Fast Burrows Wheeler Compression using All-Cores

Aditya Deshpande[1,2]
[1]*University of Illinois at Urbana Champaign*
Champaign, USA
ardeshp2@illinois.edu

P J Narayanan[2]
[2]*International Institute of Information Technology Hyderabad*
Hyderabad, India
pjn@iiit.ac.in

***Abstract*—In this paper, we present an all-core implementation of Burrows Wheeler Compression algorithm that exploits all computing resources on a system. Our focus is to provide significant benefit to everyday users on common end-to-end applications by exploiting the parallelism of multiple CPU cores and additional accelerators, viz. many-core GPU, on their machines. The all-core framework is suitable for problems that process large files or buffers in blocks. We consider a system to be made up of compute stations and use a work-queue to dynamically divide the tasks among them. Each compute station uses an implementation that optimally exploits its architecture. We develop a fast GPU BWC algorithm by extending the state-of-the-art GPU string sort to efficiently perform BWT step of BWC. Our hybrid BWC with GPU acceleration achieves a $2.9\times$ speedup over best CPU implementation. Our all-core framework allows concurrent processing of blocks by both GPU and all available CPU cores. We achieve a $3.06\times$ speedup by using all CPU cores and a $4.87\times$ speedup when we additionally use an accelerator i.e. GPU. Our approach will scale to the number and different types of computing resources or accelerators found on a system.**

***Keywords*-Burrows Wheeler Transform; Bzip2; Heterogeneous systems;**

I. Introduction

Computing platforms are parallel and heterogeneous today. CPUs with multiple identical cores are found on these platforms. In addition to CPU, they have an accelerator in form of GPU, with dozens to hundreds of simpler cores. In future, other accelerators may also be used. Obtaining higher performance on common *end-to-end* user applications on such parallel platforms has been a challenge, however. Tuned implementations of several data-parallel algorithms on graphs or matrices on multicore CPUs, manycore accelerators like the GPU and CellBE, and their combinations have been developed recently. But such operations constitute only a portion of most end-to-end applications. Pipelining different tasks effectively and obtaining even moderate performance gains for the entire end-to-end application is still a practical challenge.

In this paper, we present an *all-core* implementation of an end-to-end lossless data compression application, specifically, the *Burrows Wheeler Compression* (BWC) algorithm. BWC is a popular, open compression scheme built on Burrows Wheeler Transform (BWT) [3]. It is used widely to compress regular files, system software, gene sequences, etc. BWC typically gives 30% smaller compressed files compared to LZW based schemes [1]. Compression schemes are among the hardest to parallelize on many-core architectures like the GPU due to their irregularity. The approach developed by us scales to exploit all cores – CPU, GPU and others – present on a given computer. In contrast, only block-parallel BWC approaches on multi-core CPUs have resulted in speedup previously [10]. A recent GPU BWC effort performed slower than a single core CPU [15]. The main contributions of our work are given below.

1) We develop a fast BWT algorithm on the GPU that is built on radix sort.[1]
2) BWT and its inverse are often used in pairs. This allows us a way to speedup BWT by modifying the strings to reduce high tie-lengths. We introduce a reversible string perturbation step to increase speed at a slight reduction in compression ratio (Section III-B).
3) We partition the tasks between CPU core and the GPU, with naturally serial tasks performed on the CPU. The controlling CPU thread performs its tasks totally overlapped with the GPU computations, resulting in a fast hybrid BWC algorithm (Section III-C).
4) We develop an *all-core computation framework* to exploit heterogeneous compute cores on a system. Using this all-core framework we extend hybrid BWC (that used single controlling CPU thread and GPU) to all-core BWC. On a Intel Core i7, with only multi-core CPU, our all-core BWC achieves a $3.06\times$ speedup. This improves to $4.87\times$ when we use GPU acceleration by adding a Nvidia GTX 580 (Table II). On a low-end Intel Core2Duo, our all-core BWC achieves a $1.22\times$ speedup. This improves to $1.67\times$ speedup on using Nvidia GTX 280 (Table III). Our code is available for public use.[2]

Our all-core BWC using GPU acceleration, gives better runtime (than multi-core CPU BWC), while balancing the load between the GPU and the CPUs. This is significant since a previous effort failed to achieve speedup using GPUs [15]. We believe that the techniques and lessons from this work will motivate future work in designing all-core

[1]http://web.engr.illinois.edu/~ardeshp2/stringSort/
[2]http://cvit.iiit.ac.in/resources/bzip2GPU/bzip2Cvit.tar.gz

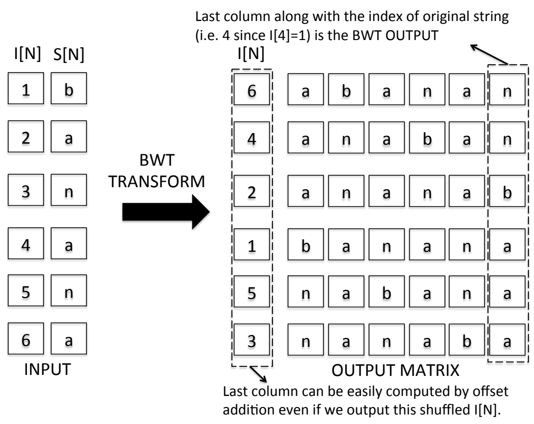

Last column along with the index of original string
(i.e. 4 since I[4]=1) is the BWT OUTPUT
I[N] S[N]
I[N]
BWT
TRANSFORM
INPUT
OUTPUT MATRIX
Last column can be easily computed by offset
addition even if we output this shuffled I[N].

