Int J Parallel Prog (2018) 46:1017–1034
https://doi.org/10.1007/s10766-017-0502-5

Survey of GPU Based Sorting Algorithms

Dhirendra Pratap Singh[1] · **Ishan Joshi**[1] · **Jaytrilok Choudhary**[1]

Received: 11 August 2016 / Accepted: 6 April 2017 / Published online: 11 April 2017

Abstract Parallel sorting algorithms are widely studied nowadays. After the introduction of parallel processors such as graphics processing unit (GPU) and easy to use parallel programming languages such as CUDA and OpenCL, literature on parallel sorting algorithms has become vast and richer with new ideas and techniques applied to solve the famous problem of sorting. This paper presents a survey of GPU based sorting algorithms. Four sorting algorithms have been selected for this survey: Radix sort, Merge sort, Sample sort and Quick sort. Methods used in those algorithms are described in brief. The performance of these algorithms as claimed by their authors is also presented. A comparative analysis based on the literature is depicted.

Keywords Radix sort · Merge sort · Quick sort · Sample sort · GPU · CUDA

1 Introduction

Sorting algorithms are commonly designed on GPUs these days. Algorithms using the divide and conquer strategy are mainly mapped on GPUs since they are apposite for parallel environment. However, GPUs are constructed for facilitating arithmetic and calculation-intensive computations; therefore, it is not straight-forward to convert sequential sorting algorithms to parallel. Still, the literature reports that a wide range

✉ Dhirendra Pratap Singh
dpsingh.manit@gmail.com

Ishan Joshi
ishanjoshi731@gmail.com

Jaytrilok Choudhary
jaytrilok@gmail.com

[1] Maulana Azad National Institute of Technology, Bhopal, India

of algorithms are implemented on GPUs. Sorting on a Central Processing Unit (CPU) is slower than sorting on a GPU.

In the current generation of processors like GPUs and multicore CPUs, several computing cores are packed on a single chip. Common design practice in the modern CPU is to devote most of the chip area to the cache mechanism and memory management. In contrast, in many core processors most of the chip area is devoted to computation. For this reason, parallel processors excel over CPUs in computationally intensive operations. However, it is difficult for the programmer to write applications for many core processors that fully utilize the parallelism provided by the hardware.

Compute unified device architecture (CUDA) greatly helps programmers to write applications for GPUs. It provides a set of C programming language extensions. These extensions are familiar, easy to use and express parallelism in a logical manner. Due to these extensions, programmers are straightway able to use multiple levels of memory hierarchy that are quite different from the CPU memory hierarchy model. Specific memory layouts used by the program play a major role in affecting the performance of the program in the case of CUDA, since it leaves the full responsibility of taking full advantage of all levels of the CUDA's explicit memory hierarchy to CUDA programmers. That is the reason CUDA programs are difficult to optimize. Performance modeling of CUDA programs is less well understood than that of OpenCL programs. In addition, GPU hardware is evolving fast. Previous generations of the GPU are very different in terms of architectural features than the current generation. A program performing well on one generation cannot be expected to perform in the same manner on the next generation. Optimizing a program for a specific GPU is of no avail since this is very tedious and time-consuming. We need an analysis of GPU sorting algorithms. How do specific algorithms perform in a specific environment? How far are factors such as reducing memory latency and scalability being addressed in the algorithm?

In this paper, four GPU sorting algorithms available in the literature—Radix sort, Quick sort, Merge sort and Sample sort are reviewed, and the methods used in those algorithms are described in brief. Finally the performance of these algorithms as claimed by their authors is presented.

2 Radix Sort

Radix sort has been one of the fastest sorting algorithms for 32 and 64 bit keys on both CPU and GPU for a long time due to its linear time performance [1]. The Radix sort algorithm has low and variable computational granularities (shift and mask operations in this case). Radix sort has fewer I/O operations. Redundant computation can be covered due to fewer I/O operations [2]. It has stronger dependence between successive elements compared to other approaches. This issue is addressed by Ha et al. [3].

Satish et al. [4] describe two GPU sorting algorithms, namely Radix sort and Merge sort. In Radix sort, fine grained parallelism is achieved by parallel scan operation. On-chip memory is used to impose block structuring on the sorting algorithm and also, for local ordering of data to improve coherence. Radix sort is a counting based sorting algorithm. It sorts one digit at a time starting from least significant digit to most significant digit of a d digit number. Either of the algorithms, Counting sort or Bucket sort can be used inside each pass of Radix sort. For each digit, the sum of the number of keys having smaller digit and the number of keys having the same digit but occurring earlier in the sequence is counted, it is referred to as the rank of the element. The element is rewritten at this position in the output sequence. This counting preserves the relative ordering of data and hence it is stable i.e. the output sequence is sorted after d passes.

A sequence is divided into tiles and assigned to different thread blocks; let us say p thread blocks. Two factors are taken care of: the number of scatters is minimized to global memory and the coherence of scatters is maximized. The first goal is accomplished by data blocking and using a digit size greater than one. To achieve the second goal, data blocks are locally sorted by the current radix 2^b digit using On-chip shared memory.

In [5], an empirical optimization technique is proposed for the generation of efficient code. It focuses on algorithmic parameters that can be altered according to different environments and GPU architectural factors. Demonstration of this empirical optimization is performed using Radix sort. d passes of Radix sort can be written as:

$$d = \left\lceil \frac{b}{r} \right\rceil \tag{1}$$

where b is the number of bits in keys and r is the number of bits processed in each pass. The rank is calculated using three steps: the first step traverses the input sequence and counts the number of elements in each of the 2^r buckets. Buckets are collected in such a way that the new value of the current bucket is calculated by adding all the lower numbered buckets and the original value of the current bucket in the second step. The xth bucket contains the starting position of all the elements whose ith element is x. The rank of an element is the value of its corresponding bucket plus the number of elements in front of it. The third step is a scattering operation in which elements are finally assigned positions according to their ranks.

The first step of the algorithm is parallelized by dividing the input sequence into blocks; each CUDA thread block will work on a different data block. The second step is parallelized using a publicaly available prefix sum primitive in the CUDA Data Parallel Primitives (CUDPP) library.

