Can GPUs Sort Strings Efficiently?

Aditya Deshpande and P J Narayanan
Center for Visual Information Technology,
International Institute of Information Technology,
Hyderabad, India

***Abstract*—String sorting or variable-length key sorting has lagged in performance on the GPU even as the fixed-length key sorting has improved dramatically. Radix sorting is the fastest on the GPUs. In this paper, we present a fast and efficient string sort on the GPU that is built on the available radix sort. Our method sorts strings from left to right in steps, moving only indexes and small prefixes for efficiency. We reduce the number of sort steps by adaptively consuming maximum string bytes based on the number of segments in each step. Performance is improved by using Thrust primitives for most steps and by removing singleton segments from consideration. Over 70% of the string sort time is spent on Thrust primitives. This provides high performance along with high adaptability to future GPUs. We achieve speed of up to 10 over current GPU methods, especially on large datasets. We also scale to much larger input sizes. We present results on easy and difficult strings defined using their after-sort tie lengths.**

I. Introduction

Sorting is an important operation in data processing. Fast and efficient sorting has been available on the GPU for a few years. Radix sort performs the best on the GPU and its performance has improved steadily since its introduction [14], [16]. The Thrust template library contains fast sort implementations, with its radix sort by Merrill and Grimshaw being the fastest [14]. Thrust sort can be enhanced with a custom comparison operator [3]. Sorting is a vital data parallel primitive on the GPU and is a critical component of several algorithms [1], [8], [7], [10], [13], [15], [19]. The use of standard primitives opens the door for automatic performance improvements when the primitive's performance improves due to an improved implementation or an improved architecture, both of which happens with sort on the GPU. A primitive based implementation can thus be adapted efficiently to newer generation of architectures that differ significantly.

Sorting long and variable length keys is still a challenge on the GPU. The most typical problem is sorting of strings of arbitrary length, which is a common occurrence in different fields. In this paper, we study string sorting that is built over existing sort primitives on the GPU. Such an approach will result in future performance improvements in the primitives to translate to string sorting with no redesign. The developers of Thrust suggest the use of a custom comparator with the fixed-length sort primitives, to perform string sorting. The performance of this depends critically on the comparator implementation and is not fast enough perhaps due to the moving of a lot of data. Davidson et al. presented the first string sort implementation on the GPUs using an efficient merge sort for long and variable length keys [6]. Their performance is good, but scalability to large datasets may not be straightforward using merge sort as opposed to radix sort.

In this paper, we present a fast and efficient GPU string sort method based on available fixed-length radix sort. Our method outperforms both methods described above on large datasets. Our method is simple and scalable. We sort the string prefixes progressively from the left, using a key length that performs the best on radix sort. A straightforward MSD radix sort, which shuffles entire strings, will require excessive data movement that will slow the operation, as stated by Davidson et al. [6]. We, therefore, use fixed-length prefixes to represent the strings and move only a 4-byte string index in addition to the key. Each sort divides the strings into buckets or segments of common prefixes. Each segment needs to be sorted in further rounds within itself. This is done using a global radix sort with a key consisting of the segment id and further prefix characters from the string. Singleton segments represent strings that have found their final place and are removed to reduce the sorting problem size for later iterations. We also use maximum number of bytes in the key for the string characters in each step. This is done by using minimum number of bytes for the segment id adaptively in each step. This allows for more string bytes to be used in each sort step. Flexible prefix selection reduces the number of iterations needed, since now a larger part of the string is sorted per iteration, and results in significantly higher performance.

We present exhaustive results on several datasets to demonstrate the applicability and scalability of our method. We use the standard benchmark datasets as well as create our own to demonstrate scaling to larger problem sizes. The average after-sort tie-length quantifies the difficulty of sorting a given dataset. We present results on datasets with large variations in the tie-length. Our main contributions include the following. (a) A fast string sort implementation that achieves a speedup ranging from 1.8 to 19.7 over the current GPU methods. We reduce the memory movement of long-key radix sort by shuffling only string ids, for high performance. We reduce the number of iterations by removing singletons as and when they are formed. We sort the longest possible prefix in each step by adaptively fixing the length of segment id field. (b) We use the optimized primitives from available libraries effectively to implement most of the steps. About 70% of the total time is spent of such primitives. This uses the architectures effectively with less effort and provides high scalability and portability to future architectures. (c) We present results on a variety of natural and synthetic datasets to demonstrate the performance of our algorithm. We also analytically estimate the expected runtime. The experimental results are within realistic limits of these expected runtimes.

We are the first to exploit radix sort effectively for string sorting on the GPU. Our approach loads the string progressively from left to right only once; the segment id acts as a proxy for the prefix for later iterations. This reduces the

memory movement greatly and improves the performance. In contrast, a merge sort approach repetitively loads each string from the global memory (whenever ties occur) in every merge step. We show a speed up of more than 10 over the best GPU string sorting approaches. We obtain a sorting throughput of 83 MKeys/s on a dataset of 1 million random strings. Our approach can scale to an order of magnitude larger input size than the previously reported GPU string sort. On a 10 million words dataset we achieve a throughput of 65 MKeys/s.

II. Related Work

We review the relevant sorting and string sorting algorithms for the CPU and the GPU in following sections.

A. CPU String Sorting Algorithms

Most sorting algorithms are designed with the assumption that input keys are a few characters or integers (i.e. fixed-length keys). For keys which span more than a few integers or characters (i.e. strings), comparison based sorting algorithms use an iterative comparator between given keys. Radix sorting algorithms create and recursively sort buckets starting from most significant to the least significant integer or character. Comparison based sorts will perform $\Omega(Nlog(N))$ comparisons, each iterating over two strings to resolve ties. Naive radix sort procedures will shuffle the entire strings in each step. Both of these could be expensive in practice.

