Prefix sums on GPUs

Bruce Merry

Department of Computer Science, University of Cape Town

GPGPU2 Workshop 2014
Outline

1. Definition and Applications
   - Motivating Problem
   - Definitions
   - Other Applications

2. Parallel Algorithms
   - Kogge-Stone
   - Brent-Kung

3. GPU Strategies
   - Reduce-then-Scan
   - Two-Level Prefix Sum
Outline

1 Definition and Applications
   - Motivating Problem
   - Definitions
   - Other Applications

2 Parallel Algorithms
   - Kogge-Stone
   - Brent-Kung

3 GPU Strategies
   - Reduce-then-Scan
   - Two-Level Prefix Sum
Problem Statement

For every object in a set, output a list of the other objects that differ by less than some amount.

This is deliberately vague: could be for n-body simulation, clustering, scattered data interpolation.
Problem Statement

For every object in a set, output a list of the other objects that differ by less than some amount. This is deliberately vague: could be for n-body simulation, clustering, scattered data interpolation.
The lists should be packed together contiguously.

Assuming one workitem per object, how do the workitems know where to start?
The lists should be packed together contiguously.

| A0 | A1 | A2 | B0 | B1 |

Assuming one workitem per object, how do the workitems know where to start?
The lists should be packed together contiguously.

Assuming one workitem per object, how do the workitems know where to start?
The lists should be packed together contiguously.

| A0 | A1 | A2 | B0 | B1 | D0 | E0 | E1 |

Assuming one workitem per object, how do the workitems know where to start?
The lists should be packed together contiguously.

| A0 | A1 | A2 | B0 | B1 | D0 | E0 | E1 |

Assuming one workitem per object, how do the workitems know where to start?
This can be solved with a multi-pass approach:

1. Every workitem counts how many records to emit, and writes this number to a buffer.

2. The buffer is processed to determine the start position for each object, and writes this position to a buffer.

3. Each workitem reads this buffer, and emits its records in the right place.
This can be solved with a multi-pass approach:

1. Every workitem counts how many records to emit, and writes this number to a buffer.

2. The buffer is processed to determine the start position for each object, and writes this position to a buffer.

3. Each workitem reads this buffer, and emits its records in the right place.
This can be solved with a multi-pass approach:

1. Every workitem counts how many records to emit, and writes this number to a buffer.
2. The buffer is processed to determine the start position for each object, and writes this position to a buffer.
3. Each workitem reads this buffer, and emits its records in the right place.
Outline

1 Definition and Applications
   - Motivating Problem
   - Definitions
   - Other Applications

2 Parallel Algorithms
   - Kogge-Stone
   - Brent-Kung

3 GPU Strategies
   - Reduce-then-Scan
   - Two-Level Prefix Sum

Prefix sums on GPUs
Bruce Merry
Definition and Applications
Motivating Problem
Definitions
Other Applications
Parallel Algorithms
Kogge-Stone
Brent-Kung
GPU Strategies
Reduce-then-Scan
Two-Level Prefix Sum
Summary
Exclusive Prefix Sum

Given an operator $\oplus$ and an identity element $I$, the *exclusive prefix sum* of $(a_0, a_1, \ldots, a_{n-1})$ is

$$(I, a_0, a_0 \oplus a_1, a_0 \oplus a_1 \oplus a_2, \ldots, a_0 \oplus \cdots \oplus a_{n-2}) = \left( \bigoplus_{j=0}^{i-1} a_j \right)$$

In other words, element $i$ is the sum of all elements strictly before $i$. 

\[
\begin{align*}
4 & \quad 3 & \quad 7 & \quad 9 & \quad 2 & \quad 3 \\
0 & \quad 4 & \quad 7 & \quad 14 & \quad 23 & \quad 25
\end{align*}
\]
Exclusive Prefix Sum

Given an operator $\oplus$ and an identity element $I$, the exclusive prefix sum of $(a_0, a_1, \ldots, a_{n-1})$ is

$$(I, a_0, a_0 \oplus a_1, a_0 \oplus a_1 \oplus a_2, \ldots, a_0 \oplus \cdots \oplus a_{n-2}) = \left( \bigoplus_{j=0}^{i-1} a_j \right)$$

In other words, element $i$ is the sum of all elements strictly before $i$. 

