Approximate and Exact Selection on GPUs

Tobias Ribizel*, Hartwig Anzt*†
*Steinbuch Centre for Computing, Karlsruhe Institute of Technology, Germany
†Innovative Computing Lab (ICL), University of Tennessee, Knoxville, USA
tobias.ribizel@student.kit.edu, hartwig.anzt@kit.edu

***Abstract*—We present a novel algorithm for parallel selection on GPUs. The algorithm requires no assumptions on the input data distribution, and has a much lower recursion depth compared to many state-of-the-art algorithms. We implement the algorithm for different GPU generations, always using the respectively-available low-level communication features, and assess the performance on server-line hardware. The computational complexity of our *SampleSelect* algorithm is comparable to specialized algorithms designed for – and exploiting the characteristics of – "pleasant" data distributions. At the same time, as the *SampleSelect* does not work on the actual values but the ranks of the elements only, it is robust to the input data and can complete significantly faster for adversarial data distributions. Additionally to the exact *SampleSelect*, we address the use case of approximate selection by designing a variant that radically reduces the computational cost while preserving high approximation accuracy.**

***Index Terms*—parallel selection algorithm, GPU, kth order statistics, approximate threshold selection**

I. Introduction

Sequence selection is an ubiquitous challenge that appears in many problem settings, from quantile selection in order statistics over determining thresholds in approximative algorithms to top-k selection in information retrieval. One of the most popular solutions to this problem is the widely-used *Quickselect* algorithm [1], a partial-sorting variant of *Quicksort* [2]. The close relationship between these two algorithms is not a singularity, but characteristic of the connection between selection and sorting algorithms. In fact, many improvements of *Quicksort* and similar partitioning-based sorting algorithms can be directly transferred to the corresponding selection algorithms, e.g., the deterministic pivot choice implemented using the *Median of medians* algorithm [3], multiple splitter elements in the *Sample sort* algorithm [4], and an implementation variant optimized for modern hardware architectures called *Super-scalar sample sort* [5].

With the rise of parallel architectures, the development of effective selection and sorting algorithms is heavily guided by parallelization aspects. The traditional concepts employed for the parallelization are based on work decomposition, and have proven to be efficient for multi-core and multi-node architectures embracing the multiple-instruction-multiple-data (MIMD) programming paradigm. Unfortunately, the same strategies largely fail to work efficiently ons modern manycore architectures like GPUs. The primary reason is that these devices are designed to operate in streaming mode, and that their performance heavily suffers from instruction-branching, non-coalesced memory access, and global communication or synchronization. As streaming processors like GPUs are becoming increasingly popular, and are nowadays adopted by a large fraction of the supercomputing facilities, there exists a heavy demand for selection algorithms that are designed to leverage the highly parallel execution model of GPUs and avoid global synchronization and communication in favor of localized communication. In response to this demand, we propose a new parallel selection algorithm for GPUs. Aiming at a sample selection algorithm featuring fine-grained parallelism, we follow a bottom-up approach by starting with the GPU hardware characteristics, and selecting algorithmic building blocks that map well to the architecture-specific operating mode. Acknowledging CUDA's asynchronous execution model, and using low-level communication features inherently supported by hardware, the new selection algorithm proofs to be competitive with other GPU-optimized selection algorithms that impose strong assumptions on the input data distribution, and superior to input-data independent state-of-the-art algorithms available in literature, open source software, or vendor libraries.

The rest of the paper is organized as follows. In Section II we recall some basic concepts of selection algorithms and their parallelization potential. Section III list efforts that also aim at parallelizing selection algorithms for GPUs. In Section IV we present the novel SampleSelect selection algorithm. We also provide details about how the SampleSelect algorithm is realized in the CUDA programming model, and how the low-level communication and synchronization features available in the distinct GPU generations are incorporated. Section V, presents a comprehensive analysis of the effectiveness, efficiency, and performance of the novel SampleSelect selection algorithm. We conclude in Section VI with a summary of the findings, and an outlook on future research.

II. Selection

For an input sequence $(x_0, \dots, x_{n-1})$, the selection problem is given by finding the element at position k in the sorted sequence $x_{i_0} \le \cdots \le x_{i_{n-1}}$, i.e., finding the kth-smallest element x_{i_k} of the sequence. In this setting, we also say that x_{i_k} has *rank* k. If the rank of an element is not unique, i.e., because the element occurs multiple times in the sequence, we assign it the smallest rank.

A. General framework

The most popular algorithms for the selection problem are all based on *partial sorting*: If we choose $b{+}1$ so-called *splitter*

```
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);
}
```

