Fast Parallel Suffix Array on the GPU

Leyuan Wang[1], Sean Baxter[2], and John D. Owens[3]

[1] Department of Computer Science, University of California, Davis,
Davis, CA 95616, USA
leywang@ucdavis.edu
[2] D. E. Shaw Research, New York, NY 10036, USA
sean.baxter@deshawresearch.com
[3] Department of Electrical & Computer Engineering, University of California, Davis,
Davis, CA 95616, USA
jowens@ece.ucdavis.edu

Abstract. We implement two classes of suffix array construction algorithms on the GPU. The first, skew, makes algorithmic improvements to the previous work of Deo and Keely to achieve a speedup of 1.45x over their work. The second, a hybrid skew and prefix-doubling implementation, is the first of its kind on the GPU and achieves a speedup of 2.3–4.4x over Osipov's prefix-doubling and 2.4–7.9x over our skew implementation on large datasets. Our implementations rely on two efficient parallel primitives, a merge and a segmented sort. We also demonstrate the effectiveness of our implementations in a Burrows-Wheeler transform and a parallel FM index for pattern searching.

Keywords: suffix array, parallel, GPU, skew, prefix-doubling, Burrows-Wheeler transform, FM index

1 Introduction

The suffix array (SA) of a string is the sorted set of all suffixes of the string. This data structure is used in a broad spectrum of applications, including data compression, bioinformatics, and text indexing. The suffix array, along with its first construction algorithm, was introduced by Manber and Myers [11] as a more space- and cache-efficient, and simpler to construct alternative to suffix trees.

The straightforward way to generate a suffix array from a string is to simply sort all suffixes of that string using a comparison-based sorting algorithm. For a string of length n, this construction takes $\mathcal{O}(n \log n)$ suffix comparisons and each suffix comparison has time complexity $\mathcal{O}(n)$, so the total time needed is $\mathcal{O}(n^2 \log n)$. The key insight to develop a more efficient algorithm is to leverage the fact that suffixes are not arbitrary strings but related to each other.

The existing suffix array construction algorithms (SACAs) that leverage this property can be divided into three classes: prefix-doubling, recursive and induced copying. The first class of SACAs, prefix-doubling, sorts the suffixes of a string by their prefixes, the length of which is doubled every iteration. The idea was originally proposed by Karp et al. [7], first applied to suffix array construction by

Manber and Myers [11] (MM), and later optimized by Larsson and Sadakane [8] (LS). LS is more efficient than MM, because it removes the unnecessary scanning of fully sorted suffixes from the previous pass. The second class of SACAs recursively sorts a subset of the suffixes, use the order of the sorted subset to infer the order of remaining subset, and finally merge the two sorted subsets to get the order of the entire set. The skew algorithm proposed by Kärkkäinen and Sanders [6] (KS) is a popular linear-time recursive algorithm. The final class of SACAs, induced copying, is non-recursive and uses already-sorted suffixes to quickly induce a complete order of the suffixes. Like the recursive formulation, their time complexity is $\mathcal{O}(n)$.

The recent explosion in data sizes and the emergence of commodity data-parallel processors motivate efficient parallel implementations of SACAs. In this paper, we focus on highly data-parallel SACAs that are suitable for implementation on devices such as many-core GPUs and multi-core CPUs. Because of their high arithmetic and memory throughput, these processors are well-suited for data-intensive computing tasks such as SACAs. However, parallelizing SACAs is a significant challenge.

Both Osipov [15] and Deo and Keely [2] have done seminal work on highly parallel SACAs on GPUs. Deo and Keely analyze the aforementioned three SACA classes and conclude that induced copying has numerous data dependencies and note a lack of parallel approaches to exploit this technique. Osipov concludes that prefix-doubling algorithms are more cost-efficient to implement on the GPU compared with the linear-time recursive skew approach, because the former only requires fast GPU radix sorting of (32-bit key, 32-bit value) pairs, while skew needs to sort large tuples by comparison-based sorting and merging. On the other hand, Deo and Keely conclude that skew is best suited for the GPU as all its phases can be readily mapped to a data-parallel architecture, while prefix-doubling has an irregular, data-dependent number of unsorted groups across phases, and the amount of work per group in each iteration is non-uniform.

Recently, Liu et al. [10] and Pantaleoni [16] have proposed scalable, space-efficient methods that exploit the sorting speed of modern GPUs for blockwise suffix sorting targeting bioinformatics applications to work specifically with large collections of relatively short DNA strings. Because the GPU has limited memory, Liu et al. focus on dividing the large inputs into several sets and sorting each set using a GPU-accelerated method while Pantaleoni solves the problem using insertion, with GPU sorting new blocks and CPU inserting the symbols into the external final result.

In this work, we address the parallel SACA problem by designing, implementing, and comparing two different formulations of SACAs on NVIDIA GPUs. We make three main contributions.

1. Our skew approach incorporates several optimizations that yield a speedup of 1.45x over Deo and Keely's implementation.
2. We also implement a hybrid non-recursive skew/prefix-doubling SACA that overcomes the parallelization challenges identified by Deo and Keely and

performs much better than Osipov's plain prefix-doubling. Comparing our two implementations, we revisit Deo and Keely's conclusions on the most appropriate formulation for parallel SACAs, demonstrate that a recursive doubling-like formulation can be efficiently mapped to GPUs and that our hybrid implementation in general produces the fastest SACA implementation on GPUs. The speedup is as high as 12.76× over Deo and Keely's skew implementation, up to 4.4× over Osipov's parallel prefix-doubling, and 7.9× over our optimized skew implementation.

