Streaming String Sort : A Hybrid String Sort on CPU+GPU

Kalpit Thakkar
Center for Visual Information and Technology
International Institute Of Information Technology
Hyderabad, India – 500032
kalpit.thakkar@research.iiit.ac.in

P J Narayanan
Center for Visual Information and Technology
International Institute Of Information Technology
Hyderabad, India – 500032
pjn@iiit.ac.in

***Abstract*—This paper explains the design of a hybrid string sort algorithm using CPU+GPU and explores the performance traits of the algorithm. Datasets which don't fit in the GPU memory can be sorted using our algorithm, although the maximum size is bound by the CPU memory and the number of strings in the dataset. Our algorithm performs the sorting by loading strings from left to right, in successive columns of length "K" and sorts them in each sorting step. Value of "K" is dynamic as singletons are removed when they are formed. CPU performs the data processing step of loading the next column of "K" bytes from each string to be sorted and GPU performs the sorting step, in an overlapped manner. A context lag between CPU and GPU is observed as they perform overlapped executions and next iteration depends on the previous. Results are shown on different datasets having sizes ranging from 2 MB to 6.4 GB and different after-sort tie-lengths. We compare our runtimes with the best CPU string sorting implementations available today and explore the possibilities for improvement in our algorithm.**

Keywords*—*String sorting, tie-length, context lag

I. Introduction

String sorting is a well studied problem. There are numerous applications where data consisting of strings needs to be sorted [PZM$^+$12], [VHPN09], [MGG12], [LGS$^+$09], [GL10], [ASA$^+$09]. Sorting strings involves handling variable-length keys as opposed to fixed-length keys. It is more challenging as (a) string lengths are variable, and swapping strings is not as simple as swapping fixed-length records, (b) strings are compared one character at a time, instead of the entire key being compared and require more instructions, and (c) strings are traditionally accessed using pointers; the strings are not moved from their original locations due to string copying costs [SW08]. With present day architectures with multiple cores, huge opportunities for exploiting the parallelism are offered by the hardware. However, the memory that is available on highly parallel architectures might not be sufficient in many scenarios, especially when large databases are involved [GGKM06]. A GPU may not be able to hold all the information required for sorting a large number of strings all at once. String sorting approaches using CPU and GPU in conjunction overcome this limitation and facilitate memory scalibility. An algorithm using external memory sorting suffers from an I/O bottleneck though [AFGV97], [FPP06], because of low latency memory transfers between the internal and external memory.

Streaming string sort is designed to sort large string data on heterogeneous multicore architectures. Fast and efficient string sort on GPU is used to sort the strings [DN13] and the string data is streamed from the CPU to the GPU, when needed. Comparison-based string sorting where CPU and GPU are used [BSK13], and a fast BWC algorithm which uses all cores available on the system, including both the CPU and GPU [DN15] are studied, both of which use a static scheme of work distribution in order to achieve overlapped execution of CPU and GPU. In streaming sort, CPU and GPU depend on each other, introducing more complexities to handling the parallelism and the context lag due to overlap. The context lag manifests itself in the form of information delay between the GPU and CPU, in terms of arrangement of strings and the number of unsorted strings.Context lag is handled by maintaining an index map on the GPU and doing an index adjustment on the GPU at the end of each step.

We provide results on benchmark datasets and datasets designed to demonstrate scalibility to large data size. Average after-sort tie-lengths define the difficulty of the dataset [DN13] and we present results on datasets with different after-sort tie-lengths. The main contribution of this paper includes (a) a scalable, hybrid, parallel string sorting algorithm on CPU+GPU platform, which uses a fast, efficient GPU string sort and (b) Comparative analysis of our algorithm and a set of parallel CPU string sorting algorithms [BES14], on high-end and low-end GPUs. Possibilities and opportunities for improvement in our algorithm are explored using the results obtained.