Figure 1: Illustration of the Burrows Wheeler Transform on the input string *banana*.

implementations for other end-to-end applications.

II. Background and Related Work

A. Burrows Wheeler Transform

Burrows and Wheeler, developed BWT [3] with following steps: (i) Start with input string $S[1...N]$ and associated index array $I[1...N]$, initialized to $1...N$, denoting the starting position of each (cyclically shifted) suffix in input string. (ii) Sort all suffixes in lexicographic order along with corresponding indices I. Final suffix array is a permutation of initial index array $I[1...N]$. (iii) Compute last elements of sorted suffixes and output it along with the index of original string in the sorted output.

Figure 1 shows the application of BWT on an input string banana. It operates on all cyclic shifts or suffixes of input string $S[1...6] = \texttt{banana}$. It generates the output matrix containing a sorted list of all cyclically shifted strings. The answer is last column of output matrix, i.e. nnbaaa, appended with number 4, since the original string occurs at 4^{th} position in output. In general, suffix sorting step of BWT is compute intensive because it involves sorting $O(N)$ suffix strings each of length $O(N)$ and these strings also have a high match length (i.e. they share long common prefix). High match length is a characteristic of compression datasets, because compression schemes are typically used when data has redundancy. This redundancy results from long repeating substrings, which causes high match length for suffix strings. In practice, cyclically shifted suffix strings match to lengths as high as 10^3-10^5 characters within a 9M char. block. This shows the compute intensive nature of BWT.

B. Sequential Burrows Wheeler Compression

BWT was used to devise a lossless BW compression scheme by Burrows and Wheeler [3]. For suffix sort, they performed a radix sort on first two characters (c_1c_2) of all suffixes followed by a modified Quicksort [2] in subsequent iterations, along with a special mechanism to handle inputs with long repeated runs. Their BWC was slow. Seward proposed a method that involved sorting only a few buckets after 2-character radix sort and cleverly synthesized sorted order for suffixes in other buckets. This resulted in an efficient and popular Bzip2 compressor. Incorporating Sadakane's algorithm [16] improved performance on worst case inputs. Synthesizing sorted order of new suffixes from previously sorted ones avoids expensive matching and results in good performance. Such fine-grained synthesis methods require synchronization and are difficult to do on GPU. Our GPU implementation uses a coarse synthesis technique developed by Kärkkäinen and Sanders [11].

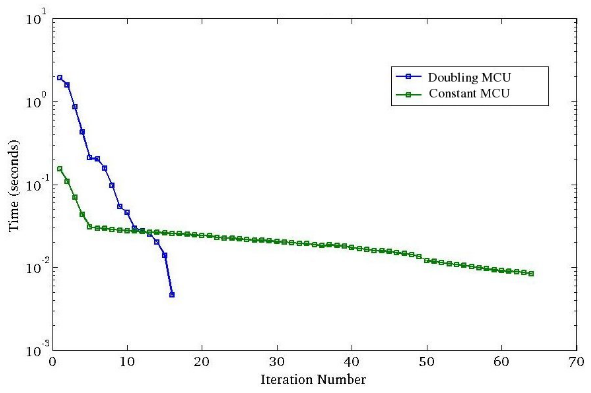
C. Parallel Burrows Wheeler Compression

Gilchrist and Cuhadar [10] exploited the inter-block parallelism for linear speedup with multiple CPU cores for BWC. They focussed on naive parallelization without modifying the basic algorithm or making it scale to larger input size.

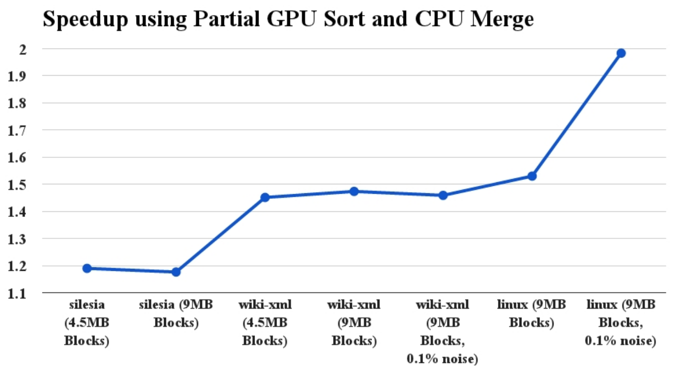
Previous GPU BWC. The only prior GPU implementation of BWT by Patel et al. [15] repeatedly sorts strings using a variable length key comparison based sort [5]. Their implementation was about $2.78\times$ slower (with BWT step dominating the runtime) than the CPU version due to the inherent difficulty of parallelizing BWC. Another attempt at building parallel BWC was abandoned due to very poor performance [4].

Edwards and Vishkin BWC. In parallel with our work, Edwards and Vishkin [8] developed an intra-block parallel BWC algorithm and compared it with CPU BWC [9]. Their algorithm is work optimal with an $O(\log N)$ time on architectures with fine-grained parallelism. They demonstrated it on their Explicit Multi-Threading (XMT) architecture but not on CPUs/GPUs. In contrast, we demonstrate better performance on multi-core CPUs and GPUs. They report a speedup of 1.8 to $2.8\times$ on XMT-64 ($\sim$ 64 cores) platform and 12 to $25\times$ on a simulated XMT-1024 ($\sim$ 1024 cores) platform. They use files from *Large Corpus*[3], which are small in size ($< 4.5MB$'s) and have a lower maximum sorting depth (< 2000). We demonstrate high performance on larger datasets with higher sorting depth (10^3 to 10^5) like Enwik8 (96MB) [14], Linux-2.6.11.tar (199MB) and *Silesia Corpus*[4] (203MB, which supersedes *Large Corpus*) [6]. It would be worthwhile to compare the runtime of our algorithm with Edwards and Vishkin [8] on same platforms and same test datasets in future.

