Parallel Lossless Data Compression on the GPU

Ritesh A. Patel
University of California, Davis
ritpatel@ucdavis.edu

Yao Zhang
University of California, Davis
yaozhang@ucdavis.edu

Jason Mak
University of California, Davis
jwmak@ucdavis.edu

Andrew Davidson
University of California, Davis
aaldavidson@ucdavis.edu

John D. Owens
University of California, Davis
jowens@ece.ucdavis.edu

ABSTRACT

We present parallel algorithms and implementations of a bzip2-like lossless data compression scheme for GPU architectures. Our approach parallelizes three main stages in the bzip2 compression pipeline: Burrows-Wheeler transform (BWT), move-to-front transform (MTF), and Huffman coding. In particular, we utilize a two-level hierarchical sort for BWT, design a novel scan-based parallel MTF algorithm, and implement a parallel reduction scheme to build the Huffman tree. For each algorithm, we perform detailed performance analysis, discuss its strengths and weaknesses, and suggest future directions for improvements. Overall, our GPU implementation is dominated by BWT performance and is 2.78× slower than bzip2, with BWT and MTF-Huffman respectively 2.89× and 1.34× slower on average.

1. INTRODUCTION

In this work, we implement parallel data compression on a GPU. We study the challenges of parallelizing compression algorithms, the potential performance gains offered by the GPU, and the limitations of the GPU that prevent optimal speedup. In addition, we see many practical motivations for this research. Applications that have large data storage demands but run in memory-limited environments can benefit from high-performance compression algorithms. The same is true for systems where performance is bottlenecked by data communication. These distributed systems, which include high-performance multi-node clusters, are willing to tolerate additional computation to minimize the data sent across bandwidth-saturated network connections. In both cases, a GPU can serve as a compression/decompression coprocessor operating asynchronously with the CPU to relieve the computational demands of the CPU and support efficient data compression with minimal overhead.

The GPU is a highly parallel architecture that is well-suited for large-scale file processing. In addition to providing massive parallelism with its numerous processors, the GPU benefits the performance of our parallel compression through its memory hierarchy. A large global memory residing on the device allows us to create an efficient implementation of a compression pipeline found in similar CPU-based applications, including bzip2. The data output by one stage of compression remains in GPU memory and serves as input to the next stage. To maintain high performance within the algorithms of each stage, we use fast GPU shared memory to cache partial file contents and associated data structures residing as a whole in slower global memory. Our algorithms also benefit from the atomic primitives provided in CUDA [12]. These atomic operations assist in the development of merge routines needed by our algorithms.

We implement our compression application as a series of stages shown in Figure 1. Our compression pipeline, similar to the one used by bzip2, executes the Burrows-Wheeler transform (BWT), the move-to-front transform (MTF), and ends with Huffman coding. The BWT inputs the text string to compress, then outputs an equally-sized BWT-transformed string; MTF outputs an array of indices; and the Huffman stage outputs the Huffman tree and the Huffman encoding of this list. The BWT and MTF apply reversible transformations to the file contents to increase the effectiveness of Huffman coding, which performs the actual compression. Each algorithm is described in greater detail in the next sections. This execution model is well suited to CUDA, where each stage is implemented as a set of CUDA kernels, and the output from kernels are provided as input to other kernels.

Although Gilchrist presents a multi-threaded version of bzip2 that uses the Pthreads library [6], the challenges of mapping bzip2 to the GPU lie not only in the implementation, but more fundamentally, in the algorithms. bzip2 is designed with little thought given to data locality and parallelism. The BWT implementation in bzip2 is based on radix sort and quicksort, which together require substantial global communication. We redesign BWT using a hierarchical merge sort that sorts locally and globally. The use of a local sort distributes the workload to the GPU cores and minimizes the global communication. The traditional MTF algorithm is strictly serial with a character-by-character dependency. We invent a novel scan-based MTF algorithm that breaks the input MTF list into small chunks and processes them in parallel. We also build the Huffman tree using a parallel reduction scheme.

We make the following contributions in the paper. First, we design parallel algorithms and GPU implementations for three major stages of bzip2: BWT, MTF, and Huffman coding. Second, we conduct a comprehensive performance analysis, which reveals the strengths and weaknesses of our par-

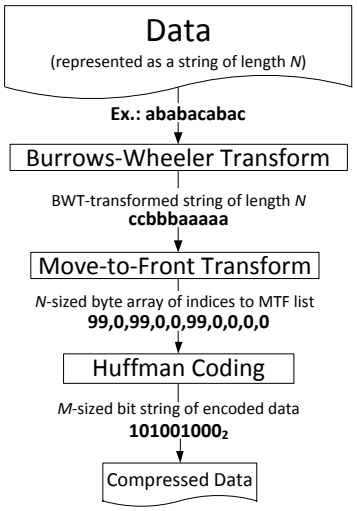

Figure 1: *Our compression pipeline consists of three stages: (1) Burrows-Wheeler Transform; (2) Move-to-Front Transform; (3) Huffman Coding.*

allel algorithms and implementations, and further suggests future directions for improvements. Third, our implementation enables the GPU to become a compression coprocessor, which lightens the processing burden of the CPU by using idle GPU cycles. We also assess the viability of using compression to trade computation for communication over the PCI-Express bus.

2. BURROWS-WHEELER TRANSFORM

The first stage in the compression pipeline, an algorithm introduced by Burrows and Wheeler [3], does not itself perform compression but applies a reversible reordering to a string of text to make it easier to compress. The Burrows-Wheeler transform (BWT) begins by producing a list of strings consisting of all cyclical rotations of the original string. This list is then sorted, and the last character of each rotation forms the transformed string. Figure 2 shows this process being applied to the string "ababacabac", where the cyclical rotations of the string are placed into rows of a block and sorted from top to bottom. The transformed string is formed by simply taking the last column of the block.

