The Ninth International Workshop on Accelerators and Hybrid Exascale Systems (AsHES)
May 20th, 2019 | Rio de Janeiro, Copacabana, Brazil

Approximate and Exact Selection on GPUs

Tobias Ribizel, Hartwig Anzt

Tobias Ribizel

Karlsruher Institut für Technologie

THE UNIVERSITY OF TENNESSEE KNOXVILLE

Selection Problem

Given an unsorted sequence of real numbers $x_0, x_1, x_2, x_3, \ldots x_{n-1}$, we want to find the element x_{i_k} such that in the sorted sequence

$$x_{i_0} \leq x_{i_1} \leq x_{i_2} \leq x_{i_3} \leq \cdots \leq \underset{\uparrow}{x_{i_k}} \leq \ldots x_{i_{n-1}}$$
$$k$$

the element x_{i_k} is located in position k.

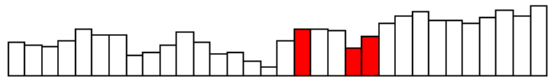
We do not necessarily need to sort the complete sequence!

- Statistics (Quantiles)
- Approximations
- Thresholds

General Approach: Partial Sorting

```
1  double select(data, rank) {
2      if (size(data) <= base_case_size) {
3          sort(data);
4          return data[rank];
5      }
6      // select splitters
7      splitters = pick_splitters(data);
8      // compute bucket sizes n_i
9      counts = count_buckets(data, splitters);
10     // compute bucket ranks r_i
11     offsets = prefix_sum(counts);
12     // determine bucket containing rank
13     bucket = lower_bound(offsets, rank);
14     // recursive subcall
15     data = extract_bucket(data, bucket);
16     rank -= offsets[bucket];
17     return select(data, rank);
18 }
```

Pick splitters

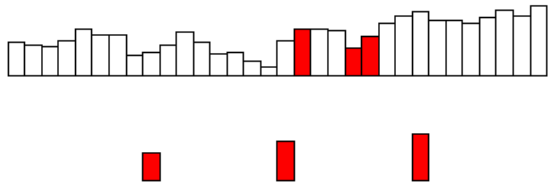


General Approach: Partial Sorting

```
1  double select(data, rank) {
2      if (size(data) <= base_case_size) {
3          sort(data);
4          return data[rank];
5      }
6      // select splitters
7      splitters = pick_splitters(data);
8      // compute bucket sizes n_i
9      counts = count_buckets(data, splitters);
10     // compute bucket ranks r_i
11     offsets = prefix_sum(counts);
12     // determine bucket containing rank
13     bucket = lower_bound(offsets, rank);
14     // recursive subcall
15     data = extract_bucket(data, bucket);
16     rank -= offsets[bucket];
17     return select(data, rank);
18 }
```

Pick splitters

Sort splitters

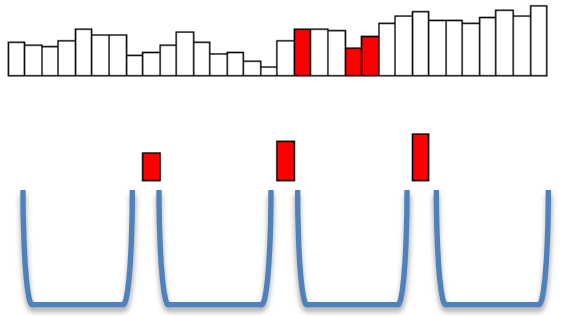


General Approach: Partial Sorting

```
1  double select(data, rank) {
2      if (size(data) <= base_case_size) {
3          sort(data);
4          return data[rank];
5      }
6      // select splitters
7      splitters = pick_splitters(data);
8      // compute bucket sizes n_i
9      counts = count_buckets(data, splitters);
10     // compute bucket ranks r_i
11     offsets = prefix_sum(counts);
12     // determine bucket containing rank
13     bucket = lower_bound(offsets, rank);
14     // recursive subcall
15     data = extract_bucket(data, bucket);
16     rank -= offsets[bucket];
17     return select(data, rank);
18 }
```

Pick splitters

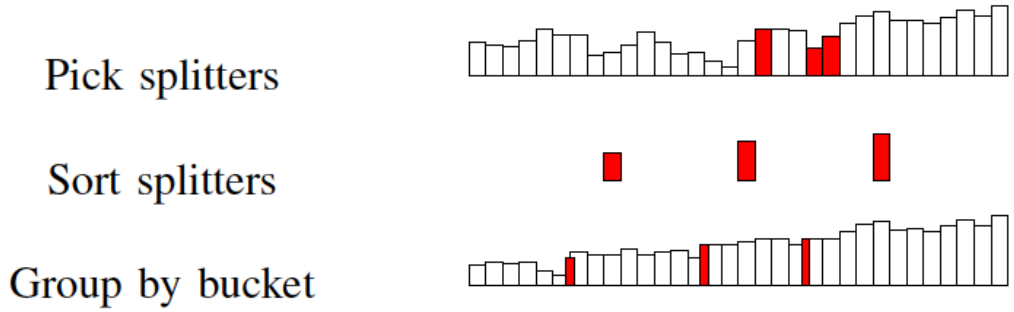
Sort splitters



Splitters separate buckets

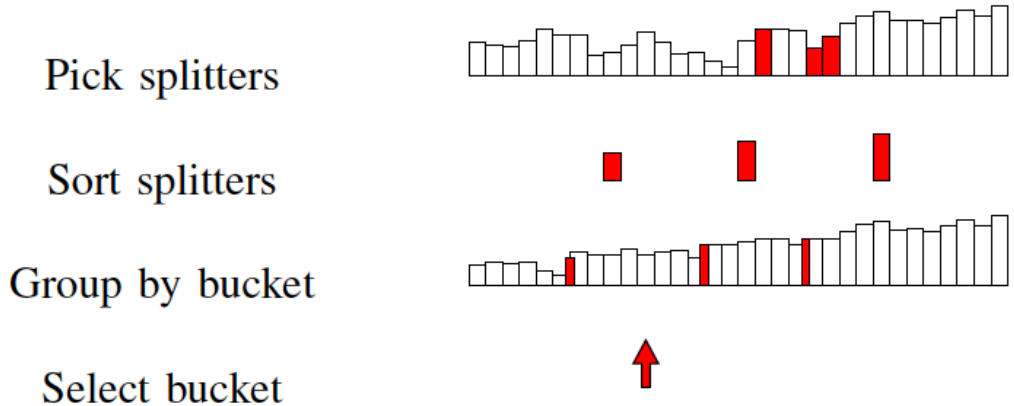
General Approach: Partial Sorting

```
1  double select(data, rank) {
2      if (size(data) <= base_case_size) {
3          sort(data);
4          return data[rank];
5      }
6      // select splitters
7      splitters = pick_splitters(data);
8      // compute bucket sizes n_i
9      counts = count_buckets(data, splitters);
10     // compute bucket ranks r_i
11     offsets = prefix_sum(counts);
12     // determine bucket containing rank
13     bucket = lower_bound(offsets, rank);
14     // recursive subcall
15     data = extract_bucket(data, bucket);
16     rank -= offsets[bucket];
17     return select(data, rank);
18 }
```



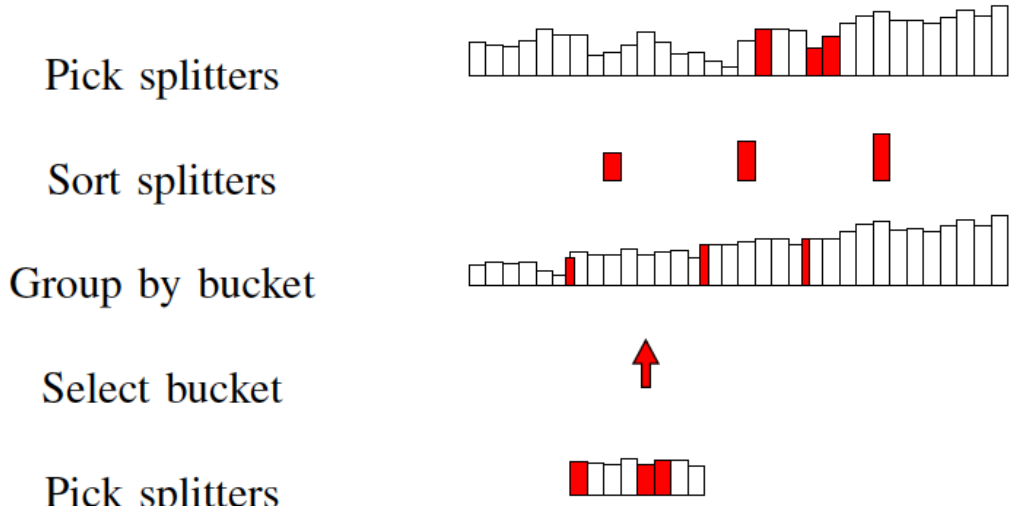
General Approach: Partial Sorting

```
1  double select(data, rank) {
2      if (size(data) <= base_case_size) {
3          sort(data);
4          return data[rank];
5      }
6      // select splitters
7      splitters = pick_splitters(data);
8      // compute bucket sizes n_i
9      counts = count_buckets(data, splitters);
10     // compute bucket ranks r_i
11     offsets = prefix_sum(counts);
12     // determine bucket containing rank
13     bucket = lower_bound(offsets, rank);
14     // recursive subcall
15     data = extract_bucket(data, bucket);
16     rank -= offsets[bucket];
17     return select(data, rank);
18 }
```



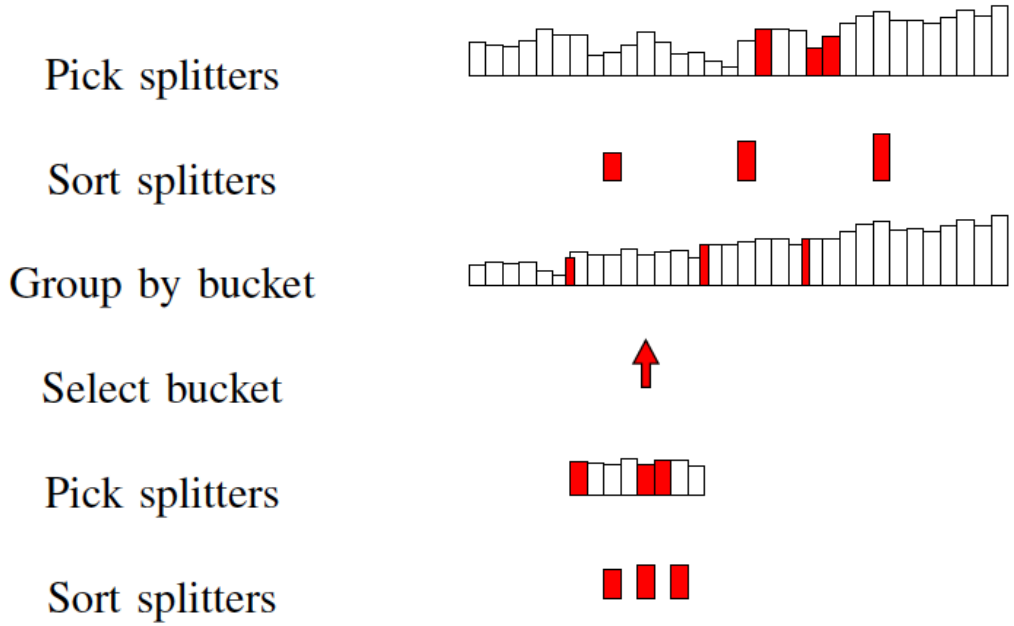
General Approach: Partial Sorting

```
1  double select(data, rank) {
2      if (size(data) <= base_case_size) {
3          sort(data);
4          return data[rank];
5      }
6      // select splitters
7      splitters = pick_splitters(data);
8      // compute bucket sizes n_i
9      counts = count_buckets(data, splitters);
10     // compute bucket ranks r_i
11     offsets = prefix_sum(counts);
12     // determine bucket containing rank
13     bucket = lower_bound(offsets, rank);
14     // recursive subcall
15     data = extract_bucket(data, bucket);
16     rank -= offsets[bucket];
17     return select(data, rank);
18 }
```



