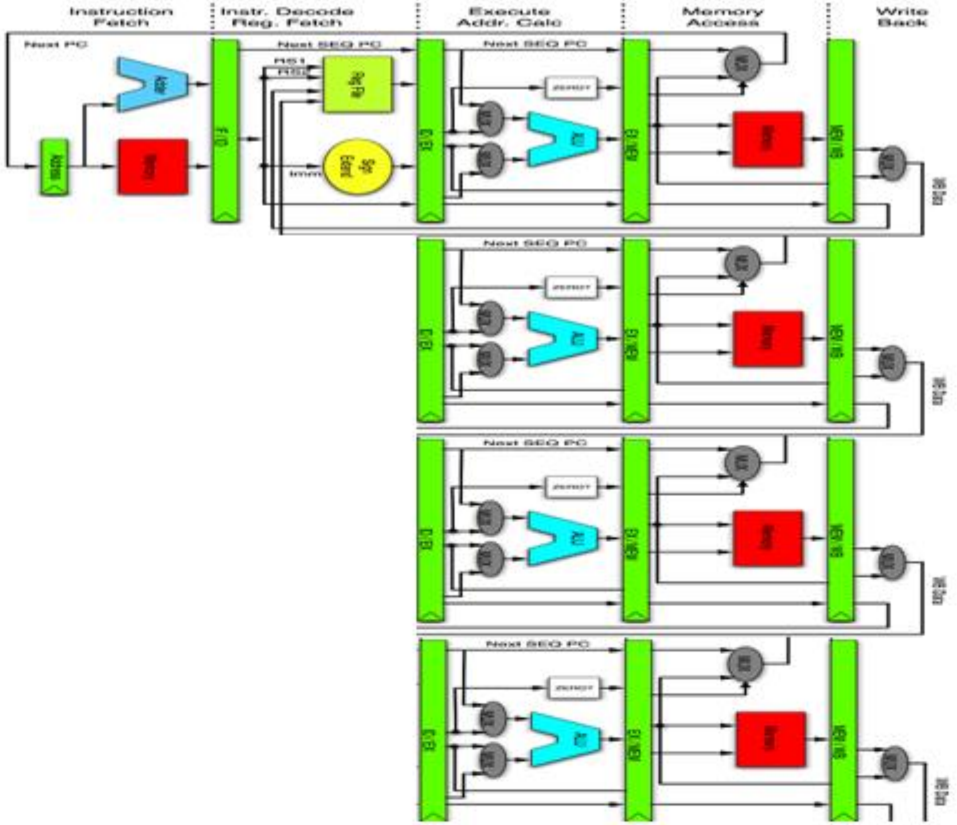


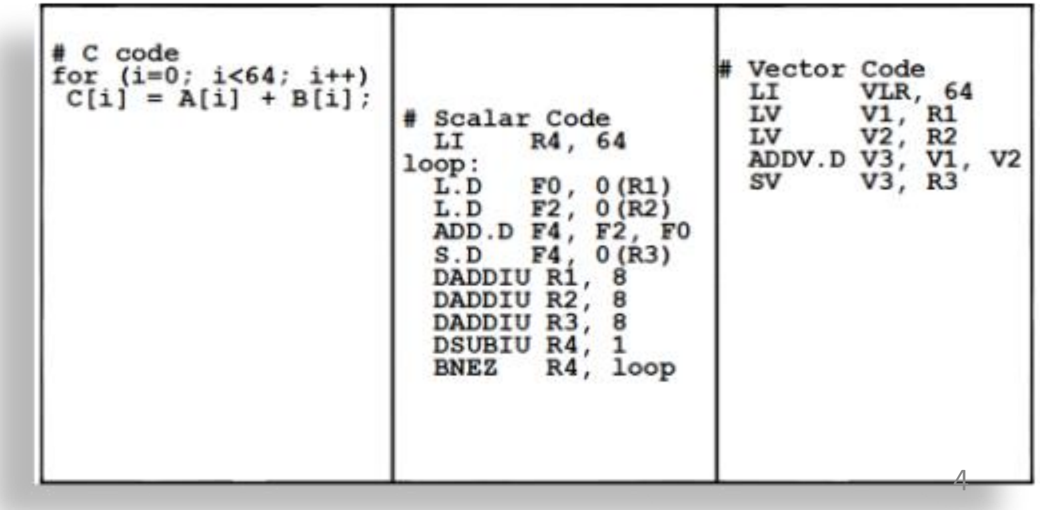
```
# C code
for (i=0; i<64; i++)
  C[i] = A[i] + B[i];

# Scalar Code
  LI     R4, 64
loop:
  L.D    F0, 0(R1)
  L.D    F2, 0(R2)
  ADD.D  F4, F2, F0
  S.D    F4, 0(R3)
  DADDIU R1, 8
  DADDIU R2, 8
  DADDIU R3, 8
  DSUBIU R4, 1
  BNEZ   R4, loop
```

Review: what is a vector processor?



Review: what is a vector processor?



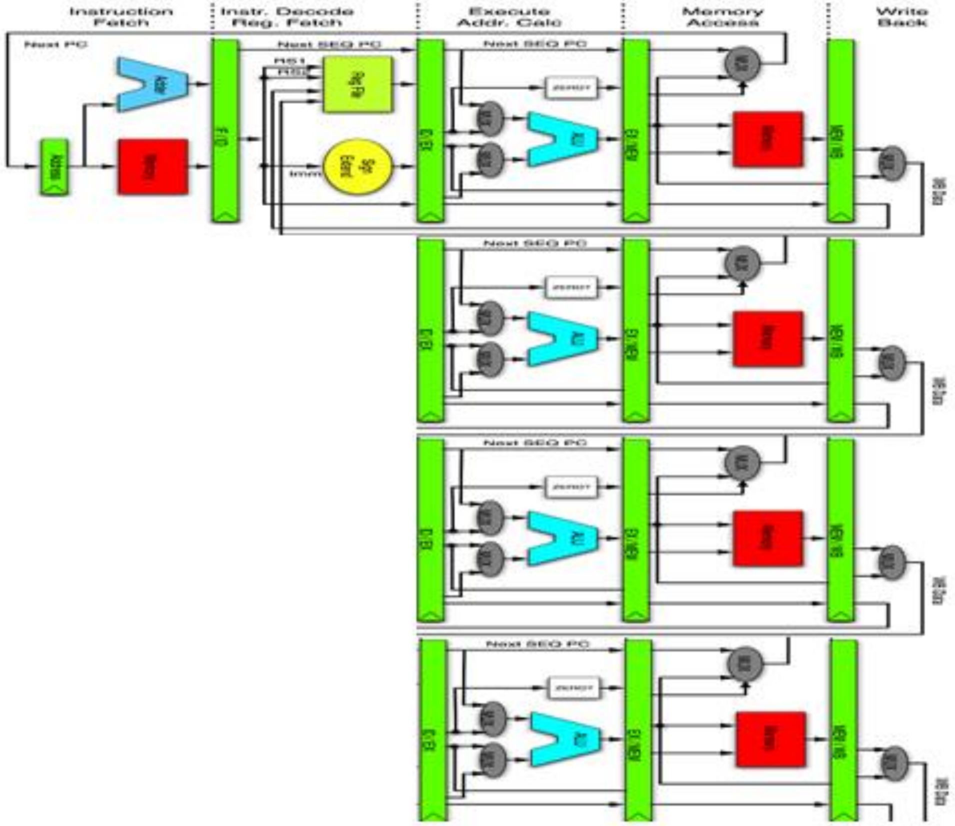


```
# C code
for (i=0; i<64; i++)
 C[i] = A[i] + B[i];
```

```
# Scalar Code
  LI     R4, 64
loop:
  L.D    F0, 0(R1)
  L.D    F2, 0(R2)
  ADD.D F4, F2, F0
  S.D    F4, 0(R3)
  DADDIU R1, 8
  DADDIU R2, 8
  DADDIU R3, 8
  DSUBIU R4, 1
  BNEZ   R4, loop
```

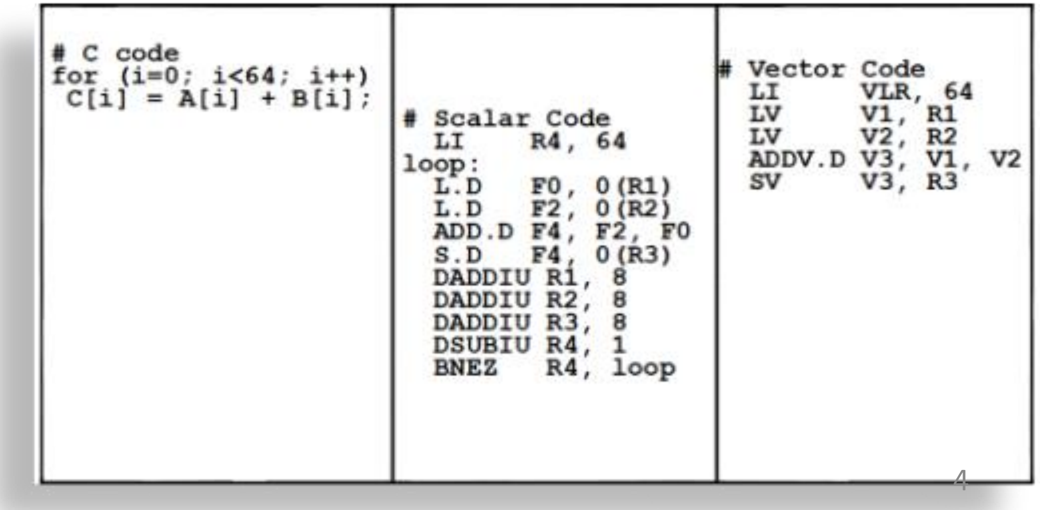
```
# Vector Code
  LI      VLR, 64
  LV      V1, R1
  LV      V2, R2
  ADDV.D V3, V1, V2
  SV      V3, R3
```

Review: what is a vector processor?