The new string produced by BWT tends to have many runs of repeated characters, which bodes well for compression. To explain why their algorithm works, Burrows and Wheeler use an example featuring the common English word "the", and assume an input string containing many instances of "the". When the list of rotations of the input is sorted, all the rotations starting with "he" will sort together, and a large proportion of them are likely to end in 't' [3]. In general, if the original string has several substrings that occur often, then the transformed string will have several places where a single character is repeated multiple times in a row. To be feasible for compression, BWT has another important property—reversibility. Burrows and Wheeler also describe the process to reconstruct the original string [3].

The main stage of BWT, the sorting of the rotated strings, is also the most computationally expensive stage. Sorting

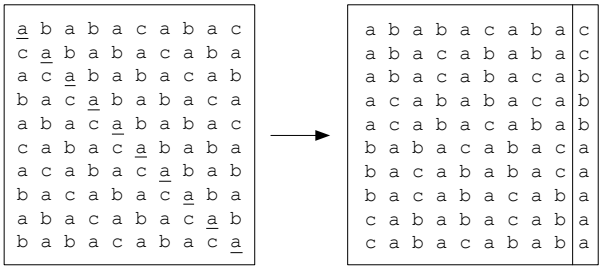

Figure 2: *BWT permutes the string "ababacabac" by sorting its cyclical rotations in a block. The last column produces the output "ccbbbaaaaa", which has runs of repeated characters.*

in BWT is a specialized case of string sort. Since the input strings are merely rotations of the original string, they all consist of the same characters and have the same length. Therefore, only the original string needs to be stored, while the rotations are represented by pointers or indices into a memory buffer.

The original serial implementation of BWT uses radix sort to sort the strings based on their first two characters followed by quicksort to further sort groups of strings that match at their first two characters [3]. Seward [18] shows the complexity of this algorithm to be $O(A \cdot n \log n)$, where A is the average number of symbols per rotation that must be compared to verify that the rotations are in order.

Sorting has been extensively studied on the GPU, but not in the context of variable-length keys (such as strings) and not in the context of very long, fixed-length keys (such as the million-character strings required for BWT). Two of the major challenges are the need to store the keys in global memory because of their length and the complexity and irregularity of a comparison function to compare two long strings.

2.1 BWT with Merge Sort

In our GPU implementation of BWT, we leverage a string sorting algorithm based on merge sort [5]. The algorithm uses a hierarchical approach to maximize parallelism during each stage. Starting at the most fine-grained level, each thread sorts 8 elements with bitonic sort. Next, the algorithm merges the sorted sequences within each block. In the last stage, a global inter-block merge is performed, in which b blocks are merged in $\log_2 b$ steps. In the later inter-block merges, only a small number of large unmerged sequences remain. These sequences are partitioned so that multiple blocks can participate in merging them.

Since we are sorting rotations of a single string, we use the first four characters of each rotation as the sort key. The corresponding value of each key is the index where the rotation begins in the input string. The input string is long and must be stored in global memory, while the sort keys are cached in registers and shared memory. When two keys are compared and found equal, a thread continually fetches the next four characters from global memory until the tie can be broken. Encountering many ties causes frequent access to global memory while incurring large thread divergence. This is especially the case for an input string with long runs of repeating characters. After sorting completes, we take

the last characters of each string to create the final output of BWT: a permutation of the input string that we store in global memory.

No previous GPU work implements the full BWT, primarily (we believe) because of the prior lack of any efficient GPU-based string sort. One implementation alternative is to substitute a simpler operation than BWT. The Schindler Transform (STX), instead of performing a full sort on the rotated input string, sorts strings by the first X characters. This is less effective than a full BWT, but is computationally much cheaper, and can be easily implemented with a radix sort [16].

3. MOVE-TO-FRONT TRANSFORM

The move-to-front transform (MTF) improves the effectiveness of entropy encoding algorithms [1], of which Huffman encoding is one of the most common. MTF takes advantage of repeated characters by keeping recently used characters at the front of the list. When applied to a string that has been transformed by BWT, MTF tends to output a new sequence of small numbers with more repeats.

MTF replaces each symbol in a stream of data with its corresponding position in a list of recently used symbols. Figure 3 shows the application of MTF to the string "ccbbbaaaaa" produced by BWT from an earlier example, where the list of recent symbols is initialized as the list of all ASCII characters. Although the output of MTF stores the indices of a list, it is essentially a byte array with the same size as the input string.

Algorithm 1 Serial Move-to-Front Transform

Input: A char-array *mtfIn*.
Output: A char-array *mtfOut*.

```
1: {Generate Initial MTF List}
2: for i = 0 → 255 do
3:     mtfList[i] = i
4: end for
5: for j = 0 → sizeof(mtfIn) − 1 do
6:     K = mtfIn[j]
7:     mtfOut[j] = K's position in mtfList
8:     Move K to front of mtfList
9: end for
```

Algorithm 1 shows pseudo-code for the serial MTF algorithm. At first glance, this algorithm appears to be completely serial; exploiting parallelism in MTF is a challenge because determining the index for a given symbol is dependent on the MTF list that results from processing all prior symbols. In our approach, we implement MTF as a parallel operation, one that can be expressed as an instance of the scan primitive [17]. We believe that this parallelization strategy is a new one for the MTF algorithm.

3.1 Parallel MTF Theory and Implementation

Each step in MTF encodes a single character by finding that character in the MTF list, recording its index, and then moving the character to the front of the list. The algorithm is thus apparently serial. We break this serial pattern with two insights:

1. Given a substring s of characters located somewhere within the input string, and without knowing anything about the rest of the string either before or following, we can generate a *partial MTF list* that computes the partial MTF for s that only contains the characters that appear in s (Algorithm 2).

