Fast Burrows Wheeler Compression Using CPU and GPU

by

Aditya Deshpande and P J Narayanan

Report No: IIIT/TR/2014/

Center for Visual Information Technology
International Institute of Information Technology
Hyderabad - 500 032, INDIA
April 2014
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 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 implementation achieves a 2.9× speedup over the 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× speedup by using all CPU cores and a 4.87× speedup using the GPU also in the all-core framework. Our approach will scale to the number and type of computing resources on a system.

General Terms: Parallel Algorithms, GPGPU, Data Compression

Additional Key Words and Phrases: Burrows Wheeler Transform, Bzip2, Suffix Sort, CUDA, Thrust

ACM Reference Format:

1. INTRODUCTION

Computing platforms are parallel today. CPUs have multiple identical cores. A GPU with dozens to hundreds of simpler cores is present on many systems. In future, other accelerators may also be part of computer systems. 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 multi-core CPUs, many-core accelerators like the GPU, and their combinations have been developed recently [Intel 2013; Bader et al. 2007; Rehman et al. 2009; Zhou et al. 2008; Deshpande et al. 2011]. 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. It is high time the benefits of using every compute core on commodity computers be made available to common users.

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) [Burrows and Wheeler 1994]. 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 [Adjeroh et al. 2008]. 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 [Gilchrist and Cuhadar 2008]. A recent GPU BWC effort performed slower than a single core CPU [Patel et al. 2012]. We develop an intra-block, fine-grained BWC algorithm on GPU that outperforms the state-of-the-art CPU implementation by Seward [Seward 2000]. We also present an all-core framework where inter-block parallelism is exploited to divide the tasks among multiple computing stations of the CPU and we use intra-block parallelism on the
fine-grained (many-core) architecture of the GPU. Our results show significant performance benefits as well as effective load balancing using all cores on the system. 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 (Section 3.2 and 5.1). We extend the GPU string sort approach developed by us \(^1\) [Deshpande and Narayanan 2013] to address the high tie-lengths common in compression datasets and use it to perform suffix sort step of BWT.

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 string perturbation step to increase speed at a slight reduction in compression ratio (Section 3.3). The known perturbations can be reversed after decompression to recover the original data block.

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 3.4).

4. We develop an all-core computation framework to exploit all compute cores on a computer system. For task/work partitioning, we present a simple strategy using work queues that can apply to several applications. We use our hybrid BWC approach on the compute stations with GPUs and the optimal BWC code from Seward [Seward 2000] on the CPU compute stations. Section 4 describes our all-core framework and its application to BWC in more detail. We modify the state-of-the-art BWC implementation to support large block sizes and concurrent processing of blocks by CPU and GPU. Our code is available for public use.\(^2\)

5. We demonstrate significant speedup as well as better compression ratios on a set of challenging datasets (Section 5.2 and 5.3). Our hybrid BWC is the first to report a speedup on the GPU. The all-core BWC produces linear speedup using only multi-core CPUs, while being compatible with present Bzip2 standards. The all-core BWC using both CPU cores and GPU gives better runtime (than multi-core CPU BWC), while balancing the load between the GPU and the CPUs.

Our implementation scales well on different combination of compute cores available on a system, providing optimal performance. On an Intel Core i7, our all-core implementation achieves a 3.06× speedup, which improves to 4.87× when an Nvidia GTX 580 is included (Table II). On a low-end Intel Core2Duo, our all-core BWC achieves a 1.22× speedup which improves to a 1.67× speedup with the addition of Nvidia GTX 280 (Table III). We believe that the techniques and the lessons from this work will motivate future work in designing all-core implementations for other end-to-end applications. A suite of similar end-to-end applications that exploit every compute core will truly allow the common user to enjoy the benefits of parallel computing on the desktop.

2. BACKGROUND AND RELATED WORK

In this section we review related work on BWC, BWT and its sequential and parallel implementations.

2.1. Burrows Wheeler Transform

Burrows and Wheeler, developed BWT [Burrows and Wheeler 1994] with following steps: (i) Start with the input string \(S[1...N]\) and associated index array \(I[1...N]\), ini-

\(^1\) Code available at http://web.iiit.ac.in/~aditya.deshapandeug08/stringSort/ and https://github.com/aditya12agg5/cudpp

\(^2\) Code for All-Core BWC available at http://cvit.iiit.ac.in/index.php?page=resources

Submitted to ACM Transactions on Parallel Computing.
Last column along with the index of original string (i.e. 4 since \([I[4]=1]\) is the BWT OUTPUT

<table>
<thead>
<tr>
<th>([IN])</th>
<th>([SN])</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>a</td>
</tr>
<tr>
<td>4</td>
<td>a</td>
</tr>
<tr>
<td>2</td>
<td>a</td>
</tr>
<tr>
<td>1</td>
<td>b</td>
</tr>
<tr>
<td>5</td>
<td>n</td>
</tr>
<tr>
<td>3</td>
<td>n</td>
</tr>
</tbody>
</table>

OUTPUT MATRIX

Last column can be easily computed by offset addition even if we output this shuffled \([IN]\).

