Parallel Computing xxx (xxxx) xxx

Contents lists available at ScienceDirect

Parallel Computing

journal homepage: www.elsevier.com/locate/parco

Parallel selection on GPUs

Tobias Ribizel[a], Hartwig Anzt[a,b,*]

[a] Steinbuch Centre for Computing, Karlsruhe Institute of Technology, Germany
[b] Innovative Computing Lab, University of Tennessee, USA

A R T I C L E   I N F O

Article history:
Received 25 July 2019
Revised 22 October 2019
Accepted 5 November 2019
Available online xxx

Keywords:
Parallel selection algorithm
GPU
Multiselection
*k*th order statistics
Approximate selection

A B S T R A C T

We present a novel parallel selection algorithm for GPUs capable of handling single rank selection (single selection) and multiple rank selection (multiselection). 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 leveraging 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 proposed *SampleSelect* algorithm does not work on the actual element values but on the element ranks of the elements only, it is robust to the input data and can complete significantly faster for adversarial data distributions. We also address the use case of approximate selection by designing a variant that radically reduces the computational cost while preserving high approximation accuracy.

1. Introduction

Selecting a single rank or multiple ranks of an unordered sequence of elements is an ubiquitous challenge that appears in many problem settings, from quantile selection in order statistics over determining thresholds in approximation algorithms to top-k selection in information retrieval. Among the most popular solutions to this problem is the heavily-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 *SampleSort* 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 hardware-aware optimizations. The traditional concepts employed for the parallelization of selection and sorting algorithms are primarily based on decomposing the input dataset. This approach based on splitting the workload across the parallel resources has 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 on 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 nowadays not only adopted by a large fraction of the supercomputing facilities but also a central ingredient of embedded devices powering the mobile market, there exists a heavy demand for selection algorithms that are designed to leverage the highly parallel execution model of GPUs by avoiding global synchronization and communication in favor of localized communication. In response to this demand, we propose a new parallel selection algorithm for GPUs that is capable of single rank selection and multiple rank selection. With the goal of efficiently leveraging the compute power of modern GPUs, we follow a bottom-up approach by starting with the GPU hardware characteristics, and designing the selection algorithm out of 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 proves 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,

* Corresponding author.
E-mail addresses: hanzt@icl.utk.edu, hartwig.anzt@kit.edu (H. Anzt).

https://doi.org/10.1016/j.parco.2019.102588
0167-8191/© 2019 Elsevier B.V. All rights reserved.

open source software, or vendor libraries. While we initially proposed the SampleSelect in [6], we here extend the functionality of the basic algorithm to handle also multiple rank selection (multiselection). Multiselection is needed, for example, if data elements within a specific range are of interest. Obviously, multiselection can always be realized via consecutive invocation of single rank selection. However, our multiselection-ready SampleSelect algorithm is much faster, introducing only moderate overhead when selecting multiple ranks instead of a single rank.

The rest of the paper is organized as follows. In Section 2 we recall some basic concepts of selection algorithms and their parallelization potential. Section 3 lists efforts in parallelizing selection and multiselection algorithms. In Section 4 we present the SampleSelect algorithm efficient in single and multiple rank selection. 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 5 presents a comprehensive analysis of the effectiveness, efficiency, and performance of the novel SampleSelect selection algorithm targeting both, single and multiple rank selection. We conclude in Section 6 with a summary of the algorithm capabilities and its performance evaluation.

2. 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. The formulation can also be extended to the *multiselection problem*, i.e., given a sequence $k_1, \dots, k_m$ of ranks, we want to find the elements $x_{i_{k_j}}$ with these ranks. Multiselection is needed, for example, when identifying elements in a range, or when computing quantiles in statistical analysis.

2.1. General framework

The most popular algorithms for the selection problem are all based on *partial sorting*: If we choose $b+1$ so-called *splitter elements* s_i $(-\infty = s_0 \le \cdots \le s_b = \infty)$, 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 buckets containing the target ranks. 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 a bucket-based selection for single and multiple rank selection is given in Fig. 1 and visualized in Fig. 2. Virtually all popular (multi-)selection algorithms are based on this approach of recursive bucket selection. Note that for multiselection, we assume the target ranks to be sorted, as every bucket then contains a contiguous range of the target ranks which significantly simplifies the recursion process.

2.2. 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). In the lowest recursion level, we sort the elements using bitonic sort to return the elements with the desired ranks 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 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: $X_1, \dots, X_s \sim \mathcal{U}(0,1)$ and (assuming sampling with replacement) independent. Thus, the sample percentiles are asymptotically normally distributed with mean p_i and standard deviation $\sqrt{p_i(1-p_i)/s}$ [7]. Consequently, we can modulate the sample size s to control the imbalance between different bucket sizes.

2.3. Approximating the kth-smallest elements

An important observation in the context of bucket-based selection algorithms is that the ranks of the splitter elements are available once all elements have been grouped into their buckets: Their ranks equal the aforementioned partial sums r_i. If the application does not require our exact ranks, but can work with an approximation like the $k \pm \varepsilon$th smallest element, the selection algorithm can be modified to terminate before reaching the lowest recursion level. In this case, it is possible to 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 remains smaller than half the maximum bucket size, and can thus effectively be controlled via modulating the number of buckets and the sample size. If the distribution of the input data is smooth, a small error in the element rank translates to a small error in the element value. However, this is not true for the general case, as for non-smooth (i.e. clustered) input data, the induced element value error can grow arbitrarily large. Approximate selection is attractive for algorithms that favor moderate inaccuracies over invoking expensive exact selection algorithms. For example, the recently developed parallel threshold ILU algorithm for GPUs (ParILUT [8]) is interested in quickly obtaining approximate thresholds [9].

