Title

Fast parallel skew and prefix-doubling suffix array construction on the GPU

Journal

Concurrency and Computation: Practice and Experience, 28(12)

ISSN

15320626

Authors

Wang, Leyuan
Baxter, Sean
Owens, John D

Publication Date

2016-08-25

DOI

10.1002/cpe.3867

Peer reviewed

CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2015; **00**:1–20
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/cpe

Fast Parallel Skew and Prefix-Doubling Suffix Array Construction on the GPU

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

[1] *University of California, Davis, Davis, CA, USA*
[2] *D. E. Shaw Research, New York, NY, USA*

SUMMARY

Suffix arrays are fundamental full-text index data structures of importance to a broad spectrum of applications in such fields as bioinformatics, Burrows-Wheeler Transform (BWT)-based lossless data compression, and information retrieval. In this work, we propose and implement two massively parallel approaches on the GPU based on two classes of suffix array construction algorithms. The first, parallel *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 theoretically analyze the two formulations of suffix array construction algorithms and show performance comparisons on a large variety of practical inputs. We conclude that, with the novel use of our efficient segmented sort, *prefix-doubling* is more competitive than *skew* on the GPU. We also demonstrate the effectiveness of our methods in our implementations of the Burrows-Wheeler transform and in a parallel FM-index for pattern searching.

Received . . .

KEY WORDS: suffix array; parallel; GPU; skew; prefix-doubling; segmented sort

1. INTRODUCTION

The suffix array (SA) of a string is the lexicographically sorted set of all suffixes of the string. The suffix array was originally designed as a space- and cache-efficient alternative to suffix trees; Manber and Myers [1] introduced the suffix array and its first construction algorithm in 1990. The SA data structure is used in a variety of applications, including string processing ("stringology"), computational biology, text indexing, and many more.

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 requires $\mathcal{O}(n \log n)$ suffix comparisons; each of those suffix comparisons have a time complexity of $\mathcal{O}(n)$. Altogether, then, the total time needed is $\mathcal{O}(n^2 \log n)$. To develop a more efficient algorithm, we leverage the insight that suffixes are not arbitrary strings but instead are 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.

- *Prefix-doubling* sorts the suffixes of a string by their prefixes, doubling the length of those prefixes every iteration. Prefix-doubling was originally proposed by Karp et al. [2], first

*Correspondence to: Leyuan Wang, Department of Computer Science, University of California, Davis, 1 Shields Ave, Davis, CA 95616, USA.
†E-mail: leywang@ucdavis.edu

applied to suffix array construction by Manber and Myers [1] (MM), and later optimized by Larsson and Sadakane [3] (LS). The advantage of LS over MM is that LS does not scan the fully sorted suffixes generated in the previous pass.
- *Recursive* SACA approaches recursively sort a subset of the suffixes, use the order of the sorted subset to infer the order of the remaining subset, and finally merge the two sorted subsets to get the order of the entire set. The popular "skew" algorithm proposed by Kärkkäinen and Sanders [4] is a linear-time instance of the recursive approach.
- *Induced copying* is a non-recursive approach that uses already-sorted suffixes to quickly induce a complete ordering of all suffixes. Its time complexity, like the recursive approach, is $\mathcal{O}(n)$.

The above approaches have traditionally been implemented on serial processors using serial programming models. The recent rise of highly-parallel commodity processors and large data sizes have spurred interest in efficient parallel implementations of SACAs. Our focus here is on SACAs suitable for highly data-parallel processors such as manycore GPUs and multicore CPUs. These processors, with their high arithmetic capability and memory bandwidth, are potentially well-suited for data-intensive computing tasks such as SACAs. However, designing a SACA that can both take full advantage of the capabilities of a highly-parallel machine as well as avoid the limitations of its programming model is a significant challenge.

In this work, we design, implement, and compare two different parallel formulations of SACAs on NVIDIA GPUs. We make four main contributions.

1. We design and implement a skew-based SACA with several optimizations that yield a speedup of 1.45x over Deo and Keely's GPU-based skew implementation [5].
2. We propose and implement a hybrid non-recursive SACA, incorporating both skew and prefix-doubling formulations, that together overcome the parallelization challenges identified by Deo and Keely. The result outperforms Osipov's GPU-based plain prefix-doubling implementation [6].
3. We compare our two implementations both theoretically and experimentally, then revisit Deo and Keely's conclusions on the most appropriate formulation for parallel SACAs. We demonstrate that a prefix-doubling based formulation can be efficiently mapped to GPUs and that our hybrid approach in practice produces the fastest SACA implementation on GPUs. Our speedups over previous work are as high as 12.76× over the skew implementation of Deo and Keely, 4.4× over the prefix-doubling implementation of Osipov, and 7.9× over our own optimized skew implementation, measured over a variety of real-world datasets.
4. We integrate our highest-performance implementation—our skew/prefix-doubling hybrid—into our GPU implementations of the Burrows-Wheeler transform (BWT) and an FM-index-based pattern search application.

We proceed to Section 2, where we give a short review of the current state of the art. We summarize the background concepts in Section 3, then demonstrate our two fast parallel SACAs in Section 4. These are the core algorithmic contributions of the paper and also include our algorithmic comparisons with Osipov's and Deo and Keely's implementations and a theoretical discussion on which SACA is the best fit for the GPU. Next, in Section 5, we compare our experimental results and implementations to the previous state of the art and analyze the results. Finally, we conclude in Section 6.

This paper is a significantly extended and improved version of our previous work [7].

2. RELATED WORK

Distributed-memory suffix array construction approaches One of the earliest parallel distributed-memory SACAs was the Futamura-Aluru-Kurtz (*FAK*) algorithm [8]. FAK exposes coarse-grained parallelism by (1) partitioning the suffixes into buckets according to their first w characters, (2) allocating buckets to individual processors, and then (3) sorting each bucket locally

DOI: 10.1002/cpe

using the sequential multikey quicksort of Bentley and Sedgewick [9]. This approach has limited parallelism, however; it is also task-parallel rather than data-parallel and would thus be more complex to map to GPUs. Finally, it allocates a different amount of work to each bucket, leading to load imbalance across processors. Together these issues make FAK a poor match for GPUs, but for more coarse-grained parallel machines, it is a potentially better fit: for instance, Abdelhadi et al. [10] recently described an MPI-based implementation of the *FAK* algorithm on cloud-based computer clusters.

Kulla and Sanders [11] proposed another parallel distributed-memory SACA, based on the linear-time skew approach of Kärkkäinen and Sanders [4]. However, their implementation has a higher time complexity than skew: it requires a comparison-based sample sort in each recursive iteration. A second disadvantage of this approach is the additional memory consumption required when recursively decomposing the string.

Flick and Aluru [12]'s more recent parallel distributed-memory SACA has a similar approach to LS. Their approach has a better worst-case run-time bound of $\mathcal{O}(T_{\text{sort}}(n,p)\log n)$, where T_{sort} is the runtime for parallel sorting. Their parallel MPI-based implementation achieves superior practical performance on human genome datasets.

Shared-memory suffix array construction approaches Some parallel SACAs leverage shared memory. Homann et al. [13] introduced the *mkESA* tool on multithreaded CPUs, which is a parallelized version of the "Deep-Shallow" algorithm of Manzini and Ferragina [14]. The authors report a speedup of less than 2x using 16 threads. Mohamed and Abouelhoda [15] proposed a parallelized variant of the bucket pointer refinement (*bpr*) algorithm of Schürmann and Stoye [16] on multicore architectures, leveraging shared memory. The authors claim that their implementation beats *mkESA*. But their results still show poor scalability (less than 1.7x speedup using 8 threads).

Shun's Problem Based Benchmark Suite (*PBBS*) [17] leverages the task-parallel Cilk Plus programming model in its parallel multicore skew implementation. *PBBS*[†] includes two parallel implementations of suffix array construction: *parallelKS*, which is skew-based and *parallelRange*, which is a parallel version of prefix-doubling. The authors claim that both of them are faster than the current fastest CPU implementation of suffix array by Yuta Mori [32] and *parallelRange* is the best one among all CPU implementations of suffix array. So we compare our results with these two implementations in Section 5.

