Lossless Data Compression on GPUs

Ritesh Patel
University of California, Davis

Jason Mak
University of California, Davis
Motivation

- Data storage

- Computation vs. Data Transfer
  - Is compress-then-send worthwhile?

- GPU
  - 100 MB
  - Less transfer time
  - More computation

- CPU
  - 20 MB
PCI-E vs. GPU bandwidth

Bandwidth (GB/s)

- GPU Global Memory Bandwidth
- PCI-Express Bandwidth

Year:
- 1995
- 1999
- 2003
- 2007
- 2011
Topics Covered

• Related Work
• Three Algorithms in the Compression Pipeline
  1. Burrows-Wheeler Transform
  2. Move-to-Front Transform
  3. Huffman Coding
• Results
• Future Work
Related Work

• Domain-specific compression
  o Floating-point data compression
  o Texture compression

• Parallel bzip2 (pbzip2)
  o Uses pthread library
  o bzip2 not data-parallel-friendly
Compression Pipeline

Input Data: \textbf{n characters}

\begin{itemize}
  \item \textbf{Burrows-Wheeler Transform}
  \begin{itemize}
    \item n characters \rightarrow Many runs of repeated characters
  \end{itemize}
  \item \textbf{Move-to-Front Transform}
  \begin{itemize}
    \item n characters \rightarrow Low entropy \rightarrow Lots of 0s, 1s, etc.
  \end{itemize}
  \item \textbf{Huffman Coding}
\end{itemize}

Output Data: \textbf{compressed}
Burrows-Wheeler Transform

- Transforms a string and gives has many runs of repeated characters
- Same characters get grouped
Burrows-Wheeler Transform

- Transforms a string and gives has many runs of repeated characters
- Same characters get grouped

Input: `ababacabac` → Output: `ccbbbbaaaaaa`
Burrows-Wheeler Transform

BWT Input: ababacabac

1) Cyclical Rotations

```
ababacabac
ababacabac
```

Input Data

Move-to-Front Transform

Huffman Coding

Output Data: compressed
Burrows-Wheeler Transform

1) Cyclical Rotations

BWT Input: ababacabac

2) Sort

Input Data

Move-to-Front Transform

Huffman Coding

Output Data: compressed
Burrows-Wheeler Transform

Sorted Cyclical Rotations

a b a b a c a b a c
a b a c a b a b a c
a b a c a b a c a b
b a b a c a b a c a
b a c a b a b a c a
b a c a b a c a b a
b a c a b a c a b a
b a c a b a c a b a

Index: 0
BWT: ccbbbaaaaa
BWT with Merge Sort

• Sorting $n$ strings $\rightarrow$ each $n$ characters long
  o $n = 1$ million per block in our implementation
• Radix sort not a good fit
• Break ties on the fly
BWT with Merge Sort

- Sorting \( n \) strings → each \( n \) characters long
  - \( n = 1 \) million per block in our implementation
- Radix sort not a good fit
- Break ties on the fly
- String sorting on GPU
  - Non-uniform

Input Data

| Pair 1: 7 ties | a | b | a | b | a | c | a | b |
| Pair 1: 7 ties | b | a | c | a | b | a | b | c |
| Pair 2: 0 ties | a | b | a | c | a | b | a | b |
| Pair 2: 0 ties | a | c | a | b | a | c | a | b |
| Pair 3: 4 ties | a | c | a | b | a | c | a | b |
| Pair 3: 4 ties | b | a | c | a | b | a | c | a |

Output Data: compressed
BWT with Merge Sort

- Sorting $n$ strings $\rightarrow$ each $n$ characters long
  - $n = 1$ million per block in our implementation
- Radix sort not a good fit
- Break ties on the fly
- String sorting on GPU
  - Non-uniform
  - Parallelize BWT with string sorting algorithm based on merge sort
  - Currently fastest string sort on GPU
Move-to-Front Transform

- Exploit clumpy characters from BWT
  - BWT: ababacabac → ccbbaaaaaa
- Improves effectiveness of entropy encoding
Move-to-Front Transform