Several specialized string sorting algorithms have been developed over time. Most popular and efficient amongst such approaches are: mutli-key quicksort [4], Burstsort [17], [18] and MSD (most significant digit) radix sort [9]. These typically use a combination of two or more of the standard sorting algorithms augmented with a few additional steps for performance. Burstsort uses *burst-trie* data structure and a standard sorting algorithm [4], [12]. It organizes strings into small buckets by inserting them into a burst-trie, such that these buckets can be sorted within the CPU cache memory. The sorted buckets are already lexicographically ordered amongst each other and need not be merged later. Kärkkäinen and Rantala engineer an efficient radix sort for strings which involves repetitive application of radix sorting from most significant digit to the least significant one [9]. They develop *counting* based methods which require pre-computing the bucket size. To achieve this, they need to make two passes over the data per radix sort step, where the first pass computes bucket sizes and the next pass scatters the data. They also develop one-pass *dynamic* methods where buckets are generated and resized on the fly. They resort to simpler sorting viz. insertion sort when buckets are small enough. They avoid shuffling entire strings by manipulating pointers, cache successive characters of strings, and use supra-alphabets (2 byte characters instead of 1 byte) for small buckets, to reduce the sorting steps.

Our algorithm comes under the category of counting based method of [9], with actual implementation exploiting the efficient standard primitives viz. sort, scatter, scan etc. on the GPU. The MSD radix sort creates buckets in a depth first manner, while we show that a breadth-first pattern is more suited for exploiting the parallelism on the GPU. Also, our implementation can compare longer length of prefixes (maximum of $\approx$ 8 characters long) per radix sort step as compared to the two character limit of MSD radix sort. In Table VIII and Section IV-F, we show that our GPU algorithm gives significant speed up ($4\times$ - $17\times$) compared to both Burstsort and MSD radix sort algorithms on standard benchmark datasets.

B. GPU String Sorting Algorithms

The GPU has emerged as a massively parallel accelerator for problems that have a strong data parallel flavor. Several high performance sorting algorithms have been designed for it. Cederman and Tsigas developed an implementation of Quicksort [5], sample sort was implemented by Leischner et al. [11]. Satish et al. developed a radix sort on GPU which outperformed an 8-core CPU by $4\times$ [16]. Their performance benefits from the reducing of scattered writes to global memory by making use of on-chip shared memory. They also developed a fast merge sort, which merged small blocks that fit in the on-chip shared memory. Merrill and Grimshaw later developed a radix sort using fast scan primitives which is the fastest sort today [14]. They introduced optimizations like adapting the radix sort granularity (digit size) to fit the underlying GPU architecture, kernel fusion and serialization of threads for appropriate steps to reduce the global memory accesses. All these implementations assume a fixed key size of 32/64 bits and cannot handle variable or long keys.

Thrust also provides efficient primitives for fixed key length radix sort and merge sort [3]. Developers of Thrust suggest creating a user-defined string class and a custom iterative comparator to extend their sort to strings[1]. Thrust supports radix sort only for basic datatypes and resorts to a merge sort when provided with user defined datatypes and/or a custom comparator. Radix sort primitives are significantly faster as compared to merge sort and such an approach fails to exploit them.

Recently, Davidson et al. developed an efficient merge sort based string sorting procedure that handles variable length keys well [6]. They prefer register packing (by creating a limited number of threads) against over-utilization of GPU. Input was divided into blocks of fixed size, $m = 1024$, and $m/8$ threads (per SM) were created to sort 8 elements each through a carefully designed bitonic sorting network. Each thread then modified its search-space (neighboring 8, 16, 32 and so on elements) to create the final sorted block of size m. In the next step, on each GPU SM two blocks were merged which doubles the size of the output block and halves the number of blocks. At the end when blocks to be merged are few, they use a scheme where all threads (or cuda blocks) cooperatively merge them. For variable length keys, they initially sort the first 4 characters and load successive characters from global memory in case of ties. As the merge procedure nears conclusion, they observe that, comparisons are made between more similar strings and ties take longer to resolve. Resolving ties involves accessing high latency global memory and also causes thread divergence, these issues cannot be easily solved. Their implementation shows a good sorting performance of 70M Strings/sec on a dataset of 1 million strings when ties are few and slows down by $4 - 5\times$ when tie length increases. Though the GPU radix sort primitives are

[1] http://goo.gl/mlwlZ

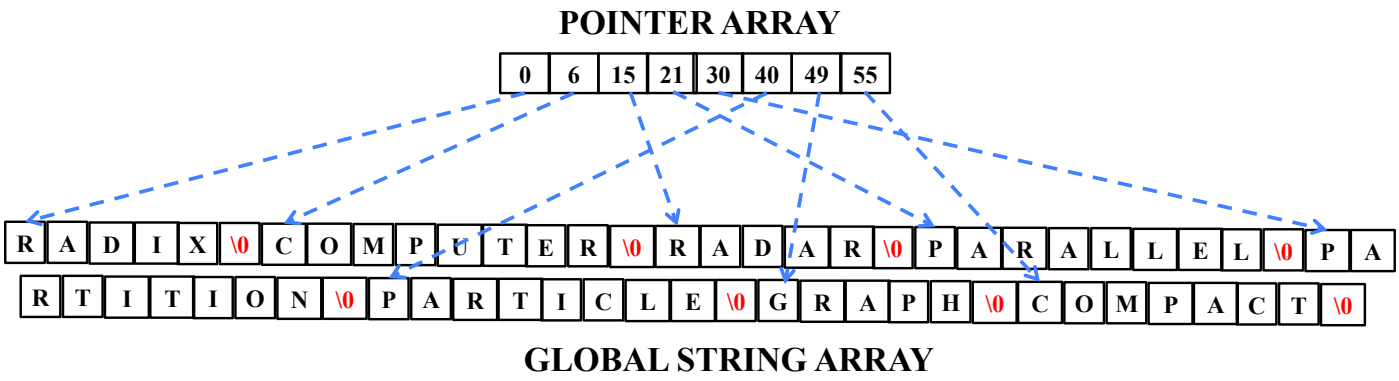

Fig. 1. The strings in the global string are delimited by null characters. The pointer array contains indices of the starting position of each string in the global string array. This pointer array is shuffled during the sort to obtain the sorted order of strings. The example we consider in this paper contains the set of strings : **radix, computer, radar, parallel, partition, particle, graph, compact.** We use this same set of strings to illustrate our sorting procedure.