Two algorithmic parameters are identified in Radix sort. The first one determines the division of workload among GPU resources. Division of the workload is specified using a three tuple (*number_of_thread_block, number_of_thread_per_block, number_of_element_per_block*). Two such tuples are needed, one for the counting step and the other for the scattering step. The second parameter is the width of the digit i.e. the number of digits processed in each pass. In earlier implementations of Radix sort, r was fixed, e.g. CUDPP uses $r=1$ in [6] and $r=4$ in [4], but here a different r is used for every pass. For prefix sum step, a three number tuple and a vector of r is needed.

Architectural factors affecting the performance of Radix sort are the number of cores, the global memory bandwidth and the frequency of GPU core and global memory.

Number of Cores When the number of available cores increases, each thread will process fewer elements. Consequently, more threads can be spawned on an increased number of cores. On the other hand, increasing the number of cores will increase the amortization of overhead of execution of threads. It is a matter of a trade-off between the two.

Global Memory Bandwidth Global memory utilization depends on the value of r. When higher r is selected, fewer passes are required and therefore, the overhead incurred by each pass is reduced. In contrast, higher r will mean that the input sequence is written to more diversified memory locations, resulting in un-coalesced memory access. It is a matter of a trade-off between the two.

Frequency of GPU Core and Global Memory When the frequency of the core is increased, the number of elements processed per thread should also be increased to fully utilize the GPU resources. However, increasing the number of elements processed per thread will increase un-coalesced memory access. It is a matter of a trade-off between the two.

Ha et al. [3] present a hardware optimized parallel Radix sort algorithm and its implementation. The presented solution does not require any special hardware extension such as atomic counting, making it general and applicable to a wide range of hardware. Contrary to previous approaches like the Radix sort of the CUDPP library, this paper shows how to implement the 4-way Radix sort. The performance of the algorithm is doubled and also the number of loops is reduced by half. An efficient order checking strategy is also implemented which would terminate the sorter early if a sorted state is reached or the input is already sorted. This algorithm is applicable for arbitrary types of data such as floats, pointers, general records and negative numbers that make it a generalized solution.

Four subsystems of the 4-way GPU Radix sort algorithm are: the order checking function, 4-way radix counting, prefix sum positioning and final mapping. The order of the input elements is checked at the beginning of the loop. If the elements of the input sequence are in order, the loop is immediately terminated. A test is implemented based on the optimized reduction operation proposed by Harris [7]. The next step of the algorithm is to compute the frequency of every element in the list. This is the first step of Radix sort. The input array is divided into blocks to make sure that the first step is being run in a parallel manner. In each of these blocks, the local frequency of all possible elements is computed. Four counters are generated so as to sort two bits per pass. For each of the four possible digits, a bit mask is generated. Shared memory parallel optimized prefix sum computation is applied to every mask at the same time on four counting bit combinations and the total count of every element in a block in the global memory is stored. The order of the data in the sorted chunk of the same radix counting bit is indicated by the local prefix sum. Data is shuffled locally in order to prevent a non-coalesced scattering effect of element-wise mapping at the final state. In the next step, by computing the prefix sum over the block sum array, local frequency lists are converted into global positions. The sorted position is calculated as

$$SortedPosition = P_d[n] + m \tag{2}$$

where d is a digit, n is the input chunk and m is the local prefix sum value.

In [8], a Radix sort algorithm is presented which is implemented in the Thrust library of CUDA. Operations of the algorithm involve iterating over digit places of numbers to be sorted from the least significant to the most significant digits. A stable *distribution sort* is performed for each digit place based upon their digit at that digit position. For an m-element sequence with l-bit keys and radix $s = 2^d$, a Radix sort of these keys will require $\frac{l}{d}$ passes of a *distribution sort* over all m keys. The asymptotic complexity of the *distribution sort* is *O(n)*. The reason behind this is that each input item needs only a fixed number of comparisons to get its position. The entire sorting process is *O(n)* if the number of digit-places is definite.

This approach is a hybrid composition of several algorithms. Hardware is efficiently used by employing memory bound parallel prefix scan routines. The prefix scan enables computing elements to dynamically and co-operatively find proper memory locations into which their output data can be placed. Keys are processed in a data parallel manner, for a given digit place, in Radix sort. Co-operation between processing elements is necessary to find out appropriate destinations for relocating their keys.

One more reason behind efficient use of hardware is applying a visitor pattern [9] of task composition in the prefix scan primitive. In this way, kernel fusion is achieved. With this pattern, the arithmetic intensity of memory bound primitives is increased. There is no need to move between the intermediate states of the program through global device memory. So, the number of memory transactions needed by the application is reduced. Elimination of load and store instructions also allows the pattern to exploit the resources. Thus, the memory workload is reduced. The prefix scan requires only a small, constant number of intermediate values that must be exchanged through global memory.

2.1 Experimental Evaluation

Satish et al. [4] evaluated their algorithms on a range of GPUs: GTX 200, 9800 GTX, 8800 Ultra, 8800 GT and 8600 GT. Their implementation of Radix sort outperformed Radix sort by Le Grand [10] implemented in GPU Gems, Radix sort by Sengupta et al. [11] implemented in CUDPP library [6], GPUSort and Merge sort implemented by Govindaraju et al. [12]. Radix sort achieved a speed increase of a factor of 4 over GPUSort. Radix sort by Sengupta et al. [11] is two times slower than the Radix sort presented in this paper. When compared with GPU-quicksort [13] on GTX 280 for uniform random sequences of 1M to 16M elements, it is three times faster than GPU-quicksort [13].

In [5], empirically optimized Radix sort implementation outperformed the Radix sort implemented by Satish et al. [4] by 17.1, 15.9, 15.7 and 23.1% on average on four different NVIDIA GPUs GTX 280, 8800 GTX, 9600 M GT and 9400 M respectively.

