A Case for Term Weighting using a Dictionary on GPUs

Toshiaki Wakatsuki, Atsushi Keyaki, and Jun Miyazaki

Department of Computer Science, School of Computing,
Tokyo Institute of Technology
wakatsuki@lsc.cs.titech.ac.jp, keyaki@lsc.cs.titech.ac.jp,
miyazaki@cs.titech.ac.jp

Abstract. This paper demonstrates a fast Okapi's BM25 term weighting method on GPUs for information retrieval by combining a GPU-based dictionary using a succinct data structure and data parallel primitives. The problem of handling documents on GPUs is to processing variable length strings such as a document itself and a word. Processing variable size of data causes many idle cores, i.e., load imbalances among threads, due to the SIMD nature of GPU architecture. Our term weighting method is carefully composed of efficient data parallel primitives to avoid load imbalance. Additionally, we implemented a haigh performance compressed dictionary on GPUs. By using this dictionary, words are converted into IDs so that costly string comparisons can be avoided. Our experimental results revealed that the proposed term weighting method on GPUs performs up to 5x faster than the MapReduce-based one on multi-core CPUs.

Keywords: GPGPU, term weighting, dictionary, parallel primitive

1 Introduction

With the spread of computers and the Web, large amounts of documents have been created. In order to process these large amounts of documents in a practical time, a method for high throughput document processing is necessary. Many researches have been carried out utilizing commodity hardware to address to this issue. MapReduce is one of the parallel programming models for processing large-scale data on large computer clusters [2]. MapReduce has widely been used for processing documents such as term weighting and inverted index construction [8].

On the other hand, GPUs have widely been adopted in many researches and applications, because they offer high performance computing by many cores and high memory bandwidth. Emerging general-purpose computing on GPUs has led to the drastic expansion of application field. For example, many high performance numerical calculation libraries support GPUs for their floating point operations [1]. Furthermore, even high performance non-numerical calculation libraries

[1] https://developer.nvidia.com/gpu-accelerated-libraries

become to support GPUs. For instance, a graph processing library utilizes GPU computing primitives and optimization strategies to achieve a balance between performance and expressiveness [18]. The effective use of data parallel primitives is the keys to construct an high performance program. Although string processing has been recognized to be unsuitable for GPUs, some GPU-accelerated algorithms were proposed such as string matching for database applications [16].

There have been some variations of MapReduce for various platforms such as GPUs [3] as well as shared memory processors [17] and clusters [2]. However, MapReduce cannot draw out the potential power of GPUs due to the load imbalance among tasks.

In this paper, we propose an implementation of an efficient dictionary on GPUs as a new data parallel primitive for document processing, and a methodology to construct efficient document processing by using the dictionary and existing data parallel primitives on GPUs. We then discuss on its applicability to a realistic application. In particular, we focus on the high performance calculation of Okapi's BM25 term weighting by using these primitives on GPUs as an example, rather than a simple string match computation. In document processing, many string comparisons need to be performed, for example, exact matches of two words and sorting a set of words. However, comparing strings are very costly calculation on GPUs, because of their variable size. The dictionary is very useful when handling many strings in document processing, because it efficiently converts each words in a document into the corresponding integer ID so that costly comparisons of strings are replaced to low cost comparisons of integers on GPUs.

When Web pages are processed, the size of vocabulary becomes very large because they contain numerous proper nouns such as names, URLs, etc. The dictionary size, however, must keep small even for large vocabulary due to the memory size limitation of GPUs. Therefore, we improved a compressed dictionary algorithm with a succinct data structure [7] for implementing a GPU-accelerated dictionary, so that a large vocabulary can be handled even with the limited memory size of GPUs.

We also conducted experiments to reveal the power of our dictionary and the suitability of its combination with other existing data parallel primitives for calculating BM25 term weights as an example of document processing.

The rest of the paper is organized as follows. Section 2 refers to the background of this study including parallel primitives, and Section 3 describes an implementation of the BM25 term weighting with our dictionary on GPUs. Experimental results are discussed in Section 4, followed by concluding remarks in Section 5.

[2] `http://hadoop.apache.org/`

2 Background

2.1 Notations

Let Σ be a finite set of alphabets. We denote the text of length n by $T = c_1c_2c_3 \dots c_n$, $c_i \in \Sigma$. For the English text, we have $\Sigma = \{a, b, \dots, z\}$, $|\Sigma| = 26$. When T is written in a natural language, it is assumed that almost T is composed of a finite vocabulary V_k where $k = |V_k|$. Then T is expressed as $T = w_1w_2w_3 \dots w_m$, $w_i \in V_k$. Let d be the ID of a document and t be the ID of a term.