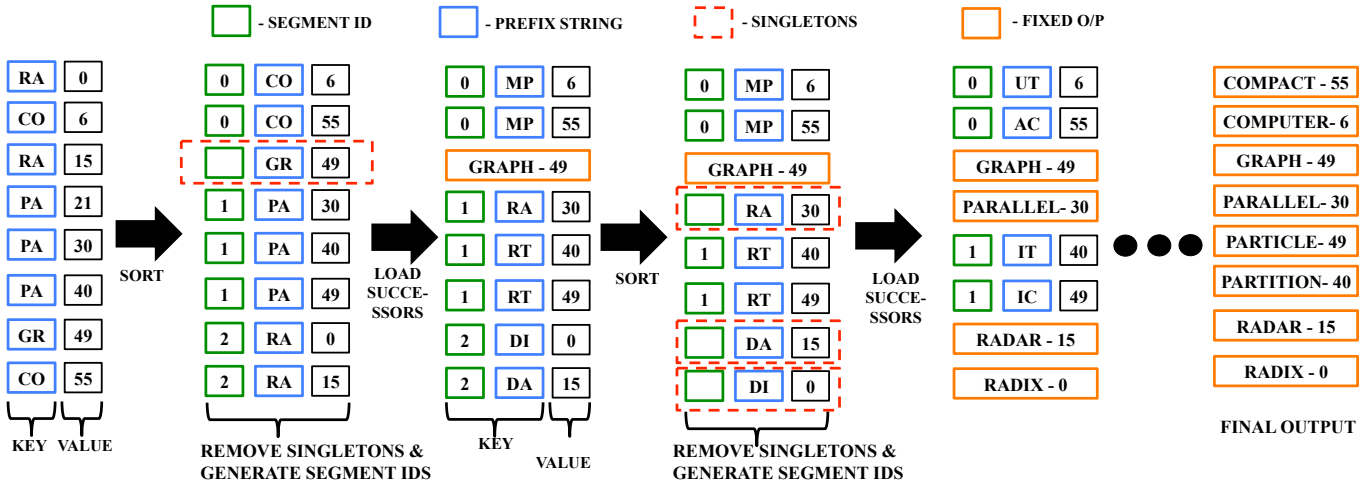

Fig. 2. Illustration of our basic GPU sorting algorithm. In this example, we load two-character prefix strings in each step. The steps of fixed-length sorting, removing singletons, generating segment ids and loading successive prefix strings are performed until we obtain the final output (i.e. all strings are singletons).

relatively faster, their string sort again uses a merge sort. They avoid radix sort because "costs of radix sort algorithm that involves direct manipulation (i.e. shuffling) of keys will scale with key length".

In a recent development, Banerjee et al. use the hybrid CPU+GPU platform and present a faster merge sort algorithm than Davidson et al. [2]. They also benchmark its sorting performance on long and variable length keys [2]. On an average, their merge sort is 20% faster for fixed-length keys and 24% faster for a dataset of random strings as compared to Davidson et al. On GTX 580 GPU, they achieve a throughput of 16 MKeys/s on a dataset of 1M random strings, while our throughput on the same platform is 83 MKeys/s. In our detailed results, for balanced comparison, we only compare to Davidson et al. Since, unlike Banerjee et al. our string sort currently does not exploit the multi-core CPU for additional performance.

We introduce a different category of efficient string sorting algorithms for the GPU, based on fast fixed-length radix sort. Instead of comparing two strings at a time in the traditional iterative manner, we show that we can efficiently compare all strings in a column-wise manner from first to the last character. Our approach uses a way to limit the shuffling to only string indexes and small prefix strings at each step, while performing all operations with fast parallel primitives. Avoiding iterative comparisons allows us to circumvent the problems of thread divergence and high latency global memory accesses faced by Davidson et al. Also, on practical datasets we see that the length of ties do not exceed more than a few hundred characters and results in Section IV-D, IV-C show that, our GPU algorithm gives better performance and scalability as compared to the merge sort based string sorting methods of Davidson et al. and the Thrust Library.

III. Our Algorithm

The input setup stores all strings in the global memory as a single contiguous array, delimited by null characters (Figure 1) and accompanied by an index/pointer array. The pointers to the strings are indices in this global string array. Fixed-length radix sort primitives are the fastest amongst all others. One way to use these primitives for performing string sorting is to load entire strings as keys and perform the radix sort operation. This will recursively create buckets from the first to

Algorithm 1 Our GPU String Sorting

```
1: Input: String Array G, Index Array I
2: Output: Shuffled Index Array O.
3: k = optimal key length                    // 8 for our platform
4: M ← load_prefix(G,k, 0)
5: offset ← k                    // load next prefix starting at offset
6: Seg ← [0, 0, ···, 0]                      // Only one segment
7: segBytes ← compute_bytes(Seg)             // 0 initially
8: K ← pack_keys(M, Seg, segBytes)
9: repeat
10:    radix_sort(key: K, value: I)
11:    F ← mark_singletons(K, O)  // F = Flag, O = Output
12:    // above step also writes index of singletons to output
13:    D ← prefix_scan(F)            // D = Destination Array
14:    K, I ← scatter(K, I, flag: F, dest : D)   // compaction
15:    Seg ← generate_segments(K)
16:    segBytes ← compute_bytes(Seg)
17:    M ← load_prefix(G, k− segBytes, offset)
18:    offset ← offset +k− segBytes
19:    K ← pack_keys(M, Seg, segBytes)
20: until no segments left
21: Output : Shuffled Index Array O
```

last character, but will involve shuffling the entire string keys at each step. Such an extensive data movement will form a huge bottleneck on the GPUs.

We develop an approach that is outlined in Algorithm 1. We exploit the efficiency of radix sort while reducing data movement by sorting records of string bytes as the key and string index as the value (line 10). Each step uses a few bytes of each string starting at a fixed offset from the left as the key; the offset is 0 for the first step (line 4). The fixed-length radix sort primitive from the library is used in each step. Strings with a common prefix so far come in adjacent positions after sorting. Strings with unique prefixes will be singletons and would already be in their final place in the sorted output. These can be marked and removed from further processing (line 11). For the remaining strings, we assign *segment ids* (stored in the Seg array) beginning with 0 for the lexicographically smallest and increment by 1 whenever the next string differs. Common prefix strings are contiguous and get the same *segment id*. Segment assignment is performed using a scan primitive after marking in parallel the locations where adjacent sorted records have different keys. Each segment represents a bucket; strings belonging to it will only shuffle among themselves without crossing segments in the final output. The *segment id*, thus, encodes the history of the sorting steps till the current one, which allows us to discard the currently sorted prefixes and load successive bytes in their place for further sorts (line 17). We do further sorts on records with a key consisting of the *segment id* on the left and next few bytes from each string on the right and a value consisting of the string pointers. The tuple of *segment id* and successive characters forms a proxy for the entire prefix of each string. The pointers in the value, together with the offset are used to load successive characters in every sort step.

