Randomized Selection on the GPU

Laura Monroe, Joanne Wendelberger, Sarah Michalak
Los Alamos National Laboratory

High Performance Graphics 2011
August 6, 2011

Los Alamos NATIONAL LABORATORY

LA-UR 11-04829

NNSA National Nuclear Security Administration

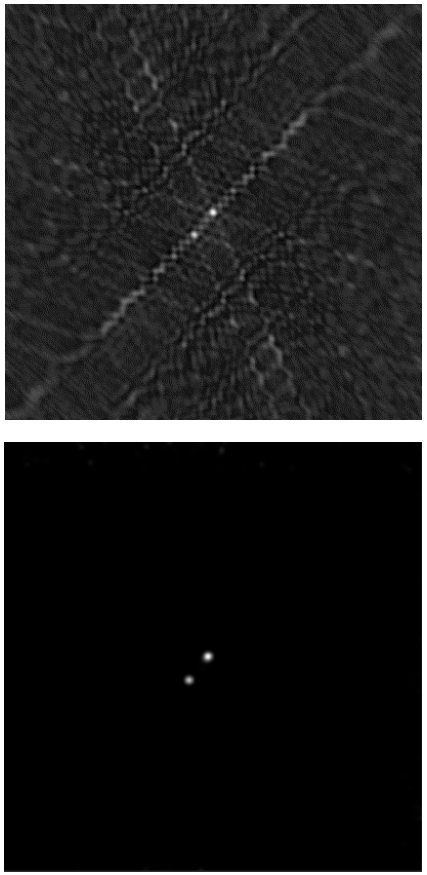
Top *k* Selection on GPU

- Output the top *k* keys and values from an unordered list of length *n*
  - Top *k* is in terms of key ordering
- Our motivating problem is from radio-astronomy

- Contributions of this work
  - Speedups of 1.5-3x over best-known GPU selection, and 3-6x over Thrust sort
  - Selections on lists up to 4x longer than Thrust sort
  - New method of selecting pivots in this randomized select

Motivating problem

- CLEAN on GPU
  - Used in radio-astronomy to remove noise from images generated by multiple antennas
  - Fast CLEAN algorithm (Clark)
    - **Chooses the *k* brightest pixels in the image**, and saves them to a clean image
    - Convolves the *k* pixels with the point-spread function via a Fast Fourier Transform (FFT), a convolution, and an inverse FFT
    - Subtracts the result from the base image to get residual image
    - This process iterates until all pixels in the residual image reach a threshold noise value. The clean image is then accepted.



Images © Bill Junor, LANL

CLEAN Constraints

- Choice of GPU for CLEAN
  - Good FLOPS/watt ratio for remote telescope locations
  - Have fast FFTs on GPU, need fast selection
- Choice of GPU for selection implementation
  - Transfer of big images a performance killer
    - Residual image changes at each of the many iterations in CLEAN
    - If FFTs on GPU but selection on CPU, big image transfer each step
- CLEAN requirements for selection
  - Requires very general version of select
  - Need both keys *and* values
    - Keys (pixel brightness) are what is ordered and selected upon
    - Values (pixel locations) are what is needed for CLEAN
  - Need all *k* keys and values, not enough to grab just *k*th

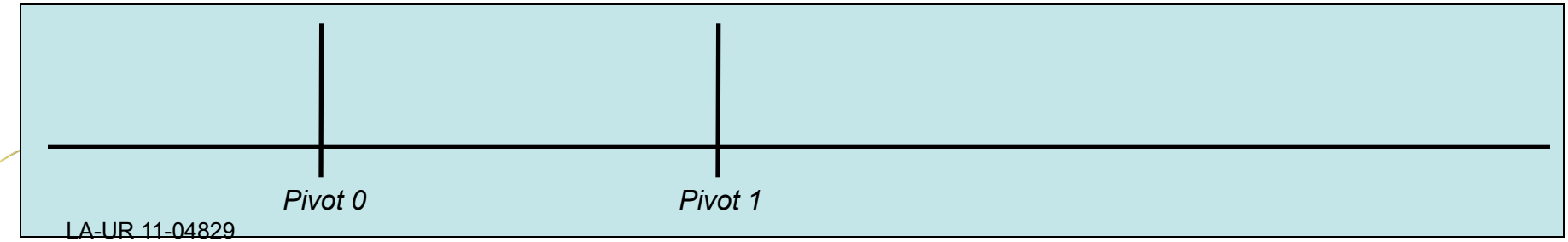
Some Previous Work on Selection

Selection is a more general CS problem.

- Serial Lazy Select (Motwani and Raghavan 1995)
- Serial Quickselect (Bleloch 1996)
- Parallel Randomized Selection (Bader 2004)
- GPU Select via Explicit Construction (Govindaraju 2004)
- GPU Select via Minimization of a Complex Function (Beliakov 2011)
- Select via a Sort (e.g., Thrust)
  - Does more than just the select
  - Good if the list never changes

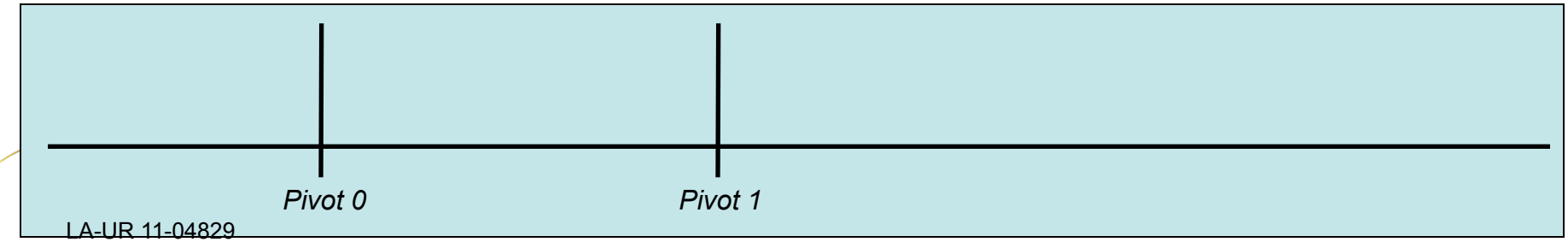
Randomized Selection Algorithm

- Choose two pivots for a partition so that:
  - The *k*th element is contained in the middle bin with probability p_k
  - The middle bin is small relative to the list



