Parallel Processing Letters
Vol. 22, No. 3 (2012) 1250008 (14 pages)
© World Scientific Publishing Company
DOI: 10.1142/S0129626412500089

World Scientific
www.worldscientific.com

DETERMINISTIC SAMPLE SORT FOR GPUS*

FRANK DEHNE

School of Computer Science, Carleton University
Ottawa, Canada, www.dehne.net, frank@dehne.net

and

HAMIDREZA ZABOLI

School of Computer Science, Carleton University
Ottawa, Canada, hzaboli@connect.carleton.ca

Received February 2011
Revised August 2011, December 2011
Published 10 July 2012
Communicated by S. Sahni

ABSTRACT

We demonstrate that parallel *deterministic* sample sort for many-core GPUs (GPU Bucket Sort) is not only considerably faster than the best comparison-based sorting algorithm for GPUs (Thrust Merge [Satish et.al., Proc. IPDPS 2009]) but also as fast as *randomized* sample sort for GPUs (GPU Sample Sort [Leischner et.al., Proc. IPDPS 2010]). 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.

Keywords: Parallel Algorithms, GPU, Sorting.

1. Introduction

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. [8]). The design of

*Research partially supported by the Natural Sciences and Engineering Research Council of Canada (NSERC) and the IBM Center for Advanced Studies Canada.

efficient GPU methods for discrete and combinatorial problems with data dependent memory access patterns is still in its infancy. In fact, there is currently still a debate even on the best *sorting* method for GPUs (e.g. [13, 11]). The comparison-based THRUST MERGE method [13] by Nadathur Satish, Mark Harris and Michael Garland of nVIDIA Corporation was considered the best sorting method for GPUs until Nikolaj Leischner, Vitaly Osipov and Peter Sanders [11] recently published a *randomized* sample sort method for GPUs (referred to as GPU SAMPLE SORT in the remainder) 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. In this paper, 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 [11].

Our experimental performance comparison shows that for uniform data distribution, which is the best case for *randomized* GPU SAMPLE SORT, *deterministic* GPU BUCKET SORT is *exactly* as fast as GPU SAMPLE SORT. However, in contrast to GPU SAMPLE SORT, the performance of the *deterministic* GPU BUCKET SORT method remains the same for any input data distribution because buckets are created deterministically and bucket sizes are guaranteed.

The main contribution of this paper is *not* a particularly novel algorithmic technique. In fact, the sorting problem has been studied for so long that most recently published sorting techniques for new parallel architectures are combinations/adaptations of previously studied algorithmic technique. This is the case for THRUST MERGE [13], GPU SAMPLE SORT[11] and the GPU BUCKET SORT method studied in this paper. The contribution of this paper is to show that a combination of known *deterministic* sample sort techniques, adapted to GPU computing, improves on GPU SAMPLE SORT[11].

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 previous work on GPU based sorting. Section 4 outlines GPU BUCKET SORT and discusses some details of our CUDA [1] implementation. In Section 5, we present an experimental performance comparison between our GPU BUCKET SORT implementation, the randomized GPU SAMPLE SORT implementation in [11], and the THRUST MERGE implementation in [13].

2. Review: GPU Architectures

As in [13] and [11], we will focus on nVIDIA's unified graphics and computing platform for GPUs [12] and associated *CUDA* programming model [1]. However, the discussion applies more generaly to GPUs that support the OpenCL standard [3]. A GPU consists of an array of streaming processors called *Streaming Multiprocessors (SMs)*. 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. Previous Work on GPU Sorting

Sorting algorithms for GPUs started to appear a few years ago and have been highly competitive. Early results include *GPUTeraSort* [7] based on bitonic merge, and *Adaptive Bitonic Sort* [9] based on a method by Bilardi et.al. [4]. *Hybrid Sort* [16] used a combination of bucket sort and merge sort, and D. Cederman et.al. [5] proposed a quick sort based method for GPUs. Both methods [16, 5] suffer from load balancing problems. Until recently, the comparison-based THRUST MERGE method [13] 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. [13] 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 [11] presented a randomized sample sort method for GPUs (GPU SAMPLE SORT) that significantly outperforms THRUST MERGE [13]. However, as also discussed in Section 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 [11] measures the runtime of GPU SAMPLE SORT for several input data distributions to document the performance variations observed for different input distributions.

