Parallel Sorting of Roughly-Sorted Sequences

CSCI 5172 Fall '16 Project

Anthony Pfaff, Jason Treadwell

December 8, 2016

Roughsort is a sorting algorithm that exploits the *radius* of a sequence to beat the linearithmic lower runtime bound of the comparison sort family of algorithms. Its structure invites implementation using parallel execution. Here we present a parallel Roughsort implementation using Nvidia's CUDA platform for heterogeneous GPGPU programming and interpret our results.

1 The Array Sorting Problem

Sorting the elements of an array is among the most classic problems of computer science, often considered to be the most fundamental algorithmic problem [11]. Not merely just for arranging data to make it easier to read, sorting algorithms are frequently employed to render graphics and to improve the performance of other algorithms (e.g., set intersection or finding the unique elements of a list).

The *comparison* sorts are an indispensable family of sorting algorithms containing

such classics as Heapsort, Mergesort and Quicksort. True to its name, algorithms in this family determine how to reorder a sequence by comparing its elements against each other according to some natural or explicitly-provided ordering; for the sake of simplicity, we shall only consider the sorting of integral elements in non-decreasing order, backed by random-access array storage[1]. A well-established result is the "linearithmic" $\Omega(n \lg n)$ runtime lower bound for the general problem of sorting an array of n elements by comparison; algorithms such as Mergesort achieve $\Theta(n \lg n)$ runtime (and are thus asymptotically optimal) whereas Quicksort, while usually outperforming Mergesort over arrays that fit in memory, has quadratic-runtime pathological cases.

Comparison-based sorting algorithms, such as Mergesort, often have divide-and-conquer structures that can easily and substantially be sped up by using *nested parallelism* and other means of parallel execution [11].

2 Sorting Roughly-Sorted Sequences

The optimal $\Omega(n \lg n)$ runtime bound of comparison sorting can be beaten when additional analysis is performed on the input sequence [11]. An algorithm by Altman and Igarashi, which we call *Roughsort*, improves upon this lower bound by exploiting the degree to which the unprocessed input array is partially sorted [8]. Specifically, Roughsort sorts an array in $\Theta(n \lg k)$ time, where k is the *radius* of the input sequence.

[1] Sequential-access storage demands distinct and divergent design considerations.

A sequence $A = \{a_0, a_1, \cdots, a_{n-1}\}$ is k-sorted if it satisfies

$$a_i \leq a_j \quad \forall\, i < j - k\,, \quad 0 \leq i \leq j < n\,,$$

where a 0-sorted sequence is fully-sorted. The radius of A is the smallest such k for which A is k-sorted. Alternatively, and perhaps more usefully, the radius of A measures how far away its most out-of-order element is from its position in the fully-sorted array:

$$\text{radius}(A) = \max\{j - i \mid j > i,\, a_i > a_j\}\,.$$

Roughsort exploits the radius of A by using a *halving* algorithm that takes a $2l$-sorted sequence and partially sorts it into an $\lfloor l - 1 \rfloor$-sorted sequence. By repeatedly halving the sortedness of A, we fully sort it in $\lg k$ runs, whence the runtime. The radius of A must therefore be given *a priori* or determined from the input in order to effectively perform the algorithm.

Some applications involve sorted arrays that are updated or extended in such a way as to only slightly perturb the ordering (i.e., the radius k of an out-of-order sequence is small) [8]. In such cases, Roughsort could be employed to quickly resort the sequences faster than by using a full $\Theta(n \lg n)$ comparison sort. Since an unsorted sequence of length n can be at most $(n-1)$-sorted, the runtime of Roughsort is $\Theta(n \lg k) = O(n \lg n)$ and is thus asymptotically-optimal among comparison-based sorting algorithms.

Similar to other comparison sorts, Roughsort has divide-and-conquer character-

istics that support parallel execution on various multiprocessor models [8, 7].

3 Sequential Implementation

As noted, Roughsort must know the radius k of the input array $A = \{a_0, a_1, \cdots, a_{n-1}\}$ in order to completely sort it. If not known, the radius can be determined in linear time [8], preserving the $\Theta(n \lg k)$ complexity of the algorithm.

To determine the radius, we must first compute the *characteristic sequences* $LR(A) = \{b_i\}$ and $RL(A) = \{c_i\}$:

$$b_i = \max\{a_0, a_1, \cdots, a_i\}, \quad c_i = \min\{a_i, a_{i+1}, \cdots, a_{n-1}\}, \quad 0 \leq i < n$$

Both sequences are easily computed in linear time and space using simple min/max prefix scans. Note that $c_i \leq b_i \ \forall \ i$. By performing a linear-time scan of these sequences and observing where they "cross" each other, we can compute the *disorder measure* sequence $DM(A) = \{d_i\}$, where each d_i represents how displaced a_i is from its position in the sorted ordering of the elements of A:

$$d_i = \max\left\{\{i - j \mid c_i < b_j\} \cup \{0\}\right\}$$

The radius of A is thus the maximum element from $DM(A)$. We employed a modification of the DM algorithm from [8] where we save memory by simply keeping track of the maximum-encountered d_i instead of storing the entire sequence; our sequential radius determination algorithm thus runs in linear time and requires $2n = \Theta(n)$ space.

The core of the sequential Roughsort implementation is the aforementioned radius halving algorithm from [8]. This algorithm halves the radius k of A in place through the following three steps:

1. Partition each consecutive run of k elements in A about the mean of the run.
2. Starting at element $a_{\lfloor k/2 \rfloor}$ to stagger the partitions, repeat step 1 and go to 3.
3. Repeat step 1 and halt.

A run of elements is "partitioned" by partially sorting it such that no element before the median of the run exceeds the median and no element thereafter is less than it. We partition each run using the `nth_element` function from the C++ STL, which implements an optimized, linear-time selection algorithm [11] such as Introselect[2] [5].

Our sequential implementation thus sorts the array A in place by determining its radius k and halving the radius $\lg k$ times, in total requiring linear space and taking $\Theta(3(n/k \cdot k) \lg k) = \Theta(n \lg k)$ time.

The halving algorithm fails to halve the radius of 1-sorted sequences, since it's impossible to stagger their radial runs; our implementation thus performs a single linear Bubblesort scan to quickly sort such sequences.

For an analysis of the sequential implementation's runtime performance, please see Section 9.

[2]Introselect is based on Quickselect, which resembles a Quicksort that searches for the nth element of the sorted array by only partitioning one side of each pivot.

