Fast K-selection Algorithms for Graphics Processing Units

TOLU ALABI, JEFFREY D. BLANCHARD, BRADLEY GORDON, and RUSSEL STEINBACH, Grinnell College

Finding the kth largest value in a list of n values is a well-studied problem for which many algorithms have been proposed. A naive approach is to sort the list and then simply select the kth term in the sorted list. However, when the sorted list is not needed, this method has done quite a bit of unnecessary work. Although sorting can be accomplished efficiently when working with a graphics processing unit (GPU), this article proposes two GPU algorithms, `radixSelect` and `bucketSelect`, which are several times faster than sorting the vector. As the problem size grows so does the time savings of these algorithms with a sixfold speedup for float vectors larger than 2^{24} and for double vectors larger than 2^{20}, ultimately reaching a 19.1 times speed-up for double vectors of length 2^{28}.

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**]: Parallel and vector implementations;; I.3.1 [**Hardware Architecture**]: Graphics Processors

General Terms: Algorithms, Design, Experimentation, Performance

Additional Key Words and Phrases: K-selection, Order Statistics, Multi-core, Graphics Processing Units, GPGPU, CUDA

1. INTRODUCTION

The k-selection problem, a well studied problem in computer science, asks one to find the kth largest value in a list of n elements. This problem is often referred to as finding the kth order statistic. This task appears in numerous applications and our motivating example is a thresholding operation, where only the k largest values in a list are retained while the remaining entries are set to zero. Both hard and soft thresholding operations are used frequently in digital signal processing algorithms, for example in the greedy algorithms used in compressed sensing [Blumensath and Davies 2010; Dai and Milenkovic 2009; Foucart 2011; Needell and Tropp 2009]. In each iteration of these algorithms, the k largest magnitudes in at least one vector are left unchanged while all other values in the vector are set to zero. This is easily accomplished by performing a k selection on the magnitudes of the vector followed by a hard threshold.

A naive method for finding the kth largest value is to sort the list and then simply obtain the value in the appropriate position in the sorted list. However, sorting the list is often unnecessary if there is not demand for the sorted list. A considerable amount of extra work is undertaken to sort entries of the list that can easily be eliminated from the set candidates for the kth largest value. In many computational algorithms, including iterative greedy algorithms used in compressed sensing, the k-selection problem appears in every

This work is supported by the National Science Foundation under grant DMS-1112612 and the Grinnell College Mentored Advanced Project (MAP) program.
Author's addresses: T. Alabi and R. Steinbach, Department of Computer Science, Grinnell College; J.D. Blanchard and B. Gordon, Department of Mathematics and Statistics, Grinnell College (J.D. Blanchard: `jeff@math.grinnell.edu`).

DOI 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

iteration and having a sorted list is not advantageous since the vector under consideration changes with each iteration. Therefore, these iterative algorithms will repeatedly perform the unnecessary operations required to sort the list. Thus a suitable k-selection can provide massive gains in performance.

With the 2010 introduction of error-correction on Nvidia's Tesla C2050/C2070 graphics processing units (GPU), heterogeneous CPU-GPU computing jumped to the forefront of high performance computing. In fact, three of the top five supercomputers on the June 2011 TOP500 list are CPU-GPU heterogeneous machines with power consumption measurements remarkably lower than the two competitive CPU only machines [TOP500.org 2011]. This CPU-GPU computing environment already has highly optimized sorting procedures available, but widely available GPU-based k-selection algorithms lag behind. In this paper, we describe two GPU implementations for k-selection algorithms, `radixSelect` and `bucketSelect`. `radixSelect` is a selection adaptation of a most-significant digit radix sort while `bucketSelect` is an iterative application of the first step in a bucket sort, namely placing the elements into a fixed number of buckets. `bucketSelect` is a GPU implementation of an algorithm originally referred to as distributive partitioning [Allison and Noga 1980]. Both `radixSelect` and `bucketSelect` demonstrate considerable speed-up over sorting the vector on the GPU with `bucketSelect` achieving speeds up to 19.1 times faster than the state-of-the-art GPU-based sorting algorithm. The algorithms are implemented in CUDA-C, Nvidia's GPU specific extension of C [NVIDIA Corporation 2011].

1.1. k-Selection from Sorting

The reader is almost certainly familiar with the serial version of Quicksort [Hoare 1961a], which sorts a list by partitioning the list according to a pivot. A pivot is selected from any number of techniques, for example the median of three or a random pivot selection. Each entry in the list is then compared to this pivot value. The list is then partitioned into two sublists, one containing only values that are greater than the pivot and the second containing the values that are less than or equal to the pivot value. The algorithm then performs the same techniques on each of the sublists.

If one is not interested in sorting the list, but instead wants to determine the kth largest value in the list, the required work can be reduced considerably. After the initial partition, one need only perform the subsequent iteration the sublist which contains the kth largest value. This is the well-known algorithm Quickselect (originally called Find [Hoare 1961b]). There are several variants of the algorithm which generally alter the method for choosing the pivot element or use more than one pivot value.

The basic idea of Quickselect is to gradually reduce the portion of the list over which you must search for the kth value, in essence performing a dimension reduction during each iteration. This idea is indicative of the general approach to developing selection algorithms, in particular to each of the algorithms discussed in this paper. These k selection algorithms follow a similar formula:

Generic k-Selection Technique

Step 1. Determine the number of bins and a rule for placing values into these bins.
Step 2. Assign the values to these bins according to the assignment rule.
Step 3. Determine which bin contains the kth largest element.
Step 4. Iterate this algorithm only on the bin containing the kth largest value.

As described above, Quickselect[1] clearly follows this pattern. Our implementations of `radixSelect` and `bucketSelect` also follow this recipe. In `radixSelect` sixteen bins are

[1]We have implemented two GPU versions of Quickselect. One is incredibly simple using the `thrust::partition` command and the other is based on GPUQuickSort [Cederman and Tsigas 2010]. While

defined with an assignment to each bin based on the sixteen possible hexadecimal digits for the current digit. For `bucketSelect`, the bins are defined with equispaced endpoints ranging from a minimum to a maximum value relevant to the active portion of the list and values are assigned to the bucket by a linear projection. Both `radixSelect` and `bucketSelect` are highly parallelizable as their assignment rules do not require interaction with other elements in the list.

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 Nvida Tesla GPUs have 14 multiprocessors with 32 cores each for a total of 448 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 wide 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.

1.3. Organization

In this paper we present three algorithms for solving the k-selection problem, `sort&choose`, `radixSelect`, and `bucketSelect`. These algorithms are detailed in Sec. 2 along with the specifics of the CUDA-based GPU implementations for `radixSelect` and `bucketSelect`. In Sec. 3, we discuss the memory requirements of the algorithms. We are primarily concerned with the observed performance of these algorithms which is described in Sec. 4. In this testing, we focus both on lists with entries from standard distributions such as uniform or normal distribution as well as adversarial distributions for both `radixSelect` and `bucketSelect`. The main focus of the paper is `bucketSelect` as it outperforms the other algorithms for non-adversarial data and is the most novel of our contributions. Finally, we compare our algorithm to recent related work by Beliakov [Beliakov 2011] and Monroe et al. [Monroe et al. 2011]. The software package, Grinnell GPU k-Selection, contains all of these algorithms with the testing suite and is available online [Alabi et al. 2011].

---

these algorithms are often faster than `sort&choose` for large vectors, neither is competitive with `radixSelect` or `bucketSelect`.

2. GPU K-SELECTION ALGORITHMS

Here we describe four algorithms for selecting the kth largest value from a list of length n. After describing the algorithm, we frame each algorithm in terms of the generic k-selection technique outlined in Sec. 1.1. Based on our motivating problem from signal processing, we consider the list as a vector containing numeric data (floats, doubles, unsigned ints, or ints). Certainly other applications may need to perform the k-selection problem on other data structures, but the algorithms can be easily tailored to the appropriate setting.

2.1. `Sort&Choose`

As discussed in Sec. 1, one method for finding the kth largest value is to sort the entire vector and then query the sorted vector for the kth largest element. Sorting the vector is a useful technique when having a sorted copy of the vector is advantageous, for example if many order statistics are needed or the vector will not change in subsequent iterations of a parent algorithm. We are only concerned with the k selection problem on GPUs and therefore use a GPU sorting algorithm for this algorithm. With CUDA 4.0 [NVIDIA 2011], Nvidia began distributing the Thrust library [Hoberock and Bell 2010] with the CUDA packages. Currently, the fastest known GPU sorting algorithm is the least significant digit radix sort implementation of Merrill and Grimshaw [Merrill and Grimshaw 2011]. This highly optimized implementation is now included in the Thrust library and is the default algorithm behind `thrust::sort`. This straightforward but often unnecessarily costly k-selection algorithm is described in Alg. 1.

It can be useful to consider `sort&choose` in the context of assigning elements to bins as described in Sec. 1.1. `sort&choose` places the elements of the vector in n bins with the assignment rule being that the ith largest value will be placed in bin $n - i$. In essence, sorting a vector requires that you solve the k-selection problem for all values of k from 1 to n. In order to complete Step 2, namely assigning the values to the appropriate bin, the vector is sorted and the assignments to bins are complete. Step 3 is now straightforward as the kth largest value is clearly in bin $n - k$. Since each bin contains exactly one element, Step 4 is unnecessary.

ALGORITHM 1: `sort&choose`

Input: Vector of length n, `vec`; k
Output: kth largest value
1. Sort the vector with `thrust::sort`;
return $vec[n - k]$

2.2. `RadixSelect`

Analogous to Quickselect from Quicksort, we implement a most significant digit radixSelect based on the highly optimized radix sort by Merrill [Merrill and Grimshaw 2011]. Rather than looking at each digit in subsequent passes through the elements of the vector, this algorithm only performs a radix sort on the digit which contains the kth largest element of the vector. Our implementation is a modification of Merrill's least significant digit radix sort in two ways. First, a least significant digit radix sort will not provide a means to identify a digit containing the kth largest value, and it is therefore not possible to reduce the problem size. Hence `radixSelect` focuses on the most significant digit. Second, we then only iterate on the digit which contains the kth largest value. This has the advantage of reducing the dimension of the problem in each iteration. When the radix only contains a single element or the digit includes the least significant bit, the algorithm terminates.

Like the Merrill algorithm, we consider a hexadecimal digit of four bits and use the one-to-one correspondence between floats or doubles and unsigned integers via a bit flipping

routine. In the first pass, we consider the four most significant bits of the entries in the vector and define a bin as each of the sixteen possible digits. Therefore, in terms of our generic k-selection technique, Step 1 is to define sixteen bins with an assignment rule based on the possible digits. Step 2 can be interpreted as simply having the current digit identify the bin to which the element is assigned. Step 3 is completed by counting the number of times each digit appears, and then using a cumulative count to determine the $KDigit$, namely the digit containing the kth largest value. The counting is performed using the code available from `thrust::sort` [Merrill and Grimshaw 2011].

Step 4 provides the reduction in workload for the algorithm. This reduction takes on two forms, essentially defining two distinct radix based selection algorithms. The first method, `inplace radixSelect`, does not copy the elements with $KDigit$ to a new location, rather it alters the elements of the vector which do not have $KDigit$ to eliminate them from contention as the kth largest element. The second method, which we call simply `radixSelect`, copies the elements whose ith digit is $KDigit$ to an auxiliary vector and iterates only on these elements. The process used in Step 4 is based on the size of the input vector. When the vector is small enough, avoiding the copy removes some overhead and provides a faster selection.

ALGORITHM 2: `inplace radixSelect`

Input: Vector, vec, of length n, the desired order statistic k, and d the number of (hexadecimal) digits in elements of vec
Output: kth largest element of vec
while $i < d$ **do**
  foreach *Digit* j **do**
    $Count[j]$ = occurrences of digit j in the ith digit;
  end
  $CumCount[j] = \sum_{i=0}^{j} Count[i]$;
  $KDigits[i] = l$ where $CumCount[l-1] < n-k$ && $CumCount[l] \geq n-k$;
  foreach $v \in vec$ **do**
    if *digit* i *of* $v < KDigits[i]$ **then**
      $v = 0$;
    end
    else if *digit* i *of* $v > KDigits[i]$ **then**
      v = Largest number representable in d digits;
    end
  end
end
$result$ = assemble the kth largest element from the values in $KDigits$;
return *result*

`inplace radixSelect`. After identifying $KDigit$ in Step 3, the elements in $KDigit$ are left unaltered and $KDigit$ is recorded. Any element in the vector which has a digit greater than the $KDigit$ has all of its bits set to 1, i.e. these elements are set to the largest possible value. Likewise, the elements whose ith digit is smaller than the $KDigit$ are set to 0 by having all of their bits set to 0. When proceeding to the subsequent digit, the kth largest element is now guaranteed to have its $(i+1)$st digit be the kth largest $(i+1)$st digit. After passing through all of the digits, the recorded digits from each iteration define the kth largest value in the original vector. This is described in Alg. 2.

