Randomized Selection on the GPU

Laura Monroe
Los Alamos National Laboratory
Los Alamos, NM, 87545
lmonroe@lanl.gov

Joanne Wendelberger
Los Alamos National Laboratory
Los Alamos, NM, 87545
joanne@lanl.gov

Sarah Michalak
Los Alamos National Laboratory
Los Alamos, NM, 87545
michalak@lanl.gov

Abstract

We implement here a fast and memory-sparing probabilistic top *k* selection algorithm on the GPU. The algorithm proceeds via an iterative probabilistic guess-and-check process on pivots for a three-way partition. When the guess is correct, the problem is reduced to selection on a much smaller set. This probabilistic algorithm always gives a correct result and always terminates. Las Vegas algorithms of this kind are a form of stochastic optimization and can be well suited to more general parallel processors with limited amounts of fast memory.

CR Categories: D.1.3 [**Concurrent Programming**]: Parallel programming. F.2.2 [**Non-numerical Algorithms and Problems**]: Non-numerical Algorithms and Problems – *Sorting and searching*. G.3 [**Probability and Statistics**]: Probabilistic Algorithms. 1.3.1 [**Computer Graphics**]: Hardware Architecture – *Graphics processors*. 1.3.1 [**Computer Graphics**]: Hardware Architecture – *Parallel processing*.

Keywords: Experimental parallel algorithms; GPGPU; Las Vegas algorithm; probabilistic algorithms; selection, *k*th order statistic.

1 Introduction

Selection of the top *k* elements from an unordered list is a basic computer science algorithm. We present a fast GPU implementation of a randomized but deterministic parallel algorithm for selecting the top *k* keys and values of the list.

In this algorithm, two pivots are guessed probabilistically and are used to partition the list into three bins. Keys and associated values are placed into the first bin if the keys are less than the first pivot, and into the middle bin if the keys are between the two pivots. If the *k*th key then falls into the middle bin, the algorithm completes with a selection on this relatively small bin. Otherwise, new pivots are chosen and the process iterates.

A top *k* selection is relatively hard to compute directly, but it is easy to check the placement of the *k*th key midway into this guess-based calculation. The algorithm is therefore a serial guess-and-check process intelligently implemented, rather than a straight calculation. The guess-and-check process always terminates, and can be specified to terminate with probability at least p_k after only one iteration. Although the algorithm uses probabilistic methods, it always calculates a correct, unordered set of all top *k* elements, and extracts both the keys and the values.

This work is published under LANL LA-UR 11-10765.

HPG 2011, Vancouver, British Columbia, Canada, August 5 – 7, 2011.
© 2011 ACM 978-1-4503-0896-0/11/0008 $10.00

In this paper, we provide a justification of the probabilistic method used and discuss GPU implementation details. We discuss several experimental test cases, including one based on real data from the scientific domain motivating this work, and present timing results from these experiments. We compare to three other GPU selection methods, including a naïve Thrust-based selection-via-sort [Hoberock and Bell 2010], an explicit construction of the *k*th element [Govindaraju 2004], and a reduction via minimization of a convex set [Beliakov 2011], and find that the method discussed in this paper currently gives the best results.

Contributions include:
1) Speedups for larger list sizes in the 1.5-3x range on the best known GPU selection, and in the 3-6x range on the selection-via-sort using Thrust radix sort.
2) Selection from lists up to 4x longer than those obtained using the Thrust sort. These approach the limits of GPU global memory in storing the original list and the *k* keys and values.
3) An in-depth analysis of the increase in timing in *k* on the GPU.
4) A new method of calculating pivots in a randomized selection.

2 Motivation

The initial motivation for this work was the desire to implement the CLEAN algorithm [Hogbom 1974] on the GPU. The CLEAN algorithm is used in radio-astronomy to remove noise from images generated from multiple antennas, as shown in Figure 1. A time-saving variant of CLEAN [Clark 1980] chooses the *k* brightest pixels in the image, convolves the *k* pixels with the point-spread function via a Fast Fourier Transform (FFT), a convolution, and an inverse FFT, and subtracts the result from the base image. This process iterates until all pixels in the residual image reach a threshold noise value.

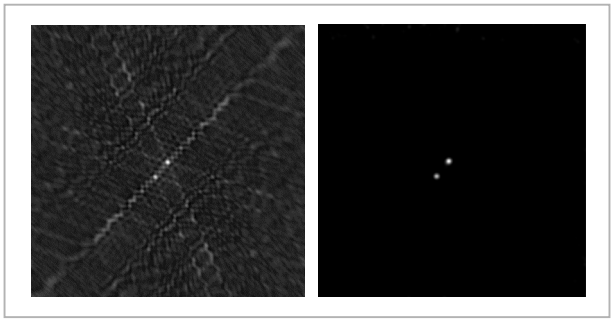

Figure 1. *An un-CLEAN-ed and CLEAN-ed image from radio-astronomy. © Bill Junor, LANL.*

We considered the GPU for implementation of CLEAN as there are fast FFTs on the GPU, and because a GPU-based system has the high FLOPS/watt ratio useful for remote compute-intensive radio-astronomy applications. Our goal was to implement a top *k* selection calculation on the GPU so that the entire CLEAN calculation stays on the GPU, avoiding PCIe transfers.

Selection is a much more general problem than its use in CLEAN,

and is applicable to a wide range of statistical and database-processing problems. For this reason, we present this selection algorithm as an independent result.

3 Prior Work

Selection of the top k elements in a list is one of the classic computer science algorithms [Knuth 1997; Cormen et al 1990], with versions that have worst-case linear time [Blum et al 1973]. Parallel selections have been implemented on general compute platforms, with an optimal parallel selection in [Akl 1984].

A parallel Quickselect was presented in [Blelloch 1996] that relies on recursive partitioning. The algorithm presented here also can be said to rely on recursive partitioning. However, the algorithm here converges far more rapidly than that presented by Blelloch: in Quickselect, the recursive partition takes place on a set having expected size around half the length of the original list, whereas here, the recursion takes place on a set of order square root of the length of the original list.

Motwani and Raghavan presented a serial probabilistic selection called the Lazy Select in [Motwani and Raghavan 1995]. Our algorithm improves on the Lazy Select by selecting pivots depending on the value of k, often selecting a narrower interval between pivots than the Lazy Select does, and thereby reducing the time of the final selection. Our algorithm is also guaranteed to terminate and is parallel in implementation.

Bader [2004] implemented a fast parallel probabilistic selection similar to the Lazy Select on a non-GPU cluster, improving on the method of [Rajasekaran and Reif 1993]. Our GPU selection is in the same spirit as this method; however, our probability calculation differs from theirs. Also, on the GPU, one must address a significant timing increase as k increases. We discuss this in this paper.

Govindaraju [2004] et al presented a kth element selection on the GPU, which is deterministic and proceeds by explicit construction of the kth key. The top k keys (and values, if desired) may then be extracted in one pass over the original list. We have implemented this on a modern GPU using Thrust library calls, but extract only the keys. We find that our keys-and-values selection is several times faster than our implementation of this keys-only selection.