4 Parallel Acceleration Using CUDA

Parallel implementations of the Roughsort and radius finding algorithms are also addressed. The aforementioned algorithms assume a specific underlying memory in achieving the subsequent parallel implementation runtimes: the concurrent-read concurrent-write parallel random access model (CRCW PRAM). Shiloach and Vishkin elaborate on this memory model, providing that it is a purely theoretical model which, *inter alia*, meets the following criteria: a shared, global memory of size $m \geq p$ is available to p processors, and up to p positions in this memory may be simultaneously read or written in one step [21]. As with Roughsort, a number of algorithms take advantage of this model's assumptions to provide performance improvements over their serial counterparts [22] [19].

Conspicuous, however, with these and other CRCW PRAM algorithms is a lack of presented implementations. The model is merely that, and thus implementing such an algorithm requires a suitable proxy which shares the characteristics of CRCW PRAM. In contemporary computing, the most notable such proxy is graphics processing unit (GPU) parallelization.

GPUs invert a core principle of general purpose computing to yield a performance improvement on specific workloads. Rather than having one or a handful of general purpose, powerful multiprocessors, GPUs opt for a different model: many less-powerful, special purpose processors. Contemporary examples include the Skylake Intel Xeon E3-1275 v5 CPU and the Nvidia GTX 1080 GPU[3]. The Xeon has 4

[3]Both were used on this project, for the evaluation of sequential and parallel implementations of these algorithms, respectively.

processor cores whereas the Nvidia card has 2560 cores [2] [1].

Nvidia—the manufacturer of the aforementioned graphics card—makes available an API suite and SDK which allows developers to readily access the GPU: CUDA. This platform provides developers a simple interface with which they can offload processing of CUDA-augmented C++ software payloads to CUDA-supported Nvidia cards.

The parallelism provided by Nvidia and CUDA in this scheme can be realized by developers who develop their payloads in specific ways. Fundamentally, programs running on an Nvidia card via CUDA operate differently than their CPU counterparts. In general, a line of code executing on an Nvidia card is not processed in isolation. Rather, the Nvidia CUDA scheme provides for single instruction, multiple thread (SIMT) execution, as opposed to the usual single instruction, single data scheme (SISD) provided by general purpose CPUs. In contrast to a single instruction accessing or updating one memory location as the result of one instruction executing on a single core of a general purpose CPU, a single instruction executing on an Nvidia card is executing inside multiple threads on multiple respective cores simultaneously, each of which operates on a different location in memory.

When a developer wants to take advantage of this parallelism, they structure their code such that the multiple results of the same instruction executing with values from different, largely contiguous memory locations is a useful component of the overall program's payload. Contiguity is an important caveat, due to the way memory accesses in Nvidia cards are structured. Effective memory bandwidth is significantly reduced when memory accesses are not grouped in such a manner as to

take advantage of the architecture, reducing memory bandwidth by over 90% on even higher-end Nvidia cards [3]. Memory access which complies with this constraint is said to have *coalescence*.

The core component of an Nvidia payload is a thread; an instance of the series of instructions in the overall CUDA payload. Multiple threads run in a group called a block. Blocks contain a developer-specified number of threads each. Between threads and blocks is the concept of a warp. Warps are collections of 32 threads executing simultaneously on separate processor cores of the Nvidia card. Usually, all threads in a warp are executing the same instruction simultaneously. However, in situations where a conditional is evaluated by members of a warp, all threads for whom the conditional yields true will execute, and subsequently all threads with a false conditional result will execute (or vice versa), in a phenomenon called *warp divergence*. This yields from the SIMT architecture, where all cores on which a warp executes must be processing the same instruction or temporarily switched off and not processing at all. Blocks are divided up into multiple warps for scheduling and execution on an Nvidia GPU. Collections of these blocks are referred to as a grid. Finally, encompassing all of these components is the kernel. Kernels are the series of instructions which threads, warps, blocks, and grids comprise an instance of and represent the sequence of code to actually be transferred to and executed on GPU.

Nvidia provides a simple mechanism by which developers can access the parallelism provided by CUDA. Each of the above divisions of the CUDA payload are provided with an identifier. A tuple composed of the thread, block, and grid iden-

tifiers[4] uniquely identifies that component of the workload. Using this information, individual threads all executing the same instruction (such as a memory read from an array) can perform that instruction on distinct data (access different memory locations in the array) [17].

5 Parallel Radius Determination

In achieving parallel radius finding, Altman and Chlebus noted that the LR, RL, and DM sequences necessary to determine the radius k of a roughsorted sequence could be computed in $O(\lg k)$, while k is again at most $n - 1$ [7], with the above underlying assumptions.

Using CUDA as a proxy for CRCW PRAM, its techniques provided a platform to develop the GPU parallel versions of sequence radius finding and subsequent roughsorting.

In support of the above discussed parallel Roughsort implementation, CUDA implementations of the RL, LR, and DM sequence determination algorithms were developed. This was done by assigning each thread a unique identifier and then, analogous to Altman and Chlebus'work, having each thread manage the corresponding slot in the global unsorted array [8]. Each thread performed the parallel min-then max-prefix operations until the respective lists were efficiently determined ordered. Sorting was determined by having each thread determine if its assigned slot in the LR or RL list was greater than the next element in the list, and having each thread globally assert any true (unsorted) result to global memory, available to all

[4]Concerns with warps are abstracted away from the developer. Consequently warp identifiers are not needed and are not available.

other threads.

Once the RL and LR lists were completed, the DM list was computed by using Altman and Chlebus's serial radius techniques, while using the $O(\lg k)$ bounded techniques introduced by their RL and LR sequence determination algorithms. However, a notable divergence from the serial version of the serial DM list algorithm was taken. The inner loop of the serial DM list algorithm added a criteria which assured that $LR[i]$ (the ith element of the LR list) was always greater than or equal to $RL[i]$ (the ith element of the RL list). Only the power of 2 greater than the actual value of k is needed for parallel Roughsort to function, so this criteria was removed in the interest of efficiency and simplicity [8].

Finally, to determine the radius from the DM list, the maximum element needed to be found. For this purpose, the *Thrust* CUDA library was used, which provides CUDA-optimized implementations of a number of C++ STL functions [6]. One such STL function is the `max_element` function. Using the *Thrust* implementation of this function, maximum elements of large arrays can be found expediently on Nvidia hardware using CUDA [4]. Using this function yielded the maximum element of the DM list, which is k. Further, to ensure all values quiesced during LR, RL, and DM list computations, a separate kernel was launched to perform the final DM computations.

