

Aalto University  
School of Science  
Degree Programme in Computer, Communication and Information Sciences

Yury Shukhrov

# Lightweight Massively Parallel Suffix Array Construction

Master's Thesis  
Espoo, May 26, 2019

Supervisor: Professor Petteri Kaski

<b>Author:</b>	Yury Shukhrov		
<b>Title:</b>	Lightweight Massively Parallel Suffix Array Construction		
<b>Date:</b>	May 26, 2019	<b>Pages:</b>	88
<b>Major:</b>	Computer Science	<b>Code:</b>	SCI3042
<b>Supervisor:</b>	Professor Petteri Kaski		
<p>The suffix array is an array of sorted suffixes in lexicographic order, where each sorted suffix is represented by its starting position in the input string. It is a fundamental data structure that finds various applications in areas such as string processing, text indexing, data compression, computational biology, and many more. Over the last three decades, researchers have proposed a broad spectrum of suffix array construction algorithms (SACAs). However, the majority of SACAs were implemented using sequential and parallel programming models. The maturity of GPU programming opened doors to the development of massively parallel GPU SACAs that outperform the fastest versions of suffix sorting algorithms optimized for the CPU parallel computing. Over the last five years, several GPU SACA approaches were proposed and implemented. They prioritized the running time over lightweight design.</p> <p>In this thesis, we design and implement a lightweight massively parallel SACA on the GPU using the prefix-doubling technique. Our prefix-doubling implementation is memory-efficient and can successfully construct the suffix array for input strings as large as 640 megabytes (MB) on Tesla P100 GPU. On large datasets, our implementation achieves a speedup of 7-16x over the fastest, highly optimized, OpenMP-accelerated suffix array constructor, <i>libdivsufsort</i>, that leverages the CPU shared memory parallelism. The performance of our algorithm relies on several high-performance parallel primitives such as radix sort, conditional filtering, inclusive prefix sum, random memory scattering, and segmented sort. We evaluate the performance of our implementation over a variety of real-world datasets with respect to its runtime, throughput, memory usage, and scalability. We compare our results against <i>libdivsufsort</i> that we run on a Haswell compute node equipped with 24 cores. Our GPU SACA is simple and compact, consisting of less than 300 lines of readable and effective source code. Additionally, we design and implement a fast and lightweight algorithm for checking the correctness of the suffix array.</p>			
<b>Keywords:</b>	suffix array construction, GPU, prefix-doubling, CUDA, skew, Burrows-Wheeler transform, algorithm engineering, parallel primitives, induced sorting		
<b>Language:</b>	English		

# Acknowledgements

First and foremost, I would like to express my sincere gratitude to my supervisor, Professor Petteri Kaski, for his continuous support, patience, and guidance throughout the course of this work. I also wish to thank him for providing me the opportunity to work on a challenging yet exciting project. His constructive feedback, immense knowledge, and valuable suggestions have contributed greatly to the improvement of the thesis.

I would like to acknowledge the use of computing resources available via project "Science-IT" at Aalto University School of Science and via CSC—the Finnish IT Center for Science. As regards the latter I would especially like to thank "Science-IT" administrators for granting me access to the Triton cluster to carry out the experiments in this thesis.

Finally, I would like to thank my parents for their support and faith in my ability to attain my goals.

Espoo, May 26, 2019

Yury Shukhrov

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>Background</b>	<b>4</b>
2.1	Notation and Terminology . . . . .	4
2.2	The Suffix Array . . . . .	7
2.3	The Burrows-Wheeler Transform . . . . .	9
2.4	Prior Work . . . . .	11
<b>3</b>	<b>Algorithms for Suffix Array Construction</b>	<b>15</b>
3.1	Prefix-Doubling Algorithms . . . . .	16
3.1.1	Manber and Myers Algorithm . . . . .	16
3.1.2	Larsson and Sadakane Algorithm . . . . .	23
3.2	Recursive Algorithms . . . . .	28
3.3	Induced Sorting Algorithms . . . . .	30
<b>4</b>	<b>GPU Parallel Programming</b>	<b>34</b>
4.1	Compute Unified Device Architecture . . . . .	34
4.2	Essentials of the Thrust Library . . . . .	36
4.2.1	Host and Device Vectors . . . . .	37
4.2.2	Interoperability . . . . .	38
4.2.3	Anonymous Kernels . . . . .	40
4.3	Parallel Primitives . . . . .	42
4.3.1	Radix Sort . . . . .	42
4.3.2	Select-Flagged . . . . .	44
4.3.3	Inclusive Prefix Sum . . . . .	46
4.3.4	Scatter . . . . .	48
4.3.5	Segmented Sort . . . . .	49
<b>5</b>	<b>Implementations</b>	<b>52</b>
5.1	Choosing the Algorithm . . . . .	53
5.2	Suffix Array Construction on the GPU . . . . .	54

5.3	The Suffix Array Checker . . . . .	66
<b>6</b>	<b>Experimental Results</b>	<b>69</b>
6.1	Hardware Specifications . . . . .	69
6.2	Performance Evaluation . . . . .	70
6.3	Scalability Analysis . . . . .	74
<b>7</b>	<b>Conclusions</b>	<b>77</b>

# Chapter 1

## Introduction

The problem of *suffix sorting* boils down to finding a lexicographic order of all suffixes in a string. The *suffix array* (SA) is an elegant and compact data structure that stores the starting positions of sorted suffixes. The concept of the SA was proposed in 1990 by Manber and Myers [53] as the space-saving substitute of the *suffix tree* [19, 24] data structure. The SA finds an important application in different areas such as stringology [16], genome analysis [2], bioinformatics [18], and many more.

For example, one can use the SA to determine quickly if a given text contains a required pattern. Instead of constructing an index structure for the text by repetitive queries and text preprocessing, the SA provides an immediate index structure solution. Thus, to find all matches of a pattern in a text one needs to identify each suffix that begins with a pattern. A simple solution requires a double pass of a *binary search* [91] to locate the first and last index of the interval containing the pattern. This kind of indexes is called *full-text indexes* [50].

Another application of the SA is in data compression. The SA plays an important role in *Burrows-Wheeler transformation* (BWT) introduced by Burrows and Wheeler in 1994 [12]. The BWT of a string is obtained by generating cyclic rotations of this string, by sorting them in lexicographic order, and by extracting the last character of these sorted strings. This transformation is reversible with minimal data overhead. The BWT can be efficiently computed as a simple modification of the SA. It is a powerful tool for transforming data into a form suitable for lossless compression algorithms incorporated in popular compressors such as *bzip2* [7] and *gzip* [78].

A trivial way to construct the suffix array of the input text is to employ a sorting algorithm (e.g., *quick sort* or *merge sort* [27, 41]) to find the lexicographic order of suffixes. This approach will result in  $O(n^2 \log n)$  time complexity because  $O(n \log n)$  comparisons are used and each string comparison

takes  $O(n)$  time. However, we can significantly improve on that complexity if we utilize the knowledge that there is some dependency between suffixes. In fact, suffixes are overlapping substrings.

Currently, suffix array construction algorithms (SACAs) that take advantage of this knowledge can be classified into three main categories: *prefix-doubling*, *recursive*, and *induced sorting* [74]. Prefix-doubling SACAs use the rank of prefixes to determine the order of suffixes, where the length of each prefix is doubled in each iteration. Recursive SACA approaches group suffixes into two subsets according to some criteria. One subset with  $2/3$  or less suffixes is recursively sorted, and its order is used to induce the order of the second subset. Once both subsets are sorted, merge sort is used to combine the results. Induced sorting algorithms use the information about the sorted subset of suffixes to induce the order of the remaining suffixes.

Over the past two decades, the majority of the proposed SACAs have been implemented using the sequential programming model. The advances and accessibility of the general-purpose computing on graphics cards (GPGPU) have spurred interest in designing an efficient SACA that would harness the power of massively parallel GPU architectures and outperform currently fastest parallel SACAs built for multi-core CPU architectures. In the past five years, there were several approaches to design and implement a shared memory parallel SACA using the GPU programming model. The most prominent is the implementation of Deo and Kelly [17], based on the *skew* algorithm [77]; the implementation of Osipov [73], based on prefix-doubling method; and the implementation of Wang *et al.* [89], based on a hybrid *skew prefix-doubling* approach.

To best of our knowledge, previous approaches to constructing a suffix array with GPGPU coding prioritized the running time of the implementation over other technical aspects. As a result, implementations in the previous approaches lacked a simple, compact and lightweight design. We are motivated to improve on the previous approaches in terms of simplicity and compactness of the GPU SACA implementation. In this thesis, we design and implement a lightweight and memory-efficient GPU SACA using modern, high-performance, state-of-the-art parallel primitives to achieve best results. We experimentally evaluate the performance of our implementation using real-world datasets and comparing the runtime, throughput, and scalability against the fastest in practice suffix sorting algorithm working in main memory [21], *libdivsufsort* [61].

This thesis is organized as follows: we begin in Chapter 2 with a brief introduction to the basic definitions and notations in stringology, we study basic properties of the SA and its inverse. Then, we describe the BTW and demonstrate how it can be efficiently constructed by the SA. Finally, we give

a short review of the prior work done in the area of suffix array construction. In Chapter 3, we discuss three fundamental methods for suffix array construction. We study working principles of the SACAs based on each method. In Chapter 4, we review the programming model for GPGPU: we begin with a brief introduction to *Compute Unified Device Architecture* (CUDA), then we cover essentials of the Thrust library that are employed in our implementation, and explain the working principles of the most efficient parallel primitives that were incorporated in our GPU SACA implementation. In Chapter 5, we present our GPU SACA implementation and justify the reason for choosing the prefix-doubling method. Additionally, we present a fast and lightweight implementation of the suffix array checker. This program serves the purpose of checking the correctness of our GPU SACA implementation. Its working principles are based on three conditions for checking the correctness of the suffix array. We discuss these conditions and describe our implementation approach. Experimental evaluation of our implementation and the corresponding results are presented in Chapter 6. We conclude this thesis in Chapter 7.

## Chapter 2

# Background

In this chapter, we give a brief introduction to basic terminologies and definitions in stringology. We describe the concept of the suffix array data structure and expose the related notation. We study the working principles of the *Burrows-Wheeler Transform* (BWT) and demonstrate how to efficiently construct a BWT string directly from the SA. The chapter ends with a review of the advances in suffix array construction.

### 2.1 Notation and Terminology

A *string* is a sequence of characters drawn from a finite set, called *alphabet*. The alphabet may contain *symbols*, *letters* or *characters*. We will usually denote an alphabet with the symbol  $\Sigma$  and its size with  $\sigma$ , that is  $|\Sigma| = \sigma$ . The size of the alphabet is the number of unique elements in it [15, 16].

**Example 2.1.** The most common type of the alphabet is the basic modern Latin alphabet of lowercase letters:

$$\Sigma = \{a, b, c, \dots, x, y, z\}.$$

This alphabet contains 26 distinct letters, that is  $\sigma = 26$ . The word  $S = \text{abracadabra}$  is an example of a string over the alphabet  $\Sigma$ .

Let  $\Sigma_1 = \{0, 1\}$  be the alphabet of size two containing binary numbers. Then strings  $\varepsilon, 0, 1, 00, 01, 10, 11, 000$ , etc. are all in  $\Sigma_1$ , where  $\varepsilon$  represents the *empty string*. This kind of alphabet is commonly used in the context of finite automata [82].

Other commonly used alphabets include the set of 256 8-bit ASCII (*American Standard Code for Information Interchange*) [49] symbols or the set of DNA (*DeoxyriboNucleic Acid*) characters [84].

We denote the set of all strings over an alphabet  $\Sigma$  by

$$\Sigma^* = \bigcup_{t=0}^{\infty} \Sigma^t = \Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cup \dots,$$

where

$$\begin{aligned} \Sigma^t &= \overbrace{\Sigma \times \Sigma \times \dots \times \Sigma}^t \\ &= \{x_1x_2\dots x_t \mid x_i \in \Sigma \text{ for } 1 \leq i \leq t\} \end{aligned}$$

denotes the set of all strings on  $\Sigma$  having length  $t$ .

The *length* of a string  $x$ , denoted by  $|x|$ , is the number of alphabet elements it contains. Given a string  $x$  of length  $n$ , we denote by  $x[i]$ , for  $i \in [0, n)$ , the character of a string  $x$  at the position  $i$  or at *index*  $i$  of  $x$  assuming that we are dealing with a *zero-indexed* string, that is a string with the first character positioned at index zero instead of one.

We can denote a string  $S$  of length  $n$  using either index separated by comma notation or *index-range* notation. In the first case, we mark the starting index and ending index of  $S$  separated by a comma. In the second case, we write the starting index and ending index of  $S$  separated by the range symbol.

**Example 2.2.** Given a string  $S$  of length  $n$ , we denote it as

$$S = S[1, n] = S[1]S[2] \dots S[n] = S[1 \dots n],$$

if it starts at index 1 and ends at index  $n$ . We can express zero-indexed strings  $T$  of length  $n$  as follows:

$$T = T[0, n - 1] = T[0]T[1] \dots T[n - 1] = T[0 \dots n).$$

Given a string  $S[0, n)$  with  $n$  characters,  $S[i \dots j] \subset S$  denotes the *substring* of  $S$  that starts at position  $i$  and ends at position  $j$ , such that  $i \leq j$ . When  $i > j$ ,  $S[i \dots j]$  denotes the empty string  $\varepsilon$ .

**Example 2.3.** Suppose we are given a string  $T[0, 11] = \text{abracadabra\$}$ . To define a substring **braca** we provide its starting and ending character indices within the array  $T$ , that is  $T[1, 4] = \text{braca}$ .

Let  $x$  be a string of length  $n$  and let  $y$  be a string of length  $m$ . The *concatenation* of  $x$  and  $y$ , denoted by  $xy$ , is the string formed by merging  $x$  and  $y$ , as in  $x_1 \dots x_n y_1 \dots y_m$ . We write  $x^n$  to denote that string  $x$  is concatenated with itself  $n$  times, that is,

$$x^0 = \varepsilon \text{ and } x^n = \overbrace{xx \dots x}^n.$$

**Example 2.4.** Let  $x = \text{abra}$  and  $y = \text{cadabra}$ . The concatenation of  $x$  and  $y$  is  $xy = \text{abracadabra}$ . To obtain  $x^2$  we append  $x$  to the end of itself, that is,  $x^2 = \text{abraabra}$ .

Given a string  $S[1 \dots n]$  with  $n$  symbols, a substring  $S[i \dots n] \subset S$  is a *suffix* of  $S$  that starts at index  $i$  and ends at the end of  $S$ . A *proper suffix* of the string  $S$  is any suffix of  $S$  such that  $1 < i \leq n$ , that is, any non-empty suffix of  $S$  that is not equal to  $S$ . A *prefix* of  $S$  is a substring  $S[1 \dots i] \subset S$  that starts from the beginning of  $S$  and ends at index  $i$ . A *proper prefix* of the string  $S$  is any prefix of  $S$  such that  $1 \leq i < n$ , that is, any non-empty prefix of  $S$  that is not equal to  $S$  [51].

**Example 2.5.** Suffixes and prefixes of the string  $T = \text{abracadabra}\$$ .

Suffix	Prefix	Index
abracadabra\$	a	0
bracadabra\$	ab	1
racadabra\$	abr	2
acadabra\$	abra	3
cadabra\$	abrac	4
adabra\$	abraca	5
dabra\$	abracad	6
abra\$	abracada	7
bra\$	abracadab	8
ra\$	abracadabr	9
a\$	abracadabra	10
\$	abracadabra\$	11

In Example 2.5, you can see that a suffix of the string  $T$  starts from the corresponding index and includes all characters in between until and including the last character. Thus, if we consider *suffix* 5, **adabra\$**, we can express it as  $T[5 \dots 11]$ . Observe that the prefix is reciprocal in some way to the suffix. Suffix index tells us that it starts from that index and spans over all characters until and including the last character. The same index tells us that the prefix is a substring that starts from the first index and ends on the suffix index. Hence, *prefix* 5, **abraca** can be defined as  $T[0 \dots 5]$ .

The *lexicographic ordering*, denoted by  $<$ , is a total ordering on the alphabet  $\Sigma$  that induces the ordering on  $\Sigma^*$  as follows. Given two strings  $s, t \in \Sigma^*$ , we say that  $s$  is lexicographically smaller than  $t$ , written as  $s < t$ , if and only if either  $s$  is a proper prefix of  $t$  or we can express the first string as  $s = xcy$  and the second string as  $t = xdz$  with  $x, y, z \in \Sigma^*$  and characters  $c, d \in \Sigma$  such that  $c < d$  [15, 70].

**Example 2.6.** Suppose we have three strings:  $x = acacc$ ,  $y = acca$  and  $z = accaac$ . To determine the lexicographic ordering of these strings we compare their first characters index-wise. Assume that  $a < c$ . The first two characters of all three strings are the same, so we need to look at the third character. We see that in string  $x$  this character is  $a$  which is smaller than in the other two strings that have character  $c$  at that index. This tells us that  $x$  is lexicographically the smallest among three strings. Next, we compare  $y$  and  $z$ . The first four characters are the same. However, we notice that  $y$  is a proper prefix of  $z$ . Hence the lexicographic ordering is  $x < y < z$ .

## 2.2 The Suffix Array

Let  $x[0 \dots n]$  be a string of length  $n + 1$  over an *indexed* alphabet  $\Sigma$  (i.e. an alphabet where each symbol is associated with an integer that belongs to some limited range). We assume that  $x[n] = \$$  is a special symbol called the *sentinel* that is usually marked with a dollar sign  $\$$  such that  $\$$  is lexicographically smaller than any other symbol in  $\Sigma$  [18].

We denote the  $i$ -th suffix of  $x$  as  $x_i = x[i \dots n]$  or *suffix  $i$* . The *suffix array* of the string  $x$ , written as  $SA_x$  or just  $SA$ , is an array of integers  $SA[0 \dots n]$ , which stores a permutation of the integers  $\{0, \dots, n\}$ , such that

$$x_{SA[0]} < x_{SA[1]} < \dots < x_{SA[n]}.$$

To put it another way,  $x_i$  is the  $j$ -th smallest suffix of the string  $x$  in ascending lexicographical order if and only if  $SA[j] = i$  [64].

**Example 2.7.** The suffix array of the string  $x = abracadabra\$$ .

Suffix	$i$	Sorted suffix	$SA[i]$
abracadabra\$	0	\$	11
bracadabra\$	1	a\$	10
racadabra\$	2	abra\$	7
acadabra\$	3	abracadabra\$	0
cadabra\$	4	acadabra\$	3
adabra\$	5	adabra\$	5
dabra\$	6	bra\$	8
abra\$	7	bracadabra\$	1
bra\$	8	cadabra\$	4
ra\$	9	dabra\$	6
a\$	10	ra\$	9
\$	11	racadabra\$	2

Example 2.7 tells us that  $x_{11}$  is the 0-th smallest suffix,  $x_{10}$  is the 1-th smallest suffix, and so on (assuming that we use zero-indexed string and lexicographical order of the characters). Notice that  $|x| = |\text{SA}|$  and that SA is the permutation of the set  $\{0, \dots, 11\}$ .

The *inverse suffix array* of the string  $x$ , written as  $\text{ISA}_x$  or just ISA, is an array of integers  $\text{ISA}[0 \dots n]$ , so that for any  $i$  with  $0 \leq i \leq n$  the equality  $\text{ISA}[\text{SA}[i]] = i$ . More precisely, the relation between SA and ISA can be expressed as follows:

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

The inverse suffix array is also called the *lexicographic ranks* of suffixes because  $\text{ISA}[i]$  specifies the rank of the  $i$ -th suffix among the lexicographically ordered suffixes [74]. The expression  $\text{ISA}[i] = j$  implies that suffix  $x_i$  is the  $j$ -th smallest suffix of the string  $x$  in ascending lexicographical order [51, 64]. We can compute SA and ISA, one from the other, in linear time as follows:

$$\text{SA}[\text{ISA}[i]] = i, \quad 0 \leq i \leq n \quad \text{and} \quad \text{ISA}[\text{SA}[j]] = j, \quad 0 \leq j \leq n.$$

**Example 2.8.** The suffix array and the inverse suffix array of the string  $x = \text{abracadabra}\$$ .

Suffix	$i$	Sorted suffix	$\text{SA}[i]$	$\text{ISA}[i]$
abracadabra\$	0	\$	11	3
bracadabra\$	1	a\$	10	7
racadabra\$	2	abra\$	7	11
acadabra\$	3	abracadabra\$	0	4
cadabra\$	4	acadabra\$	3	8
adabra\$	5	adabra\$	5	5
dabra\$	6	bra\$	8	9
abra\$	7	bracadabra\$	1	2
bra\$	8	cadabra\$	4	6
ra\$	9	dabra\$	6	10
a\$	10	ra\$	9	1
\$	11	racadabra\$	2	0

Example 2.8 tells us that  $x_0$  is the 3-th smallest suffix,  $x_1$  is the 7-th smallest suffix, and so on. To compute SA from ISA we take the following steps:  $\text{SA}[\text{ISA}[0]] = \text{SA}[3] = 0$ ,  $\text{SA}[\text{ISA}[1]] = \text{SA}[7] = 1$ , and so on. To compute ISA from SA we take the following steps:  $\text{ISA}[\text{SA}[0]] = \text{ISA}[11] = 0$ ,  $\text{ISA}[\text{SA}[1]] = \text{ISA}[10] = 1$ , and so on. Obviously, we can compute SA from ISA and vice versa in linear time.

## 2.3 The Burrows-Wheeler Transform

The *Burrows-Wheeler Transform* (BWT) [12] is a process of transforming an input string into another string that has a structure suitable for efficient compression. A remarkable property of this structure is that a string generated by the BWT tends to contain a large number of substrings composed of consecutive identical characters. To take advantage of this property and efficiently code a string generated by the BWT, a *Move-to-Front* (MTF) [9, 76] and a *Huffman* [31, 87] or *arithmetic* [93] coders are used.

The BTW is reversible, which enables a recovery of the original string with minimal data overhead. The BWT of a string  $x[0 \dots n]$  is generated as follows: (1) Append a sentinel symbol  $\$$  to the end of the string  $x$  such that  $x[n] = \$$  (2) Construct a  $(n + 1) \times (n + 1)$  matrix  $R$  containing all rotations (cyclic shifts) of  $x$ . (3) Sort rotations in  $R$  lexicographically. (4) The last column of  $R$  is the output of the BWT [12, 71].

**Example 2.9.** Application of the BWT with input string  $x = \text{abracadabra}$ .

$i$	Rotations ( $R$ )	Sorted rotations	BWT[ $i$ ]
0	abracadabra\$	\$abracadabra	a
1	bracadabra\$a	a\$abracadabr	r
2	racadabra\$ab	abra\$abracad	d
3	acadabra\$abr	abracadabra\$	\$
4	cadabra\$abra	acadabra\$abr	r
5	adabra\$abrac	adabra\$abrac	c
6	dabra\$abraca	bra\$abracada	a
7	abra\$abracad	bracadabra\$a	a
8	bra\$abracada	cadabra\$abra	a
9	ra\$abracadab	dabra\$abraca	a
10	a\$abracadabr	ra\$abracadab	b
11	\$abracadabra	racadabra\$ab	b

Observe that when cyclic shifts are applied on the string **abracadabra\$**, the sentinel symbol **\$** shifts from right to left until it reaches the first position in the last rotation. The output of the transformation is **ard\$rcaaaabb**. Although the length of the input string is small, the structure of the output string contains substrings of consecutive identical characters, which are easy to compress using the coders mentioned at the beginning of this chapter. In practice, input strings of sufficiently large length are likely to generate

more substrings of consecutive identical characters, as a result of the BTW transformation, and hence the compression efficiency tends to increase.

The efficiency of the original approach for computing the BTW of a string boils down to the efficiency of sorting lexicographically the rotations of the *Burrows-Wheeler Matrix* (BWM). In other words, sorting the rotations is a bottleneck in the BWT with respect to the time complexity of this operation. A naive approach is to sort the rotations using a comparison-based sorting algorithm such as *quicksort* or *mergesort*. However, a more sophisticated and efficient approach is to employ the suffix array to solve the problem of sorting the rotations [3, 46].

Observe that the sorted rotations resemble the sorted suffixes obtained from the suffix array of the string  $x = \text{abracadabra\$}$ . We notice in Example 2.10 that each sorted rotation can be obtained by extracting a corresponding sorted suffix of  $x$  and prepending it to the beginning of  $x$ .