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

...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 the 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 the 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 \(S[1...6] = \text{banana}\). It generates the output matrix containing a sorted list of all cyclically shifted strings. The answer is the last column of output matrix, i.e. \(\text{nbaaa}\), appended with number 4, since the original string occurs at 4th position in the output. In general, the 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. For example, if there are 2 same substrings, say a few 1000 characters long, that are repeated in the input, then many suffix strings will have these few 1000 characters matching. Resolving ties between all these strings during sorting is expensive. In practice, cyclically shifted suffix strings match to lengths as high as \(10^3\) to \(10^5\) characters within a 9M character block. This shows the compute intensive nature of BWT.

### 2.2. Sequential Burrows Wheeler Compression

BWT was used to devise a lossless BW compression scheme by Burrows and Wheeler [Burrows and Wheeler 1994]. For suffix sort, they performed a radix sort on first two characters \((c_1c_2)\) of all suffixes followed by a modified Quicksort [Bentley and Sedgewick 1997] in subsequent iterations. They also developed a special mechanism to handle inputs with long repeated runs on the same character. Their BWC was slow. Seward proposed a method that generated 256 depth one buckets based on the third character \((c_3)\) after the two-character radix sort [Seward 2000]. Within each bucket, they sorted only those suffixes starting with \(c_3c_4\) (such that \(c_3 \neq c_4\)) and cleverly synthesized sorted order for suffixes in other buckets (of the form \(c_s,c_3\)). This resulted in an efficient and popular Bzip2 file compressor. Incorporating Sadakane’s algorithm

Submitted to ACM Transactions on Parallel Computing.
Fast Burrows Wheeler Compression Using CPU and GPU

[Sadakane 1998] improved the performance even on worst case inputs. Synthesizing sorted order of new suffixes from previously sorted ones avoids expensive matching and results in good performance of these methods. Such fine-grained synthesis methods require synchronization and are difficult to do on the GPU architecture. Our GPU implementation uses a coarse synthesis technique developed by Kärkkäinen and Sanders [Kärkkäinen and Sanders 2003], wherein after sorting only \((2/3)^{rd}\) of the suffix strings, the rest can easily be sorted and merged.

2.3. Parallel Burrows Wheeler Compression

Gilchrist and Cuhadar [Gilchrist and Cuhadar 2008] 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. [Patel et al. 2012] repeatedly sorts strings using a variable length key comparison based sort [Davidson et al. 2012]. Their implementation was about 2.78× 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 [Bzip2Cuda 2011]. We build our BWT on a GPU string sorting algorithm developed by us [Deshpande and Narayanan 2013] that uses radix sort. Our GPU BWT implementation achieves more than 2× speedup. Our implementation scales to larger block sizes giving better compression ratios.

Edwards and Vishkin BWC. In parallel with our work, Edwards and Vishkin [Edwards and Vishkin 2013a] developed an intra-block parallel BWC algorithm and compared it with the CPU BWC [Edwards and Vishkin 2013b]. 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 multi-core CPUs. In contrast, we demonstrate better performance on multi-core CPUs and GPUs. They report a speedup of 1.8 to 2.8× on XMT-64 (~ 64 cores) platform and 12 to 25× on a simulated XMT-1024 (~ 1024 cores) platform. They use files from Large Corpus\(^3\), which are small in size (< 4.5MB’s) by today’s standards and have a low maximum sorting depth (< 2000). We demonstrate high performance on large datasets with higher sorting depth \((10^5 \text{ to } 10^7)\) like Enwik\(8\) (96MB) [Mahoney 2006], Linux-2.6.11.tar (199MB) [Kernel 2005] and Silesia Corpus\(^4\) (203MB, which supersedes Large Corpus) [Deorowicz 2003]. It would be worthwhile to compare the runtime of our algorithm with Edwards and Vishkin [Edwards and Vishkin 2013a] on same platforms and same test datasets in the future.

3. 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. In principle, string sorting can be performed on GPU by using a custom comparator in a GPU merge sort algorithm\(^5\) [Patel et al. 2012; Davidson et al. 2012]. This custom comparator iteratively compares two strings. In a merge sort, input is recursively split into small buckets and these buckets can

---

\(^3\)http://corpus.canterbury.ac.nz/descriptions/


\(^5\)http://goo.gl/mlwlZ

Submitted to ACM Transactions on Parallel Computing.
be sorted independent of each other. Parallelism of GPU is also leveraged to perform merging of buckets co-operatively using 2 or more thread blocks/SM’s. But, the iterative comparisons discussed earlier are performed at every merge step. Moreover, to perform such comparisons during successive merge steps, same string is loaded again and again from the high latency global memory. Threads in a warp diverge because of varying length comparisons. A traditional GPU merge sort based string sort runs 2 – $5 \times$ slower than CPU when used to perform suffix sorting [Patel et al. 2012]. Thus, instead we use a radix sort based string sorting procedure.

### 3.1. Review of Radix Sort based String Sort

A radix sort based approach can also be used to perform string/suffix sorting. Using a radix sort procedure, suffix sorting can be done in a 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, 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 the merge sort based methods.