[3]http://corpus.canterbury.ac.nz/descriptions/
[4]http://sun.aei.polsl.pl/ sdeor/index.php?page=silesia



(a) Sorting with Doubling MCU is expensive but requires very few iterations, while Constant MCU sorting is faster but takes very many iterations. Thus, we use constant MCU for first few iterations and then resort to doubling MCU.



(b) Speedup of 1.2 to 2$\times$ is obtained by using our partial sort and merge strategy as compared to performing full sorting on different datasets.

Figure 2: Study of optimizations to string sort for performing BWT.

BWT in sequence alignment. Another line of related work on parallel BWC is found in the literature of BWT constructed for sequence alignment in bio-informatics applications. This work deals only with the BWT step and not the entire BWC. Nvidia distributes a high performance library *nvbio* for this purpose[5]. The BWT/suffix sort algorithm in this library is inspired from Liu et al. [13]. Note that, this algorithm uses radix sort at its core and finds splitters to reduce the radix sort problem size. They make naive repeated use of radix sort for string sorting step. While, in our work, the core radix sort based string sort has been augmented with many optimizations [7]. These optimizations have been shown to improve runtime when compared to naive repeated use of radix sort. In addition, we have also developed many BWC specific techniques (Section III-A).

III. CPU and GPU Hybrid BWC Algorithm

Burrows Wheeler Transform (BWT) is the most crucial step of BWC and one that occupies a significant chunk of its runtime. Computing BWT is equivalent to suffix sorting of all suffix strings (shown in Figure 1) of the input string. We can treat each suffix as a separate string and perform a string sort. A radix sort based approach is used to perform string/suffix sorting. Using radix sort, suffix sorting can be done in single sort step with $O(N)$ length keys, which is impractical. Alternatively, if each of the suffix strings are sorted on its first few (8-bit) characters, we can get the first column of sorted suffixes (Figure 1), with repeated occurrences clubbed together (in a bucket). Their order can be corrected by sorting on the next characters of each suffix, within its bucket. Since sorting now only needs to be done within a bucket, all buckets are independent of each other and can be processed in parallel. Note that, as opposed to merge sort [15], [5], in our radix sort based method all characters (from left to right) of the string/suffix are accessed only once. By performing minimal number of global memory accesses a radix sort based string sorting method runs much faster than merge sort based methods. More details of string sorting technique can be found in [7].

A. *Modified String Sort for BWT*

The string sorting approach works well only when the input has ties up to a few 100 characters [7]. As we saw earlier, BWT has relatively much higher number of ties (10^3 to 10^5 characters). We develop some BWT specific optimizations to address this –

Doubling MCU Length. The match length between suffix strings (i.e. length of longest common prefix of all suffix strings) determine the maximum number of fixed-length sorts required. This is referred to as the sorting depth. Large sorting depths result from long substrings repeating many times, which degrades GPU BWT performance. To address this, we double the MCU length after a few steps. This reduces the number of sort steps as longer substrings are being compared in each successive iteration. As shown in Figure 2a, if we use doubling right from the start, initial sort steps are very costly as compared to constant sized (k character) MCUs. But, the total number of sort steps with doubling are very less. To obtain the best results, we use the faster constant size MCU for the first 16 iterations and then double the MCU length. For example, one input dataset has a sorting depth of 960 characters. We double the MCU length after 16 sort steps. So, we perform $16 * 4 = 64$ character comparisons with 4-byte MCUs. Now, only 7 more sort steps are required to cover the remaining 896 ($8 + 16 + 32 + ... + 512 > 896$) characters. Doubling MCU length after 16 sort steps gives us a speedup of $1.8\times$ in

[5]http://nvlabs.github.io/nvbio/sufsort_page.html

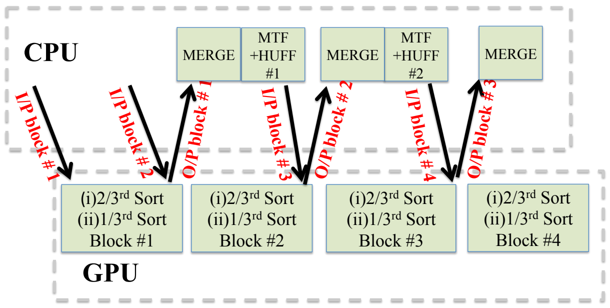

CPU
MERGE
MTF +HUFF #1
MERGE
MTF +HUFF #2
MERGE
I/P block # 1
I/P block # 2
O/P block # 1
I/P block # 3
O/P block # 2
I/P block # 4
O/P block # 3
(i)2/3rd Sort (ii)1/3rd Sort Block #1
(i)2/3rd Sort (ii)1/3rd Sort Block #2
(i)2/3rd Sort (ii)1/3rd Sort Block #3
(i)2/3rd Sort (ii)1/3rd Sort Block #4
GPU

Figure 3: An illustration of the CPU+GPU hybrid BWC pipeline. The merge, MTF and Huffman steps are done on the CPU in a fully overlapped manner with the partial sorts on the GPU of succeeding block.

this particular case.