During the preparation of this paper, a new publication [10] introduced an in-place GPU sort based on an optimized version of bitonic sort. Bitonic sort is simpler (smaller constant time factors) but requires $O(n \log^2 n)$ work. As observed in [10], their bitonic sort based method outperforms sample sort based methods for small data sets but their own experiments (Figure 8 in [10]) also show that for increasing data size this advantage is lost, with a break even point at approx. 64 Million data items.

4. GPU Bucket Sort: *Deterministic* Sample Sort for GPUs

In this section we outline GPU BUCKET SORT, a *deterministic* sample sort algorithm for GPUs, and discuss our CUDA implementation of GPU BUCKET SORT. An overview of GPU BUCKET SORT is shown in Algorithm 1. 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

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)

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 [9] based on [4], 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 [4]. 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 (about $2K$ for the GTX 2XX/Tesla series and $6K$ for the Fermi). 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 Section 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 [7], adaptive bitonic sort [9] based on [4], and GPU SAMPLE SORT [11]. Our experiments indicate that for up to 16 M data items, simple bitonic sort is still faster than even GPU SAMPLE SORT [11] 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.

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 Figure 1 and can be implemented with coalesced memory accesses in global memory. Each row in Figure 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

Fig. 1. Illustration Of Step 7 In Algorithm 1

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 $B_j = A_{1j} \cup A_{2j} \cup ... \cup A_{mj}$ has at most $\frac{2n}{s}$ data items [15]. 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$s$ (see Section 5 below), 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.

5. Experimental Results and Discussion

For our experimental evaluation, we executed Algorithm 1 on five different GPUs (nVIDIA Tesla, GTX 285/2GB, GTX 285/1GB, GTX 260, and Fermi GTX 480) for various data sets of different sizes, and compared our results with those reported in [13] and [11] which are the current best GPU sorting methods and outperform previous methods such as e.g. [16]. Unfortunately, we were unable to compare our work with [14] because the authors did not supply us with their code and their published performance data is for a different GPU (GTX 280) that we did not have available, uses 32-BIT keys (instead of 64-BIT keys used in [13], [11] and our paper), and reported their performance only on small data sets up to 64 million data items.

Figure 2 shows some important performance characteristics of the five different GPUs. Figure 4 shows a comparison of the runtimes of our GPU Bucket Sort implementation on the Tesla C1060, GTX 260, GTX 285 (with 2 GB memory) and Fermi GTX 480 for varying number of data items. Each data point shows the average of 100 experiments. The observed variance was less than 1 ms for all data

	Tesla C1060	GTX 285 (2 GB)	GTX 285 (1 GB)	GTX 260	Fermi GTX 480
Number of cores	240	240	240	216	480
Core clock rate (MHz)	602	648	648	576	700
Global memory size (GB)	4	2	1	0.896	1.5
Memory clock rate (MHz)	1600	2322	2484	1998	1848
Memory bandwidth (GB/sec)	102	149	159	112	177

Fig. 2. Performance Characteristics For nVIDIA Tesla C1060, GTX 285 with 2 GB memory, GTX 285 with 1 GB memory, GTX 260 and Fermi GTX 480. (Source: [2])

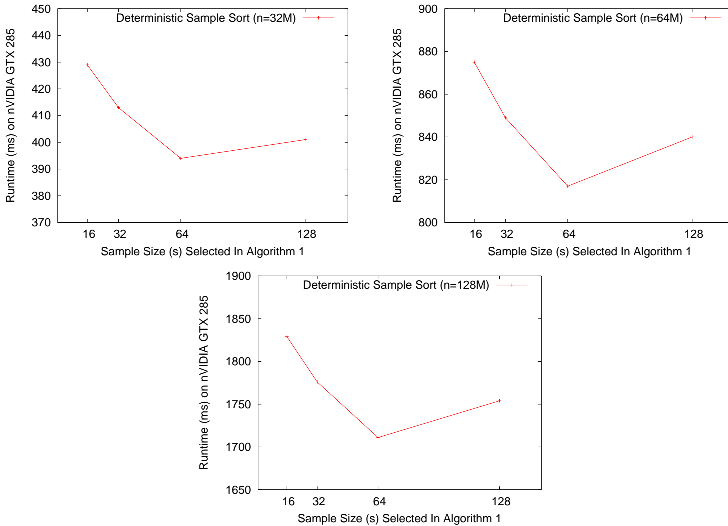


Fig. 3. Runtime Of Algorithm 1 As A Function Of Selected Sample Size s For Fixed $n = 32M$, $n = 64M$, and $n = 128M$.