GPU suffix array construction approaches Osipov [6] and Deo and Keely [5] have done seminal work on developing highly parallel, shared-memory GPU algorithms for suffix array construction. Deo and Keely analyze the three SACA classes that we enumerated in Section 1; they note that induced copying has numerous data dependencies which are challenging to address with a parallel approach. Of the two remaining classes, Deo and Keely conclude that skew is best suited for the GPU: they note that all of skew's 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. In contrast, Osipov concludes that prefix-doubling algorithms are a better fit than skew for the GPU, noting that prefix-doubling only requires fast GPU radix sorting of (32-bit key, 32-bit value) pairs, while skew needs a more expensive sort on large tuples (for instance, by comparison-based sorting and merging). Deo and Keely implemented and tuned a parallel variant of skew on GPUs; Osipov demonstrated a GPU implementation based on LS.

In more recent work, Pantaleoni [18] and Liu et al. [19] both proposed scalable, space-efficient methods targeting bioinformatics applications. They leverage the GPU's fast sorting capabilities to address workloads consisting of large collections of relatively short DNA strings. Pantaleoni's approach focuses on prefix-doubling; Liu et al. use Kärkkäinen's blockwise suffix sorting method [20].

[†] http://www.cs.cmu.edu/~pbbs/index.html

DOI: 10.1002/cpe

Table I. 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

3. BACKGROUND & PRELIMINARIES

We begin with the algorithmic background for the string algorithms that we have implemented. Section 3.1 provides the notation for suffix array construction that we use throughout the paper. Readers already familiar with the Burrows-Wheeler Transform (Section 3.2), the FM index (Section 3.3), the three classes of SACAs (Section 3.4), and GPU terminology (Section 3.5) can skip to Section 4.

3.1. 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 $j \bmod 3 \equiv 0$ be S_0, and suffixes with starting position k where $k \bmod 3 \equiv 1$ be S_1.

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

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

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 $\text{SA}_{[12]}$ and that of S_0 as $\text{SA}_{[0]}$, correspondingly for the inverse suffix array, $\text{ISA}_{[12]}$ and $\text{ISA}_{[0]}$, and we denote the lexicographic ranks of S_1 by $\text{ISA}_{[1]}$.

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 a corresponding inverse suffix array ISA_h.

3.2. 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:

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

DOI: 10.1002/cpe

Table II. C[c] and Occ(c, k) of "annb$aa".

c	$	a	b	n
C[c]	0	1	4	5

	a	n	n	b	$	a	a
	1	2	3	4	5	6	7
$	0	0	0	0	1	1	1
a	1	1	1	1	1	2	3
b	0	0	0	1	1	1	1
n	0	1	2	2	2	2	2

```
1  backward_search(P[1,p])
2  i ←p, c ←P[p], First ←C[c], Last ←C[c+1]
3  while (First ≤Last && (i ≥2)) do
4      c ←P[i−1]
5      First ←C[c]+Occ(c,First−1)+1
6      Last ←C[c]+Occ(c,Last)
7      i ←i−1
8  if (Last < First)
9      return "no rows prefixed by P[1,p]"
10 else
11     return <First,Last>
```

Figure 1. Counting the number of occurrences

	F						L	SA
1	**$**	b	a	n	a	n	**a**	6
2	**a**	$	b	a	n	a	**n**	5
3	**a**	n	a	$	b	a	**n**	3
4	**a**	n	a	n	a	$	**b**	1
5	**b**	a	n	a	n	a	**$**	0
6	**n**	a	$	b	a	n	**a**	4
7	**n**	a	n	a	$	b	**a**	2

Figure 2. Determine positions

Table I shows an example of the SA, ISA and BWT of the input string "banana$".

3.3. The FM index

Proposed by Ferragina and Manzini [21], 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 counts how many times a pattern occurs in BWT-compressed text and determines the locations of the occurrences without decompressing it. Their algorithm leverages two nice properties of the suffix array: (1) all the suffixes prefixed by a pattern P occupy a contiguous portion of the suffix array, denoted as a subarray; (2) the subarray has a range [First, Last] where First is actually the lexicographic position of P among all the ordered suffixes.

LF (last-to-first) mapping is a key to their fast pattern searching technique. Taking the Burrows-Wheeler transform (BWT) of an input text—say, "banana$"—results in a matrix composed of sorted cyclic permutations of the string, as shown in Figure 2. The result "annb$aa" corresponds to the last column of the matrix. If we label the first column of the sorted permutation matrix as an array F and the last column as an array L, a last-to-first column mapping LF(i) from a character L[i] to F[j] can be defined, with the help of two tables C[c] and Occ(c, k), where C[c] is the number of occurrences of the characters lexically smaller than c in L and Occ(c, k) is the number of occurrences of character c in the prefix L[1..k]. An example of C[c] and Occ(c, k) for the input string "banana$" is shown in Table II. The last-to-first mapping is then defined as follows:

$$\text{LF}(i) = \text{C}[\text{L}[i]] + \text{Occ}(\text{L}[i], i).$$

The algorithm is called backward_search because it searches pattern P from the last character to the first. LF mapping is repeatedly applied to find the range of rows prefixed by successively longer suffixes of P until the range becomes empty, in which case P does not occur in the orginal input text, or until we run out of suffixes, where the size of the range represents the number of occurrences. Figure 1 shows the pseudocode for counting the occurrences of pattern P. Using

DOI: 10.1002/cpe

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
m	m	i	i	s	s	i	i	s	s	i	i	p	p	i	i	$
0 ($)	1 (i)								2 (m)		3 (p)		4 (s)			
17	3	4	7	8	11	12	15	16	1	2	13	14	5	6	9	10
0	1	2				3	4		5	6	7	8	9		10	
17	16	3	7	11	15	12	4	8	2	1	14	13	6	10	5	9
0	1	2	3	4		5	6		7	8	9	10	11		12	
17	16	15	11	3	7	12	4	8	2	1	14	13	6	10	5	9
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
17	16	15	11	7	3	12	8	4	2	1	14	13	10	6	9	5

Figure 3. Prefix-doubling example for the input string "mmiissiissiippii$" with each suffix's index above (line 1). First we group the index of suffixes into five buckets based on their first characters and give a rank (ISA) to each bucket in lexicographic order (line 2). Suffixes that are singletons in a bucket are fully sorted and don't need to be considered in the following steps. In the next step, we compare the rank of *suffix* $i+1$ (ISA$[i+1]$) to decide the order of *suffix* i, i.e., for bucket 1 in line 2, to sort *suffix* 3 and *suffix* 4, we compare ISA[4] which is 1 with ISA[5] which is 8, and get *suffix* 3 comes before *suffix* 4 (line 3). Following the same routine, in step three, we compare ISA$[i+2]$ of not fully sorted suffixes (line 4), and in step four, we compare ISA$[i+4]$ to generate the final result where all suffixes are in independent buckets (line 5).

Table II's representation of C and Occ with the input example of P="ana", Figure 1 results in an output range of $[3, 4]$. Given the suffix array as shown in Figure 2, we can directly locate the postions of P's occurrences in the original string "banana$"—SA[2]=3 and SA[3]=1. We refer the reader to the original paper [21] for further detail.

We implement backward_search on the GPU by mapping the algorithm into massively parallel primitives. We construct C and Occ using the DeviceHistogram kernel from Merrill's *CUB* Library[‡] and the cudppMultiscan kernel from CUDA Data Parallel Primitives (*CUDPP*) Library[§]. Instead of storing a whole suffix array, we use a sampled suffix array to save space and still manage to find the locations of occurrences in constant time. Our experimental results are shown in Section 5.

3.4. *Suffix Array Construction Algorithms*

For each of the three classes of SACAs, we outline the main ideas of the algorithm and the conclusions of Deo and Keely with respect to its parallelization.

3.4.1. Prefix-Doubling Algorithms Prefix-doubling sorts prefixes of the input string to find the lexicographic order of suffixes. The first step of the algorithm is a fast single-character sort. On each successive step, the length of the prefix is doubled, and the prefixes are sorted again; the key idea is that given an h-order of suffixes (suffixes are already sorted by their h-length prefixes), we can deduce their $2h$-order in linear time. Consider any two suffixes, say *suffix* i and *suffix* j, with identical h-length prefixes. We can deduce their order according to their $2h$-length prefixes by using the current relative order of *suffix* $i+h$ and *suffix* $j+h$, which is already known according to the h-order. After applying this routine to all suffixes of the h-order, we can get the $2h$-order of suffixes. Once a prefix is unique, its position in the sorted prefix list is fixed, and its associated suffix has its final rank. Since the longest suffix has size n, all suffixes will be sorted in at most $\mathcal{O}(\log n)$ iterations. A running example is shown in Figure 3.