Note that the *segment id* cannot exceed the problem size. If there are many common substrings in the dataset, there will be fewer segments. We repeat the process, sorting more of the string and removing singletons until all strings are singletons. Each radix sorting step involves shuffling only a fixed-length record of the key plus the string index as the value. We are able to exploit the high performance of the parallel radix sort for all steps, without resorting to less parallel sorting methods midway. This enables us to scale to larger datasets and also datasets with longer ties better, as shown in Sections IV-C, IV-D,IV-F.

Our approach is illustrated in Figure 2, where we sort the set of strings: *radix, computer, radar, parallel, partition, particle, graph and compact*, organized in memory as shown in Figure 1. In this example, we load 2-character prefix strings before every sort step. After the first sort step we see that the prefix *gr* is a singleton, thus the position of *graph* in output array is fixed as 3. The other prefix strings *co, pa, ra* are assigned incremental segment ids. We then use the value part which consists of pointers of the strings to load successive 2-character prefixes for each string. The next sort is performed on the tuple of segment ids and the newly loaded 2-character prefix. This generates more singletons *ra, da, di* which fixes the positions of the strings *parallel, radar, radix* respectively. The same process repeats for one more iteration after which all strings become singletons and we obtain the sorted order.

A. Singleton Elimination

We write a parallel kernel to perform singleton elimination. First a flag array is created with 0 if a prefix is a singleton – that is, it is different from its predecessor and its successor – and 1 otherwise. An exclusive prefix sum of the flag array gives us the destination indices for each remaining string. We compact the original records with flag of 1 using a scatter primitive and the destination indices. Singleton elimination reduces the problem size for the subsequent iterations and reduces the memory requirements. In practice, singleton elimination improves the string sort performance by $1.1\times$ to $4.5\times$ as shown in Table III. The use of library primitives for scan and scatter helps in adapting to other systems.

B. Optimal Key Length and Adaptive Segment Ids

The radix sort primitives support keys of lengths 1, 2, 4 and 8 bytes efficiently on current Nvidia GPUs. Beyond 8

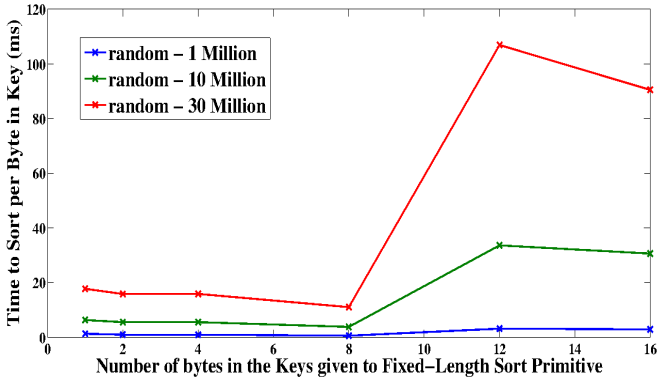

Fig. 3. Performance of Thrust's fixed-length sort primitive with varying key length.

Dataset	Details	Size (MB)	After Sort Tie Length Max.	Avg.
artificial-2	10^6 same strings of A repeated $\approx$ 100 times	98	101	101
artificial-4	10^7 strings of characters a to i	162	80	13
artificial-5	10^6 strings containing A but varying length (1 to 100)	50	100	50
dictcalls	100187 strings of opcodes	2.2	35	15
random	10^6 random strings of length $\approx$ 100	97	5	2
words	$10^7 \approx 10$ million words with no duplicates	118	48	7
url	$10^7 = 10$ million strings containing urls	305	215	29
genome	$31623000 \approx 30$ million strings of a, t, g, c	302	9	8
sentences	≈ 1.8 million sentences from gutenberg project	124	63	17
pc-filelist	≈ 10 million strings containing filepaths	656	180	59

TABLE I. DETAILS OF THE DATASETS USED IN OUR EXPERIMENTS. WE CREATE AND USE SENTENCES AND PC-FILELIST TO PARTICULARLY BENCHMARK OUR CODE ON TYPICAL DATASETS WITH HIGH NUMBER OF TIES. THE OTHER DATASETS ARE STANDARD DATASETS USED IN PREVIOUS STRING SORTING LITERATURE [9], [18].

Dataset	Thrust Comparator Sort			Our GPU String Sort						Speed Up	α
	Mem. Setup	Sort	Total	Mem. Setup	Iterations (k)	Sort (t1)	Scatter/ Scan (t2)	CUDA Kernels (t3)	Total		
artificial-2	122	2652	2744	91	15	35	31	61	219	12.5	3.6
artificial-4	113	5699	5813	106	15	197	44	87	436	13.3	1.6
artificial-5	97	1808	1906	60	17	69	24	20	175	10.8	1.6
dictcalls	81	32	113	50	6	7	3	1	62	1.8	1.5
random	182	75	257	70	1	6	1	5	84	3.0	2
words	105	2016	2122	97	9	97	22	35	252	8.4	1.5
url	155	8559	8714	153	37	416	102	264	936	9.3	1.8
genome	155	14064	14219	275	2	189	92	164	720	19.7	2.3
sentences	131	895	1026	79	11	65	18	36	199	5.1	1.8
pc-filelist	230	9003	9234	225	33	705	164	313	1409	6.5	1.6

TABLE II. COMPARISON OF RUNTIME (IN MILLISECONDS) OF THE THRUST COMPARATOR BASED STRING SORT AND OUR GPU STRING SORT. THE TABLE SHOWS THE SPLIT IN RUNTIMES FOR DIFFERENT STEPS OF OUR STRING SORTING ALGORITHM. THE VALUE $\alpha = (t1 + t2 + t3)/t1$.

bytes, the values are tupled together and compared with 2 or more standard comparisons [3]. Figure 3 shows the normalized sorting time (per byte of the key) on an Nvidia GeForce GTX 580 GPU for different key lengths. Beyond 8-byte keys, there is a sharp increase in this normalized per byte sorting cost. This suggests that the individual sorting steps should use 8-byte keys for maximum performance. This number may be different on other accelerators and on future Nvidia GPUs. Our algorithm should use appropriate key-lengths on each.