II. Related Work

In this section, previous sorting and string sorting algorithms on CPU, GPU and a heterogeneous (CPU + GPU) architecture have been noted.

A. CPU Algorithms

There are several CPU string sorting algorithms that have been developed in the past. Some noteworthy string algorithms are: Multi-key quicksort [BS97] is an algorithm that combines quicksort and radix sort. Burstsort [SZR07], [SW08] combines burst-trie [HZW02] with multi-key quicksort and radix sort [MBM93]. MSD radix sort [KR09] organizes the strings into buckets recursively based on each successive character in the string corresponding to that recursion depth. A two-pass counting method, where first pass counts the number of buckets required and second pass scatters the string data and a one-pass dynamic method, where the buckets are generated and resized dynamically during execution, have been developed [KR09]. MSD radix sort uses insertion sort for sorting small buckets and does not move entire strings but manipulates the string pointers. Burstsort inserts the strings into a data structure called *burst-trie*, in turn organizing them into buckets which can be sorted within CPU cache memory. Once sorted in cache, these strings do not need to be merged because the order is already lexicographic in nature. [BES14] developed a parallel super scalar sample sort for sorting strings on CPU which gains high speedups over previous parallel and sequential string sorting algorithms on CPU.

B. GPU Algorithms

An implementation of quicksort on GPU was developed [CT10]. A GPU sample sort [LOS10] is a randomized version of the GPU quick sort. At any given iteration, a set of splitters are chosen that divides the list into many parts and each sublist is processed independently in parallel. An implementation of radix sort [SHG09] on GPU divided each pass of the radix sort into four phases which were synchronized globally. The focus is on minimizing scatters to global memory and maximizing coherence among scatters. Merrill and Grimshaw designed a radix sort, which is the fastest GPU implementation of radix sort available [MG11]. An efficient merge sort on GPU has been developed [DTGO12] which handles strings well, however, it accesses costly global memory to resolve ties. Deshpande and Narayanan developed a GPU string sort [DN13] which falls into the category of counting method of [KR09]. It loads strings from left to right in steps, moving only indices and small prefixes for efficiency. They reduce the number of sort steps by adaptively consuming maximum string bytes based on the number of segments in each step. Using Thrust primitives and removing singletons improves their performance. A comparison-based string sorting algorithm for a multi-GPU system was developed [TVJ$^+$13]. It uses a pivot selection algorithm and avoids data marshalling steps altogether alongwith load balancing for the multi-GPU system.

C. CPU+GPU Algorithms

Banerjee et al. developed a comparison-based sort using a CPU+GPU platform [BSK13]. It uses the idea of GPU sample sort and divides the input list into independent sublists using splitters chosen uniformly at random. These independent sublists are then sorted recursively and finally merged on the GPU. Their implementation reported upto 24% faster performance on a dataset of random strings. Their string sort is a combination of sample sort and merge sort.

III. Streaming String Sort

In this section, the algorithm for streaming string sort is explained. The pseudocode is given in Algorithm 1 and explained in detail in section C. Depending on the size of input, we use only GPU, only CPU or CPU+GPU for string sorting. If the input fits in GPU memory and can be sorted using only GPU, such that the number of strings is above a threshold, no CPU is involved in string sorting. In a CPU+GPU mode, GPU performs the string sorting and CPU performs only data processing. However, when the number of strings is below a threshold, only CPU is used.

