Fast Parallel Suffix Array on the GPU

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

University of California, Davis, CA, USA

D. E. Shaw Research, NY, USA

27th August 2015

Acknowledgments

- We acknowledge Mrinal Deo, Vitaly Osipov and Jacopo Pantaleoni for sharing their original data for comparisons.
- Thanks Owens group for good advice on implementation and feedback on early drafts of the paper.
- We appreciate the funding support of the National Science Foundation under grants OCI-1032859 and CCF-1017399, and UC Lab Fees Research Program Award 12-LR-238449.

Fundamental Concepts

- The Suffix Array (SA) and Inverse Suffix Array (ISA):

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

- The Burrows-Wheeler Transform (BWT):

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

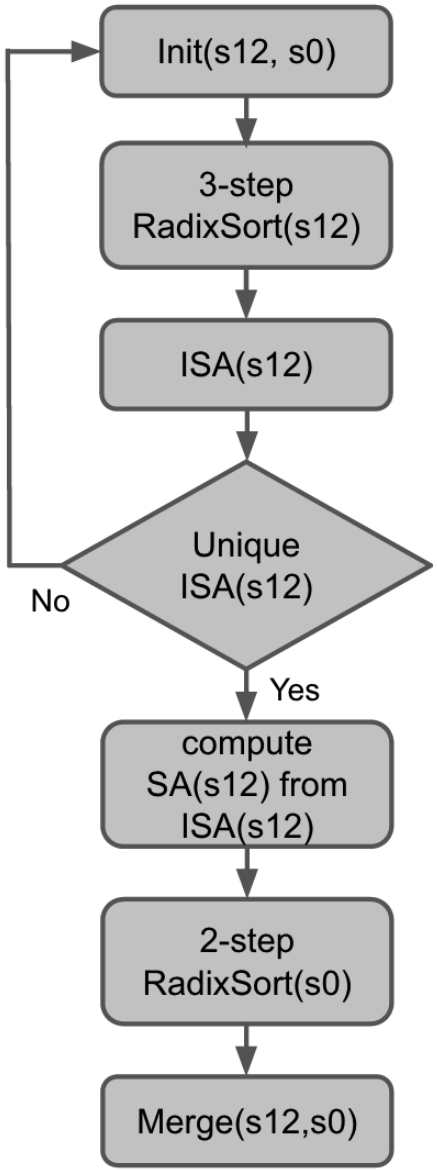
i	Suffix	Sorted Suffix	SA[i]	ISA[i]	Sorted Rotations	BWT[i]
0	banana$	$	6	4	$banana	a
1	anana$	a$	5	3	a$banan	n
2	nana$	ana$	3	6	ana$ban	n
3	ana$	anana$	1	2	anana$b	b
4	na$	banana$	0	5	banana$	$
5	a$	na$	4	1	na$bana	a
6	$	nana$	2	0	nana$ba	a

- The FM (Full-text, Minute-space) index

Suffix Array Construction Algorithms (SACAs)