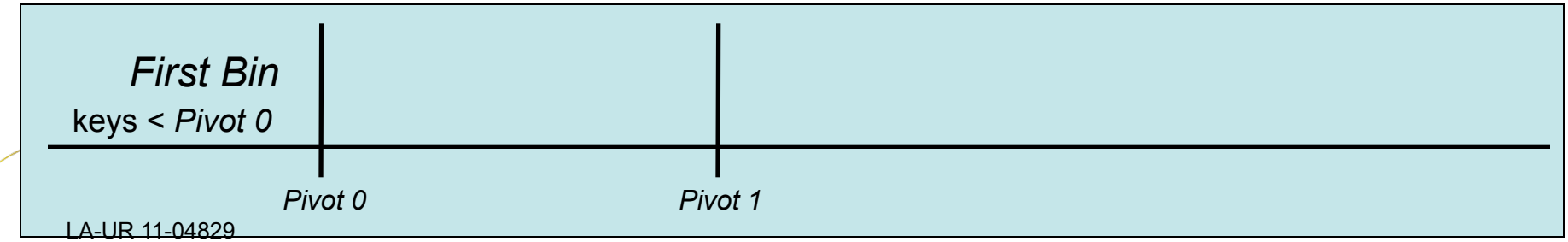
Randomized Selection Algorithm

- Choose two pivots for a partition so that:
  - The *k*th element is contained in the middle bin with probability p_k
  - The middle bin is small relative to the list
- Partition the list



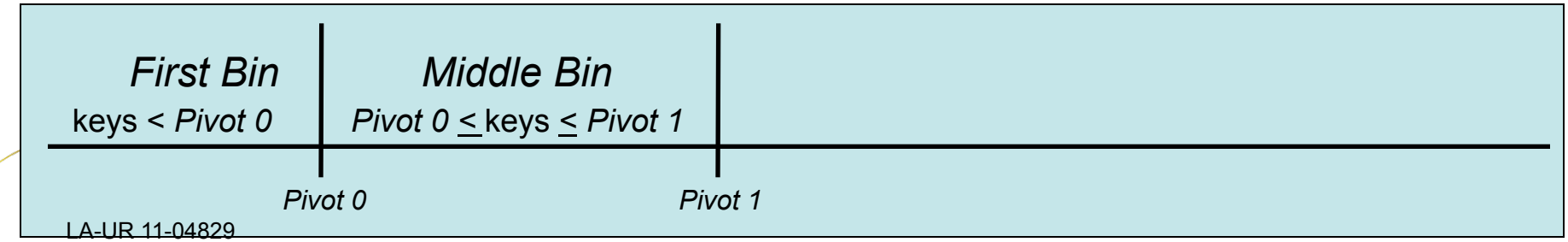
Randomized Selection Algorithm

- Choose two pivots for a partition so that:
  - The *k*th element is contained in the middle bin with probability p_k
  - The middle bin is small relative to the list
- Partition the list
  - First bin holds elements with keys < *Pivot 0*



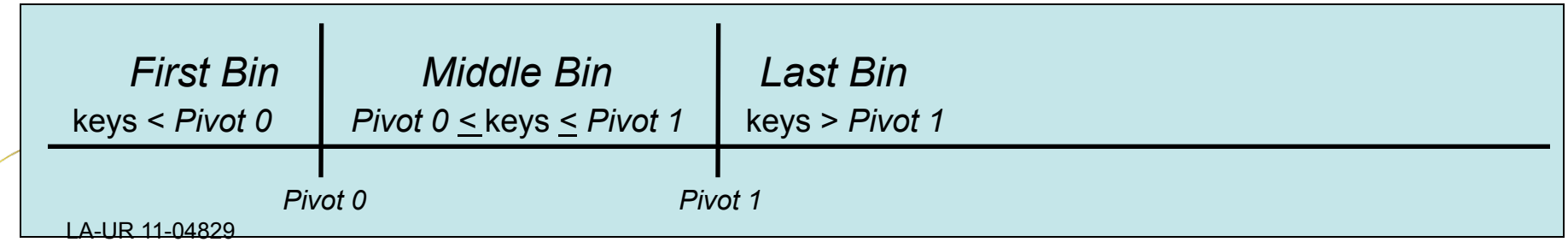
Randomized Selection Algorithm

- Choose two pivots for a partition so that:
  - The *k*th element is contained in the middle bin with probability p_k
  - The middle bin is small relative to the list
- Partition the list
  - First bin holds elements with keys < *Pivot 0*
  - Middle holds elements with keys between *Pivot 0* and *Pivot 1*



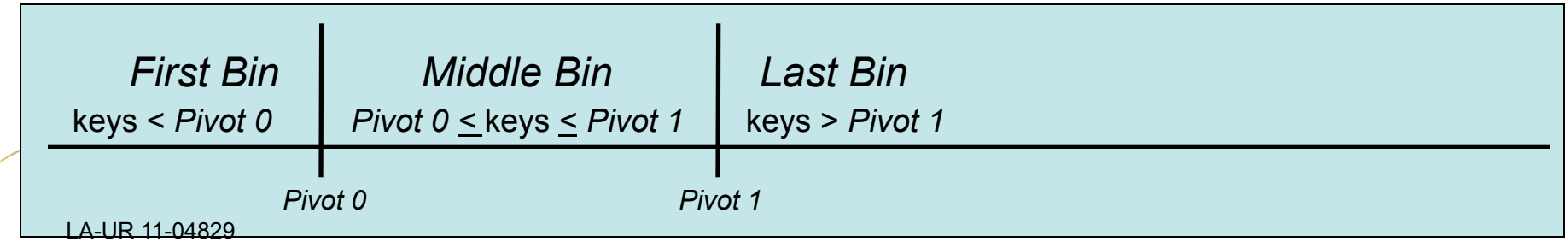
Randomized Selection Algorithm

- Choose two pivots for a partition so that:
  - The *k*th element is contained in the middle bin with probability p_k
  - The middle bin is small relative to the list
- Partition the list
  - First bin holds elements with keys < *Pivot 0*
  - Middle holds elements with keys between *Pivot 0* and *Pivot 1*
  - Last holds elements with keys > *Pivot 1*



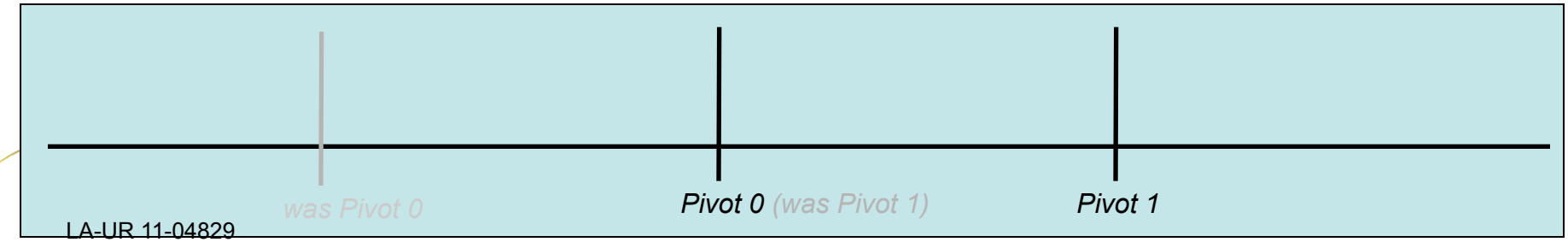
Randomized Selection Algorithm

- Choose two pivots for a partition so that:
  - The *k*th element is contained in the middle bin with probability p_k
  - The middle bin is small relative to the list
- Partition the list
  - First bin holds elements with keys < *Pivot 0*
  - Middle holds elements with keys between *Pivot 0* and *Pivot 1*
  - Last holds elements with keys > *Pivot 1*
- Is *k*th in middle bin?
  - Yes, if number in first bin < *k* and number in first and middle bins $\geq$ *k*