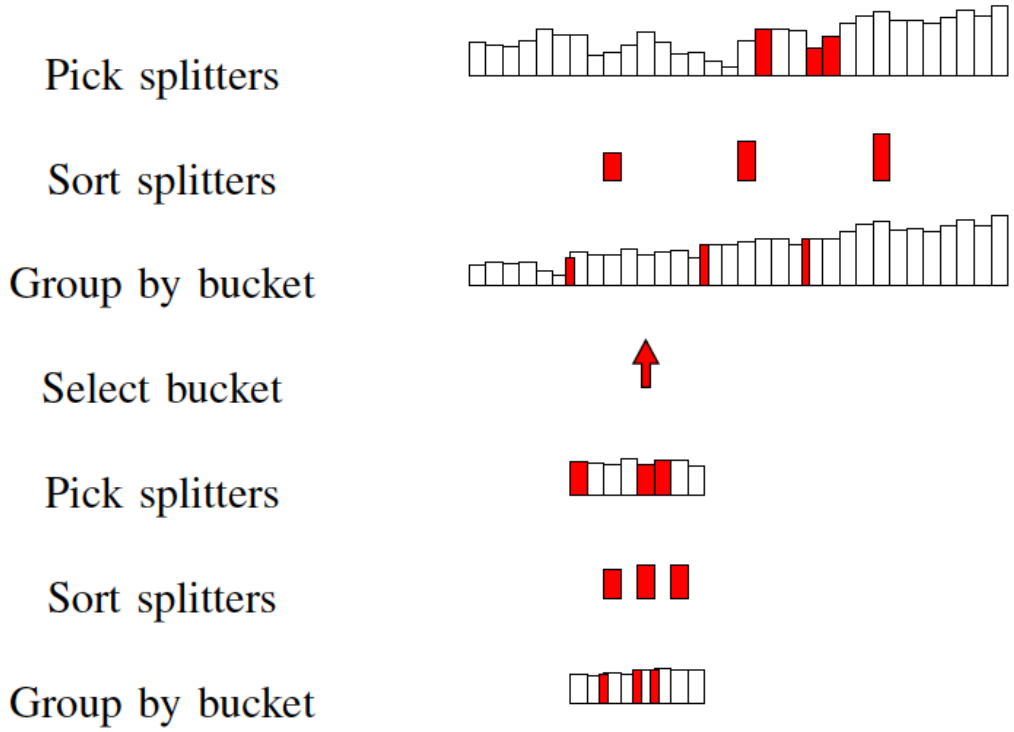
General Approach: Partial Sorting

```
1  double select(data, rank) {
2      if (size(data) <= base_case_size) {
3          sort(data);
4          return data[rank];
5      }
6      // select splitters
7      splitters = pick_splitters(data);
8      // compute bucket sizes n_i
9      counts = count_buckets(data, splitters);
10     // compute bucket ranks r_i
11     offsets = prefix_sum(counts);
12     // determine bucket containing rank
13     bucket = lower_bound(offsets, rank);
14     // recursive subcall
15     data = extract_bucket(data, bucket);
16     rank -= offsets[bucket];
17     return select(data, rank);
18 }
```



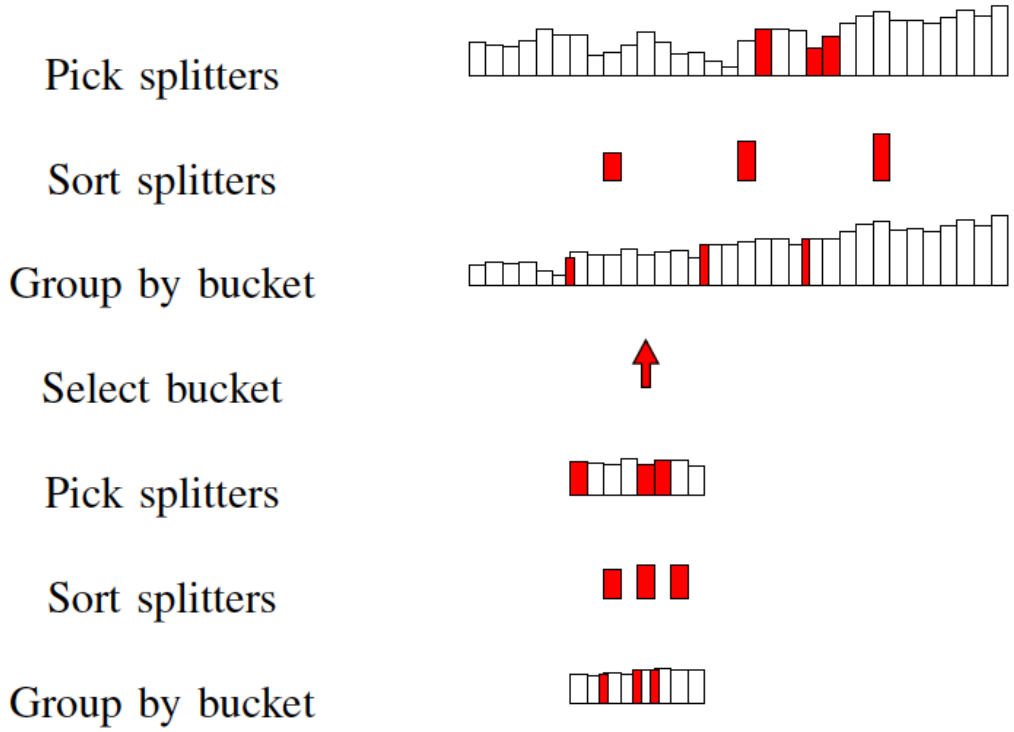
General Approach: Partial Sorting

```
1  double select(data, rank) {
2      if (size(data) <= base_case_size) {
3          sort(data);
4          return data[rank];
5      }
6      // select splitters
7      splitters = pick_splitters(data);
8      // compute bucket sizes n_i
9      counts = count_buckets(data, splitters);
10     // compute bucket ranks r_i
11     offsets = prefix_sum(counts);
12     // determine bucket containing rank
13     bucket = lower_bound(offsets, rank);
14     // recursive subcall
15     data = extract_bucket(data, bucket);
16     rank -= offsets[bucket];
17     return select(data, rank);
18 }
```



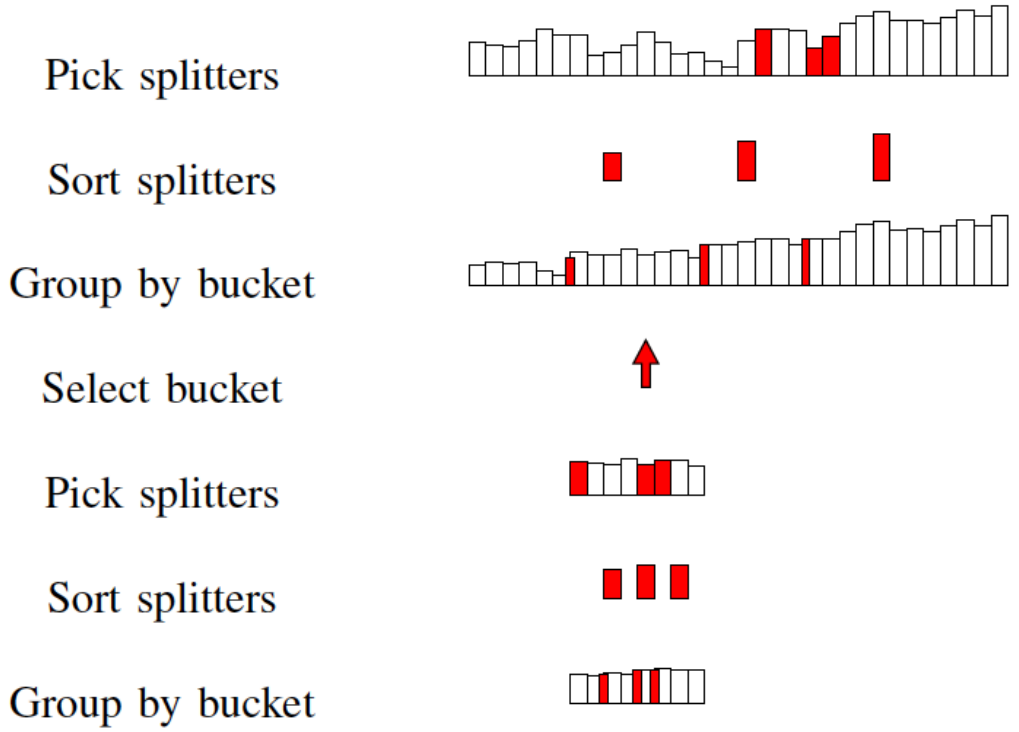
Implementation Aspects

```
1  double select(data, rank) {
2      if (size(data) <= base_case_size) {
3          sort(data);
4          return data[rank];
5      }
6      // select splitters
7      splitters = pick_splitters(data);
8      // compute bucket sizes n_i
9      counts = count_buckets(data, splitters);
10     // compute bucket ranks r_i
11     offsets = prefix_sum(counts);
12     // determine bucket containing rank
13     bucket = lower_bound(offsets, rank);
14     // recursive subcall
15     data = extract_bucket(data, bucket);
16     rank -= offsets[bucket];
17     return select(data, rank);
18 }
```