Ha et al. [3] evaluated the performance of their algorithm on an Opteron AMD 275 quad dual-core 2.2 Ghz, system with 6 GB of memory and 1024 K L1 cache equipped with an NVIDIA Geforce 8800 GTX GPU. Algorithms with which the 4-way Radix sort was compared were: the optimized CPU Radix sort as proposed by Herf [14], the STL sort [15], the multi-threaded TBB parallel Quick sort by Intel [16], the Radix-16 sort by Le Grand [10] and an 8-way parallel version of this Radix sort. Both with uniform and Gaussian distribution inputs, the coalesced 4-way Radix sort gave the best performance; that was between 1.5 and 2.2 times faster than the Radix-16 that also ran on CUDA by Le Grand [10]. A Speed increase of 1.5 was achieved from the 4-way non-coalesced mapping radix to the final version of 4-way radix. This demonstrates the impact of coalesced access on the performance of CUDA GPU implementation. CPU Radix sort was 1.5-2.2 times slower than this algorithm. STL sort [15] was reportedly slower than this algorithm due to expensive thread creation. In comparison to the performance of value sorting, a reduction by a factor between 1.5 and 1.8 is experienced. A non-coalesced shuffling pattern is the reason for the slow performance.

The algorithm of [8] is at least two times faster than the sorting algorithm described by Satish et al. [4] on all fully programmable NVIDIA GPUs and 3.7 times faster on the GT200 based models. Running this algorithm on older GPUs like the GT80, it was observed that it outperformed Intel Core 2 quad-core processors.

3 Merge Sort

Belonging to the Insertion sort family, the Merge sort is an apposite algorithm for parallel processors. Merge sort is of divide and conquer nature, which is why it is suitable for implementation on GPUs. It has been observed that the Merge sort is used with some other algorithms to form a hybrid approach for sorting. For external sorting, when the input sequence is stored in a large external memory, Merge sort is the preferred technique [4].

In [17], an algorithm is presented for sorting large lists in *n logn* time complexity. Initially, a list is partitioned into sub-lists to fully utilize the parallelism of the GPU. The list is partitioned using a GPU based Bucket sort or Quick sort. Partitions are sorted independently using a custom Merge sort. Merge sort works on flout4 vectors instead of one element.

The Bucket sort divides the list such that elements of list *k* are smaller than elements of list *k+1*. The Bucket sort has two passes. The first pass makes of list of pivot points which can be obtained either by interpolation or by constructing a histogram. It counts the number of elements moving in each bucket. It also remembers the bucket number and position inside the bucket for each element. Elements are moved to their new position in the second pass.

The number of partitions of the list is at least double the number of streaming processors in the GPU. Parallelism in the algorithm is increased by increasing the number of partitions. Since, the number of threads executing in parallel will never be less than the number of partitions at any point in time. However, increasing the number of partitions too much will increase the work of the Bucket sort or Quick sort.

Sub-lists can be independently sorted using Merge sort. Flout4 elements are internally sorted using a kernel. One thread is assigned a pair of lists to be merged. The output is the sorted merge of these two lists. Flout4 vectors are taken from lists depending on which one has the lowest element on the top. This flout4 element is compared with the flout4 element already taken in the previous step such that the lowest four elements are separated from the highest four elements. Internal sorting of these two flout4 vectors take place. The list with the lowest four elements joins the sorted stream of elements.

The Merge sort [4] performs sorting by merging sorted sequences in parallel and pair-wise. It has three steps:

1. Input is divided into p equal sized tiles.
2. p tiles are sorted in parallel through thread blocks.
3. Sorted tiles are merged.

For sorting individual data tiles, Batchers odd-even Merge sort is used. The third step of the algorithm is performed using a pair-wise merge tree of *log p* depth.

When the number of sub-sequences to be merged decreases geometrically, parallel merging becomes less efficient due to reduced utilization of resources. This algorithm tries to show the fine grained parallelism in the merging step. Parallel merging is the most time-consuming process. It uses two strategies to design an efficient parallel merge routine. The first is to use a divide and conquer approach at each level of the merge tree. The second is to employ parallel binary searches to find out the final position of each element in the sorted sequence.

If the sub-sequences are less than $t=256$, they are merged using single thread block t. The rank of every element present in both the sub-sequences in the merged sequence is computed. Rank of an element let $a_i \in A$ is calculated as

$$rank(a_i; C) = i + rank(a_i; B) \tag{3}$$

where C is the merged sequence and A and B are the two sub-sequences. $rank(a_i; B)$ is count of the elements $b_j \in B$ with $b_j < a_i$ and it is computed efficiently using binary search. Previous steps are followed for the contents of B also.

Larger arrays can be merged by dividing them into tiles of size t that can be merged independently through thread blocks. Splitters are used to divide the array. Splitters are sequences constructed after selecting every tth element in every sub-sequence. Splitters of two sub-sequences are than merged to form a single splitter.

The algorithm [18] is based on Merge sort. It introduces a new sorting algorithm, namely GPU-Warp sort. Firstly, the input sequence is divided into sub-sequences which are sorted using bitonic networks. This is followed by Merge sort, which merges all the sub-sequences to get the final sequence. In a warp, threads are executed synchronously. The algorithm takes advantage of this synchronous execution in order to map sorting tasks efficiently to GPU architecture. An adequate homogeneous parallel comparison is given to all the threads in a warp to circumvent branch divergence. Merge sort sustains coalesced global memory accesses. This algorithm performs the following four operations.

Barrier Free Bitonic Sort If blocks are taken as a unit to sort each sub-sequence, each adjacent stage needs to be globally synchronized. This is the reason that warp is chosen here as a unit instead of block. The algorithm becomes barrier free due to the synchronous execution of threads in a warp. Some other features to be noticed are: firstly, n elements correspond to $\frac{n}{2}$ compare and swap operations in each stage. Secondly, half of the compare and swap operations form ascending pairs. Descending pairs are formed by the rest of the operations except the last phase, where all the comparisons have the same sequence order. At least 128 elements are kept in every sub-sequence. All 32 threads within a warp perform distinct compare and swap operations at the same time to avoid divergent operations appearing within a warp and all these operations form sorted pairs with the same order. Sub-sequences are loaded in the shared memory.