3. We integrate our hybrid implementation into our GPU implementation of the Burrows-Wheeler transform (BWT) and an FM index-based pattern search application.

2 Background & Preliminaries

We begin with the algorithmic background for the string algorithms we implemented. Section 2.1 provides the notation for suffix array construction that we use throughout the paper. Readers already familiar with the Burrows-Wheeler Transform (Section 2.2), the FM index (Section 2.3), and GPU terminology (Section 2.4) can skip to Section 3.

2.1 The Suffix Array

Consider an input string x of length $n \geq 1$ ending with a lexicographically smallest suffix (\$). We denote the suffix starting at position i (i.e., $x[i, \ldots, n-1]$) by *suffix i*. For convenience, let suffixes with starting position i where $i \bmod 3 \not\equiv 0$ be S_{12}, suffixes with starting position j where j mod $3 \equiv 0$ be S_0, and suffixes with starting position k where k mod $3 \equiv 1$ be S_1.

The ***suffix array*** (SA) of x is defined as an $n+1$ length array such that $\mathrm{SA}[j]=i$ means "*suffix i* is the jth suffix of x in ascending lexicographical order". The ***inverse suffix array*** (ISA) is defined as follows:

$$\mathrm{ISA}[i]=j \iff \mathrm{SA}[j]=i$$

This implies that *suffix i* has rank j in lexicographic order. ISA is also called the *lexicographic ranks* of suffixes. For convenience, we denote the suffix array of S_{12} by $\mathrm{SA}_{[12]}$ and that of S_0 as $\mathrm{SA}_{[0]}$, correspondingly for the inverse suffix array, $\mathrm{ISA}_{[12]}$ and $\mathrm{ISA}_{[0]}$, and we denote the lexicographic ranks of S_1 by $\mathrm{ISA}_{[1]}$.

Both algorithms we describe sort prefixes with increasing length $h \geq 1$. We will refer to this partial ordering as an h-order of suffixes. Suffixes that are equal under h-order are given the same rank, and put into the same h-group. If the sorting process is stable, h-groups with a larger h are refinements over their counterparts with a smaller h. Suffixes in a partial h-order are stored with their indexes in an approximate suffix array SA_h, and their ranks in ISA_h.

2.2 The Burrows-Wheeler Transform

The BWT of a string is generated by lexicographically sorting the cyclic shift of the string to form a string matrix and taking the last column of the matrix. The BWT groups repeated characters together by permuting the string; it is also reversible, which means the original string can be recovered. These two characteristics make BWT a popular choice for a compression pipeline stage (for instance, bzip2). It is directly related to the suffix array: the sorted rows in the matrix are essentially the sorted suffixes of the string and the first column of the matrix reflects a suffix array. The BWT of a string x can be computed from its SA as follows:

$$\mathrm{BWT}[i] = \begin{cases} x[\mathrm{SA}[i]-1] & \text{if } \mathrm{SA}[i] > 0 \\ \$ & \text{if } \mathrm{SA}[i] = 0 \end{cases}$$

Table 1 shows an example of the SA, ISA and BWT of the input string “banana” as follows.

Table 1. SA, ISA and BWT for the example string “banana”.

i	Suffix	Sorted Suffix	SA[i]	ISA[i]	Sorted Rotations	BWT[i]
0	banana$	$	6	4	$banana	a
1	anana$	a$	5	3	a$banan	n
2	nana$	ana$	3	6	ana$ban	n
3	ana$	anana$	1	2	anana$b	b
4	na$	banana$	0	5	banana$	$
5	a$	na$	4	1	na$bana	a
6	$	nana$	2	0	nana$ba	a

2.3 The FM index

Proposed by Ferragina and Manzini [4], the FM (Full-text, Minute-space) index is a compressing and indexing method that allows compression of input text while still supporting fast arbitrary pattern searches. It is a lightweight *compressed suffix array* that combines the BWT and the suffix array data structure. The compressed index can be used to efficiently find the number of occurrences of a pattern from the text, as well as locate the position of each occurrence. The authors describe an algorithm called backward_search that calculates how many times a pattern occurs in BWT-compressed text without decompressing it. We refer the reader to the original paper [4] for further detail.

2.4 The Graphics Processor Unit (GPU)

In the following discussion we use NVIDIA CUDA terminology. Modern GPUs are massively parallel processors that support tens of thousands of hardware-scheduled threads running simultaneously. These threads are organized into blocks and the hardware schedules blocks of threads onto hardware cores. High-end NVIDIA GPUs have on the order of 16 streaming-processor (SP) cores, each of which contains 32-wide SIMD (single-instruction, multiple-data) units that run 32 threads in lockstep. GPUs also feature a memory hierarchy of per-thread registers, per-block shared memory, and off-chip global DRAM accessible to all threads. CUDA programs ("kernels") specify the number of blocks and threads per block under a SIMT (single-instruction, multiple-thread) programming model. Lindholm et al. [9] provides more detail on modern GPU hardware and Nickolls et al. [14] on the GPU programming model.

Efficient GPU programs have enough work per kernel to keep all hardware cores busy (load-balancing); strive to reduce thread divergence (when neighboring threads branch in different directions); aim to access memory in large contiguous chunks to maximize achieved memory bandwidth (coalescing); and minimize communication between CPU and GPU. Designing an SACA that achieves all of these goals is a significant challenge. We also prioritize using high-performance parallel algorithmic GPU primitives (e.g., scan, radix sort, compact, segmented sort) when applicable.

