Fast In-Memory Suffix Sorting on the GPU

ABSTRACT

We design and implement a parallel suffix sorting algorithm on the GPU (Graphics Processing Unit) based on prefix-doubling. Speficically, we first sort the suffixes by their initial characters, and then divide the partially sorted suffix groups by their lengths and further sort each type of groups. The number of initial characters, the dividing criteria, and the type-dependent sorting strategies are optimized to achieve the best overall performance.

We have evaluated our implementation in comparison with the fastest CPU-based suffix sorting implementation as well as the latest GPU-based suffix sorting implementation. On a server with an NVIDIA M2090 GPU and two Intel Xeon E5-2650 CPUs, our implementation achieves a throughput of up to 52MB per second, and a speedup of up to 6.6x and 2.5x over the CPU-based and GPU-based suffix sorting implementation, respectively.

Keywords

Suffix Array, Prefix Doubling, Parallel Sorting, GPGPU

1. INTRODUCTION

A suffix array (SA) represents all suffixes of a string in the lexicographical order [23]. Compared with suffix trees, suffix arrays are more space efficient. Moreover, with an additional data structure – Longest Commom Prefix(LCP) array, most operations on a suffix tree can be implemented on a suffix array. As a result, suffix arrays have been widely used in data compression, such as the Burrows-Wheeler Transform (BWT) [4], text indexing and searching, e.g., FM-index [6], and bioinformatics [22].

Constructing the suffix array for a string involves generating all suffixes of the original string and sorting them by the lexicographical order. This suffix sorting problem has been studied intensively over the past two decades, and many suffix sorting algorithms have been proposed. Existing suffix sorting algorithms roughly fall within three classes: Prefix Doubling [23, 21], Recursive [13, 14, 18, 19, 16, 15, 17], and Induced Copying algorithms [12, 36, 24, 25, 35]. Even though these algorithms sort suffixes using different strategies, almost all of them exploit the fact that each suffix is a sub-string of a set of other suffixes.

Search using an SA is efficient; however, constructing the SA, or suffix sorting, is time consuming. In recent years several parallel and distributed suffix sorting algorithms have been proposed, namely on multicores [7, 11, 29], on distributed clusters [20, 26] as well as on manycore GPUs [5, 31, 37].

In this paper, we first compare different kinds of serial algorithms for suffix sorting and analyze their suitability for parallelization on the GPU. We then study the state-of-the-art GPU-based parallel suffix sorting algorithms and propose our own. Our GPU-based algorithm is based on the most efficient prefix doubling algorithm by Larsson and Sadakane [21] (referred to as the LS algorithm).

The biggest challenge for parallelizing the LS algorithm is that, after sorting the suffixes according to their initial h_0 characters, we need to sort a large number of *unsorted group*s of suffixes with group size (number of suffixes in the group) varied significantly [5, 31].

To address this challenge, we develop a hybrid and efficient segmented sorting approach. The main idea is the following: in each prefix-doubling sorting step, we first refine the groups with the same initial h_0 symbols to extract the *unsorted group*s (i.e., groups with sizes greater than one); next we divide the *unsorted groups* into multiple types by their group sizes, and we sort each type of unsorted groups with type-specific strategies.

We tune our GPU-LS algorithm by experimentally selecting the suitable h_0, the group size, and type-specific sort strategies. We further implement the state-of-the-art GPU-based suffix sorting algorithm, denoted GPU-skew [5] for comparison. We evaluate our GPU-LS in comparison with the fastest CPU-based, multi-threaded suffix sorting algorithm, *libdivsufsort* [2], as well as the most recent GPU-based suffix sorting implementation – GPU-skew [5]. The results show that our algorithm achieves a speedup up to 6.6 times over *libdivsufsort*, and up to 2.5 times over GPU-skew.

In summary, we make the following contributions.

1. We identify the technical challenges in performing parallel suffix sorting on the GPUs, and analyze how these algorithms could be improved.
2. We reduce the suffix sorting problem to the problem of sorting a large number of small, non-consecutive segments (*groups*) in a large array, and propose a GPU-based hybrid sorting algorithm for these non-consecutive segments. This problem is known to be tough for the SIMD (Single-Instruction Multiple-Data) GPU architecture. Furthermore, our algorithm is designed elaborately for the suffix sorting scenario, in which the sizes (numbers of suffixes) of unsorted segments drop dramatically in each round of sorting. To the best of our knowledge, this work is the first attempt to sort suffixes using segmented sorting strategies on the GPU.
3. We experimentally evaluate our algorithm in comparison with both the fastest CPU-based and GPU-based suffix sorting implementations on representative benchmark datasets and analyze the performance results.

The remainder of this paper is organized as follows. In section 2, we introduce the background of suffix array and work on GPGPU. In section 3, we describe the sequential and parallel suffix sorting algorithms, and identify the technical challenges of parallel suffix sorting on the GPUs. We present our solutions of parallel suffix sorting in Section 4. We evaluate and discuss the performance of our algorithm in Section 5, and make a conclusion in Section 6.

2. BACKGROUND

In this section, we introduce the background of suffix array (SA) and related work on General-Purpose computing on the GPU (GPGPU).

2.1 Suffix Array

The input of the suffix sorting algorithm is a string. Consider a string $T = t_0t_1t_2...t_{n-1}$ of length $n \geq 1$ defined on an alphabet Σ. We add a lexicographically smallest character, denoted as \$, to the end of the string as t_n. $T_i = t_it_{i+1}...t_{n-1}\$$ represents a suffix of T whose start position is i. The **suffix array** (SA) of T is an integer array of size n that contains the permutation of $[0, n-1]$. $SA[i] = k$ means that T_k is the lexicographically ith smallest suffix among all the suffixes of T.

Many suffix sorting algorithms make use of another array, referred to as the **inverse suffix array** (ISA). The ISA stores the rank value for each suffix, so that $ISA[i] = k$ means T_i is the lexicographically kth smallest suffix among all the suffixes of T. After the suffix sorting is finished, the suffix array SA and inverted suffix array ISA can be derived one from the other in $O(n)$ time. That is, after each suffix gets a unique rank value, we have $SA[i] = k \Leftrightarrow ISA[k] = i$.

The suffixes of T is in **h-order** when they are sorted by the initial h charters of each suffix [23, 21], represented as SA_h and its corresponding inverse suffix array represented as ISA_h. An **h-group** is a maximal sequence of adjacent suffixes in SA_h that share the same initial h characters [32], and the length (size) of an h-group is the number of suffixes it contains. All the suffixes within the same h-group have the same **h-rank**. An h-group of length one (containing a single suffix) is a **sorted group**, and an h-group with length greater than one is an **unsorted group**.

2.2 GPGPU

GPUs have been used as parallel computing platforms or hardware accelerators for a wide range of applications. Related to this work, there has been earlier work on sorting [8, 34, 28, 27, 33], segmented sorting [3]. These sorting algorithms are for sorting a general array as opposed to an array of suffixes of the same string.