Bitonic Based Merge Sort If the sub-sequences to be merged are A and B, the warp fetches the lowest $\frac{t}{2}$ elements from both A and B where, t is the size of warp buffer in shared memory. After a Bitonic merge sort, the lowest $\frac{t}{2}$ elements of the result are the output and the remaining $\frac{t}{2}$ elements remain in the buffer. The next $\frac{t}{2}$ entries of the buffer are filled with either $\frac{t}{2}$ elements from A or B, depending on the outcome of comparing the maximum elements fetched last time.

Split into Small Tiles The number of sequences to be merged geometrically decrease in the Merge sort. The efficiency and parallelism of the algorithm are reduced. To cope with this situation, sequences are divided into smaller tiles. Suppose l is the number of sequences of size n before split operation and each sequence is divided into s sub-sequences. The following equation should be satisfied

$$\forall a \in \text{sub-sequence } (x, i), \forall b \in \text{sub-sequence } (y, j) : a \leq b, \text{ where}$$
$$0 \leq x < l, 0 \leq y < l, 0 \leq i < j < s.$$

Final Merge Sort Merging of sub-sequences created in the step 3 is done in step 4. All the sub-sequences can be independently merged. Therefore, it is generating sufficient parallelism. Complexity of GPU-warpsort is defined as:

$$O(n) + O(nlogn) = O(nlogn) \tag{4}$$

3.1 Experimental Evaluation

Sintorn et al. [17] used Intel Core 2 Duo system working at 2.66 GHz for the experimental evaluation. They used a single GeForce 8800 GTS-512 graphics card and also dual cards. As claimed by the authors, the algorithm is 6.2 times faster than STL Introsort [19] which ran on a single core of a CPU, on a single GPU and 11 times faster if compared with the dual GPU version. The authors claimed the dual GPU version was 1.8 times faster than the single GPU version. It was 10% faster than the GPU Radix sort [11] and 2.5 times faster than the GPUSort algorithm [12].

In [4], the algorithm was tested on a range of GPUs—GTX 200, 9800 GTX, 8800 Ultra, 8800 GT and 8600 GT. In the case of Merge sort, a speed increase of two was achieved over GPUSort [12]. Performance of this algorithm and the one suggested by Le Grand [10] was almost the same for up to 1M elements, after which this algorithm excelled. It was slower than the Radix sort algorithm implemented in the paper [4]. For large input sizes, it outperformed CUDPP Radix sort [11].

The processor used for testing the algorithm of [18] was AMD Opteron880 working at 2.4 GHz and the graphics card used was NVIDIA 9800GTX+. The algorithm was compared with the sorting algorithm designed by Satish et al. [4], the Quick sort designed by Cederman et al. [13] and the Bitonic sort of Baraglia et al. [20]. GPU-Warpsort [18] performed 20% efficient when compared to the Merge sort of Satish et al. [4] and 30% efficient for sequences larger than 4M. GPU-Warpsort [18] was 1.7 times faster than GPU-quicksort [13] for key only sequences. It performed substantially better than the Bitonic sort of Baragalia et al. [20].

4 Sample Sort

Sample sort is or has been the fastest sorting algorithm if the inter-process communication is high [4]. It selects a subset of the input. This subset is referred to as splitters. Splitters are sorted by some other procedure. The input sequence is divided into buckets using these splitters. Each bucket is sorted in parallel and the result is the concatenation of these buckets. However, performance of the Sample sort degrades if the number of elements per processor is low [21,22].

Leischner et al. [23] describe GPU based Sample sort algorithm. The algorithm needs a comparison procedure on keys only. On the other hand, other approaches directly manipulate the binary representation of data. This algorithm decreases number of access to global memory. The reason behind the reduction is that it processes the data in various multi-way phases instead of a larger number of two-way phases. Coarse level parallelism is achieved by imposing block structuring. This enables data parallel computation of individual input tiles. Computation is performed by blocks of fine grained concurrent threads. Memory intensive data structures are stored in fast per-block memory, which is why global memory accesses are reduced. Huge volumes of data can also be processed by assigning a variable number of elements per thread.

The first step of the algorithm [23] is for the splitters to split the input sequence into k buckets bound by successive splitters. All the buckets are sorted recursively and the output is a concatenation of these buckets. The input sequence is divided into p number of tiles and p is evaluated as:

$$p = \left\lceil \frac{n}{t \times l} \right\rceil \tag{5}$$

where t is the number of threads in a block and l is the number of elements processed by each thread. In step 2, for all elements in its tile, the thread block computes the bucket indices. The number of elements in each bucket is computed by the thread block. A per-block k-entry histogram is also stored in the global memory by the thread block. In step 3, global bucket offsets in the output are calculated by perfoming a prefix sum over the $k \times p$ histogram tables stored in a column-major order. In step 4, bucket indices are again calculated by each thread block for all elements in its tile. Thread block also calculates local offsets in the buckets. Elements are stored at their appropriate output locations using the global offsets calculated in the previous step.

Dehne et al. [24] present GPU Bucket sort. The performance of GPU Bucket sort is the same as that of the randomized sample sort method in [23], as far as the best case is concerned. However, the performance of GPU Bucket sort is similar for any kind of data distribution. There are two reasons for this. Firstly, the buckets are created deterministically. Secondly, bucket sizes are guaranteed. The authors also claim the algorithm to be memory efficient since it is able to sort considerably larger data sets within the same memory limits of the GPUs.

In the GPU Backet sort [24], first the array is split into m sub-lists $A_1, \ldots, A_m$ containing $\frac{n}{m}$ elements. Where, $\frac{n}{m}$ is the size of the shared memory at each streaming multiprocessor (SM). All the sub-lists are sorted locally on the SM, using its shared memory as cache. Equidistant samples are selected from each sorted sub-list. If the number of samples in each sub-list is s, the total number of samples is sm. All samples are sorted in global memory. Equidistant samples are selected from the sorted list of sm samples. These samples are called global samples and are s in number. The location of each of the s global samples in A_i is determined. This operation will create for each A_i a partitioning into s buckets. For each bucket, its starting location in the final sorted sequence is calculated through a parallel prefix sum operation. Buckets are relocated to the location found in the previous step and the newly created array consists of s sub-lists. These sub-lists are sorted locally on every SM.

