Optimizing Huffman Decoding for Error-Bounded Lossy Compression on GPUs

Cody Rivera (cjrvera1@crimson.ua.edu)
Sheng Di (sdi1@anl.gov)
Jiannan Tian (jiannan.tian@wsu.edu)
Xiaodong Yu (xyu@anl.gov)
Dingwen Tao (dingwen.tao@wsu.edu)
Franck Cappello (cappello@mcs.anl.gov)
Background: Use Cases

- **HPC Application Needs**
  - HPC applications are generating **increasingly large amounts of data**
  - Applications include large scale simulations, deep neural networks
  - e.g., Hardware/Hybrid Accelerated Cosmology Code (HACC) (S. Habib et. al.) [1], which generates roughly **22 petabytes** per simulation run

- **In-Memory Caching**
  - Fast memory, especially on GPUs, is a **scarce** resource
  - Can cache data more economically by **compressing** and **decompressing** intermediate data
  - e.g., Wu et. al.’s work on quantum circuit simulation [17], where compression reduces total RAM usage from **32 exabytes** to **768 terabytes**
Background: Lossy Compression/SZ

- **Lossy compression on scientific data**
  - Offers much *higher compression ratios* than lossless compression by trading a little bit of accuracy
  - An example: **SZ**, a prediction-based lossy compression that achieves high compression ratios [5]
    - Actively *developed* and researched
    - CPU, GPU (**cuSZ**), and domain-specific (**DeepSZ**, **PastriSZ**) versions
  - We focus on **SZ/cuSZ** (over **ZFP**) for the following reasons:
    - Less distortion/*higher PSNR* at a given bitrate
    - Compression error can be explicitly *bounded* by the user
An Overview of cuSZ

- **Prediction**
  - Predict data points using a data-fitting Lorenzo predictor (Ibarria et al.) [7]

- **Quantization**
  - Determine the prediction error for each point and classify it as an integer we call a quantization code

- **Huffman Coding**
  - Losslessly compress quantization codes
Background: Huffman Coding

- **Huffman Coding**
  - Classic lossless variable-length compression technique introduced by David Huffman in 1952
  - Example: ABAACDAA (16 bits at 2 bits per character)
    - Encoded Text: 1010110110011 (13 bits)
    - Compression Ratio: 16/13 ≈ 1.23

Example Huffman Tree and Codebook

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Codeword</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1</td>
</tr>
<tr>
<td>D</td>
<td>00</td>
</tr>
<tr>
<td>B</td>
<td>010</td>
</tr>
<tr>
<td>C</td>
<td>011</td>
</tr>
</tbody>
</table>

Example Huffman Tree and Codebook
Motivation: Why Optimize Decoding?

- **cuSZ’s current Huffman coding**
  - Encoding performance: average 25.7 GB/s in production [9], 138.3 GB/s experimentally (J. Tian et. al., IPDPS ‘21) [10]
  - Decoding performance: average 32.3 GB/s

- **Research Focus**
  - Decompression is needed for data post-analysis as well as retrieval from in-memory caches
  - Huffman decoding, however, is the primary bottleneck for cuSZ, taking up 83% of the time in a recent version
Increasing Parallelism in Decoding

- **Coarse-Grained Parallelism**
  - Many points per thread, few threads
  - cuSZ’s current Huffman decoder
  - Does not map well to GPU architectures

- **Fine-Grained Parallelism**
  - Few points per thread, many threads
  - Maps more effectively to the GPU’s massive parallelism
The Self-Synchronization Property

- Tendency for Huffman codes to correct themselves if a few bits were skipped, first written about by Ferguson and Rabinowitz [11]
- Example: 111000010111000
  - Correct decoding
    - (11)(10)(00)(010)(11)(10)(00)
    - CBADCBA
  - Incorrect decoding
    - 11000010111000
    - CAAB
    - Is eventually correct

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Codeword</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>00</td>
</tr>
<tr>
<td>B</td>
<td>10</td>
</tr>
<tr>
<td>C</td>
<td>11</td>
</tr>
<tr>
<td>D</td>
<td>010</td>
</tr>
<tr>
<td>E</td>
<td>011</td>
</tr>
</tbody>
</table>

Self-Synchronizing Codebook
Finding Thread Boundaries

- Consider the message
  - BACACCDBAAEBBA

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Codeword</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>00</td>
</tr>
<tr>
<td>B</td>
<td>10</td>
</tr>
<tr>
<td>C</td>
<td>11</td>
</tr>
<tr>
<td>D</td>
<td>010</td>
</tr>
<tr>
<td>E</td>
<td>011</td>
</tr>
</tbody>
</table>

Subsequence 0, 1  
\[
\begin{array}{c}
0001100 \\
11111001
\end{array}
\]

Subsequence 2, 3  
\[
\begin{array}{c}
01000000 \\
11101000
\end{array}
\]
Finding Thread Boundaries