We pack the segment ids and prefix strings into the key. We can use 4 bytes to represent the segment id and the rest for subsequent string bytes. This will process the strings slower when the number of segments are small. We adapt to the number of segments and use the minimum number of bytes to represent the segment id at each step. This varies from 0 bytes for the first step (all strings are in the same segment) to 3 bytes in practice in our examples (problem size itself was less than 2^{24} except in one case). The remaining bytes are used to load the prefix strings. This adaptive scheme allows us to compare more number of characters in each step, reducing the number of iterations. On low entropy data the adaptive scheme allows us to have fewer bytes for the segment id, which implies more characters are compared per sort step and more new segments can be identified. For high entropy data, segment id consumes more bytes and relatively lesser number of characters are compared per sort step. This is still reasonable because strings are already highly segmented and only a few more comparisons are needed to resolve ties. The results in Table IV show that we get significant speed up with the adaptive scheme.

IV. RESULTS AND PERFORMANCE ANALYSIS

We use efficient parallel primitives provided by the Thrust Library (v1.6.0) [3]. Our setup has Nvidia GeForce GTX 580 GPU (compute v2.0) and we use CUDA software version 4.0. We also demonstrate our results on Nvidia Tesla K20C (compute v3.5), with the Kepler architecture. For K20C, we use CUDA software version 5.0. We present the performance of our string sorting algorithm on different datasets (Table I) in this section. The runtimes that we measure for all CPU and GPU algorithms are typically in milliseconds and do not include the File I/O time.

A. Thrust performance on varying key size

Figure 3 presents the performance of Thrust's fixed-length radix sort primitive for different key lengths. The key length changes from 1 to 16 bytes. The total time divided by the key length in bytes, to sort 1 to 30 million random keys plus 4-byte value per key record, is shown in the figure. The per byte sorting time reduces as key length varies from 1 byte to 8 bytes. Beyond 8 bytes, Thrust shifts to a slower merge sort algorithm [3]. This results in a sharp increase in the per byte sort time. The optimal per byte sort time is thus obtained for 8 byte keys. We, therefore, fix the key size to 8 bytes for the GPU radix sort operations we perform. Please note that this parameter

Dataset	With Singleton Elimination - Time (ms)	Without Singleton Elimination - Time (ms)	Speed Up
artificial-2	367	357	0.9
artificial-4	601	2237	3.7
artificial-5	299	317	1.1
dictcalls	28	38	1.3
random	23	31	1.3
words	423	1340	3.1
url	1160	5331	4.5
sentences	219	408	1.8
pc-filelist	1723	4791	2.7

TABLE III. COMPARISON OF RUNTIME (IN MILLISECONDS) OF OUR GPU STRING SORTING ALGORITHM WITH AND WITHOUT THE OPTIMIZATION OF SINGLETON REMOVAL. TO DECOUPLE THE OPTIMIZATIONS AND STUDY THEM SEPARATELY WE MAINTAIN A FIXED SEGMENT ID SIZE IN ABOVE EXPERIMENTS. THE RUNTIME IMPROVES WITH SINGLETON REMOVAL.

Dataset	Fixed Size Segment Id		Adaptive Size Segment Id		Speed Up
	# Iterations	Total Time	# Iterations	Total Time	
artificial-2	26	367	15	127	2.8
artificial-4	21	601	15	328	1.8
artificial-5	26	299	17	113	2.6
dictcalls	9	28	6	11	2.5
random	2	23	1	12	1.9
words	13	423	9	154	2.7
url	54	1160	37	782	1.4
genome	3	1488	2	445	3.3
sentences	16	219	11	119	1.8
pc-filelist	46	1723	33	1182	1.4

TABLE IV. COMPARISON OF RUNTIME (IN MILLISECONDS) OF OUR GPU STRING SORTING ALGORITHM WITH AND WITHOUT THE OPTIMIZATION OF ADAPTIVE SIZE FOR SEGMENT BYTES. THIS SHOWS THAT OUR ADAPTIVE SCHEME REDUCES THE NUMBER OF ITERATIONS AND PROVIDES US A SIGNIFICANTLY BETTER RUNTIME.

may vary on other GPUs and certainly on other accelerators. The k parameter in Algorithm 1 can be set to the optimum value when adapting our method to other architectures.

B. Datasets

The details of the datasets used in our experiments are given in Table I. The top 8 (artificial-2 to genome) are standard datasets used in the CPU string sorting literature [9], [18]. They have data of varying size from natural sources such as list of english words, urls, genome sequences as well as synthetic ones such as list of repeated strings of same character (artificial-2), list of repeated strings of same character but varying lengths (artificial-5), random strings (random), words formed from a few characters (artificial-4), etc. Good performance on these datasets indicates the robustness of the sorting algorithm. The GPUs are particularly good to process larger datasets. Hence, we create additional datasets such as `pc-filelist` and `sentences`. The `pc-filelist` was created by listing all ($\approx$ 2 million) files on a typical server class machine starting with the root character "/". This was replicated 5 times with a different prefix for each copy (viz. "node1/" to "node5/") to obtain a dataset of 10 million strings. For the `sentences` dataset, we extract sentences from ≈ 40 e-books of the Gutenberg project. Large after-sort tie length is an indicator of the difficulty of sorting a dataset [6]. These strings have many ties to each other since many files share common base directory path and many sentences have similar beginning. This makes them difficult, yet not artificial, inputs for string sorting. Their large size also allows us to test for the scalability of our approach. These datasets as well as our code will be available for others to use.

C. Comparison to Enhanced Thrust Sort

In Table II, we compare the performance of our radix sort based GPU string sort to Thrust's comparator based string sort described in section II-B. We use the implementation of Thrust comparator based method that is available as part of a suffix sorting code[2]. This implementation is very slow as it does not support bulk copy of the user-defined string class. Copying one string at a time is very slow. We enhanced the code to support bulk transfers so as to facilitate its running on large datasets. Table II shows that we obtain a speed ups ranging from $1.8\times$ to $19.7\times$. We analyze and justify this performance in the following sections.