Beliakov [2011] presents a new GPU selection algorithm on ArXiv that proceeds via minimization of a convex function. This method is similar to that discussed in this paper since the selection is reduced to a smaller selection on a set defined by two pivots, but is very different in that the pivots are selected by optimization theory rather than probabilistically. This method selects the kth key, but no other keys or values. We compare to the results presented by Beliakov and find that the randomized method is faster.

One can always implement selection by sorting the keys and values of the list and grabbing the top k keys and values in order, using existing efficient sorts. We compare to this method as well, using the fast Thrust radix sort to perform the sort.

The selection shown here is a Las Vegas algorithm [Babai 1979; Motwani and Raghavan 1995; Motwani and Raghavan 1996]. A Las Vegas algorithm is one of a class of probabilistic computation algorithms [De Leeuw et al 1955] in which the methods are probabilistic, but the result is always correct. This type of algorithm can be used for stochastic optimization, as in this case, where it is used to optimize time spent and memory used.

4 The GPU Programming Model

The GPU is a massively parallel SIMD architecture, having a number of processors, each with many cores, and a hierarchical memory structure. Each processor has a small amount of register space, accessible to individual threads within the processor, and a small amount of fast shared memory available to threads and blocks executing on that processor. There is a moderate amount of slower global memory on the GPU available to all threads and blocks. There is finally limited bandwidth through the PCIe bus between the GPU and the main memory on the node.

We use NVIDIA's CUDA architecture and programming model [NVIDIA 2010] to implement this algorithm. CUDA programs are organized into kernels. Each kernel is executed by individual threads, which run in parallel on the cores of the GPU. The threads are organized into blocks, each of which is executed within a single processor. CUDA provides a mechanism called coalescing to hide latency for data moves to and from GPU global memory, and optimizes moves to and from main memory across the PCIe bus through use of pinned memory.

Six aspects of GPU architecture that especially impact implementation of this selection are the following:

1) *The parallelism available on the GPU*. Our largest runs used up to 64 million threads, delivering good thread-level parallelism, a major advantage of GPU architecture.

2) *SIMD and conditionals*. Partitioning the list forces the use of conditionals that can lead to divergent threads because of the GPU SIMD model of computation. The super-scalar technique [Sanders and Winkel 2004] can alleviate this divergence. However, we find that this takes slightly longer on the two-level tree generated.

3) The small amount of fast shared memory near the processors. The more fast shared memory a block needs, the fewer blocks can be resident on the processor, and the slower the kernel.

4) The increased bandwidth (and speed) gained using coalesced rather than uncoalesced writes to global memory. This gives an incentive to save all of the top k keys and values in shared memory before global memory writes, but requires more shared memory use. Thus, 3) and 4) represent some trade-off.

5) The latency from the main memory to the GPU across the PCIe bus. This adds to the total time to do the select, so if the overall calculation is on the GPU, it is better to select there as well.

6) The limited amount of global memory on the GPU. This limits the overall size of the lists that can be processed at once.

5 The Randomized Selection

5.1 Terminology

A *key-value pair* is defined as per usual for an associative array: a key is the data upon which the selection is performed, and the value is a pointer to that data.

In this implementation of selection, we identify "top k" with "k minimal". However, the "top k" concept can be trivially generalized to encompass any comparison that imposes a strict weak ordering on the set of keys, for example, "k maximal".

5.2 High-level Presentation of the Algorithm

This is a three-part algorithm, composed of: 1) a probabilistic choice of two *pivots* used for a partition, 2) the partition itself, and 3) a reduced selection on the smallest part of the partition and the extraction of the top k elements. The original list is left unchanged by this algorithm.

The pivot choice. We consider a set of randomly chosen keys (called *splitters*) from the original list. We sort the splitters and using probabilistic calculations choose two of these to serve as pivots for a partition of the list into three *bins*. The pivots are chosen with the goals that that the middle bin will contain the kth key with a desired probability p_k, and that the middle bin will also be relatively small in size.

The partition. Each key of the list is queried. Counts are kept of the keys that would fall into each bin, and after the query, the counts are prefix-summed. This suffices to show if the kth has fallen into the middle bin. If not, two more pivots are chosen, and the process iterates until the middle bin contains the kth. At that point, a second pass is made on the list to actually do the partition.

The reduced selection. This is done on the middle bin, which should be small in size. The selected part of the middle bin, along with the entire first bin, constitutes the top k keys and values.

The idea of this algorithm is to reduce the heavy processing of the selection problem to the much smaller middle bin. We examine each element, as must happen in any selection, but then do no more processing except upon those elements in the middle bin.

5.3 Input parameters

Three parameters are input into the algorithm: 1) n, the list length, 2) k, the number of elements to be selected, and 3) p_k, the desired probability that the middle bin contain the kth key.

The length of the original list n is limited by the amount of global memory available. We implement here for n a power of two, and attain the highest such power-of-two n that will fit into GPU global memory (allowing for smaller helper matrices).

The parameter k varies between 1 and n. The larger the quantile k/n is, the longer the time is that it takes to select the elements.

A larger probability p_k of success implies fewer repetitions but a longer final selection, as the middle bin likely has more keys. We chose p_k empirically, and found that runs with p_k >40% produced similar timings, with performance degradation for p_k <30%.

The number of splitters, *numSplitters,* is internally set. The choice of *numSplitters* determines the timings of the splitter sort and the middle bin select. We chose *numSplitters* so the initial sort and the final select are on lists of similar sizes. We have achieved acceptable overall results in our experiments by setting *numSplitters* to be $8*\sqrt{n}$. As n gets bigger, the value of *numSplitters* matters less, because the sort and select have much less impact proportionally on the overall timing.

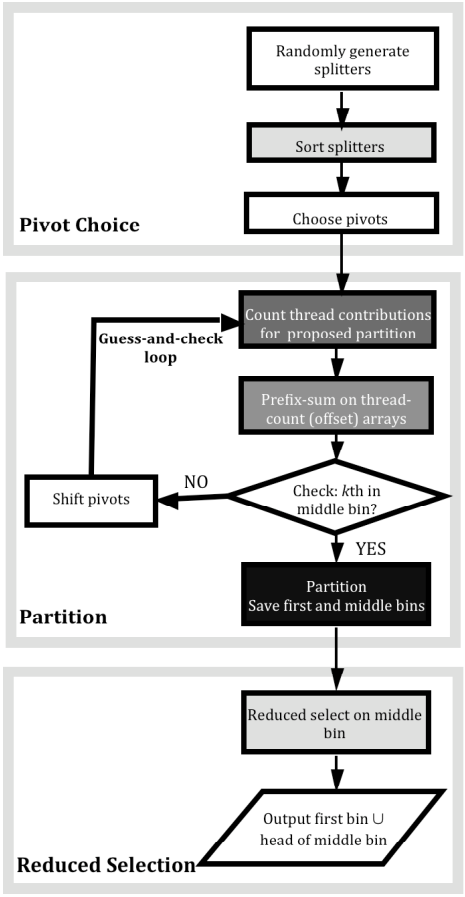

Figure 2. *Flow chart representing the selection algorithm. Those kernels making heavy use of the GPU are shaded, with the darkest the most time-consuming.*

6 Implementation of the Selection

In this section, we present algorithmic details and implementation of some of the functions shown in Figure 2. We implement for 32-bit integer keys, and extract both keys and values, and make an effort throughout to spare global memory, in order to select the highest possible n and k. The CPU is used only for the overall algorithmic control structure (necessarily), and for the generation of bucket probabilities for pivot choice (for convenience). All other computation is on the GPU.