Implementation:

- Instruction fetch control logic shared
- Same instruction stream executed on
- Multiple pipelines
- Multiple different operands in parallel

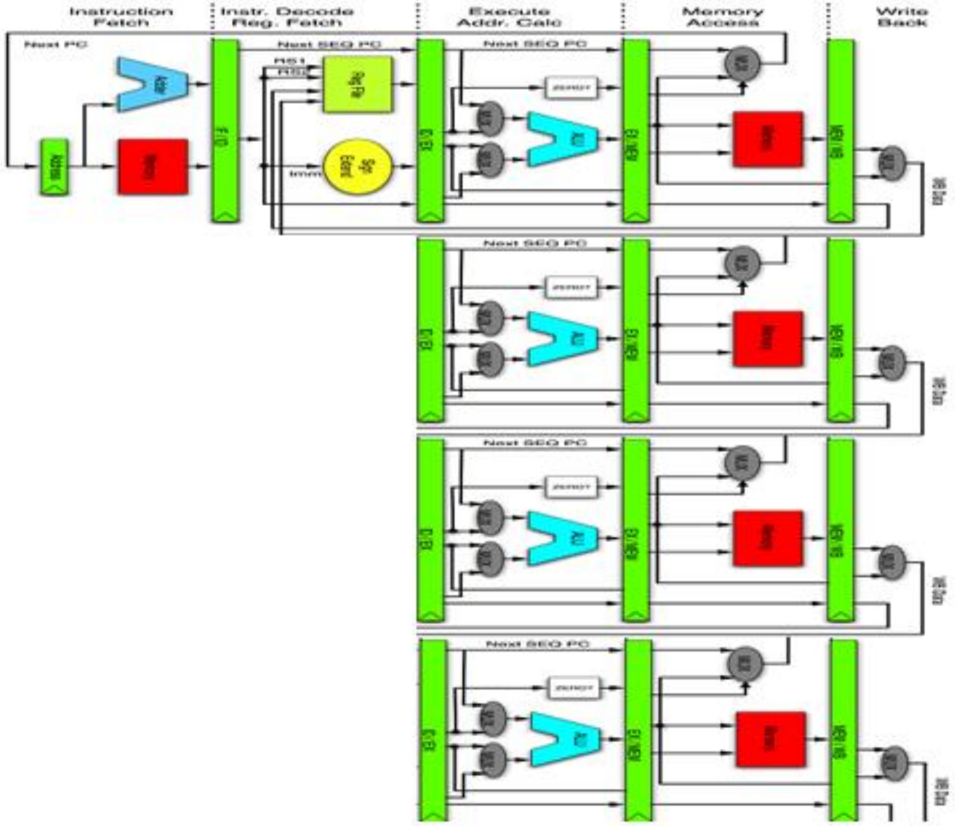


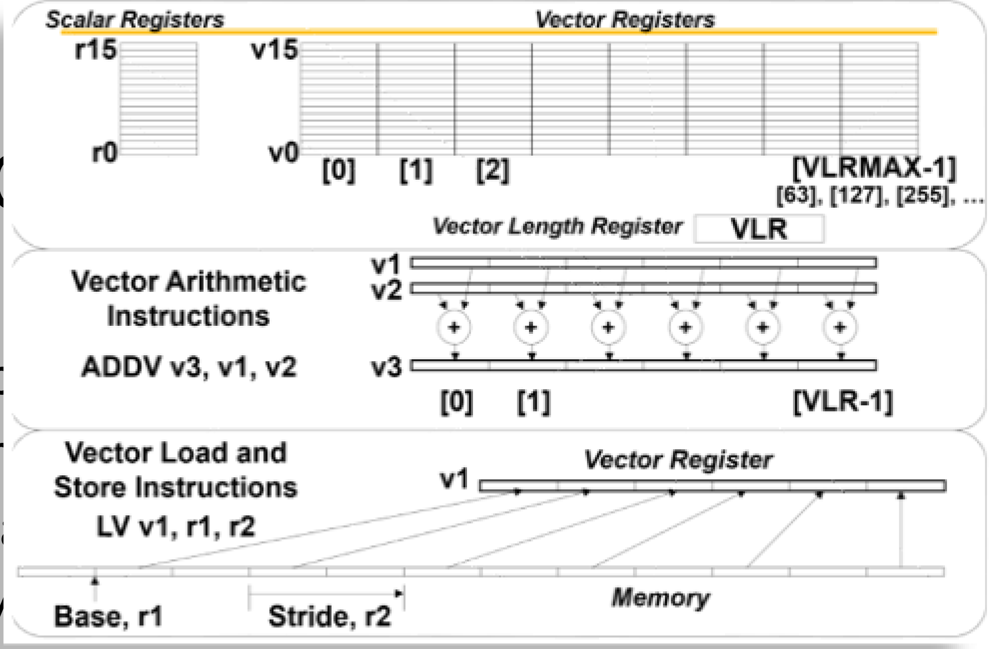
```
# C code
for (i=0; i<64; i++)
 C[i] = A[i] + B[i];
```

```
# Scalar Code
  LI     R4, 64
loop:
  L.D    F0, 0(R1)
  L.D    F2, 0(R2)
  ADD.D  F4, F2, F0
  S.D    F4, 0(R3)
  DADDIU R1, 8
  DADDIU R2, 8
  DADDIU R3, 8
  DSUBIU R4, 1
  BNEZ   R4, loop
```

```
# Vector Code
  LI     VLR, 64
  LV     V1, R1
  LV     V2, R2
  ADDV.D V3, V1, V2
  SV     V3, R3
```

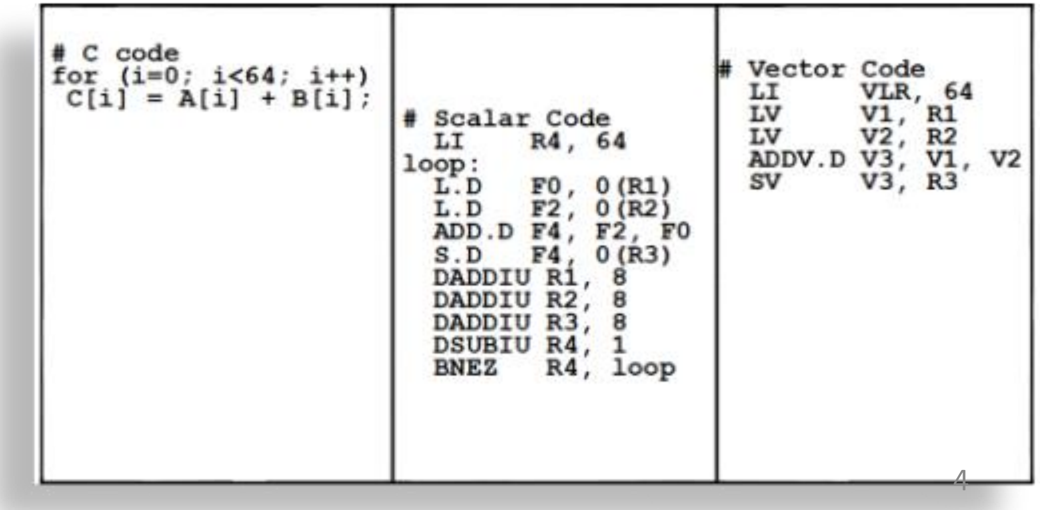
Review: what is a vector proc





Imp

- Ir
- Sa
- M
- Multiple different operands in parallel



```
# C code
for (i=0; i<64; i++)
  C[i] = A[i] + B[i];
```

```
# Scalar Code
  LI     R4, 64
loop:
  L.D    F0, 0(R1)
  L.D    F2, 0(R2)
  ADD.D  F4, F2, F0
  S.D    F4, 0(R3)
  DADDIU R1, 8
  DADDIU R2, 8
  DADDIU R3, 8
  DSUBIU R4, 1
  BNEZ   R4, loop
```

```
# Vector Code
  LI     VLR, 64
  LV     V1, R1
  LV     V2, R2
  ADDV.D V3, V1, V2
  SV     V3, R3
```

Review: Hardware multi-threading

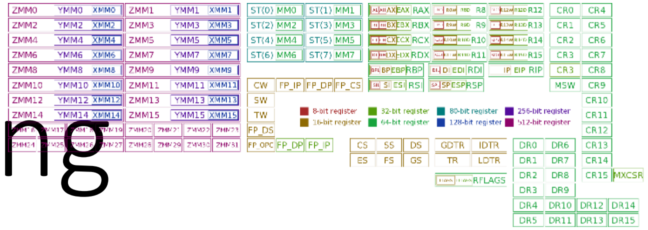
Review: Hardware multi-threading

- Address memory bottleneck

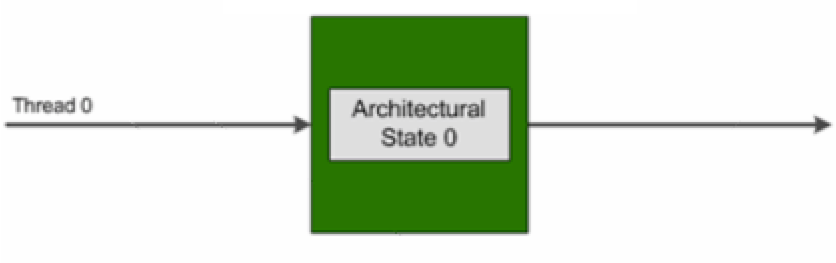
Review: Hardware multi-threading

- Address memory bottleneck
- Share exec unit across
  - Instruction streams
  - Switch on stalls

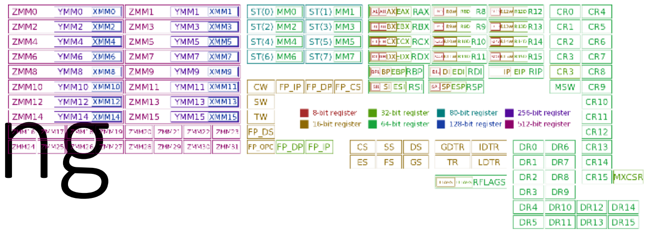
Review: Hardware multi-threading



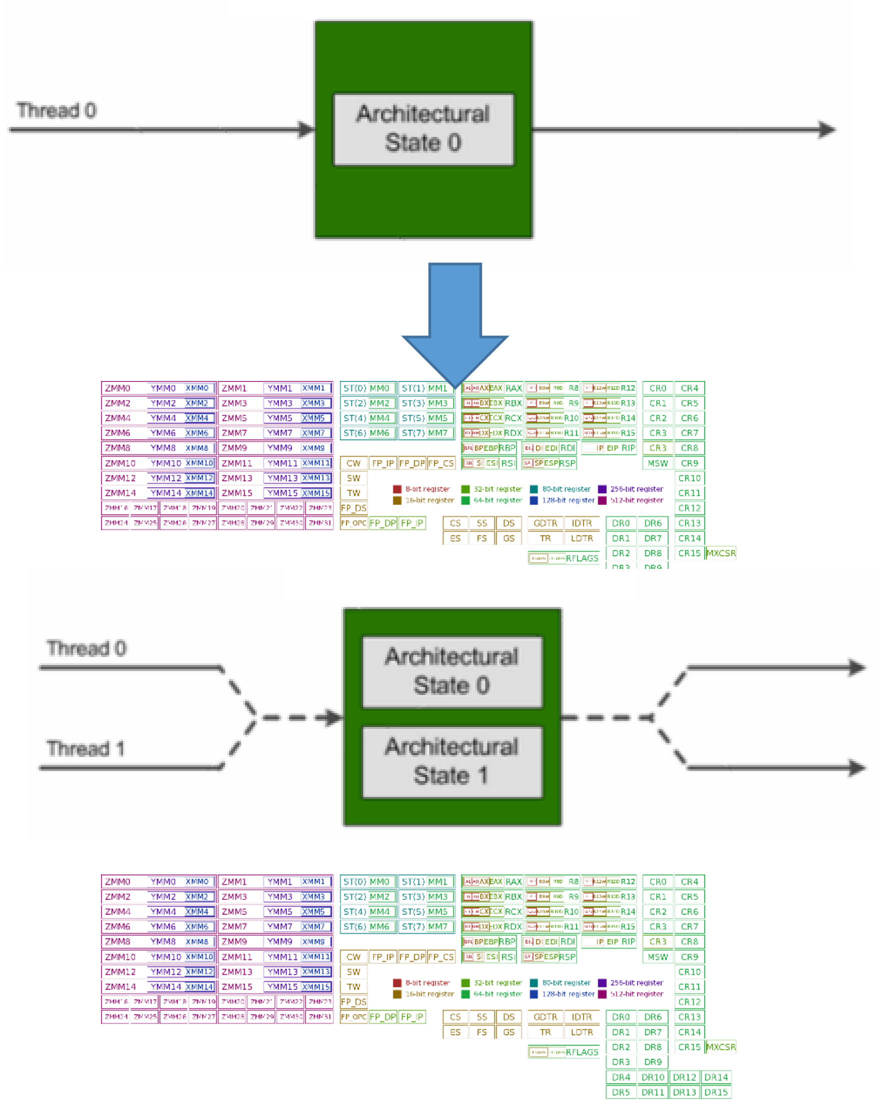
- Address memory bottleneck
- Share exec unit across
  - Instruction streams
  - Switch on stalls