Sort Time and Total Time: Table II shows the split of the times taken by different steps of the string sort for all the datasets. The memory setup time includes the time for allocating memory on GPU as well as doing all the data transfers between CPU and GPU. We also measure the time taken by the sort primitive ($t1$), scatter/scan primitives ($t2$) and the CUDA kernels that perform the remaining operations ($t3$). We see from the results that sorting is the most expensive step on all large datasets. The only exception is artificial-2 dataset, which sorts arrays of equal values each time. Thrust handles this specially and provides very fast sorting [14]. Typically, if we exclude the memory setup time, we see that the total time varies in a small band of 1.5 to 2.3 times the time for the radix sort. We denote this by a factor α, shown in the last column of Table II. In practice, as of today, we can empirically expect α to be bound by ≈ 2.5 for our algorithm.

Expected vs. Achieved Time: Given that the basic sorting primitive takes on average time t per iteration, the total time for an iteration can be estimated to be $\alpha \times t$. With k iterations, then estimated total time is $\alpha \times t \times k$. If the throughput of fixed-length radix sort primitive is given in p MKeys/s and if the string sorting problem has size N million strings, then the estimated time per iteration is $1000/p \times N$ milliseconds. Thus, the expected total time for string sort in milliseconds is

$$T_{exp}(ms) = \alpha \times 1000/p \times N \times k \approx t1 + t2 + t3. \quad (1)$$

Fixed-length radix sort primitive of thrust offers a sorting performance of 1GKeys/s = 1000 MKeys/s [14]. But in practice, on our architecture, we achieve a throughput of 500 to 760 MKeys/s from the radix sort primitive, thus we fix $p = 750$ in equation 1 and calculate the expected runtime for $\alpha = 2.5$. The results in Table V show that we achieve better runtime than the expected time on artificial-4, words, pc-filelist and url datasets (each of which has 10 million strings and also long ties). Such an analysis, shows that we can predict the performance of the string sorting algorithm based on a fixed-length sorting primitive. This prediction needs parameters such as input size (N), performance of the fixed-length sort primitive (p) and max ties in the input (i.e. some indicator of the value of k). Our results in Table V, show that our achieved performance is within practical limits of the expected performance in many

[2]https://github.com/bzip2-cuda/bzip2-cuda/tree/master/lib

Dataset	Achieved Runtime $(t1 + t2 + t3)$	Expected Runtime $(\alpha \times 1000/p \times N \times k)$
artificial-2	127	50
artificial-4	328	500
artificial-5	113	57
dictcalls	11	2
random	12	4
words	154	300
url	782	1233
genome	445	200
sentences	119	66
pc-filelist	1182	1100

TABLE V. ACHIEVED VS. EXPECTED RUNTIME (IN MILLISECONDS) FOR OUR RADIX SORT BASED GPU STRING SORTING ALGORITHM.

Dataset	GTX 580 *(Fermi)*	K20C *(Kepler)*	Speed Up
artificial-4	436	472	0.92
url	936	899	1.04
genome	720	642	1.21
pc-filelist	1409	1121	1.25
pc-filelist $\times 2$	-	2574	-

TABLE VII. COMPARISON OF RUNTIME (IN MILLISECONDS) ON THE KEPLER K20C AND GTX 580 GPU. PC-FILELIST DATASET IS REPLICATED TWICE ($\approx 20 Mstrings$) TO CREATE PC-FILELIST $\times 2$. K20C CAN PROCESS THIS LARGE INPUT BECAUSE OF ITS HIGH GLOBAL MEMORY, WHICH IS NOT POSSIBLE ON GTX 580.

cases and it justifies the speed up that we obtain over other methods.

Singleton Elimination and Adaptive Segment ID: Removal of singletons progressively reduces the sorting problem size. This allows the fixed-length sorting primitive to perform better. In equation 1, this translates to reducing the impact of N in successive iterations. Table III shows that we achieve a speed up between 1.1 to 4.5 for different datasets after eliminating singletons. There is no speed up for the artificial-2 dataset, since it consists of equal strings and no singletons are found. For url, artificial-4 and pc-filelist datasets, which have large number of strings and long ties we see that our singleton elimination technique performs particularly well giving us a speed up of 4.5, 3.7 and 2.7 respectively. Using the minimum number of bytes for the segment id field enables us to reduce the total number of iterations as more number of characters are compared per iteration. This reduces k in equation 1. Table IV shows that we achieve a further speed up of $1.4\times$ to $3.3\times$ by using the adaptive scheme for segment id bytes. Both, these optimizations aim at reducing the expected runtime for our GPU algorithm and also yield good results in practice.

Thrust Comparator based String Sort: The enhanced Thrust sort for strings using a custom comparator performs poorly (Table II). This is due to the use of the slower merge sort as well as the loading successive characters from the high-latency global memory to resolve ties. These accesses need to be performed repeatedly per string in *every* merge step. Our method loads the string from start till the ties are resolved only once per sort step. Our approach makes full use of the fact that the fixed-length radix sort primitive ensures limited global memory accesses [14]. Reducing the high latency global memory accesses and exploiting fast primitives of radix sort and scan, help us achieve much better performance than the comparator based string sort of Thrust, Table II shows that we achieve a speed up of 5 to 10 on large datasets.

Time spent on Thrust Primitives: Table VI presents the fraction of total execution time that is spent using Thrust primitives of sort, scan and scatter. This measure is an indicator of the adaptability of our methods to newer implementations and architectures. For large datasets with long ties (i.e., url, aritificial-4, sentences, pc-filelist) we see that greater than 65% of the time is used by Thrust primitives. On an average the Thrust primitives use 70% of the execution time. These results show that our algorithm without any redesign can benefit well from any future improvements to these primitives or the basic architecture.

D. Comparison to Davidson et al. [6]

We compare with the method by Davidson et al. using the CUDPP code from them[3] [6]. This code is still under development and lacks a few optimizations mentioned in their paper. It presently does not scale to the input size that we use for most datasets. On the wiki-words dataset they used, we obtained a runtime of 20 ms, while they reported about 14 ms in their paper [6]. This is an easy dataset, with very few ties (average ≈ 5) and only 1 million elements. Hence, the small difference in runtime is not very significant. Scalability to large datasets is a particular strength of our approach since it is totally based on the radix sort primitive, as opposed to the slower merge sort they use. On 1 million random

[3]http://code.google.com/p/cudpp/source/checkout

Dataset	% Time used by Thrust Primitives $(t1 + t2)/(t1 + t2 + t3) \times 100$
artificial-2	51
artificial-4	73
artificial-5	82
dictcalls	90
random	58
words	77
url	66
genome	63
sentences	69
pc-filelist	73

TABLE VI. THE % OF TOTAL TIME (W/O MEMORY SETUP) USED FOR EXECUTION BY THE THRUST SORT, SCATTER AND SCAN PRIMITIVES. ON AN AVERAGE, 70% OF THE TOTAL TIME IS UTILIZED BY THRUST PRIMITIVES.