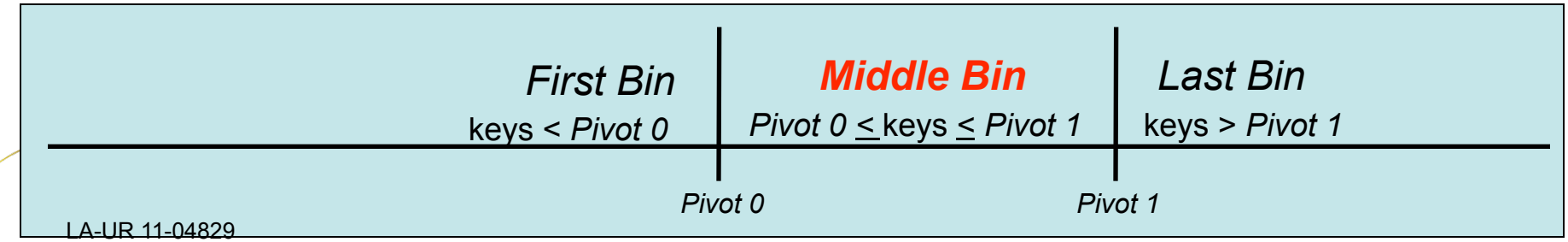
Randomized Selection Algorithm

- Choose two pivots for a partition so that:
  - The *k*th element is contained in the middle bin with probability p_k
  - The middle bin is small relative to the list
- Partition the list
  - First bin holds elements with keys < *Pivot 0*
  - Middle holds elements with keys between *Pivot 0* and *Pivot 1*
  - Last holds elements with keys > *Pivot 1*
- Is *k*th in middle bin?
  - Yes, if number in first bin < *k* and number in first and middle bins > *k*
  - If not, shift pivots in appropriate direction and iterate until it is



Randomized Selection Algorithm

- Choose two pivots for a partition so that:
  - The *k*th element is contained in the middle bin with probability p_k
  - The middle bin is small relative to the list
- Partition the list
  - First bin holds elements with keys < *Pivot 0*
  - Middle holds elements with keys between *Pivot 0* and *Pivot 1*
  - Last holds elements with keys > *Pivot 1*
- Is *k*th in middle bin?
  - Yes, if number in first bin < *k* and number in first and middle bins > *k*
  - If not, shift in appropriate direction and iterate until it is
- Select on the middle bin (we used select-via-sort)



Randomized Selection Algorithm

- Choose two pivots for a partition so that:
  - The *k*th element is contained in the middle bin with probability p_k
  - The middle bin is small relative to the list
- Partition the list
  - First bin holds elements with keys < *Pivot 0*
  - Middle holds elements with keys between *Pivot 0* and *Pivot 1*
  - Last holds elements with keys > *Pivot 1*
- Is *k*th in middle bin?
  - Yes, if number in first bin < *k* and number in first and middle bins > *k*
  - If not, shift in appropriate direction and iterate until it is
- Select on the middle bin (we used select-via-sort)
- Write out the selected part of middle + the first

Algorithm observations

- Essentially a reduced selection
  - Selection is on a smaller middle bin
- Pivot choice is heart of algorithm
- The action is in reads-to/writes-from shared memory
  - The partition takes most of the time because of this
- Everything on the GPU except
  - The overall control
  - The probability sums in the pivot choice
    - For convenience used GSL on CPU (validated, has numerical tricks)

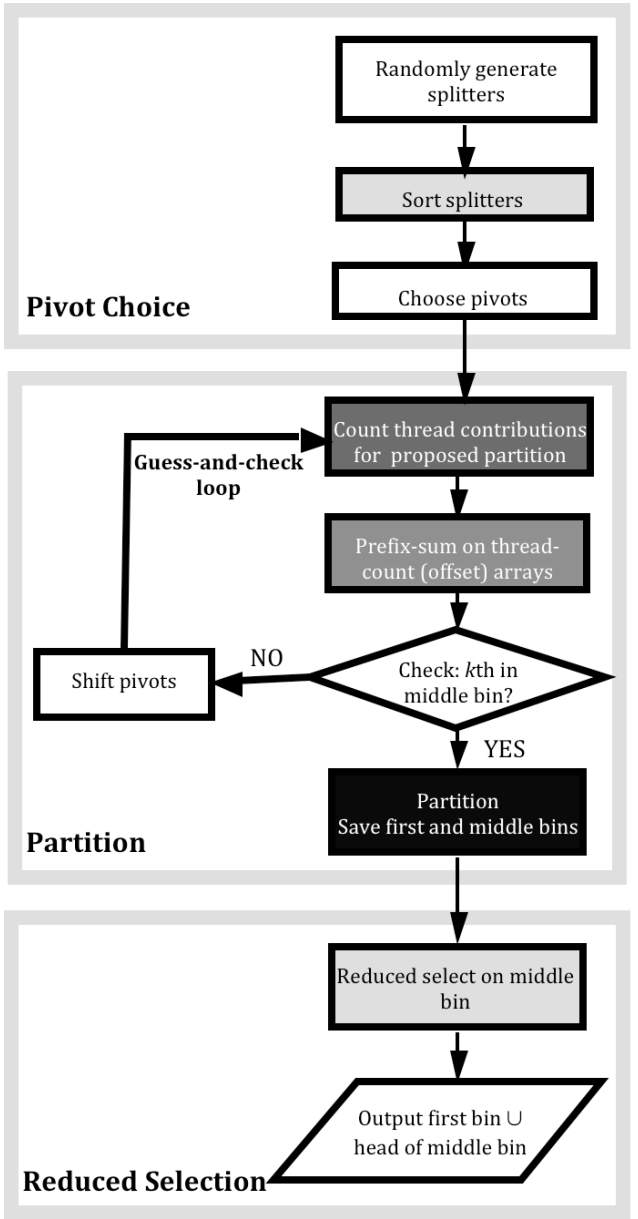
Guess and check

- This is a Las Vegas algorithm.
  - Las Vegas algorithm = probabilistic, but gives correct result
  - An example of stochastic optimization
- Relatively slow to directly select *but*
- Fast to calculate the guess
- Fast to do the calculations, given the guess
- Fast and easy to check correctness of pivot guess
- Can guess pivots with arbitrary accuracy
- Guess + check faster than direct calculation

Parameters

- n, the list length
  - This is limited by the amount of global memory
  - We implemented only powers of 2, but can generalize
- k, the number of elements to be selected
  - Varies between 1 and n
  - Gives rise to the quantile k/n
  - Quantile k/n allows comparison between different n
- p_k, the desired probability that the middle bin contains the kth element
  - Larger p_k implies fewer repetitions, but a larger final selection

