Parallel Sorting of Roughly-Sorted Sequences

CSCI 5172 Fall '16 Project

Anthony Pfaff, Jason Treadwell

December 7, 2016

Sorting a collection according to some ordering among its items is among the most classic problems of computer science. A well-established result is the linearithmic (i.e. $O(n \lg n)$) optimal upper bound for sorting sequences of length n by comparison. Commonly-used sorting algorithms such as Mergesort and Quicksort have $\Theta(n \lg n)$ and $\Omega(n \lg n)$ runtimes, respectively.

We can exploit the ordering of *roughly-sorted* sequences to sort them in $O(n \lg k)$ time, where k is the *radius* of a sequence S or the smallest k such that S is k-*sorted.*[4] A k-sorted sequence $\{a_0, a_1, \cdots, a_n\}$ satisfies $a_i \leq a_j \ \forall \ 1 \leq i \leq j \leq n, i \leq j - k$. Since an unsorted sequence can be at most n-sorted, the worst-case runtime of this algorithm has complexity $O(n \lg n)$.

We intend to study and implement sorting of roughly-sorted sequences using both sequential and parallel[3] versions of the algorithm. In particular, I shall investigate the applicability of the method over very large sequences, accelerated using graphics

hardware. A GPU is dedicated graphics hardware that also excels at performing frequent, basic computations over large amounts of data. The use of GPUs for numerical applications is a recent and rapidly-growing development in computer science and software engineering. Suitable hardware is now commonly available in consumer devices and may herald a paradigm shift in how we efficiently perform common and indispensable tasks like sorting.

I propose to develop implementations of the method using C and Nvidia's CUDA framework, comparing the results with sequential implementations running on both the GPU and CPU as well as with the classic Mergesort and Quicksort algorithms. If time permits, I would also like to compare the method to parallel versions of the latter two algorithms.

The aforementioned algorithms assume a specific underlying memory in achieving the provided 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>p is available to p processors, and up to p positions in this memory may be simultaneously read or written in one step[13]. As with roughsort, a number of algorithms take advantage of this model's assumptions to provide efficiency improvements over their serial counterparts[14][12].

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 [Jason's CPU information] on a CPU, versus [Jason's GPU information] on a GPU.

Nvidia–the manufacturuer of the aforemeitoned 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/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 memory locations is a useful component of the overall program's payload. Nvidia CUDA provides simple mechanisms by which to achieve this.

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 the same processor core on 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, 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 divvyed 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[1] 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).[?]

Using these 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 sequences were developed. This was done by assigning each thread a unique identifier and then, analagous to Altman and Chlebus'work, having each thread manage the corresponding slot in the global unsorted array [4]. 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 other threads.

Once the RL and LR lists were completed, the DM list was computed by using Altman et. al.'s serial radius techniques, while using the log(n) 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 B[i] (the ith element of the LR list) was always greater than or equal to C[i] (the ith element of the RL list). As discussed, only the power of 2 greater than the actual

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

value of k is actually needed for parallel roughsort to function, so this criteria was removed in the interest of efficiency and simplicity[4].

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++ Standard Template Library (STL) functions[2]. One such STL function is the std::max_element function. Using the Thrust implementation of this function, maximum elements of large arrays can be found expediently on Nvidia hardware using CUDA[1]. 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.

[graphs here]