3. 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. [10] 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 [11] – it is based on the reformulation of the median selection as a convex optimization problem. Monroe et al. [12] 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. [13] 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

```
double select(data, rank) {
    if (size(data) <= base_case_size)
        sort(data);
        return data[rank];
    // select splitters
    splitters = sample(data);
    // compute bucket sizes n_i
    counts = count(data, splitters);
    // compute bucket ranks r_i
    offsets = prefixsum(counts);
    // determine bucket containing rank
    bucket = binarysearch(offsets, rank);
    // recursive subcall
    data = filter(data, bucket);
    rank -= offsets[bucket];
    return select(data, rank);
```

```
double[] multi_select(data, ranks)
    if (size(data) <= base_case_size)
        sort(data);
        return data[ranks];
    // select splitters
    splitters = sample(data);
    // compute bucket sizes n_i
    counts = count(data, splitters);
    // compute bucket ranks r_i
    offsets = prefixsum(counts);
    // determine buckets containing ranks
    [mask, subranks] =
            compute_bucket_mask(offsets, ranks);
    // adjust ranks for subcalls by previous counts
    subranks -= offsets;
    // copy data from unmasked buckets
    data = filter(data, mask, offsets);
    // recursive subcalls
    for (bucket b : buckets)
        subdata = data[offsets[b]:offsets[b + 1]];
        result += multi_select(subdata, subranks);
    return result;
```

Fig. 1. High-level overview of a bucket-based selection algorithm: Selecting a single rank (left) and multiple ranks (right).

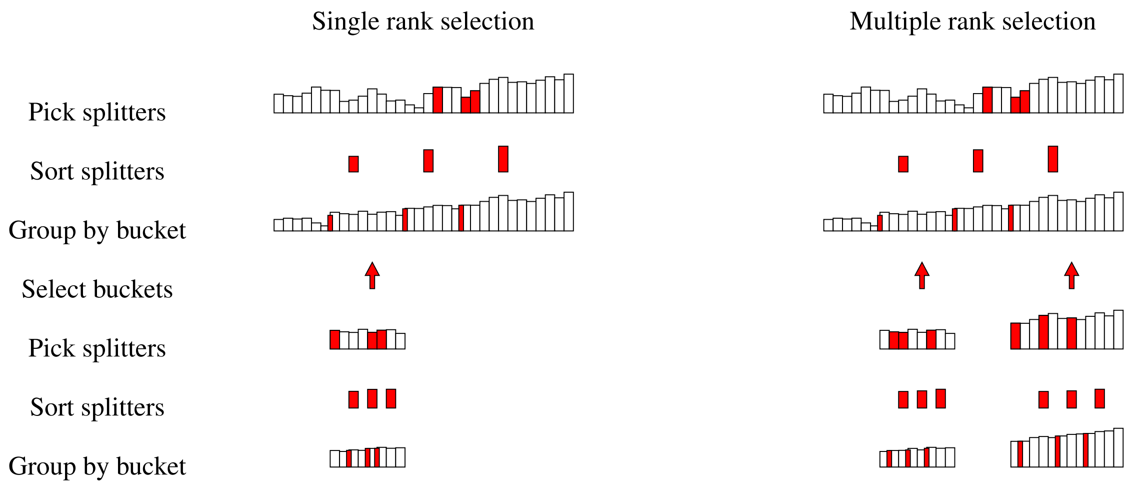

Fig. 2. Visualization of bucket-based partial sorting: Single rank selection (left) and multiselection (right).

distributed systems have been presented by Hübschle-Schneider and Sanders [14].

The (parallel) multiselection problem was studied in the PRAM parallel programming model by Olariu and Wen [15] as well as in a mesh-connected setting by Shen [16]. Orthogonally to that, Kaligosi et al. [17] provided a theoretical upper bound of the optimal runtime of multiselection algorithms in the sequential case.

While all these algorithms were designed to select only a single k-smallest element or the top-k elements from a sequence, both BucketSelect and RadixSelect can be extended in the framework from Fig. 1 to build an algorithm for multiselection. Unlike most previous efforts designing selection routines for GPUs, our algorithm is purely comparison-based, i.e., we only use the relative order and ranks of elements to determine the kth-smallest element(s). This especially means that the algorithm can operate on more complex data types like tuples with lexicographic order, with only minimal changes.

4. Implementation

4.1. Optimizing for memory bandwidth

Any algorithm selecting a target rank from a non-ordered sequence needs to access 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 on average $(1+\varepsilon)n$ element read- and write operations for a single input rank, with a small and configurable ε parameter and auxiliary storage of size smaller $n/4$ in single precision arithmetic and $n/8$ in double precision arithmetic in terms of the input element size, respectively.

For multiselection, it is not possible to derive tight general bounds, as complexity and storage requirements heavily depend on the distribution of the target ranks. In particular, all target ranks being clustered results in complexity and storage requirements close to the single selection case, while both metrics grow for uniformly distributed target ranks. Nevertheless, in comparison to QuickSelect sorting the complete dataset, separating the input data into multiple buckets in SampleSelect results in traversing a much smaller area of the recursion tree.

4.2. SampleSelect

At its core, our SampleSelect implementation consists of three elementary kernels, whose pseudocode is also listed in Fig. 4:

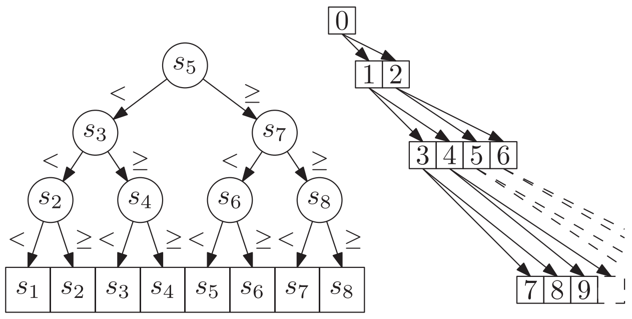

Fig. 3. Search tree based on bucket splitters $s_1, \ldots, s_8$ (left) and its implicit array storage order (right).

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. Using a bucket bitmask, the `filter` kernel extracts the elements of a single bucket (single rank selection) or a set of buckets (multiple rank selection).

