Selecting Multiple Order Statistics with a Graphics Processing Unit

JEFFREY D. BLANCHARD, ERIK OPAVSKY, and EMIRCAN UYSALER, Grinnell College

Extracting a set of multiple order statistics from a huge data set provides important information about the distribution of the values in the full set of data. This article introduces an algorithm, bucketMultiSelect, for simultaneously selecting multiple order statistics with a graphics processing unit (GPU). Typically, when a large set of order statistics is desired the vector is sorted. When the sorted version of the vector is not needed, bucketMultiSelect significantly reduces computation time by eliminating a large portion of the unnecessary operations involved in sorting. For large vectors, bucketMultiSelect returns thousands of order statistics in less time than sorting the vector while typically using less memory. For vectors containing $2^{28}$ values of type double, bucketMultiSelect selects the 101 percentile order statistics in less than 95 ms and is more than 8× faster than sorting the vector with a GPU optimized radix sort.

Categories and Subject Descriptors: D.1.3 [Concurrent Programming]: Parallel programming; F.2.2 [Non-numerical Algorithms and Problems]: Sorting and searching; G.4 [Mathematical Software]: General Terms: Algorithms, Design, Experimentation, Performance

1. INTRODUCTION

Selecting an order statistic is a well studied, ubiquitous problem in scientific computing. As the big data phenomenon continues to mature, providing meaningful and accurate descriptions of the huge data sets will become increasingly important. Order statistics are a straightforward tool in this endeavor. For example, the set of 101 percentile order statistics offers a manageable, detailed description of the distribution of the full data set. Typical order statistic selection algorithms are designed to select one order statistic at a time. In this paper, we introduce a parallelized multiple order statistic selection algorithm designed for graphics processing units (GPU) which is able to return thousands of order statistics from huge data sets in less time than sorting the data on a GPU.

There is a large collection of literature devoted to the theory and application of order statistics, for example [Arnold et al. 1992; David and Nagaraja 2003]. The propensity of the work on order statistics considers identifying or approximating the distribution of the order statistics. Common methods for garnering information about order statistics and their distributions include bootstrapping and resampling methods. These techniques provide statistical confidence regarding the distribution of the order statistics and therefore of the full data set. The algorithm presented here exactly identifies a large set of order statistics thereby replacing probabilistic confidence in the order statistics with exact values. A thorough discussion and implementation of this algorithm can be found in [Blanchard et al. 2013].

This work is supported by the National Science Foundation under grant DMS-1112612 and the Grinnell College Mentored Advanced Project program.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.

© 2013 ACM 0000-0000/2013/-ART00 $15.00
DOI: http://dx.doi.org/10.1145/0000000.0000000
ough introduction and description of applications of order statistics can be found in [Evans et al. 2006].

Many algorithms exist for precisely identifying a single order statistic, such as Quickselect or Find [Hoare 1961b] based on the well-known Quicksort [Hoare 1961a]. In 2011, several GPU selection algorithms were announced including an optimization based algorithm cuttingPlane [Beliakov 2011], a randomized but deterministic selection randomizedSelect [Monroe et al. 2011], a radix selection radixSelect [Alabi et al. 2012], and an algorithm based on distributive partitioning bucketSelect [Alabi et al. 2012]. The performance of these four algorithms was extensively compared in [Alabi et al. 2012] and all four algorithms are implemented in the software GGKS: Grinnell GPU k-selection [Alabi et al. 2011].

A naive method of selecting an order statistic, hereafter called sort&choose, is to sort the vector and then simply retrieve the desired value. At the time of the study [Alabi et al. 2012], the fastest known sorting algorithm for GPUs was Merrill and Grimshaw’s radix sort [Merrill and Grimshaw 2011] implemented as thrust::sort in the Thrust library [Hoberock and Bell 2010]. In [Alabi et al. 2012], all four GPU selection algorithms were shown to select any order statistic from data sets with more than $2^{20}$ numerical entries in less time than a thrust::sort implementation of sort&choose. Moreover, bucketSelect and randomizedSelect were observed to have comparable speeds on vectors of floats and are consistently faster than the optimization and radix selection algorithms. For a vector of uniformly distributed, double precision values, bucketSelect was the fastest algorithm boasting a 19× speed-up over sort&choose using thrust::sort.

1.1. Selecting Multiple Order Statistics

Rather than retrieving a single order statistic, we consider the task of precisely selecting a large set of order statistics from a huge data set [Alsuwaiyel 2001; Cardinal and Fiorini 2013; Kaligosi et al. 2005]. Again, sort&choose is a straightforward and very common method of completing this task: simply sort the vector and extract the desired set of order statistics. When a sorted version of the vector is useful for some future task, this is an appropriate algorithm. However, if the sorted version of the vector will not be utilized other than to identify order statistics, a considerable amount of wasted computation has occurred. Alternative single order statistic selection algorithms such as cuttingPlane, randomizedSelect, or radixSelect reduce the set of candidates for a single order statistic by eliminating the majority of the values in the vector. In contrast, bucketSelect assigns each value in the data set to a bucket; when multiple order statistics are desired, these bucket assignments retain information on the locations of all order statistics. This paper introduces an extension of bucketSelect for identifying multiple order statistics, bucketMultiSelect.

bucketMultiSelect is an extension of bucketSelect and is therefore based on selection via distributive partitioning [Allison and Noga 1980]. When implemented as a serial algorithm, selection by distributive partitioning is not competitive with alternative algorithms such as Quickselect. However, in bucketSelect and bucketMultiSelect the act of distributive partitioning is implemented through a communication-avoiding computation which assigns the values in the list to a set of buckets. As the assignment of each entry in the vector is independent of the assignment of any other entry, the algorithm is ripe for parallelization. When the number of buckets is large enough to substantially reduce the candidates for the desired order statistics, the algorithm achieves significant computational gains. For example, in this paper bucketMultiSelect utilizes 11264 buckets to select order statistics from vectors of lengths $2^{20}$ to $2^{29}$. For vectors smaller than $2^{22}$, it is typically faster to sort the vector on the GPU than to employ bucketMultiSelect. The upper limit of $2^{29}$ is defined by the global memory constraint of the NVIDIA Tesla K40 for double precision vectors; for float and unsigned integer vectors of length $2^{30}$ bucketMultiSelect successfully finds large sets of order statistics faster than sort&choose, but bucketMultiSelect is
memory constrained for float and unsigned integer vectors much larger than $2^{30}$. The use of 11264 buckets is partially constrained by the available shared memory on the GPU’s multiprocessors. As shared memory capacities increase on future generation GPUs, increasing the number of buckets will be advantageous for bucketMultiSelect.

The basic outline of bucketMultiSelect is the following: consider a vector of $n$ values from which $k$ order statistics are desired.

1. **Uniform Kernel Density Estimator**: through random sampling, identify approximate order statistics to define a step-function approximation of the distribution of the full data set.

2. **Create Buckets**: using the uniform kernel density estimator, define $B$ buckets as intervals from the minimum to maximum value in the full data set. The kernel density estimator ensures these buckets will contain roughly $n/B$ values.

3. **Assign Values to Buckets**: through a linear projection, assign the values in the data set to one of the $B$ buckets.

4. **Identify Active Buckets**: by counting the number of values assigned to each bucket, identify at most $k$ buckets containing the desired set of order statistics.

5. **Dimension Reduction**: reduce the problem by copying the values in the active buckets to an auxiliary vector and update the set of desired order statistics.

6. **sort&choose**: sort the vector of active values and retrieve the $k$ desired order statistics.