`radixSelect`. As mentioned above, this method accomplishes Step 4 by copying the elements in $KDigit$ to an auxiliary vector. The algorithm then iterates only on the elements of this new vector. This is described in Alg. 3.

ALGORITHM 3: `radixSelect`

Input: Vector, vec, of length n, the desired order statistic k, and d the number of (hexadecimal) digits in elements of vec
Output: kth largest element of vec
while $n > 1$ && $i < d$ **do**
  foreach *Digit* j **do**
    $Count[j]$ = occurrences of digit j in the ith digit;
  end
  $CumCount[j] = \sum_{i=0}^{j} Count[i]$;
  $KDigit = l$ where $CumCount[l-1] < n-k$ && $CumCount[l] \geq n-k$;
  $vec = \{v_i | v_i \in vec$ and the ith digit of $v_i = KDigit\}$;
  $k = k - (n - CumCount[KDigit])$;
  $n = Count[KDigit]$;
end
return $vec[n-k]$

Although we have essentially developed two radix based selection algorithms, the algorithm we test uses both methods to complete Step 4. If the input vector is less than 2^{22}, Step 4 is accomplished via `inplace radixSelect`. Otherwise the algorithm uses `radixSelect`. One important advantage of `inplace radixSelect` is that it requires no additional memory other than the counts of the sixteen digits and recording the digits which identify the kth largest element. As the other algorithms in this paper require auxiliary vectors, `inplace radixSelect` is able to solve the selection problem on vectors twice as long as the largest vectors for `sort&choose`, `radixSelect`, and `bucketSelect`.

2.3. `BucketSelect`

Many parallel sorting algorithms use an initial preprocessing step of subdividing the vector into buckets of relatively equal valued entries and then sort each bucket in parallel. In this algorithm, we simply repeat this bucketing process in each iteration and completely eliminate any sorting. First, we determine the size of the buckets we wish to use, assign each element to a bucket and then identify the bucket which contains the kth largest element. We then focus only on the entries in this bucket, the $Kbucket$, and again project the entries in the $Kbucket$ into a new set of buckets. After a finite number of iterations, the bucket width is less than the machine precision for the vector type and therefore must contain elements of equal value. Since the kth largest element of the list is in this bucket, we have identified the kth largest entry. This algorithm for selection is also known as distributive partitioning [Allison and Noga 1980].

This algorithm leaves the input vector unchanged, unlike any of the previously described selection algorithms. For example, performing `sort&choose` on a vector will alter the vector in that the entries will be moved to their appropriate position in the sorted version of the vector. If the parent application requires the unsorted vector, a copy of the vector will have to be provided to `sort&choose`. `bucketSelect` leaves the entries of the vector in their initial position and simply records the bucket in which they are assigned in a working vector whose life span is the call to `bucketSelect`.

There are many free parameters for this algorithm, most notably the number of buckets. One is initially tempted to use n buckets; for a vector of uniformly distributed elements the expected number of elements assigned to each bucket is one and the algorithm will terminate in a single pass. Even for other distributions the expected number of entries in each bucket will be fantastically smaller than the length of the list providing a major reduction in problem size. However, there is a trade off between the number of buckets and the cost of identifying which bucket contains the desired value as this identification

requires a cumulative count of the entires in all preceding buckets. As the expected number of elements assigned to a bucket increases, the cost of counting the number of elements in each bucket increases. As shown in Alg. 4-5, each time we assign an element to a particular bucket we increment a counter for the number of elements in that bucket. For our parallel implementation, this could cause conflicts without the use of an atomic function.

Atomic functions ensure that multiple simultaneous requests to perform the same task are not lost due to conflicts. Instead, each request is put in a queue and executed serially. We use the atomic increment function `atomicInc` to update the counter for each bin as we assign an element to that bin. When used globally, this causes significant performance degradation. Therefore, we wish to use the atomic functions primarily in shared memory. Since the amount of shared memory on each multiprocessor is restricted, our choice of the number of buckets is likewise restricted. We have chosen (empirically) to use $2^{10} = 1024$ buckets[2] in order to place bucket counters into shared memory, use our atomic functions on shared memory, and then simply report back the values of each of the shared bucket counters to a global bucket counter.

2.3.1. Implementation of `bucketSelect`. Here we describe the details of our implementation of `bucketSelect`. First of all, this is a CUDA-based GPU algorithm and therefore requires control parameters. In particular, each task will be subdivided into blocks, with each block taking a certain number of threads. To maximize the parallel efficiency (which also provides the best empirical performance) we set the number of blocks to the number of multiprocessors on the GPU and the number of threads per block to the maximum number of threads per block[3]. We first describe a typical iteration in terms of the generic selection model from Sec. 1.1 and then describe the full implementation of the algorithm in terms of *phases*.

Generic Steps of `bucketselect`. **Step 1.** In every iteration, the number of bins, which we call buckets in this algorithm, is fixed at $B + 1$. In our implementation, we use $B = 2^{10}$ in order to exploit the shared memory as discussed above. The extra bucket is a dummy bucket where values already eliminated as candidates for the kth largest value are assigned.

The assignment rule is a linear projection of each candidate element in the vector to one of the B buckets. In each iteration, the assignment rule is given a maximum and minimum value, `max` and `min`. The B active buckets are then defined by partitioning the interval $[\texttt{min}, \texttt{max}]$ into B equal length buckets of width $(\texttt{max} - \texttt{min})/B$. The vector entries are assigned to the bucket containing that entry.

Step 2. The assignment to buckets is accomplished by recording the appropriate bucket in a vector of n integers, `bucket`, via the linear projection

$$\texttt{bucket}[i] = \left\lfloor \frac{B}{\texttt{max} - \texttt{min}} \left(\texttt{vec}[i] - \texttt{min}\right) \right\rfloor. \tag{1}$$

If $\texttt{vec}[i] < \texttt{min}$ or $\texttt{vec}[i] > \texttt{max}$, the entry is assigned to the dummy bucket. Thus, relatively small values are assigned to small numbered buckets while larger values are assigned to larger numbered buckets.

During the assignment process, the number of entries assigned to each bucket is recorded in a vector of B integers, `counter`. To exploit the performance gains in using shared memory, a copy of `counter` is sent with each block to each multiprocessor's shared memory. The copy of `counter` is available to each thread in a given block which weighed in our decisions to set the number of blocks to the number of multiprocessors and to set $B = 2^{10}$. The count is performed using an atomic function, `atomicInc`, which eliminates potential conflicts by

[2]With the potential for the development of GPUs with larger shared memory capacities, altering this algorithm to use a larger number of buckets will likely lead to improved performance.

[3]While these values are GPU dependent, they are easily obtained with the built-in CUDA function `cudaGetDeviceProperties()`.

atomically performing the task

$$\text{If } \texttt{bucket}[i] = b, \quad \texttt{counter}[b] = \texttt{counter}[b] + 1. \tag{2}$$

When a block completes the execution of all assigned threads, the shared memory copy of `counter` is then added to the global memory copy of `counter` via the atomic function `atomicAdd`.

Step 3. When all blocks have completed Step 2, `counter` contains the complete count of entries assigned to each bucket defined in Step 1. To identify the bucket containing the kth largest value, we simply need a cumulative count of all buckets up the the $Kbucket$. Due to its relatively small size, `counter` is passed to the CPU and the cumulative sum performed serially, terminating once the cumulative count exceeds $k - 1$. It is certainly possible to transform `counter` into a vector of cumulative totals on the GPU, for example using a scan function. However, the improved communication speed of modern high performance computing GPUs eliminates the perceived advantage of avoiding the memory transfer to the CPU when `counter` is not too large. Empirically, performing this task heterogeneously is preferred. This cumulative sum identifies both the bucket containing the kth largest element, $Kbucket$, and the number of elements in $Kbucket$ which the next iteration must consider.

Step 4. The first three steps provide the necessary information for the subsequent iteration. The assignment vector `bucket` is used to identify candidates for the kth largest value as only those elements $\texttt{vec}[i]$ with $\texttt{bucket}[i] = Kbucket$ are eligible. Depending on the phase described below, the elements in $Kbucket$ are either copied to a new vector or `max` and `min` are updated so that all eliminated entries will be assigned to the dummy buckets in the subsequent iteration.

Phases of `bucketSelect`. The generic advantage of performing a selection over a complete sort is the expected reduction in workload at each iteration. Typically, the reduction in workload is a realized reduction in problem size, such as when Quickselect only performs a subsequent iteration over a subset of the list. In a selection, each iteration identifies a subset of the input vector which contains candidate values for the kth largest value. `bucketSelect` uses two different approaches to reducing the workload, one a problem size reduction and the other a computational reduction. These differences define our two *phases* of `bucketSelect`. In Phase I, the workload reduction will take the form of a physical problem reduction by creating a new, smaller vector containing only the candidate values from the previous iteration. Phase II will simply reduce the work required while leaving the vector unaltered.

In Step 2 above, the highly parallel nature of the assignment process given by (1) is countered by the necessarily serial nature of counting. The serial nature of counting is protected in Step 2 using atomic functions. However the atomic counting procedure, even when exploiting shared memory, is the principal performance burden of Step 2. Phase II will implement a workload reduction by altering the bounds of the bucket while maintaining the actual dimension of the vector. Rather than performing the work required to create a new smaller vector or to move all the elements in $Kbucket$ to a known location in the old vector, Phase II will simply query the entire list again. However, if the element was not assigned to Kbucket in the previous iteration it will quickly be assigned to a dummy bucket of which no count is required. Clearly, this could be fantastically wasteful if the vector was many orders of magnitude larger than the number of candidate values. Thus, entry into Phase I or Phase II is based on the current size of the vector under consideration. In other words, if the initial vector is large (e.g. longer than 2^{21}), Phase I will first create a smaller vector which is passed to Phase II.

Phase I. This algorithm is outlined in Alg. 4. If the initial vector is larger than 2^{21}, `bucketSelect` begins in Phase I. In order to define the buckets, the maximum and min-

imum values in the vector are obtained and assigned to the values `max` and `min`[4]. This is implemented using `thrust::minmax` which finds both values simultaneously. If `max` = `min`, the algorithm has found the kth largest value and terminates. Otherwise, Phase I performs Steps 2 and 3 as described above. Now let $Kcount$ = `counter`[$Kbucket$] be the number of elements assigned to the bucket containing the kth largest value in the vector. At this point, `bucketSelect` allocates memory for a new vector of length $Kcount$ and writes all entries in $Kbucket$ to this new vector.

ALGORITHM 4: `bucketSelect`: Phase I

Input: Vector, vec, of length n, the desired order statistic k, and a number of buckets, B
Output: kth largest element of vec

```
while n > Cutoff do
    max = thrust::max(vec);
    min = thrust::min(vec);
    if max == min then
        return max;
    end
    range = max − min;
    slope = (B − 1)/range;
    assignBuckets(vec, n, B, slope, min, bucket, counter);
    KBucket = FindKBucket(counter, B, k);
    KCount = counter[KBucket];
    if Kcount == 1 then
        if bucket[index] = Kbucket then
            return vec[bucket[index]];
        end
    end
    else
        if KBucket! = 0 then
            k = k − sum + KCount;
        end
        copyElements(vec, n, bucket, KBucket, newVec);
        PhaseII(newVec, Kcount, k, numBlocks);
    end
end
```

Since the initial vector is still intact after Step 2, the entries which were assigned to $Kbucket$ are identified by the corresponding values in the assignment vector `bucket`. At this point, the actual locations of the targeted candidates are unknown which presents a challenge for writing them to the new vector. If one was willing to accomplish this task serially, there would be no challenge as one would pass through `bucket` and write `vec`[i] to the next available position in the new vector whenever `bucket`[i] = $Kbucket$. However, to do this in parallel on the GPU, we must maintain a count of how many places in the new vector have already been filled with the candidates from $Kbucket$. Again, this serialization is accomplished by using an atomic increment of a control parameter.

