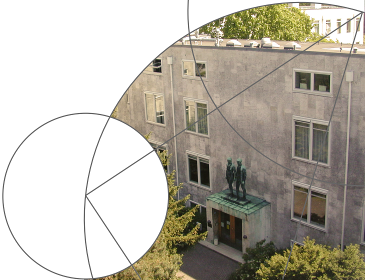
Faculty of Science

Multireduce and Multiscan on Modern GPUs

Master's thesis defence

Marco Eilers
Department of Computer Science



April 15, 2014

Overview

Reduce and scan

Works on any monoid M with binary function $\odot$

Reduce and scan

Works on any monoid M with binary function $\odot$

Examples with integers and addition

Reduce and scan

Works on any monoid M with binary function $\odot$

Examples with integers and addition

Input vector

4	1	3	9	0	5	5	2

Reduce and scan

Works on any monoid M with binary function $\odot$

Examples with integers and addition

Input vector

4	1	3	9	0	5	5	2

Reduce: Sum of all values

30

Reduce and scan

Works on any monoid M with binary function $\odot$

Examples with integers and addition

Input vector

4	1	3	9	0	5	5	2

Reduce: Sum of all values

30

Scan: Sums of all prefixes

0	4	5	8	17	17	23	28

Reduce and scan

Works on any monoid M with binary function $\odot$

Examples with integers and addition

Input vector

4	1	3	9	0	5	5	2

Reduce: Sum of all values

30

Scan: Sums of all prefixes

0	4	5	8	17	17	23	28

Segmented reduce/scan: Partition input into contiguous segments and reduce/scan them separately

Multireduce

Input *labels*

0	2	2	1	0	2	0	1	1	1	2	0

Input values

4	1	3	9	0	5	5	2	7	1	3	5

Multireduce

Input *labels*

0	2	2	1	0	2	0	1	1	1	2	0

Input values

4	1	3	9	0	5	5	2	7	1	3	5

Output:

Multireduce

Reduce the values for each label separately.

0	2	2	1	0	2	0	1	1	1	2	0

4	1	3	9	0	5	5	2	7	1	3	5

Multireduce

Reduce the values for each label separately.

0	2	2	1	0	2	0	1	1	1	2	0

4	1	3	9	0	5	5	2	7	1	3	5

Multireduce

Reduce the values for each label separately.

0	2	2	1	0	2	0	1	1	1	2	0

4	1	3	9	0	5	5	2	7	1	3	5

14		

Multireduce

Reduce the values for each label separately.

0	2	2	1	0	2	0	1	1	1	2	0

4	1	3	9	0	5	5	2	7	1	3	5

14	19	

Multireduce

Reduce the values for each label separately.

0	2	2	1	0	2	0	1	1	1	2	0

4	1	3	9	0	5	5	2	7	1	3	5

14	19	12

Multiscan

Indentical inputs:

0	2	2	1	0	2	0	1	1	1	2	0

4	1	3	9	0	5	5	2	7	1	3	5

Multiscan

Indentical inputs:

0	2	2	1	0	2	0	1	1	1	2	0

4	1	3	9	0	5	5	2	7	1	3	5

Different output:

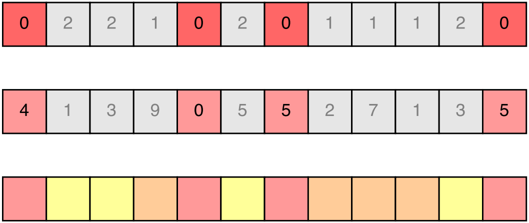
Multiscan

Scan the values for each label separately

0	2	2	1	0	2	0	1	1	1	2	0
4	1	3	9	0	5	5	2	7	1	3	5

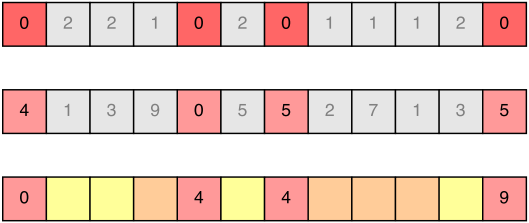
Multiscan

Scan the values for each label separately



Multiscan

Scan the values for each label separately



Multiscan

Scan the values for each label separately

0	2	2	1	0	2	0	1	1	1	2	0

4	1	3	9	0	5	5	2	7	1	3	5

0			0	4		4	9	11	18		9

Multiscan

Scan the values for each label separately

0	2	2	1	0	2	0	1	1	1	2	0

4	1	3	9	0	5	5	2	7	1	3	5

0	0	1	0	4	4	4	9	11	18	9	9

Applications

Multiscan has been suggested as a primitive to build entire machines around.

Applications

Multiscan has been suggested as a primitive to build entire machines around.
Immediate applications are

- Radix sort
- Sparse matrix vector multiplication

Applications

Multiscan has been suggested as a primitive to build entire machines around.

Immediate applications are

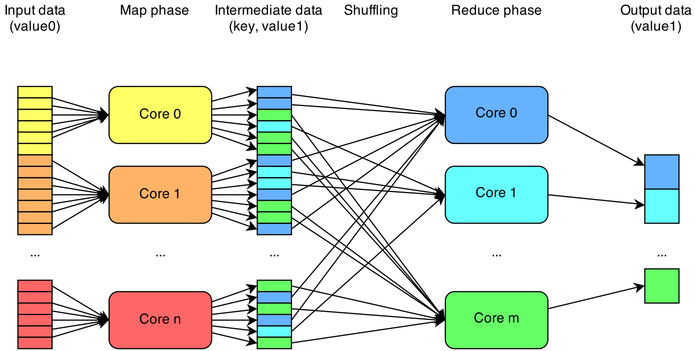
- Radix sort
- Sparse matrix vector multiplication

Multireduce generalizes important operations

- Histograms
- Scattering

General multireduce can be used for MapReduce framework

Applications



Sequential algorithms

Sequential algorithms for both operations are trivial.

```
function multiReduce(int n, int m, int indices[n], int values[n], int buckets[m]) :
    for i = 0 to m − 1 do
        buckets[i] = 0;
    end
    for i = 0 to n − 1 do
        buckets[indices[i]] += values[i];
    end
end
```

Sequential algorithms

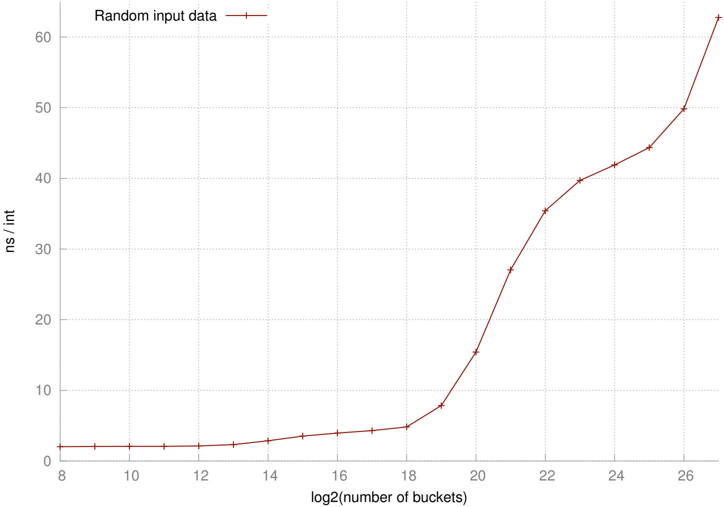
Sequential algorithms for both operations are trivial.

