Parallel Sorting for GPUs

Frank Dehne and Hamidreza Zaboli

Abstract Selim Akl has been a ground breaking pioneer in the field of parallel sorting algorithms. His 'Parallel Sorting Algorithms' book [12], published in 1985, has been a standard text for researchers and students. Here we discuss recent advances in parallel sorting methods for many-core GPUs. We demonstrate that parallel *deterministic* sample sort for GPUs (GPU BUCKET SORT) is not only considerably faster than the best comparison-based sorting algorithm for GPUs (THRUST MERGE) but also as fast as *randomized* sample sort for GPUs (GPU SAMPLE SORT). However, *deterministic* sample sort has the advantage that bucket sizes are guaranteed and therefore its running time does not have the input data dependent fluctuations that can occur for *randomized* sample sort.

1 Introduction

Selim Akl has been a ground breaking pioneer in the field of parallel sorting algorithms. His 'Parallel Sorting Algorithms' book [12], published in 1985, has been a standard text for researchers and students. Here we discuss recent advances in parallel sorting methods for many-core GPUs.

Modern graphics processors (*GPU*s) have evolved into highly parallel and fully programmable architectures. Current many-core GPUs can contain hundreds of processor cores on one chip and can have an astounding performance. However, GPUs are known to be hard to program and current general purpose (i.e. non-graphics) GPU applications concentrate typically on problems that can be solved using fixed and/or regular data access patterns such as image processing, linear algebra, physics simulation, signal processing and scientific computing (see e.g. [7]). The design of efficient GPU methods for discrete and combinatorial problems with

F. Dehne (✉)
Carleton University, Ottawa, Canada
e-mail: frank@dehne.net

H. Zaboli
IBM, Ottawa, Canada
e-mail: hamedzaboli@gmail.com

data dependent memory access patterns is still in its infancy. The comparison-based THRUST MERGE method [11] by Nadathur Satish, Mark Harris and Michael Garland of nVIDIA Corporation was considered the best sorting method for GPUs. Nikolaj Leischner, Vitaly Osipov and Peter Sanders [9] recently published a *randomized* sample sort method for GPUs (referred to as GPU SAMPLE SORT) that significantly outperforms THRUST MERGE. However, a disadvantage of the *randomized* sample sort method is that its performance can vary for different input data distributions because the data is partitioned into buckets that are created via *randomly* selected data items. Here we demonstrate that *deterministic* sample sort for GPUs, referred to as GPU BUCKET SORT, has the same performance as the *randomized* sample sort method (GPU SAMPLE SORT) in [9].

The remainder of this paper is organized as follows. Section 2 reviews some features of GPUs that are important in this context. Section 3 reviews recent GPU based sorting methods. Section 4 outlines GPU BUCKET SORT and discusses some details of our CUDA [1] implementation. In Sect. 5, we present an experimental performance comparison between our GPU BUCKET SORT implementation, the randomized GPU SAMPLE SORT implementation in [9], and the THRUST MERGE implementation in [11].

2 Review: GPU Architectures

As in [9, 11], we will focus on nVIDIA's unified graphics and computing platform for GPUs [10] and associated *CUDA* programming model [1]. However, the discussion applies more generaly to GPUs that support the OpenCL standard [2]. A GPU consists of an array of streaming processors called *Streaming Multiprocessors (SM*s). Each SM contains several processor cores and a small size low latency local *shared memory* that is shared by its processor cores. All SMs are connected to a *global* DRAM *memory* through an interconnection network. The global memory is arranged in independent memory partitions and the interconnection network routes the read/write memory requests from the processor cores to the respective global memory partitions, and the results back to the cores. Each global memory partition has its own queue for memory requests and arbitrates among the incoming read/write requests, seeking to maximize DRAM transfer efficiency by grouping read/write accesses to neighboring memory locations (referred to as *coalesced* global memory access). Memory latency to global DRAM memory is optimized when parallel read/write operations can be grouped into a minimum number of sub-arrays of contiguous memory locations.

It is important to note that data accesses from processor cores to their SM's local shared memory are at least an order of magnitude faster than accesses to global memory. This is our main motivation for using a sample sort based approach. An important property of sample sort is that the number of times the data has to be accessed in global memory is a small fixed constant. At the same time, *deterministic* sample sort provides a partitioning into independent parallel workloads and also gives guarantees for the sizes of those workloads. For GPUs, this implies that we are able to utilize the local shared memories efficiently and that the number of data transfers between gloabl memory and the local shared memories is a small fixed constant.