The kernel density estimator allows the creation of buckets which are assigned an approximately uniform number of values, namely $n/B$ values per bucket. This is the key step to providing a substantial dimension reduction: rather than sorting a vector of length $n$, bucketMultiSelect sorts a vector of length approximately $\frac{k}{B} n$. When $B = 11264$, $k = 101$, and $n = 2^{26}$, bucketMultiSelect sorts approximately 602,000 values rather than 67 million. Even with the initial overhead, this results in considerable performance gains when selecting multiple order statistics from huge data sets. bucketMultiSelect is capable of selecting thousands of order statistics faster than sort&choose; for $n = 2^{26}$ bucketMultiSelect selects the 101 percentile order statistics from uniformly distributed double precision values in approximately 27 ms, roughly $6.5 \times$ faster than sort&choose. When $n = 2^{28}$, the acceleration grows to more than $8 \times$; see Tab. VI.

<table>
<thead>
<tr>
<th>Type Distribution</th>
<th>Type Distribution</th>
<th>Type Distribution</th>
<th>Type Distribution</th>
<th>Type Distribution</th>
<th>Type Distribution</th>
<th>Type Distribution</th>
</tr>
</thead>
<tbody>
<tr>
<td>UINT</td>
<td>Float</td>
<td>Double</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uniform</td>
<td>Uniform</td>
<td>Uniform</td>
<td>Normal</td>
<td>Normal</td>
<td>Uniform</td>
<td>Normal</td>
</tr>
<tr>
<td>2^{21}</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>257</td>
</tr>
<tr>
<td>2^{22}</td>
<td>233</td>
<td>221</td>
<td>161</td>
<td>201</td>
<td>0</td>
<td>3371</td>
</tr>
<tr>
<td>2^{23}</td>
<td>1025</td>
<td>1012</td>
<td>1024</td>
<td>965</td>
<td>512</td>
<td>4578</td>
</tr>
<tr>
<td>2^{24}</td>
<td>1828</td>
<td>1676</td>
<td>1622</td>
<td>1597</td>
<td>1114</td>
<td>5559</td>
</tr>
<tr>
<td>2^{25}</td>
<td>2271</td>
<td>2162</td>
<td>2079</td>
<td>2088</td>
<td>1638</td>
<td>6108</td>
</tr>
<tr>
<td>2^{26}</td>
<td>2482</td>
<td>2349</td>
<td>2249</td>
<td>2256</td>
<td>1817</td>
<td>6484</td>
</tr>
<tr>
<td>2^{27}</td>
<td>2601</td>
<td>2490</td>
<td>2380</td>
<td>2395</td>
<td>1960</td>
<td>6698</td>
</tr>
<tr>
<td>2^{28}</td>
<td>2609</td>
<td>2484</td>
<td>2380</td>
<td>2388</td>
<td>1941</td>
<td>6877</td>
</tr>
</tbody>
</table>

Table I indicates the largest number$^1$ of order statistics for which bucketMultiSelect selects a set of order statistics faster than sort&choose. The values in the vector are of type unsigned integer (UINT), single precision (Float), or double precision (Double) and are drawn from a random distribution. The details of the experimental set-up are described in Sec. 3. In Tab. I, the sets of order statistics are uniformly spaced meaning $k$ order statistics provide the $100/(k - 1)$ percentiles, i.e. the set of desired order statistics is

$^1$A different set of tests will yield similar, but not identical, values for Tab. I.
\{s_1, s_2, \ldots, s_k, n\} \text{ where } s_i = \left\lfloor \frac{i}{k} \times n \right\rfloor \text{ for } i = 1, \ldots, k - 1. \text{ Therefore, 11 uniformly spaced order statistics provide the deciles, 101 order statistics provide the percentiles, 201 order statistics the 1/2-percentiles, 1001 order statistics provide the 1/10-percentiles, etc. Additional performance information is included in Sec. 3.}

1.2. General Purpose GPU Computing

Graphics processing units continue to grow in popularity as a low-cost, low-energy, highly effective platform for parallel computations. A GPU consists of a certain number of multiprocessors each with a large number of cores. For high performance computing applications, error correcting code is available on NVIDIA’s Tesla architecture making GPU computing a trustworthy alternative to CPU-only implementations. The NVIDIA Tesla K40 GPU has 15 multiprocessors with 192 cores each for a total of 2880 cores.

We implement our algorithms in CUDA-C, NVIDIA’s extension of C which allows for relatively straightforward management of the parallelization. The CUDA programming model [NVIDIA Corporation 2011] consists of writing kernels which provide a set of instructions to be executed on each core. The parallelization is broken into blocks with each block assigned a number of threads. The execution of a kernel is effortlessly sent to the GPU with control parameters determining the number of blocks and the number of threads per block. The programmer is free to select a range of combinations for the blocks and threads per block.

Each multiprocessor is assigned a block and all associated threads. The multiprocessor has a limited amount of constant and shared memory which is available to all threads in the block. The use of local memory, either shared or constant, can significantly increase the speed at which the kernel is executed. While the memory available to each multiprocessor is limited, a high performance GPU is also equipped with a reasonable amount of global memory. All blocks and all threads can access the GPU’s global memory although the global memory access is slower than accessing the shared or constant memory on each multiprocessor. CUDA-C is also equipped with a straightforward method for allocated shared memory with each block, again by passing a control parameter in the kernel call. The algorithms in this paper utilize the shared memory when possible to improve performance. The CUDA Programming Guide [NVIDIA Corporation 2011] contains a much more thorough discussion of the CUDA programming model including the use of the various types of memory and the execution of kernels.

The algorithm presented here has been tailored for its implementation on GPUs. However, the algorithm is applicable to any multi-core architecture by altering the GPU specific subroutines and tuning appropriate parameters. The GPU implementation was selected based on the growing popularity and wide availability of high performance GPUs.

2. AN ALGORITHM FOR SELECTING MULTIPLE ORDER STATISTICS: BUCKETMULTISELECT

In this section, we provide the details of the algorithm and specific choices for the implementation on GPUs. This algorithm draws on many of the strengths of the single order statistic selection algorithms described in Sec. 1. In Sec. 2.1, random sampling similar to bootstrapping and the technique used in randomizedSelect [Monroe et al. 2011] permits a rapid definition of buckets which will each contain a roughly uniform number of values from the full data set. The fast linear projection into buckets is borrowed from bucketSelect [Alabi et al. 2012]. Finally, the guarantees of sort&choose are applied to a reduced vector containing the candidates for the set of desired order statistics. The full algorithm bucketMultiSelect is defined in Sec. 2.5 with pseudo code given in Alg. 5. This pseudo code is written in terms of four subroutines which are detailed in Secs. 2.1–2.4. For the CUDA implementation, most kernels are executed with 1024 threads per block, and the number of blocks equal to the number of multi-processors on the GPU.
2.1. Creating the buckets

While `bucketSelect` is the fastest algorithm on non adversarial distributions, the algorithm struggles when faced with adversarial vectors [Alabi et al. 2012]. In `bucketSelect` the buckets are defined as equal width intervals from the minimum to maximum value in the vector. When the entries in the vector are far from uniformly distributed, the algorithm suffers greatly. Borrowing from the bootstrapping techniques and `randomizedSelect`, `bucketMultiSelect` first determines a set of \( P \) kernel density intervals (KD intervals) which form a piecewise defined uniform kernel density estimator for the values in the vector. Alternatively, the construction of the KD intervals can be interpreted as a first level of a sample sort [Dehne and Zaboli 2012; Leischner et al. 2010]. Each KD interval is then assigned an equal number of buckets thereby placing more buckets at the concentration of the density function for the values in the vector.