Review: Hardware multi-threading

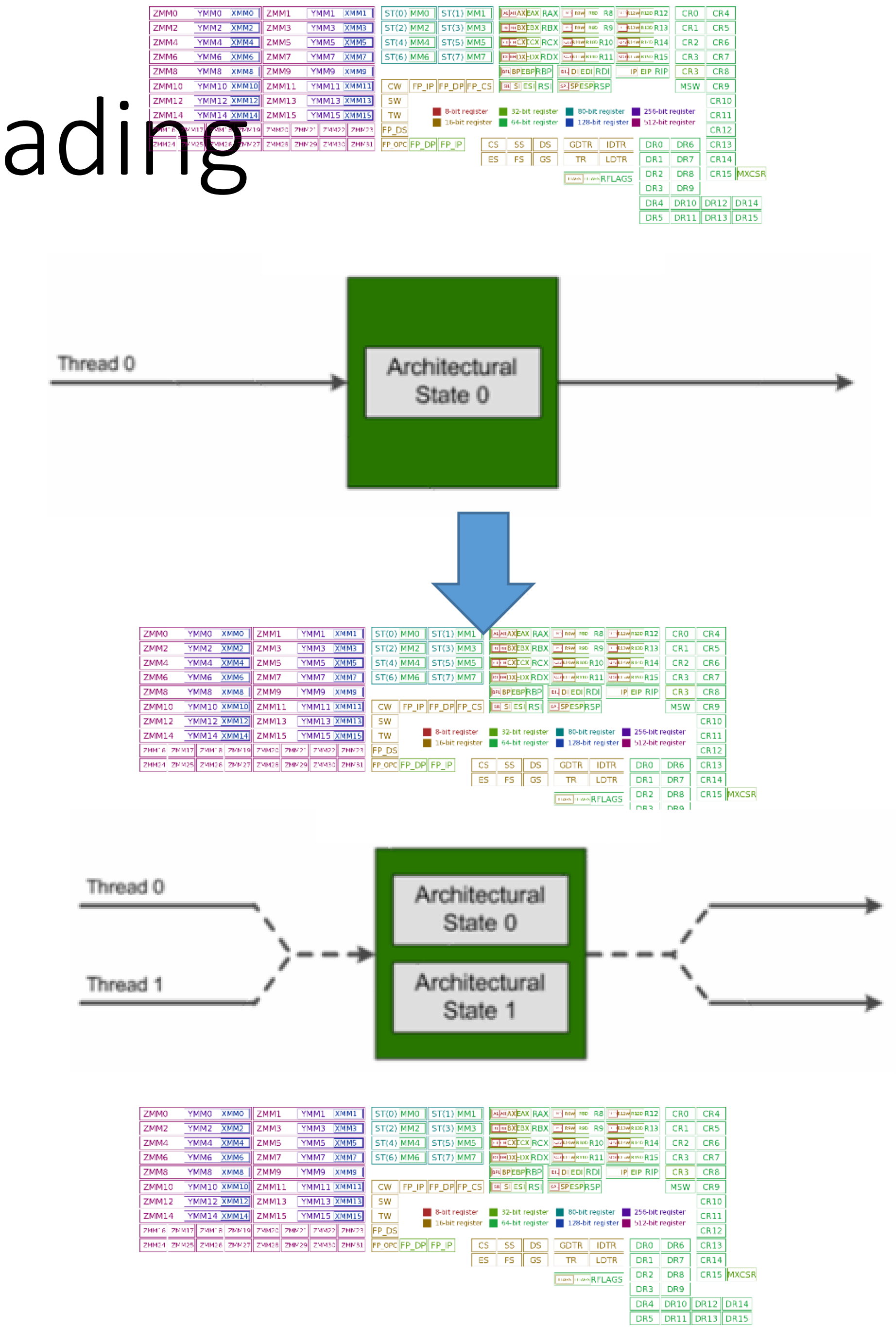


- Address memory bottleneck
- Share exec unit across
  - Instruction streams
  - Switch on stalls

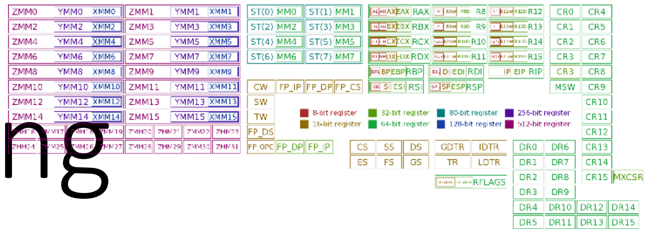


Review: Hardware multi-threading

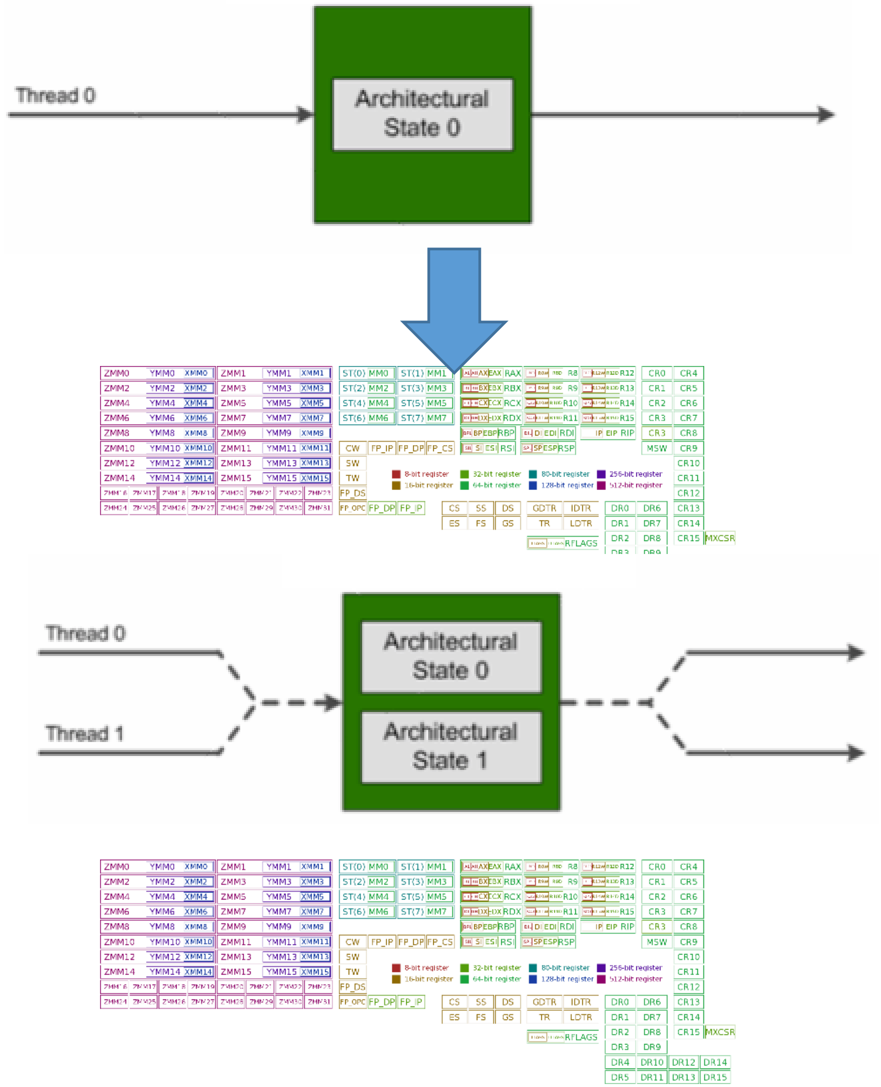
- Address memory bottleneck
- Share exec unit across
  - Instruction streams
  - Switch on stalls
- Looks like multiple cores to the OS



Review: Hardware multi-threading



- Address memory bottleneck
- Share exec unit across
  - Instruction streams
  - Switch on stalls
- Looks like multiple cores to the OS
- Three variants:
  - Coarse
  - Fine-grain
  - Simultaneous



Programming Model

- ***GPUs are I/O devices, managed by user-code***
- “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

- ***GPUs are I/O devices, managed by user-code***
- “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 Patterns
  - Map
  - Scatter, Gather
  - Reduction
  - Scan
  - Search, Sort

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 Patterns
  - Map
  - Scatter, Gather
  - Reduction
  - Scan
  - Search, Sort

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

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

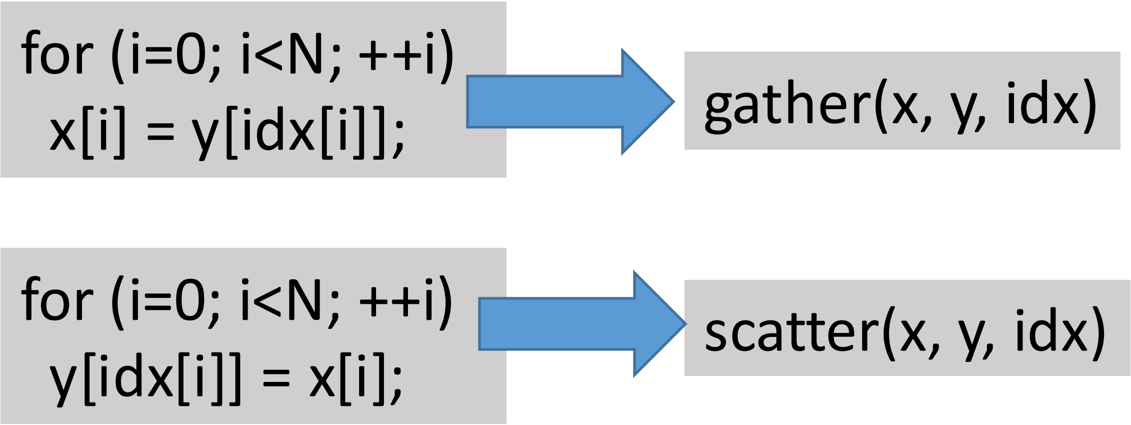
- Gather:
    - Read multiple items to single /packed location
- Scatter:
    - Write single/packed data item to multiple locations