points since GPU Bucket Sort is deterministic and any fluctuation observed was due to noise on the GPU (e.g. operating system related traffic). All three curves show a growth rate very close to linear which is encouraging for a problem that requires $O(n \log n)$ work. Not surprisingly, GPU Bucket Sort performs best on the Fermi. Interestingly, it performs better on the GTX 285 than both Tesla and GTX 260, and it even performs better on the GTX 260 than on the Tesla C1060. Note that the memory bandwidth for the GTX 260 is higher than for the Tesla C1060. This indicates that GPU Bucket Sort is memory bandwidth bound which is expected for sorting methods since the sorting problem requires only very little

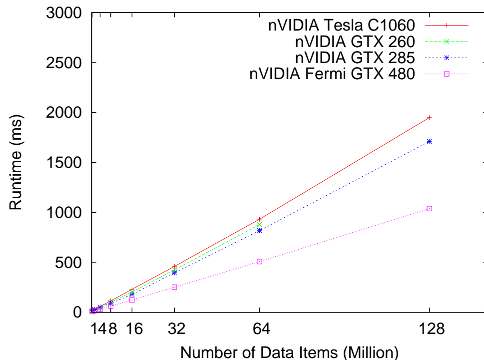
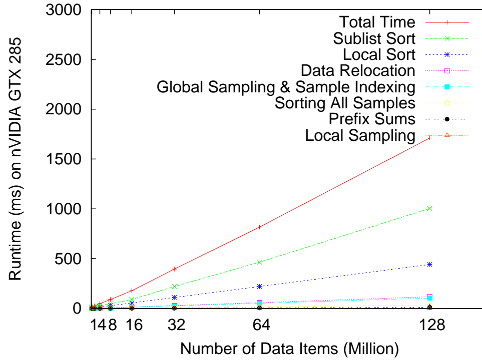


Fig. 4. Performance Of Deterministic Sample Sort For GPUs (GPU Bucket Sort). Total runtime for varying number of data items on different GPUs: nVIDIA Tesla C1060, GTX 260 and GTX 285.

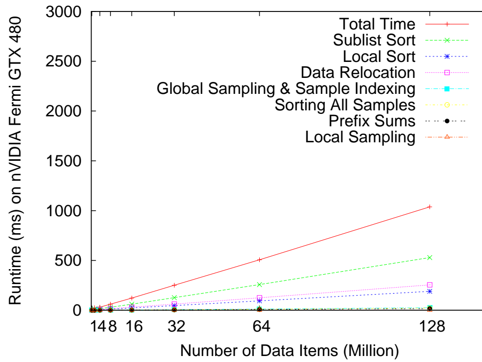
computation but a large amount of data movement. For individual steps of GPU Bucket Sort, the order can sometimes be reversed. For example, we observed that Step 2 of Algorithm 1 (local sort) runs faster on the Tesla C1060 than on the GTX 260 since this step is executed locally on each SM. Because this step is compute-bound, its performance is largely determined by the number of SMs and the performance of the SM's cores. Step 3 is compute-bound because it is taking samples locally and sorting them. Step 4 needs to sort data in global memory and is therefore memory bandwidth bound. Steps 5, 6, and 7 are all compute-bound steps because they do not need significant data movements to/from global memory. Steps 8 and 9 are memory bandwidth bound because these steps include high loads of data movement and sorting in global memory. In summary, the entire algorithm is memory bandwidth bound because the steps that are taking most of the time are memory bandwidth bound while compute bound steps contribute only a small fraction of the total time.