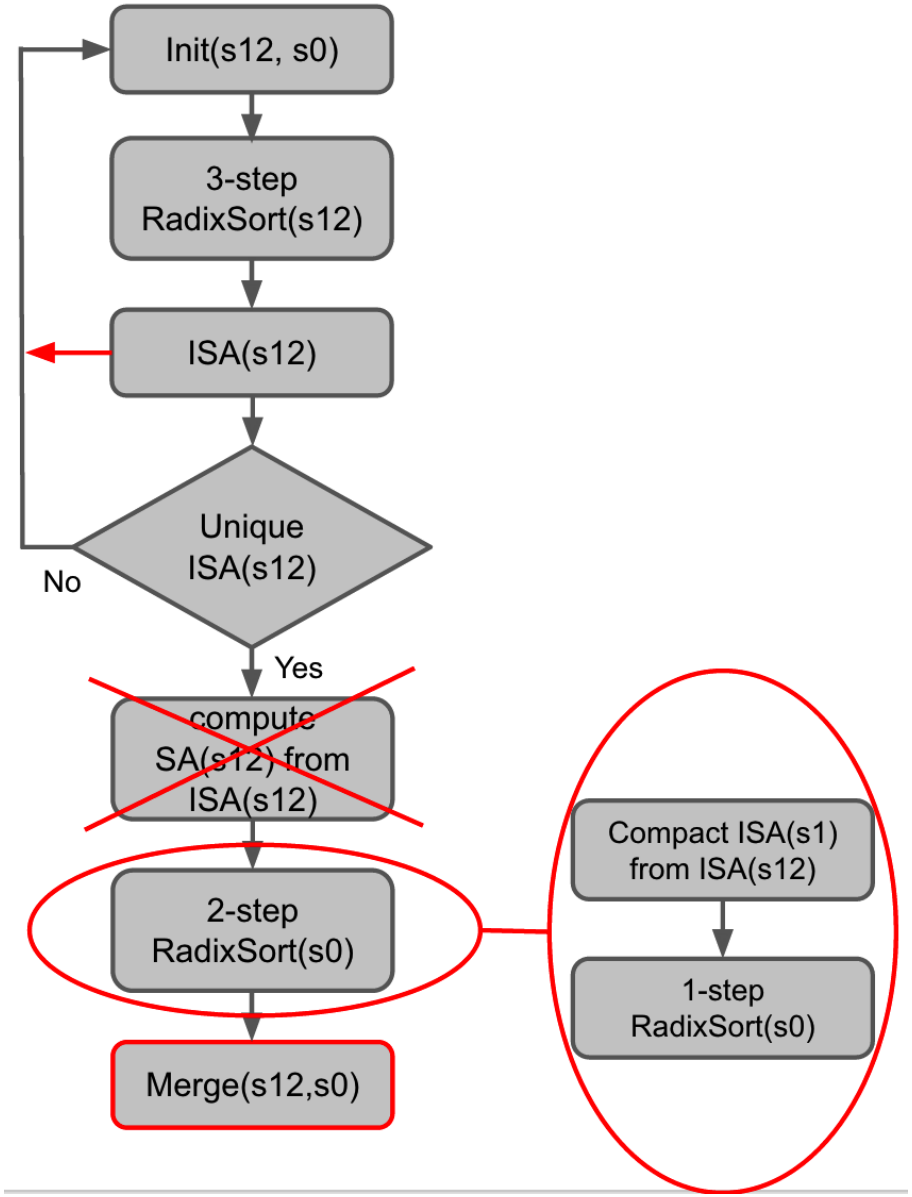
- Prefix-doubling Algorithms $\mathcal{O}(n \log n)$
  - Manber and Myers (MM)
  - Larsson and Sadakane (LS)
- Recursive Algorithms $\mathcal{O}(n)$
  - Kärkkäinen and Sanders (KS)
- Induced Copying $\mathcal{O}(n)$
  - Two-stage induced copying
  - Pure induced copying (SA-IS)

Skew-based approach by Deo and Keely



- Advantages: Mapped to GPU based primitives; Almost no communications between CPU and GPU
- Limitations: Some redundant steps; There is communication and synchronization between CPU and GPU; Load-imbalanced merge primitive.