2. We can efficiently combine two partial MTF lists for two adjacent substrings to create a partial MTF list that represents the concatenation of the two substrings (Algorithm 3).

We combine these two insights in our parallel divide-and-conquer implementation. We divide the input strings into small 64-character substrings and assign each substring to a thread. We then compute a partial MTF list for each substring per thread, then recursively merge those partial MTF lists together to form the final MTF list. We now take a closer look at the two algorithms.

Iteration	MTF List	Transformed String
cbbbaaaaa	…abc… (ASCII)	[99]
cbbbaaaaa	c…ab…	[99,0]
cbbbaaaaa	c…ab…	[99,0,99]
cbbbaaaaa	bc…a…	[99,0,99,0]
cbbbaaaaa	bc…a…	[99,0,99,0,0]
cbbbaaaaa	bc…a…	[99,0,99,0,0,99]
cbbbaaaaa	abc…	[99,0,99,0,0,99,0]
cbbbaaaaa	abc…	[99,0,99,0,0,99,0,0]
cbbbaaaaa	abc…	[99,0,99,0,0,99,0,0,0]
cbbbaaaaa	abc…	[99,0,99,0,0,99,0,0,0,0]

Figure 3: *The MTF transform is applied to a string produced by BWT, "cccbbbaaaaa". In the first step, the character 'c' is found at index 99 of the initial MTF list (the ASCII list). Therefore, the first value of the output is byte 99. The character 'c' is then moved to the front of the list and now has index 0. At the end of the transform, the resulting byte array has a high occurrence of 0's, which is beneficial for entropy encoding.*

Algorithm 2 MTF Per Thread

Input: A char-array *mtfIn*.
Output: A char-array *myMtfList*.

```
1: Index = 0
2: J = Number of elements per substring
3: for i = ((threadID + 1) × J) − 1 → threadID × J do
4:     mtfVal = mtfIn[i]
5:     if mtfVal does NOT exist in myMtfList then
6:         myMtfList[Index++] = mtfVal
7:     end if
8: end for
```

Algorithm 2 describes how we generate a partial MTF list for a substring s of length n. In this computation, we need only keep track of the characters in s. For example, the partial MTF list for "dead" is [d,a,e], and the partial list for "beef" is [f,e,b]. We note that the partial MTF list is simply the ordering of characters from most recently seen to least recently seen. Our implementation runs serially within each thread, walking s from its last element to its first element and recording the first appearance of each character in s.

Our per-thread implementation is SIMD-parallel so it runs efficiently across threads. Because all computation occurs internal to a thread, we benefit from fully utilizing the registers in each thread processor, and n should be as large as possible without exhausting the hardware resources within a thread processor. The output list has a maximum size of n or the number of possible unique characters in a string (256 for 8-bit byte-encoded strings), whichever is smaller. After each thread computes its initial MTF list, there will be N/n independent, partial MTF lists, where N is the length of our input string.

Algorithm 3 AppendUnique()

Input: Two MTF lists $List_n, List_{n-1}$.
Output: One MTF list $List_n$.

```
1: j = sizeof(List_n)
2: for i = 0 → sizeof(List_{n-1}) − 1 do
3:     K = List_{n-1}[i]
4:     if K does NOT exist in List_n then
5:         List_n[j++] = K
6:     end if
7: end for
```

To reduce two successive partial MTF lists into one partial MTF list, we use the *AppendUnique()* function shown in Algorithm 3. This function takes characters from the first list that are absent in the second list and appends them in order to the end of the second list. For example, applying *AppendUnique()* to the two lists [d,a,e] for the string "dead" and [f,e,b] for the string "beef" results in the list [f,e,b,d,a]. This new list orders the characters of the combined string "deadbeef" from most recently seen to least recently seen. Note that this merge operation only depends on the size of the partial lists (which never exceed 256 elements for byte-encoded strings), not the size of the substrings.

To optimize our scan-based MTF for the GPU, we adapt our implementation to the CUDA programming model. First, we perform a parallel reduction within each thread-block and store the resulting MTF lists in global memory. During this process, we store and access each thread's partial MTF list in shared memory. Next, we use the MTF lists resulting from the intra-block reductions to compute a global block-wise scan. The result for each block will be used by the threads of the subsequent block to compute their final outputs independent of threads in other blocks. In this final step, each thread-block executes scan using the initial, partial lists stored in shared memory and the lists outputted by the block-wise scan stored in global memory. With the scan complete, each thread replaces symbols in parallel using its partial MTF list outputted by scan, which is the state of the MTF list after processing prior characters. Figure 4 shows a high-level view of our algorithm.

MTF can be formally expressed as an exclusive scan computation, an operation well-suited for GPU architectures [17]. Scan is defined by three entities: an input datatype, a binary associative operation, and an identity element. Our datatype is a partial MTF list, our operator is the function *AppendUnique()*, and our identity is the initial MTF list (for a byte string, simply the 256-element array a where $a[i] = i$). It is likely that the persistent-thread-based GPU scan formulation of Merrill and Grimshaw [10] would be a good match for computing the MTF on the GPU.