**Example 2.10.** The relation between the BWT and the SA with input string  $x = \text{abracadabra\$}$ .

$i$	Sorted rotations	Sorted suffixes of $x$	SA[ $i$ ]	BWT[ $i$ ]
0	\$abracadabra	\$	11	a
1	a\$abracadabr	a\$	10	r
2	abra\$abracad	abra\$	7	d
3	abracadabra\$	abracadabra\$	0	\$
4	acadabra\$abr	acadabra\$	3	r
5	adabra\$abrac	adabra\$	5	c
6	bra\$abracada	bra\$	8	a
7	bracadabra\$a	bracadabra\$	1	a
8	cadabra\$abra	cadabra\$	4	a
9	dabra\$abraca	dabra\$	6	a
10	ra\$abracadab	ra\$	9	b
11	racadabra\$ab	racadabra\$	2	b

The relation between the BWT and the SA can be formulated as follows:

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

This relation implies that BWT[ $i$ ] of the string  $x$  can be constructed by taking a character that is positioned to the left of the suffix in the suffix array SA[ $i$ ].

## 2.4 Prior Work

In this section, we give a short review of the most notable SACAs and classify them according to three categories: sequential implementations, parallel CPU implementations and massively parallel GPU implementations. The sequential implementation class of algorithms includes all SACAs that run in a single thread and execute CPU instructions sequentially. A class of parallel CPU implementations includes SACAs that leverage multi-core CPU parallelism and shared-memory data access. These are multi-threaded SACAs that can be powered by some application programming interface (API) that supports multi-platform shared memory multiprocessing programming. Finally, the last class includes SACAs developed for massively parallel shared-memory GPU architectures.

### Sequential Implementations

The concept of the suffix array was introduced in 1990 by Manber and Myers (MM) [53] along with the first  $O(n \log n)$  [47] suffix sorting algorithm, which consumed less space than alternative suffix tree implementations. Their work was motivated by the results of Karp *et al.* in 1972 [38] who was working on pattern matching in strings using array and tree data structures. The MM algorithm was later improved and optimized by Larsson and Sadakane (LS) [45]. Both the MM and the LS algorithms used *prefix-doubling* technique. According to [74], the LS algorithm was ten times faster than the MM algorithm in practice.

In 2003, independent groups of researchers developed different linear-time SACAs that leveraged recursive *divide-and-conquer* principles. Among them the most prominent are: Ko and Aluru (KA) [42]; Kärkkäinen and Sanders (KS) [37]; Kim *et al.* (KSPP) [39] and Hon *et al.* (HSS) [30].

A plethora of SACAs of different time and space complexities based on *induced sorting* method have been proposed among which are a few notable ones: Itoh and Tanaka (IT, 1999) [33]; Seward (S, 2000) [80]; Burkhardt and Kärkkäinen (BK, 2003) [11]; Manzini and Ferragina (MF, 2004) [56]; Schürmann and Stoye (SS, 2005) [79]; Baron and Bresler (BB, 2005) [5]; Maniscalco and Puglisi (MP, 2007) [55]; Nong *et al.* (SA-IS, 2009) [67]. In 2010, Yuta Mori improved the performance of Nong *et al.* SA-IS algorithm and released an open-source version in a public repository [62].

In 2018, Li *et al.* [47] obtained the first in-place linear-time SACA that takes constant workspace for read-only input strings over integer alphabets, where  $|\Sigma| = O(n)$ . They improved the result of Nong [66] by extending the

alphabet size to  $O(n)$  for in-place sorting and by reducing the workspace to  $O(1)$ . Li *et al.* in-place linear-time SACA is based on the induced sorting framework developed in [42] (which is also used in the following SACAs [23, 66–69, 74]).

## Parallel CPU Implementations

In 2009, Homann *et al.* [29] introduced an open-source program for constructing enhanced suffix arrays *mkESA*. The *mkESA* was based on Manzini and Ferragina [56] *Deep-Shallow* SACA that Homann *et al.* managed to map onto a multi-core CPU architecture. The authors achieved a speedup of less than 2x using 16 threads against single threaded execution of *mkESA*.

In 2010, Mohamed and Abouelhoda [60] proposed a parallelized version of the *the bucket pointer refinement (bpr)* algorithm of Schürmann and Stoye [79]. Their algorithm the *pbpr* incorporates Seward’s [80] method and they tested it on 8 core machine. Mohamed and Abouelhoda claim that the *pbpr* outperforms the *mkESA*. Their results show that the *pbpr* algorithm achieves less than 1.7x speedup using eight threads [89].

One of the fastest, robust and lightweight SACAs of this category was implemented by Yuta Mori. Mori’s algorithm, *libdivsufsort* (sometimes referred as *divsufsort*), is based on induced sorting technique and parallelized by the OpenMP application programming interface (API) [72]. Mori evaluated the performance of its implementation in the benchmarking contest [63]. A plethora of SACAs participated in this contest. Among them two algorithms were parallelized for shared memory multiprocessing: the *Archon4* algorithm by Dmitry Malyshev [52] and the *MSufSort3* algorithm by Michael Maniscalco [54]. All participating in the contest algorithms were extensively tested over different corpora. Mori’s algorithm showed the best total running time. Fischer and Kurpicz gave a concise description of *libdivsufsort* by dismantling the source code that has never been documented in an academic context. According to Fischer and Kurpicz [21, 22], *libdivsufsort* is still the fastest and the most space-conscious way to construct the suffix array on the CPU.

In 2012, Shun *et al.* [81] released an open-source Problem Based Benchmark Suite (PBBS), which includes two shared memory parallel implementations of the *skew* [37] and prefix-doubling [45] algorithms. Both SACAs from the PBBS are implemented using Cilk Plus extension [32], which is deprecated since 2018 and requires a separate installation on the machine. Authors claim that both versions of their implementations are faster than *libdivsufsort*.

Recently, Nong *et al.* [44] developed the first linear-time and in-place

parallel suffix sorting algorithm for constant alphabets the *pSACAK*, which is essentially a parallelized version of the *SACAK* [66] algorithm. Additionally, Nong *et al.* [43] described the parallelization process of the *sais* [67] SACA. They called a parallelized version the *psais*. Both, the *pSACAK* and the *psais* algorithms were developed using Cilk Plus extension. Their results suggest that the *pSACAK* runs 34 percent faster than *psais* and consumes 5 percent less memory. Authors compared their implementations against the *pKS* algorithm (parallel version of the *skew* [37] algorithm from PBBS) and against *libdivsufsort*. According to their results, the *pSACAK* is just 5 percent faster than the *pKS*, but uses 4 times less memory. The *pSACAK* achieves an average speedup of 2.85x over *libdivsufsort* and requires up to 0.3 percent more space. It remains uncertain whether the authors ran *libdivsufsort* with maximum number of OpenMP threads, since they referred to Mori’s algorithm as sequential.

## Massively Parallel GPU Implementations

In 2013, Deo and Keely [17] mapped a recursive *skew* algorithm onto GPUs. They implemented a GPU version of the *skew* algorithm using OpenCL and achieving up to 34x and 5.8x speedup over a single threaded CPU implementation using a discrete GPU and *accelerated processing unit* (APU) respectively. In their work, Deo and Keely also described the GPU implementation of Kasai *longest common prefix* (LCP) algorithm. Their GPU implementation of LCP achieves a speedup of up to 25x and 4.3x on discrete GPU and APU respectively.

Deo and Keely conclude that the *skew* algorithm is the best candidate for GPU implementation, because, as they claim, all phases of the algorithm can be efficiently parallelized using GPU programming paradigms. On the other hand, Deo and Keely claim that induce sorting SACA can be efficiently mapped onto GPUs only if data-dependency issue will be solved by finding an efficient way to parallelize between different phases and finding a way how to sort strings of irregular size efficiently. They also consider prefix-doubling to be unsuitable for GPU because of the increasing number of unsorted buckets in each level, non-uniform amount of work in each unsorted bucket and difficulties in managing the increasing number of unsorted buckets.

In 2012, Vitaly Osipov [73] implemented a prefix doubling SACA on the GPU. He claimed that to his best knowledge it was the first prefix-doubling SACA mapped onto GPU. According to Osipov, his GPU algorithm is the result based on a few modifications applied to MM and LS algorithms. In particular, in his GPU solution, he reduced the number of radix sort passes to one and applied a filtering criterion to eliminate large overheads caused

by re-sorting.

Osipov also compared the running time of his algorithm against *libdivsuf-sort*. Osipov’s results suggest that his algorithm achieved around 6x speedup only on one dataset, while on other datasets the speedup over *libdivsuf-sort* was up to 4. Osipov stated that he ran *libdivsuf-sort* on 4 core CPU. He concludes that prefix-doubling algorithms are better suitable for mapping onto GPUs compared to recursive algorithms like the *skew*. He justifies his opinion by claiming that sorting and merging in the *skew* are more expensive, while prefix-doubling requires only efficient GPU sorting of four-byte key-value pairs.

In 2015, Wang *et al.* [89] proposed and developed two SACAs on the GPU. They implemented the *skew* algorithm on the GPU with some algorithmic modifications and improvements over Deo and Kelly’s algorithm achieving a speedup of 1.45x over their work. In addition to that, they also implemented a hybrid GPU SACA that combined both the *skew* and prefix-doubling approaches in its design and outperformed Osipov’s algorithm by 2.3-4.4x, and their own GPU *skew* implementation by 2.4-7.9x on large datasets. Wang *et al.* claim that their hybrid algorithm is the first in its kind. Performance improvement over Deo and Kelly’s *skew* algorithm and Osipov’s prefix-doubling algorithm was achieved by taking advantage of parallel primitives such as a *merge* and a *segmented sort* [6].

## Chapter 3

# Algorithms for Suffix Array Construction

In this chapter, we study three main classes of algorithms that are used for suffix array construction. Each class of such algorithms applies a specific technique to succeed in its task. We analyze the main ideas behind each technique and investigate how they are implemented by some of the most prominent SACAs of each class.

The prefix-doubling technique was first introduced and applied for suffix array construction by Manber and Myers [53] in 1990. Their work was inspired and motivated by Karp *et al.* [38] who proposed the idea of the technique in 1972. Later, Larsson and Sadakane [45] optimized the MM algorithm achieving around 10x speedup without applying any parallelism [74].

In 2003 and 2004, several novel linear-time recursive suffix array construction algorithms were developed. They are Kärkkäinen and Sanders' [37] algorithm, Ko and Aluru's [42] algorithm, Kim *et al.*'s [39] algorithm and Hon *et al.*'s [30] algorithm. They are based on a similar recursive divide-and-conquer model.

The first algorithm based on induced sorting technique was proposed by Itoh and Tanaka [33] in 1999. From 2000 until 2006, other induced sorting SACAs were developed. They are Seward's [80] algorithm, Burkhardt and Kärkkäinen's [11] algorithm, Manzini and Ferragina's [56] algorithm, Schürmann and Stoye's [79] algorithm, Baron and Bresler's [5] algorithm and Maniscalco and Puglisi's [55] algorithm. In 2009, Nong *et al.*'s [67] developed fast and lightweight induced sorting SACA, which was later improved and optimized without any parallelism by Mori [62].

## 3.1 Prefix-Doubling Algorithms

The idea of the prefix-doubling approach is to deduce the order of suffixes from the lexicographic ranks of their prefixes. Initially, we sort suffixes lexicographically by their prefixes of length one. Essentially, this means that we sort lexicographically each suffix according to the first character. Then, we group suffixes that start with identical character into *buckets*.

A bucket that contains only one suffix is called *singleton* and considered sorted. A bucket that contains more than one suffix is called *non-singleton* and requires sorting.

In each round, we double the prefix length of each suffix in all non-singleton buckets, ignoring singleton ones. Then we use the ranks of prefixes from the previous round to obtain the ranks of prefixes in the current round and to sort suffixes in each non-singleton bucket according to the new ranks of their prefixes. In other words, we use the relative order of suffixes computed in round  $t$  to deduce their order in round  $t + 1$ . Once the suffix can be uniquely distinguished by the rank of the corresponding prefix, its position in the suffix array is fixed, and it is marked as a singleton. The doubling process terminates when all buckets are singleton.

We say that suffixes are in  $h$ -order if they are sorted lexicographically according to their first  $h$  symbols. The  $h$ -bucket contains suffixes in their  $h$ -order. Suffixes in  $h$ -order may have the same ranks when the value of  $h$  is relatively small compared to the string length.

The length of the prefix is defined by  $h$ -order of suffixes with all singleton buckets representing suffixes that are already sorted by their  $h$ -length prefixes. The advantage of this method is that once we know the  $h$ -order of suffixes we can determine their positions within  $2h$ -buckets in time  $O(n)$  [89].

When we scan suffixes in their  $h$ -order, we deduce the order of suffix  $i$  from the lexicographic rank of its  $i + h$  prefix computed in round  $h - 1$  (where round 0 is the initial single character sorting). In each iteration, we double  $h$ , that is  $h = 2^k$  for  $k \in \{0, 1, 2, \dots\}$ . It takes at most  $O(\log n)$  rounds until every  $2h$ -bucket is a singleton. Hence, the total running time of this method is  $O(n \log n)$ .

### 3.1.1 Manber and Myers Algorithm

The Manber and Myers (MM) suffix sorting algorithm runs in  $\lceil \log_2(n + 1) \rceil$  stages. In each stage, it performs an implicit  $2h$ -sort by scanning buckets from left to right. This operation takes  $O(n)$  time. To implicitly sort the

suffixes, Manber and Myers leverage the following idea.

Consider two suffixes  $i$  and  $j$  that reside in the same bucket. We want to determine their  $h$ -order. For this purpose, we need to compare lexicographically their  $i + h$  and  $j + h$  prefixes. We assume that we know the relative order of suffixes  $i + h$  and  $j + h$ . Hence, we can use their order to sort suffixes  $i$  and  $j$ . This idea leads to the following observation:

**Observation 1.** *Denote by  $SA_h$  the suffix array in  $h$  stage. Let  $i \in [a, b]$  be the position of the suffix that belongs to the bucket occupying the interval  $[a, b]$  in  $SA_h$ . As we traverse  $SA_h$  left to right, then each  $h$ -order suffix*

$$SA_h[i] - h > 0$$

*defines the  $2h$ -order of suffixes within the corresponding  $h$ -buckets.*

This observation means the following. Consider the first bucket, as we scan  $SA_h$  left to right, and let suffix  $i$  be the first suffix in this bucket, such that  $i - h \geq 0$ . Since suffix  $i$  is ranked first in its  $h$ -bucket, then suffix  $i - h$  should be ranked first in its  $2h$ -bucket, because the next  $h$  symbols of suffix  $i - h$  are the first  $h$  symbols of suffix  $i$ . Next, we move suffix  $i - h$  to the first position in its  $h$ -bucket. For each suffix  $i$ , the algorithm moves suffix  $i - h$  to the next available position in its  $h$ -bucket.

The detailed pseudo-code of the MM implementation is available in Algorithm 1. Initially, we create two boolean arrays  $B_h$  and  $B_{2h}$ , and three integer arrays  $SA$ ,  $Cnt$ , and  $ISA$ . The size of all arrays is  $n$ .  $SA$  is the suffix array that stores suffixes in  $h$ -order.  $ISA$  is the inverse of  $SA$ , defined as  $ISA[SA[i]] = i$  for all  $i \in [0, n]$ .  $B_h$  marks the bucket heads (i.e. helps to identify left and right boundaries of  $h$ -bucket).  $B_{2h}$  marks prefixes that were moved.  $Cnt$  stores the next available position in the bucket that is being currently scanned. The first prefix is moved to the top of its  $h$ -bucket,  $Cnt[i]$  is incremented, and the next prefix goes to the second position and so on.

In line 4 of Algorithm 1, we perform the initialization of  $SA$ ,  $B_h$  and  $B_{2h}$ . We sort suffixes of the input string  $T = T[0]T[1] \dots T[n-1]$  according to the first symbol and store the result into  $SA$ . This procedure can be done by running key-value radix sort, where elements of  $T$  are keys and elements of  $SA$  are values. Initially, we fill  $SA$  with consecutive integers from  $[0, n)$ . The initial values of  $B_h$  are computed by comparing lexicographically the first symbol of the adjacent  $h$ -order suffixes, namely,  $T[SA[i-1]]$  and  $T[SA[i]]$ . The initial values of  $B_{2h}$  are set to 0.

After initialization, the algorithm enters the main **while** loop (line 6). It executes until all buckets are fully sorted, that is, each bucket is a singleton. This condition is satisfied when all values of  $B_h$  are set to 1 or when the

number of buckets is  $n$ . At the beginning of each iteration, we compute the ranks of suffixes in  $h$ -order, namely,  $ISA[SA[i]] = i$  and set  $Cnt[i]$  to 0 for all  $0 \leq i < n$ .

In line 15, we begin the traversal of  $SA$  left to right. Let  $[a, b]$  be the interval occupied by the  $h$ -bucket that is currently being scanned within  $SA$ . We denote by  $s$  the rank of every suffix obtained from  $SA[i] - h \geq 0$ . For every  $i$ ,  $a \leq i \leq b$ , we set  $ISA[s] = ISA[s] + Cnt[s]$ , increment  $Cnt[s]$ , and set  $B_{2h}$  to 1. All suffixes that are uniquely distinguished by the ranks of their  $h$ -prefixes are moved to the position defined by  $Cnt$  within the corresponding  $h$ -bucket.

Before we move to the next bucket, we scan the current bucket again and update the values of  $B_{2h}$  as shown in line 25. For all suffixes that were moved, we identify the leftmost suffix and use its position to mark the head of its  $2h$ -bucket, while resetting the  $B_{2h}$  values for other suffixes within the same  $h$ -bucket. More precisely, we set  $B_{2h}$  to 0 for all  $v \in [ISA[s] + 1, u - 1]$  such that the position of every moved suffix  $s$  is marked in  $B_{2h}$  and

$$u = \min\{t : t > ISA[s] \text{ and } (B_h[t] \text{ or not } B_{2h}[t])\}.$$

This expression implies that we identify the left boundary of each  $2h$ -bucket, while the right boundary is preserved by  $B_h$ . Below, we provide a running example of the MM implementation available in the Algorithm 1 using the input string  $T = \text{abracadabra}\$,$  where  $\$$  is the sentinel. At the beginning of stage  $h = 1$ , we have:

$i$	$B_h[i]$	$B_{2h}[i]$	$Cnt[i]$	$ISA[i]$	$SA[i]$
0	1	1	1	1	11 = \$
1	1	0	0	6	0 = abracadabra\$
2	0	0	0	10	3 = acadabra\$
3	0	0	0	1	5 = adabra\$
4	0	0	0	8	7 = abra\$
5	0	0	0	1	10 = a\$
6	1	0	0	9	1 = bracadabra\$
7	0	0	0	1	8 = bra\$
8	1	0	0	6	4 = cadabra\$
9	1	0	0	10	6 = dabra\$
10	1	0	0	1	2 = racadabra\$
11	0	0	0	0	9 = ra\$

At this point,  $SA[i]$  is partitioned into 6 buckets according to the first symbol of each suffix:  $\$, a, b, c, d, r$ . Suffixes in  $SA[i]$  for  $i = \{0, 8, 9\}$  are already

sorted and hence their buckets are singleton. We scan  $SA[i]$ , consider one  $h$ -bucket at time and perform the following operations:

---

**Algorithm 1:** The MM algorithm

---

```

1  procedure MM( $T, n$ )
   Input:  $T$  — input string,  $n$  — input string length.
   Output: Sorted lexicographically suffixes of  $T$  stored in  $SA$ .
2  Define integer arrays:  $SA(n)$ ,  $ISA(n)$ ,  $Count(n)$ 
3  Define boolean arrays:  $B_h(n)$ ,  $B_{2h}(n)$ 
4  Initialize*  $SA$ ,  $B_h$  and  $B_{2h}$ 
5   $h \leftarrow 1$ 
6  while  $\exists$  non-singleton  $h$ -bucket do
7      for each bucket $[a, b]$ † do
8           $Count[a] \leftarrow 0$ 
9          for  $i \in [a, b]$  do
10              $ISA[SA[i]] \leftarrow a$ 
11         end
12     end
13      $Cnt[ISA[n-h]] \leftarrow Cnt[ISA[n-h]] + 1$ 
14      $B_{2h}[ISA[n-h]] \leftarrow true$ 
15     for each bucket $[a, b]$  do
16         for  $i \in [a, b]$  do
17              $s \leftarrow SA[i] - h$ 
18             if  $s \geq 0$  then
19                  $head \leftarrow ISA[s]$ 
20                  $ISA[s] \leftarrow head + Cnt[head]$ 
21                  $Cnt[head] \leftarrow Cnt[head] + 1$ 
22                  $B_{2h}[ISA[s]] \leftarrow true$ 
23             end
24         end
25         for  $i \in [a, b]$  do
26              $s \leftarrow SA[i] - h$ 
27             if  $s \geq 0$  and  $B_{2h}[ISA[s]]$  then
28                  $u \leftarrow \min\{t : t > ISA[s] \text{ and } (B_h[t] \text{ or not } B_{2h}[t])\}$ 
29                 for  $v \in [ISA[s] + 1, u - 1]$  do
30                      $B_{2h}[v] \leftarrow false$ 
31                 end
32             end
33         end
34     end
35     for  $i \in [0, n - 1]$  do
36          $SA[ISA[i]] \leftarrow i$ 
37          $B_h[i] \leftarrow B_h[i]$  or  $B_{2h}[i]$ 
38     end
39      $h \leftarrow 2h$ 
40 end
41 for  $i \in [0, n - 1]$  do
42      $ISA[SA[i]] \leftarrow i$ 
43 end
44 return  $SA$ 
45 End

```

---

\*Initialization is explained in the text.

<sup>†</sup>We denote by bucket $[a, b]$  the bucket that is currently being scanned in  $SA_h$ , where  $a$  is the left and  $b$  is the right boundaries of this bucket.

**Scanning bucket 1:**  $T_{10}$  is moved to the position 1 of its bucket,  $SA[0] - 1 = 10$ ,  $ISA[10] = 1$ , set  $ISA[10] = 1 + Cnt[1]$ ,  $B_{2h}[1] = 1$  and increment  $Cnt[1]$ . **Scanning bucket 2:**  $T_2$  is moved to the position 10 of its bucket,  $SA[2] - 1 = 2$ ,  $ISA[2] = 10$ , set  $ISA[2] = 10 + Cnt[10]$ ,  $B_{2h}[10] = 1$  and increment  $Cnt[10]$ .  $T_4$  is moved to the position 8 of its bucket,  $SA[3] - 1 = 4$ ,  $ISA[4] = 8$ , set  $ISA[4] = 8 + Cnt[8]$ ,  $B_{2h}[8] = 1$  and increment  $Cnt[8]$ .  $T_6$  is moved to the position 9 of its bucket,  $SA[4] - 1 = 6$ ,  $ISA[6] = 9$ , set  $ISA[6] = 9 + Cnt[9]$ ,  $B_{2h}[9] = 1$  and increment  $Cnt[9]$ .  $T_9$  is moved to the position 11 of its bucket,  $SA[5] - 1 = 9$ ,  $ISA[9] = 10$ , set  $ISA[9] = 10 + Cnt[10]$ ,  $B_{2h}[11] = 1$  and increment  $Cnt[10]$ .  $B_{2h}[11]$  is set to 0 (by scanning the bucket again). **Scanning bucket 3:**  $T_0$  is moved to the position 2 of its bucket,  $SA[6] - 1 = 0$ ,  $ISA[0] = 1$ , set  $ISA[0] = 1 + Cnt[1]$ ,  $B_{2h}[2] = 1$  and increment  $Cnt[1]$ .  $T_7$  is moved to the position 3 of its bucket,  $SA[7] - 1 = 7$ ,  $ISA[7] = 1$ , set  $ISA[7] = 1 + Cnt[1]$ ,  $B_{2h}[3] = 1$  and increment  $Cnt[1]$ .  $B_{2h}[3]$  is set to 0 (by scanning the bucket again). **Scanning bucket 4:**  $T_3$  is moved to the position 4 of its bucket,  $SA[8] - 1 = 3$ ,  $ISA[3] = 1$ , set  $ISA[3] = 1 + Cnt[1]$ ,  $B_{2h}[4] = 1$  and increment  $Cnt[1]$ . **Scanning bucket 5:**  $T_5$  is moved to the position 5 of its bucket,  $SA[9] - 1 = 5$ ,  $ISA[5] = 1$ , set  $ISA[5] = 1 + Cnt[1]$ ,  $B_{2h}[5] = 1$  and increment  $Cnt[1]$ . **Scanning bucket 6:**  $T_1$  is moved to the position 6 of its bucket,  $SA[10] - 1 = 1$ ,  $ISA[1] = 6$ , set  $ISA[1] = 6 + Cnt[6]$ ,  $B_{2h}[6] = 1$  and increment  $Cnt[6]$ .  $T_8$  is moved to the position 7 of its bucket,  $SA[11] - 1 = 8$ ,  $ISA[8] = 6$ , set  $ISA[8] = 6 + Cnt[6]$ ,  $B_{2h}[7] = 1$  and increment  $Cnt[6]$ .  $B_{2h}[7]$  is set to 0 (by scanning the bucket again). At the end of stage  $h = 1$ , we have:

$i$	$B_h[i]$	$B_{2h}[i]$	$Cnt[i]$	$ISA[i]$	$SA[i]$
0	1	1	1	2	11 = \$
1	1	1	5	6	10 = a\$
2	1	1	0	10	0 = abracadabra\$
3	0	0	0	4	7 = abra\$
4	1	1	0	8	3 = acadabra\$
5	1	1	0	5	5 = adabra\$
6	1	1	2	9	1 = bracadabra\$
7	0	0	0	3	8 = bra\$
8	1	1	1	7	4 = cadabra\$
9	1	1	1	11	6 = dabra\$
10	1	1	2	1	2 = racadabra\$
11	0	0	0	0	9 = ra\$

Notice that during  $h$ -bucket scan, we derived new positions for suffixes  $T_{i-h}$  and stored them into  $ISA[i]$ . At the end of the stage, we performed the actual

moving, which is demonstrated in Algorithm 1 in lines 35-38. Additionally, we merged the values of  $B_h$  and  $B_{2h}$  by performing logical OR operation between them and storing the results into  $B_h$ .