- Each symbol in the data is replaced by its index in the list
  - Initial list: ...abcd... (ASCII)
- Recently seen characters are kept at front of the list
  - Long sequences of identical symbols replaced by many zeros

ccbbbaaaaa → Move-to-Front Transform → 99, 0, 99, 0, 0, 99, 0, 0, 0, 0
## Move-to-Front Transform

<table>
<thead>
<tr>
<th>Input</th>
<th>MTF List</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>ccbbaaaaa</td>
<td>...abc... (ASCII)</td>
<td>[99]</td>
</tr>
</tbody>
</table>

- ‘c’ occurs at 99\textsuperscript{th} index $\rightarrow$ Output ‘99’
Move-to-Front Transform

- ‘c’ occurs at 99\textsuperscript{th} index $\rightarrow$ Output ‘99’
- Move ‘c’ to front of the MTF list
- Shift all previous elements to the right
## Move-to-Front Transform

<table>
<thead>
<tr>
<th>Input</th>
<th>MTF List</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>ccbbbaaaaa</td>
<td>...abc... (ASCII)</td>
<td>[99]</td>
</tr>
<tr>
<td>ccbbbaaaaa</td>
<td>c...ab...</td>
<td>[99, 0]</td>
</tr>
</tbody>
</table>

- ‘c’ occurs at 0\(^{th}\) index $\Rightarrow$ Output ‘0’
## Move-to-Front Transform

<table>
<thead>
<tr>
<th>Input</th>
<th>MTF List</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>ccbbbaaaaa</td>
<td>...abc... (ASCII)</td>
<td>[99]</td>
</tr>
<tr>
<td>ccbbbaaaaa</td>
<td>c...ab...</td>
<td>[99, 0]</td>
</tr>
</tbody>
</table>

- ‘c’ occurs at 0<sup>th</sup> index → Output ‘0’
- ‘c’ already at front, no shifts
Move-to-Front Transform

<table>
<thead>
<tr>
<th>Input</th>
<th>MTF List</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>cccbbbaaaaaa</td>
<td>...abc... (ASCII)</td>
<td>[99]</td>
</tr>
<tr>
<td>cccbbbaaaaaa</td>
<td>c...ab...</td>
<td>[99, 0]</td>
</tr>
</tbody>
</table>

- ‘c’ occurs at 0th index → Output ‘0’
- ‘c’ already at front, no shifts
- Repeat for all elements
## Move-to-Front Transform

<table>
<thead>
<tr>
<th>Iteration</th>
<th>MTF List</th>
<th>Transformed String</th>
</tr>
</thead>
<tbody>
<tr>
<td>cccbbbaaaaa</td>
<td>...abc... (ASCII)</td>
<td>[99]</td>
</tr>
<tr>
<td>cccbbbaaaaa</td>
<td>c...ab...</td>
<td>[99, 0]</td>
</tr>
<tr>
<td>cccbbbbaaaa</td>
<td>c...ab...</td>
<td>[99, 0, 99]</td>
</tr>
<tr>
<td>cccbbbaaaaa</td>
<td>bc...a...</td>
<td>[99, 0, 99, 0]</td>
</tr>
<tr>
<td>cccbbbaaaaa</td>
<td>bc...a...</td>
<td>[99, 0, 99, 0, 0]</td>
</tr>
<tr>
<td>cccbbbaaaaa</td>
<td>bc...a...</td>
<td>[99, 0, 99, 0, 0, 99]</td>
</tr>
<tr>
<td>cccbbbaaaaa</td>
<td>abc...</td>
<td>[99, 0, 99, 0, 0, 0]</td>
</tr>
<tr>
<td>cccbbbaaaa</td>
<td>abc...</td>
<td>[99, 0, 99, 0, 0, 0, 0]</td>
</tr>
<tr>
<td>cccbbbaaaaa</td>
<td>abc...</td>
<td>[99, 0, 99, 0, 0, 0, 0, 0]</td>
</tr>
</tbody>
</table>