Fig. 1. High-level overview of a bucket-based selection algorithm

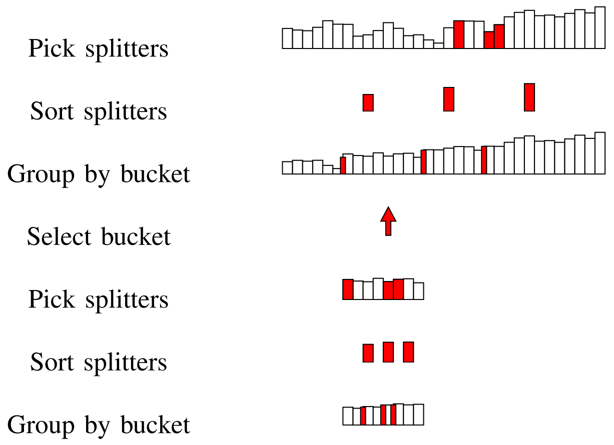

Fig. 2. Visualization of bucket-based partial sorting

elements s_i ($-\infty = s_0 \leq \cdots \leq s_b = \infty$), we can partition the input dataset into b *buckets* containing the element intervals $[s_i, s_{i+1})$. An important consequence of this partitioning is that, aside from the element values, we also partition their *ranks* in the sorted sequence: Let n_i be the number of elements in the ith bucket, i.e., the number of elements from the input sequence contained in $[s_i, s_{i+1})$. Then these elements have ranks in the interval $[r_i, r_{i+1})$, where $r_i = \sum_{j=0}^{i-1} n_j$ is the combined number of elements in all previous buckets.

Based on this observation, we can formulate a general framework for exact selection: After determining the element count for each bucket, it suffices to recursively proceed only within the bucket containing the target rank. Specifically, for identifying the element of rank k ($k \in [r_i, r_{i+1})$) we proceed with searching for the element with rank $k - r_i$ in this bucket. The algorithmic framework for such a bucket-based selection is given in Figure 1 and visualized in Figure 2. Virtually all popular selection algorithms are based on this approach of recursive bucket selection.

B. Splitter selection

The choice of splitters in a bucket-based selection algorithm has a strong influence on the recursion depth, and thus the total runtime of the resulting algorithm. In the general case, the optimal splitters separate the input elements in b buckets of equal size n/b. This results in an algorithm that needs at most $\log_b \frac{n}{B} + 1$ recursive steps, where B is the base case size (lowest recursion level), below which we sort the elements to return the kth-smallest element directly. Without considering the computational overhead of choosing the splitters, their optimal values are the $p_i = i/b$ percentiles of the input dataset. In practice, these can be approximated by the corresponding percentiles of a sufficiently large random sample. In terms of the relative element ranks, the average error introduced by considering only of a small sample of size s of the complete dataset can be estimated as follows: The relative ranks of the sampled elements, i.e., the ranks normalized to $[0, 1]$, are approximately uniformly distributed: $X_1, \ldots, X_s \sim \mathcal{U}(0, 1)$ and, assuming sampling with replacement, independent. Thus the sample percentiles are asymptotically normal-distributed with mean p_i and standard deviation $\sqrt{p_i(1 - p_i)/s}$ [6]. We can thus use the sample size s to control the imbalance between different bucket sizes.

C. Approximating the kth-smallest element

An important observation in the context of bucket-based selection algorithms is that after all elements have been grouped into their buckets, the ranks of the splitter elements are already available: Their ranks equal the aforementioned partial sums r_i. If the application does not require our algorithm to compute the kth-smallest element exactly, but can work with an approximation like the $k \pm \varepsilon$th smallest element, the selection algorithm can be modified to terminate before the lowest recursion level is reached. In this case, we can compute the approximate kth order statistic as the splitter s_i whose rank r_i is closest to k. In terms of the element ranks, the error is at worst half the maximum bucket size, and can thus be controlled by the number of buckets and sample size. If the distribution of the input data is smooth, the small error in the element rank translates into a small error of the element value. However, this is not true for the general case, as for noisy input data, the induced error can grow arbitrary large.

III. Related work

In the past, different strategies aiming at efficient selection on GPUs were explored. The first implementation of a selection algorithm on GPUs was presented by Govindaraju et. al. [7] for the problem of database operations. The proposed algorithm recursively bisects the value range of the binary representation of the input data. A different approach was proposed by Beliakov [8] – it is based on the reformulation of the median selection as a convex optimization problem. Monroe et. al. [9] published a Las-Vegas algorithm for choosing two splitters that bound a small bucket containing the kth-smallest element with high probability. Alabi et. al. [10] were the first to use a larger number of buckets in their selection algorithm, either by uniformly splitting the input value range (BucketSelect), or based on the RadixSort algorithm (RadixSelect). Furthermore, significant advances in the theoretical treatment of communication-minimal parallel selection algorithms as well as the practical implementation thereof on distributed systems have been presented by Hübschle-Schneider and Sanders [11].

Unlike most previous works on GPU selection, our algorithm is purely comparison-based, i.e., we only use the relative order and ranks of elements to determine the kth-smallest element.

IV. Implementation

A. Optimizing for memory bandwidth

Every algorithm for sequence selection needs to read each element at least once. The classical QuickSelect algorithm applied to a sequence of length n needs to read and write $2n$ elements on average, using auxiliary storage of size $n/2$ if the input cannot be overwritten. As the sort- and selection algorithms are memory bound on GPU architectures (which implies that the data access volume correlates with the runtime), we aim at developing an algorithm with a lower memory access volume. The SampleSelect algorithm we propose requires $(1+\varepsilon)n$ element read and write operations on average, with a small and configurable ε parameter and auxiliary storage of size at most $n/4$.[1]

B. SampleSelect

At its core, our SampleSelect implementation consists of three elementary kernels:

1) The `sample` kernel builds a sorted set of splitters.
2) The `count` kernel traverses all data, and determines the size of the distinct buckets.
3) The `filter` kernel extracts the elements of a single bucket.

a) Sample kernel: To select a suitable splitter set for the following steps, our `sample` kernel first loads a small sample of elements into shared memory, and sorts them using a bitonic sorting network [12]. From the resulting set, we pick the i/b percentiles for $i = 1, \dots, b-1$, and store them in global memory.

b) Count kernel: At its core, the `count` kernel is the combination of two important steps: First we need to identify which bucket an element belongs into. Then we need to increment the shared counter for this bucket. While the bucket index could be identified using a binary search on the sorted splitter array, the involved index calculations are rather complicated.