At the beginning of stage  $h = 2$ , we have:

$i$	$B_h[i]$	$B_{2h}[i]$	$Cnt[i]$	$ISA[i]$	$SA[i]$
0	1	1	0	2	11 = \$
1	1	1	1	6	10 = a\$
2	1	1	0	10	0 = abracadabra\$
3	0	0	0	4	7 = abra\$
4	1	1	0	8	3 = acadabra\$
5	1	1	0	5	5 = adabra\$
6	1	1	0	9	1 = bracadabra\$
7	0	0	0	2	8 = bra\$
8	1	1	0	6	4 = cadabra\$
9	1	1	0	10	6 = dabra\$
10	1	1	0	1	2 = racadabra\$
11	0	0	0	0	9 = ra\$

The number of buckets increased from 6 to 9. From the previous stage the algorithm successfully sorted three suffixes:  $T_3, T_5$  and  $T_{10}$ . At this point, we have 6 singleton buckets and 3 buckets that still needs to be sorted. We reset the  $Cnt[i]$  and  $B_{2h}[i]$  by setting  $Cnt[i] = 0$  for all  $i$ ,  $B_{2h}[ISA[n - h]] = 1$  and incrementing  $Cnt[ISA[n - h]]$ . Then we compute new values for  $ISA[i]$  that are derived from the new bucket numbers. We perform same operations as in previous stage but with values from the table above. As a result, at the end of stage  $h = 2$ , we have:

$i$	$B_h[i]$	$B_{2h}[i]$	$Cnt[i]$	$ISA[i]$	$SA[i]$
0	1	1	0	2	11 = \$
1	1	1	1	7	10 = a\$
2	1	1	2	11	0 = abracadabra\$
3	0	0	0	4	7 = abra\$
4	1	1	1	8	3 = acadabra\$
5	1	1	1	5	5 = adabra\$
6	1	1	2	9	8 = bra\$
7	1	1	0	3	1 = bracadabra\$
8	1	1	1	6	4 = cadabra\$
9	1	1	1	10	6 = dabra\$
10	1	1	2	1	9 = ra\$
11	1	1	0	0	2 = racadabra\$

We moved  $T_9$  to 10,  $T_8$  to 6,  $T_5$  to 5,  $T_1$  to 7,  $T_3$  to 4,  $T_6$  to 9,  $T_2$  to 11,  $T_4$  to 8,  $T_0$  to 2,  $T_7$  to 3.

At the beginning of stage  $h = 4$ , we have:

$i$	$B_h[i]$	$B_{2h}[i]$	$Cnt[i]$	$ISA[i]$	$SA[i]$
0	1	1	0	2	11 = \$
1	1	1	0	7	10 = a\$
2	1	1	0	11	0 = abracadabra\$
3	0	0	0	4	7 = abra\$
4	1	1	0	8	3 = acadabra\$
5	1	1	0	5	5 = adabra\$
6	1	1	1	9	8 = bra\$
7	1	1	0	2	1 = bracadabra\$
8	1	1	0	6	4 = cadabra\$
9	1	1	0	10	6 = dabra\$
10	1	1	0	1	9 = ra\$
11	1	1	0	0	2 = racadabra\$

From the previous stage we uniquely sorted suffixes  $T_1, T_2, T_8$ , and  $T_9$ . As a result, at the beginning of this stage, we have 10 out of 11 singleton buckets. Only two suffixes have the same rank  $T_0$  and  $T_7$  and hence needs to be sorted. After executing lines 7-39, we obtain the following results at the end of stage  $h = 4$ :

$i$	$B_h[i]$	$B_{2h}[i]$	$Cnt[i]$	$ISA[i]$	$SA[i]$
0	1	1	0	3	11 = \$
1	1	1	0	7	10 = a\$
2	1	1	2	11	0 = abracadabra\$
3	0	1	0	4	7 = abra\$
4	1	1	1	8	3 = acadabra\$
5	1	1	1	5	5 = adabra\$
6	1	1	1	9	8 = bra\$
7	1	1	1	2	1 = bracadabra\$
8	1	1	1	6	4 = cadabra\$
9	1	1	1	10	6 = dabra\$
10	1	1	0	1	9 = ra\$
11	1	1	1	0	2 = racadabra\$

We moved  $T_7$  to 2,  $T_6$  to 9,  $T_3$  to 4,  $T_1$  to 7,  $T_4$  to 8,  $T_0$  to 3,  $T_2$  to 11, and  $T_5$  to 5. The algorithm updates  $SA$  and  $Bh$ . The main while loop terminates

since all buckets are sorted. The algorithm reconstructs  $ISA[i]$  from  $SA[i]$  for all  $i$  and outputs the suffix array  $SA$ .

### 3.1.2 Larsson and Sadakane Algorithm

Bucket sorting is one of the main factors affecting the efficiency of prefix-doubling SACAs. Manber and Myers [53] (MM) were first to describe and apply the prefix-doubling technique in their SACA using linear-time bucket sorting. The working principles of the MM algorithm are based on scanning suffixes at  $h$ -order, implicitly sorting them by moving to the top of their bucket and relabeling buckets in  $2h$ -order. This whole process takes linear time. However, MM is inefficient in practice due to a large amount of redundant work [74].

According to [11, 17, 74], Larsson and Sadakane [45] (LS) algorithm is one of the most efficient implementations based on the prefix-doubling technique in the non-parallel setting. Although its time complexity is the same as the MM algorithm,  $O(n \log n)$ , in practice, it runs much faster. The LS algorithm is based on the MM algorithm with several important adjustments. First, the LS eliminates unnecessary scanning. Second, it avoids a large amount of redundant work such as idle regrouping of already sorted parts of ISA. Third, it keeps the memory space at the same level as the MM.

The LS algorithm does not scan the whole array; instead, it keeps track of sorted buckets and ignores them in further rounds. The LS explicitly sorts each  $h$ -bucket using the *Bentley-McIlroy quicksort* (BMQS) [8]. The BMQS is an optimized version of the quicksort that employs split-end partitioning and incorporates an efficient solution for elements swapping. The BMQS groups equal elements and brings them to the middle of the array, while only distinct elements are recursively sorted. This method is highly efficient for inputs with a large number of equal elements, as it takes less time to swap equal elements and bring them to the middle of the array than to make recursive calls on them.

Larsson and Sadakane formulated the problem as follows. Let  $T[0 \dots n] = T[0]T[1] \dots T[n]$  be a string comprised of  $n+1$  characters, such that the actual input string occupies the subarray  $T[0 \dots n)$  and  $T[n] = \$$  is the sentinel. In each stage  $h$ , an integer array  $I$  stores suffixes in their  $h$ -order. Values of  $I$  are integers that belong to the interval  $[0, n]$ . In other words, integer array  $I$  is a suffix array that orders suffixes lexicographically according to the first  $h$  symbols.  $I$  is in the final state when each suffix can be uniquely distinguished. Algorithm LS employs the following observation of Manber and Myers:

**Observation 2.** *When we scan suffixes in  $h$ -order, we can use the rank of suffix  $T_{i+h}$  as a sorting key for suffix  $T_i$  to place it in its  $2h$ -order.*

Observation 2 finds its application in the LS algorithm in the following way. Initially, we place all suffixes in their 1-order by sorting them lexicographically considering only the first character of each suffix. Next, when  $h \geq 1$ ,  $h = 2^i$ , the position of the suffix  $T_i + 2^{h-1}$  computed in  $\dagger$ stage  $h - 1$  is a sorting key for suffix  $T_i$ . In each iteration, we double  $h$  and consider twice as many symbols per suffix as in the previous iteration. In total, this process takes  $O(\log n)$  iterations until we turn every bucket into a singleton one.

Larsson and Sadakane introduce the following concepts in the description of their implementation. A subarray  $I[i \dots j]$  containing a maximal number of adjacent suffixes that are lexicographically equal according to the first  $h$  symbols is a *group*. The *group number* of a group is a position of the last suffix belonging to this group, i.e., for all suffixes in the group  $I[i \dots j]$  the group number is  $j$ . A group that holds only one suffix is a *sorted group*; otherwise, the group is *unsorted*. All neighboring sorted groups are merged into a *combined sorted group*.

The LS algorithm maintains three integer arrays  $I$ ,  $V$ , and  $L$ . The role of  $I$  and  $V$  is similar to the role of the  $h$ -order suffix array and its inverse. Larsson and Sadakane use  $I$  to store suffixes in their  $h$ -order. We use  $V$  to map each  $h$ -order suffix to its rank computed from the group number it belongs to. Compared to the MM approach, Larsson and Sadakane label groups by the position of the rightmost suffix in each group. We use  $L$  to keep track of the groups' lengths and facilitates the process of merging the sorted groups. If a group occupying a subarray  $I[i \dots j]$  is unsorted then we set  $L[i] = j - i + 1$ , and if it is a combined sorted group then we negate the value and set  $L[i] = -(j - i + 1)$ . Negative values that are used to mark the length of the combined sorted group enable the algorithm to jump over the sorted groups.

In the first step, we assign a position to each suffix directly from the input string  $T$ . Then suffixes are sorted by their initial positions and stored into  $I$ . The algorithm associates each suffix  $T_i$  with its position  $i$ . Each suffix is sorted by its first character, that is for every suffix  $T_i$ , the algorithm uses  $T[i]$  as a sorting key and its position  $i$  as a value. Then the algorithm initializes  $V$  by assigning the group number to each suffix, identifies sorted and unsorted groups and stores the results into  $L$ . This concludes the initialization stage and yields the 1-order. The algorithm keeps sorting  $I$  in stages doubling the value of  $h$  in each stage. Note that

---

\*Stage  $h = 0$  takes place when we sort suffixes according to the first symbol.

**Observation 3.** *When suffixes are in  $h$ -order, each suffix  $T_i$  that belongs to a sorted group has a distinct rank deduced from its  $i + h$  prefix.*

This means that once some groups are sorted, the positions of suffixes in these groups are fixed in  $I$ . The problem is reduced to rearranging the unsorted groups. We sort each unsorted group  $I[j \dots k]$  by using  $V[I[i] + h]$  as a sorting key for suffix  $i$ , for all  $i \in I[j \dots k]$ . Unique group numbers obtained from  $V[I[i] + h]$  define the partition of  $I[j \dots k]$  into new groups. This places  $I$  in  $2h$ -order. We compute group numbers for  $2h$ -order and update  $L$  accordingly. A high-level description of the LS algorithm (basic version) is given below:

1. Fill  $I$  with a sequence of numbers from 0 to  $n$  representing the starting position of each suffix in the input string  $x = x_0x_1 \dots x_n$ . Sort  $I$  using  $x_i$  as the key for  $i$ . Set  $h$  to 1.
2. For each suffix  $i$ ,  $0 \leq i \leq n$ , set  $V[i] = j$ , where  $j$  is the rightmost position of each group in  $I$  which contains suffix  $i$ .
3. For each unsorted group occupying the subarray  $I[i \dots j]$  set  $L[i] = j - i + 1$ , and if it is a combined sorted group set  $L[i] = -(j - i + 1)$ .
4. Process each unsorted group occupying the subarray  $I[k \dots l]$  with BMQS, using  $V[I[i] + h]$  as the key for each suffix  $i$  in  $I[k \dots l]$ .
5. For each pair of unique keys  $V[I[i] + h]$  and  $V[I[j] + h]$ , such that  $i \neq j$ , where suffix  $i$  and suffix  $j$  are in unsorted group  $I[k \dots l]$  mark the splitting positions  $i$  and  $j$ .
6. Double  $h$ . Combine the length of sorted groups, use splitting positions to partition  $I$  into new groups, updating  $V$  and  $L$  accordingly.
7. If all groups are sorted, i.e.  $L[0] = -n$ , then stop. Otherwise, go to 4.

We run the LS algorithm with the input string `abracadabra$`. During the initialization (steps 1-3), we sort the suffixes using  $x_i$  as the key for  $i$  and store the result into  $I$ . First, we fill  $I$  with numbers that represent the starting positions of suffixes

$$I = \begin{matrix} & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 \\ \text{a} & \text{b} & \text{r} & \text{a} & \text{c} & \text{a} & \text{d} & \text{a} & \text{b} & \text{r} & \text{a} & \text{b} & \text{r} & \text{a} & \$ \end{matrix},$$

then we sort suffixes according to the first symbol and create groups with lexicographically equal symbols

$$I = \$ \begin{matrix} & 11 & | & 0 & 3 & 5 & 7 & 10 & | & 1 & 8 & | & 4 & 6 & | & 2 & 9 \\ \text{a} & \text{a} & \text{a} & \text{a} & \text{a} & \text{a} & \text{a} & \text{a} & \text{a} & \text{b} & \text{b} & \text{c} & \text{d} & \text{r} & \text{r} & \end{matrix},$$

where each group is separated by the vertical bar and the starting position of each  $h$ -order suffix is displayed above it. Next, we compute group numbers by using the rightmost position of the suffix in each group as we scan  $I$  left to right. We create new groups, assign group numbers to each suffix in the input string and store the result into  $V$ :

$$\begin{array}{cccccccccccc} 0 & & 5 & & 7 & 8 & 9 & 11 & & & & & & \\ \$ & | & a & a & a & a & a & | & b & b & | & c & | & d & | & r & r & \\ \Rightarrow & & V = & a & b & r & a & c & a & d & a & b & r & a & \$ . \end{array}$$

In the next step, we compute the length of each group and place them in  $L$ :

$$L = \$ \begin{array}{cccccccc} -1 & 5 & & 2 & -1 & -1 & 2 & \\ | & a & a & a & a & a & | & b & b & | & c & | & d & | & r & r . \end{array}$$

Thus groups are arranged as follows: group 5 has length 5, group 7 and 11, both have length 2. Groups 0,8, and 9 have a negative length and considered sorted.

**Example 3.1.** We run the LS algorithm (basic version) with the input string  $x = \text{abracadabra}\$$  and a sentinel symbol  $\$$  appended to the end of the string. The algorithm proceeds from top to bottom. At the beginning of each doubling stage  $h$ , we list the keys  $V[I[i] + h]$  that are used to sort suffixes  $I[i]$  in each unsorted group. We demonstrate the content of  $V$ ,  $L$  and how it is updated across different stages.

$h$	$i$	0	1	2	3	4	5	6	7	8	9	10	11
	$x_i$	a	b	r	a	c	a	d	a	b	r	a	\$
	$I[i]$	<u>11</u>	0	3	5	7	10	1	8	<u>4</u>	<u>6</u>	2	9
	$V[I[i]]$	0	5	5	5	5	5	7	7	8	9	11	11
	$L[i]$	-1	5					2		-2		2	
1	$V[I[i] + h]$		7	8	9	7	0	11	11			5	5
	$I[i]$		<u>10</u>	0	7	<u>3</u>	<u>5</u>	1	8			2	9
	$V[I[i]]$		1	3	3	4	5	7	7			11	11
	$L[i]$	-2		2		-2		2		-2		2	
2	$V[I[i] + h]$			11	11			4	1			8	0
	$I[i]$			0	7			<u>8</u>	<u>1</u>			<u>9</u>	<u>2</u>
	$V[I[i]]$			3	3			6	7			10	11
	$L[i]$	-2		2		-8							
4	$V[I[i] + h]$			8	0								
	$I[i]$			<u>7</u>	<u>0</u>								
	$V[I[i]]$			2	3								
	$L[i]$	-12											
	$I[i]$	11	10	7	0	3	5	8	1	4	6	9	2

In Example 3.1, once the group is sorted, we underline the suffix it hosts to show that its position is fixed in  $I$  and to help the reader to visualize and track down the final result. To compare  $h$ -order suffixes lexicographically we use the following symbol  $\vee$ .

At the beginning of stage  $h = 1$ , we have three unsorted groups 5, 7, and 11. To sort suffixes of the group 5, we need to compare them according to the first two symbols:  $\mathbf{ab} \vee \mathbf{ac} \vee \mathbf{ad} \vee \mathbf{ab} \vee \mathbf{a\$}$ . Since the suffixes are already sorted according to their first symbols, the task is reduced to comparing  $\mathbf{b} \vee \mathbf{c} \vee \mathbf{d} \vee \mathbf{b} \vee \mathbf{\$}$ . We call BMQS and supply the following  $V[I[i] + 1]$  keys 7, 8, 9, 7, 0 and corresponding 0, 3, 5, 7, 10 values. The result of the sorting is 0, 7, 7, 8, 9 which places suffixes in the following order 10, 0, 7, 3, 5. We mark the splitting positions defined by unique keys 0, 8, 9 and process the next unsorted group. Processing the unsorted groups 7 and 11 we notice that suffixes in these groups yield equal  $V[I[i]+1]$  keys and cannot be distinguished lexicographically at this stage. The algorithm proceeds to the next stage updating  $V$  and  $L$  accordingly.

At the beginning of stage  $h = 2$ , we have:

$$I = \mathbf{\$} \overset{11}{\underline{a}} \overset{10}{\underline{a}} \overset{0}{\underline{a}} \overset{7}{\underline{a}} \overset{3}{\underline{a}} \overset{5}{\underline{a}} \overset{1}{\underline{b}} \overset{8}{\underline{b}} \overset{4}{\underline{c}} \overset{6}{\underline{d}} \overset{2}{\underline{r}} \overset{9}{\underline{r}},$$

and the groups are arranged as follows

$$\overset{0}{\underline{\$}} \overset{1}{\underline{a}} \overset{3}{\underline{a}} \overset{4}{\underline{a}} \overset{5}{\underline{a}} \overset{7}{\underline{a}} \overset{8}{\underline{b}} \overset{9}{\underline{b}} \overset{11}{\underline{c}} \overset{11}{\underline{d}} \overset{11}{\underline{r}} \overset{11}{\underline{r}} \Rightarrow V = \overset{3}{\underline{a}} \overset{7}{\underline{b}} \overset{11}{\underline{r}} \overset{4}{\underline{a}} \overset{8}{\underline{c}} \overset{5}{\underline{a}} \overset{9}{\underline{d}} \overset{3}{\underline{b}} \overset{7}{\underline{r}} \overset{11}{\underline{a}} \overset{1}{\underline{b}} \overset{0}{\underline{\$}}.$$

All groups, except 3, 7, and 11 are sorted. The keys  $V[I[i] + 2]$  for suffixes in group 3 are  $V[0 + 2] = 11$  and  $V[7 + 2] = 11$ . Since the keys are not unique, the algorithm proceeds to the next unsorted group. The keys for suffixes in group 7 are  $V[1 + 2] = 4$  and  $V[8 + 2] = 1$ . The keys for suffixes in group 11 are  $V[2 + 2] = 8$  and  $V[9 + 2] = 0$ . In this pass, groups 7 and 11 are fully sorted, as the keys for suffixes in these groups are unique. Splitting positions are 1, 2, 8, and 9. They are used to update  $V$  and  $L$  accordingly. At the beginning of stage  $h = 4$ , we have:

$$I = \mathbf{\$} \overset{11}{\underline{a}} \overset{10}{\underline{a}} \overset{0}{\underline{a}} \overset{7}{\underline{a}} \overset{3}{\underline{a}} \overset{5}{\underline{a}} \overset{8}{\underline{b}} \overset{1}{\underline{b}} \overset{4}{\underline{c}} \overset{6}{\underline{d}} \overset{9}{\underline{d}} \overset{2}{\underline{r}},$$

and the groups are arranged as follows

$$\overset{0}{\underline{\$}} \overset{1}{\underline{a}} \overset{3}{\underline{a}} \overset{4}{\underline{a}} \overset{5}{\underline{a}} \overset{6}{\underline{a}} \overset{7}{\underline{a}} \overset{8}{\underline{b}} \overset{9}{\underline{b}} \overset{10}{\underline{c}} \overset{11}{\underline{d}} \overset{11}{\underline{d}} \overset{11}{\underline{r}} \overset{11}{\underline{r}} \Rightarrow V = \overset{3}{\underline{a}} \overset{7}{\underline{b}} \overset{11}{\underline{r}} \overset{4}{\underline{a}} \overset{8}{\underline{c}} \overset{5}{\underline{a}} \overset{9}{\underline{d}} \overset{3}{\underline{b}} \overset{6}{\underline{d}} \overset{10}{\underline{a}} \overset{1}{\underline{b}} \overset{0}{\underline{\$}}.$$

Only group 3 is left unsorted. The algorithm proceeds to group 3 and computes  $V[I[i] + 4]$  keys. The key for suffix 0 is  $V[0 + 4] = 8$  and the key for suffix 7 is  $V[7 + 4] = 0$ . Suffix 7 comes before suffix 0 according to the keys. The algorithm marks splitting positions 0 and 7, updates  $V$  and  $L$ . All groups are now sorted, the algorithm terminates and outputs  $I$ .

## 3.2 Recursive Algorithms

The *skew* algorithm (also referred as DC3) [36] is a linear-time recursive SACA for integer alphabets developed by Kärkkäinen and Sanders [37]. Recursive SACAs have three steps: (1) Construct a subset of suffixes of size  $2/3$  or less. Recursively sort this subset. (2) Construct a subset of the remaining suffixes and sort it using the result of (1). (3) Merge both subsets into one. To illustrate this approach, we describe the method of Kärkkäinen and Sanders.

Let  $T[0, n) = t_0 t_1 \dots t_{n-1}$  be the input string of  $n$  characters over the alphabet  $\Sigma = \{1, 2, \dots, \sigma\}$  and let  $t_j = \$$  for  $j \geq n$  be a special character (called sentinel) smaller than any other alphabet character. The *skew* algorithm takes the following steps:

1. Build an array of suffixes starting at positions  $S_{12} = \{i : i \bmod 3 \neq 0\}$ . Define

$$S_1 = \{i : i \bmod 3 = 1\} \text{ and } S_2 = \{i : i \bmod 3 = 2\}.$$

2. Sort  $S_{12}$  uniquely:

- (a) Construct strings

$$\begin{aligned} R_1 &= [t_1 t_2 t_3][t_4 t_5 t_6] \dots [t_{\max S_1} t_{\max S_1+1} t_{\max S_1+2}], \\ R_2 &= [t_2 t_3 t_4][t_5 t_6 t_7] \dots [t_{\max S_2} t_{\max S_2+1} t_{\max S_2+2}], \end{aligned}$$

where  $[t_i t_{i+1} t_{i+2}]$  is a triple of first three characters from  $S_1$  and  $S_2$  suffixes. Let  $R = R_1 R_2$  be the concatenation of  $R_1$  and  $R_2$ .

- (b) Encode each triple of  $R$  with a corresponding value of  $R_1 R_2$ .
  - (c) Obtain a lexicographic naming for each triple of  $R$  by running radix sorting on it and mapping a rank to its position in  $R$ .
  - (d) If lexicographic naming is not unique, run recursively the *skew* algorithm by passing ranks of  $R$  as input.
3. Sort  $S_0$ :
    - (a) For each  $i \bmod 3 = 0$ , generate a tuple  $(T[i], \text{ISA}_{12}[i+1])$ , where  $T[i]$  is a single character at position  $i$  and  $\text{ISA}_{12}[i+1]$  denotes the rank in  $S_{12}$  of the suffix  $i+1$ .
    - (b) Apply Radixsort to the tuples  $(T[i], \text{ISA}_{12}[i+1])$ .
  4. Merge  $S_0$  and  $S_{12}$ :

(a) If  $j \in S_1$  then compare formed tuples

$$(T[j], \text{ISA}_{12}[j+1]) \vee (T[i], \text{ISA}_{12}[i+1]).$$

(b) If  $j \in S_2$  then compare formed triples

$$(T[j], T[j+1], \text{ISA}_{12}[j+2]) \vee (T[i], T[i+1], \text{ISA}_{12}[i+2]).$$

The execution of the algorithm is illustrated using the string

$$T[0, 10] = \overset{0}{\text{a}} \overset{1}{\text{b}} \overset{2}{\text{r}} \overset{3}{\text{a}} \overset{4}{\text{c}} \overset{5}{\text{a}} \overset{6}{\text{d}} \overset{7}{\text{a}} \overset{8}{\text{b}} \overset{9}{\text{r}} \overset{10}{\text{a}},$$

where the final suffix array will be  $\text{SA} = (10, 7, 0, 3, 5, 8, 1, 4, 6, 9, 2)$ .