6 Radius Determination Runtime Analysis

Running these kernels yielded disappointing results, however. Figures 1 and 2 compare the runtime of the sequential algorithm to the parallel. For random lists

composed together in such a manner as to yield a specific k value for the list (as discussed later in the paper) [5], sequential CPU computations exceeded performance on GPU for any tested n when values of k exceeded 6, as see in Figure 2[6]. Figure 1 reports more positive results for k equal to 2, showing that parallel run times were equalled or exceeded by sequential runtimes for tested values n equal to or in excess of 150,000, though this result was at best constant factor improvement over the sequential implementation and does not demonstrate the expected $O(\lg k)$ runtime, emphasizing a more fundamental issue with the platform, implementation, or both.

[5]Lists which had a specific k value were generated ahead of time and then evaluated using the radius finders to prove both correctness and provide for validity in this experimentation

[6]These results reversed when performed in a Windows environment, compiled with Visual Studio 2015 Community and run on commodity hardware. The authors suspect this to be a false result and posit that optimization differences on the C++ STL `nth_element` function are the cause. This invites further investigation, however.

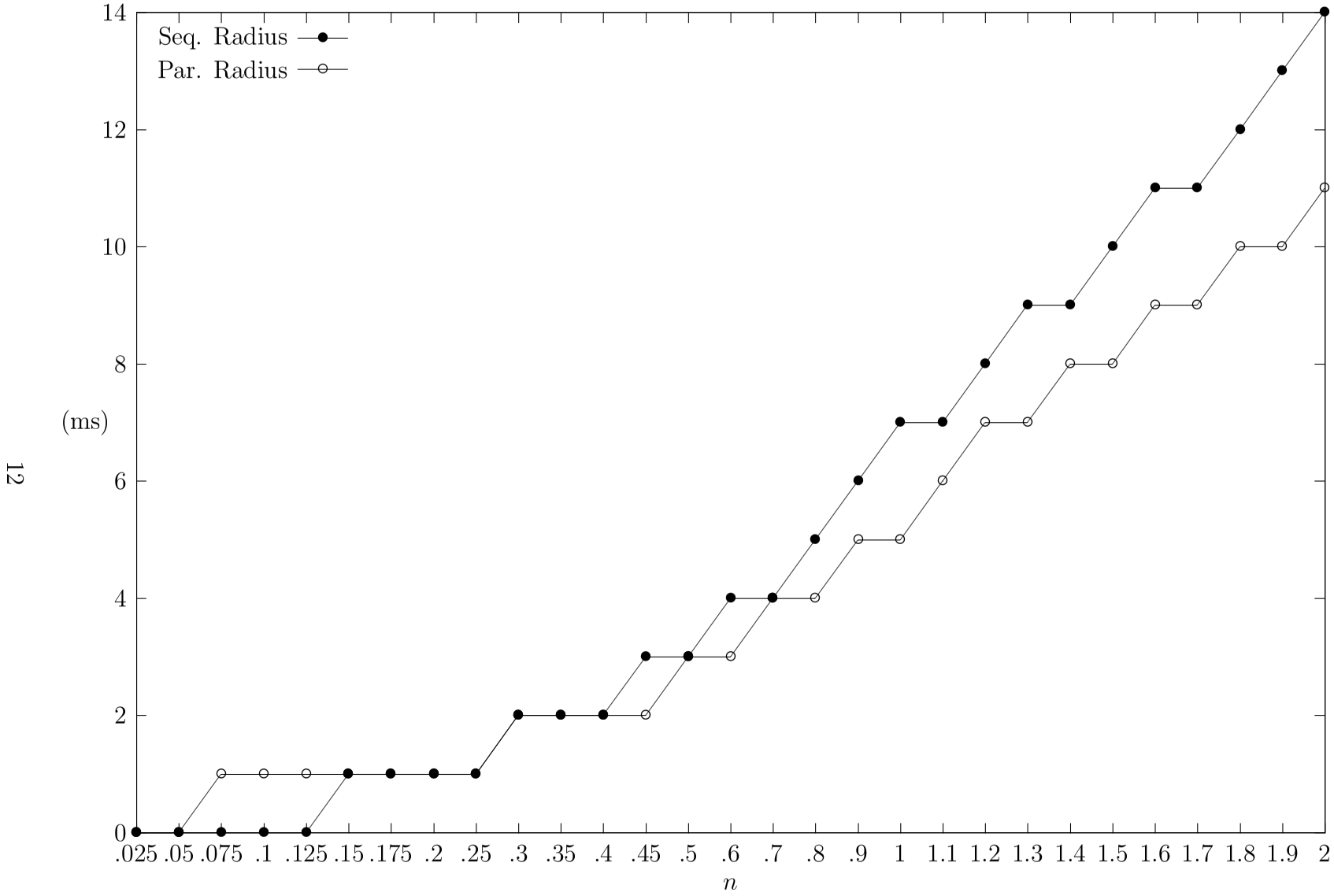

Figure 1: *Radius Determination Runtimes over Arrays of Length* $n \cdot 10^6$, $k = 2$