Partial GPU sorts and CPU merge. In suffix sort, it is not necessary to sort all strings, we can sort only a subset of the original strings and synthesize the sorted order for rest. This synthesis is possible because the input strings are cyclically shifted. Suppose, we sort all strings at indices $i \pmod 3) \neq 0$ (denoted by set $I_{1,2}$), we can generate the sorted order for all suffixes at $i \pmod 3) = 0$ (denoted by set I_0). This is done as follows: we compare the first character of the two I_0 strings, if it is unequal we obtain the sorted order. If it is equal, the strings beginning from next characters of both suffixes correspond to suffixes in $I_{1,2}$, for which we already know the sorted order. Also, it is easy to merge I_0 and $I_{1,2}$, since in at most two comparisons of next characters, we hit two suffixes that belong to $I_{1,2}$. Similar sort and merge approaches have been in practice in the CPU suffix sorting literature [11], [12], [17]. We perform the two sorting steps on GPU and move the merge step to CPU as shown in Figure 3. This allows us to utilize the idle CPU cycles while GPU is performing sort on subsequent block. Figure 2b shows that we obtain a speedup of 1.2 to 2× after using this optimization.

B. String Perturbation

Large sorting depth comes from repeated long substrings. Runtime can reduce greatly if we can reduce long ties by perturbing the string. This works for BWT based compression schemes because a known perturbation can be undone after decompression. Different perturbations were tried by us. Inserting a random character at fixed positions in the input string worked the best, as it forcefully breaks long ties. It should be noted that the BWT for this modified input string is not the same as BWT of the original input string. Since BWT and inverse BWT (IBWT) are used in pairs, random characters that occur at fixed positions can be removed to restore the original string after IBWT.

Algorithm 1 Hybrid BWC Algorithm

```
1:  Input: File $\mathbf{F}$
2:  $[\mathbf{B_1}, \mathbf{B_2}, \cdots, \mathbf{B_n}]$ = split-blocks($\mathbf{F}$, $\mathbf{N}$)
    // split file into size N blocks
3:  $[\mathbf{B_1}, \cdots, \mathbf{B_n}] \leftarrow$ perturb($[\mathbf{B_1}, \cdots, \mathbf{B_n}]$, interval)
    // random char is added after interval
4:  for $\mathbf{B_p}$ in $[\mathbf{B_1}, \mathbf{B_2}, \cdots, \mathbf{B_n}]$
5:      $\mathbf{I} = [\mathbf{1} \cdots \mathbf{N}]$
    // Index Array, denotes starting position of each string
6:      $\mathbf{I_{1,2}} \leftarrow \{\mathbf{I[i]} \mid \mathbf{I[i] (mod \ 3) = 1}$ or $\mathbf{2}\}$
7:      $\mathbf{I_0} \leftarrow \{\mathbf{I[i]} \mid \mathbf{I[i] (mod \ 3) = 0}\}$
8:      $\mathbf{I_{1,2}} \leftarrow$ GPU-modified-string-sort($\mathbf{B_p}, \mathbf{I_{1,2}}$, lim)
    // MCU doubled after lim iter.
9:      $\mathbf{I_0} \leftarrow$ GPU-non-iterative-sort($\mathbf{B_p}, \mathbf{I_{1,2}}, \mathbf{I_0}$)
    // synthesize sorted order
    /* CPU computation is fully overlapped with asynchronous GPU calls */
10:     $\mathbf{I} \leftarrow$ CPU-merge($\mathbf{B_p}, \mathbf{I_{1,2}}, \mathbf{I_0}$)
    // non-iterative merge of two sorted suffix sets
11:     bwt $\leftarrow$ CPU-offset-addition($\mathbf{I}, \mathbf{B_p}, \mathbf{N}$) $\oplus$ index-of($\mathbf{0}, \mathbf{I}$)
    // generate BWT
12:     result $\leftarrow$ CPU-mtf-huffman(bwt)
    // perform MTF and Huffman encoding
13: end for
```

The compressed file size increases slightly as we increase the entropy by adding random characters, but our results (Section V-A) show that this increase is reasonable. This optimization also provides us a way to trade-off compression time against compression ratio. The speedup obtained by string perturbation is very useful on datasets with very high sorting depths viz. *linux-2.6.11.tar* (8.2× speedup after 1% perturbation as shown in Table I).

C. Overview of Hybrid BWC Algorithm

We have developed a hybrid BWC algorithm that makes use of a single CPU core and the GPU. In the design of our hybrid BWC algorithm we take into account differences between CPU and GPU and map the appropriate operations to the appropriate compute platform. The BWC algorithm consists of three steps: (i) Burrows Wheeler Transform, (ii) Move to Front Transform (MTF), and finally (iii) Huffman encoding. Typically BWT computation itself takes about 80-90% of the total computation time. MTF and tree building step of Huffman coding are completely serial and it is difficult to extract performance by mapping these to a data-parallel model. Based on these observations, we perform the bulk of BWC computation i.e. BWT operation on the GPU (using steps discussed in Section III-A) and we perform the remaining computations of MTF and Huffman encoding on the controlling CPU thread. Also note that, as discussed earlier, during BWT the merge step after partial sorts is also performed on CPU. This hybrid BWC is illustrated in Figure 3. Barring the last block, the merge, MTF and Huffman operations of all the blocks are performed in a fully overlapped manner with partial sorts on GPU. This makes good use of the idle CPU cycles.