This radix sort based string sorting technique, for the GPU, has been described in detail in [Deshpande and Narayanan 2013]. A brief description of it is given here. As shown in Figure 2, our method sorts strings from left to right and each part of the string is accessed only once. Our string sort groups characters to form multi-character units (MCUs) of $k$ characters each ($k = 4$ to $8$, Figure 2 uses $k = 2$ for simplifying the illustration) and sorts using them as keys. On performing the first sort of $k$ characters we obtain a few unique occurrences of MCUs, i.e., singletons or size 1 buckets (viz. $GR$ in Figure 2). The sorted order for strings corresponding to these singleton MCUs (i.e. $GRAPH$ which corresponds to $GR$ in Figure 2) is now known and we can remove them from further sort iterations. On the other hand, repeated occurrences of MCUs get clubbed together within a bucket (viz. $2 CO$’s, $3 PA$’s, $2 RA$’s in Figure 2). The ordering of these strings, within each bucket, can be corrected by looking at the successor MCUs. Note that, all the buckets are lexicographically ordered and their MCUs should...
not mix with each other in future sort steps. To ensure this, bucketID is assigned as most significant few bytes of each MCU (marked by green boxes in Figure 2). In the remaining part of MCU (i.e. \( k - \text{len}(\text{bucketID}) \)), we load successor MCUs. The process of: (i) sorting (using fastest parallel radix sort), (ii) creating buckets and eliminating singletons (using parallel scan and scatter primitives) and (iii) loading successors (embarrassingly parallel) continues till all buckets are of size 1 (i.e. no buckets will be left after next elimination). The final sorted output is obtained at this stage.

At each step for the string sort, we perform global sorts by replacing MCUs with their successors and maintaining bucket ordering information to record history of previous sorts. This allows us to use the fast GPU radix sort primitive to perform each sort step. Also, by identifying singleton MCUs, we eliminate strings as soon as their position in output becomes fixed. This progressively reduces the sort problem size and improves runtime. We sort maximum portion of the string per sort step by adaptively using minimum bytes to record bucketID within each MCU. In all the above steps, we exploit sort, scan and scatter primitives on the GPU. Exploiting fast standard GPU primitives and reducing high latency global memory accesses, allows our GPU string sort to provide \( 10 \times \) improvement over previous methods. More details about the impact of each optimization on runtime could be found in [Deshpande and Narayanan 2013]. The pseudo code is given in Appendix.

3.2. Modified String Sort for BWT

For BWT, all \( N \) suffix strings are each of length \( N \) and these strings share long matching prefixes with each other. This is because, compression schemes are typically used on datasets with high redundancy or match length (in form of matching substrings). Matching substrings give rise to matching prefixes for suffix strings. Long matching prefixes results in large number of iterations for the string sort algorithm. The string sorting approach works well only when the input has ties up to a few 100 characters [Deshpande and Narayanan 2013]. Suffix sorting step of BWT has relatively much higher number of ties (\( 10^3 \) to \( 10^5 \) characters). In such a scenario sorting using constant-sized MCUs (\( k \) characters long) takes too much time. Thus, we perform costly sorts on longer MCUs, but reduce the number of iterations by doubling the MCU length after every iteration. We use the property of suffix strings being

Submitted to ACM Transactions on Parallel Computing.
cyclically shifted to only sort a subset of strings and generate the sorted order for rest. These two sets of sorted suffix strings are then merged by utilizing idle CPU cycles. These BWT specific optimizations are discussed in more detail in subsequent sections –

**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. We use the terms match length and sorting depth interchangeably depending on the context. Large sorting depths result from long substrings repeating many times, which degrades the 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 3a, 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 \(\times 4 = 64\) 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 \(1.8 \times (1.06s to 0.58s per block)\) as compared to using constant sized MCUs in this particular case. Each sort step with longer MCU takes more time compared to the case of constant sized MCU, but the large drop in number of sort steps results in a much improved overall performance.

**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 \(i \ (mod\ 3) \neq 0\) (denoted by set \(I_{1,2}\)), we can generate the sorted order for all suffixes at \(i \ (mod\ 3) = 0\) (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 [Kärkkäinen and Sanders 2003; Kim et al. 2003; Seward 2000]. We adapt the efficient approach by [Kärkkäinen and Sanders 2003] to our GPU implementation. We perform the two sorting steps on GPU and move the merge step to CPU as shown in Figure 4. This allows us to utilize the idle CPU cycles while GPU is performing sort operation for subsequent block in BWC. Figure 3b shows that we obtain a speedup of 1.2 to \(2 \times\) as a result of using this optimization.

The optimizations discussed above are limited only to the BWT step of BWC. Specific to the problem of BWC, we develop another optimization to forcefully break long ties in the input. This is discussed in Section 3.3.