Dataset	Our GPU String Sort	CPU String Sorts	
		Burstsort	MSD Radix Sort
artificial-2	219	880	2496
artificial-4	436	4000	4468
artificial-5	175	730	948
dictcalls	62	50	44
random	84	166	188
words	252	3700	2968
url	936	9270	8832
genome	720	8210	9888
sentences	199	-	968
pc-filelist	1409	24920	13220

TABLE VIII. COMPARISON OF RUNTIME (IN MILLISECONDS) OF OUR GPU STRING SORTING ALGORITHM WITH STATE-OF-THE-ART CPU ALGORITHMS FOR STRING SORT. FOR THESE EXPERIMENTS, WE USE THE NVIDIA GTX 580 GPU AND INTEL CORE2DUO E7500 CPU.

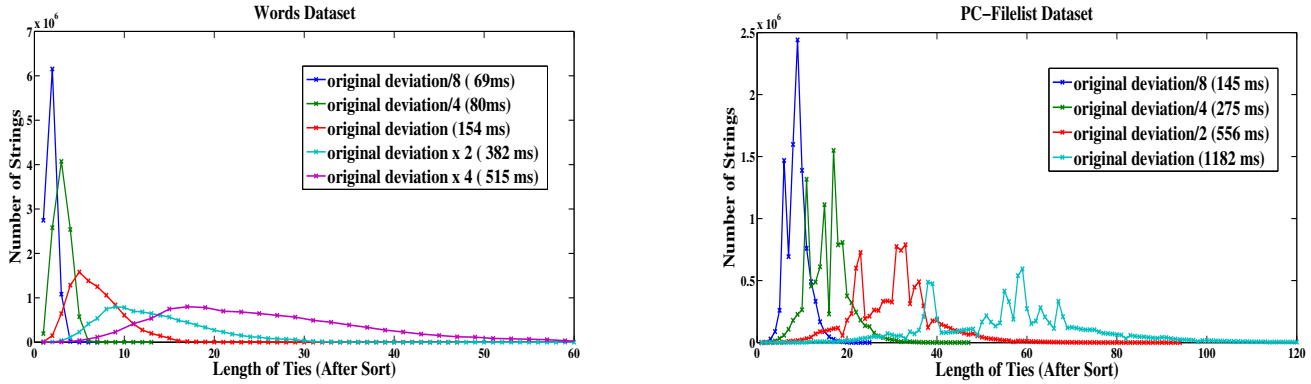

Fig. 4. In this figure, we vary the deviation of the sorted ties histogram for two datasets: words (less ties) and pc-filelist (high ties). To reduce the deviation by a factor of k, we take every k^{th} character of each string. To increase deviation by a factor of k, each character of the string is replicated k times. The runtimes for our GPU algorithm are indicated in the legend. Datasets having histograms with low deviation are easier to sort than those with high deviation. The runtime varies linearly with the change in deviation, indicating we can even handle inputs with high ties as efficiently as possible.

strings and a larger words dataset with 10 million strings, their implementation takes 27 ms and 2100 ms respectively. Our string sort takes only 12 ms and 154 ms respectively on these datasets, we get a speed up of 2 to 13×. The comparison will be more meaningful only when a fully optimized and stable version of their code is available.

E. Performance on the Kepler GPU

Table VII compares the performance of our string sort on Nvidia K20C and Nvidia GTX 580 GPU. K20C is of a different architecture family (kepler) and has a relatively slower clock speed of 706 MHz as compared to 772 MHz of GTX 580 (fermi). But, it has a global memory of $\approx 5GB$ and can process much larger inputs than GTX 580 ($\approx 2GB$ global memory). It also has 2496 cores, while the GTX 580 has only 512. In our experiments on sorting random integer data with the Thrust sort primitive, we see that the K20C performs only marginally better as compared to GTX 580. Thus, using the K20C we achieve a speedup of 1.21 and 1.25 on genome and pc-filelist datasets respectively. On an even larger dataset, pc-filelist ×2, created by concatenating the pc-filelist dataset twice, K20C gives a runtime of 2.57s. The same dataset takes about 38.1s and 27.8s on the CPU using Burstsort and MSD radix sort respectively (i.e. speed up of 14 and 10). Also, the global memory limit in GTX 580 GPU is prohibitive for processing this large input. Thus, the new K20C GPU gives slightly better performance and is scalable to much larger inputs as compared to GTX 580. Note, since our code is primarily primitive based, no tuning of the thread/block grid parameters is required to achieve speed up on Kepler. Also, any future improvements to the primitives that result from new features viz. dynamic parallelism, hyperQ etc. of the Kepler architecture will be directly inherited by our string sort.

F. Comparison with CPU Algorithms

Table VIII compares the performance of our GPU string sorting algorithm with the state of the art CPU algorithms for string sorting: Burstsort [17], [18] and MSD radix sort [9]. For Burstsort, we use the code provided by the authors of [17]. For MSD radix sort we use the efficient code available as part of a standard string sorting library[4]. The speedup is very significant except on the very simple dictcalls dataset. This gives a feel of the speed up that can be expected using a GPU for those who use string sorting on the CPU.

G. Performance with After-Sort Tie Length

The sorted ties histogram quantifies the difficulty of sorting a given dataset [6], [9]. For each string, the average of number of prefix characters that match to the strings just before and after it in the sorted order is called its tie-length. This measure is an upper bound on the number of prefix characters that can match to the given string from the entire dataset. The histogram of these tie-lengths is called the sorted ties histogram. If the sorted ties histogram is short (i.e. deviation is small) and steep, many strings have small tie lengths. Such an input is easy to sort. On the other hand, if it has a large spread (i.e. deviation is large), there are many strings with long ties and the dataset is difficult to sort.