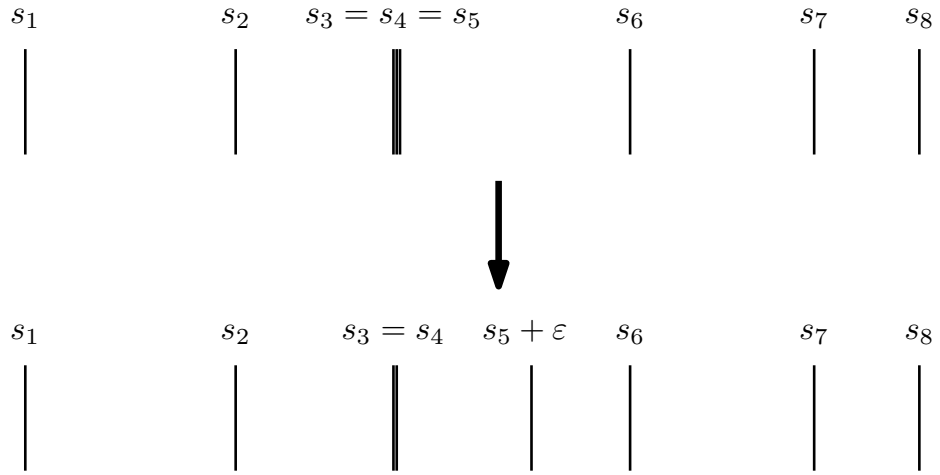
Implementation Aspects

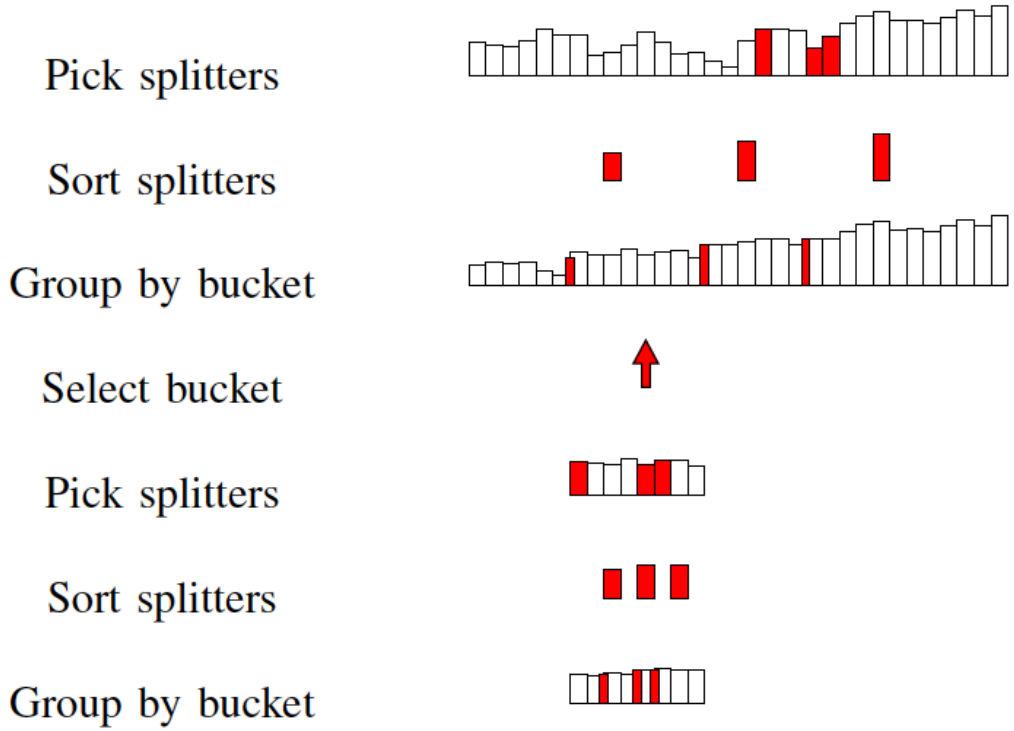
- We only copy elements of the bucket we are interested in;



Implementation Aspects

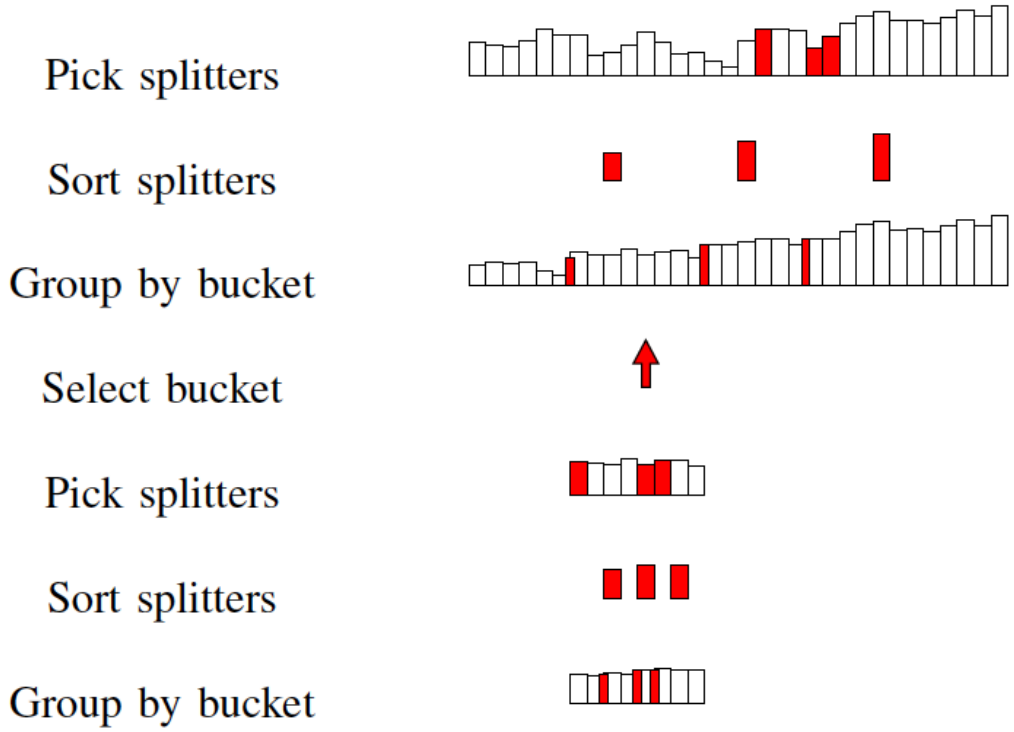
- We only copy elements of the bucket we are interested in;
- In case of identical splitter elements, they are placed in an *equality bucket*;
- If target rank is in an *equality bucket*, the algorithm can terminate early by returning lower bound;





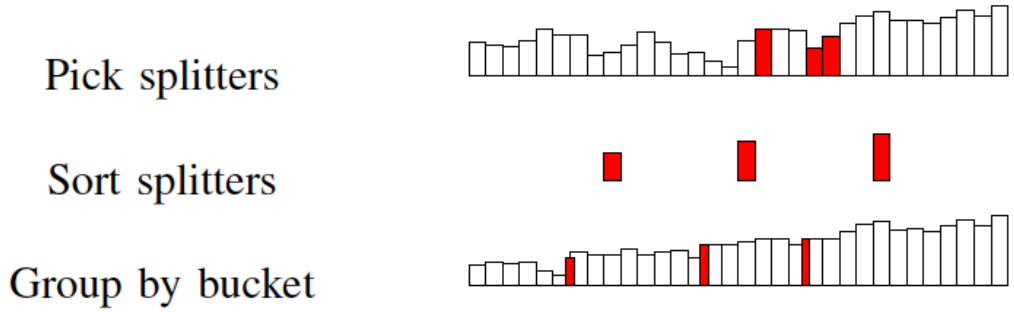
Implementation Aspects

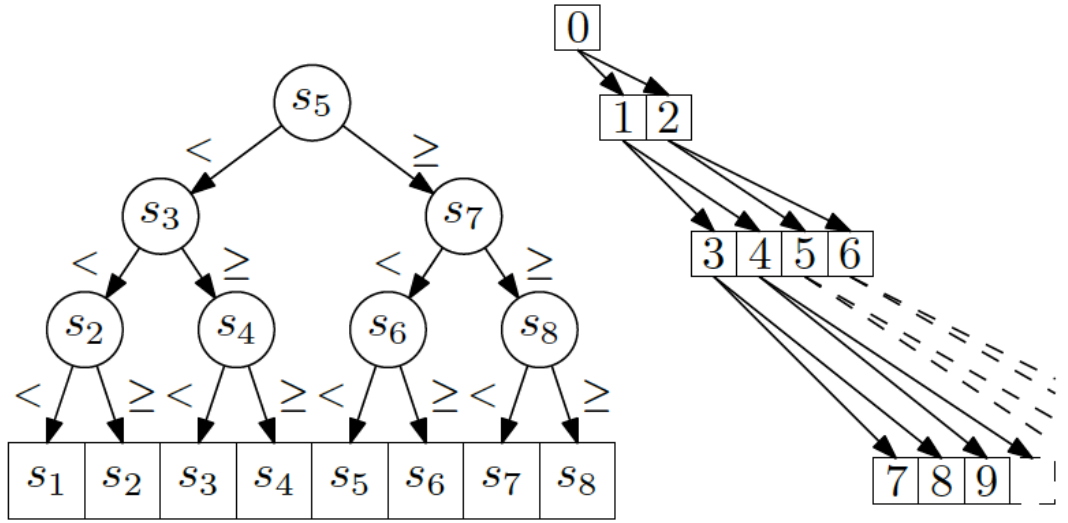
- We only copy elements of the bucket we are interested in;
- In case of identical splitter elements, they are placed in an *equality bucket*;
- If target rank is in an *equality bucket*, the algorithm can terminate early by returning lower bound;
- For sorting the splitters, small input datasets, and the lowest recursion level a *bitonic sort* in shared memory is used;



Implementation Aspects

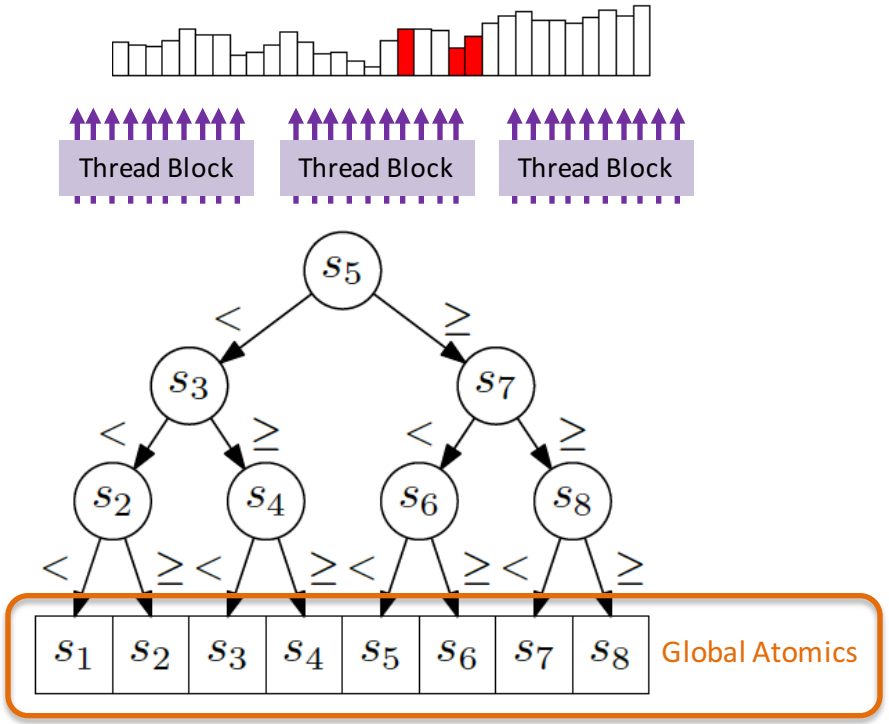
- We only copy elements of the bucket we are interested in;
- In case of identical splitter elements, they are placed in an *equality bucket*;
- If target rank is in an *equality bucket*, the algorithm can terminate early by returning lower bound;
- For sorting the splitters, small input datasets, and the lowest recursion level a *bitonic sort* in shared memory is used;
- Use a *binary search tree* to sort elements into the buckets;