**3.3. 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. For example,
on a 4.5M character block of `linux-2.6.11.tar` adding a random character after every 1000th character reduced the maximum sorting depth from 65472 to 960 characters and the average sorting depth from 10078 to 825 characters (Figure 6, bottom right). 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. The compressed file size increases slightly as we increase the entropy by adding random characters, but our results (Section 5.2) show that this increase is reasonable. This optimization also provides us a way to trade-off compression time against compression ratio. Figure 6 (bottom right) and Table I demonstrate the significant improvement (8.2× speedup for `linux-2.6.11.tar` with 0.1% perturbation on 9MB blocks) in runtime obtained after using string perturbation. The speedup obtained by string perturbation is very useful on datasets with very high sorting depths viz. `linux-2.6.11.tar`.

### 3.4. 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. Patel et al. [Patel et al. 2012] implemented all three stages on the GPU with the hope of performing on the fly compression/decompression during data transfers. All three stages were individually slower on the GPU as compared to the CPU, with BWT step experiencing the maximum slowdown. Typically BWT computation itself takes about 80-90% of the total computation time on the CPU. The 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 3.2) 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 4. Barring the last block, the merge, MTF and Huffman operations...
of all the blocks are performed in a fully overlapped manner with the partial sorts on
the GPU. This makes good use of the idle CPU cycles, when GPU is busy doing the
sort operation, and provides a throughput for BWC that nearly equals the GPU BWT
throughput.

The pseudo code of our hybrid BWC algorithm is given in Algorithm 1. Our algorithm
takes as input a file $F$ (line 1) and splits it into multiple blocks ($B_1, B_2, \ldots, B_n$) each of
size $N$ (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, $I$, which denotes the starting position
of each suffix string (line 5). These blocks are perturbed (line 3) and then undergo sort steps of BWT
after the other (line 4). We create an index array, $I$, which denotes the starting position
each of the suffix strings (line 5). Note that, since cyclically shifted suffix strings are used
during BWT, each of the $N$ suffix strings also has a length of $N$. We use modified string
sorting method described above, along with doubling MCU optimization to sort strings
that occur at positions $I_{1,2}$ (line 8). After this string sort, $I_{1,2}$ contains the indices of
strings in sorted order. We use this sorted order to non-iteratively synthesize the sorted
order for strings at positions $I_0$ (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. The sort steps on GPU are thus
overlapped with other BWC steps on the CPU as shown in Figure 4.

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. The design of our hybrid BWC
algorithm makes best use of both CPU and GPU. Table I and Figure 7 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.

4. THE ALL-CORE FRAMEWORK
The all-core framework shown in Figure 5 is detailed in this section. A typical 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

\begin{algorithm}
\caption{Hybrid Burrows Wheeler Compression Algorithm}
1: Input: File $F$
2: $[B_1, B_2, \ldots, B_n] = \text{split-blocks}(F, N)$ \hspace{1em} // split file into size $N$ blocks
3: $[B_1, B_2, \ldots, B_n] \leftarrow \text{perturb-blocks}([B_1, B_2, \ldots, B_n], \text{interval})$ \hspace{1em} // random char is
\hspace{1em} added after interval
4: for $B_p$ in $[B_1, B_2, \ldots, B_n]$
\hspace{1em} 5: \hspace{1em} $I = \{1 \cdots N\}$ \hspace{1em} // Index Array, denotes starting position of each string
\hspace{1em} 6: \hspace{1em} $I_{1,2} \leftarrow \{I[i] \mid I[i](\text{mod} \ 3) = 1 \text{ or } 2\}$
\hspace{1em} 7: \hspace{1em} $I_0 \leftarrow \{I[i] \mid I[i](\text{mod} \ 3) = 0\}$
\hspace{1em} 8: \hspace{1em} $I_{1,2} \leftarrow \text{GPU-modified-string-sort}(B_p, I_{1,2}, \text{lim})$ \hspace{1em} // MCU doubled after $\text{lim}$ iter.
\hspace{1em} 9: \hspace{1em} $I_0 \leftarrow \text{GPU-non-iterative-sort}(B_p, I_{1,2}, I_0)$ \hspace{1em} // synthesize sorted order
\hspace{1em} \hspace{1em} /* CPU computation is fully overlapped with asynchronous GPU calls */
\hspace{1em} 10: \hspace{1em} $\text{bwt} \leftarrow \text{CPU-merge}(B_p, I_{1,2}, I_0)$ \hspace{1em} // non-iterative merge of two sorted suffix sets
\hspace{1em} 11: \hspace{1em} $\text{result} \leftarrow \text{CPU-mtf-huffman}(\text{bwt})$ \hspace{1em} // perform MTF and Huffman encoding
\hspace{1em} 12: end for
\end{algorithm}
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 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 application in this paper, though extension to other applications is straightforward.

