The SIAM Conference on Parallel Processing for Scientific Computing (SIAM PP)
February 14th, 2020 | Seattle, Washington, USA

Approximate and Exact (Multi-)Selection on GPUs

Tobias Ribizel[1], Hartwig Anzt[1,2]

[1]Karlsruhe Institute of Technology

[2]University of Tennessee

HPC-Seminar – FAU Erlangen
23.03.2021

ific Computing (SIAM PP)

Approximate and Exact (Multi-)Selection on GPUs

Tobias Ribizel[1], Hartwig Anzt[1,2]

[1]Karlsruhe Institute of Technology

[2]University of Tennessee

Karlsruher Institut für Technologie

THE UNIVERSITY OF TENNESSEE KNOXVILLE

Selection Problem

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

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

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

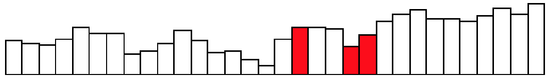
We do not necessarily need to sort the complete sequence!

- Statistics (Quantiles)
- Top-k selection
- Thresholds

General Approach: Partial Sorting

```
double select(data, rank) {
    if (size(data) <= base_case_size) {
        sort(data);
        return data[rank];
    }
    // select splitters
    splitters = pick_splitters(data);
    // compute bucket sizes n_i
    counts = count_buckets(data, splitters);
    // compute bucket ranks r_i
    offsets = prefix_sum(counts);
    // determine bucket containing rank
    bucket = lower_bound(offsets, rank);
    // recursive subcall
    data = extract_bucket(data, bucket);
    rank -= offsets[bucket];
    return select(data, rank);
}
```

Pick splitters

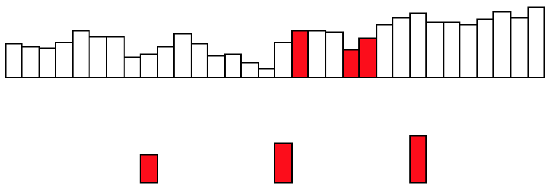


General Approach: Partial Sorting

```
double select(data, rank) {
    if (size(data) <= base_case_size) {
        sort(data);
        return data[rank];
    }
    // select splitters
    splitters = pick_splitters(data);
    // compute bucket sizes n_i
    counts = count_buckets(data, splitters);
    // compute bucket ranks r_i
    offsets = prefix_sum(counts);
    // determine bucket containing rank
    bucket = lower_bound(offsets, rank);
    // recursive subcall
    data = extract_bucket(data, bucket);
    rank -= offsets[bucket];
    return select(data, rank);
}
```

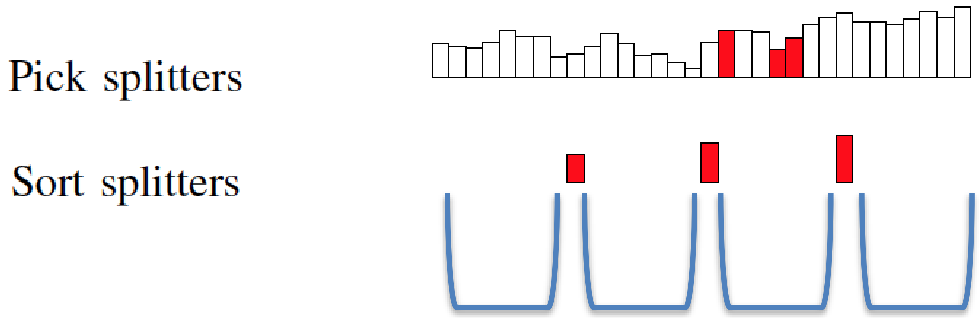
Pick splitters

Sort splitters



General Approach: Partial Sorting

```
double select(data, rank) {
    if (size(data) <= base_case_size) {
        sort(data);
        return data[rank];
    }
    // select splitters
    splitters = pick_splitters(data);
    // compute bucket sizes n_i
    counts = count_buckets(data, splitters);
    // compute bucket ranks r_i
    offsets = prefix_sum(counts);
    // determine bucket containing rank
    bucket = lower_bound(offsets, rank);
    // recursive subcall
    data = extract_bucket(data, bucket);
    rank -= offsets[bucket];
    return select(data, rank);
}
```



Splitters separate buckets

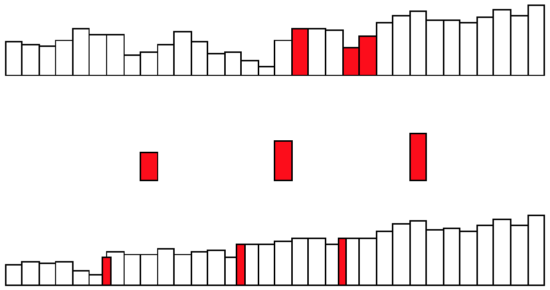
General Approach: Partial Sorting

```
double select(data, rank) {
    if (size(data) <= base_case_size) {
        sort(data);
        return data[rank];
    }
    // select splitters
    splitters = pick_splitters(data);
    // compute bucket sizes n_i
    counts = count_buckets(data, splitters);
    // compute bucket ranks r_i
    offsets = prefix_sum(counts);
    // determine bucket containing rank
    bucket = lower_bound(offsets, rank);
    // recursive subcall
    data = extract_bucket(data, bucket);
    rank -= offsets[bucket];
    return select(data, rank);
}
```

Pick splitters

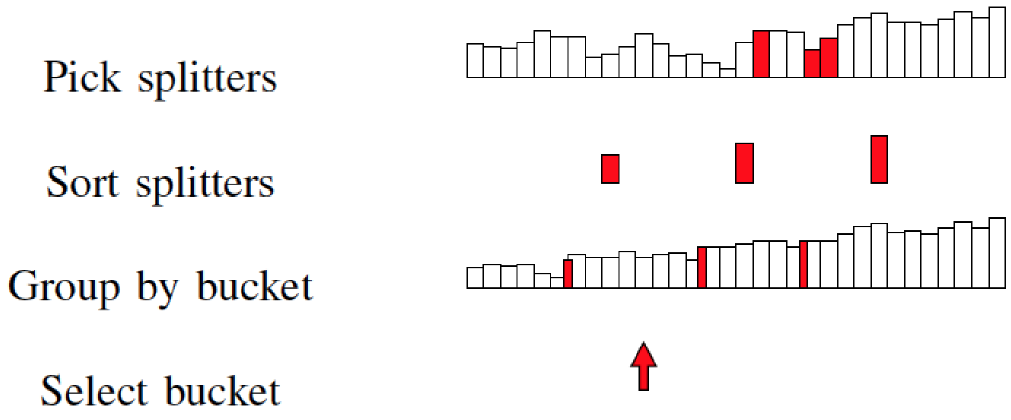
Sort splitters

Group by bucket



General Approach: Partial Sorting

```
double select(data, rank) {
    if (size(data) <= base_case_size) {
        sort(data);
        return data[rank];
    }
    // select splitters
    splitters = pick_splitters(data);
    // compute bucket sizes n_i
    counts = count_buckets(data, splitters);
    // compute bucket ranks r_i
    offsets = prefix_sum(counts);
    // determine bucket containing rank
    bucket = lower_bound(offsets, rank);
    // recursive subcall
    data = extract_bucket(data, bucket);
    rank -= offsets[bucket];
    return select(data, rank);
}
```



General Approach: Partial Sorting

```
double select(data, rank) {
    if (size(data) <= base_case_size) {
        sort(data);
        return data[rank];
    }
    // select splitters
    splitters = pick_splitters(data);
    // compute bucket sizes n_i
    counts = count_buckets(data, splitters);
    // compute bucket ranks r_i
    offsets = prefix_sum(counts);
    // determine bucket containing rank
    bucket = lower_bound(offsets, rank);
    // recursive subcall
    data = extract_bucket(data, bucket);
    rank -= offsets[bucket];
    return select(data, rank);
}
```