At this point, a new vector of length $Kcount$ has been created internally which contains only values which are candidates for the output. This has reduced the problem size from n to $Kcount$. If $Kcount$ is still too large, larger than 2^{21}, Phase I is repeated. Otherwise, the new vector is passed to Phase II. For typical data, this problem reduction will be significant.

[4]Obviously, finding the maximum or minimum value is a special case of the k-selection problem which can be solved with a single parallel scan through the list. In our implementation, if $k = 1$ (maximum) or $k = n$ (minimum), we simply call `thrust::max` or `thrust::min`.

However, for adversarial data, this reduction could be as small as one element. Since the allocation of memory for the new vector and writing the values from unknown locations requires an atomic control parameter, Phase I never repeats more than two times.

Phase II. Phase II typically receives a vector of no more than 2^{21} elements, either from phase I or because the input vector is shorter than $2^{21}+1$ and Phase I was bypassed. Phase II will never alter the vector and requires no memory allocation or copying of elements from the vector. In the first iteration, `max` and `min` are determined by finding the maximum and minimum elements in the vector using `thrust::minmax` as described above in Phase I. Again, if `max` = `min` the algorithm has succeeded. If not, Steps 2 and 3 are performed and $Kbucket$ and $Kcount$ identified.

ALGORITHM 5: `bucketSelect`: Phase II

Input: Vector, vec, of length n, the desired order statistic k, and a number of buckets, B
Output: kth largest element of vec

```
max = thrust::max(vec);
min = thrust::min(vec);
if max == min then
    return max;
end
range = max − min;
slope = (B − 1)/range;
assignBuckets(vec, n, B, slope, min, bucket, counter);
KBucket = FindKBucket(counter, B, k, sum);
KCount = counter[KBucket];
while KBucket > 1 && (max − min) > 0 do
    min = maximum(min, min + KBucket/slope);
    max = minimum(max, min + 1/slope);
    k = k − sum + KCount;
    if max − min > 0 then
        slope = (B − 1)/(max − min);
        reassignBucket(vec, n, B, slope, min, bucket, counter, KBucket);
        sum = 0;
        KBucket = findKBucket(counter, B, k, sum);
        KCount = counter[KBucket];
    end
    else
        return max;
    end
end
return GetKValue(vec, n, bucket, KBucket)
```

If $Kcount = 1$, the algorithm has succeeded. If not, the algorithm enters Step 4 and iterates on the entire vector. Here the workload is reduced by redefining `max` and `min`. Since the locations of the elements in $Kbucket$ are unknown, identifying the true maximum and minimum values in $Kbucket$ would be costly. Instead, the values are updated to be the boundaries of $Kbucket$, namely

$$\texttt{min}_{new} = \texttt{min} + Kbucket\left(\frac{\texttt{max}-\texttt{min}}{B}\right); \qquad \texttt{max}_{new} = \texttt{min}_{new} + \left(\frac{\texttt{max}-\texttt{min}}{B}\right). \tag{3}$$

Phase II then passes to the next iteration using these new values for `max` and `min`. Therefore, in the subsequent iteration, only the elements identified in the current iteration as belonging to $Kbucket$ will require the computations from (1) and (2). The iterations continue until

Table I. Memory Requirements for `sort&choose`, `radixSelect`, and `bucketSelect` for a vector of length n.

	Altered Input Vector				Preserved Input Vector			
	Worst Case		Expected (Uniform)		Worst Case		Expected (Uniform)	
	float/int	double	float/int	double	float/int	double	float/int	double
`sort&choose`	$2n$	$2n$	$2n$	$2n$	$3n$	$3n$	$3n$	$3n$
`radixSelect`	$2n$	$2n$	$2n$	$2n$	$3n$	$3n$	$3n$	$3n$
`bucketSelect`	$3n$	$2.5n$	$2.001n$	$1.501n$	$3n$	$2.5n$	$2.001n$	$1.501n$

either $Kcount = 1$ or $\texttt{max} - \texttt{min}$ has reached machine precision for the data type of the initial vector.

3. ANALYSIS OF ALGORITHMS

The three algorithms described in Sec. 2 are all comparable in memory requirements; see Tab. I. Since `sort&choose` and `radixSelect` are based on Merrill's `thrust::sort`, the memory requirements are the same. Besides some constant amount of memory, these algorithms require two vectors on which to read and write in each iteration, one working vector and one auxiliary vector which is updated for the subsequent iteration. Thus the memory requirement for `sort&choose` and `radixSelect` is $2n * \texttt{sizeof}(type)$. Both of these algorithms alter the input vector by either sorting the vector or, in the case of `radixSelect`, partially sorting the vector. If the input vector is required by a parent algorithm, a copy of the vector must be given to the selection algorithm. In this case the memory requirement is $3n * \texttt{sizeof}(type)$.

To determine the candidates for the kth largest value in each iteration, `bucketSelect` constructs an an assignment vector `bucket` as described in Sec. 2.3. This assignment vector is a vector of integers. To determine the bucket which contains the remaining candidates, `bucketSelect` also forms a vector of integers whose length is the number of buckets. In our implementation, this is fixed at 1024. Also, in Phase I of `bucketSelect`, a new vector is created and the elements assigned to the $Kbucket$ are written to this vector. The worst case size of this vector is $n-1$ while for a uniformly distributed vector, the expected length of this vector is $2^{-10} * n$. Hence the worst case memory requirements for `bucketSelect` is a constant plus $(2n-1) * \texttt{sizeof}(type) + n * \texttt{sizeof}(int)$. However the expected memory requirement for a uniformly distributed input vector is essentially $1.001n * \texttt{sizeof}(type) + n * \texttt{sizeof}(int)$. If the input vector is of type float or int, the fact that the assignment vector is of type int is irrelevant for memory purposes. However, when the input vector is of type double, the memory requirement for the assignment vector is $.5n$. The memory requirements appear in Tab. I.

The analysis of these algorithms in terms of operations is certainly penetrable. However, it is not precisely relevant to the experimental nature of this article and is therefore omitted. The number of operations is dependent on the number of iterations performed by each of the algorithms. `sort&choose` and `radixSelect` have a maximum number of iterations fixed at the number of bits divided by four for the input data type (8 for integers or floats, 16 for doubles). The requisite number of iterations for `bucketSelect` is variable and depends on the make-up of the vector. The following theorem establishes a worst case number of iterations for `bucketSelect`.

THEOREM 3.1. *`bucketSelect` will select the kth largest entry in a finite number of iterations given by*

$$maxIterations = \left\lceil \left| \log_{1024} \frac{\texttt{max} - \texttt{min}}{d} \right| \right\rceil \tag{4}$$

where `max` and `min` are the maximum and minimum values in the input vector and d is the smallest difference between any two distinct elements in the list.

The proof of Thm. 3.1 is straightforward. It is important to note that this is a worst case result which occurs only for highly adversarial vectors.

4. PERFORMANCE EVALUATION

This section details the observed performance of the GPU k-selection algorithms described in Sec. 2. After describing our methodology and hardware configuration in Sec. 4.1, we present the data which shows both `radixSelect` and `bucketSelect` solve the selection problem considerably faster than `sort&choose`. We present findings for some standard distributions (Sec. 4.2), evaluate the effects of data type on the algorithms (Sec. 4.2.1), investigate the dependence of the algorithms on the distribution (Sec. 4.2.2) and on the value of k (Sec. 4.2.3), and attack `bucketSelect` with adversarial vectors (Sec. 4.2.5).

4.1. Testing Environment

We developed a testing suite which can be used on any platform with a CUDA-capable GPU with compute capability 1.3 or higher. The testing suite creates a vector, runs each of the chosen algorithms on the same vector, and records the output and timing for each algorithm. This data is written to a file and two functions are used to compile the data into comma separated files for processing. We then use Matlab to generate the tables and plots which help interpret the performance of the algorithms.

In order to determine that the algorithms have selected the correct value for a given problem, the out put of the algorithms are recorded. We make the assumption that `sort&choose` will always select the correct value. Therefore, the output of any other algorithm is compared to the output of `sort&choose`; if an algorithm were to make a different selection, an error flag is set in the output data. This permits us to easily check that the algorithms are performing correctly. Furthermore, to ensure the data was reasonably clean, the device was reset after 50 tests using a built-in CUDA function depending on the CUDA version.

4.1.1. Hardware Configuration for Testing. We performed testing on two configurations at Grinnell College, one equipped with Nvidia's Tesla C2070 and the other with Nvidia's GTX480. As we are primarily concerned with algorithms to be used in high performance computing (HPC) applications, we are more interested in the data from the C2070 configuration. The C2070 is a HPC GPU which includes error correcting code. As the C2070 has fewer multiprocessors than the GTX480, it necessarily has a smaller degree of parallelization. This combined with error correction shows slower algorithm performance on the C2070 than on the GTX480. The results presented here were obtained using the C2070; testing on the GTX480 showed similar performance of the algorithms.

— **C2070** Running Debian with CUDA 3.2., the testing configuration has multiple Intel Xeon 5620 CPUs @ 2.40 GHz and an NVIDIA Tesla C2070 GPU. The C2070 has 14 multiprocessors each with 32 cores for a total of 448 cores. The C2070 has 6GB GDDR5 total dedicated (global) memory.

Each of the algorithms is memory limited in that the length of the vectors on which they can perform selection is determined by the global memory of the GPU. The C2070 permitted lengths between 2^{29} and 2^{30} for floats[5] and unsigned integers and between 2^{28} and 2^{29} for doubles. As described below, we tested vectors whose length were powers of two, thus the largest vectors tested are of length 2^{29} on the C2070.

4.1.2. Problem Formulation. In our testing suite, we built a problem generation function which creates two primary random distributions for testing. The two distributions[6] are

[5] `inplace radixSelect` enables us to solve the k-selection problem on vectors with more than 2^{30} elements.
[6] Throughout the paper, $\mathcal{U}(a, b)$ is the uniform distribution on the interval $(a, b]$ while $\mathcal{N}(\mu, \sigma^2)$ is the normal distribution with mean μ and variance σ^2.

the uniform distribution $\mathcal{U}(0,1)$ and the normal distribution $\mathcal{N}(0,1)$. We tested three data types: floats, doubles, and unsigned integers[7]. These three data types and two distributions were combined into five non adversarial testing distributions:

— **UniformUINT** The vector was populated with unsigned integers which were uniformly distributed over the full range of unsigned integers: $\{0, 1, \ldots, 2^{32} - 1\}$.
— **UniformFLOAT** The vector was populated with floats from $\mathcal{U}(0,1)$.
— **NormalFLOAT** The vector was populated with floats from $\mathcal{N}(0,1)$.
— **UniformDOUBLE** The vector was populated with doubles from $\mathcal{U}(0,1)$.
— **NormalDOUBLE** The vector was populated with doubles from $\mathcal{N}(0,1)$.

The problems were generated randomly on the GPU using the pseudo random number generator provided in CUDA, namely CURAND. In Sec. 4.2, the length of the vectors tested were $n = 2^p$ for $p \in \{14, 15, \ldots, 29\}$ for floats and unsigned integers and $p \in \{14, 15, \ldots, 28\}$ on doubles. For each value of n, we selected 25 values of k, namely $k = 2$,

$$k = \lfloor \alpha n \rfloor \text{ for } \alpha \in \{.01, .025\} \cup \{.05, .10, .15, \ldots, .85, .90, .95\} \cup \{.975, .99\},$$

and $k = n - 1$. For each pair (n, k), we generated 50 problems for each of the five testing distributions listed above. The three algorithms were tested on the same set of 38,750 problems for each of the five non adversarial testing distributions for a total of 581,250 tests.

Additionally, we performed 10 tests for each pair (n, k) on three variations of these distributions for another 69,750 tests. The half-normal distribution was formed by creating a vector with entries from $\mathcal{N}(0,1)$ and then taking the absolute value of all entries. This distribution would be expected to reduce the speed of both `radixSelect` and `bucketSelect` as it would double the number of nearly equal values in a vector of the same length. We conjectured that increasing the variance in the normal distribution or expanding the range of values from the uniform distribution would prove advantageous to `radixSelect` while having little effect on `bucketSelect`. To test these conjectures, we created non-adversarial problems from $\mathcal{N}(0, 100)$ and $\mathcal{U}(-10^6, 10^6)$.

In Sec. 4.2.5, we discuss an additional 93,000 tests of the algorithms against adversarial vectors specifically designed to penalize the use of atomic functions in `bucketSelect` or to simply remove the advantage of a workload reduction in any selection algorithm. Finally, in Sec. 5, we report the results of 271,250 tests which include two recent GPU k-selection algorithms introduced in 2011. The following sections investigate the efficacy of the algorithms based on these 1,015,250 tests.