Our Solution

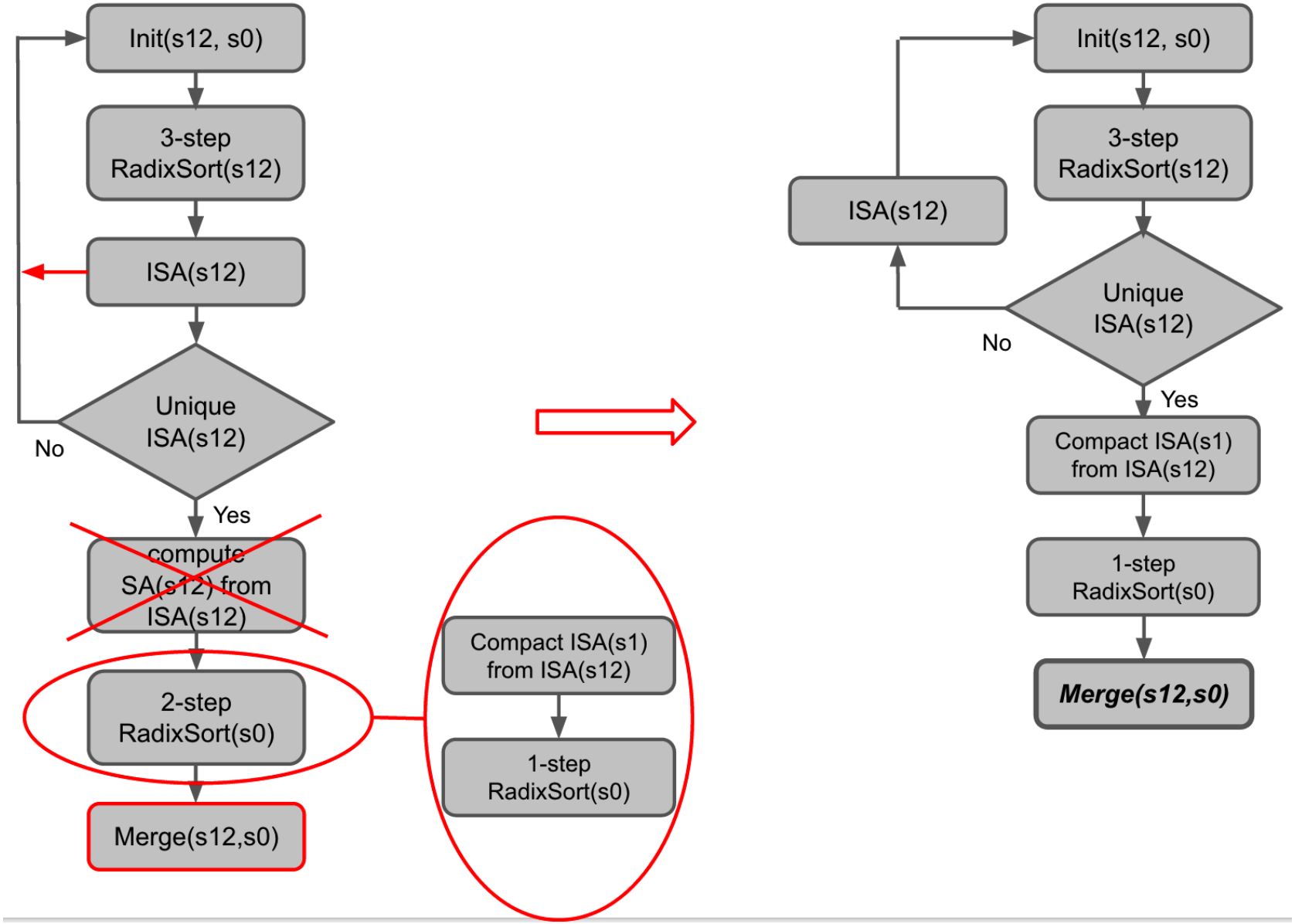


- Only compute ranks (ISA(s12)) when the suffixes are not fully sorted, thus saves us from computing SA(s12) from ISA(s12) at the end of recursion;
- Faster 1-step radix sort + trivial compact operation replacing 2-step radix sort;

Resulting in only 2/3 memory bandwidth in the recursive part and 3 fewer memory transactions in the last round

- Proposed a more efficient load-balanced merge primitive.

Our Solution



Efficient Merge Primitive

Two keys

- dividing the two sorted inputs into independent chunks of equal sized work;
- ensuring that the outputs of each of those chunks of work are contiguous in the final merged output.