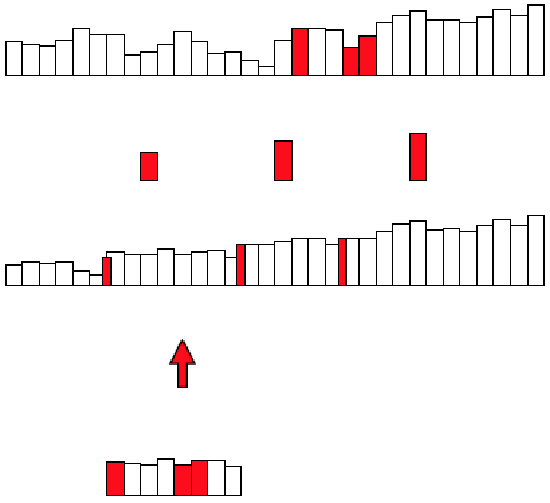
Pick splitters

Sort splitters

Group by bucket

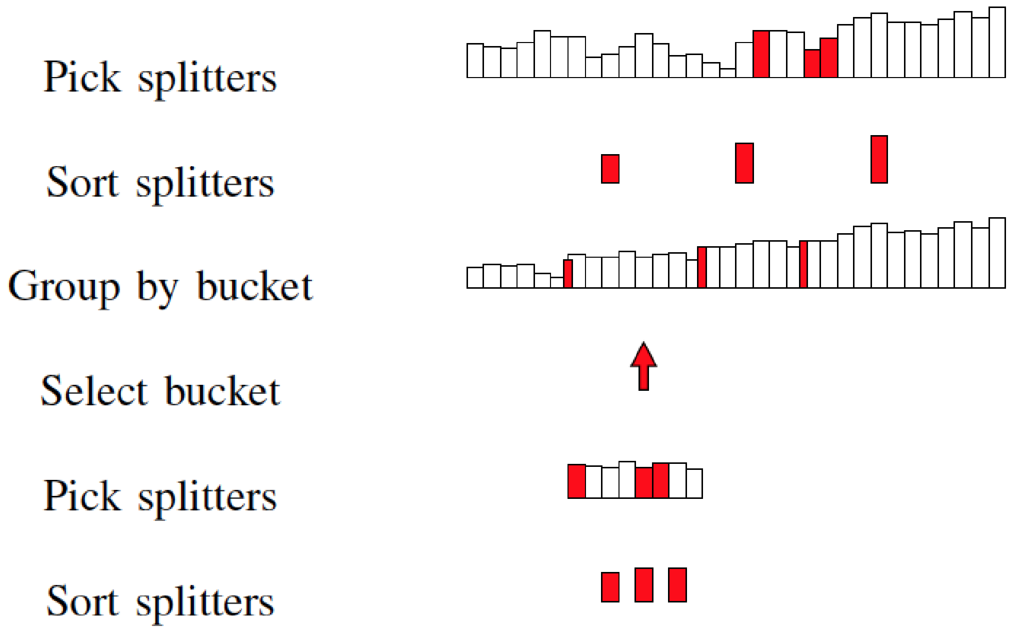
Select bucket

Pick splitters



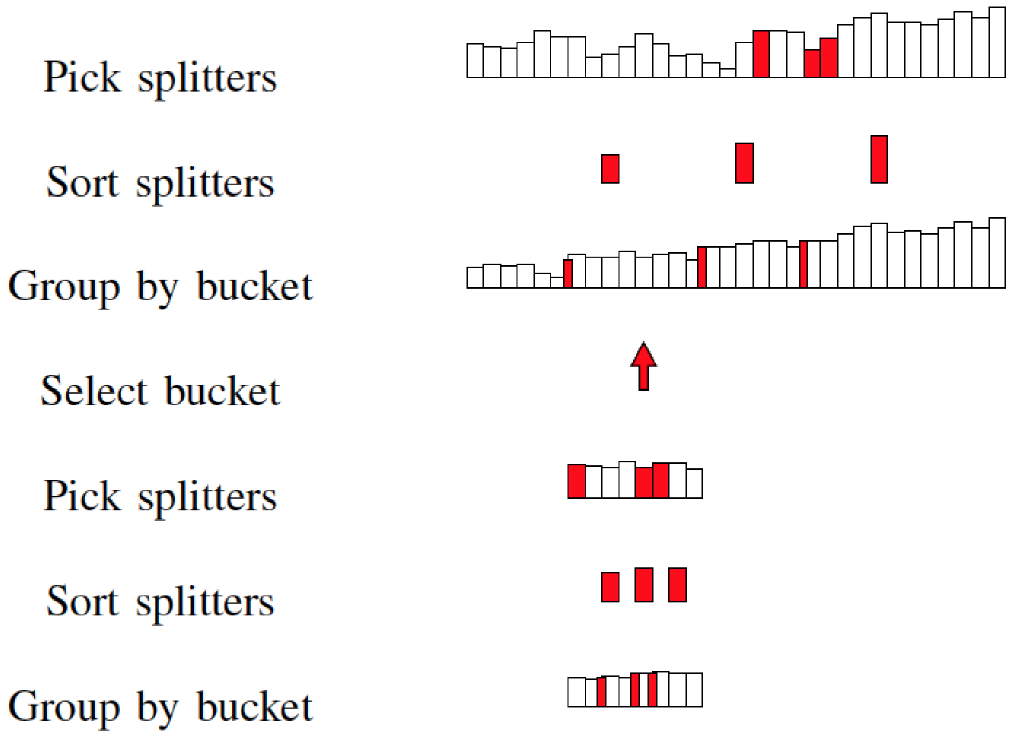
General Approach: Partial Sorting

```
double select(data, rank) {
    if (size(data) <= base_case_size) {
        sort(data);
        return data[rank];
    }
    // select splitters
    splitters = pick_splitters(data);
    // compute bucket sizes n_i
    counts = count_buckets(data, splitters);
    // compute bucket ranks r_i
    offsets = prefix_sum(counts);
    // determine bucket containing rank
    bucket = lower_bound(offsets, rank);
    // recursive subcall
    data = extract_bucket(data, bucket);
    rank -= offsets[bucket];
    return select(data, rank);
}
```



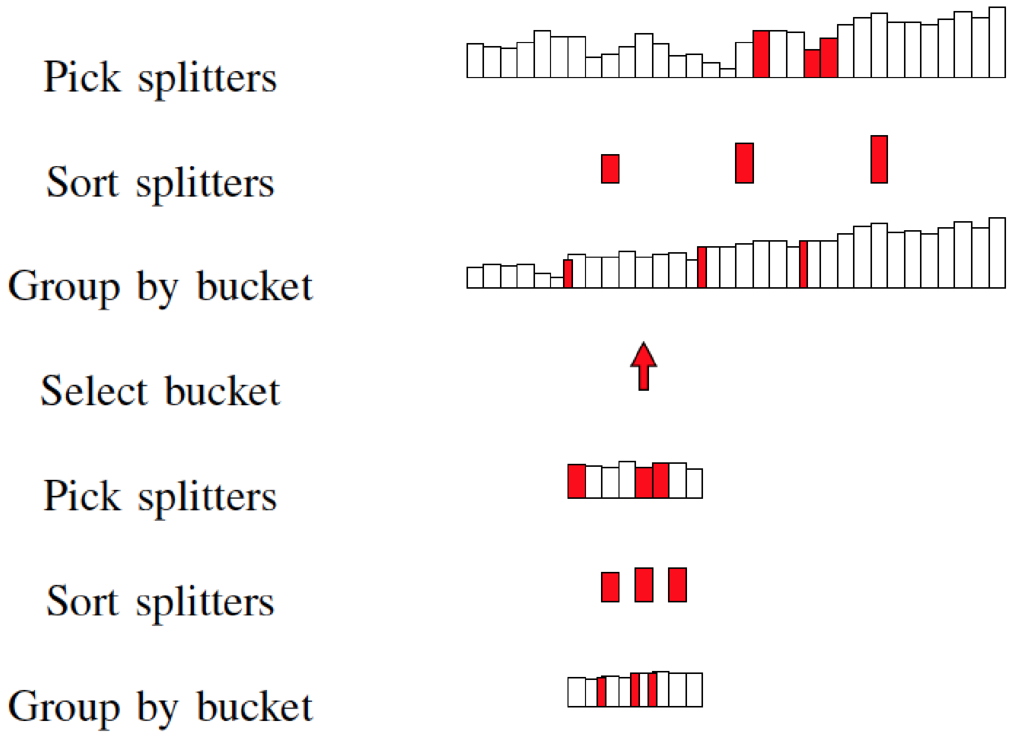
General Approach: Partial Sorting

```
double select(data, rank) {
    if (size(data) <= base_case_size) {
        sort(data);
        return data[rank];
    }
    // select splitters
    splitters = pick_splitters(data);
    // compute bucket sizes n_i
    counts = count_buckets(data, splitters);
    // compute bucket ranks r_i
    offsets = prefix_sum(counts);
    // determine bucket containing rank
    bucket = lower_bound(offsets, rank);
    // recursive subcall
    data = extract_bucket(data, bucket);
    rank -= offsets[bucket];
    return select(data, rank);
}
```