Currently major GPU vendors include NVIDIA, AMD, and Intel. NVIDIA's CUDA is one of the earliest programming frameworks for GPGPU applications. OpenCL is a later standard for GPU programming across different GPUs. However, the OpenCL implementations are still vendor-specific, and are actively evolving. Additionally, the performance of OpenCL programs are less optimized than that of programs implemented in a framework native to the GPU hardware. Due to performance considerations, we use NVIDIA's CUDA for GPU programming on an NVIDIA GPU in our work.

In CUDA, developers write GPU programs, called kernels, to process data elements in arrays. Each kernel program is executed by a number of *threadblocks* in parallel. Developers can specify the number of thread blocks and the number of threads per block; however, thread scheduling is done by the runtime system in the unit of 32-thread *warps*. Threads in a block can share a piece of small (usually at tens of kilobytes) but fast, on-chip memory, called the shared memory. In contrast to the fast, small shared memory, the GPU device memory is of several gigabytes and has a high bandwidth, but the latency is also high. A useful feature of the GPU memory is coalesced access, which happens when threads in a warp access consecutive memory addresses and these accesses are grouped into a single memory transaction.

In our work, we employ the following strategies for performance: (1) data-parallel programming, i.e., utilizing data-parallel primitives such as sort, scan, scatter, filter, and so on, to fully exploit the massive thread parallelism in the GPU; (2) use of the shared memory for low latency; (3) use of coalesced access for memory bandwidth utilization.

3. SUFFIX SORTING ALGORITHMS

3.1 Sequential Algorithms

According to the survey by Puglisi [32], there are three major classes of sequential suffix sorting algorithms, namely Prefix-Doubling Algorithms, Induce Copying Algorithms and Recursive Algorithms. We briefly introduce each class of algorithms, and analyze their characteristics and suitability for parallelization on the GPU.

Prefix Doubling. The original Prefix-Doubling suffix sorting algorithm (referred to as MM algorithm) [23] by Manber and Myers works as follows: (1) sort the suffixes according to their first characters and get SA_1 and ISA_1, and (2) sort the suffixes in multiple passes. In pass $j (j \geq 0)$ sorting,

each suffix $SA_h[i], (h = 2^j)$ is sorted with $ISA_h[SA_h[i]]$ as the primary key, and $ISA_h[SA_h[i]+h]$ as the secondary key to get SA_{2h}. The maximum number of passes required is $\log(n)$ as in each pass h doubles. Both step (1) and (2) can be parallelized with a linear time key-value sorting algorithm, such as radix sort. However, the MM algorithm itself is inefficient in practice due to the global sorting of the suffix array in each pass, as many of the suffixes have already been in sorted groups from a previous pass.

An improved Prefix-Doubling algorithm (referred to as LS algorithm) was proposed by Larsson and Sadakane [21]. The first step of the LS algorithm is almost the same as that of the MM algorithm, whereas in the second step, the LS algorithm sorts each unsorted h-group separately instead of sorting the global SA_h in each pass j. Because all the suffixes in the same h-group shares the same rank, each suffix $SA_h[i]$ of them can be sorted simply using $ISA_h[SA_h[i]+h]$ as the primary key. By breaking the large array sort into a large number of small array sorts, the LS algorithm achieved a speedup up to an order of magnitude over the MM algorithm [21], even though both have the same worse-case time complexity of $O(n \log n)$.

At a first glance, the LS algorithm is not so easily parallelizable on the GPU as the MM algorithm, since it is not easy to handle a large number of small, unsorted groups efficiently on the GPU, especially the number of unsorted groups may increase as the sorting passes progress. However, we find that this challenge can be overcome by organizing these small unsorted groups into different types and sorting each type of groups using a suitable method, i.e. radix sort or merge sort. In the experiments section, we demonstrate that the overhead of such organization is worthwhile as it reduces redundant sorting significantly.

Recursive Algorithms. Recursive suffix sorting algorithms work as follows. Consider the original string T as $T^{(0)}$. In each recursive step $i(i \geq 0)$, $T^{(i)}$ is split into two disjoint strings $T^{(i+1)}$ and $T^{'(i+1)}$. After the suffix array for $T^{(i+1)}$ (or $SA^{(i+1)}$) is recursively computed, $SA^{(i+1)}$ is used to induce the suffix array of $T^{'(i+1)}$ (or $SA^{'(i+1)}$). Finally $SA^{(i+1)}$ and $SA^{'(i+1)}$ are merged to get $SA^{(i)}$.

The skew algorithm (also called the DC3 algorithm) is a typical recursive algorithm [13, 14]. In this approach, string $T^{(i+1)}$ is formed from $T^{(i)}$ with characters at position i, $(i \equiv 1 \bmod 3)$ or $(i \equiv 2 \bmod 3)$, whereas string $T^{'(i+1)}$ is formed from $T^{(i)}$ with characters at position $(i \equiv 0 \bmod 3)$. In each recursion, the problem size is reduced to 2/3 of the problem from the previous recursion. Once $SA^{(i+1)}$ is sorted, both deriving $SA^{'(i+1)}$ from $SA^{(i+1)}$ and merging $SA^{(i+1)}$ with $SA^{'(i+1)}$ to get $SA^{(i)}$ take linear time. The overall computation to get $SA^{(0)}$ takes $O(n)$ time. In addition to the skew algorithm, there are several other variants of recursive algorithms [18, 19, 16, 15, 17]. They all follow the same basic idea, except that they split $T^{(i)}$ into $T^{(i+1)}$ and $T^{'(i+1)}$ and merge $SA^{(i+1)}$ with $SA^{'(i+1)}$ using different strategies.

Among all the recursive algorithms, the skew algorithm is the most straightforward one to be parallelized on the GPU since the data for each round of recursion are taken at regular, fixed positions, and they can be sorted using an efficient GPU-based radix sort.

Induced Copying. The first induced copying algorithm was proposed by Itoh and Tanaka [12]. Their algorithm works in the following steps. (1) The suffixes of an input string T are split into two types: type A or type B. T_i is a type A suffix if $t_i > t_{i+1}$; otherwise it is a type B suffix. Each type B suffix is put into its 1-*group* according to its first character. (2) The type B suffixes in each 1-*group* are sorted using certain string sorting algorithms. This step produces the final order for type B suffixes. (3) After type B suffixes are sorted and in their final positions, the suffix array is scanned multiple times to put each type A suffix into its final position.

There are several other variants of induced copying algorithms [36, 24, 25, 35]. All these algorithms have an $O(n^2 \log(n))$ worst case time complexity, since in the second step each suffix is sorted as an independent string. However, they are usually fast in practice. The fastest CPU suffix sorting library, *libdivsursort* [2] is an implementation mainly based on induced copying.

The first step of induced copying can parallelized easily on the GPU, whereas in the second step, parallelizing a common string sorting process is very difficult to achieve a good performance on the GPU due to the irregular sizes of strings. In the last and third step, as the final position of a type A suffix in the suffix array depends on the results from the previous scan pass, this multi-pass scanning is sequential in nature and difficult to be parallelized.

3.2 Distributed and Parallel Algorithms

In recent years several distributed and parallel suffix sorting algorithms have been proposed.