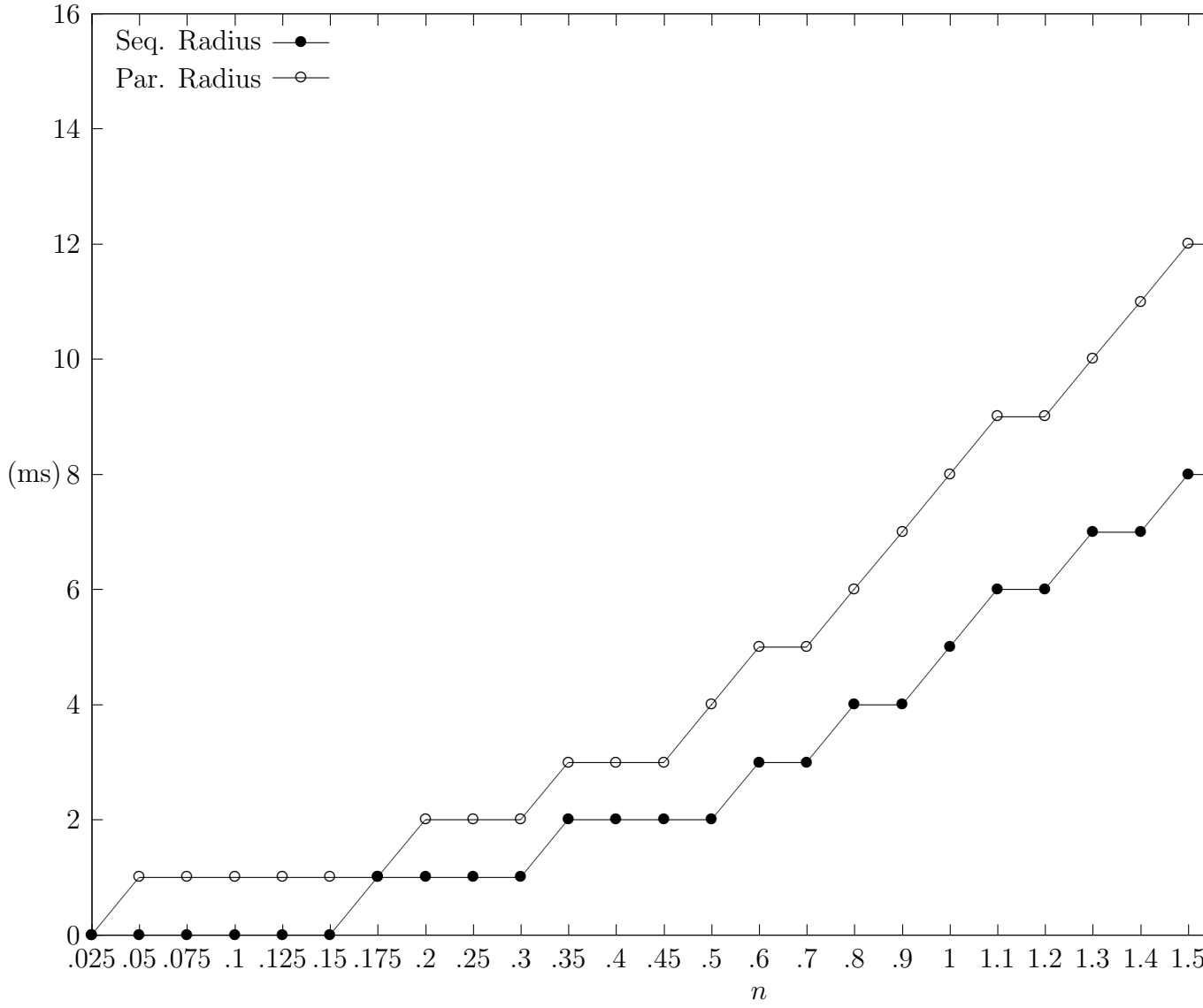
Running these kernels yielded disappointing results, however. For random lists composed together in such a manner as to yield a specific k value for the list[2], sequential CPU computations exceeded performance on GPU for any tested n when values of k exceeded 6, as see in Figure 2.[3]. Figure 1 reports 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 a roughly linear improvement over the sequential implementation.

The source papers theorized that parallel algorithms would run at a O(log(n))

[2]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

[3]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 suspect that optimization differences on the C++ STL nth_element function are the cause. This begets further investigation, however

bound, so these contrary result must then be explained. There are several possible reasons for this which should be considered, as there are similartities 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 location 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, 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 access 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[7].


Seq. Radius
Par. Radius
16
14
12
10
8
6
4
2
0
(ms)
.025 .05 .075 .1 .125 .15 .175 .2 .25 .3 .35 .4 .45 .5 .6 .7 .8 .9 1 1.1 1.2 1.3 1.4 1.5
n

References

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

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

[3] Tom Altman and Bogdan Chlebus. Sorting roughly sorted sequences in parallel. *Information Processing Letters*, 33(6), February 1990.

[4] Tom Altman and Yoshihide Igarashi. Roughly sorting: Sequential and parallel approach. *Journal of Information Processing*, 12(2), 1989.

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

[6] Gavin Clarke. Torvalds shoots down call to yank 'backdoored' Intel RdRand in Linux crypto. `http://www.theregister.co.uk/2013/09/10/torvalds_on_rrrand_nsa_gchq/`, September 2013. [Online; accessed 20-April-2016].

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

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

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

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

[11] George Marsaglia. Xorshift rngs. *Journal of Statistical Software*, 8(14), July 2003.

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

[13] Yossi Shiloach and Uzi Vishkin. *Finding the maximum, merging and sorting in a parallel computation model.* Springer Berlin Heidelberg, Berlin, Heidelberg, 1981.

[14] Yossi Shiloach and Uzi Vishkin. An o(logn) parallel connectivity algorithm. *Journal of Algorithms*, 3(1):57 – 67, 1982.

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