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>3</td>
<td>7</td>
<td>9</td>
<td>2</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>4</td>
<td>7</td>
<td>14</td>
<td>23</td>
<td>25</td>
<td></td>
</tr>
</tbody>
</table>
Inclusive Prefix Sum

Given an operator $\oplus$ and an identity element $I$, the inclusive prefix sum of $(a_0, a_1, \ldots, a_{n-1})$ is

$$(a_0, a_0 \oplus a_1, a_0 \oplus a_1 \oplus a_2, \ldots, a_0 \oplus \cdots \oplus a_{n-1}) = \left( \bigoplus_{j=0}^{i} a_j \right)$$

In other words, element $i$ is the sum of all elements before and including $i$.
Inclusive Prefix Sum

Given an operator $\oplus$ and an identity element $I$, the *inclusive prefix sum* of $(a_0, a_1, \ldots, a_{n-1})$ is

$$(a_0, a_0 \oplus a_1, a_0 \oplus a_1 \oplus a_2, \ldots, a_0 \oplus \cdots \oplus a_{n-1}) = \left( \bigoplus_{j=0}^{i} a_j \right)$$

In other words, element $i$ is the sum of all elements before and including $i$. 

\[
\begin{array}{ccccccc}
4 & 3 & 7 & 9 & 2 & 3 \\
4 & 7 & 14 & 23 & 25 & 28
\end{array}
\]
Outline

1 Definition and Applications
   - Motivating Problem
   - Definitions
   - Other Applications

2 Parallel Algorithms
   - Kogge-Stone
   - Brent-Kung

3 GPU Strategies
   - Reduce-then-Scan
   - Two-Level Prefix Sum

Prefix sums on GPUs
Bruce Merry

Definition and Applications
Motivating Problem
Definitions
Other Applications

Parallel Algorithms
Kogge-Stone
Brent-Kung

GPU Strategies
Reduce-then-Scan
Two-Level Prefix Sum

Summary
Other Applications

- Compaction: select all objects that satisfy a predicate
- Partitioning: rearrange objects that satisfy a predicate before the others
- Sorting: radix sort is just repeated partitioning
- Visibility: an object is visible if it is not preceded by a taller one (using max operator instead of +)
- Meshing: each cell produces an variable number of triangles
Other Applications

- Compaction: select all objects that satisfy a predicate
- Partitioning: rearrange objects that satisfy a predicate before the others
  - Sorting: radix sort is just repeated partitioning
  - Visibility: an object is visible if it is not preceded by a taller one (using max operator instead of +)
  - Meshing: each cell produces an variable number of triangles
Other Applications

- Compaction: select all objects that satisfy a predicate
- Partitioning: rearrange objects that satisfy a predicate before the others
- Sorting: radix sort is just repeated partitioning
- Visibility: an object is visible if it is not preceded by a taller one (using max operator instead of +)
- Meshing: each cell produces an variable number of triangles
Other Applications

- Compaction: select all objects that satisfy a predicate
- Partitioning: rearrange objects that satisfy a predicate before the others
- Sorting: radix sort is just repeated partitioning
- Visibility: an object is visible if it is not preceded by a taller one (using max operator instead of +)
- Meshing: each cell produces an variable number of triangles
Other Applications

- Compaction: select all objects that satisfy a predicate
- Partitioning: rearrange objects that satisfy a predicate before the others
- Sorting: radix sort is just repeated partitioning
- Visibility: an object is visible if it is not preceded by a taller one (using max operator instead of +)
- Meshing: each cell produces an variable number of triangles
Atomics offer an alternative way to allocate unique memory per work-item, but

- Suffer heavy contention, which is slow (but getting better all the time)
- Do not preserve the original ordering
- Do not give reproducible ordering