2.2 Term weighting

The weight of term t in document d on Okapi's BM25 term weighting scheme [15] is defined by Equation (1).

$$w_{td} = \log \frac{N}{df_t} \cdot \frac{(k_1 + 1)tf_{td}}{k_1((1 - b) + b \times (L_d/L_{ave})) + tf_{td}} \quad (1)$$

where N is the number of documents, df_t is the number of documents that contain a term t, tf_{td} is the frequency of term t in document d, L_d is the length of document d, and L_{ave} is the average of all L_d. The variables k_1 and b are tuning parameters.

2.3 Term weighting method using MapReduce

Map Each document is assigned to a map worker. The map worker extracts terms from the document assigned, and then, generates key-value pairs, each of which contains term t as a key document ID d as a value. If the same term appears multiple times, pairs are generated multiple times. The generated pairs are stored, and the length of pairs is equal to that of document d.

Reduce A reduce worker processes the group of key-value pairs that have the same term t as a key. From aggregated values, the list of document ID d is obtained. When term t appears multiple times in document d, the same number of d exist in the list. For each d, tf_{td} is obtained by counting the number of d in the list. In addition, df_t is calculated by counting the number of unique d in the list.

We sort the list of d and use the sorted list of d to obtain tf_{td}, df_t, and w_{td}. First, the whole list is scanned to obtain df_t. Then, the list is scanned again to obtain the tf_{td}, while w_{td} is calculated in each time.

2.4 Dictionary

A dictionary is a data structure that handles a set of strings. A set of strings with supplementary information such as ID and description is stored in the dictionary

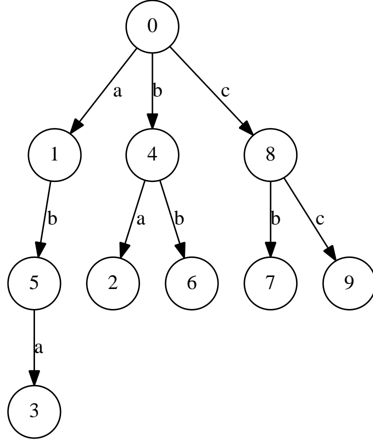

Fig. 1. A example of trie.

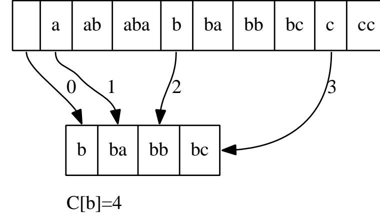

Fig. 2. A example transition from each node by b.

Table 1. A example of STT.

N	0	1	2	3	4	5	6	7	8	9
a	1	-1	-1	-1	2	3	-1	-1	-1	-1
b	4	5	-1	-1	6	-1	-1	-1	7	-1
c	8	-1	-1	-1	-1	-1	-1	-1	9	-1

beforehand. After that, we can use this dictionary to judge whether a certain string is included in the dictionary or not. If the string is included, its supplementary information is retrieved. In the areas managing large-scale vocabulary and/or requiring limited memory capacity, the ideal dictionary algorithm needs to have the features that achieve minimal memory footprint and fast lookup operation. Martínez-Prieto et al. conducted a comparative study of compressed string dictionaries from the viewpoint of both theoretical and practical performances [9].

Dictionary Encoding Let us consider a mapping that converts a word w into the corresponding ID t. Using this mapping, T is expressed as a dictionary-encoded sequence of integers $T = t_1 t_2 t_3 \ldots t_m$.

Instead of expressing T as string, dictionary encoding enables an efficient comparison of words and saving memory space in most situations.

STT Trie is one of the data structures that support retrieving information using variable length keys [4]. It is represented as a tree structure. The edges are labeled with a character. Each node corresponds to a prefix of the key. The

information associated with the key can be retrieved by tracing the edges from the root. The state transition table (STT) stores the next node ID for all nodes and for all characters explicitly.

For example, Figure 1 and Table 1 show the trie and the corresponding STT that has aba, ba, bb, cb, and cc as keys. In this example, -1 denotes that the transition is unavailable.

XBW In this paper, we apply compressed prefix matching with XBW proposed by Hon et al. [7] to convert words on GPUs. Note that we use the algorithm without further compression because of noticeable overhead. For theoretical details, see [7].

The reverse prefix is defined as the concatenation of characters from the node to the root. The ID of each node is assigned by lexicographical order of the associated reverse prefix. Note that the ID of the root node is 0. An example of the assigned IDs appears in Figure 1.