To define \( P \) KD intervals, we randomly sample 1024 entries from the vector. From these samples, we employ `sort&choose` to select \( P - 1 \) order statistics from the sample. These \( P - 1 \) order statistics are stored in a vector \( \text{pivots} \). The minimum and maximum values in the vector are added to the beginning and end of \( \text{pivots} \). These pivots now completely define \( P \) KD intervals \( [\text{pivots}[i-1], \text{pivots}[i]) \). The step function taking the constant value \( \left( \frac{P}{\text{pivots}[i] - \text{pivots}[i-1]} \right)^{-1} \) on the interval \( [\text{pivots}[i-1], \text{pivots}[i]) \) is a (modified) kernel density estimator for the entries in the vector: let \( p_i = \text{pivots}[i] \) and define

\[
\hat{f}(x) = \frac{1}{P} \sum_{i=1}^{P} \frac{1}{p_i - p_{i-1}} \chi_{[0,1)} \left( \frac{x - p_{i-1}}{p_i - p_{i-1}} \right) = \frac{1}{P} \sum_{i=1}^{P} \frac{1}{p_i - p_{i-1}} \chi_{[p_{i-1}, p_i)}(x),
\]

(1)

where \( \chi_I \) is the indicator function for the interval \( I \). The function \( \hat{f} \) defined by (1) is a rough approximation of the density function for the values in the vector. In our implementation, \( P = 16 \) and the 15 order statistics selected from the sample are \( \{22 + 70j\}^{14}_{j=0} \). The offset of 22 was selected from empirical observations to properly define the KD intervals. The construction of the KD intervals and slopes is described in Subroutine 1, Step 1.

---

**Subroutine 1: bucketMultiSelect: CreateBuckets**

**Input:** Vector, \( \text{vec} \), vector length \( n \), number of KD intervals \( P \), total number of buckets \( B \), minimum and maximum values in \( \text{vec} \) \( \text{min} \), \( \text{max} \)

**Output:** Endpoints of KD intervals \( \text{pivots} \), slopes for KD intervals \( \text{slopes} \)

1. Define endpoints of KD intervals as \( \text{pivots} \);
   - Generate 1024 uniformly distributed random numbers;
   - Use random numbers to sample 1024 elements from \( \text{vec} \);
   - Sort the samples, \( \text{sortedsamples} \);
   - Define \( P + 1 \) pivots;
     - for \( j = 1; j < P; j = j + 1 \) do
       - \( \text{pivots}[j] = \text{sortedsamples}[22 + 70*(j-1)] \)
     - end
   - \( \text{pivots}[0] = \text{min} \) and \( \text{pivots}[P] = \text{max} \);
2. Define slopes for each KD interval;
   - Number of buckets per KD interval: \( b = B/P \);
   - for \( i = 1; i < P; i = i + 1 \) do
     - \( \text{slopes}[i-1] = b/(\text{pivots}[i]-\text{pivots}[i-1]) \)
   - end

The uniform kernel density estimator serves as a proxy for the density of the values in the vector. We now define \( B \) buckets under the assumption\(^2\) that the values in the vector

\(^2\)For detail on the confidence given to this assumption, see Sec. 2.5.3.
are approximately uniformly distributed along each KD interval. Under this assumption, we assign an equal number of buckets \( b = B/P \) to each KD interval. For the follow-on projection step in Subroutine 2, the slope of the line from the point \((0, 0)\) to the point \((\text{pivots}[i] - \text{pivots}[i - 1], b)\) is computed in Subroutine 1, Step 2 and stored in the vector \( \text{slopes} \). These slopes allow a linear projection of the values in the vector into the buckets \( 0, \ldots, b - 1 \) as described in Sec. 2.2. The natural inclination to choose a very large number of buckets conflicts with available shared memory capacity and the need to identify those buckets containing the desired order statistics. In our implementation, \( B = 11264 \) and \( b = 704 \) as this was empirically identified as the most consistent choice of parameters for large vectors while permitting liberal use\(^3\) of shared memory.

### 2.2. Assigning Values to the Buckets

The first step in assigning a value to the buckets is finding the KD interval containing the value. This is accomplished through traversing a binary search tree as described in [Leischner et al. 2010] over the left endpoints of the KD intervals stored in shared memory as \( \text{pivots} \). Once the correct KD interval is identified we assign the values of the vector to the buckets through a linear projection. Define the linear functions

\[
L_i(x) = \frac{b}{p_i - p_{i-1}} (x - p_{i-1}) = \text{slopes}[i - 1] \times (x - \text{pivots}[i - 1]),
\]

where \( p_i = \text{pivots}[i] \) and \( b = B/P \) is the number of buckets per KD interval. For the values in each KD interval, \( x \in [p_{i-1}, p_i) \), the linear projection simply determines the largest integer in \( \{0, \ldots, b - 1\} \) that is no greater than \( L_i(x) \), i.e. \( P_i(x) = \lfloor L_i(x) \rfloor \). To determine the appropriate bucket for assignment, the cumulative number of buckets assigned to all preceding KD intervals is then added to the linear projection. Finally, we must keep track of the number of values assigned to each bucket and increment a counter on the bucket stored in \( \text{bucketCount} \). This process is detailed in Subroutine 2, Step 3.

---

**Subroutine 2: bucketMultiSelect: AssignBuckets (CUDA Kernel)**

**Input:** Vector, \( \text{vec} \), vector length \( n \), endpoints of KD intervals \( \text{pivots} \), number of KD intervals \( P \), slopes for KD intervals \( \text{slopes} \), number of Buckets \( B \), control parameter \( \text{offset} \)

**Output:** Assignments to buckets \( \text{buckets} \), array of bucket counters by block \( \text{CounterArray} \)

1. Get thread index \( \text{threadIndex} \);
2. Copy \( \text{pivots} \) (as binary search tree) and \( \text{slopes} \) to shared memory and initialize \( \text{bucketCount} \) in shared memory;
3. Assign values in \( \text{vec} \) to buckets and increment counter;
   
   ```
   for i = \text{threadIndex}; i < n; i = i + \text{offset} do
     \text{Binary search for KD interval } p \in \{1, \ldots, P\};
     \text{Number of buckets for all previous KD interval } \text{PreCount} = (p - 1)B/P;
     \text{buckets}[i] = \lfloor \text{slopes}[p - 1] \times (\text{vec}[i] - \text{pivots}[p - 1]) \rfloor + \text{PreCount};
     \text{if buckets}[i] = B then /* Close right endpoint of final bucket */
       \text{buckets}[i] = B - 1;
     \text{end}
     \text{bucketCount[buckets}[i]] = \text{bucketCount[buckets}[i]] + 1; /* using atomicInc */
   \text{end}
   \text{Copy bucketCount to global memory array } \text{CounterArray};
   ```

---

In the CUDA implementation, several factors accelerate the bucket assignment process. First, the assignment of each value in the vector is clearly independent of all other vector

---

\(^3\)As noted in Sec. 1.1, when shared memory capacities increase on future GPUs, increasing the number of buckets will enhance the performance of \( \text{bucketMultiSelect} \).
elements. Therefore, the linear projection is completely parallelized. When incrementing the counter for each bucket, conflicts are possible when two threads simultaneously assign a value to the same bucket. To avoid such conflicts, we utilize the built-in atomic function `atomicInc` which places simultaneous requests in a queue for serial execution. To further avoid conflicts and speed up memory access, the values in `pivots`, `slopes`, and `bucketCount` are all stored in shared memory. Information stored in shared memory is accessible only to the block utilizing the multiprocessor; therefore simultaneous updates from different blocks do not present a conflict or add to an atomic queue. To minimize the number of communications between shared and global memory, each thread is tasked with assigning buckets to multiple values. By traversing the vector via jumps of size $offset = numBlocks \times threadsPerBlock$, we exploit coalesced memory access for each warp of 32 threads.