**Step 1.** We construct  $S_1, S_2$ , and  $S_{12}$ :

$$S_1 = \{1, 4, 7, 10\}, S_2 = \{2, 5, 8\}, \text{ and } S_{12} = \{1, 4, 7, 10, 2, 5, 8\}.$$

**Step 2.** To sort  $S_{12}$  we form the following strings of triples:

$$R_1 = [\text{bra}] [\text{cad}] [\text{abr}] [\text{a$$}], \text{ and } R_2 = [\text{rac}] [\text{ada}] [\text{bra}].$$

Concatenating  $R_1$  and  $R_2$  we get:

$$R = \underset{1}{[\text{bra}]} \underset{4}{[\text{cad}]} \underset{7}{[\text{abr}]} \underset{10}{[\text{a$$}]} \underset{2}{[\text{rac}]} \underset{5}{[\text{ada}]} \underset{8}{[\text{bra}]}.$$

Radix sorting and generating lexicographic ranks:

$$\underset{1}{[\text{a$$}]} \underset{2}{[\text{abr}]} \underset{3}{[\text{ada}]} \underset{4}{[\text{bra}]} \underset{4}{[\text{bra}]} \underset{5}{[\text{cad}]} \underset{6}{[\text{rac}]}.$$

Mapping ranks to triples in  $R$ :

$$R = \underset{4}{[\text{bra}]} \underset{5}{[\text{cad}]} \underset{2}{[\text{abr}]} \underset{1}{[\text{a$$}]} \underset{6}{[\text{rac}]} \underset{3}{[\text{ada}]} \underset{4}{[\text{bra}]}.$$

Let  $T_{12} = (4, 5, 2, 1, 6, 3, 4)$  be an array of lexicographic ranks of  $R$ . Since rank 4 appears twice, ranks are not distinct, the algorithm is applied recursively returning the array  $(4, 5, 1, 0, 6, 2, 3)$ , whose positions refer to the positions in  $T_{12}$ . Mapping them to positions in  $T$  we get  $S_{12} = (10, 7, 5, 8, 1, 4, 2)$ , which is uniquely sorted.

**Step 3.** To sort  $S_0$ , we form tuples according to the protocol and then compare them. Composed tuples:

$$\begin{array}{lll} 0: \text{abracadabra} & \rightarrow & (\text{a}, \text{ISA}_{12}[1]) \rightarrow (\text{a}, 5) \\ 3: \text{acadabra} & \rightarrow & (\text{a}, \text{ISA}_{12}[4]) \rightarrow (\text{a}, 6) \\ 6: \text{dabra} & \rightarrow & (\text{d}, \text{ISA}_{12}[7]) \rightarrow (\text{d}, 2) \\ 9: \text{ra} & \rightarrow & (\text{r}, \text{ISA}_{12}[10]) \rightarrow (\text{r}, 1) \end{array}$$

Running Radixsort we get

$$T[0] < T[3] < T[6] < T[9] \text{ because } (\mathbf{a}, 5) < (\mathbf{a}, 6) < (\mathbf{d}, 2) < (\mathbf{r}, 1).$$

Thus sorted  $S_0 = (0, 3, 6, 9)$ .

**Step 4.** To illustrate the merging procedure we construct a Table 3.1 and fill it with tuples and triples as defined in the algorithm protocol.

$S_0$				$S_{12}$						
0	3	6	9	10	7	5	8	1	4	2
( $\mathbf{a}, 5$ )	( $\mathbf{a}, 6$ )	( $\mathbf{d}, 2$ )	( $\mathbf{r}, 1$ )	( $\mathbf{a}, 0$ )	( $\mathbf{a}, 4$ )			( $\mathbf{b}, 7$ )	( $\mathbf{c}, 3$ )	
( $\mathbf{a}, \mathbf{b}, 7$ )	( $\mathbf{a}, \mathbf{c}, 3$ )	( $\mathbf{d}, \mathbf{a}, 4$ )	( $\mathbf{r}, \mathbf{a}, 0$ )			( $\mathbf{a}, \mathbf{d}, 2$ )	( $\mathbf{b}, \mathbf{r}, 1$ )			( $\mathbf{r}, \mathbf{a}, 6$ )

Table 3.1: In our running example, we have that  $T[10, 12] < T[0, 12]$  since  $(\mathbf{a}, 0) < (\mathbf{a}, 5)$ . Also we have that  $T[9, 12] < T[2, 12]$  since  $(\mathbf{r}, \mathbf{a}, 0) < (\mathbf{r}, \mathbf{a}, 6)$ . In the following table we include all possible tuples and triples which are used for comparison in a merging routine.

Merging routine yields a final suffix array  $SA = (10, 7, 0, 3, 5, 8, 1, 4, 6, 9, 2)$ . Each step except the recursive call runs in time  $O(n)$ . The recursive call is executed over a string  $T_{12}$ , whose length is  $\lceil 2n/3 \rceil$ . Thus, the running time of the algorithm is defined by recurrence  $T(n) = T(2n/3) + O(n)$ . By Master Theorem [94], the solution to this recurrence is  $T(n) = O(n)$ , which is linear as expected.

### 3.3 Induced Sorting Algorithms

Induced sorting technique deduces the order of unsorted suffixes from the set of already-sorted suffixes that are classified according to some criteria. Induced sorting SACAs are efficient and fast in practice. For instance, *libdivsort* [61] is based on induced sorting, and it is considered by [21, 22, 35] to be the fastest SACA in practice that operates in main memory.

To illustrate this approach, we describe the method of Nong *et al.* [67], that incorporates a combination of *LMS*-substrings to reduce the problem and a *pure induced sorting* to facilitate the propagation of the order of suffixes.

Let  $X[1, n] = x[1]x[2] \dots x[n]$  be the input string of  $n$  characters over an indexed alphabet  $\Sigma$  and let the last character of  $x$  be the sentinel such that  $x[n] = \$$ .

**S-type and L-type suffixes.** A suffix  $X[i, n]$  is classified as S-type (smaller) suffix if  $X[i, n] < X[i + 1, n]$ , otherwise if  $X[i, n] > X[i + 1, n]$  it is classified as L-type (larger) suffix. In case, when  $X[i, n] = X[i + 1, n]$ , then  $X[i, n]$  is classified as type of  $X[i + 1, n]$ . A suffix that consists of only the sentinel, such that  $X[n, n] = \$$ , is classified as S-type.

**LMS-type suffixes and substrings.** A suffix  $X[i, n]$ , for  $1 < i \leq n$ , is classified as LMS-type (leftmost S-type) suffix if  $X[i]$  is S-type suffix and  $X[i - 1]$  is L-type suffix. A substring  $X[i, j]$  is classified as LMS-type substring if both  $X[i, n]$  and  $X[j, n]$  are LMS-type suffixes, and the type of suffix  $X[k, n]$ ,  $i < k < j$  is not LMS-type. A suffix that consists of only the sentinel, such that  $X[n, n] = \$$ , is also considered as LMS-type substring.

The order of LMS-type substrings is determined by comparing them lexicographically. In case, when the characters of both substrings are equal, we break the tie by checking the type of corresponding suffixes. S-type suffixes have higher priority than L-type suffixes.

**LMS-type prefixes.** A prefix  $X[1, i]$  is classified as LMS-type prefix if it consists of a single LMS-type suffix, or a suffix  $X[j, n]$  is classified as LMS-type suffix, where  $j$  is the first position after  $i$ . If suffix  $X[i, n]$  is classified as S-type suffix, then a LMS-prefix  $X[1, i]$  is also of S-type. Similarly, if suffix  $X[i, n]$  is classified as L-type suffix, then a LMS-prefix  $X[1, i]$  is also of L-type.

---

**Algorithm 2:** The SAIS algorithm

---

```

1 function SAIS ( $X, SA, n, \sigma$ );
   Input :  $X$  is the input string,  $SA$  is an empty suffix array,  $n$  is the
           length of  $X$ ,  $\sigma$  is the size of indexed alphabet  $\Sigma$ .
   Output:  $SA$  containing sorted lexicographically suffixes (their
           starting positions) of  $X$ .
2 Define four integer arrays:  $t(n), X_1(n), P_1(n), B(n)$ ;
3 Classify all the S-type and L-type suffixes and store them in  $t$ ;
4 Classify all the LMS-substrings and store them in  $P_1$ ;
5 Induced-sort all the LMS-substrings with the aid of  $P_1$  and  $B$ ;
6 Create  $X_1$  using the names of LMS-substrings;
7 if each character in  $X_1$  is unique then
8   |  $SA_1[X_1[i]] = i$  for all  $i$ ;
9 else
10  | Recursively call SAIS( $X_1, SA_1, n_1, \sigma_1$ );
11 end
12 Induce  $SA$  from  $SA_1$ ;
13 return

```

---

The implementation of Nong *et al.* is described in Algorithm 2. Below, we provide a running example of this algorithm using the input string  $X = \text{abracadabra}\$$ . We assume that  $X$  is a zero-indexed array of characters that ends with a sentinel  $\$$ . Initially, we scan  $X$ , classify S/L-type suffixes and store the result in type array  $t$ . We mark all the LMS-type suffixes by  $*$  and proceed to execute the algorithm in three steps.

**Step 1.** We classify suffixes 3, 5, 7, and 11 to be LMS-type. Next, we identify and label the buckets as follows. We call a bucket containing consecutive identical characters  $i$  as *bucket  $i$* . The algorithm groups suffixes with identical first character into 6 buckets associated with characters  $\$, a, b, c, d,$  and  $r$ , as illustrated in lines 5 and 6. We set all values of SA to negative one and scan  $X$  to assign all LMS-type suffixes into corresponding buckets. Since LMS-type suffixes 3, 5, and 7 begin with the identical character  $a$ , they are all put into bucket  $a$ . LMS-suffix 11 is put into bucket  $\$$ . At this point, all LMS-prefixes of length one are sorted.

**Step 2.** For illustration purpose, we mark with  $\hat{\cdot}$  the head of the bucket that is currently being scanned. Additionally, we use the symbol  $@$  to show which element of SA is currently being processed. When we are processing the first element in SA, which is  $SA[0] = 11$  (line 9), we use this value to discover that suffix 10 starts with character  $a$  and is  $L$ -type. Then we add it to bucket  $a$  and shift the head one step forward. We repeat this process until we reach the end of SA. By that time, all the  $L$ -type LMS-prefixes in SA are sorted (line 18). When  $\hat{\cdot}$  points to the space between two buckets, then one of them or both are full.

**Step 3.** Now, we use the sorted  $L$ -type prefixes to induce the order of the all LMS-prefixes with length larger than one. We mark the right boundary of each bucket with  $\hat{\cdot}$  and scan SA from right to left. When we are processing the last element in SA, which is  $SA[11] = 2$  (line 21), we use this value to discover that suffix 1 starts with character  $b$  and is  $S$ -type. Then we append it to the bucket  $b$  and shift  $\hat{\cdot}$  one step to the left. We repeat this procedure until we reach the beginning of SA. When scanning SA is completed, all the LMS-prefixes are arranged according to the order deduced from the sorted  $L$ -type prefixes (line 33). Next, to reduce the problem and apply a divide-and-conquer approach, we divide  $X$  into a smaller array  $X_1$  and fill it with the names generated for  $LMS$ -substrings as follows. We map suffixes 3, 5, 7 and 11 to 2, 3, 1, and 0 (line 35). Since each value of  $X_1$  is unique, we compute  $SA_1$  directly from  $X_1$ . The algorithm induces SA from  $SA_1$  and outputs the result.

```

00 Index: 00 01 02 03 04 05 06 07 08 09 10 11
01 X:      a b r a c a d a b r a $
02 t:      S S L S L S L S S L L S
03 LMS:           *      *      *      *
04 Step 1:
05 Bucket:  $          a          b          c          d          r
06 SA: {11} {-1 -1 07 03 05} {-1 -1} {-1} {-1} {-1 -1}
07 Step 2:
08 SA: {11} {-1 -1 07 03 05} {-1 -1} {-1} {-1} {-1 -1}
09      @^      ^      ^      ^      ^      ^
10      {11} {10 -1 07 03 05} {-1 -1} {-1} {-1} {-1 -1}
11      ^      @      ^      ^      ^      ^
12      {11} {10 -1 07 03 05} {-1 -1} {-1} {-1} {09 -1}
13      ^      ^      @      ^      ^      ^
14      {11} {10 -1 07 03 05} {-1 -1} {-1} {06} {09 -1}
15      ^      ^      @      ^      ^      ^
16      {11} {10 -1 07 03 05} {-1 -1} {-1} {06} {09 02}
17      ^      ^      @      ^      ^      ^
18      {11} {10 -1 07 03 05} {-1 -1} {04} {06} {09 02}
19      ^      ^      ^      ^      ^      ^

20 Step 3:
21 SA: {11} {10 -1 07 03 05} {-1 -1} {04} {06} {09 02}
22      ^      ^      ^      ^      ^      @^
23      {11} {10 -1 07 03 05} {-1 01} {04} {06} {09 02}
24      ^      ^      ^      ^      ^      @      ^
25      {11} {10 -1 07 03 05} {08 01} {04} {06} {09 02}
26      ^      ^      ^      ^      @^      ^
27      {11} {10 -1 07 03 05} {08 01} {04} {06} {09 02}
28      ^      ^      ^      @^      ^      ^
29      {11} {10 -1 07 03 05} {08 01} {04} {06} {09 02}
30      ^      ^      @      ^      ^      ^
31      {11} {10 -1 00 03 05} {08 01} {04} {06} {09 02}
32      ^      ^      ^      @      ^      ^
33      {11} {10 07 00 03 05} {08 01} {04} {06} {09 02}
34      ^      ^      ^      ^      ^      ^
35 X1:  2  3  1  0

```

## Chapter 4

# GPU Parallel Programming

This chapter covers key aspects and formulations of the GPU programming that were applied in practice during the implementation stage. We discuss and demonstrate the application of a broad spectrum of parallel primitives that we incorporated into our implementation. We focus on three high-performance libraries that were employed to accelerate and optimize our GPU algorithm.

We begin with a brief introduction to GPU architecture and CUDA. In Section 4.2, we discuss the essential features of the Thrust library that found its application in our solution. Section 4.3 is dedicated to parallel primitives, state-of-the-art algorithms that significantly improved the performance of our implementation.

### 4.1 Compute Unified Device Architecture

The GPU architecture is different from the universal CPU architecture in the sense that it has a predetermined specialization incorporated in it. Graphics-intensive applications imply parallel data processing and GPU was originally intended for parallel computations. GPU architecture is organized in a way to support the execution of the massive number of concurrent threads. GPU is equipped with a relatively large number of arithmetic logic units (ALU's), which are combined into groups, and implement the parallel computing model.

A modern GPU is equipped with many cores, where a single core is called a *streaming processor* and a set of cores is a *streaming multiprocessor* (SM). SMs are organized in a grid, where for each SM there is a block of cores that belong to it. The number of threads per core may be on the order of few thousands. Each core implements a single-instruction, multiple-thread

(SIMT) model that controls the execution of threads. The threads that belong to the same block can share common resources, communicate with one another, and they can be synchronized [4, 40, 65].

The GPU is oriented for computation-intensive tasks and takes advantage of a single-instruction, multiple-data (SIMD) parallelism. To exploit the productivity of the SIMD parallelism and leverage the power of massively parallel GPU architecture, NVIDIA introduced in 2006 *Compute Unified Device Architecture* (CUDA). Today, CUDA is a popular platform and model for programming NVIDIA GPUs. CUDA provides affordable many-core parallelism with a low learning curve for C and C++ developers. CUDA supports cross-platform development and offers wrappers for other popular programming languages such as Python and Java. For the remainder of this thesis, we will refer to CUDA C simply as CUDA [48].

CUDA program can be logically divided into two parts, the first part (controlling) is executed on the CPU, the second part (computational) is executed on the GPU. In CUDA terminology, the CPU is called the *host* and the GPU is called the *device* [92]. The host has access to system resources and hardware. It controls the workflow, initiates the communication with the device, delegates the work to the device, and collects the results from the device. The device can be viewed as a workhorse, that performs every computationally-intensive task that the host assigned to it, and reports back the results. CUDA allows a programmer to specify which part of the CUDA code will execute on the CPU and which part will execute on the GPU [20, 88].

CUDA employs *kernels* to run the device-wide code. A kernel is a CUDA function that is launched by the host by specifying the number of thread blocks and the number of threads associated with each block [85]. The thread hierarchy of each kernel is organized into a grid composed of thread blocks. Each thread block is scheduled to run on the available SM. In which particular order SMs execute thread blocks is unknown to the user. The SM executes one thread block at a time. To leverage the SIMD parallelism, the SM partitions each thread block into a block of 32 threads that is called a *warp*. Warps run in parallel and are controlled by the SIMT unit [14, 75].

The programmable device memory is classified into the following types: *global memory*, *shared memory*, *constant memory*, *local memory*, *texture memory*, and *registers*. Global memory resembles the CPU RAM in the sense that it is the largest and the slowest memory on the device. All threads can access it during the lifespan of the active SM that they belong to. When the host sends data to the device, it is typically copied into global memory. Constant memory is located in the constant cache module of the SM. It is an immutable type of memory that is useful for storing read-only data and

it provides fast access when all threads from a warp read from the same memory location. Texture memory is oriented for graphics applications and is tuned for the two-dimensional spatial locality. Similarly to constant memory, it is a read-only cache on the chip. Shared memory is declared within the kernel and the access to it is shared between all threads from the same thread block. Its lifespan coincides with the lifespan of the kernel in which it was declared. Shared memory is much more superior than local and global memory in terms of speed because it is on-chip memory (similar to the CPU level one cache) that exhibits high throughput and low latency. Registers and local memory are used within the kernel scope and allocated during the kernel runtime execution. When a user declares automatic variables inside the kernel, they are stored in registers, which provide the fastest memory access. Automatic variables, arrays, large data structures that are declared inside the kernel and cannot be stored in registers, are stored in local memory, which is a subdivision of global memory. Thus, local memory is as slow as global memory, due to low throughput and high latency [13, 25].

## 4.2 Essentials of the Thrust Library

The Thrust library is a collection of high-performance massively parallel algorithms oriented for CUDA programming. Parallel primitives offered by the Thrust library are based on C++ templates adapted for CUDA ecosystem. Thrust creates a high-level abstraction layer on top of CUDA that enables the development of accelerated massively parallel applications and facilitates the implementation of the code. The Abstract Programming Interface (API) of the Thrust library allows the developers to convert CUDA code into a form supported by Thrust and vice versa. The thrust programming model is based on C++ Standard Template Library (STL) which significantly reduces the learning curve of CUDA ecosystem for experienced C++ developers [28].

The high-level abstractions provided by Thrust facilitate rapid development and deployment of the application since Thrust takes over the control of repetitive, time-consuming and tedious routines like determining kernel launch parameters and launching kernels, managing device memory allocations, deallocations, and data transfers, synchronizing threads, ensuring the use of fast memory when possible, collecting the results [4, 14].

Thrust automates the process of launching kernel-specific routines trying to perform them in a near-optimal manner, applying optimization and following best practices whenever possible. To take advantage of the SIMD parallelism of the GPU, Thrust aims to achieve the highest occupancy of the kernel by maximizing the number of active threads and minimizing the num-

ber of idle streaming multiprocessors. It launches the kernel with parameters that ensure the highest occupancy [20].

By delegating to Thrust the configuration of kernel launch parameters and control over kernel execution resources we get implicit access to the automatic occupancy calculator for our kernel. Additionally, we obtain near-optimal use of kernel resources such as the number of registers and the amount of shared memory. In addition to this, Thrust facilitates the robustness of the application by checking that grid dimensions do not exceed the allowed limit and by adjusting the size of large user-defined types to satisfy the requirements of the compute capabilities of the device [40, 57].

### 4.2.1 Host and Device Vectors

The data structures that are used by Thrust to store data are called *host* and *device vectors*. Thrust vectors are similar to C++ STL vectors in the sense that they can be resized dynamically, they are generic, and they automatically control memory allocation and deallocation. Host vectors live in the CPU memory while device vectors live in the GPU memory.

**Listing 4.1.** Basic application of Thrust vectors.

```
1  int main(void) {
2
3      host_vector<int> h_v(1 << 14);
4      generate(h_v.begin(), h_v.end(), rand);
5      device_vector<int> d_v = h_v;
6      sort(d_v.begin(), d_v.end());
7      copy(d_v.begin(), d_v.end(), h_v.begin());
8
9      return 0;
10 }
```

Basic application of Thrust vectors can be found in Listing 4.1. In line 3, we create the host vector of size  $2^{14}$  integers. We then fill it with random numbers using Thrust's `generate` function. In line 5, we can see how to create a device vector, allocate the device memory for it and copy the content of the host vector to the device vector. All it takes just one line. We invoke Thrust's `sort` function that sorts in parallel all values of device vector on the GPU. Finally, we copy the sorted values back into the host vector [13, 77].

**Listing 4.2.** Printing elements of the device vector.

```
1 void print(const device_vector<int>& d_vec) {
2     for (int value : d_vec)
3         cout << " " << value;
4     cout << "\n";
5 }
```

Vectors can be read or modified using the array subscript notation or using a range-based `for` loop as shown in Listing 4.2. This comes handy when we need to debug our application and observe how vector data was modified after some operation. However, each time we access data from the device vector, Thrust performs in the background a data transfer from the device to the host. Therefore, this operation is resource-demanding and should be used only for debugging purpose [83].

**Listing 4.3.** Different ways to create and initialize device vectors.

```
1 int main(void) {
2
3     device_vector<int> v1(1000);
4     device_ptr<int> v_ptr = v1.data();
5     device_vector<int> v2(v1);
6     device_vector<int> v3(v_ptr, v_ptr + 1000);
7     device_vector<int> v4(v1.begin(), v1.end());
8
9     return 0;
10 }
```

There are different ways to initialize and populate Thrust vectors [25, 75]. Listing 4.3 demonstrates different ways to declare and initialize device vectors. In line 3, we create a vector `v1` of 1000 integers. We then obtain a reference to the vector `v1` and store it into device pointer `v_ptr`. Vector `v2` is created and initialized from copying `v1` to `v2`. Vector `v3` is initialized from iterator range in "pointer style". The last vector `v4` is initialized from iterator range in "STL style". The flexibility of Thrust vectors allows to initialize them even from C++ STL vectors in a similar way as was demonstrated in Listing 4.3.

## 4.2.2 Interoperability

Thrust's interoperability is an important feature that allows the programmer to switch easily between Thrust environment and CUDA environment

in both directions. This property provides an opportunity to combine the compactness and robustness of the Thrust code with the raw efficiency of CUDA. For example, the programmer can use Thrust vectors and the device memory occupied by them in CUDA kernels with some minor code modification. In addition to this, the interoperability between Thrust and CUDA makes it possible to connect some high-efficient CUDA libraries like CUB and ModernGPU to harness the power of their parallel primitives and to improve the overall performance of the program [86, 88].

The transition between Thrust and CUDA is straightforward, and comparable with interfacing C++ STL to standard C code. We can get a CUDA pointer to the GPU memory occupied by the device vector via the raw pointer cast. The raw pointer of the device vector is obtained by calling the built-in Thrust function.

**Listing 4.4.** Translating code from Thrust to CUDA.

```

1  size_t N = 2048;
2  // create device vector of size N
3  device_vector<int> d_v(N);
4  // extract raw pointer to device memory occupied by d_v
5  int *d_ptr_raw = raw_pointer_cast(d_v.data());
6  // use d_ptr_raw in built-in CUDA functions
7  cudaMemset(d_ptr_raw, 0, N, sizeof(int));
8  // launch CUDA kernel passing a raw pointer
9  cuda_kernel<<<N/256, 256>>>(N, d_ptr_raw);
10 // memory is automatically freed

```

In Listing 4.4, the code snippet shows how to convert code from Thrust to CUDA. Thrust provides a function `raw_pointer_cast` to obtain the raw pointer `d_ptr_raw` from the device vector `d_v`. We can then use this pointer as we would if we obtained it from `cudaMalloc` call. For example, we can pass it as a parameter to CUDA kernels or CUDA built-in functions such as `cudaMemset` [1, 40].