Algorithm 1 Streaming Sort

```
1: $K \leftarrow$ Max. bytes that can be allocated per string
2: LOADNEXTKBYTES()
3: CUDAMEMCPY($hostToDevice$)
4: REPEAT
5:     threadID $\leftarrow$ OMP_GET_THREAD_NUM
6:     if threadID $= 0$ then
7:         GPUSTRINGSORT()
8:         GPULOADNEXTKEYS()
9:         GPUELIMINATESINGLETONS()
10:        GPUSETSEGMENTBYTES()
11:        if GPUSTEPCOMPLETE then
12:            GPUINDEXADJUSTMENT()
13:            GPUENDSTEP
14:    if threadID $= 1$ then
15:        LOADNEXTKBYTES()
16:        CUDAMEMCPY($hostToDevice$)
17:    waitForThreadsMerge
18:    CUDAMEMCPY($deviceToHost$)          ▷ index array
19: UNTIL all are sorted
```

A. Overall System

The algorithm is divided into two parts: GPU string sorting loop and CPU loop loading the next bytes from each string that is to be sorted. Both these parts are executed in an overlapped manner using OpenMP and the overall system is shown in Figure 1. GPU string sorting loop handled by Thread 1 is built on the GPU string sort [DN13], with some changes in order to handle the context lag between CPU and GPU as they are running in parallel. The modifications are explained in the section B. The context lag is explained in detail in section D.

B. GPU String Sort

The significant features of the GPU string sort upon which we build are (a) instead of moving entire strings in memory, only its pointers (indices) are shuffled, (b) singletons are removed (buckets which contain a single string) as they are formed, to reduce the problem size in each iteration, (c) length of segment ID for each key is adaptively fixed to sort longest possible prefix in each step [DN13]. The string sort was modified due to CPU running in parallel with GPU and strings not fitting on GPU all at once. Due to CPU and GPU running in an overlapped manner, CPU lags behind GPU by one sorting step. The next sorting step depends on previous one in terms of the order of strings because loading next keys for each string requires the order to be maintained and the CPU does not know the order until GPU finishes its step. The major modifications include (a) an index map is maintained now on GPU which maps the string indices in device array to the string number in string array prepared on host, (b) at the end of each GPU step, the index map is used to modify device index array so that GPU loads the next keys from the correct position in string array for each string. A minor modification is made: while loading the next keys on the GPU, the required characters were accessed one by one from global memory. They are now fetched by two 8-byte aligned accesses from global memory and stored in a register. Accessing them from a register rather than the global memory in the GPU kernel avoids some costly global accesses.

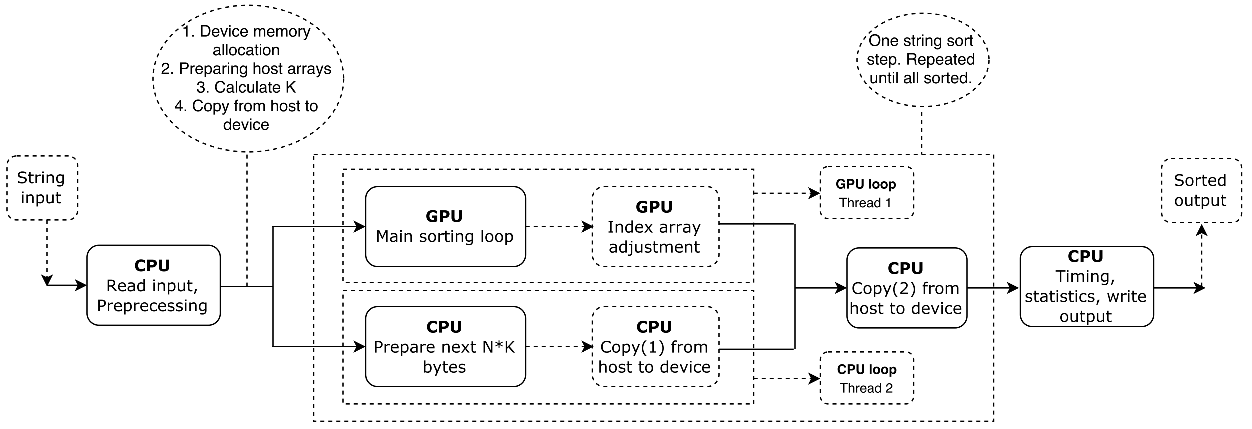
1. Device memory allocation
2. Preparing host arrays
3. Calculate K
4. Copy from host to device
One string sort step. Repeated until all sorted.
String input
CPU Read input, Preprocessing
GPU Main sorting loop
GPU Index array adjustment
CPU Prepare next N*K bytes
CPU Copy(1) from host to device
GPU loop Thread 1
CPU loop Thread 2
CPU Copy(2) from host to device
CPU Timing, statistics, write output
Sorted output