When copying the `bucketCount` vector from shared memory to global memory, each block is assigned a column in a $B \times numBlocks$ array `CounterArray`. Since the thread index is computed in the same manner for every kernel, each block is assigned the same set of threads in future kernels using an identical choice of number of blocks and threads per block. By storing bucket counts in `CounterArray`, future kernels, including the parallel reduction in Subroutine 4, are provided specific information about the values and bucket assignments for the elements considered by the block. This significantly accelerates Subroutine 4.

### 2.3. Identifying the Active Buckets

Following Subroutine 2, we have a vector, `buckets`, of $n$ unsigned integers containing the bucket assignments and a $B \times numBlocks$ array `CounterArray`, of unsigned integers describing the number of values assigned to each bucket by each of the blocks. In this section, we detail the task of identifying the buckets which contain the desired order statistics. To do so, we first form a cumulative sum of the bucket counters in `CounterArray` and copy the final column to the host memory. The host copy of the total counts for each bucket is denoted $bucketCount_h$ with the subscript identifying its location in host memory. Performing the identification on the device (GPU) is also possible, although empirical investigations showed no advantage. (Potential problem specific optimizations are discussed in Sec. 2.5.2.)

#### Subroutine 3: `bucketMultiSelect::identifyActiveBuckets`

**Input:** Array of bucket counters `CounterArray`, number of buckets $B$, desired order statistics $DesiredOrderStats$, number of order statistics $k$

**Output:** Active buckets `ActiveBuckets`, cumulative counts up to active buckets `ActiveBucketPreCounts`

1. Cumulative sum: `CounterArray = cumulatesum(CounterArray); /* CUDA kernel */`
2. Copy last column of `CounterArray` to host memory;
   ```
   for(j = 0; j < B; j = j + 1)
   |
   | bucketCount_h[j] = CounterArray[j, numBlocks - 1];
   ```
3. Identify active buckets containing a desired order statistic;
   ```
   sum = bucketCount_h[0];
   active = 0;
   for(i = 0; i < k; i = i + 1)
   |
   | s = DesiredOrderStats[i];
   | while sum $< s$ $\&$ active $< B$
   | |
   | active = active + 1;
   | |
   | sum = sum + bucketCount_h[active];
   |
   ActiveBuckets[i] = active;
   ActiveBucketPreCounts[i] = sum - bucketCount_h[active];
   ```

ACM Journal Name, Vol. 0, No. 0, Article 00, Publication date: 2013.
We define an active bucket as a bucket to which at least one desired order statistic has been assigned. With the set of desired order statistics in increasing order, identifying the active buckets is a simple traversal of the list of bucket counters. By progressively summing the number of values in each bucket, as soon as the cumulative number of values assigned to the buckets is larger than the desired order statistic, this bucket must contain the desired order statistic. Therefore, a simple loop on the host rapidly identifies the active buckets. Simultaneously, the cumulative counts of values assigned to all preceding buckets are stored for each active bucket. This results in a list of \( k \) unsigned integers indexing the active buckets, \text{ActiveBuckets}, and the associated cumulative counts of preceding buckets, \text{ActiveBucketPreCounts}.

### 2.4. Reducing the Problem Size

Subroutines 2 and 3 have now produced an assignment of all values in the vector to \( B \) buckets and identified the active bucket for each of the desired order statistics. We now perform the essential task of reducing the vector to only the active buckets. Since the number of values assigned to each bucket is approximately \( n/B \) and we are trying to identify \( k \) order statistics, this results in a reduction from \( n \) possible elements to roughly \( k \times n \) elements. In a serial implementation, one would simply traverse the vector of bucket assignments and copy each entry of the vector assigned to an active bucket into a new auxiliary vector. Subroutine 4, Step 4 executes this task in parallel on the GPU.

**Subroutine 4: bucketMultiSelect: Reduce**

Input: Vector \( \text{vec} \), bucket assignments \( \text{buckets} \), array of bucket counters by block \( \text{CounterArray} \),
active buckets \( \text{ActiveBuckets} \), control parameter \( \text{offset} \), cumulative counts up to active buckets \( \text{ActiveBucketPreCounts} \), desired order statistics \( \text{DesiredOrderStats} \)

Output: Reduced vector \( \text{ReducedVector} \), updated order statistics \( \text{UpdatedOrderStats} \)

1. Determine a unique set of active buckets;
   \( \text{UniqueActive} = \text{unique elements in } \text{ActiveBuckets} \);
2. Update cumulative counts to each unique bucket;
3. Update desired order statistics for the reduced vector;
   \( \text{UpdatedOrderStats} = \text{adjusted DesiredOrderStats} \) for reduced vector;
4. Copy values in \( \text{vec} \) assigned to buckets in \( \text{UniqueActive} \) to the auxiliary vector \( \text{ReducedVector} \);
   Get thread index \text{threadIndex}:
   Load \( \text{UniqueActive} \) (as binary search tree) to shared memory;
   Compute \( \text{blockOffset} = B \times \text{blockIndex} \) for \( \text{CounterArray} \);
   for \( i = \text{threadIndex}; i < n; i = i + \text{offset} \) do
     if \( \text{buckets}[i] \in \text{UniqueActive} \) then
       /* Binary Search Tree */
       \( \text{ReducedVector} [\text{atomicDec(CounterArray}\text{buckets}[i] + \text{blockOffset}) - 1] = \text{vec}[i]; \)
     end
   end

First, redundant active buckets, i.e., buckets containing multiple desired order statistics, have all repeats removed and a vector of at most \( k \) unique active buckets is identified. The cumulative counts preceding the unique active buckets are updated. The desired order statistics are then recomputed to match the total number of elements in all unique active buckets. Finally, by returning the appropriate cumulative count in \( \text{CounterArray} \) to the shared memory of each block, the values assigned to the unique buckets are identified through traversing a binary search tree as in [Leischner et al. 2010] and copied to an auxiliary vector, \( \text{ReducedVector} \). By scheduling the kernels with identical block and thread parameters, the cumulative counts in \( \text{CounterArray} \) serve as predefined memory allocations for each unique bucket and each block. While coalesced memory transfers are unlikely, an
atomic decrement eliminates bank conflicts in writing to ReducedVector. This produces a rapid search and copy of the values in the vector which remain candidates for the desired order statistics. (Potential problem specific optimizations are discussed in Sec. 2.5.2.)

2.5. Main Algorithm: bucketMultiSelect

2.5.1. Implementation of bucketMultiSelect. This section details the use of Subroutines 1–4 to substantially reduce the time required to select multiple order statistics from a large vector. In Sec. 3 empirical performance measures show that bucketMultiSelect offers considerable acceleration over sorting the entire list; see Tab. VI. In fact, for large vectors of unsigned integers, floats or doubles, bucketMultiSelect is able to select thousands of order statistics faster than sorting the vector; see Tab. I. For a typical application, the 101 percentile order statistics provide a highly accurate representation of the full data set; see Fig. 5.