As the earliest parallel work, Futamura et al. [7] proposed a very simple parallel suffix sorting algorithm for multicore CPUs. Their algorithm does not belong to any of the above three categories. In their algorithm, the suffixes are first sorted according to the initial few characters. This step is similar in many of the suffix sorting algorithms. Next, they sort the suffixes in each bucket in parallel using common string sorting algorithms. Their implementation shows a good speedup and scalability on an IBM-SP2 machine in comparison with the sequential string sorting algorithm. However, it is unclear how their performance is in comparison with a sequential suffix sorting algorithm.

Menon et al. [26] present a suffix sorting algorithm based on the MapReduce distributed programming model. Their algorithm is similar to Futamura's work: in the *Map* stage they partition the entire suffix array into multiple disjoint ranges, and each range is independently sorted in the *Reduce* stage.

Parallel Prefix-Doubling Algorithms. Sun and Ma [37] implemented the original MM algorithm [23] using CUDA. Their implementation achieved a 10x speedup compared with the sequential MM algorithm. However, the MM algorithm is usually 10x slower than the LS algorithm [32,

31].

Osipov [31] improved the parallel MM algorithm on the GPU. They proposed to compact the suffix array after each sorting step to remove some singleton (sorted) groups for further sorting steps. Through such filtering, the number of suffixes to be sorted decreases after each iteration. However, the remaining (unsorted) suffixes are still sorted globally rather than sorted separately within each group.

Parallel Recursive Algorithms. Kulla and Sanders [20] proposed a distributed implementation of the skew algorithm on a cluster using MPI. They sorted a string of complete Human Genome(total 2.928 GBytes) in 181 seconds on an Itanium cluster with 64x2 cores. Deo and Keely [5] implemented a parallel skew algorithm on AMD GPUs and APUs using OpenCL. Their implementation achieved a speedup up to 34x over a single-thread CPU-based skew algorithm. Due to the unavailability of the source code of this implementation and the differences in the GPU platforms and languages, we implemented this GPU-skew suffix sorting algorithm on the NVIDIA GPU using CUDA and compared with our GPU-LS algorithm.

Parallel Induced Copying Algorithms. In the induced copying category, so far there are only parallel implementations on the multicore CPUs, as incuded copying algorithms are hard to parallelize on GPU platforms. Specifically, the work by Homann and Fleer [11] is a parallel implementation of the Deep-Shallow suffix sorting algorithm [24] with a speedup of around 2x. Mohamed and Abouelhoda [29] implemented a parallel Bucket-Pointer Refinement(BPR) algorithm [35], and their implementation achieved speedups of no more than 2x than the sequential BPR algorithm.

4. PARALLEL SUFFIX SORTING

We describe our GPU-based parallel suffix sorting algorithm GPU-LS in this section.

4.1 Overview

Due to the SIMD architecture, GPUs are advantageous in processing massive number of elements in lock-steps. As a result, the MM algorithm and the skew algorithm can be easily parallelized on the GPU, whereas algorithms such as Deep-Shallow and Bucket-Pointer Refinement are hard to parallelize on the GPU even though they have a better performance on multicore CPUs than MM and Skew. In our algorithm design, we consider both the efficiency of the baseline sequential algorithm as well as the amount of data parallelism in the algorithm so as to take advantage of the GPU architecture.

We choose to parallelize the LS algorithm on the GPU as it avoids extensively resorting of groups that are already sorted and it is ranked one of the very efficient suffix sorting algorithms (even though it is not the fastest in practice).

In our GPU-LS, the problem of suffix sorting is reduced to the following segmented sorting problem.

Segmented Sorting Problem Given an array A of size N, a head array H and a length array L, both H and L are of length n, for all $j(j < n)$, sort the unsorted segment $A[H[j], H[j] + L[j]]$, $H[j] + L[j] \leq H[j + 1]$.

The problem of GPU-based sorting has been well studied in recent years [8, 34, 28, 27, 3, 33]. However, most of them focus on sorting a large array globally, and there are little work focusing on the problem of sorting a large number of small segments. Our problem is also different from the segmented sorting problem in the *modernGPU* library [3] since the segments in our problem are non-consecutive, i.e., there may be very long sorted intervals between two unsorted segments.

We see two possible solutions for our segmented sorting problem.

- Sort each unsorted segment using a GPU-based sorting library, such as the radix sort in the Thrust library [10].
- Regard all sorted intervals and unsorted segments as a series of consecutive segments and sort them all together using the segmented sort routine provided by the *modernGPU* library [3].

The problem in the first solution is that sorting each segment individually can not fully utilize the GPU resources, since an unsorted segment may be small. The second solution, sorting both the sorted intervals and the unsorted segments, is not efficient either because there will be a large amount of redundant resorting on the intervals that are already sorted.

To address these problems, we propose a new solution for our segmented sorting problem, which makes up the core of our GPU-LS algorithm. In our solution, we mark the unsorted segments as multiple types according to their lengths (number of elements in the group). Then, we sort all segments of the same type in parallel. Since the segments of the same type have similar lengths, they can be handled by the GPU more efficiently, taking advantage of the GPU's data parallelism.

In our GPU-LS, the suffix array SA is mapped to A in the segmented sorting problem, and each *unsorted group* is mapped to an unsorted segment. We maintain the following auxiliary data structures in addition SA and ISA.

- **unsorted suffix index array** ($USIA$). The unsorted suffix index array $USIA_h$ stores the indices in SA_h of each suffix belonging to a certain unsorted *h-group*. $USIA_h$ maintains the same order for all unsorted suffixes as they are in SA_h.
- **unsorted group head flag** ($UGHF$). The $UGHF_h$ flag marks the head of each unsorted group in $USIA_h$ with 1 and the other positions with 0. Both $USIA_h$ and $UGHF_h$ are of the same length in each round of sorting.
- **unsorted group head array** ($UGHA$). The $UGHA_h$ records the head index of each unsorted *h-group* in SA_h.

Algorithm 1: Parallel Suffix Sorting

1. Generate $USKA_{h0}$ using the first h_0 character of each suffix;
2. radix sort ($USKA_{h0}$, SA_{h0}) ; //radix sort SA according the $USKA_{h0}$
3. derive ISA_{h0}; //compute the rank of each suffix in SA_{h0}
4. derive *unsorted* h_0-*groups*; //compute $UGHA_{h0}$, $UGLA_{h0}$, $USIA_{h0}$, $UGHF_{h0}$
5. $h = h_0$;
6. while (there are *unsorted group*s) do
7. {
8. divide *unsorted h-groups* according to their lengths;
9. sort Small-type *h-groups* in parallel;
10. sort Medium-type *h-groups* in parallel;
11. sort Large-type *h-groups* in parallel;
12. update ISA_h;
13. split *unsorted h-groups*; //update $UGHA_h$, $UGLA_h$, $USIA_h$, $UGHF_h$
14. $h = h*2$;
15. }

Figure 1: The algorithm skeleton of our GPU-LS.

- **unsorted group length array** ($UGLA$). The $UGLA_h$ records the length of each unsorted h-*group* in SA_h. Both $UGHA_h$ and $UGLA_h$ are of the same length in each round of sorting.
- **unsorted suffix key array** ($USKA$). After we finish sorting each unsorted h-*group* in SA_h (i.e., we get SA_{2h}), $USKA_h$ will store the key $ISA_h[SA_h[i]+h]$ for each suffix $SA_h[i]$ in the unsorted h-*group*. This key will be used to split h-*group*s to $2h$-*group*s and derive ISA_{2h}.