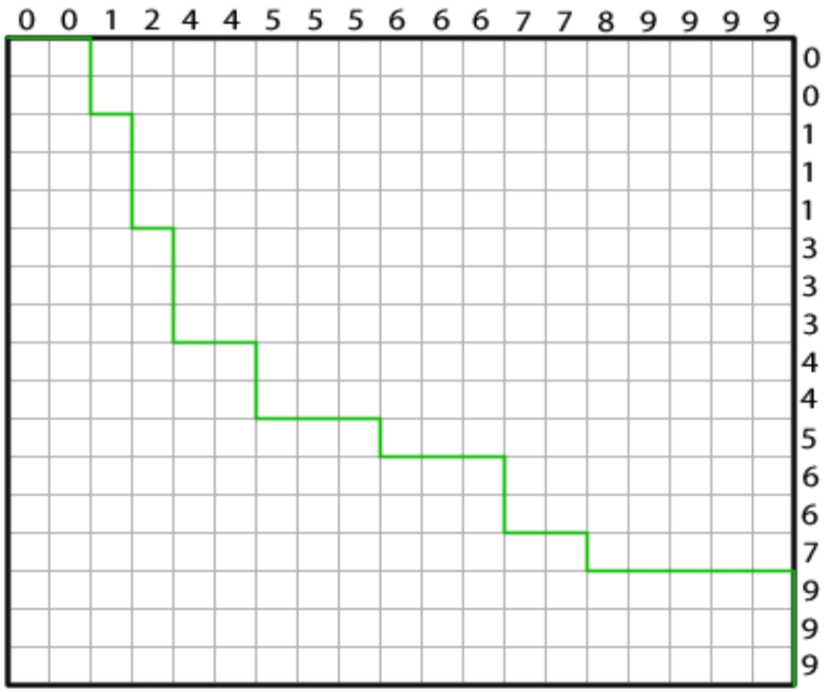
Identify split points

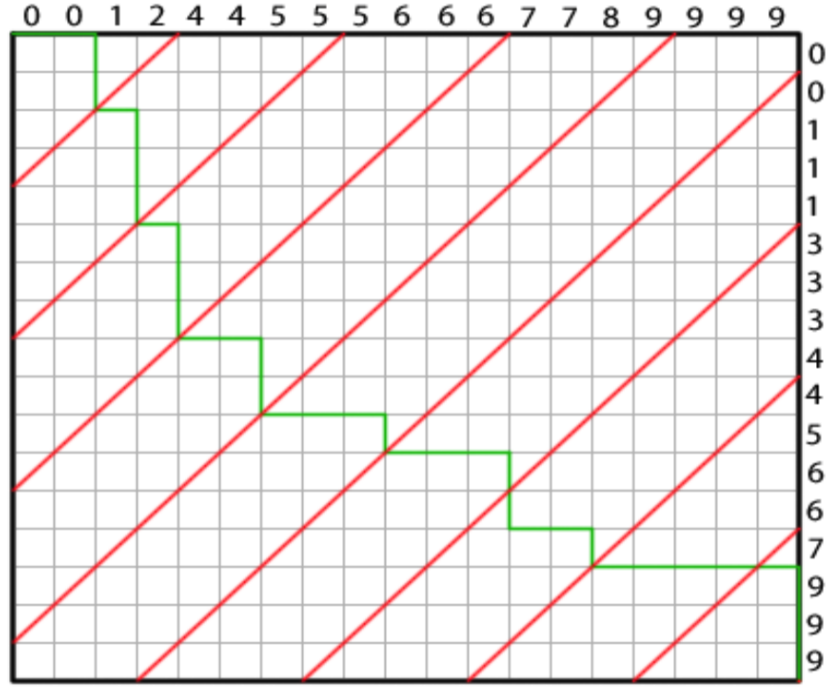
- Use merge path to transform a 2-D search to 1-D search along a diagonal that connects the two input arrays.

[0] Code is available at `http://nvlabs.github.io/moderngpu/merge.html`.

Efficient Merge Primitive

Merge Path

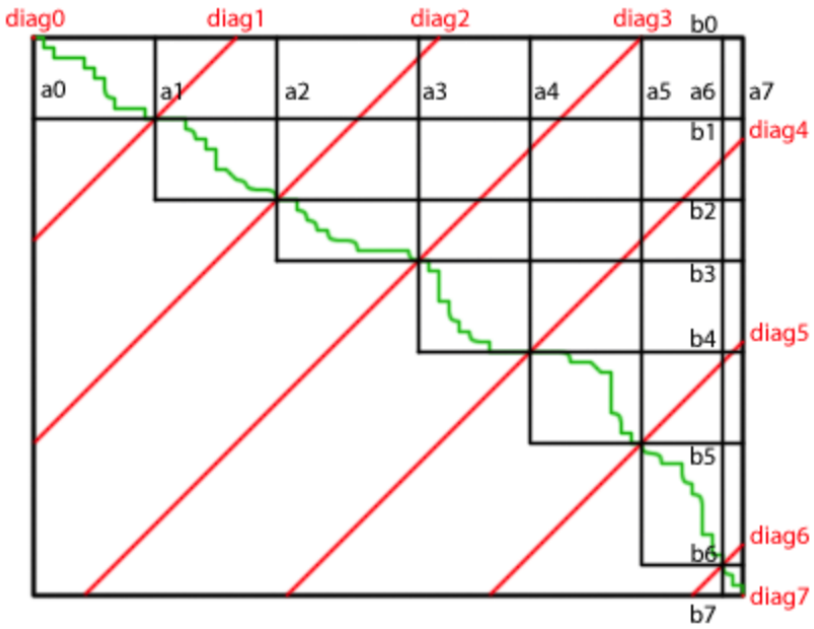


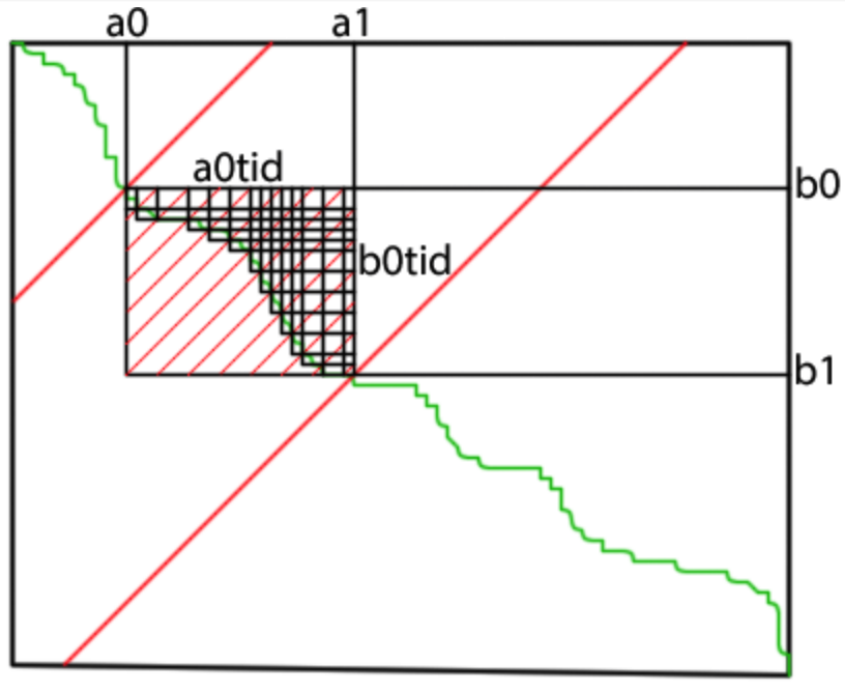


[0]Images obtained from `https://nvlabs.github.io/moderngpu/bulkinsert.html`.