4.1 Experimental Evaluation

Experimental evaluation of Sample sort algorithm [23] was performed on a sequence of floats: 32-bit and 64-bit integers and key-value pairs where both keys and values were 32-bit integers. Performance of the algorithm was compared with many algorithms such as Thrust [8] and CUDPP [6] Radix sort and Thrust Merge sort [4]. GPU-quicksort [13], hybrid sort [17] and bbsort [25] were also included. The processor used in the experimental evaluation was an Intel Q6600 2.4 GHz machine. The NVIDIA Tesla C1060 was the GPU used. The performance gain of Sample sort over the best comparison-based sorting algorithm GPU Thrust Merge sort [4] was at least 25% and on average 68% for uniformly distributed keys. For the same data distribution, Sample sort was two times faster than GPU-quicksort [13] on average. The performance gain over the highly optimized GPU Thrust Radix sort [8] was at least 63% and on average was double for 64 bit integer keys.

Dehne et al. [24] used NVIDIA Tesla, GTX 285 and GTX 260 for the experimental evaluation. The performance of the algorithm was compared with the sorting methods presented by Satish et al. [4] and Leischner et al. [23]. Compared with Tesla and GTX 260, GPU Bucket sort performed better on the GTX 285. Performance was better on the GTX 260 than the Tesla. It outperformed the Thrust Merge procedure proposed by Satish et al. [4]. This algorithm and the Randomized Sample sort performed nearly identically on the GTX 285 and Tesla C1060. For the entire range of data sizes, a linear growth rate was claimed by authors in the case of GPU Bucket sort for GTX 285 and Tesla C1060 and it maintained a fixed sorting rate (number of sorted data item per time unit).

5 Quick Sort

In the Quick sort algorithm, a list of elements is taken and partitioned around a specific pivot element. The exact position of the pivot element in the sorted list is found. The lists are recursively partitioned until they become too small to partition. Quick sort is the fastest and most studied algorithm in CPU architecture. However, it was reported as not particularly suitable for GPUs until Cederman et al. [13] presented GPU-quicksort.

Sengupta et al. [11] essentially illustrates implementation of segmented scan primitives. These general purpose data parallel primitives are useful for a broad range of applications. As part of the demonstration of applications of segmented scan primitives, authors have implemented a parallel version of the Quick sort algorithm. The authors used the method proposed by Blelloch [26] to implement Quick sort in CUDA.

Segments of the input are processed in parallel. The algorithm is suitable for segmented scan primitive due to communications between elements (threads) inside a single segment. A pivot element is chosen in each segment (the first element of the segment). The pivot element is distributed across the segments. The input element is compared to the pivot. Greater-than or greater-than-or-equal are compared accordingly in alternating passes of the algorithm. A segmented vector containing true and false is produced by the comparison operation. This segmented vector is used to split-and-segment the input. As a result, smaller elements are placed at the head of the vector and larger elements are placed at the end of the vector. Instead of the 3-way split used by Blelloch [26], here a 2-way split is used. Whether or not the output is sorted is checked by global reduction at each step. In the last step, a parallel merge operation is applied to all the blocks.

Cederman et al. [13] introduce a Quick sort algorithm suitable for a highly parallel multicore GPU. It takes advantage of the high bandwidth of GPUs by reducing the amount of book-keeping and inter-thread synchronization needed. Inter-thread synchronization can be minimized by coalescing read operations and constraining threads so that memory accesses are kept to a minimum. This algorithm partitions the list iteratively. The next iteration begins when lists of previous iterations are partitioned.

A sequence to be partitioned is divided logically into sections. Each section is processed by a thread block. Each thread in the thread block keeps track of the number of elements it has seen larger and smaller than the pivot. Each thread stores this information in two arrays in shared local memory. A cumulative sum is calculated to find out the index of each element. Threads will write their assigned elements in new position in the auxiliary buffer. When the number of sub-sequences is large enough that each thread block can be assigned one, the algorithm enters in the second phase. There is no need for inter-thread block synchronization. When the sub-sequences become small enough to be sorted entirely in the fast local memory, the authors suggest using a different sorting algorithm which performs well when the size of the list approaches the number of threads.

In [27], a recursive version of Quick sort is proposed. It solves two problems associated with GPU-quicksort. First, partitioning of data is not guaranteed to be uniform. Some sub-lists may be longer than others. Since GPU-quicksort is an iterative algorithm and the iteration is completed when all the sub-lists of that iteration are processed, smaller lists will have to wait for the longer ones to complete. Second, in GPU-quicksort, kernel launch for partitioning the list is done on the CPU side. After every iteration, the information of the index and size of the sub-list is communicated back to the CPU and it will decide the number of threads and thread blocks accordingly.

With the increase in the depth of the recursion tree, the number of kernel launches is also increased. Communication between the CPU and GPU becomes an important factor. NVIDIA CUDA Quick sort [27] solves this problem by dynamic parallelism. Dynamic parallelism enables recursive parallelism in GPU. At the top level, a single kernel partitions the sequence into two groups, and then launches two Quick sort kernels. One kernel is assigned to the group having elements smaller than the pivot and the other is assigned to the group having elements larger than the pivot. The kernel at the top knows the index and the size of the two groups; it will have the information on whether to launch a kernel and how many threads to use. The parent kernel is able to launch its child kernels immediately after partitioning the list. The program progresses in an asynchronous manner. The kernel launches its two children in a separate stream. CUDA streams are executed simultaneously, which means the two sub-sorts will run in parallel.

Manca et al. [28] suggest an iterative version of Quick sort namely CUDA-quicksort. It is an improvement over GPU-quicksort [13]. Low inter-block synchronization and coalesced memory access make it faster than GPU-quicksort and NVIDIA CUDA Quick sort [27] by factors of 3 and 2 respectively. In GPU-quicksort, list is divided into *k* equally sized sections. The number of thread blocks is fixed a priori. Section size is decided by dividing the list size by the number of thread blocks available. However, in this algorithm the section size is fixed. The number of thread blocks is decided by dividing the list size by the section size. In the beginning, as in the original Quick sort, a list is partitioned into two sub-lists around a pivot element. Elements smaller than the pivot go to the left partition and elements larger than the pivot go to the right partition. A pivot is placed in between these two partitions. Thread blocks work on their assigned section independently of each other. Each section is divided into two sub-lists. These sub-lists are created in shared memory. Then the sub-lists are written to global memory.