Figure 1 shows the skeleton of our GPU-LS algorithm. It is based on the sequential LS algorithm [21]. We start by parallel sorting each suffix according their initial h_0 characters to get SA_{h_0} and ISA_{h_0} (Line 1-2 in Algorithm 1). Next we derive ISA_{h_0} (Line 3) and unsorted h_0-*group*s (Line 4). Subsequently, a prefix-doubling loop is performed (Line 8-14). In the loop, we first divide *unsorted group*s generated from the previous loop into several types according to their lengths. Next we sort each type of unsorted groups in parallel using a suitable sorting method. At the end of the loop, we update ISA_h to get ISA_{2h} and split unsorted h-*group*s into $2h$-*group*s accordingly. Each *unsorted* h-*group* is identified with its head index (in $UGHA_h$) and its length (in $UGLA_h$) and we use the group head index of each h-*group* as its h-*rank*.

In the following, we describe the algorithm in more detail.

4.2 Initial Sorting

The initial sorting is done based on the first h_0 characters of each suffix. We construct the key for each suffix T_i using its first h_0 characters, and then perform key-value sorting. We have several alternatives to choose the value of h_0. For text strings with 8-bit characters, we can choose $h_0 = 1$ with each key represented as a single character, or $h_0 = 4$ (or 8) with each key represented as an integer (or long integer). In this step, we choose to use the GPU-based radix sort as it is more efficient than comparison based sort for sorting long arrays [27].

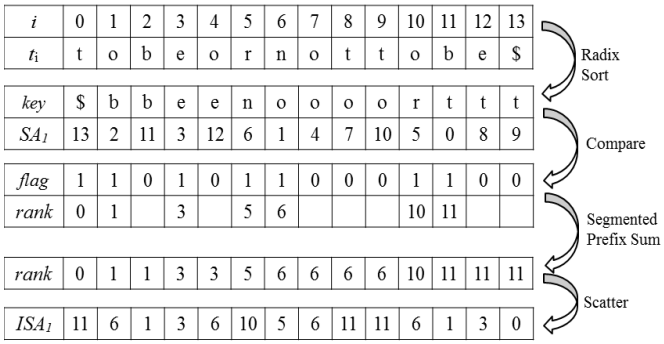

Figure 2: An example of initial sorting and ISA_{h_0} computation for input string *tobeornottobe* ($h_0 = 1$).

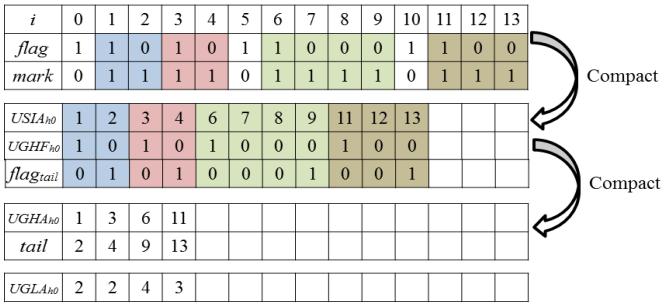

Figure 3: An example of deriving *unsorted group*s after initial sorting. There are four unsorted segments illustrated with four different colors.

Figure 2 illustrates the process of parallel ISA_{h_0} computation. After initial sorting we compare each key with its predecessor. If they are different, we mark the current key as the head of a new h_0-group, and use the current index as its h_0-rank. We propagate the h_0-rank value to all the suffixes within the same group by a segmented prefix-sum operation, and scatter the ranks according to SA_{h_0} to get ISA_{h_0}. By ranking each h-*group* with its head index, the ISA can be updated locally for each *unsorted group* in subsequent sorting steps. If the number of unique ranks equals the string length, the sorting will terminate, otherwise we will derive the unsorted h_0-*group*s in the next step and enter the prefix-doubling loop to sort them.

The process to derive the unsorted h_0-*group*s (Line 4 in Algorithm 1) is also illustrated in Figure 3. We first mark the h_0-*group*s with a temporary mark array by comparing each head flag with its successor: if both of them are 1, the h_0-*group* in the current position is a sorted group and we mark it with 0. The index array and flag array are compacted using the mark to get the $USIA_{h_0}$ and $UGHF_{h_0}$, respectively. and the tail flags are then computed from head flags. The heads and tails for unsorted groups are computed by compacting the head flags and tail flags. Finally, we compute the length of each unsorted group. The compacted index array ($USIA_{h_0}$), the head flag array ($UGHF_{h_0}$), head ar-

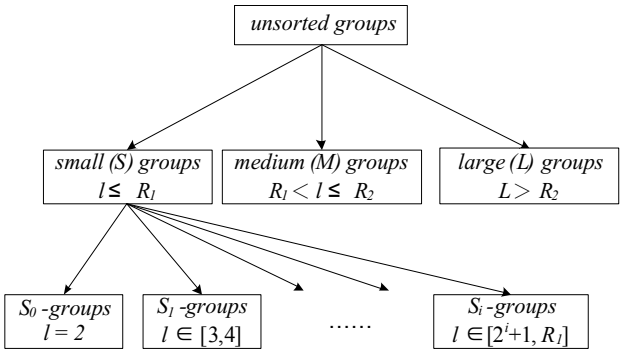

Figure 4: The scheme to divide *unsorted group*s into various types.

Algorithm 2: bitonic segmented sort *S*-type groups

Input: R_1, SA_h, ISA_h, group head array and length array

1. for i from 0 to ($\log(R_1)$-1)
2. {
3. Sort groups of type S_i in parallel, with each thread block sorting $R_1/(2^{i+1})$ groups.
4. }

ray ($UGHA_{h_0}$) and length array ($UGLA_{h_0}$) are all used in subsequent sorting steps.

4.3 Sorting Unsorted Groups

After the initial sorting, we have a large number of *unsorted* h_0*-group*s.

The general idea of our strategy to processing these unsorted groups is that, if one unsorted group is too small to saturate even one single thread block, we will combine multiple such groups and sort them using one thread block to better utilize the GPU resources; otherwise we will sort one group using one or more thread blocks.

To achieve the high data parallelism goal, we first divide the unsorted groups into multiple types according to their lengths. Our dividing scheme is shown in Figure 4. Specifically, we set two thresholds R_1 and R_2 to divide the groups into three types: small-sized groups (S type), medium-sized groups (M type) and large-sized groups (L type). An S type group cannot saturate a single thread block. To process them more efficiently, the S type groups are further divided into sub-types by their logarithmic sizes, and multiple groups of the same sub-type are processed in parallel. For example, if each thread block is to sort 256 elements, we will combine every 128 S_0-type groups or 64 S_1-type groups and sort them by one thread block.

The problem of dividing unsorted groups is reduced to the problem of splitting a list of elements into a number of disjoint partitions, which can be achieved by using a parallel splitting or sorting operation on the GPU.

Sorting S-type Groups.