function multiScan(int n, int m, int $indices[n]$, int $values[n]$, int $result[n]$) **:**

```
int buckets[m];
for i = 0 to m − 1 do
    buckets[i] = 0;
end
for i = 0 to n − 1 do
    result[i] = buckets[indices[i]];
    buckets[indices[i]] += values[i];
end
```

end

Performance problems



Performance problems

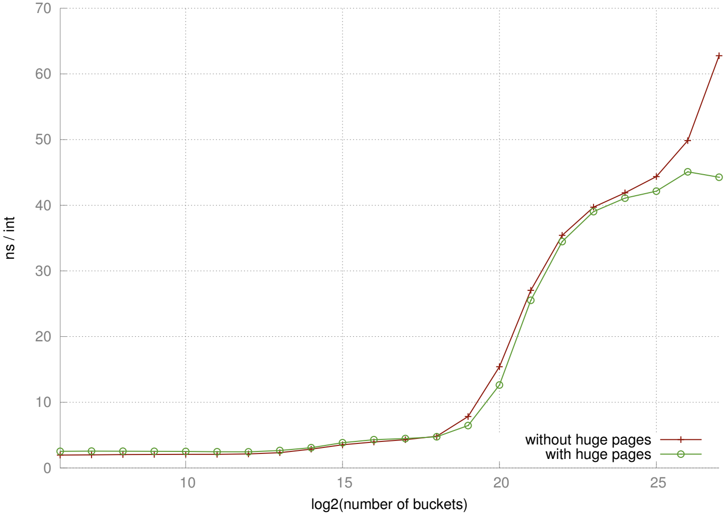
- TLB misses
- Cache misses

Solutions

Solutions

Huge pages can be used to mitigate the problem of TLB misses

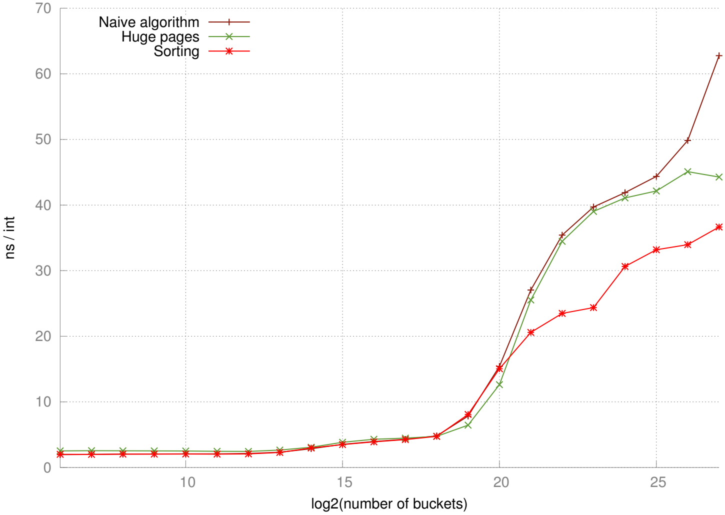
Solutions


70
60
50
40
30
20
10
0
ns / int
10
15
20
25
log2(number of buckets)
without huge pages
with huge pages

Solutions

Sorting the input by labels turns random bucket accesses into sequential ones and avoids cache misses

Solutions



Naive algorithm
Huge pages
Sorting
ns / int
log2(number of buckets)
0
10
20
30
40
50
60
70
10
15
20
25

Applicability of sorting approach

Sorting *can* lead to a performance gain in some scenarios.
Think accesses to disk, network, ...

Applicability of sorting approach

Sorting *can* lead to a performance gain in some scenarios.
Think accesses to disk, network, ...
However...

- This is highly dependent on the system's memory architecture
- Finding the right parameters is non-trivial.