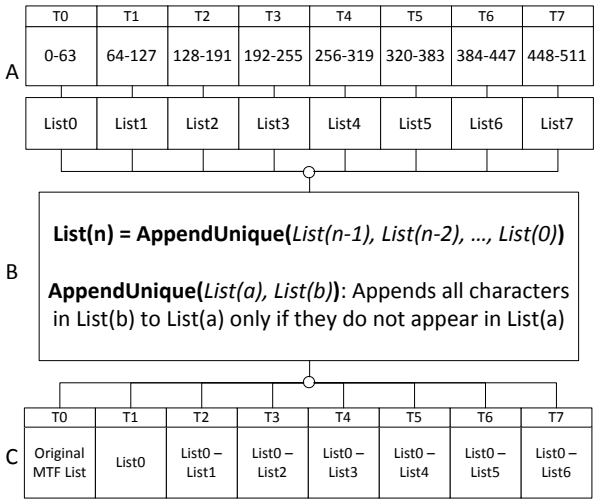

Figure 4: *Parallel MTF steps: A) Each thread computes a partial MTF list for 64 characters (Algorithm 2). B) An exclusive scan executes in parallel with AppendUnique() as the scan operator (Algorithm 3). C) The scan output for each thread holds the state of the MTF list after all prior characters have been processed. Each thread is able to independently compute a part of the MTF transform output.*

3.2 Reverse MTF Transform

The reverse MTF transform, shown in Algorithm 4, restores the original sequence by taking the first index from the encoded sequence and outputting the symbol found at that index in the initial MTF list. The symbol is moved to the front of the MTF list, and the process is repeated for all subsequent indices stored in the encoded sequence. Like MTF, reverse MTF appears to be a highly serial algorithm because we cannot restore an original symbol without knowing the MTF list that results from restoring all prior symbols. A similar scan-based approach can be used, in which the intermediate data is a MTF list and the operator is a permutation of a MTF list. We leave a parallel implementation of reverse MTF, as outlined in this section, for future work.

Algorithm 4 Serial Reverse Move-to-Front Transform

Input: A char-array *mtfRevIn*.
Output: A char-array *mtfRevOut*.

```
1: {Generate Initial MTF List}
2: for i = 0 → 255 do
3:     mtfList[i] = i
4: end for
5: for j = 0 → sizeof(mtfRevIn) − 1 do
6:     K = mtfRevIn[j]
7:     mtfRevOut[j] = mtfList[K]
8:     Move mtfList[K] to front of mtfList
9: end for
```

4. HUFFMAN CODING

Huffman coding is an entropy-encoding algorithm used in

data compression [7]. The algorithm replaces each input symbol with a variable-length bit code, where the length of the code is determined by the symbol's frequency of occurrence. There are three main computational stages in Huffman coding: (1) generating the character histogram, (2) building the Huffman tree, and (3) encoding data.

First, the algorithm generates a histogram that stores the frequency of each character in the data. Next, we build the Huffman tree as a binary tree from the bottom up using the histogram. The process works as follows. At each step, the two least-frequent entries are removed from the histogram and joined via a new parent node, which is inserted into the histogram with a frequency equal to the sum of the frequencies of its two children. This step is repeated until one entry remains in the histogram, which becomes the root of the Huffman tree.

Once the tree is built, each character is represented by a leaf in the tree, with less-frequent characters at a deeper level in the tree. The code for each character is determined by traversing the path from the root of the tree to the leaf representing that character. Starting with an empty code, a zero bit is appended to the code each time a left branch is taken, and a one bit is appended each time a right branch is taken. Characters that appear more frequently have shorter codes, and full optimality is achieved when the number of bits used for each character is proportional to the logarithm of the character's fraction of occurrence. The codes generated by Huffman coding are also known as "prefix codes", because the code for one symbol is never a prefix of a code representing any other symbol, thus avoiding ambiguity that can result from having variable-length codes. Finally, during the encoding process, each character is replaced with its assigned bit code.

In trying to parallelize the construction of the Huffman tree, we note that as nodes are added and joined, the histogram changes, so we cannot create nodes in parallel. Instead, we perform the search for the two least-frequent histogram entries in parallel using a reduction scheme. Parallelizing the encoding stage is difficult because each Huffman code is a variable-length bit code, so the location to write the code in the compressed data is dependent on all prior codes. In addition, variable-length bit codes can cross byte boundaries, which complicates partitioning the computation. To perform data encoding in parallel, we use an approach that encodes into variable-length, byte-aligned bit arrays.

The bzip2 application implements a modified form of Huffman coding that uses multiple Huffman tables [19]. The implementation creates 2 to 6 Huffman tables and selects the most optimal table for every block of 50 characters. This approach can potentially result in higher compression ratios. For simplicity and performance, we do not employ this technique in our implementation.

4.1 Histogram

To create our histogram, we use an algorithm presented by Brown et al. that builds 256-entry per-thread histograms in shared memory and then combines them to form a thread-block histogram [2]. We compute the final global histogram using one thread-block.

4.2 Huffman Tree

We use a parallel search algorithm to find the two least-frequent values in the histogram. Our algorithm, shown in Figure 5, can be expressed as a parallel reduction, where each thread searches four elements at a time. Our reduce operator is a function that selects the two least-frequent values out of four. Each search completes in $\log_2 n - 1$ stages, where n is the number of entries remaining in the histogram.

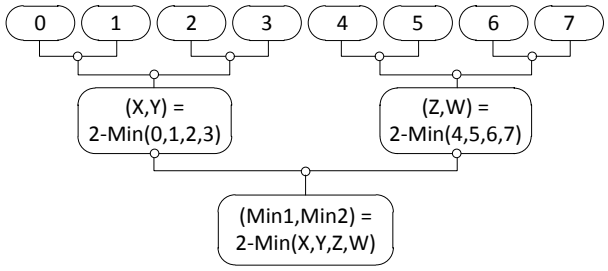

Figure 5: *We use a parallel reduction to find the two lowest occurrences. This technique is used in the building of the Huffman tree.*

4.3 Encoding

We derive our GPU implementation for Huffman encoding from a prior work by Cloud et al. [4], who describe an encoding scheme that packs Huffman codes into variable-length, byte-aligned bit arrays but do not give specific details for their implementation in CUDA. Our implementation works in the following way. We use 128 threads per thread-block and assign 4096 codes to each block, thereby giving each thread 32 codes to write in serial. Similar to Cloud et al., we byte-align the bit array written by each parallel processor, or each thread-block in our case, by padding the end with zeroes so that we do not have to handle codes crossing byte boundaries across blocks and so that we can later decode in parallel. We also save the size of each bit array for decoding purposes. To encode variable-length codes in parallel, each block needs the correct byte offset to write its bit array, and each thread within a block needs the correct bit offset to write its bit subarray (we handle codes crossing byte boundaries across threads). To calculate these offsets, we use a parallel scan across blocks and across threads within each block, where the scan operator sums bit array sizes for blocks and bit subarray sizes for threads. After encoding completes, our final output is the compressed data, which stores the Huffman tree, the sizes of the bit arrays, and the bit arrays themselves interleaved with small amounts of padding.