The pseudo code of our hybrid BWC algorithm is given in Algorithm 1. Our algorithm takes as input a file $\mathbf{F}$ (line 1) and splits it into multiple blocks ($[\mathbf{B_1}, \mathbf{B_2}, \cdots, \mathbf{B_n}]$) each of size $\mathbf{N}$ (line 2). These blocks are perturbed (line 3) and then undergo sort steps of BWT on the GPU and remaining merge, mtf and huffman encoding steps on the CPU one after the other (line 4). We create an index array, $\mathbf{I}$, which denotes the starting position of each suffix string (line 5). Note that, since cyclically shifted suffix strings are used during BWT, each of the $\mathbf{N}$ strings also has a length of $\mathbf{N}$. We use modified string sorting method described above, along with doubling MCU optimization to sort strings that occur at positions $\mathbf{I_{1,2}}$ (line 8). After this string sort, $\mathbf{I_{1,2}}$ contains the indices of strings in sorted order. We use this sorted order to non-iteratively synthesize the sorted order for strings at positions $\mathbf{I_0}$ (line 9). The results of partial sorts are handed over to the CPU for performing merge and remaining BWC steps (line 10 to 12), and GPU simultaneously begins the sort steps on the next block.

In our algorithm for all operations on GPU, we use the fastest sort, scatter and scan primitives. These primitives are tuned for every new architecture and there are also algorithmic improvements which improve their performance. Our GPU BWC, built on these primitives, can directly inherit all these improvements and is adaptable to future architectures without requiring any re-design. Table I and Figure 5 show that our hybrid BWC gives a max. $2.9\times$ speedup over standard CPU BWC implementation i.e. Bzip2. In our hybrid BWC we only use a single CPU core. We further improve our speedup by using the other idle CPU cores through our all-core framework as shown in Tables II and III.

IV. The All-Core Framework

The *all-core* framework shown in Figure 4 is detailed in this section. A typical heterogeneous computation platform consists of multi-core CPUs, many-core GPUs, and/or other accelerators. The specific platform we focus on consists of a multi-core CPU and a GPU, but the framework extends easily to others. We treat each CPU thread as a compute station or *CoSt*. The number of CoSts can exceed the number of CPU cores with hyperthreading. Each GPU with a controlling CPU thread is another CoSt. The GPU programming models are getting more flexible and multiple CoSts may coexist on a physical GPU in the future. Each CoSt can be assigned a specific task. Each CoSt is free to choose the best possible strategy to perform the given task. A CPU core as a CoSt will attack the problem sequentially while a GPU as CoSt will resort to data-parallelism. For task

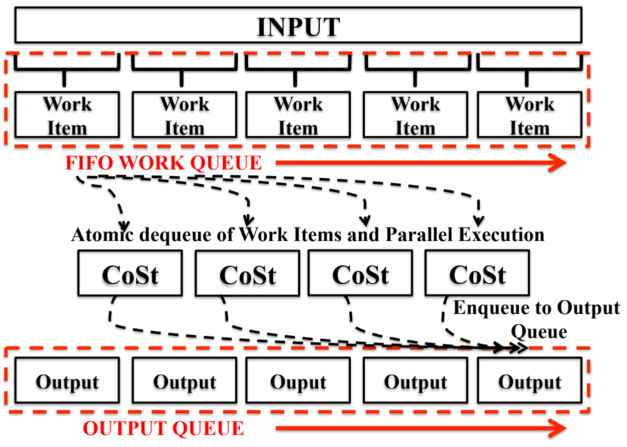


Figure 4: Work Queue based all-core framework.

partitioning we present a simple, but generic strategy based on *work queues* that can be used for several problems. Each CoSt dequeues an appropriate task from the work queue, processes it, and enqueues the results to an output queue. This framework is best suited for applications that process large data, but in independent blocks. There may be some pre-processing before independent blocks are formed and some post-processing to combine the outputs of independent blocks. Several applications fit the work queue model such as video encoding, decoding, and transcoding, lossy or lossless data compression, etc. We confine our attention to BWC compression in this paper, though extension to other applications is straightforward.

A. BWC in All-Core Framework

BWC on large data buffers (files or others) is performed by dividing it into blocks which are processed independently. A CoSt processes a block. We add the entire data buffer to a work-queue. Each CoSt, when free, removes a work item of appropriate size from the queue and processes it. Since our target architecture has two types of CoSts, (i) CPU core and (ii) GPU with a single controlling CPU thread, we use two BWC implementations. CPU CoSts use the best sequential implementation of BWC. On the GPU we use our hybrid BWC algorithm. We call this implementation which uses both, CPU (BWC by [17]) and GPU (our hybrid BWC as described in III-C), as the all-core BWC. It is also possible to use only CPU cores in our all-core framework, this gives us a multi-core BWC. The multi-core BWC is similar to [10], but it is fully compatible with Bzip2 compression standard. In Table II and III we show the speedup achieved using all-core BWC and multi-core BWC. Note that the use of GPU acceleration enables our all-core approach to achieve additional speedup over multi-core CPUs.

