Prefix sums on GPUs

Bruce Merry

Department of Computer Science, University of Cape Town

GPGPU2 Workshop 2014

Outline

Outline

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.

Output Format

The lists should be packed together contiguously.

A0	A1	A2					

Assuming one workitem per object, how do the workitems know where to start?

Output Format

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?

Output Format

The lists should be packed together contiguously.

A0	A1	A2	B0	B1	D0		

Assuming one workitem per object, how do the workitems know where to start?

Output Format

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?

Output Format

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?

Solution

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.

Solution

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.

Solution

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

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.

4	3	7	9	2	3
0	4	7	14	23	25

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.

4	3	7	9	2	3
0	4	7	14	23	25

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.

4	3	7	9	2	3
4	7	14	23	25	28

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.

4	3	7	9	2	3
4	7	14	23	25	28

Outline

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

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

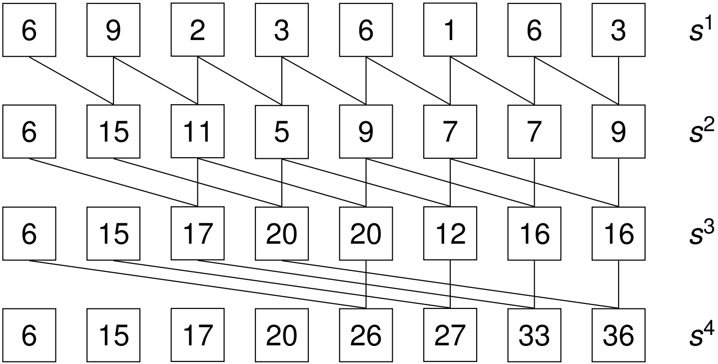
Idea

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


6 9 2 3 6 1 6 3 s^1
6 15 11 5 9 7 7 9 s^2
6 15 17 20 20 12 16 16 s^3
6 15 17 20 26 27 33 36 s^4

Pseudo-code

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

$i \leftarrow$ workitem ID;
foreach *power-of-two* t **from** 1 **to** N **do**
    $x \leftarrow a_i$;
    if $t \leq i$ **then**
        $x \leftarrow x \oplus a_{i-t}$;
    `barrier();`
    $a_i \leftarrow x$;
    `barrier();`

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.

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

Idea

For an exclusive scan:

- Add pairs of adjacent elements:

$$p_i = a_{2i} \oplus a_{2i+1}$$

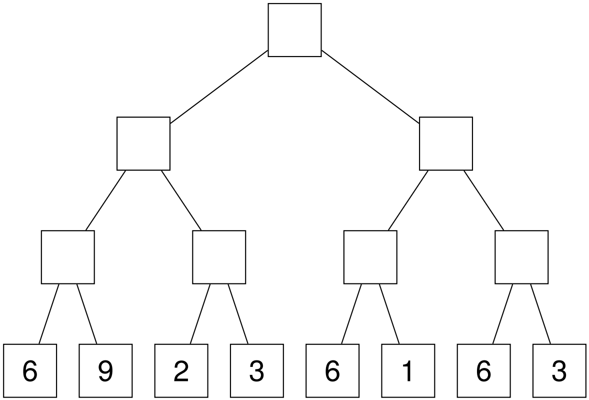
- Recursively scan these sums:

$$q_i = \bigoplus_{j=0}^{i-1} p_i = \bigoplus_{j=0}^{2i-1} a_j$$

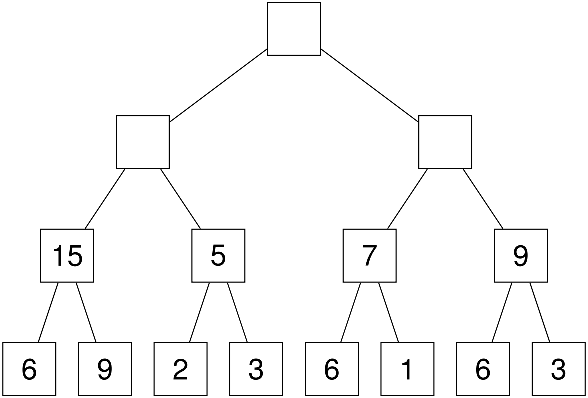
- Use these sums to compute the result:

$$s_{2i} = q_i, s_{2i+1} = q_i \oplus a_{2i}$$

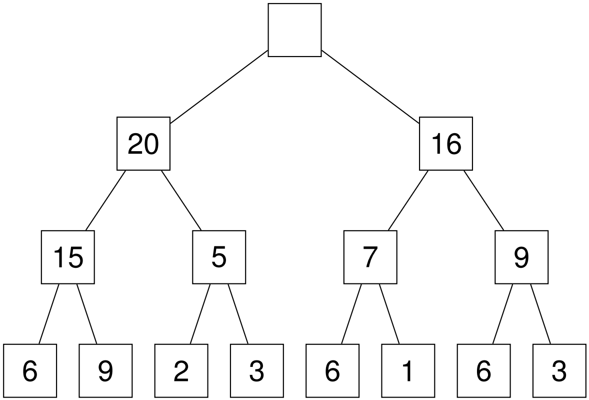
Example



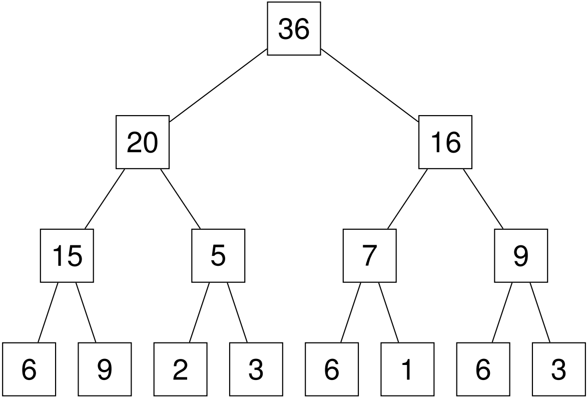
Example



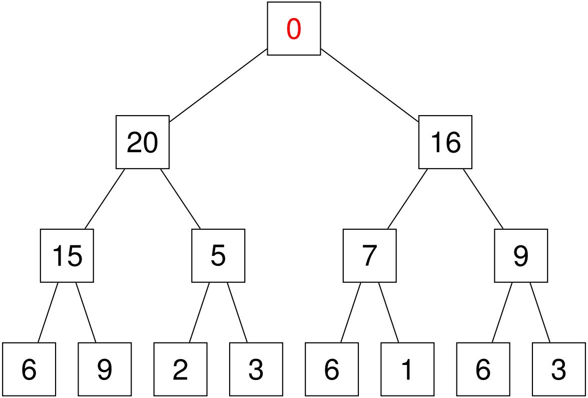
Example



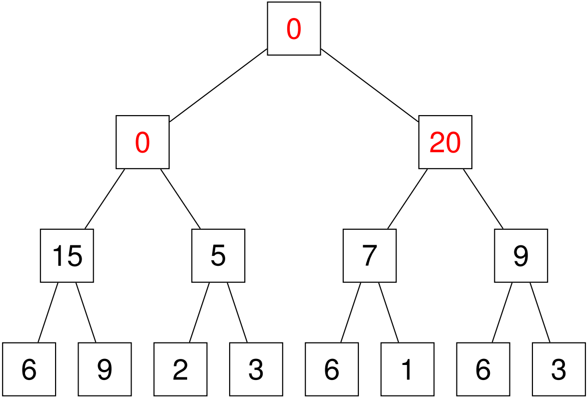
Example



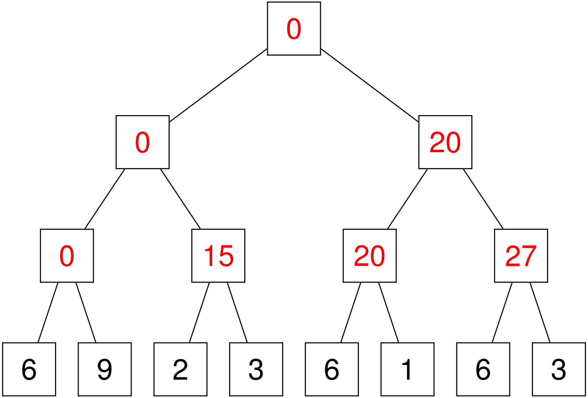
Example



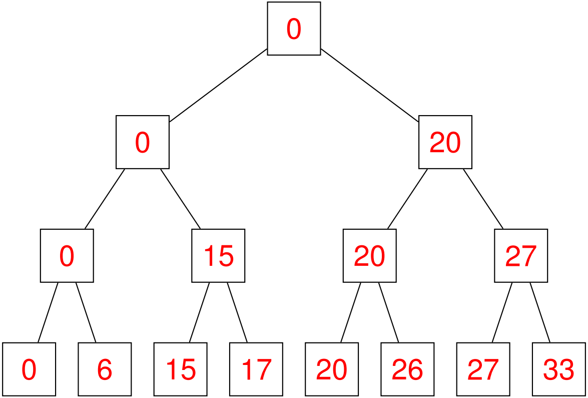
Example



Example



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.

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.

Pseudo-code

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

Copy a_i to b_{i+N} for $i \in [0, N)$;
for $t \leftarrow n - 1$ **downto** 1 **do**
  for $i \leftarrow 0$ **to** $2^t - 1$ **do** in parallel
    $b_{2^t+i} \leftarrow b_{2^{t+1}+i} \oplus b_{2^{t+1}+i+1}$;

```
// Exclusive prefix sum of two elements
```

$b_3 \leftarrow b_2$;
$b_2 \leftarrow I$;
for $t \leftarrow 1$ **to** $n - 1$ **do**
  for $i \leftarrow 0$ **to** $2^t - 1$ **do** in parallel