Manber and Myers's approach [1] (MM) induces the $2h$-order of suffixes from the h-order by scannning SA_h. During the scan, MM needs random accesses to ISA$[i+h]$ for each *suffix* i it

[‡] http://nvlabs.github.io/cub/
[§] http://cudpp.github.io/

DOI: 10.1002/cpe

encounters in non-local order. In order to correctly place the suffixes to their respective h-groups, the scan has to be performed in linear order. Groups in singletons still need to be scanned in MM until all suffixes are fully sorted. The above three characteristics make MM space- and work-inefficient. Larsson and Sadakane [3] (LS) choose to use ternary split quicksort (TSQS) to construct SA_1. They sort each h-group independently using $ISA[i + h]$ as the key for each *suffix* i in the h-group. The ISA stores the h-group rank during the construction process. In each iteration, they identify positions that are singletons and only process non-singleton h-groups, avoiding a large amount of redundant work compared to MM.

The performance of prefix-doubling is clearly dependent on the performance of sorting. Let's consider a brute-force parallel approach to suffix array construction: doing a full sort of all prefixes. The dominant sorting algorithm on the GPU is a radix sort [22], but its cost is proportional to key length, and with long keys (prefixes), the cost of a radix sort is unviable. The alternative is a comparison-based sort [23], but it would also not deliver competitive performance. Whether serial or parallel, for an efficient prefix-doubling implementation, we cannot afford the cost of a full sort on prefixes of any significant length; instead we must take advantage of partial prefix orders from previous iterations in sorting the current set of prefixes.

We can think of each iteration as producing a set of buckets that are dependent on the prefixes considered in that iteration. Suffixes in the same bucket have an indeterminate order (their prefixes are identical), but the buckets themselves are ordered: once two suffixes are placed into different buckets, their order is fixed by the order of their buckets. An efficient prefix-doubling implementation takes advantage of the ordering of buckets from previous iterations. Each successive iteration considers a longer set of prefixes and may refine each bucket into multiple buckets. Deo and Keely [5] conclude that prefix-doubling is unsuitable for data-parallel architectures: the number of buckets and the amount of work per bucket is irregular and data-dependent, and thus difficult to parallelize. While they turn to a different formulation, we successfully build an efficient prefix-doubling implementation, which we describe in Section 4.2.

3.4.2. Recursive algorithms The skew algorithm is one of the recursive algorithms that runs in worst-case linear time [24]. Recursive SACAs have three steps: (1) choose and recursively sort a subset (typically 2/3 or fewer) of the suffixes; (2) use the order of the sorted subset to infer the order of the remaining subset; and (3) merge the two sorted subsets to get the order of the entire set. Different SACAs make different choices in step (1) in terms of selecting a subset. The key to the entire SACA's linear-time complexity is ensuring that step (2) is cheap. Step (3) requires an efficient merge operation.

The recursion in step (1) is among the most challenging to parallelize; recursion is often problematic with the GPU programming model. Here, we need to check at the end of each step if the sequence is fully sorted and continue to recurse if it is not. Deo and Keely [5] note that this is the only operation that requires communication between the CPU and GPU, and that the other operations map well to a data-parallel machine. They characterize skew as the most suitable and scalable algorithm for GPUs.

3.4.3. Induced Copying Induced copying SACAs are non-recursive and use already-sorted suffixes to quickly induce a complete order of the remaining suffixes. Like the recursive formulation, their time complexity is $\mathcal{O}(n)$. Induced copying SACAs are currently the fastest CPU implementations and we compare our results with the best one, *libdivsufsort*. To illustrate this approach, we describe the method of Itoh and Tanaka [25], *two stage induced copying*.

First they define suffix i of a string x to be type A or type B as follows:

$$\text{type A: } x[i] > x[i+1]$$
$$\text{type B: } x[i] \leq x[i+1]$$

Itoh and Tanaka's insight is that once type B suffixes are sorted, it is straightforward to find the order of type A suffixes. Their algorithm has three steps:

DOI: 10.1002/cpe

1. Initial bucket sort. This step assigns each suffix to a bucket according to its first character. Now, consider all suffixes associated with a single bucket (they all have the same first character). When properly sorted, all type A suffixes in that bucket (whose second characters are smaller than their first characters) would come before all type B suffixes (whose second characters are larger). In this step, we only store the indexes of type B suffixes into the bucket (we know they will be at the end of the bucket).
2. Sort each bucket's suffixes with a string sort algorithm. Note that buckets at this point only contain type B suffixes. Other induced copying SACAs try to improve the sorting method, but this is still the most time-consuming step. After this step, type B suffixes are in their final positions.
3. Induce the positions of the type A suffixes and complete the SA. We walk through the partially completed suffix array from left to right, and for each sorted *suffix* i we encounter, if *suffix* $i-1$ is an A type suffix that has not been placed in the SA, we assign *suffix* $i-1$ to the leftmost empty position in its bucket in the SA. All type A suffixes that fall into their corresponding buckets during the left-to-right scan will always appear in sorted order, as each *suffix* i is already in its final position in the SA when the scan reaches position i.

Deo and Keely conclude that there are many data dependencies in this class of SACAs and the most time-consuming parts are either serial in nature or hard to parallelize on GPUs because of the need for communication between compute units. In general, the first step can be easily parallelized and should be straightforward to map to GPUs. The second step is a challenge for the same reason as the bucket sort in the prefix-doubling SACAs: the irregular number of buckets and sizes of each bucket. The final step traverses an array of suffixes from left to right, making changes along the way. Changes at one point affect decisions made later in the array in the same iteration. This step is inherently sequential and is different from the sorting scheme of the previous two classes of SACAs that partition data at each step, exposing great amounts of parallelism. Both of the previous two classes of SACAs are basically like building a radix tree that create smaller and smaller partitions. But with induced copying, the partitioning outcomes are much less clear. While suffixes are essentially sorted like a radix tree (modulo the A/B type classification), they are not put into independent partitions, because the pointer that is doing the sweep can modify data in any partition at any time. Though we do not explore an induced-copying SACA in this work, it remains intriguing future work.

3.5. The Graphics Processor Unit (GPU)

Today's GPUs target two similar programming models for general-purpose programmability, CUDA [26] (developed by NVIDIA) and OpenCL [27] (managed by Khronos). In the following discussion we primarily use NVIDIA CUDA terminology and where appropriate, add OpenCL terms in parentheses.

The modern GPU is a massively parallel processor that supports tens of thousands of hardware-scheduled threads running simultaneously. These threads are organized into blocks (work-groups) and the hardware schedules blocks of threads onto hardware cores (CUDA: streaming multiprocessors, OpenCL: compute units). High-end NVIDIA GPUs have on the order of 16 cores, each of which contains 32-wide SIMD (single-instruction, multiple-data) processors (CUDA: CUDA cores, OpenCL: SIMD units) that run 32 threads (work-items) in lockstep. GPUs also feature a memory hierarchy of per-thread registers, per-block shared memory (per-work-group local 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. [28] provides more detail on modern GPU hardware and Nickolls et al. [29] 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

DOI: 10.1002/cpe

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.

4. ALGORITHMS & ANALYSIS

We implement two efficient parallel SACA approaches on the GPU: skew (Section 4.1) and a skew/prefix-doubling hybrid (Section 4.2). We also do a comparison between the two classes of parallel SACAs (Section 4.3).

4.1. Parallel Skew Algorithm

We first implement the skew algorithm based on the method of Kärkkäinen and Sanders [4]. Our CUDA implementation is similar to the OpenCL implementation of Deo and Keely [5], but with several algorithmic optimizations; we compare our skew implementation to Deo and Keely's in Figure 4.

Both of our implementations begin by extracting S_{12} and S_0 from an input string (line 2 of Figure 4), then launching a 3-step least significant digit (LSD) radix sort (from Merrill and Grimshaw's paper [22]). This sort finds the order of S_{12} based on their first triplets (line 3 to line 5). In the first iteration, we initialize each triplet to the first three characters of each S_{12}. Then in the succeeding (recursive) iterations, we use the ranks of the triplets as the value for the key-value sort.