Algorithm flow chart



Pivot selection

- Randomly select some number (*numSplitters)* of elements from the list and call these splitters
  - We empirically chose *numSplitters* = 8*√*n*, to balance first and last sorts
- Sort the splitters
- Imagine that the list is partitioned into buckets defined by these splitters
  - Each bucket has very roughly "more or less" the same number of elements
- This has the effect of flattening the distribution and allows one to reason probabilistically about the number of elements in each

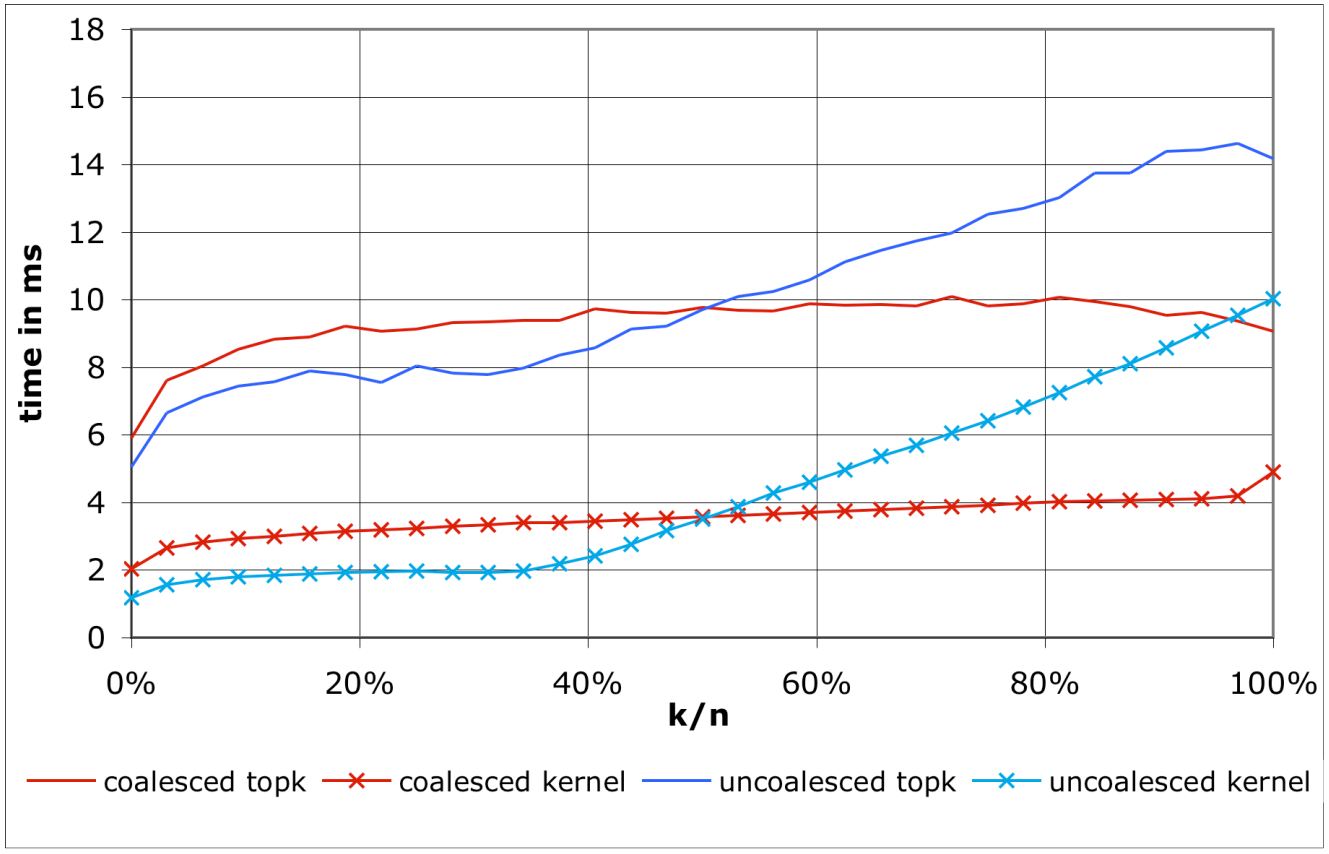
Pivot selection

- The probability that the *k*th is in the *i*th bucket is approximately
  $$C(numSplitters, i)\ (k/n)^i\ (1-(k/n))^{numSplitters-i}$$
- Binomial distribution approximation
  - *numSplitters* random trials
  - Success is defined as "the kth key < the splitter tried"
  - Success has probability *k*/*n*
  - Approximation may be weak for kth key outside splitters, but impact minimal if probabilities of end buckets small
- Starting at the bucket most likely to hold the *k*th, $k/(n/(numSplitters+1))$, add probabilities, incrementing buckets on each side, until the desired probability is exceeded.
  - Can do this as buckets are not overlapping
  - Binomial distribution is approximated by normal

Query and Partition kernels

- First pass -- counting loop
  - Count thread contributions to first and middle bin
  - Scan-add (prefix-sum) these for offsets
  - Compare total elements falling into the first and into the first and middle to see if the *k*th falls into the middle
  - If not, then shift the pivots in the appropriate direction and repeat
- Second pass - partition
  - Once you have good pivots, partition and write
- No atomics
- Closely coupled kernels account for most of the time spent

Coalesced vs. uncoalesced writes in partition-and-write kernel



GPU impact on algorithm

- Parallelism a great strength
  - Ran on up to 64 million threads for the partition kernels
- Small amount of fast memory near processors
- Bandwidth gain in using coalesced writes
  - But to do these coalesced writes also need to use more shared memory
- SIMD, divergent threads during partition
  - Super-scalar technique in (Sanders and Winkel 2004) didn't help for our two-level tree
- Slow reads across PCIe bus
  - If calculation is on GPU, best to select there too
- Limited global memory on GPU
  - Limits size of list that can be processed

