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

Programming GPUs
  CUDA
    Basics: getting something working
    Advanced: making it perform
Review
Each SM has multiple vector units (4)
32 lanes wide → warp size
Each SM has multiple vector units (4)
32 lanes wide → warp size
Vector units use *hardware multi-threading*
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 \(\rightarrow\) warp size
Vector units use hardware multi-threading
Execution \(\rightarrow\) a grid of thread blocks (TBs)
Each TB has some number of threads
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
Each SM has multiple vector units (4)  
32 lanes wide $\rightarrow$ warp size  
Vector units use **hardware multi-threading**  
Execution $\rightarrow$ 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....*
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 $f(x)$

$\text{map}(A, f) \rightarrow \text{apply } f(x) \text{ on all elements in } A$

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) \rightarrow apply f(x) on all elements in A

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

```java
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)
Reduce

Input
   - Associative operator \( \text{op} \)
   - Ordered set \( s = [a, b, c, \ldots, z] \)

\( \text{Reduce}(\text{op}, s) \) returns
   \[ a \text{ op } b \text{ op } c \ldots \text{ op } z \]
Reduce

Input
- Associative operator $\text{op}$
- Ordered set $s = \{a, b, c, \ldots, z\}$

$\text{Reduce}(\text{op}, s)$ returns

$$a \text{ op } b \text{ op } c \ldots \text{ op } z$$

for ($i = 0; i < N; ++i$) {
    $\text{accum } += (\text{point}[i] \times \text{point}[i])$
}
Reduce

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

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

\[ \text{accum} = \text{reduce}(*, \ \text{point}) \]

Why must \( op \) be associative?
Reduce

Input

Associative operator \( \text{op} \)

Ordered set \( s = [a, b, c, \ldots z] \)

Reduce(\( \text{op} \), \( s \)) returns

\[ a \text{ op } b \text{ op } c \ldots \text{ op } z \]

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

\[
\text{accum} = \text{reduce}(*, \text{point})
\]

Why must \( \text{op} \) be associative?
Scan (Prefix Sum)

Input
- Associative operator $\textbf{op}$
- Ordered set $s = [a, b, c, \ldots, z]$
- Identity $I$

$\text{scan}(op, s) = [I, a, (a \textbf{ op } b), (a \textbf{ op } b \textbf{ op } c) \ldots]$

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 $\rightarrow$ key
Example: Parallel GroupBy

Group a collection by key
Lambda function maps elements → key

\[ \text{var res} = \text{ints.GroupBy}(x \mapsto x); \]
Example: Parallel GroupBy

Group a collection by key
Lambda function maps elements → key

```csharp
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 $\rightarrow$ key

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

![Diagram showing the grouping process](image-url)
Example: Parallel GroupBy

Group a collection by key
Lambda function maps elements $\rightarrow$ key

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

```csharp
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
Parallel GroupBy

Process each input element in parallel
- grouping ~ shuffling
- input item $\rightarrow$ 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 $\rightarrow$ 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

```
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 $\rightarrow$ output offset such that groups are contiguous
Parallel GroupBy

Process each input element in parallel
  grouping ~ shuffling
  input item \rightarrow output offset such that groups are contiguous
Parallel GroupBy

Process each input element in parallel

grouping ~ shuffling
input item $\rightarrow$ output offset such that groups are contiguous
Parallel GroupBy

Process each input element in parallel
  grouping ~ shuffling
  input item \rightarrow 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
```
Parallel GroupBy

Process each input element in parallel

- grouping ~ shuffling
- input item $\rightarrow$ 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?

![Diagram showing an example of Parallel GroupBy]

- **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**
Parallel GroupBy

Process each input element in parallel

- grouping ~ shuffling
- input item \( \rightarrow \) output offset such that groups are contiguous
- output offset = group offset + item number
- ... but how to get the group offset, item number?

<table>
<thead>
<tr>
<th>ints</th>
<th>10</th>
<th>30</th>
<th>20</th>
<th>10</th>
<th>20</th>
<th>30</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>res</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>30</td>
<td>30</td>
<td>20</td>
<td>20</td>
</tr>
</tbody>
</table>

Start index of each group in the output sequence

Number of elements in each group

Number of groups and input \( \rightarrow \) 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?