3 Algorithms & Analysis

We implement two fast parallel SACAs, skew (Section 3.1) and a skew/prefix-doubling hybrid (Section 3.2).

3.1 Parallel Skew Algorithm

```
dk-SA (int* T, int* SA, int length)
Initialize Mod12() // form triplet s12, s0
RadixSort (s12) // LSD radix sort 1st char
RadixSort (s12) // LSD radix sort 2nd char
RadixSort (s12) // LSD radix sort 3rd char
lexicRankOfTriplets (s12)
if (!allUniqueRanks) then
    dk-SA () // Recurse
    storeUniqueRanks()
else
    computeSAFromUniqueRank ()
RadixSort ($\mathrm{ISA}_{[1]}$)
RadixSort (s0)
Merge (s0,s12)
```

```
skew-SA (int* T, int* SA, int length)
Initialize Mod12() // form triplet s12, s0
RadixSort (s12) // LSD radix sort 1st char
RadixSort (s12) // LSD radix sort 2nd char
RadixSort (s12) // LSD radix sort 3rd char
if (!allUniqueRanks) then
    lexicRankOfTriplets (s12)
    skew-SA ()// Recurse
    storeUniqueRanks()
Compact ($\mathrm{ISA}_{[12]}$) // compact out the order
    of $\mathrm{ISA}_{[1]}$
RadixSort (s0)
Merge (s0,s12)
```

Fig. 1. Left, Deo and Keely's skew implementation pseudocode; right, ours.

Our first approach is implementing the skew algorithm using massively parallel kernels based on KS [6] and similar to the OpenCL implementation of Deo and Keely [2]. We compare our implementation of skew with Deo and Keely's in Fig. 1 and now describe several algorithmic optimizations over their work.

Both methods start by extracting S_{12} and S_0 from an input string (line 2) and launching a 3-step least significant digit (LSD) radix sort using Merrill and Grimshaw's approaches [12] to find the order of S_{12} based on their first triplets (line 3 to line 5). In the first iteration, each triplet is composed of the first three characters of each S_{12}. The ranks of the triplets are then used as the value for the key-value sort in the recursive iterations.

The ranks are computed by counting unique triplets. In practice, this is done by first comparing each triplet against its predecessor, storing a flag of 1 whenever they are unequal, and then doing a prefix-sum of the list of flags. We use the same flagging method as Deo and Keely to tell if S_{12} are fully sorted (line 6 right and line 7 left), but we do the prefix-sum only when the suffixes are not fully sorted instead of immediately after computing the flagging list (line 7 right and line 6 left). This change saves us from having to compute $\mathrm{SA}_{[12]}$ from $\mathrm{ISA}_{[12]}$ if we are at the end of the recursion and the suffixes are fully sorted (line 10 and 11 left). After the recursion, they continue to compute $\mathrm{SA}_{[0]}$ by sorting S_0 with a 2-step LSD radix sort of (the first character of *suffix* i, rank of *suffix* $i+1$) pairs where *suffix* $i \in S_0$ and thus *suffix* $i+1 \in S_{12}$ whose rank is known from the previous steps (line 13 and 14 left). We use a faster one-step radix sort because the order of the ranks of S_1 (equivalent to $\mathrm{ISA}_{[1]}$) can be filtered out from $\mathrm{ISA}_{[12]}$ (result of line 7 right) using a compact operation (line 11 and 12 right).

The above optimizations allow our implementation to only use 2/3 of Deo and Keely's memory bandwidth in the recursive part, and 3 fewer memory transactions in the last round.

Finally, we optimize the merge step that combines the two sorted suffix arrays $\mathrm{SA}_{[0]}$ and $\mathrm{SA}_{[12]}$ while avoiding load-imbalance. Deo and Keely use the merge technique of Satish et al. [18], binary search, and memory locality optimizations of Davidson et al. [1]. Their work suffers from load-imbalance due to having two separate-sized lists being processed independently. Instead, we utilize vectorized sorted search to map threads and blocks to equally sized sections of each partition, which successfully avoids load-imbalance. This method is based on the Merge Path approach of Green et al. [5] and implemented as a merge primitive in the second author's *Modern GPU* library[4] and is described here for the first time in the literature.

The two keys to an efficient GPU merge operation are (1) dividing the two sorted inputs into independent chunks of equal sized work and (2) ensuring that the outputs of each of those chunks of work are contiguous in the final merged output. This contrasts with Deo and Keely's approach, where chunk size is not uniform. The key operation, then, is to identify the split points. The obvious way would require a two-dimensional search across both input arrays; Merge Path instead describes an elegant transformation to an one-dimensional search along a diagonal that connects the two input arrays.

[4] Code is available at `http://nvlabs.github.io/moderngpu` and described in `http://nvlabs.github.io/moderngpu/merge.html`.

Our implementation performs a two-part, hierarchical split: first dividing the entire input into equal-sized tiles that can be assigned to blocks, then dividing each tile into equal-sized subtiles that can be assigned to threads. The merge is completely parallel (not cooperative) between threads; its inputs are in shared memory and its outputs are in registers. The result is a highly load-balanced, parallel-friendly implementation that achieves a throughput of greater than half the peak bandwidth of the GPU, compared to 12.1% of the theoretical peak for the implementation by Green et al. [5].