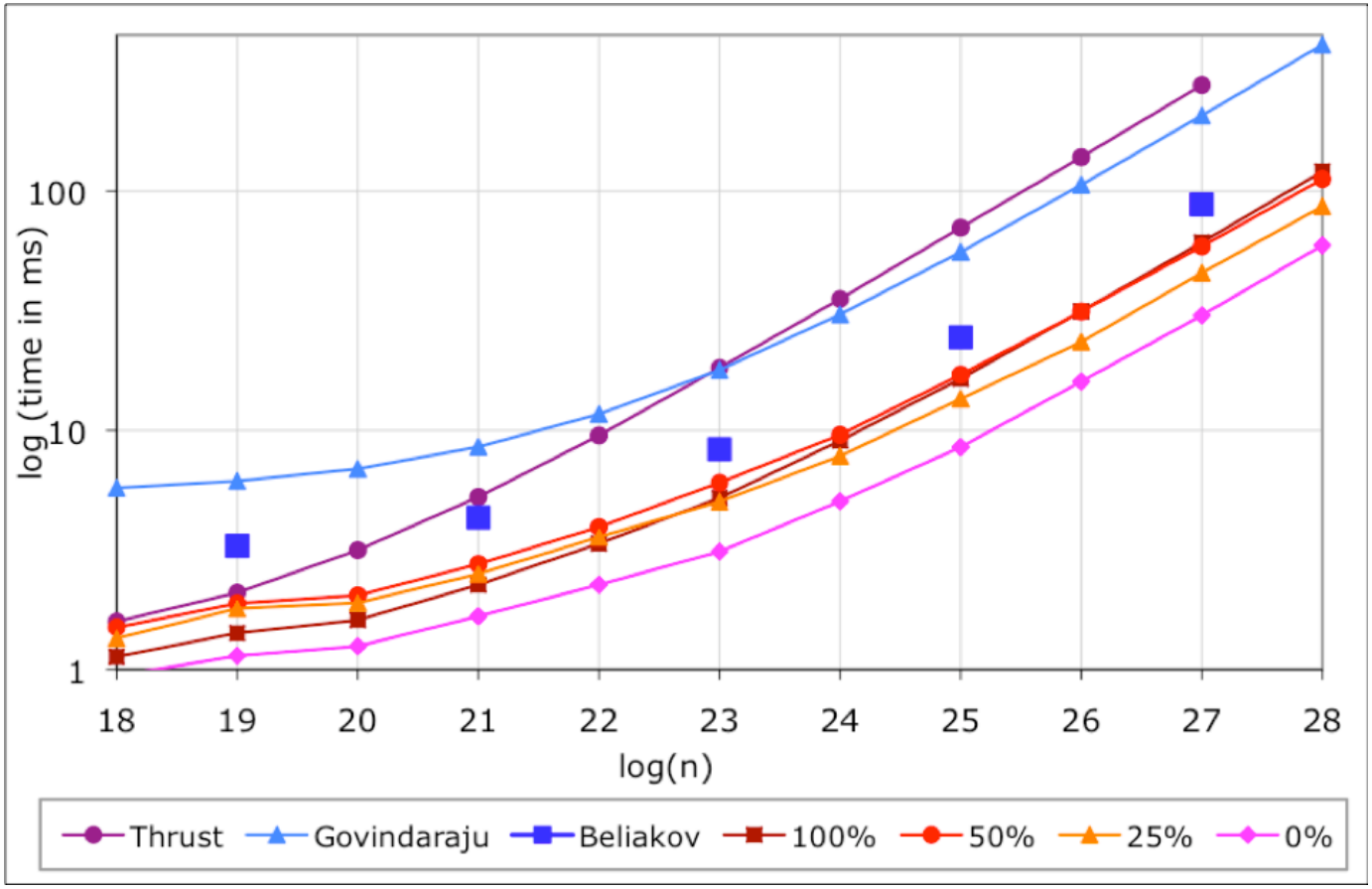
Verification

- This is an example of stochastic optimization
  - If the wrong pivots are chosen, the answer will still be correct but the timing will be slower
- We verified the code statistically as part of our experimentation
- Did a series of experiments consisting of 10K runs each, and compared the observed number of times the *k*th was in different buckets with the corresponding calculated bucket probabilities for different values of *k*
- Results were consistent with correct implementation of the algorithm

Experimental results

- Algorithms compared
  - This randomized top *k* selection
  - Thrust-based select-via-sort
  - Direct construction of *kth* element (Govindaraju 2004)
  - Construction by minimization of a convex function (Beliakov 2011)
- Data compared
  - Real radio-astronomy data
  - Random data
  - In-order and backward-order data
  - Data with repetition
  - All integer
- Platforms
  - NVIDIA Quadro 6000 (mostly), Quadro 5000, GTX 285, 8800 GT

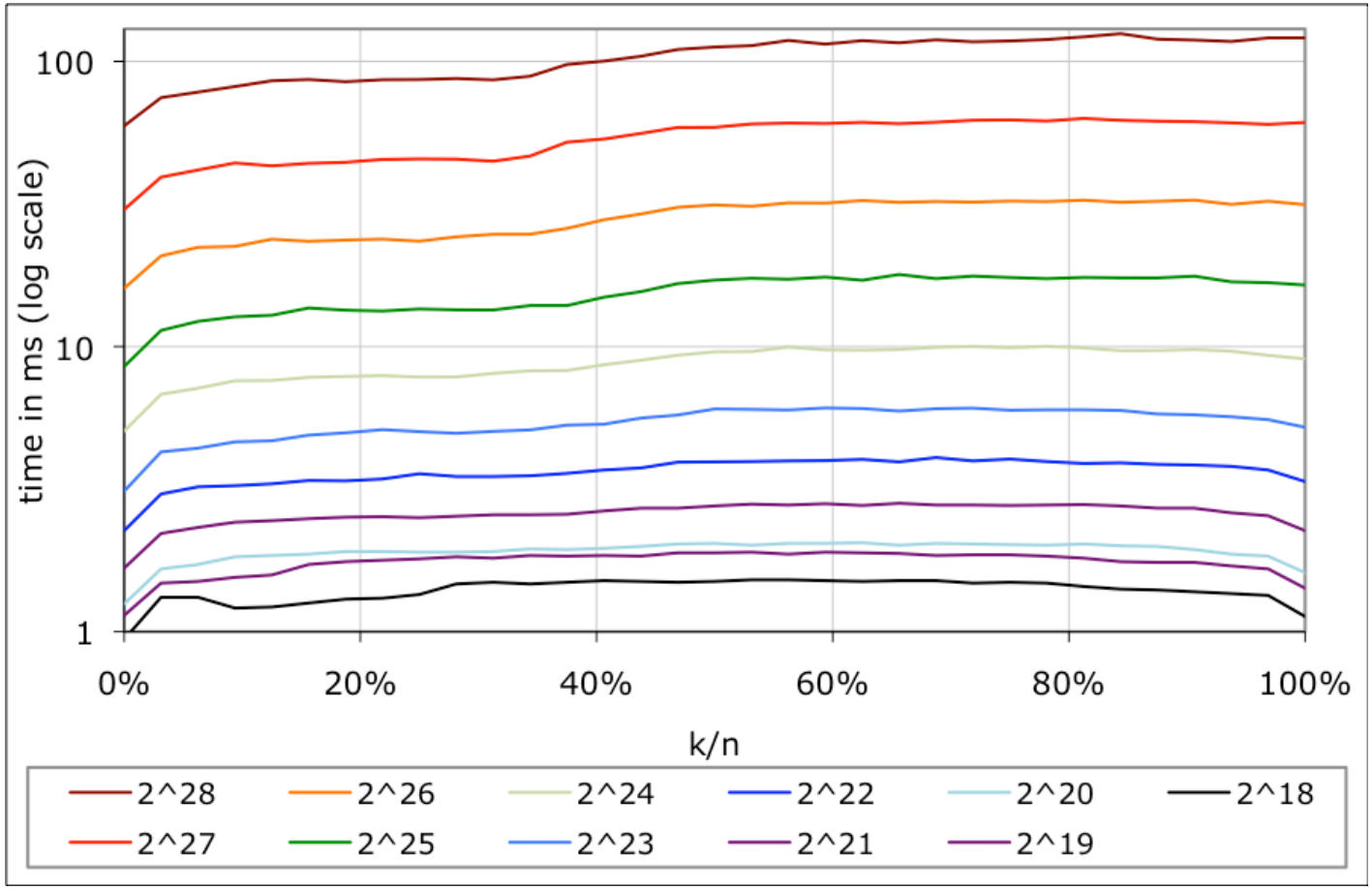
1.5-3x faster than best selections
3-6x faster than Thrust select-via-sort