Sample kernel To select a suitable splitter set for the following steps, our `sample` kernel first reads a small sample of elements into shared memory and sorts them using a bitonic sorting network [18]. From the resulting set, we pick the i/b percentiles for $i = 1, \ldots, b-1$, and store them in global memory. These percentiles separate the input data into b roughly equal-sized buckets.

Count kernel The `count` kernel combines two important steps: First, it identifies the target bucket an element belongs into. Then it increments the shared counter for this bucket. The bucket index could be identified using a binary search on the sorted splitter array, but the involved index calculations would be complicated and expensive.

To alleviate this bottleneck, we decided to place the splitters in a complete binary search tree that is implicitly stored in an array, like suggested in [5]. The indexing of this array is based on an 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. For efficient processing on standard hardware, we use a single byte to store each oracle, which limits the scope of this approach to at most 256 buckets. The indexing and search tree traversal are visualized in Fig. 3.

Filter kernel The `filter` kernel scans over all oracles, loading only the elements belonging to the bucket(s) containing the target rank, and then stores these elements. Writing the elements of these buckets in contiguous fashion requires using shared counters that hold the next unused index in the contiguous storage for each bucket.

4.3. 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 $s_a = \ldots = s_e < s_{e+1}$ 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(s) are contained in such an equality bucket, the algorithm can terminate early by returning the corresponding lower bound splitter. Without this modification, the algorithm cannot be guaranteed to terminate, as a sequence of equal elements would always be placed in the same bucket. Without equality buckets however, we would not be able to determine that this bucket only contains equal elements, which would lead to an infinite recursion.

4.4. 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 [18] 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 operations. Our implementation is an extension of the bitonic sorting kernels with n-to-m data binding introduced by Hou et al. [19], using register shuffle operations to sort the data within a warp. As a trade-off between local performance and register pressure/maximal occupancy, we store $n = 4$ local elements per thread. In the distinct algorithm implementations, the bitonic sorting implementation is used for the splitter selection in SampleSelect, the pivot selection in the QuickSelect algorithm we implemented for performance comparison, and for the recursion base case in both algorithms.

4.5. Recursion

As the recursion depth of our algorithms is not exactly known a-priori, and communication/synchronization between the host processor and the GPU usually introduces significant latencies, we use CUDA Dynamic Parallelism to keep the control flow completely on the GPU (This feature allows GPU kernels to asynchronously launch new subsequent kernels). Acknowledging that all kernels launched from the CPU or a single thread block on the GPU will be executed in the launch order, we are able to implement a simple tail-recursion. For the purpose of this recursion, we introduce additional kernels that select the bucket(s) containing the kth-smallest element(s), and compute the kernel launch parameters for the subsequent recursion level.

Specifically, for multiselection, we execute a binary search on the input ranks for every bucket in parallel. This determines where the splitter ranks are placed in the sorted rank sequence, and thus determining which rank(s) can be found in which bucket. For each non-empty bucket, a multiselection kernel for the next recursion level is invoked. Launching many sub-kernels simultaneously can result in substantial overhead. To mitigate this effect, for a larger number of subcalls, we instead launch a combined kernel where each thread block processes the elements from a single input bucket from the previous recursion level. This has the additional advantage that we need no reduction and global synchronization, as all counts are accumulated in shared memory.

4.6. Reference implementation: QuickSelect

As a reference point in the performance evaluation, we implemented a GPU version of the QuickSelect algorithm, and apply the same performance optimizations like for the SampleSelect implementation. 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 bi-partitioned. This difference leads to simpler treatment of a single input element, but in general requires more recursion levels and more data access operations than SampleSelect.

As a basic building block, we implemented a branchless bipartition kernel that realizes the selection by growing the array of elements smaller than the pivot from the left and elements larger sample

```
for (i = 0; i < n_sample; i++) // parallel
    sample[i] = input[random(size)];
sort(sample);                  // parallel
for (i = 0; i < n_bucket; i++) // parallel
    splitter[i] = sample[i*n_sample/n_bucket];
```

reduce

```
// when using shared-memory atomics:
// compute block-wise partial sums
for (i = 0; i < n_buckets; i++)    // parallel
    for (b = 0; b < n_blocks; b++)
        block_partial[b + 1][i] =
            block_partial[b][i] +
            block_counts[b][i];
    counts[i] = block_partial[n_blocks][i];

bucket_partial = prefixsum(counts) // parallel
```

count

```
for (i = 0; i < size; i++)     // parallel
    el = input[i];
    t = 0;
    for (lvl = 0; lvl < tree_height; lvl++)
        t = 2 * t + (el < tree[t] ? 1 : 2);
    bucket = t - (tree_width - 1);
    counts[bucket]++; // atomic
    oracles[i] = bucket;

// when using shared-memory atomics:
// store partial sum for thread block b
for (i = 0; i < n_bucket; ++i) // parallel
    block_counts[b][i] = counts[i];
```

filter

```
// initialize offsets for thread block b
for (i = 0; i < n_buckets; i++) // parallel
    offset[i] = bucket_partial[i]
        + block_partial[b][i] // shared memory

for (i = 0; i < size; i++) // parallel
    bucket = oracles[i];
    if (bucket_mask.contains(bucket))
        output[offsets[bucket]] = input[i];
        offsets[bucket]++; // atomic
```

Fig. 4. Elementary kernels for SampleSelect (Pseudocode).

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

```
int bucket;
int mask = 0xffffffff; // start with full warp
for (int b = 0; b < tree_height; b++)
    bool bit = bucket & (1 << b) != 0;
    int step_mask = ballot(bit);
    if (bit)
        // keep threads that have the bit set
        mask = mask & step_mask;
    else
        // keep threads that have the bit unset
        mask = mask & ~step_mask;
```

Fig. 5. Branchless partitioning algorithm for QuickSelect (left) and warp-aggregation for bucket indexes (right).

than the pivot from the right. Pseudocode for the bipartition kernel is provided in Fig. 5.