The bucket probabilities could in principle be calculated on the GPU, but for this project we used GNU Scientific Library (GSL) [GNU 2009] on the CPU, as it is verified and avoids some underflow errors on probability calculations. We used CUDPP [CUDA 2010; Satish et al 2009] for the small internal sorts, random selections and one of the prefix-sums. (Thrust shows slightly worse performance on the sizes of sets sorted internally.) We compiled using CUDA 3.2 [NVIDIA 2010] and *g++*.

6.1 The Pivot Choice

6.1.1 Random Generation and Sort of Splitters

Let the n keys be put into some order $\{a_0, \ldots, a_{n-1}\}$. Randomly select *numSplitters* values from $\{0, \ldots, n\text{-}1\}$. For any randomly

selected integer i, a_i is a splitter. Any efficient GPU sort can be used for the sort.

6.1.2 Probabilistic Choice of Pivots

We discuss in this section the procedural method for choosing the two pivots. Justification of this method is presented in Section 7.1. A flow chart representing this calculation is shown in Figure 3. The entire probabilistic choice calculation described here is determined by n, k and *numSplitters*, and is done on the CPU, with only *bucketRange* sent across the PCIe bus to the GPU.

We use the sorted splitters to conceptually partition the list, thus "flattening out" the distribution. This partition produces *buckets* that typically contain roughly the same number of keys (when there is not an extreme amount of repetition of keys), so one can reason probabilistically about the number of elements in each. We use this conceptual structure to estimate the probability of the kth key falling into a given concatenation of buckets.

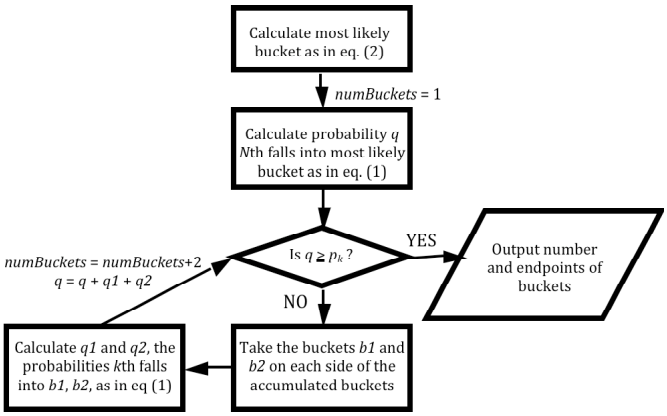

Figure 3. *Flow chart representing the pivot calculation.*

Let $\{s_0, s_1, \ldots, s_{numSplitters-1}\}$ be the list of sorted splitters and consider an element from the original list having key x. Define a conceptual set of buckets as follows:
If $x < s_0$, x would fall into the 0th bucket.
If $s_{i-1} \le x < s_i$, x would fall into the ith bucket.
If $x > s_{numSplitters-1}$, x would fall into the *numSplitters* bucket.

Given the specified desired probability p_k, we calculate a range of contiguous buckets into which the kth key would fall with at least probability p_k and choose the two splitters defining this range of buckets as pivots for the three-bin histogram. To obtain the probability that a set of buckets holds the kth element, we add the probabilities associated with each of the buckets in the set. We can do this since the buckets are non-overlapping, and so the probabilities are independent. The probability that the ith bucket holds the kth value is

$$C(numSplitters, i)\, p^i\, (1-p)^{numSplitters-i} \qquad (1)$$

where $C(numSplitters, i)$ denotes the number of combinations of *numSplitters* items taken i at a time and $p = k/n$.

We start with the bucket most likely to hold the kth key, which is

$$mostLikelyBucket = k/(\, n/(\, numSplitters+1)) \qquad (2)$$

since the number of keys in a bucket defined by the splitters has expected value $n/(numSplitters+1)$. We then calculate the probability that the kth key is in that bucket as in (1). We check to see if that probability is greater than the desired probability p_k, and if so, we are done. Otherwise, we include one bucket on either side of the bucket already chosen, add the probabilities associated with these new buckets to the original probability, and test to see if the cumulative probability is at least the desired probability p_k. We iterate in this way until we have a set of $2*bucketRange+1$ buckets for which the cumulative probability is at least p_k. Formally, the cumulative probability is

$$\sum_{i=mostLikelyBucket-bucketRange}^{mostLikelyBucket+bucketRange} C(numSplitters, i)\, p^i\, (1-p)^{numSplitters-i} \qquad (3)$$

where *bucketRange* is the number of iterations needed to reach the desired probability. The two endpoints of the consolidated set of numBuckets = 2*bucketRange+1 buckets are identified as the pivots for the partition. Let these be denoted as Pivot0 and Pivot1. Then

$$Pivot_0 = s_{mostLikelyBucket-bucketRange-1}$$
$$Pivot_1 = s_{mostLikelyBucket+bucketRange}$$

6.2 The Partition

This section describes the GPU implementation of the kernels that handle the partition. The partition executes two passes through the list, and partitions based on the pivots chosen, as described in the pseudocode in Figure 4. The first pass collects thread offsets and calculates whether the kth element is in the proposed middle bin. The second pass actually does the partition on keys and values, and saves the first and middle bins.

This set of kernels accounts for the linearity of this algorithm in n. All other kernels are sub-linear. The partition-and-save kernel accounts for the increase in k, because of the writes to global memory needed when saving the bins of the partition.

The partition kernels are tightly coupled as all use the same kernel parameters *gridDim* and *blockDim*, and generate and use a common set of offset arrays whose dimensions correspond to the kernel parameters, as shown in Figure 4. Non-optimal choices of *gridDim* and *blockDim* can reduce the speed of the kernels (and thus the overall algorithm), and can also reduce the size of the list that can be processed, as they impact the size of offset arrays stored in global memory.

Through experimentation on NVIDIA Quadros 5000 and 6000, we saw the best results when *blockDim* was 128 and *keysPerThread* was 8. On the NVIDIA GTX 285, we saw our best results when *blockDim* was 64 and *keysPerThread* was 8. These parameters are likely to differ on other GPUs, so experimentation is needed to determine the best parameters. Since we processed lists having as many as 2^{29} elements on the Quadro 6000, we had *gridDim* sizes up to $2^{29}/(2^7*2^3)=2^{19}$ blocks and had 2^{26}=64 million threads for each of the partition kernels.