3.2 Skew/Prefix-doubling

Deo and Keely's work marked a significant milestone as the first implementation of a linear-time SACA skew on the GPU. However, skew on GPUs has two significant disadvantages:

1. As the skew formulation is recursive, we cannot parallelize across iterations.
2. At the end of each iteration, we may have sets of triplets that are fully sorted. However, to keep the algorithm recursive, we cannot declare these fully sorted suffixes complete and leave them out of further iterations; instead we must process them on every iteration, which results in a large amount of redundant work.

To address these two disadvantages, we turn to a combination of skew and prefix-doubling, which turns out to be a better fit for modern GPU architectures.

In our implementation, we still leverage our skew framework: we keep the first step of skew, which reduces the string size by a factor of 2/3, and the final skew merge stage, which is trivial. Only after the first step of skew do we transition to our non-recursive better-performing prefix-doubling implementation. In the first stage, we select all S_{12} suffixes, forming 3-character substrings, and do a 25-bit radix sort (25 bits for 3 chars from a constant alphabet in the range $[0 \dots 255]$ plus a sentinel letter \$) on those substrings. Then we compute the ranks of S_{12} and assign the ranks into an inverse suffix array ($\mathrm{ISA}_{[12]}$). From now on, we work with suffixes in partially-sorted order rather than text order. In other words, after this initial radix sort, all suffixes with the same 3-character prefix are contiguous in our array (i.e.,"Fun" is next to "Funicular") no matter where they appear in the original text.

Next, in our prefix-doubling step, we sort by ($\mathrm{ISA}[\mathrm{SA}[i]+\delta]$, $\mathrm{ISA}[\mathrm{SA}[i]+2\delta]$) pairs, where δ is the length of prefix which doubles in each iteration until all suffixes are in their own *segment*s, which we define as a set of suffixes that are equal up to the current substring length. These are each given a rank (the index of the first element in the segment within the string). The rank of the segment next to the current one is used as the key for the next pass, and on each iteration, we double the length of the prefix. The key to high prefix-doubling performance is our ability to sort efficiently within segments, even though the number of segments and their sizes are non-uniform and are not known at compile time (this is the specific concern about prefix-doubling raised by Deo and Keely). We address this with an efficient *segmented sort* primitive, which we describe next. In our implementation, we also identify suffixes at the end of each iteration that are singletons in their own segments—their final positions in the suffix array are fixed, so we re-rank and compact them out of the working suffix array. This way, future iterations only need to consider suffixes whose final positions are not yet fixed.

Segmented sort The input to segmented sort is a contiguous list of segments with a variable number of unsorted items per segment; the output is the same list of segments but with items sorted within each segment. One way to solve this problem is to sort each segment one at a time, but it is likely on a highly parallel machine that many (or even most) segments will not have enough work to fill the machine. Another approach is to do a full sort over all items, but this is inefficient because it ignores the significant work that has already been completed in classifying the items into segments. We wish to both work on all segments simultaneously but still leverage the presence of segments. The challenge for an efficient *segmented sort* implementation is the variation in the size and number of segments. This method is implemented as a segmented-sort primitive in the second author's *Modern GPU* library[5] and is described here for the first time in the literature. An efficient segmented sort is the difference-maker in developing a competitive prefix-doubling implementation.

The core of our segmented sort implementation is merging, in the same style as the previously-described merge kernel. For illustrative purposes, consider a full merge sort of a single segment. We would begin by dividing the work into equal-sized blocks, sort each block of elements independently, then use our efficient merge to merge blocks of work together, starting with many small merges and concluding with one large merge. We have previously claimed that the most efficient way for us to merge is to use fixed-size blocks of work, which gives us straightforward parallelization and perfect load balancing.

How do we adapt such a merge in the presence of segments? We must respect the segmentation during the merge, and the way we do this is using a key insight: During a merge of two contiguous lists, *the only segment that is affected by the merge is one that spans the boundary between two blocks.* All other segments involved in this merge are copied without change from input to output. We illustrate this in Fig. 2.

The final optimization is early exit. The number of input boundaries is cut in half on each iteration, so once a segment no longer crosses an active input boundary, we can conclude that segment is fully sorted and mark it as inactive. A tile with no active segments is done with its work and can exit. Especially with a large number of small segments, this early-exit optimization dramatically decreases the number of passes over the data, the required memory bandwidth, and the overall runtime.

[5] Code is available at `http://nvlabs.github.io/moderngpu` and described in `http://nvlabs.github.io/moderngpu/segsort.html`.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$\underset{\wedge}{\boldsymbol{m}}$	$\boldsymbol{m}$	$\boldsymbol{i}$	$\boldsymbol{i}$	$\boldsymbol{s}$	$\underset{\wedge}{s}$	i	i	$\underset{\wedge}{\boldsymbol{s}}$	$\boldsymbol{s}$	$\boldsymbol{i}$	$\boldsymbol{i}$	$\boldsymbol{p}$	$\underset{\wedge}{p}$	i	i
(0	1	2	$3)\ \vert$	(4	5	6	$7)\vert$	(8	9	10	$11)\vert$	(12	13	14	15)
$(\underset{\wedge}{\boldsymbol{i}}$	$\boldsymbol{i}$	$\boldsymbol{m}$	$\boldsymbol{m})\vert$	$(\boldsymbol{s}$	$\underset{\wedge}{i}$	i	$s)\vert$	$(\underset{\wedge}{\boldsymbol{i}}$	$\boldsymbol{i}$	$\boldsymbol{s}$	$\boldsymbol{s})\ \vert$	$(\boldsymbol{p}$	$\underset{\wedge}{i}$	i	$p)$
(0	1	2	$3\ \vert$	4	5	6	$7)\vert$	(8	9	10	$11\ \vert$	12	13	14	15)
$(\underset{\wedge}{\boldsymbol{i}}$	$\boldsymbol{i}$	$\boldsymbol{m}$	$\boldsymbol{m}\vert$	$\boldsymbol{s}$	$\underset{\wedge}{i}$	i	$s)\vert$	$(\underset{\wedge}{\boldsymbol{i}}$	$\boldsymbol{i}$	$\boldsymbol{p}$	$\boldsymbol{s}\ \vert$	$\boldsymbol{s}$	$\underset{\wedge}{i}$	i	$p)$
0	1	2	$3\ \vert$	4	5	6	$7\ \vert$	8	9	10	$11\ \vert$	12	13	14	15
$\underset{\wedge}{\boldsymbol{i}}$	$\boldsymbol{i}$	$\boldsymbol{m}$	$\boldsymbol{m}\vert$	$\boldsymbol{s}$	$\underset{\wedge}{i}$	i	$s\ \vert$	$\underset{\wedge}{\boldsymbol{i}}$	$\boldsymbol{i}$	$\boldsymbol{p}$	$\boldsymbol{s}\ \vert$	$\boldsymbol{s}$	$\underset{\wedge}{i}$	i	p