**Diagram:**

- **Input Data**: cccbbbaaaaa
- **Burrows-Wheeler Transform**: cccbbbaaaaa -> [99, 0, 99, 0, 0, 0, 0, 0]
- **Move-to-Front Transform**: [99, 0, 99, 0, 0, 0, 0, 0] -> [99, 0, 99, 0, 0, 0, 0, 0, 0]
- **Huffman Coding**: [99, 0, 99, 0, 0, 0, 0, 0, 0, 0] -> compressed output

**Output Data:** compressed
Parallel MTF

- MTF appears to be highly serial
  - Character-by-character dependency
Parallel MTF

- MTF appears to be highly serial
  - Character-by-character dependency
- Goal: generate multiple MTF lists at different indices
  - Compute MTF output in parallel
Parallel MTF

- MTF appears to be highly serial
  - Character-by-character dependency
- Goal: generate multiple MTF lists at different indices
  - Compute MTF output in parallel
- How to parallelize?
  - Two key insights
    1. Generate partial MTF list of a substring $s$
    2. Combine two adjacent partial MTF lists (using “concatenation”)
Example Parallel MTF

Input Data

Input Data

Move-to-Front Transform

Move-to-Front Transform

Huffman Coding

Huffman Coding

Output Data: compressed

Burrows-Wheeler Transform

Burrows-Wheeler Transform

Output Data: compressed

B A D A D D D A C C C B B A A A
Example Parallel MTF

Input Data

Burrows-Wheeler Transform

Move-to-Front Transform

Huffman Coding

Output Data: compressed

Part 1

MTF List

B A D A D D D A C C C B B A A A

End of part 1

A D B
Example Parallel MTF

Part 1

B A D A D D D A

Part 2

C C C B B A A A

Final MTF List

A B C D

Input Data

Burrows-Wheeler Transform

Move-to-Front Transform

Huffman Coding

Output Data: compressed
Example Parallel MTF

Break input into 2 chunks

2 partial MTF lists (in parallel)

Input Data

List 1

List 2

A D B

A B C

B A D A D D D A

C C C B B A A A A

Burrows-Wheeler Transform

Move-to-Front Transform

Huffman Coding

Output Data: compressed
Example Parallel MTF

2 partial MTF lists (in parallel)

Appends all unique symbols from list 1 to list 2
Example Parallel MTF

2 partial MTF lists (in parallel)

Same MTF list as we had before!

Input Data

Burrows-Wheeler Transform

Move-to-Front Transform

Huffman Coding

Output Data: compressed
Example Parallel MTF

Break input into 4 chunks

Input Data

Burrows-Wheeler Transform

Move-to-Front Transform

Huffman Coding

Output Data: compressed

B A D A D D D A C C C B B A A A

B A D A

D D D A

C C C B

B A A A
Example Parallel MTF

Break input into 4 chunks

Generate partial MTF lists

Input Data

Burrows-Wheeler Transform

Move-to-Front Transform

Huffman Coding

Output Data: compressed
Example Parallel MTF

MTF list at index 4

Operator → Appends all unique symbols to current list
Example Parallel MTF

Operator → Appends all unique symbols to current list
Example Parallel MTF

![Diagram showing parallel MTF process]

- **Input Data**: A D B
- **Output Data**: B C A D

Operator → Appends all unique symbols to current list

MTF list at index 12
Example Parallel MTF

Operator \(\rightarrow\) Appends all unique symbols to current list

MTF list at index 16

Input Data

Burrows-Wheeler Transform

Move-to-Front Transform

Huffman Coding

Output Data: compressed
Example Parallel MTF

Generated lists at indices: 4, 8, 12, 16

Operator → Appends all unique symbols to current list
Example Parallel MTF

Operator → Appends all unique symbols to current list

Input Data

Burrows-Wheeler Transform

Move-to-Front Transform

Huffman Coding

Output Data: compressed
Example Parallel MTF

Operator → Appends all unique symbols to current list
Example Parallel MTF

- Operator → Appends all unique symbols to current list

- Input Data
  - 41

- Output Data: compressed

- A D B
- A D
- B C
- A B

- A D B
- A B C
- A B C D

- Move-to-Front Transform
- Burrows-Wheeler Transform
- Huffman Coding
Example Parallel MTF