We compute ranks by counting unique triplets. In practice, we do so by first comparing each triplet against its predecessor, storing a flag of 1 whenever they are unequal, and then computing a prefix-sum on 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). However, we make several significant changes to their approach, beginning with the following two memory-bandwidth-centered optimizations:

- While Deo and Keely immediately compute the prefix-sum after computing the flagging list (line 6 left), we only compute the prefix-sum if the suffixes are not fully sorted (line 7 right). With this change, we do not have to compute $\text{SA}_{[12]}$ from $\text{ISA}_{[12]}$ if we are at the end of the recursion and the suffixes are fully sorted (line 10 and 11 left).
- After the recursion, Deo and Keely continue to compute $\text{SA}_{[0]}$ by sorting S_0 with a 2-step LSD radix sort of pairs constructed from (the first character of *suffix* i, rank of *suffix* $i+1$). We note for these pairs, *suffix* $i \in S_0$ and thus *suffix* $i+1 \in S_{12}$, and we know its rank from the previous steps (line 13 and 14 left). Thus we opt for a faster one-step radix sort because the order of the ranks of S_1 (equivalent to $\text{ISA}_{[1]}$) can be filtered out from $\text{ISA}_{[12]}$ (result of line 7 right) using a compact operation (line 11 and 12 right).

The above two optimizations reduce the memory bandwidth required by our implementation: we use only 2/3 of Deo and Keely's memory bandwidth in the recursive part, and 3 fewer memory transactions in the final round.

In the final step (line 14 left), Deo and Keely use Satish et al.'s merge technique [30] and binary search, and leverage the memory locality optimizations of Davidson et al. [23]. Unfortunately, Deo and Keely's implementation suffers from load-imbalance, because it has two differently-sized lists that must be processed independently. Thus in our implementation, we focuse on the merge step that combines the two sorted suffix arrays $\text{SA}_{[0]}$ and $\text{SA}_{[12]}$ to implement it in a load-balanced way. At a high level, we utilize vectorized sorted search to map threads and blocks to equally-sized sections of each partition; this mapping is load-balanced across blocks, permitting a faster implementation better suited for the GPU. We base our method on Green et al.'s Merge Path approach [31]; this *merge* primitive also appears in the second author's *Modern GPU* library[¶].

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

DOI: 10.1002/cpe

```
dk−SA(int* T, int* SA, int length)
Initialize Mod12() // form triplets 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)
    dk−SA() // Recurse
    storeUniqueRanks()
else
    computeSAFromUniqueRank()
RadixSort(ISA[1])
RadixSort(s0)
Merge(s0,s12)
```

```
skew−SA(int* T, int* SA, int length)
Initialize Mod12() // form triplets 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)
    lexicRankOfTriplets(s12)
    skew−SA() // Recurse
    storeUniqueRanks()
Compact(ISA[12]) // compact out the order
    of ISA[1]
RadixSort(s0)
Merge(s0,s12)
```

Figure 4. Left, Deo and Keely's skew implementation pseudocode; right, ours.

Let's take a closer look at our merge implementation. Deo and Keely's approach does not have a uniform chunk size assigned to each block and thus suffers from load imbalance. So our first challenge is load-balancing: when we divide up the two sorted inputs into chunks and distribute them to thread blocks, we must ensure that those chunks have uniform size. Our second challenge is memory coalescing: the output of each chunk will ideally occupy contiguous, coherent regions in memory. We can address both challenges by judiciously choosing the split points between different chunks in the two sorted inputs. The obvious way to choose these split points would require a two-dimensional search across both input arrays, but the Merge Path approach describes a simpler technique: it transforms the expensive two-dimensional search into a simpler one-dimensional search along a diagonal that connects the two input arrays.

Our merge implementation performs a two-part, hierarchical split by leveraging Merge Path partitioning and coarse-grained searching: we first divide the entire input into equal-sized tiles that can be assigned to blocks, then divide 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 Green et al.'s original implementation [31].

4.2. Parallel Skew/Prefix-doubling

Deo and Keely concluded that skew was the best approach for GPU implementation; their work, the first implementation of linear-time skew for suffix array construction, certainly demonstrated that the skew approach was viable and could deliver good performance. However, we note two significant disadvantages for skew on GPUs:

1. Because skew is inherently recursive, we cannot parallelize across iterations. This restricts our parallelism opportunities.
2. At the end of each iteration, some sets of triplets may be fully sorted. However, to maintain the recursive formulation of algorithm, we cannot simply declare that these fully sorted suffixes are complete and leave them out of further iterations; instead we must (re-)process them on every iteration. As a result, we must perform a large amount of redundant work.

These two disadvantages keep us from fully exploiting the compute capabilities of our GPUs. Thus we consider a different strategy: a novel hybrid combination of skew and prefix-doubling. We demonstrate that we can achieve better performance with this hybrid and conclude that it is a better fit for modern GPU architectures.

From our skew implementation, we keep two components: (1) the first step of skew, which reduces the string size by a factor of 2/3, and (2) the final skew merge stage, which is trivial. These two components bracket our prefix-doubling implementation; after the first step of skew, we transition

DOI: 10.1002/cpe

```
1  osipov−SA(int∗ T, int∗ SA, int length)
2  Initialize SA_4 // sort suffixes by their first 4
       chars
3  Initialize ISA_4[i] //4−rank of the head of i's
       4−group in SA_4
4  size ←n, h ←4
5  while (size > 0)
6      Scan SA_h and generate tuples
           (SA_h[j] − h, ISA_h[SA_h[h] − h],
           ISA_h[SA_h[j]])
7      RadixSort tupples by 2nd component
           stably //contains SA_2h
8  Refine h−heads of h−groups // Re−rank
9  Update ISA_2h // contains ISA_2h
10 Filter and Compact SA_2h
11 size ←sizeof(SA_2h)
12 h ←2h
```

```
1  spd−SA(int∗ T, int∗ SA, int length)
2  Initialize Mod12() // form triplets s12, s0
3  RadixSort(s12) // 25−bit radix sort on
       triplet s12
4  ComputeRanks ISA_[12]
5  size ←2/3 n, h ←6
6  while (size > 0)
7      SegmentedSort
           (ISA[SA[i]+h],ISA[SA[i]+2h])
8      Update ISA_2h and Compact SA_2h
9      size ←sizeof(SA_2h)
10     h ←2h
11 Compact(ISA_[12]) // compact out the order
       of ISA_[1]
12 RadixSort(s0)
13 Merge(s0,s12)
```

Figure 5. Left, Osipov's parallel prefix-doubling description; right, our skew/prefix-doubling.

to our non-recursive better-performing prefix-doubling implementation, and when it is done, we return to the final step of skew.

We begin with skew's first step. Here we select all S_{12} suffixes, forming 3-character substrings, and do a 25-bit radix sort on those substrings. (25 bits can represent 3 chars from a constant alphabet in the range $[0 \ldots 255]$ plus a sentinel letter \$.) Next, we compute the ranks of S_{12} and assign the resulting ranks into an inverse suffix array ($\text{ISA}_{[12]}$). From now on, we work with suffixes in our new partially-sorted order rather than the original text order. (Thus after this initial radix sort, all suffixes with the same 3-character prefix are contiguous in our array—"Key" is next to "Keyword"—no matter where they appear in the original text.

Next, we turn to our prefix-doubling implementation. In it, we sort by (ISA[SA[i]+δ], ISA[SA[i]+2δ]) pairs. Here δ is the current length of the prefix; this length doubles in each iteration until all suffixes are in their own *segment*s. We define a segment as a set of suffixes that are equal up to the current substring length. Each segment is assigned 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. If we identify a suffix at the end of an iteration that is a singleton in its own segment, we can conclude that its final position in the suffix array is fixed. We can then fix their final positions in the output suffix array, compact those singletons out of the working suffix array (removing them from any further processing), and re-rank the remaining segments. Future iterations only consider suffixes whose final positions are not yet fixed.

Recall from Section 2 that Deo and Keely were concerned that "prefix-doubling has an irregular, data-dependent number of unsorted groups across phases". Addressing this concern is crucial for performance and one of our core contributions in this work. We must sort efficiently within each segment, even though the number of segments and their sizes are non-uniform and not known at compile time. We address this with an efficient *segmented sort* primitive, which we describe next.

Segmented sort The input to segmented sort is a contiguous, ordered list of segments with a variable number of unsorted items per segment; the output is the list of segments in the same order but with items sorted within each segment.

One possible way to implement a segmented sort 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. It is thus desirable to consider approaches which can simultaneously sort multiple segments at the same time. So a second approach is a full sort over all items, but this does more work than necessary: it ignores the significant work that has already been completed in classifying

DOI: 10.1002/cpe

the items into segments. Ideally, we would leverage the presence of segments but also work on all segments simultaneously.