Another critical issue for the performance of CUDA implementations is conditional branching. CUDA programs typically execute very large numbers of threads. In fact, a large number of threads is required for hiding latencies of global memory accesses. The GPU has a hardware thread scheduler that is built to manage tens of thousands and even millions of concurrent threads. All threads are divided into blocks, and each block is executed by an SM. An SM executes a thread block by breaking it into groups called *warps* and executing them in parallel. The cores within an SM share various hardware components, including the instruction decoder. Therefore, the threads of a warp are executed in SIMT (single instruction, multiple threads) mode, which is a slightly more flexible version of the standard SIMD (single instruction, multiple data) mode. The main problem arises when the threads encounter a conditional branch such as an IF-THEN-ELSE statement. Depending on their data, some threads may want to execute the code associated with the "true" condition and some threads may want to execute the code associated with the "false" condition. Since the shared instruction decoder can only handle one branch at a time, different threads can not execute different branches concurrently. They have to be executed in sequence, leading to performance degradation. Recent GPUs provide a small improvement through an instruction cache at each SM that is shared by its cores. This allows for a "small" deviation between the instructions carried out by the different cores. For example, if an IF-THEN-ELSE statement is short enough so that both conditional branches fit into the instruction cache then both branches can be executed fully in parallel. However, a poorly designed algorithm with too many and/or large conditional branches can result in serial execution and very low performance.

3 GPU Sorting Methods

Early sorting algorithms for GPUs include *GPUTeraSort* [6] based on bitonic merge, and *Adaptive Bitonic Sort* [8] based on a method by Bilardi et al. [3]. *Hybrid Sort* [14] used a combination of bucket sort and merge sort, and Cederman et al. [4] proposed a quick sort based method for GPUs. Both methods [4, 14] suffer from load balancing problems. Until recently, the comparison-based THRUST MERGE method [11] by Nadathur Satish, Mark Harris and Michael Garland of nVIDIA Corporation was considered the best sorting method for GPUs. THRUST MERGE uses a combination of odd-even merge and two-way merge, and overcomes the load balancing problems mentioned above. Satish et al. [11] also presented an even faster GPU radix sort method for the special case of integer sorting. Yet, a recent paper by Nikolaj Leischner, Vitaly Osipov and Peter Sanders [9] presented a randomized sample sort method for GPUs (GPU SAMPLE SORT) that significantly outperforms THRUST MERGE [11]. However, as also discussed in Sect. 1, the fact that GPU

Sample Sort is a *randomized* method implies that its performance can vary with the distribution of the input data because buckets are created through randomly selected data items. For example, the performance analysis presented in [9] measures the runtime of GPU Sample Sort for several input data distributions to document the performance variations observed for different input distributions.

4 GPU Bucket Sort*: Deterministic* Sample Sort For GPUs

In this section we outline GPU Bucket Sort, a *deterministic* sample sort algorithm for GPUs. An overview of GPU Bucket Sort is shown in Algorithm 1 below. It consists of a local sort (Step 1), a selection of samples that define balanced buckets (Steps 3–5), moving all data into those buckets (Steps 6–8), and a final sort of each bucket (Step 9). In our implementation of GPU Bucket Sort we introduced several adaptations to the structure of GPUs, in particular the two level memory hierarchy, the large difference in memory access times between those two levels, and the small size of the local shared memories. We experimented with several bucket sizes and number of samples in order to best fit them to the GPU memory structure. For sorting the selected sample and the bottom level sorts of the individual buckets, we experimented with several existing GPU sorting methods such as bitonic sort, adaptive bitonic sort [8] based on [3], and parallel quick sort.

The following discussion of our implementation of GPU Bucket Sort will focus on GPU performance issues related to shared memory usage, coalesced global memory accesses, and avoidance of conditional branching. Consider an input array A with n data items in global memory and a typical local shared memory size of $\frac{n}{m}$ data items.