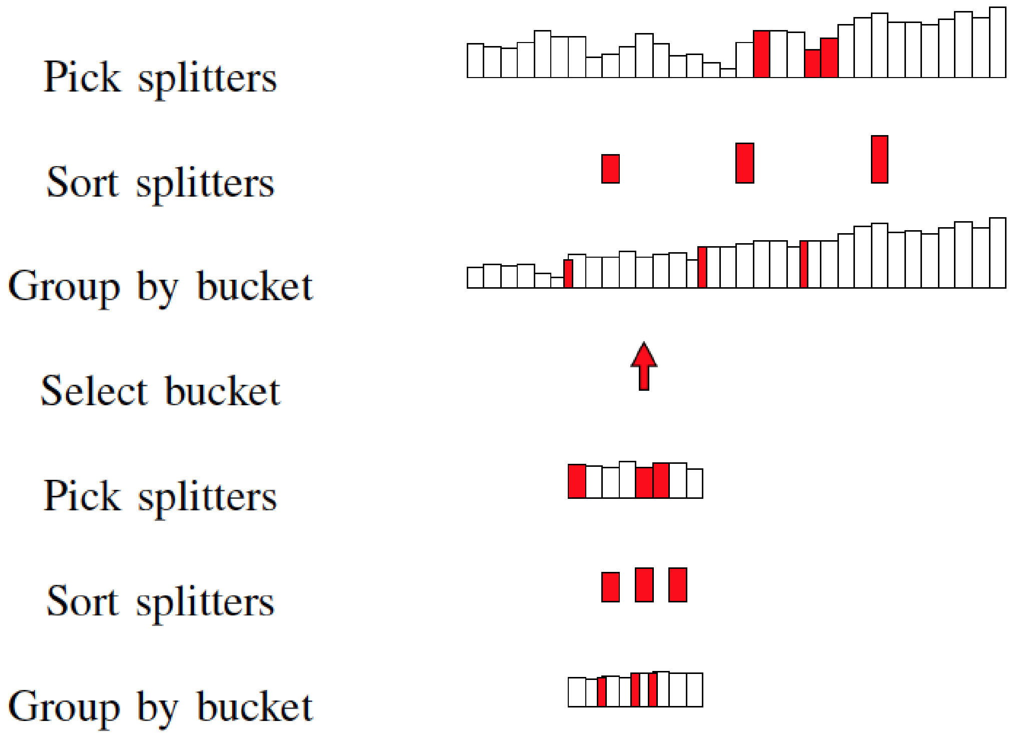
Implementation Aspects

```
double select(data, rank) {
    if (size(data) <= base_case_size) {
        sort(data);
        return data[rank];
    }
    // select splitters
    splitters = pick_splitters(data);
    // compute bucket sizes n_i
    counts = count_buckets(data, splitters);
    // compute bucket ranks r_i
    offsets = prefix_sum(counts);
    // determine bucket containing rank
    bucket = lower_bound(offsets, rank);
    // recursive subcall
    data = extract_bucket(data, bucket);
    rank -= offsets[bucket];
    return select(data, rank);
}
```



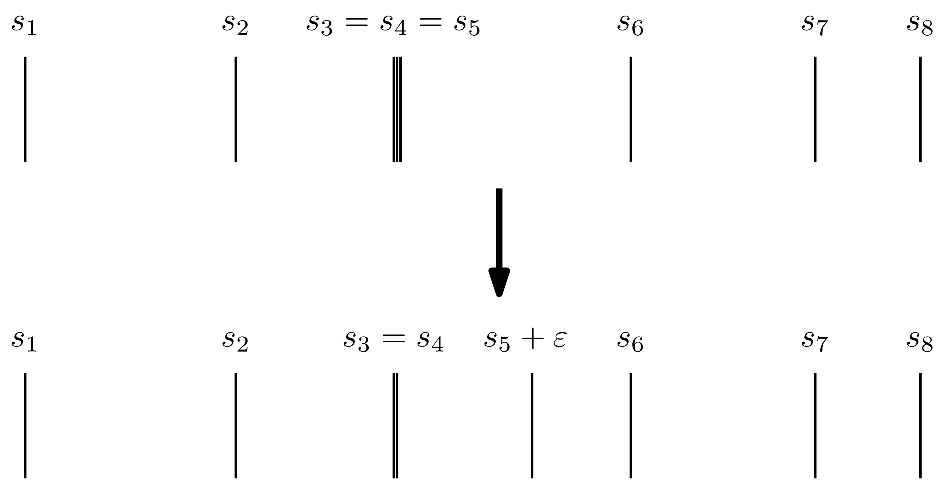
Implementation Aspects

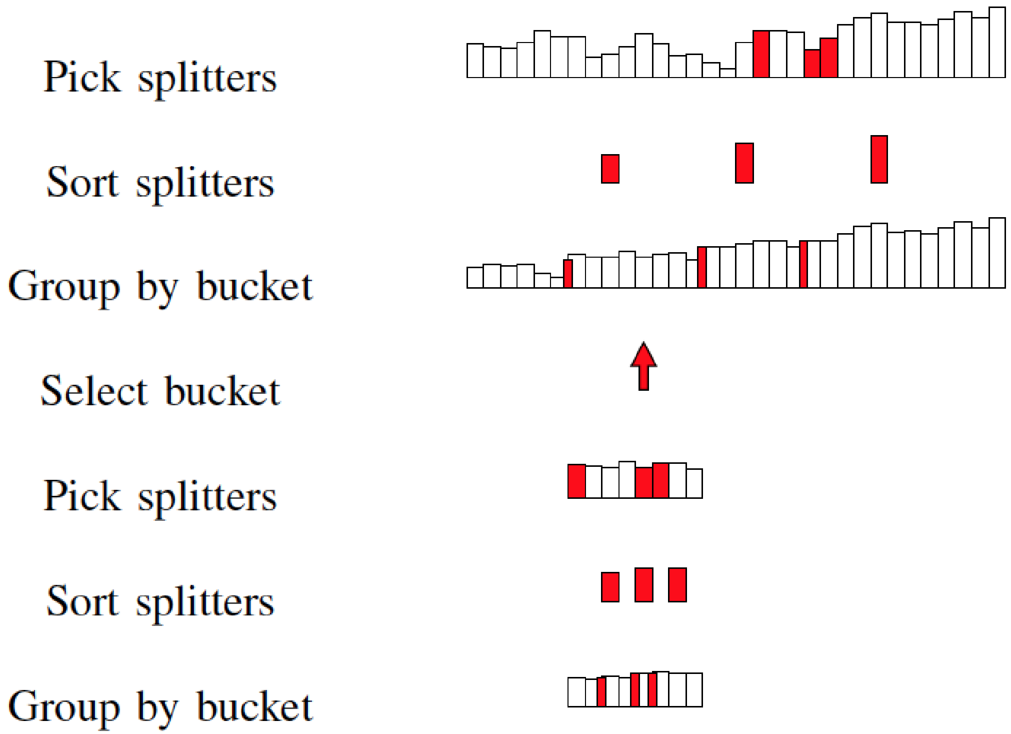
- We only copy elements of the buckets we are interested in;



Implementation Aspects

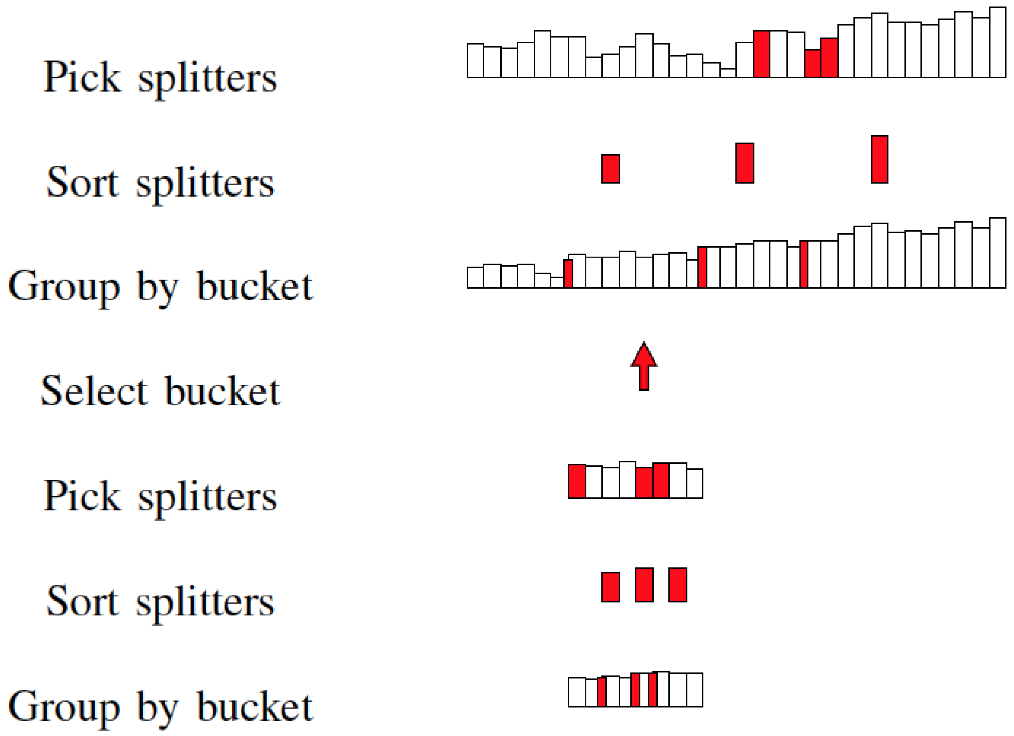
- We only copy elements of the buckets we are interested in;
- In case of identical splitter elements, they are placed in an *equality bucket*;
- If target rank is in an *equality bucket*, the algorithm can terminate early by returning lower bound;