4.2. `radixSelect` and `bucketSelect` vs. `sort&choose`

We begin with the UniformUINT data where the vector was populated with uniformly distributed unsigned integers from across the range of unsigned integers. This distribution should be the most favorable distribution for `radixSelect` as this distribution is equivalent to a uniform sampling of all possible bit patterns. For `bucketSelect` any uniform distribution is expected to minimize the effects of any atomic functions as each bucket will be expected to contain roughly $2^{-10} * n$ elements from the vector. Also, as seen from Thm. 3.1, `bucketSelect` will require no more than 4 iterations for UniformUINT. Table II shows a sampling of the odd powers of 2 from 2^{17} to 2^{29}. We see from both Tab. II and Tab. VII that both `radixSelect` and `bucketSelect` significantly outperform `sort&choose` with 3-7 times speed-ups for uniformly distributed unsigned integers.

The raw data in Tab. II shows the consistent performance of each of the algorithms on this distribution. `sort&choose` performs precisely the same operations regardless of the distribution and should have little fluctuation between the minimum and maximum times

[7]We also tested signed integers, but observed no difference between signed and unsigned integers.

Table II. Minimum, mean, and maximum timings (ms) for `sort&choose`, `radixSelect`, and `bucketSelect`, Unsigned Integers (Uniform), C2070.

	`sort&choose`			`radixSelect`			`bucketSelect`		
n	min	mean	max	min	mean	max	min	mean	max
2^{17}	0.8402	0.9012	1.0061	0.9985	1.0279	1.1120	0.7445	0.8349	1.0637
2^{19}	1.8003	1.8743	2.0573	1.4419	1.4777	1.5542	1.0103	1.1506	1.3798
2^{21}	5.0848	5.1750	5.3209	2.1664	2.5107	2.8934	1.6571	2.0013	2.3966
2^{23}	18.6836	18.8710	19.0239	4.4635	4.6379	5.0512	3.5818	3.8061	4.1530
2^{25}	72.7715	73.5514	74.2858	12.8250	13.3656	13.9775	10.7120	10.9807	11.3913
2^{27}	288.6710	291.9060	294.7190	46.2728	47.2903	48.6638	39.7620	40.1594	40.9113
2^{29}	1162.4300	1171.1700	1181.1500	181.2060	184.6030	189.8990	154.9020	156.5890	157.3720

required to solve a selection problem. `radixSelect` achieves greater acceleration when each iteration of the algorithm significantly reduces the problem size for the subsequent iteration. This is consistently achieved for UniformUINT with the minimum and maximum times within 9% of the mean time for $n \geq 2^{25}$. `bucketSelect` has even smaller variance as the problem size grows with the minimum and maximum times within 1.5% of the mean time for $n = 2^{29}$.

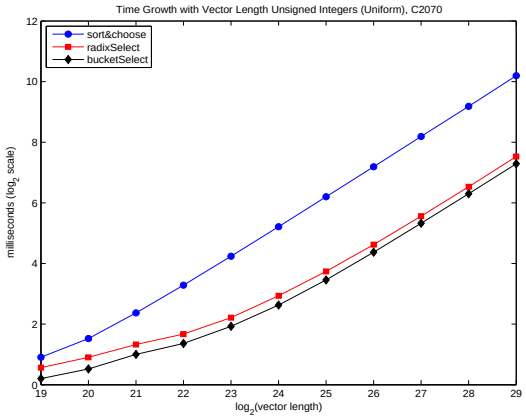
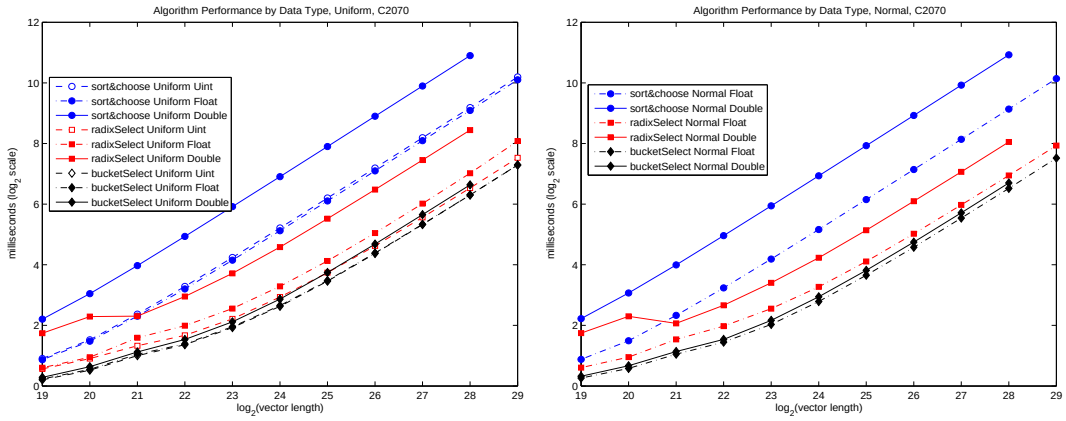


Fig. 1. Algorithm Performance for `sort&choose`, `radixSelect`, and `bucketSelect`: mean times ($\log_2(ms)$) on UniformUINT.

Figure 1 shows the mean timings for all $n \geq 2^{19}$ of the three algorithms with the times plotted on a $\log_2(ms)$ scale. For small n, `sort&choose` occasionally solved the selection problem faster than the other algorithms, but as the algorithm grows `bucketSelect` is fastest on nearly every test. `radixSelect` has the occasional victory when k is relatively extreme (close to 1 or close to n) since the most significant radix is often very small in these cases.

When the distribution is changed to either $\mathcal{U}(0,1)$ or $\mathcal{N}(0,1)$, the most significant digit is less likely to fantastically reduce the problem size for most values of k. This causes a reduced mean performance for `radixSelect` although as the problem size grows we clearly see that `radixSelect` is superior to `sort&choose`. `bucketSelect` is seemingly undeterred by the change in distribution and the advantages `bucketSelect` over `sort&choose` remain evident.



(a) (b)

Fig. 2. Performance by data type under a fixed distribution: (a) Uniform and (b) Normal.

Table III. Minimum, mean, and maximum timings (ms) for `sort&choose`, `radixSelect`, and `bucketSelect`, Floats U(0,1), C2070.

	`sort&choose`			`radixSelect`			`bucketSelect`		
n	min	mean	max	min	mean	max	min	mean	max
2^{17}	0.8308	0.8903	1.0265	1.0225	1.0548	1.1461	0.7355	0.8460	1.2775
2^{19}	1.7486	1.8265	1.9671	1.4601	1.5195	1.5996	1.0185	1.1742	1.7571
2^{21}	4.8523	4.9372	5.0640	1.4437	3.0173	3.7883	1.2106	2.0673	3.1138
2^{23}	17.5978	17.7587	17.9086	3.5481	5.8720	8.2731	3.5182	3.8798	4.2849
2^{25}	68.2046	68.8952	70.7823	11.8739	17.4619	28.2056	10.7288	11.1427	11.4554
2^{27}	270.3860	273.3170	281.7800	44.9640	64.8031	107.4910	39.4910	40.3515	40.7312
2^{29}	1085.2900	1097.5100	1125.3000	136.7980	270.9400	422.8280	154.7980	156.7760	157.4460

In Fig. 2 we observe the mean times of the three algorithms on the remaining four distributions: UniformFLOAT, UniformDOUBLE[8] , NormalFLOAT, NormalDOUBLE. Interestingly, `bucketSelect` is nearly immune to the change in data type from floats to doubles while `sort&choose` and `radixSelect` are required to check twice as many bits. This results in a noticeable performance gain for `bucketSelect` compared to the other two algorithms. The dependence of the algorithms on data type, distribution, and the value of k are discussed more fully in the following subsections. For a selection of vector lengths, n, the minimum, mean, and maximum times are reported for each algorithm solving the selection problem on a vector whose entries are drawn from $\mathcal{U}(0,1)$ in Tab. III (floats) and Tab. IV (doubles). The timings for solving the selection problem with a vector with entries from $\mathcal{N}(0,1)$ are shown in Tab. V (floats) and Tab. VI (doubles).

The data in Tables III-VI show considerable speed-up for both `radixSelect` and `bucketSelect` over sorting the vector. Moreover, the data reveals important performance features of both `radixSelect` and `bucketSelect`. When `radixSelect` is able to substantially reduce the problem size early in the algorithm, impressive minimum times are achieved. However, the range of timings between the maximum and minimum timings is relatively substantial. On the other hand, `bucketSelect` has a much narrower band of timings for each of the testing distributions. This is graphically depicted in Fig. 3. `sort&choose` is

[8]The sudden speed-up of `radixSelect` on doubles at $n = 2^{21}$ is an artifact of the choice of cut-off to change from `inplace radixSelect` to `radixSelect`. In our tests, we used the same cut-off for both floats and doubles to demonstrate the benefits of tuning these parameters.

Table IV. Minimum, mean, and maximum timings (ms) for `sort&choose`, `radixSelect`, and `bucketSelect`, Doubles U(0,1), C2070.

	`sort&choose`			`radixSelect`			`bucketSelect`		
n	min	mean	max	min	mean	max	min	mean	max
2^{17}	1.7640	1.8399	1.9653	1.9394	1.9967	2.0757	0.8832	0.9786	1.1879
2^{19}	4.5168	4.6152	4.8051	3.2622	3.3442	3.4445	1.0718	1.2198	1.5236
2^{21}	15.5830	15.6925	15.8151	2.3295	4.9401	5.8905	1.8574	2.1775	2.4955
2^{23}	60.1907	60.4227	60.6914	6.1209	13.1398	15.8347	4.1724	4.3416	4.6007
2^{25}	238.2380	239.0420	240.0740	20.9202	45.8662	55.1469	13.1708	13.4117	13.7231
2^{27}	950.4660	953.8190	956.2140	80.4464	175.3450	212.0500	50.0332	50.2917	50.6080
2^{28}	1905.9800	1909.9900	1914.9400	159.7100	347.8840	421.8760	99.2446	99.5465	99.9913

Table V. Minimum, mean, and maximum timings (ms) for `sort&choose`, `radixSelect`, and `bucketSelect`, Floats N(0,1), C2070.

	`sort&choose`			`radixSelect`			`bucketSelect`		
n	min	mean	max	min	mean	max	min	mean	max
2^{17}	0.8335	0.8953	1.0288	1.0224	1.0585	1.1579	0.6442	0.8587	1.2857
2^{19}	1.7605	1.8375	1.9728	1.4662	1.5211	1.6274	0.8887	1.2004	1.7248
2^{21}	4.9038	5.0301	5.1456	1.9187	2.9056	3.4272	1.4315	2.0715	2.9173
2^{23}	17.8502	18.2376	18.4496	3.3861	5.8592	6.8204	3.5346	4.0917	4.5858
2^{25}	69.7808	71.0147	71.8757	10.5613	17.2451	20.5934	11.5556	12.5846	13.6666
2^{27}	277.4780	281.7280	284.9330	38.6123	62.8593	75.5137	43.8558	46.4916	49.8771
2^{29}	1108.0300	1128.3700	1140.7200	149.9080	243.9630	294.6750	174.4140	183.4080	212.2650

Table VI. Minimum, mean, and maximum timings (ms) for `sort&choose`, `radixSelect`, and `bucketSelect`, Doubles N(0,1), C2070.