In **Steps 1 and 2** of Algorithm 1, we split the array A into m sublists of $\frac{n}{m}$ data items each and then locally sort each of those m sublists. More precisely, we create m thread blocks of 512 threads each, where each thread block sorts one sublist using one SM. Each thread block first loads a sublist into the SM's local shared memory using a coalesced parallel read from global memory. Note that, each of the 512 threads is responsible for $\frac{n}{m}/512$ data items. The thread block then sorts a sublist of $\frac{n}{m}$ data items in the SM's local shared memory. We tested different implementations for the local shared memory sort within an SM, including quicksort, bitonic sort, and adaptive bitonic sort [3]. In our experiments, bitonic sort was consistently the fastest method, despite the fact that it requires $O(n \log^2 n)$ work. The reason is that, for Step 2 of Algorithm 1, we always sort a small fixed number of data items, independent of n. For such a small number of items, the simplicity of bitonic sort, it's small constants in the running time, and it's perfect match for SIMD style parallelism outweigh the disadvantage of additional work.

In **Step 3** of Algorithm 1, we select s equidistant samples from each sorted sublist. (The implementation of Step 3 is built directly into the final phase of Step 2 when the sorted sublists are written back into global memory.) Note that, the sample size s is a free parameter that needs to be tuned. With increasing s, the sizes of buckets created in Step 8 decrease and the time for sorting those buckets (Step 9) decreases as well. However, the time for managing the buckets (Steps 3–7) grows with increasing s. This trade-off will be studied in Sect. 5 where we show that $s = 64$ provides the best performance. In **Step 4**, we sort all sm selected samples in global memory, using all available SMs in parallel. Here, we compared GPU bitonic sort [6], adaptive bitonic sort [8] based on [3], and GPU SAMPLE SORT [9]. Our experiments indicate that for up to 16 M data items, simple bitonic sort is still faster than even GPU SAMPLE SORT [9] due to its simplicity, small constants, and complete avoidance of conditional branching. Hence, Step 4 was implemented via bitonic sort. In **Step 5**, we again select s equidistant *global samples* from the sorted list of sm samples. Here, each thread block/SM loads the s global samples into its local shared memory where they will remain for the next step.

Input: An array A with n data items stored in global memory.
Output: Array A sorted.

1. Split the array A into m sublists A_1, ..., A_m containing $\frac{n}{m}$ items each where $\frac{n}{m}$ is the shared memory size at each SM.
2. *Local Sort*: Sort each sublist A_i (i=1,..., m) locally on one SM, using the SM's shared memory as a cache.
3. *Local Sampling*: Select s equidistant samples from each sorted sublist A_i (i=1,..., m) for a total of sm samples.
4. *Sorting All Samples*: Sort all sm samples in global memory, using all available SMs in parallel.
5. *Global Sampling*: Select s equidistant samples from the sorted list of sm samples. We will refer to these s samples as *global samples*.
6. *Sample Indexing*: For each sorted sublist A_i (i=1,..., m) determine the location of each of the s global samples in A_i. This operation is done for each A_i locally on one SM, using the SM's shared memory, and will create for each A_i a partitioning into s buckets A_{i1},..., A_{is} of size a_{i1},..., a_{is}.
7. *Prefix Sum*: Through a parallel prefix sum operation on a_{11},..., a_{m1}, a_{12},..., a_{m2}, ..., a_{1s},..., a_{ms} calculate for each bucket $A_{ij}(1 \leq i \leq m, 1 \leq j \leq s,)$ its starting location l_{ij} in the final sorted sequence.
8. *Data Relocation*: Move all sm buckets $A_{ij}(1\leq i \leq m, 1\leq j \leq s)$ to location l_{ij}. The newly created array consists of s sublists B_1, ..., B_s where $B_j = A_{1j} \cup A_{2j} \cup ... \cup A_{mj}$ for $1\leq j \leq s$.
9. *Sublist Sort*: Sort all sublists B_j, $1\leq j \leq s$, using all SMs.

Algorithm 1: GPU BUCKET SORT (Deterministic Sample Sort For GPUs)

In **Step 6**, we determine for each sorted sublist A_i (i=1, ..., m) of $\frac{n}{m}$ data items the location of each of the s global samples in A_i. For each A_i, this operation is done locally by one thread block on one SM, using the SM's shared memory, and will create for each A_i a partitioning into s buckets A_{i1},..., A_{is} of size a_{i1},..., a_{is}. Here, we apply a parallel binary search algorithm to locate the global samples in A_i. More precisely, we first take the $\frac{s}{2}$-th global sample element and use one thread to perform a binary search in A_i, resulting in a location $l_{s/2}$ in A_i. Then we use two threads to perform two binary searches in parallel, one for the $\frac{s}{4}$-th global sample element in the part of A_i to the left of location $l_{s/2}$, and one for the $\frac{3s}{4}$-th global sample element in the part of A_i to the right of location $l_{s/2}$. This process is iterated $\log s$ times until all s global samples are located in A_i. With this, each A_i is split into s buckets A_{i1},..., A_{is} of size a_{i1},..., a_{is}. Note that, we do not simply perform all s binary searches fully in parallel in order to avoid memory contention within the local shared memory [1].