4.1. 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. When done, it adds the output – a compressed bit stream in case of BWC – to the output queue and gets another item from the queue for processing. A manager thread on CPU performs the post-processing, which involves combining the output bit streams, adding headers, etc., and creates the final output. Each CoSt performs the same task, but uses an implementation that best fits its architecture. 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 [Seward 2000]) and GPU (our hybrid BWC as described in 3.4), as the all-core BWC. It is also possible to use only the CPU cores in our all-core framework, this gives us a multi-core BWC (which runs in parallel on multiple CPU cores). The multi-core BWC is similar to parallel imple-
mentation by [Gilchrist and Cuhadar 2008], but it is fully compatible with the Bzip2 compression standard. In Table II and III we show the speedup achieved using all-core BWC and multi-core BWC. These results show that our all-core BWC improves on the speedup obtained by using only CPU cores in the multi-core BWC. The development of hybrid BWC enables our all-core approach to achieve this additional speedup. Our work-queue based all-core framework keeps all the CoSts busy while there is more work to do. Thus, the framework provides great flexibility in mixing different types of compute stations while balancing the loads on them according to their capacities.

4.2. BW Decompression in All-Core Framework
Burrows Wheeler decompression has the same block level parallelism as compression and we use our all-core framework to gain speedup. We develop an all-core implementation of decompression where GPU CoSt performs the Inverse BWT step according to algorithm given by Patel [Patel 2012]. This algorithm involves 3 steps of sorting (1 byte key), list ranking and scatter. Efficient primitives to perform all these operations are available on the GPU. Since inverse BWT is a simple operation, GPU does not achieve a significant speedup over CPU. Thus, the overall speedup is linear in number of CoSt’s used through our all-core framework. BW compression is a tougher problem than decompression and we choose to focus on it in this work.

5. EXPERIMENTAL RESULTS
We evaluate the performance of our GPU BWT (Section 5.1), hybrid BWC (Section 5.2), multi-core and all-core BWC (Section 5.3) on different datasets and on different CPU and GPU platforms. The GPU BWC (Algorithm 1), was implemented on Nvidia GPUs using Thrust primitives for sort (key and value), scatter and scan [Hoberock and Bell 2010]. The following standard datasets for lossless data compression were used in our experiments:

— Enwik8 [Mahoney 2006]: 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 [Wikipedia 2014].

— Publicly available source of linux kernel 2.6.11 (199MB) [Kernel 2005].

— Silesia data corpus, a widely used standard data compression benchmark which has large files from various sources viz. database, codes, medical images etc. [Deorowicz 2003]. We tar (concatenate) all the files and use the tarred file (silesia.tar) as a dataset.

In our results, we benchmark the performance of our GPU BWT against the state-of-the-art BWT on CPU, i.e. BWT implemented in Bzip2 file compressor [Seward 2000] and we show the effectiveness of our string perturbation optimization for some datasets (Section 5.1). We also compare the performance of our hybrid BWC pipeline against single-core CPU BWC (Section 5.2). These results show that our hybrid BWC pipeline performs about 2.9× better than the highly tuned CPU BWC and thus, about 8× faster than the previous GPU BWC [Patel et al. 2012] (which was 2.78× slower than CPU). The reader should note that this is the first time a speedup has been achieved on GPU for the problem of BWC. Our hybrid BWC also gives better compression ratio in lesser time by using large block sizes, as compared to the maximum 900KB block size supported by standard CPU 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 5.3). The dataset and the code used for our experiments is available at http://cvit.iiit.ac.in/index.php?page=resources.

Submitted to ACM Transactions on Parallel Computing.
5.1. Results: GPU BWT

We benchmark the performance of both CPU and GPU BWT algorithms by varying block size (900Kb to 9MB) and measuring the runtime on datasets with different sorting depths, i.e. for 9MB blocks the maximum sorting depth varies from 960 (enwik8) to 2,62,080 (linux-2.6.11.tar). The BWT implementation in Bzip2 on the CPU is considered to be the state-of-the-art and we use it for comparison. Figure 6 shows that our GPU algorithm achieves speedup for large block sizes (> 4.5MB) since the GPU is not utilized well for smaller ones. We later demonstrate that these large block sizes also provide better compression (Section 5.2). Figure 6 shows that the runtime for GPU sort operations increases with increase in the sorting depth (indicated by dashed red line). On large block sizes, we achieve a speedup whenever the maximum sorting depth is upper bound by about $10^4$. The maximum speedup, of $2.5 \times$, is achieved for wiki-xml using 9MB blocks. For datasets, with an order of magnitude higher sorting depth viz. linux-2.6.11.tar, the GPU BWT is about $4 \times$ slower than the CPU. This is addressed using string perturbation.

Effect of String Perturbation. In Figure 6 (bottom right), we vary the % of perturbation from 0 (0 random characters added) to 1% (1 random character after every 100th) for the high sorting depth (practical worst-case) linux-2.6.11.tar dataset and measure the runtime. As we increase the % perturbation, both the maximum and average sorting depth reduces (since long ties are broken by random characters) and runtime improves (indicated by dashed red line). We see that at 0.01% perturbation itself, the GPU performs marginally better than the CPU. Beyond 0.01% perturbation, the speedup of GPU over CPU becomes significant. On linux-2.6.11.tar dataset with 9MB blocks the GPU algorithm goes from being $4 \times$ slower to $1.7 \times$ faster as the perturbation
Table I: Impact of Block Size and String Perturbation on Runtime/Compressed File Size for our hybrid BWC and standard CPU BWC i.e. Bzip2 by [Seward 2000]