	`sort&choose`			`radixSelect`			`bucketSelect`		
n	min	mean	max	min	mean	max	min	mean	max
2^{17}	1.7644	1.8423	1.9782	1.9396	1.9954	2.1071	0.7704	0.9818	1.2341
2^{19}	4.5634	4.6706	4.8172	3.2628	3.3527	3.6237	0.9244	1.2505	1.4816
2^{21}	15.7238	15.9528	16.1091	2.5058	4.1852	5.0495	1.5914	2.2007	2.6501
2^{23}	60.7145	61.5074	61.8848	5.7465	10.5815	12.7828	3.9226	4.4825	4.9255
2^{25}	240.8890	243.4500	244.6180	18.9366	35.2653	42.6335	13.1120	14.0837	15.2460
2^{27}	961.4290	971.5180	975.9660	71.5328	133.8990	162.9150	50.0018	52.4536	56.1033
2^{28}	1924.9400	1944.6100	1955.7400	141.4560	265.1480	323.1220	99.2814	103.8820	110.4690

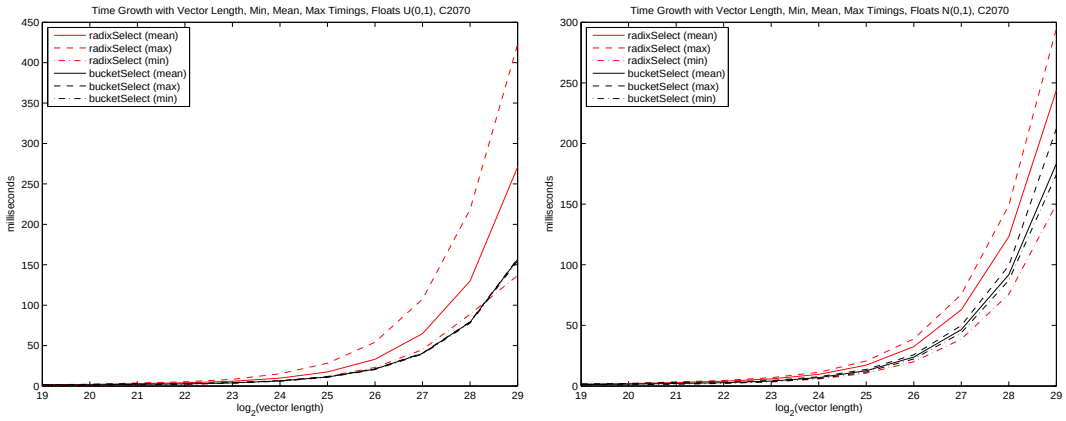
not included in this figure as the range of timings is expected to be rather small given that the algorithm sorts the entire vector for every vector.

From Tab. II-VI, it is clear that the algorithm with the fastest mean time for $n \geq 2^{17}$ for each distribution is `bucketSelect`[9]. In Tab. VII, we represent the acceleration of solving the k-selection problem by a ratio of `sort&choose` to `bucketSelect` or `radixSelect`. `bucketSelect` exhibits impressive speed-up over `sort&choose`, especially when working with doubles. For uniformly distributed floats, `bucketSelect` solves the selection problems 1.9 times faster than `sort&choose` when $n = 2^{20}$ and grows to a 7.0 times acceleration for $n = 2^{29}$. For doubles, `bucketSelect`'s advantage is amplified achieving an 19.1 times speed-up at $n = 2^{28}$. For vectors drawn from a normal distribution, similar speed-ups are achieved reaching a factor of 18.7 for normally distributed doubles.

In the following subsections, we investigate the dependence of the algorithms on varying one factor in the problem, namely data type, distribution, or the values of k.

4.2.1. Algorithm Dependence on Data Type. In this section, we investigate the dependence of each of the algorithms on the data type. For `sort&choose` it is expected that the number of bits is the only factor which should effect the timings. The dependence of `sort&choose` on

[9]For $n \leq 2^{15}$, `sort&choose` has the fastest mean time and performs the fastest selection in most cases. For $n = 2^{16}$, either `bucketSelect` or `sort&choose` has the fastest mean timing dependent on distribution.



(a) (b)

Fig. 3. Minimum, Mean, and Maximum timings (ms) for `radixSelect` and `bucketSelect`: (a) Uniform-FLOAT $\mathcal{U}(0,1)$, (b) NormalFLOAT $\mathcal{N}(0,1)$.

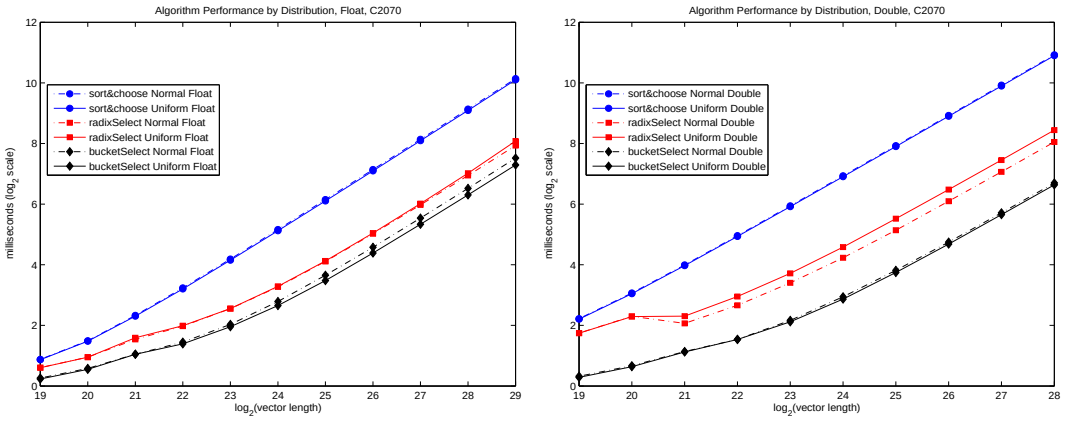
Table VII. Speed-up: Ratio of timings for `sort&choose` to `bucketSelect` and `sort&choose` to `radixSelect`, Uniformly Distributed Elements, C2070.

	Uniform						Normal			
	`sort&choose` / `bucketSelect`			`sort&choose` / `radixSelect`			`sort&choose` / `bucketSelect`		`sort&choose` / `radixSelect`	
n	uint	float	double	uint	float	double	float	double	float	double
2^{17}	1.0794	1.0524	1.8800	0.8767	0.8441	0.9215	1.0427	1.8765	0.8459	2.0325
2^{18}	1.3187	1.2906	2.6579	1.0653	1.0277	1.0822	1.2739	2.6385	1.0308	2.4271
2^{19}	1.6290	1.5556	3.7837	1.2684	1.2021	1.3801	1.5307	3.7349	1.2080	2.6810
2^{20}	2.0045	1.9020	5.3049	1.5350	1.4408	1.6883	1.8807	5.2749	1.4541	3.0976
2^{21}	2.5858	2.3882	7.2066	2.0611	1.6363	3.1765	2.4283	7.2490	1.7312	1.9018
2^{22}	3.7862	3.5097	10.5498	3.0598	2.3155	3.9503	3.4566	10.7218	2.3993	2.1831
2^{23}	4.9580	4.5772	13.9170	4.0688	3.0243	4.5984	4.4572	13.7218	3.1126	2.3606
2^{24}	5.9994	5.5254	16.4609	4.8542	3.5625	5.0063	5.1837	15.8995	3.7136	2.4441
2^{25}	6.6982	6.1830	17.8234	5.5030	3.9455	5.2117	5.6430	17.2859	4.1180	2.5040
2^{26}	7.0522	6.5492	18.5865	5.9367	4.1442	5.3540	5.9095	18.1046	4.3451	2.5463
2^{27}	7.2687	6.7734	18.9657	6.1726	4.2177	5.4397	6.0598	18.5215	4.4819	2.5527
2^{28}	7.3838	6.9055	19.1869	6.2864	4.2009	5.4903	6.1182	18.7194	4.5590	2.5524
2^{29}	7.4793	7.0005	–	6.3443	4.0507	–	6.1522	–	4.6252	–

the number of bits is evident in Fig. 2 as it takes more than twice as long for `sort&choose` to solve the problem for the 64 bit doubles than for the 32 bit floats or unsigned integers. The same effect of bit length would also be expected of `radixSelect`. However, this dependence is considerably less pronounced when `radixSelect` solves the selection problem on a normally distributed vector as the algorithm exploits the advantages of having a smaller number of elements with the same digit earlier in its iterative process.

Both `sort&choose` and `radixSelect` examine the bits of the entries. On the other hand, `bucketSelect` performs its assignment task computationally. The process of assigning elements to a vector to the buckets is mostly independent of the data type with a very small penalty for projecting 64 bit elements instead of 32 bit elements. This is clearly depicted in Fig. 2 where `bucketSelect` has relatively consistent mean timings over each of the data types.

4.2.2. Algorithm Dependence on Distribution. Similar to the previous subsection, we vary one aspect of the testing to investigate the dependence of the algorithms on distribution, either uniformly or normally distribution elements in the vector. In Fig. 4, the data type is fixed



(a) (b)

Fig. 4. Performance by distribution under a fixed data type: (a) floats, (b) doubles.

as either a float or a double while the distribution is altered from $\mathcal{U}(0,1)$ to $\mathcal{N}(0,1)$. Again, `sort&choose` is expected to be unaffected by the distribution of the elements in the vector since it will pass over every bit of every element. Both of our algorithms are expected to have some dependence on distribution, however Fig. 4 shows that both `radixSelect` and `bucketSelect` are not significantly affected by a change from the uniform distribution to the normal distribution. The advantages of reducing the workload for the algorithm are evident in the surprising fact that `radixSelect` actually solves the selection problem faster on the normal distribution. A change in distribution certainly alters the dependence of `radixSelect` and `bucketSelect` on the value of k. This is discussed in Sec. 4.2.3.

4.2.3. Algorithm Dependence on k. In our testing, we selected a wide range of values for k as described in Sec. 4.1.2. In the plots in this section, the value of k is not spaced linearly at the ends; rather the ends have $k = 2$, $k = \lfloor .01n \rfloor$, $k = \lfloor .025n \rfloor$, and $k = \lfloor .975n \rfloor$, $k = \lfloor .99n \rfloor$, $k = n - 1$ while the remainder of the values of k are sampled in increments of $.05 * n$. In Fig. 5 the dependence of `radixSelect` and `bucketSelect` on the value of k is plotted for a sampling of values of $n = 2^{23}, \ldots, 2^{29}$ for both $\mathcal{U}(0,1)$ and $\mathcal{N}(0,1)$. Likewise, Fig. 6 has similar plots for the data type double. The dependence on k across all n and both data types shows that the k dependence of the algorithms is inherent in the distribution. Clearly, `sort&choose` is independent of k.

As expected, `radixSelect` is dependent on k as it performs a most significant digit partition in each iteration. When the vector is populated with entries of relatively equal magnitude, the most significant bits will be similar. Therefore, when the kth largest value in the vector has an uncommon digit earlier in its bit sequence, the algorithm will perform faster. This is evident in the dependence of `radixSelect` on k for both uniformly and normally distributed elements. In particular, in the normal distribution, the extreme values of k (near 1 or n) have relatively few other entries of equal magnitude in the vector. Furthermore, the observed acceleration when searching for the median is expected since the median is expected to be very close to zero.

The dependence on k for `bucketSelect` also is expected. For a vector with uniformly distributed elements, the value of k should have very little impact. For this distribution, it is expected that the number of candidates for the kth largest value should always be decreasing proportionally with the number of buckets (in our tests, the number of buckets is 1024). At the same time, when `bucketSelect` is asked to find the kth largest value of a vector whose elements are not uniformly distributed, the reduction in workload will be less