According to [33], bitonic sort is often the fastest for small sequence of data. Therefore, we choose bitonic sort as the basis for sorting S-type *unsorted group*s. We propose a bitonic segmented sorting algorithm, shown in Algorithm 2. It sorts different sub-types of small groups in sequence whereas within each type all small groups are sorted in parallel.

Figure 5 gives an example of sorting S_3-type *unsorted group*s using one thread block. For illustration purpose, we assume each block contains 64 threads and can sort 4 S_3-type *unsorted group*s. As the *unsorted group*s are divided, each block reads information (head index and length) of four S_3-type groups, and then loads the suffixes for the four groups from SA_h to the shared memory. Moreover, for each suffix $SA_h[i]$ loaded, $ISA_h[SA_h[i]+h]$ is also loaded to the shared memory as the key field for comparing two suffixes. If the length of any group is smaller than their maximum length (16 for S_3-type), the empty space after the group tail will be padded. Next a bitonic sort is performed on the suffixes in the shared memory based on their keys.

This sorting processes requires fewer bitonic steps than the bitonic sort for the global array, since we only need to ensure the sort order in each segment. In the example, four bitonic steps are required to sort the 64 elements in the shared memory, which contains four segments. After the suffixes in the shared memory are sorted, they are written back to SA_h. The key $ISA_h[SA_h[i]+h]$ for each suffix $SA_h[i]$ is written to a temporary array and will be used to derive ISA_{2h}.

In summary, our sorting of S-type groups is very efficient in that : 1) all operations are performed on the shared memory; 2) it requires fewer bitonic steps than sorting the entire array globally; and 3) it does not require inter-block synchronization, since the sorting for each sub-type of groups is finished with one kernel call.

Sorting M-type Groups.

We propose two alternatives to sort M-type *unsorted group*s.

The first alternative is to sort each group with multiple thread blocks using radix sort. We set up a threshold R_{2t} to split each M-type group into several sub-segments, each of which is less than or equal to R_{2t}. Each round of the radix sort contains three steps: (1) block-level histogram; (2) inter-block scan; (3) scatter. In step (1), each thread block processes several sub-segments and the histogram for each sub-segment is computed in parallel. In step (2), we perform a scan on the histograms of the sub-segments that belong to the same *unsorted group*, so that we get the global scatter offset for each sub-segment. In step (3), the entries of each sub-segment is scattered according to the the offset computed from step (2).

The second alternative is to sort the M-type groups using the GPU segmented merge sort. We apply the segmented sorting approach as in the *ModernGPU* project [3]. The process is illustrated in Figure 6. Since the segmented sorting can only sort consecutive segments, we copy the unsorted

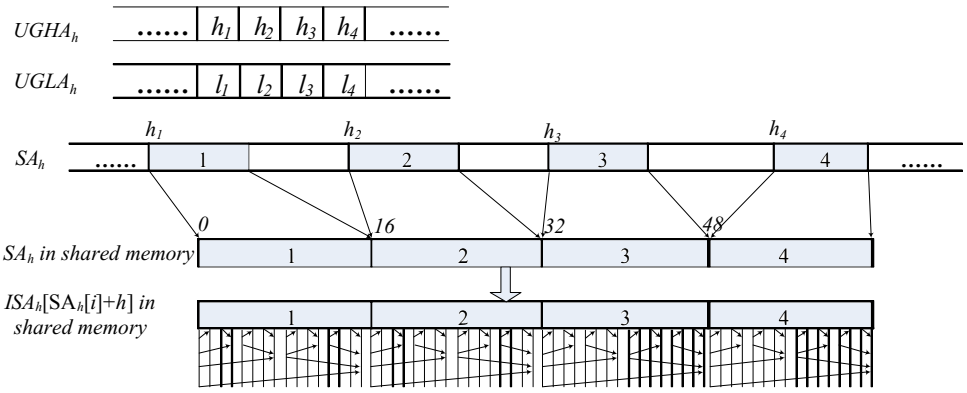

Figure 5: An example of sorting four S-type *unsorted group*s in a single thread block with a variant of bitonic sort. Shade segments in SA_h are *unsorted group*s of the same sub-type (in the range $[9, 16]$ here) whereas clear segments are *unsorted group*s of other types or *sorted group*s. ↗ and ↘ mean sorting in ascending order and descending order, respectively.

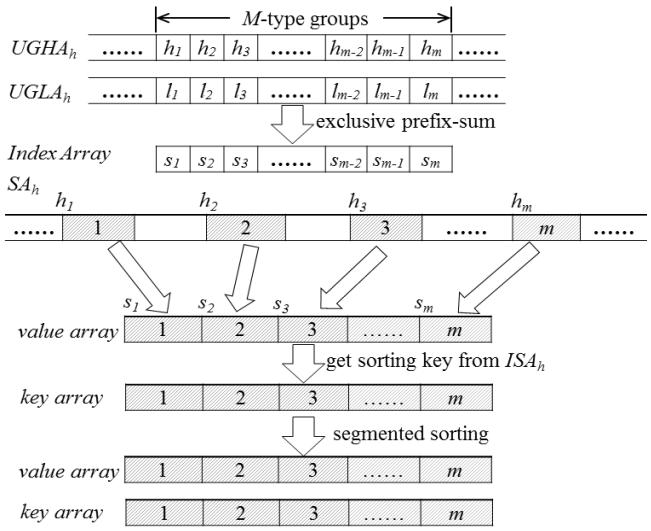

Figure 6: An example of sorting M-type groups using segmented sorting API.

suffixes belonging to M-type groups from SA_h to a temporary array (*value array* in Figure 6) to construct consecutive segments. Since the M-type groups are relatively larger than S-type ones, the suffixes in each group are copied using all threads within one thread block cooperatively to achieve coalesced memory access and utilize memory bandwidth. After getting the sorting key $ISA_h[SA_h[i] + h]$ for each suffix $SA_h[i]$ and storing them in the *key array*, we use segmented merge sort to sort the suffixes in the *value array* based on their keys in the *key array* and the segment indices in the *index array*. Finally, the sorted suffix segments are copied back to their original positions in SA_h and their keys are copied to $USKA_h$.

Sorting L-type Groups.

After the S-type and M-type unsorted h-*group*s are sorted, we are left with only L-type unsorted groups to get SA_{2h}. Our strategy is to sort each such group with an existing sorting API, such as the radix sort in the thrust, since these L-type groups are usually very large and each of them can fully utilize the GPU resources.

4.4 Updating ISA and Splitting Groups

After all three types of groups are sorted, the SA_h has becomes SA_{2h}. We update the ISA_h to get ISA_{2h} and split the unsorted groups with the following steps.

1. We mark the head of each $2h$-*group* contained in certain *unsorted* h-*group* in parallel. For each thread *tid*, we get the entry $i = USIA_h[tid]$, and compare the suffix key $USKA_h[i]$ with its predecessor $USKA_h[i-1]$. If they are different, it indicates a head position of a $2h$-*group*, and we mark $UGHF_h[tid]$ with 1; otherwise $UGHF_h[tid]$ will remain unchanged.

2. We construct a temporary $Rank_{tmp}$ for computing the rank of each suffix. For each thread *tid*, if $UGHF_h[tid]$ is 1, $Rank_{tmp}[i]$ is set to *tid*; otherwise, it is set to 0. Then, we perform a segmented prefix-sum scan on $Rank_{tmp}$ with $UGHF_h$ as the segment flag to get the new rank for each suffix.