Scatter and Gather

- Gather:
  - Read multiple items to single /packed location
- Scatter:
  - Write single/packed data item to multiple locations
- Inputs: x, y, indeces, N

Scatter and Gather

- Gather:
    - Read multiple items to single /packed location
- Scatter:
    - Write single/packed data item to multiple locations
- Inputs: x, y, indeces, 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, indeces, N

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

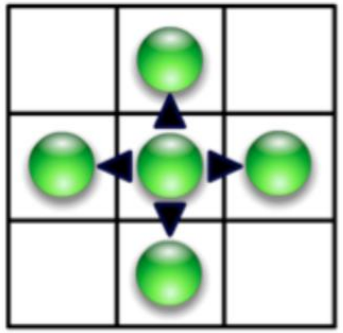
→

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

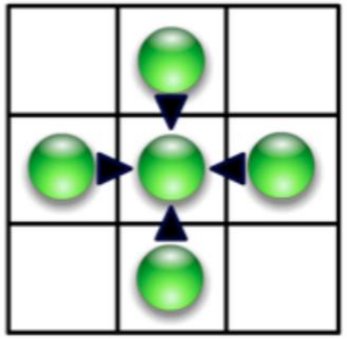
```
for (i=0; i<N; ++i)
  y[idx[i]] = x[i];
```

→

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



Scatter



Gather

Reduce

Reduce

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

Reduce

- Input
  - Associative operator op
  - Ordered set s = [a, b, c, … z]
- Reduce(op, s) returns a op b op c … op z

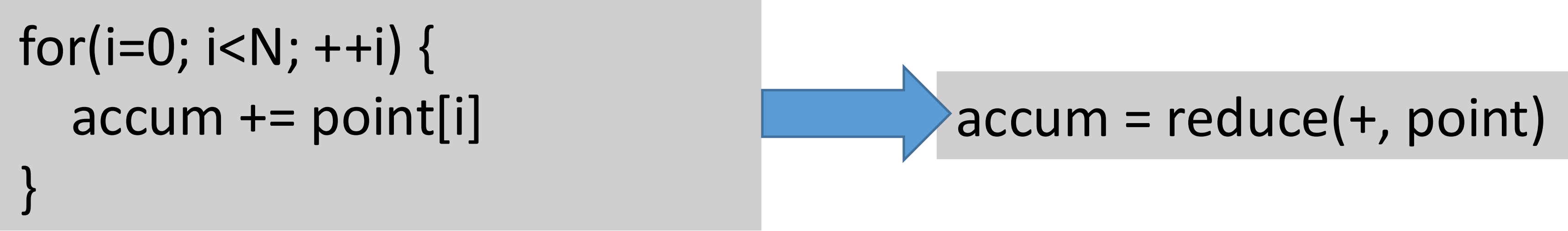
Reduce

- 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]
}
```

Reduce

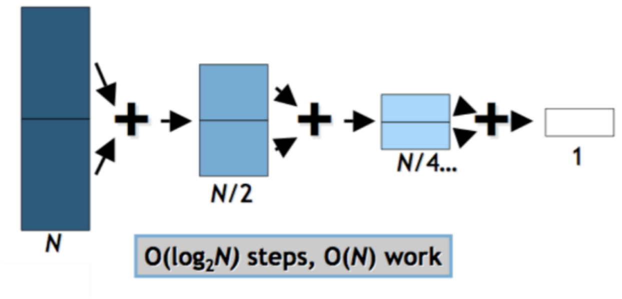
- Input
    - Associative operator op
    - Ordered set s = [a, b, c, … z]
- Reduce(op, s) returns a op b op c … op z

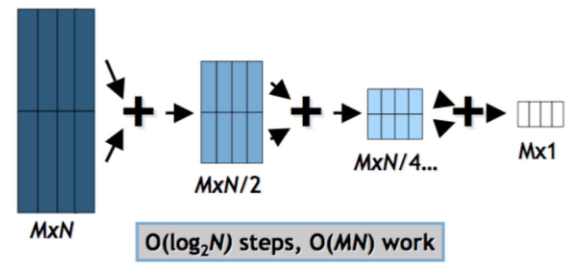


```
for(i=0; i<N; ++i) {
    accum += point[i]
}
```

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

Reduce



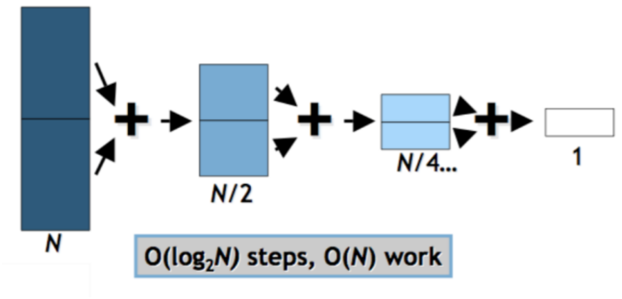


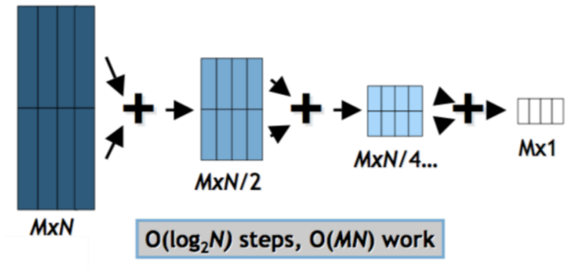
- Input
  - Associative operator op
  - Ordered set s = [a, b, c, … z]
- Reduce(op, s) returns a op b op c … op z

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

```
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]
}
```

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

Why must op be associative?

Scan (prefix sum)

Scan (prefix sum)

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

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 (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) …]

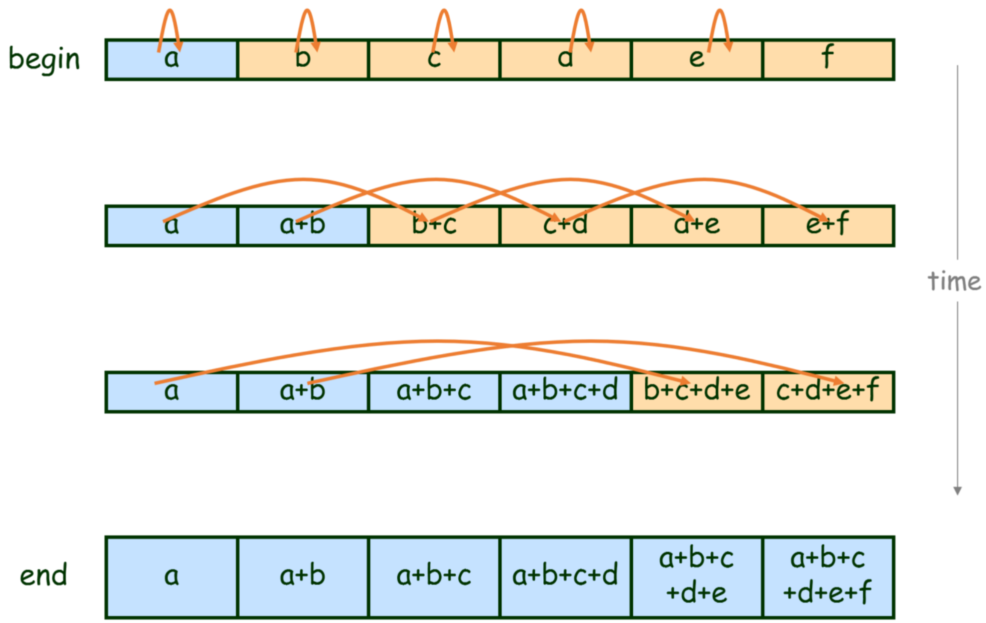
- Scan is the workhorse of parallel algorithms:
  - Sort, histograms, sparse matrix, string compare, …

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

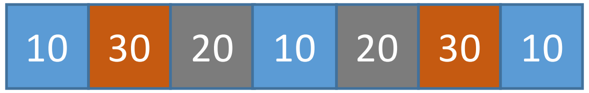
- Group a collection by key
- Lambda function maps elements → key

```
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);
```

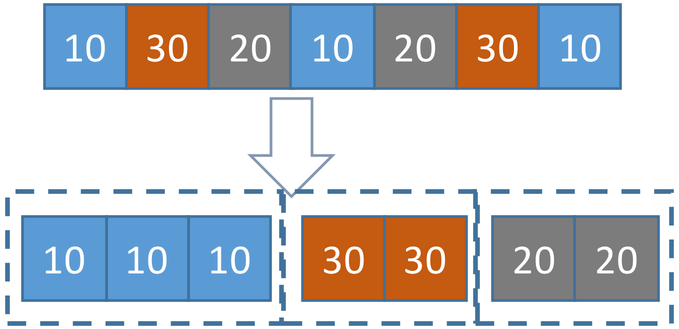


10 30 20 10 20 30 10

Example: Parallel GroupBy

- Group a collection by key
- Lambda function maps elements → key

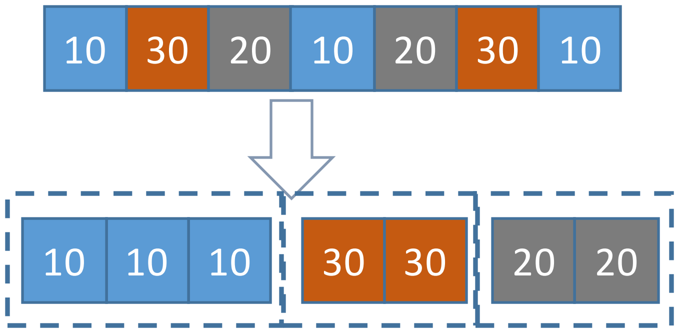
```
var res = ints.GroupBy(x => x);
```



Example: Parallel GroupBy

- Group a collection by key
- Lambda function maps elements → key

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



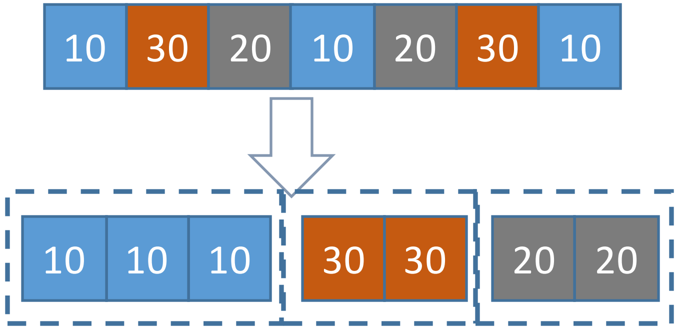
```
foreach(T elem in PF(ints))
{
   key   = KeyLambda(elem);

   group = GetGroup(key)