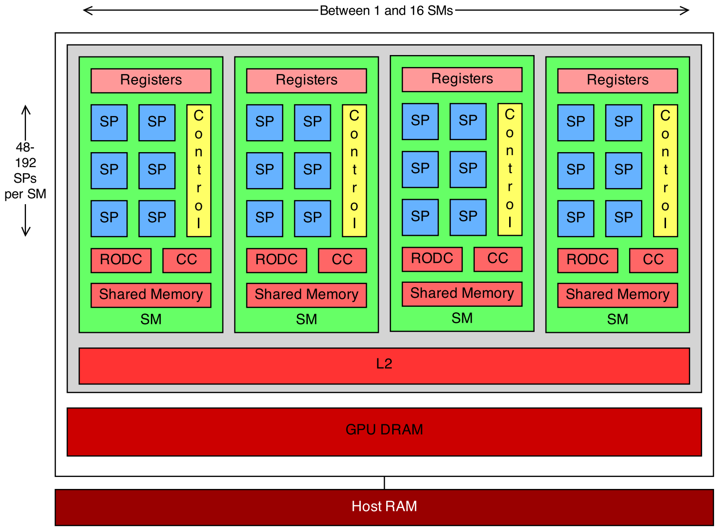
Applicability of sorting approach

Sorting *can* lead to a performance gain in some scenarios.
Think accesses to disk, network, ...
However...

- This is highly dependent on the system's memory architecture
- Finding the right parameters is non-trivial.

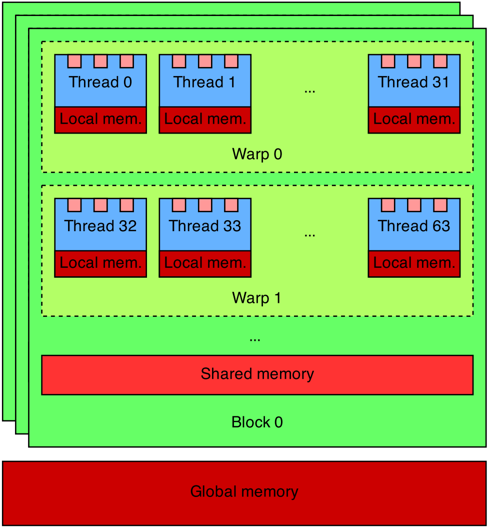
Developed a simulator to predict performance.

GPUs



Between 1 and 16 SMs
48-
192
SPs
per SM
Registers
SP
Control
RODC
CC
Shared Memory
SM
L2
GPU DRAM
Host RAM

CUDA architecture



CUDA: Performance requirements

CUDA: Performance requirements

- Hundreds of threads per SM

CUDA: Performance requirements

- Hundreds of threads per SM
- Limited memory bandwidth

CUDA: Performance requirements

- Hundreds of threads per SM
- Limited memory bandwidth
- Latency hiding

CUDA: Performance requirements

- Hundreds of threads per SM
- Limited memory bandwidth
- Latency hiding
- Coalesced global memory access

CUDA: Performance requirements

- Hundreds of threads per SM
- Limited memory bandwidth
- Latency hiding
- Coalesced global memory access
- Bank conflicts

CUDA: Performance requirements

- Hundreds of threads per SM
- Limited memory bandwidth
- Latency hiding
- Coalesced global memory access
- Bank conflicts
- Atomic operations

CUDA: Performance requirements

- Hundreds of threads per SM
- Limited memory bandwidth
- Latency hiding
- Coalesced global memory access
- Bank conflicts
- Atomic operations
- Synchronization

CUDA: PRAM algorithm performance

CUDA: PRAM algorithm performance

- Often not massively parallel, sometimes too much

CUDA: PRAM algorithm performance

- Often not massively parallel, sometimes too much
- Usually assume SIMD
  - Require sync after each step...
  - ...which is either slow or impossible

CUDA: PRAM algorithm performance

- Often not massively parallel, sometimes too much
- Usually assume SIMD
  - Require sync after each step...
  - ...which is either slow or impossible
- No concept of block-level shared memory

CUDA: PRAM algorithm performance

- Often not massively parallel, sometimes too much
- Usually assume SIMD
  - Require sync after each step...
  - ...which is either slow or impossible
- No concept of block-level shared memory
- Generally not much attention paid to memory layout
  - Often uncoalesced memory accesses
  - Often bank conflicts
  - Generally less focus on avoiding memory accesses

Possibilities for algorithm design

Assume, a multireduce algorithm is given.
There are two possibilities to use it:

Possibilities for algorithm design

Assume, a multireduce algorithm is given.
There are two possibilities to use it:

1. Apply it directly to the entire input

Possibilities for algorithm design

Assume, a multireduce algorithm is given.
There are two possibilities to use it:

1. Apply it directly to the entire input
2. Partition the input into several segments, apply the algorithm to all segments in parallel, then combine partial results

Possibilities for algorithm design

Input values	3	1	2	0	4	1	1	3	2	1	4	4	0	3	1	3
Input labels	2	0	1	0	0	1	2	2	1	1	0	1	2	1	1	0

Core 0 | Core 1 | Core 2 | Core 3

Store the partial reductions for each segment

← nSegments →

nBuckets				
	1	4	4	3
	2	1	7	4
	3	4	0	0

Reduce partial results for each bucket

12
14
7

Possibilities for algorithm design

If partitioning is used, intermediate data can be stored...

Possibilities for algorithm design

If partitioning is used, intermediate data can be stored...

- ... in global memory
    - High latency
    - Coalescing required
    - Large

Possibilities for algorithm design

If partitioning is used, intermediate data can be stored...

- ... in global memory
  - High latency
  - Coalescing required
  - Large
- ... in shared memory
  - Low latency
  - Danger of bank conflicts
  - Small

Possibilities for algorithm design

If partitioning is used, intermediate data can be stored...

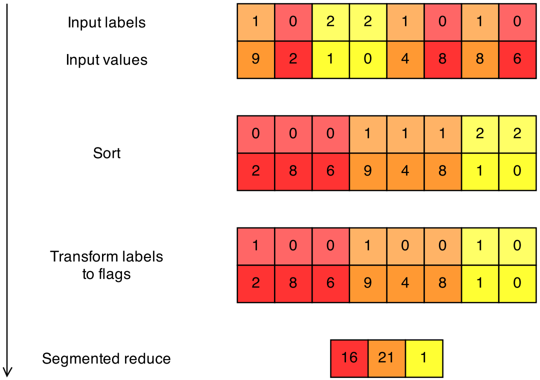
- ... in global memory
  - High latency
  - Coalescing required
  - Large
- ... in shared memory
  - Low latency
  - Danger of bank conflicts
  - Small

This results in three ways to implement (almost) any given algorithm.

Sort-based algorithms

Idea:

1. Sort input by labels, so that values for identical labels are adjacent
2. Reduce/scan values for each label...
   - ... using a segmented reduce/scan



Adapting the sequential algorithm

Idea: Apply the partitioning trick again on a block level.
Let each thread work sequentially on its own segment of the input data, calculating its own partial result ("bucket set").

Adapting the sequential algorithm

Idea: Apply the partitioning trick again on a block level.
Let each thread work sequentially on its own segment of the input data, calculating its own partial result ("bucket set").
Advantages:

- Arbitrary amounts of parallelism
- Basic algorithm is efficient

Adapting the sequential algorithm

Idea: Apply the partitioning trick again on a block level.
Let each thread work sequentially on its own segment of the input data, calculating its own partial result ("bucket set").
Advantages:

- Arbitrary amounts of parallelism
- Basic algorithm is efficient

Problems:

- Random access reads/writes
- Input data is required in contiguous blocks, but fastest access is strided
- Many partial results have to be combined

Optimization for multireduce

Many of the most frequently used operators are commutative.

Optimization for multireduce

Many of the most frequently used operators are commutative.
In this case:

- Input data can be processed in any order
- Several threads can share a bucket set
    - (as long as write conflicts are handled somehow)

Optimization for multireduce

Many of the most frequently used operators are commutative.
In this case:

- Input data can be processed in any order
- Several threads can share a bucket set
    - (as long as write conflicts are handled somehow)

Consequently

- Less partial results to process
- Optimal data access pattern without further effort
- Less cache misses
- Atomic operations needed

Example of multireduce in shared memory

- Adaptation of sequential algorithm
- For commutative operators only
- Bucket sets stored in shared memory
- Derived from histogram algorithm

Example of multireduce in shared memory

- Adaptation of sequential algorithm
- For commutative operators only
- Bucket sets stored in shared memory
- Derived from histogram algorithm

Memory layout is the deciding factor.