Atomics have the advantage of allowing for single-pass algorithms.
Outline

1. Definition and Applications
   - Motivating Problem
   - Definitions
   - Other Applications

2. Parallel Algorithms
   - Kogge-Stone
   - Brent-Kung

3. GPU Strategies
   - Reduce-then-Scan
   - Two-Level Prefix Sum

Prefix sums on GPUs
Bruce Merry
Definition and Applications
Motivating Problem
Definitions
Other Applications
Parallel Algorithms
Kogge-Stone
Brent-Kung
GPU Strategies
Reduce-then-Scan
Two-Level Prefix Sum
Summary
Idea

Let $s_i^t$ be the sum of the (up to) $t$ inputs ending with $a_i$. Then

$$s_i^{2t} = s_{i-t}^t \oplus s_i^t.$$ 

We start with $(s_i^1) = (a_i)$, then compute $(s_i^2), (s_i^4), (s_i^8)$ and so on, up to $(s_i^N)$, in $O(\log_2 N)$ iterations, to give an inclusive prefix sum.
Example
Pseudo-code

```plaintext
foreach power-of-two \( t \) from 1 to \( N \) do
    for \( i \leftarrow t \) to \( N - 1 \) do in parallel
        \( a_i \leftarrow a_{i-t} \oplus a_i; \)
```
Work-item Pseudo-code

\[
i \leftarrow \text{workitem ID}; \\
\text{foreach power-of-two } t \text{ from 1 to } N \text{ do} \\
\hspace{1em} x \leftarrow a_i; \\
\hspace{1em} \text{if } t \leq i \text{ then} \\
\hspace{2em} x \leftarrow x \oplus a_{i-t}; \\
\hspace{1em} \text{barrier}(); \\
\hspace{1em} a_i \leftarrow x; \\
\hspace{1em} \text{barrier}();
\]

Prefix sums on GPUs
Bruce Merry

Definition and Applications
Motivating Problem
Definitions
Other Applications

Parallel Algorithms
Kogge-Stone
Brent-Kung

GPU Strategies
Reduce-then-Scan
Two-Level Prefix Sum

Summary
The working register $x$ can be reused between loop iterations without reloading.

The `if` statement can be eliminated by padding at the front with zeros.

Shared memory can be used to reduce global memory accesses.
Optimizations

- The working register $x$ can be reused between loop iterations without reloading.
- The if statement can be eliminated by padding at the front with zeros.
- Shared memory can be used to reduce global memory accesses.
Optimizations

- The working register $x$ can be reused between loop iterations without reloading.
- The if statement can be eliminated by padding at the front with zeros.
- Shared memory can be used to reduce global memory accesses.
Properties

- It is **work-inefficient**: it performs $O(N \log N)$ operations in total
  - About $2 \log_2 N$ barriers
  - About $N \log N$ reads and $N \log N$ writes
- Memory access pattern is good: sequential accesses
Properties

- It is *work-inefficient*: it performs $O(N \log N)$ operations in total.
- About $2 \log_2 N$ barriers.
- About $N \log N$ reads and $N \log N$ writes.
- Memory access pattern is good: sequential accesses.
Properties

- It is *work-inefficient*: it performs $O(N \log N)$ operations in total.
- About $2 \log_2 N$ barriers.
- About $N \log N$ reads and $N \log N$ writes.
- Memory access pattern is good: sequential accesses.
Properties

- It is work-inefficient: it performs $O(N \log N)$ operations in total
- About $2 \log_2 N$ barriers
- About $N \log N$ reads and $N \log N$ writes
- Memory access pattern is good: sequential accesses
Outline

1. Definition and Applications
   - Motivating Problem
   - Definitions
   - Other Applications

2. Parallel Algorithms
   - Kogge-Stone
   - Brent-Kung

3. GPU Strategies
   - Reduce-then-Scan
   - Two-Level Prefix Sum
For an exclusive scan:

- Add pairs of adjacent elements:
  \[ p_i = a_{2i} \oplus a_{2i+1} \]