Figure 1: OVERALL SYSTEM ORGANIZATION

C. The Algorithm

The pseudocode for the streaming sort is given in Algorithm 1. Assuming number of strings is above a threshold, if input fits on the available GPU memory after allocating some constant amount of memory required for sorting, string sorting is performed using only GPU. The lines of code in the loop marked by **REPEAT** is what is referred to as a "step". It is also shown in Figure 1. This loop contains a GPU loop and a CPU loop. These loops run in an overlapped manner and hence there is a context lag between the two. Context lag is handled by using an index map (d_IM) to maintain the same order of strings on both. In GPU loop, when the *N*K* bytes are exhausted, an index adjustment is done on for handling the context lag (line 14).

1) Context lag: This takes place between CPU and GPU due to execution in an overlapped fashion. CPU prepares data for step **i+1** when GPU performs sorting step **i**. The CPU does not know what would be the value of N and the structure of d_IV for step **i+1** until step **i** is completed on GPU. Hence, there is a context lag between CPU and GPU in terms of information about the number and order of strings. Due to context lag, CPU loads the strings for step **i+1** according to information from step **i-1**. It results in loading more strings than required. Though context lag forces some extra work, it ensures correctness in loading the next keys for each string remaining after GPU completes sorting step **i**. The order of the strings loaded by CPU is realized on GPU by index adjustment, explained below.

2) Index adjustment: To understand index adjustment, the structure of the device index array needs to be understood first. Each entry in the device index array has two parts (a) the index of the string in the global string array on host (h_SV) and (b) the string number in the device string array prepared by the host (d_SV). Part (a) of d_IV remains constant throughout the string sorting procedure, as we do not move the entire strings around. Part (b) of each entry keeps changing according to the string number in d_SV, as the order keeps changing (or may remain same) in each step. Index of each string in d_SV would be a multiple of K and hence only the string number is required. The index map is a mapping from a string number to it's number in the device string array prepared by the host. It is used to modify part (b) of d_IV such that for an entry d_IV[i] it's new value for part (b) would be d_IM[(d_IV[i]<<32)>>32]. The index adjustment is done on GPU. Once the device index array is modified, the index map is modified according to it, because device index array is sent to host. CPU uses the order of strings specified by device index array to prepare the string array.

Figure 2 explains the algorithm using first two sorting steps on the input shown. When sorting step-2 is running, the CPU loads (for step-3), K (= 2) bytes from N (= 7) strings as N = 7 after step-1. This explains the context lag. Now, to understand the index adjustment, after step-1 is over, part(b) of index array does not change as the initial index map is of the form d_IM[i] = i. However, the index map changes in step-1 as strings have shuffled due to sorting. If value of part(b) in index array is V and the entry is at index I in index array, the index map stores it as d_IM[V] = I, basically saying that the string number was (V+1) initially but now it is (I+1). Using this index map, it can be observed that in step-2 the part(b) of index array is modified and in turn the index map is also modified. This explains the concept of index adjustment.

IV. Experimental Results and Analysis

We will explain the experiments performed in order to test and analyze the performance of our algorithm.

A. Datasets

The datasets used for testing the algorithm are listed in Table 1 with their descriptions. In order to test the algorithm properly, there was a need to prepare datasets with sizes greater than GPU memory. Hence, the datasets *norandom*, *random* and *partlyrandom* have been prepared. The other datasets are standard benchmark datasets[1] and *filelist* is a real world dataset which has been prepared by listing the filepaths obtained from a server.