Example of multireduce in shared memory

Two approaches for histogram algorithms:

1. Per-block bucket sets
   - Large amounts of parallelism
   - Bank and write conflicts
2. Per-thread bucket sets
   - No conflicts at all
   - Limited parallelism

Example of multireduce in shared memory

Two approaches for histogram algorithms:

1. Per-block bucket sets
    - Large amounts of parallelism
    - Bank and write conflicts
2. Per-thread bucket sets
    - No conflicts at all
    - Limited parallelism

My approach combines the best of both

- Large amounts of parallelism
- Conflicts only where they don't hurt

Example of multireduce in shared memory

Two approaches for histogram algorithms:

1. Per-block bucket sets
    - Large amounts of parallelism
    - Bank and write conflicts
2. Per-thread bucket sets
    - No conflicts at all
    - Limited parallelism

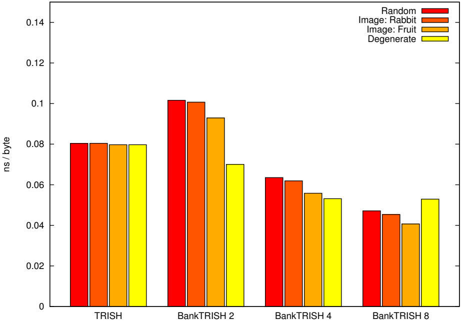
My approach combines the best of both

- Large amounts of parallelism
- Conflicts only where they don't hurt

Resulting multireduce is faster than the best histogram algorithm, but much more general.

Application to histogramming

Algorithm can be transferred back to histogramming, resulting in a 40% speedup on average.



Multiscan algorithms

Options for multiscan:

Multiscan algorithms

Options for multiscan:

1. Sorting approach
    - More complicated

Multiscan algorithms

Options for multiscan:

1. Sorting approach
   - More complicated
2. Adaptation of sequential algorithm
   - Similar to non-commutative multireduce case

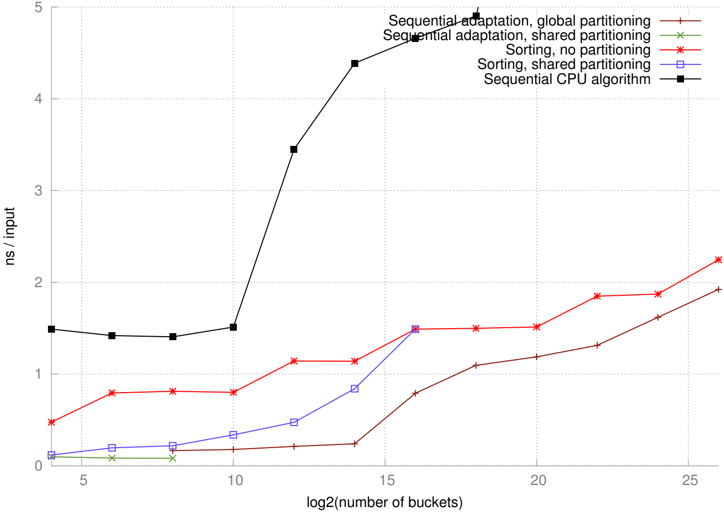
Multiscan algorithms

Options for multiscan:

1. Sorting approach
    - More complicated
2. Adaptation of sequential algorithm
    - Similar to non-commutative multireduce case
3. PRAM algorithm by Sheffler
    - Frequent memory accesses
    - Frequent synchronizations
    - Much data, few threads