Note that the GTX 285 and Fermi GTX 480 remained the fastest machines, even for all individual steps. We note that GPU Bucket Sort can sort up to $n = 64M$ data items within the 896 MB memory available on the GTX 260 (see Figure 4). On the GTX 285 with 2 GB memory and Tesla C1060 our GPU Bucket Sort implementation can sort up to $n = 256M$ and $n = 512M$ data items, respectively (see Figures 6&7). On the Fermi GTX 480 with 1.5GB memory it is able to sort up to $n = 128M$ data items.

Figure 5 shows in detail the time required for the individual steps of Algorithm 1 when executed on a GTX 285 and Fermi GTX 480. 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



(a)

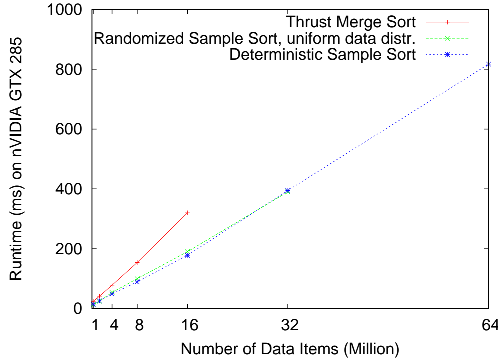


(b)

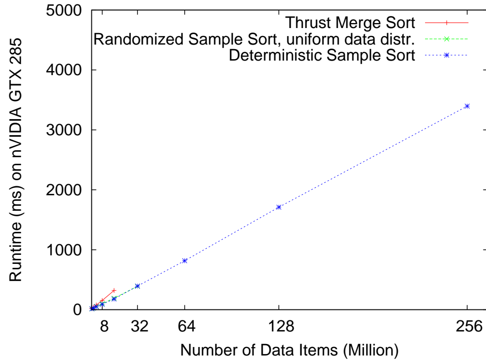
Fig. 5. Performance Of Deterministic Sample Sort For GPUs (GPU Bucket Sort). Total runtime and runtime for individual steps of Algorithm 1 for varying number of data items. (a) For nVIDIA GTX 285 and and (b) for nVIDIA Fermi GTX 480.

when memory accesses can be coalesced (see Section 2). Note that, the sample size s in Algorithm 1 is a free parameter that needs to be tuned. With increasing s, the sizes of sublists B_j created in Step 8 of Algorithm 1 decrease and the time for Step 9 decreases as well. However, the time for Steps 3-7 grows with increasing s. This trade-off is illustrated in Figure 3 which shows the total runtime for Algorithm 1 as a function of s for fixed $n = 32M, 64M, 128M$. As shown in Figure 3, the total runtime is smallest for $s = 64$, which is the parameter value chosen for our GPU Bucket Sort code.

Figures 6 and 7 show a comparison between GPU Bucket Sort and the current best GPU sorting methods, *randomized* GPU Sample Sort [11] and Thrust Merge [13]. Figure 6 shows the runtimes for all three methods on a GTX 285 and Figure 7 shows the runtimes of all three methods on a Tesla C1060. Note that, [13] and [11] did not report runtimes for the GTX 260 and Fermi GTX 480. For GPU Bucket Sort, all runtimes are the averages of 100 experiments, with less



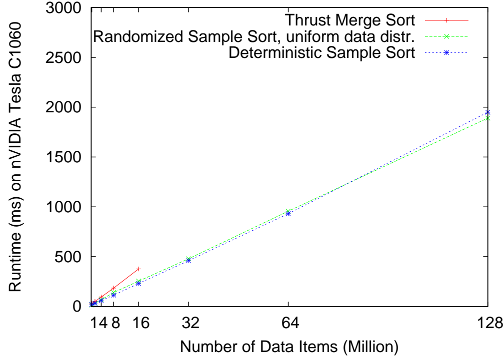
(a) Number of Data Items Up To 64,000,000.



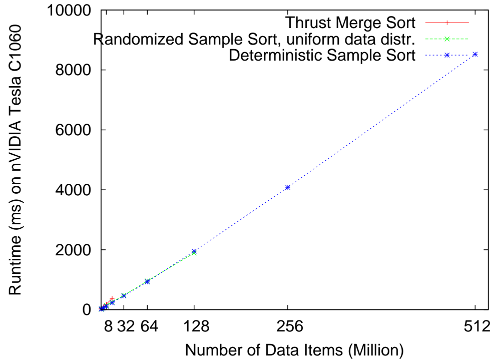
(b) Number of Data Items Up To 256,000,000.