6.2.1 The Offset Calculation Kernels

The list is queried, and the thread-offset arrays are incremented whenever an element from the list will fall into the corresponding bin. The thread-offset arrays are prefix-summed to establish

thread offsets and to obtain a block count. The block-offset arrays are also prefix-summed, to obtain block offsets.

```
int threadOffset [2][gridDim][blockDim];
int blockOffset [2][gridDim];

while (! kth_In_Middle_Bin)
    QueryCount <gridDim, blockDim >
        (list, pivots, threadOffset);
    OffsetCalc(threadOffsets, blockOffsets);
    Check_If_kth_In_Middle;
    if (! kth_In_Middle_Bin)
      Shift_Pivots( pivots );
Partition <gridDim, blockDim > ( list, pivots, threadOffsets,
blockOffsets, First, Middle);

kernel QueryCount(list, pivots, threadOffsets)
Begin
    for ( key in list )
      if ( key < pivots[0] )
        threadOffsets[0][blockId][threadId]++;
      else if ( key ≤ pivots[1] )
        threadOffsets[1][ blockId][ threadId]++;
End

OffsetCalc(threadOffsets, blockOffsets)
Begin
    Prefix_Sum <gridDim, blockDim > ( threadOffsets[0] );
    Prefix_Sum <gridDim, blockDim > (threadOffsets[1]);
    blockOffsets[0][blockId] = threadOffsets[0][
blockId][blockDim-1];
    blockOffsets[1][ blockId] = threadOffsets[1][
blockId][blockDim-1];
End

Shift_Pivots( pivots, numBuckets, splitters, endpoint)
Begin
    if kth_Is_In_First_Bin
      endpoint [1] = endpoint[0] ;
      endpoint [0] = endpoint[0] – numBuckets ;
    if kth_Is_In_Last_Bin
      endpoint[0] = endpoint[1] ;
      endpoint[1] = endpoint[1] + numBuckets ;
    pivots [0] = splitters [endpoint[0]] ;
    pivots [1] = splitters [endpoint[1]] ;
End

kernel Partition(list, pivots, threadOffsets, blockOffsets,
First, Middle)
Begin
    for ( key in list )
      if ( key < pivots[0] )
        Put_Key_In_First_Bin;
      else if ( key ≤ pivots[1] )
        Put_Key_In_Middle_Bin;
End
```

Figure 4. *Pseudocode representing the partition.*

6.2.2 Check: Is *k*th in Middle Bin?

This is the decision point of the guess-and-check loop shown in Figure 2. The middle bin contains the *k*th when 1) the count of the first bin is less than *k* and 2) the added count of the first and middle bins is greater than or equal to *k*.

6.2.3 Shift Pivots

If the pivot choice was unsuccessful, we choose new pivots by shifting in the appropriate direction over the original set of splitters, holding the number of buckets constant, as shown in Figure 4. Since the pivots are shifted in one direction by a constant number of buckets over a finite number of buckets, the algorithm must terminate.

6.2.4 Partition-and-Save

At this point, a pair of pivots has been chosen so the *k*th element lies in the middle bin. We now partition the list in parallel using the pivots and the offset arrays, and save the first and middle bins.

As *k* gets larger, more queries succeed and more keys and values are written, so the time of this kernel increases as well. Since the query on list elements succeeds with frequency dependent on the distribution of the list, writes to memory occur with unpredictable frequency, so this kernel can share performance characteristics with graph-type algorithms (which perform less well on GPUs), especially if writes are done directly to slower global memory.

One can write to global memory efficiently using coalesced writes, by saving elements to shared memory and transferring them in contiguous chunks to contiguous addresses in global memory. However, using the small shared memory to hold saved elements increases the per-block use of that memory, which reduces the blocks resident per processor, and so reduces the overall speed of the kernel. The small amount of fast shared memory on current GPUs thus forces a choice between using inefficient uncoalesced writes to global memory or using efficient coalesced writes requiring more shared memory per kernel. We investigated both approaches, and also a hybrid of the two.

6.2.4.1 Coalesced vs. Uncoalesced Writes

For the low-shared-memory, uncoalesced-write approach, we write to global memory each time a key/value pair is identified as falling into the first or middle bins. For the high-shared-memory, coalesced-write approach, we allocate enough shared memory to store all key/value pairs processed by a block (in case the query should succeed for all), then at the end, write only the identified key/value pairs to global memory in a coalesced manner. We also tried a hybrid coalesced write, collecting key/value pairs into small regions of shared memory varying in size, and writing to global when the regions filled up.

Uncoalesced Best for Small *k*, Coalesced Best for Large *k*

For the GPUs we tested, the uncoalesced write gives better results for small k/n, and the coalesced write is better for large k/n. The hybrid write gave the worst performance, as the small areas fill up with unpredictable frequency, forcing interrupts in thread processing across the block to do the write.

Different applications may require consistently small or large k/n (for example, our radio-astronomy application uses relatively small k/n), so optimal performance requires understanding the behavior of both approaches and the crossover point. We analyze this in depth throughout the rest of this section.

Crossover Between Coalesced and Uncoalesced

The exact timings and crossover point are dependent on the shared memory and the write speeds of the GPU and the value of *keysPerThread*, so empirical observation is recommended. The crossover point is the same for all n on a given GPU, if *keysPerThread* is held constant, because both the coalesced and uncoalesced versions of the kernel increase by the same factor when n is increased and the quantile k/n is fixed.

The bandwidth of the writes determines the slope of the timing plots across k. The plot for coalesced writes will have a smaller slope than that of uncoalesced individual writes, because of the efficient use of bandwidth. The additional shared memory needed for the coalesced approach accounts for the difference between coalesced and uncoalesced for small k/n, since the work done for small k is roughly the same for each approach.

Let *slope* be the timing plot slope for coalesced or uncoalesced writes, and let *yint* be the intersection of the projected line with the y-axis, both obtained by taking timings for a pair of non-extreme N. Then the crossover point of the two is at
$x = (yint_{coalesced} - yint_{uncoalesced}) / (slope_{uncoalesced} - slope_{coalesced})$
The uncoalesced *yint* is less than the coalesced as the uncoalesced kernel uses less shared memory. At the same time, the *slope* for coalesced is less than that for uncoalesced. Therefore, x is positive and there must be a crossover.

Representative Observed Timings

Linearity of both coalesced and uncoalesced approaches can be seen in Figure 5. We also show the timings for an uncoalesced write requesting the extra shared memory needed for the coalesced approach. The graph of uncoalesced writes shows almost flat slopes up through k/n to around 0.35, then shows increased slope. We believe this is due to the beneficial cache for uncoalesced writes, as this does not occur on GPUs lacking cache.