4.7. Reference implementation: BucketSelect

We also compare our SampleSelect implementation to the previously mentioned BucketSelect algorithm. In its core approach, it is identical to SampleSelect except for the splitter choice: Instead of using a larger sample, it only uses the maximum and minimum and derives the uniformly spaced splitters $min + i/(b\cdot(max - \min))$, which allows us to derive the bucket index of an element x as

$$\left\lfloor \frac{x - min}{max - min}\cdot b\right\rfloor.$$

4.8. 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 either on shared or on global memory. However, operations on global memory usually require a large degree of synchronization (across all thread blocks), 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.

The pseudocode in Fig. 4 shows how this process is incorporated into SampleSelect: The `count` kernel computes the bucket counts for the elements processed by each thread block. They are then stored in global memory, and a prefix sum of these partial counts is computed in the `reduce` kernel. Step 3 is incorporated into the `filter` kernel, in which the result of the atomic operation is the target index for the location where the current element is stored. This is attractive as both kernels (`count` and `filter`) operate on the same element indexes, hence the prefix sums from one operation can be used in the next. The use of shared-memory atomics in `filter` is comparable to the filtering approach introduced in [20], 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 using atomics is that the algorithm performance can be impacted by atomic collisions. These collisions occur if multiple threads issue atomic operations on the same operand/memory location. While even for uniformly distributed datasets there exists a significant chance for these collisions [21], they are expected to occur frequently for “nasty” distributions containing clusters. A mitigation strategy that reduces the number of atomic collisions is *warp-aggregation* [22]. 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. 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 Fig. 5: For a fixed thread, the loop computes a bitmask containing all threads of the warp that increase the same bucket index (and would thus incur an atomic collision). In the implementation of the bucket count kernel, the mask computation can be overlapped with the searchtree traversal, potentially hiding latencies from shared memory access.

4.9. Tuning parameters

The SampleSelect and QuickSelect implementations feature several tuning parameters and configuration options for hardware-aware and problem-specific optimizations: ***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. ***Sample size*** A larger sample used to select the bucket splitters generally improves the splitter quality. In 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. ***Number of buckets*** A larger number of buckets increases the accuracy of a single recursion level, and therewith decreases the recursion depth of SampleSelect. However, 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. ***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 generally increases the register pressure of the kernel, potentially reducing the occupancy per streaming multiprocessor (SM). ***Atomics*** The performance characteristics of global and shared memory atomics are very architecture-dependent. Furthermore, the warp-aggregation alleviating the performance impact of atomic collisions occurring for “nasty” distributions introduces some overhead for the general case. ***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 depth, we consider the impact negligible.

4.10. Kernel fusion

Aside from its stand-alone form, the SampleSelect kernel is amenable to kernel fusion [23] if not only the kth-smallest element is required, but, for example, all elements from a contiguous rank range or all elements larger than the kth-smallest element are of interest (the latter is often denoted 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 which are contained in the desired rank range. As the splitters are ordered, the recursion still needs to descend only into the buckets containing the delimiters of the rank range.

5. Experiments

In the experimental evaluation of the SampleSelect implementation, we assess its performance for different parameter configurations in comparison to the QuickSelect implementation, as well as a BucketSelect [13] implementation we developed for comparison by replacing the element classification in SampleSelect. We consider two GPU architectures belonging to distinct compute generations, and a set of input datasets varying in size and value distribution. For full experiment reproducibility, we make the source code and all benchmark data available at https://github.com/upsj/gpu_selection under the permissive GNU GPLv3 license.

5.1. 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} elements, containing $d = 1, 16, 128, 1024$ and n distinct values. Using datasets with $d < n$ allows us to evaluate the performance impact of repeating elements resulting in atomic collisions. For each dataset, we chose the target ranks randomly out of a uniform distribution to simulate a variety of different workloads for single selection.

For the multiselection, we evaluate two different distributions of input ranks: `uniform` contains 32 ranks evenly distributed over the dataset range, which corresponds to a worst-case input for both, QuickSelect and SampleSelect, as each rank is contained in a different bucket. `clustered` contains the $\log_2 n$ ranks $2^i < n$, which is close to a best case for both selection algorithms, as most buckets can be discarded early.

To account for variations introduced by the random target rank selection, we run each experiment on 10 distinct input datasets and report the average data along with the standard deviation. We ensure correctness of the SampleSelect implementation by comparing the results to a reference solution based on the sorted input data computed using the `std::sort` algorithm from the C++ standard library.

5.2. 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 1. The kernels are compiled using the CUDA 10.1 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, BucketSelect and SampleSelect are all linear-time algorithms for a fixed number of input ranks. As performance metric, we consider the “throughput” which derives as ratio between the dataset cardinality (not accounting for the distribution of the values) and the algorithm runtime. To reflect stochastic effects incurred by considering different input datasets and hardware jitter, we always consider 10 different input datasets and report the arithmetic mean along with the standard deviation.

5.3. Parameter tuning

As elaborated, SampleSelect features a list of parameters amenable to hardware-specific tuning. In Fig. 6 we analyze the