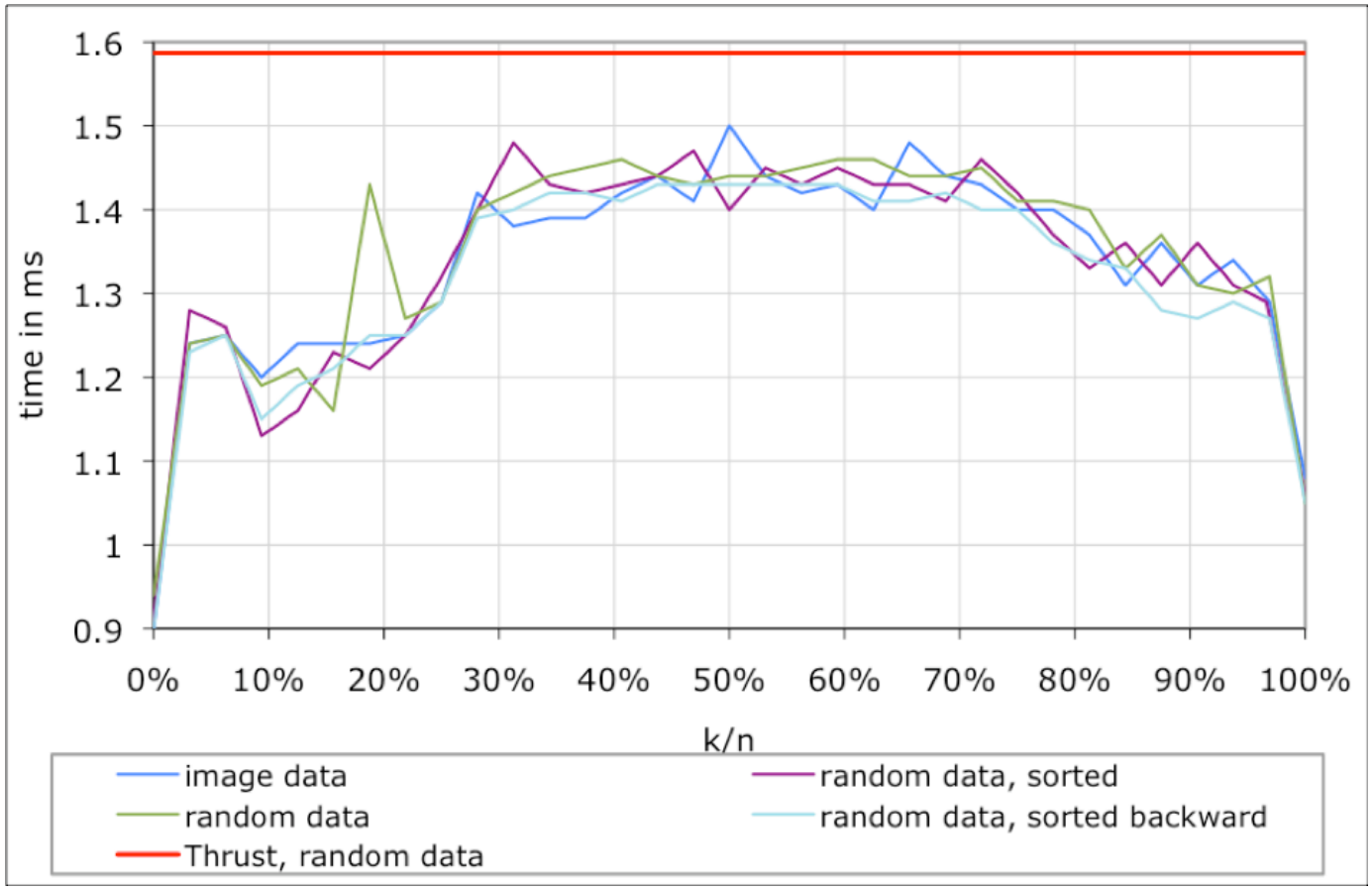
Handles list lengths *n* 2-4x bigger than Thrust select-via-sort

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}

Timings in terms of list length *n*



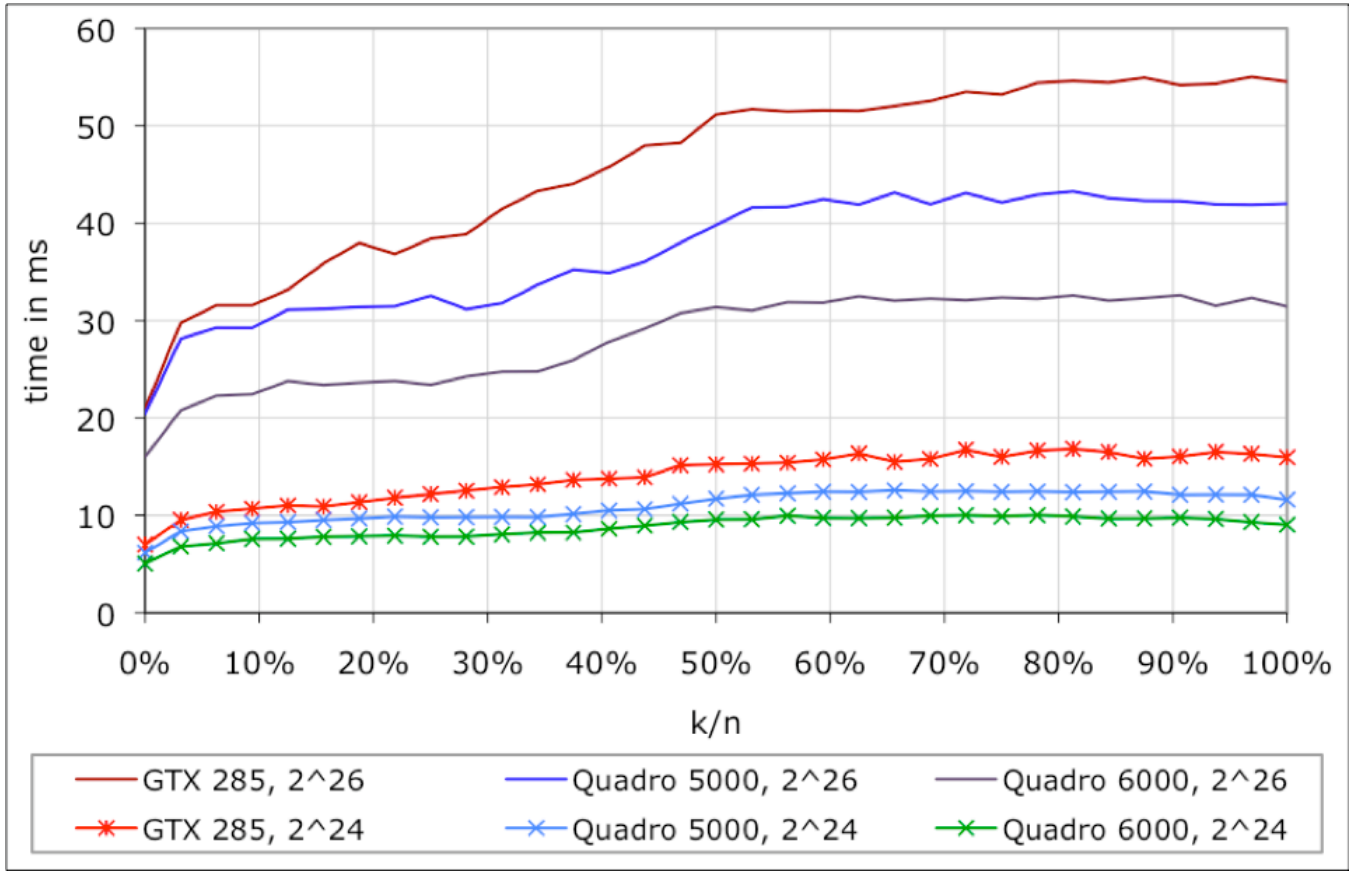
Real radio-astronomy data closely tracks random data