We scatter the rank of each suffix to its corresponding position in ISA_h. Only the ranks for suffixes in *unsorted* h-*group*s are updated, and we get ISA_{2h} after the scattering. If all entries in $UGHF_h$ are 1, each suffix will have a unique rank and the suffix sorting is finished. Otherwise, we will prepare $USIA_{2h}$, $UGHF_{2h}$, $UGHA_{2h}$ and $UGLA_{2h}$ in the next step for further sorting.

3. We find *sorted* $2h$-*group*s by comparing each entry in $UGHF_h$ with its successor: if both entries are 1, the current entry lies in a *sorted* $2h$-*group*. The $USIA_h$ and $UGHF_h$ are compacted to remove the indices and flags for *sorted* $2h$-*group*s, and we get $USIA_{2h}$ and $UGHF_{2h}$. The $UGHA_{2h}$ is computed by compacting $USIA_{2h}$ using $UGHF_{2h}$ as the compaction mask. To compute $UGLA_{2h}$, we first derive the tail flag for *unsorted* $2h$-*group* from $UGHF_{2h}$. Then we compact $USIA_{2h}$ using the tail flag array to get the tail

Table 1: Hardware and Software Configuration

Item	Configuration
CPU	two Intel E5-2650 CPUs
GPU	NVIDIA Tesla M2090
OS	CentOS 6.3
CUDA	Version 5.5
Main Memory	64GB

index array. The $UGLA_{2h}$, which stores the length of each *unsorted* $2h$*-group*, is computed using its head index and tail index.

5. EXPERIMENTS

5.1 Experimental Setup

In this section, we evaluate our parallel suffix sorting approach on a server. The hardware and software configuration of the server is shown in Table 1. We compare the performance of our suffix sorting algorithm (referred to as *GPU_LS*) with the fastest CPU-based multithreaded suffix sorting library, *libdivsufsort*; we also compare ours with the most recent GPU-based suffix sorting implementation [5], referred to as *GPU_skew*, which is based on the skew algorithm [13].

We implement our algorithm using C programming language and NVIDIA CUDA API. The device code is compiled using nvcc compiler with *-O3* flag for highest level optimization and *-m64* for using 64-bit address, while the host code is compiled using gcc compiler with the same flags. The GPU time is measured using CUDA API function *cudaEventRecord* and *cudaEventElapsedTime*. The time of copying the input strings into GPU memory and copying the result out is excluded. For *libdivsufsort*, we use its default compile settings, and its running time on CPUs is measured by the C function *gettimeofday*. For each test, we run the program five times and take the average time.

Since we do not have access to the source code of *GPU_skew*, we implement the parallel skew algorithm following the description in the orginal [5] using CUDA and try our best to optimize the code to improve its performance. In this algorithm, we sort the triplets using CUDA thrust radix sort, which is based on [27]. In the top-level recursion, each triplet is represented as three characters in the original string so we pack it into one integer and sort the triplets using 32-bit integer radix sort. However, in the subsequent rounds of recursion each triplet is represented as three rank values (96 bits in total), which cannot use any built-in basic types for radix sort. Therefore, we sort the triplets in two rounds: in the first round we sort the triplets by their last two entries using a 64-bit long integer radix sort; in the second round, we sort the triplets by the first entry using a 32-bit integer radix sort. We use the parallel merge in thrust v1.7, which is an implementation of the GPU Merge Path algorithm [9], to merge the two parts of sorted suffixes in each recursion.

Since our algorithm for sorting small groups is memory latency bound, to hide the latency of memory access and improve the occupancy of streaming multiprocessors, we use the offline optimization tool *CUDA_Occupancy_calculator* [30] to get the maximum number of elements sorted by a single thread block when the peak performance is reached. In our experimental environment, the maximum warp occupancy was reached when there were 256 threads in each thread block and each thread block sorted 256 S-type entries. The impact of the R_1 threshold on the overall performance is studied in detail in section 5.1.

Table 2: Characteristics of test data sets.

Name	Avg. LCP	Max. LCP	$\|\Sigma\|$	StringLength
chr22.dna	1979	199999	4	34,553,758
howto	267	70720	197	39,422,105
etext99	1108	286352	146	105,277,340
rfc	93	3445	120	116,421,901
sprot34.dat	89	7373	66	109,617,186

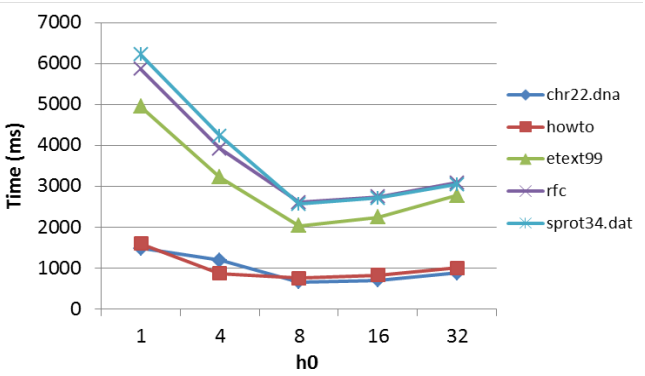

Figure 7: The overall run time with different h_0.

We evaluate the performance of all three implementations using benchmark data from the *Large Canterbury Corpus* [1]. These data sets have been widely used to benchmark the performance of different suffix sorting algorithms. The characteristics of the data set is summarized in Table 2. As shown in the table, the maximum string length is about 116 million characters (bytes). The average LCP (Longest Common Prefix) and maximum LCP equals the average and maximum number of initial characters to be compared to make the suffixes fully sorted,respectively. Therefore, they provide a rough estimation of the suffix sorting workload for the string: the longer the LCP, the heavier the workload.

5.2 Parameter Tuning in GPU_LS

We first study the performance of our algorithm with different choices of h_0. We use the first strategy, multi-thread block radix sort, to sort m-type groups, and set R_1 and R_2 to 256 and 65536, respectively. In the later experiments we will test the performance impact of R_1 and R_2. When h_0 value is equal to 1, 4, or 8, we represent each key as an element of a basic data type (*char*, *int* or *long*) and sort them in one round. When h_0 is greater than 8, we cannot sort them in one round; instead, we first sort the suffixes by their first $h_0 = 8$ characters to get SA_8, and then sort the remainder using the parallel MM algorithm to get SA_{h_0} with $h_0 = 16$. Similarly we can get SA_{32} with more rounds of sorting every eight characters.

The performance result with different choices of h_0 is plotted

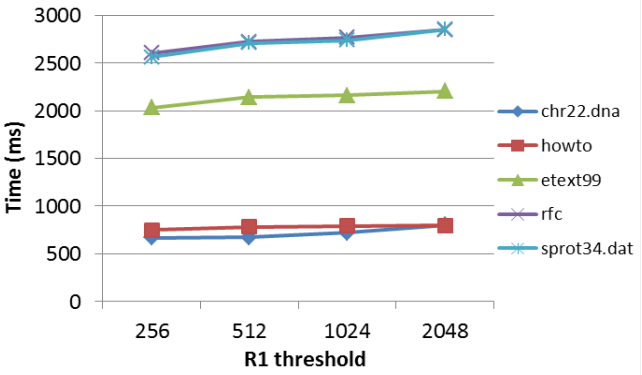

Figure 8: Effect of R_1.