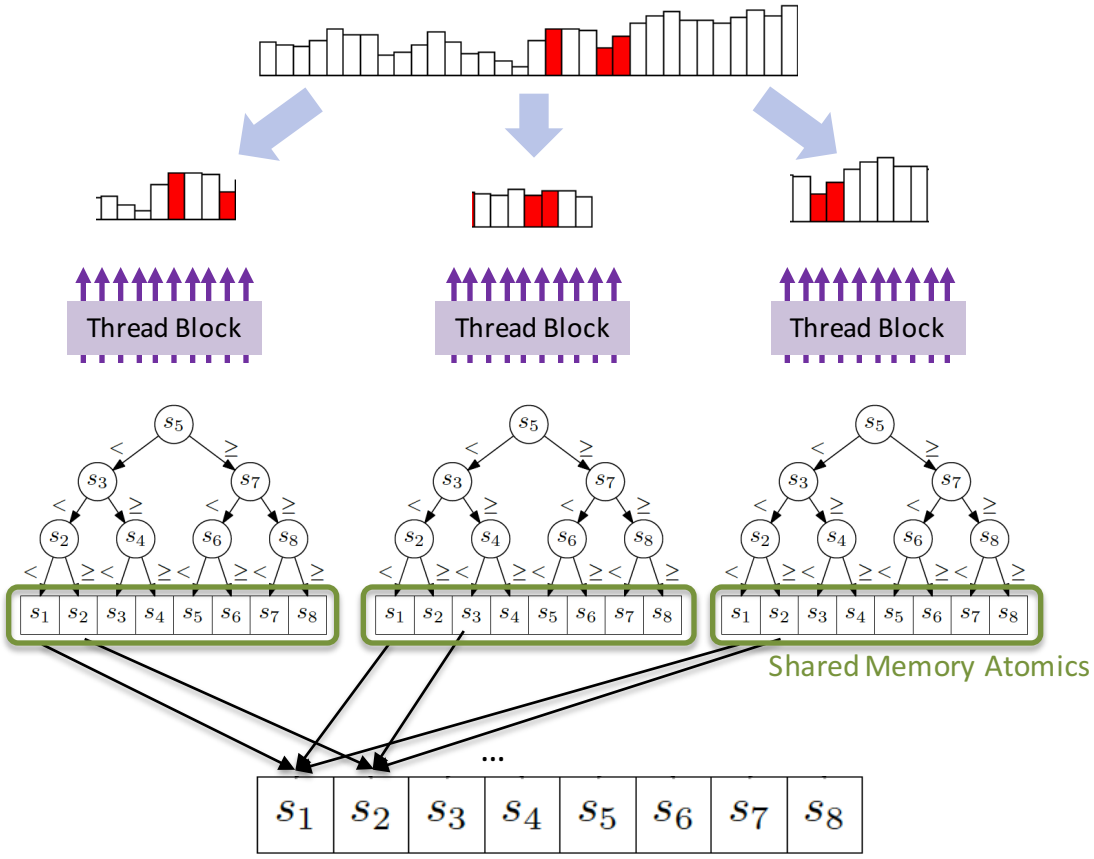
Parallelization & Communication

Global Memory Atomics



- Run SampleSelect using all resources on complete data set;
- Use global atomics to generate bucket counts;

Shared Memory Atomics



- Split data set into chunks, assign to thread blocks;
- Each thread block runs bucket count on its data;
- Use a global reduction to get global bucket counts;

Experiment Setup

- 2 distinct GPU architectures
- Input datasets with 2^{16} to 2^{28} elements
- *d= 1, 16, 128, 1024, n* distinct values
- All results averaged over 10 runs
- Single precision input data

- Comparison against QuickSelect kernel
- QuickSelect and SampleSelect have same performance optimization level
- Correctness check using C++ std::nth_element

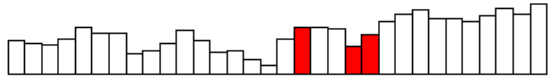
	NVIDIA K40	NVIDIA V100
Architecture	Kepler	Volta
DP Performance	1.2 TFLOPs	7 TFLOPs
SP Performance	3.5 TFLOPs	14 TFLOPs
HP Performance	–	112 TFLOPs
SMs	13	80
Operating Freq.	0.75 GHz	1.53 GHz
Mem. Capacity	5 GB	16 GB
Mem. Bandwidth	208 GB/s	900 GB/s
Sustained BW	146 GB/s	742 GB/s
L2 Cache Size	1.5 MB	6 MB
L1 Cache Size	64 KB	128 KB
	2013	*2017*