For each character c, the bit-vector B_c is constructed. The i-th element of the bit-vector B_c indicates the node i can transition by a character c if the i-th element is 1. The function $\text{rank}(i, B_c)$ is defined as the number of 1s in $B_c[0, i)$. Additionally, for each character c, the smallest node ID that node's reverse prefix begins c is stored as $C[c]$.

The node transition by c from the node x is obtained by the following procedure.

1. If x-th element of the B_c is 0, there is no valid transition.
2. If there is valid transition, the next node ID is calculated by $\text{rank}(x, B_c) + C[c]$.

Figure 2 shows example transitions from each node by b. The nodes that can transition by b have an arrow labeled with a rank. Some examples are shown below:

- $B_b = 1100100010$
- Transition by b from root node.
  $\text{rank}(0, B_b) + C[b] = 0 + 4 = 4$
- Transition by b from node 8.
  $\text{rank}(8, B_b) + C[b] = 3 + 4 = 7$

bit-vector For the bit-vector supporting rank in $O(1)$ time, RRR [14] is used. The bit-vector is divided into blocks of length t bits. The block is classified according to the number of 1s in the bits. Thus, $\binom{t}{k}$ blocks belong to class k and each block has unique index r. Therefore, the pair (k, r) identifies the block. For decoding from the pair to bits, there are two methods, using a pre-computed table or computing on the fly [12]. Additionally, a superblock groups some blocks and stores rank and pointer at beginning bit of the superblock.

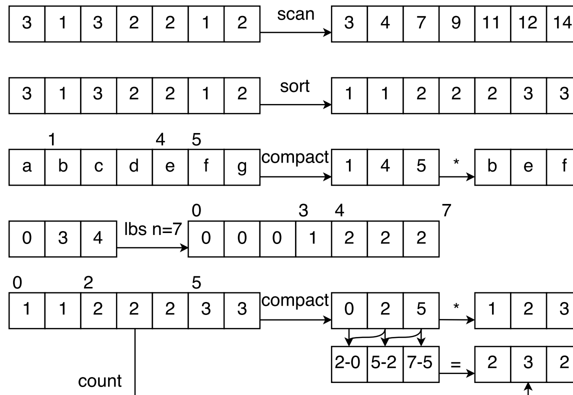

Fig. 3. Example behavior of parallel primitives.

2.5 GPU Architecture

In this paper, we used a GTX 970, which is a GPU with Maxwell architecture, for evaluation and the program is written using CUDA. For further details of the GPU architecture, see the official documents [13].

All threads within a warp perform one common instruction in a lockstep. When the threads within a warp follow different execution paths, each path is executed one by one while the threads following the others discard the results. This is called warp divergence which is one of the causes of low performance.

Global memory is basically accessed via 128-byte memory transaction. When all threads within a warp access continuous 128-byte memory region, these memory accesses are coalesced. This coalesced access is important to achieve maximum performance of memory access.

2.6 Data parallel primitives

Data parallel primitives on GPUs are the fundamental algorithms that are used as building blocks for constructing programs. There has been proposed some efficient algorithms such as sort [11], scan [6], and merge [5]. In addition, there are several implementations and libraries available. In this paper, we use ModernGPU [1] library.

In this section, we describe data parallel primitives composing our proposed term weighting method. Figure 3 shows an example behavior of each primitive.

Scan Scan, or prefix sum, takes an associative binary operator $\oplus$ and an array$[a_0, a_1, a_2, ..., a_{n-1}]$ as input, and then generates an array $[a_0, (a_0 \oplus a_1), ..., (a_0 \oplus a_1 \oplus ... \oplus a_{n-1})]$. If the i-th output includes the i-th input, it is called inclusive scan. Instead, if the i-th output does not include the i-th input, it is called exclusive scan.

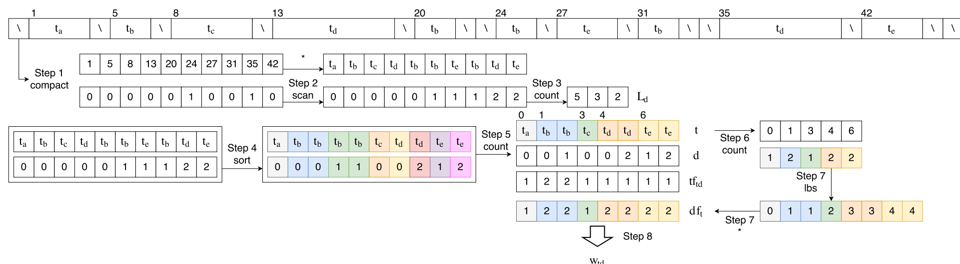

Fig. 4. Overview of proposed term weighting method using data parallel primitives.