The reverse process of translating code from CUDA to Thrust is also straightforward. Listing 4.5 illustrates this approach. To wrap a raw pointer by a device pointer we invoke the function `device_pointer_cast`. Once the cast is completed, the device pointer `d_ptr` stores the memory address of the raw pointer, which used a `cudaMalloc` call to reserve a piece of memory on the GPU. From the device pointer, Thrust retrieves all information about the device memory that it points to. This information includes the starting address of the memory, the type of data it can store, and the size of allocated memory. Now, the device pointer can be used in Thrust functions and its memory can be accessed from the host [4, 20].

**Listing 4.5.** Translating code from CUDA to Thrust.

```
1   size_t N = 2048;
2   // declare raw pointer
3   int *d_ptr_raw;
4   // allocate memory for d_ptr_raw on the GPU
5   cudaMalloc(&d_ptr_raw, N, sizeof(int));
6   // wrap raw pointer with device pointer
7   device_ptr<int> d_ptr = device_pointer_cast(d_ptr_raw);
8   // use device pointer in Thrust function
9   sort(d_ptr, d_ptr + N);
10  // write to device memory from the host via d_ptr
11  d_ptr[0] = 5;
12  // free device memory occupied by d_ptr_raw
13  cudaFree(d_ptr_raw);
```

### 4.2.3 Anonymous Kernels

*Lambda expressions* or simply *lambdas* allow to implicitly define functions that are not bound to an identifier. This type of functions are called *anonymous functions*. The basic signature of C++ lambda includes the *capture clause*, the parameter list, and the lambda body. The capture clause specifies the type and method of the capture. The parameter list and the body resemble the definition of the explicit function. Lambdas can be executed asynchronously.

GPU lambdas made their debut in CUDA 7.5 toolkit. Their syntax is similar to the syntax of C++ lambdas. In the context of CUDA, GPU lambdas define kernels implicitly in a more compact way. They automate the process of kernel launch and termination. GPU lambdas can be executed both on the host and on the device. To activate this feature, the programmer should specify the execution type by placing `__host__` `__device__` specifier between the capture clause and the parameter list specifiers. CUDA application that contains GPU lambdas must be compiled with `-expt-extended-lambda` flag.

The code snippet in Listing 4.6 demonstrates how to create and execute a heterogeneous lambda. In this example, lambda is annotated with `__host__` `__device__` specifier to facilitate heterogeneous programming. Based on the conditional statement, we can launch it either on the GPU or the CPU. In both cases the `for_each` construct will run in asynchronous setting.

GPU lambdas are built around "parallel-for" construct that enables asynchronous code execution inside the lambda block, which acts as an implicitly defined kernel body and can be considered as the *anonymous kernel*. The

anonymous kernel is launched using `for_each` construct that is defined in Thrust and specifying the iterator range, the method of capture and explicit arguments.

**Listing 4.6.** Application of the heterogeneous lambda.

```
1 void sum_lambda(float *u, float *v, float c, int n) {
2
3     using namespace thrust;
4     auto t = counting_iterator(0);
5     int limit = 10000000;
6
7     auto lmd = [=] __host__ __device__ (int j) {
8         v[j] = c * u[j] + v[j];
9     };
10
11    if(n > limit)
12        for_each(device, t, t + n, lmd);
13    else
14        for_each(host, t, t + n, lmd);
15 }
```

**Listing 4.7.** Running anonymous kernel with device vectors.

```
1 void vector_sum(device_vector<int>& a,
2 device_vector<int>& b, device_vector<int>& c) {
3
4     int *d_a = raw_pointer_cast(a.data());
5     int *d_b = raw_pointer_cast(b.data());
6     int *d_c = raw_pointer_cast(c.data());
7
8     int n = a.size();
9     auto t = counting_iterator<int>(0);
10
11    for_each(t, t + n, [=] __device__(int j) {
12        d_c[j] = d_a[j] + d_b[j];
13    });
14 }
```

Listing 4.7 demonstrates how to add two vectors by running an anonymous kernel with pointers to device vectors. Function `vector_sum` receives three references to devices vectors, which are then converted to raw pointers. Next, we query the size of vectors and set the range for the iterator. We then call

`for_each` construct that works as a `for` loop running from 0 until  $n$  in parallel where *thread index* is passed to variable  $j$ . The rest of the body performs the same routine as if it was executed inside the CUDA kernel, with an exception that in lambda case Thrust controls thread pool and ensures that the code inside `for_each` body stops being executed when iterator reaches its upper limit.

## 4.3 Parallel Primitives

In this section, we introduce the reader to the parallel primitives that we used in our GPU implementation. By parallel primitives, we mean a highly tuned parallel constructs optimized for the GPU programming model that provide state-of-the-art algorithms to enhance the performance of the application. We will discuss the essential parallel primitives that we borrowed from three CUDA libraries: Thrust, CUDA Unbound (CUB), and Modern GPU (MGPU).

CUB provides high-level abstractions for complex parallel operations which can be expressed in a few lines of code and executed sequentially despite the complexity of underlying parallelism. Performance-wise, CUB's parallel algorithms are more efficient than Thrust's analogs. CUB supports generic programming and multivariate compile-time parametrization. It targets high granularity to achieve better performance and applies adaptive tuning scheme for choosing the best parameters to accommodate kernel specialization and resource usage [58].

MGPU is a high-performance library for accelerated CUDA programming. It is a collection of header files written in C++ and adapted for CUDA ecosystem. MGPU provides some unique parallel primitives such as *segmented sort* that is highly tuned, highly efficient, user-friendly, and performance-wise the best solution for its task that is available today [6].

### 4.3.1 Radix Sort

Although Thrust's CUDA backend has some significant performance improvements over previous releases and assuming the fact that is built on top of CUB, the performance of CUB's radix sort showed some slight advantages over Thrust's radix sort as shown in Figure 4.1.

In this section, we cover CUB's radix sort for sorting 4-byte key-value pairs. In CUB's context, this operation can be accomplished by running `cub::DeviceRadixSort::SortPairs` algorithm, which sorts key-value pairs into increasing order. The input key-value pairs remain unaltered as the

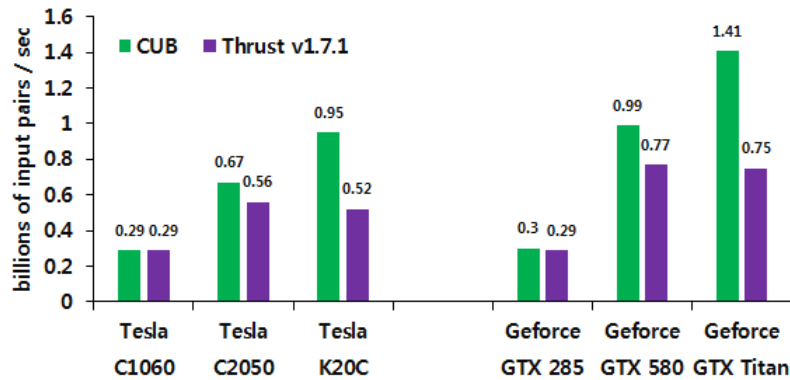


Figure 4.1: Performance comparison of device-wide radix sort using 32 millions of unsigned integer key-value pairs. [58]

algorithm uses temporary buffers of size  $2n$ , where  $n$  is the size of the key array, to store the intermediate and final results [58].

**Listing 4.8.** Key-value radix sort demonstration.

```

1 void radix_sort(int *d_in_keys, int *d_out_keys,
2 int *d_in_vals, int *d_out_vals, int n) {
3
4     void *d_tmp = NULL;
5     size_t tmp_size = 0;
6
7     DeviceRadixSort::SortPairs(d_tmp,
8 tmp_size, d_in_keys, d_out_keys,
9 d_in_vals, d_out_vals, n);
10
11    cudaMalloc(&d_tmp, tmp_size);
12
13    DeviceRadixSort::SortPairs(d_tmp,
14 tmp_size, d_in_keys, d_out_keys,
15 d_in_vals, d_out_vals, n);
16
17    cudaFree(d_tmp);
18 }

```

**Example 4.9.** Running the algorithm with the following values:

	Input Data		Output Data
d_in_keys	5, 3, 2, 8, 6, 7, 4, 1	d_out_keys	1, 2, 3, 4, 5, 6, 7, 8
d_in_vals	1, 2, 3, 4, 5, 6, 7, 8	d_out_vals	8, 3, 2, 7, 1, 5, 6, 4

The code snippet of CUB’s key-value radix sort can be found in Listing 4.8. The function `radix_sort` accepts five parameters: `d_in_keys` — a pointer of type `int` to the keys stored in the device memory, `d_out_keys` — a pointer of type `int` to the device memory location where sorted keys will reside, `d_in_vals` — a pointer of type `int` to values stored in the device memory, `d_out_vals` — a pointer of type `int` to the device memory location where sorted values will reside and the number of key-value pairs to sort.

Initially, we define a temporary buffer, and its size initialized to null. Then we run `SortPairs` routine from `DeviceRadixSort` class which will determine the required number of bytes for temporary buffer. In line 11, we call `cudaMalloc` to allocate a required number of bytes for a temporary buffer in the device memory. Finally, we run `SortPairs` again which will use temporary buffer to store intermediate results and then write final results to the output data containers. The memory allocated for the temporary buffer is later released.

### 4.3.2 Select-Flagged

This parallel primitive is available in `DeviceSelect` module of CUB, which provides filters and compacts selected items according by specified criterion applied on data that reside in the device memory. Figure 4.2 illustrates `DeviceSelect::Flagged` performance on different NVIDIA GPUs using 32-bit integers as input. [58].

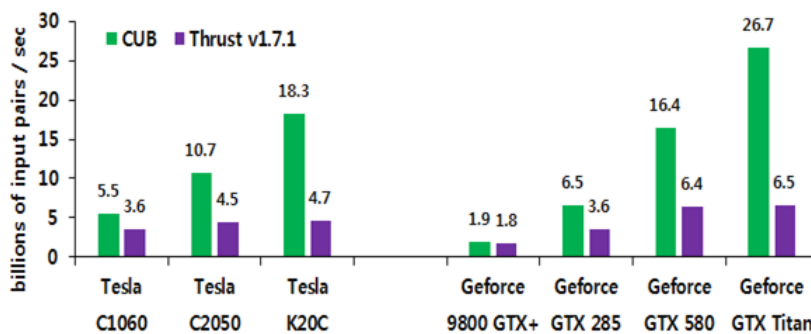


Figure 4.2: Performance comparison of device-wide select-flagged using 32 millions of integers, where 16 millions of them were randomly selected. [58]

The select-flagged operation is straightforward and comes handy when an array of items should be filtered according to the array of flags. The general idea is schematized in Figure 4.3.

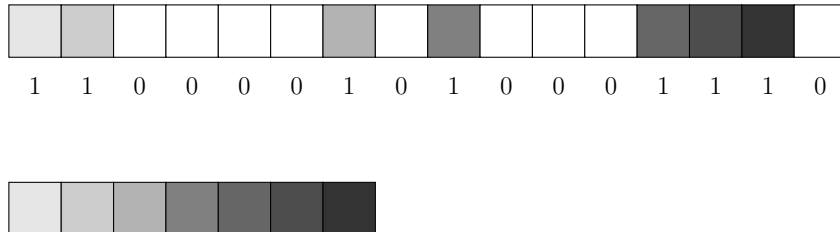


Figure 4.3: High-level scheme of select-flagged operation. [58]

The algorithm scans the flag array. Whenever it encounters a set flag, it checks its position and copies the item from input array located at that position into the compacted output array maintaining the original relative ordering.

**Listing 4.10.** Select-flagged demonstration.

```

1 void select_flagged(int *d_in, uint8_t *d_bits, int *d_out,
2                   int *d_n_bits, int n) {
3
4     void *d_tmp = NULL;
5     size_t tmp_size = 0;
6
7     DeviceSelect::Flagged(d_tmp, tmp_size,
8                          d_in, d_bits, d_out, d_n_bits, n);
9
10    cudaMalloc(&d_tmp, tmp_size);
11
12    DeviceSelect::Flagged(d_tmp, tmp_size,
13                        d_in, d_bits, d_out, d_n_bits, n);
14
15    cudaFree(d_tmp);
16 }

```

**Example 4.11.** Running the algorithm with the following values:

Input Data		Output Data	
d_in	3, 6, 9, 8, 7, 0, 5, 1	d_out	3, 6, 8, 1, 0, 0, 0, 0
d_bits	1, 1, 0, 1, 0, 0, 0, 1	d_n_bits	4

The code snippet of select-flagged parallel primitive is available in Listing 4.10. The function `select_flagged` accepts five parameters: `d_in` — a pointer of type `int` to the input data stored in the device memory, `d_bits` — a pointer of type `unsigned char` to the array of bits (flags), `d_out` — a pointer of type `int` to device container that will store values that satisfy the selection criterion, `d_n_bits` — a pointer of type `int` to the device memory location that will store number of selected values and the size of input data.

Similarly to the CUB’s radix sort algorithm, we define an empty temporary buffer and do one pass of `Flagged` algorithm with the intent to determine the size of the temporary buffer. Once we know the size, we allocate the device memory for it. In the second pass, it is used to store intermediate results and serves as a helper data structure in the algorithm’s behavior. In the end, we destroy the temporary buffer and free memory.

### 4.3.3 Inclusive Prefix Sum

There are several GPU libraries that provide a parallel solution to compute *inclusive prefix sum*. However, the algorithm from CUB’s library showed better performance than Thrust’s alternative operation as shown in Figure 4.4. This parallel primitive can be found in CUB’s `DeviceScan` class, which hosts parallel constructs for running a prefix scan on data stored in the GPU memory.

The idea of the *prefix scan* implies element-wise reduction according to the binary associative operator. Prefix sum is the instance of prefix scan, where the binary operation is the addition. It is a straightforward process of adding each prefix to the sum of prefixes before it. Formally, let  $U = \{u_1, u_2, \dots, u_n\}$  be the sequence of input elements. The prefix sum of  $U$  is the output sequence  $V = \{v_1, v_2, \dots, v_n\}$  defined as

$$\begin{aligned} v_1 &= u_1, \\ v_2 &= v_1 + u_1, \\ &\dots \\ v_n &= v_{n-1} + u_n. \end{aligned} \tag{4.1}$$

Prefix sum is inclusive when  $u_i$  is included in the computation of the output  $v_i$ . CUB’s device-oriented scan module is implemented with a ”decoupled look-back” approach, described in Merrill and Garland [59] report. This approach requires only a single pass to conduct a global prefix scan and performs approximately  $2n$  data access operations. Authors claim that it is as fast as `memcpy` operation. A high-level scheme of the algorithm design is illustrated in Figure 4.5 [58].

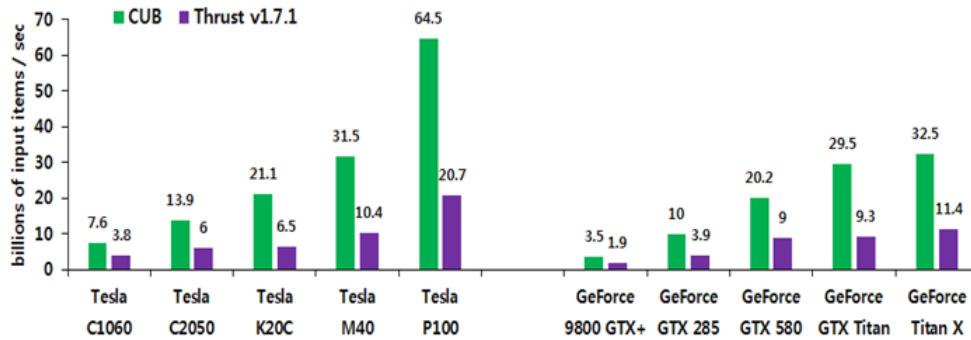


Figure 4.4: Performance comparison of the device-wide prefix scan using 32 millions of integers. [58]

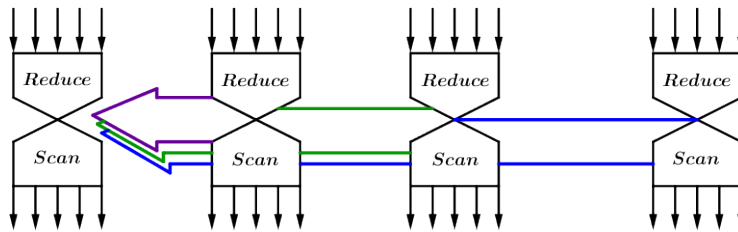


Figure 4.5: Single-pass adaptive look-back prefix scan. [59]

`DeviceScan::InclusiveSum` primitive works as follows. Given an input sequence, the algorithm generates a "moving sum." That is, value at position 2 is summed with the value at position 1. Value at position 3 is summed with the value at position 2 and so on. An application of `InclusiveSum` primitive can be found in Listing 4.12.

The function `InclusiveSum` accepts three parameters: `d.in` — a pointer of type `unsigned char` to the input data stored in the device memory, `d.out` — a pointer of type `int` to the device memory, where the output will be stored and the size of input data.

We define an empty temporary buffer and do one pass of the `inclusive_sum` algorithm with the intent to determine the size of the temporary buffer. Once we know the size, we allocate the device memory for it. In the second pass, it is used to store intermediate results and serves as a helper data structure in the algorithm's behavior. In the end, we destroy the temporary buffer and free memory.

**Listing 4.12.** Inclusive prefix sum demonstration.

```

1 void inclusive_sum(uint8_t *d_in, int *d_out, int n) {
2
3     void *d_tmp = NULL;
4     size_t tmp_size = 0;
5
6     DeviceScan::InclusiveSum(d_tmp, tmp_size,
7     d_in, d_out, n);
8
9     cudaMalloc(&d_tmp, tmp_size);
10
11    DeviceScan::InclusiveSum(d_tmp, tmp_size,
12    d_in, d_out, n);
13
14    cudaFree(d_tmp);
15 }

```

**Example 4.13.** Running the algorithm with the following values:

Input Data				Output Data			
d_in	1, 1, 0, 0, 0, 1, 0, 1	d_out	1, 2, 2, 2, 2, 3, 3, 4				

### 4.3.4 Scatter

Scatter is a parallel primitive available in the Thrust library. It copies data stored in contiguous memory into nonconsecutive memory blocks. Given input array  $A_{in}[1 \dots n]$ , index map  $M[1 \dots n]$  and output array  $A_{out}[1 \dots n]$ , scatter can be formulated as  $A_{out}[M[j]] = A_{in}[j]$  for all  $1 \leq j \leq n$ . A high-level scheme of scatter operation is illustrated in Figure 4.6 [26].

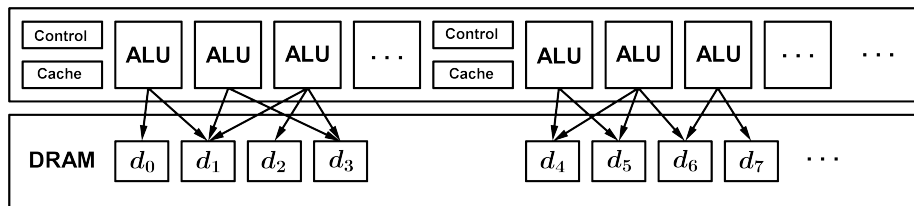


Figure 4.6: High-level scheme of scatter operation.

In context of Thrust, scatter is defined as parallel primitive that copies elements from a source range into an output array according to a map [28].

To demonstrate the work of scatter operation we run it with the following parameters:

$$M = [8, 1, 9, 0, 2, 11, 5, 3, 4, 7, 6, 10], A_{in} = [1, 2, 2, 2, 3, 3, 4, 5, 6, 7, 7, 7].$$

Invoking `Scatter( $A_{in}, M, A_{out}$ )`, the algorithm uses  $M$  as a map and outputs

$$A_{out} = [2, 2, 3, 5, 6, 4, 7, 7, 1, 2, 7, 3].$$

**Listing 4.14.** Scatter demonstration.

```

1 void Scatter(device_vector<int> &d_in,
2 device_vector<int> &d_map, device_vector<int> &d_out) {
3
4     scatter(device, d_in.begin(), d_in.end(),
5           d_map.begin(), d_out.begin());
6 }

```

### 4.3.5 Segmented Sort

The segmented sort is a high-performance variant of merge sort that operates on non-uniform random data. Segmented sort allows us to sort adjacent irregular-length segments of an array in parallel. Sorted intervals are defined by an array of starting positions of each segment that are called *segment heads*. MGPU provides one of the fastest if not the fastest segmented sort parallel primitive. Furthermore, MGPU segmented sort detects when segments are fully sorted and takes advantage of early-exit opportunities, which solves the problem of redundant resorting and improves throughput [6].

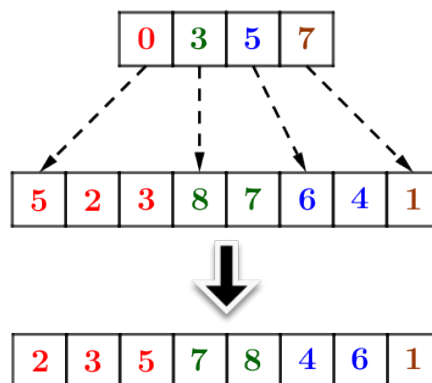


Figure 4.7: Segmented sort in action.

The main idea behind the segmented sort is illustrated in Figure 4.7. Given an array of segment heads and an array of keys to be sorted in each segment. The algorithm returns a segment-wise sorted array of keys. For better visualization, we marked each segment head with a different color. All keys of the same color belong to the same segment.

**Listing 4.15.** Demonstration of key-value segmented sort.

```

1  standard_context_t context;
2
3  vector<int> h_keys{5, 2, 3, 8, 7, 6, 4, 1};
4  vector<int> h_vals{1, 2, 3, 4, 4, 5, 6, 7};
5  vector<int> h_segs{0, 3, 5, 7, 0, 0, 0, 0};
6
7  device_vector<int> d_keys(h_keys);
8  device_vector<int> d_vals(h_vals);
9  device_vector<int> d_segs(h_segs);
10
11  int num_keys = 8;
12  int num_segs = 4;
13
14  int *keys_ptr = raw_pointer_cast(d_keys.data());
15  int *vals_ptr = raw_pointer_cast(d_vals.data());
16  int *segs_ptr = raw_pointer_cast(d_segs.data());
17
18  segmented_sort(keys_ptr, vals_ptr, num_keys,
19  segs_ptr, num_segs, less_t<int>(), context);

```

However, sometimes the task requires to sort both segment keys and satellite values. MGPU solution can deal with this problem efficiently. We demonstrate how to tackle this problem in Listing 4.15.

**Example 4.16.** Running the algorithm with the following values:

Input Data		Output Data	
keys_ptr	5, 2, 3, 8, 7, 6, 4, 1	keys_ptr	2, 3, 5, 7, 8, 4, 6, 1
vals_ptr	1, 2, 3, 4, 4, 5, 6, 7	vals_ptr	2, 3, 1, 4, 4, 6, 5, 7
segs_ptr	0, 3, 5, 7, 0, 0, 0, 0	segs_ptr	0, 3, 5, 7, 0, 0, 0, 0

We execute the code snippet in Listing 4.15 using the input values shown in Example 4.16. First, we initialize C++ STL vectors with given values. Next, we create device vectors and copy data to the device memory. The number of keys corresponds to the number of values, which is 8 in this example. We inform the algorithm that the input vector has 4 segments and their heads are

stored in `segs_ptr`. Although we have 8 items in `segs_ptr`, the algorithm will take the first 4. We then call the MGPU function `segmented_sort` passing the required parameters.

## Chapter 5

# Implementations

In this chapter, we present a massively parallel GPU implementation of suffix array construction based on the prefix-doubling method. Our implementation has good scalability and lightweight design. The essential tools that we utilized for suffix array construction are parallel primitives such as inclusive prefix sum, radix sort, select-flagged, scatter, and segmented sort. The efficiency and performance of our implementation merely depend on the performance of these primitives. High-level abstractions provided by Thrust, CUB and MGPU libraries allowed us to achieve a simple, memory-efficient and compact design of our algorithm.

In Section 5.1, we discuss the feasibility of three main algorithmic methods available for suffix array construction, in terms of how efficiently each of them can be mapped onto GPUs. We also disclose the main reasons for choosing the prefix-doubling method as well as the pros and cons of this method compared to its alternatives.

In Section 5.2, we describe and discuss the details of our implementation. In particular, we provide a high-level pseudo-code of our implementation. We follow this pseudo-code step by step explaining the role of each part of our algorithm as well as listing the source code of key functions and elaborating on their role in solving the suffix array construction problem.