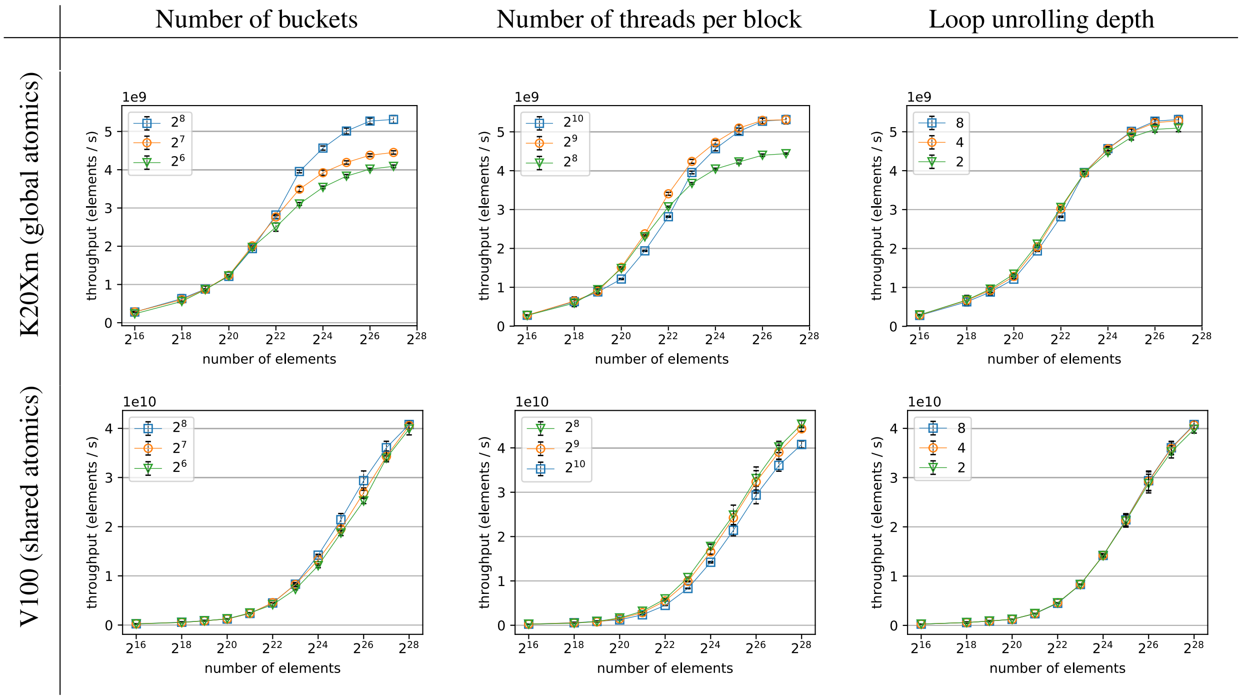

Fig. 6. Parameter tuning benchmarks (single precision). Based on preliminary experiments [6], we only visualize the performance using global memory atomics on the K20Xm and shared memory atomics on the V100, as these are the fastest configurations on the respective platform.

Table 1
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.

	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

effect of different parameter choices on the overall performance of the single selection algorithm implementations [6]. We notice that 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 by the loop unrolling depth. The number of threads accumulated in a block (for a fixed number of buckets) should be chosen as large as possible on the K20 architecture, while the V100 GPU favors smaller thread blocks.

5.4. Performance comparison for single selection

In Fig. 7 we present the performance analysis from [6] assessing the algorithm throughput for different input sizes to compare the shared-memory variants and global-memory variants of QuickSelect, BucketSelect and SampleSelect for single rank selection. 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”, “bucket-g” and “quick-g”) are generally faster than their shared-memory counterparts (“sample-s”, “bucket-s” and “quick-s”, respectively). Independent of the precision format, the performance differences are significant in particular for the QuickSelect algorithm. On the other hand, 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 the SampleSelect almost matches its single precision throughput. As the atomics always operate on 32bit integers, this suggests that the atomic operations expose the bottleneck for the SampleSelect implementation, whereas the performance of the QuickSelect algorithm is primarily 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 (that practically exceed the GPUs' main memory capacity). Despite the data distribution being close-to-optimal for the BucketSelect algorithm, the performance comparison reveals that BucketSelect barely outperforms SampleSelect for single precision input on the K20Xm, and it is inferior for all other configurations. This, again, indicates that the bottleneck in these algorithms not being the element classification itself, but the histogram-like bucket counting operation. Furthermore, as BucketSelect was previously reported to be the fastest GPU selection algorithm for uniform input data [10–13], our SampleSelect algorithm is competitive to BucketSelect even in the best-case-setting. Moreover, as SampleSelect operates on the element ranks