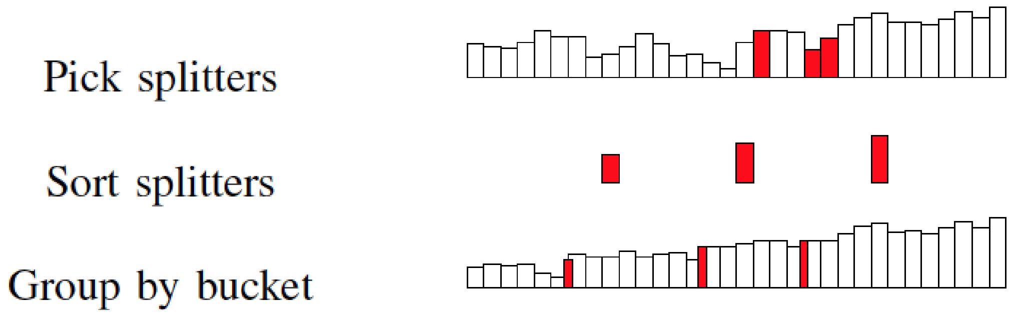
Implementation Aspects

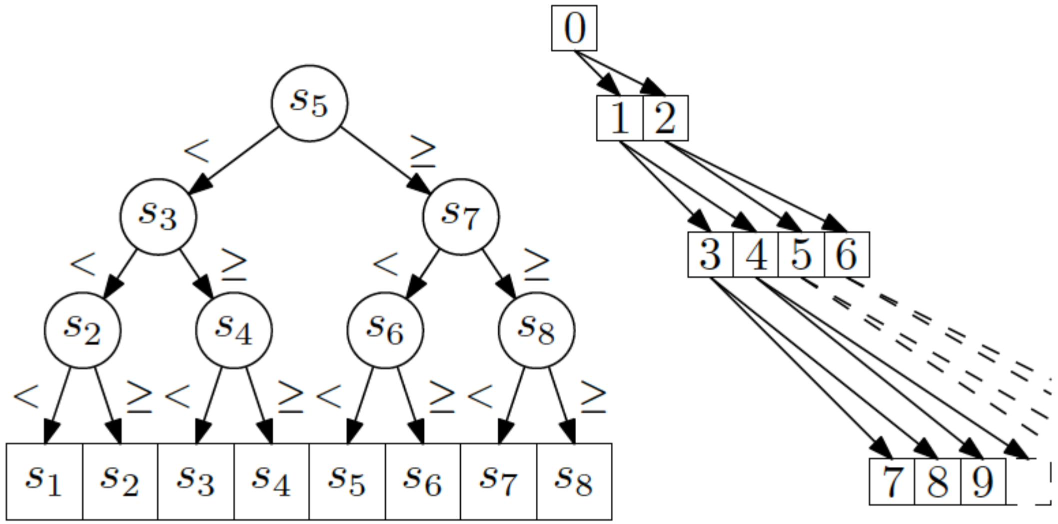
- We only copy elements of the buckets we are interested in;
- In case of identical splitter elements, they are placed in an *equality bucket*;
- If target rank is in an *equality bucket*, the algorithm can terminate early by returning lower bound;
- For sorting the splitters, small input datasets, and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



Implementation Aspects

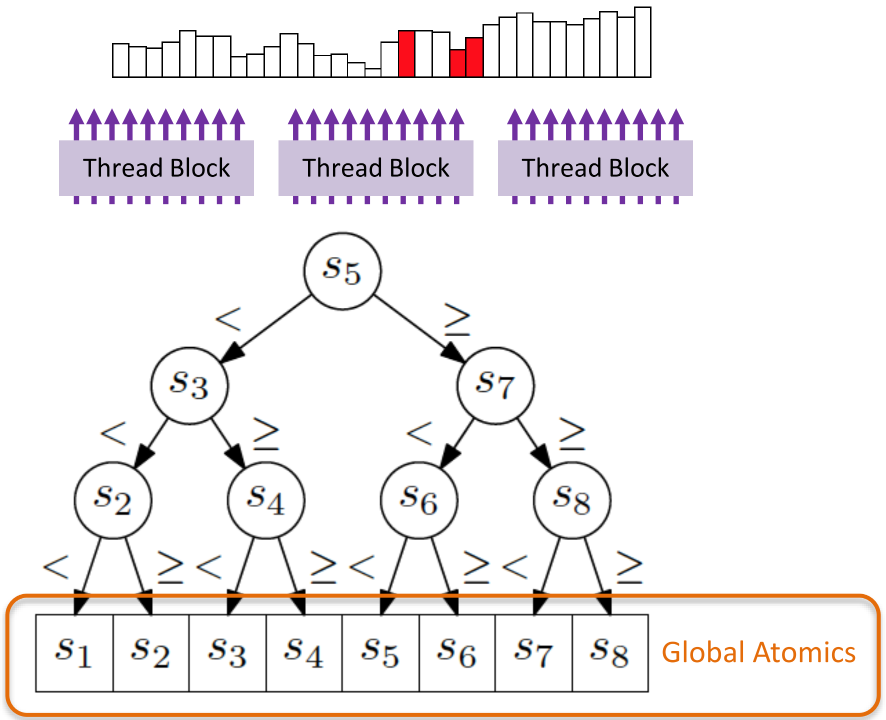
- We only copy elements of the buckets we are interested in;
- In case of identical splitter elements, they are placed in an *equality bucket*;
- If target rank is in an *equality bucket*, the algorithm can terminate early by returning lower bound;
- For sorting the splitters, small input datasets, and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





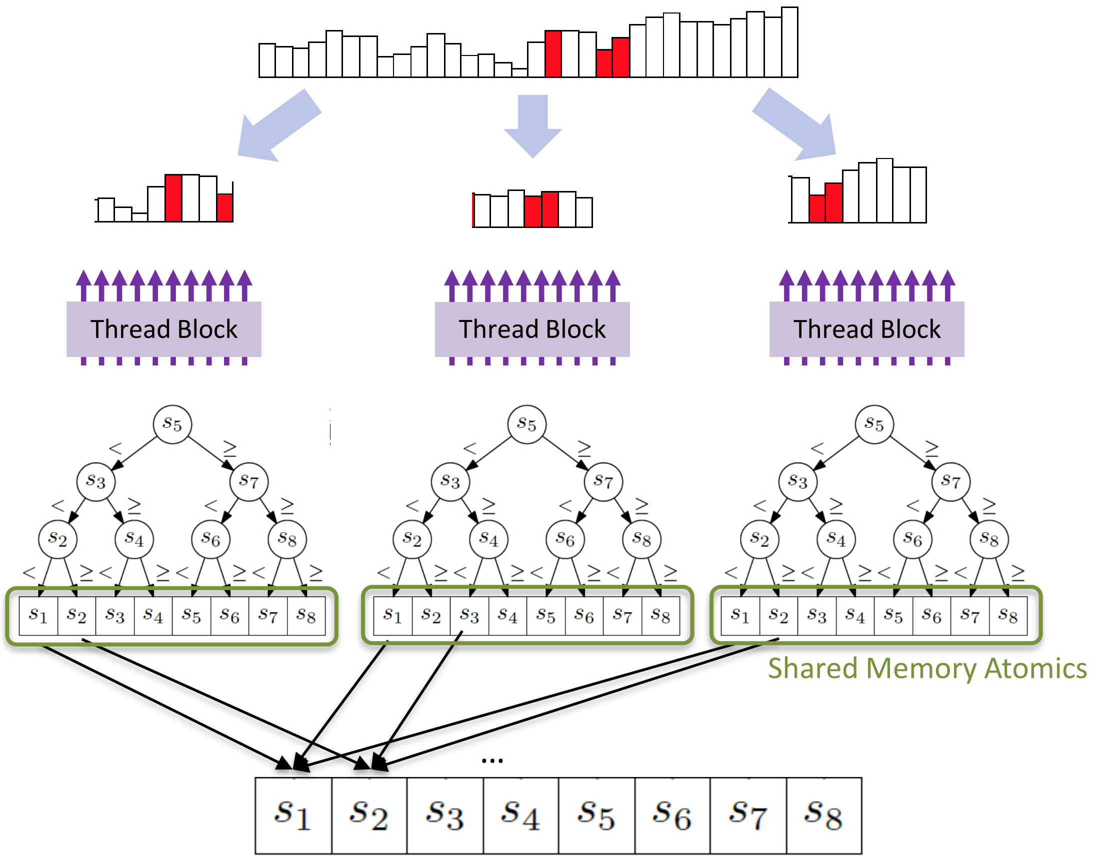
Parallelization & Communication

Global Memory Atomics



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

Shared Memory Atomics



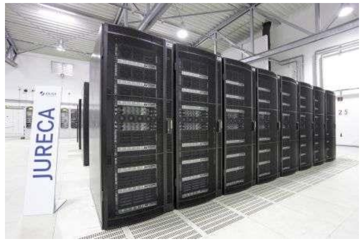
- Split data set into chunks, assign to thread blocks;
- Each thread block runs bucket count on its data;
- Use a global reduction to get global bucket counts;