Generated lists at indices: 4, 8, 12, 16

Input Data

Burrows-Wheeler Transform

Move-to-Front Transform

Huffman Coding

Output Data: compressed
Example Parallel MTF

MTF Lists

Input Data

Output Data: compressed

(Initial MTF List)
Example Parallel MTF

Input Data

Burrows-Wheeler Transform

Move-to-Front Transform

Huffman Coding

Output Data: compressed

---

**Initial MTF List**

<table>
<thead>
<tr>
<th>In</th>
<th>MTF Lists</th>
</tr>
</thead>
<tbody>
<tr>
<td>B A D A</td>
<td>A B C...</td>
</tr>
<tr>
<td>D D D A</td>
<td>A D B...</td>
</tr>
<tr>
<td>C C C B</td>
<td>A D B...</td>
</tr>
<tr>
<td>B A A A</td>
<td>B C A D...</td>
</tr>
</tbody>
</table>

MTF Out

<table>
<thead>
<tr>
<th>0</th>
<th>4</th>
<th>8</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>[1,1,3,1]</td>
<td>[1,0,0,1]</td>
<td>[3,0,0,3]</td>
<td>[0,2,0,0]</td>
</tr>
</tbody>
</table>
So far...

Input

ababacabac

Burrows-Wheeler Transform

Grouped characters

ccbbbaaaaa

Move-to-Front Transform

Lots of 0s

99, 0, 99, 0, 0, 99, 0, 0, 0, 0

Huffman Coding

Output Data: compressed
Huffman Coding

- Final stage that performs actual compression
- Replace each character with a bit code
- Characters that occur more often get shorter codes
- Huffman Coding is more effective with repetitive data (after MTF)
Huffman Coding

“HUFF” \(\rightarrow\) 00 01 1 1
32 bits 6 bits
Huffman Coding

- Huffman Stages:
  1. Generate 256-bin histogram
  2. Build Huffman tree
  3. Replace characters with codes
Huffman Histogram

- Histogram
  - Build a 256-entry histogram to count characters
  - To do this on the GPU, divide the input data among threads with each thread maintaining its own histogram
  - Merging histograms requires atomic operations
Huffman Tree

• Huffman Tree Algorithm
  o Remove two lowest counts, combine to form composite node
  o Composite node “inserted” into histogram with combined counts
  o Each step is dependent on the previous step
  o Parallelize finding the two lowest counts (maximum 256-way parallelism)
Huffman Tree Algorithm

**Histogram**
- H: 1
- U: 1
- F: 2

“HUFF”
Huffman Tree Algorithm

Histogram
H: 1
U: 1
F: 2

Input Data
Burrows-Wheeler Transform
Move-to-Front Transform
Huffman Coding
Output Data: compressed
Huffman Tree Algorithm

Histogram
F: 2

Input Data
Burrows-Wheeler Transform
Move-to-Front Transform
Huffman Coding
Output Data: compressed
Huffman Tree Algorithm

Histogram
H+U: 2
F: 2

Input Data
Burrows-Wheeler Transform
Move-to-Front Transform
Huffman Coding
Output Data: compressed
Huffman Tree Algorithm

Histogram
H+U: 2
F: 2
Huffman Tree Algorithm

Histogram
(empty)

Input Data
Burrows-Wheeler Transform
Move-to-Front Transform
Huffman Coding
Output Data: compressed
Huffman Tree Algorithm

Left → 0  Right → 1

Huffman Codes
H: 00  
U: 01  
F: 1

Input Data

Burrows-Wheeler Transform

Move-to-Front Transform

Huffman Coding

Output Data: compressed
Huffman Coding

• Huffman coding assigns **prefix codes**
• No codes share the same prefix

**Huffman Codes**
- H: 00
- U: 01
- F: 1
Huffman Coding

Huffman Codes
H: 00
U: 01
F: 1

“HUFF” → 000111
32 bits → 6 bits
Huffman Coding

- Replace characters with codes
  - To do in parallel, divide input among threads
  - Hard to do on the GPU because codes are variable-length
  - Each thread must calculate the correct offset to begin writing its code