Thus, we decided to employ a technique introduced in [5] and place the splitters in a complete binary search tree that is implicitly stored in an array. The indexing of this array is based on the approach often used in binary heaps: For a tree node at index i, its parent has index $\lfloor \frac{i-1}{2} \rfloor$ and its children have the indexes $2i+1$ and $2i+2$. To reduce the memory footprint necessary to identify elements from a single bucket, we memoize the bucket index for each element (called *oracle*) in as few bits as possible – on realistic hardware, that means we use a single byte to store each oracle, limiting us to at most 256 buckets. The indexing and search tree traversal are visualized in Figures 4 and 3.

[1]For single precision inputs based on the parameters providing the greatest performance. Double precision inputs only require only half that amount.

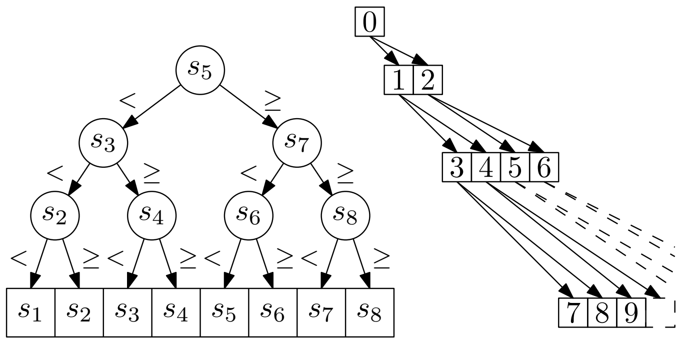

Fig. 3. Search tree based on bucket splitters $s_1, \dots, s_8$ (left) and its implicit array storage order (right).

```
double element = data[idx];
double tree[2 * tree_width - 1];
int i = 0;
for (int l = 0; l < tree_height; l++)
    i = 2 * i + (element < tree[i] ? 1 : 2);
int bucket = i - (tree_width - 1);
counts[bucket]++;
oracles[idx] = bucket;
```

Fig. 4. Loop for traversing the implicit search tree.

c) Filter kernel: The `filter` kernel scans through all oracles, loading only the elements belonging to a fixed bucket and stores them contiguously using a shared counter that stores the next unused index in the contiguous storage.

C. Repeating elements

Initially, the SampleSelect algorithm is designed for sequences of pairwise different elements, each of them having a unique rank. However, small modification introduced in [5] enable SampleSelect to handle equal elements: In case identical splitters $s_a = \dots = s_e < s_{e+1}$ occur, the equal elements are sorted into the eth bucket together with all elements smaller than s_{e+1}. Replacing s_e by $\tilde{s}_e = s_e + \varepsilon$ enables to place identical elements in an *equality bucket*. In case the target element is contained in such an equality bucket, the algorithm can terminate early by just returning the corresponding lower bound splitter.

D. Sorting small inputs

Different stages of selection algorithms require the efficient sorting of small element sets. For this purpose, we implement a simple bitonic sorting kernel [12] operating in shared memory. As bitonic sorting requires explicit synchronization, the kernel needs to be restricted to a single thread block, as this is the largest thread group that is guaranteed to be scheduled on the same multiprocessor (SM) and capable of leveraging shared memory atomics. In the distinct algorithm implementations, the bitonic sorting implementation is used for the splitter selection in SampleSelect, pivot selection in QuickSelect, and for the recursion base case in both algorithms.

E. Recursion

As the recursion depth of our algorithms is not exactly known a-priori, and communication/synchronization between

```
int l = 0, r = size - 1;
double pivot;
for (int i = 0; i < size; i++)
    bool smaller = data[i] < pivot;
    int o = smaller ? l : r;
    l += smaller ? 1 : 0;
    r -= smaller ? 0 : 1;
    out[o] = data[i];
```

Fig. 5. Branchless partitioning algorithm for Quickselect.

the host processor and the GPU can be a source of large latencies, we use CUDA Dynamic Parallelism to keep the control flow completely on the GPU.[2] For this purpose, we introduced additional kernels that select the bucket containing the kth-smallest element, and compute the kernel launch parameters for the subsequent recursion level.

F. Reference implementation: Quickselect

As a reference point in the performance evaluation, we implemented a GPU version of the Quickselect algorithm, and employ the same performance optimizations like for the implementation of the SAMPLESELECT algorithm we propose in this work. While SAMPLESELECT chooses a large number of splitters and (conceptually) partitions the elements into the resulting buckets, QUICKSELECT only chooses a single so-called *pivot element* based on which the input data is bipartitioned. This difference leads to simpler treatment of a single element, but in general requires more recursion levels and more read and write operations to input elements than SAMPLESELECT.

As a basic building block, we implemented a branchless bipartition kernel that processes the selection by growing the array of elements smaller than the pivot from the left and elements larger than the pivot from the right. The core of the bipartition kernel is provided in Figure 5.