Sort It is considered that radix sort [11] is the fastest sorting algorithm on GPUs. However, there is a limitation on the property of keys to perform the radix sort. For example, the radix sort cannot perform with variable length keys such as strings. In comparison sorting algorithms, merge sort based on efficient merge [5] is competitive with the radix sort. Still sorting variable length keys with the merge sort yields suboptimal performance on GPUs. We used the merge sort [1] as an implementation of sort primitive.

Compact Compact, or filter, extracts the elements that satisfy a predicate from input arrays. It consists of three steps. First, it marks the elements that satisfy a condition. Then, it calculates their indices. Finally, it generates output arrays using the indices.

Load balancing search Load balancing search takes an array of length k in which elements represent the positions of boundary of an array of length n that desired to be generated. Then, it generates an array of length n in which elements represent the indices to denote what part of an array of length k. It can be regarded as a special case of a vectorized sorted search [1].

Count Count is the operation that counts the numbers of unique keys in an input array. It is not so much primitive as composition of primitives. However, we describe it here because we employ it several times in our proposed method. First, it sorts an input array if needed. Then, it extracts the indices of boundaries using compact. The predicate for compact is whether the key is not equal to the preceding key. After that, it calculates the number of keys by subtracting the index from the succeeding index for each index. The corresponding key with the number can be retrieved by using the indices of boundaries.

3 Term weighting method using data parallel primitives

Assumption We calculate all w_{td} corresponding to the pair $\langle t, d \rangle$ when document d contains term t. A set of documents is represented as a sequence of

terms separated by a space. The boundary between documents is represented as a blank line.

Step 1: Extract word In order to calculate term weights, extracting terms from a document is needed. We assume that documents are preprocessed by the method as described in above, hence the following conditions hold:

1. The character is the first character of a word, if the character is an alphabet and the preceding is a blank character.
2. The character is the first character of a document, if the character is an alphabet and the preceding is a new line character.

An array of indices denoting the positions of the first character of each term can be obtained by compact using predicate 1. Additionally, an array of Boolean denoting whether the corresponding term is the first term in the document. It is used to generate a document index in the next step.

Step 2: Assign document ID We have no information on which term belongs which document. Using the inclusive scan of sum, we can obtain an array of monotonic values that the terms within the same document have the same value. We use these values as document ID d.

Step 3: Calculate document length For calculating BM25, the number of terms in each document is needed. It can be obtained by count primitive to an array of document IDs.

Step 4: Sort In this step, the positions of terms and the document IDs are sorted by the position of the term as the key and the document ID as the value. The position of the term is compared with the corresponding term by string comparison. We use stable sort so that document IDs can preserve monotonic order within the same terms.

Step 5: Calculate tf_{td} The number of elements that have the same term and document ID is $tf_t d$, which is obtained by count primitive.

Step 6: Calculate df_t In the previous step, the array of unique pairs of term t and document ID d is obtained. The number of the elements that have the same term is df_t in this array. In the same way, df_t is obtained by count primitive.

Step 7: Assign df_t In order to calculate w_{td}, we need both tf_{td} and df_t. If one thread handles one df_t, it is easy to obtain tf_{td} from df_t by calculating offsets and iterating df_t times. In this way, however, some threads may calculate large amounts of BM25 weights while others do a few. Thus, a significant load imbalance occurs and the warp divergence hurts performance.

To evade this and balance the load, we create an expanded array of df_t of which element has a one-to-one correspondence to tf_{td}. This array is generated by load balancing search.

Step 8: Calculate BM25 Since the values necessary for the calculation are obtained by the above steps, BM25 weights are calculated with Equation (1).

3.1 Implementation of the dictionary

We implemented the dictionary with XBW as described in section 2.4. First, the block size is set to 15. Class k ranges $[0, \ldots, 15]$. Thus, class k is represented as a 4-bit integer. The superblock groups 16 blocks. A rank, a pointer, and 16 classes are stored in a structure. The size of the structure is 16 bytes. This structure is handled as built-in vector type uint4 so that it can be fetched in a single memory transaction.

We use the table for decoding a pair (k, r). The number of entries is 2^{15} and the size of each entry shares 2 bytes. Thus, the size of the table is 64 KB in total. The constant $\lceil \log \binom{t}{k} \rceil$ and offsets for accessing the table are cached in the shared memory.

Modification to term weighting method It is assumed that the dictionary is constructed in advance and resided in CPU side memory. The dictionary is transferred to GPU side memory along with input documents. We add the converting step before the sort step. This step converts the positions of terms into IDs using the dictionary. After this step, comparisons of the terms are replaced by those of IDs.