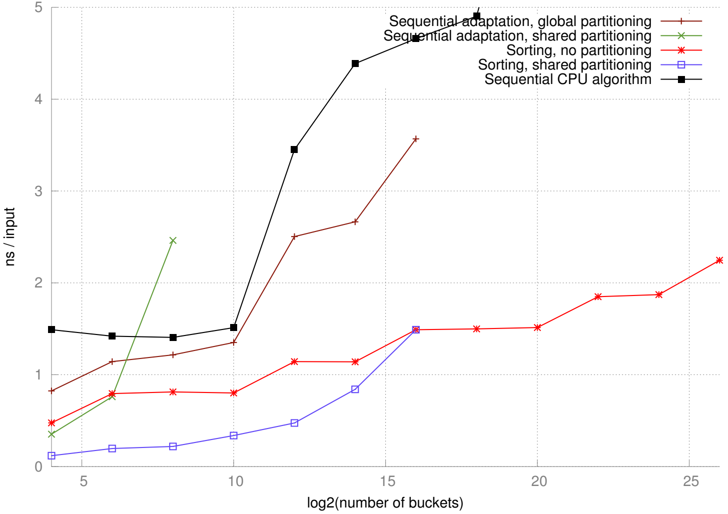
Performance of multireduce

With commutative operator:

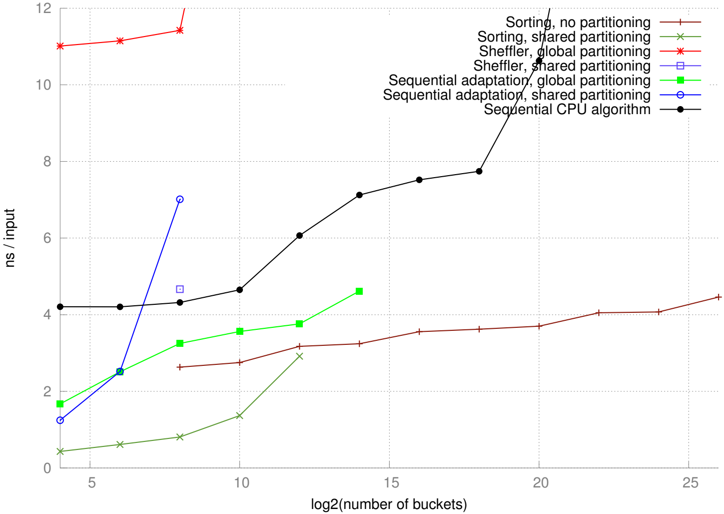


Performance of multireduce

With non-commutative operator:



Performance of multiscan



Conclusion

Conclusion

- Theoretically optimal algorithm is not always the best in practice (caching problems)

Conclusion

- Theoretically optimal algorithm is not always the best in practice (caching problems)
- GPU always results in speedup over CPU

Conclusion

- Theoretically optimal algorithm is not always the best in practice (caching problems)
- GPU always results in speedup over CPU
- Multireduce can be implemented efficiently
    - Number of buckets and commutativity have big influence

Conclusion

- Theoretically optimal algorithm is not always the best in practice (caching problems)
- GPU always results in speedup over CPU
- Multireduce can be implemented efficiently
  - Number of buckets and commutativity have big influence
- and can be recommended as parallel primitive

Conclusion

- Theoretically optimal algorithm is not always the best in practice (caching problems)
- GPU always results in speedup over CPU
- Multireduce can be implemented efficiently
    - Number of buckets and commutativity have big influence
- and can be recommended as parallel primitive
- Multiscan cannot

Conclusion

- Theoretically optimal algorithm is not always the best in practice (caching problems)
- GPU always results in speedup over CPU
- Multireduce can be implemented efficiently
  - Number of buckets and commutativity have big influence
- and can be recommended as parallel primitive
- Multiscan cannot
- Multireduce algorithm can be applied directly to improve histogramming

Future work

Future work

- Explore sorting for better cache use
    - Improve simulator

Future work

- Explore sorting for better cache use
    - Improve simulator
- Algorithm optimization

Future work

- Explore sorting for better cache use
    - Improve simulator
- Algorithm optimization
- Applications of multireduce

Future work

- Explore sorting for better cache use
    - Improve simulator
- Algorithm optimization
- Applications of multireduce
- Language integration

Thank you for your attention!

Questions?