We conclude this chapter by presenting a fast and lightweight algorithm for checking the correctness of the suffix array. We call this algorithm the *suffix array checker*. The theory behind its work is summarized in the theorem proposed by Burkhardt and Kärkkäinen [11]. They stated three conditions that must be satisfied to decide on the correctness of the suffix array. We elaborate on these conditions and describe our implementation approach.

## 5.1 Choosing the Algorithm

Deo and Kelly [17] noted that data dependencies in parallelizing across inducing steps and lack of an efficient solution for sorting irregularly sized strings are major bottlenecks in mapping SACAs based on induced sorting method to GPU.

Due to insufficiently researched parallelization capabilities of the induced sorting method for GPUs\*, the choice of the algorithm that could be efficiently mapped to GPUs is reduced to prefix-doubling or *DC-recursive* based methods. Although, in theory, the running time of Kärkkäinen and Sanders skew algorithm [37] is linear, in practice, it was shown [74] that it has worse performance than some algorithms running in  $O(n \log n)$  time.

According to Wang *et al.* [89] there are two significant disadvantages of mapping the *skew* algorithm to GPUs. The recursive nature of the *skew* algorithm eliminates the possibility to parallelize across recursive calls, which in turn greatly restricts the parallelism efficiency. Additionally, the *skew* algorithm performs a large amount of redundant work by resorting triplets that are already sorted as there is no simple way to indicate which triples are fully sorted and ignore them in further iterations.

From our observations, we noticed that it is rather difficult to debug the *skew* algorithm on the GPU due to its recursive structure. CUDA debugging alone can be a very cumbersome process, and the recursive formulation considerably increases debugging difficulty. Some "bug" in the recursive part of the *skew* algorithm on GPU can be challenging to spot and fix as the recursive part consists of several kernels and there are no simple ways to monitor data modification between the iterations.

The drawbacks of the prefix-doubling method with respect to parallelization are less more critical compared with the drawbacks of the recursive *skew* approach. According to Osipov [73], the MM [53] prefix-doubling algorithm does not distinguish between fully sorted buckets and those that still require sorting, which results in redundant work done on resorting singleton buckets. On the other hand, the LS [45] algorithm filters singleton buckets and applies the BMQS to sort non-singleton buckets, but makes the load balancing difficult due to irregularity in the size of non-singleton buckets.

The disadvantages of the *skew* implementation on the GPU keep us from fully exploiting the compute capabilities of our GPU. For that reason, we consider prefix-doubling approach a better candidate for our implementation. Through extensive experiments and observations, we concluded that all parts

---

\*To best of our knowledge, by the time of writing this thesis, no one has proposed an efficient implementation approaches to induce sorting SACAs for GPUs.

of the MM algorithm could be parallelized and mapped successfully onto GPU architecture. We address the issues of load balancing by using parallel primitives like *segmented sort*. In our implementation, we do not use filtering of singleton buckets because this operation requires extra work of removing all singleton buckets and respectively removing all elements from related data structures that exhibit dependency with singleton buckets. Instead, we rely on the segmented sort that is designed to deal with irregular-size bucket sorting in the most efficient and productive way.

Much of the Thrust CUDA back-end is written in terms of CUB library, which provides high-level abstractions for every level of CUDA ecosystem. We use Thrust essentials like device vector, raw pointers, device lambdas, and high-performance parallel primitives from CUB and MGPU libraries to optimize our implementation, make it robust and lightweight. Furthermore, debugging Thrust application is as simple as debugging CPU application, which plays an important role for the developer.

## 5.2 Suffix Array Construction on the GPU

Our implementation is described in Algorithm 3. The algorithm expects three input parameters. An input string  $T$  over the indexed alphabet. An empty suffix array  $SA$  that will act as a  $h$ -order buffer for intermediate values and eventually it will contain the starting positions of lexicographically sorted suffixes of  $T$ . We also supply the length of the input string as the parameter  $n$ .

The algorithm begins by copying input data to device vectors. That is,  $T$  is a device vector of length  $n$  that contains the input string and  $SA$  is an empty device vector of length  $n$ . We will use the following input to show how our algorithm works across all stages. Let  $T = \text{abracadabra\$}$  and  $n = 12$ . Next, we define five additional device vectors:

$$keys(n), b\_heads(n), b\_ranks(n), ISA(n) \text{ and } num\_segs(1).$$

We will describe the role of each vector. Note that each defined vector is of the same size  $n$ , except  $num\_segs(1)$ . Device vector  $keys$  will serve two purposes. Initially, it will store the alphabet values of each character in  $T$ , where the values are the starting position of suffixes in  $SA$  with respect to  $h$ -order. During the prefix-doubling process, the device vector  $keys$  will store the ranks obtained from  $ISA$ .

Device vector  $b\_heads$  is used to store bucket heads that are flags that mark the beginning and the end of each bucket. They are intended to distinguish between different buckets and suffixes that they contain. We define a

bucket head to be the leftmost element in each bucket as we scan the buckets from left to right.

Device vector  $b\_ranks$  is used to store bucket ranks. It assigns a unique rank to each bucket. All suffixes of the same bucket inherit the rank of the bucket they belong to.

As we know from Chapter 2,  $ISA$  is the inverse suffix array. In our implementation  $ISA$  stores the ranks of suffixes with respect to  $h$ -order. Values of  $ISA$  are computed from bucket ranks and are used for updating values of  $keys$ .

Device vector  $num\_segs$  stands for the number of segments, i.e., buckets. It stores a single integer value, but it must be allocated in the device memory.

---

**Algorithm 3:** GPU prefix-doubling SACA

---

```

1 procedure PD-SA( $T, SA, n$ )
  Input:  $T$  — input string,  $SA$  — empty suffix array,  $n$  — input
    string length.
  Output: Sorted lexicographically suffixes of  $T$  stored in  $SA$ .
2  Define device vectors:  $keys(n), b\_heads(n), b\_ranks(n),$ 
3   $ISA(n), num\_segs(1)$ 
4  Pack32( $T, keys, SA, b\_heads, n$ )
5  RadixSort32( $keys, SA, n$ )
6   $h \leftarrow 4$ 
7  while  $\top$  do
8    MarkBucketHeads( $keys, b\_heads, n$ )
9    if every bucket is singleton then
10   | Exit loop
11   end
12   UpdateBucketHeads( $b\_heads, b\_ranks, n$ )
13   ComputeISA( $SA, b\_ranks, n$ )
14   UpdateKeys( $keys, ISA, SA, b\_heads, b\_ranks, h, n$ )
15   GetSegmentHeads( $b\_ranks, ISA, b\_heads, num\_segs, n$ )
16   SegmentedSort( $keys, ISA, SA, num\_segs, n$ )
17    $h \leftarrow 2h$ 
18 end
19 return  $SA$ 
20 End

```

---

Once we initialized 5 device vectors listed in lines 2–3, we call the function Pack32. This function converts 4 consecutive 8-bit characters into a single 32-bit integer. Our task is to pack as many characters as possible into a

single integer. It is apparent that we cannot pack more than 4 characters into a single integer. This operation allows to compact the input string, apply 32-bit radix sort and launch prefix-doubling loop from  $h = 4$ .

**Listing 5.1.** Source code of Pack32 function.

```

1 void Pack32(device_vector<uint8_t>& T,
2 device_vector<int>& keys, device_vector<int>& SA,
3 device_vector<uint8_t>& b_heads, int n) {
4
5     uint8_t *T_ptr = raw_pointer_cast(T.data());
6     uint8_t *b_heads_ptr = raw_pointer_cast(b_heads.data());
7     int *keys_ptr = raw_pointer_cast(keys.data());
8     int *SA_ptr = raw_pointer_cast(SA.data());
9     auto r = counting_iterator<int>(0);
10
11     for_each(r, r + n, [=] __device__(int i) {
12         b_heads_ptr[i] = 0;
13         SA_ptr[i] = i;
14
15         int p32 = T_ptr[i];
16         p32 <<= 8;
17         if (i + 1 < n) {
18             p32 |= T_ptr[i + 1];
19         }
20         p32 <<= 8;
21         if (i + 2 < n) {
22             p32 |= T_ptr[i + 2];
23         }
24         p32 <<= 8;
25         if (i + 3 < n) {
26             p32 |= T_ptr[i + 3];
27         }
28
29         keys_ptr[i] = p32;
30     });
31 }

```

To pack 4 characters into an integer we invoke an anonymous kernel via device lambda. Thread controller spawns a particular number of threads, each of which launches the anonymous kernel with its thread id. Each thread reads 4 consecutive characters from the string and packs it into a 32-bit integer by shifting 8 bits to the left and concatenating the resulting bit sequence.

**Listing 5.2.** Source code of RadixSort32 function.

```
1 void RadixSort32(device_vector<int>& keys,
2 device_vector<int>& SA, uint32_t num_items) {
3
4     uint32_t *d_keys = raw_pointer_cast(keys.data());
5     uint32_t *d_values = raw_pointer_cast(SA.data());
6
7     size_t temp_storage_bytes = 0;
8     void *d_temp_storage = NULL;
9
10    DoubleBuffer<uint32_t> d_cub_keys;
11    DoubleBuffer<uint32_t> d_cub_values;
12    d_cub_keys.d_buffers[d_cub_keys.selector] = d_keys;
13    d_cub_values.d_buffers[d_cub_values.selector] = d_values;
14
15    cudaMalloc((void**) &d_cub_keys.d_buffers[
16    d_cub_keys.selector ^ 1], sizeof(uint32_t) * num_items);
17    cudaMalloc((void**) &d_cub_values.d_buffers[
18    d_cub_values.selector ^ 1], sizeof(uint32_t) * num_items);
19
20    DeviceRadixSort::SortPairs(d_temp_storage,
21    temp_storage_bytes, d_cub_keys, d_cub_values, num_items);
22    cudaMalloc(&d_temp_storage, temp_storage_bytes);
23
24    DeviceRadixSort::SortPairs(d_temp_storage,
25    temp_storage_bytes, d_cub_keys, d_cub_values, num_items);
26
27    cudaMemcpy(d_keys, d_cub_keys.d_buffers[d_cub_keys.selector],
28    sizeof(uint32_t) * num_items, cudaMemcpyDeviceToDevice);
29    cudaMemcpy(d_values, d_cub_values.d_buffers[
30    d_cub_values.selector], sizeof(uint32_t) * num_item,
31    cudaMemcpyDeviceToDevice);
32
33    cudaFree(d_temp_storage);
34    if (d_cub_values.d_buffers[1])
35        cudaFree(d_cub_values.d_buffers[1]);
36    if (d_cub_keys.d_buffers[1])
37        cudaFree(d_cub_keys.d_buffers[1]);
38 }
```

Additionally, we exploit the same anonymous kernel to set the values of *b\_heads* to zero and fill *SA* with its indices. The implementation of *Pack32* function is available in Listing 5.1.

After we packed all suffixes of length 4 into 32-bit integers and stored the result into *keys*, we apply 32-bit radix sort. We perform a stable radix sort by keys having *SA* as values. For that purpose, we call the implemented `RadixSort32` function.

For the sake of illustration of the algorithm work with the input string  $T = \text{abracadabra\$}$ , we will omit the packing, since the string is too short and will not allow us to demonstrate the effect of each function inside the prefix-doubling loop due to early exit, as when  $h = 4$  all buckets are fully sorted. Hence, let us assume that  $h = 1$  and during the invocation of `Pack32` we just copy values of  $T$  into *keys* and initialize *b\_heads* and *SA* omitting the packing.

During initialization stage we call two functions: `Pack32` which returns

```

T:
 97 98 114 97 99 97 100 97 98 114 97 36
keys:
 97 98 114 97 99 97 100 97 98 114 97 36
SA:
 0 1 2 3 4 5 6 7 8 9 10 11
b_heads:
 0 0 0 0 0 0 0 0 0 0 0 0
```

and `RadixSort32` which sorts *keys* and *SA* as key-value pairs and returns

```

keys:
 36 97 97 97 97 97 98 98 99 100 114 114
SA:
 11 0 3 5 7 10 1 8 4 6 2 9
```

If we consider each value of *keys* as character we get:

$$keys = \$ \overset{11}{\underset{0}{a}} \overset{3}{\underset{3}{a}} \overset{5}{\underset{7}{a}} \overset{7}{\underset{10}{a}} \overset{1}{\underset{1}{b}} \overset{8}{\underset{4}{b}} \overset{4}{\underset{6}{c}} \overset{6}{\underset{2}{d}} \overset{2}{\underset{9}{r}},$$

where each bucket is clearly separated by vertical bar.

After initialization, our algorithm launches the main prefix-doubling loop. The loop will run until all buckets are singleton, that is, fully sorted. Next, we will discuss and describe in details the role of each function inside the `while` loop. In line 8 of Algorithm 3, we call a function `MarkBucketHeads`. The role of this function is to mark the bucket heads and store the result in *b\_heads*. The implementation of this function is available in Listing 5.3.

**Listing 5.3.** Source code of MarkBucketHeads function.

```
1 void MarkBucketHeads(device_vector<int>& keys,  
2 device_vector<uint8_t>& b_heads, int n) {  
3  
4     int *keys_ptr = raw_pointer_cast(keys.data());  
5     uint8_t *b_heads_ptr = raw_pointer_cast(b_heads.data());  
6     auto r = counting_iterator<int>(0);  
7  
8     for_each(r, r + n, [=] __device__(int i) {  
9         if (b_heads_ptr[i] == 1) {  
10            return;  
11        }  
12        else if (i == 0) {  
13            b_heads_ptr[i] = 1;  
14        }  
15        else if (keys_ptr[i] != keys_ptr[i - 1]) {  
16            b_heads_ptr[i] = 1;  
17        }  
18    });  
19 }
```

We begin by casting a raw pointer to device vectors *keys* and *b\_heads*. We initialize *b\_heads* by setting its values to zero. Next, we execute device lambda in line 8. The conditional statement in line 9 tells the active thread to return if the bucket head at index *i* is already set. The conditional statement in line 12 tells the active thread to set the bucket head at index *i* if that index is the first index in the *b\_heads* vector. By default, the bucket head for the first bucket is always set. The last conditional statement in line 15 tests if the key at position *i* - 1 is different from the key at the current position *i*. If this statement is true, it implies that flag at position *i* should be set and become a new bucket head. This is a simple and logical criterion to check for a new bucket head. If two adjacent keys are the same, then they belong to the same bucket, and if they are different, they must be in different buckets as each bucket contains lexicographically equal elements (i.e. their key values must be the same).

Invoking MarkBucketHeads when  $h = 1$  returns

```
b_heads:  
1 1 0 0 0 0 1 0 1 1 1 0
```

Once the bucket heads are marked, we check if all buckets are fully sorted. For that purpose, we scan buckets. All buckets are singleton if all flags in

*b\_heads* are set. We can simply check for the smallest element in *b\_heads* by using parallel primitives like `min_element` or `reduce`. If the smallest value in *b\_heads* is 1 then we have successfully sorted all buckets, we can exit the loop and copy the results back to the host. Otherwise, we continue with the loop and call the function `UpdateBucketRanks`.

The task of the function `UpdateBucketRanks` is to update bucket ranks. The implementation of this function is available in Listing 5.4.

**Listing 5.4.** Source code of `UpdateBucketRanks` function.

```

1 void UpdateBRanks(device_vector<uint8_t>& b_heads,
2 device_vector<int>& b_ranks, int n) {
3
4     uint8_t *b_heads_ptr = raw_pointer_cast(b_heads.data());
5     int *b_ranks_ptr = raw_pointer_cast(b_ranks.data());
6
7     void *d_temp_storage = NULL;
8     size_t temp_storage_bytes = 0;
9
10    DeviceScan::InclusiveSum(d_temp_storage, temp_storage_bytes,
11    b_heads_ptr, b_ranks_ptr, n);
12
13    cudaMalloc(&d_temp_storage, temp_storage_bytes);
14
15    DeviceScan::InclusiveSum(d_temp_storage, temp_storage_bytes,
16    b_heads_ptr, b_ranks_ptr, n);
17
18    cudaFree(d_temp_storage);
19 }

```

To update bucket ranks we perform an inclusive scan on *b\_heads*. Before calling `UpdateBucketRanks` we have:

```

b_heads:
1 1 0 0 0 0 1 0 1 1 1 0

```

Invoking `UpdateBucketRanks` returns

```

b_ranks:
1 2 2 2 2 2 3 3 4 5 6 6

```

The result can be interpreted as follows. The first bucket is a singleton, and its rank is 1, the second bucket contains 5 elements, and all of them have

rank 2, the third bucket contains 2 elements, and they both are assigned rank 3, the next two buckets are singleton, and the last bucket has two elements with rank 6.

We then compute  $SA$  ranks and store them into  $ISA$  vector. Our task is to compute  $ISA[SA[i]] = b\_ranks[i]$ . This can be achieved in different ways. The simplest way is to run a kernel that will perform reading and writing to global memory. This strategy can be improved by aiming for maximum memory coalescing and using a grid-stride loop. Another way is to run a key-value radix sort with  $SA$  items serving as keys and  $b\_ranks$  items serving as values. After sorting, we perform fast device to device copy from  $b\_ranks$  to  $ISA$ . Finally, we can use parallel primitive called `scatter` to carry out the required computations.

We have tried various approaches to compute  $ISA[SA[i]] = b\_ranks[i]$ , but the efficiency of each approach is reduced to the speed of global memory access, which is a bottleneck in this type of computations. Since all approaches showed similar performance, we decided to stick to the most code-saving one, that is implemented by Thrust `scatter` primitive. Listing 5.5 demonstrates our approach.

**Listing 5.5.** Source code of `ComputeISA` function.

```

1 void ComputeISA(device_vector<int>& SA,
2 device_vector<int>& b_ranks, device_vector<int>& ISA) {
3
4     scatter(thrust::device, b_ranks.begin(), b_ranks.end(),
5           SA.begin(), ISA.begin());
6 }

```

We call `ComputeISA` with the following input data:

```

SA:
 11 0 3 5 7 10 1 8 4 6 2 9
b_ranks:
 1 2 2 2 2 2 3 3 4 5 6 6

```

which returns

```

ISA:
 2 3 6 2 4 2 5 2 3 6 2 1

```

`Scatter` copies elements from the source array  $b\_ranks$  into the output array  $ISA$  according to the map  $SA$ . All indices in the map array must be unique.

Once we know the rank of suffixes at  $h$ -order, we can compute their  $2h$ -order in linear time and store the result into *keys*. For that purpose, we invoke the function `UpdateKeys`. Its implementation is available in Listing 5.6.

**Listing 5.6.** Source code of `UpdateKeys` function.

```

1 void UpdateKeys(device_vector<int>& keys,
2 device_vector<int>& ISA, device_vector<int>& SA,
3 device_vector<uint8_t>& b_heads,
4 device_vector<int>& b_ranks, int h, int n) {
5
6     int *ISA_ptr = raw_pointer_cast(ISA.data());
7     int *SA_ptr = raw_pointer_cast(SA.data());
8     int *keys_ptr = raw_pointer_cast(keys.data());
9     int *b_ranks_ptr = raw_pointer_cast(b_ranks.data());
10    uint8_t *b_heads_ptr = raw_pointer_cast(b_heads.data());
11    auto r = counting_iterator<int>(0);
12
13    for_each(r, r + n, [=] __device__(int i) {
14        b_ranks_ptr[i] = i;
15        if (b_heads_ptr[i] == 1 && i == n - 1) {
16            return;
17        }
18        else if (b_heads_ptr[i] == 1 && b_heads_ptr[i + 1]) {
19            return;
20        }
21        int h_suffix = SA_ptr[i] + h;
22        if (h_suffix >= n) {
23            keys_ptr[i] = -h_suffix;
24        }
25        else {
26            keys_ptr[i] = ISA_ptr[h_suffix];
27        }
28    }

```

In lines 6–10 we retrieve raw pointers to the device memory allocated for each data structure used within the function scope. In line 13, we define an anonymous kernel in the form of device lambda that will perform required computations in parallel on the device side. We reuse the memory of *b\_ranks* to store the indices that we will need to compute segment heads in the next step.

Programming logic facilitates an early exit in case if a singleton bucket is detected. It involves two conditional statements. The first conditional

statement in line 16 checks if the last bucket head is set. If so, the key at position  $i$  remains unchanged and thread  $i$  returns. The second conditional statement in line 20 implies that bucket  $i$  is a singleton if and only if bucket head at position  $i$  is set and bucket head at position  $i + 1$  is also set.

If early exit was not triggered then we should compute  $h$ -order of suffix  $i$  that is  $ISA[SA[i] + h]$  and store the result into *keys*. We may have a situation when the index  $SA[i] + h$  of *ISA* is out of bound. In this case we simply negate the value of  $SA[i] + h$ , because it must be lexicographically less than the rest elements of the bucket.

Continuing with our example and data obtained from previous actions, we proceed and call `UpdateKeys` with the following input data:

```
SA:
 11 0 3 5 7 10 1 8 4 6 2 9
b_heads:
 1 1 0 0 0 0 1 0 1 1 1 0
ISA:
 2 3 6 2 4 2 5 2 3 6 2 1
```

The expected output is

```
Keys:
 36 3 4 5 3 1 6 6 99 100 2 2
b_ranks:
 0 1 2 3 4 5 6 7 8 9 10 11
```

Before we can apply bucket sorting routine, we need to determine starting positions of each bucket and number of buckets. In the next two operations we refer to each bucket as a segment. To obtain required parameters we call `GetSegmentHeads` function. The implementation is available in Listing 5.7.

The function `GetSegmentHeads` copies all indices based on the *flags* map. These indices constitute starting positions of each bucket or in other words segment heads. The values of *flags* are the same as the values of *b\_heads*, they assist in mapping process. Number of segments are stored into *num\_segs*.

**Listing 5.7.** Source code of GetSegmentHeads function.

```
1 void GetSegmentHeads(device_vector<int>& in,  
2 device_vector<int>& out, device_vector<uint8_t>& flags,  
3 device_vector<int>& num_segs, int n) {  
4  
5     int *in_ptr = raw_pointer_cast(in.data());  
6     int *out_ptr = raw_pointer_cast(out.data());  
7     uint8_t *flags_ptr = raw_pointer_cast(flags.data());  
8     int *num_segs_ptr = raw_pointer_cast(num_segs.data());  
9  
10    void *d_temp_storage = NULL;  
11    size_t temp_storage_bytes = 0;  
12  
13    DeviceSelect::Flagged(d_temp_storage, temp_storage_bytes,  
14    in_ptr, flags_ptr, out_ptr, num_segs_ptr, n);  
15  
16    cudaMalloc(&d_temp_storage, temp_storage_bytes);  
17  
18    DeviceSelect::Flagged(d_temp_storage, temp_storage_bytes,  
19    in_ptr, flags_ptr, out_ptr, num_segs_ptr, n);  
20  
21    cudaFree(d_temp_storage);  
22 }
```

We call GetSegmentHeads with the following values:

```
b_ranks:  
0 1 2 3 4 5 6 7 8 9 10 11  
b_heads:  
1 1 0 0 0 0 1 0 1 1 1 0
```

which returns

```
ISA:  
0 1 6 8 9 10 5 2 3 6 2 1  
num_segs:  
6
```

We reuse *ISA* memory to store segment heads. Number of segments indicate the range of *ISA* memory that should be read to get segment heads. To sort buckets we invoke SegmentedSort function. The implementation is demonstrated in Listing 5.8.

**Listing 5.8.** Source code of SegmentedSort function.

```

1 void SegmentedSort(device_vector<int>& keys,
2 device_vector<int>& ISA, device_vector<int>& SA,
3 device_vector<int>& num_segs , int n, context_t& context) {
4
5     int *ISA_ptr = raw_pointer_cast(ISA.data());
6     int *SA_ptr = raw_pointer_cast(SA.data());
7     int *keys_ptr = raw_pointer_cast(keys.data());
8
9     segmented_sort(keys_ptr, SA_ptr, n, ISA_ptr,
10 (int) num_segs[0], less_t<int>(), context);
11 }

```

The `segmented_sort` parallel primitive reads segment heads from *ISA* using device vector *num\_segs* for access range. Then it sorts elements within each segment bounded by segment heads.