Efficient Merge Primitive

Merge Path





[0]Images obtained from `https://nvlabs.github.io/moderngpu/merge.html`.

Parallel Skew algorithm comparisons

```
1  dk-SA (int* T, int* SA, int length)
2  Initialize Mod12() // form triplet s12, s0
3  RadixSort (s12) // LSD radix sort 1st char
4  RadixSort (s12) // LSD radix sort 2nd char
5  RadixSort (s12) // LSD radix sort 3rd char
6  lexicRankOfTriplets (s12)
7  if (!allUniqueRanks) then
8       dk-SA () // Recurse
9       storeUniqueRanks()
10 else
11      computeSAFromUniqueRank ()
12 RadixSort (ISA[1])
13 RadixSort (s0)
14 Merge (s0,s12)
```

```
1  skew-SA (int* T, int* SA, int length)
2  Initialize Mod12() // form triplet s12, s0
3  RadixSort (s12) // LSD radix sort 1st char
4  RadixSort (s12) // LSD radix sort 2nd char
5  RadixSort (s12) // LSD radix sort 3rd char
6  if (!allUniqueRanks) then
7       lexicRankOfTriplets (s12)
8       skew-SA ()// Recurse
9       storeUniqueRanks()
10 Compact (ISA[12]) // compact out the order
       of ISA[1]
11 RadixSort (s0)
12 Merge (s0,s12)
```

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

Limitations of Parallel Skew

- Recursive, cannot parallelize across iterations;
- Have to re-sort some fully sorted suffixes in order to keep the recursive routine.

A Hybrid of Parallel Skew/Prefix-doubling

A better fit for modern GPU architectures

- Keep the first step and the final merge stage of skew;
- After reducing the string size by 2/3, we transition to prefix-doubling;
- Sort by (ISA[SA[i]+δ], ISA[SA[i]+2δ]) pairs;
- High-performance prefix-doubling using our efficient *segmented sort*;
- Filtered out suffixes that are fully sorted at the end of each iteration.

Improvements against Osipov's plain prefix-doubling

- 3-character radix sort is less expensive than Osipov's 4-character one;
- *Segmented sort* has better locality than radix sort across global memory;
- Induction step is cheaper than radix sort in sorting the remaining 1/3 suffixes.

Segmented Sort

- Input: segments of unsorted items
- Output: same lists of segments within which items are sorted
- Challenge: variation in the size and number of segments

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

Parallel Prefix-doubling based Algorithm comparisons

```
1  Initialize SA_4 by sorting suffixes by
   their first 4 characters
2  Initialize ISA_4[i] with the 4-rank of
   i =head of i's 4-group in SA_4
3  size = n, h = 4
4  while size > 0 do
5      Scan SA_h and generate tuples
       (SA_h[j] − h, ISA_h[SA_h[j] −
       h], ISA_h[SA_h[j]])
6      RadixSort tuples by 2nd component
           stably // contains SA_2h
7      Refine h-heads of h-groups
           // Re-rank
8      Update ISA_2h // contains ISA_2h
9      Filter and Compact SA_2h
10     size = size of SA_2h
11     h = h * 2
```

```
1  Initialize Mod12() // form triplets s12,
       s0
2  RadixSort (s12) // 25-bit radix sort
       on triplets s12
3  ComputeRanks ISA_[12]
4  size = (2/3)n, h = 6
5  while size > 0 do
6      SegmentedSort
       (ISA[SA[i]+h], ISA[SA[i]+2h])
7      Update ISA_2h and Compact SA_2h
8      size = size of SA_2h
9      h = h * 2
10 Compact (ISA_[12]) // compact out the
       order of ISA_[1]
11 RadixSort (s0)
12 Merge (s0,s12)
```

Figure 2: Left, Osipov's parallel prefix-doubling description; right, our skew/prefix-doubling.