<table>
<thead>
<tr>
<th>Dataset (Size)</th>
<th>Block Size</th>
<th>(i) Compression time for our hybrid BWC (s),</th>
<th>(ii) Compression time for CPU BWC (s) [Seward 2000],</th>
<th>(iii) Compressed file size in MB's (same for both)</th>
</tr>
</thead>
<tbody>
<tr>
<td>enwik8 (96MB)</td>
<td>900KB</td>
<td>10.07, 10.81, 27.66</td>
<td>10.03, 10.85, 27.70</td>
<td>9.91, 10.88, 28.09</td>
</tr>
<tr>
<td></td>
<td>9MB</td>
<td>8.31, 15.23, 24.86</td>
<td>8.30, 15.22, 24.91</td>
<td>8.33, 15.82, 25.33</td>
</tr>
<tr>
<td>wiki-xml (151MB)</td>
<td>900KB</td>
<td>36.88, 38.29, 15.29</td>
<td>36.56, 38.16, 15.39</td>
<td>33.85, 37.83, 16.19</td>
</tr>
<tr>
<td></td>
<td>4.5MB</td>
<td>30.42, 60.78, 13.66</td>
<td>30.14, 60.76, 13.77</td>
<td>26.97, 60.55, 14.55</td>
</tr>
<tr>
<td>linux-2.6.11.tar (199MB)</td>
<td>900KB</td>
<td>84.88, 24.93, 35.35</td>
<td>48.01, 24.69, 35.46</td>
<td>23.82, 22.17, 36.44</td>
</tr>
<tr>
<td></td>
<td>4.5MB</td>
<td>133.54, 45.66, 33.10</td>
<td>41.37, 44.02, 33.23</td>
<td>24.17, 39.88, 34.26</td>
</tr>
<tr>
<td></td>
<td>9MB</td>
<td>196.64, 53.59, 32.51</td>
<td>45.55, 51.77, 32.65</td>
<td>23.81, 32.11, 33.69</td>
</tr>
<tr>
<td>silesia.tar (203MB)</td>
<td>900KB</td>
<td>39.56, 29.85, 52.06</td>
<td>39.14, 29.69, 52.17</td>
<td>28.98, 29.32, 52.97</td>
</tr>
<tr>
<td></td>
<td>4.5MB</td>
<td>34.60, 39.57, 50.06</td>
<td>29.52, 39.63, 50.19</td>
<td>22.97, 32.67, 51.03</td>
</tr>
<tr>
<td></td>
<td>9MB</td>
<td>36.10, 46.73, 49.57</td>
<td>28.85, 46.92, 49.70</td>
<td>24.55, 46.31, 50.55</td>
</tr>
</tbody>
</table>

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).

Time for Merge. Figure 6 also shows that the time required for the overlapped CPU merge operation is smaller than the time of the GPU sort step and it is constant for a given input size, no matter what the dataset (indicated by a solid black line). This allows us to fully overlap the merge computation with the sort operation on GPU.

5.2. Results: Hybrid BWC

We also measure the runtime of our hybrid BWC (GPU performs the partial sorts and CPU performs the merge, MTF and Huffman in a fully overlapped manner as described in Section 3.4) against the state-of-the-art CPU BWC implementation in Bzip2 file compressor. Table I gives the total runtime and compressed file size on different datasets using different block sizes and with varying percentage of perturbation. Our hybrid BWC is fully compatible with the Bzip2 standard and for any given input it generates the same compressed file as the CPU Bzip2 (provided both algorithms use the same block size and 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 [Burrows and Wheeler 1994]. 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
Fig. 7: 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 also gives some gain in compression ratio.

of our hybrid BWC improves. Even for the worst-case linux dataset we beat the CPU with perturbation $\geq 0.01\%$ 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 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 7 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. This is a significant improvement over previous GPU implementation which was 2.78× slower and also slightly worse in compression ratio as compared to CPU [Patel et al. 2012]. 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 7). In practice, 0.1% perturbation allows us to improve the runtime for our hybrid BWC and still keeps the compressed file size below the state-of-the-art (obtained by CPU BWC on 900KB blocks). The gain $(2.9 - 8.4\%)$ in compression ratio is indicated by green line in Figure 7. In the results that follow, we fix our block size to 9MB and perturbation to 0.1%.

Decompression. In Figure 8 we measure the change in the runtime for BW decompression with different block size and % perturbation. We see that there is a marginal increase in runtime when the block size increases and no change with perturbation. Thus, our modifications do not have a drastic impact on the decompression runtime. We leverage the block-level parallelism present in BW decompression to gain a linear speedup with CoSts through our all-core framework.

5.3. 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).
Fig. 8: The decompression time (about 7s) is relatively small as compared to the compression time (about 30s) for wiki-xml dataset. The runtime increases slightly with increase in block size and % perturbation does not affect the decompression runtime.