The words dataset has strings that are smaller in length. The average tie length is 7 characters. To study the performance of our algorithm with respect to average tie-length, we modify the sorted ties histogram of this dataset as follows. We increase the deviation by a factor of k by replicating every character of the given string k times. Similarly, we reduce the deviation by a factor of k by taking every k^{th} character of each string. Note, area under the curve remains the same for all these modified histograms. From Figure 4, we see that when deviation changes by a factor of $1/8$, $1/4$, 2 and 4, the runtime changes by $0.4\times$, $0.5\times$, $2.4\times$ and $3.3\times$ respectively. Our runtime demonstrates a near-linear scaling with deviation. On reducing the deviation below $1/4$, the tie length becomes so low that only 1-2 sort operations are required. Similarly for the pc-filelist dataset, we change deviation by factors of $1/2$, $1/4$ and $1/8$ and obtain runtimes that are $0.47\times$, $0.23\times$ and $0.12\times$ the original runtime. The pc-filelist dataset was designed as a stress test for our algorithm, with a large number of strings (10 million) and very long ties. A near-linear performance on different spreads of histograms for this dataset shows that our algorithm can handle the entire spectrum of sorted ties histogram efficiently.

[4]https://github.com/rantala/string-sorting

The pc-filelist dataset is such that we use up all memory available on the GPU to perform the sort. Increasing the deviation will involve replicating the characters and will require more memory. To handle such cases, we can modify our approach to stream successive prefixes of all strings to the GPU memory periodically to substitute the used prefixes. This can be explored in our future work and will allow our algorithm to scale to even larger input size. It is not as straightforward to scale merge sort algorithms, because they will require entire strings to be in memory to perform iterative comparisons at least for the final merge step.

V. Conclusions

We presented an efficient string sort method for the GPUs, based on the fast radix sort primitive in the paper. We got a speed up of more than 10 over the best GPU string sorting approaches. Our approach scales to larger datasets easily. Reducing memory movement, removing singleton segments early, and consuming maximum number of string bytes in each sort-step are the key factors behind our high performance. Most of the time was spent on optimized primitives from available libraries. We presented results on a variety of natural and synthetic datasets. The analytical expected runtime matched the actual time quite closely.

We would like to explore the scalability of our method to extreme conditions in the future, when the set of strings to be sorted does not fit the GPU memory. Our method can handle this by streaming parts of the dataset from the CPU as and when needed, as we need to access the string strictly from the left to the right only. Strings can thus be divided into sections by columns and streamed to the GPU. The streaming can overlap with the computations to get very high throughput in string sort. We are exploring this currently.

References

[1] D. A. Alcantara, A. Sharf, F. Abbasinejad, S. Sengupta, M. Mitzenmacher, J. D. Owens, and N. Amenta. Real-time parallel hashing on the gpu. *ACM Trans. Graph.*, 28(5):154:1–154:9, Dec. 2009.

[2] D. S. Banerjee, P. Sakurikar, and K. Kothapalli. Fast, scalable parallel comparison sort on hybrid multicore architectures. In *Workshop On Accelerators for Hybrid Exascale Systems (AsHES), IPDPS'13*, 2013.

[3] N. Bell and J. Hoberock. Thrust: A productivity-oriented library for cuda. *GPU Computing Gems Jade Edition*, page 359, 2011.

[4] J. L. Bentley and R. Sedgewick. Fast algorithms for sorting and searching strings. In *Proceedings of the eighth annual ACM-SIAM symposium on Discrete algorithms*, SODA '97, pages 360–369, 1997.

[5] D. Cederman and P. Tsigas. Gpu-quicksort: A practical quicksort algorithm for graphics processors. *J. Exp. Algorithmics*, 14:4:1.4–4:1.24, Jan. 2010.

[6] A. Davidson, D. Tarjan, M. Garland, and J. D. Owens. Efficient parallel merge sort for fixed and variable length keys. In *Innovative Parallel Computing*, page 9, May 2012.

[7] K. Garanzha and C. Loop. Fast ray sorting and breadth-first packet traversal for gpu ray tracing. In *Computer Graphics Forum*, volume 29, pages 289–298. Wiley Online Library, 2010.

[8] B. He, N. K. Govindaraju, Q. Luo, and B. Smith. Efficient gather and scatter operations on graphics processors. In *Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*, 2007.

[9] J. Kärkkäinen and T. Rantala. Engineering radix sort for strings. In *Proceedings of the 15th International Symposium on String Processing and Information Retrieval*, SPIRE '08, pages 3–14, 2009.

[10] C. Lauterbach, M. Garland, S. Sengupta, D. Luebke, and D. Manocha. Fast bvh construction on gpus. In *In Proc. Eurographics 09*, 2009.

[11] N. Leischner, V. Osipov, and P. Sanders. Gpu sample sort. In *2010 IEEE International Symposium on Parallel Distributed Processing (IPDPS)*, pages 1–10, 2010.

[12] P. M. McIlroy, K. Bostic, and M. D. McIlroy. Engineering radix sort. *COMPUTING SYSTEMS*, 6:5–27, 1993.

[13] D. Merrill, M. Garland, and A. Grimshaw. Scalable gpu graph traversal. *SIGPLAN Not.*, 47(8):117–128, Feb. 2012.

[14] D. Merrill and A. Grimshaw. High performance and scalable radix sorting: A case study of implementing dynamic parallelism for GPU computing. *Parallel Processing Letters*, 21(02):245–272, 2011.

[15] R. A. Patel, Y. Zhang, J. Mak, and J. D. Owens. Parallel lossless data compression on the GPU. In *Proceedings of Innovative Parallel Computing (InPar'12)*, May 2012.

[16] N. Satish, M. Harris, and M. Garland. Designing efficient sorting algorithms for manycore gpus. In *Parallel Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1–10, 2009.

[17] R. Sinha and A. Wirth. Engineering burstsort: Towards fast in-place string sorting. In C. McGeoch, editor, *Experimental Algorithms*, volume 5038 of *Lecture Notes in Computer Science*, pages 14–27. 2008.

[18] R. Sinha, J. Zobel, and D. Ring. Cache-efficient string sorting using copying. *J. Exp. Algorithmics*, 11, Feb. 2007.

[19] V. Vineet, P. Harish, S. Patidar, and P. J. Narayanan. Fast minimum spanning tree for large graphs on the gpu. In *Proceedings of the Conference on High Performance Graphics 2009*, HPG '09, pages 167–171, New York, NY, USA, 2009. ACM.