Fig. 2. Segmented Sort example. Consider an input string composed of 16 random characters grouped into four irregular segments (the first row). The head of each segment is marked with carets. First, we divide the characters equally into four blocks of four elements each, then launch four "blocksorts" to sort four inputs each while maintaining segment order. Next, we merge the first block with the second and the third block with the fourth. Note in the merge of the third and fourth blocks, two separate segments are involved, but only the first segment—the one that crosses the boundary between the two inputs—changes as a result of the segmented merge. Finally, there's one active input boundary left in the middle but with no segment crossing it, which means all segments are fully sorted and there's no need for further merging, so this is an early-exit. The final result is segments in the same order as the input, but sorted within each segment (the last row).

Comparison against plain prefix-doubling implementation Our hybrid prefix-doubling method has several optimizations over the pure prefix-doubling implementation by Osipov [15]. He modifies MM by replacing chunks of (32-bit key, 32-bit value) radix sort with a single (32-bit key, 64-bit value) radix sort. At the end of each iteration, he filters out fully-sorted suffixes to avoid unnecessary re-sorting, similar to LS. Throughout the implementation, he uses parallel primitives including prefix-sum, radix sort, random gather from and scatter to memory based on Merrill's *back40computing* library[6].

In our method, the first step of skew—a single pass of (32-bit key, 25-bit value) radix sort—is inexpensive and gives us a reduction ratio of 0.67. This is significantly better than Osipov's initial sorting of the first 4 characters. Also, our massively parallel *segmented sort* primitive has better locality than radix-sorting integer tuples across global memory. Furthermore, our induction step in the skew framework is cheaper than a radix sort when sorting the remaining 1/3 suffixes. Though we need an additional merge in the final step, our parallel merge primitive is quite efficient (see Section 3.1). We compare our approach with Osipov's pure prefix-doubling implementation in Fig. 3.

[6] `https://code.google.com/p/back40computing/`

```
Initialize SA$_4$ by sorting suffixes by their first 4 characters
Initialize ISA$_4[i]$ with the 4-rank of $i$ =head of $i$'s 4-group in SA$_4$
$size = n, h = 4$
while $size > 0$ do
    Scan SA$_h$ and generate tuples (SA$_h[j] - h$,ISA$_h$[SA$_h[j] - h$],ISA$_h$[SA$_h[j]$])
    RadixSort tuples by 2nd component stably // contains SA$_{2h}$
    Refine $h$-heads of $h$-groups // Re-rank
    Update ISA$_{2h}$ // contains ISA$_{2h}$
    Filter and Compact SA$_{2h}$
    $size = size$ of SA$_{2h}$
    $h = h * 2$
```

```
Initialize Mod12() // form triplets s12, s0
RadixSort (s12) // 25-bit radix sort on triplets s12
ComputeRanks ISA$_{[12]}$
$size = \frac{2}{3}n, h = 6$
while $size > 0$ do
    SegmentedSort (ISA[SA[$i$]+$h$],ISA[SA[$i$]+2$h$])
    Update ISA$_{2h}$ and Compact SA$_{2h}$
    $size = size$ of SA$_{2h}$
    $h = h * 2$
Compact (ISA$_{[12]}$) // compact out the order of ISA$_{[1]}$
RadixSort (s0)
Merge (s0,s12)
```

Fig. 3. Left, Osipov's parallel prefix-doubling description; right, our skew/prefix-doubling.

Skew vs. prefix-doubling Skew with a difference cover modulo 3 is a "prefix tripling" technique[7], tripling the pace at which it samples its ranks each round. It is more efficient as a prefix-tripler than an integer alphabet sort, because the 2-integer segmented sort of prefix-doubling is certainly much faster than the 3-integer radix sort of skew. In its radix sort, skew uses the most significant digit simply to get the suffix back in its original segment, which comes for free with prefix-doubling's segmented sort. Furthermore, skew cannot drop fully-sorted suffixes, because it needs to transform their ranks into the new coordinate system in which they will be sampled by the remaining unsorted suffixes. With prefix-doubling, suffixes are ranked in the same coordinate system (i.e., where they would be placed in the final sorted suffix array) throughout the computation, and since there is no need to re-rank fully-sorted suffixes, we can remove them from the problem.