Fig. 6. Comparison between Deterministic Sample Sort (GPU Bucket Sort), Randomized Sample Sort (GPU Sample Sort) [11] and Thrust Merge [13]. Total runtime for varying number of data items on an nVIDIA GTX 285. ([13] and [11] provided data only for up to 16M and 32M data items, respectively.)

than 1 ms observed variance. For *randomized* GPU Sample Sort and Thrust Merge, the runtimes shown are the ones reported in [11] and [13]. For Thrust Merge, performance data is only available for up to $n = 16M$ data items. For larger values of n, the current Thrust Merge code shows memory errors [6]. As reported in [11], the current *randomized* GPU Sample Sort code can sort up to $32M$ data items on a GTX 285 with 1 GB memory and up to $128M$ data items on a Tesla C1060. Our GPU Bucket Sort implementation appears to be more memory efficient. GPU Bucket Sort can sort up to $n = 256M$ data items on a GTX 285 with 2GB memory and up to $n = 512M$ data items on a Tesla C1060. Therefore, Figures 6a and 7a show the performance comparison with higher resolution for up to $n = 64M$ and $n = 128M$, respectively, while Figures 6b and 7b show the performance comparison for the entire range up to $n = 256M$ and $n = 512M$, respectively.



(a) Number of Data Items Up To 128,000,000.



(b) Number of Data Items Up To 512,000,000.

Fig. 7. Comparison between Deterministic Sample Sort (GPU Bucket Sort), Randomized Sample Sort (GPU Sample Sort) [11] and Thrust Merge [13]. Total runtime for varying number of data items on an nVIDIA Tesla C1060.([13] and [11] provided data only for up to 16M and 128M data items, respectively.)

We observe in Figures 6a and 7a that, as reported in [11], *randomized* GPU Sample Sort [11] significantly outperforms Thrust Merge [13]. Most importantly, we observe that *randomized* sample sort (GPU Sample Sort) [11] and *deterministic* sample sort (GPU Bucket Sort) show nearly identical performance on both, the GTX 285 and Tesla C1060. Note that, the experiments in [11] used a GTX 285 with 1 GB memory whereas we used a GTX 285 with 2 GB memory. As shown in Figure 2, the GTX 285 with 1 GB has a slightly better memory clock rate and memory bandwidth than the GTX 285 with 2 GB which implies that the performance of *deterministic* sample sort (GPU Bucket Sort) on a GTX 285 is actually a few percent better than the performance of randomized sample sort (GPU Sample Sort).

The data sets used for the performance comparison in Figures 6 and 7 were uniformly distributed, random data items. The data distribution does not impact

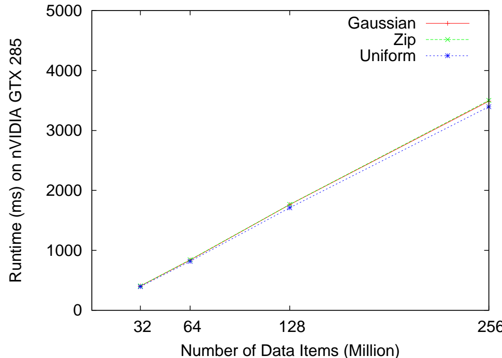


Fig. 8. Performance of Deterministic Sample Sort (GPU Bucket Sort) for Different Input Data Distributions.

the performance of *deterministic* sample sort (GPU Bucket Sort) but has an impact on the performance of *randomized* sample sort (GPU Sample Sort). In fact, the uniform data distribution used for Figures 6 and 7 is a *best case* scenario for *randomized* sample sort where all bucket sizes are nearly identical. Figure 8 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 time performance of our algorithm.