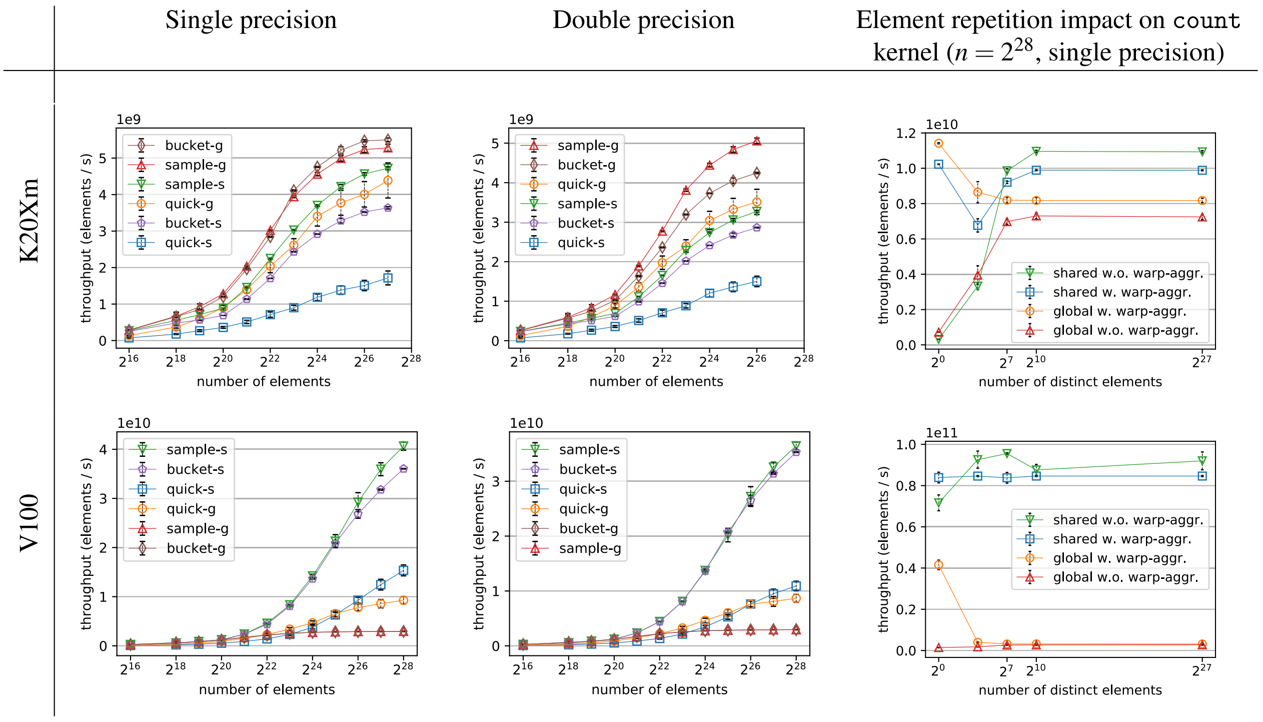

Fig. 7. Comparison of different single selection algorithms (left and middle) on the K20Xm and V100 GPUs and the impact of repeating elements on atomic collisions and warp-aggregation (right). *sample, bucket* and *quick* denote SampleSelect, BucketSelect and QuickSelect, respectively. The suffixes *-g* and *-s* denote kernels using shared-memory and global-memory atomics.

only, its performance remains mostly unaffected from adversarial input datasets.

5.5. Data distribution and intra-warp communication

On the right-hand side of Fig. 7, we assess the influence of the data distribution, in particular the impact of collisions resulting from element repetition [6]. The distinct communication strategies differ in the effectiveness 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 for the general case. The fast shared-memory atomics (initially introduced with the Maxwell architecture [24]) make warp-aggregation unnecessary on the V100 GPU.

5.6. Runtime breakdown

In Fig. 8, we visualize for the different kernels the runtime breakdown of a single recursion level of SampleSelect and QuickSelect on the V100 and K20Xm GPUs with the respectively fastest configuration (global-memory atomics on K20m and shared-memory atomics on V100). We observe that the recording of oracles (“count with write”) introduces only negligible overhead to the runtime of the `sample` and `count` kernels of SampleSelect, as we can see in the two middle green bars. Opposed to that, the reduction for shared-memory atomics becomes more expensive. The reason behind is that in addition to the total bucket counts, also the partial sums need to be stored, 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 for single selection in SampleSelect, which can likely be linked to the larger memory footprint of the elements compared to their oracles. The runtime for the multiselection `filter` kernel of SampleSelect included in Fig. 8 reflects a setting where 128 of the 256 buckets are extracted. Even though the total amount of elements being written to main memory is only half compared to the QuickSelect kernel, the cost of the multiselection `filter` kernel is much higher. The reason behind is that the element writes require random access and use multiple atomic counters to keep track of the next free index for each bucket. However, the lower cost of a QuickSelect kernel comes at the cost of a much deeper recursion hierarchy, and hence a much higher number of kernel invocations.

5.7. Multiselection

Fig. 9 shows the performance of the multiselection implementations of QuickSelect and SampleSelect on the `clustered` and `uniform` input datasets, with the RadixSort algorithm from NVIDIA’s CUB library [25] as a baseline. We consider both single and double precision arithmetic. For brevity, we limit the analysis on the distinct hardware architectures to the algorithm configurations that performed best in the single selection case. This is the shared-memory atomics variant on the V100 GPU and the global-memory atomics variant on the K20m GPU. Independent of hardware architecture and precision, the `clustered` multiselection is faster than the `uniform` multiselection for large inputs and SampleSelect outperforms QuickSelect by more than 2 × . This reflects the fact that clustered target ranks require less buckets to be extracted in the recursion levels. On the K20m GPU, the clustered target ranks result in about 35% higher throughput compared to uniformly distributed target ranks for SampleSelect. On the V100, SampleSelect reaches a roughly 25% larger

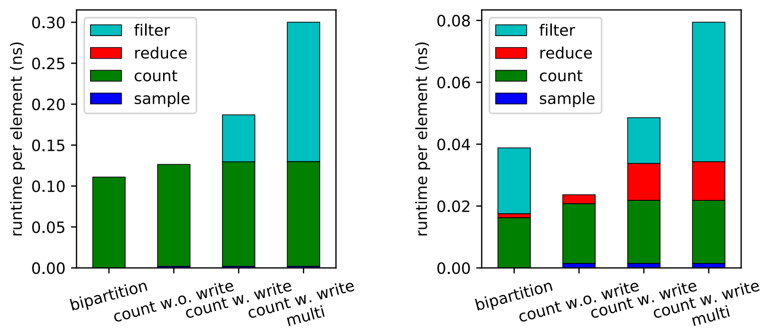

Fig. 8. Runtime breakdown for the elementary kernels with $n = 2^{24}$ (single precision) using global-memory atomics on a K20Xm (left) and shared-memory atomics on a V100 GPU (right). The kernels are (from left to right): bipartitioning for QuickSelect (bipartition), counting the number of elements for SampleSelect (count w.o. write), counting and extracting the elements from a single bucket (count w. write), counting and copying the elements from every second bucket (count w. write multi).