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*



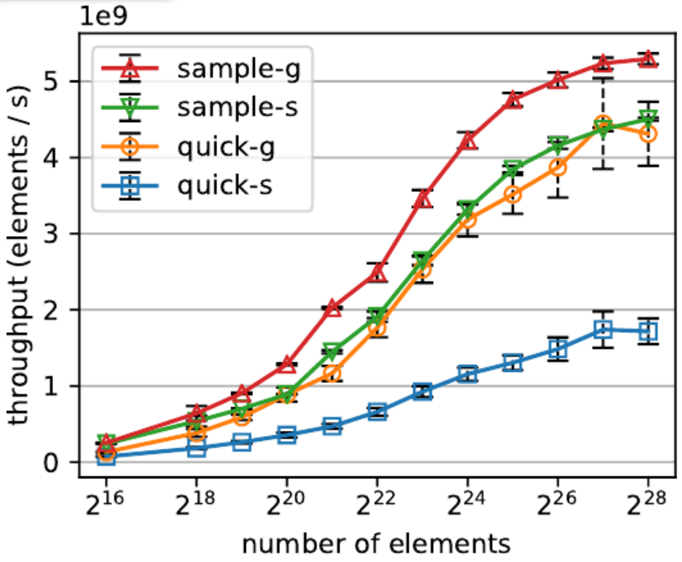
#44@TOP500

#1@TOP500

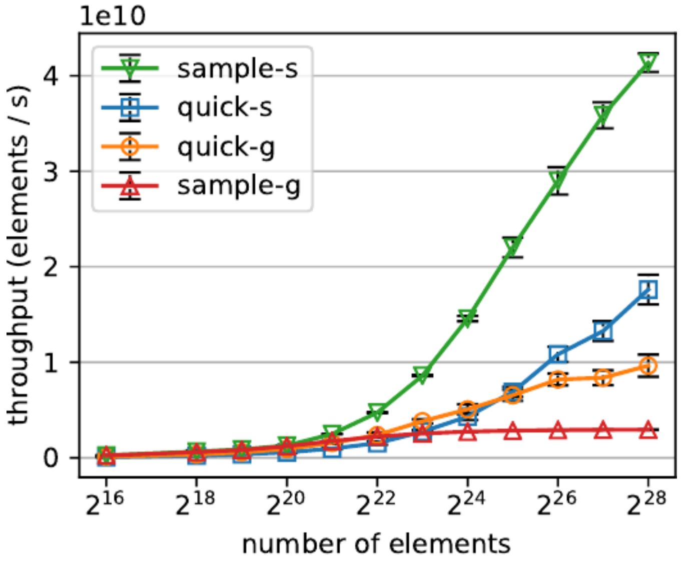
Kernel Performance: **Global vs. Shared Atomics**

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

NVIDIA K40



NVIDIA V100

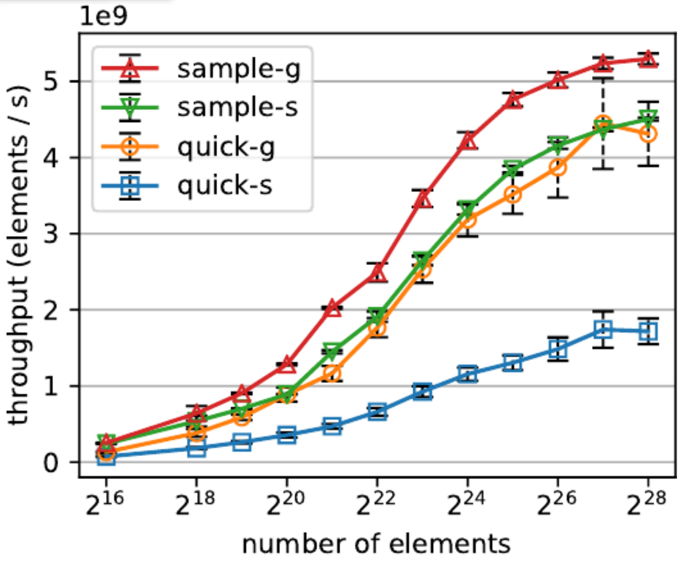


Larger performance variation for QuickSelect as we are more likely to run into the "*Worst-Case*" performance.

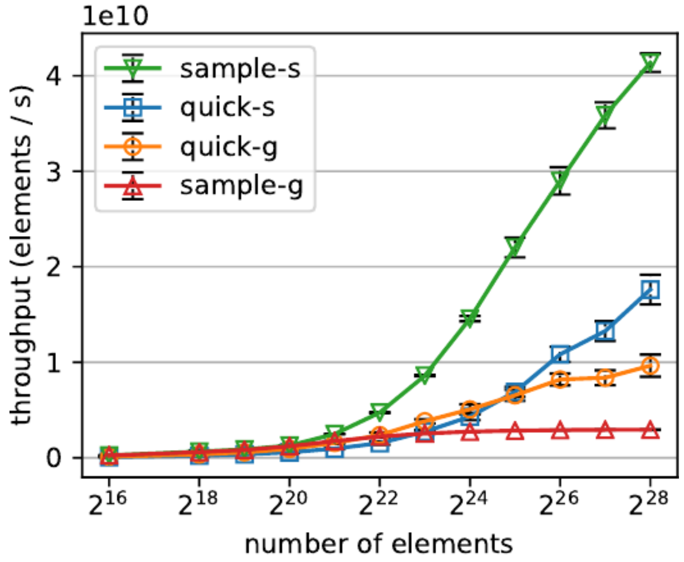
Kernel Performance: **Global vs. Shared Atomics**

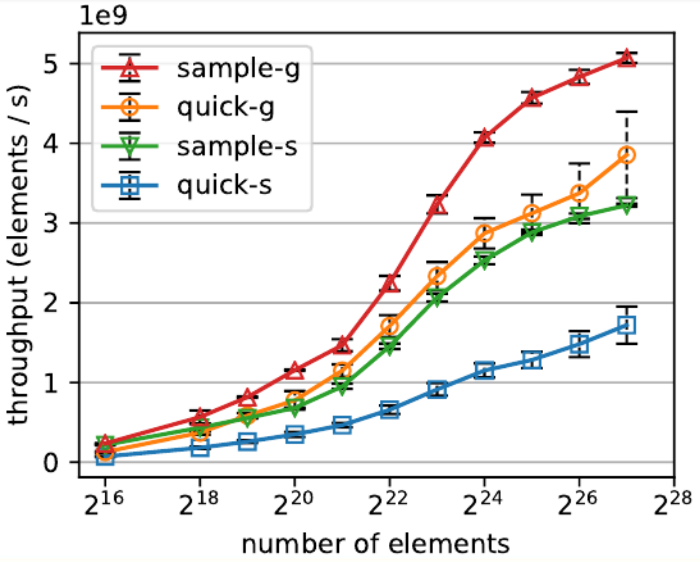
-g : global memory atomics
-s: shared memory atomics