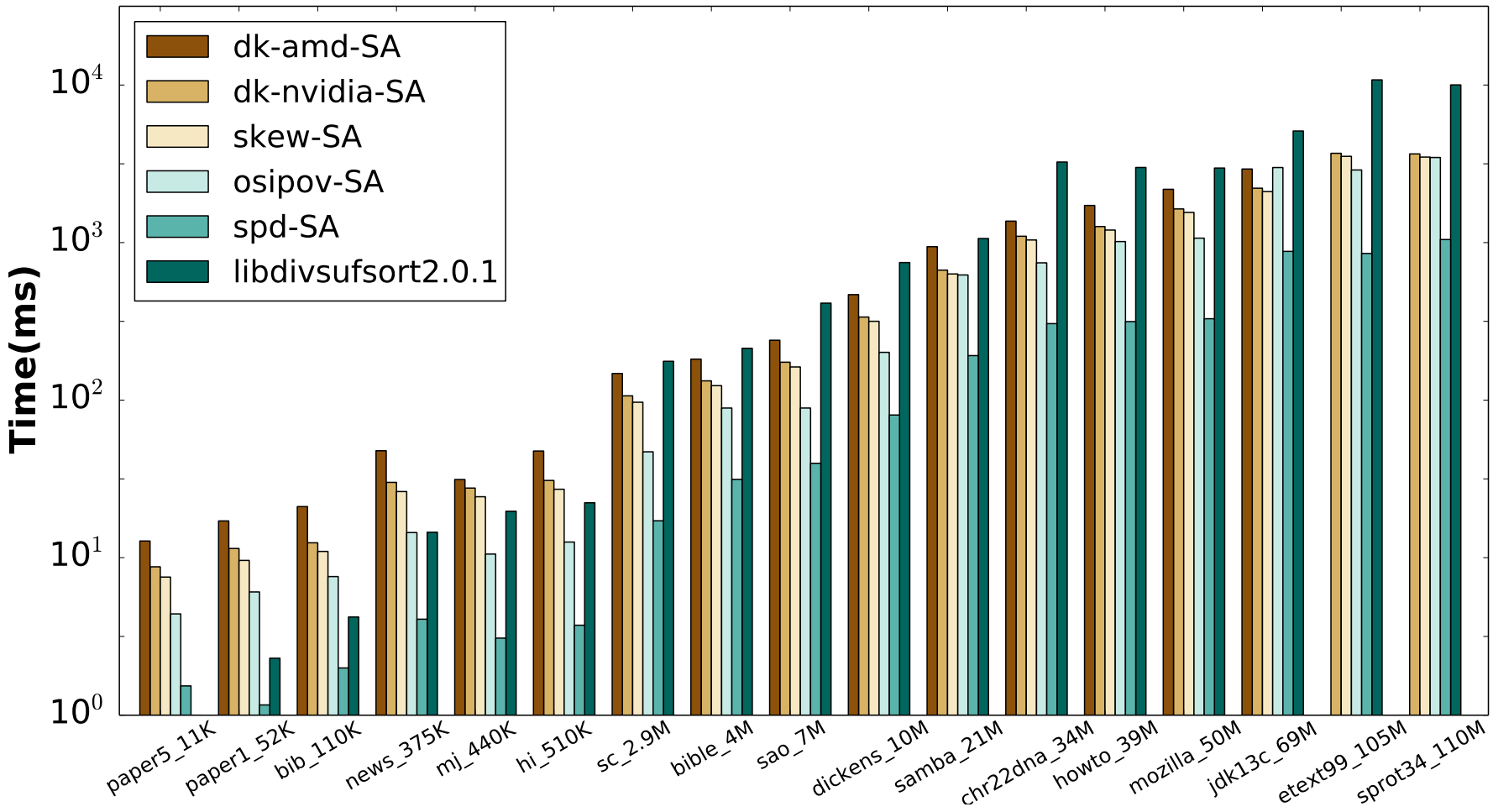
Skew vs prefix-doubling

- Skew is essentially a "prefix tripling" technique, tripling the pace at which it samples its ranks each round;
- 2-integer segmented sort of prefix-doubling is much faster than the 3-integer radix sort of skew;
- In its radix sort, skew uses the most significant digit simply to get the suffix back in its original segment, which comes for free with prefix-doubling's segmented sort;
- Skew cannot drop fully-sorted suffixes for it needs to transform their ranks into the new coordinate system in which they will be sampled by the remaining unsorted suffixes, but with prefix-doubling, suffixes are ranked in the same coordinate system throughout the computation;
- Skew has a solid reduction ratio of 0.67, regardless of the data while prefix-doubling has a worst-case reduction ratio of 1.0 but has a more favorable reduction ratio on real-world text.