4.4 Decoding

Huffman decoding is a straightforward algorithm that processes one bit at a time, thus avoiding the bit-wise operations needed to handle codes across byte boundaries. To decode in serial, we traverse the Huffman tree, starting from the root. We take the left branch when a zero is encountered and the right branch when a one is encountered. When we reach a leaf, we write the uncompressed character represented by the leaf and restart our traversal at the root. To perform decoding in parallel on the GPU, we assign each bit array to a thread so that each thread is responsible for writing 4096 uncompressed characters. Although this simple approach uses less parallelism than our encoding implementation, we

do not determine this to be a performance issue because decoding, unlike encoding, does not require the extra scan operations and bitwise operations needed to support writing variable-length datatypes.

5. RESULTS

Experimental Setup.

Our test platform uses a 3.2 GHz Intel Core i5 CPU, an NVIDIA GTX 460 graphics card with 1 GB video memory, CUDA 4.0, and the Windows 7 operating system. In our comparisons with a single-core CPU implementation of bzip2, we only measure the relevant times in bzip2, such as the sort time during BWT, by modifying the bzip2 source code to eliminate I/O as a performance factor. We take a similar approach for MTF and Huffman encoding.

For our benchmarks, we use three different datasets to test large realistic inputs. The first of these datasets, `linux-2.6.11.1.tar` (203 MB), is a tarball containing source code for the Linux 2.6.11.1 kernel. The second, `enwik8` (97 MB), is a Wikipedia dump taken from 2006 [9]. The third, `enwiki-latest-abstract10.xml` (151 MB), is a Wikipedia database backup dump in the XML format [20].

bzip2 Comparison.

Table 1 displays our results for each dataset. bzip2 has an average compress rate of 7.72 MB/s over all three benchmarks, while our GPU implementation is 2.78× slower with a compress rate of 2.77 MB/s. Compared to our GPU implementation, bzip2 also yields better compression ratios because it uses additional compression algorithms such as run-length encoding and multiple Huffman tables [19]. In terms of similarities, our implementation and bzip2 both compress data in blocks of 1 MB. In addition, BWT contributes to the majority of the runtime in both implementations, with an average contribution of 91% to the GPU runtime and 81% to the bzip2 runtime. We find the average performance of MTF and Huffman coding on the GPU to be similar to that of bzip2. Our BWT and MTF+Huffman implementations are respectively 2.89× and 1.34× slower than bzip2 on average.

Memory Bandwidth.

One motivation for this work is to determine whether on-the-fly compression is suitable for optimizing data transfers between CPU and GPU. To answer this question, we analyze data compression rates, PCI-Express bandwidth, and GPU memory bandwidth. Given the numbers of the latter two, we compute the budget for the compression rate. Figure 6 shows the various compression rates required in order for on-the-fly compression to be feasible. These rates range from 1 GB/s (PCIe BW = 1 GB/s, Compress Ratio[1] = 0.01) to 32 GB/s (PCIe BW = 16 GB/s, Compress Ratio = 0.50). Our implementation is not fast enough, even at peak compression rates, to advocate "compress then send" as a more optimal strategy than "send uncompressed". Assuming a memory bandwidth of 90 GB/s and a compression rate of 50%, the required compression rate for encoding is 15 GB/s according to Figure 6, which means the total memory access allowed per character is fewer than 90/15 = 6 bytes. A

[1]Compress Ratio = $\frac{Compressed\ size}{Uncompressed\ size}$

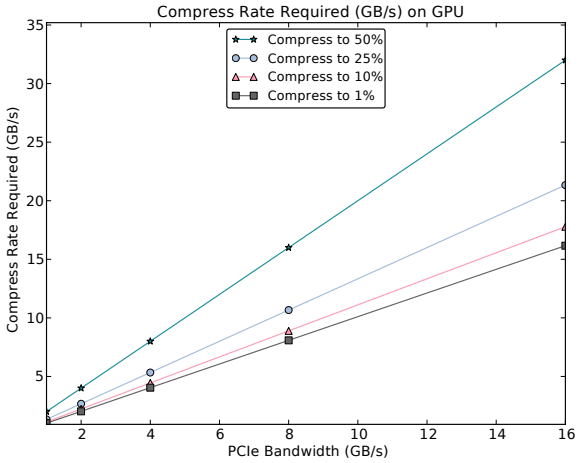

Figure 6: *The required compression rate (encode only) to make "compress then send" worthwhile as a function of PCI-Express bandwidth.*

rough analysis of our implementation shows that our BWT, MTF, and Huffman coding combined require 186 bytes of memory access per character.

However, we believe that technology trends will help make compress-then-send more attractive over time. Figure 7 shows the past and current trends for bus and GPU memory bandwidth. Global memory (DRAM) bandwidth on GPUs is increasing more quickly than PCIe bus bandwidth, so applications that prefer doing more work on the GPU rather using an external bus are aligned with these trends.

Analysis of Kernel Performance.