Step 7 uses a prefix sum calculation to obtain for all buckets their starting location in the final sorted sequence. The operation is illustrated in Fig. 1 and can be implemented with coalesced memory accesses in global memory. Each row in Fig. 1 shows the a_{i1},..., a_{is} calculated for each sublist. The prefix sum is implemented via a parallel column sum (using all SMs), followed by a prefix sum on the columns sums (on one SM in local shared memory), and a final update of the partial sums in each column (using all SMs).

In **Step 8**, the sm buckets are moved to their correct location in the final sorted sequence. This operation is perfectly suited for a GPU and requires one parallel coalesced data read followed by one parallel coalesced data write operation. The newly created array consists of s sublists B_1, ..., B_s where each $B_j = A_{1j} \cup A_{2j} \cup \ldots \cup A_{mj}$ has at most $\frac{2n}{s}$ data items [13]. In **Step 9**, we sort each B_j using the same bitonic sort implementation as in Step 4. We observed that for our choice of s, each B_j contains at most $4M$ data items. For such small data sets, simple bitonic sort is again the fastest sorting algorithm for each B_j due to bitonic sort's simplicity, small constants, and complete avoidance of conditional branching.

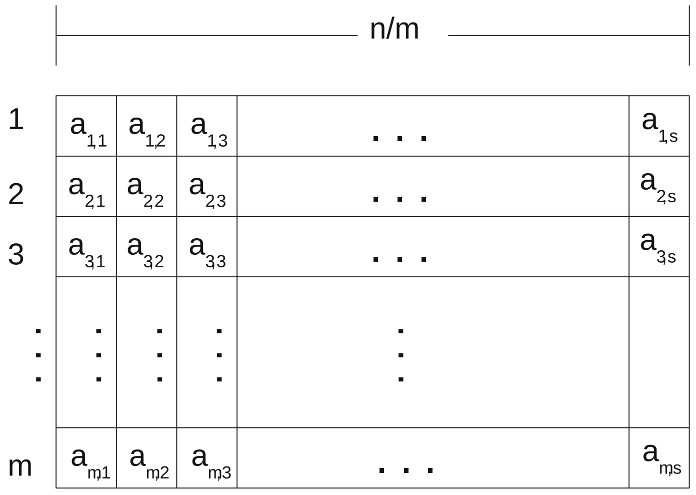
n/m
1
2
3
m
a1,1 a1,2 a1,3 . . . a1,s
a2,1 a2,2 a2,3 . . . a2,s
a3,1 a3,2 a3,3 . . . a3,s
am,1 am,2 am,3 . . . am,s

Fig. 1 Illustration of step 7 in Algorithm 1

5 Experimental Results and Discussion

Figure 2 shows in detail the time required for the individual steps of Algorithm 1 when executed on an NVIDIA Fermi GTX 480 GPU. We observe that *sublist sort* (Step 9) and *local sort* (Step 2) represent the largest portion of the total runtime of GPU BUCKET SORT. This is very encouraging in that the "overhead" involved to manage the deterministic sampling and generate buckets of guaranteed size (Steps 3–7) is small. We also observe that the *data relocation* operation (Step 8) is very efficient and a good example of the GPU's great performance for data parallel access when memory accesses can be coalesced.