NVIDIA K40

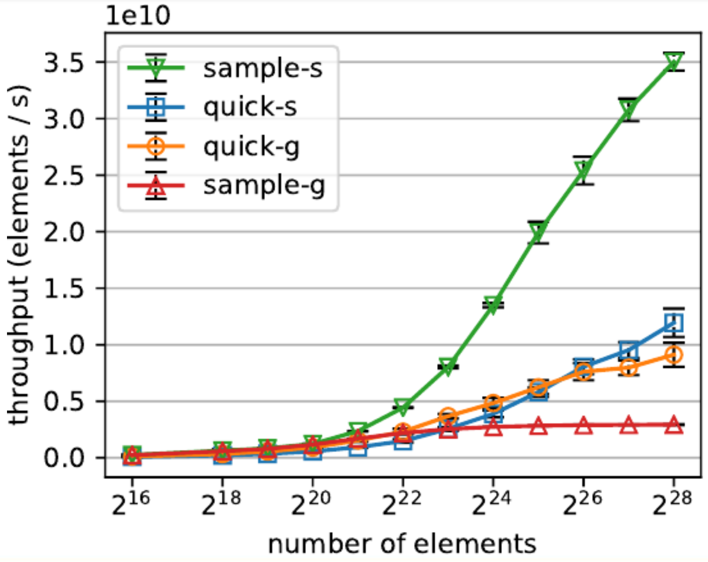


NVIDIA V100



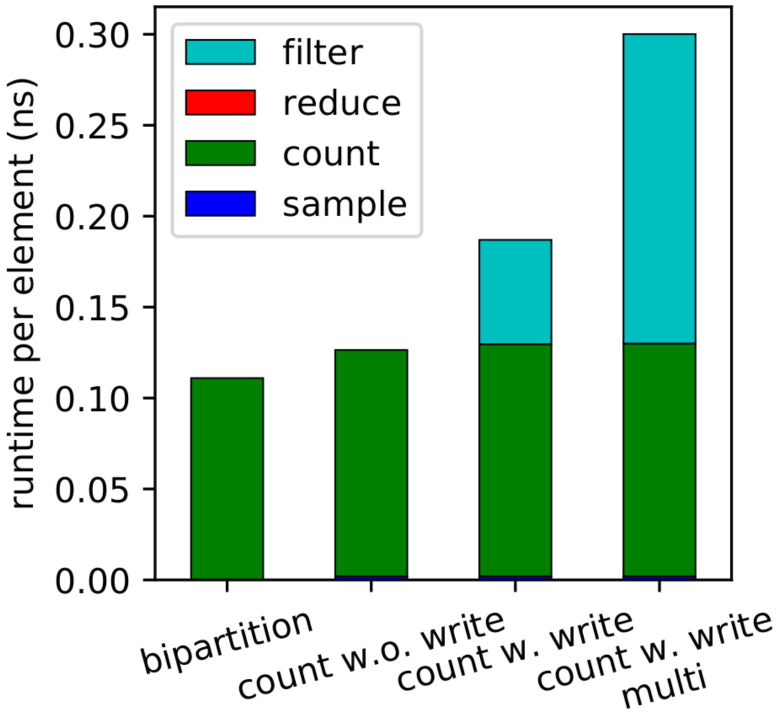


double precision

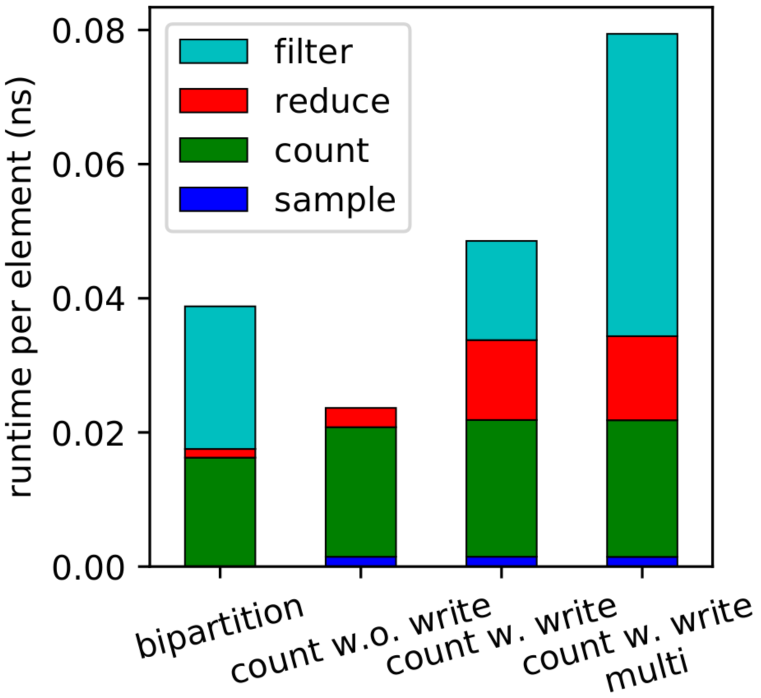


Runtime breakdown

NVIDIA K40
(global atomics)



NVIDIA V100
(shared atomics)

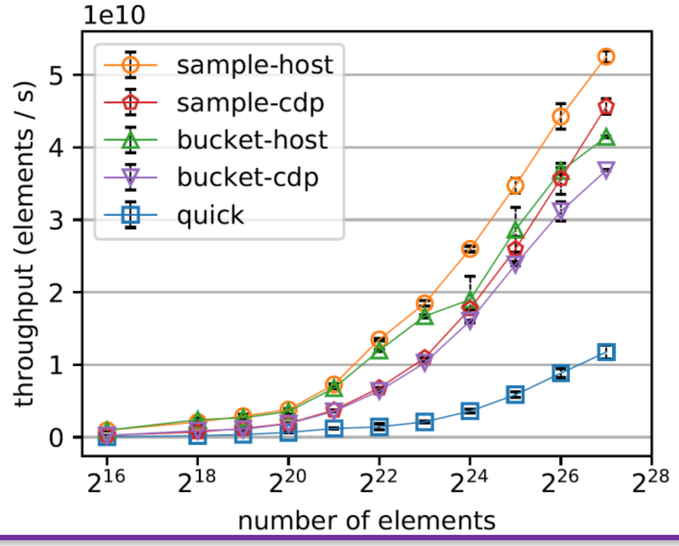


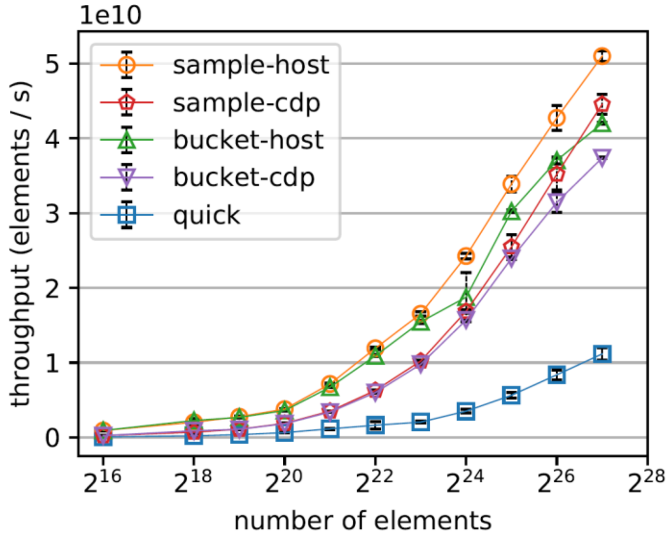
$n = 2^{24}$, single precision

Kernel Performance: **Host vs. Device recursion**

-host: Host launches recursive kernels
-cdp: CUDA dynamic parallelism

NVIDIA A100

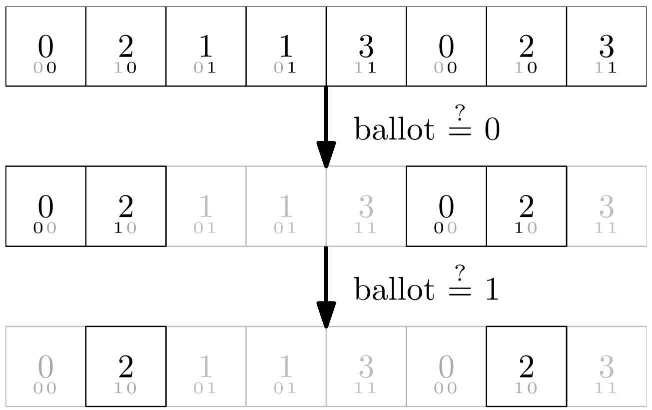




double precision

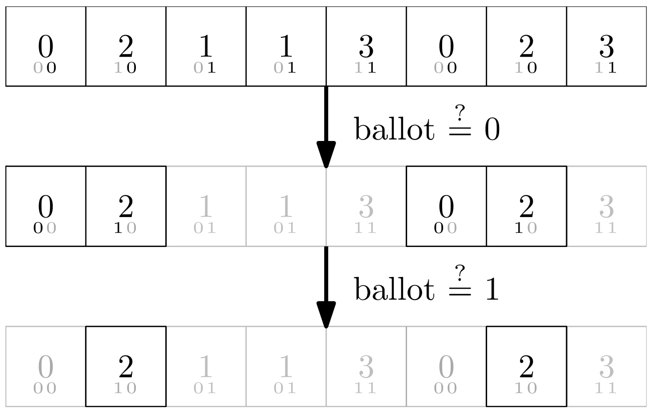
Kernel Optimization: **Element Repetition**