   group.Add(elem);
}
```

Example: Parallel GroupBy

- Group a collection by key
- Lambda function maps elements → key

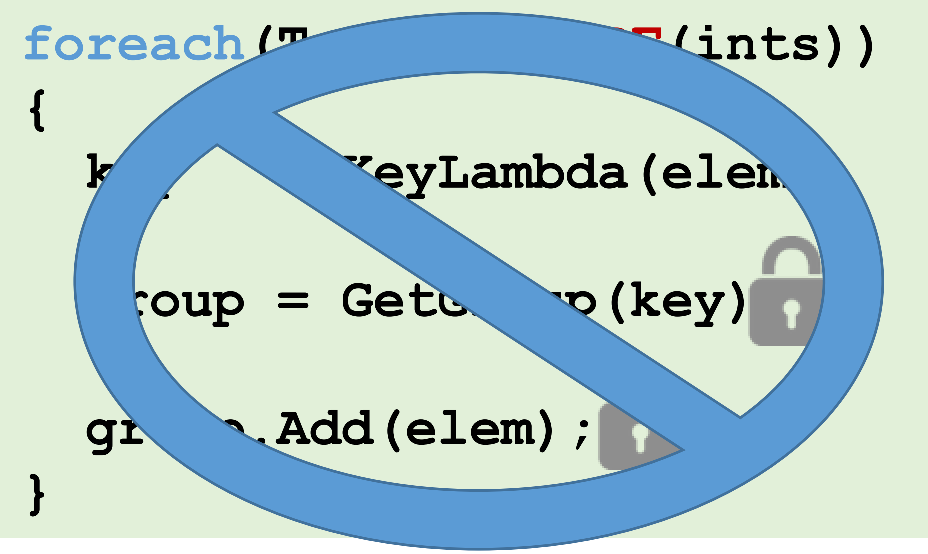
```
var res = ints.GroupBy(x => x);
```

- *Insufficient Parallelism*
- *Requires synchronization*



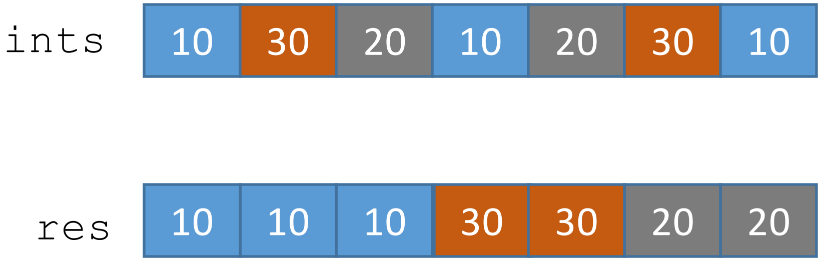
10 30 20 10 20 30 10

10 10 10 | 30 30 | 20 20



```
foreach(T          (ints))
{
  k           eyLambda(elem

   roup = GetG    p(key)