If there is a term that is not listed in the dictionary, this term is treated as an unknown term, and the same ID is assigned for all unknown terms. Note that the existence of unknown terms never affects the term weight of other terms, although the term weight of unknown terms becomes the same value.

4 Evaluation

In this section, we compare the following four methods and evaluate the performance:

MapReduce Phoenix++ (MRP)
: This is a MapReduce based method on multi-core processors using Phoenix++ [17]. We conduct experiments using 8 threads, which is the same as the number of logical cores.

MapReduce Mars revised (MRM)
: This is a MapReduce based method on GPUs using Mars [3]. We replaced the original sort algorithm to the merge-based sort primitive for fair comparison. The overhead of sorting is a bottleneck in the shuffle step.

Table 2. Statistics of datasets.

	n_w	n_{dl}	n_{tf}	n_{df}
100MB	17,881,505	24,411	11,029,756	447,663
200MB	35,767,202	48,725	22,074,419	687,255
300MB	53,651,412	73,163	33,119,396	882,477
400MB	71,532,437	97,683	44,123,823	1,051,534
500MB	89,409,102	122,178	55,165,752	1,203,716

Parallel Primitives (PP)
This is a data parallel primitives based method on GPUs without a dictionary.

Parallel Primitives with Dictionary (PPD)
This is a data parallel primitives based method on GPUs with the dictionary to convert terms into IDs.

We measured the execution times from when all inputs reside in CPU side memory until calculated results are stored back to CPU side memory. Hence, data transfer time between the CPU and the GPU is included if a GPU is used.

4.1 Setup

We collect frequencies of terms from around 50 million English pages in TREC ClueWeb09 Category B[3]. The terms are case insensitive and consist of only alphabets. Thus, $\Sigma = \{a, b, \ldots, z\}$ and $|\Sigma| = 26$. We used top-k terms when the vocabulary size is k.

We used artificially created documents for evaluation. The words in the document are randomly selected by using discrete distribution based on the above frequencies of terms. The lengths of the documents are randomly determined by using lognormal distribution of which parameters are $\mu = 6.0, \sigma = 1.1$. Table 2 shows the statistics of the datasets generated. Here, n_w is the total number of words, n_{dl} is the number of documents, n_{tf} is that of tf_{td}, and n_{df} is that of df_t.

The experiments were conducted on a PC with an Intel Core i7-6700K, 16GB of DDR4 memory, and an NVIDIA GeForce GTX 970, running on Ubuntu 14.04 and CUDA 7.5.

4.2 Results

Figure 5 compares the performances of four methods. Both the proposed methods based on parallel primitives on GPUs perform better than multi-core CPUs. In particular, PPD, which uses the dictionary, outperforms MRP by a factor of 5.0-5.1 in terms of runtime. The effect of using the dictionary is observed by comparing PPD and PP. PPD achieves reduction of overall runtime by a factor of 3.8-4.5. By contrast, MRM, which uses Mars on GPUs, fails to achieve

[3] `http://trec.nist.gov/`

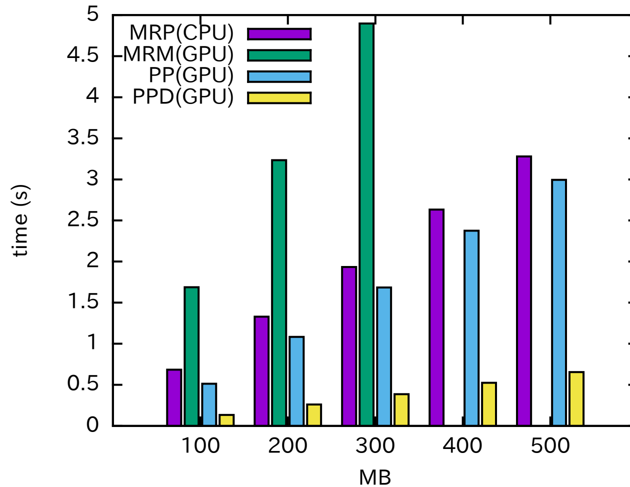

Fig. 5. Execution time of methods.

performance gain against MRP. Furthermore, the memory requirement is larger than the other methods; therefore, MRM cannot run with more than 400MB of datasets.

Figure 6 shows breakdowns of the execution times of MapReduce-based MRP and MRM running on CPUs and GPUs, respectively. The shuffle step aggregates key-value pairs. This step is implemented by sort in Mars and hash tables in Phoenix++, respectively. Thus, the time for shuffle is included within map and reduce steps in Phoenix++. The merge step gathers the results of each reduce worker into one list of key-value pairs. Although the reduce step of MRMdoes not contain any operation handling strings, this step occupies a large portion of execution time. This is due to load imbalance within threads which detoriorates performance.