**ALGORITHM 5: bucketMultiSelect**

**Input:** Vector vec, length of vector n, desired order statistics DesiredOrderStats, number of order statistics k, number of buckets B, number of KD intervals P

**Output:** Values of order statistics OrderStatValues

1 **Initialization:**
   - Set numBlocks and threadsPerBlock;
   - offset = numBlocks * threadsPerBlock;
   - [min, max] = thrust::minmax(vec);
   - if min = max then /* vec contains only one unique value. */
     | return OrderStatValues[1 : k] = min;
   end
2 **Subroutine 1:**
   - [pivots, slopes] = CreateBuckets(vec, n, P, B, min, max);
3 **Subroutine 2:**
   - [buckets, CounterArray] = AssignBuckets(vec, n, pivots, P, slopes, B, offset);
4 **Subroutine 3:**
   - [ActiveBuckets, ActiveBucketPreCounts] = IdentifyActiveBuckets(CounterArray, B, DesiredOrderStats, k);
5 **Subroutine 4:**
   - [ReducedVector, UpdatedOrderStats] = Reduce(vec, buckets, CounterArray, ..., ...., ActiveBuckets, ActiveBucketPreCounts, DesiredOrderStats);
6 **sort&choose:**
   - ReducedVector = gpusort(ReducedVector)
   - Copy values of desired order statistics to output vector
   - return OrderStatValues = ReducedVector[UpdatedOrderStats]

To invoke the subroutines of the preceding sections, bucketMultiSelect begins with an initialization. The number of blocks and threads per block are set for the execution of the kernels. The CUDA implementation is designed to use 1024 threads per block with the number of blocks set as the number of multiprocessors on the GPU. With these parameters set, Subroutine 2 is tasked with assigning several values to their appropriate buckets. To exploit coalesced memory transfers, each thread moves through the vector in steps of size offset = numBlocks * threadsPerBlock. To aid in defining the buckets via the kernel density estimator in Subroutine 1, the minimum and maximum values\(^4\) in the vector are obtained using a function from Thrust [Hoberock and Bell 2010].

\(^4\)An alternative version that does not require the minimum and maximum values was also implemented. This algorithm’s slightly reduced overhead results in faster identification of order statistics away from the extremes, but is noticeably slower when extreme order statistics near 1 or n are requested.
After completing the initialization, Subroutine 1 defines buckets according to a uniform kernel density estimator obtained via random sampling. Subroutine 2 then assigns the values in the vector to an appropriate bucket while fully utilizing shared memory and the independence of the computational, linear assignment for a full parallelization. The buckets which contain candidates for the desired order statistics are then identified with Subroutine 3. Subroutine 4 copies the remaining candidates for the order statistics to an auxiliary vector and updates the order statistics to coincide with the new, reduced vector. Finally, bucketMultiSelect employs the stability of sort&choose on the reduced vector and writes the values of the desired order statistics to the output vector via a parallelized copy.

2.5.2. Potential Optimization Strategies. The implementation of bucketMultiSelect balances performance across the extremely large problem space where GPU architecture, data type, vector length, vector distribution, order statistic distribution, and the number of order statistics can significantly impact performance. Our implementation is founded on the desire to validate the algorithmic idea of bucketMultiSelect rather than to create an optimized piece of software. Several implementation specifics could be tuned to improve performance at the expense of obfuscating the impact of algorithmic ideas in the empirical testing in Sec. 3.

The only aspect of our implementation which adapts to a characteristic of a problem instance is the sorting algorithm employed in the final sort&choose phase (Alg. 5, Step. 6). Since the naive method of sort&choose should utilize the fastest available sorting routine, the implementation of sort&choose, both as an independent algorithm and as a subroutine of bucketMultiSelect, employs different libraries for 32 bit and 64 bit data types. For vectors of type float and unsigned integer, sort&choose utilizes the highly optimized device radix sort, 

\texttt{cub::DeviceRadixSort::SortKeys}, from the CUB library [Merrill 2013]. For vectors of type double, sort&choose utilizes the similarly optimized merge sort, 

\texttt{mgpu::MergeSortKeys}, from the ModernGPU library [Baxter 2013]. Finally, the choice of $B = 11264$ total buckets is dependent on the use of an NVIDIA Tesla K40 with 96GB of shared memory per multi-processor.

Table II. Mean bucketMultiSelect (Alg. 5) Subroutine timings (ms) selecting deciles, percentiles, and 1/10-percentiles, K40, 25 tests. Total Number of Buckets: $B = 11264$.

<table>
<thead>
<tr>
<th>Uniform Floats</th>
<th>SubRoutine</th>
<th>length</th>
<th>#OS</th>
<th>Initialization</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>s&amp;c</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>$2^{26}$</td>
<td>11</td>
<td>4.20</td>
<td>0.88</td>
<td>8.08</td>
<td>0.14</td>
<td>5.72</td>
<td>0.96</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$2^{26}$</td>
<td>101</td>
<td>4.29</td>
<td>0.89</td>
<td>8.08</td>
<td>0.15</td>
<td>8.89</td>
<td>1.99</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$2^{26}$</td>
<td>1001</td>
<td>4.34</td>
<td>0.78</td>
<td>8.10</td>
<td>0.18</td>
<td>16.44</td>
<td>6.59</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$2^{8}$</td>
<td>11</td>
<td>12.19</td>
<td>0.89</td>
<td>32.32</td>
<td>0.14</td>
<td>22.95</td>
<td>2.00</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$2^{8}$</td>
<td>101</td>
<td>12.20</td>
<td>0.88</td>
<td>32.31</td>
<td>0.15</td>
<td>35.01</td>
<td>3.96</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$2^{8}$</td>
<td>1001</td>
<td>12.32</td>
<td>0.78</td>
<td>32.36</td>
<td>0.17</td>
<td>65.65</td>
<td>22.08</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Uniform Doubles</th>
<th>SubRoutine</th>
<th>length</th>
<th>#OS</th>
<th>Initialization</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>s&amp;c</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>$2^{26}$</td>
<td>11</td>
<td>5.25</td>
<td>1.36</td>
<td>8.31</td>
<td>0.15</td>
<td>5.89</td>
<td>1.77</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$2^{26}$</td>
<td>101</td>
<td>5.08</td>
<td>1.34</td>
<td>8.30</td>
<td>0.16</td>
<td>8.94</td>
<td>3.07</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$2^{26}$</td>
<td>1001</td>
<td>5.24</td>
<td>1.47</td>
<td>8.33</td>
<td>0.18</td>
<td>16.69</td>
<td>13.36</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$2^{8}$</td>
<td>11</td>
<td>15.65</td>
<td>1.36</td>
<td>33.21</td>
<td>0.16</td>
<td>23.00</td>
<td>2.84</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$2^{8}$</td>
<td>101</td>
<td>15.44</td>
<td>1.34</td>
<td>33.19</td>
<td>0.15</td>
<td>35.20</td>
<td>6.93</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$2^{8}$</td>
<td>1001</td>
<td>15.64</td>
<td>1.49</td>
<td>33.27</td>
<td>0.18</td>
<td>67.20</td>
<td>54.45</td>
</tr>
</tbody>
</table>

In order to understand the potential effect of improving the implementation, Tab. II provides insight into the computational cost of each phase of Alg. 5 for various data types, vector lengths, and numbers of uniformly spaced order statistics. The timing of the initialization phase includes the allocation of all predefined memory requirements. Similarly,
Table III. Mean bucketMultiSelect (Alg. 5) Subroutine timings (ms) selecting deciles, percentiles, and 1/10-quantiles, K40, 25 tests. Total Number of Buckets: $B = 8192$.