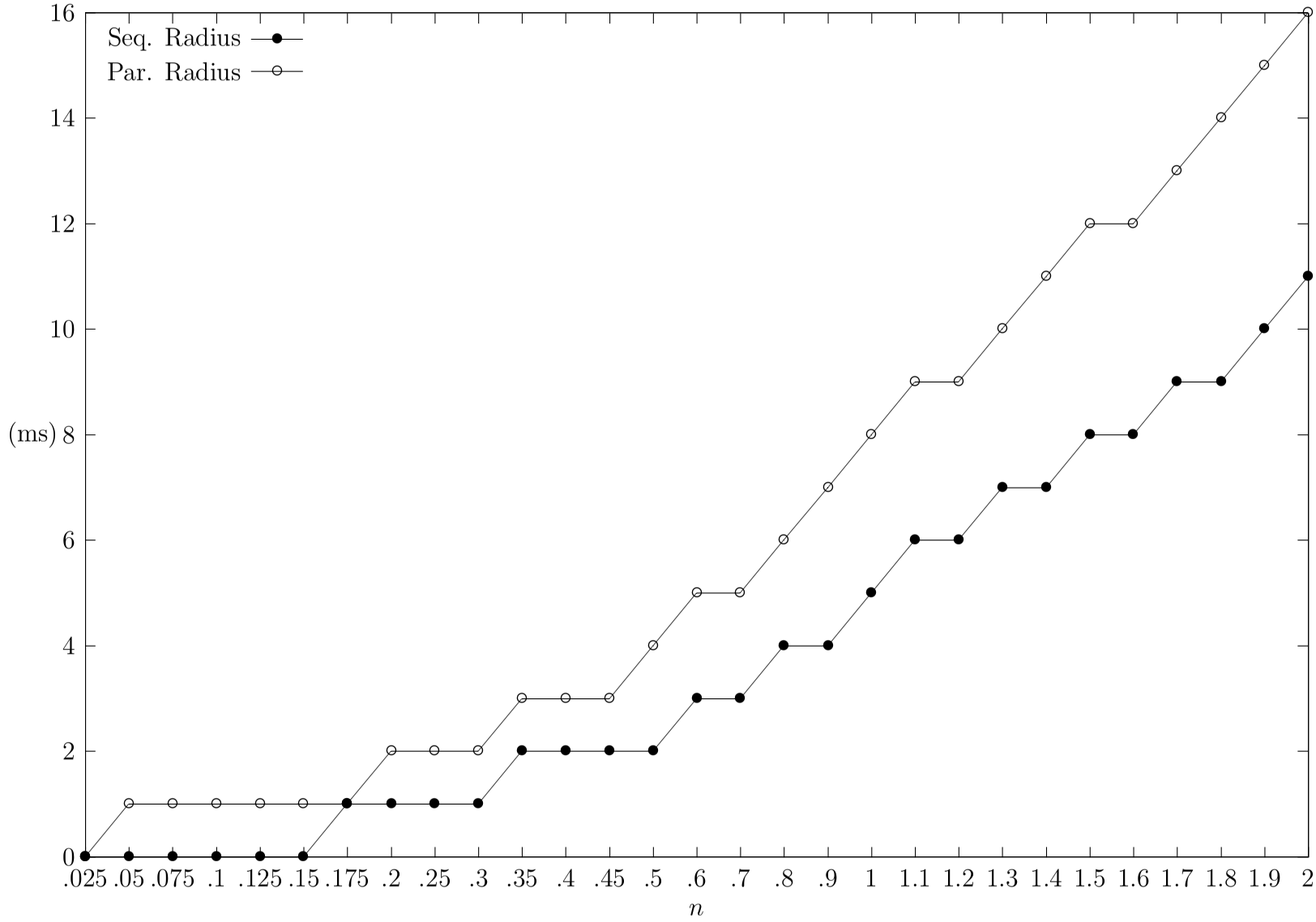

Figure 2: *Radius Determination Runtimes over Arrays of Length* $n \cdot 10^6$, $k = 100$

7 Parallel Roughsort Implementation and Results

We first attempted to implement the parallel Roughsort described in [8]. This approach follows the sequential version from Section 3, except in that the segment partitions are all launched in parallel at each of the 3 steps, synchronizing between steps. We employed Nvidia's *Thrust* library, which provides a parallel CUDA implementation of the C++ STL.

Regrettably, `Thrust` has yet to provide an `nth_element` selection algorithm implementation, so we replaced it with calls to `Thrust`'s `sort` algorithm, which performs a "state-of-the-art", highly-optimized radix sort implementation when invoked over arrays of primitive integer elements [18, 16]. Radix sort is a generalized counting sort that runs in linear time when operating over fixed-size array elements. While radix sort requires additional storage (unlike Introselect, which was used in the sequential implementation), this additional storage ought to be acceptable as the segments we wish to partition ought to be relatively small.

Each `Thrust sort` call issues at least one kernel launch; we attempted to run radix sorts over the entire array in parallel by establishing a fixed number of CUDA *streams*. Whereas kernels launched through the same stream are executed in order, up to 16 kernels launched through distinct streams can be run concurrently on the GPU, available resources permitting [17]. We established a fixed number of CUDA streams and launched each `sort` call through them in a circular, round-robin fashion.

Unfortunately, while this implementation was very simple to code and correctly sorted its input, it was thousands of times slower than the sequential implementation. Running the implementation through CUDA's `nvvp` profiler showed that the

`sort` calls weren't being launched in parallel as expected, possibly due to some non-obvious synchronization mechanism that made concurrent launches of this function impossible.

We then implemented the parallel Roughsort approach described in [7], which prescribes the parallel Mergesort algorithm by Cole in lieu of weak partitioning and is implemented as follows for sorting the array A with radius k in place:

1. In parallel, sort all consecutive runs of $2k$ elements in A.
2. Repeat step 1, but begin the first run at a_k; halt.

Unlike the sequential algorithm, this approach fully sorts each array segment and thus needs only two batches of sorts without intra-radial staggering. Furthermore, just a single iteration of this process suffices to fully sort the k-sorted input.

We implemented this approach using a CUDA kernel where each thread in the launch is responsible for sorting exactly one segment from step 1 or 2; the steps are divided into separate kernel launches, each given an offset from which begin segmenting the array. While the threads in each launch run concurrently[7], each thread sorts its designated segment sequentially. In contrast to [7], we used `Thrust`'s radix sort instead of its Mergesort implementation in order to leverage the linear runtime of the former. While this approach to the parallel Roughsort was far more successful than the previous, it still struggled against its competitors (see Section 9).

[7]For large arrays, the GPU can't host every thread simultaneously and must sequentially execute the thread blocks in batches.

8 Random k-Sorted Sequence Generation

We developed a k-sorted random sequence generator to test our implementations. The generator takes k and the length of the desired array n as arguments (specified by the user on the command line) and allocates a contiguous array of n 32-bit random integers. In order to avoid pseudorandom statistical weaknesses, we sourced these random integers from the high-end, cryptographically-secure hardware random number generator included in recent Intel processors [14].

We k-sort the random array A by fully sorting it using the C++ STL before perturbing the sorted elements. This perturbance is achieved by randomly choosing sorted elements a_p, a_q such that $q - p = k$ in order to guarantee the k-sortedness of the sequence. To increase the rigor of the randomized k-sorting, each consecutive, disjoint run of at most $k + 1$ array elements (excluding a_p and a_q) is then randomly shuffled to produce a random subarray that is itself at most k-sorted.