Our BWT implementation first groups the input string into blocks, then sorts the rotated strings within each block, and finally merges the sorted strings. We sort a string of 1 million characters by first dividing the string into 1024 blocks with 1024 strings assigned to each block. The local block-sort stage on the GPU is very efficient and takes only 5% of the total BWT time, due to little global communication and the abundant parallelism of divided string blocks. The rest of the string sort time is spent in the merging stages, where we merge the 1024 sorted blocks in $\log_2 1024 = 10$ steps. The merge sort is able to utilize all GPU cores by dividing the work up into multiple blocks as discussed in section 2.1. However, the number of ties that need to be broken results in poor performance. For each tie, a thread must fetch an additional 8 bytes (4 bytes/string) from global memory, leading to high thread divergence.

We analyze the number of matching prefix characters, or tie lengths, between strings for each pair of strings during BWT. Figure 8 shows that there are a higher fraction of string pairs with long tie lengths in BWT than there are in a typical string-sort dataset of random words. This is the same set of words used as the primary benchmark in the work by Davidson et al. [5], which achieves a sort rate of 30 Mstrings/s on the same GPU.

Figure 9 shows the performance comparison between the serial MTF and our parallel scan-based MTF. The runtime of the parallel MTF is the time it takes to compute

Table 1: *GPU vs. bzip2 Compression.*

File (Size)	Compress Rate (MB/s)	BWT Sort Rate (Mstring/s)	MTF+Huffman Rate (MB/s)	Compress Ratio ($\frac{Compressed\ size}{Uncompressed\ size}$)
enwik8 (97 MB)	GPU: 7.37 bzip2: 10.26	GPU: 9.84 bzip2: 14.2	GPU: 29.4 bzip2: 33.1	GPU: 0.33 bzip2: 0.29
linux-2.6.11.1.tar (203 MB)	GPU: 4.25 bzip2: 9.8	GPU: 4.71 bzip2: 12.2	GPU: 44.3 bzip2: 48.8	GPU: 0.24 bzip2: 0.18
enwiki-latest-abstract10.xml (151 MB)	GPU: 1.42 bzip2: 5.3	GPU: 1.49 bzip2: 5.4	GPU: 32.6 bzip2: 69.2	GPU: 0.19 bzip2: 0.10

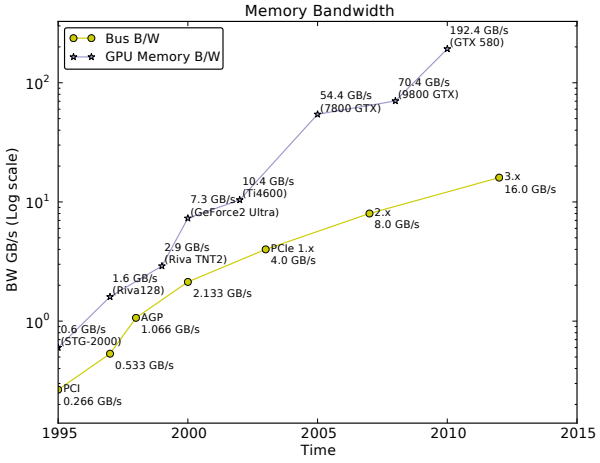

Figure 7: *Available PCI-Express and GPU memory bandwidth over time. The rate of GPU global memory bandwidth is growing at a much higher rate than bus bandwidth.*

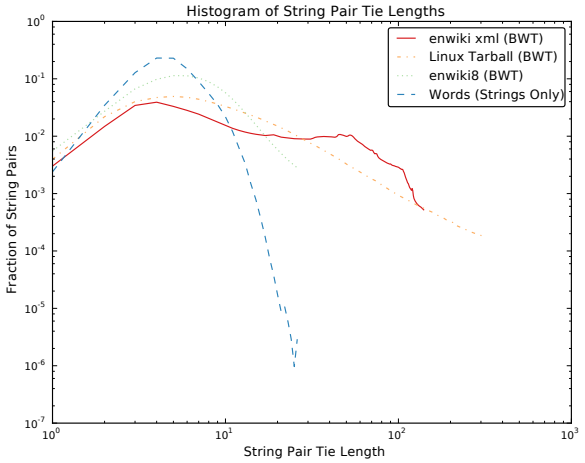

Figure 8: *Datasets that exhibit higher compress ratios also have a higher fraction of string pairs with long tie lengths. Sorting long rotated strings in BWT yields substantially more ties than other string-sorting applications (e.g. "words"). These frequent ties lead to lower sort rates on these datasets (3–20× slower), as shown in Table 1.*

individual MTF lists plus the time it takes for all threads to compute the MTF output using these lists. The computation of the output requires each thread to replace 64 characters in serial and contributes most to the total parallel MTF runtime. Initially, we stored both the entire 256-byte MTF lists and the input lists in shared memory. Due to excessive shared memory usage, we could only fit 4 thread-blocks, or 4 warps (with a block size of 32 threads) per multiprocessor. This resulted in a low occupancy of 4 warps/48 warps=8.3% and correspondingly low instruction and shared memory throughput. To improve the occupancy, we changed our implementation to only store partial MTF lists in shared memory. This allows us to fit more and larger thread-blocks on one multiprocessor, which improves the occupancy to 33%. We varied the size of partial lists and found the sweet spot that saves shared memory for higher occupancy and at the same time reduces global memory accesses. Figure 10 shows the MTF local scan performance as a function of partial list size in shared memory. The optimal points settle around 40 characters per list for all three datasets, which is the number we use in our implementation.

Figure 11 shows the runtime breakdown of Huffman coding. For all three datasets, building the Huffman tree contributes to approximately 80% of the total time. The lack of parallelism is a major performance limiting factor during Huffman tree construction. Since we build the Huffman tree from a 256-bin histogram, we have at most 128-way parallelism. As a result, we only run a single thread-block of 128 threads on the entire GPU, which leads to poor performance.