B. Discussion

From Table IV it can be observed that fixed-K has better runtime than dynamic-K. Using a fixed-K throughout sorting makes the CPU work less while preparing the data for next

[1]Sinha's collection from https://panthema.net/2013/parallel-string-sorting

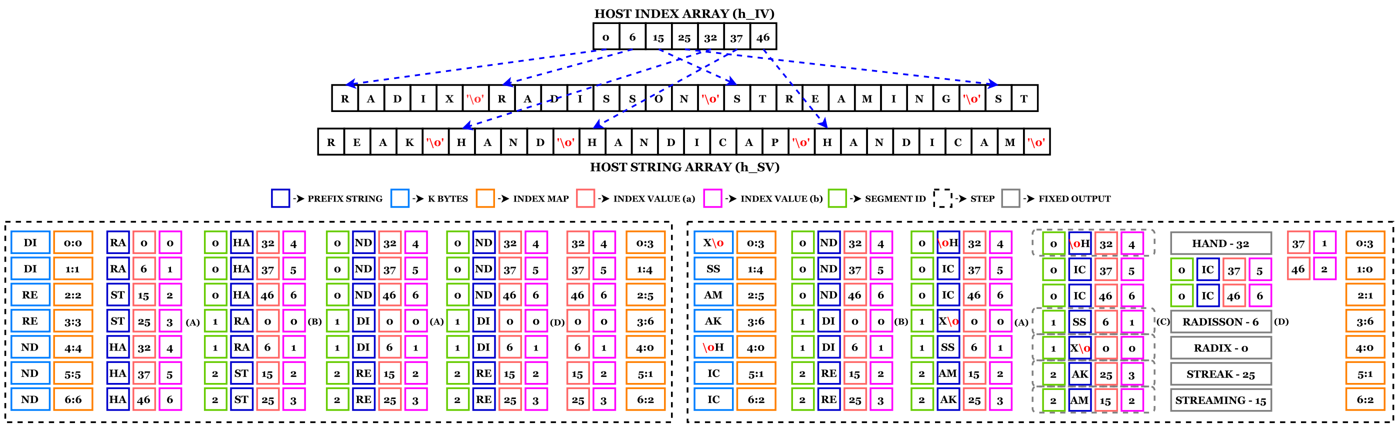

Figure 2: TRANSITIONS ARE MARKED AS (A), (B), (C) AND (D). (A) MEANS SORT, (B) MEANS LOAD SUCCESSORS, (C) MEANS ELIMINATE SINGLETONS AND (D) MEANS INDEX ADJUSTMENT. THE DIAGRAM EXPLAINS THE FIRST TWO SORTING STEPS OF SORTING THE INPUT ACCORDING TO OUR ALGORITHM.

Name	Size	Description	Max. tie length	Avg. tie length
dictcalls	2.2 MB	100187 strings containing opcodes	35	15
artificial-5	50 MB	10^6 strings with A, varying lengths (1 to 100)	100	50
random	97 MB	10^6 strings of length approx. 100	5	2
artificial-2	98 MB	10^6 strings with A repeated approx. 100 times	101	100
artificial-4	162 MB	10^7 strings of characters a to i	80	13
words	238 MB	Approx. 19 million words with no duplicates	157	7
genome	302 MB	31623000, approx. 30 million strings of a,t,g,c	9	8
url	305 MB	10^7 strings containing URLs	215	29
filelist	971 MB	10^7 strings containing filepaths	228	25
urls_large	2.4 GB	4*10^7 URLs, subset of big URL dataset of [BES14]	3404	56
norandom	4.8 GB	10^7 strings containing only “a”, having different lengths	1006	78
totalrandom	4.8 GB	10^7 random strings, having different lengths	7	3
partlyrandom	6.4 GB	10^7 strings, made by permuting a set of 4 strings of fixed length, different number of times	495	66