For real-world texts, this makes prefix-doubling more efficient than skew. Skew has a solid reduction ratio of 0.67, regardless of the data. Prefix-doubling has a worst-case reduction ratio of 1.0 (if the pass fails to resolve any suffixes), but has a reduction ratio on real-world text that is usually favorable.

4 Experiments & Results

In this section we present a detailed experimental evaluation of our implementations of suffix array algorithms. For a more thorough comparison, we re-implement Deo and Keely's method using a current state-of-the-art radix sort primitive from Merrill's *CUB* library[8] merge primitive on an NVIDIA GPU using CUDA. For convenience, we call Deo and Keely's OpenCL parallel skew

[7] A difference cover D modulo h, denoted by D_h, is a set of integers $i \in \{0, \ldots, h-1\}$ such that $i \equiv k - j \pmod h$ for some $j, k \in D_h$. For example, $\{1, 2\}$ is a difference cover modulo 3 and $\{1, 2, 4\}$ is a difference cover modulo 7.

[8] `http://nvlabs.github.io/cub/`

implementation on an AMD GPU *dk-amd-SA*, our CUDA implementation of Deo and Keely's approach *dk-nvidia-SA*, Osipov's parallel prefix-doubling *osipov-SA*, our parallel skew implementation *skew-SA*, and our parallel hybrid skew/prefix-doubling implementation *spd-SA*.

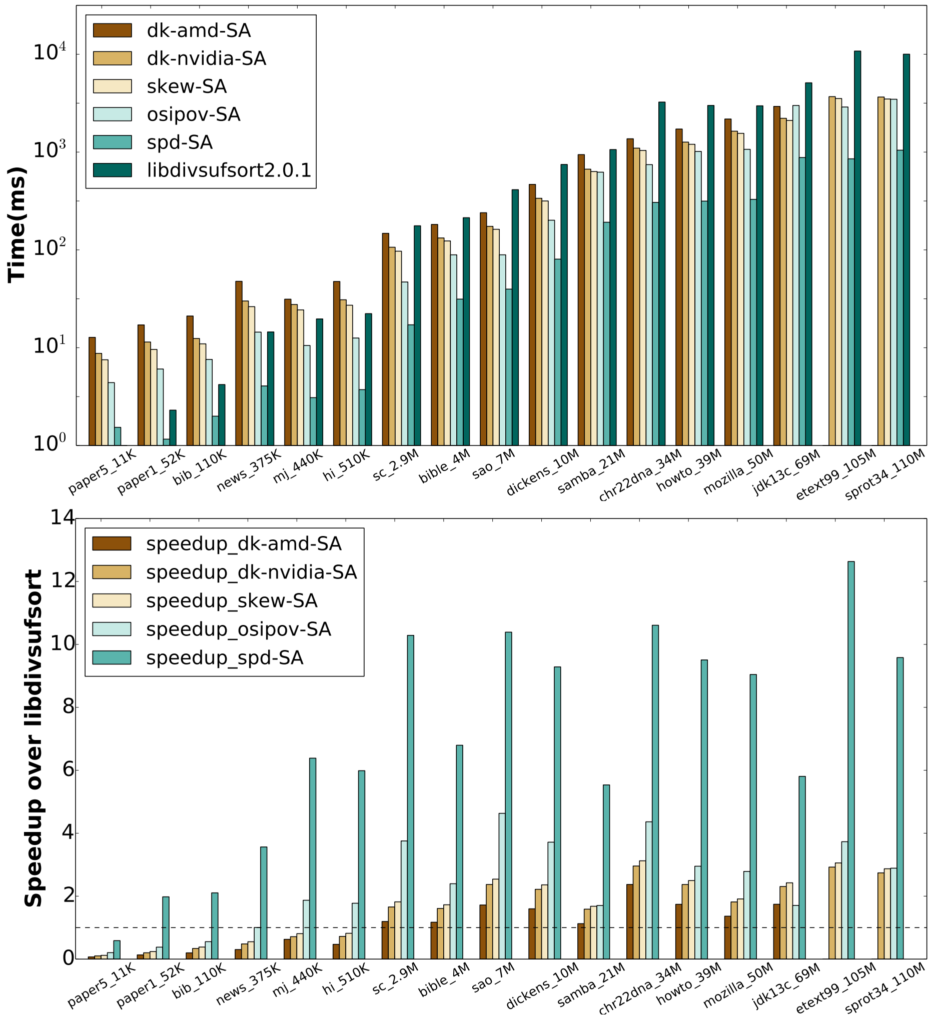
Our experimental setup is an Intel Core i7-3770K 3.5 GHz 4-core machine with 16 GB RAM and 8 MB L3 cache. We used an NVIDIA Tesla K20c GPU (launch date: November 2012; process: 28 nm; peak single-precision floating-point throughput: 3.524 TFLOPS; peak memory bandwidth: 208 GB/s). Deo and Keely's OpenCL implementation was on an AMD Radeon 7970 GPU (launch date: December 2011; process: 28 nm; peak single-precision floating-point throughput: 3.789 TFLOPS; peak memory bandwidth: 264 GB/s). The AMD GPU has slight peak performance advantages over the NVIDIA GPU we used, but despite differences in programming environment and GPU architecture, we believe results from the two GPUs are directly comparable. We compiled and ran dk-nvidia-SA, skew-SA, spd-SA, and osipov-SA using CUDA 6.0 and Visual Studio 2010 on 64-bit Windows 7.