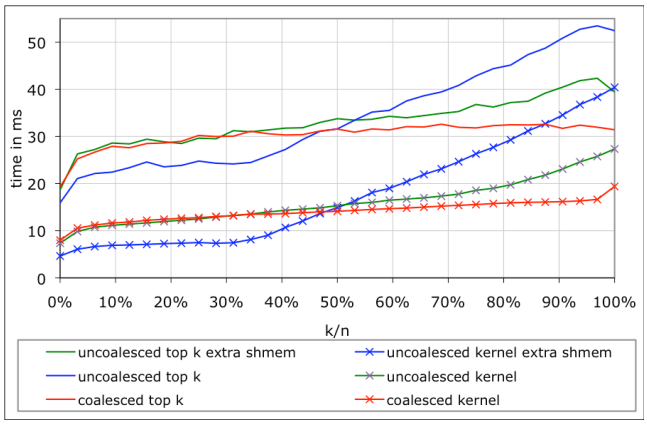

Figure 5. *Timings on the Quadro 6000 for the partition-and-save kernel and for the algorithm, on a random unsorted length 2^{26} list*

6.3 Reduced Select and Generation of Top *k* List

The kth element is now known to be in the middle bin, all elements of the first bin are known to be of rank less than k, and elements that do not fall into the first or middle bins are known to be of rank greater than k. Let the first bin contain M elements. It is sufficient to select for the top k-M elements in the middle bin, using any selection. For simplicity, here we use a CUDPP sort.

7 Justification of Probabilistic Guess of Pivots

We derive the probability that the kth key falls into a given bucket, as in (1), and then show that the probability for the buckets may simply be added, as in (3).

7.1 Probability *k*th Key Falls Into *i*th Bucket

To determine the probability that the kth value is in a specific bucket or set of buckets, we make use of probabilistic analysis using a binomial approximation.

We define q_p to be the pth quantile of the keys, i.e., the value of q_p is chosen such that $Prob\ (x \le q_p) = p$, for p in [0,1] for any randomly sampled key x. Therefore, the probability that any one of the randomly sampled splitters described above is less than or equal to q_p is p. To obtain a bucket that contains the kth value, we choose $p=k/n$. Then the probability that the kth key lies in the ith bucket is equal to the probability that q_p is in the ith bucket. For q_p to fall in the ith bucket, we need i of the splitters to be less than or equal to q_p and *numSplitters*-i of the splitters to be greater than q_p. Using the sorted splitters $\{s_0, s_1, \ldots, s_{numSplitters-1}\}$, this occurs when $s_{i-1} \le q_p < s_i$.

The probability that i of the sorted splitters are less than or equal to q_p may be approximated by the probability that i of the randomly drawn unsorted splitters are less than or equal to q_p and is given by $C\ (numSplitters,\ i)\ p^i\ (1\text{-}p)^{numSplitters-i}$, for $i = 0,\ldots, numSplitters$, where the notation $C(a, b)$ denotes the number of combinations of a items taken b at a time. This is simply the commonly used expression arising from the Binomial distribution for i successes in *numSplitters* events, where p is the probability of a success and $(1\text{-}p)$ is the probability of a failure. See for example, [Hogg and Craig 1978]. If the kth key lies outside the range of the splitters, the approximation may be poor for the end buckets, but the impact will be minimal if the tail probabilities associated with the end buckets are small.

7.2 Addition of Probabilities

Because the buckets are constructed in a non-overlapping manner, q_p can only lie within a single bucket, and the probability of lying within a range of buckets may be obtained by adding the probabilities of lying within each of the individual buckets in the range as indicated in (3). Note that this result is consistent with a formula in [David 1981] for the probability that a quantile of a continuous distribution is in the interval between two specified ranked samples.

7.3 Concatenation of Buckets

The number of preliminary buckets in the range is calculated by adding a series of probabilities for individual buckets. First, the probability that the kth key is actually in the most likely preliminary bucket is calculated. If that probability is greater than or equal to p_k, then we are done. Otherwise, the probability that the kth key falls into the preliminary buckets on either side of the most likely bucket is calculated, and that probability is added to the first probability. This process iterates until the cumulative probability is greater than or equal to p_k.

Since the number of preliminary buckets is determined before running the Top k probabilistic selection algorithm, p_k is unlikely

to be matched exactly, and is therefore taken as a lower bound. Because this process is probabilistic, for some percentage (approximately 1- p_k) of these runs, the pivot-pair guesses will fail, i.e. the middle bin will not contain the kth value.

Clearly, for any probability p_k, there is some range of preliminary buckets that will satisfy or exceed it. For example, the entire range of buckets contains the kth key with probability 1. However, the larger the probability p_k, the more preliminary buckets are consolidated into the middle bin and the longer the middle bin sort will take. This is an optimization problem in the implementation of the algorithm and depends on the sort performance and the performance of the algorithm as a whole.

8 Algorithmic Analysis

8.1 Linearity in Terms of *n*

Reads and writes dominate the selection: the algorithm essentially reads n keys, does a (short) query, and writes k keys and values. We hold k/n fixed and discuss algorithmic complexity in terms of n on a kernel-by-kernel basis, and show linearity or sub-linearity of all kernels.

8.1.1 Linear Kernels (Partition Kernels)

The query for offsets is linear in n. It performs n reads, n queries, k writes to shared memory and $2*gridDim*blockDim = 2*n/keysPerThread$ coalesced writes to global memory. Since k increases with n when k/n is held fixed, writes to shared memory are linear in n, and since *keysPerThread* is a constant, the writes to global memory are as well.

The prefix-sum of the thread-offset arrays is linear in n. The array is of dimension *gridDim* x *blockDim*. The number of rows $gridDim = n/(blockDim*keysPerThread)$ is linear in n, since *blockDim* and *keysPerThread* are constant, and each of the *gridDim* parallel scan-adds on the constant-length *blockDim* rows are of constant time. The scan-add of the block-offset arrays is also linear in n [Sengupta et al 2008].

The partition-and-save is linear in n. Let the quantile represented by k/n be fixed, and consider the kernel run on input with length $a*n$. Since the read and query take place on $a*n$, they take a times as long as when run on input of length n. The number of elements selected is $a*k$, so the query succeeds a times as often, which means the writes take a times as long. This implies that this kernel is linear in n, when k/n is constant.

8.1.2 Sub-linear Kernels

The generation of splitters randomly generates *numSplitters* values and extracts the associated keys from the list. The number of splitters is a multiple of $\sqrt{n}$, so this kernel is $O(\sqrt{n})$. The sort on splitters uses the CUDPP sort. This is a radix sort, so is linear in terms of the list sorted [Satish et al 2009]. Since the number of splitters is a multiple of $\sqrt{n}$, the splitter sort will be order $O((\sqrt{n})$.

The selection on the middle bin again uses the CUDPP sort. We have observed the number of buckets, *numBuckets*, to be approximately $O(n^{1/4})$, and the expected value of the middle bin is a multiple of $(\sqrt{n})*numBuckets$, assuming the repetition of keys in the middle bin is not large. In this case, the sort on the middle bin is $O(\sqrt{n}*numBuckets) = O(\sqrt{n})*O(numBuckets) = O(n^{3/4})$. The pivot guess performs the calculation in (1) for each of the buckets to be concatenated. As per observation, the growth in the number of buckets is approximately $O(n^{1/4})$. The check and the pivot shift are constant in n.

There is impact on performance when there is a large amount of repetition of keys that have order slightly greater than or equal to k. In that case, the middle bin is likely to contain all of these keys, so the sort on this large middle bin dominates the selection.