**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, 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×), 16.4 (1.59×), 7.7s (1.28×) and 11.0s (1.58×) 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× for all the datasets compared to the about 3.24× 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 a combination of Intel Core2Duo E6750 CPU with Nvidia GTX 280 and Nvidia Quadro FX 3700 GPUs (Table III). These are less powerful GPUs with limited support for parallelism as compared to Nvidia GTX 580 and we observe a performance comparable or even worse than CPU for our hybrid BWC. Except for the linux dataset, we see that in both cases the all-core BWC improves upon the best speedup obtained by the multi-core BWC. The improvement is relatively more significant for the GTX 280, as it is a more powerful GPU compared to Quadro FX 3700. The best speedup obtained on the GTX 280 setup is 1.67×, on the Quadro FX 3700 we achieve 1.48×, while the multi-core BWC gives best speedup of 1.22×. Though the Nvidia Quadro FX 3700 is on average two times slower compared to the dual-core CPU, our pipeline is such that it effectively balances the load and still provides an overall speedup with full resource utilization. Also, to investigate the reasons for slowdown on linux dataset, we ran our code while keeping track of the times at which the items are dequeued from the work queue. It so happens that GTX 280 and some CPU threads dequeue the last remaining blocks around 35.2s. The CPU being faster terminates earlier.

Submitted to ACM Transactions on Parallel Computing.
Table II: All-Core and Multi-core BWC results on high-end CPU+GPU system

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Total time for all-core BWC (CPU+GPU) (s), (blocks processed by GPU / #total blocks)</th>
<th>Speedup (bold) vs. single CPU (underlined)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0 CPU 1 CPU 2 CPU 3 CPU 4 CPU 5 CPU 6 CPU 7 CPU</td>
<td></td>
</tr>
<tr>
<td>enwik8</td>
<td>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)</td>
<td>4.05</td>
</tr>
<tr>
<td>wiki-xml</td>
<td>27.8 (18/18) 23.6 (13/18) 19.8 (10/18) 18.4 (10/18) 16.0 (8/18) 15.9 (7/18) 18.8 (6/18)</td>
<td>4.87</td>
</tr>
<tr>
<td>linux</td>
<td>23.8 (22/22) 14.3 (13/22) 10.8 (8/22) 9.3 (8/22) 9.1 (7/22) 7.7 (7/22) 7.9 (6/22)</td>
<td>4.16</td>
</tr>
<tr>
<td>silesia</td>
<td>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)</td>
<td>4.20</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Total time for multi-core BWC (CPU only) (s) (No GPU involved)</th>
<th>Speedup (bold) vs. single CPU (underlined)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1 CPU 2 CPU 3 CPU 4 CPU 5 CPU 7 CPU 8 CPU</td>
<td></td>
</tr>
<tr>
<td>enwik8</td>
<td>15.8 9.1 6.4 5.0 5.0 4.9 4.9</td>
<td>3.22</td>
</tr>
<tr>
<td>wiki-xml</td>
<td>79.9 44.5 32.3 28.4 25.6 26.2 26.1</td>
<td>3.06</td>
</tr>
<tr>
<td>linux</td>
<td>32.1 17.6 13.3 10.7 10.0 9.9 3.24</td>
<td></td>
</tr>
<tr>
<td>silesia</td>
<td>46.3 30.9 21.5 21.4 21.1 17.4 18.4</td>
<td>2.66</td>
</tr>
</tbody>
</table>

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). We use 9MB blocks and 0.1% perturbation, best runtime for each dataset is indicated in bold. We achieve $3.06 \times$ speedup with multi-core BWC, which improves to $4.87 \times$ with our all-core BWC. 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.

leaving the GPU to be a bottleneck. There is always a possibility of last block going to the slowest CoSt and affecting the runtime. A possible and an interesting way to avoid this would be to learn the behavior of runtimes (during execution of initial work items) of all CoSts and avoid slower CoSts from dequeuing work items at the very end.

6. CONCLUSION

In this paper, we presented an all-core framework to exploit all computing resources 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 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 heterogeneous processing platforms. We expect the all-core framework to be useful for applications like video encoding and decoding, other data compression schemes, etc.

Submitted to ACM Transactions on Parallel Computing.
### Table III: All-Core and Multi-Core BWC results on low-end CPU+GPU system