For evaluation, we use the same input datasets as Deo and Keely along with two larger datasets. The input strings range in size from 10 KB to 110 MB and are collected from the Calgary Corpus, Large Canterbury Corpus, Manzinis Corpus, Protein Corpus, and Silesia Corpus [13]. We compare the four GPU implementation results against Mori's highly tuned, OpenMP-assisted CPU implementation libdivsufsort 2.0.1 [13] based on induced copying on a 4-core PC, using its own internal runtime measurement, which excludes disk access time.

Fig. 4 summarize our performance results and we make the following observations:

- On datasets of sufficient size (on the order of 1 MB for the skew implementations, smaller for spd-SA), all four GPU implementations are faster than the CPU baseline. Roughly speaking, the skew implementations are twice as fast as the CPU version, osipov-SA has a 4× speedup, and spd-SA's speedup ranges from 6× to 11×.
- Macroscopically, the fluctuations in the speedups of spd-SA and osipov-SA for the same datasets suggest that the behavior of our hybrid prefix-doubling spd-SA is similar to that of osipov-SA, and our skew-SA with dk-amd-SA and dk-nvidia-SA.
- spd-SA is 2.3× to 4.4× faster than osipov-SA.
- skew-SA is consistently 1.45× faster than dk-amd-SA and 1.1× faster than dk-nvidia-SA.
- Both prefix-doubling based GPU implementations outperform the three skew based methods on most datasets.

In general the performance of the GPU implementations track each other. The datasets with the highest speedups on GPUs are those with a non-uniform prefix distribution (e.g., *chr22dna*, which contains DNA sequences of only 4 different characters), whereas more uniformly distributed prefixes yield smaller speedups. The peaks of the speedup happen because the GPU implementations



Dataset mozilla_50M	dk-amd-SA	dk-nvidia-SA	skew-SA	osipov-SA	spd-SA
Runtime (ms)	2178.5	1634.9	1553.9	1066.7	328.6
Throughput (MB/s)	23	30.6	32.2	46.9	152.2

Fig. 4. Runtimes (top) of five suffix array construction implementations over corpus datasets; the datasets are those chosen by Deo and Keely [2] in addition to two larger datasets for which we have no dk-amd-SA measurements. The CPU implementation libdivsufsort is the baseline for speedup comparisons (bottom). The five GPU implementations are dk-amd-SA, dk-nvidia-SA, osipov-SA, skew-SA, and spd-SA.

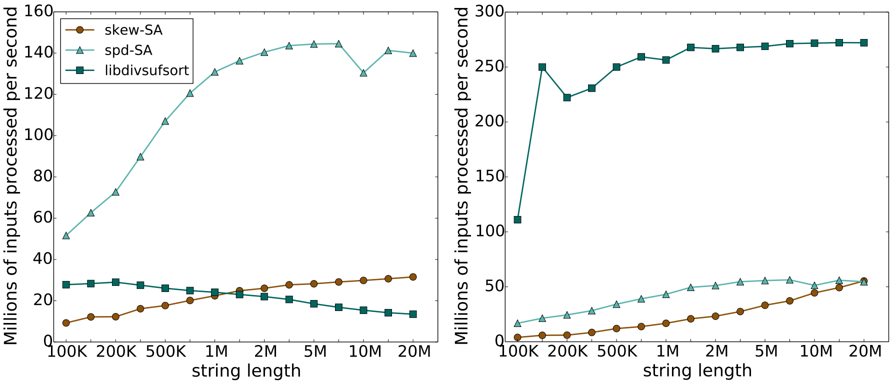


Fig. 5. Left, throughput on plain text "enwik8" as dataset size scales; right, throughput on a dataset consisting only of the repeated letter 'A', using the same legend as left graph.

(especially our hybrid skew/prefix-doubling) are faster on non-uniform prefixes. For skew, a more uniform dataset results in more iterations in the recursive step, and thus takes longer time. For prefix-doubling, uniform datasets give us fewer segments for separation and thus result in less parallelism.

We take a closer look at two datasets for skew-SA and spd-SA. The first is a scalability test on increasing amounts of text data from the English Wikipedia dump "enwik8"[9], shown in Fig. 5 at left. In general, the larger the dataset, the higher the throughput; it takes an input size of many millions of characters for both approaches to reach the throughput asymptote. At 10 MB, skew-SA has a 2× speedup and spd-SA a 9× speedup over libdivsufsort.

The second is an artificial dataset composed of only the repeated single character 'A'. This is a pathologically bad case for prefix-doubling because (except for suffixes near the end of the string) every input position has an identical prefix on every iteration until the last one, so spd-SA cannot divide the prefixes into multiple segments—they all land in the same segment. Moreover, because those prefixes in that segment are lexicographically identical, they have worst-case sorting behavior. Skew's performance is much more predictable; although skew must recurse all the way to the base case and cannot finish early, it is not pathologically bad as with prefix-doubling. Nonetheless, except for very large inputs, spd-SA's performance still exceeds skew-SA's. Induced copying is much better suited for this dataset. For a 10 MB all-'A' input, libdivsufsort completes in 40 ms, compared with 224 ms for skew-SA and 196 ms for spd-SA.

Application tools implementation The most recent release of the CUDA Data Parallel Primitives (CUDPP) Library[10] uses our optimized skew implementation in its parallel Burrows-Wheeler Transform (BWT) [17] and bzip2 data com-

[9] `http://cs.fit.edu/~mmahoney/compression/textdata.html`

[10] `http://cudpp.github.io/`

pression functions. As predicted, both gain significant speedups from replacing string sort with a suffix array algorithm [3]. We also implement our fast hybrid skew/prefix-doubling in a parallel BWT and use it as a partial step in implementing parallel FM index backward_search, along with CUB's DeviceHistogram routine and cudppMultiscan from CUDPP 2.2. We measured the performance of the parallel BWT and FM index based on our fast hybrid skew/prefix-doubling on "enwik8" and "chr22dna" datasets and show our results in Table 2.