G. Shared counters

A core functionality of all aforementioned kernels is the atomic increment of counters shared by a large thread group that processes the data in parallel. For this purpose, the CUDA language provides a set of atomic operations that can operate on shared and global memory. However, operations on global memory usually require a large degree of synchronization, and can thus quickly become detrimental to the kernel performance. On the other hand, the much faster shared memory atomics can only be used to synchronize within a single thread block, thus requiring additional reduction operations to combine the partial results to global counts.

For both, the selection kernel and the bipartitioning kernel, the atomic counters in global memory can be replaced with a hierarchy of atomics working on different memory levels. This can be realized by

1) Executing the kernel (selection/bipartitioning) once, but only accumulating the atomic operations for a single thread block in a shared-memory counter and storing this block-local partial sum.
2) Computing a prefix sum (also sometimes referred to as exclusive scan) over all block-local partial sums. These sums denote the boundaries of memory areas each thread block will write to, thus assigning an index range to each thread block. This operation is denoted by `reduce` in the following descriptions.
3) Executing the kernel (selection/bipartitioning) a second time, this time using the index ranges computed by the previous step to assign an unique index to each output element.

In case of our SAMPLESELECT implementation, the `count` kernel does not require the partial results of the atomic counters in shared memory. Hence, the last step is merged with the `filter` kernel, instead. This works because both kernels operate on exactly the same element indexes, so the prefix sums from one kernel can be used in the other one. The implementation of `filter` using shared-memory atomics follows the approach introduced in [13], but differs in the sense that instead of storing predicate bits as an intermediate step, it stores the bucket indexes in the oracles.

A consequence of the use of atomics is the significant performance impact of atomic collisions. These collisions occur in case multiple threads execute atomic operations on the same operand/memory location. For a moderate number of buckets, this is likely due to the Birthday paradox [14]. A mitigation strategy that reduces the number of atomic collisions is *warp-aggregation* [15]. The idea is to use warp-local communication to synchronize among the threads of a warp (1 warp contains 32 threads), and issue only a single atomic operation for each atomic counter in a warp. As a side-effect, this technique can reduce the number of atomic operations, especially in cases where only few different operands/memory locations get updated atomically. While warp-aggregation is usually used on global counters that get updated by each thread, the same techniques can also be used in the histogram-like bucket count operation, as demonstrated in Figure 6: For a fixed thread, the loop computes a bitmask containing all threads of the warp that computed the same bucket index (and would thus introduce a atomic collision). In the implementation of the bucket count kernel, the mask computation can also be interleaved with the searchtree traversal, potentially hiding latencies from shared memory access.

H. Tuning parameters

The SAMPLESELECT and QUICKSELECT implementation have a number of tuning parameters and configuration options which can be modified to optimize for the best performance on different architectures or input data distributions.

a) Work distribution: The launch parameters, i.e., the number of thread blocks and threads per block can have a significant impact on the overall performance of a kernel.

b) Sample size: A larger sample used to select the bucket splitters generally improves the splitter quality. In

[2]CUDA Dynamic Parallelism allows kernels on the GPU to asynchronously launch new kernels. By utilizing that all kernels launched from the CPU or a single thread on the GPU will be executed in the order they were launched in, we were able to implement a simple tail-recursion.

```
int bucket;
int mask = 0xffffffff;
for (int b = 0; b < tree_height; b++)
    bool bit = bucket & (1 << b) != 0;
    int step_mask = ballot(bit);
    if (bit)
        // keep all threads that have the bit set
        mask = mask & step_mask;
    else
        // keep all threads that don't have the bit
            set
        mask = mask & ~step_mask;
```

Fig. 6. Warp-aggregation for bucket indices

consequence, it may decrease the variation in runtime or approximation error due to imbalances between the bucket sizes. However, it also increases the splitter-selection overhead, and can (if the sample size exceeds the shared memory size) require a more complex splitter-selection kernel.

c) Number of buckets: A larger number of buckets increases the accuracy of a single recursion level, and therewith decreases the the recursion depth of exact SAMPLESELECT. However, a it also increases the amount of shared memory needed to store the partial bucket for the `count` kernel, and increases the overhead of the reduction operation when using shared memory atomics.

d) Unrolling: If data traversal is unrolled for a single thread, the compiler is able to reorder instructions from consecutive iterations such that memory access latencies can be reduced. However, unrolling also potentially increases the register pressure of the kernel, reducing the maximum occupancy per streaming multiprocessor (SM).

e) Atomics: The performance characteristics of global and shared memory atomics is very architecture-dependent. Furthermore, the warp-aggregation alleviating the performance impact of atomic collisions introduces some overhead for the general case.

f) Base case: The input size at which the algorithm switches to resort a simple sorting-based selection kernel potentially impacts the overall execution time. However, as the input size decreases exponentially with the recursion level, we consider the impact negligible.

I. Kernel fusion

Aside from its stand-alone form, the SAMPLESELECT kernel is amenable to kernel fusion [16] in the situation that not only the kth-smallest element is required, but also all larger elements are of interest (often described as top-k selection). This can be achieved by modifying the `filter` kernel such that it copies not only elements from the target bucket, but also from all buckets containing larger elements. As the splitters are ordered, the recursion still only needs to descend into the target bucket, but all elements from larger buckets are guaranteed to be part of the top-k selection.