We call `SegmentedSort` with the following inputs:

```

SA:
 11 0 3 5 7 10 1 8 4 6 2 9
Keys:
 36 3 4 5 3 1 6 6 99 100 2 2
ISA:
 0 1 6 8 9 10 5 2 3 6 2 1
num_segs:
 6

```

The function returns the following outputs:

```

SA:
 11 10 0 7 3 5 1 8 4 6 2 9
Keys:
 36 1 3 3 4 5 6 6 99 100 2 2

```

Recall that *keys* contain *h*-order ranks of suffixes except for already sorted suffixes deduced from singleton buckets. If we partition *keys* by segment heads we get

$$\{36\}, \{3, 4, 5, 3, 1\}, \{6, 6\}, \{99\}, \{100\}, \{2, 2\}$$

with the corresponding values of *SA*

$$\{11\}, \{0, 3, 5, 7, 10\}, \{1, 8\}, \{4\}, \{6\}, \{2, 9\}$$

Notice that at  $h = 1$  order we can only sort three elements from the second bucket. Elements from the third and sixth bucket are lexicographically equal and cannot be uniquely distinguished in this round. The algorithm executes few more rounds until buckets  $\{0, 7\}$ ,  $\{1, 8\}$ ,  $\{2, 9\}$  are fully sorted.

### 5.3 The Suffix Array Checker

Checking whether the implementation of an algorithm produced the correct result is not always a straightforward process. Often it is a challenging and even impossible task. An algorithm that serves the purpose of detecting errors in the output is a *result checker* [10, 90]. The result checker is programmed to verify the result according to some criteria. To check that the result of the suffix array satisfies its properties, we introduce the concept of the *suffix array checker*. It is an algorithm that uses the information about the input string to decide the correctness of its suffix array. In this section, we describe a compact, fast and lightweight suffix array checker based on the theorem proposed by Burkhardt and Kärkkäinen [11].

**Theorem 1.** *Given an input string  $T[0 \dots n - 1] = T[0]T[1] \dots T[n - 1]$  and an array of integers  $SA[0 \dots n - 1] = SA[0]SA[1] \dots SA[n - 1]$ . The array  $SA$  contains lexicographically sorted suffixes of  $T$  if and only if the following conditions are satisfied:*

1.  $\forall 0 \leq i \leq n - 1, SA[i] \in [0, n - 1]$ .
2.  $\forall 0 \leq i \leq n - 1, T[SA[i]] \geq T[SA[i - 1]]$ .
3.  $\forall 1 \leq i \leq n - 1$  such that  $SA[i - 1] \neq n - 1$  and  $T[SA[i - 1]] = T[SA[i]]$ ,  $\exists j, k \in [0, n - 1]$  such that  $SA[j] = SA[i - 1] + 1$ ,  $SA[k] = SA[i] + 1$  and  $j < k$ .

We omit the proof here. Condition 1 checks that all elements are within range of the array. Condition 2 ensures that suffixes in  $SA$  are ordered correctly based on the first character only. Condition 3 verifies the order of suffix  $i$  based on the information obtained about suffix  $i - 1$ .

The first two conditions are straightforward and can be checked in  $O(n)$  time using a single loop. Condition 3 can be checked by making use of the inverse suffix array  $ISA$ , if it is available. In that case, we can compute  $k$  and  $j$  directly from  $ISA$  as follows:

$$\begin{aligned} j &= ISA[SA[i - 1] + 1], \\ k &= ISA[SA[i] + 1]. \end{aligned} \tag{5.1}$$

We can construct  $ISA$  directly from  $SA$  in linear time:

$$ISA[SA[i]] = i, \text{ for all } 0 \leq i \leq n - 1. \quad (5.2)$$

However, this approach will require an extra  $O(n)$  memory space to store the values of  $ISA$ . We can do better if we employ the following observation [11].

**Observation 4.** *Consider suffixes that occupy the subarray  $SA[i \dots j]$  and start with an identical character. Then we have:*

$$SA[i] + 1 < SA[i + 1] + 1 < \dots < SA[j] + 1.$$

*The above holds for every suffix except the sentinel suffix as there is no entry after it.*

This observation gives rise to the following idea. We can create and maintain a lookup table  $L$  of size  $O(\sigma)$  and  $O(1)$  access time. For each unique character in the input string  $T$ , we generate a rank and update it accordingly (this step will be explained in the text). As we scan the suffix array left to right, we check  $SA[i]$  by comparing  $SA[i - 1]$  with  $SA[L[T[SA[i] - 1]]]$ . If these two values are not the same than  $SA[i]$  values is incorrect (according to Theorem 1 and Observation 4).

We accumulated all ideas presented above into a pseudo-code available in Algorithm 4. It presents a complete implementation of the suffix array checker in pseudo-code. The first two conditions of the Theorem 1 are checked in lines 2-11. Line 12 defines a lookup table  $L$ , which is an array of size  $\sigma$ . Lines 13-25 describe the initialization of the table and its use in checking the condition 3 of the Theorem 1. Verification the first two conditions of Algorithm 4 takes  $O(n)$  time and no extra memory space. To maintain the lookup table we need  $O(\sigma)$  memory space, which is a more efficient solution than keeping the inverse suffix array which will use  $O(n)$  space. However, the alphabet size in practice is much smaller than the input string length. Once we have initialized the lookup table, we run a single loop and check each entry of the suffix array only once, exploiting the property of the Observation 4 and reading from the table in  $O(1)$ . As a result, we have a lightweight and fast suffix array checker that we extensively tested on all datasets that we used in our experiments and also on small inputs.

---

**Algorithm 4:** Suffix Array Checker

---

```
1 procedure SUFCHECK( $T, SA, n$ )
  Input:  $T$  — input string,  $SA$  — suffix array,  $n$  — input string
  length.
  Output: Success if all conditions are satisfied, Failure
  otherwise.
2   for all  $i \in [0, n)$  do
3     | if  $SA[i] < 0$  or  $SA[i] \geq n$  then
4     | | return Failure
5     | end
6   end
7   for all  $i \in [1, n)$  do
8     | if  $T[SA[i-1]] > T[SA[i]]$  then
9     | | return Failure
10    | end
11  end
12  Define lookup table  $L[\sigma]$ , where  $\sigma$  is an alphabet size
13  for  $i \leftarrow n-1$  to 0 do
14    |  $L[T[SA[i]]] \leftarrow i$ 
15  end
16   $L[T[n-1]] \leftarrow L[T[n-1]] + 1$ 
17  for  $i \leftarrow 0$  to  $n-1$  do
18    | if  $SA[i] > 1$  then
19    | |  $c \leftarrow T[SA[i]-1]$ 
20    | | if  $SA[L[c]] + 1 \neq SA[i]$  then
21    | | | return Failure
22    | | end
23    | |  $L[c] \leftarrow L[c] + 1$ 
24    | end
25  end
26  return Success
27 End
```

---

## Chapter 6

# Experimental Results

In this chapter, we evaluate our GPU prefix-doubling implementation of suffix array construction *gpu-pd* and compare it with the fastest [21, 22, 34, 35], lightweight, OpenMP-parallelized suffix array constructor, *libdivsufsort* [61], which was developed by Yuta Mori for shared memory multi-core CPU architecture. *Libdivsufsort* is heavily optimized by exploiting the speedup features of OpenMP framework as well as the smart application of concurrency aimed to harness the power of multi-core processors. The implementation of *libdivsufsort* is relatively complex and advanced. It is based on induced sorting technique with some excellent tuning. A comprehensive analysis of the working principles of *libdivsufsort* was conducted by Fischer and Kurpicz [21]. Mori compared the performance of *libdivsufsort* with a plethora of most prominent CPU SACAs (some of them were parallelized and optimized for multi-core systems) over different corpora. According to the results [63], *libdivsufsort* showed the best total running time over all datasets.

### 6.1 Hardware Specifications

In this chapter, we will report on hardware and software specifications that were used in our experiments.

**Compute node specification.** Intel Xeon Processor E5-2680 v3 running at 2.5 GHz. Microarchitecture: Haswell with 22 nm semiconductor technology. Number of cores: 24 cores, 12 cores per CPU. Additional features: 30 MB L3 cache, 256 GB of DDR4-2133 RAM, hyper-threading disabled.

**GPU specification.** NVIDIA Tesla P100 GPU running at 1328.500 MHz. Architecture: NVIDIA Pascal with 16 nm semiconductor technology. Number of CUDA cores: 3584 CUDA cores, 56 streaming multiprocessors (SM) and 64 cores per SM. GPU memory: 15987 MB of free memory, with a to-

tal of 16280 MB and 64-bit pointers. Memory clock: 715.000 MHz x 4096 bits and 732.2 GB/s maximum bandwidth. Compute capability: 6.0. Error-correcting code memory (ECC memory) is enabled.

**Software specification.** Operating system: Ubuntu 16.04.5 LTS (GNU/Linux 4.4.0-109-generic x86\_64). GNU C Compiler: version gcc 5.5.0. Cuda compilation tools: release 9.2, V9.2.88.

## 6.2 Performance Evaluation

To evaluate the performance of our GPU implementation we carried out several experiments over different datasets. The goal of these experiments was to measure the running time and throughput of our GPU implementation compared with *libdivsufsort*. We used different types of corpora in each experiment. In the first experiment, we used datasets from Manzini corpus. This corpus was commonly used for benchmarking in many research papers related to the field of suffix array construction. The data it contains is considered well-rounded in terms of data type, alphabet size, string length, and correlation. In the second experiment, we used corpora generated from several different sources and containing datasets of real-world data.

### Manzini Corpus

Datasets from this corpus were partially used by Deo and Kelly [17], Osipov [73] and Wang *et al.* [89]. Deo and Kelly used only three datasets from Manzini corpus: `chr22.dna` (34 MB), `howto` (39 MB), and `jdk13c` (70 MB) including `mozilla` (50 MB) that is not currently available in Manzini corpus. Osipov used only two datasets from this corpus: `chr22.dna` (34 MB) and `howto` (39 MB). Wang *et al.* used the following datasets from Manzini corpus: `chr22.dna` (34MB), `etext99` (105MB), `howto` (39MB), `jdk13c` (70 MB), `sprot34.dat` (110 MB), `w3c2` (104 MB), and `mozilla` (50 MB).

Memory consumption analysis showed that our algorithm can successfully process input datasets as large as 640 MB and requires for a string of length  $n$  a total of  $25n$  byte storage in the GPU memory. This makes our implementation 1.3-1.5x more memory-efficient than GPU SACAs of by Deo and Kelly, Osipov and Wang *et al.*

In our experiments, we used all datasets from Manzini corpus. As you can see in Figure 6.1, the smallest speedup of 4.1x was achieved on the `w3c2` (104 MB) dataset. The largest speedup of 8.4x was achieved on the `sprot34.dat` (110 MB) dataset. The smallest throughput of 49.7 millions of characters processed per second was recorded on the `w3c2` (104 MB) input. The largest

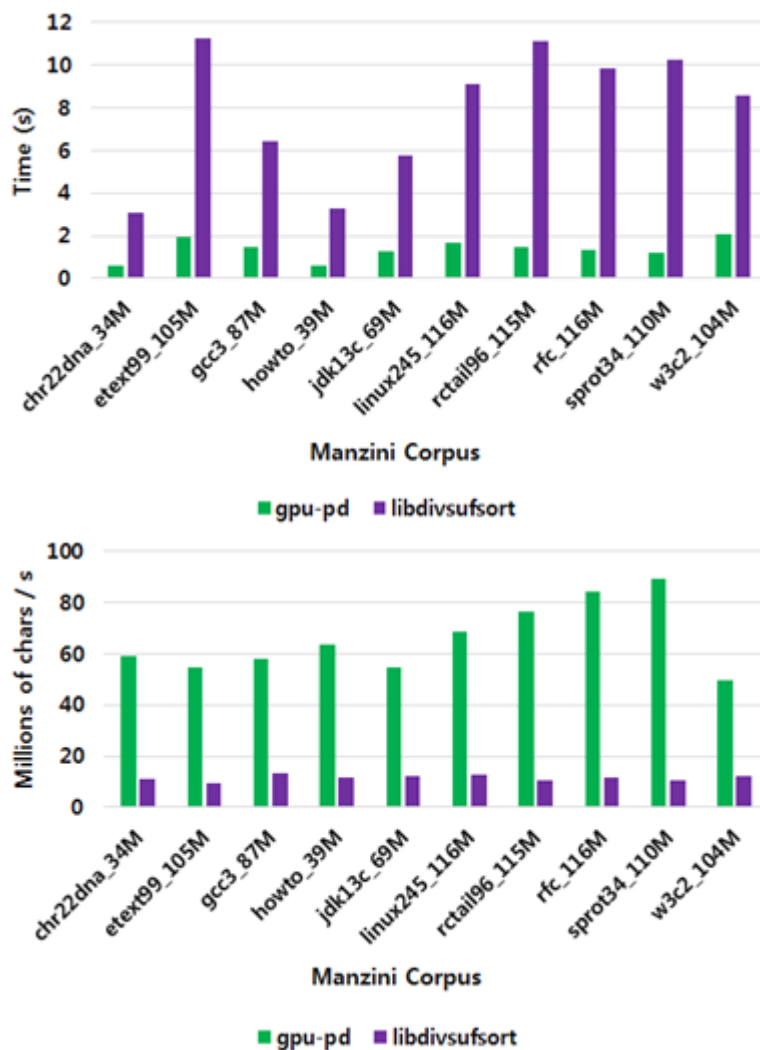


Figure 6.1: Runtimes (top figure) and throughputs (bottom figure) over Manzini corpus datasets.

throughput of 89.6 millions of characters processed per second was observed on the `sprot34.dat` (110 MB) input.

## Corpora 600

Corpora 600 comprises of different corpus datasets that are 600 MB in size with two datasets of 550 MB size. Corpora 600 is aimed to evaluate the performance of our *gpu-pd* implementation using inputs of a size that is close to the maximum input size that *gpu-pd* can process without running out of

memory. Datasets that are part of Corpora 600 have different alphabet size and contain real-world data. Bellow, we will briefly describe the content and origins of each dataset in Corpora 600:

- **amazon\_reviews** — Amazon Reviews Corpus. It contains a few million Amazon reviews from different categories of products.
- **hacker\_news** — Hacker News Corpus. It contains a subset of all Hacker News articles.
- **twitter\_support** — Corpus of Customer Support on Twitter. It contains over 3 million tweets and replies from the biggest brands on Twitter.
- **articles\_us** — Corpus of American Articles. It contains 143,000 articles from 15 American publications.
- **urban\_dict** — Urban Dictionary Words And Definitions Corpus. It contains 2.6 million words with ratings from urban dictionary.
- **enron\_emails** - Enron Emails Corpus. It contains 500,000 emails from 150 employees of the Enron Corporation.
- **nyt\_comments** - Corpus of New York Times Comments. It contains comments on articles published in the New York Times.
- **scotus** — Corpus of American Court Decisions. It contains 130 million words in 32,000 Supreme Court decisions from the 1790s to the current time.
- **ubuntu\_dialogs** — Ubuntu Dialogue Corpus. It contains 26 million turns from natural two-person dialogues.
- **blog** — Blog Authorship Corpus. It contains over 600,000 posts from more than 19 thousand bloggers.

The numerical data in the name of each dataset defines its size in MB. Experimental results are illustrated in Figure 6.2.

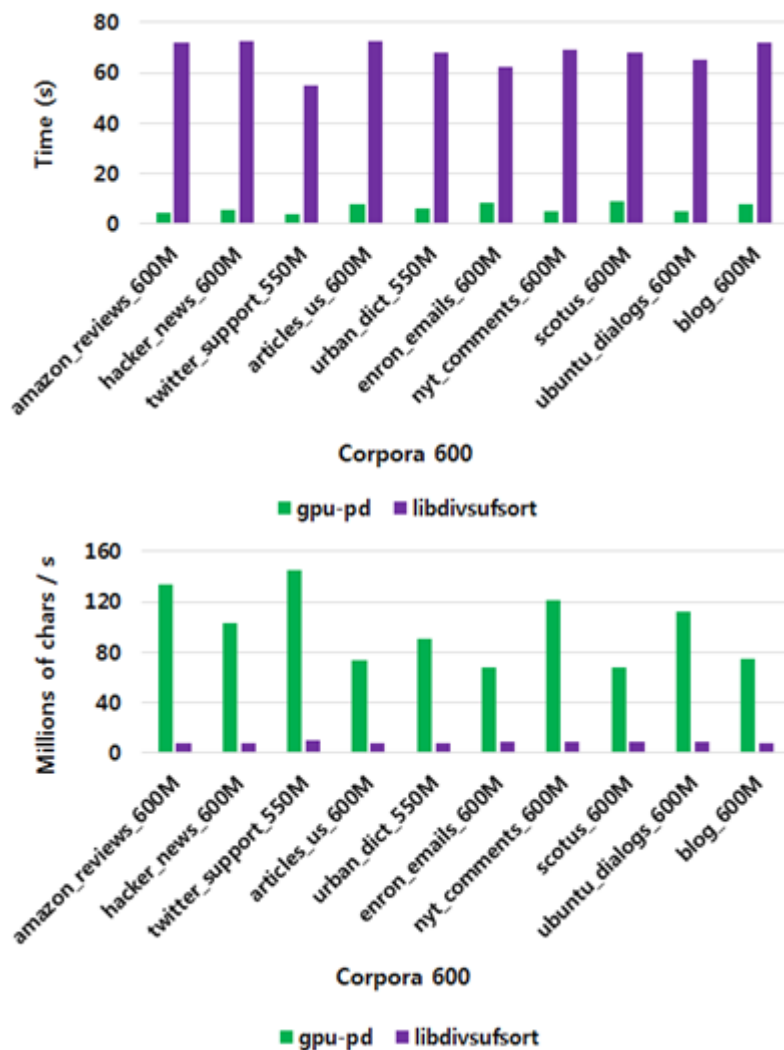


Figure 6.2: Runtimes (top figure) and throughputs (bottom figure) over Corpora 600 datasets.

With Corpora 600 datasets our implementation achieved the smallest speedup of 7.1x on the `enron_emails` (600 MB) dataset. The largest speedup of 16.1x happened on the `amazon_reviews` (600 MB) dataset. The smallest throughput of 68.4 millions of characters processed per second was recorded on the `scotus` (600 MB) dataset. The largest throughput of 144.7 millions of characters processed per second was observed on the `twitter_support` (550 MB) dataset.

## 6.3 Scalability Analysis

To investigate the scalability of our implementation we carried out two different types of experiments. In the first experiment, our task was to analyze the scalability of our algorithm as we increased the input string length. In the second experiment, we studied how our implementation scales when we supplied the worst-case inputs. For prefix-doubling it takes place when the input string consists of the same character repeated many times. We compared the results against *libdivsufsort*.

### Scalability with Dataset Size

In this experiment, our goal was to analyze how our implementation scales compared with *libdivsufsort* using datasets that were uniformly partitioned and originated from the same data source. We carried out experiments using 10 input datasets generated from the concatenation of English text files selected from *etext02* to *etext05* collections of *Gutenberg Project* and a dump of Wikipedia filtered to contain a pure English text.

The largest string length we used in this experiment is 600 MB. Small inputs cannot fully saturate the GPU and compensate for expensive data transfer between host and device. To compensate for this drawback the input string length should be sufficiently large. It is apparent, that sufficiently small inputs diminish the advantage of GPU compared to the efficiency of the parallel CPU approaches.

More objective results are achieved when the input size is sufficiently large to saturate all GPU cores and compensate the cost of data transfer between the host and device. As you can see in Figure 6.3, on both charts, the throughput of *gpu-pd* is increasing as the input size gets larger. On the other hand, the throughput of *libdivsufsort* is slowly decreasing.

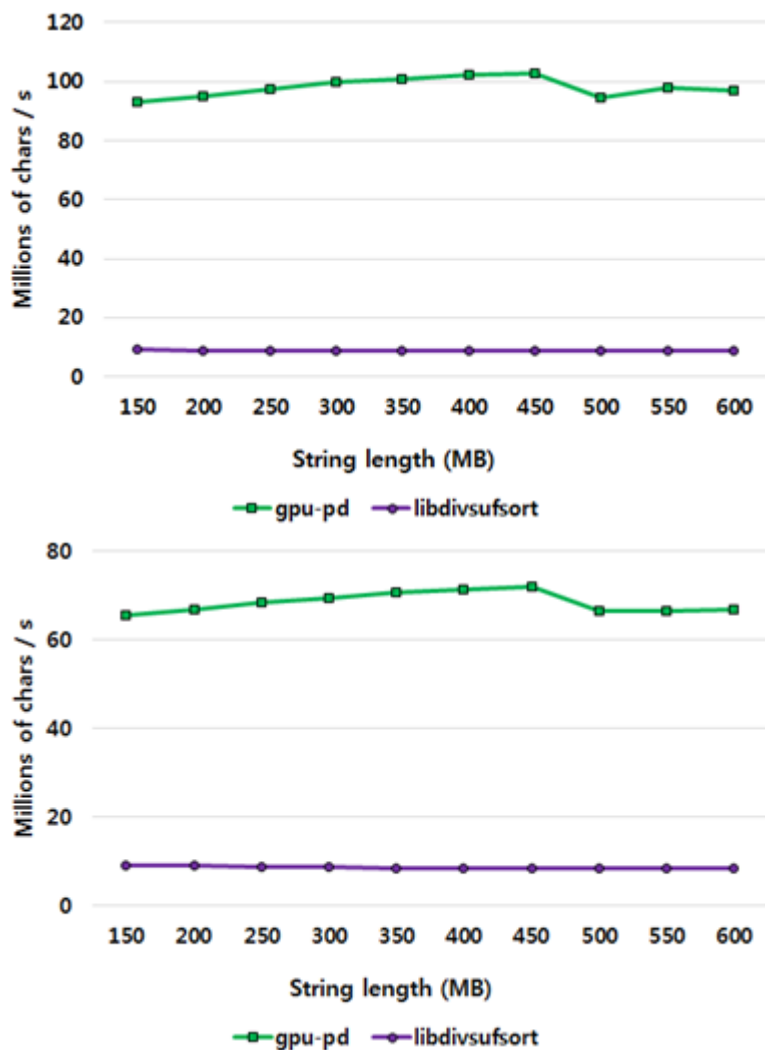


Figure 6.3: Top, bottom: suffix array construction throughput on plain text *enwik9* and on concatenation of English text files selected from *etext02* to *etext05* collections of *Gutenberg Project* as a function of dataset size.

## Scalability with Worst-Case Input

In this experiment, we analyze the scalability of our algorithm against *libdivsufsort* by simulating a worst-case scenario with respect to inputs. We generate 10 strings that range in length from 150 MB to 600 MB with 50 MB step. Each string consists of a repeated character 'A'.

For prefix-doubling SACA such sequences of the same letter repeated many times are considered hard to process because this scenario requires the algorithm to do a lot of extra work to break a single large bucket of

prefixes into singleton ones. In other words, the algorithm spends more than in average time to separate prefixes into unique buckets, because the algorithm requires extra work to distinguish between prefixes that appear to be lexicographically identical. Hence, the algorithm places initially all prefixes into a single bucket and then does a lot of extra work to segment this bucket. This scenario exhibits worst-case sorting behavior.

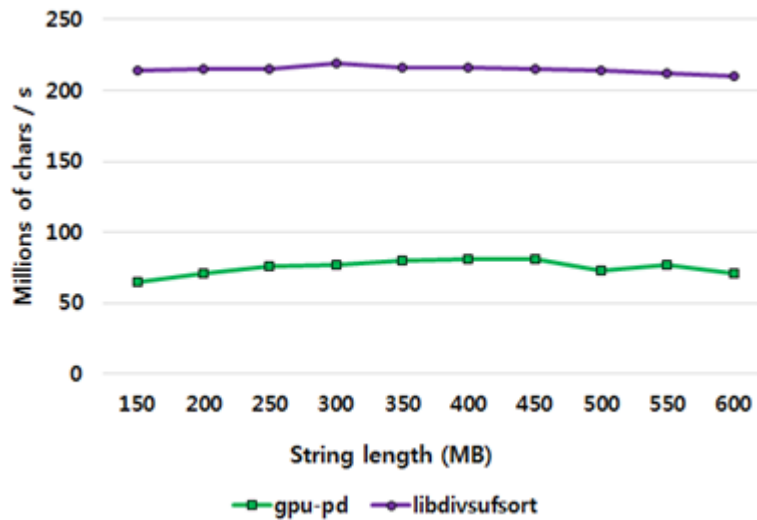


Figure 6.4: Suffix array construction throughput on a dataset consisting only of the repeated letter 'A'.

Figure 6.4 shows the results of the worst-case scenario for our GPU prefix-doubling implementation in terms of input type and size. Even with the worst-case inputs, our algorithm shows continuous growth in throughput. For *libdivsufsort* this test is easy because it is based on induced sorting method, which is much better adapted for dealing with large substrings composed of identical characters. Although the throughput of *libdivsufsort* is 2.7-3.3 times larger than the throughput of our GPU implementation over the input size range, we can clearly see that its throughput is slowly decreasing. We observe that on the input string of 600 MB, *libdivsufsort* runs in 2.85 seconds, compared with 8.47 seconds for *gpu-pd*. We notice that the curves of our algorithm in Figure 6.3 and 6.4 have relatively stable trends.