Table 2. Throughput of the BWT and FM index's backward_search using our spd-SA.

Dataset	enwik8	chr22.dna
BWT (Millions of characters/s)	132.5	116.4
FM index (Millions of characters/s)	28.6	77

5 Conclusions

Much of the interesting work in GPU computing has been the result of brute-force techniques, judiciously applied. Often, GPU computing practitioners have found that the loss of efficiency by using brute force is more than offset by the performance advantages of the GPU. Of the three classes of suffix array construction algorithms, skew is perhaps the most suitable for brute-force methods, and was chosen by Deo and Keely, and ourselves when we began our work.

However, the maturation of GPU computing is leading to the development of elegant, efficient, load-balanced algorithmic building blocks that are designed for, and run well on the GPU. The merge and segmented sort implementations in this paper make the difference between an SACA that is uncompetitive vs. an SACA that is best in class. We expect that the next frontier in GPU SACAs will be tackling the third class of SACAs—induced copying. The research challenge is to determine whether the inherent algorithmic efficiency of their CPU implementation will translate into the GPU domain.

Acknowledgments

We thank to Yangzihao Wang for the initial implementation and good advice along the way. We would like to acknowledge Mrinal Deo for providing their paper's original data, Vitaly Osipov for sharing his paper's source code for comparision, and both Jason Mak and Carl Yang for feedback on early drafts of the paper. We appreciate the funding support of the National Science Foundation under grants OCI-1032859 and CCF-1017399, and UC Lab Fees Research Program Award 12-LR-238449.

References

1. Davidson, A., Tarjan, D., Garland, M., Owens, J.D.: Efficient parallel merge sort for fixed and variable length keys. In: Proceedings of Innovative Parallel Computing. InPar '12 (May 2012)
2. Deo, M., Keely, S.: 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. pp. 197–206. PPoPP '13 (Feb 2013)
3. Edwards, J.A., Vishkin, U.: Parallel algorithms for Burrows-Wheeler compression and decompression. Theoretical Computer Science 525, 10–22 (13 Mar 2014)
4. Ferragina, P., Manzini, G.: Opportunistic data structures with applications. In: Proceedings of the 41st Annual Symposium on Foundations of Computer Science. pp. 390–398. FOCS 2000 (Nov 2000)
5. Green, O., McColl, R., Bader, D.A.: GPU merge path: A GPU merging algorithm. In: Proceedings of the 26th ACM International Conference on Supercomputing. pp. 331–340. ICS '12 (2012)
6. Kärkkäinen, J., Sanders, P.: Simple linear work suffix array construction. In: Proceedings of the 30th International Conference on Automata, Languages and Programming. pp. 943–955. ICALP'03, Springer-Verlag, Berlin, Heidelberg (2003), http://dl.acm.org/citation.cfm?id=1759210.1759301
7. Karp, R.M., Miller, R.E., Rosenberg, A.L.: Rapid identification of repeated patterns in strings, trees and arrays. In: Proceedings of the Fourth Annual ACM Symposium on Theory of Computing. pp. 125–136. STOC '72 (May 1972)
8. Larsson, N.J., Sadakane, K.: Faster suffix sorting. Theoretical Computer Science 387(3), 258–272 (2007)
9. Lindholm, E., Nickolls, J., Oberman, S., Montrym, J.: NVIDIA Tesla: A unified graphics and computing architecture. IEEE Micro 28(2), 39–55 (Mar/Apr 2008)
10. Liu, C.M., Luo, R., Lam, T.W.: GPU-accelerated BWT construction for large collection of short reads. arXiv preprint arXiv:1401.7457 (2014)
11. Manber, U., Myers, G.: Suffix arrays: A new method for on-line string searches. In: Proceedings of the First Annual ACM-SIAM Symposium on Discrete Algorithms. pp. 319–327. SODA '90 (Jan 1990)
12. Merrill, D., Grimshaw, A.: Revisiting sorting for GPGPU stream architectures. Tech. Rep. CS2010-03, Department of Computer Science, University of Virginia (Feb 2010)
13. Mori, Y.: libdivsufsort, version 2.0.1. https://code.google.com/p/libdivsufsort/wiki/SACA_Benchmarks (2010)
14. Nickolls, J., Buck, I., Garland, M., Skadron, K.: Scalable parallel programming with CUDA. ACM Queue pp. 40–53 (Mar/Apr 2008)
15. Osipov, V.: Parallel suffix array construction for shared memory architectures. In: Proceedings of the 19th International Conference on String Processing and Information Retrieval. pp. 379–384. SPIRE'12, Springer-Verlag (2012)
16. Pantaleoni, J.: A massively parallel algorithm for constructing the BWT of large string sets. arXiv.org abs/1410.0562(1410.0562v1) (Oct 2014)
17. Patel, R.A., Zhang, Y., Mak, J., Owens, J.D.: Parallel lossless data compression on the GPU. In: Proceedings of Innovative Parallel Computing (May 2012)
18. Satish, N., Harris, M., Garland, M.: Designing efficient sorting algorithms for manycore GPUs. In: Proceedings of the 23rd IEEE International Parallel and Distributed Processing Symposium (May 2009)