9 Implementation Performance Analysis

We tested our sequential and parallel Roughsort implementations against sequential and parallel Mergesort (as implemented by the C++ STL and by CUDA `Thrust`, respectively) as well as against our simple, sequential Bubblesort implementation. We chose Mergesort since it's a classic sort algorithm that also scales very well with parallel execution, capable of achieving a speedup of at least $\Theta(n/\lg^2 n)$ when given unlimited processors for a runtime of $\Theta(\lg^3 n)$ [11]. While Bubblesort has a quadratic runtime upper bound, its lightweight, linear lower bound emerges when

given nearly-sorted sequences and provides a means of comparing our implementation's performance for small values of k.

All tests were run on a workstation sporting a high-end, quad-core Intel CPU as well as an Nvidia GeForce GTX 1080—a recent and high-end[8], consumer-grade GPU suitable for heavy computational workloads over 32-bit values.

The time needed to transfer data to the GPU prior to running the parallel sorts isn't counted in the runtime results, since we'd expect such sorting operations to run as part of larger GPU workloads and not in isolation. Since our parallel radius determiner wasn't complete until late in development and ran slower than the sequential implementation over arrays with larger radii, our parallel Roughsort implementation used the sequential implementation to compute k, the runtime of which wasn't figured into the results.

Figures 3 and 4 compare all five sorting algorithms, varying n while fixing k to 2 and 15, respectively. Roughsort ought to have a runtime advantage with the smaller radius that dissipates as we increase k. Figures 5 and 6 invert the axes and vary k while fixing n at $750 \cdot 10^3$ and $1.25 \cdot 10^6$, respectively. For each tested (n, k), all five sorters were run over the same, k-sorted random array of length n; their runtimes were averaged over eight such runs. The runtime axis is logarithmic in every graph.

Figure 3 demonstrates the utility of the sequential Roughsort algorithm, which beats even the highly-optimized Mergesort from the C++ STL at handling the nearly-sorted input arrays. This victory is greatly overshadowed by the sequential Bubblesort, which is operating close to its optimal domain.

[8]At the time of this writing.

The next battery of tests in Figure 4 raises k to 15, causing the sequential Roughsort's advantage over Mergesort to evaporate. The parallel Roughsort generally outperforms Bubblesort here (which fairs better than expected against the sequential Mergesort in a trend that can't last for much longer to the right of the graph) and defeats the sequential Roughsort. Unexpectedly, the parallel implementation exhibits about a factor-of-ten speedup over the $k = 2$ graph; this shows that the overhead of launching many CUDA threads over small array subsequences substantially counterbalances the greater runtime of sorting larger subsequences.

Figures 5 and 6 fix n and vary k, yielding similar results. For arrays of these lengths, the parallel Roughsort's crossover point lies somewhere around the interval $20 \leq k \leq 30$, where it manages to best the three sequential algorithms. An invariant among all four graphs is the utter dominance of the parallel Mergesort implementation provided by *Thrust*, whose runtimes never depart from the x-axis.

These results suggest that our parallel Roughsort's constant runtime factor and crossover point are relatively high. We were unable to observe a benefit from its superior asymptotic bound over nearly-sorted arrays that could fit within the GPU's memory.