<table>
<thead>
<tr>
<th>Uniform Floats</th>
<th>SubRoutine</th>
<th>length</th>
<th>#OS</th>
<th>Initialization</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>s&amp;c</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>$2^{26}$</td>
<td>11</td>
<td>4.33</td>
<td>0.90</td>
<td>8.08</td>
<td>0.13</td>
<td>5.73</td>
<td>0.90</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$2^{26}$</td>
<td>101</td>
<td>4.25</td>
<td>0.90</td>
<td>8.08</td>
<td>0.14</td>
<td>9.11</td>
<td>2.15</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$2^{26}$</td>
<td>1001</td>
<td>4.29</td>
<td>0.77</td>
<td>8.08</td>
<td>0.16</td>
<td>18.23</td>
<td>8.49</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$2^{28}$</td>
<td>11</td>
<td>12.24</td>
<td>0.90</td>
<td>32.22</td>
<td>0.12</td>
<td>22.98</td>
<td>2.10</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$2^{28}$</td>
<td>101</td>
<td>12.22</td>
<td>0.90</td>
<td>32.22</td>
<td>0.14</td>
<td>35.84</td>
<td>4.71</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$2^{28}$</td>
<td>1001</td>
<td>12.32</td>
<td>0.79</td>
<td>32.36</td>
<td>0.16</td>
<td>73.23</td>
<td>29.76</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Uniform Doubles</th>
<th>SubRoutine</th>
<th>length</th>
<th>#OS</th>
<th>Initialization</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>s&amp;c</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>$2^{26}$</td>
<td>11</td>
<td>5.04</td>
<td>1.34</td>
<td>8.31</td>
<td>0.12</td>
<td>5.85</td>
<td>1.96</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$2^{26}$</td>
<td>101</td>
<td>5.07</td>
<td>1.33</td>
<td>8.31</td>
<td>0.14</td>
<td>9.16</td>
<td>3.38</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$2^{26}$</td>
<td>1001</td>
<td>5.26</td>
<td>1.52</td>
<td>8.31</td>
<td>0.16</td>
<td>18.15</td>
<td>18.40</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$2^{28}$</td>
<td>11</td>
<td>15.45</td>
<td>1.34</td>
<td>33.19</td>
<td>0.12</td>
<td>22.99</td>
<td>2.94</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$2^{28}$</td>
<td>101</td>
<td>15.44</td>
<td>1.33</td>
<td>33.19</td>
<td>0.14</td>
<td>36.07</td>
<td>8.61</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$2^{28}$</td>
<td>1001</td>
<td>15.53</td>
<td>1.51</td>
<td>33.18</td>
<td>0.16</td>
<td>73.95</td>
<td>78.21</td>
</tr>
</tbody>
</table>

the sort&choose phase timings includes the process of freeing any memory that was not released in a prior subroutine. The effect of increasing the number of buckets $B$ can be seen by comparing the use of $B = 8192$ buckets in Tab. III with the use of $B = 11264$ buckets in Tab. II. In both tables, note the limited impact of data type on Subroutines 1–4. In the remainder of this subsection, we describe some potential ways to accelerate an implementation based on prior knowledge of the problem.

**Initialization.** The primary cost in the initialization phase is determining the minimum and maximum value in the vector. In our implementation, this is performed using thrust::minmax which returns both values in a single pass. For very large vectors of floats and integers, $n \gtrsim 2^{28}$, implementing this task by using both a minimum and maximum reduction via the CUB library improves performance. Alternatively, one might adapt a future optimized library’s reduction function to obtain both the minimum and maximum in a single pass.

**Subroutine 1.** The current creation of the buckets has a relatively small impact on the overall cost of the implementation. However, when dealing with adversarial data it may be critical to take a larger sample when selecting the pivots. Doing so could ultimately change the impact of this subroutine.

**Subroutine 2.** In bucketMultiSelect the task of assigning the elements to a bucket consists of $\lceil \log_2(P) \rceil$ comparisons in a binary search tree followed by a linear projection. The computational bucket assignment calls for the use of a large number of buckets, $B$. One interpretation of this task is a comparison based assignment to the $P$ buckets defined by the KD intervals, followed by a refinement of each big bucket to $b = B/P$. This is akin to combining the sample sort ideas of using a small number of partitions as in [Leischner et al. 2010] and a large number of partitions as in [Dehne and Zaboli 2012]. The distinguishing characteristic in bucketMultiSelect is the computational assignment process rather than a comparison based assignment.

When selecting a single order statistic with bucketSelect [Alabi et al. 2012], the algorithm iteratively applies the computational bucket assignment to the lone active bucket; iterative refinement of the partitions is also used in the GPU sample sort [Leischner et al. 2010]. An iterative refinement of the buckets in bucketMultiSelect is also possible and potentially advantageous. As seen in Tabs. II–III, the effect of increasing the number of buckets ultimately accelerates the subsequent tasks in Subroutine 4 and the sort&choose phase. A similar, potentially greater effect can be achieved through iterative refinement at
the cost of increasing the time required to complete the assignment process. Moreover, this requires redefining the linear projections and tailoring them to each of the active buckets. While this is inconsequential for a very small number of active buckets, simply sorting the active elements provides a relatively stable implementation which fosters a clearer insight into the algorithmic idea of reduction.

Subroutine 3. Like Subroutine 1, the identification of the active buckets is a small fraction of the total cost of the algorithm. The current implementation uses a heterogeneous CPU - GPU scheme to identify the active buckets. As stated in Sec. 2.3, it is possible to accomplish this task entirely on the GPU using a prefix sum and scan. Our implementation serially identifies the active buckets in order to balance the cost of the task for very large numbers of order statistics. For many order statistics, the serial CPU implementation is advantageous while for small numbers of order statistics a GPU only implementation could potentially improve this low cost subroutine.

Subroutine 4. The reduction step is the area most ripe for optimization and is the principal cost of the algorithm as the number of order statistics increases. For a moderate number of order statistics, \( k \lesssim 200 \), the use of the CounterArray and atomicDec to copy the elements to an auxiliary vector is rather effective. However, the implementation has two weaknesses: the use of an atomic function and the lack of coalesced memory transfers. As the number of order statistics increases, these weaknesses negatively impact the performance of bucketMultiSelect. When selecting thousands of order statistics, one should alter the implementation to enforce coalesced memory access. For example, simply replacing the copy to global memory in Alg. 4, Step 4, with the creation of a flag vector for use with the optimized partition function in CUB can improve this subroutine when selecting large numbers of order statistics.

sort&choose. Our implementation of bucketMultiSelect chooses the sorting routine used in the sort&choose phase of the algorithm based on the data type. For floats and unsigned integers, the device wide radix sort in the CUB library is employed, while the ModernGPU merge sort is used for vectors of type double. The creation of the reduced vector containing the active elements currently uses predefined memory locations so that the reduced vector is partially sorted into the active buckets. A potential improvement on the current implementation is to exploit this information to sort each active bucket individually. In terms of implementation, this is most desirable when the expected number of active buckets is relatively small or the expected number of elements in each active bucket is relatively small. The current implementation choice is based on the desire for a single implementation which is effective over a large class of problems. If the expected length of the reduced vector is small enough, a CPU based quickMultiSelect algorithm could also be beneficial (See Sec. 3.2).

2.5.3. Analysis of bucketMultiSelect. Since bucketMultiSelect is not an iterative algorithm, the analysis is rather straightforward. Ignoring the parallelization, we can simply count the number of operations. Identification of the minimum and maximum in the Initialization, the linear projection to buckets in Subroutine 2, and the identification of elements in the active buckets in Subroutine 4 are each \( O(n) \). Subroutines 1 and 3 require far fewer operations defined by the number of pivots, \( P \), buckets, \( B \), or desired order statistics, \( k \). The final sort&choose phase relies on the choice of sorting algorithm; if the sorting algorithm is \( O(S(n)) \) then bucketMultiSelect is \( O(\max\{n,S(n)\}) \). If the implementation exhibits an average parallel efficiency of \( \alpha \) over \( q \) cores, bucketMultiSelect requires \( \frac{O(\max\{n,S(n)\})}{\alpha q} \) operations.

Sorting routines typically require an auxiliary vector and alter the input vector; if the input vector must be preserved, sort&choose must first store a copy of the input vec-
Table IV. Worst case and Expected Memory requirements for bucketMultiSelect.