Dataset (Size)	Block Size	**(i) Compression time for our hybrid BWC (s), (ii) Compression time for CPU BWC (s) [17], (iii) Compressed file size in MB's (same for both)**			
		0% Perturbation	*0.01% Perturbation*	*0.1% Perturbation*	*1% Perturbation*
enwik8 (96MB)	900KB	**10.07**, 10.81, 27.66	**10.03**, 10.85, 27.70	**9.91**, 10.88, 28.09	**8.87**, 10.97, 31.32
	4.5MB	**7.29**, 13.12, **25.62**	**7.29**, 13.11, **25.67**	**7.31**, 13.10, **26.09**	**7.60**, 13.22, 29.36
	9MB	**8.31**, 15.23, **24.86**	**8.30**, 15.22, **24.91**	**8.33**, 15.82, **25.33**	**8.63**, 15.27, 28.66
wiki-xml (151MB)	900KB	**36.88**, 38.29, 15.29	**36.56**, 38.16, 15.39	**33.85**, 37.63, 16.19	**23.49**, 32.07, 21.89
	4.5MB	**30.42**, 60.78, **13.66**	**30.14**, 60.76, **13.77**	**26.97**, 60.55, **14.55**	**15.96**, 48.52, 19.82
	9MB	**31.51**, 80.76, **13.13**	**31.12**, 80.77, **13.24**	**27.62**, 79.94, **14.04**	**15.79**, 66.12, 19.07
linux-2.6.11.tar (199MB)	900KB	84.86, 24.93, 35.35	48.01, 24.69, 35.46	32.84, 23.21, 36.44	**22.78**, 22.17, 44.19
	4.5MB	133.54, 45.66, **33.10**	41.37, 44.02, **33.23**	**24.17**, 39.88, **34.26**	**14.24**, 26.65, 42.31
	9MB	196.64, 53.59, **32.51**	45.55, 51.77, **32.65**	**23.81**, 32.11, **33.69**	**14.37**, 29.64, 41.81
silesia.tar (203MB)	900KB	39.56, 29.65, 52.06	36.14, 29.69, 52.17	**28.98**, 29.32, 52.97	**23.06**, 27.46, 59.49
	4.5MB	34.60, 39.57, **50.06**	**29.52**, 39.63, **50.19**	**22.97**, 32.67, **51.03**	**16.81**, 36.07, 57.54
	9MB	36.10, 46.73, **49.57**	**28.85**, 46.92, **49.70**	**24.55**, 46.31, **50.55**	**17.74**, 41.94, 57.11

Table I: This table shows impact of block size, string perturbation on runtime and compressed file size (CPU BWC runtime is that of the standard `Bzip2` and the GPU BWC runtime is that of our `hybrid BWC` implementation). Bold values indicate cases where we get either better compression and/or runtime compared to the baseline i.e. standard CPU BWC on the default 900KB blocks (denoted by underline).

V. Experimental Results

We evaluate the performance of our hybrid BWC (Section V-A), multi-core and all-core BWC (Section V-B) on different datasets and on different CPUs and GPUs. The following standard datasets for lossless data compression were used in our experiments:

- Enwik8 [14]: The first 10^8 bytes of the English wikipedia dump on March 3, 2006. We also use wikipedia's enwiki-latest-abstract-10.xml (henceforth, referred to as wiki-xml) dataset [18].
- Publicly available source of linux kernel 2.6.11 (199MB).
- Silesia data corpus, a widely used standard data compression benchmark which has large files from various sources viz. database, codes, medical images etc. [6]. We tar (concatenate) all the files and use the tarred file (silesia.tar) as a dataset.

In our results, we compare the performance of our hybrid BWC pipeline against single-core CPU BWC (Section V-A). These results show that our hybrid BWC pipeline performs about $2.9\times$ better than the highly tuned CPU BWC and thus, about $8\times$ faster than the previous GPU BWC [15]. The reader should note that this is the first time a speedup has been achieved on GPU for BWC. Finally, we show the speedup we achieve through multi-core BWC and our all-core BWC implementation on a high-end as well as a low-end system (Section V-B). The dataset and the code used for our experiments is available at http://cvit.iiit.ac.in/resources/bzip2GPU/bzip2Cvit.tar.gz.

A. Results: Hybrid BWC

We measure the runtime of our hybrid BWC (as described in Section III-C) against the state-of-the-art CPU BWC implementation in Bzip2. Table I gives the total runtime and compressed file size on different datasets using different block sizes and with varying percentage of perturbation. Larger block size provides better compression, this is because BWT can now group together characters from a larger area for the MTF and Huffman steps and this has already been demonstrated by Burrows and Wheeler [3]. The performance of our hybrid BWC algorithm improves with increase in block size (except only for linux dataset with no perturbation), while the CPU performance becomes worse. For wiki-xml dataset, from 900KB to 9MB block size (without perturbation) the runtime of our hybrid algorithm improves by 15% (36.88s to 31.51s) while the runtime of the CPU BWC more than doubles (38.29s to 80.76s). Also, the compressed file size reduces by 14% (15.29 to 13.13MB) with 9MB blocks. Thus, our hybrid BWC scales better with block size and can be used to obtain better compression in lesser time as compared to CPU BWC.

To further improve our speedup and address worst-case datasets viz. linux-2.6.11.tar, we use string perturbation. We see that with increase in perturbation (random characters added), the runtime of CPU BWC is nearly same but the runtime of our hybrid BWC improves. Even for the worst-case linux dataset we beat the CPU with perturbation $>= 0.01\%$ and on large blocks. Through perturbation we are adding additional entropy to the input and the compressed file size increases. The current state-of-the-art runtime/compression is provided by CPU BWC running with 900Kb block size and no perturbation (marked by underline). Table I shows