Table I: DATASETS USED FOR THE EXPERIMENTS

Dataset	GTX 580		GTX Titan		CPU string sort				Speedup
	# Steps	Time	# Steps	Time	pS^5-unroll	pS^5-equal	pRS-16bit	radixR_CE7	
dictcalls	1	**8**	1	10	22	21	21	**12**	1.5
artificial-5	1	77	1	**75**	171	**166**	290	295	2.2
random	1	9	1	**8**	140	144	**107**	119	13.4
artificial-2	1	106	1	**83**	**216**	220	562	464	2.6
artificial-4	1	206	1	**152**	**627**	705	837	1125	4.1
words	1	243	1	**175**	**1136**	1185	1211	1862	6.5
genome	–	–	1	**262**	**1433**	1592	1996	2526	5.5
url	1	432	1	**316**	**734**	756	1069	1945	2.3

Table II: TIME IN MILLISECONDS TO SORT THE BENCHMARK DATASETS ON THREE DIFFERENT ARCHITECTURES. (I) GEFORCE GTX 580 + INTEL CORE I7 4790K QUAD-CORE 4.4GHZ, (II) GEFORCE GTX TITAN + INTEL CORE I7 4790K QUAD-CORE 4.4GHZ, (III) INTEL CORE I7 4790K QUAD-CORE 4.4GHZ (32G RAM). THE FOUR SORTING ALGORITHMS USED ARE FROM [BES14] WITH THE SAME NOTATION.

step. It wastes GPU memory when the number of strings remaining are very less, while dynamic-K doesn't waste memory. However, dynamic-K increases CPU time. There is a tradeoff between better use of memory bandwidth on GPU and faster CPU data preparation step. The speedup in CPU preparation step may be good but the total CPU time speedup is negligible or no speedup is noticed at all, as a fixed-K also means that GPU would exhaust the strings much faster. Dynamic-K is better than fixed-K in terms of using memory bandwidth and fixed-K implementation uses more number of steps than taken by a dynamic-K one. These observations lead us to believe that using dynamic-K is a better algorithm choice than fixed-K if better memory bandwidth utilization is needed and fixed-K is better if a faster runtime is required.

Obtaining results on different architectures has provided an insight into how the algorithm behaves on different GPUs, having different global memories and compute capabilities. GTX Titan has compute v3.5 (6 GB of global memory) and GTX 580 has compute v2.0 (1.5 GB of global memory), as reported by the CUDA SDK v7.5 that we have used. Table II and III show the runtimes on different architectures for all the datasets. No runtime shown means either the CPU memory was inadequate or the number of strings was high enough to disallow an acceptable initial value of K. This is one drawback of the algorithm. The speed of CPU in preparing the next column of strings to be sorted varies according to architecture. Less number of steps do not necessarily imply that the total time taken for sorting would be less. Larger value of K implies more CPU time in preparation step, but less number of steps. Smaller value of K implies less CPU time in preparation step, but more number of steps. In Table II and III , we compare our runtimes with the runtimes of four best CPU algorithms chosen from the algorithms in [BES14]. The code of the four implementations[2] has been put together for ease of comparison among them and we utilized it for testing purposes. For benchmark datasets, our algorithm runs in only-GPU mode as they fit in GPU memory and we gain a maximum speedup of 13x for the dataset *random*. *filelist* also runs in only-GPU mode and gains a speedup of 2.4x over the best CPU string sorting time. For the datasets *norandom*, *totalrandom* and *partlyrandom*, the timings are interesting. Our

[2]https://panthema.net/2013/parallel-string-sorting/parallel-string-sorting-0.6.5.tar.bz2

Dataset	GTX 580		GTX Titan		CPU string sort				Speedup
	# Steps	Time	# Steps	Time	pS^5-unroll	pS^5-equal	pRS-16bit	radixR_CE7	
filelist	1	991	1	**692**	1669	**1643**	3619	4545	2.4
norandom	38	23065	11	**12966**	11921	**9705**	22849	32946	0.7
totalrandom	1	**374**	1	580	3243	3258	**3142**	3458	8.4
partlyrandom	10	3685	8	3281	4999	**4751**	6659	7690	1.4
urls_large	–	–	137	9641	4077	**4056**	7337	12590	0.4