Figure 7 shows breakdowns of the execution times of data parallel primitives based methods. In PP method, the steps handling variable length strings such as sort, the calculations of tf and df occupy a large portion of execution time. In particular, sorting dominates overall execution time. By contrast, PPD method which employs the dictionary reduces the overheads of costly sorting and comparing of strings by converting terms into IDs beforehand. Although the converting step is added, the cost of converting and sorting by IDs is small enough compared to sorting by strings.

PPD achieves a significant speedup by using the dictionary. In compensation for using the dictionary, there are two disadvantages. First, an additional cost for converting is involved. Second, it cannot calculate weights of the terms that are not listed in the dictionary. Moreover, the execution time without data transfer occupies about 45%. However, it is important to use a fast and compact

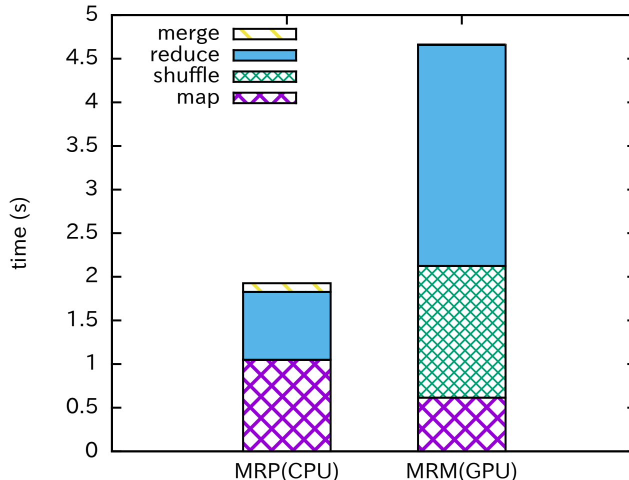

Fig. 6. Breakdown of excexution time of MapReduce based methods.

dictionary for the applications which need to handle large vocabulary such as information retrieval.

4.3 Evaluation of Dictionary

In this section, we compare the proposed implementation of the dictionary with a naive one based on a trie using a state transition table (STT). STT is the fastest method if memory access cost is constant. In reality, the memory subsystem of GPU is complicated and varies depending on architecture [19, 10]. It must be noted that these algorithms for dictionary are inherently inevitable to random access. Hence, their performances are heavily influenced by memory hierarchy and cache replacement algorithm of hardware.

Setup We used the following two types of text:

T_{freq}
This is an artificial text using discrete distribution as same as the section 4.1.

T_{unif}
This is an artificial text using uniform distribution.

There are two parameters m and k for texts. The parameter m is the number of words in a text and k is that of words in vocabulary.

The thread i converts the $i+kN$-th word, where N is the number of threads and k is natural number. We measured execution times in changing the number of threads and that of blocks. The minimal execution times are shown in the following results.

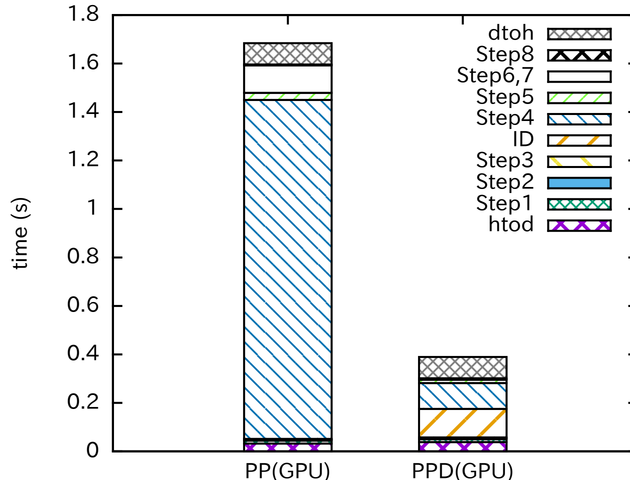

Fig. 7. Breakdown of execution time of data parallel primitive based mehods. dtoh: data transfer time from GPU to CPU. htod: data transfer time from CPU to GPU. each step corresponding to steps of section 3.

Results The memory footprint of the dictionary in our setup is shown in Figure 8. For large vocabulary, the XBW requires about 40x less memory footprint than STT.

Seeing Figure 9, the execution time is proportional to the size of input in all cases. The performance of STT is heavily depending on the vocabulary size. By contrast, the proposed method slightly increases the execution time.