<table>
<thead>
<tr>
<th>Type</th>
<th>Unit/Float</th>
<th>Double</th>
</tr>
</thead>
<tbody>
<tr>
<td>bucketMultiSelect</td>
<td>4n</td>
<td>3.5n</td>
</tr>
<tr>
<td></td>
<td>( (2 + 2\frac{k}{B})n )</td>
<td>( (1.5 + 2\frac{k}{B})n )</td>
</tr>
</tbody>
</table>

The sorting algorithms selected in our implementation require a memory allocation of 2n altering the input vector or 3n to preserve the original vector. On the other hand, bucketMultiSelect preserves the original input vector. The length n auxiliary vector required by bucketMultiSelect is a vector of unsigned integers for the bucket assignment. bucketMultiSelect also has the additional memory required by the sorting algorithm in the final phase. In the worst case Subroutine 4 returns of vector of length \( n \). Thus, the worst case memory requirement for a vector of type \( T \) is \( 3n \cdot \text{sizeof}(T) + n \cdot \text{sizeof}(\text{uint}) \). For vectors containing uniformly distributed data, bucketMultiSelect sorts a vector of expected length \( \frac{1}{2}n \). Thus the 3n worst case term is replaced by an expected memory requirement for bucketMultiSelect of \( \left(1 + 2\frac{k}{B}\right)n \cdot \text{sizeof}(T) \). Tab. IV lists the memory requirements for bucketMultiSelect, where we note \( \text{sizeof}(\text{uint}) = \text{sizeof}(\text{double})/2 \). For a typical application selecting \( k = 101 \) order statistics from a vector of type double and using \( B = 11264 \) buckets, bucketMultiSelect requires a memory allotment of approximately 1.52n while sort&choose requires an allocation of 2n or 3n to preserve the original vector.

Finally, the sample size in Subroutine 1 to form the KD intervals is 1024. The uniform kernel density estimator is designed to identify 16 KD intervals which support 1/16 of the entries in the full data set. Treating the endpoints of the KD intervals as order statistics defining population proportions, we employ sampling theory for determining sample sizes required for estimating a proportion. From [Thompson 2002, Sec. 5.4], the sample size of 1024 provides a 95% confidence level the proportions supported on each KD interval is within 0.0353 of the intended proportion of 0.0625. Increasing the sample size improves the accuracy of the KD intervals in exchange for an increase in computation time.

3. PERFORMANCE EVALUATION

The following empirical data was obtained through the testing suite included in the software distribution, GGMS: Grinnell GPU Multi-Selection [Blanchard et al. 2013]. The testing was performed on machines with two Intel Xeon E5-2637 CPUs @ 3.50GHz and four NVIDIA Tesla K40 GPU. The NVIDIA Tesla K40 has 15 multiprocessors each with 192 cores for a total of 2880 CUDA cores. All findings reported in this subsection utilize a single GPU and are a result of at least 25 independent tests for each problem instance. The data points in Figs. 1–4 and the mean values in Tabs.II–VI are computed as a modified mean which excludes the minimum and maximum values to mitigate against potential impact of outliers and present typical algorithm behavior.

3.1. Random Data

A problem instance in this subsection consists of generating a random vector of length \( n \) and a set of \( k \) desired order statistics. The random vectors are populated with entries from one of four random vector distributions: uniform, normal, half normal, and Cauchy. A vector from the uniform vector distribution is populated with entries drawn from the uniform distribution on the interval \((0,1)\), namely \( U(0,1) \). The normal vector distribution populates a vector with elements from the standard normal distribution, \( \mathcal{N}(0,1) \), with the normal distribution values generated using the Box-Muller transform. The half normal vector distribution is created by taking the absolute value of entries drawn from the normal distribution.

\(^5\)Very rare anomalies were observed for both bucketMultiSelect and sort&choose in the testing process. The problem instances with observed anomalies was not consistent across testing runs.

ACM Journal Name, Vol. 0, No. 0, Article 00, Publication date: 2013.
Cauchy distribution is derived by applying the transform $C(x) = \tan\left(\pi(x - 1/2)\right)$ to the entries of a vector drawn from the uniform vector distribution. The cuRand random number generator is used to generate each problem instance.

The distribution of the $k$ desired order statistics from the possible set of order statistics, $\{1, \ldots, n\}$, has an important impact on the speed of bucketMultiSelect. When the set of $k$ desired order statistics falls into $k$ distinct buckets, the reduction step in Subroutine 4 is minimized. Alternatively, when the desired order statistics fall into $\ell < k$ unique buckets, a smaller number of elements are copied to the auxiliary vector and sorted. Intuitively, this implies that uniformly spacing the set of desired order statistics throughout the set $\{1, \ldots, n\}$ will be the most challenging distribution of order statistics for bucketMultiSelect. This is also verified empirically as shown in Fig. 1. In Fig. 1, the vector from which the order statistics are selected is populated with $2^{26}$ floats (a) or doubles (b) from the uniform vector distribution $U(0,1)$.

Alternative distributions for the desired order statistics are uniform random, normal random, clustered, and sectioned. For uniform random order statistics, a set of $k$ integers is selected from the discrete uniform distribution on $\{1, 2, \ldots, n\}$. Similarly, for normal random order statistics, $k$ random values are drawn from $\mathcal{N}(n/2, n/14)$ and truncated to an integer. The clustered distribution selects groups of 9 order statistics centered at $\lfloor k/9 \rfloor$ uniformly random elements from $\{1, \ldots, n\}$. Finally, the sectioned distribution requests a set of $k$ consecutive order statistics with an initial location selected uniformly at random. We see from Fig. 1 that the algorithm behaves as expected with the sectioned and clustered distributions being selected substantially faster than the others as these distribution are highly likely to fall into a much smaller number of unique buckets. When $k \lesssim 1000$, the uniform, uniform random, and normal random order statistic distributions are likely to require $k$ unique buckets and therefore yield similar timings for bucketMultiSelect. The apparent jumps in the curves occur when the number of active buckets passes a power of 2 thereby requiring an additional level in the binary search tree of Subroutine 4. Since sort&choose finds all $n$ order statistics, the distribution of the desired order statistics has no impact on its performance.

We focus on the most challenging distribution for the desired order statistics, namely $k$ uniformly spaced order statistics from 1 to $n$. In Fig. 2, we observe the growth in time to recover $k$ uniformly spaced order statistics in a vector of $n = 2^{26}$ floats or doubles drawn from the uniform (a) or normal (b) vector distributions. The computational linear assignment
Fig. 2. For a vector of length $n = 2^{26}$, time growth with the number of uniformly spaced order statistics for various vector distributions: (a) uniform, (b) normal, (c) half normal, (d) Cauchy.

In Table VI, we see that for vectors of length $2^{26}$ populated with entries from the uniform or normal vector distributions, `bucketMultiSelect` selects the 101 percentile order statistics in less than $95ms$ for floats or doubles and is more than $8 \times$ faster than sorting a vector a doubles. The time required to sort the vector in `sort&choose` grows exponentially with the vector length. In Fig. 3(a), we see that `bucketMultiSelect` has a sub exponential growth in time required to find 101 uniformly spaced order statistics in a vector from the uniform distribution. For all three vector types (uint, float, double), the time advantage of `bucketMultiSelect` over `sort&choose` increases as the length of the vector increases. Figure 3(b) shows nearly identical results for vectors from the normal distribution. It is clear from Fig. 3(a)-(b) and from Table VI that `bucketMultiSelect` is particularly well-suited for selecting multiple order statistics from vectors of type double; `bucketMultiSelect` is...
nearly as efficient on doubles as on floats. Figure 3 (c) and (d) depict the performance gain of `bucketMultiSelect` over `sort&choose` for vectors from the half normal and Cauchy vector distributions.