## Chapter 7

# Conclusions

In this thesis, we designed and implemented a lightweight, simple and memory efficient SACA using the CUDA programming model. We studied three main classes of algorithms for suffix array construction (Chapter 3). We demonstrated their working principles by describing the most prominent algorithms of each class. We described the algorithm of Manber and Myers [53], the algorithm of Larson and Sadakane [45], the algorithm of Kärkkäinen and Sanders [37], and the algorithm of Nong *et al.* [67].

We analyzed the parallelization properties of each SACA and observed the following. Induced sorting method exhibits a low level of parallelization capabilities according to Deo and Keely [17], and it remains unknown whether this method can be efficiently applied for suffix sorting on GPUs. The skew algorithm has several disadvantages when it comes to mapping it to GPU architectures. First, its recursive structure restricts parallelism efficiency. Second, it performs a large amount of redundant work by resorting triplets that are already sorted. Third, *bugs* (errors in the code) that appear in the recursive part are difficult to spot and fix on the GPU. Finally, the prefix-doubling approach has the following drawbacks. First, the original method of Manber and Myers does not distinguish between sorted and unsorted buckets, which results in unnecessary processing of already sorted buckets and relatively large data overhead. Second, the irregularity in buckets partitioning may lead to performance downgrade.

By analyzing the drawbacks of each method, we concluded that prefix-doubling SACA is the best candidate for translation to the GPU domain. Each step of this method can be efficiently parallelized. Modern and high-performance parallel primitives like segmented sort solve the problem of unnecessary scanning of sorted buckets and irregularity in buckets partitioning. Our GPU SACA is based on the prefix-doubling method. We combined some ideas of MM and LS to achieve the best result. To achieve simplicity and

compactness in the design of our implementation, we used the abstractions provided by Thrust, CUB and MGPU libraries.

We provided an overview of the basic CUDA concepts and GPU architecture. Additionally, we described the most efficient GPU parallel primitives that we incorporated into our implementation to enhance its performance, reduce memory consumption, make it more robust, simple and compact. We demonstrated the working principles of high-level abstractions and parallel constructs employed in our implementation.

We then presented the design and implementation of our algorithm. First, we formulated the principles of our implementation in a high-level pseudo-code. Then we elaborated on the role of each data structure and subroutine used in our implementation. We described each step of our algorithm and presented the source code for each part of our implementation. For clarity, we illustrated the effect of our source code by providing running examples.

Our primary goal was to design and implement a lightweight, memory-efficient, compact, and simple GPU SACA implementation. We achieved this goal by presenting the source code of our implementation, which is less than 300 lines of effective code. Additionally, we designed and implemented a fast and lightweight tool for checking the correctness of the suffix array, the *suffix array checker*. We reviewed the theory behind this tool and presented its implementation in pseudo-code. Our suffix array checker implementation verifies the correctness of the suffix array in  $O(n)$  time and consumes  $O(\sigma)$  extra memory space.

In the previous chapter, we evaluated the performance of our implementation by carrying out several experiments using real-world datasets. In our evaluation, we compared the runtime, throughput, and scalability against the fastest in practice multithreaded SACA, *libdivsufsort*. To emphasize the memory-efficiency of our algorithm, we assembled a corpora consisting of different real-world datasets with 600 MB input strings which is 6 times larger than the largest input processed by Wang *et al.* hybrid GPU implementation, 9 times larger than the largest input processed by Deo and Kelly, 15 times larger than the largest input processed by Osipov.

Experimentally, we established an upper bound for the memory consumption of our GPU algorithm to be  $25n$ , where  $n$  is the input length in bytes. The runtime of our algorithm scales linearly with the input string length and the curves of our implementation have relatively stable trends as our experimental results demonstrate. Our algorithm outperformed *libdivsufsort* in all experiments that we carried out, except the worst-case experiment, where induced sorting approach showed a complete dominance over prefix-doubling approach. Our GPU prefix-doubling implementation achieved a speedup of 16.1x over *libdivsufsort* on the `amazon_reviews` (600 MB) dataset. Our al-

gorithm showed a relatively high throughput of 144.7 millions of characters processed per second on the `twitter_support` (550 MB) dataset.

We believe that there is still room for improvement. The profiling showed that the main bottleneck in our implementation is the computation of the  $h$ -order *ISA*. As we discussed in Section 5.2, there are several ways to perform this computation. However, the efficiency of each method boils down to random reading and writing to global GPU memory. This step could be improved if the access to global memory is minimized or by finding a more efficient way to compute the  $h$ -order *ISA*.

Another improvement involves the translation of our implementation into a multi-GPU domain. For that purpose, one need to parallelize the architecture of our algorithm across interconnected GPUs. This task requires finding a way how to distribute the work evenly and efficiently between multiple GPUs, and how to combine results from each GPU such that they are consistent across different stages.

Recently, Fischer and Kurpicz [22] presented two new approaches for suffix array construction in a distributed environment. They proposed and implemented a distributed prefix-doubling SACA and a distributed induced copying SACA. They claim that their distributed induced copying SACA is the first of its kind and is currently the most lightweight SACA in a distributed setting. The breakthrough and advances in distributed SACA development may get us closer to the discovery of powerful and efficient techniques for multi-GPU SACA development.

Finally, we expect to see a GPU SACA based on an induced sorting method. Induced sorting SACAs of Itoh and Tanaka [33], Nong *et al.* [67] and Mori [61, 62] showed a great potential and efficiency. The question remains open whether it is feasible to utilize the efficiency of induced sorting technique for suffix sorting on the GPU. Translating this powerful algorithmic technique into GPU domain would give rise for the use of effective approaches to tackle some of the most challenging parallelization problems.

We conclude, that our main contribution is the design and implementation of the GPU prefix-doubling SACA that showed to be more memory-efficient and lightweight compared to the previous GPU SACAs of Deo and Kelly [17], Osipov [73] and Wang *et al.* [89]. We also showed that the design of our implementation is simpler and more compact than in previous GPU SACA approaches.

# Bibliography

- [1] AAMODT, T. M., FUNG, W. W. L., AND ROGERS, T. G. *General-Purpose Graphics Processor Architectures*. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2018.
- [2] ABOUELHODA, M. I., KURTZ, S., AND OHLEBUSCH, E. The enhanced suffix array and its applications to genome analysis. In *Algorithms in Bioinformatics, Second International Workshop, WABI 2002, Rome, Italy, September 17-21, 2002, Proceedings* (2002), pp. 449–463.
- [3] ADJEROH, D., BELL, T., AND MUKHERJEE, A. *The Burrows-Wheeler Transform:: Data Compression, Suffix Arrays, and Pattern Matching*. Springer Science & Business Media, 2008.
- [4] BARLAS, G. *Multicore and GPU Programming: An integrated approach*. Elsevier, 2014.
- [5] BARON, D., AND BRESLER, Y. Antisequential suffix sorting for bwt-based data compression. *IEEE Trans. Computers* 54, 4 (2005), 385–397.
- [6] BAXTER, S. *MGPU, version 2.12*, 2016. (Available from: <https://github.com/moderngpu/moderngpu>) [Accessed on May 26, 2019].
- [7] BAXTER, S. *The bzip2 and libbzip2 official home page, version 1.0.2*, 2004. (Available from: <http://www.sourceware.org/bzip2/>) [Accessed on May 26, 2019].
- [8] BENTLEY, J. L., AND MCILROY, M. D. Engineering a sort function. *Softw., Pract. Exper.* 23, 11 (1993), 1249–1265.
- [9] BENTLEY, J. L., SLEATOR, D. D., TARJAN, R. E., AND WEI, V. K. A locally adaptive data compression scheme. *Communications of the ACM* 29, 4 (1986), 320–330.
- [10] BLUM, M., AND KANNAN, S. Designing programs that check their work. *J. ACM* 42, 1 (1995), 269–291.

- [11] BURKHARDT, S., AND KÄRKKÄINEN, J. Fast lightweight suffix array construction and checking. In *Combinatorial Pattern Matching, 14th Annual Symposium, CPM 2003, Morelia, Michocán, Mexico, June 25-27, 2003, Proceedings* (2003), pp. 55–69.
- [12] BURROWS, M., AND J. WHEELER, D. A block-sorting lossless data compression algorithm. *Digital Systems Research Center Research Reports 1* (1995).
- [13] CHENG, J., GROSSMAN, M., AND MCKERCHER, T. *Professional Cuda C Programming*. John Wiley & Sons, 2014.
- [14] COOK, S. *CUDA programming: a developer's guide to parallel computing with GPUs*. Newnes, 2012.
- [15] CROCHEMORE, M., HANCART, C., AND LECROQ, T. *Algorithms on strings*. Cambridge University Press, 2007.
- [16] CROCHEMORE, M., AND RYTTER, W. *Jewels of stringology*. World Scientific, 2002.
- [17] DEO, M., AND KEELY, S. Parallel suffix array and least common prefix for the GPU. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP '13, Shenzhen, China, February 23-27, 2013* (2013), pp. 197–206.
- [18] ERCIYES, K. *Distributed and Sequential Algorithms for Bioinformatics*, vol. 23 of *Computational Biology*. Springer, 2015.
- [19] FARACH, M. Optimal suffix tree construction with large alphabets. In *38th Annual Symposium on Foundations of Computer Science, FOCS '97, Miami Beach, Florida, USA, October 19-22, 1997* (1997), pp. 137–143.
- [20] FARBER, R. *CUDA application design and development*. Elsevier, 2011.
- [21] FISCHER, J., AND KURPICZ, F. Dismantling divsufsort. In *Proceedings of the Prague Stringology Conference 2017, Prague, Czech Republic, August 28-30, 2017* (2017), pp. 62–76.
- [22] FISCHER, J., AND KURPICZ, F. Lightweight distributed suffix array construction. In *Proceedings of the Twenty-First Workshop on Algorithm Engineering and Experiments, ALENEX 2019, San Diego, CA, USA, January 7-8, 2019*. (2019), pp. 27–38.

- [23] FRANCESCHINI, G., AND MUTHUKRISHNAN, S. In-place suffix sorting. In *Automata, Languages and Programming, 34th International Colloquium, ICALP 2007, Wroclaw, Poland, July 9-13, 2007, Proceedings* (2007), pp. 533–545.
- [24] GIEGERICH, R., KURTZ, S., AND STOYE, J. Efficient implementation of lazy suffix trees. *Softw., Pract. Exper.* *33*, 11 (2003), 1035–1049.
- [25] GUIDE, D. Cuda c programming guide. *NVIDIA, May* (2019).
- [26] HE, B., GOVINDARAJU, N. K., LUO, Q., AND SMITH, B. Efficient gather and scatter operations on graphics processors. In *Proceedings of the ACM/IEEE Conference on High Performance Networking and Computing, SC 2007, November 10-16, 2007, Reno, Nevada, USA* (2007), p. 46.
- [27] HOARE, C. A. R. Algorithm 64: Quicksort. *Commun. ACM* *4*, 7 (1961), 321.
- [28] HOBEROCK, J., AND BELL, N. *Thrust, version 1.8.2*, 2015. (Available from: <https://thrust.github.io/>) [Accessed on May 26, 2019].
- [29] HOMANN, R., FLEER, D., GIEGERICH, R., AND REHMSMEIER, M. *mkESA: enhanced suffix array construction tool*. *Bioinformatics* *25*, 8 (2009), 1084–1085.
- [30] HON, W., SADAKANE, K., AND SUNG, W. Breaking a time-and-space barrier in constructing full-text indices. In *44th Symposium on Foundations of Computer Science (FOCS 2003), 11-14 October 2003, Cambridge, MA, USA, Proceedings* (2003), pp. 251–260.
- [31] HUFFMAN, D. A. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE* *40*, 9 (Sep. 1952), 1098–1101.
- [32] INTEL CORPORATION. Intel cilk plus, an extension to the c and c++ languages to support data and task parallelism, 2017. (Available from: <https://www.cilkplus.org/>) [Accessed on May 26, 2019].
- [33] ITOH, H., AND TANAKA, H. An efficient method for in memory construction of suffix arrays. In *Sixth International Symposium on String Processing and Information Retrieval and Fifth International Workshop on Groupware, SPIRE/CRIWG 1999, Cancun, Mexico, September 21-24, 1999* (1999), pp. 81–88.

- [34] KÄRKKÄINEN, J., AND KEMPA, D. Engineering a lightweight external memory suffix array construction algorithm. In *Proceedings of the 2nd International Conference on Algorithms for Big Data , Palermo, Italy, April 07-09, 2014.* (2014), pp. 53–60.
- [35] KÄRKKÄINEN, J., KEMPA, D., PUGLISI, S. J., AND ZHUKOVA, B. Engineering external memory induced suffix sorting. In *Proceedings of the Ninteenth Workshop on Algorithm Engineering and Experiments, ALENEX 2017, Barcelona, Spain, Hotel Porta Fira, January 17-18, 2017.* (2017), pp. 98–108.
- [36] KÄRKKÄINEN, J., AND SANDERS, P. Simple linear work suffix array construction. In *Automata, Languages and Programming, 30th International Colloquium, ICALP 2003, Eindhoven, The Netherlands, June 30 - July 4, 2003. Proceedings* (2003), pp. 943–955.
- [37] KÄRKKÄINEN, J., SANDERS, P., AND BURKHARDT, S. Linear work suffix array construction. *J. ACM* 53, 6 (2006), 918–936.
- [38] KARP, R. M., MILLER, R. E., AND ROSENBERG, A. L. Rapid identification of repeated patterns in strings, trees and arrays. In *Proceedings of the 4th Annual ACM Symposium on Theory of Computing, May 1-3, 1972, Denver, Colorado, USA* (1972), pp. 125–136.
- [39] KIM, D. K., JO, J., AND PARK, H. A fast algorithm for constructing suffix arrays for fixed-size alphabets. In *Experimental and Efficient Algorithms, Third International Workshop, WEA 2004, Angra dos Reis, Brazil, May 25-28, 2004, Proceedings* (2004), pp. 301–314.
- [40] KIRK, D. B., AND HWU, W. W. *Programming Massively Parallel Processors - A Hands-on Approach.* Morgan Kaufmann, 2010.
- [41] KNUTH, D. Section 5.2. 4: Sorting by merging. *The Art of Computer Programming 3* (1998), 158–168.
- [42] KO, P., AND ALURU, S. Space efficient linear time construction of suffix arrays. *J. Discrete Algorithms* 3, 2-4 (2005), 143–156.
- [43] LAO, B., NONG, G., CHAN, W. H., AND PAN, Y. Fast induced sorting suffixes on a multicore machine. *The Journal of Supercomputing* 74, 7 (2018), 3468–3485.
- [44] LAO, B., NONG, G., CHAN, W. H., AND XIE, J. Y. Fast in-place suffix sorting on a multicore computer. *IEEE Trans. Computers* 67, 12 (2018), 1737–1749.

- [45] LARSSON, N. J., AND SADAKANE, K. Faster suffix sorting. *Theor. Comput. Sci.* 387, 3 (2007), 258–272.
- [46] LI, H., AND DURBIN, R. Fast and accurate long-read alignment with burrows-wheeler transform. *Bioinformatics* 26, 5 (2010), 589–595.
- [47] LI, Z., LI, J., AND HUO, H. Optimal in-place suffix sorting. In *2018 Data Compression Conference, DCC 2018, Snowbird, UT, USA, March 27-30, 2018* (2018), p. 422.
- [48] LINDHOLM, E., NICKOLLS, J., OBERMAN, S. F., AND MONTRYM, J. NVIDIA tesla: A unified graphics and computing architecture. *IEEE Micro* 28, 2 (2008), 39–55.
- [49] MACKENZIE, C. E. *Coded-Character Sets: History and Development*. Addison-Wesley Longman Publishing Co., Inc., 1980.
- [50] MÄKINEN, V. Compact suffix array - A space-efficient full-text index. *Fundam. Inform.* 56, 1-2 (2003), 191–210.
- [51] MÄKINEN, V., BELAZZOUGUI, D., CUNIAL, F., AND TOMESCU, A. I. *Genome-Scale Algorithm Design: Biological Sequence Analysis in the Era of High-Throughput Sequencing*. Cambridge University Press, 2015.
- [52] MALYSHEV, D. *Archon, version 4*, 2015. (Available from: <https://github.com/kvark/dark-archon/releases>) [Accessed on May 26, 2019].
- [53] MANBER, U., AND MYERS, E. W. Suffix arrays: A new method for on-line string searches. *SIAM J. Comput.* 22, 5 (1993), 935–948.
- [54] MANISCALCO, M. *Msort, version 3*, 2011. (Available from: <https://github.com/michaelmaniscalco/msort>) [Accessed on May 26, 2019].
- [55] MANISCALCO, M. A., AND PUGLISI, S. J. An efficient, versatile approach to suffix sorting. *ACM Journal of Experimental Algorithmics* 12 (2007), 1.2:1–1.2:23.
- [56] MANZINI, G., AND FERRAGINA, P. Engineering a lightweight suffix array construction algorithm. *Algorithmica* 40, 1 (2004), 33–50.
- [57] MATLOFF, N. *Parallel computing for data science: with examples in R, C++ and CUDA*. Chapman and Hall/CRC, 2015.
- [58] MERRILL, D. *CUB, version 1.8.0*, 2018. (Available from: <https://nvlabs.github.io/cub/index.html>) [Accessed on May 26, 2019].

- [59] MERRILL, D., AND GARLAND, M. Single-pass parallel prefix scan with decoupled look-back. Tech. rep., NVIDIA Technical Report NVR-2016-002, 2016.
- [60] MOHAMED, H., AND ABOUELHODA, M. Parallel suffix sorting based on bucket pointer refinement. In *Biomedical Engineering Conference (CIBEC), 2010 5th Cairo International* (2010), IEEE, pp. 98–102.
- [61] MORI, Y. *libdivsufsort, version 2.0.2*, 2015. (Available from: <https://github.com/y-256/libdivsufsort>) [Accessed on May 26, 2019].
- [62] MORI, Y. *sais, version 2.4.1*, 2010. (Available from: <https://sites.google.com/site/yuta256/sais>) [Accessed on May 26, 2019].
- [63] MORI, Y. Suffix array benchmarking contest, 2015. (Available from: [https://github.com/y-256/libdivsufsort/blob/wiki/SACA\\_Benchmarks.md](https://github.com/y-256/libdivsufsort/blob/wiki/SACA_Benchmarks.md)) [Accessed on May 26, 2019].
- [64] NAVARRO, G. *Compact Data Structures - A Practical Approach*. Cambridge University Press, 2016.
- [65] NICKOLLS, J., BUCK, I., GARLAND, M., AND SKADRON, K. Scalable parallel programming with cuda. In *ACM SIGGRAPH 2008 classes* (2008), ACM, p. 16.
- [66] NONG, G. Practical linear-time  $O(1)$ -workspace suffix sorting for constant alphabets. *ACM Trans. Inf. Syst.* 31, 3 (2013), 15.
- [67] NONG, G., ZHANG, S., AND CHAN, W. H. Linear suffix array construction by almost pure induced-sorting. In *2009 Data Compression Conference (DCC 2009), 16-18 March 2009, Snowbird, UT, USA* (2009), pp. 193–202.
- [68] NONG, G., ZHANG, S., AND CHAN, W. H. Linear time suffix array construction using d-critical substrings. In *Combinatorial Pattern Matching, 20th Annual Symposium, CPM 2009, Lille, France, June 22-24, 2009, Proceedings* (2009), pp. 54–67.
- [69] NONG, G., ZHANG, S., AND CHAN, W. H. Two efficient algorithms for linear time suffix array construction. *IEEE Trans. Computers* 60, 10 (2011), 1471–1484.
- [70] OHLEBUSCH, E. *Bioinformatics Algorithms: Sequence Analysis, Genome Rearrangements, and Phylogenetic Reconstruction*. Oldenbusch Verlag, 2013.

- [71] OHLEBUSCH, E., BELLER, T., AND ABOUELHODA, M. I. Computing the burrows-wheeler transform of a string and its reverse in parallel. *J. Discrete Algorithms* 25 (2014), 21–33.
- [72] OPENMP ARCHITECTURE REVIEW BOARD. *OpenMP, version 5.0*, 2018. (Available from: <https://www.openmp.org/>) [Accessed on May 26, 2019].
- [73] OSIPOV, V. Parallel suffix array construction for shared memory architectures. In *String Processing and Information Retrieval - 19th International Symposium, SPIRE 2012, Cartagena de Indias, Colombia, October 21-25, 2012. Proceedings* (2012), pp. 379–384.
- [74] PUGLISI, S. J., SMYTH, W. F., AND TURPIN, A. A taxonomy of suffix array construction algorithms. *ACM Comput. Surv.* 39, 2 (2007), 4.
- [75] ROSE, C. *Cuda Succinctly*. CreateSpace Independent Publishing Platform, 2017.
- [76] RYABKO, B. Y. Data compression by means of a "book stack". *Problemy Peredachi Informatsii* 16, 4 (1980), 16–21.
- [77] SANDERS, J., AND KANDROT, E. *CUDA by example: an introduction to general purpose GPU programming*. Addison-Wesley, 2011.
- [78] SCHINDLER, M. *gzip homepage*, version 1.12, 2002. (Available from: <http://www.compressconsult.com/gzip/>) [Accessed on May 26, 2019].
- [79] SCHÜRMAN, K., AND STOYE, J. An incomplex algorithm for fast suffix array construction. In *Proceedings of the Seventh Workshop on Algorithm Engineering and Experiments and the Second Workshop on Analytic Algorithmics and Combinatorics, ALENEX /ANALCO 2005, Vancouver, BC, Canada, 22 January 2005* (2005), pp. 78–85.
- [80] SEWARD, J. On the performance of BWT sorting algorithms. In *Data Compression Conference, DCC 2000, Snowbird, Utah, USA, March 28-30, 2000*. (2000), pp. 173–182.
- [81] SHUN, J., BLELLOCH, G. E., FINEMAN, J. T., GIBBONS, P. B., KYROLA, A., SIMHADRI, H. V., AND TANGWONGSAN, K. Brief announcement: the problem based benchmark suite. In *24th ACM Symposium on Parallelism in Algorithms and Architectures, SPAA '12, Pittsburgh, PA, USA, June 25-27, 2012* (2012), pp. 68–70.

- [82] SIPSER, M. Introduction to the theory of computation. *SIGACT News* 27, 1 (1996), 27–29.
- [83] SOYATA, T. *GPU Parallel Program Development Using CUDA*. Chapman and Hall/CRC, 2018.
- [84] TARHIO, J., AND PELTOLA, H. String matching in the DNA alphabet. *Softw., Pract. Exper.* 27, 7 (1997), 851–861.
- [85] UJALDON, M. CUDA achievements and GPU challenges ahead. In *Articulated Motion and Deformable Objects - 9th International Conference, AMDO 2016, Palma de Mallorca, Spain, July 13-15, 2016, Proceedings* (2016), pp. 207–217.
- [86] VAIDYA, B. *Hands-On GPU-Accelerated Computer Vision with OpenCV and CUDA: Effective techniques for processing complex image data in real time using GPUs*. Packt Publishing, 2018.
- [87] VITTER, J. S. Design and analysis of dynamic huffman codes. *Journal of the ACM (JACM)* 34, 4 (1987), 825–845.
- [88] WALKER, R. C., AND GOETZ, A. W. *Electronic Structure Calculations on Graphics Processing Units: From Quantum Chemistry to Condensed Matter Physics*. John Wiley & Sons, 2016.
- [89] WANG, L., BAXTER, S., AND OWENS, J. D. Fast parallel skew and prefix-doubling suffix array construction on the GPU. *Concurrency and Computation: Practice and Experience* 28, 12 (2016), 3466–3484.
- [90] WASSERMAN, H., AND BLUM, M. Software reliability via run-time result-checking. *J. ACM* 44, 6 (1997), 826–849.
- [91] WILLIAMS, JR., L. F. A modification to the half-interval search (binary search) method. In *Proceedings of the 14th Annual Southeast Regional Conference* (New York, NY, USA, 1976), ACM-SE 14, ACM, pp. 95–101.
- [92] WILT, N. *The Cuda Handbook: A Comprehensive Guide to GPU Programming*. Pearson Education, 2013.
- [93] WITTEN, I. H., NEAL, R. M., AND CLEARY, J. G. Arithmetic coding for data compression. *Commun. ACM* 30, 6 (1987), 520–540.

- [94] YAP, C. A real elementary approach to the master recurrence and generalizations. In *Theory and Applications of Models of Computation - 8th Annual Conference, TAMC 2011, Tokyo, Japan, May 23-25, 2011. Proceedings* (2011), pp. 14–26.