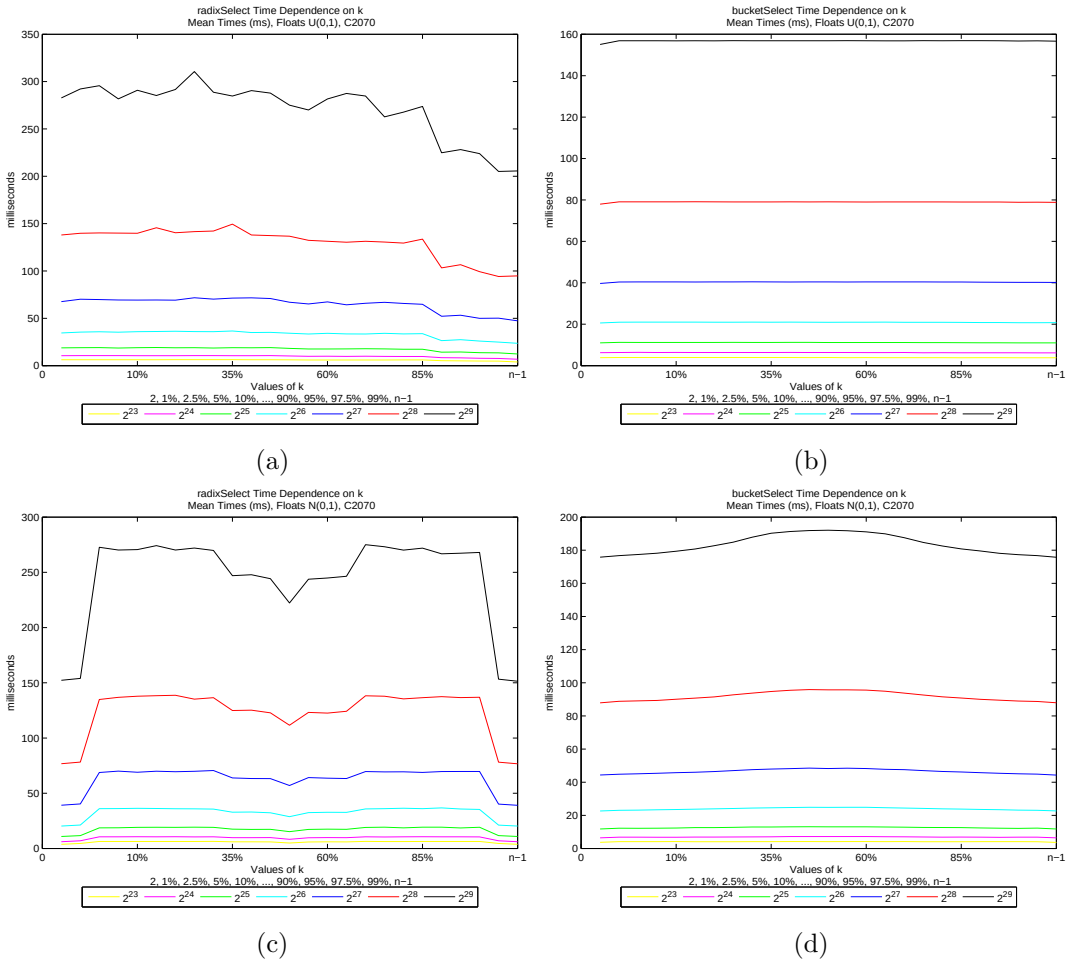


Fig. 5. Algorithm dependence on k for floats: (a),(c) `radixSelect` and (b),(d) `bucketSelect`. (a),(b) UniformFLOAT $\mathcal{U}(0,1)$; (c),(d) NormalFLOAT $\mathcal{N}(0,1)$.

substantial whenever there are many elements of similar size relative the range of potential candidates remaining. However, this dependence is not significant after the first iteration. For example, after projecting a normally distributed vector into buckets, each bucket should contain elements that are much more uniformly distributed than in the previous iteration.

4.2.4. Additional Distributions. In this section, we provide similar analysis of algorithm performance on vectors populated with floats drawn from three additional non adversarial distributions: the half normal distribution, $\mathcal{U}(-10^6, 10^6)$, and $\mathcal{N}(0, 100)$. As discussed in Sec. 4.1.2, these non adversarial distribution are minor changes to those tested extensively in the previous sections. The half normal distribution is expected to roughly double the number of equal sized elements in the vector. By taking the absolute value of a vector drawn from $\mathcal{N}(0,1)$, we create a slightly more difficult task for both `radixSelect` and `bucketSelect`. The algorithms' performance on the half-normal distribution is shown in Fig. 7 with the relative speed-up of `bucketSelect` and `radixSelect` to `sort&choose` show in Tab. VIII. The noticeable increase in maximum timings for `bucketSelect` on the half-normal distribution are all observed when k is close to n. When k is close to n the algorithm

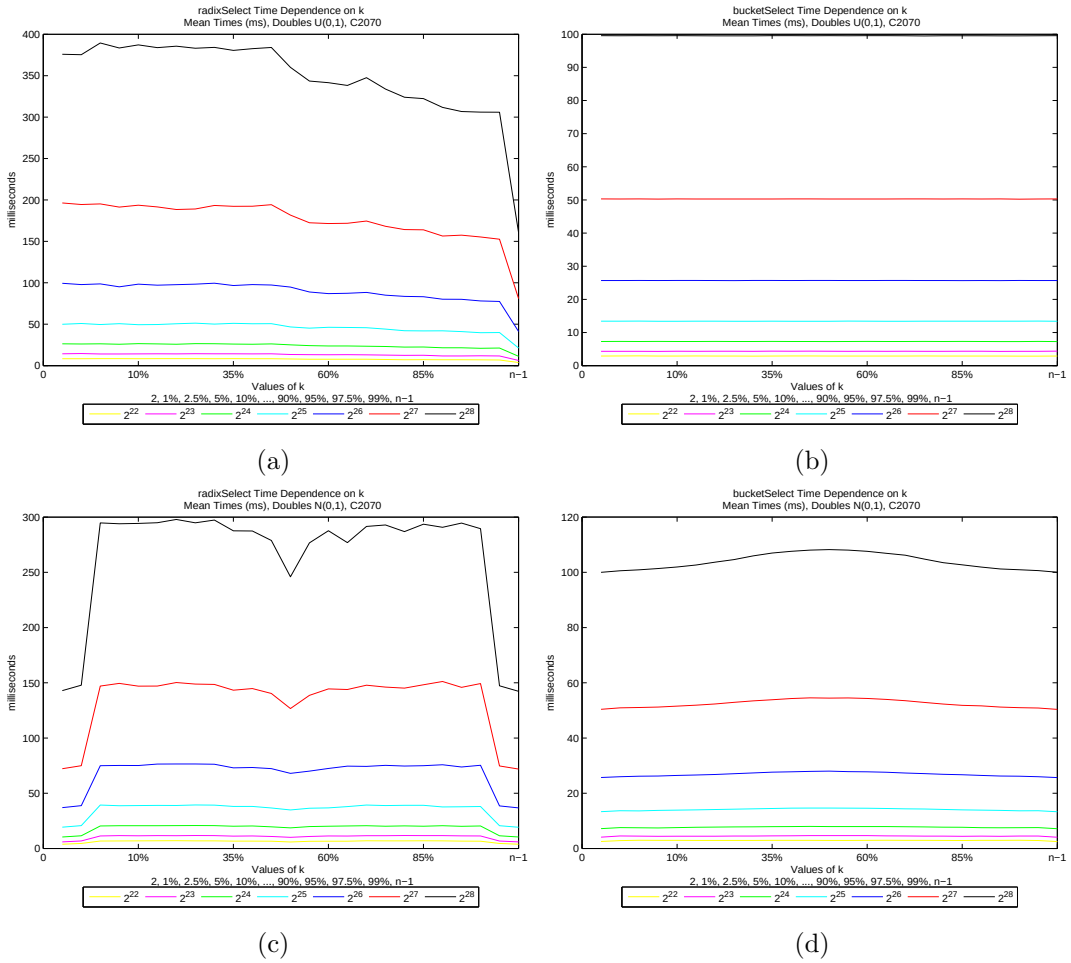
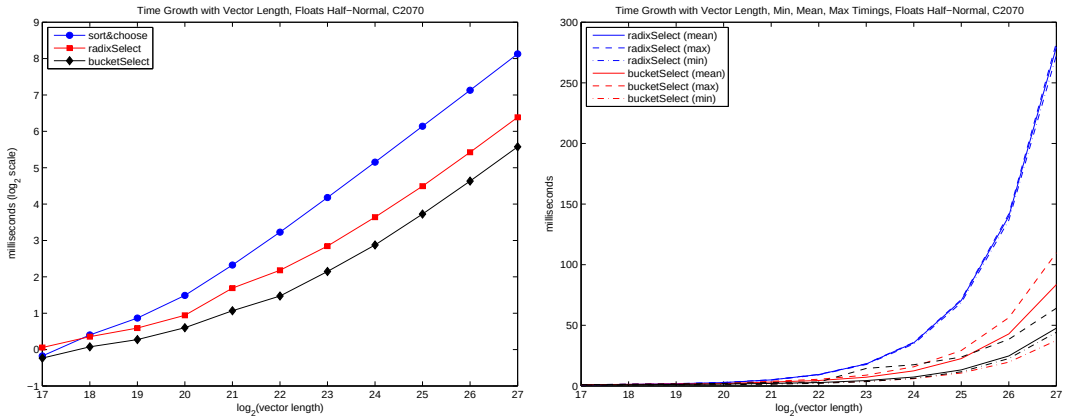


Fig. 6. Algorithm dependence on k for doubles: (a),(c) `radixSelect` and (b),(d) `bucketSelect`. (a),(b) UniformDOUBLE $\mathcal{U}(0,1)$; (c),(d) NormalDOUBLE $\mathcal{N}(0,1)$.

is asked to find an element that is close to the minimum value in the list. For the half-normal distribution, a large portion of the list is close to the minimum value. Therefore, the initial workload reduction is much smaller when physically reducing the length of the list in Phase I of `bucketSelect` (Alg. 4).

The other two distributions simply expand the range of values by scaling the standard distributions above to $\mathcal{U}(-10^6, 10^6)$ and $\mathcal{N}(0, 100)$. The larger range of values in a vector from either of these distributions will require a larger variety of bit patterns and should provide an advantage for `radixSelect`. Again, `bucketSelect` is relatively unaffected by this scaling since it takes the range of values in the vector into consideration when assigning elements to their respective buckets. In Tab. VIII we see that while `bucketSelect` maintains its advantage over both other algorithms, the advantage over `radixSelect` is reduced from the distributions with a smaller range of values. Figure 8 shows the mean timings ($\log_2(ms)$) as n grows from 2^{17} to 2^{27}. The data supports our conjectures about the behavior of the algorithms.

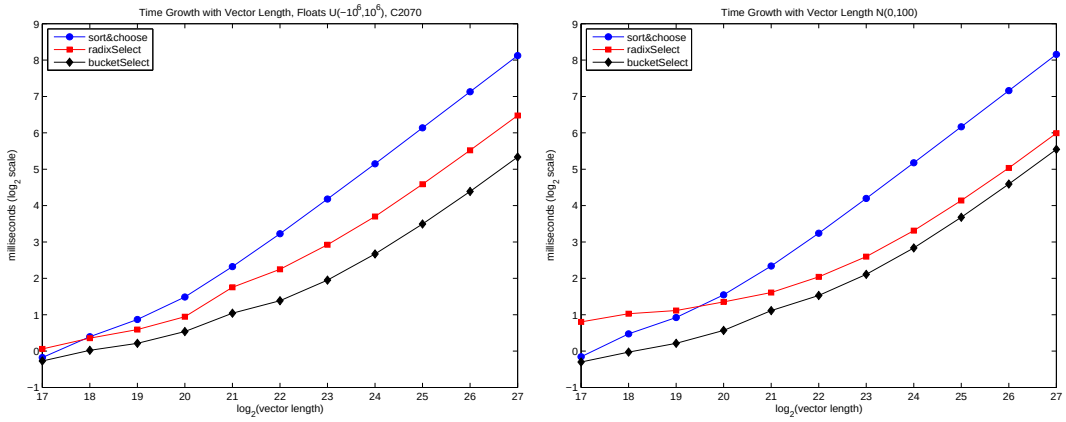


(a) (b)

Fig. 7. Algorithm Performance on the half-normal distribution: (a) mean times ($\log_2(ms)$) (b) minimum, mean, and maximum times (ms).

Table VIII. Speed-up on Additional Distributions: Ratio of timings for `sort&choose` to `bucketSelect`and `radixSelect`, Floats, C2070.

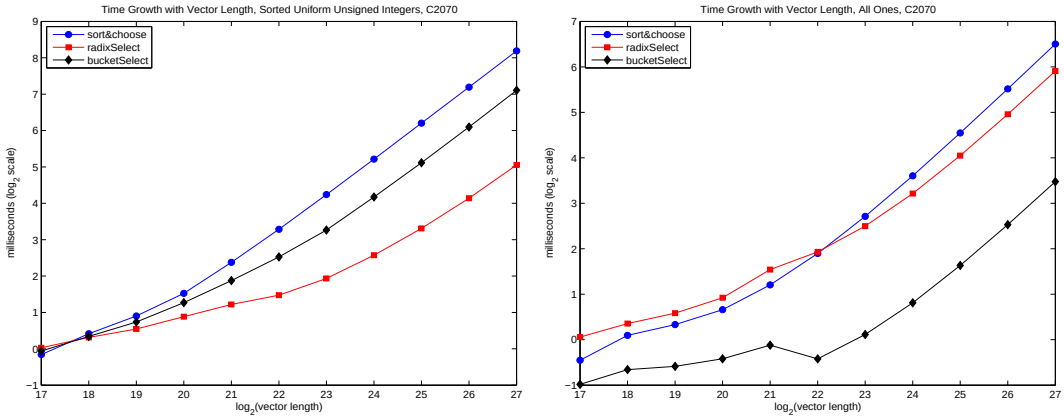
n	$\frac{\texttt{sort\&choose}}{\texttt{bucketSelect}}$ Half-Normal	$\mathcal{U}(-10^6, 10^6)$	$\mathcal{N}(0, 100)$	$\frac{\texttt{sort\&choose}}{\texttt{radixSelect}}$ Half-Normal	$\mathcal{U}(-10^6, 10^6)$	$\mathcal{N}(0, 100)$
2^{17}	1.0422	1.0672	1.0607	0.8511	0.8482	0.8627
2^{19}	1.5075	1.5750	1.5275	1.2099	1.2107	1.2132
2^{21}	2.3913	2.4297	2.4292	1.5534	1.4807	1.7524
2^{23}	4.0924	4.6847	4.3683	2.5230	2.3882	3.1614
2^{25}	5.3312	6.2487	5.6082	3.1314	2.9247	4.1817
2^{27}	5.8644	6.9122	6.0302	3.3435	3.1324	4.6038



(a) (b)

Fig. 8. Algorithm Performance on additional distributions, floats, mean times ($\log_2(ms)$): (a) $\mathcal{U}(-10^6, 10^6)$; (b) $\mathcal{N}(0, 100)$.