#44@TOP500

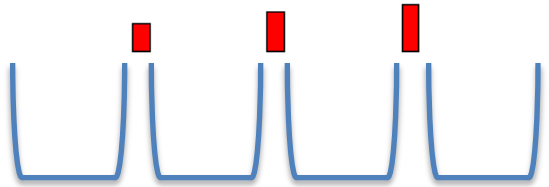
#1@TOP500

Kernel Optimization I: **Bucket Count**

Pick splitters



Sort splitters



Splitters separate buckets

Kernel Optimization I: **Bucket Count**

NVIDIA K40

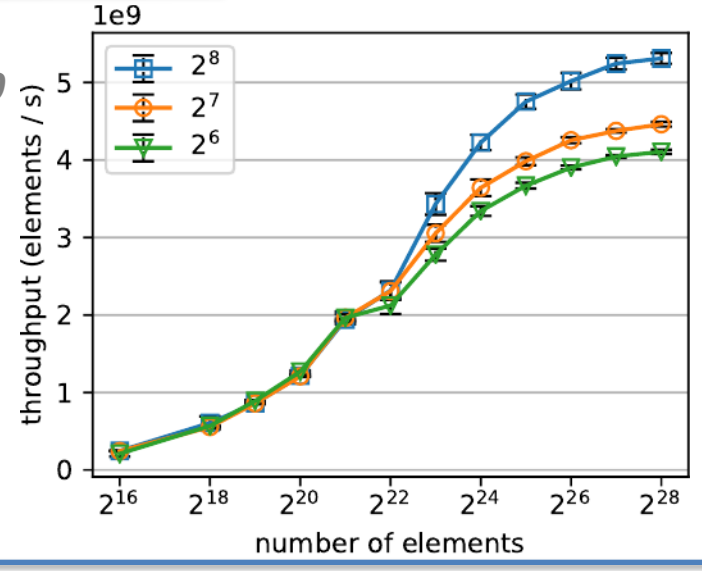
NVIDIA V100

Global memory atomics

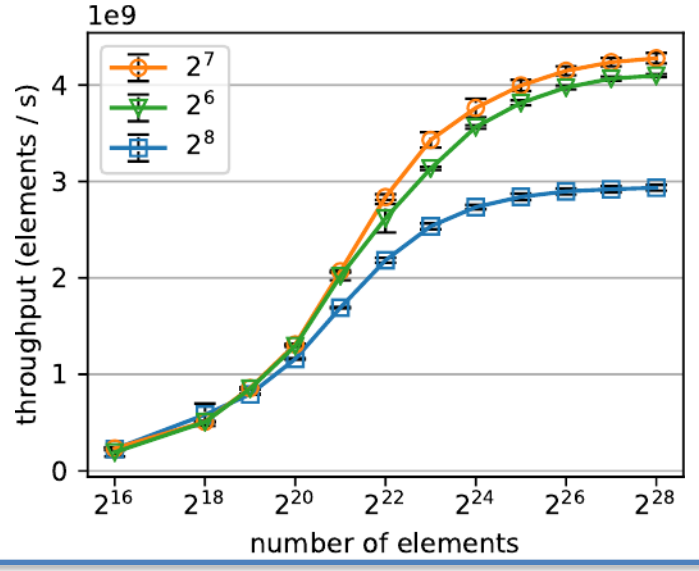
Shared memory atomics

Kernel Optimization I: **Bucket Count**

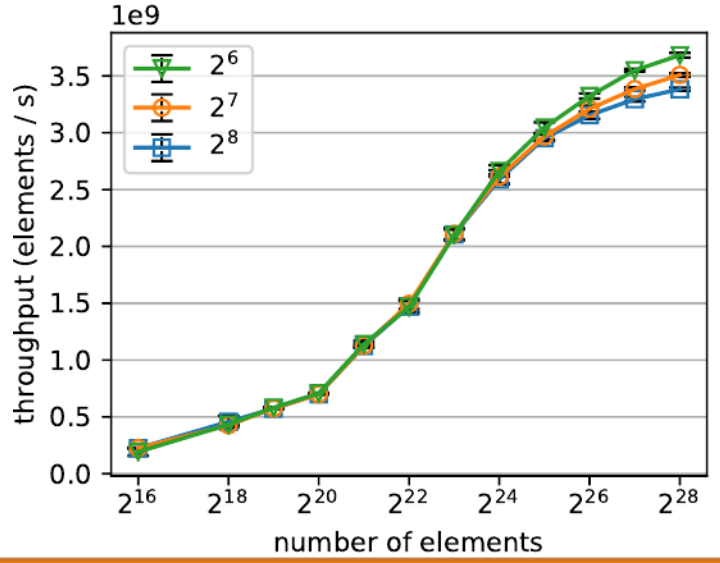
NVIDIA K40



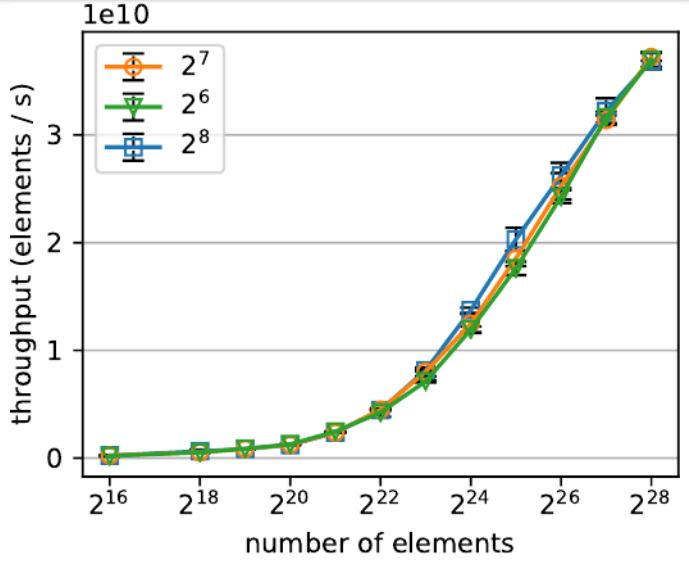
NVIDIA V100



Global memory atomics

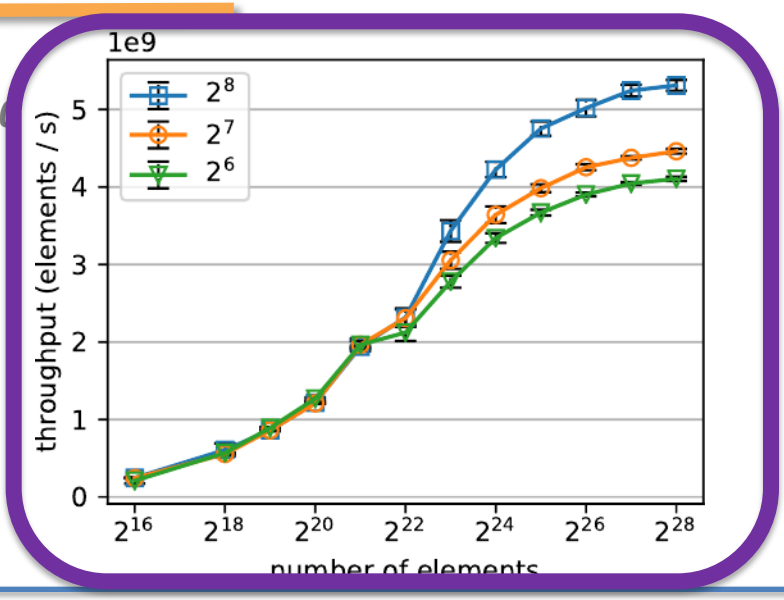


Shared memory atomics

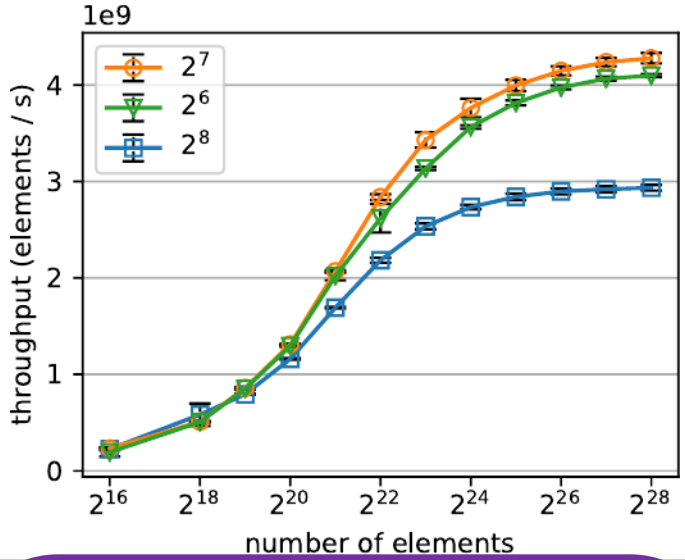


Kernel Optimization I: **Bucket Count**

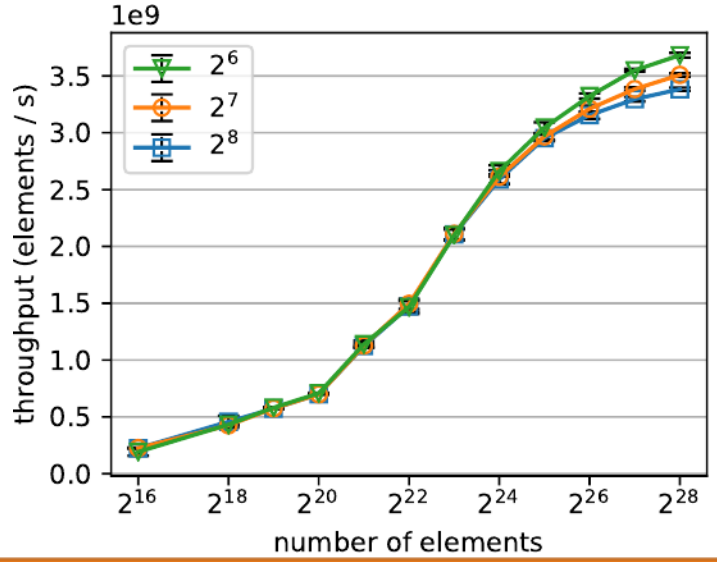
NVIDIA K4



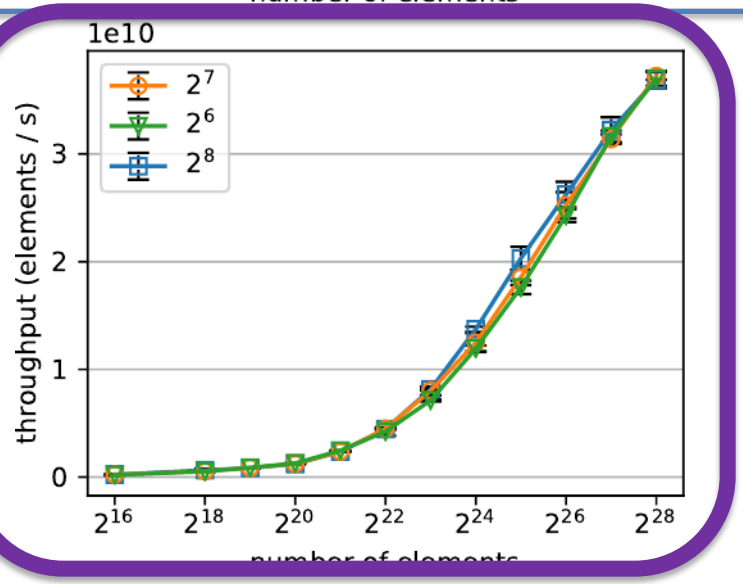
NVIDIA V100



Global memory atomics

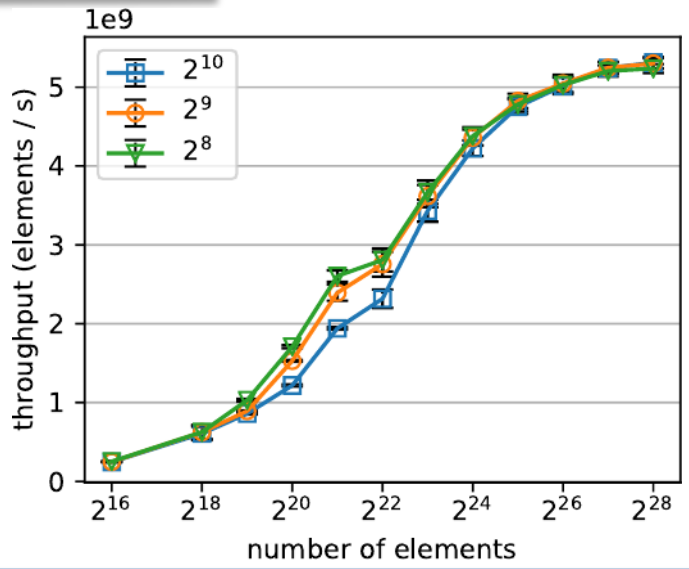


Shared memory atomics

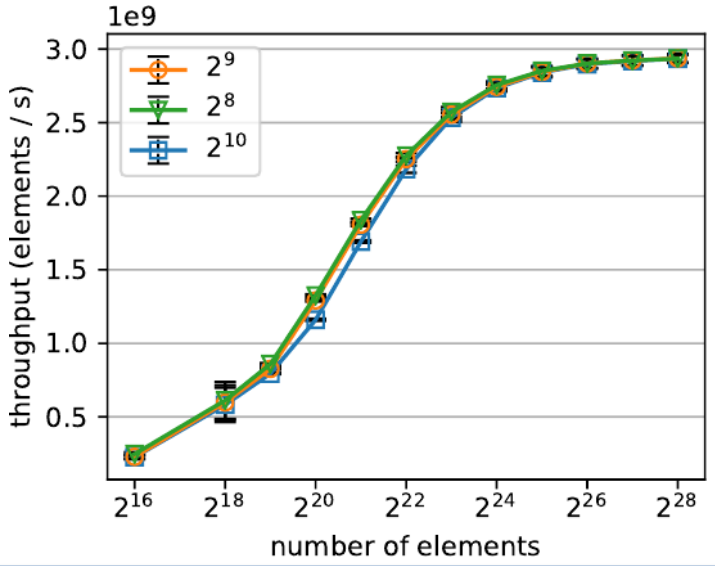


Kernel Optimization II: **#Threads per Block**