```
ints  |  res
---   |---
10    | 10
30    | 10
20    | 30
10    | 30
20    | 20
30    | 20
10    | 10
```

- Start index of each group in the output sequence
- Number of elements in each group
- Number of groups and input → group mapping
GroupBy with Parallel Primitives

10  30  20  10  20  30  10
GroupBy with Parallel Primitives
GroupBy with Parallel Primitives

Assign group IDs

<table>
<thead>
<tr>
<th>Group ID</th>
<th>10</th>
<th>20</th>
<th>30</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>
**GroupBy with Parallel Primitives**

Assign group IDs

- Group ID:
  - 0
  - 1
  - 2

Compute group sizes

- Group ID:
  - 0
  - 1
  - 2
- Group Size:
  - 3
  - 2
  - 2
GroupBy with Parallel Primitives

<table>
<thead>
<tr>
<th></th>
<th>10</th>
<th>20</th>
<th>30</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>10</td>
<td>20</td>
<td>30</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Assign group IDs

<table>
<thead>
<tr>
<th>Group ID :</th>
<th>0</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
</table>

Compute group sizes

<table>
<thead>
<tr>
<th>Group ID :</th>
<th>0</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Group Size :</td>
<td>3</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

Compute start indices

<table>
<thead>
<tr>
<th>Group ID :</th>
<th>0</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Group Start Index :</td>
<td>0</td>
<td>3</td>
<td>5</td>
</tr>
</tbody>
</table>
GroupBy with Parallel Primitives

Assign group IDs

Group ID:

Group Size:

Compute group sizes

Group ID:

Group Size:

Compute start indices

Group ID:

Group Start Index:

Write Outputs

Write Outputs
GroupBy with Parallel Primitives

Assign group IDs

Compute group sizes

Compute start indices

Write Outputs

Sorting or hashing
GroupBy with Parallel Primitives

Assign group IDs

Group ID:
- 0
- 1
- 2

Group Size:
- 3
- 2
- 2

Sorting or hashing

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

Compute group sizes

Group ID:
- 0
- 1
- 2

Group Size:
- 3
- 2
- 2

Compute start indices

Group ID:
- 0
- 1
- 2

Group Start Index:
- 0
- 3
- 5

Write Outputs

- 10 10 10 20 20 30 30
GroupBy with Parallel Primitives

Assign group IDs

Compute group sizes

Compute start indices

Write Outputs

Sorting or hashing

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

prefix sum of group sizes
GroupBy with Parallel Primitives

1. **Assign group IDs**
   - Group ID: 0 1 2 2 2
   - Group Size: 3 2 2

2. **Compute group sizes**
   - Group ID: 0 1 2 2 2
   - Group Size: 3 2 2

3. **Compute start indices**
   - Group ID: 0 1 2 2 2
   - Group Start Index: 0 3 5

4. **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 with Parallel Primitives

**Assign group IDs**

<table>
<thead>
<tr>
<th>Group ID</th>
<th>Group Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>3</td>
</tr>
<tr>
<td>20</td>
<td>2</td>
</tr>
<tr>
<td>30</td>
<td>3</td>
</tr>
</tbody>
</table>

**Compute group sizes**

<table>
<thead>
<tr>
<th>Group ID</th>
<th>Group Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>3</td>
</tr>
<tr>
<td>20</td>
<td>2</td>
</tr>
<tr>
<td>30</td>
<td>3</td>
</tr>
</tbody>
</table>

**Compute start indices**

<table>
<thead>
<tr>
<th>Group ID</th>
<th>Group Start Index</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>20</td>
<td>3</td>
</tr>
<tr>
<td>30</td>
<td>5</td>
</tr>
</tbody>
</table>

**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...
Parallel Patterns
Parallel Patterns

Thrust:
Large set of algorithms
~75 functions
~125 variations

Flexible
User-defined types
User-defined operators

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>reduce</td>
<td>Sum of a sequence</td>
</tr>
<tr>
<td>find</td>
<td>First position of a value in a sequence</td>
</tr>
<tr>
<td>mismatch</td>
<td>First position where two sequences differ</td>
</tr>
<tr>
<td>inner_product</td>
<td>Dot product of two sequences</td>
</tr>
<tr>
<td>equal</td>
<td>Whether two sequences are equal</td>
</tr>
<tr>
<td>min_element</td>
<td>Position of the smallest value</td>
</tr>
<tr>
<td>count</td>
<td>Number of instances of a value</td>
</tr>
<tr>
<td>is_sorted</td>
<td>Whether sequence is in sorted order</td>
</tr>
<tr>
<td>transform_reduce</td>
<td>Sum of transformed sequence</td>
</tr>
</tbody>
</table>
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_malloc`, `scalable_allocator`, `cache_aligned_allocator`
- Mutual exclusion: `mutex`, `spin_mutex`, `queuing_mutex`, `spin_rwlock_mutex`, `queuing_rwlock_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?