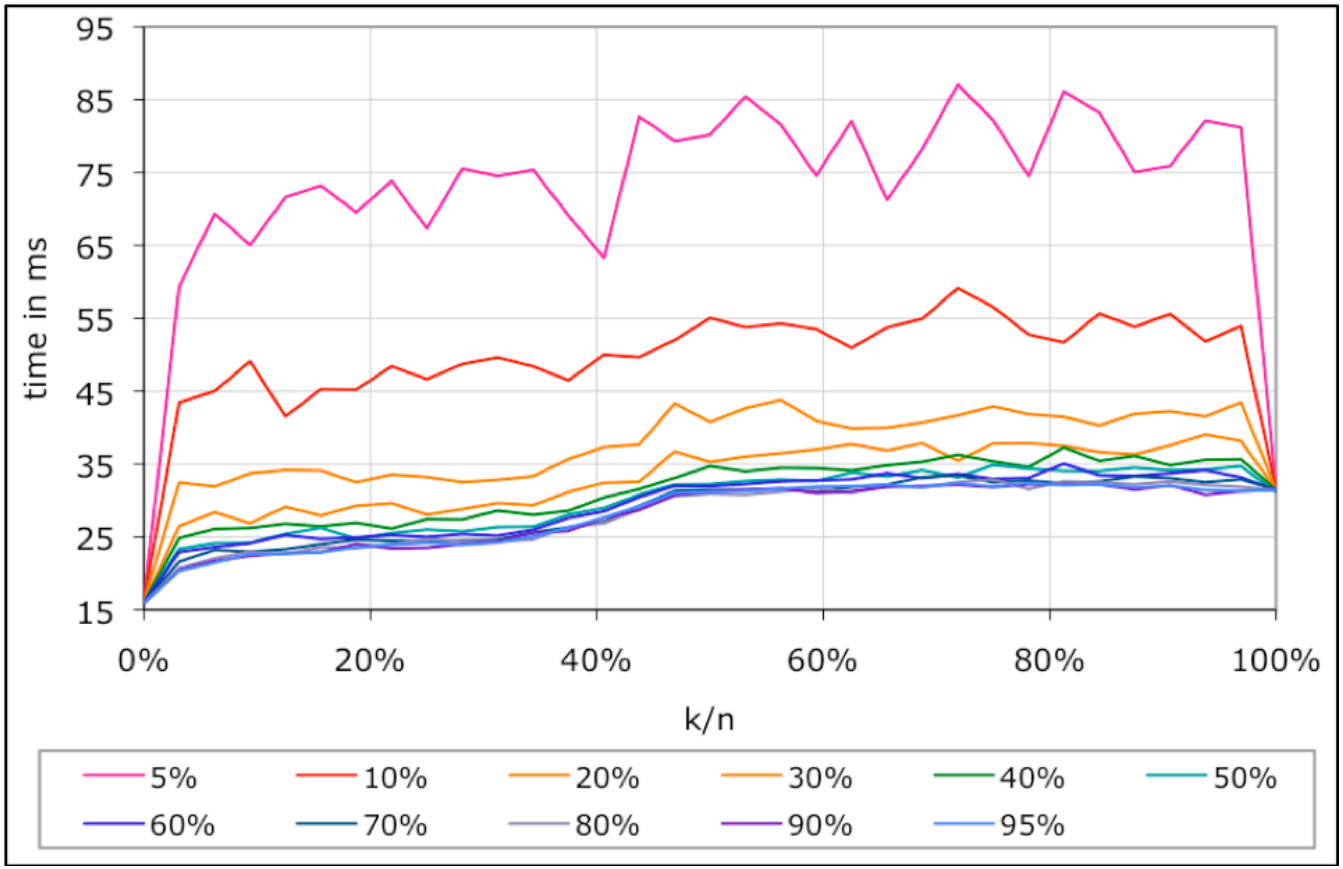
When is this algorithm slow?

- We identified one case when this is slower than select-via-sort
  - *Lots* of repetition AND
  - The *k*th key occurs near the repetition
- In that case, the select-via-sort can be faster
- We found the break-even to be when between half and three-quarters of the elements repeat (and the *k*th happens to be one of them)
  - Because nearly a full sort is done in the final select, along with all of the other work