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

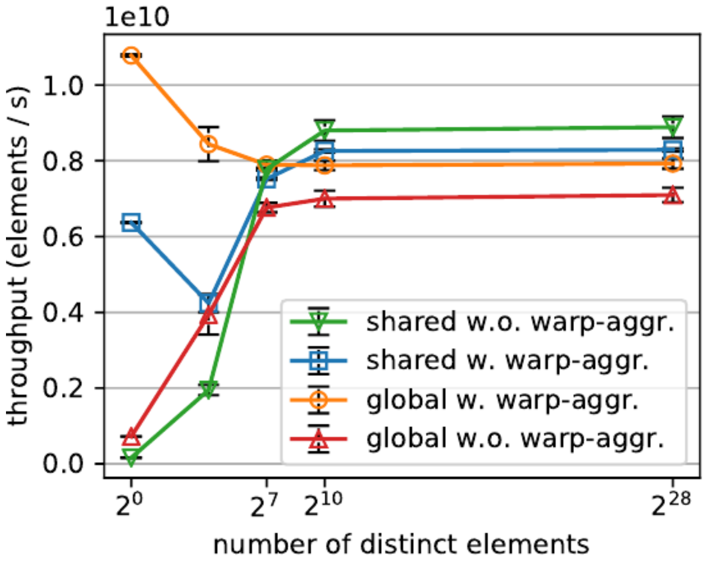


Kernel Optimization: **Element Repetition**

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

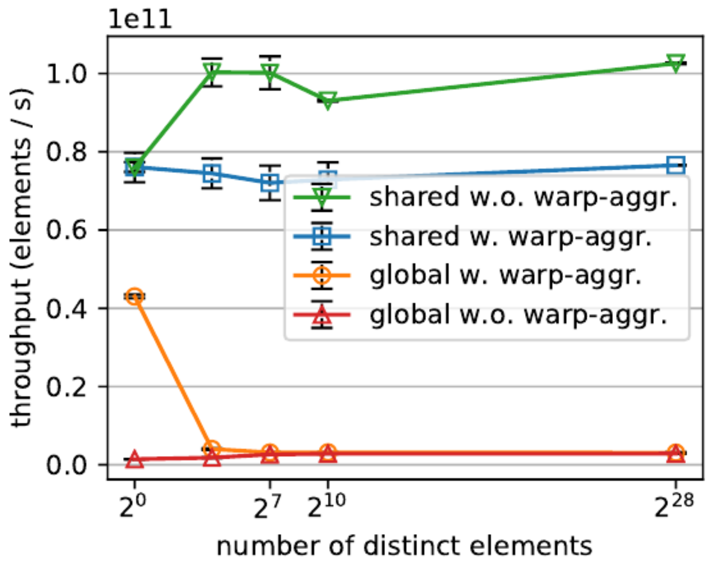


NVIDIA K40



more collisions

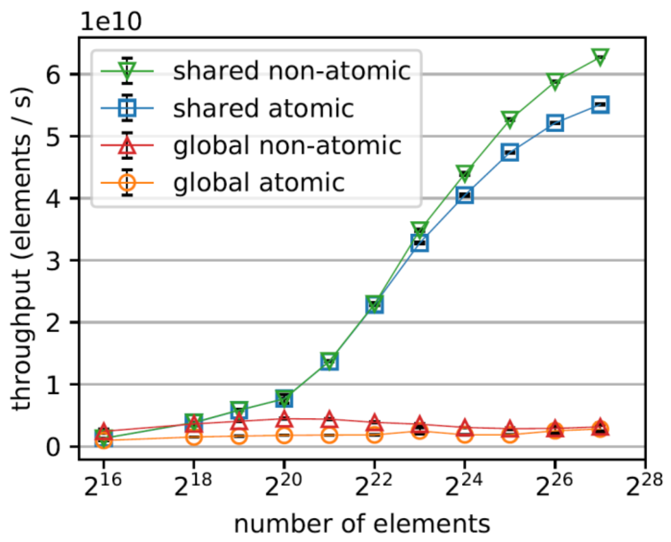
NVIDIA V100



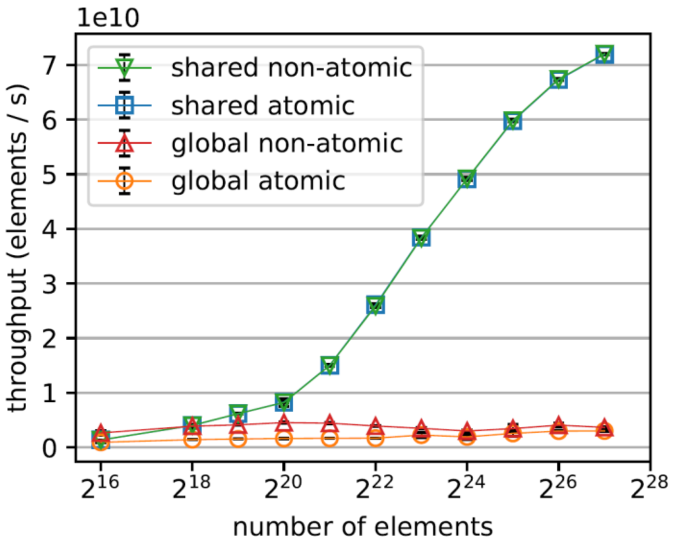
more collisions

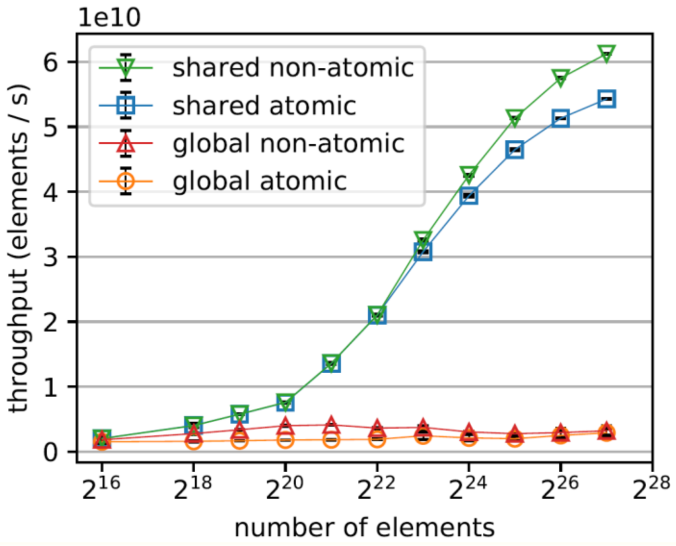
Kernel Performance: **What if we needed no atomics?**

NVIDIA A100 with warp-aggregation

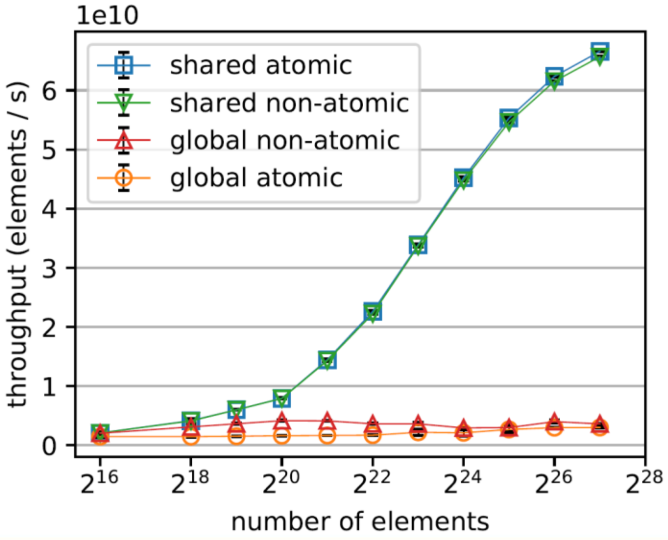


NVIDIA A100 without warp-aggregation





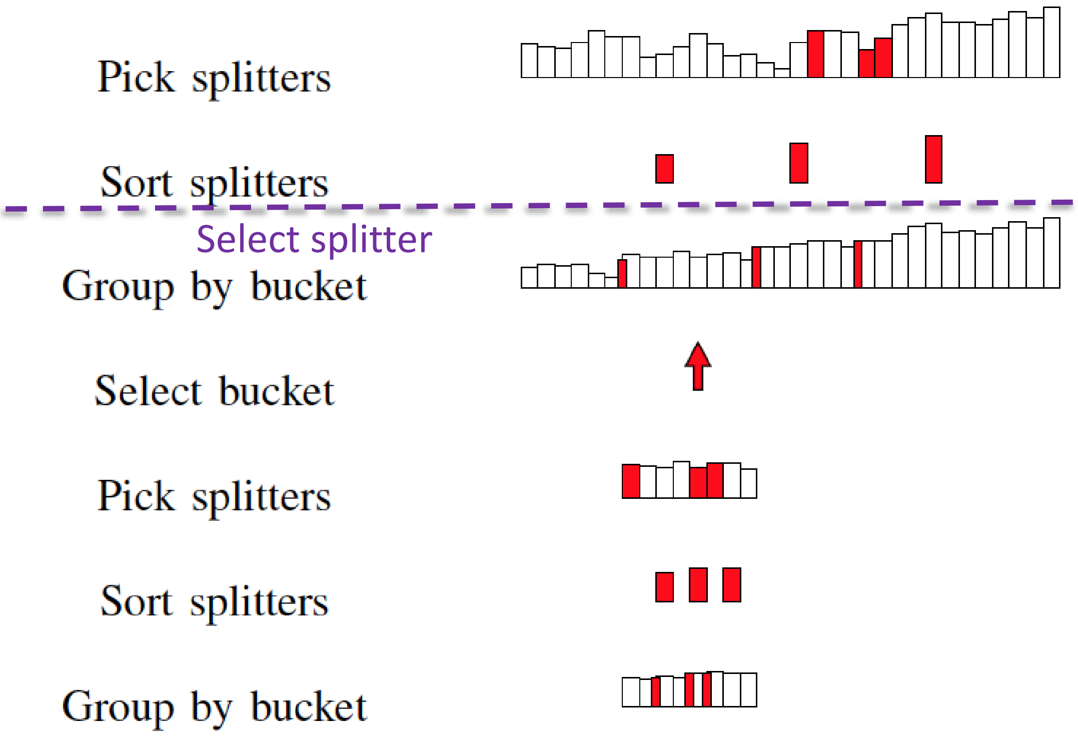
double precision



Approximate Selection

We do not descend 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)



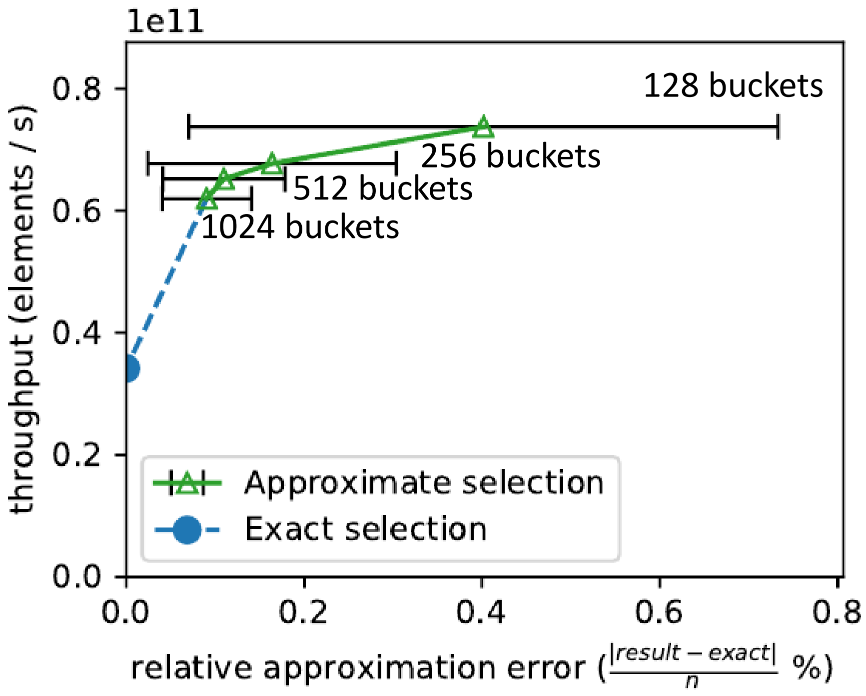
Approximate Selection

We do not descend 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:

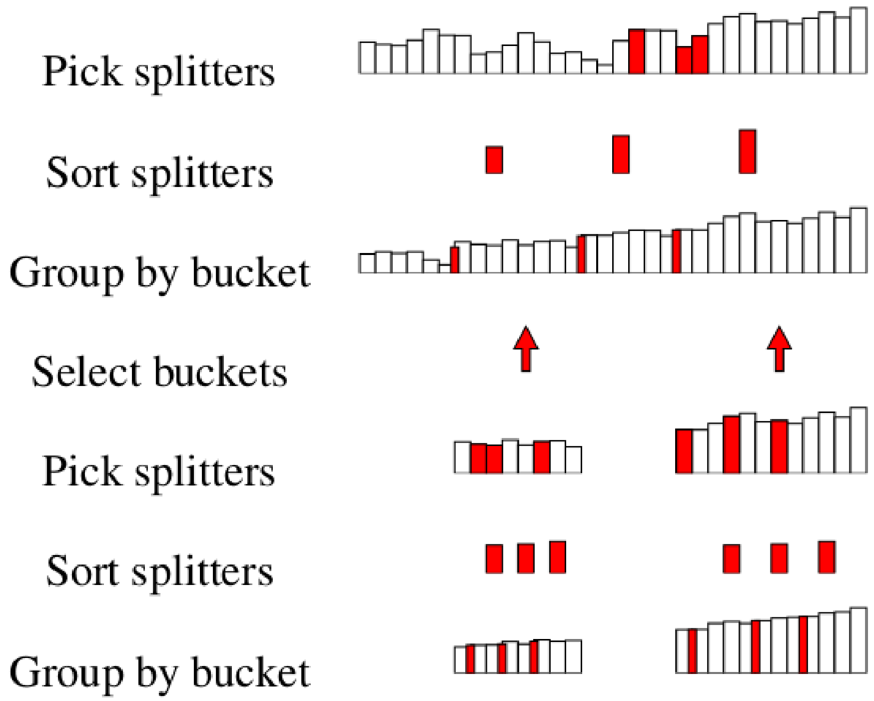
- 2^{28} uniformly distributed single precision values
- Approximate selection uses 1 level only
- We report statistics over 10 runs



Multiple Selection

Generalization: Select elements at *multiple ranks* $k_1, \dots, k_m$ simultaneously

- Determine which buckets contain k_i using binary search
- Extract elements from all these buckets simultaneously
- Launch multiple subcalls using CUDA *dynamic parallelism*

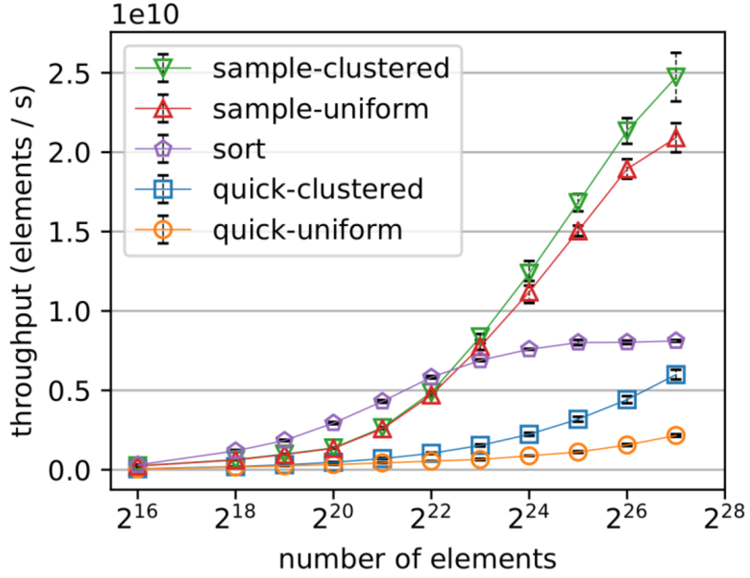


Multiple Selection

Generalization: Select elements at *multiple ranks* $k_1, \ldots, k_m$ simultaneously

- Determine which buckets contain k_i using binary search
- Extract elements from all these buckets simultaneously
- Launch multiple subcalls using CUDA *dynamic parallelism*
- Comparison with *QuickSelect* and CUB *RadixSort*
- Input ranks: *clustered* with $k_i = 2^i$ (best case)
  uniform with $k_i = \frac{i}{32} \cdot n$ (worst case)

NVIDIA V100

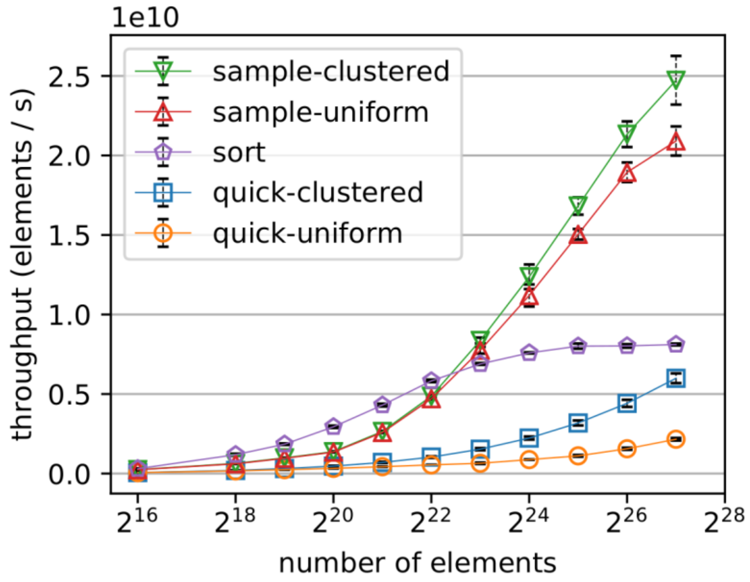


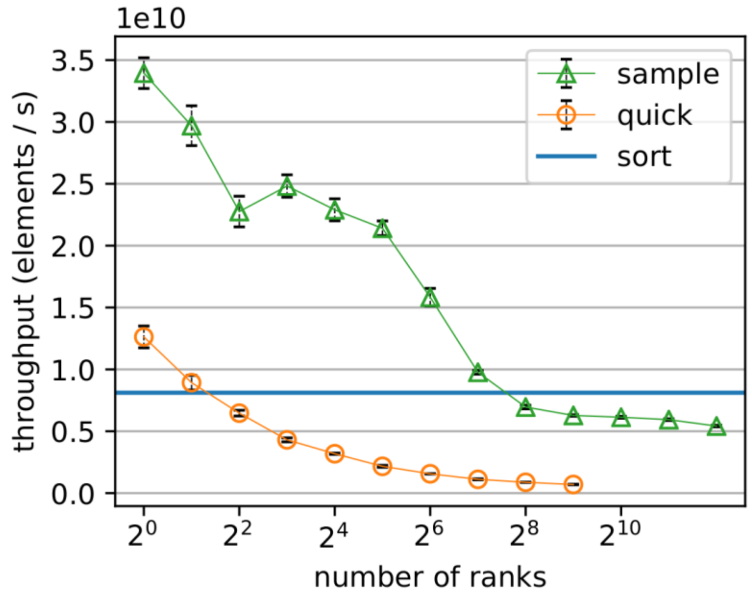
Multiple Selection

Generalization: Select elements at *multiple ranks* $k_1, \dots, k_m$ simultaneously

- Determine which buckets contain k_i using binary search
- Extract elements from all these buckets simultaneously
- Launch multiple subcalls using CUDA *dynamic parallelism*
- Comparison with *QuickSelect* and CUB *RadixSort*
- Input ranks: *uniform* with $k_i = \frac{i}{\#ranks} \cdot n$ for $n = 2^{27}$

NVIDIA V100





Summary and Outlook

- 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
- Multiple selection faster than sorting for up to 128 ranks

From a performance engineering standpoint (overgeneralized take-aways ☺):

- Hardware support beats warp-aggregation for atomics
- Shared-memory atomics are blazingly fast
- Host-side kernel launches outperform dynamic parallelism for tail recursion
- Pruning your recursion tree can be worthwile (if you still have enough parallelism left)

HELMHOLTZ
RESEARCH FOR GRAND CHALLENGES

Helmholtz Impuls und Vernetzungsfond
VH-NG-1241

Karlsruher Institut für Technologie

THE UNIVERSITY OF TENNESSEE KNOXVILLE

References

1. T. Ribizel and H. Anzt, “Approximate and Exact Selection on GPUs,“ Proceedings of the 9th AsHES Workshop at IPDPS, 2019
2. T. Ribizel and H. Anzt, “Parallel selection on GPUs,” Parallel Computing, vol. 91, p. 102588, Mar. 2020

Karlsruher Institut für Technologie

THE UNIVERSITY OF TENNESSEE KNOXVILLE