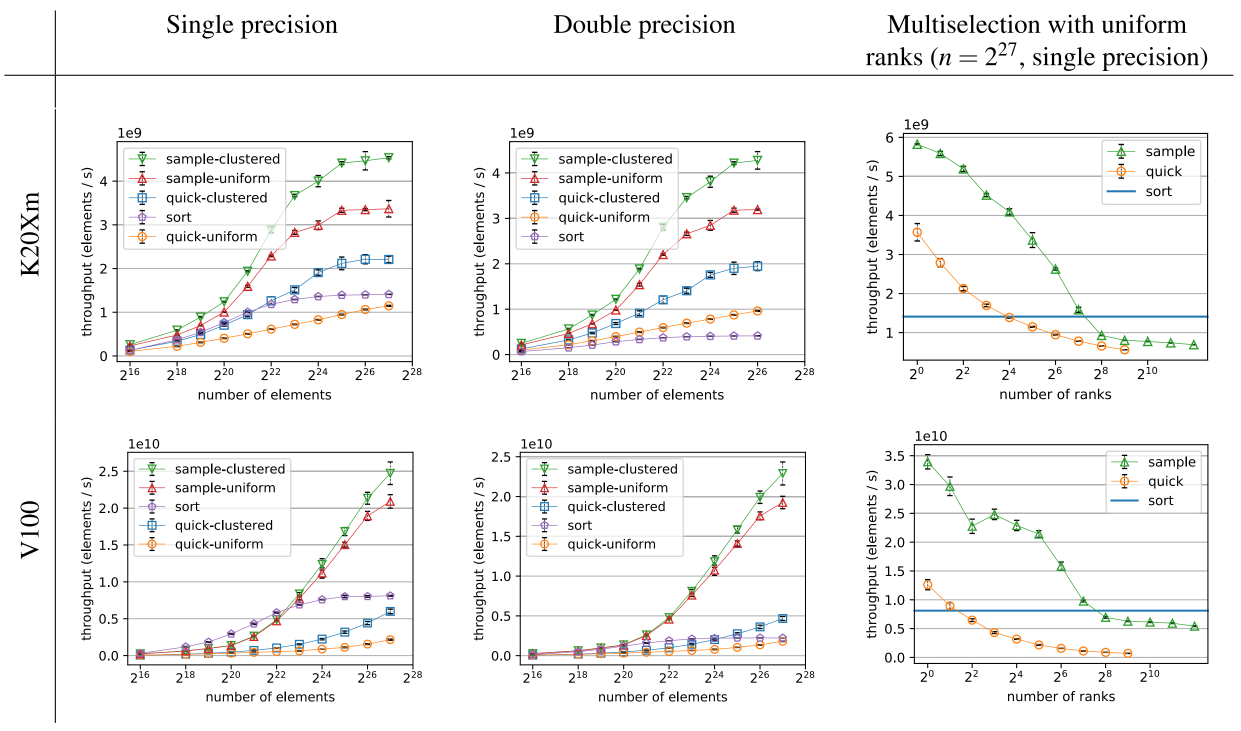

Fig. 9. Comparison of different multiselection algorithms using global-memory atomics on the K20Xm and shared-memory atomics on the V100 GPUs. *sample* and *quick* and *sort* denote SampleSelect, QuickSelect and RadixSort, respectively. The suffixes *-clustered* and *-uniform* denote `clustered` and `uniform` input ranks.

throughput for clustered ranks. If we compare to the single selection setting on the K20m architecture, the multiselection throughput of SampleSelect decreases to about 80% for the `clustered` ranks and about 60% for the `uniform` ranks. On the V100, the performance difference to the single selection case is more significant. For the `clustered` target ranks, the multiselection SampleSelect reaches roughly 60% of the single selection throughput, for `uniform` target ranks the multiselection SampleSelect reaches about 50% of the single selection throughput. RadixSort reaches less than half of the total throughput of SampleSelect for larger inputs, but still manages to outperform QuickSelect in some configurations for single-precision inputs on both GPUs. For double precision, the throughput of RadixSort drops much faster than all other algorithms, most likely due to its larger dependency on the element size and increased memory footprint. For a scenario where we increase the number of (uniformly distributed) ranks we want to select from the input, Fig. 9 additionally shows the runtime of SampleSelect and QuickSelect in comparison to the RadixSort baseline: SampleSelect is faster than RadixSort for up to 128 ranks, while QuickSelect is dominated by the sorting algorithm much earlier. These results indicate that the SampleSelect multiselection performance could be improved by a better fine-tuning for smaller input sizes.

5.8. Approximate selection

Many problem settings do not require an accurate selection result, but can accept an element close to the target rank. 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. Thus, we are not limited by the 256 bucket-limit that is imposed by the oracle storage (explained in Section 4). Obviously, this introduces some approximation error, while radically reducing the computational (and memory) cost. In Fig. 10 we visualize the throughput performance (y-axis) and relative

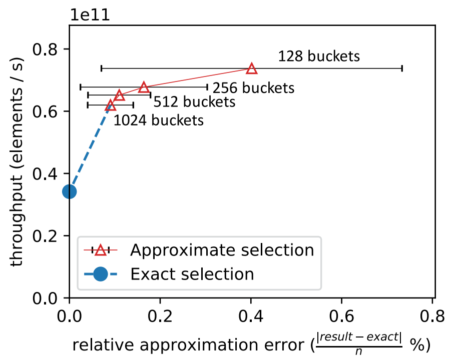

Fig. 10. Error–throughput plot for $n = 2^{28}$ (single precision) and different bucket counts (128, 256, 512, 1024) as well as the exact SampleSelect baseline using shared-memory atomics on a V100 GPU.

approximation error in terms of the element rank (x-axis) for both the (exact) SampleSelect implementation and the inexact SampleSelect variant [6]. The problem setting uses 2^{28} uniformly-distributed single precision values, the approximate SampleSelect algorithm (red 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 Sample-Select (blue circle). For larger bucket counts, the accuracy increases, and when using 1024 buckets, 50% runtime savings come at the price of an average relative approximation error smaller than 0.1%. An important observation in this context is that the performance impact of a larger bucket count is relatively small, while the error has a large variability based on the random sample choice. In consequence, for approximate selection it is advisable to always push the bucket count to the hardware-supported limit (i.e. $b \leq 1024$ on older NVIDIA GPUs).

6. Conclusion

We have proposed a new parallel selection algorithm for GPUs that is capable of handling single rank selection and multiple rank selection. The SampleSelect algorithm is based on a partial selection strategy using a set of splitters for partitioning the input dataset, and employs low-level synchronization mechanism to preserve much of the asynchronous execution mode of modern GPUs. In comparison to state-of-the-art GPU implementations that impose strong assumptions on the input data distribution, SampleSelect is competitive in runtime while being immune to the effects of unpleasant data distributions. SampleSelect strongly outperforms the standard QuickSelect algorithm for both, single and multiple selection on the GPU. We also propose an approximate SampleSelect variant that terminates before reaching the lowest recursion level. Despite introducing moderate approximation errors, approximate SampleSelect is interesting for algorithms requiring the quick approximation of target elements.