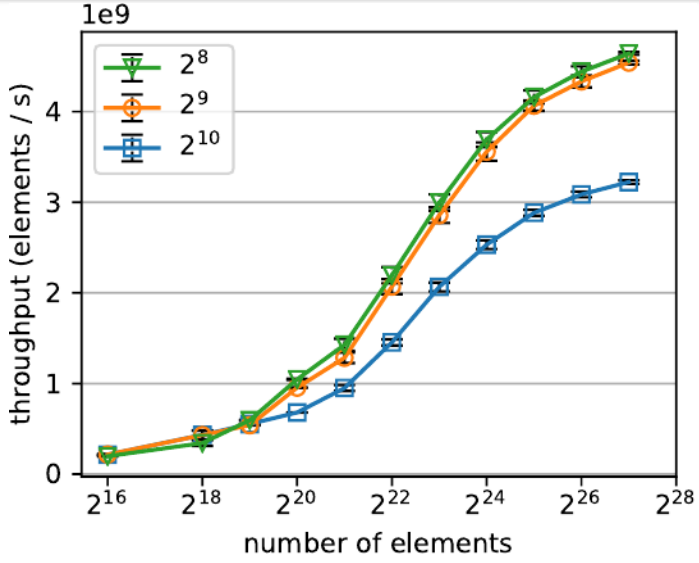
NVIDIA K40



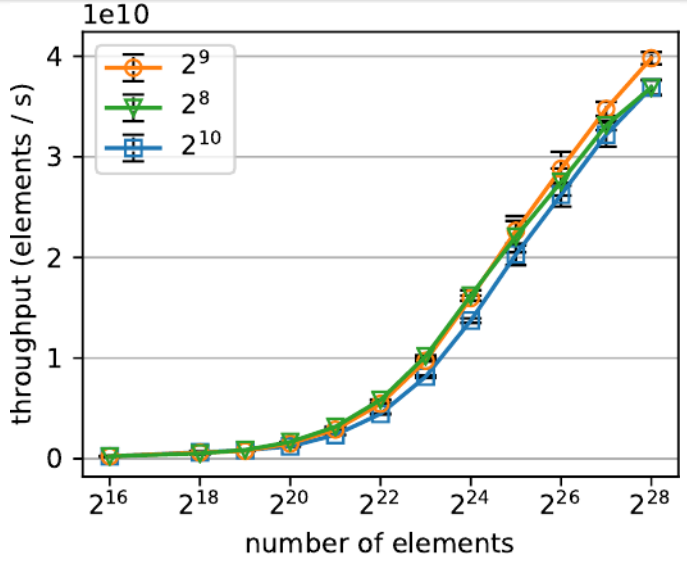
NVIDIA V100



Global memory atomics

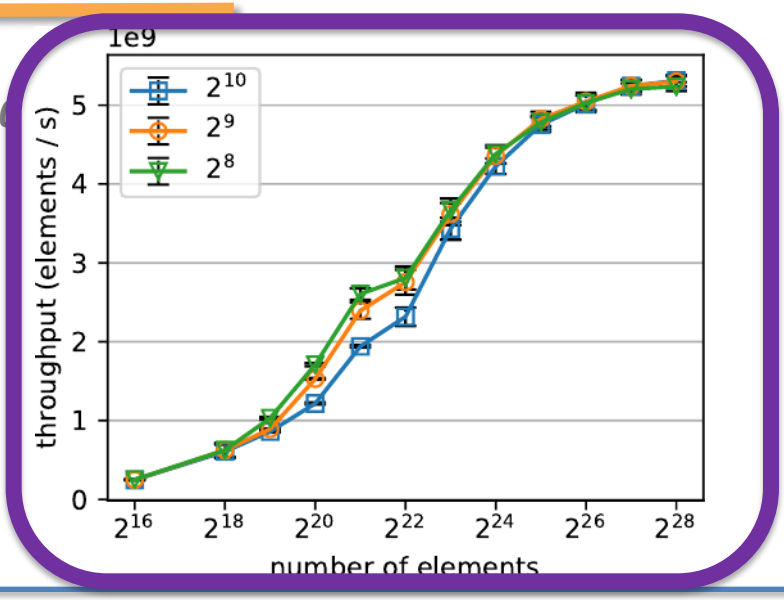


Shared memory atomics

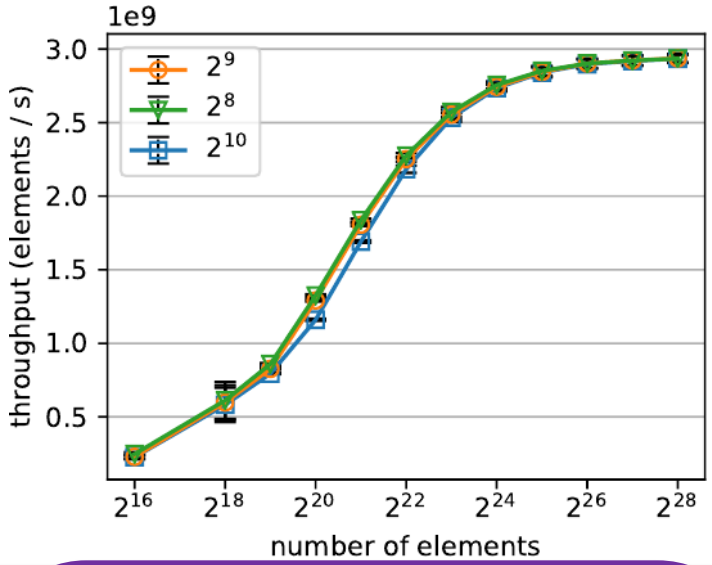


Kernel Optimization II: **#Threads per Block**

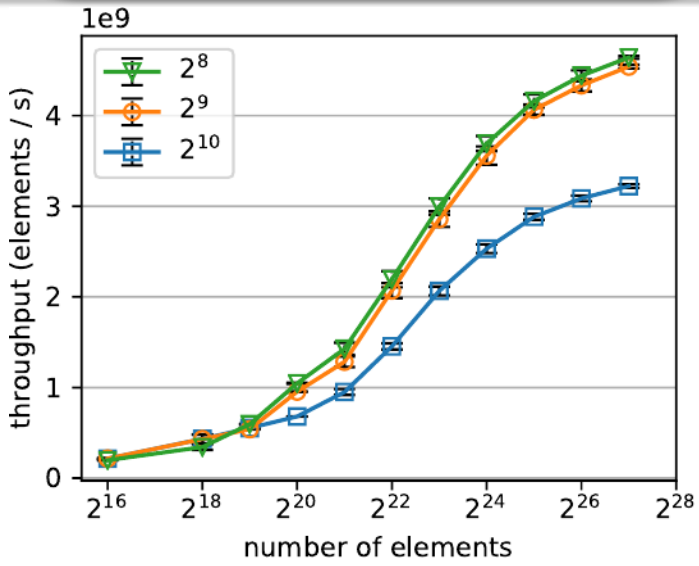
NVIDIA K40



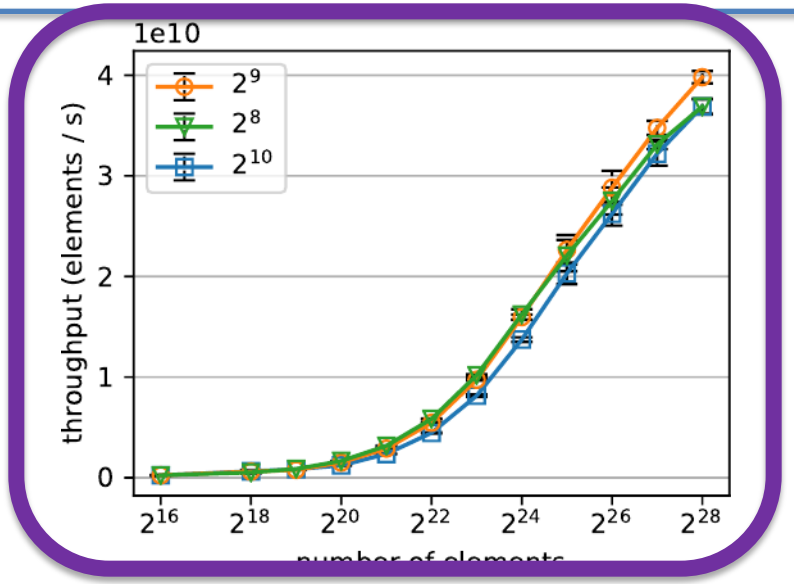
NVIDIA V100



Global memory atomics

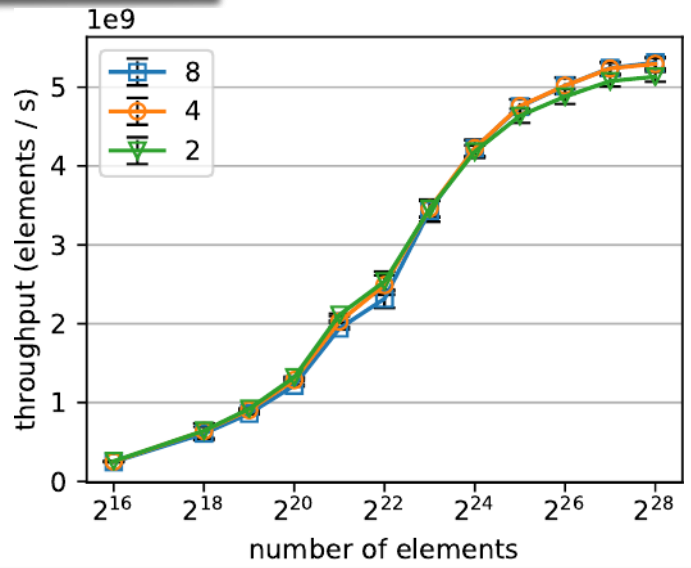


Shared memory atomics

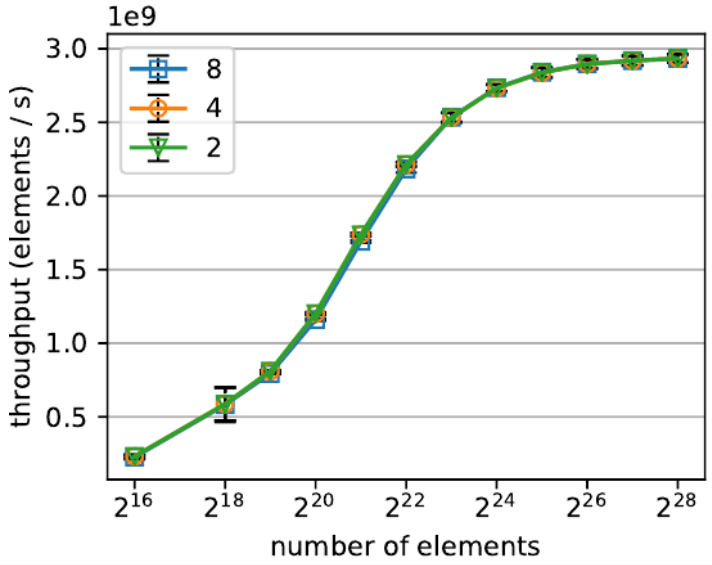


Kernel Optimization III: **Loop Unrolling**

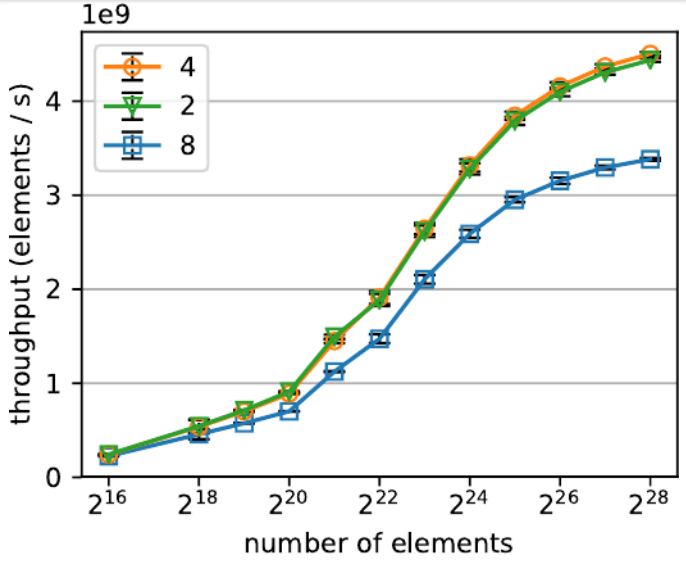
NVIDIA K40



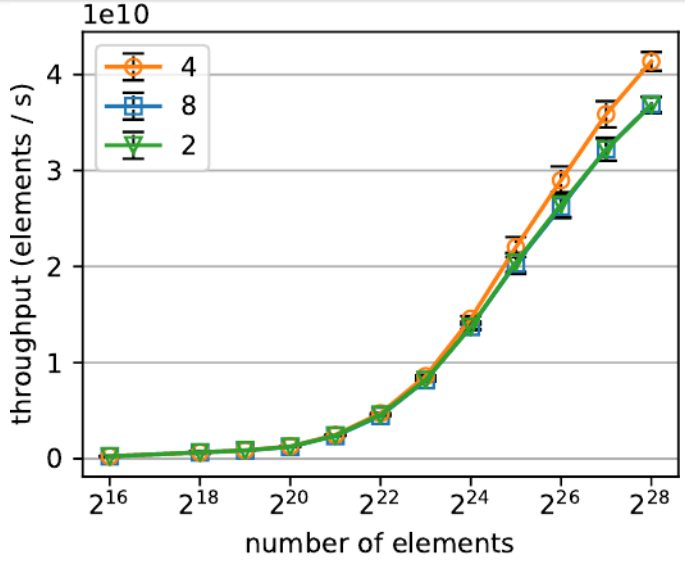
NVIDIA V100



Global memory atomics

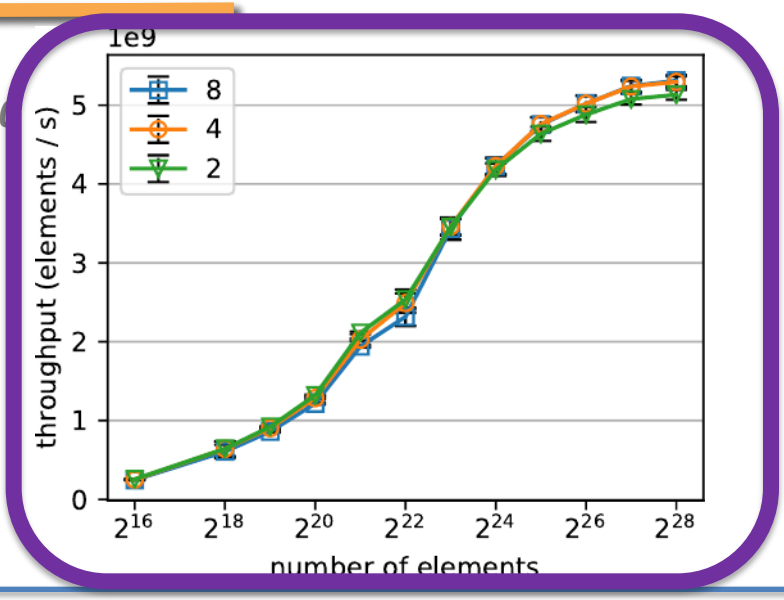


Shared memory atomics

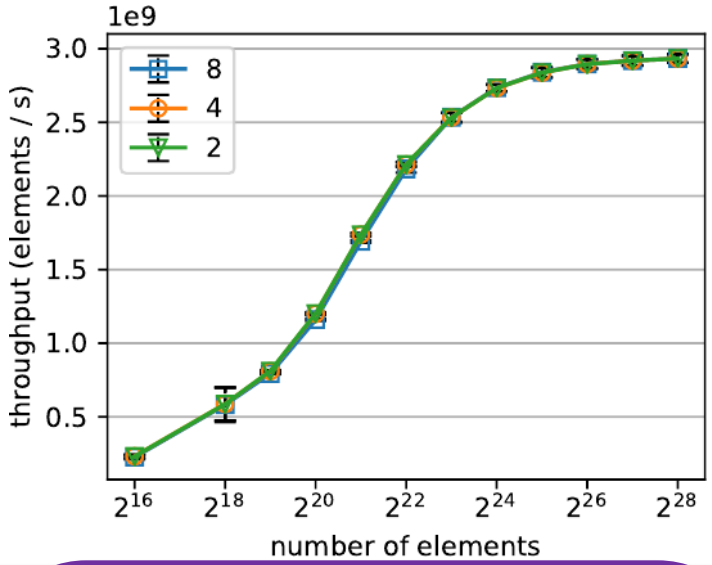


Kernel Optimization III: **Loop Unrolling**

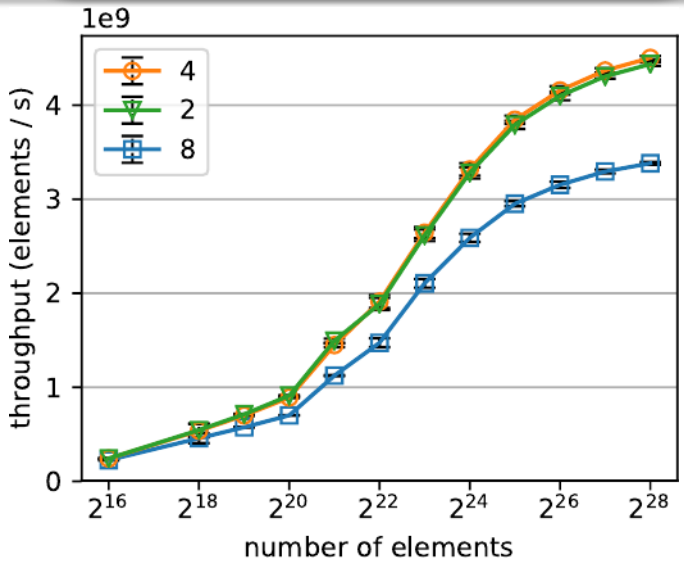
NVIDIA K40