  gr    p.Add(elem);
}
```

Parallel GroupBy

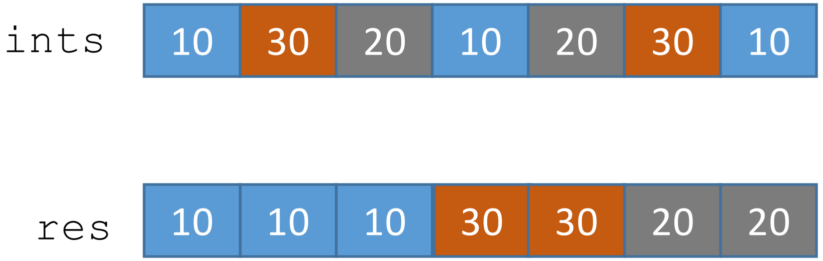


ints	10	30	20	10	20	30	10
res	10	10	10	30	30	20	20

Parallel GroupBy

Process each input element in parallel

- grouping ~ shuffling
- input item → output offset such that groups are contiguous

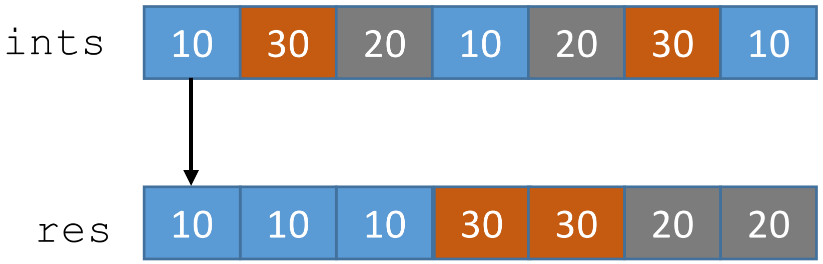


`ints`	10	30	20	10	20	30	10
`res`	10	10	10	30	30	20	20

Parallel GroupBy

Process each input element in parallel

- grouping ~ shuffling
- input item → output offset such that groups are contiguous



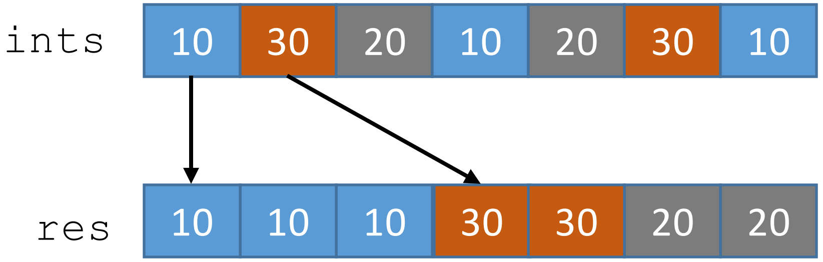
ints 10 30 20 10 20 30 10

res 10 10 10 30 30 20 20

Parallel GroupBy

Process each input element in parallel

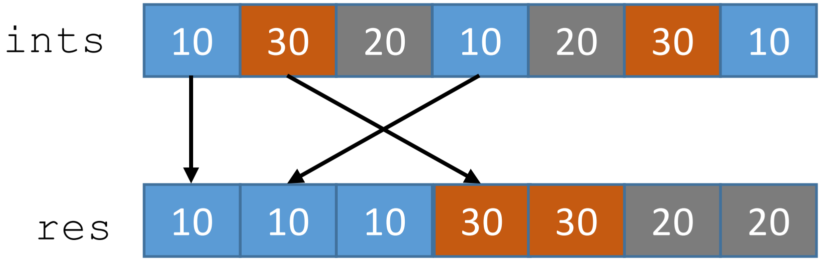
- grouping ~ shuffling
- input item → output offset such that groups are contiguous



Parallel GroupBy

Process each input element in parallel

- grouping ~ shuffling
- input item → output offset such that groups are contiguous

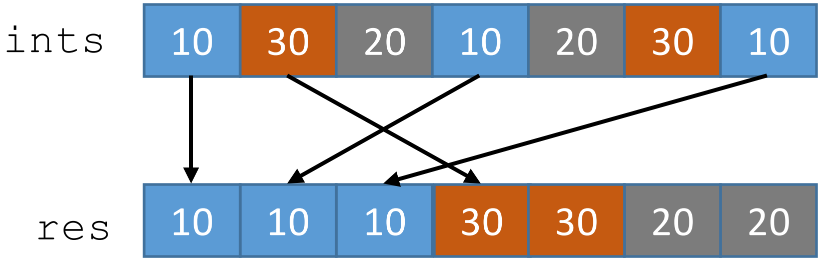


ints	10	30	20	10	20	30	10
res	10	10	10	30	30	20	20

Parallel GroupBy

Process each input element in parallel

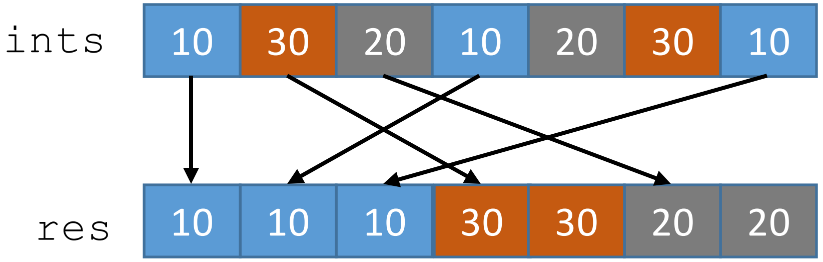
- grouping ~ shuffling
- input item → output offset such that groups are contiguous



Parallel GroupBy

Process each input element in parallel

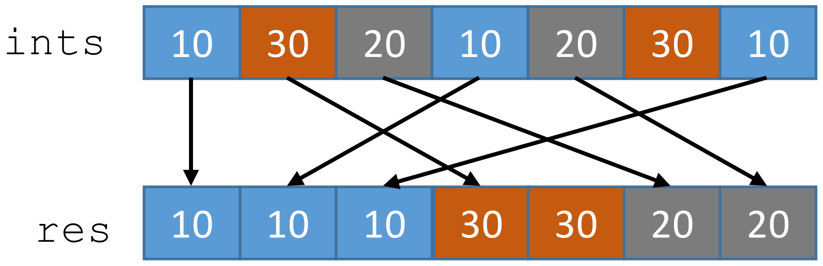
- grouping ~ shuffling
- input item → output offset such that groups are contiguous



Parallel GroupBy

Process each input element in parallel

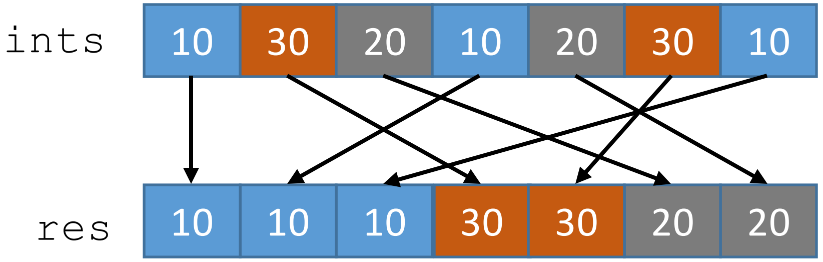
- grouping ~ shuffling
- input item → output offset such that groups are contiguous



Parallel GroupBy

Process each input element in parallel

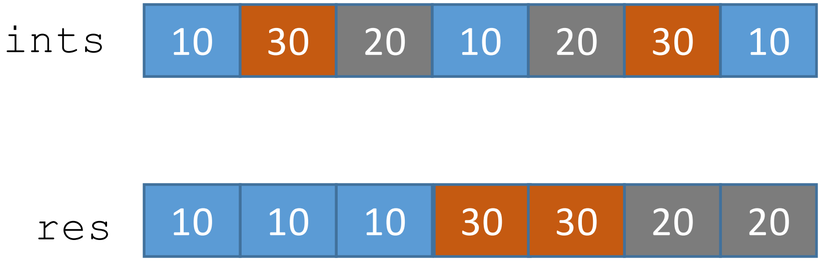
- grouping ~ shuffling
- input item → output offset such that groups are contiguous



Parallel GroupBy

Process each input element in parallel

- grouping ~ shuffling
- input item → output offset such that groups are contiguous

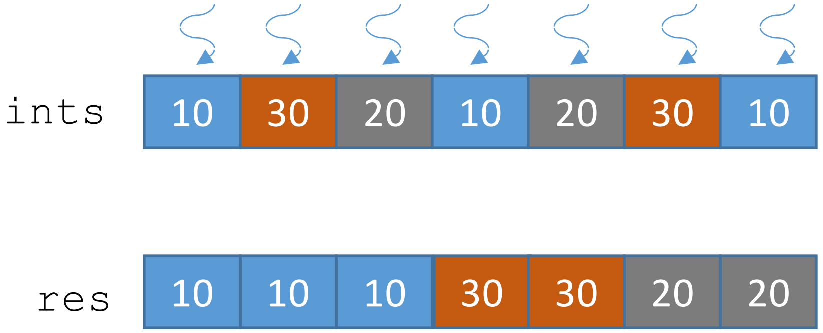


ints	10	30	20	10	20	30	10
res	10	10	10	30	30	20	20

Parallel GroupBy

Process each input element in parallel

- grouping ~ shuffling
- input item → output offset such that groups are contiguous