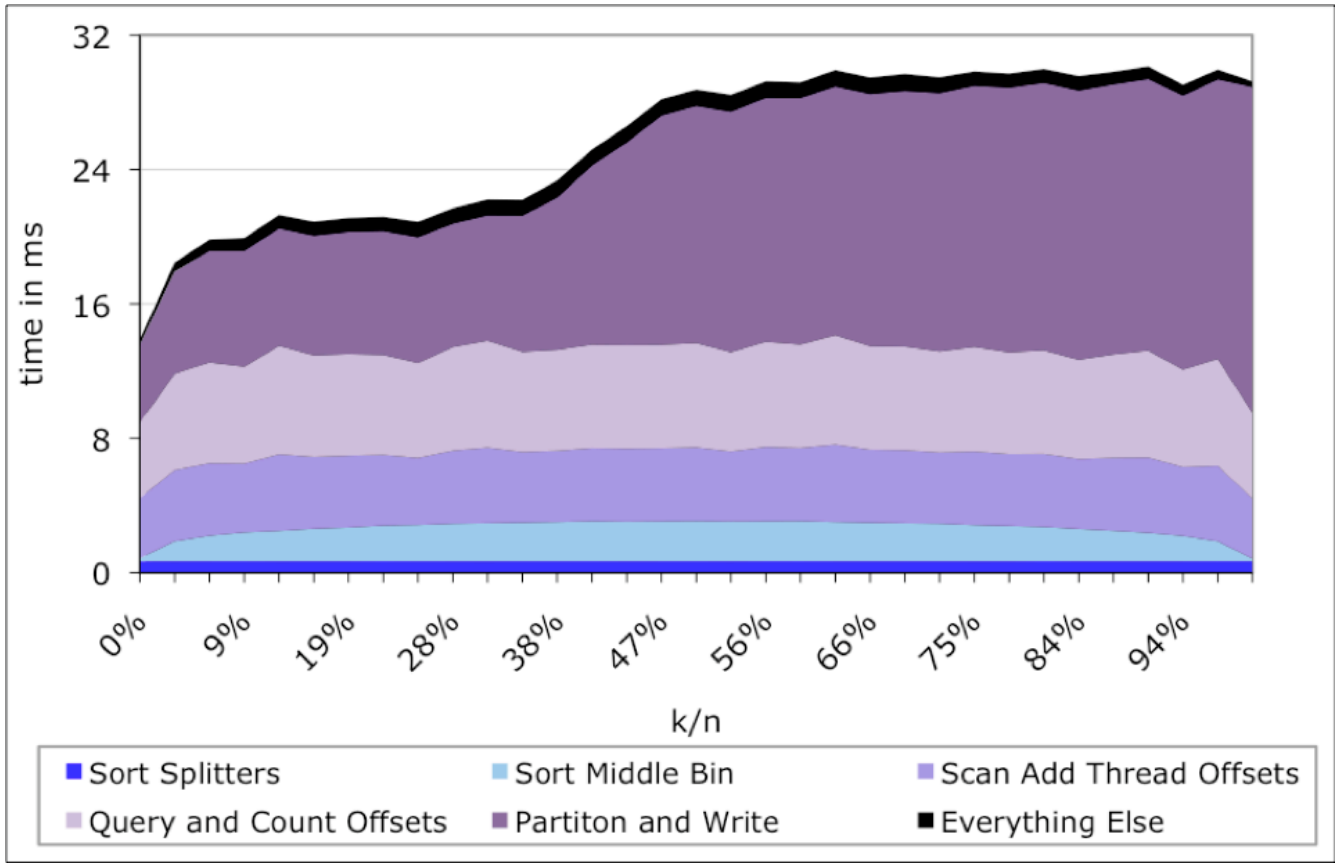
Timings for different GPUs



Probabilities p_k give similar timings when $p_k > 30\%$



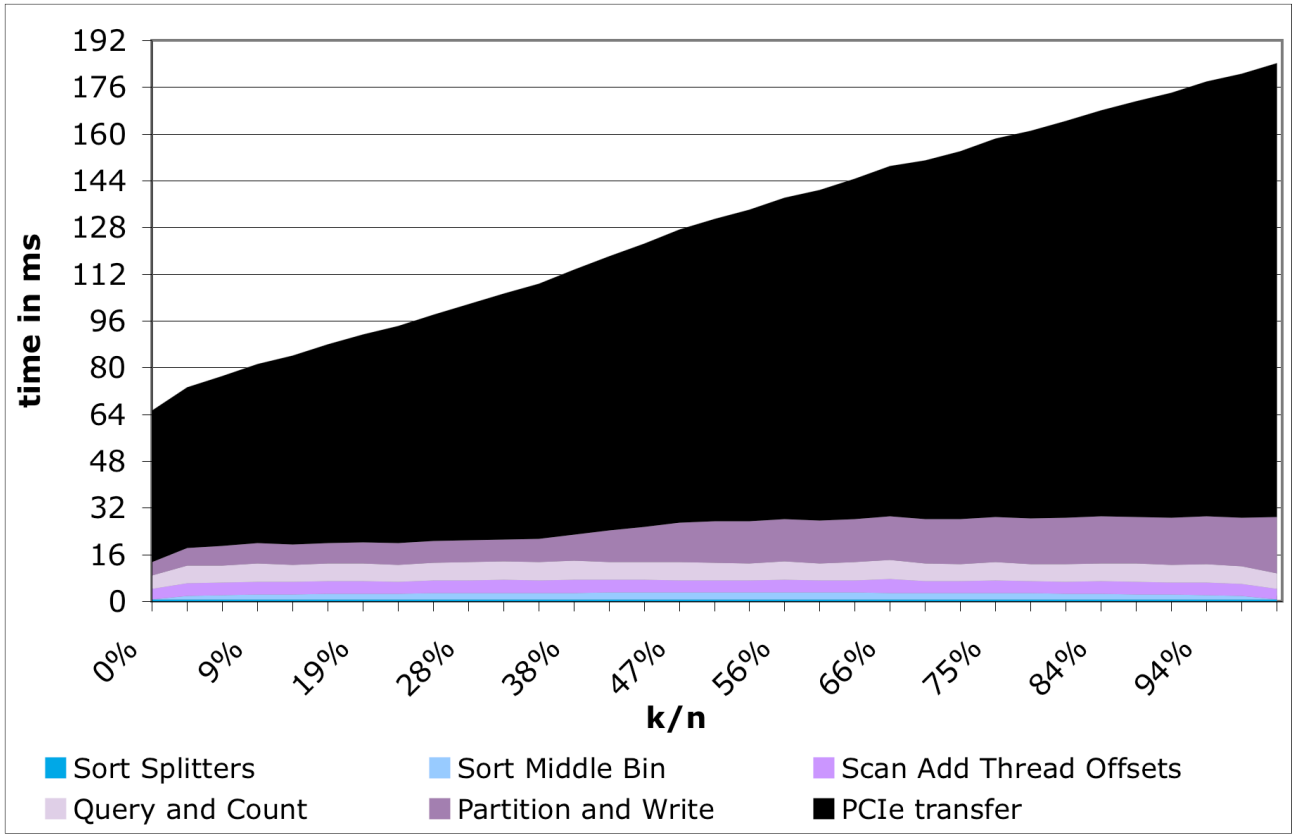
Kernel timings, $n = 2^{26}$



If you're on the GPU, select there too

Same $n=2^{26}$ run as previous slide

PCIe transfer time overwhelms the select time



Conclusions

- We have shown a fast randomized selection on the GPU using a Las Vegas algorithm.
  - Faster than other GPU-based selects we've examined
    - Even those that do less (i.e., only grab the *k*th, only grab keys)
- Select on the GPU when you're already there
- Use this randomized select if
  - The list changes between calls to select
  - You want longer lists than a sort handles on your GPU
- Use select-via-sort *only* if
  - You are selecting on the same list many times
    - That's the added value of doing a sort first
  - You know the series of lists have more than half repeated keys and know that the *k*th is one of them most of the time