V. EXPERIMENTS

In the experimental evaluation of the SAMPLESELECT implementation, we assess its performance for different parameter configurations in comparison to the QUICKSELECT implementation. We consider two GPU architectures belonging to distinct compute generations, and a set of input datasets varying in size and value distribution.

	K20Xm	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

TABLE I
KEY CHARACTERISTICS OF THE HIGH-END NVIDIA GPUS. THE HALF (HP) PERFORMANCE OF THE V100 IS FOR THE 8 TENSOR CORES. THE SUSTAINED MEMORY BANDWIDTH IS MEASURED USING THE BANDWIDTH TEST SHIPPING WITH THE CUDA SDK.

A. Input data

As the SAMPLESELECT algorithm is sensitive only to the distribution of the element ranks, not the actual numeric values, we consider datasets generated as uniform distribution across a pre-defined set of distinct values. Specifically, we generate input datasets with sizes from $n = 2^{16}$ to 2^{28}, containing $d = 1, 16, 128, 1024$ and n distinct values. Using $d < n$ allows to evaluate the performance impact of repeating elements. For each input dataset, we also chose a random rank uniformly at random to simulate a variety of different workloads. To account for variation induced by random rank selection, we run each experiment on 10 distinct input dataset and report the average data along with the variation.

To ensure the correctness of the SAMPLESELECT implementation, we compare the results to a reference solution computed by the `std::nth_element` algorithm from the C++ standard library.

B. Hardware environment

We run experimental analysis on two different GPU models – The Tesla K20Xm and the Tesla V100. Their basic performance characteristics are listed in Table I. The kernels are compiled using the CUDA 9.2 compiler with code generation for the highest compute capability enabled. To minimize the impact of random noise, we measure the execution time for each kernel 10 times using the CUDA Runtime API (`cudaEventRecord` and `cudaEventElapsedTime`), and report the average results along with the variation.

QUICKSELECT and SAMPLESELECT are both linear-in-time algorithms. As performance metric, we take the total execution time of the selection process in relation to the dataset cardinality (not accounting for the distribution of the values), invert the ratio, and obtain "throughput" as dataset-size / algorithm runtime.

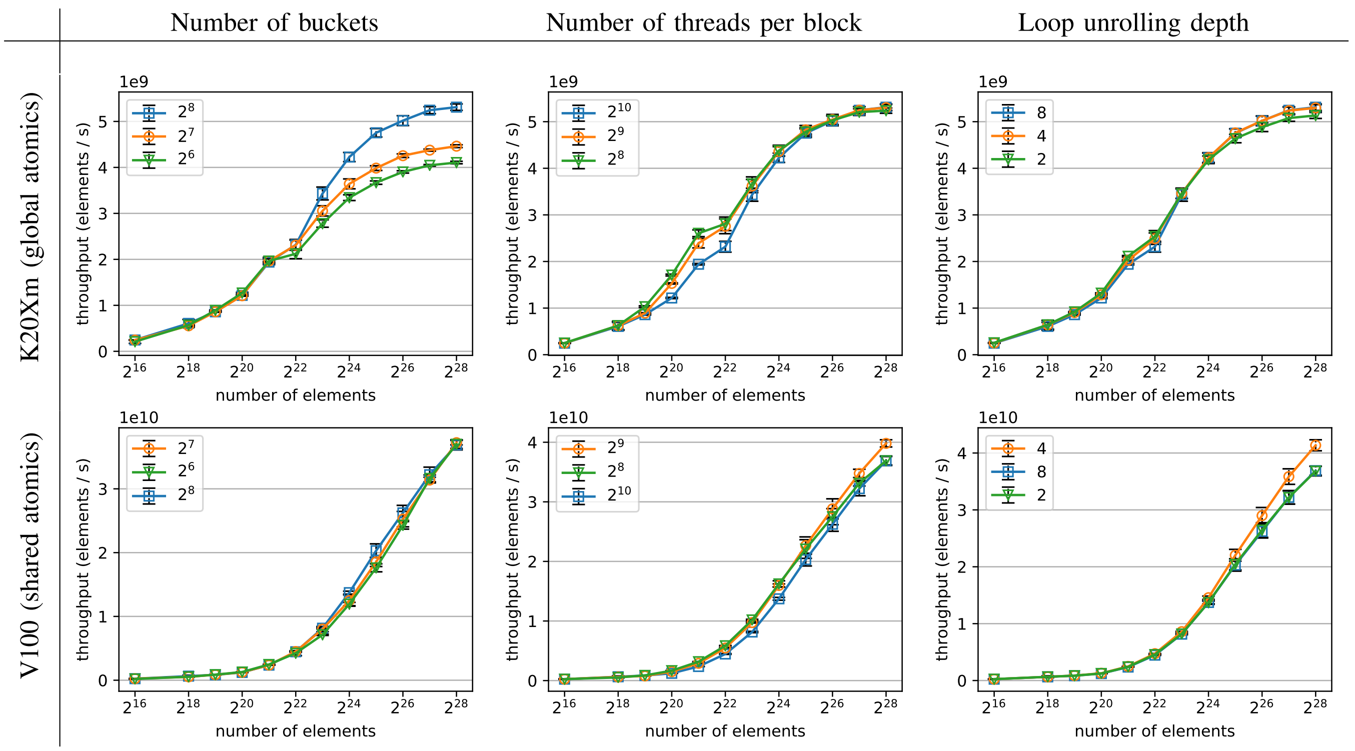

Fig. 7. Parameter tuning benchmarks (single precision). Based on preliminary experiments, we only visualize the performance using global memory atomics on the K20Xm and shared memory atomics on the V100, as these are the fastest configurations on the respective platform.