Input Data

Output Data: compressed

```
t0  t1  t2  t3  t4
1   0010 000011 0010 000011
```
Results

• GPU vs. Bzip2
  o Overall
    • GPU 2.78x slower
  o BWT
    • GPU 2.9x slower (91% of runtime)
  o MTF + Huffman
    • GPU 1.34x slower
## Results

### Benchmark results

<table>
<thead>
<tr>
<th>File (Size)</th>
<th>Compress Rate (MB/s)</th>
<th>BWT Sort Rate (Mstrings/s)</th>
<th>MTF+Huffman Rate (MB/s)</th>
<th>Compress Ratio Compressed Size Uncompressed Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Text (97 MB)</td>
<td>GPU: 7.37 bzip2: 10.26</td>
<td>GPU: 9.84 bzip2: 14.2</td>
<td>GPU: 29.4 bzip2: 33.1</td>
<td>GPU: 0.33 bzip2: 0.29</td>
</tr>
<tr>
<td>Source Code (203 MB)</td>
<td>GPU: 4.25 bzip2: 9.8</td>
<td>GPU: 4.71 bzip2: 12.2</td>
<td>GPU: 44.3 bzip2: 48.8</td>
<td>GPU: 0.24 bzip2: 0.18</td>
</tr>
<tr>
<td>XML (151 MB)</td>
<td>GPU: 1.42 bzip2: 5.3</td>
<td>GPU: 1.49 bzip2: 5.4</td>
<td>GPU: 32.6 bzip2: 69.2</td>
<td>GPU: 0.19 bzip2: 0.10</td>
</tr>
</tbody>
</table>

Worst performance during string sort and overall

Best amount of compression
Results

• BWT Performance
  o GPU – 91% of runtime
  o Bzip2 – 81% of runtime
  o Tie Analysis:
    • Amount of compression is data dependent
    • Better compression leads to longer ties and poor performance
Results

- MTF Performance
  - Majority of runtime during character replacement
  - Moving element to front, shifting all other elements
  - Thread divergence
Results

• Huffman Performance
  o Majority of runtime during Huffman tree building
  o 256-way parallelism is inadequate for the GPU
Decompression

- **Reverse Burrows-Wheeler transform**
  - Much faster than forward BWT
  - Requires only 1 character-sort
  - Similar to linked-list traversal
  - Preliminary implementation is an extension of Hillis and Steele’s parallel linked-list traversal algorithm
Decompression

- **Reverse Burrows-Wheeler transform**
  - Much faster than forward BWT
  - Requires only 1 character-sort
  - Similar to linked-list traversal
  - Preliminary implementation is an extension of Hillis and Steele's parallel linked-list traversal algorithm

- **Reverse Move-to-Front transform**
  - Parallel approach uses similar algorithm as forward-MTF
    - Generate partial MTF lists
    - Combine adjacent lists
Decompression

- Reverse Burrows-Wheeler transform
  - Much faster than forward BWT
  - Requires only 1 character-sort
  - Similar to linked-list traversal
  - Preliminary implementation is an extension of Hillis and Steele’s parallel linked-list traversal algorithm

- Reverse Move-to-Front transform
  - Parallel approach uses similar algorithm as forward-MTF
    - Generate partial MTF lists
    - Combine adjacent lists

- Huffman decoding
  - Decode each encoded block in parallel
Future Work

• Explore other GPU-based string sorts that can better handle long strings with many ties
• Develop a Huffman tree building algorithm that has more than 256-way parallelism
• Overlap GPU compression and PCI-Express data transfer
Conclusion

• Implemented parallel lossless data compression on the GPU

• Parallelized BWT, MTF, and Huffman Coding
  o Developed a novel algorithm for MTF
  o BWT string sort contributes to the majority of runtime

• Implementation is slow but may be better suited for future GPU architectures and other parallel environments
Acknowledgements

- NSF grants OCI-1032859 and CCF-1017399
- HP Labs Innovation Research Program
- Discussion and feedback
  - Shubho Sengupta (Intel)
  - Adam Herr (Intel)
  - Anjul Patney (UC Davis)
  - Stanley Tzeng (UC Davis)