Figures 3 and 4 show a comparison between GPU BUCKET SORT and the current best GPU sorting methods, *randomized* GPU SAMPLE SORT [9] and THRUST MERGE [11] on a NVIDIA Tesla C1060 GPU. For GPU BUCKET SORT, all runtimes are the averages of 100 experiments, with less than 1 ms observed variance. For *randomized* GPU SAMPLE SORT and THRUST MERGE, the runtimes shown are the ones reported in [9, 11]. For THRUST MERGE, performance data is only available for up to $n = 16$ M data items. For larger values of n, the current THRUST MERGE code shows memory errors [5]. As reported in [9], the current *randomized* GPU SAMPLE SORT code can sort up to 128 M data items on a Tesla C1060. Our GPU BUCKET SORT implementation appears to be more memory efficient. GPU BUCKET SORT can sort up to $n = 512$ M data items on a Tesla C1060. Figures 3 and 4 show the performance comparison with higher resolution for up to $n = 128$ M and for the entire range up to $n = 512$ M, respectively. We observe that, as reported in [9], *randomized* GPU SAMPLE SORT [9] significantly outperforms THRUST MERGE [11]. Most importantly, we observe that *randomized* sample sort (GPU SAMPLE SORT) [9] and *deterministic* sample sort (GPU BUCKET SORT) show nearly identical performance.

The data sets used for the performance comparison in Figs. 3 and 4 were uniformly distributed, random data items. The data distribution does not impact the performance

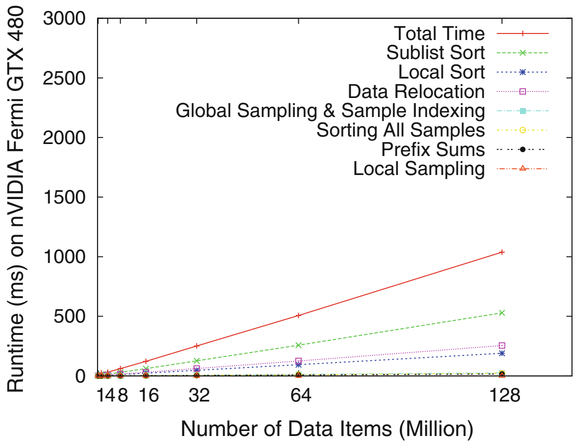

Fig. 2 Performance of deterministic sample sort for GPUs (GPU BUCKET SORT). Total runtime and runtime for individual steps of Algorithm 1 for varying number of data items

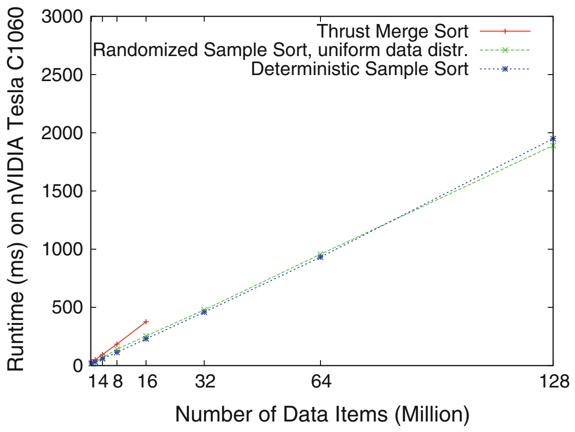

Fig. 3 Comparison between deterministic sample sort (GPU BUCKET SORT), Randomized Sample Sort (GPU SAMPLE SORT) [9] and THRUST MERGE [11]. Total runtime for varying number of data items up to 128,000,000. (*Note* [11] and [9] provided data only for up to 16 and 128 M data items, respectively.)

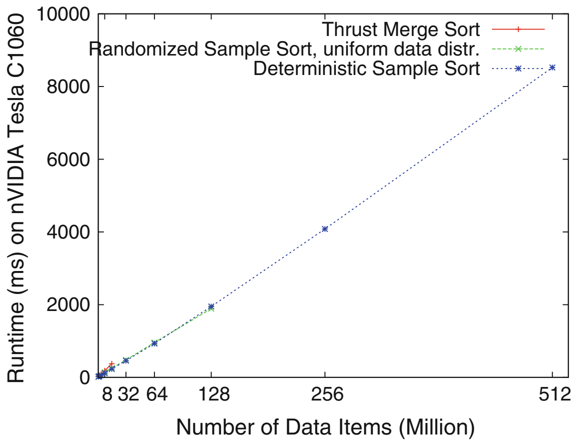

Fig. 4 Comparison between deterministic sample sort (GPU BUCKET SORT), Randomized sample sort (GPU SAMPLE SORT) [9] and THRUST MERGE [11]. Total runtime for varying number of data items up to 512,000,000. (*Note* [11] and [9] provided data only for up to 16 and 128 M data items, respectively.)

of *deterministic* sample sort (GPU BUCKET SORT) but has a considerable impact on the performance of *randomized* sample sort (GPU SAMPLE SORT) [9]. In fact, the uniform data distribution used for Figs. 3 and 4 is a *best case* scenario for *randomized* sample sort where all bucket sizes are nearly identical. Figure 5 shows that our deterministic sample sort (GPU BUCKET SORT) is stable under different types of data distribution. We tested three types of data distribution: Uniform, Gaussian, and Zipf. As seen in the figure, different input data distributions have little influence on the performance of GPU BUCKET SORT.