In this algorithm, inter-thread block synchronization takes place through atomic primitives. It differs from the GPU-quicksort [13] in the way sub-lists are written to global memory. In the GPU-quicksort, each thread moves its assigned data to the global memory using the cumulative sum. Threads access the global memory in an un-coalesced manner. CUDA-quicksort optimizes memory access by sorting the elements in the shared memory before being written to the global memory. Threads compute the offset of their assigned elements using the cumulative sum. Offset is used to move the items to their proper sub-list in the shared memory. In the shared memory, elements which are smaller than the pivot are separated from the elements which are larger than the pivot. Threads write the elements in the consecutive memory locations in the global memory.

5.1 Experimental Evaluation

In [11], the processor used for testing the algorithm was an Intel Xeon working at 3.00 GHz. The graphics card used was an NVIDIA GeForce 8800 GTX. The algorithm was compared with four algorithms: Split based Radix sort per block [29], STL sort [15], CPU C implementation of Quicksort and Split based Global Radix sort. GPU based Quick sort was the slowest among all of them. Book-keeping instructions made the program slow. Book-keeping instructions managed the active regions of shared memory. Book-keeping instructions and staging of regions together result in a large number of registers. The occupancy of the program was inevitably reduced (according to the authors, profiler show occupancy of only $\frac{1}{6}$). These were the two reasons cited by the authors for the slow performance of Quick sort.

Experimental evaluation of [13] was performed on a dual-processor, dual-core AMD Opteron 1.8 GHz machine. NVIDIA 8600GTS 256 MiB with four multiprocessors and the high-end NVIDIA 8800GTX 768 MiB with 16 multiprocessors were the two graphics processors used to run the algorithm. Algorithms compared with GPU-quicksort [13] were GPUsort [30], Radix Merge [29], Global Radix [11], Hybrid sort [17] and STL Introsort [19]. Performance on the high-end GPU was three times better than the low-end GPU. GPU-quicksort [13] performed better on all the distributions than the compared sorting algorithms on the high-end processors. On the low-end processors it was also on a par with the all compared algorithms.

In [27], reportedly, the algorithm was 1.5 times faster than GPU-quicksort [13] when tested on 12 core Intel Xeon E5-2667 2.90 GHz and a GPU NVIDIA Tesla Kepler k20.

CUDA-quicksort [28] was tested on a 12 core Intel Xeon E5-2667 2.90 GHz processor and a GPU NVIDIA Tesla Kepler k20 machine. The algorithm was compared with GPU-quicksort [13], CDP Quick sort [27], the Radix sort of the thrust library [8], based on [4], Bitonic sort [31] and Merge sort [4]. CUDA-quicksort was able to outperform all of them except the Radix sort due to the complexity factor. However, when compared with 96-bit data structure items CUDA-quicksort outperformed the Thrust Radix sort [8] and achieved a speed increase ranging from 1.58× to 2.18×.

6 Result Analysis

It can be observed in the Table 1 that Radix sort is a highly studied algorithm on the GPU. Quick sort and Merge sort are also quite often executed on GPUs due to their divide and conquer nature. In Fig. 1, a comparative analysis of the performance of the algorithms presented in this survey is shown. Algorithms connected with an arrow are compared in the literature. The arrow points towards the slower algorithm.

Satish et al. [4] presented Radix sort and Merge sort. Radix sort outperformed the Merge sort algorithm presented in the same paper and it was the fastest sorting algorithm of that time. That is why it is observed in Fig. 1 that much research has focused on surpassing the performance achieved by Radix sort of Satish et al. [4] and much has succeeded in doing so. Radix sort implemented in Thrust library of CUDA is a perfect example of this. Radix sort by Huang et al. [5] provided an optimization technique derived empirically and, as claimed by the authors, it also outmatched Radix sort of Satish et al. [4]. Merge sort is often accompanied with some other algorithm to form a hybrid solution for sorting, as in [17]. The most important part of the Merge sort, the merge routine is often used due to its divide and conquer nature.

Deterministic Sample sort is one of the fastest sorting algorithm presented in [24]. Though the authors have not compared their algorithm with all the relevant and competing algorithms. Their algorithm was shown to be faster than GPU Sample sort [23], which in turn was faster than many algorithms such as GPU-quicksort [13], Hybrid sort by Sintorn et al.[17] etc.

Table 1 Summary of GPU based sorting algorithms

S. no.	Name of the algorithm	Base algorithm	Environment used	Improvement
1	Fast Parallel GPU Sorting using a Hybrid Algorithm	Merge sort and Bucket sort	Processor: Intel Core2 Duo, NVIDIA Graphics Card: GeForce 8800 GTS-512	6.2 times faster than STL Introsort [19], 10% faster than GPU Radix sort [11] and 2.5 times faster than Bitonic sort algorithm [12]
2	Designing Efficient Sorting Algorithms for Many Core GPU	Radix sort and Merge sort	NVIDIA Graphics Card: GTX 200, 9800 GTX, 8800 Ultra, 8800 GT, and 8600 GT	Radix sort is upto 4 times faster than Graphics based GPUSort [30]. Also 2 times faster than CUDA based Radix sort [11]. Merge sort is also twice as fast as GPUSort [12]
3	GPU Quick sort: A Practical Quicksort Algorithm for Graphics Processor	Quick sort	Processor: AMD Opteron 1.8 GHz machine. NVIDIA Graphics card: 8600GTS 256 MiB and the high-end 8800GTX 768 MiB	Performance on high-end GPU is 3 times better than low-end GPU. Outperformed, on all the distributions, GPUsort [12], Radix Merge [29], Global radix [11], Hybrid sort [17] and STL Introsort [19]
4	An Empirically Optimized Radix sort for GPU	Radix sort	NVIDIA Graphics Card: GTX 280, 8800 GTX, 9600M GT and 9400M	Faster than best Radix sort [4] implementation by 17.1, 15.9, 15.7 and 23.1% on four different GPUs
5	Fast 4-way parallel Radix sorting on GPUs	Radix sort	Processor; Opteron AMD 275 2.2Ghz machine, NVIDIA Graphics card: Geforce 8800 GTX	1.5 and 2.2 times faster than the radix-16 [10]. 1.5-2.2 times faster than CPU Radix sort [14]
6	Revisiting Sorting for GPGPU Stream Architectures	Radix sort	NVIDIA Graphics card: GT200 AND G80	2 times faster than sorting algorithm described by Satish et al. [4] on all fully programmable GPU and 3.7 times faster on GT200