What makes a segmented sort particularly challenging is the variation in the size and number of segments. One possible approach might be maintaining segment IDs as the most significant bits of the key (to maintain segment stability) while choosing an appropriate sorting method for each individual segment. While this approach helps to address the issues with the two methods in the previous paragraph, it would be complex to implement and maintain, and its efficiency would be difficult to predict. Like our previously-described merge kernel, the real challenge here is characterizing the problem in a way that allows us to distribute constant amounts of work to independent chunks, even in the presence of varying segment sizes.

In our implementation, we divide the input into equal-sized "blocks", and then launch "blocksorts" to sort within each block while maintaining segment order. Then we use a sequence of iterative merge operations to get the final result.

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 primitive described in Section 4.1 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 Figure 6.

The final optimization is early exit (also shown in Figure 6). 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.

Our method is implemented as a segmented-sort primitive in the second author's *Modern GPU* library[‖]. An efficient segmented sort is the difference-maker in developing a competitive prefix-doubling implementation.

Comparison against plain prefix-doubling implementation Our hybrid prefix-doubling method makes several improvements to Osipov's prefix-doubling implementation [6]. Osipov modifies MM by replacing multiple (32-bit key, 32-bit value) radix sort with a single (32-bit key, 64-bit value) radix sort. His implementation also filters out fully-sorted suffixes to avoid unnecessary re-sorting at the end of each iteration, similar to LS. Throughout his implementation, he leverages the parallel primitives of Merrill's *back40computing* library[**], including prefix-sum, radix sort, and random gather from and scatter to memory.

Osipov begins his implementation with a sort of the first 4 characters of each suffix. Instead, we begin with the first step of skew—a (32-bit key, 25-bit value) radix sort. This sort is inexpensive and gives us a reduction ratio of 0.67. Throughout our implementation we use our *segmented sort* primitive, which has better locality than radix-sorting integer tuples in global memory. Furthermore, when we sort the remaining 1/3 suffixes, our induction step in the skew framework is cheaper than a radix sort. Though we require an additional merge in the final step, our parallel merge primitive is

[‖]Code is available at http://nvlabs.github.io/moderngpu and described in http://nvlabs.github.io/moderngpu/segsort.html.
[**]https://code.google.com/p/back40computing/

DOI: 10.1002/cpe

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
m ^	***m***	***i***	***i***	***s***	*s* ^	*i*	*i*	***s*** ^	***s***	***i***	***i***	***p***	*p* ^	*i*	*i*
(0	1	2	3)	(4	5	6	7)	(8	9	10	11)	(12	13	14	15)
(***i*** ^	***i***	***m***	***m***)	(***s***	*i* ^	*i*	*s*)	(***i*** ^	***i***	***s***	***s***)	(***p***	*i* ^	*i*	*p*)
(0	1	2	3	4	5	6	7)	(8	9	10	11	12	13	14	15)
(***i*** ^	***i***	***m***	***m***	***s***	*i* ^	*i*	*s*)	(***i*** ^	***i***	***p***	***s***	***s***	*i* ^	*i*	*p*)
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
i ^	***i***	***m***	***m***	***s***	*i* ^	*i*	*s*	***i*** ^	***i***	***p***	***s***	***s***	*i* ^	*i*	*p*

Figure 6. 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).

quite efficient (see Section 4.1). We compare Osipov's pure prefix-doubling implementation to our approach in Figure 5.

4.3. Skew vs. Prefix-doubling

In skew, we build and sort 3-character substrings for the suffixes S_{12}, which yields $\text{SA}_{[12]}$. Then we use the sorted order of S_1 to induce the order of S_0 by outputting $(x[i], \text{ISA}[i+1])$ pairs, where $i \bmod 3 \equiv 0$, in the sorted order of S_1, then perform an 8-bit radix sort on $x[i]$ to put S_0 in order. Finally, we merge the sorted $\text{SA}_{[0]}$ and $\text{SA}_{[12]}$ arrays.

The key question for a skew implementation is how to sort the S_{12} subset. Most implementations re-rank the sorted 3-character substrings to form a new integer alphabet; re-write the suffixes in unsorted order in terms of this new integer alphabet, with one rank replacing a 3-character substring; then select a new S_{12} subset; and recurse. This approach makes the function satisfyingly recursive, but it results in repeated sorts of the same suffixes, with an associated loss of overall performance. The prefix-doubling approach eliminates repeated sorting of the same suffixes: once a suffix is placed into a bucket, we know that it will stay in that bucket and will come before all later buckets and after all earlier buckets. 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. With both methods, we have the choice of filtering out fully-sorted suffixes in each round. With prefix-doubling, which is non-recursive, this helps performance tremendously and has no drawbacks.

As for sorting, skew with a difference cover modulo 3 is a "prefix-tripling" technique[††], 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. As well, skew's prefix-tripling has the drawback of making ranking more complicated. Ignoring the difficulty of sorting within segments for now, we see that for the core of the algorithms, prefix-doubling is more efficient than skew. In its radix sort, skew uses the

[††]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.

DOI: 10.1002/cpe

most significant digit simply to get the suffix back in its original segment, which comes for free with prefix-doubling's segmented sort.

Together, the above advantages make prefix-doubling more efficient than skew, at least with real-world texts. Skew does have the advantage of predictability with its reduction ratio per pass of 0.67, independent of the input data. In contrast, prefix-doubling has a worst-case reduction ratio of 1.0 per pass (if the pass fails to resolve any suffixes), but has a practical reduction ratio on real-world text that is typically favorable.

5. EXPERIMENTS & RESULTS

In this section we evaluate our implementations of suffix array algorithms, and compare them to the implementations and published results of prior work. We also re-implement Deo and Keely's skew approach on an NVIDIA GPU using CUDA with two enhancements: a current state-of-the-art radix sort primitive from Merrill's *CUB* library[‡] and the merge primitive that we use in our own implementation (from the second author's *Modern GPU* library). In the remainder of this section, we refer to Deo and Keely's (original) OpenCL parallel skew implementation on an AMD GPU *dk-amd-skew*, our CUDA reimplementation of Deo and Keely's approach *dk-nvidia-skew*, Osipov's parallel prefix-doubling *osipov-pd*, our parallel skew implementation *wbo-skew*, and our parallel hybrid skew/prefix-doubling implementation *wbo-skew/pd*.

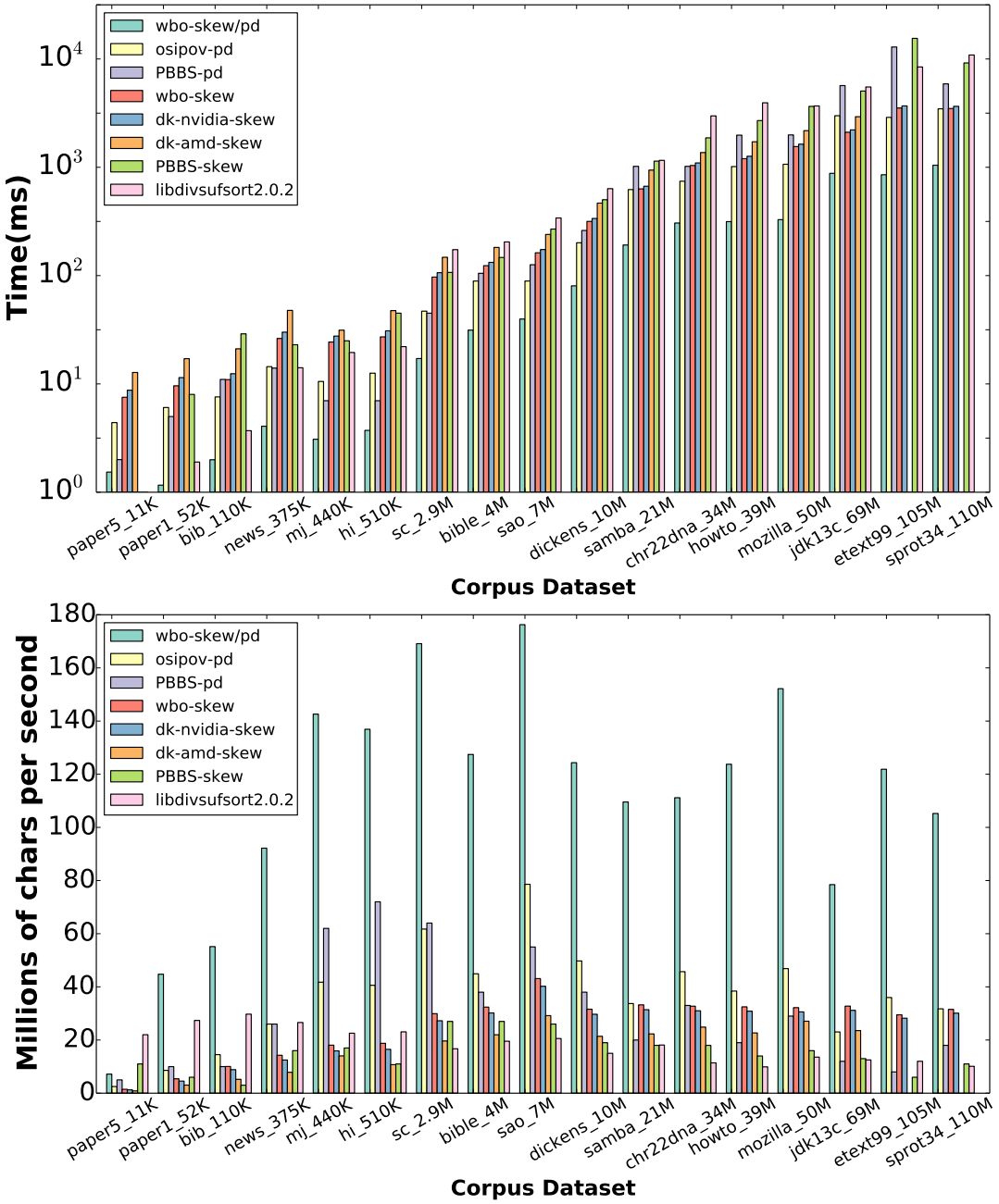
We evaluate all results on an Intel Xeon E5-2637 v2 3.5 GHz 4-core system with 786 GB RAM and 15 MB L3 cache. We used an NVIDIA Tesla K20c GPU (launch date: November 2012; process: 28 nm; peak single-precision floating-point throughput: 3.524 TFLOPS; peak memory bandwidth: 208 GB/s). Deo and Keely's 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-skew, wbo-skew, wbo-skew/pd, and osipov-pd using CUDA 6.0 and Visual Studio 2010 on 64-bit Windows 7.