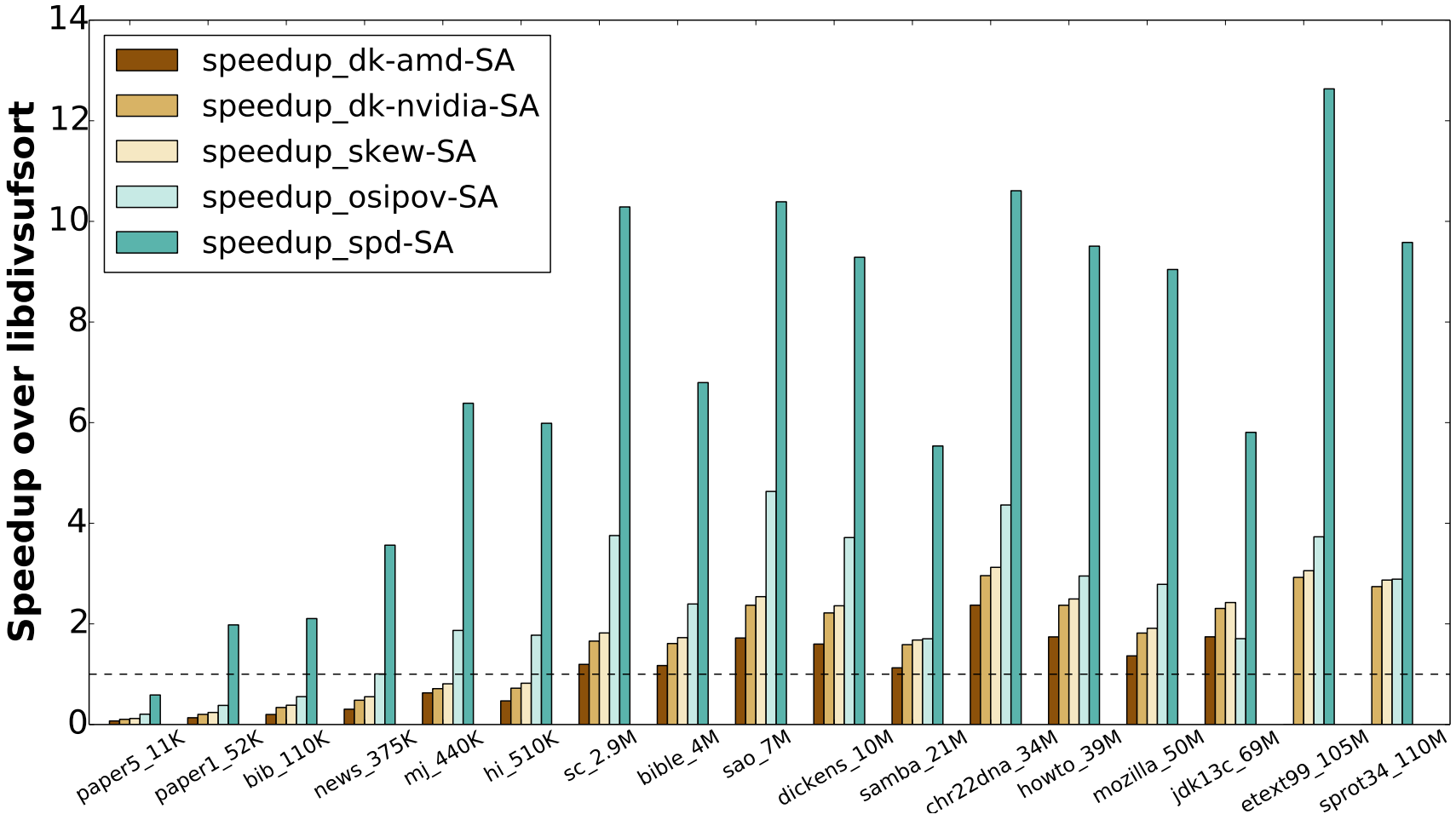
Experimental Setup

- An Intel Core i7-3770K 3.5 GHz 4-core machine with 16 GB RAM and 8 MB L3 cache.
- An NVIDIA Tesla K20c GPU (launch date: November 2012; process: 28 nm; peak single-precision floating-point throughput: 3.524 TFLOPS; peak memory bandwidth: 208 GB/s)
- We use CUDA 6.0 and Visual Studio 2010 on 64-bit Windows 7 platform
- The input strings range in size from 10 KB to 110 MB and are collected from the Calgary Corpus, Large Canterbury Corpus, Manzini's Corpus, Protein Corpus, and Silesia Corpus
- We compare the four GPU implementation results against Mori's highly tuned, OpenMP-assisted CPU implementation libdivsufsort 2.0.1 based on induced copying on a 4-core PC, using its own internal runtime measurement, which excludes disk access time.