4.2.5. Adversarial Distributions. As with most selection algorithms, it is relatively easy to identify vectors which will cause particular trouble for the algorithms. The only algorithm that will not suffer based on the distribution of the elements in the vector is `sort&choose`; `sort&choose` is completely stable as it performs precisely the same operations regardless of the make-up of the vector. Here we tested some obvious adversarial vectors for `bucketSelect`. In the preceding discussion, the value of `radixSelect` is unclear as our alternative algorithm `bucketSelect` outperforms `radixSelect` in almost every situation presented above. However, `radixSelect` shares some of the stability of `sort&choose` in that its worst case performance is the need to examine every bit for every element in the vector. This stability will prove advantageous when faced with large adversarial vectors.



(a) (b)

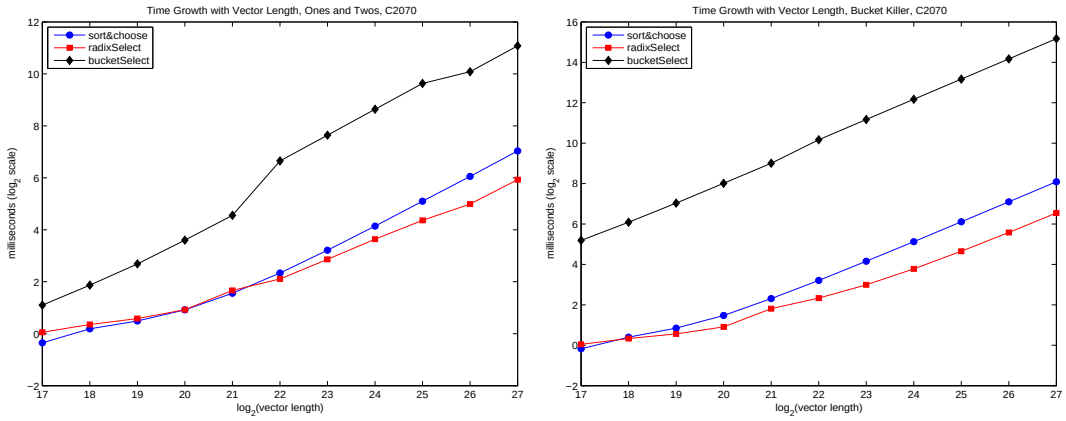
Fig. 9. Algorithm Performance on adversarial distributions, mean times ($\log_2(ms)$): (a) sorted unsigned integers; (b) a vector of all ones.

As described in Sec. 2.3, `bucketSelect` uses atomic functions for counting the number of entries assigned to a given bucket. One relatively obvious way to construct an adversary for the atomic functions in `bucketSelect` is to give the algorithm a sorted list. Our first adversarial vector is a sorted list of unsigned integers across the range of possible unsigned integers. While this is an adversarial vector, it is plausible that one might face a vector with like entries in close proximity in the list. In this case, the atomic queues in the shared memory could be long. We see in Fig. 9(a) that `bucketSelect` is still up to two times faster than `sort&choose`. Importantly, as the vector length grows `radixSelect` becomes the fastest selection algorithm beating `bucketSelect` by a factor of 4.

A second attack on the atomic functions in `bucketSelect` is the naive idea of filling the vectors with a single value. Here we created a vector of all ones. `bucketSelect` necessarily computes the maximum and minimum value in the vector and therefore is able to identify the fact that the vector contains only a single value. Interestingly, `sort&choose` is still required to pass through the entire list while `radixSelect` has been designed to recognize that no problem reduction is possible in each iteration. Therefore, in each iteration `radixSelect` will identify that there are no distinct digits in the current digit place and simply jump to the next radix. This allows `radixSelect` to obtain an advantage over `sort&choose` as the vector length grows, and `radixSelect` eventually solves the problem up to 2 times faster than `sort&choose`.

A more methodical development of adversarial vectors clearly demonstrates the liability when the vector has a structure which will force long queues from the atomic functions.

For example, if the vector is length 10^6 and contains only values from $\{1, 2\}$ with 95% of the entries being 1, the atomic function counting the elements assigned to the first (zeroth) bucket will have a queue of length 9.5×10^6 (although this is distributed over the blocks). Fortunately, the algorithm will terminate after one iteration, but we can force a growing number of iterations by changing our set of values in an adversarial fashion discussed later. To demonstrate this effect, we construct two adversarial vectors which greatly penalize `bucketSelect`. First, a vector with entries from $\{1, 2\}$ will not only cause problems for `bucketSelect` but will also remove much of the advantage of `radixSelect` since each iteration will not significantly reduce the problem size. As the vector length grows, we see the stability of `radixSelect` pay off as it is eventually 2.1 times faster than `sort&choose`. Understandably, `bucketSelect` is not as fortunate and takes up to 23 times as long as `sort&choose`. This is shown in Fig. 10(a).



(a) (b)

Fig. 10. Algorithm Performance on adversarial distributions, mean times ($\log_2(ms)$): (a) vector containing ones and twos; (b) Bucket Killer.

Even more menacing is a vector we have appropriately named Bucket Killer. This vector was formed to cause two simultaneous problems. First, we want the problem reduction obtained in any iteration to be very small; that is each iteration only eliminates a few elements from contention. Second, in doing so, we have forced a single bin to take on all but a few of the entries in the vector of length n, thus creating a queue up to length $n - 1$ in the atomic counting. A simple proxy for this vector is comprised of each of the values 2^p for $p \in \{-32, -31, \ldots, 31, 32\}$. We then fill the remaining $n - 65$ entries in the vector with random values near 2^{-32}. This causes both massive atomic queuing and minimal problem reduction. Fortunately, `radixSelect` is able to handle this vector in a reasonable fashion, but the vector lives up to our moniker. The mean timings ($\log_2(ms)$) are shown in Fig. 10(b). On average, `bucketSelect` now takes 135 times as long as `sort&choose` and up to 395 times as long as `radixSelect` to solve the k-selection problem on the Bucket Killer with $n = 2^{27}$.

Finally, `bucketSelect` has one more adversary, namely its need to define the buckets computationally. It is possible to cause `bucketSelect` to actually fail. When assigning elements of the vector to buckets as defined by (1), one must divide the number of buckets by the difference between the maximum value and the minimum value. Therefore, if we construct a vector filled with entries very near machine zero, this will cause a computational division by zero. Because the computations in `bucketSelect` are done in double precision, this problem was not realized for the data type float. However, a vector populated with

doubles near machine zero can cause `bucketSelect` to fail. While this is clearly a problem, the likelihood of encountering a vector of this nature in a typical application is small enough to accept this fault of `bucketSelect`.

These adversarial vectors are clearly unlikely to occur in typical data. If it is known that data might contain such vectors other selection algorithms are clearly in order. For example, if faced with a vector that contains only the values 1 and 2, one could simply sum the vector and easily identify the kth largest value for any k. Moreover, the user is likely to have some idea of the data which will be considered by the algorithms. If the user is aware of the possibility of having a vector that is populated with a small number of fixed values, or a vector with many entries near machine zero, `radixSelect` is the algorithm of choice for large values of n. However, if the user is faced with data that is likely to contain a large range of values that are not machine zero, `bucketSelect` is clearly the fastest currently known algorithm. This is further demonstrated in the following section where two other new GPU k-selection algorithms are compared against the algorithms we have discussed thus far.

5. RELATED WORK: `CUTTINGPLANE` AND `RANDOMIZEDSELECT`

In the past year, two other k-selection algorithms have been proposed which use sophisticated techniques for selecting pivot elements. In March 2011, Beliakov introduced a selection algorithm which chooses pivots based on convex optimization [Beliakov 2011]. In August 2011, Monroe et al. introduced a deterministic algorithm based on a random selection of pivots [Monroe et al. 2011]. Both of these algorithms outperform `sort&choose` and have their own advantages and disadvantages. Interestingly, the algorithms proposed in this paper, `radixSelect` and `bucketSelect`, each have comparable performance to one of these algorithms. In most cases `bucketSelect` has the fastest mean times of all four GPU selection algorithms.

5.1. `cuttingPlane`

Beliakov introduced a selection algorithm which chooses pivots based on optimization techniques which he refers to as Kelly's cutting plane method [Beliakov 2011]. We henceforth refer to this k-selection algorithm as `cuttingPlane`. In that paper, the author establishes that `cuttingPlane` is faster than `sort&choose`. In our testing on a C2070, we found `cuttingPlane` to perform better than reported in [Beliakov 2011]. This algorithm is comparable to `radixSelect` while `bucketSelect` is faster in nearly every case we tested. Figure 11 shows the mean times of all five algorithms (including `randomizedSelect` described in Sec. 5.2) under the same conditions as described in Sec. 4.1. In this case, we performed representative testing over 2^p for $p \in \{14, \ldots, 29\}$ on the C2070 configuration. Notably, `cuttingPlane` is faster than `radixSelect` when solving the selection problem on doubles drawn from the uniform distribution. However, `bucketSelect` has mean timings considerably below those of `cuttingPlane`. `cuttingPlane` is not capable of performing a selection for unsigned integers.

As mentioned above, the testing suite assumes that `sort&choose` will always return the correct value and checks the other algorithms against this value. Unfortunately, `cuttingPlane` struggles on the uniform distribution when $k = 2$; in 1600 tests performed when $k = 2$ over the range of n, `cuttingPlane` failed 310 times (or roughly 19% of the time). In each instance, it returns the value 0. We believe this is due to the computational nature in which the algorithm selects its pivots. When $k = 2$, we are selecting the second largest uniformly distributed value in the interval $(0, 1]$. It is likely the case that roundoff errors in the optimization phase of `cuttingPlane` determine incorrect pivots defining an incorrect set of possible values. In [Beliakov 2011] the author focuses on finding the median; it appears `cuttingPlane` does not suffer from these errors when searching for order statistics which lie well inside the range of values in the vector.