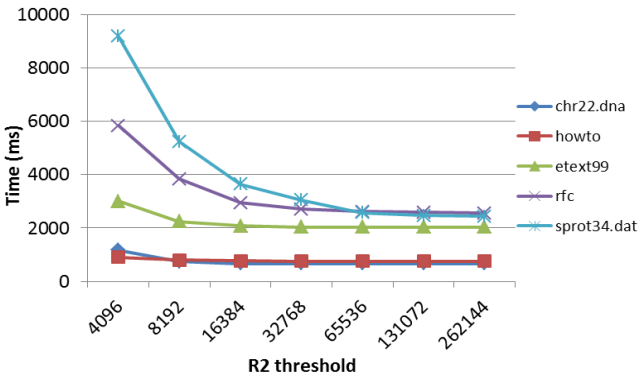

Figure 9: Effect of R_2.

in Figure 7. All five test strings show the same trend: when h_0 increases from 1 to 8, the performance increases and when h_0 further increases from 8 to 32, the performance decreases. The peak performance is achieved when $h_0 = 8$.

The performance improvement with the increase of h_0 from 1 to 8 is because sorting with more initial characters in one round results in more and smaller *unsorted groups* on average, where our group sorting algorithm performs well. In contrast, the performance degrades when h_0 is greater than 8, due to the high overhead of applying the MM algorithm to get SA_{16} and SA_{32}. The time differences between different strings are mainly due to their different LCPs and lengths. In the subsequent experiments, we use $h_0 = 8$ as the default for best performance.

Next we study the impact of the R_1 threshold on the overall performance. Due to the size limit of the shared memory for bitonic segmented sort, the maximum value for R_1 is 2048 in our environment. We use the second alternative, the GPU-based segmented merge sort, to sort the M-type groups. The results are plotted in Figure 8. In this figure, we observe that the overall run time increases slightly with the increase of R_1. The reason for the favorable effect of a small R_1 value is that our bitonic segmented sorting algorithm is designed for small groups. For larger groups, e.g., groups larger than 256 elements, the bitonic sorting will require more rounds, which hurts its efficiency. Therefore, we set the R_1 value to 256, which is also confirmed by the offline optimization tool *CUDA_Occupancy_calculator*.

We further test the performance using different R_2 settings, which are used to separate M-type and L-type *unsorted groups*. We vary the R_2 value from 4096 to 262144 at exponential steps, and the result is shown in Figure 9. We observe that as R_2 increases, the running time first drops significantly, and then flattens after R_2 is greater than 65536. This phenomenon confirms that our strategy of focusing on segmented sorting of a large number of small groups simultaneously is effective in that it improves the GPU resource utilization. When the threshold is large enough, the performance remains almost constant, as sorting of each large group can fully utilize the resources with a global sorting routine.

5.3 Comparison of Algorithms

We first study the time breakdown of our GPU_LS algorithm as well as the GPU_skew [5] to compare their performance charactersitics.

We divide the GPU_LS performance into six time components: initial sorting, S-type group sorting, M-type group sorting, L-type group sorting, ISA updating (after each sorting step) and group processing (include group splitting and dividing a group into different types). In contrast, the GPU_skew is divided into four time components: radix sort, merge, compute rank and other. The two algorithms have different time components due to their inherent differences in the algorithms.

The time breakdowns for GPU_skew and GPU_LS are shown in Figure 10 (a) and (b) respectively. In GPU_skew, the most time-consuming component is to merge the two groups of sorted suffixes $SA^{(i+1)}$ and $SA'^{(i+1)}$ in each recursion step. The second time-consuming is to sort $SA^{(i+1)}$ and $SA'^{(i+1)}$. The other components in total consume no more than 20% of the total time.

In GPU_LS, the four sorting components together take about one half of the overall time. The two most time-consuming components are updating ISA and group processing. Even though the group processing time is considerable, this overhead is worthwhile in that by such preprocessing, the unsorted groups can be sorted more uniformly and more efficiently. As a result, GPU_LS is more than 2x faster than GPU_skew.

Finally, we compare the overall performance of our algorithm with both *GPU_skew* and *libdivsufsort*.

As the latest *libdivsufsort* has multithreading support using OpenMP, we try different OpenMP thread configurations for each dataset and choose the number of threads that reaches the peak performance. Surprisingly for these large datasets, the best performance was always reached in the single-threaded configuration in our environment. For the two GPU-based algorithms, we also apply the configuration that leads to their peak performance.

The overall performance results are presented in Table 3. Our *GPU_LS* outperforms both *GPU_skew* and *libdivsufsort*

Table 3: Performance Comparison of GPU_LS with GPU_skew and *libdivsufsort*.

Name	*GPU_LS* (seconds)	*GPU_skew* (seconds)	*libdivsufsort* (seconds)	speedup (*GPU_skew* / *GPU_LS*)	speedup (*libdivsufsort* / *GPU_LS*)	*GPU_LS* Throughput (MB/sec)
chr22.dna	**0.665**	1.580	3.433	2.38	5.16	51.96
howto	**0.750**	1.786	3.650	2.38	4.87	52.56
etext99	**2.033**	5.052	13.430	2.48	6.61	51.78
rfc	**2.607**	5.561	11.520	2.13	4.42	44.66
sprot34.dat	**2.562**	5.257	12.040	2.05	4.70	42.79

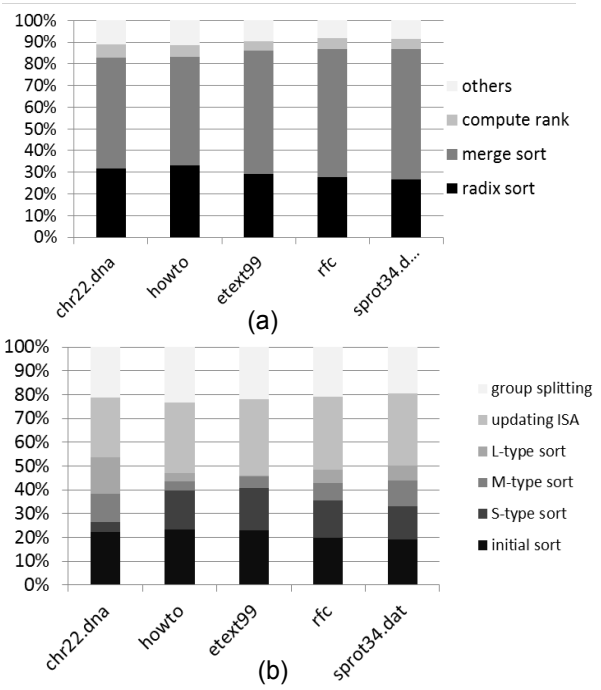

Figure 10: The time breakdown for GPU_skew (a) and our GPU_LS algorithm (b).

for all test strings. The speedup of *GPU_LS* over *libdivsufsort* ranges from 4.42x up to 6.61x, and that over GPU_skew is up to 2.48x. The throughput of *GPU_LS* is up to 52MB/sec, in comparison with 25MB/sec of *GPU_skew* [5].