Figure 11 indicates the execution time per character, while total memory usage is shown in Figure 10. The input text is T_{freq}, $m = 100$M, and imitates an actual text by using term frequencies. The execution time of STT increases along with vocabulary size. For large vocabulary, the proposed method performs better than STT in this scenario.

When the input text is T_{unif}, the behavior is shown in Figure 12. Both methods increase execution times along with vocabulary size. When using uniform distribution, the characteristic of text changes according to vocabulary size. The variance of length of words becomes larger because a rare word is often longer. The mean and variance of length of words in a text are shown in Table 3. The variance of length of words causes warp divergence. It is considered that the proposed method suffers the more effect of warp divergence, since it requires more operations than STT.

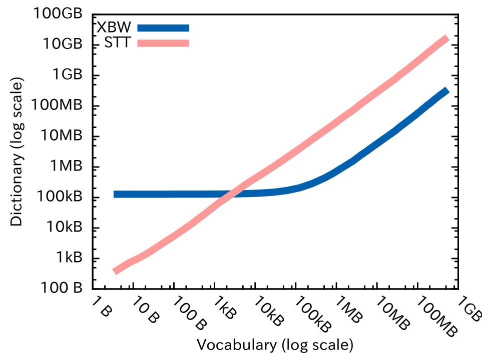

Fig. 8. Memory footprint of the dictionary.

Table 3. Mean and variance of length of word in a text.

Vocabulary size k		10K	100K	1M	10M
T_{freq}	Mean	4.55	4.78	4.84	4.86
	Variance	6.34	6.91	7.09	7.24
T_{unif}	Mean	6.66	7.12	7.84	9.44
	Variance	6.42	7.04	8.32	17.2

5 Conclusion

In this paper, we proposed an implementation of the efficient compressed dictionary on GPUs and a methodology to construct a very fast BM25 term weighting algorithm with a dictionary and existing data parallel primitives on GPUs as an example of major document processing. The experimental results showed that the proposed dictionary requires 40x less memory footprint than STT, and that GPUs are able to achieve obvious speedup against CPUs on processing of documents by properly composing the dictionary and data parallel primitives.

Although the performance for converting words into integer IDs depends on characteristics of text and vocabulary, the proposed dictionary is competitive with STT in our setup when the size of vocabulary is large. By using the dictionary to avoid string comparison, GPUs with dictionary performs up to 5.1x faster than multi-core CPUs and up to 4.1x faster than GPUs without the dictionary. Use of the dictionary has proven to be very effective in efficiently performing the processing of documents, thereby the importance of a compact and fast dictionary has been clarified.

Further studies are needed in order to conclude applicability of the data parallel primitives and the dictionary to other document processing. In terms of the dictionary, the simplest parallelization method is used for converting words into IDs. Since the processing time of each word depends on the length of a word, it is considered that some sophisticated scheduling method rearranging

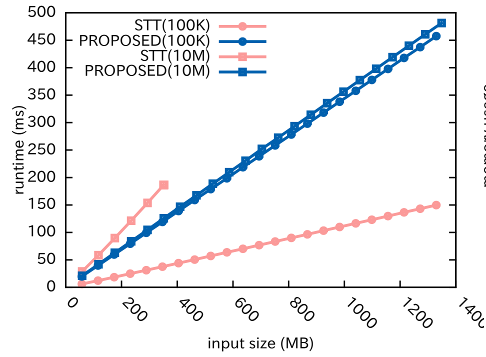

Fig. 9. Runtime in milliseconds of converting words where T_{freq}, k = 100K, 10M, and varying the length of text m.

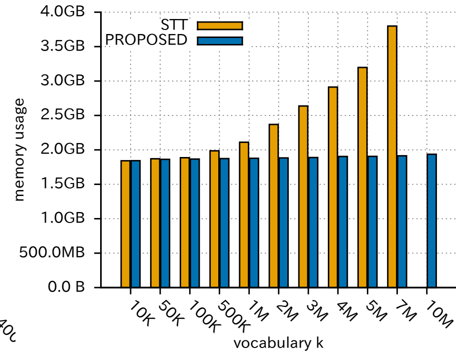

Fig. 10. Total memory usage on GPUs of converting words where T_{freq}, m = 100M, and varying the number of words in vocabulary.

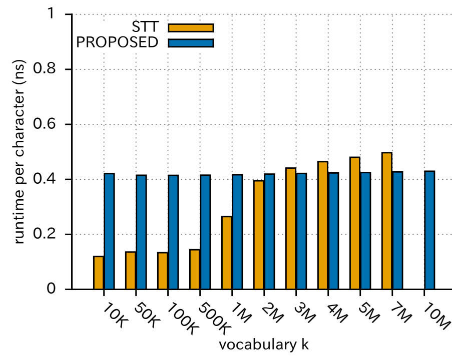

Fig. 11. Runtime per character in nanoseconds of converting words where T_{freq}, m = 100M, and varying the number of words in vocabulary.