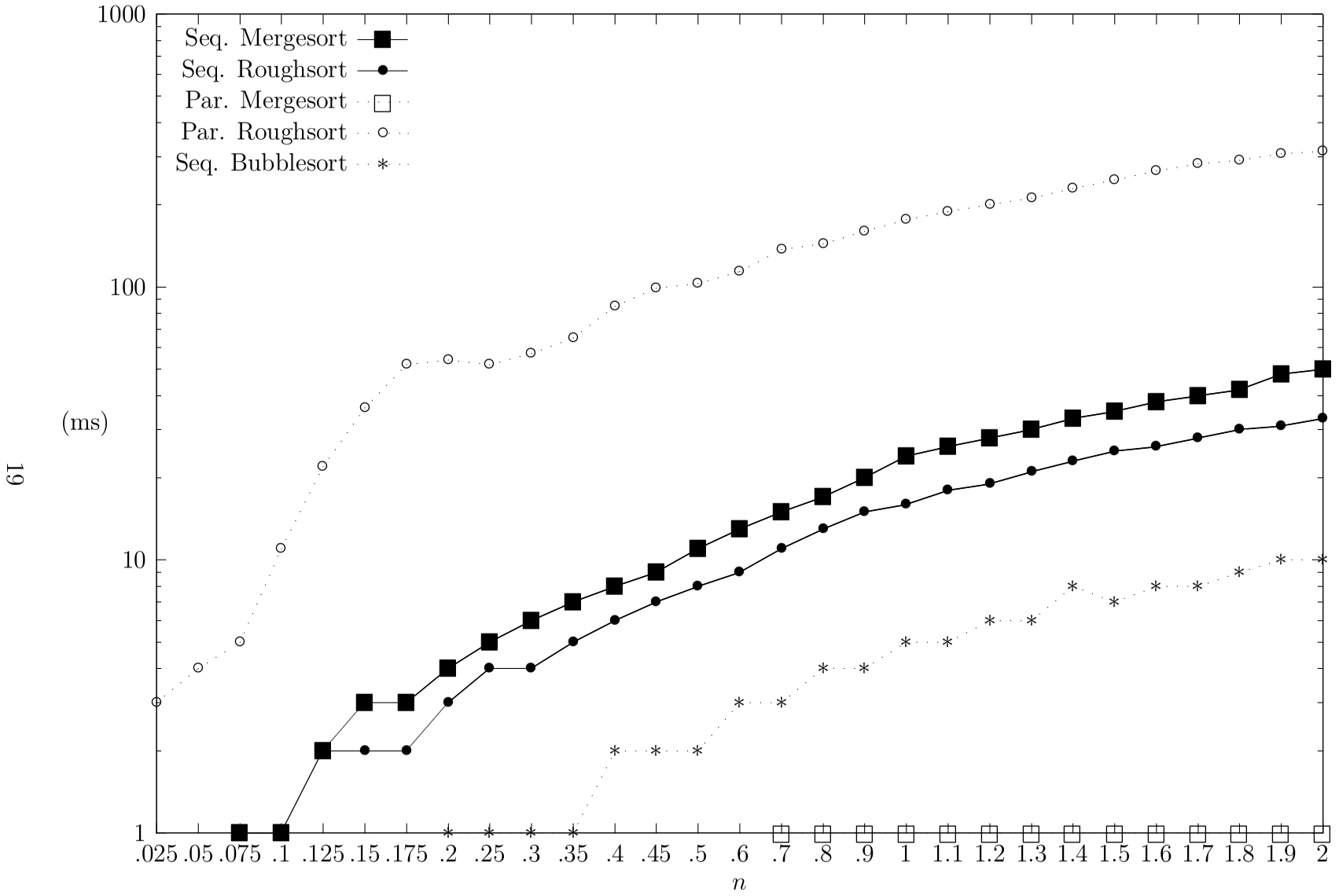

Figure 3: *Sort Runtimes over Arrays of Length* $n \cdot 10^6$, $k = 2$

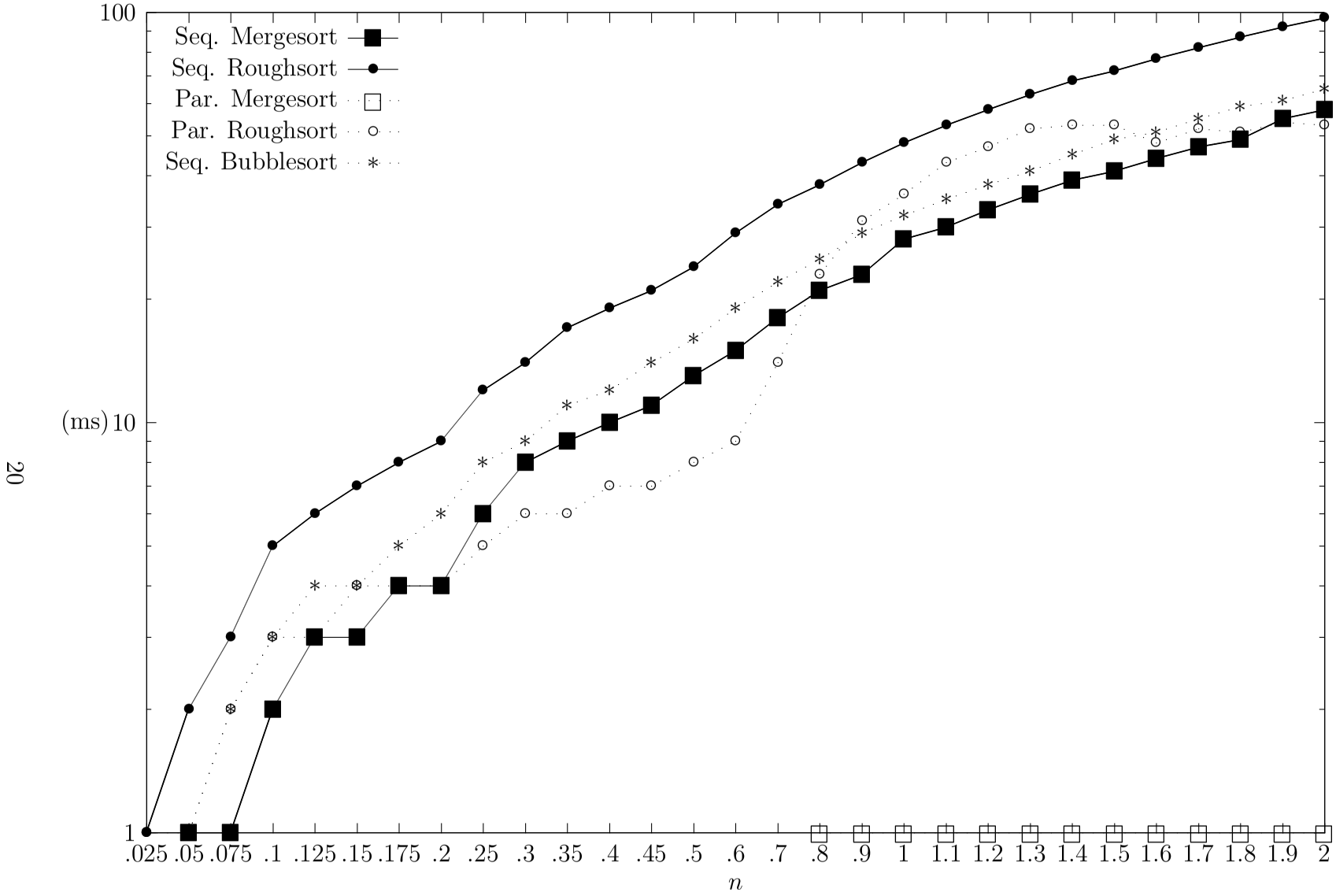

Figure 4: *Sort Runtimes over Arrays of Length* $n \cdot 10^6$, $k = 15$

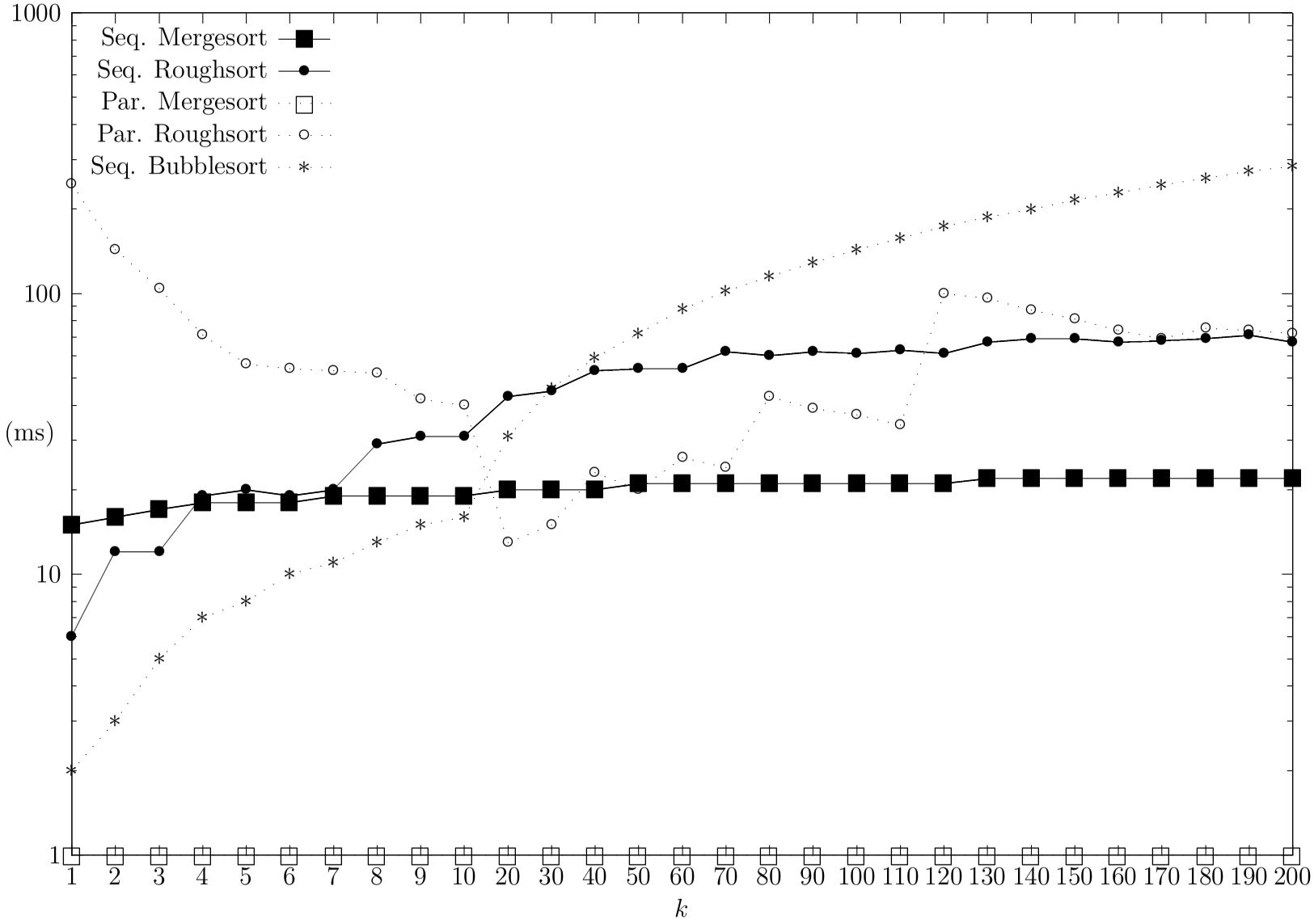

Figure 5: *Sort Runtimes over Arrays of Radius k*, $n = 0.75 \cdot 10^6$

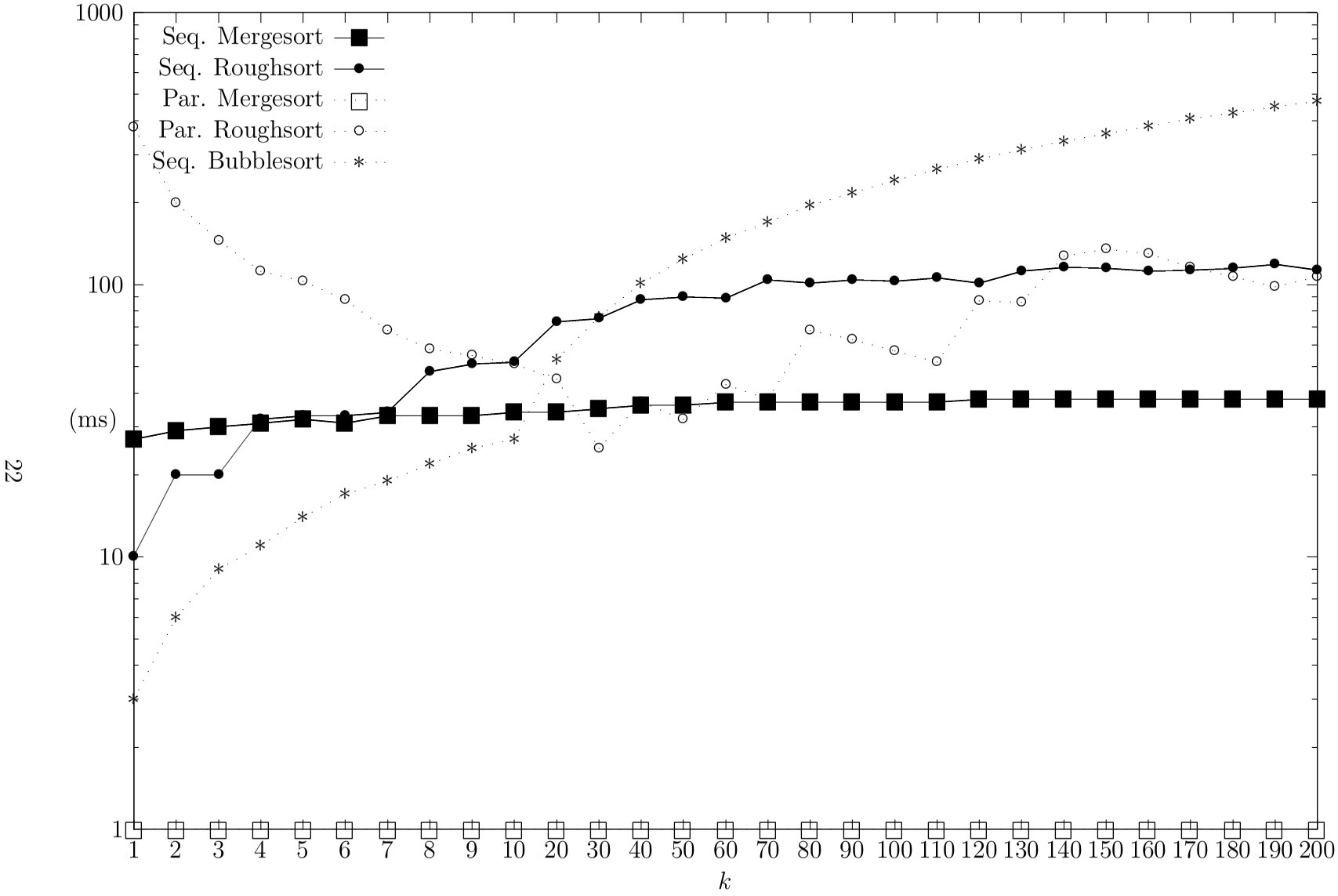

Figure 6: *Sort Runtimes over Arrays of Radius k*, $n = 1.25 \cdot 10^6$

10 Explanation of Results

The source papers theorized that parallel radius finding would run at a $O(\lg(k))$ bound, so these contrary result must then be explained. There are several possible reasons for this which should be considered, as there are similarities but also differences between GPU and the assumed CRCW PRAM. Given many GPU processor threads may update multiple memory locations simultaneously, and likewise many memory locations may be read in parallel on GPU, the underlying NVIDIA hardware that CUDA provides access to is reminiscent of CRCW PRAM. However, GPUs differ from true CRCW PRAM in fundamental ways. First, GPU memory is finite, while PRAM is of arbitrary size. Furthermore, for GPU memory access times to provide the concurrent read and write characteristics reminiscent of PRAM, these memory operations must be performed on contiguous portions of GPU memory in a *coalescing manner*, whereas CRCW PRAM has no such ordering limitation. Foremost in the differences between CRCW PRAM and GPU architectures however is the warp divergence concept addressed above. What this implies for memory operations is that workloads exist which may prevent all but one thread in a warp from performing a memory operation at a particular time. Consequently, there are practical limitations to what a GPU may achieve, and these limitations restrict the workloads which are practical to run on a GPU. Not only must a GPU workload constrain its memory usage within practical limitations, the accesses to that memory must be orderly, and the underlying algorithm performing those accesses must be amenable to parallelization on GPU if the advantages of GPU processing over CPU processing are to be realized [12]. Likewise, while the parallel Roughsort results were more in

line with expectations, that it failed to best the heavily optimized `Thrust` Mergesort implementation confirms that issues impacting parallel radius finding similarly affect parallel Roughsort's performance.