- Recursively scan these sums:
  \[ q_i = \bigoplus_{j=0}^{2i-1} p_i = \bigoplus_{j=0}^{i-1} a_j \]

- Use these sums to compute the result:
  \[ s_{2i} = q_i, \quad s_{2i+1} = q_i \oplus a_{2i} \]
Example
Example
Example
Example

Prefix sums
on GPUs

Bruce Merry

Definition and
Applications
Motivating Problem
Definitions
Other Applications

Parallel
Algorithms
Kogge-Stone
Brent-Kung

GPU
Strategies
Reduce-then-Scan
Two-Level Prefix
Sum

Summary
Example
Memory Arrangement

**In-place** Each sum replaces the second element of the pair being summed. No extra memory, but has bad bank conflicts.

**Out-of-place** Each level of the tree stored contiguously. Requires double the memory, but conflicts are only 2-way.
In-place
Each sum replaces the second element of the pair being summed. No extra memory, but has bad bank conflicts.

Out-of-place
Each level of the tree stored contiguously. Requires double the memory, but conflicts are only 2-way.
Pseudo-code

Out-of-place exclusive sum, for $N = 2^n$:

Copy $a_i$ to $b_{i+N}$ for $i \in [0, N)$;

\begin{verbatim}
for t ← n − 1 downto 1 do
    for i ← 0 to $2^t − 1$ do in parallel
        $b_{2^t+i} ← b_{2^{t+1}+i} \oplus b_{2^{t+1}+i+1}$;
// Exclusive prefix sum of two elements
$b_3 ← b_2$;
b_2 ← i;
for t ← 1 to n − 1 do
    for i ← 0 to $2^t − 1$ do in parallel
        $b_{2^{t+1}+i+1} ← b_{2^t+i} \oplus b_{2^{t+1}+i}$;
        $b_{2^{t+1}+i} ← b_{2^t+i}$;
Copy $b_{i+N}$ to $s_i$ for $i \in [0, N)$;
\end{verbatim}
Per-workitem Pseudo-code

Uses $\frac{N}{2}$ work-items:

\begin{verbatim}
Uses $\frac{N}{2}$ work-items:
\begin{verbatim}
\begin{align*}
  i &\leftarrow \text{work-item ID}; \\
  b_{i+N} &\leftarrow a_i; \\
  b_{i+N+N/2} &\leftarrow a_{i+N/2}; \\
  \text{barrier}(); \\
  \text{for } t \leftarrow n - 1 \text{ downto } 1 \text{ do} \\
    &\text{ if } i < 2^t \text{ then} \\
    &\quad b_{2^t+i} \leftarrow b_{2^t+1+2i} \oplus b_{2^t+1+2i+1}; \\
    &\quad \text{barrier}(); \\
  \end{align*}
\end{verbatim}
\end{verbatim}

\begin{verbatim}
\begin{align*}
  \text{if } i = 0 \text{ then} \\
    &a_3 \leftarrow a_2; \\
    &a_2 \leftarrow i; \\
    \text{barrier}(); \\
  \text{for } t \leftarrow 1 \text{ to } n - 1 \text{ do} \\
    &\text{ if } i < 2^t \text{ then} \\
    &\quad b_{2^t+1+2i+1} \leftarrow b_{2^t+i} \oplus b_{2^t+1+2i}; \\
    &\quad b_{2^t+1+2i} \leftarrow b_{2^t+i}; \\
    &\quad \text{barrier}(); \\
  \end{align*}
\end{verbatim}

\begin{verbatim}
\begin{align*}
  s_i &\leftarrow b_{i+N}; \\
  s_{i+N/2} &\leftarrow b_{i+N+N/2}; \\
\end{align*}
\end{verbatim}
\end{verbatim}
Properties

- Work-efficient: $O(N)$ addition operations
- Still requires about $2 \log_2 N$ barriers
- Requires about $4N$ reads and $3N$ writes
- Only $\frac{N}{2}$ work-items required
- Has branching, but it is coherent
Properties