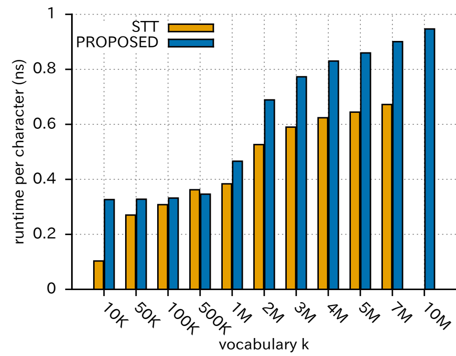

Fig. 12. Runtime per character in nanoseconds of converting words where T_{unif}, m = 100M, and varying the number of words in vocabulary.

words can balance the load among threads and achieve more improvement in its performance.

References

1. Baxter, S.: moderngpu 2.0, https://github.com/moderngpu/moderngpu/
2. Dean, J., Ghemawat, S.: MapReduce: Simplified data processing on large clusters. In: Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6. pp. 10–10. OSDI'04, USENIX Association (2004)
3. Fang, W., He, B., Luo, Q., Govindaraju, N.K.: Mars: Accelerating mapreduce with graphics processors. IEEE Transactions on Parallel and Distributed Systems 22(4), 608–620 (2011)
4. Fredkin, E.: Trie Memory. Communications of the ACM 3(9), 490–499 (Sept 1960)
5. Green, O., McColl, R., Bader, D.A.: GPU Merge Path: A GPU merging algorithm. In: Proceedings of the 26th ACM International Conference on Supercomputing. pp. 331–340. ICS '12, ACM (2012)
6. Harris, M., Sengupta, S., Owens, J.D.: Parallel prefix sum (scan) with cuda. In: Nguyen, H. (ed.) GPU Gems 3. Addison Wesley (Aug 2007)
7. Hon, W.K., Ku, T.H., Shah, R., Thankachan, S.V., Vitter, J.S.: Faster compressed dictionary matching. Theoretical Computer Science 475, 113 – 119 (2013)
8. Lin, J., Dyer, C.: Data-Intensive Text Processing with MapReduce. Morgan and Claypool Publishers (2010)
9. Martínez-Prieto, M.A., Brisaboa, N., Cnovas, R., Claude, F., Navarro, G.: Practical compressed string dictionaries. Information Systems 56, 73 – 108 (2016)
10. Mei, X., Chu, X.: Dissecting GPU memory hierarchy through microbenchmarking. IEEE Transactions on Parallel and Distributed Systems 28(1), 72–86 (2017)
11. Merrill, D., Grimshaw, A.: High performance and scalable radix sorting: a case study of implementing dynamic parallelism for gpu computing. Parallel Processing Letters 21(02), 245–272 (2011)
12. Navarro, G., Providel, E.: Fast, small, simple rank/select on bitmaps. In: Proceedings of the 11th International Conference on Experimental Algorithms. pp. 295–306. SEA'12, Springer-Verlag (2012)
13. NVIDIA: CUDA Toolkit Documentation. http://docs.nvidia.com/cuda/
14. Raman, R., Raman, V., Satti, S.R.: Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets. ACM Transactions on Algorithms 3(4) (Nov 2007), article No. 43
15. Robertson, S.E., Walker, S., Jones, S., Hancock-Beaulieu, M., Gatford, M.: Okapi at TREC-3. In: Proceedings of The 3rd Text REtrieval Conference. pp. 109–126 (1994)
16. Sitaridi, E.A., Ross, K.A.: GPU-accelerated string matching for database applications. The VLDB Journal 25(5), 719–740 (2016)
17. Talbot, J., Yoo, R.M., Kozyrakis, C.: Phoenix++: Modular mapreduce for shared-memory systems. In: Proceedings of the Second International Workshop on MapReduce and Its Applications. pp. 9–16. MapReduce '11, ACM (2011)
18. Wang, Y., Davidson, A., Pan, Y., Wu, Y., Riffel, A., Owens, J.D.: Gunrock: A high-performance graph processing library on the gpu. In: Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. pp. 11:1–11:12. PPoPP '16, ACM (2016)

19. Wong, H., Papadopoulou, M., Sadooghi-Alvandi, M., Moshovos, A.: Demystifying GPU microarchitecture through microbenchmarking. In: IEEE International Symposium on Performance Analysis of Systems and Software. pp. 235–246. ISPASS 2010 (2010)