Figures 6b and 7b show the performance of GPU Bucket Sort for up to $n = 256M$ and $n = 512M$, respectively. For both architectures, GTX 285 and Tesla C1060, we observe a very close to linear growth rate in the runtime of GPU Bucket Sort for the entire range of data sizes. This is very encouraging for a problem that requires $O(n \log n)$ work. In comparison with *randomized* GPU Sample Sort, the linear curves in Figures 6b and 7b show that our GPU Bucket Sort implementation maintains a fixed *sorting rate* (number of sorted data items per time unit) for the entire range of data sizes, whereas it is shown in [11] that the sorting rate for *randomized* GPU Sample Sort fluctuates and often starts to decrease for larger values of n.

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 [13], the best comparison-based sorting algorithm for GPUs, and it is exactly as fast as *randomized* sample sort for GPUs (GPU Sample Sort)[11] when the input data sets used are uniformly distributed, which is a *best case* scenario for randomized sample sort. However, as observed in [11], the performance of *randomized* GPU Sample Sort fluctuates with the input data distribution whereas GPU Bucket Sort does not show such

fluctuations. GPU Bucket Sort showed a fixed *sorting rate* (number of sorted data items per time unit) for the entire range of data sizes tested (up to $n = 512M$ data items), whereas it is shown in [11] that the sorting rate for *randomized* GPU Sample Sort fluctuates and often starts to decrease for larger values of n. In addition, our GPU Bucket Sort implementation appears to be more memory efficient because GPU Bucket Sort is able to sort considerably larger data sets within the same memory limits of the GPUs.

References

[1] *NVIDIA CUDA Programming Guide*. nVIDIA Corporation, www.nvidia.com.
[2] *NVIDIA GPU Technical Specifications*. nVIDIA Corporation, www.nvidia.com.
[3] *The OpenCL Specification 1.0*. Khronos OpenCL Working Group, 2009.
[4] G. Bilardi and A. Nicolau. Adaptive bitonic sorting. An optimal parallel algorithm for shared-memory machines. *SIAM J Comput*, 18(2):216–228, 1989.
[5] D. Cederman and P. Tsigas. A practical quicksort algorithm for graphics processors. In *Proc. European Symposium on Algorithms (ESA)*, volume 5193 of *LNCS*, pages 246–258, 2008.
[6] M. Garland. Private communication. nVIDIA Corporation, 2010.
[7] N. Govindaraju, J. Gray, R. Kumar, and D. Manocha. GPUTeraSort: high performance graphics co-processor sorting for large database management. In *Proc. International Conference on Management of Data (SIGMOD)*, pages 325 – 336, 2006.
[8] GPGPU.ORG. General-purpose computation on graphics hardware.
[9] A. Greb and G. Zachmann. GPU-ABiSort: Optimal parallel sorting on stream architectures. In *Proc. Int'l Parallel and Distributed Processing Symposium (IPDPS)*, 2006.
[10] H.Peters, O.Schulz-Hildebrandt, and N.Luttenberger. Fast in-place, comparison-based sorting with cuda: a study with bitonic sort. *J.Concurrency and Computation: Practice and Experience*, 23:681–693, 2011.
[11] N. Leischner, V. Osipov, and P. Sanders. GPU sample sort. In *Proc. Int'l Parallel and Distributed Processing Symposium (IPDPS)*, pages 1–10, 2010.
[12] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym. NVIDIA Tesla: A unified graphics and computing architecture. *IEEE Micro*, 28(2):39–55, 2008.
[13] N. Satish, M. Harris, and M. Garland. Designing efficient sorting algorithms for manycore GPUs. In *Proc. Int'l Parallel and Distributed Processing Symposium (IPDPS)*, 2009.
[14] Nadathur Satish, Changkyu Kim, Jatin Chhugani, Anthony D. Nguyen, Victor W. Lee, Daehyun Kim, and Pradeep Dubey. Fast sort on cpus and gpus: a case for bandwidth oblivious simd sort. In *Proc. International Conference on Management of Data (SIGMOD)*, 2010.
[15] H. Shi and J. Schaeffer. Parallel sorting by regular sampling. *J. Par. and Dist. Comp.*, 14:362–372, 1992.
[16] E. Sintorn and U. Assarsson. Fast parallel GPU-sorting using a hybrid algorithm. *J. of Parallel and Distributed Computing*, 68(10):1381–1388, 2008.