8.2 Linearity in Terms of *k/n*

Only the performance of the partition queries, the pivot guess calculation, and the section on the middle bin are affected by changes in k/n, when n is fixed. Of these, only the partition-and-save kernel time increases significantly as k/n increases.

8.2.1 The Partition-and-Save Kernel

This kernel comprises three parts: 1) reading the original list into memory, 2) a query on all n elements of the list, and 3) writes (coalesced or uncoalesced) when the query succeeds. When n is held constant, the time for 1) the reads and 2) the query is the same for all k/n. The queries succeed in direct proportion to k/n. Since the writes are serial (whether individually for uncoalesced writes or in chunks for coalesced writes), both approaches are linear in k. This is true for any GPU and is not empirical.

8.2.2 Other Kernels that Vary Over *k/n*

The query on offsets changes over k/n only because of shared memory writes. These add only slightly to overall time, so this query does not add much to the timing across k/n. The pivot calculation and the sort on the middle bins depend entirely on the number of buckets consolidated, and because these are bounded above at $k/n = 50\%$, these are sub-linear.

8.3 Space Analysis of the Algorithm

Throughout the algorithm, the original list is retained in its unsorted state. Two arrays of size k are saved to store the keys and values of the top k elements. These arrays are required for any non-streaming selection algorithm, as are thread- and block-offset arrays for a parallel selection.

With the entire selection on the GPU, the GPU's global memory imposes a limit on space available as follows:
$sizeof(int) * (n + 2*k + 2*n/keysPerThread + 4*n/keysPerBlock + c*\sqrt{n} + 5)$
This accounts for the arrays holding the original list, the top k keys and values arrays, 2 thread-offset arrays, 4 block-offset arrays (non-scan-added and scan-added) and a few helper arrays. Note that *keysPerThread* and *keysPerBlock* are free parameters and may be sent as needed, By increasing these (and thus increasing work done per thread), it is sometimes possible to run on larger n and k.

8.3.1 Experimental Results on Maximal List Size

This selection processed twice the maximum list size (with no restriction on k) than the Thrust selection-via-sort on the Quadro 5000 and GTX 285, as in Figure 6. This selection also processed

even larger n, when k was restricted. For such extreme lists, we could select even larger k by increasing *keysPerThread*, at cost of slower performance. We required the randomized selection to leave the original list unchanged to meet the needs of the radio-astronomy example, so for the purposes of this experiment, we imposed the same requirement on the Thrust selection-via-sort.

GPU	*global memory*	*max n*	*max k/n*	*max n, no k restriction*	*thrust max n*
6000	6 GB	2^{29}	0.34	2^{28}	2^{28}
5000	2.5 GB	2^{28}	0.53	2^{27}	2^{26}
285	1 GB	2^{27}	0.13	2^{26}	2^{25}

Figure 6. *Maximum n with attained quantile k/n in shaded area, and maximum n with no restriction on k (i.e., works for all k).*

9 Experimental Study

9.1 Algorithms Compared and Data Analyzed

We compared four algorithms, and show results in Figure 8:
1) The randomized selection algorithm. These studies show typical execution of the algorithm as presented. This algorithm collects all k keys and values.
2) A selection-via-sort. The list keys and values are sorted using Thrust sort, and the top k keys and values are identified.
3) A direct construction of the kth element [Govindaraju 2004]. The kth key is built up explicitly over log(n) passes over the list. Our implementation of this method collects all k keys but no values. This means that timings shown underestimate the time it would take to complete all the work done by our algorithm.
4) A construction by minimization of a convex function [Beliakov 2011]. We compare to Beliakov's results on floats. These were run on a Tesla C2050, which is computationally equivalent to the Quadro 6000 we used. This method collects only the kth key, and no values, so timings shown underestimate the time needed to duplicate the randomized top k.

We experimented on the following list data:
1) Lists of randomly generated 32-bit integer keys, of length from 2^{18} to 2^{29} elements, unsorted, sorted and sorted backwards.
2) A real 32-bit image from radio-astronomy, of size 2^{18} pixels.

We examined scaling by varying the parameters n and k/n (using this quantile of keys selected permits comparisons between different values of n). Except on the probability study, we calculated using a requested probability p_s of 0.80. We ran in sets of 100 runs and averaged the results to smooth out occasional jitter for a representative result. The timings shown use the hybrid uncoalesced/coalesced approach to the partition-and-save kernel. All lists are unsorted, unless otherwise stated.

9.2 Platforms Used

We tested on the following NVIDIA GPUs.
1) The Quadro 5000 has 2.5 GB of global memory, 11 multiprocessors with 32 streaming cores each for a total of 352 cores, 64 K of shared memory/cache per multiprocessor, and 120 GB/s memory bandwidth.
2) The Quadro 6000 has 6 GB of global memory, 14 multiprocessors with 32 streaming cores each for a total of 448 cores, 64 K of shared memory/cache per multiprocessor, and 144 GB/s memory bandwidth.
3) The GeForce GTX 285 has 1 GB of global memory, 30 multiprocessors with 8 streaming cores each for a total of 240 cores, 64 K of shared memory per multiprocessor, and 159 GB/s memory bandwidth.
We show results from runs on the Quadro 6000, except for a comparison of performance of three cards, in Figures 13 and 14.

9.3 Results

Linearity in n and n/k is shown in Figures 7 and 8. Timings increase as k/n increases, as shown in Figure 7, for an overall increase in timing of around 40-60% (not including the outlier k=1 case). Figure 8 shows a linear increase in timings with respect to n for each k/n value.

Figure 8 also shows a comparison to the Thrust selection-via-sort method and also to our implementation of Govindaraju's method [2004] and to Beliakov's reported timings [2011]. Probably the most accurate comparison between Beliakov's method and the randomized Top k would be when the randomized algorithm is run for k/n approximately 0%, since in that case, exactly one element is extracted. In that case, we show a 1.5 to 3 times improvement over the timings reported by Beliakov.

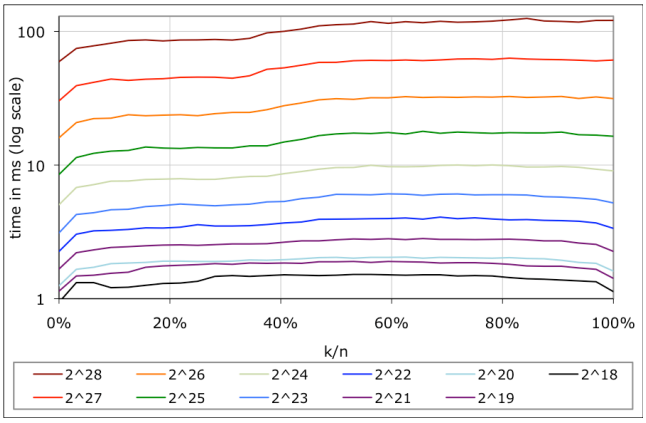

Figure 7. *Timings for different n. Each line represents a different value of n.*