11 Conclusion and Further Research

Ultimately, results for this project did not result in a parallel Roughsort implementation which convincingly bested other comparison sort algorithms. Most problematic was the linear performance of the parallel radius finder, which was expected to run in logarithmic time. Roughsort results were encouraging, but ultimately parallel Mergesort dominated on all tested values of k, demonstrating the impracticality of the parallel Roughsort implementation for workloads which would clearly benefit more from parallel Mergesort. The authors posit that this is largely a consequence of the differences between CRCW PRAM and the practical limitations of Nvidia GPU architectures. In the same vein, sequential Roughsort was bested in the domain of low k values by Bubblesort, due to Bubblesort's expected good performance on nearly sorted sequences, closing off yet another avenue in which Roughsort might have utility.

However, as a first implementation, this contributes to the body of work and provides several future avenues for exploration and possible improvement over these results. The practical considerations of the GPU hardware were not explicitly addressed while architecting and designing parallel implementations of the mentioned algorithms. The dominating `Thrust` Mergesort implementation was once in the same situation, where additional optimizations were discovered and then implemented.

Work may now be done to coalesce memory access requests across threads in the aforementioned payloads. Likewise, optimizations which might have the algorithms take advantage of more performant thread-local memory, specify beneficial cache parameters, avoid warp divergence, and generally address practical constraints of the GPU can now be investigated [17]. Further exploration may also be undertaken into CUDA implementations of CRCW parallel algorithms which might address the limitations seen in these results. For instance, an implementation of Cole's parallel median algorithm might enable a more efficient parallel Roughsort [10].

Finally, this body of work introduces a useful concept: the k value. Not only does determining the radius of a sequence provide for the possibility of selecting the best sorting algorithm for the situation, being able to generate sequences with an arbitrary k value is useful in the testing and evaluation of sorting algorithms, as was done here. Not only could this work be cumulatively used to implement sorting techniques which select the optimal algorithm for the situation based on the k value, evaluation of sorting algorithms becomes more scientific and thus effective.

References

[1] Geforce gtx 1080. http://www.geforce.com/hardware/10series/geforce-gtx-1080#specs. [Online; accessed 8-December-2016].

[2] Intel xeon processor e3-1275 v5 (8m cache, 3.60 ghz) specifications. http://ark.intel.com/products/88177/Intel-Xeon-Processor-E3-1275-v5-8M-Cache-3_60-GHz. [Online; accessed 8-December-2016].

[3] How to access global memory efficiently in cuda c/c++ kernels. https://devblogs.nvidia.com/parallelforall/how-access-global-memory-efficiently-cuda-c-kernels/, January 2013. [Online; accessed 8-December-2016].

[4] thrust: Extrema. https://thrust.github.io/doc/group__extrema.htm, March 2015. [Online; accessed 7-December-2016].

[5] std::nth_element. http://en.cppreference.com/w/cpp/algorithm/nth_element, March 2016. Accessed: 2016-12-07.

[6] Thrust :: Cuda toolkit documentation. http://docs.nvidia.com/cuda/thrust/, September 2016. [Online; accessed 7-December-2016].

[7] T. Altman and B. Chlebus. Sorting roughly sorted sequences in parallel. *Information Processing Letters*, 33(6), February 1990.

[8] T. Altman and Y. Igarashi. Roughly sorting: Sequential and parallel approach. *Journal of Information Processing*, 12(2), 1989.

[9] J. Cheng, M. Grossman, and T. McKercher. *Professional CUDA C Programming*. Wrox, 1st edition, September 2014.

[10] R. Cole and C. Yap. A parallel median algorithm. *Information Processing Letters*, 20(3), April 1985.

[11] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, 3rd edition, 2009.

[12] Frank Dehne and Kumanan Yogaratnam. Exploring the limits of gpus with parallel graph algorithms. *arXiv preprint arXiv:1002.4482*, 2010.

[13] Forsell and Martti. On the performance and cost of some pram models on cmp hardware. *International Journal of Foundations of Computer Science*, 21(03):387–404, 2010.

[14] Intel. *Intel Digital Random Generator (DRNG) Software Implementation Guide*, 1.1 edition, August 2012.

[15] D. E. Knuth. *The Art of Computer Programming*, volume 3. Addison-Wesley, 2nd edition, 1998.

[16] N. Leischner, V. Osipov, and P. Sanders. GPU sample sort. September 2009.

[17] NVIDIA Corporation. CUDA C Programming Guide. `https://docs.nvidia.com/cuda/cuda-c-programming-guide`, September 2016. Accessed: 2016-12-06.

[18] OrangeOwl. Sorting with CUDA libraries. `http://www.orangeowlsolutions.com/archives/900`, May 2014. Accessed: 2016-12-06.

[19] S. Rajasekaran and J. H. Reif. Optimal and sublogarithmic time randomized parallel sorting algorithms. *SIAM Journal on Computing*, 18(3):594–607, 1989.

[20] R. Senser. *GPU Declarative Framework*. PhD thesis, University of Colorado Denver, November 2014.

[21] Y. Shiloach and U. Vishkin. *Finding the maximum, merging and sorting in a parallel computation model*. Springer Berlin Heidelberg, Berlin, Heidelberg, 1981.

[22] Y. Shiloach and U. Vishkin. An O(log n) parallel connectivity algorithm. *Journal of Algorithms*, 3(1):57–67, 1982.

[23] N. Wilt. *The CUDA Handbook: A Comprehensive Guide to GPU Programming*. Addison-Wesley Professional, 1st edition, June 2013.