In our evaluation, we use the same input datasets as Deo and Keely together with two larger datasets. The input strings from these datasets range in size from 10 KB to 110 MB and are collected from the Calgary Corpus, Large Canterbury Corpus, Manzini's Corpus, Protein Corpus, and Silesia Corpus [32]. We compare the four GPU implementation results against Mori's highly tuned, OpenMP-assisted CPU implementation libdivsufsort 2.0.2 [32] based on induced copying on a 4-core PC, using its own internal runtime measurement, which excludes disk access time. We also compare our results with the two Intel Cilk Plus accelerated CPU implementions *PBBS-skew* and *PBBS-pd* in the problem based benchmark suite (PBBS)[†] which are claimed to be the current fastest CPU implementations by Shun et al..

Figure 7, 8 and 9 summarize our performance results. We make the following observations:

- On datasets of sufficient size (on the order of 1 MB for the skew implementations, smaller for prefix-doubling-based implementations), all five GPU implementations and the two CPU implementations from PBBS are faster than the CPU baseline libdivsufsort 2.0.2. Roughly speaking, the GPU skew-based implementations are twice as fast as libdivsufsort, osipov-pd has a 4× speedup, and our hybrid prefix-doubling-based wbo-skew/pd's speedup ranges from 6–11×.
- Macroscopically, the performance of GPU and CPU implementations from the same SACA class track each other—the fluctuations in the throughputs of wbo-skew/pd, osipov-pd and PBBS-pd for the same datasets suggest that the behavior of our wbo-skew/pd is similar to that of osipov-pd and PBBS-pd, and our wbo-skew's behavior is similar to dk-amd-skew, dk-nvidia-skew and PBBS-skew.

DOI: 10.1002/cpe



Method	chr22dna_34M	mozilla_50M	w3c2_104M
dk-amd-SA	1368	2178.5	—
dk-nvidia-SA	1096.8	1634.9	3423
skew-SA	1038.8	1553.9	3261.4
osipov-SA	743.8	1066.7	5729.6
spd-SA	**305.9**	**328.6**	**1312.3**

Figure 7. Runtimes (top figure) and throughputs (bottom figure) of five GPU suffix array construction implementations as well as three CPU implementations over corpus datasets; the datasets are those chosen by Deo and Keely [5] in addition to two larger datasets for which we have no dk-amd-SA measurements. Specific runtimes of different methods for four datasets are listed (bottom table) in milliseconds (bold indicates the fastest). The five GPU implementations are wbo-skew/pd, osipov-pd, wbo-skew, dk-nvidia-skew, and dk-amd-skew. The three CPU implementations are PBBS-pd, PBBS-skew and libdivsufsort2.0.2.

Prepared using cpeauth.cls
DOI: 10.1002/cpe

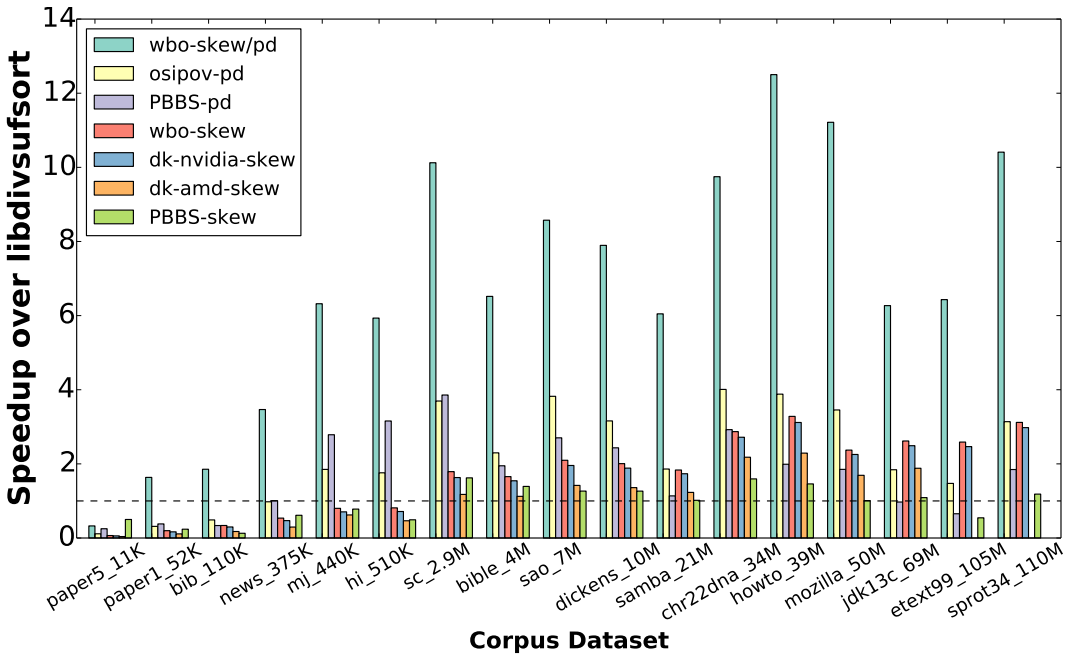


Figure 8. Speedup comparisons of five GPU suffix array construction implementations (wbo-skew/pd, osipov-pd, wbo-skew, dk-nvidia-skew and dk-amd-skew) as well as two current fastest CPU implementations (PBBS-pd and PBBS-skew). The CPU implementation of libdivsufsort 2.0.2 is the baseline for speedup comparisons.

- Our hybrid skew/prefix-doubling wbo-skew/pd is 2.3–4.4× faster than Osipov's plain prefix-doubling osipov-pd and have a speedup of 2.3–9.8× over the current fastest prefix-doubing-based CPU implementation PBBS-pd (both of the highest speedups happen on the *w3c2_104M* dataset).
- Our wbo-skew is consistently 1.45× faster than dk-amd-skew, 1.1× faster than dk-nvidia-skew and 1.1–4.8× faster than the current fastest skew-based CPU implementation PBBS-skew. We thus validate the performance improvements we made both in terms of core primitives (specifically, our merge approach) as well as our algorithmic changes to skew.
- Both prefix-doubling-based GPU implementations outperform the three skew-based methods on most datasets, prefix-doubling-based CPU implementation PBBS-pd always outperforms skew-based PBBS-skew and is even faster than GPU skew-based implementations on some datasets. Our hybrid wbo-skew/pd significantly outperforms every GPU and CPU routine on every dataset.