C. *Parameter tuning*

SAMPLESELECT features a list of parameters amenable to hardware-specific tuning. In Figure 7 we analyze the effect of different parameter choices on the overall performance of the algorithm implementations. In particular on the older K20 architecture, the SAMPLESELECT performance benefits from maximizing the number of buckets (within the limits of what a thread block allows for). At the same time the performance of the SAMPLESELECT implementation remains mostly unaffected from the number of threads accumulated in a block (for a fixed number of buckets) and the loop unrolling depth.

D. *Performance comparison with* QUICKSELECT

In Figure 8 we present the throughput plots for different input sizes to compare the shared-memory variants and global-memory variants of QUICKSELECT and SAMPLESELECT. For completeness, we consider both, single and double precision inputs.

A central observation is that the overall performance winner is architecture-specific. On the older K20Xm GPU, the implementations based on global-memory-communication ("sample-g" and "quick-g") are generally faster than their shared-memory counterparts ("sample-s" and "quick-s", respectively). In both precision formats, the differences are quite significant in particular for the QUICKSELECT algorithm. Oppose to this, the newer V100 GPU heavily favors the variants based on shared-memory-communication. There, the shared-memory variant of SAMPLESELECT is more than 10x faster than the global-memory variant, while the performance gap between the QUICKSELECT implementations is much smaller. For larger input datasets, SAMPLESELECT outperforms QUICKSELECT by a small margin on the K20Xm, but is more than twice faster on the V100. The performance gap increases for double precision inputs, where SAMPLESELECT achieves a throughput only slightly smaller than for single-precision inputs, As the atomics always operate on 32bit integers, this suggests that the atomic operations expose the bottleneck for the SAMPLESELECT implementation, oppose to the QUICKSELECT algorithm whose performance is more limited by the memory bandwidth. While randomness effects challenge a comprehensive roofline analysis, we estimate the SAMPLESELECT algorithm to achieve about one third of the peak bandwidth of the V100 GPU. Performance trends indicate that even higher efficiency values may be attainable for larger input datasets.

Comparing the SAMPLESELECT implementation with previous implementations of GPU selection algorithms ([7], [8], [9], [10]) proved difficult, as we were unable to obtain the source code of these implementations or run benchmarks on similar GPUs to compare with their reported performance numbers.

The most fair comparison would be between the fastest algorithm from [10], namely BUCKETSELECT, evaluated on a NVIDIA Tesla C2070 GPU and our SAMPLESELECT algorithm on the K20Xm.[3] On $n = 2^{27}$ uniformly distributed single-precision floating point numbers, the authors of BUCKETSELECT report a mean runtime of 40.16 ms, while our SAMPLESELECT algorithm takes 25.6 ms on average. This

[3]The peak memory bandwidth of the K20Xm is only 40% larger than of the C2070, however, the floating-point performance is roughly 3.5x larger.

difference is probably to a large degree due to the aforementioned hardware differences, as the BUCKETSELECT algorithm is based on the same fundamental approach, but their splitter choice is optimized for uniformly distributed data, simplifying their bucket index calculation significantly.

Our algorithm is thus competitive in the optimal use case for BUCKETSELECT and doesn't suffer from the existence of adversarial input datasets.

E. Data distribution and intra-warp communication

On the right-hand side of Figure 8, we assess the impact of the data distribution, in particular the collisions resulting from multiple occurrences of the same value. The distinct communication strategies differ in the efficiency to mitigate the effects: On the older K20Xm GPU, atomic collisions have a large impact on the runtime of both, shared-memory as well as global-memory atomics. This impact can be avoided by using the aforementioned warp-aggregation technique for histogram calculations, while incurring only a small performance penalty in the general case. The fast shared-memory atomics (initially introduced with the Maxwell architecture [17]) make warp-aggregation unnecessary on the V100 GPU.

F. Runtime breakdown

In Figure 9, we visualize the runtime breakdown for the different kernels of a single recursion level of SAMPLESELECT and QUICKSELECT on the V100 GPU. We observe that the recording of oracles ("count with write") has only negligible impact on the runtime of the `sample` and `count` kernels of SAMPLESELECT. Oppose to that, the following reduction becomes more expensive, as additionally to the total bucket counts, also the partial sums need to be computed, as those are used by the following `filter` kernel. The `count` kernel of QUICKSELECT completes much faster, as it only compares the elements against a single pivot element, and updates two atomic counters. At the same time, the `filter` kernel is much slower than the corresponding kernel of SAMPLESELECT, probably due to the larger memory footprint of the elements compared to their oracles. In the end, the QUICKSELECT algorithm needs a much deeper recursion hierarchy, which implies that the the QUICKSELECT needs a much higher number of kernel invocations.