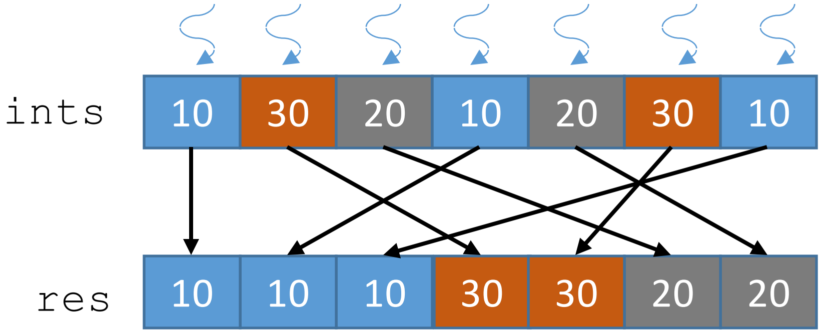
ints 10 30 20 10 20 30 10

res 10 10 10 30 30 20 20

Parallel GroupBy

Process each input element in parallel

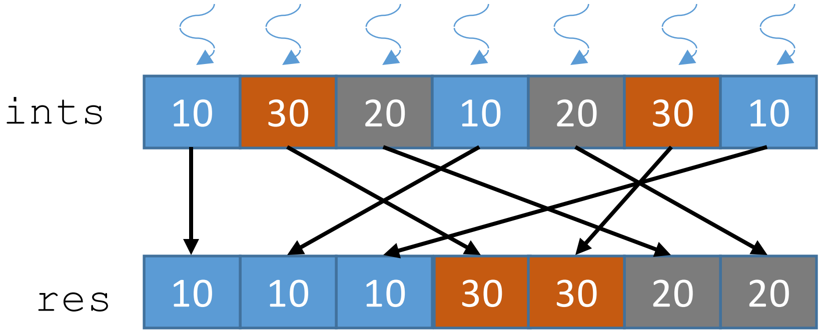
- grouping ~ shuffling
- input item → output offset such that groups are contiguous



Parallel GroupBy

Process each input element in parallel

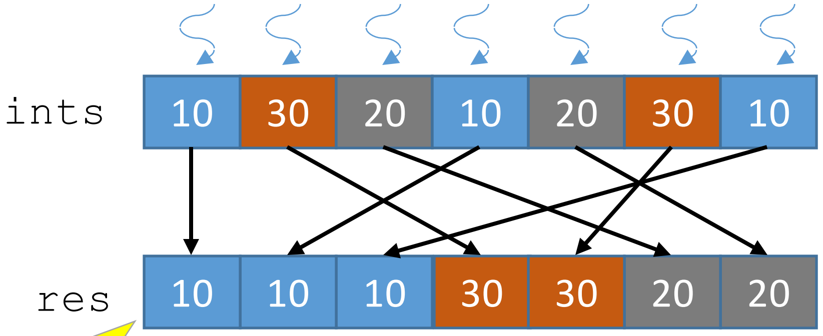
- grouping ~ shuffling
- input item → output offset such that groups are contiguous
- output offset = group offset + item number
- … but how to get the group offset, item number?



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?

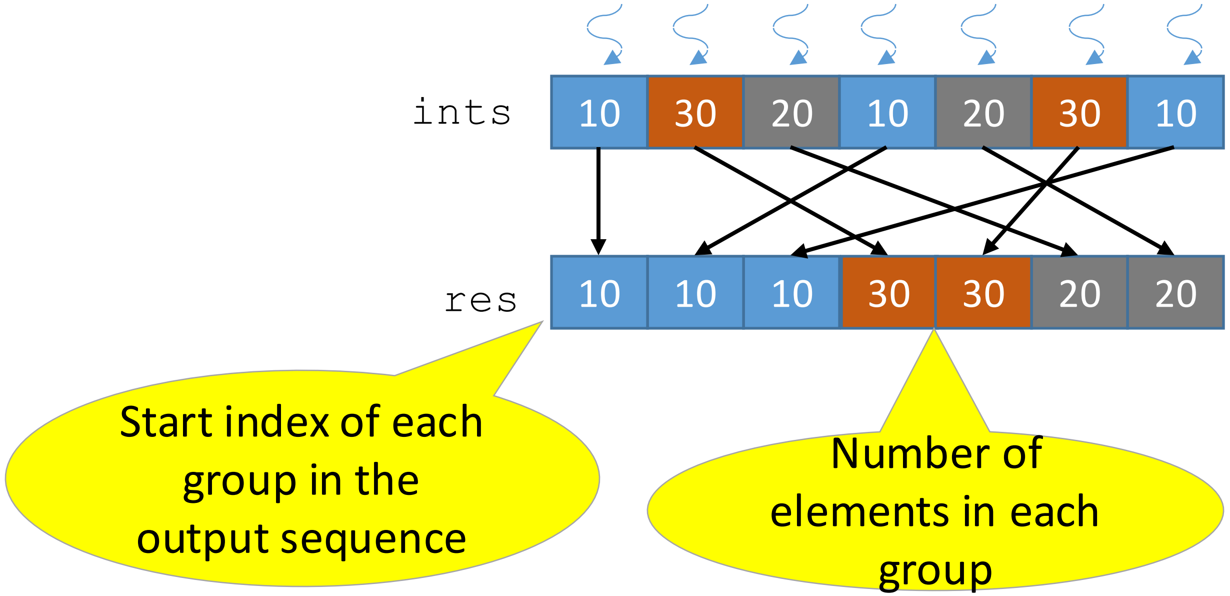


Start index of each group in the output sequence

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?



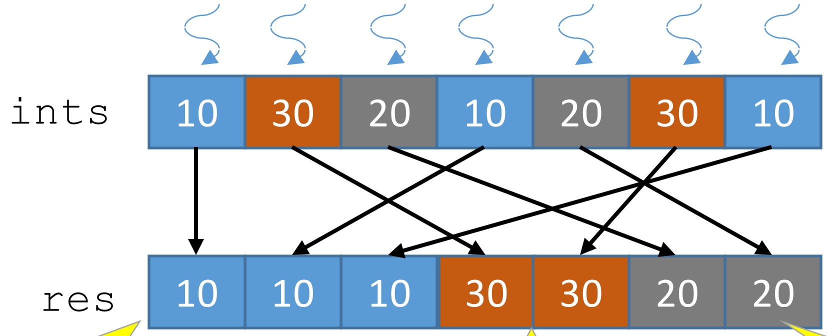
Start index of each group in the output sequence

Number of elements in each group

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?



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

Start index of each group in the output sequence

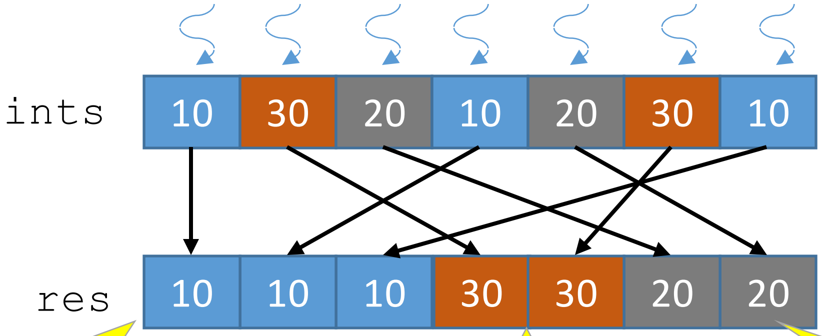
Number of elements in each group

Number of groups and input → group mapping

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?



Start index of each group in the output sequence

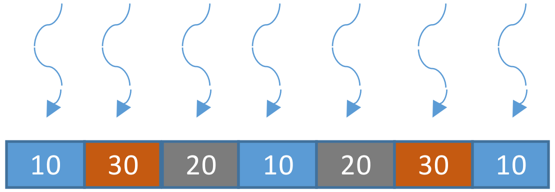
Number of elements in each group

Number of groups and input → group mapping

GroupBy using parallel primitives

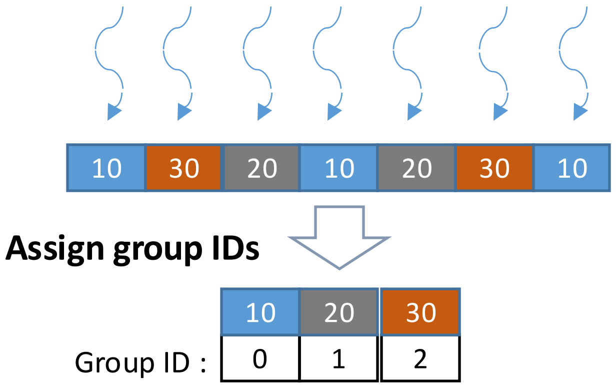
10	30	20	10	20	30	10

GroupBy using parallel primitives



10	30	20	10	20	30	10

GroupBy using parallel primitives

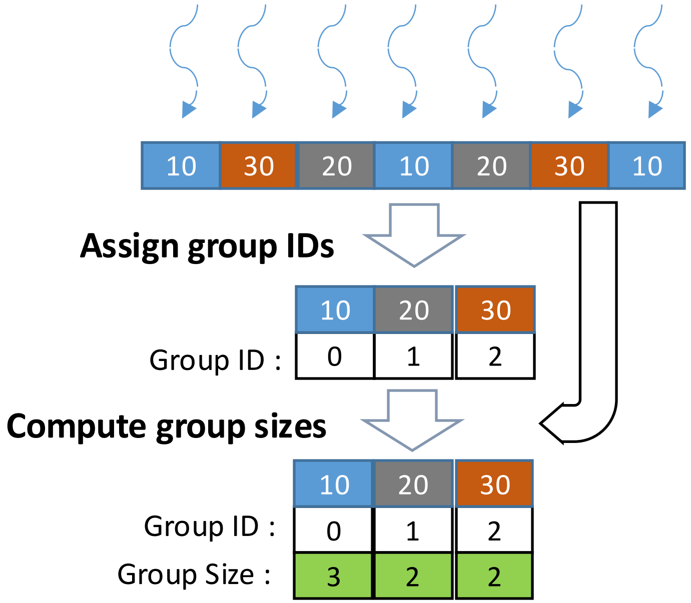


10	30	20	10	20	30	10

Assign group IDs

	10	20	30
Group ID :	0	1	2

GroupBy using parallel primitives



10	30	20	10	20	30	10

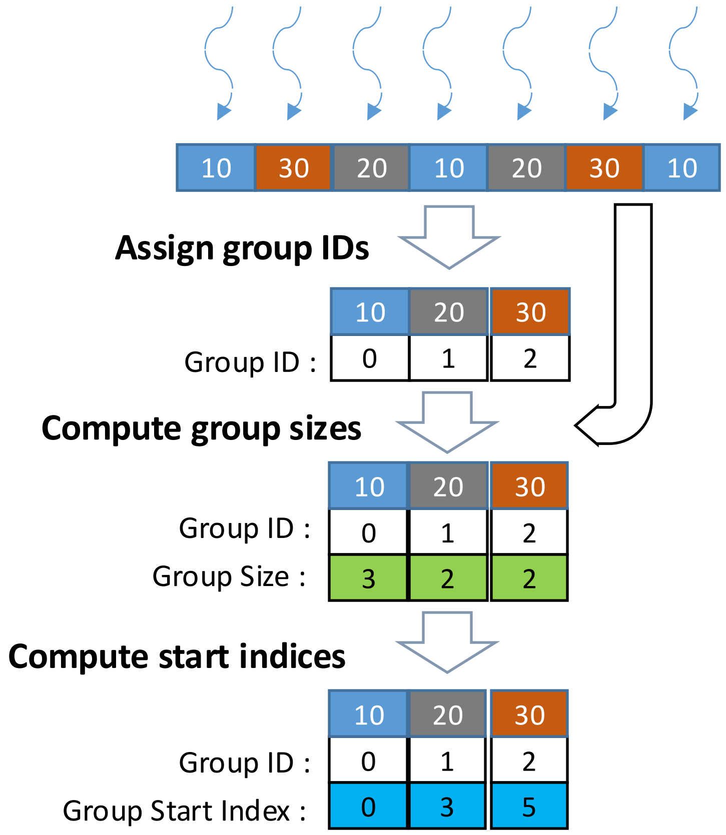
Assign group IDs

	10	20	30
Group ID :	0	1	2

Compute group sizes

	10	20	30
Group ID :	0	1	2
Group Size :	3	2	2

GroupBy using parallel primitives



10	30	20	10	20	30	10

Assign group IDs

	10	20	30
Group ID :	0	1	2

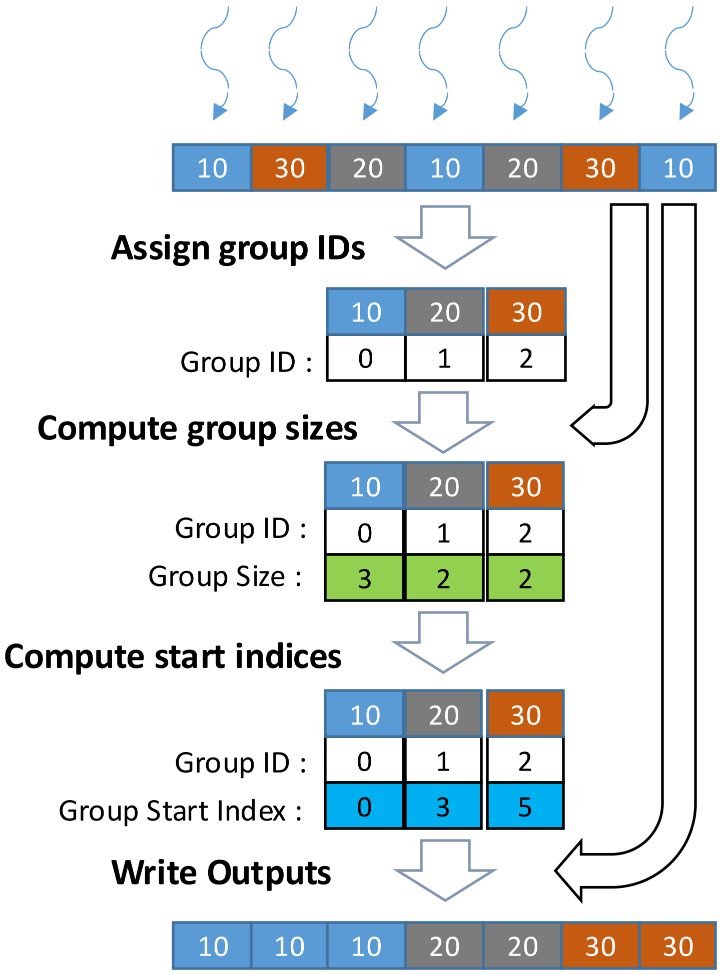
Compute group sizes

	10	20	30
Group ID :	0	1	2
Group Size :	3	2	2

Compute start indices