6. DISCUSSION

The lossless data compression algorithms discussed in this paper highlight some of the challenges faced in compressing generic data on a massively parallel architecture. Our results show that the compression of generic data on the GPU for the purpose of minimizing bus transfer time is far from being a viable option; however, many domain-specific compression techniques on the GPU have proven to be beneficial [13, 14, 21] and may be a better option.

Of course, a move to single-chip heterogeneous architectures will reduce any bus transfer cost substantially. However, even on discrete GPUs, we still see numerous uses for our implementation. In an application scenario where the GPU is idle, the GPU can be used to offload compression tasks, essentially as a compression coprocessor. For applications with constrained memory space, compression may be worthwhile even at a high computational cost. For

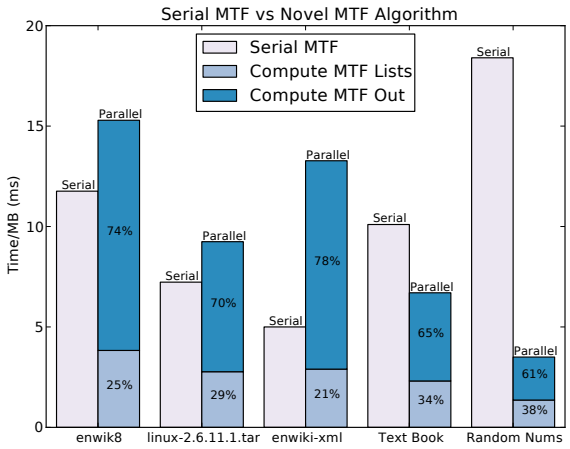

Figure 9: *Serial MTF vs. parallel scan-based MTF with five datasets: enwik8, linux tarball, enwiki xml, textbook, and random numbers (1-9).*

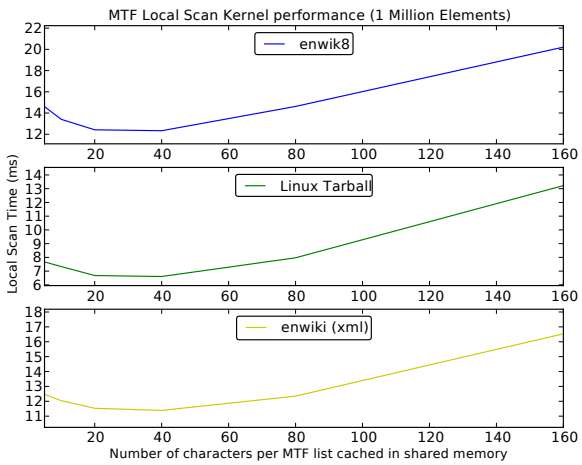

Figure 10: *MTF local scan performance as a function of partial list size in shared memory. The performance is shown for three datasets: enwik8, linux tarball, and enwiki xml.*

distributed/networked supercomputers, the cost of sending data over interconnect is prohibitively high, higher than a traditional bus, and compression is attractive in those scenarios. Finally, from a power perspective, the cost of computation falls over time compared to communication, so trading computation for communication is sensible to reduce overall system power in computers of any size.

Bottlenecks.

The string sort in our BWT stage is the major bottleneck of our compression pipeline. Sorting algorithms on the GPU have been a popular topic of research in recent years [8, 11, 15]. The fastest known GPU-based radix-sort by Merrill and Grimshaw [11] sorts key-value pairs at a rate of 3.3 GB/s (GTX 480). String sorting, however, is to the best of our knowledge a new topic on the GPU, and our implementation was based on the fastest available GPU-based string sort [5]. Table 1 shows that we achieve sort rates between 1.49 Mstrings/s to 9.84 Mstrings/s when sorting strings with lengths of 1 million characters. Datasets with higher compression ratios have lower sort rates and hence lower throughput: our "string-ties" analysis (Figure 8) shows that the highest compressed file in our dataset (`enwiki-latest-abstract10.xml`) has a much higher fraction of sorted string pairs with longer tie lengths than our least compressed dataset (`enwik8`), which in turn has more and longer ties than Davidson et al.'s "words" dataset. Additionally, to separate out the cost of ties from the cost of sorting, we computed the BWT on a string where every set of four characters were random and unique. Using this string we achieved a sort rate of 52.4 Mstrings/s, 5× faster than our best performing dataset (`enwik8`) and 37× faster than our worst performing dataset (`enwiki-latest-abstract10.xml`). It is clear that our performance would benefit most of all from a sort that is optimized to handle string comparisons with many and lengthy ties.

The performance of our parallel MTF algorithm compared to a CPU implementation varies depending on the dataset. On our three datasets, we have seen both a slowdown on the GPU as well as a 5× speedup. The performance of our MTF transform diminishes as the number of unique characters in the dataset increases. Nearly 72% of the runtime in our MTF algorithm is spent in the final stage, where each thread is able to independently compute a part of the the final MTF transform. Currently we break up the input MTF list into small lists of a fixed size. Our experiments show that the list size greatly affects the runtime distribution of MTF algorithmic stages, and the optimal list size is data-dependent. To address this problem, we hope to employ adaptive techniques in our future work.

The bottleneck of the Huffman encoding stage is the Huffman tree building, which can exploit at most 128-way parallelism. Building the Huffman tree contributes to ∼78% of the Huffman runtime, while the histogram and encoding steps contribute to ∼16% and ∼6% of the runtime, respectively. Further performance improvement of the Huffman tree building requires a more parallel algorithm that can fully utilize all GPU cores.

Overall, we are not able to achieve a speedup over bzip2. More important than the comparison, though, is the fact that the required compression rate (1 GB/s to 32 GB/s) for compress-then-send over PCIe to be worthwhile is much higher than that of bzip2 (5.3 MB/s to 10.2 MB/s). Our implementation needs to greatly reduce aggregate global memory bandwidth to approach this goal, and also develop more efficient fine-grained parallel algorithms for all steps in the pipeline. We believe that our efforts in this direction, and our identification and analysis of the performance bottlenecks of our parallel algorithms and implementations, will enable us and others to develop and refine further efforts toward lossless compression techniques. bzip2 was certainly not designed with an eye toward parallelism, so we believe that further research toward parallel-friendly algorithms for compression is an important future direction.