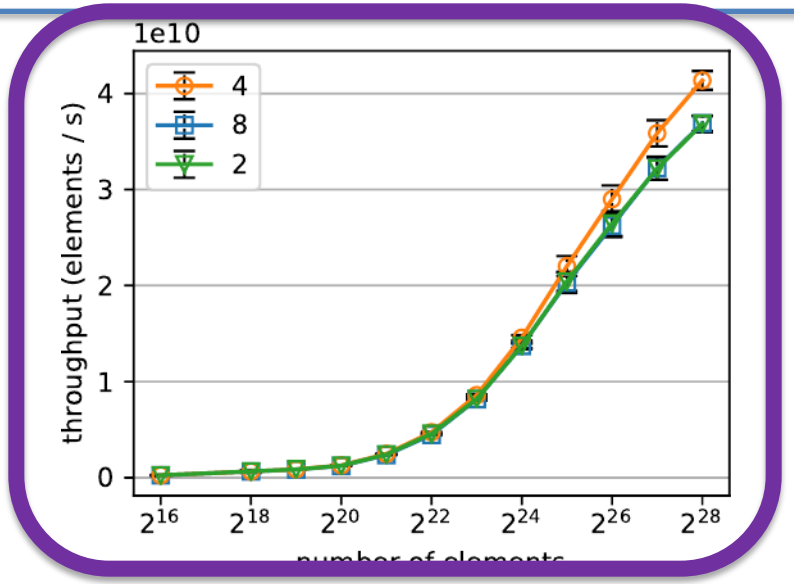
NVIDIA V100



Global memory atomics



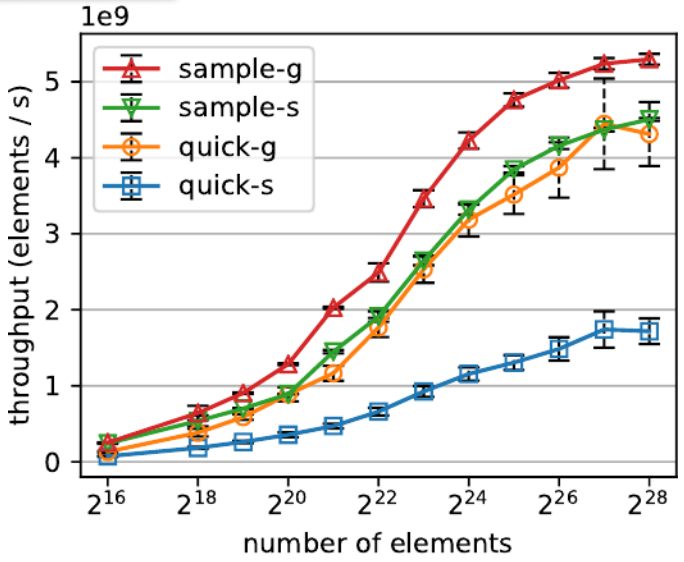
Shared memory atomics



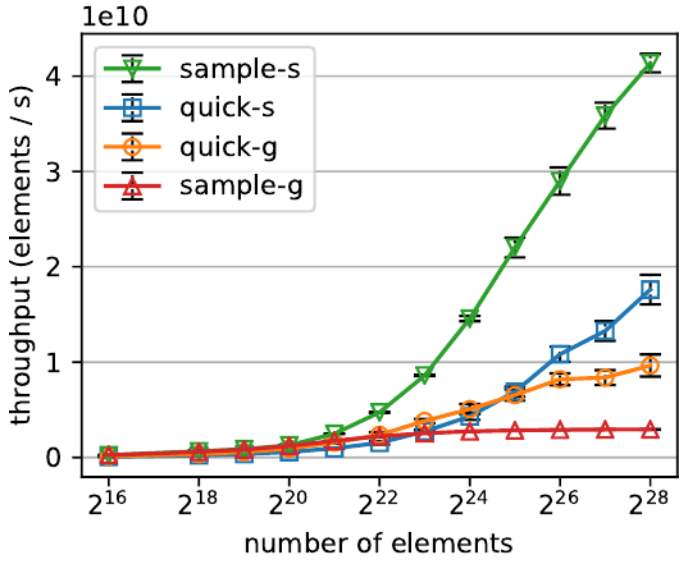
Kernel Optimization IV: **Global vs. Local Atomics**

-g : global memory atomics
-s: shared memory atomics

NVIDIA K40



NVIDIA V100

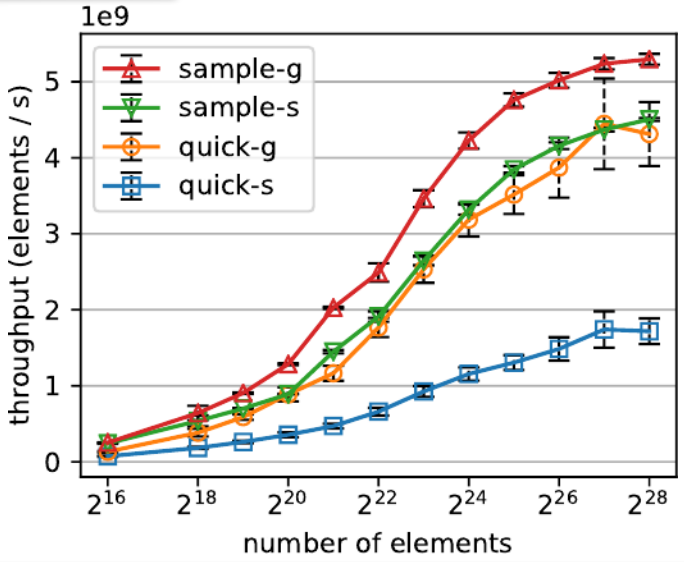


Larger performance variation for QuickSelect as we are more likely to run into the *"Worst-Case"* performance.

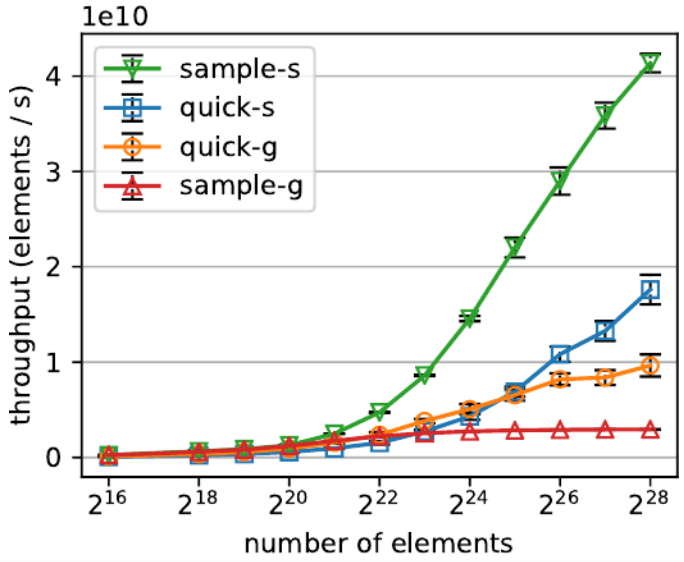
Kernel Optimization IV: **Global vs. Local Atomics**

-g : global memory atomics
-s: shared memory atomics

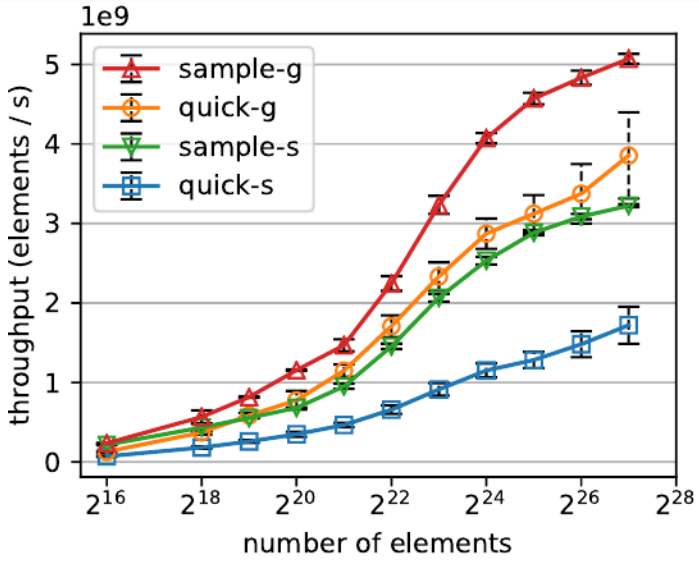
NVIDIA K40

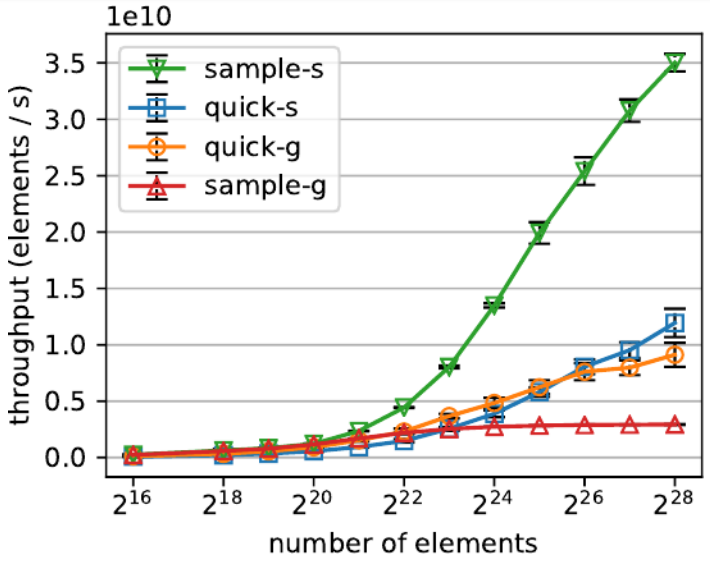


NVIDIA V100



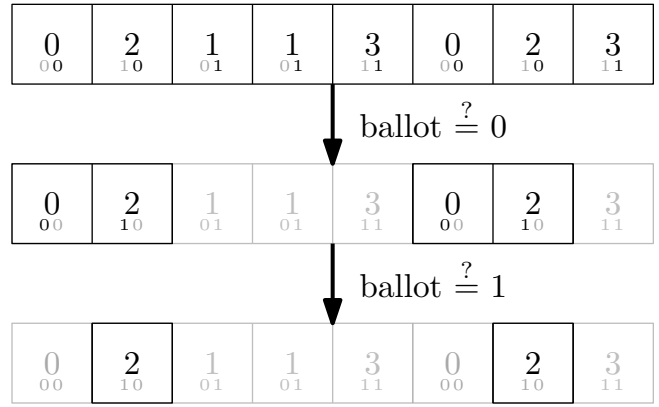
double precision





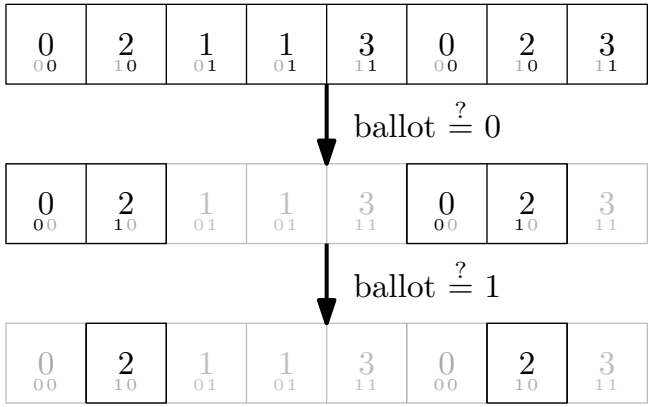
Kernel Optimization V: **Element Repetition**

Idea: use warp aggregations to mitigate the performance impact from atomic collisions.