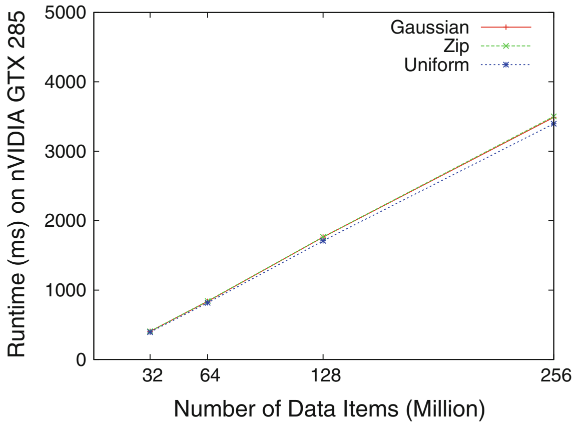

Fig. 5 Performance of deterministic sample sort (GPU BUCKET SORT) for different input data distributions

6 Conclusions

In this paper, we presented a *deterministic* sample sort algorithm for GPUs, called GPU BUCKET SORT. Our experimental evaluation indicates that GPU BUCKET SORT is considerably faster than THRUST MERGE [11], the best comparison-based sorting algorithm for GPUs, and it is exactly as fast as *randomized* sample sort for GPUs (GPU SAMPLE SORT) [9] when the input data sets used are uniformly distributed, which is a *best case* scenario for randomized sample sort. However, as observed in [9], the performance of *randomized* GPU SAMPLE SORT fluctuates with the input data distribution whereas GPU BUCKET SORT does not show such fluctuations.

References

1. NVIDIA CUDA Programming Guide. nVIDIA Corporation. www.nvidia.com
2. The OpenCL Specification 1.0. Khronos OpenCL Working Group (2009)
3. Bilardi, G., Nicolau, A.: Adaptive bitonic sorting. An optimal parallel algorithm for shared-memory machines. SIAM J. Comput. **18**(2), 216–228 (1989)
4. Cederman, D., Tsigas, P.: A practical quicksort algorithm for graphics processors. In: Proceedings of European Symposium on Algorithms (ESA), vol. 5193 of LNCS, pp. 246–258 (2008)
5. Garland, M.: Private Communication. nVIDIA Corporation (2010)
6. Govindaraju, N., Gray, J., Kumar, R., Manocha, D.: GPUTeraSort: high performance graphics co-processor sorting for large database management. In: Proceedings of International Conference on Management of Data (SIGMOD), pp. 325–336 (2006)
7. GPGPU.ORG. General-purpose computation on graphics hardware
8. Greb, A., Zachmann, G.: GPU-ABiSort: optimal parallel sorting on stream architectures. In: Proceedings of International Parallel and Distributed Processing Symposium (IPDPS) (2006)
9. Leischner, N., Osipov, V., Sanders, P.: GPU sample sort. In: Proceedings of International Parallel and Distributed Processing Symposium (IPDPS), pp. 1–10 (2010)

10. Lindholm, E., Nickolls, J., Oberman, S., Montrym, J.: NVIDIA Tesla: a unified graphics and computing architecture. IEEE Micro. **28**(2), 39–55 (2008)
11. Satish, N., Harris, M., Garland, M.: Designing efficient sorting algorithms for manycore GPUs. In: Proceedings of International Parallel and Distributed Processing Symposium (IPDPS) (2009)
12. Selim, G.A.: Parallel Sorting Algorithms. Academic Press (1985)
13. Shi, H., Schaeffer, J.: Parallel sorting by regular sampling. J. Par. Dist. Comp. **14**, 362–372 (1992)
14. Sintorn, E., Assarsson, U.: Fast parallel GPU-sorting using a hybrid algorithm. J. Parallel Distrib. Comput. **68**(10), 1381–1388 (2008)