Table III: TIME IN MILLISECONDS TO SORT THE DATASETS (USED FOR SHOWING SCALABILITY) ON THREE DIFFERENT ARCHITECTURES. (I) GEFORCE GTX 580 + INTEL CORE I7 4790K QUAD-CORE 4.4GHZ, (II) GEFORCE GTX TITAN + INTEL CORE I7 4790K QUAD-CORE 4.4GHZ, (III) INTEL CORE I7 4790K QUAD-CORE 4.4GHZ (32G RAM). THE FOUR SORTING ALGORITHMS USED ARE FROM [BES14] WITH THE SAME NOTATION.

Dataset	GTX Titan				GTX 580			
	Dynamic K		Fixed K		Dynamic K		Fixed K	
	# Steps	Total	# Steps	Total	# Steps	Total	# Steps	Total
norandom	13	**12684**	16	12966	38	23065	53	21194
totalrandom	1	585	1	580	1	384	1	**374**
partlyrandom	4	4634	8	**3281**	10	3935	26	4284
urls_large	10	9720	137	**9641**	–	–	–	–

Table IV: RUNTIME IN MILLISECONDS FOR FIXED-K AND DYNAMIC-K

algorithm performs really well on the *totalrandom* dataset, gaining a speedup of 8.4x. The dataset *norandom* has all strings of same character and *urls_large* has URLs with lots of similarity among them. It is a bad case for our algorithm and it performs worse than CPU. On *partlyrandom*, CPU performs as good as our streaming string sort. This means that CPU performs better than our algorithm on datasets which have long tie-lengths, as in our algorithm breaking the ties requires loading many successive columns from the string data.

V. Conclusion

As radix sort is the fastest on GPUs, it should be used to develop string sorting algorithms which use it in order to gain performance. Our algorithm is an exploration of this possibility. Our algorithm is not suitable for datasets that fit in the GPU memory as the context lag between the CPU and GPU cannot be avoided when they are running in an overlapped manner, which would make our algorithm perform worse than an efficient only-GPU implementation. When the number of strings is below a threshold and the tie-lengths are high, an only-GPU or CPU+GPU implementation takes far longer than only-CPU implementation. The advantage of this algorithm is that there is no need to perform a static work distribution initially for gaining parallelism by dividing work into independent blocks; it is actually handled on the fly. The use of a fixed-K when CPU+GPU mode is used, though wastes some GPU memory, makes the sorting faster by avoiding long CPU times.

References

[AFGV97] Lars Arge, Paolo Ferragina, Roberto Grossi, and Jeffrey Scott Vitter. On sorting strings in external memory (extended abstract). STOC '97, pages 540–548, 1997.

[ASA+09] Dan A. Alcantara, Andrei Sharf, Fatemeh Abbasinejad, Shubhabrata Sengupta, Michael Mitzenmacher, John D. Owens, and Nina Amenta. Real-time parallel hashing on the gpu. *ACM Trans. Graph.*, 28(5):154:1–154:9, 2009.

[BES14] Timo Bingmann, Andreas Eberle, and Peter Sanders. Engineering parallel string sorting. *CoRR*, 2014.

[BS97] Jon L. Bentley and Robert Sedgewick. Fast algorithms for sorting and searching strings. SODA '97, pages 360–369, 1997.

[BSK13] Dip Sankar Banerjee, Parikshit Sakurikar, and Kishore Kothapalli. Fast, scalable parallel comparison sort on hybrid multicore architectures. IPDPSW '13, pages 1060–1069, 2013.