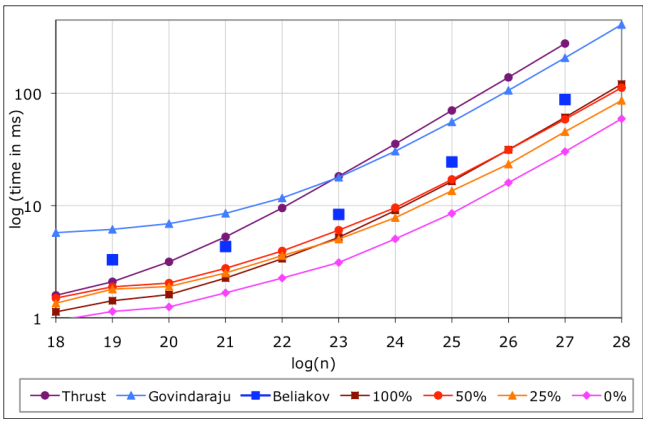

Figure 8. *Timings for different k/n. Each line marked by a percentage represents a different value for k/n. These are compared to three other methods of GPU selection.*

Figure 9 shows a 4-6x speedup over Thrust-based selection-via-sort, for larger n, with speedups as high as 9x. Figure 10 shows an overall kernel breakdown for $n = 2^{26}$, representative of larger n. The partition-and-write dominates as k/n gets larger.

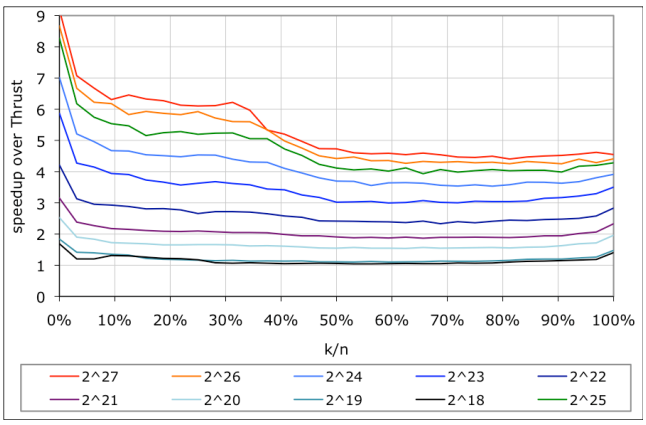

Figure 9. *Speedup over Thrust selection-via-sort, for different n.*

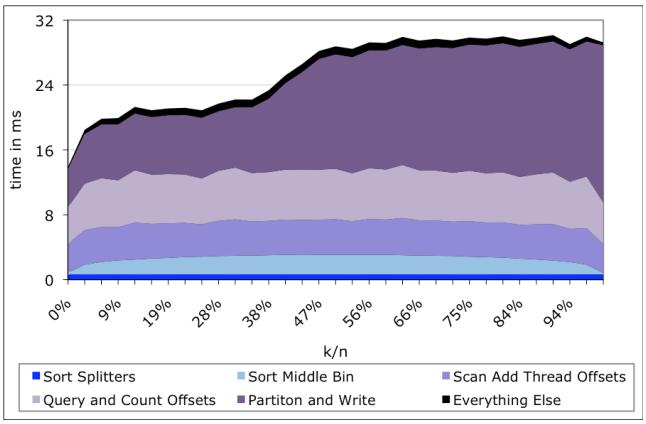

Figure 10. *A kernel timing breakdown, on a dataset of size $n=2^{26}$.*

Figure 11 compares requested success probabilities for a representative $n = 2^{26}$, and shows that the calculations run with probabilities between 40% and 95% track each other closely. Similar observations hold for other n, with the slight caveat that the optimal probability choice increases as n increases, due to the decreasing influence of the middle bin selection for such large n.

Figure 12 compares timings on a real radio-astronomy dataset to a random test dataset of the same size and to the Thrust selection-via-sort. All randomized top k runs track each other closely.

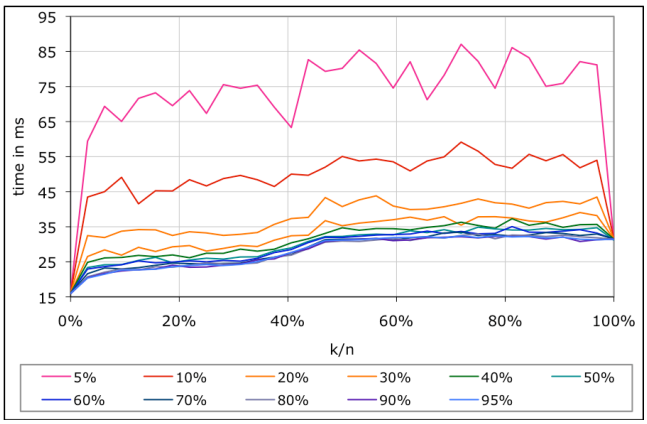

Figure 11. *A comparison of runs requesting different success probabilities, on a dataset of size $n=2^{26}$.*

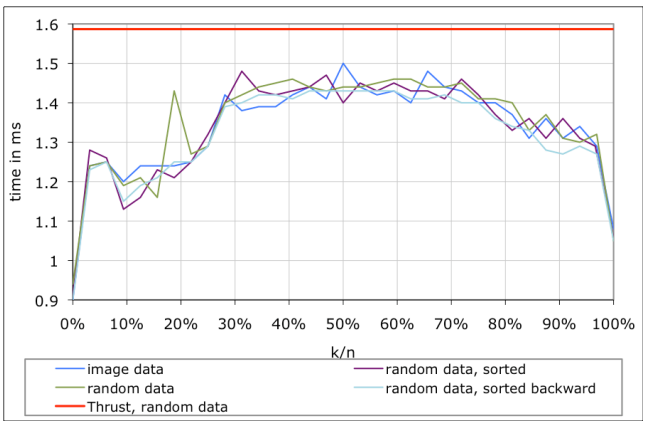

Figure 12. *A comparison of a real radio-astronomy image of size $n=2^{18}$ to random data (unsorted, sorted forward and sorted backward) and to Thrust sort-and-select.*

Figures 13 and 14 show comparisons between the three cards tested in depth: the NVIDIA GTX 285, the NVIDIA Quadro 5000 and the NVIDIA Quadro 6000. We saw performance on the Quadro 6000 around 25-35% faster than on the Quadro 5000, and performance on the Quadro 5000 around 25-35% faster than on the GTX 285.