6. CONCLUSIONS

We have presented GPU-LS, a parallel, prefix-doubling based suffix sorting algorithm on the GPU. The main advantage of LS algorithm is that it avoids sorting prefixes that are already sorted. However, the challenge is that, after the initial sort of the suffixes by their first few characters, we need to sort a large number of small unsorted groups. This challenge is even more severe on the GPU in that sorting of small groups under-utilizes the GPU data parallelism.

We analyze the challenges in our parallel suffix sorting problem and reduce it to a general parallel segmented sorting problem. Our strategy is to divide the groups into different types according to their lengths and create uniform data-parallel workloads for the GPU. We sort each type of suffixes using different sorting strategies. In particular, for the small groups that cannot saturate a single thread block, we sort multiple groups within one thread block, whereas for very large groups we sort each of them directly using the existing sorting API.

Despite the overhead of group dividing, our algorithm is faster than the fastest known CPU-based suffix sorting algorithm as well as the latest GPU-based suffix sorting implementation.

7. REFERENCES

[1] Large canterbury corpus. `http://corpus.canterbury.ac.nz/descriptions/#large`.

[2] libdivsufsort-a lightweight suffix-sorting library. `https://code.google.com/p/libdivsufsort/`.

[3] S. Baxter. Modern gpu library. `http://nvlabs.github.io/moderngpu/`.

[4] M. Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithm. Technical report, Digital Equipment Corporation, 1994.

[5] M. Deo and S. Keely. Parallel suffix array and least common prefix for the gpu. In *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 197–206. ACM, 2013.

[6] P. Ferragina and G. Manzini. Indexing compressed text. *Journal of the ACM (JACM)*, 52(4):552–581, 2005.

[7] N. Futamura, S. Aluru, and S. Kurtz. Parallel suffix sorting. In *In Proc. 9th International Conference on Advanced Computing and Communications*, pages 76–81. IEEE, 2001.

[8] N. Govindaraju, J. Gray, R. Kumar, and D. Manocha. Gputerasort: high performance graphics co-processor sorting for large database management. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 325–336. ACM, 2006.

[9] O. Green, R. McColl, and D. A. Bader. Gpu merge path: a gpu merging algorithm. In *Proceedings of the 26th ACM international conference on Supercomputing*, pages 331–340. ACM, 2012.

[10] J. Hoberock and N. Bell. Thrust: C++ template library for cuda, 2009.

[11] R. Homann, D. Fleer, R. Giegerich, and M. Rehmsmeier. mkesa: enhanced suffix array construction tool. *Bioinformatics*, 25(8):1084–1085, 2009.

[12] H. Itoh and H. Tanaka. An efficient method for in memory construction of suffix arrays. In *String Processing and Information Retrieval Symposium, 1999 and International Workshop on Groupware*,

pages 81–88. IEEE, 1999.

[13] J. Kärkkäinen and P. Sanders. Simple linear work suffix array construction. In *Automata, Languages and Programming*, pages 943–955. Springer, 2003.

[14] J. Kärkkäinen, P. Sanders, and S. Burkhardt. Linear work suffix array construction. *Journal of the ACM (JACM)*, 53(6):918–936, 2006.

[15] D. K. Kim, J. Jo, and H. Park. A fast algorithm for constructing suffix arrays for fixed-size alphabets. In *Experimental and Efficient Algorithms*, pages 301–314. Springer, 2004.

[16] D. K. Kim, J. S. Sim, H. Park, and K. Park. Linear-time construction of suffix arrays. In *Combinatorial Pattern Matching*, pages 186–199. Springer, 2003.

[17] D. K. Kim, J. S. Sim, H. Park, and K. Park. Constructing suffix arrays in linear time. *Journal of Discrete Algorithms*, 3(2):126–142, 2005.

[18] P. Ko and S. Aluru. Space efficient linear time construction of suffix arrays. In *Combinatorial Pattern Matching*, pages 200–210. Springer, 2003.

[19] P. Ko and S. Aluru. Space efficient linear time construction of suffix arrays. *Journal of Discrete Algorithms*, 3(2):143–156, 2005.

[20] F. Kulla and P. Sanders. Scalable parallel suffix array construction. *Parallel Computing*, 33(9):605–612, 2007.

[21] N. J. Larsson and K. Sadakane. Faster suffix sorting. *Theoretical Computer Science*, 387(3):258–272, 2007.

[22] H. Li and R. Durbin. Fast and accurate short read alignment with burrows–wheeler transform. *Bioinformatics*, 25(14):1754–1760, 2009.

[23] U. Manber and G. Myers. Suffix arrays: a new method for on-line string searches. *siam Journal on Computing*, 22(5):935–948, 1993.

[24] G. Manzini and P. Ferragina. Engineering a lightweight suffix array construction algorithm. In *AlgorithmsąłESA 2002*, pages 698–710. Springer, 2002.

[25] G. Manzini and P. Ferragina. Engineering a lightweight suffix array construction algorithm. *Algorithmica*, 40(1):33–50, 2004.

[26] R. K. Menon, G. P. Bhat, and M. C. Schatz. Rapid parallel genome indexing with mapreduce. In *Proceedings of the second international workshop on MapReduce and its applications*, pages 51–58. ACM, 2011.

[27] D. Merrill and A. Grimshaw. Revisiting sorting for gpgpu stream architectures. Technical Report CS2010-03, University of Virginia, Department of Computer Science, Charlottesville, VA, USA, 2010.

[28] 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.

[29] H. Mohamed and M. Abouelhoda. Parallel suffix sorting based on bucket pointer refinement. In *Biomedical Engineering Conference (CIBEC), 2010 5th Cairo International*, pages 98–102. IEEE, 2010.

[30] Nvidia. Cuda occupancy calculator. `http://developer.download.nvidia.com/compute/cuda/CUDA_Occupancy_calculator.xls`.

[31] V. Osipov. Parallel suffix array construction for shared memory architectures. In *String Processing and Information Retrieval*, pages 379–384. Springer, 2012.

[32] S. J. Puglisi, W. F. Smyth, and A. H. Turpin. A taxonomy of suffix array construction algorithms. *ACM Computing Surveys (CSUR)*, 39(2):4, 2007.

[33] 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. IEEE, 2009.

[34] N. Satish, C. Kim, J. Chhugani, A. D. Nguyen, V. W. Lee, D. Kim, and P. Dubey. 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*, pages 351–362. ACM, 2010.

[35] K.-B. Schürmann and J. Stoye. An incomplex algorithm for fast suffix array construction. *Software: Practice and Experience*, 37(3):309–329, 2007.

[36] J. Seward. On the performance of bwt sorting algorithms. In *Data Compression Conference, 2000. Proceedings. DCC 2000*, pages 173–182. IEEE, 2000.

[37] W. Sun and Z. Ma. Parallel lexicographic names construction with cuda. In *Parallel and Distributed Systems (ICPADS), 2009 15th International Conference on*, pages 913–918. IEEE, 2009.