[CT10] Daniel Cederman and Philippas Tsigas. Gpu-quicksort: A practical quicksort algorithm for graphics processors. *J. Exp. Algorithmics*, pages 4:1.4–4:1.24, 2010.

[DN13] A. Deshpande and P. J. Narayanan. Can gpus sort strings efficiently? In *International Conference on High Performance Computing*, pages 305–313, 2013.

[DN15] Aditya Deshpande and P. J. Narayanan. Fast burrows wheeler compression using all-cores. IPDPSW '15, pages 628–636, 2015.

[DTGO12] A. Davidson, D. Tarjan, M. Garland, and J. D. Owens. Efficient parallel merge sort for fixed and variable length keys. In *InPar '12*, pages 1–9, 2012.

[FPP06] Rolf Fagerberg, Anna Pagh, and Rasmus Pagh. *External String Sorting: Faster and Cache-Oblivious*, pages 68–79. STACS '06, Springer Berlin Heidelberg, Berlin, Heidelberg, 2006.

[GGKM06] Naga Govindaraju, Jim Gray, Ritesh Kumar, and Dinesh Manocha. Gputerasort: High performance graphics co-processor sorting for large database management. SIGMOD '06, pages 325–336, 2006.

[GL10] Kirill Garanzha and Charles Loop. Fast ray sorting and breadth-first packet traversal for gpu ray tracing. *Computer Graphics Forum*, pages 289–298, 2010.

[HZW02] Steffen Heinz, Justin Zobel, and Hugh E. Williams. Burst tries: A fast, efficient data structure for string keys. *ACM Trans. Inf. Syst.*, pages 192–223, 2002.

[KR09] Juha Kärkkäinen and Tommi Rantala. Engineering radix sort for strings. SPIRE '08, pages 3–14, 2009.

[LGS+09] C. Lauterbach, M. Garland, S. Sengupta, D. Luebke, and D. Manocha. Fast bvh construction on gpus. *Computer Graphics Forum*, pages 375–384, 2009.

[LOS10] N. Leischner, V. Osipov, and P. Sanders. Gpu sample sort. In *IPDPS '10*, pages 1–10, 2010.

[MBM93] Peter M. McIlroy, Keith Bostic, and M. Douglas McIlroy. Engineering radix sort. *COMPUTING SYSTEMS*, 6:5–27, 1993.

[MG11] DUANE MERRILL and ANDREW GRIMSHAW. High performance and scalable radix sorting: A case study of implementing dynamic parallelism for gpu computing. *Parallel Processing Letters*, pages 245–272, 2011.

[MGG12] Duane Merrill, Michael Garland, and Andrew Grimshaw. Scalable gpu graph traversal. *SIGPLAN Not.*, pages 117–128, 2012.

[PZM+12] R. A. Patel, Y. Zhang, J. Mak, A. Davidson, and J. D. Owens. Parallel lossless data compression on the gpu. In *InPar '12*, pages 1–9, 2012.

[SHG09] Nadathur Satish, Mark Harris, and Michael Garland. Designing efficient sorting algorithms for manycore gpus. IPDPS '09, pages 1–10, 2009.

[SW08] Ranjan Sinha and Anthony Wirth. *Engineering Burstsort: Towards Fast In-Place String Sorting*, pages 14–27. Experimental Algorithms Workshop '08, Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[SZR07] Ranjan Sinha, Justin Zobel, and David Ring. Cache-efficient string sorting using copying. *J. Exp. Algorithmics*, 2007.

[TVJ+13] Ivan Tanasic, Lluís Vilanova, Marc Jordà, Javier Cabezas, Isaac Gelado, Nacho Navarro, and Wen-mei Hwu. Comparison based sorting for systems with multiple gpus. GPGPU-6, pages 1–11, 2013.

[VHPN09] Vibhav Vineet, Pawan Harish, Suryakant Patidar, and P. J. Narayanan. Fast minimum spanning tree for large graphs on the gpu. HPG '09, pages 167–171. ACM, 2009.