	10	20	30
Group ID :	0	1	2
Group Start Index :	0	3	5

GroupBy using parallel primitives



10	30	20	10	20	30	10

Assign group IDs

	10	20	30
Group ID :	0	1	2

Compute group sizes

	10	20	30
Group ID :	0	1	2
Group Size :	3	2	2

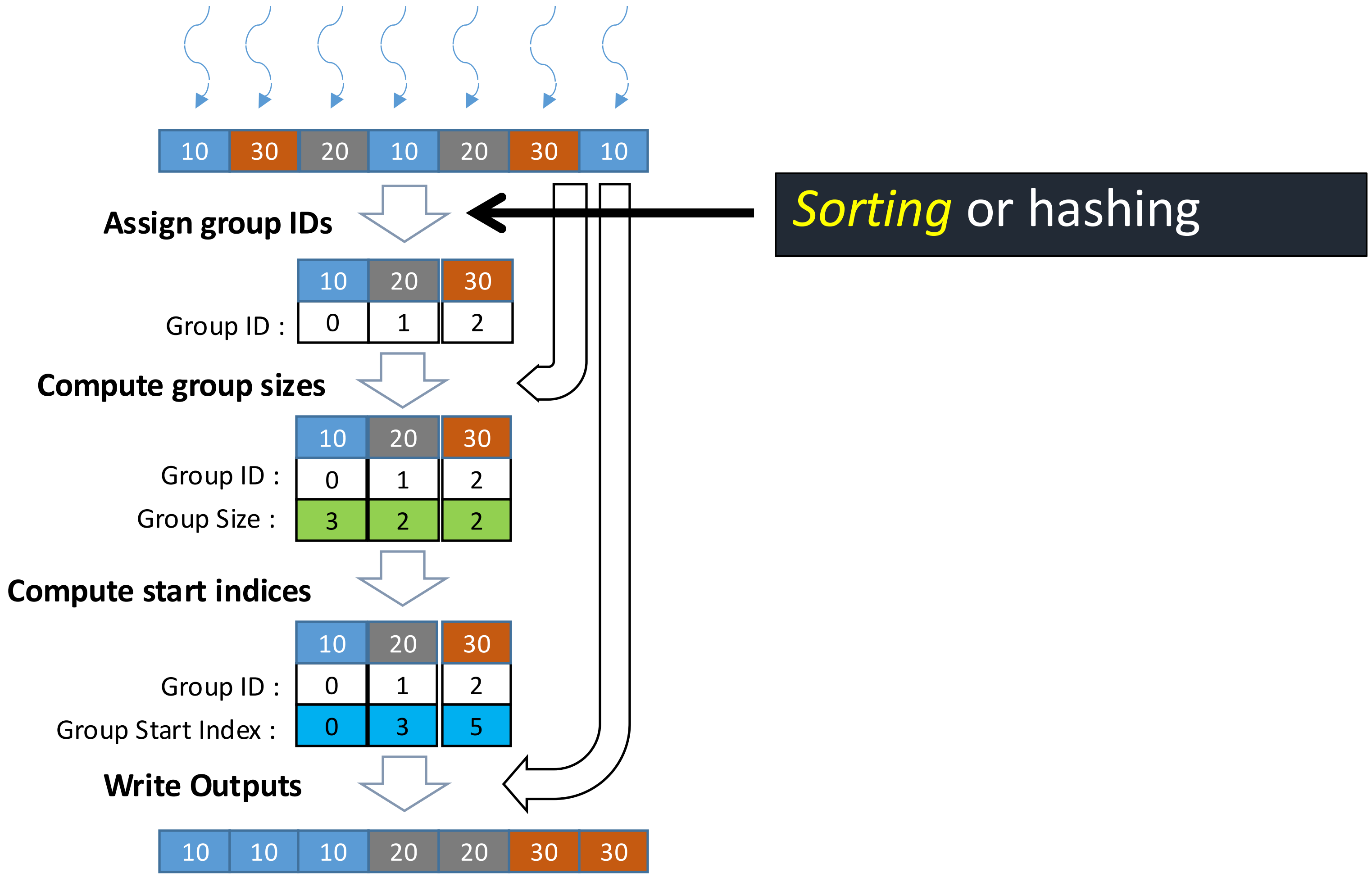
Compute start indices

	10	20	30
Group ID :	0	1	2
Group Start Index :	0	3	5

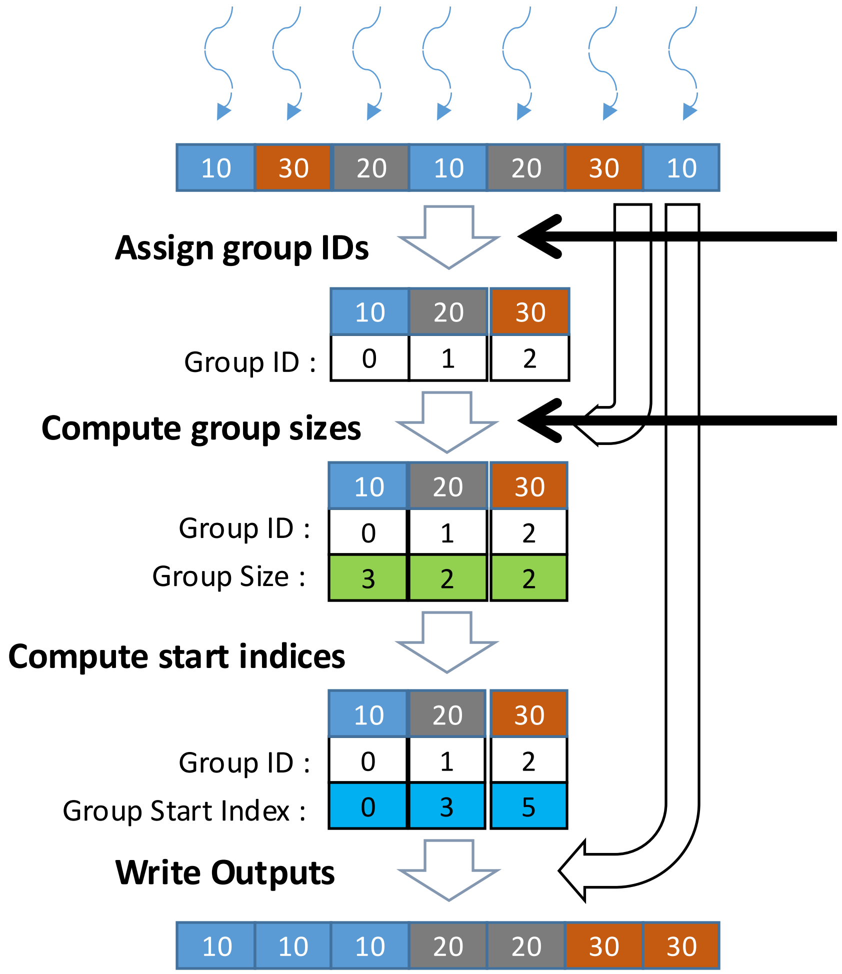
Write Outputs

10	10	10	20	20	30	30

GroupBy using parallel primitives



GroupBy using parallel primitives



10	30	20	10	20	30	10

Assign group IDs

	10	20	30
Group ID :	0	1	2

Compute group sizes

	10	20	30
Group ID :	0	1	2
Group Size :	3	2	2

Compute start indices

	10	20	30
Group ID :	0	1	2
Group Start Index :	0	3	5

Write Outputs

10	10	10	20	20	30	30

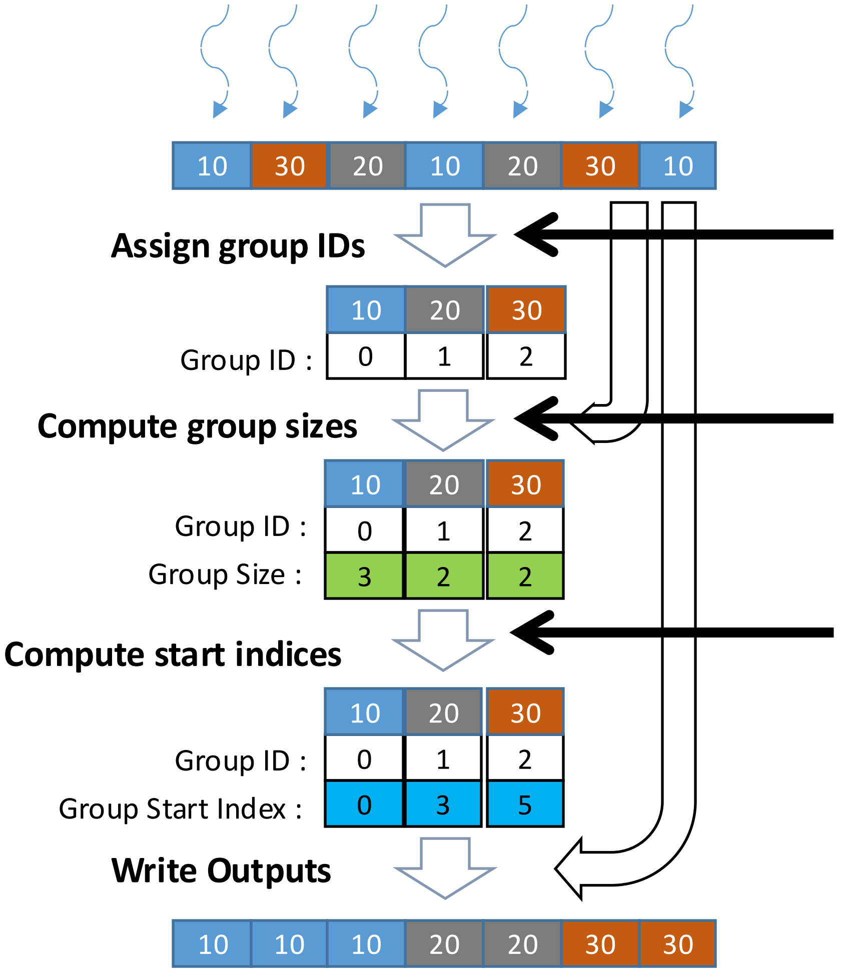
Sorting or hashing

Hash table lookup: group ID

-- Uses atomic increment

-- ***map***

GroupBy using parallel primitives



Sorting or hashing

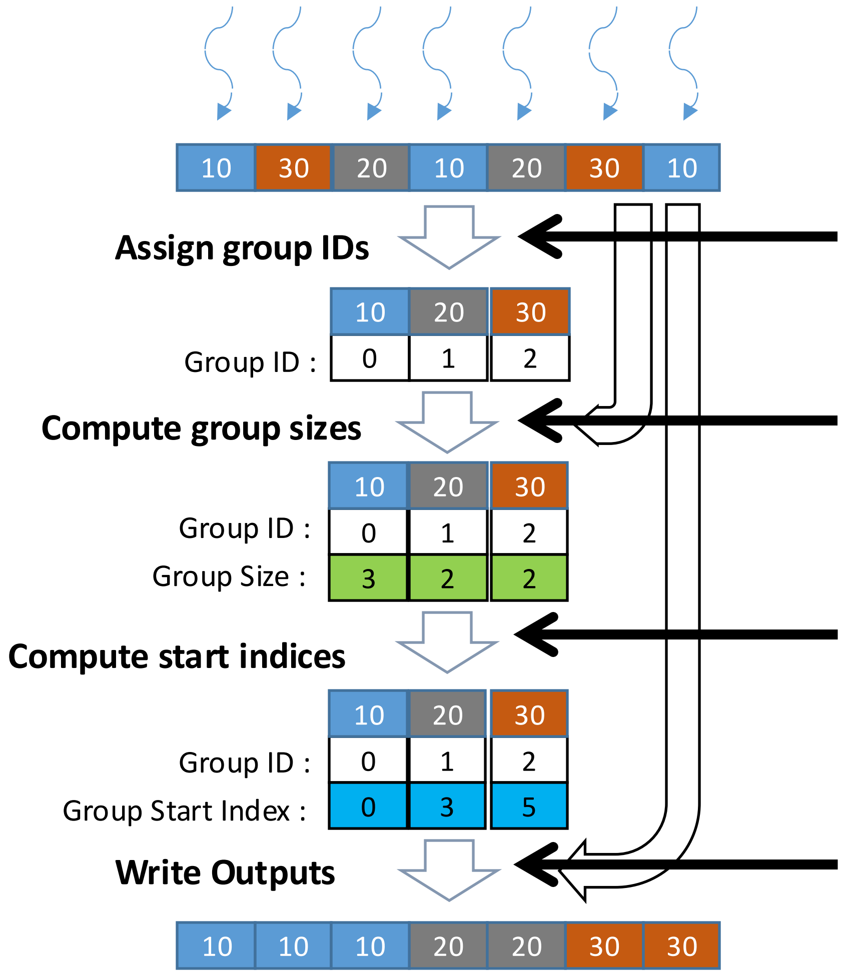
Hash table lookup: group ID

-- Uses atomic increment

-- ***map***

prefix sum of group sizes

GroupBy using parallel primitives



10	30	20	10	20	30	10

Assign group IDs

	10	20	30
Group ID :	0	1	2

Compute group sizes

	10	20	30
Group ID :	0	1	2
Group Size :	3	2	2

Compute start indices

	10	20	30
Group ID :	0	1	2
Group Start Index :	0	3	5

Write Outputs

10	10	10	20	20	30	30

Sorting or hashing

Hash table lookup: group ID

-- Uses atomic increment

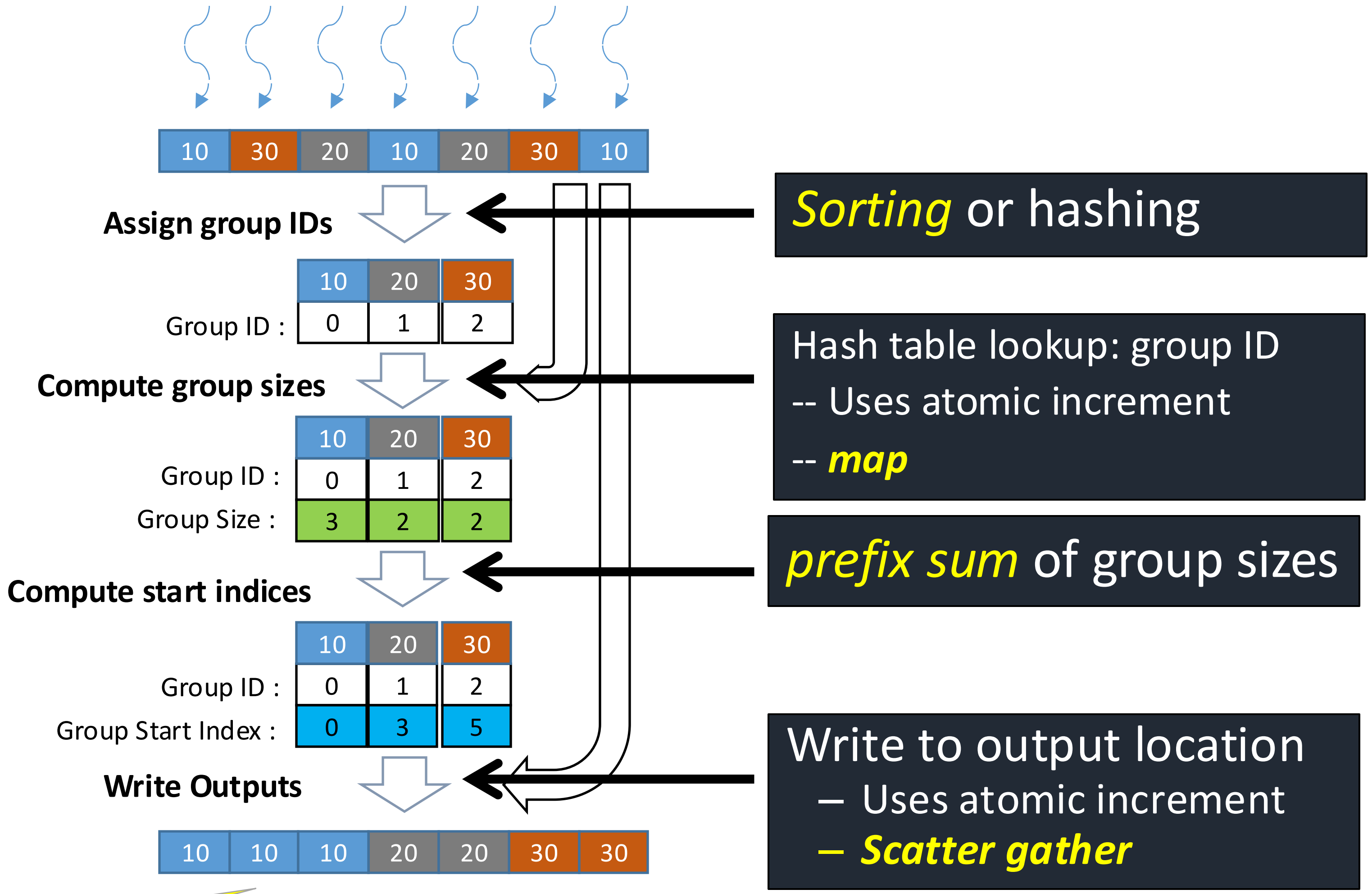
-- ***map***

prefix sum of group sizes

Write to output location

- Uses atomic increment
- ***Scatter gather***

GroupBy using parallel primitives



10	30	20	10	20	30	10

Assign group IDs

	10	20	30
Group ID :	0	1	2

Compute group sizes

	10	20	30
Group ID :	0	1	2
Group Size :	3	2	2

Compute start indices

	10	20	30
Group ID :	0	1	2
Group Start Index :	0	3	5

Write Outputs

10	10	10	20	20	30	30

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…

Sort

Many variations

- Enumeration sort
- Bitonic sort
- Merge sort
- Parallel Quicksort
- Radix sort
- Sample sort
- …

Summary

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

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