G. Approximate selection

Many problem settings do not require an accurate selection process, but can accept an approximate splitter identification. For this setting, we reduce the SAMPLESELECT algorithm to a single recursion level. This "approximate SAMPLESELECT" algorithm computes only the bucket counts, and selects the splitter that is closest to the target rank. Obviously, this introduces some approximation error, while radically reducing the computational (and memory) work. In Figure 10 we visualize the throughput performance (y-axis) and relative approximation error in terms of the element rank (x-axis) for both the (exact) SAMPLESELECT implementation and the inexact SAMPLESELECT variant. The problem setting uses 2^{28} uniformly-distributed single precision values, the approximate SAMPLESELECT algorithm (green triangles) is evaluated for configurations using 128, 256, 512, and 1024 buckets. Obviously, the accuracy decreases for smaller bucket counts, and for using only 64 buckets, the relative approximation error grows up to almost 1%. At the same time, this variant executes almost three times faster than the exact SAMPLESELECT (blue circle). For larger bucket counts, the accuracy increases, and when using 1024 buckets, 50% runtime savings compared to the exact SAMPLESELECT come with an average relative approximation error of .1%. An important observation in this context is that while the error has a large variability based on the random sample choice, the performance impact of a larger bucket count is rather small, so in the approximate selection algorithm, it seems advisable to always use the maximal bucket count for which the `sample` and `count` kernels stay within the shared memory limits ($b \leq 1024$ on older NVIDIA GPUs).

VI. CONCLUSION

We have proposed a new parallel selection algorithm for GPUs. The SAMPLESELECT algorithm employs a set of splitters for partitioning the input dataset, and low-level synchronization mechanism to preserve much of the asynchronous execution mode of modern GPUs. The SAMPLESELECT is competitive to specialized In comparison to state-of-the-art GPU implementations that impose strong assumptions on the input data distribution, the SAMPLESELECT is competitive in runtime while being immune to of coping also with unpleasant data distributions. We also propose an approximate SAMPLESELECT variant that terminates before reaching the lowest recursion level, therewith introducing moderate approximation errors. This variant can still be attractive for problem settings where the (significantly faster) identification of approximate splitters is acceptable. In future research we will research on extending the SAMPLESELECT algorithm to other typical selection applications like multiple sequence selection, and the extension to a complete sorting algorithm.

ACKNOWLEDGMENTS

This work was supported by the "Impuls und Vernetzungsfond" of the Helmholtz Association under grant VH-NG-1241.

REFERENCES

[1] C. A. R. Hoare, "Algorithm 65: Find," *Commun. ACM*, vol. 4, no. 7, pp. 321–322, Jul. 1961.

[2] ——, "Quicksort," *The Computer Journal*, vol. 5, no. 1, pp. 10–16, 1962.

[3] M. Blum, R. W. Floyd, V. Pratt, R. L. Rivest, and R. E. Tarjan, "Time bounds for selection," *Journal of Computer and System Sciences*, vol. 7, no. 4, pp. 448–461, 1973.

[4] W. D. Frazer and A. C. McKellar, "Samplesort: A sampling approach to minimal storage tree sorting," *J. ACM*, vol. 17, no. 3, pp. 496–507, 1970.

[5] P. Sanders and S. Winkel, "Super scalar sample sort," in *Algorithms – ESA 2004*, 2004, pp. 784–796.

[6] F. Mosteller, "On some useful "inefficient" statistics," *The Annals of Mathematical Statistics*, vol. 17, no. 4, pp. 377–408, 1946.

[7] N. K. Govindaraju, B. Lloyd, W. Wang, M. Lin, and D. Manocha, "Fast computation of database operations using graphics processors," in *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '04. ACM, 2004, pp. 215–226.