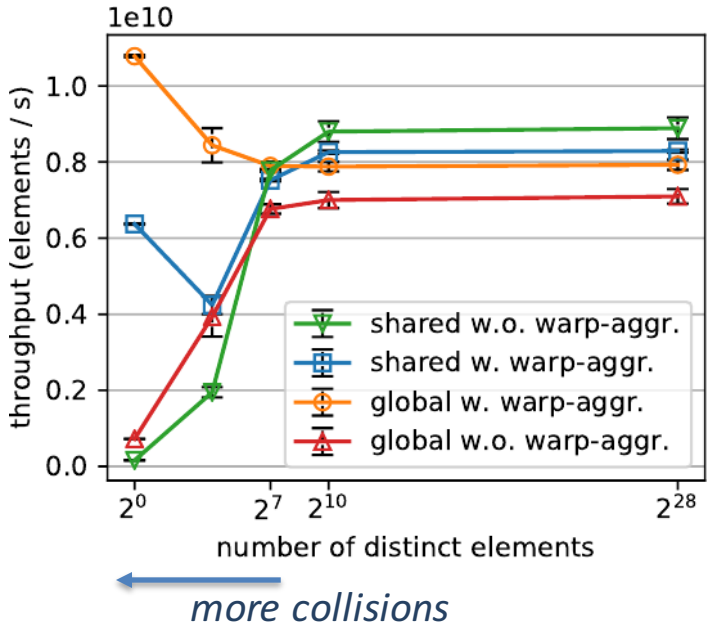


Kernel Optimization V: **Element Repetition**

Idea: use warp aggregations to mitigate the performance impact from atomic collisions.

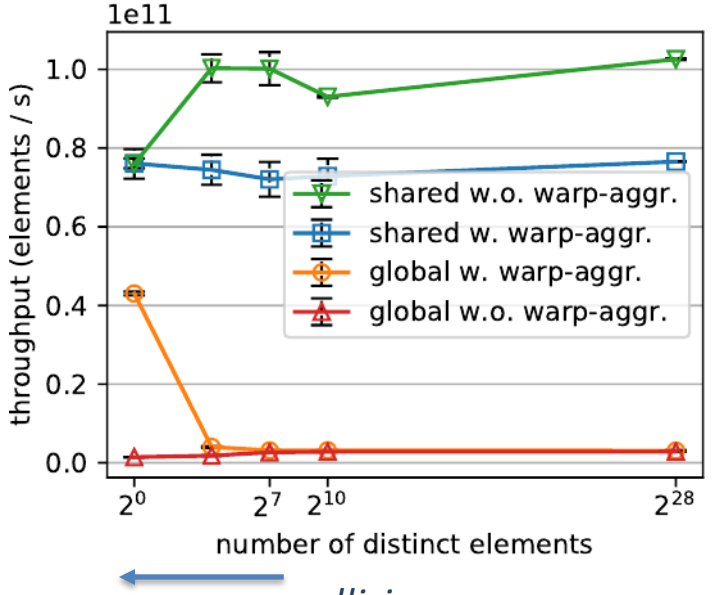


NVIDIA K40



more collisions

NVIDIA V100

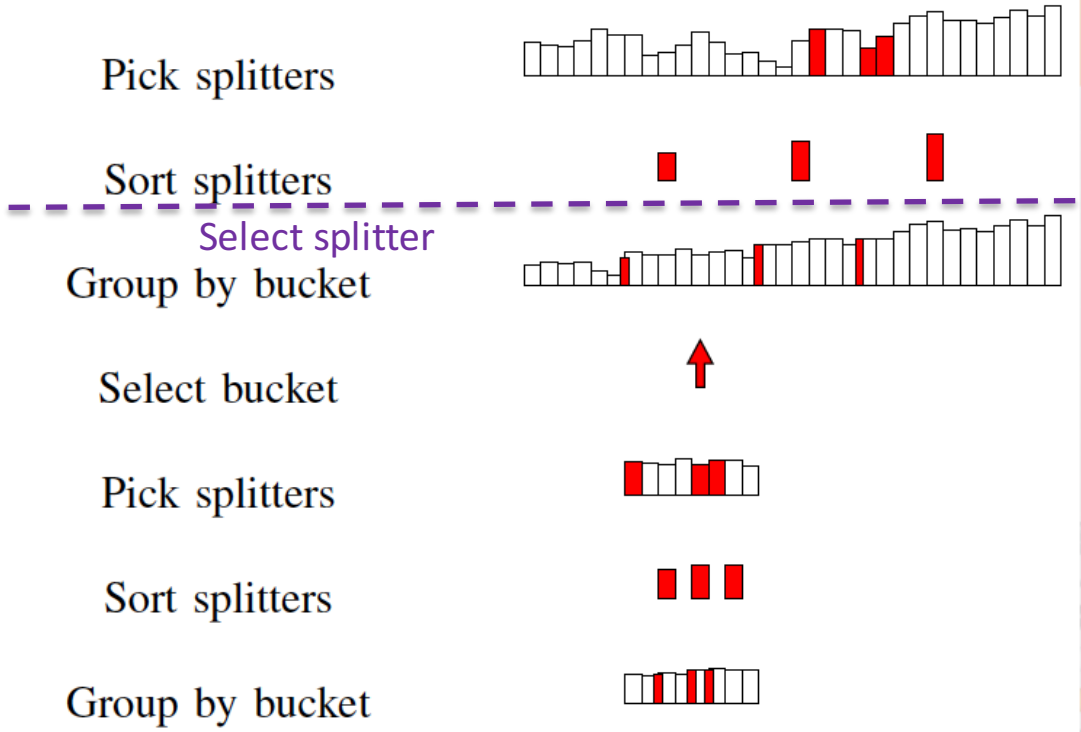


more collisions

Approximate Selection

We do not descent to the lowest level of the recursion tree, but limit to one single bucket selection.

- Accuracy depends on the number of splitters vs. dataset size
- Accuracy independent of value distribution (works on ranks, only)



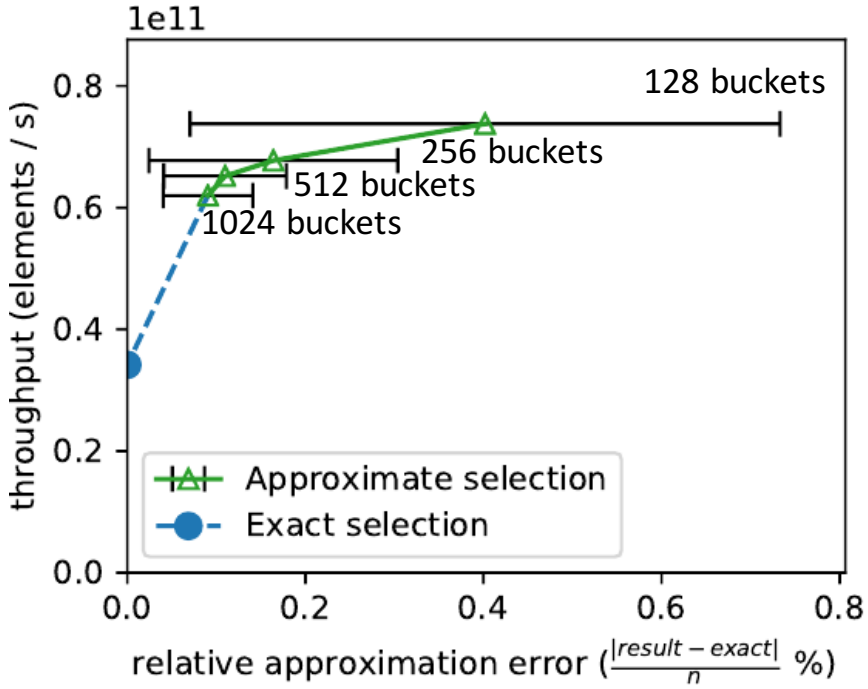
Approximate Selection

We do not descent to the lowest level of the recursion tree, but limit to one single bucket selection.

- Accuracy depends on the number of splitters vs. dataset size
- Accuracy independent of value distribution (works on ranks, only)

Test problem:

- 2^{28} uniformly distributed single precision values
- Approximate selection uses 1 level only
- We report statistics over 10 runs



Summary and Outlook

http://bit.ly/SampleSelectGPU

- **SampleSelect kernel much faster than QuickSelect**
- **36% (single) 48% (double) of experimental peak memory bandwidth on NVIDIA V100**
- **Approximate selection >2x faster than exact selection**

Tobias Ribizel

Karlsruher Institut für Technologie

THE UNIVERSITY OF TENNESSEE KNOXVILLE

HELMHOLTZ RESEARCH FOR GRAND CHALLENGES

Helmholtz Impuls und Vernetzungsfond
VH-NG-1241

Summary and Outlook

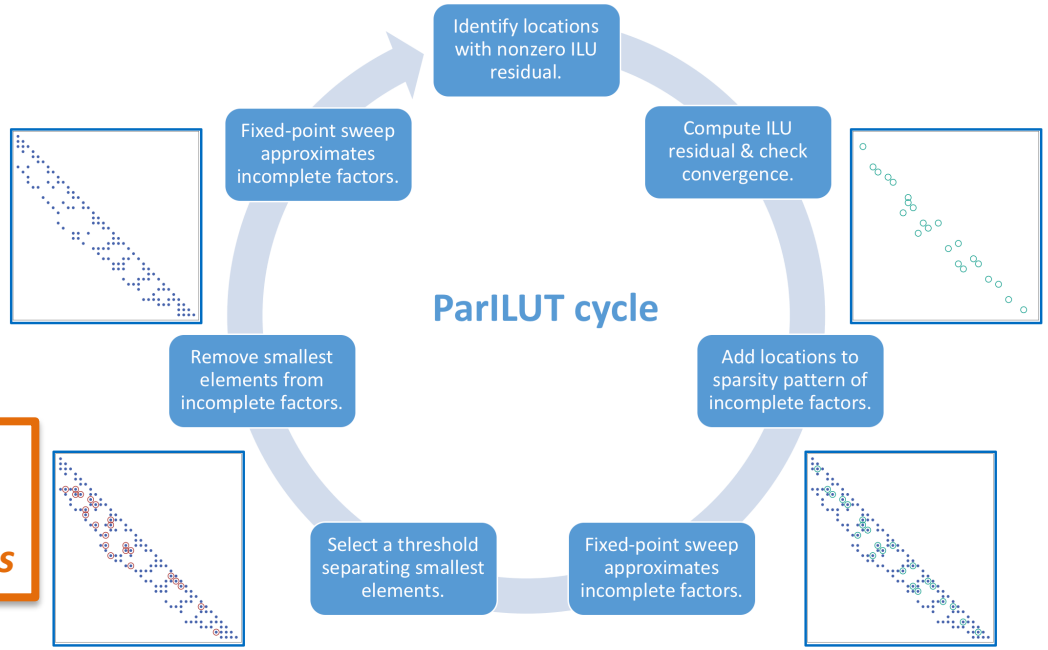
http://bit.ly/SampleSelectGPU

- **SampleSelect kernel much faster than QuickSelect**
- **36% (single) 48% (double) of experimental peak memory bandwidth on NVIDIA V100**
- **Approximate selection >2x faster than exact selection**

Tobias Ribizel

Karlsruher Institut für Technologie

THE UNIVERSITY OF TENNESSEE KNOXVILLE



What is approximate selection good for?

IPDPS Main Track, Tuesday 1:30
Session 6: GPU Computing:
ParILUT - A Parallel Threshold ILU for GPUs

HELMHOLTZ
RESEARCH FOR GRAND CHALLENGES

Helmholtz Impuls und Vernetzungsfond
VH-NG-1241