#### NVIDIA Quadro FX 3700 (GPU) + Intel Core2Duo E6750 (CPU)

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Total time for all-core BWC (CPU+GPU) (s), (#blocks processed by GPU / #total blocks)</th>
<th>Best Speedup (bold) vs. single CPU (underlined)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 CPU</td>
<td>1 CPU</td>
<td>2 CPU</td>
</tr>
<tr>
<td>enwik8</td>
<td>35.2 (12/12)</td>
<td>22.5 (7/12)</td>
</tr>
<tr>
<td>wiki-xml</td>
<td>201.4 (18/18)</td>
<td>114.5 (8/18)</td>
</tr>
<tr>
<td>linux</td>
<td>176.5 (22/22)</td>
<td>65.1 (6/22)</td>
</tr>
<tr>
<td>silesia</td>
<td>164.5 (23/23)</td>
<td>66.4 (8/23)</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Total time for all-core BWC (CPU+GPU) (s), (#blocks processed by GPU / #total blocks)</th>
<th>Best Speedup (bold) vs. single CPU (underlined)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 CPU</td>
<td>1 CPU</td>
<td>2 CPU</td>
</tr>
<tr>
<td>enwik8</td>
<td>20.5 (12/12)</td>
<td>19.3 (6/12)</td>
</tr>
<tr>
<td>wiki-xml</td>
<td>91.6 (18/18)</td>
<td>77.8 (12/18)</td>
</tr>
<tr>
<td>linux</td>
<td>77.3 (22/22)</td>
<td>46.1 (11/22)</td>
</tr>
<tr>
<td>silesia</td>
<td>74.3 (23/23)</td>
<td>57.0 (14/23)</td>
</tr>
</tbody>
</table>

#### Only Intel Core2Duo E6750 (CPU)

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Total time for multi-core BWC (CPU only) (s) Different number of CPU threads (No GPU involved)</th>
<th>Best Speedup (bold) vs. single CPU (underlined)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 CPU</td>
<td>2 CPU</td>
<td>3 CPU</td>
</tr>
<tr>
<td>enwik8</td>
<td>26.4</td>
<td>22.4</td>
</tr>
<tr>
<td>wiki-xml</td>
<td>129.9</td>
<td>106.0</td>
</tr>
<tr>
<td>linux</td>
<td>47.6</td>
<td>36.2</td>
</tr>
<tr>
<td>silesia</td>
<td>69.59</td>
<td>55.26</td>
</tr>
</tbody>
</table>

For this table, we use a low-end Intel Core2Duo E6750 CPU, Nvidia Quadro FX 3700 (low-end) and Nvidia GTX 280 (medium-end) GPUs. The table shows runtimes for all-core BWC (CPU+GPU) and multi-core BWC (CPU only). We use 9MB blocks and 0.1% perturbation, best runtimes for each dataset are indicated in bold. Using all-core BWC on both these setups, allows us to improve on the speedup achieved by the multi-core BWC (except for linux dataset).

REFERENCES


Submitted to ACM Transactions on Parallel Computing.
Fast Burrows Wheeler Compression Using CPU and GPU


APPENDIX

GPU String Sort. The pseudo code of algorithm for GPU String sort [Deshpande and Narayanan 2013] is given in Algorithm 2. The algorithm exploits efficiency of radix sort while reducing data movement by sorting records of string bytes as the key and string index as the value (line 11). Each step uses a few bytes (k) of each string starting at a fixed offset from the left as the key; the offset is 0 for the first step (line 5). The fixed-length radix sort primitive from Thrust library is used in each step. Strings with a common prefix so far come in adjacent positions after sorting. Strings with unique prefixes will be singletons and would already be in their final place in the sorted output. These can be marked and removed from further processing (line 12-15). For the remaining strings, we assign bucket ids (stored in the BucketID array) beginning with 0 for the lexicographically smallest and increment by 1 whenever the next string differs. Common prefix strings are contiguous and get the same bucket ID. This ID assignment is performed using a scan primitive after marking in parallel the locations where adjacent sorted records have different keys (line 16). Strings belonging to a bucket will only shuffle among themselves without crossing buckets in the final output. The bucket ID, thus, encodes the history of the sorting steps till the current one. This allows the currently sorted prefixes to be discarded and we can load successive bytes in their place for further sorts (line 18). Further sorts are performed on records with a key consisting of: bucket id on the left and next few bytes from each string on
Algorithm 2 GPU String Sorting

1: Input: String Array $G$ // global string array, NULL delimits separate strings
2: Input: Index Array $I$ // starting indices of strings in global array
3: Output: Shuffled Index Array $O$.
4: $k$ = optimal key length // 8 for our platform
5: $M$ ← load_prefix($G, k, 0$) // load next prefix starting at offset
6: offset ← $k$ // load next prefix starting at offset
7: BucketID ← $[0, 0, \ldots, 0]$ // Only one bucket with size equal to input size
8: buckBytes ← compute_bytes(BucketID) // 0 initially
9: $K$ ← pack_keys($M$, BucketID, buckBytes)
10: repeat
11: radix_sort(key: $K$, value: $I$) // $F = \text{Flag}$, $O = \text{Output}$
12: $F$ ← mark_singletons($K$, $O$)
13: // above step also writes index of singletons to output
14: $D$ ← prefix_scan($F$) // $D = \text{Destination Array}$
15: $K, I$ ← scatter($K, I$, flag: $F$, dest : $D$) // compaction
16: BucketID ← generate_segments($K$)
17: buckBytes ← compute_bytes(BucketID)
18: $M$ ← load_prefix($G, k - buckBytes, offset$)
19: offset ← offset + $k - buckBytes$
20: $K$ ← pack_keys($M$, BucketID, buckBytes)
21: until no buckets left
22: Output : Shuffled Index Array $O$

the right and a value consisting of: the string pointers. The process of sorting more characters of the string is repeated until all strings become singletons (line 20).

Received April 2014; revised ; accepted