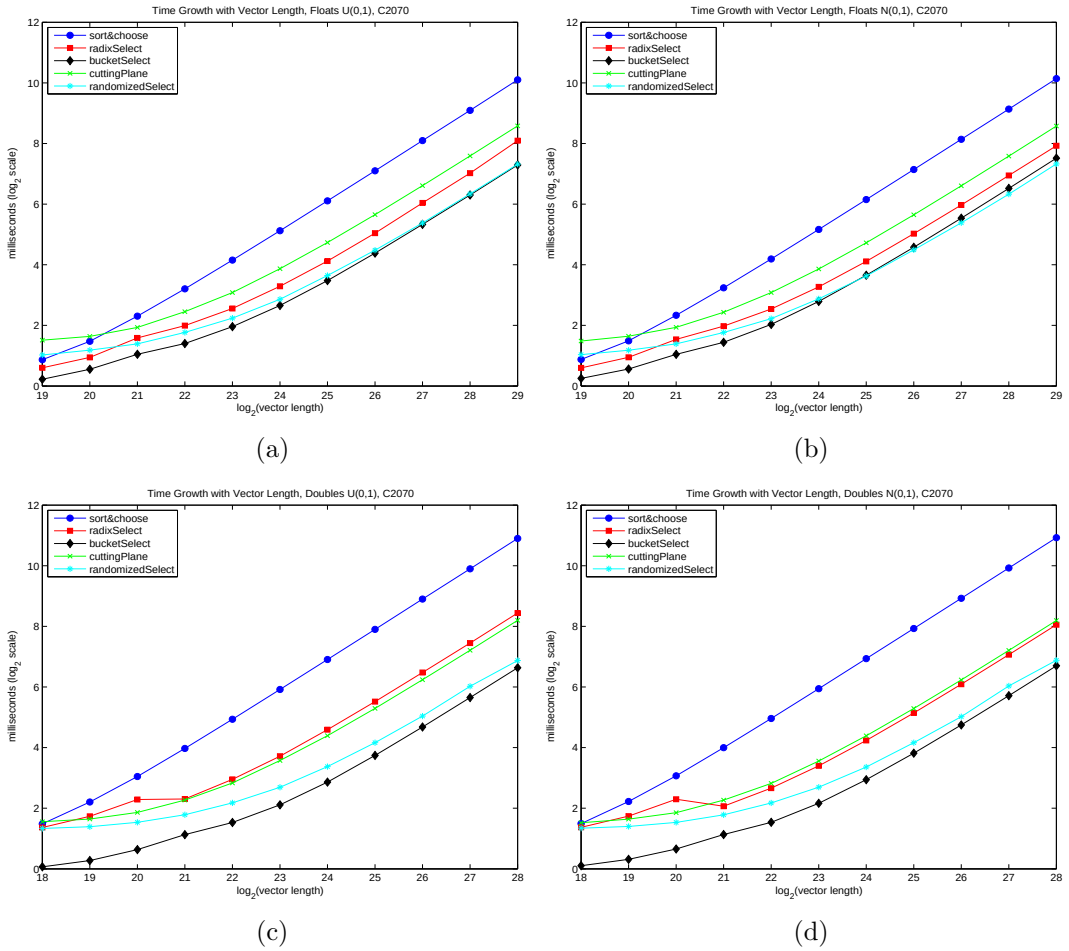


Fig. 11. Comparison of algorithms including `cuttingPlane` and `randomizedSelect` for (a),(b) floats and (c),(d) doubles drawn from (a),(c) $\mathcal{U}(0,1)$ and (b),(d) $\mathcal{N}(0,1)$.

5.2. `randomizedSelect`

An alternative method for selecting the pivots is to randomly select the pivot elements in a manner that guarantees the algorithm will succeed. This is the approach taken by Monroe et al. in the algorithm we refer to as `randomizedSelect` [Monroe et al. 2011]. In this algorithm, a random subset of elements from the vector are chosen as endpoints for assignment bins. Using the binomial distribution, the algorithm determines the probability that the kth largest value appears in each bin. The bins are then combined into three bins with the middle bin having a certain probability, p_k, that it contains the kth largest element. The elements of the vector are then assigned to these three bins and if the kth largest element lies in the middle bin, the middle bin is sorted. In [Monroe et al. 2011], the authors performed testing on unsigned integers with $p_k = .8$. In our tests, we found that using $p_k = .9$ provided fasters mean times on floats and doubles.

Our implementation of `randomizedSelect` is based on [Monroe et al. 2011] and personal correspondence with Monroe. In addition to the the difference in setting $p_k = .9$ there are two other differences to the implementation by Monroe et al. First, they use CUDPP sort; this sorting technique is not capable of sorting doubles and the most recent release of

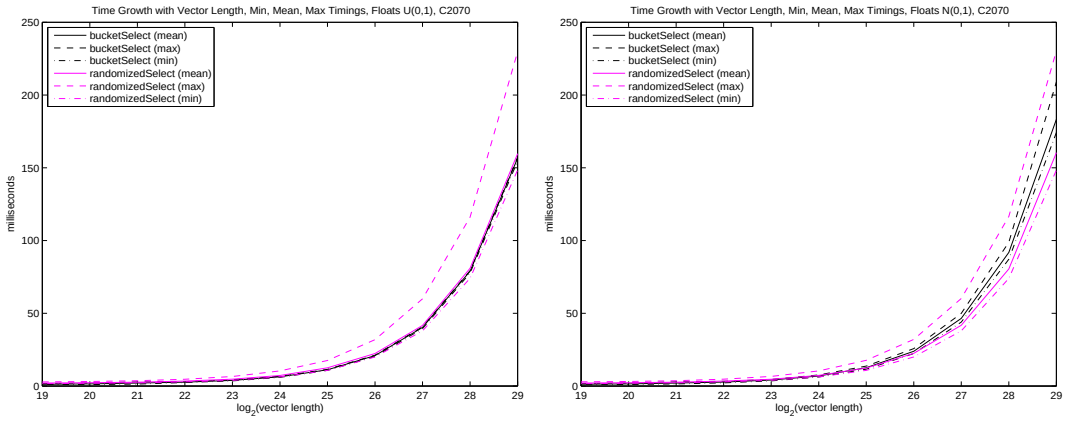
Table IX. Speed-up: Ratio of timings for `randomizedSelect` to `bucketSelect`, Uniformly Distributed Elements, C2070.

	randomizedSelect / bucketSelect				
	Uniform			Normal	
n	uint	float	double	float	double
2^{17}	2.2199	2.1910	2.4841	2.1856	2.4781
2^{18}	1.9938	1.9658	2.4074	1.9373	2.3588
2^{19}	1.7689	1.7412	2.1651	1.7201	2.1211
2^{20}	1.5905	1.5461	1.8686	1.5348	1.8368
2^{21}	1.3159	1.2697	1.5740	1.2703	1.5702
2^{22}	1.3123	1.2879	1.5664	1.2532	1.5579
2^{23}	1.2313	1.2144	1.4946	1.1407	1.4444
2^{24}	1.1848	1.1534	1.4235	1.0598	1.3350
2^{25}	1.1365	1.1235	1.3382	0.9825	1.2732
2^{26}	1.0787	1.0688	1.2839	0.9416	1.2081
2^{27}	1.0547	1.0329	1.2961	0.8976	1.2501
2^{28}	1.0226	1.0254	1.1788	0.8764	1.1333
2^{29}	1.0212	1.0195	–	0.8750	–

CUDPP has adopted the radix sort of Merrill and Grimshaw [CUDPP.org 2011]. Therefore, our implementation of `randomizedSelect` uses `thrust::sort` to sort the middle bin. Second, the implementation by Monroe et al. was actually a *Top-k* algorithm which returned a vector containing all elements of the list which are at least as large the kth largest element. Our implementation is a pure k-selection algorithm and is therefore faster. Finally, it is possible that our method of counting the number of elements in each bin is not the same method of counting implemented by Monroe et al. Despite these differences, we believe our implementation is representative of the performance of `randomizedSelect` and is the only publicly available implementation [Alabi et al. 2011].

From Fig. 11(a),(b), we see that `bucketSelect` is noticeably faster than `randomizedSelect` for smaller problems on floats, and that the performance of the two algorithms is comparable as the vector length grows. As expected, the randomized selection of pivots makes `randomizedSelect` more immune to variations in distribution of the elements in the vector. Since `bucketSelect` does have some dependence on distribution, the mean times for `randomizedSelect` are smaller than the mean times for `bucketSelect` on larger vectors of floats drawn from the normal distribution. When solving the k-selection problem for vectors of type double, `bucketSelect` always outperforms `randomizedSelect`. This is likely a result of the sorting step for `randomizedSelect`.

While Fig. 11 graphically depicts the relationship between the mean times for `randomizedSelect` and `bucketSelect` we present the ratio of the mean timings in Tab. IX. A ratio greater than 1 indicates that the mean time for `bucketSelect` was smaller than the mean time for `randomizedSelect` while a ratio less than 1 indicates that `randomizedSelect` was faster than `bucketSelect` on average. Table IX is further indication that `bucketSelect` and `randomizedSelect` have comparable performance on average. It is worth noting that `randomizedSelect` has a larger variance of times resulting from instances where the random selection of pivots does not place the kth largest element into the middle bin. On the other hand, `bucketSelect` has a more narrow band of timings as shown in Fig. 12. In Fig. 12 we show the minimum, mean, and maximum times for floats drawn from both the uniform distribution (where `bucketSelect` has lower mean times throughout) and the normal distribution (where `randomizedSelect` has lower mean times for larger vectors). Notice that the maximum times for `bucketSelect` are always lower than the maximum times for `randomizedSelect` while the minimum times have the opposite relationship.



(a) (b)

Fig. 12. Minimum, Mean, and Maximum Times for `bucketSelect` and `randomizedSelect` for floats drawn from (a) $\mathcal{U}(0,1)$ and (b) $\mathcal{N}(0,1)$.

6. CONCLUSIONS

In this paper we have introduced two new GPU k-selection algorithms, `radixSelect` and `bucketSelect`, and exhibited their performance against the benchmark of `sort&choose`. It is clear that these algorithms outperform `sort&choose` with `bucketSelect` achieving a 19 times acceleration on large vectors of doubles. Furthermore, we have noted some of the vulnerability of these algorithms, in particular that `bucketSelect` is susceptible to poor performance against highly adversarial vectors. The use of atomic functions for counting is the main vulnerability in `bucketSelect` and could potentially be removed. However, it is also clear that these adversarial distributions are unlikely to be encountered in practice. Finally, we compared these algorithms to two other GPU k-selection algorithms, `cuttingPlane` and `randomizedSelect`. When solving the k-selection problem on uniformly distributed floats or on non-adversarial vectors of doubles, `bucketSelect` is currently the fastest GPU k-selection algorithm.

ACKNOWLEDGMENTS

The authors would like to thank Jared Tanner of the University of Edinburgh who provided many of the ideas in `bucketSelect` to the second author during similar work on compressed sensing algorithms. The authors would also like to thank the NVIDIA Corporation who provided the GPUs after establishing Grinnell College as a CUDA Teaching Center. Jerod Weinman, John Stone, and Jeff Leep were also important in acquiring, setting-up, and administering the CUDA capable machines used for this research.

REFERENCES

Alabi, T., Blanchard, J. D., Gordan, B., and Steinbach, R. 2011. GGKS: Grinnell GPU k-Selection. Version 1.0.0, `http://code.google.com/p/ggks/`.

Allison, D. and Noga, M. 1980. Selection by distributive partitioning. *Information Processing Letters 11,* 1, 7 – 8.

Beliakov, G. 2011. Parallel calculation of the median and order statistics on gpus with application to robust regression. *Computing Research Repository abs/1104.2.*

Blumensath, T. and Davies, M. 2010. Normalized iterative hard thresholding: Guaranteed stability and performance. *Selected Topics in Signal Processing, IEEE Journal of 4,* 2, 298 –309.

Cederman, D. and Tsigas, P. 2010. Gpu-quicksort: A practical quicksort algorithm for graphics processors. *J. Exp. Algorithmics 14*, 4:1.4–4:1.24.

CUDPP.org. 2011. Cudpp change log. `http://cudpp.googlecode.com/svn/tags/2.0/doc/html/changelog.html`.

Dai, W. and Milenkovic, O. 2009. Subspace pursuit for compressive sensing signal reconstruction. *IEEE Trans. Inform. Theory 55,* 5, 2230–2249.

Foucart, S. 2011. Hard thresholding pursuit: an algorithm for compressive sensing. *SIAM J. Num. Anal.*. in press.

Hoare, C. A. R. 1961a. Algorithm 64: Quicksort. *Commun. ACM 4*, 321–.

Hoare, C. A. R. 1961b. Algorithm 65: find. *Commun. ACM 4*, 321–322.

Hoberock, J. and Bell, N. 2010. Thrust: A parallel template library. Version 1.3.0, `http://www.meganewtons.com/`.

Merrill, D. and Grimshaw, A. 2011. High performance and scalable radix sorting: A case study of implementing dynamic parallelism for GPU computing. *Parallel Processing Letters 21,* 02, 245–272.

Monroe, L., Wendelberger, J., and Michalak, S. 2011. Randomized selection on the gpu. In *Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics*. HPG '11. ACM, New York, NY, USA, 89–98.

Needell, D. and Tropp, J. 2009. CoSaMP: Iterative signal recovery from incomplete and inaccurate samples. *Appl. Comp. Harm. Anal. 26,* 3, 301–321.

NVIDIA. 2011. Cuda toolkit 4.0. `http://developer.nvidia.com/cuda-toolkit-40`.

NVIDIA Corporation 2011. *NVIDIA CUDA C Programing Guide, version 4.0*. NVIDIA Corporation.

TOP500.org. 2011. Top 500 List - June 2011. `http://top500.org/list/2011/06/`.

Received December 2011; revised ; accepted