    $b_{2^{t+1}+i+1} \leftarrow b_{2^t+i} \oplus b_{2^{t+1}+i}$;
    $b_{2^{t+1}+i} \leftarrow b_{2^t+i}$;
Copy b_{i+N} to s_i for $i \in [0, N)$;

Per-workitem Pseudo-code

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

$i \leftarrow$ work-item ID;
$b_{i+N} \leftarrow a_i$;
$b_{i+N+\frac{N}{2}} \leftarrow a_{i+\frac{N}{2}}$;
`barrier();`
for $t \leftarrow n-1$ **downto** 1 **do**
  if $i < 2^t$ **then**
    $b_{2^t+i} \leftarrow b_{2^{t+1}+2i} \oplus b_{2^{t+1}+2i+1}$;
  `barrier();`
if $i = 0$ **then**
  $a_3 \leftarrow a_2$;
  $a_2 \leftarrow I$;
`barrier();`
for $t \leftarrow 1$ **to** $n-1$ **do**
  if $i < 2^t$ **then**
    $b_{2^{t+1}+2i+1} \leftarrow b_{2^t+i} \oplus b_{2^{t+1}+2i}$;
    $b_{2^{t+1}+2i} \leftarrow b_{2^t+i}$;
  `barrier();`
$s_i \leftarrow b_{i+N}$;
$s_{i+\frac{N}{2}} \leftarrow b_{i+N+\frac{N}{2}}$;

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

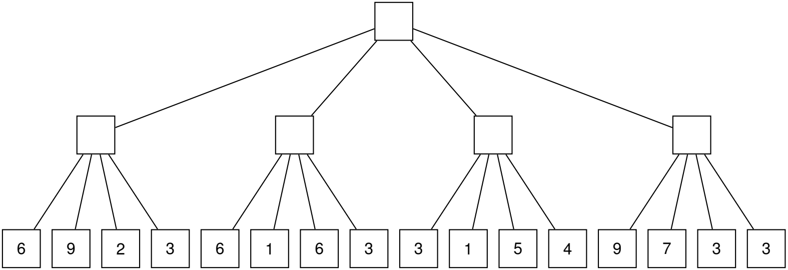
Applying one of these at a larger (multi-workgroup) scale has issues:

- 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

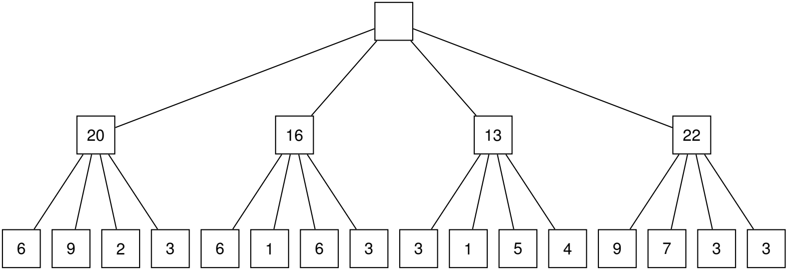
Generalizing Brent-Kung

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



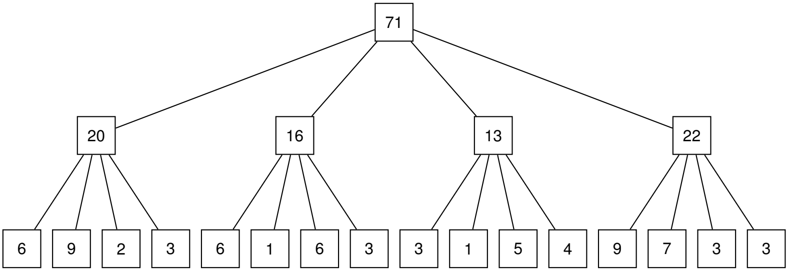
Generalizing Brent-Kung

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



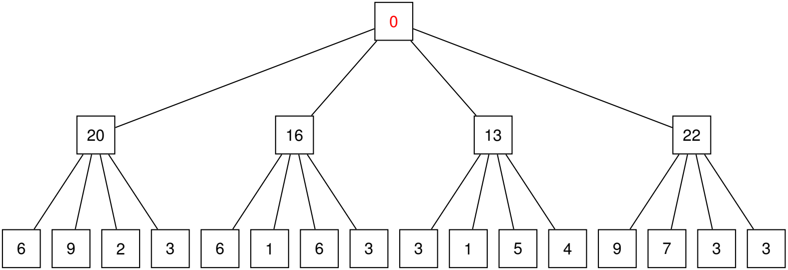
Generalizing Brent-Kung

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



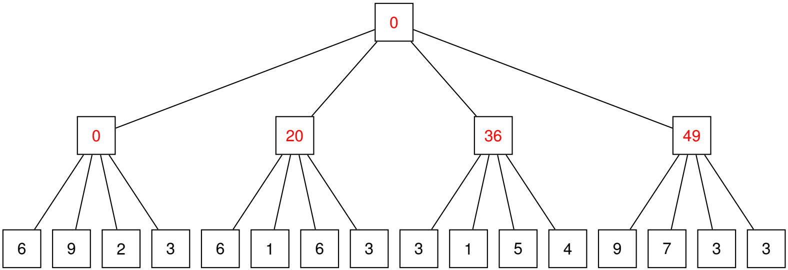
Generalizing Brent-Kung

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



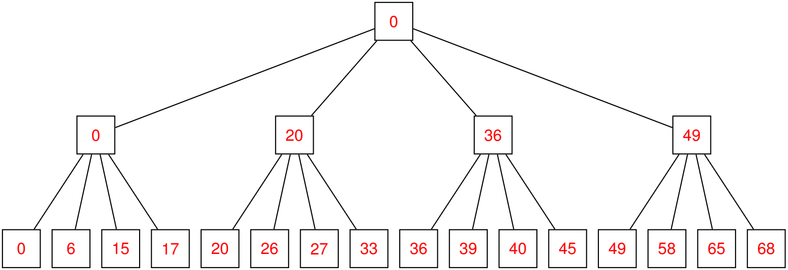
Generalizing Brent-Kung

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



Generalizing Brent-Kung

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



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

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

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

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

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

Outline

Reducing Parallelism

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.

Reducing Parallelism

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

Prefix-Summing a Block

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 $2G$.

Prefix-Summing a Block

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 $2G$.

Prefix-Summing a Block

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

Summary

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

References

- Guy E. Blelloch.
  Prefix sums and their applications.
  Technical Report CMU-CS-90-190, Computer Science Department, Carnegie Mellon University, November 1990.
- Duane Merrill and Andrew Grimshaw.
  Parallel scan for stream architectures.
  Technical Report CS2009-14, Department of Computer Science, University of Virginia, December 2009.