Declaration of Competing Interest

The authors declare that they do not have any financial or non-financial conflict of interests.

Acknowledgments

This work was supported by the “Impuls und Vernetzungsfond” of the Helmholtz Association under grant VH-NG-1241. The authors would like to thank the anonymous reviewers for their constructive comments.

References

[1] C.A.R. Hoare, Algorithm 65: find, Commun. ACM 4 (7) (1961) 321–322, doi:10.1145/366622.366647.
[2] C.A.R. Hoare, Quicksort, Comput. J. 5 (1) (1962) 10–16, doi:10.1093/comjnl/5.1.10.
[3] M. Blum, R.W. Floyd, V. Pratt, R.L. Rivest, R.E. Tarjan, Time bounds for selection, J. Comput. Syst. Sci. 7 (4) (1973) 448–461, doi:10.1016/S0022-0000(73)80033-9.
[4] W.D. Frazer, A.C. McKellar, Samplesort: a sampling approach to minimal storage tree sorting, J. ACM 17 (3) (1970) 496–507, doi:10.1145/321592.321600.
[5] P. Sanders, S. Winkel, Super scalar sample sort, in: Algorithms – ESA 2004, 2004, pp. 784–796, doi:10.1007/978-3-540-30140-0_69.
[6] T. Ribizel, H. Anzt, Approximate and exact selection on GPUs, in: 2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), 2019, pp. 471–478, doi:10.1109/IPDPSW.2019.00088.
[7] F. Mosteller, On some useful “inefficient” statistics, Ann. Math. Stat. 17 (4) (1946) 377–408, doi:10.1214/aoms/1177730881.
[8] H. Anzt, T. Ribizel, G. Flegar, E. Chow, J. Dongarra, ParILUT - a parallel threshold ILU for GPUs, in: 2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS), 2019, pp. 231–241, doi:10.1109/IPDPS.2019.00033.
[9] H. Anzt, E. Chow, J. Dongarra, ParILUT—a new parallel threshold ILU factorization, SIAM J. Scientif. Comput. 40 (4) (2018) C503–C519, doi:10.1137/16M1079506.
[10] N.K. Govindaraju, B. Lloyd, W. Wang, M. Lin, D. Manocha, Fast computation of database operations using graphics processors, in: Proceedings of the 2004&nbsp;ACM SIGMOD International Conference on Management of Data, in: SIGMOD ’04, 2004, pp. 215–226, doi:10.1145/1007568.1007594.
[11] G. Beliakov, Parallel calculation of the median and order statistics on gpus with application to robust regression, 2011.
[12] L. Monroe, J. Wendelberger, S. Michalak, Randomized selection on the GPU, in: Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics, in: HPG ’11, 2011, pp. 89–98, doi:10.1145/2018323.2018338.
[13] T. Alabi, J.D. Blanchard, B. Gordon, R. Steinbach, Fast *K*-selection algorithms for graphics processing units, J. Exp. Algorithmics 17 (2012), doi:10.1145/2133803.2345676. 4.2:4.1–4.2:4.29
[14] L. Hübschle-Schneider, P. Sanders, Communication efficient algorithms for top-k selection problems, in: 2016&nbsp;IEEE International Parallel and Distributed Processing Symposium (IPDPS), 2016, pp. 659–668, doi:10.1109/IPDPS.2016.45.
[15] S. Olariu, Z. Wen, An efficient parallel algorithm for multiselection, Parallel Comput. 17 (6) (1991) 689–693, doi:10.1016/S0167-8191(05)80059-6.
[16] Hong Shen, Efficient parallel algorithms for selection and multiselection on mesh-connected computers, in: Proceedings 13th International Parallel Processing Symposium and 10th Symposium on Parallel and Distributed Processing. IPPS/SPDP 1999, 1999, pp. 426–430, doi:10.1109/IPPS.1999.760511.
[17] K. Kaligosi, K. Mehlhorn, J.I. Munro, P. Sanders, Towards optimal multiple selection, in: L. Caires, G.F. Italiano, L. Monteiro, C. Palamidessi, M. Yung (Eds.), Automata, Languages and Programming, Lecture Notes in Computer Science, 2005, pp. 103–114, doi:10.1007/11523468_9.
[18] K.E. Batcher, Sorting networks and their applications, in: Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference, in: AFIPS ’68 (Spring), 1968, pp. 307–314, doi:10.1145/1468075.1468121.
[19] K. Hou, W. Liu, H. Wang, W.-c. Feng, Fast segmented sort on GPUs, in: Proceedings of the International Conference on Supercomputing, in: ICS ’17, ACM, 2017, pp. 12:1–12:10, doi:10.1145/3079079.3079105. Event-place: Chicago, Illinois
[20] D. Bakunas-Milanowski, V. Rego, J. Sang, C. Yu, A fast parallel selection algorithm on GPUs, in: 2015 International Conference on Computational Science and Computational Intelligence (CSCI), 2015, pp. 609–614, doi:10.1109/CSCI.2015.132.
[21] F.H. Mathis, A generalized birthday problem, SIAM Rev. 33 (2) (1991) 265–270, doi:10.1137/1033051.
[22] A. Adinets, Optimized filtering with warp-aggregated atomics. URL https://devblogs.nvidia.com/cuda-pro-tip-optimized-filtering-warp-aggregated-atomics/
[23] J. Aliaga, J. Pérez, E.S. Quintana-Orti, Systematic fusion of cuda kernels for iterative sparse linear system solvers, in: Euro-Par 2015: Parallel Processing, 2015, pp. 675–686, doi:10.1007/978-3-662-48096-0_52.
[24] N. Sakharnykh, Fast histograms using shared atomics on Maxwell. URL https://devblogs.nvidia.com/gpu-pro-tip-fast-histograms-using-shared-atomics-maxwell/
[25] D. Merrill, NVIDIA Research, CUB library. URL https://nvlabs.github.io/cub/