Table 1 continued

S. no.	Name of the algorithm	Base algorithm	Environment used	Improvement
7	GPU Sample sort	Sample sort	Processor: Intel Q6600 2.4 GHz machine, NVIDIA Graphics card: Tesla C1060	At least 25% and on average 68% faster than GPU Thrust Merge sort [4], and on average more than 2 times faster than GPU-quicksort [13]. At least 63% and on average 2 times faster than GPU Thrust Radix sort [4] for 64 bit integer keys
8	Deterministic Sample sort for GPUs	Sample sort	NVIDIA Graphics Card: Tesla, GTX 285 and GTX 260	Performance identical with Randomized Sample sort on GTX 285 and Tesla C1060
9	High Performance Comparison based Sorting Algorithm for Many Core GPUs	Merge sort and Bitonic sort	Processor: AMD Opteron880 working at 2.4 GHz, NVIDIA Graphics Card: 9800GTX+ with 512 MB Graphics RAM	Performs 20% better than Merge sort of Satish et al. [4]. Performs substantially better than Bitonic sort of Baraglia [20] and Quick sort of Cederman et al. [13]
10	Scan Primitives for GPU Computing	Quick sort	Processor: Intel Xeon working at 3.00 GHz. NVIDIA Graphics Card: NVIDIA GeForce 8800 GTX	Slower than Split based Radix sort per block [29], STL sort [15], CPU C implementation of Quick sort, Split based Global Radix sort
11	NVIDIA CUDA Dynamic Parallel Quick sort	Quick sort	Processor: Intel Xeon E5-2667 2.90 GHz NVIDIA Graphics card: NVIDIA Tesla Kepler k20	1.5 times faster than GPU-quicksort [13]
12	CUDA-Quicksort: An Improved GPU-based Implementation of Quick sort	Quick sort	Processor: Intel Xeon E5-2667 2.90 GHz NVIDIA Graphics card: NVIDIA Tesla Kepler k20	Outperformed GPU-quicksort [13], CDP Quick sort [27], the Radix sort of the thrust library [8], based on [4], Bitonic sort [31] and Merge sort [4]

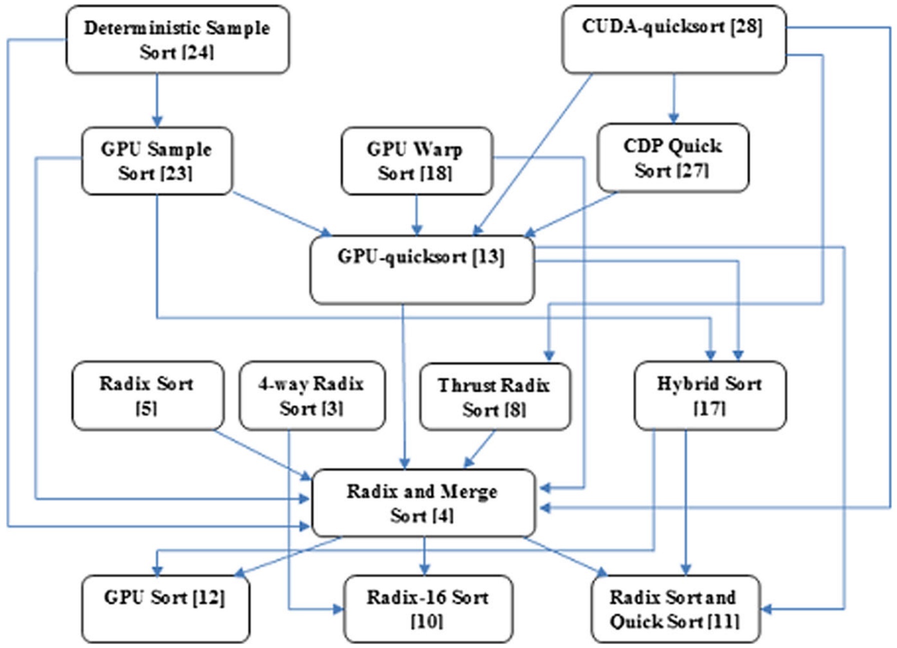

Fig. 1 Performance comparison of the algorithms

Sengupta et al. [11] made a first attempt to implement Quick sort on the GPU. However, the performance of Quick sort was slow. Cederman et al. [13] implemented Quick sort on a GPU and it was the first comprehensive and appropriate attempt to do so. NVIDIA CUDA Dynamic Parallel Quick sort [27] uses dynamic parallelism to implement Quick sort. Dynamic parallelism is an appropriate solution for Quick sort since it is of a recursive nature. Manca et al. [28] improved upon GPU-quicksort [13] and achieved appreciable results. CUDA-quicksort [28] being the most recent algorithm, it outperforms many of its predecessors including NVIDIA CUDA Dynamic Parallel Quick sort [27], Thrust Radix sort [8] for certain distributions and Bitonic sort [31].

7 Conclusion

This paper presented a survey of GPU sorting algorithms. Four sorting algorithms were selected for this survey: Radix sort, Merge sort, Sample sort and Quick sort. Working and performances of these algorithms as claimed by their authors were also presented. Finally, specific details of these algorithms are summarized in a table. A study of these algorithms shows that CUDA-quicksort, being the most recent algorithm and it outperforms many of its predecessors, including NVIDIA CUDA Dynamic Parallel Quick sort which uses dynamic parallelism, Thrust Radix sort for certain distributions, and Bitonic sort. Deterministic Sample sort also performs better than its counterparts. Deterministic Sample sort outperforms GPU Sample sort which in turn is better than many sorting algorithms such as GPU-quicksort, Hybrid sort etc. Due to the highly optimal complexity of Radix sort, it is a difficult to outrival algorithm on GPU.