- Work-efficient: $O(N)$ addition operations
- Still requires about $2 \log_2 N$ barriers
  - Requires about $4N$ reads and $3N$ writes
  - Only $\frac{N}{2}$ work-items required
- Has branching, but it is coherent
Properties

- Work-efficient: $O(N)$ addition operations
- Still requires about $2 \log_2 N$ barriers
- Requires about $4N$ reads and $3N$ writes
- Only $\frac{N}{2}$ work-items required
- Has branching, but it is coherent
Properties

- Work-efficient: $O(N)$ addition operations
- Still requires about $2 \log_2 N$ barriers
- Requires about $4N$ reads and $3N$ writes
- Only $\frac{N}{2}$ work-items required
- Has branching, but it is coherent
Properties

- Work-efficient: $O(N)$ addition operations
- Still requires about $2 \log_2 N$ barriers
- Requires about $4N$ reads and $3N$ writes
- Only $\frac{N}{2}$ work-items required
- Has branching, but it is coherent
Motivation

Applying one of these at a larger (multi-workgroup) scale has issues:

- Synchronisation: no inter-workgroup synchronisation, so barriers must be kernel-instance boundaries
- Memory usage: need $O(N)$ working space
- Bandwidth: requires $O(N \log N)$ for Kogge-Stone, about $7N$ for Brent-Kung
Outline

1. Definition and Applications
   - Motivating Problem
   - Definitions
   - Other Applications

2. Parallel Algorithms
   - Kogge-Stone
   - Brent-Kung

3. GPU Strategies
   - Reduce-then-Scan
   - Two-Level Prefix Sum
Generalizing Brent-Kung

The Brent-Kung tree doesn’t have to be binary:
The Brent-Kung tree doesn’t have to be binary:
Generalizing Brent-Kung

The Brent-Kung tree doesn’t have to be binary:
The Brent-Kung tree doesn’t have to be binary:
Generalizing Brent-Kung

The Brent-Kung tree doesn’t have to be binary:

```
  0
 /|
/  |
6  9
 /|
6  2 3
 /|
6  1 3
   /|
  3 3 3
```

The numbers in the tree represent the values at each node, and the tree structure shows how these values are combined. The Brent-Kung tree can be generalized to non-binary trees, providing flexibility in parallel algorithms.
Generalizing Brent-Kung

The Brent-Kung tree doesn’t have to be binary:

```
0
/     \
/      \
0      20
|      /
|     /  \
6     15  20
|     |    |
17    26   27
|    |    |
20   33   36
|  |  |  |
6   39  40
|  |  |
36  45  49
|  |  |
49  58  65
|  |
58  68
```
Reduce-then-Scan strategy

1. Divide elements into blocks of size $M$.
2. Use a workgroup per block to compute sum of each block.
3. Recursively prefix-sum the block sums.
4. Use a workgroup per block to prefix-sum each block, starting from the result from the previous level.

Steps 2 and 4 can use any parallel reduction/prefix sum algorithm.
Reduce-then-Scan strategy

1. Divide elements into blocks of size $M$.
2. Use a workgroup per block to compute sum of each block.
3. Recursively prefix-sum the block sums.
4. Use a workgroup per block to prefix-sum each block, starting from the result from the previous level.

Steps 2 and 4 can use any parallel reduction/prefix sum algorithm.
Reduce-then-Scan strategy

1. Divide elements into blocks of size $M$.
2. Use a workgroup per block to compute sum of each block.
3. Recursively prefix-sum the block sums.
4. Use a workgroup per block to prefix-sum each block, starting from the result from the previous level.

Steps 2 and 4 can use any parallel reduction/prefix sum algorithm.
Reduce-then-Scan strategy

1. Divide elements into blocks of size $M$.
2. Use a workgroup per block to compute sum of each block.
3. Recursively prefix-sum the block sums.
4. Use a workgroup per block to prefix-sum each block, starting from the result from the previous level.

Steps 2 and 4 can use any parallel reduction/prefix sum algorithm.
Reduce-then-Scan strategy

1. Divide elements into blocks of size $M$.
2. Use a workgroup per block to compute sum of each block.
3. Recursively prefix-sum the block sums.
4. Use a workgroup per block to prefix-sum each block, starting from the result from the previous level.