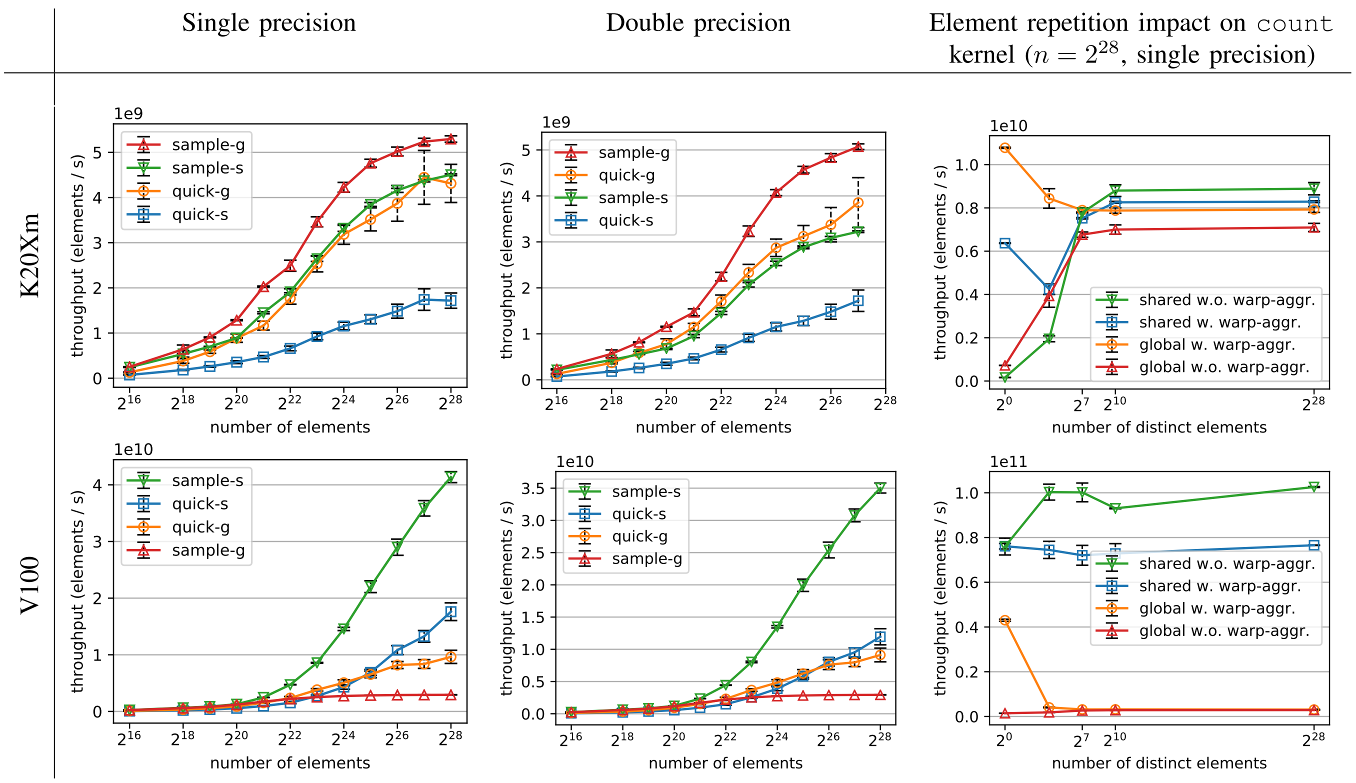

Fig. 8. Comparison of different selection algorithms (left and middle) on the K20Xm and V100 GPUs and the impact of repeating elements on atomic collisions and warp-aggregation (right). *sample* and *quick* denote SAMPLESELECT and QUICKSELECT, respectively. The suffixes *-g* and *-s* denote kernels using shared-memory and global-memory atomics.

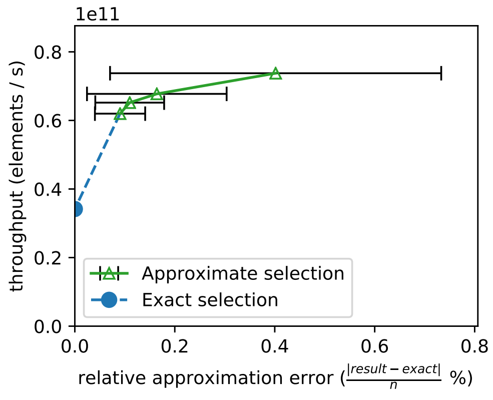

Fig. 10. Error–throughput plot for $n = 2^{28}$ (single precision) and different bucket counts (128, 256, 512, 1024) as well as the exact SAMPLESELECT baseline on the V100 GPU

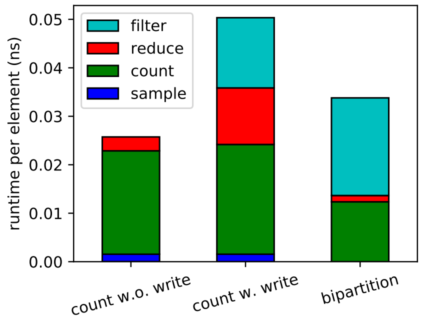

Fig. 9. Runtime breakdown for the elementary kernels using shared-memory atomics on a V100 GPU ($n = 2^{24}$, single precision)

[8] G. Beliakov, “Parallel calculation of the median and order statistics on GPUs with application to robust regression,” 2011, unpublished preprint: arXiv 1104.2732.

[9] L. Monroe, J. Wendelberger, and S. Michalak, “Randomized selection on the GPU,” in *Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics*, ser. HPG ’11, 2011, pp. 89–98.

[10] T. Alabi, J. D. Blanchard, B. Gordon, and R. Steinbach, “Fast *K*-selection algorithms for graphics processing units,” *J. Exp. Algorithmics*, vol. 17, pp. 4.2:4.1–4.2:4.29, Oct. 2012.

[11] L. Hübschle-Schneider and P. Sanders, “Communication efficient algorithms for top-k selection problems,” in *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2016, pp. 659–668.

[12] K. E. Batcher, “Sorting networks and their applications,” in *Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference*, ser. AFIPS ’68 (Spring). ACM, 1968, pp. 307–314.

[13] D. Bakunas-Milanowski, V. Rego, J. Sang, and C. Yu, “A fast parallel selection algorithm on GPUs,” in *2015 International Conference on Computational Science and Computational Intelligence (CSCI)*, Dec 2015, pp. 609–614.

[14] F. H. Mathis, “A generalized birthday problem,” *SIAM Rev.*, vol. 33, no. 2, pp. 265–270, 1991.

[15] A. Adinets. Optimized filtering with warp-aggregated atomics. [Online]. Available: https://devblogs.nvidia.com/cuda-pro-tip-optimized-filtering-warp-aggregated-atomics/

[16] J. Aliaga, J. Pérez, and E. S. Quintana-Orti, “Systematic fusion of cuda kernels for iterative sparse linear system solvers,” in *Euro-Par 2015: Parallel Processing*, 08 2015, pp. 675–686.

[17] N. Sakharnykh. Fast histograms using shared atomics on Maxwell. [Online]. Available: https://devblogs.nvidia.com/gpu-pro-tip-fast-histograms-using-shared-atomics-maxwell/