References

1. Satish, N., Kim, C., Chhugani, J., Nguyen, A.D., Lee, V.W., Kim, D., Dubey, P.: Fast sort on CPUs and GPUs: a case for bandwidth oblivious SIMD sort. In: Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data, pp. 351–362 (2010)
2. Merrill, D., Grimshaw, A.: High performance and scalable radix sorting: a case study of implementing dynamic parallelism for GPU computing. Parallel Proc. Lett. **21**(2), 245–272 (2011)
3. Ha, L., Kruger, L., Silva, C.T.: Fast four-way parallel radix sorting on GPUs. Comput. Graph. Forum **28**(8), 2368–2378 (2009)
4. Satish, N., Harris, M., Garland, M.: Designing efficient sorting algorithms for manycore GPUs. In: Proceedings of the IEEE International Symposium on Parallel and Distributed Processing (IPDPS), pp. 1–10 (2009)
5. Huang, B., Gao, J., Li, X.: An empirically optimized radix sort for GPU. In: Proceedings of the IEEE International Symposium on Parallel and Distributed Processing with Applications, pp. 234–241 (2009)
6. Harris, M., Owens, J., Sengupta, S., Zhang, Y., Davidson, A.: Cudpp: Cuda Data Parallel Primitives Library (2007). Accessed Aug 2015
7. Harris, M.: Optimizing Parallel Reduction in CUDA. Technical Report. NVIDIA Developer Technology Website/projects/reduction/doc/reduction.pdf (2007)
8. Merrill, D.G., Grimshaw, A.S.: Revisiting sorting for GPGPU stream architectures. In: Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques, pp. 545–546 (2010)
9. Gamma, E., Johnson, R., Helm, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-oriented Software. Addison-Wesley, Reading (1994)
10. Grand, S.L.: Broad-phase collision detection with CUDA. In: Nguyen, H. (ed.) GPU Gems 3, pp. 677–697. Addison Wesley, Reading (2007)
11. Sengupta, S., Harris, M., Zhang, Y., Owens, J.D.: Scan primitives for GPU computing. In: Proceedings of the 22nd ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware, pp. 97–106 (2007)
12. Govindaraju, N., Gray, J., Kumar, R., Manocha, D.: GPUTeraSort: High performance graphics coprocessor sorting for large database management. In: Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data, pp. 325–336 (2006)
13. Cederman, D., Tsigas, P.: Gpu-quicksort: A practical quicksort algorithm for graphics processors. J. Exp. Algorithmics **14**(1.4) (2009)
14. Herf, M.: Radix Tricks. http://stereopsis.com/radix.html (2001). Accessed Jan 2016
15. GCC. Standard Template Library. http://gcc.gnu.org (2008). Accessed Nov 2015
16. Intel threading building blocks 2.1. http://www.threadbuildingbuildingblocks.org (2008). Accessed Sept 2015
17. Sintorn, E., Assarsson, U.: Fast parallel GPU-sorting using a hybrid algorithm. J. Parallel Distrib. Comput. **68**(10), 1381–1388 (2008)
18. Ye, X., Fan, D., Lin, W., Yuan, N., Ienne, P.: High performance comparison-based sorting algorithm on many-core GPUs. In: Proceedings of the IEEE International Symposium on Parallel and Distributed Processing (IPDPS), pp. 1–10 (2010)
19. Musser, D.R.: Introspective sorting and selection algorithms. Softw. Pract. Exp. **27**(8), 983–993 (1997)
20. Baraglia, R., Capannini, G., Nardini, F.M., Silvestri, F.: Sorting using bitonic network with CUDA. In: Proceedings of the 7th Workshop on Large-Scale Distributed Systems for Information Retrieval (LSDS-IR), Boston, USA (2009)
21. Dusseau, A.C., Culler, D.E., Schauser, K.E., Martin, R.P.: Fast parallel sorting under LogP: experience with the CM-5. IEEE Trans. Parallel Distrib. Syst. **7**(8), 791–804 (1996)
22. Blelloch, G.E., Leiserson, C.E., Maggs, B.M., Plaxton, C.G., Smith, S.J., Zagha, M.: A comparison of sorting algorithms for the connection machine CM-2. In: Proceedings of the Third Annual ACM Symposium on Parallel Algorithms and Architectures, pp. 3–16 (1991)
23. Leischner, N., Osipov, V., Sanders, P.: GPU sample sort. In: Proceedings of the IEEE International Symposium on Parallel and Distributed Processing (IPDPS), pp. 1–10 (2010)

24. Dehne, F., Zaboli, H.: Deterministic sample sort for GPUs. Parallel Process. Lett. **22**(3), CoRR. arXiv:1002.4464 (2012)
25. Chen, S., Qin, J., Xie, Y., Zhao, J., Heng, P.A.: A fast and flexible sorting algorithm with cuda. In: Proceedings of the International Conference on Algorithms and Architectures for Parallel Processing, pp. 281–290 (2009)
26. Blelloch, G.E.: Vector Models for Data-Parallel Computing. MIT Press, Cambridge (1990)
27. Cuda toolkit documentation 6.5. http://docs.nvidia.com/cuda/cuda-samples/index.html. Accessed July 2015
28. Manca, E., Manconi, A., Orro, A., Armano, G., Milanesi, L.: CUDA-quicksort: an improved GPU-based implementation of quicksort. Concurr. Comput.: Pract. Exp. **28**(1), 21–43 (2016)
29. Harris, M., Sengupta, S., Owens, J.D.: Parallel prefix sum (scan) with CUDA. In: Nguyen, H. (ed.) GPU Gems 3, pp. 851–876. Addison Wesley, Reading (2007)
30. Govindaraju, N.K., Raghuvanshi, N., Henson, M., Manocha, D.: A Cache-efficient Sorting Algorithm for Database and Data Mining Computations Using Graphics Processors. University of North Carolina (2005)
31. Batcher, K.E.: Sorting networks and their applications. In: Proceedings of the 1968 Spring Joint Computer Conference, pp. 307–314 (1968)