NVIDIA GTX 580 (GPU) + Intel Core i7 920 (CPU)								
Dataset	**Total time for all-core BWC (CPU+GPU) (s), (#blocks processed by GPU / #total blocks)** Different number of CPU threads in addition to one CPU+GPU thread							Speedup (bold) vs. single CPU (underlined)
	0 CPU	**1 CPU**	**2 CPU**	**3 CPU**	**4 CPU**	**6 CPU**	**7 CPU**	
enwik8	8.4 (12/12)	6.5 (9/12)	5.8 (7/12)	4.7 (6/12)	4.7 (5/12)	4.6 (5/12)	**3.9 (4/12)**	4.05
wiki-xml	27.6 (18/18)	23.6 (13/18)	19.6 (10/18)	**16.4 (10/18)**	16.6 (8/18)	18.9 (7/18)	18.6 (6/18)	4.87
linux	23.8 (22/22)	14.3 (13/22)	10.88 (8/22)	9.3 (8/22)	9.1 (7/22)	**7.7 (5/22)**	7.9 (4/22)	4.16
silesia	24 (23/23)	16.8 (16/23)	13.3 (12/23)	12.6 (12/23)	12.7 (11/23)	11.4 (9/23)	**11.0 (8/23)**	4.20

Only Intel Core i7 920 (CPU)								
Dataset	**Total time for multi-core BWC (CPU only) (s)** Different number of CPU threads (No GPU involved)							Speedup (bold) vs. single CPU (underlined)
	1 CPU	**2 CPU**	**3 CPU**	**4 CPU**	**5 CPU**	**7 CPU**	**8 CPU**	
enwik8	15.8	9.1	6.4	5.0	5.0	4.9	**4.9**	3.22
wiki-xml	79.9	44.5	32.3	28.4	25.6	26.2	**26.1**	3.06
linux	32.1	17.6	13.3	10.7	10.0	10.3	**9.9**	3.24
silesia	46.3	30.9	21.5	21.4	21.1	**17.4**	18.4	2.66

Table II: For this table, we use a high-end system with Intel Core i7 CPU and Nvidia GTX 580 GPU. The table shows runtime for all-core BWC (CPU+GPU) and multi-core BWC (CPU only). Also, if we compare n CPU threads to $n-1$ CPU threads and 1 CPU+GPU thread, the runtimes of latter are better. This again shows our hybrid BWC is faster than CPU BWC.

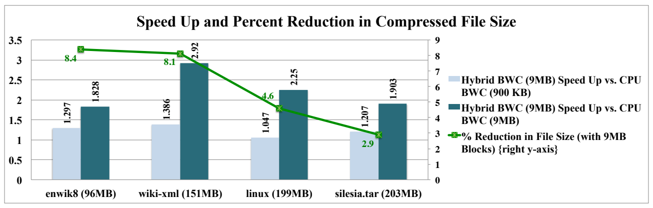


Figure 5: Our hybrid BWC (with 9MB blocks) pipeline performs marginally better than CPU BWC with 900KB blocks (which does much less work) and gives max. 2.9× speedup when compared to CPU BWC with 9MB blocks. Using 9MB blocks gives gain in compression ratio.

that with 0.1% perturbation and block size greater than $4.5MB$, we obtain better runtime as well as compression (marked by bold values) on all 4 datasets as compared to the state-of-the-art (marked by underline). The light-blue bars in Figure 5 show the speedup ($1.04-1.38\times$) obtained with respect to BWC on 900KB blocks and green line in the same figure shows the corresponding reduction in compressed file size ($2.9-8.4\%$). Note that, in comparison to the state-of-the-art, our hybrid BWC implementation (using 9MB Blocks) is outperforming the CPU, even when the CPU is doing much less work (BWT performance is worse than linear in block size) by using only 900KB blocks. Also, to achieve the same compression ratio (using large block size) on the CPU will take much more time as compared to our hybrid algorithm (the speedup when both standard CPU and our hybrid BWC uses 9MB block size is indicated by dark-blue bars in Figure 5).

B. *Results: All-Core BWC*

We use the work queue based all-core framework and run multi-core BWC (i.e. only the CPU cores) and all-core BWC (which uses both all CPU cores and GPU) on all our datasets. The block size is fixed to 9MB and perturbation to 0.1%, since we obtained optimal performance for these parameters (Table I).

High-End System. In the first setup, we compare the performance of the multi-core and all-core BWC on a high end system comprising of Intel Core i7 920 CPU and Nvidia GTX 580 GPU. The results are given in Table II. The 4-core Intel i7 CPU, with hyper-threading supports 8 threads at a time efficiently and in practice,

NVIDIA GTX 280 (GPU) + Intel Core2Duo E6750 (CPU)

Dataset	**Total time for all-core BWC (CPU+GPU) (s), (#blocks processed by GPU / #total blocks)** Different #CPU threads in addition to 1 CPU+GPU thread					Best Speedup (bold) vs. single CPU (underlined)
	0 CPU	**1 CPU**	**2 CPU**	**3 CPU**	**4 CPU**	
enwik8	20.5 (12/12)	19.3 (6/12)	**19.8 (5/12)**	20.1 (4/12)	20.0 (3/12)	1.33
wiki-xml	90 (18/18)	**77.4 (12/18)**	77.6 (10/18)	77.8 (8/18)	85.1 (6/18)	1.67
linux	77.3 (22/22)	46.1 (11/22)	40.3 (7/22)	39.8 (6/22)	**39.2 (5/22)**	1.21
silesia	74.3 (23/23)	57.0 (14/23)	**53.3 (10/32)**	53.6 (9/23)	54.0 (10/23)	1.30

Only Intel Core2Duo E6750 (CPU)

Dataset	**Total time for multi-core BWC (CPU only) (s)** Different number of CPU threads (No GPU involved)					Best Speedup (bold) vs. single CPU (underlined)
	1 CPU	**2 CPU**	**3 CPU**	**4 CPU**	**5 CPU**	
enwik8	26.4	22.4	**21.9**	22.5	22.3	1.20
wiki-xml	129.9	**106.0**	108.3	109.6	108.1	1.22
linux	47.6	**36.2**	37.5	37.7	37.7	1.31
silesia	69.59	55.26	**53.77**	56.1	56.0	1.29