Steps 2 and 4 can use any parallel reduction/prefix sum algorithm.
Analysis

Assuming that $M$ is reasonably large:

- About $\log_M N$ kernel instances
- Most memory accesses can be to local memory
- Slightly over $2N$ global reads
- Slightly over $N$ global writes
- Slightly over $O(N \log M)$ barrier instructions
Assuming that $M$ is reasonably large:

- About $\log_M N$ kernel instances
- Most memory accesses can be to local memory
  - Slightly over $2N$ global reads
  - Slightly over $N$ global writes
  - Slightly over $O(N \log M)$ barrier instructions
Analysis

Assuming that $M$ is reasonably large:

- About $\log_M N$ kernel instances
- Most memory accesses can be to local memory
- Slightly over $2N$ global reads
- Slightly over $N$ global writes
- Slightly over $O(N \log M)$ barrier instructions
Assuming that $M$ is reasonably large:

- About $\log_M N$ kernel instances
- Most memory accesses can be to local memory
- Slightly over $2N$ global reads
- Slightly over $N$ global writes
- Slightly over $O(N \log M)$ barrier instructions
Assuming that $M$ is reasonably large:

- About $\log_M N$ kernel instances
- Most memory accesses can be to local memory
- Slightly over $2N$ global reads
- Slightly over $N$ global writes
- Slightly over $O(N \log M)$ barrier instructions
Outline

1. Definition and Applications
   - Motivating Problem
   - Definitions
   - Other Applications

2. Parallel Algorithms
   - Kogge-Stone
   - Brent-Kung

3. GPU Strategies
   - Reduce-then-Scan
   - Two-Level Prefix Sum
The more parallelism one uses in a prefix sum, the higher the overheads become. Therefore, only use as much parallelism as is necessary to saturate the hardware.
The more parallelism one uses in a prefix sum, the higher the overheads become. Therefore, only use as much parallelism as is necessary to saturate the hardware.
Fixed Block Count

Use the same reduce-then-scan strategy, but

- Fix the **number** of blocks at $C$, set $M = \frac{N}{C}$
- Fix a work-group size $G$
- $C$ should be tuned so that $C \times G$ workitems saturate the device
- $C$ should be small enough that only 2 levels are required
Each block has size $M$ but workgroups only have $G$ workitems. How does a workgroup prefix-sum a block? Serially. In sub-blocks of size $G$ or $2G$. 
Each block has size $M$ but workgroups only have $G$ workitems. How does a workgroup prefix-sum a block? **Serially.** In sub-blocks of size $G$ or $2G$. 
Each block has size $M$ but workgroups only have $G$ workitems. How does a workgroup prefix-sum a block? **Serially.** In sub-blocks of size $G$ or $2G$. 
Advantages

- Only three kernel instances, two of which use the full GPU
- Only $O(N \log G)$ barrier instructions
- Only $O(C)$ extra global memory
Advantages

- Only three kernel instances, two of which use the full GPU
- Only $O(N \log G)$ barrier instructions
- Only $O(C)$ extra global memory
Advantages

- Only three kernel instances, two of which use the full GPU
- Only $O(N \log G)$ barrier instructions
- Only $O(C)$ extra global memory
Summary

- Parallel prefix sum is hard work
  - GPUs need parallelism, but algorithm works best with least parallelism
  - With good implementation, can be bandwidth-limited
Summary

- Parallel prefix sum is hard work
- GPUs need parallelism, but algorithm works best with least parallelism
- With good implementation, can be bandwidth-limited
Parallel prefix sum is hard work

GPUs need parallelism, but algorithm works best with least parallelism

With good implementation, can be bandwidth-limited
Guy E. Blelloch.
Prefix sums and their applications.

Duane Merrill and Andrew Grimshaw.
Parallel scan for stream architectures.
Technical Report CS2009-14, Department of Computer Science, University of Virginia, December 2009.