Uniform vs. non-uniform prefix distributions We note that the datasets with the highest speedups on GPUs are those with a non-uniform prefix distribution (e.g., *chr22dna*, which contains DNA sequences composed of only 4 different characters). Datasets with more uniformly distributed prefixes yield smaller speedups. Non-uniform prefixes lead to higher speedups because the GPU implementations, particularly our hybrid skew/prefix-doubling implementation, are faster on datasets with non-uniform prefixes. Why? In the skew formulation, a more uniform dataset results in more iterations in the recursive step, leading to a longer runtime. In a prefix-doubling formulation, uniform datasets result in fewer segments for separation and thus expose less parallelism.

DOI: 10.1002/cpe

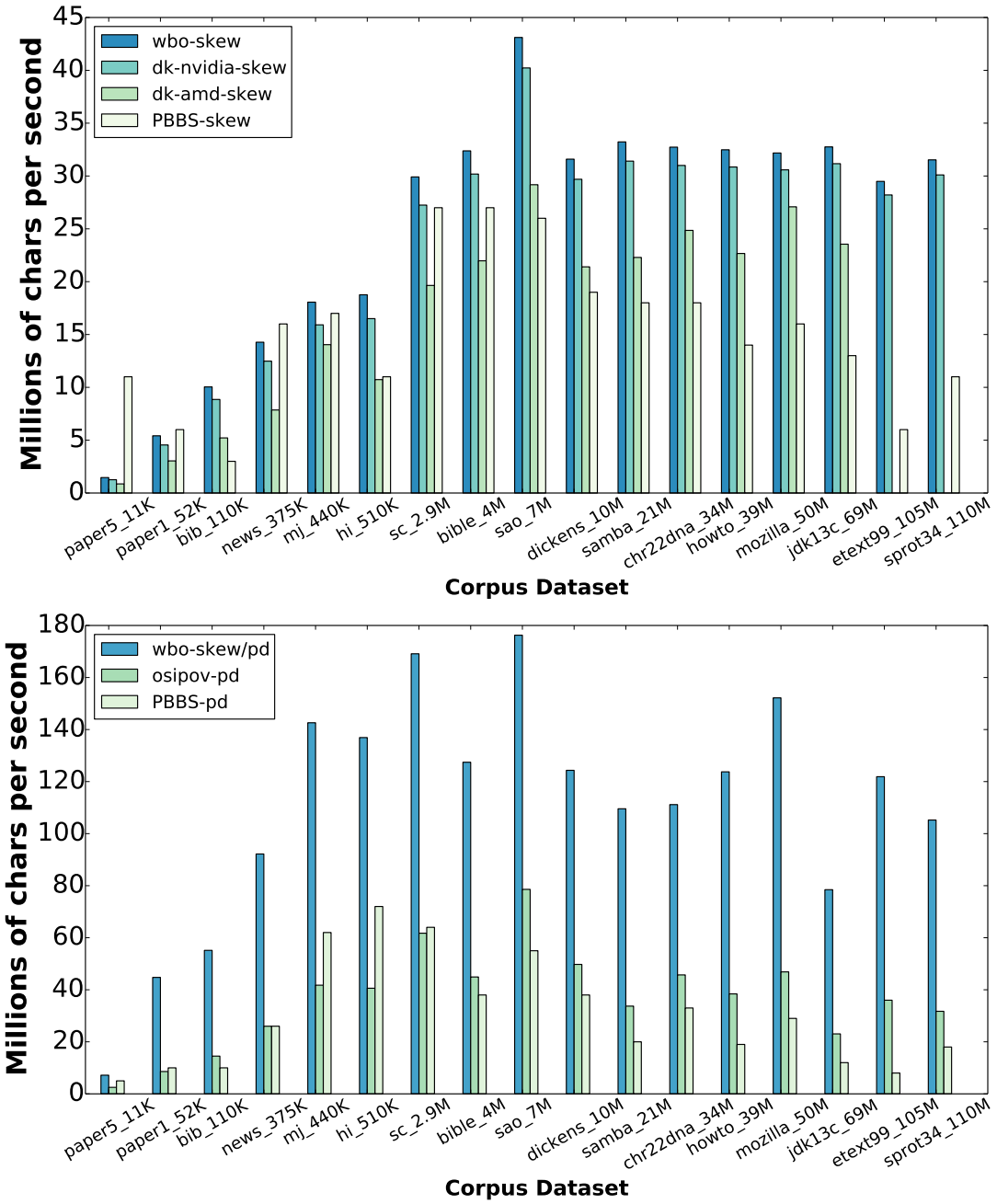


Figure 9. Throughputs of skew-based suffix array construction implementations (top figure) and prefix-doubling-based suffix array construction implementations (bottom figure). The skew-based GPU implementations are wbo-skew, dk-nvidia-skew, dk-amd-skew, and the CPU one is PBBS-skew. The prefix-doubling-based GPU implementations are wbo-skew/pd, osipov-dp and the CPU one is PBBS-pd.

Scalability with dataset size How do skew-SA and spd-SA scale with larger datasets? We consider increasing amounts of text data from the English Wikipedia dump "enwik8"[‡‡], shown in Figure 10 at left. In general, as the dataset becomes larger, the throughput increases. Both approaches require input sizes of many millions of characters to reach the throughput asymptote.

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

DOI: 10.1002/cpe

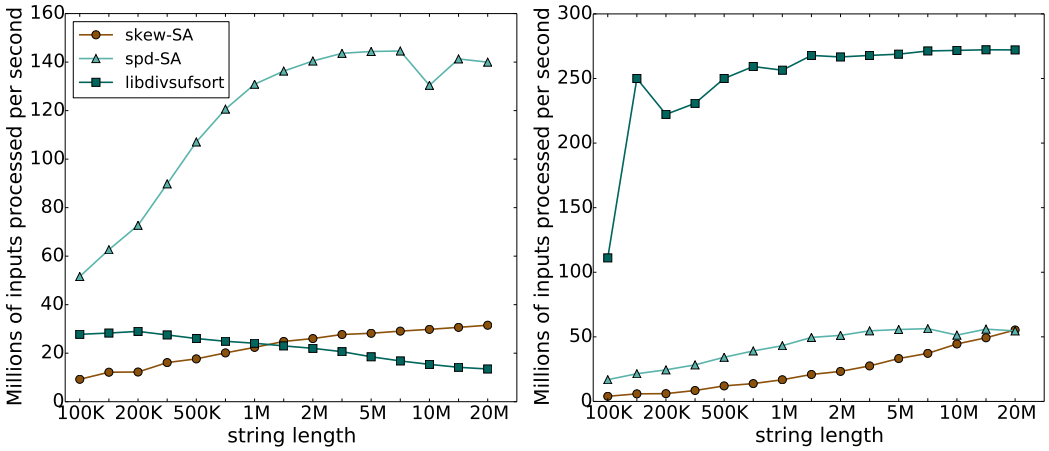


Figure 10. Left, suffix array construction throughput on plain text "enwik8" as a function of dataset size; right, suffix array construction throughput on a dataset consisting only of the repeated letter 'A', using the same legend as the left graph.

With a 10 MB input dataset, skew-SA has a 2× speedup and spd-SA a 9× speedup over *libdivsufsort*.

Scalability with worst-case input We consider an artificial dataset composed of only the repeated single character 'A', shown in Figure 10 at right. For prefix-doubling, this is a pathologically bad case: except for suffixes near the end of the string, every input position has an identical prefix on every iteration until the last one, so a prefix-doubling approach 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. The performance of a skew approach is more predictable (and non-pathological). With this dataset, skew must recurse all the way to the base case and cannot finish early. However, on each iteration, skew has a reduction ratio of 0.67, just as it would on any dataset. Nonetheless, except for very large inputs, the performance of spd-SA still exceeds that of skew-SA.

This dataset is much better suited for an induced copying approach. For a 10 MB all-'A' input, *libdivsufsort* (the CPU implementation based on induced copying) runs in 40 ms, compared with 224 ms for skew-SA and 196 ms for spd-SA.

Suffix array construction as a building block The latest release (version 2.2) of our CUDA Data Parallel Primitives (CUDPP) Library[§] contains our optimized skew implementation. We use this implementation in CUDPP's Burrows-Wheeler Transform (BWT) [33] and its bzip2 data compression procedures. Both previously used a full string sort rather than an optimized suffix array; as we might expect [34], using our suffix array construction implementation enables significant speedups for both.