Table III: For this table, we use a low-end Intel Core2Duo E6750 CPU and Nvidia GTX 280 (low/med-end) GPU. The table shows runtimes for all-core BWC (CPU+GPU) and multi-core BWC (CPU only).

increasing the number of threads above 8 did not improve the performance. Thus in our experiments we vary the number of CPU threads between 1 to 8. In this range, as expected, for both implementations, the performance generally improves with more threads. Also in Table II, on adding a single additional CPU thread to the CPU+GPU thread, we see that GPU still processes 9 out of 12 blocks, 13 out of 18 blocks for enwik8 and wiki-xml respectively. This reaffirms that our hybrid BWC is 2 times faster as compared to CPU BWC on these datasets and the work-load is balanced according to speed of CoSts. The best runtimes on enwik8, wiki-xml, linux and silesia.tar datasets for the multi-core BWC are $4.9s$, $26.1s$, $9.9s$, $17.4s$ respectively, which improve to $3.9s$ ($1.25\times$), 16.4 ($1.59\times$), $7.7s$ ($1.28\times$) and $11.0s$ ($1.58\times$) using the all-core BWC. If we look at the speedup with respect to the single-core CPU BWC implementation, we see that our all-core BWC achieves a consistent speedup greater than $4\times$ for all the datasets compared to the about $3.24\times$ speedup obtained by the multi-core BWC. This shows that our all-core CPU+GPU BWC scales to all the available cores (GPU in addition to the CPU) in the system and provides maximum speedup.

Low-End System. In the second setup, we compare the performance of multi-core BWC and all-core BWC on Intel Core2Duo E6750 CPU with Nvidia GTX 280 (Table III). This is a less powerful GPU with limited support for parallelism as compared to Nvidia GTX 580. The best speedup obtained on GTX 280 setup is $1.67\times$, while the multi-core BWC gives best speedup of $1.22\times$.

VI. Conclusion

In this paper, we presented an all-core framework to exploit accelerators available on a user's system. We demonstrated the practical utility of our all-core framework by implementing the Burrows Wheeler Compression (BWC) pipeline. We demonstrated a speedup on BWC on the GPU for the first time. Our hybrid BWC achieves good speedup over highly tuned CPU BWC. It also handles large block size efficiently, providing better compression ratio along with better runtime as compared to state-of-the-art. Our all-core framework uses all CPU cores and GPU cores effectively, balancing the load between available resources. Everyday users haven't gained much from enhanced computing power of today's heterogeneous parallel processing platforms. This is an area that needs attention as multi-core CPUs, many-core GPUs, and other accelerators become more prevalent. The ideas of all-core framework and results on an *end-to-end* application we presented can improve the application performance on emerging accelerator based platforms.

References

[1] D. Adjeroh, T. Bell, and A. Mukherjee. *The Burrows-Wheeler Transform:: Data Compression, Suffix Arrays, and Pattern Matching*. Springer, 1 edition, July 2008.

[2] J. L. Bentley and R. Sedgewick. Fast algorithms for sorting and searching strings. In *SODA*, pages 360–369, 1997.

[3] M. Burrows and D. Wheeler. A block sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, 1994.

[4] G. Bzip2Cuda. Github bzip2-cuda, http://bzip2-cuda.github.io/, 2011.

[5] A. Davidson, D. Tarjan, M. Garland, and J. D. Owens. Efficient parallel merge sort for fixed and variable length keys. In *Proceedings of Innovative Parallel Computing (InPar '12)*, May 2012.

[6] S. Deorowicz. Universal lossless data compression algorithms, 2003.

[7] A. Deshpande and P. J. Narayanan. Can gpus sort strings efficiently? In *High Performance Computing (HiPC), 2013 International Conference on*, 2013.

[8] J. A. Edwards and U. Vishkin. Brief announcement: Truly parallel burrows-wheeler compression and decompression. In *Proceedings of ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '13, 2013.

[9] J. A. Edwards and U. Vishkin. Empirical speedup study of truly parallel data compression. 2013.

[10] J. Gilchrist and A. Cuhadar. Parallel lossless data compression using the burrows wheeler transform. *Int. J. Web Grid Serv.*, 4(1):117–135, May 2008.

[11] J. Kärkkäinen and P. Sanders. Simple linear work suffix array construction, 2003.

[12] D. K. Kim, J. S. Sim, H. Park, and K. Park. Linear-time construction of suffix arrays. In *Combinatorial Pattern Matching*, pages 186–199, 2003.

[13] C. Liu, R. Luo, and T. W. Lam. Gpu-accelerated BWT construction for large collection of short reads. *CoRR*, abs/1401.7457, 2014.

[14] M. Mahoney. Enwik8, http://mattmahoney.net/dc/text.html, 2006.

[15] R. A. Patel, Y. Zhang, J. Mak, and J. D. Owens. Parallel lossless data compression on the GPU. In *Proceedings of Innovative Parallel Computing (InPar '12)*, May 2012.

[16] K. Sadakane. A fast algorithm for making suffix arrays and for burrows-wheeler transformation. In *In Proceedings Of The IEEE DCC*, 1998.

[17] J. Seward. On the performance of bwt sorting algorithms. In *Data Compression Conference*, pages 173–182. IEEE Computer Society, 2000.

[18] Wikipedia. http://dumps.wikimedia.org/enwiki/latest/, 2014.