Runtime Comparisons



Speedup Comparisons



Special Experiments

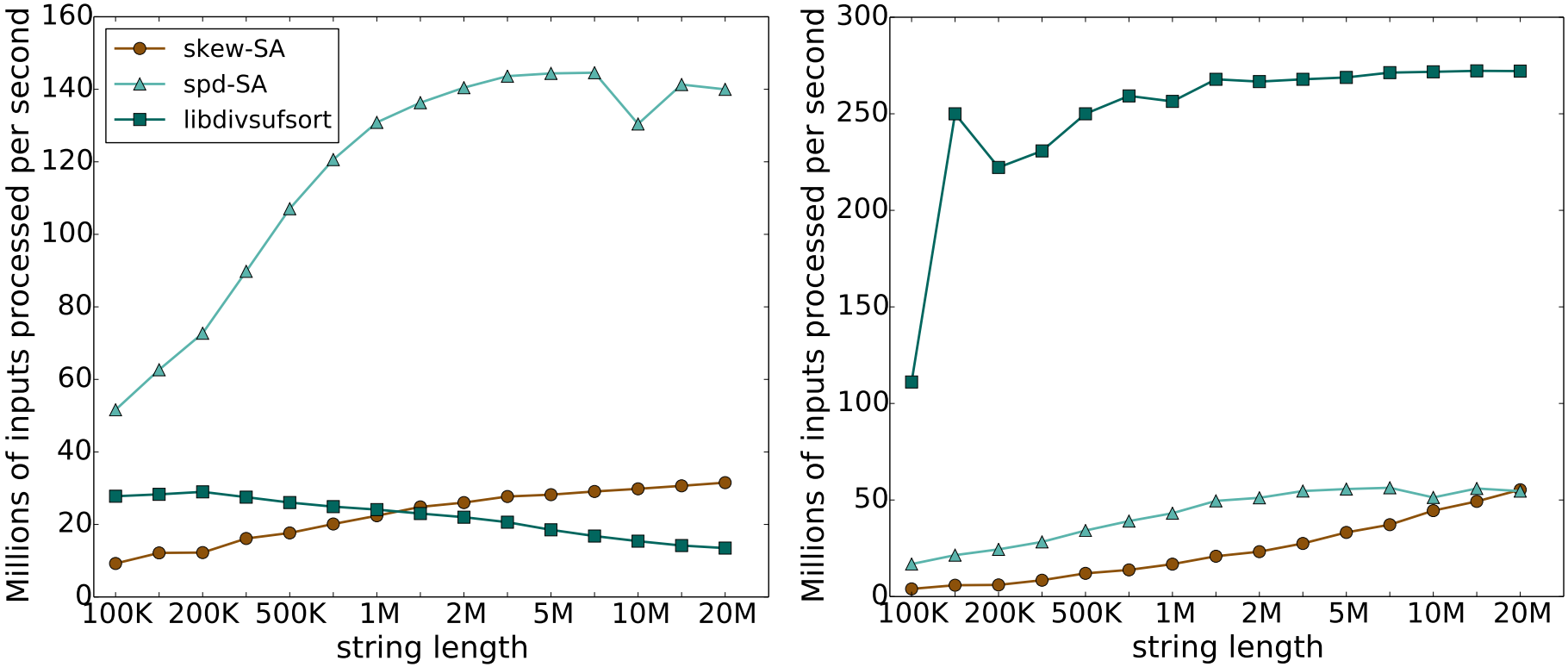


Figure 3: Left, throughput on plain text "enwik8" as dataset size scales; right, throughput on a dataset consisting only of the repeated letter 'A', using the same legend as left graph.

Application tools implementation

Table 1: Throughput of the BWT and FM index's backward_search using our spd-SA.

Dataset	enwik8	chr22.dna
BWT[1] (Millions of characters/s)	132.5	116.4
FM index (Millions of characters/s)	28.6	77

[1] Code is available at `https://github.com/cudpp/cudpp`

Conclusions

- Much of the interesting work in GPU computing has been the result of brute-force techniques, judiciously applied;
- Of the three classes of suffix array construction algorithms, skew is perhaps the most suitable for brute-force methods, and was chosen by Deo and Keely, and ourselves when we began our work;
- The maturation of GPU computing is leading to the development of elegant, efficient, load-balanced algorithmic building blocks that are designed for, and run well on the GPU;
- The merge and segmented sort implementations in this paper make the difference between an SACA that is uncompetitive vs. an SACA that is best in class.

Future Work

We expect that the next frontier in GPU SACAs will be tackling the third class of SACAs—induced copying. The research challenge is to determine whether the inherent algorithmic efficiency of their CPU implementation will translate into the GPU domain.

- Thank You.

Our skew implementation can be found in the most updated version of CUDA Data Parallel Primitives Library (CUDPP2.2)
http://cudpp.github.io/

- Questions?

Contact Info: leywang@ucdavis.edu