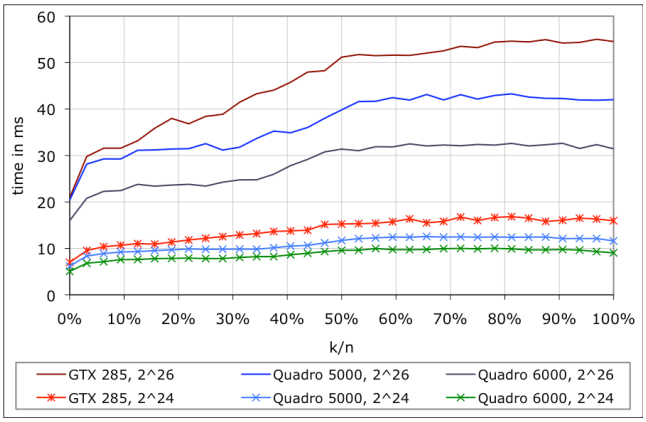

Figure 13. *Timings for the Quadro 6000, Quadro 5000 and GTX 285, for representative $n = 2^{24}$ and 2^{26}.*

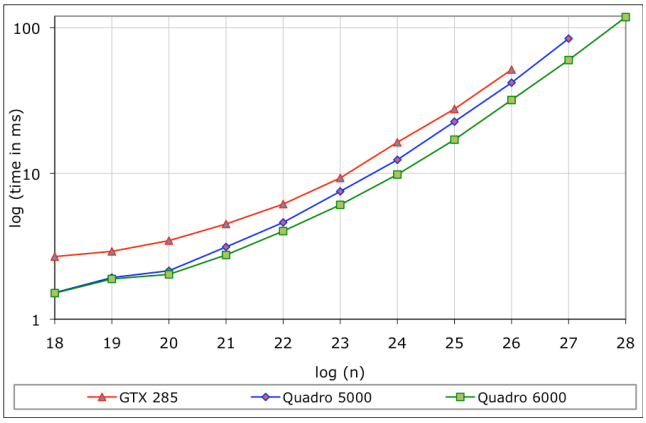

Figure 14. *Timings for the Quadro 6000, Quadro 5000 and GTX 285, for a representative k/n = 0.625.*

10 Future Work

Future work will include an analysis of optimized selection on a modern multicore CPU, and a comparison to that on the GPU. The CPU has the advantage of large fast memory, large cache and no need to transfer large lists across the PCIe bus. The GPU has the advantage of massive parallelism. If other calculations are already on the GPU, it is worthwhile to select on the GPU as well, but it is an open question in the general case which would be most advantageous, and where the crossover (in terms of CPU cores, k, and n) would be. Another interesting investigation is a comparison of GPU radix selection to this randomized selection, in view of the success of radix sorts on the GPU.

Acknowledgements

We thank Mark Harris for the use of his implementation of the selection algorithm discussed in [Govindaraju 2004]. We thank John Bent, Nathan Brown, Nathan DeBardeleben, Andy DuBois, David DuBois, Josip Loncaric, Rick Picard and Scott Vander Wiel for discussion, Bill Junor for presenting the radio-astronomy problem inspiring the work, Dave Modl for computer support, and the reviewers for thoughtful comments. This work was funded by Los Alamos National Laboratory LDRD Project #20080729DR.

References

AKL, S.G. 1984. An Optimal Algorithm for Parallel Selection. *Information Processing Letters,* 19.

BABAI, L. 1979. Monte Carlo Algorithms in Graph Isomorphism Testing, *Tech. Rep. DMS* 79-10, Universite de Montreal.

BADER, D.A. 2004. An Improved Randomized Selection Algorithm With an Experimental Study, *Journal of Parallel and Distributed Computing,* 64(9):1051-1059.

BELIAKOV, G., 2011. Parallel Calculation of the Median and Order Statistics on GPUs with Application to Robust Regression. arXiv:1104.2732v1.

BLELLOCH, G., 1996. Programming Parallel Algorithms. *Communications of the ACM, 39(3).*

BLUM, M., FLOYD R.W., PRATT, V., RIVEST, R., AND TARJAN, R.. 1973. Time Bounds for Selection, *J. Comput. System Sci.* 7 448-461.

CLARK, B.G. 1980. An Efficient Implementation of the Algorithm 'CLEAN'. *Astron. Astrophys.* 89, 377-378.

CORMEN, T.H., LEISERSON, C.E., RIVEST, R.L., AND STEIN, C. 1990. Probabilistic Analysis and Randomized Algorithms, *Introduction to Algorithms*, Second Edition. MIT Press and McGraw-Hill. pp. 91–122.

CUDA Data Parallel Primitives Library. 2010. *http://code.google.com/p/cudpp*

DAVID, H.A. 1981. *Order Statistics*, 2nd Edition, John Wiley & Sons, New York.

DE LEEUW, K., MOORE, E.F., SHANNON, C.E., AND SHAPIRO, N. 1955. Computability by Probabilistic Machines. *Automata Studies*, Princeton University Press, Princeton, NJ. 183-212.

GALASSI, M. ET AL. 2009. GNU Scientific Library Reference Manual, Third Edition.

GNU Scientific Library. 2009. *http://www.gnu.org/software/gsl/.*

GODIYAL, A., HOBEROCK, J., GARLAND, M., AND HART, J. C., 2008. Rapid Multipole Graph Drawing on the GPU. *Proc. Graph Drawing.*

GOVINDARAJU, N., LLOYD, B., WANG, W., LIN, M., AND MANOCHA, D., 2004. Fast Computation of Database Operations Using Graphics Processors. *Proc. of ACM SIGMOD.*

HARRIS, M., SENGUPTA, S., AND OWENS, J.D. 2007. Parallel Prefix Sum (Scan) with CUDA. *GPU Gems 3*, pp. 851–876. Addison Wesley.

HOBEROCK, J. AND BELL, N. 2010. Thrust: a Parallel Template Library, *http://code.google.com/p/thrust/*

HÖGBOM, J.A. 1974. Aperture Synthesis with a Non-Regular Distribution of Interferometric Baselines. *Astron. Astrophys. Supp.* 15, 417-426.

HOGG, R.V. AND CRAIG, A.T. 1978. Introduction to Mathematical Statistics, 4th Edition, Macmillan.

KNUTH, D. 1997. Sorting by Selection. *The Art of Computer Programming, Volume 3: Sorting and Searching*, Third Edition. Addison-Wesley.

MOTWANI, R. AND RAGHAVAN. P. 1995. *Randomized Algorithms*, Cambridge University Press.

MOTWANI, R. AND RAGHAVAN. P. 1996. Randomized Algorithms, *ACM Computing Surveys (CSUR),* v.28 n.1, p.33-37.

NVIDIA CUDA C Programming Guide, version 3.1.1. 2010, *http://developer.nvidia.com/object/gpucomputing.html.*

RAJASEKARAN, S. AND REIF, J.H. 1993. Derivation of Randomized Algorithms for Sorting and Selection. *Parallel Algorithm Derivation And Program Transformation*, Kluwer Academic Publishers.

SANDERS, P. AND WINKEL, S. 2004. Super Scalar Sample Sort. *Proc. European Symposium on Algorithms (ESA)*, volume 3221 of LNCS.

SATISH, N, HARRIS, M. AND GARLAND M. 2009. Designing Efficient Sorting Algorithms for Manycore GPUs. *Proceedings of the 23rd IEEE International Parallel & Distributed Processing Symposium.*

SENGUPTA, S., HARRIS, M. AND GARLAND M. 2008. Efficient parallel scan algorithms for GPUs. *NVIDIA Technical Report NVR-2008-003.*