### 3.2. Order Statistic Selection: CPU vs. GPU

The multiple order statistic selection problem is well studied for CPU implementations, e.g. [Alsuwaiyel 2001; Cardinal and Fiorini 2013; Kaligosi et al. 2005]. In this subsection, we compare a typical `quickMultiSelect` CPU implementation based on [Alsuwaiyel 2001] with the GPU implementations of `sort&choose` and `bucketMultiSelect`. Figure 4 depicts the performance of these three algorithms on vectors of length $2^{10} \leq n \leq 2^{24}$ and provides an indication of the problem sizes and data types for which each algorithmic paradigm is best suited. For 32 bit data types (floats, uints) we see that for problems with length $n \lesssim 2^{12}$, `quickMultiSelect` on the CPU should be used. For moderate sized problems with $2^{12} \lesssim n \lesssim 2^{22}$, the GPU based `sort&choose` is the fastest algorithmic paradigm. Finally, for large problems with $n \gtrsim 2^{22}$, `bucketMultiSelect` finds the 101 percentile order statistics in the least time. For vectors of type double, `sort&choose` is best suited for problems in the range $2^{13} \lesssim n \lesssim 2^{21}$ while `quickMultiSelect` should be used for smaller problems and `bucketMultiSelect` for larger problems.
Figure 4 also provides some insight into the overhead of the implementations for sort&choose and bucketMultiSelect. We see that the time required for the GPU algorithms is essentially a constant overhead cost up to some critical length. The larger overhead costs of bucketMultiSelect could potentially be reduced through further software optimization as discussed in Sec. 2.5.2. The anticipated exponential growth in run time for a CPU based quickMultiSelect is evident in Fig. 4 while Fig. 3 and Tab. VI provide insight into the performance advantage of bucketMultiSelect over sort&choose for larger problems.

3.3. Adversarial and Real Data

The uniform kernel density estimator in Subroutine 1 aids in defining buckets which will have nearly uniform assignment counts. For most vector distribution, the KD intervals are highly effective. However, very heavy tailed distributions can minimize the effect of the Kernel Density Estimator. For example, the performance of bucketMultiSelect on a vector populated with entries from the Cauchy distribution is degraded when compared to the uniform, normal, and half normal distribution. Tables I and VI with Figs. 2, 3 show this somewhat degraded performance for the Cauchy distribution. For the uniform, normal, and
Table V. Selecting multiple order statistics from power readings data [Kolter and Johnson 2011]. Mean timings (ms) from 25 tests.

<table>
<thead>
<tr>
<th>Data Set</th>
<th>#entries</th>
<th>#unique</th>
<th>#OS</th>
<th>bMS</th>
<th>s&amp;c</th>
</tr>
</thead>
<tbody>
<tr>
<td>houses</td>
<td>42,661,878</td>
<td>367,431</td>
<td>11</td>
<td>21.486</td>
<td>34.583</td>
</tr>
<tr>
<td></td>
<td>101</td>
<td></td>
<td></td>
<td>23.860</td>
<td>34.741</td>
</tr>
<tr>
<td></td>
<td>1001</td>
<td></td>
<td></td>
<td>31.265</td>
<td>35.066</td>
</tr>
<tr>
<td>houses_small</td>
<td>42,372,537</td>
<td>363,393</td>
<td>11</td>
<td>17.282</td>
<td>34.728</td>
</tr>
<tr>
<td></td>
<td>101</td>
<td></td>
<td></td>
<td>20.630</td>
<td>34.826</td>
</tr>
<tr>
<td></td>
<td>1001</td>
<td></td>
<td></td>
<td>27.237</td>
<td>34.690</td>
</tr>
</tbody>
</table>

Half normal distributions, the similar performance of bucketMultiSelect demonstrates the effective definition of buckets from the KD intervals in Subroutine 1.

However, bucketMultiSelect is still susceptible to adversarial distributions. In particular, when the values in the vector are from a discrete distribution with a small number of unique values, the bucketing process will have long queues for the atomic counting. Moreover, when the number of unique values in the vector is roughly the same as the number of desired order statistics, the reduction step (Subroutine 4) will have no effect. Hence, the overhead in bucketMultiSelect, Alg. 5, Steps 1–5, is wasted as an identical vector to the original is sorted in Step 6. As an example, when the vector is populated with random integers from 0 to 100, bucketMultiSelect requires roughly 4× as much time as sort&choose to recover the 101 order statistics. This algorithm is not suitable for vectors with a limited number of unique values.

Often real data can take on a limited number of unique values. In the following example, we utilize data from the REDD data set [Kolter and Johnson 2011]. These data sets contain actual power readings from several houses including power readings from individual circuits. The full data set, houses, consists of over 42 million entries with less than 367,500 unique values. This raw data contains many outliers, which could be interpreted as adversarial for bucketMultiSelect due to the impact on Subroutine 1. The overwhelming majority of the values fall in the interval $[-200, 200]$. We refer to the data in this interval as houses_small which contains less than 363,500 unique values. Even though there are relatively few unique values and extreme outliers, selecting the decile, percentile, and 1/10th percentile order statistics is faster with bucketMultiSelect than with sort&choose as detailed in Tab. V.

As an example of the application of selecting multiple order statistics, we form approximate piecewise linear6 probability density functions (pdf) using the decile and percentile order statistics. The pdf for the data set houses_small using the eleven decile order statistics appears as the green (dashed) curve in Fig. 5(a) while the pdf for the 101 percentile order statistics is the red (solid) curve. The pdfs are overlaid on top of a normalized histogram using 100 bins of equal size from -200 to 200. Note that the percentile order statistics provide accurate information on the full data set of more than 42 million values. The small, seemingly inaccurate oscillations in the pdf are smoothed when forming the cumulative distribution functions in Fig. 5(b). The cumulative distribution function from the percentile order statistics provides very accurate estimates of the probability a value in the full data set will fall in a given range of values.

4. CONCLUSION

We have presented an algorithm, bucketMultiSelect, for selecting multiple order statistics with a graphics processing unit. For large data sets, bucketMultiSelect is capable of selecting thousands of order statistics in less time than the standard practice of sorting the vector. Moreover, the use of random sampling to form a kernel density estimator and define the assignment process into buckets make bucketMultiSelect relatively stable across many random distributions. Since the set of 101 percentile order statistics provides a highly

---

6 Any spline will work, but a linear interpolant reduces unnecessary oscillation.
Fig. 5. The *houses small* data set. Approximate probability density functions (a) and cumulative distribution functions (b) from the 11 decile order statistics (green, dashed) and from the 101 percentile order statistics (red, solid) overlaid on a normalized histogram (a) and a cumulative, normalized histogram (b) with 100 equal sized bins from -200 to 200.

accurate representation of the full data set, bucketMultiSelect is almost always preferable to sort&choose for selecting order statistics with a GPU for large vectors. In particular, for vectors of type double, bucketMultiSelect provides $8 \times$ acceleration over sorting when selecting the percentile order statistics. While many sampling and bootstrapping techniques provide probabilistic estimates of the distribution of order statistics, bucketMultiSelect provides the precise set of actual order statistics.

ACKNOWLEDGMENTS

The authors would like to thank John Stone of Grinnell College for posing a question leading to this project and whose technical support was critical throughout. The authors thank Patrick Slough for his contributions to the CPU quickMultiSelect. The authors acknowledge the NVIDIA Corporation who provided the GPUs after establishing Grinnell College as a CUDA Teaching Center with PI Jerod Weinman. The authors also thank the anonymous referees for their valuable suggestions.

REFERENCES


Received August 2013; revised February 2015; accepted
Table VI. Mean timings and acceleration ratios for selecting quartiles, deciles, percentiles, and 1/10-percentiles, K40.

<table>
<thead>
<tr>
<th>Vector Type</th>
<th>Uniform</th>
<th>Normal</th>
<th>Vector Distribution</th>
<th>Uniform</th>
<th>Normal</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Float</td>
<td></td>
<td></td>
<td>Float</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Double</td>
<td></td>
<td></td>
<td>Double</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Unsigned Integers</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

ACM Journal Name, Vol. 0, No. 0, Article 00, Publication date: 2013.