7. CONCLUSION

Our results in this paper suggest that our implementation on current GPU architectures is not capable of provid-

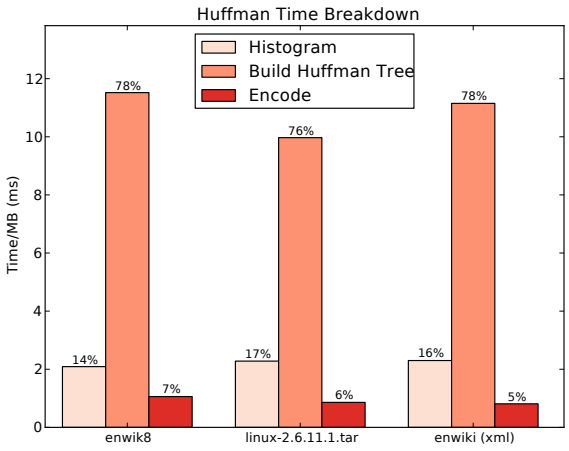

Figure 11: *The runtime breakdown of Huffman coding. The majority of the time is spent during the tree building stage due to a lack of parallelism discussed in section 5.*

ing enough performance benefits for on-the-fly lossless data compression. Compression rates must be at least 1 GB/s for compress-then-send to be a viable option. Compared to serial bzip2, our overall performance is currently 2.78× slower, but our implementation enables the GPU to become a compression coprocessor, and we see opportunities for significant improvements going forward.

Our immediate future directions for this work include: (1) finding a way to mitigate ties by fetching larger (more than four-byte) blocks; (2) autotuning the list size and stage transition points for MTF; (3) developing a more parallel algorithm for the Huffman tree building; (4) overlapping GPU compression and PCI-Express data transfer; and (5) alternate approaches to string sort that are well-suited for BWT-like workloads.

Acknowledgments

Many thanks to Adam Herr and Shubhabrata Sengupta of Intel, and Anjul Patney and Stanley Tzeng of UC Davis. Their insights and expertise were invaluable to the completion of this project. We appreciate the support of the HP Labs Innovation Research Program and the National Science Foundation (grants OCI-1032859 and CCF-1017399).

References

[1] Jon Louis Bentley, Daniel D. Sleator, Robert E. Tarjan, and Victor K. Wei. A locally adaptive data compression scheme. *Communications of the ACM*, 29:320–330, April 1986.

[2] Shawn Brown and Jack Snoeyink. Modestly faster GPU histograms. In *Proceedings of Innovative Parallel Computing (InPar '12)*, May 2012.

[3] M. Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithm. Technical Report SRC-RR-124, Digital Equipment Corporation, 10 May 1994.

[4] Robert L. Cloud, Matthew L. Curry, H. Lee Ward, Anthony Skjellum, and Purushotham Bangalore. Accelerating lossless data compression with GPUs. *CoRR*, arXiv:abs/1107.1525v1, 21 June 2011.

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

[6] Jeff Gilchrist. Parallel BZIP2 data compression software, 2003. `http://compression.ca/pbzip2/`.

[7] David A. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the Institute of Radio Engineers*, 40(9):1098–1101, September 1952.

[8] Peter Kipfer and Rüdiger Westermann. Improved GPU sorting. In Matt Pharr, editor, *GPU Gems 2*, chapter 46, pages 733–746. Addison Wesley, March 2005.

[9] Matt Mahoney. Data compression programs, 2006. `http://mattmahoney.net/dc`.

[10] Duane Merrill and Andrew Grimshaw. Parallel scan for stream architectures. Technical Report CS2009-14, Department of Computer Science, University of Virginia, December 2009.

[11] Duane Merrill and Andrew Grimshaw. Revisiting sorting for GPGPU stream architectures. Technical Report CS2010-03, Department of Computer Science, University of Virginia, February 2010.

[12] NVIDIA Corporation. NVIDIA CUDA compute unified device architecture, programming guide, 2011. `http://developer.nvidia.com/`.

[13] Molly A. O'Neil and Martin Burtscher. Floating-point data compression at 75 Gb/s on a GPU. In *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units*, GPGPU-4, pages 7:1–7:7, March 2011.

[14] Anton Pereberin. Hierarchical approach for texture compression. In *Proceedings of Graphicon*, pages 195–199, August 1999.

[15] Nadathur Satish, Mark Harris, and Michael Garland. Designing efficient sorting algorithms for manycore GPUs. In *Proceedings of the 23rd IEEE International Parallel and Distributed Processing Symposium*, May 2009.

[16] Michael Schindler. A fast block-sorting algorithm for lossless data compression. In *Proceedings of the Conference on Data Compression*, page 469, March 1997.

[17] Shubhabrata Sengupta, Mark Harris, Yao Zhang, and John D. Owens. Scan primitives for GPU computing. In *Graphics Hardware 2007*, pages 97–106, August 2007.

[18] Julian Seward. On the performance of BWT sorting algorithms. In *Proceedings of the Data Compression Conference*, pages 173–182, March 2000.

[19] Julian Seward. The bzip2 and libbzip2 official home page, 2002. `http://www.bzip.org`.

[20] Wikipedia. Wikimedia downloads, 2010. `http://dumps.wikimedia.org`.

[21] Givon Zirkind. AFIS data compression: An example of how domain specific compression algorithms can produce very high compression ratios. *ACM SIGSOFT Software Engineering Notes*, 32:1–26, November 2007.