We also incorporate our fast hybrid skew/prefix-doubling implementation in a parallel BWT and use it as a step in implementing parallel FM index backward_search (Figure 1), along with CUB's DeviceHistogram routine and cudppMultiscan from CUDPP 2.2. We measure the performance of the parallel BWT and FM index backward_search based on our fast hybrid skew/prefix-doubling implementation on two representative real-world datasets "enwik8" (first 10^8 bytes of the English Wikipedia dump on Mar.3, 2006) and "chr22dna" (34 megabytes DNA sequences from Manzini's Corpus) and show our results in Table III.

DOI: 10.1002/cpe

Table III. Throughputs 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

6. 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. Delivering an efficient parallelization and implementation of induced copying for GPUs would hopefully open the door to effective techniques for some of the most challenging parallelization problems.

ACKNOWLEDGEMENTS

Thanks to Yangzihao Wang for the initial implementation and good advice along the way. Thanks also to 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 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. Manber U, Myers G. Suffix arrays: A new method for on-line string searches. *Proceedings of the First Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '90, 1990; 319–327.
2. Karp RM, Miller RE, Rosenberg AL. Rapid identification of repeated patterns in strings, trees and arrays. *Proceedings of the Fourth Annual ACM Symposium on Theory of Computing*, STOC '72, 1972; 125–136, doi: 10.1145/800152.804905.
3. Larsson NJ, Sadakane K. Faster suffix sorting. *Theoretical Computer Science* 2007; **387**(3):258–272, doi:10.1016/j.tcs.2007.07.017.
4. Kärkkäinen J, Sanders P. Simple linear work suffix array construction. *Proceedings of the 30th International Conference on Automata, Languages and Programming*, *ICALP '03*, vol. 2719, Baeten JCM, Lenstra JK, Parrow J, Woeginger GJ (eds.). Springer-Verlag: Berlin, Heidelberg, 2003; 943–955, doi:10.1007/3-540-45061-0_73.
5. Deo M, Keely S. Parallel suffix array and least common prefix for the GPU. *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '13, 2013; 197–206, doi: 10.1145/2442516.2442536.
6. Osipov V. Parallel suffix array construction for shared memory architectures. *Proceedings of the 19th International Conference on String Processing and Information Retrieval*, SPIRE'12, Springer-Verlag, 2012; 379–384, doi: 10.1007/978-3-642-34109-0_40.
7. Wang L, Baxter S, Owens JD. Fast parallel suffix array on the GPU. *Euro-Par 2015: Parallel Processing*, *Lecture Notes in Computer Science*, vol. 9233, Träff JL, Hunold S, Versaci F (eds.). Springer Berlin Heidelberg, 2015; 573–587, doi:10.1007/978-3-662-48096-0_44.
8. Futamura N, Aluru S, Kurtz S. Parallel suffix sorting. *Technical Report Paper 64*, Syracuse University, Electrical Engineering and Computer Science 1 Jan 2001. URL http://surface.syr.edu/eecs/64.
9. Bentley JL, Sedgewick R. Fast algorithms for sorting and searching strings. *Proceedings of the Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '97, Society for Industrial and Applied Mathematics: Philadelphia, PA, USA, 1997; 360–369. URL http://dl.acm.org/citation.cfm?id=314161.314321.

10. Abdelhadi A, Kandil AH, Abouelhoda M. Cloud-based parallel suffix array construction based on MPI. *2014 Middle East Conference on Biomedical Engineering (MECBME)*, 2014; 334–337, doi:10.1109/MECBME.2014.6783271.
11. Kulla F, Sanders P. Scalable parallel suffix array construction. *Parallel Comput.* Sep 2007; **33**:605–612, doi:10.1016/j.parco.2007.06.004.
12. Flick P, Aluru S. Parallel distributed memory construction of suffix and longest common prefix arrays. *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '15, ACM: New York, NY, USA, 2015; 16:1–16:10, doi:10.1145/2807591.2807609.
13. Homann R, Fleer D, Giegerich R, Rehmsmeier M. mkESA: enhanced suffix array construction tool. *Bioinformatics* 15 Apr 2009; **25**(8):1084–1085, doi:10.1093/bioinformatics/btp112.
14. Manzini G, Ferragina P. Engineering a lightweight suffix array construction algorithm. *Algorithmica* Jun 2004; **40**(1):33–50, doi:10.1007/s00453-004-1094-1.
15. Mohamed H, Abouelhoda M. Parallel suffix sorting based on bucket pointer refinement. *5th Cairo International Biomedical Engineering Conference*, CIBEC 2010, 2010; 98–102, doi:10.1109/CIBEC.2010.5716066.
16. Schürmann KB, Stoye J. An incomplex algorithm for fast suffix array construction. *ALENEX/ANALCO*, 2005; 78–85.
17. Shun J, Blelloch GE, Fineman JT, Gibbons PB, Kyrola A, Simhadri HV, Tangwongsan K. Brief announcement: The problem based benchmark suite. *Proceedings of the Twenty-fourth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '12, 2012; 68–70, doi:10.1145/2312005.2312018.
18. Pantaleoni J. A massively parallel algorithm for constructing the BWT of large string sets. *arXiv.org* Oct 2014; **abs/1410.0562**(1410.0562v1).
19. Liu CM, Luo R, Lam TW. GPU-accelerated BWT construction for large collection of short reads. *arXiv.org* Jan 2014; **abs/1410.7457**(1410.7457v1).
20. Kärkkäinen J. Fast BWT in small space by blockwise suffix sorting. *Theor. Comput. Sci.* Nov 2007; **387**(3):249–257, doi:10.1016/j.tcs.2007.07.018.
21. Ferragina P, Manzini G. Opportunistic data structures with applications. *Proceedings of the 41st Annual Symposium on Foundations of Computer Science*, FOCS 2000, 2000; 390–398, doi:10.1109/SFCS.2000.892127.
22. Merrill D, Grimshaw A. Revisiting sorting for GPGPU stream architectures. *Technical Report CS2010-03*, Department of Computer Science, University of Virginia Feb 2010, doi:10.1145/1854273.1854344. URL https://sites.google.com/site/duanemerrill/RadixSortTR.pdf.
23. Davidson A, Tarjan D, Garland M, Owens JD. Efficient parallel merge sort for fixed and variable length keys. *Proceedings of Innovative Parallel Computing*, InPar '12, 2012, doi:10.1109/InPar.2012.6339592.
24. Farach M. Optimal suffix tree construction with large alphabets. *Proceedings of the 38th Annual Symposium on Foundations of Computer Science*, FOCS '97, 1997; 137–143, doi:10.1109/SFCS.1997.646102.
25. Itoh H, Tanaka H. An efficient method for in memory construction of suffix arrays. *Proceedings of the String Processing and Information Retrieval Symposium & International Workshop on Groupware*, SPIRE '99, 1999; 81–88, doi:10.1109/SPIRE.1999.796581.
26. NVIDIA Corporation. NVIDIA CUDA C programming guide Sep 2015. PG-02829-001_v7.5.
27. Howes L, Munshi A. *The OpenCL Specification (Version 2.1, Document Revision 23)* 11 Nov 2015. http://www.khronos.org/registry/cl/specs/opencl-2.1.pdf.
28. Lindholm E, Nickolls J, Oberman S, Montrym J. NVIDIA Tesla: A unified graphics and computing architecture. *IEEE Micro* Mar/Apr 2008; **28**(2):39–55, doi:10.1109/MM.2008.31.
29. Nickolls J, Buck I, Garland M, Skadron K. Scalable parallel programming with CUDA. *ACM Queue* Mar/Apr 2008; **6**(2):40–53, doi:10.1145/1365490.1365500.
30. Satish N, Harris M, Garland M. Designing efficient sorting algorithms for manycore GPUs. *Proceedings of the 23rd IEEE International Parallel and Distributed Processing Symposium*, 2009, doi:10.1109/IPDPS.2009.5161005.
31. Green O, McColl R, Bader DA. GPU merge path: A GPU merging algorithm. *Proceedings of the 26th ACM International Conference on Supercomputing*, ICS '12, 2012; 331–340, doi:10.1145/2304576.2304621.
32. Mori Y. libdivsufsort, version 2.0.2. https://github.com/y-256/libdivsufsort 2015.
33. Patel RA, Zhang Y, Mak J, Owens JD. Parallel lossless data compression on the GPU. *Proceedings of Innovative Parallel Computing*, 2012, doi:10.1109/InPar.2012.6339599.
34. Edwards JA, Vishkin U. Parallel algorithms for Burrows-Wheeler compression and decompression. *Theoretical Computer Science* 13 Mar 2014; **525**:10–22, doi:10.1016/j.tcs.2013.10.009.