- **Beginning of the procedure**

Legend:
- **Thread Position**
- **Synchronization Point**
- **Verified Synchronization Point**

Index 0, Index 1, Index 2, Index 3

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Codeword</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>00</td>
</tr>
<tr>
<td>B</td>
<td>10</td>
</tr>
<tr>
<td>C</td>
<td>11</td>
</tr>
<tr>
<td>D</td>
<td>010</td>
</tr>
<tr>
<td>E</td>
<td>011</td>
</tr>
</tbody>
</table>

Subsequence 0, 1

10001100 11111001

Subsequence 2, 3

01000000 11101000
Finding Thread Boundaries

- Each thread decodes a subsequence

Legend:
- Thread Position
- Synchronization Point
- Verified Synchronization Point

Index 0, Index 1, Index 2, Index 3

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Codeword</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>00</td>
</tr>
<tr>
<td>B</td>
<td>10</td>
</tr>
<tr>
<td>C</td>
<td>11</td>
</tr>
<tr>
<td>D</td>
<td>010</td>
</tr>
<tr>
<td>E</td>
<td>011</td>
</tr>
</tbody>
</table>

Subsequence 0, 1:

<table>
<thead>
<tr>
<th>B</th>
<th>A</th>
<th>C</th>
<th>A</th>
<th>C</th>
<th>C</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Subsequence 2, 3:

<table>
<thead>
<tr>
<th>D</th>
<th>A</th>
<th>A</th>
<th>C</th>
<th>B</th>
<th>B</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Subsequence 1, 3:

| 1 | 1 | 1 | 1 | 1 | 0 | 0 | 1 |

Subsequence 0, 2:

| 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
Finding Thread Boundaries

- **Synchronization points are initialized**

Legend:

- Thread Position
- Synchronization Point
- Verified Synchronization Point

Index 0, Index 1, Index 2, Index 3

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Codeword</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>00</td>
</tr>
<tr>
<td>B</td>
<td>10</td>
</tr>
<tr>
<td>C</td>
<td>11</td>
</tr>
<tr>
<td>D</td>
<td>010</td>
</tr>
<tr>
<td>E</td>
<td>011</td>
</tr>
</tbody>
</table>

Subsequence 0, 1

1 0 0 0 1 1 0 0 1 1 1 1 1 0 0 0 1

Subsequence 2, 3

0 1 0 0 0 0 0 0 1 1 1 0 1 0 0 0
Finding Thread Boundaries

- Each synchronization point is verified by the previous thread

Legend:
- Thread Position
- Synchronization Point
- Verified Synchronization Point

Index 0, Index 1, Index 2, Index 3

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Codeword</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>00</td>
</tr>
<tr>
<td>B</td>
<td>10</td>
</tr>
<tr>
<td>C</td>
<td>11</td>
</tr>
<tr>
<td>D</td>
<td>010</td>
</tr>
<tr>
<td>E</td>
<td>011</td>
</tr>
</tbody>
</table>

Subsequence 0, 1

Subsequence 2, 3
Finding Thread Boundaries

- Once this is done, each thread will decode parts of the following correctly
  - BACACCBDBAAEBBA

Legend:

- Thread Position
- Synchronization Point
- Verified Synchronization Point

Index 0, Index 1, Index 2, Index 3

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Codeword</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>00</td>
</tr>
<tr>
<td>B</td>
<td>10</td>
</tr>
<tr>
<td>C</td>
<td>11</td>
</tr>
<tr>
<td>D</td>
<td>010</td>
</tr>
<tr>
<td>E</td>
<td>011</td>
</tr>
</tbody>
</table>

Subsequence 0, 1

Subsequence 2, 3
Another Approach to Fine-grained Parallelism

- **Gap Arrays**
  - Determining synchronization points requires **redundant** decoding
  - Yamamoto et. al. propose a solution: **precompute** the start points for each thread at encoding time, put them in a gap array [13], and use them for fast decoding

![Diagram of gap arrays and subsequences]

Subsequence 0, 1

\[
1 0 0 0 1 1 0 0 \quad 1 1 1 1 1 0 0 1
\]

Subsequence 2, 3

\[
0 1 0 0 0 0 0 0 \quad 1 1 1 0 1 0 0 0
\]

A gap array: \{0, 0, -2, -1\}
Implementing and Optimizing Decoders

- Adaptation
  - Change from single-byte input to multi-byte input

- Thread-Level Optimization (for self-sync)
  - With self-synchronization, adjacent threads may decode very different amounts of data - divergence
  - Program with the thread hierarchy in mind

- Memory Optimization (for both)
  - Use wider/vector loads and stores
  - Use the GPU's shared memory to cache decoded results

CUDA C programming guide, [6]
Motivation for Memory Optimizations

- **High-compression ratio data**
  - Often found in scientific computing/cuSZ workflows, especially where the data has been well-predicted
  - Significant performance penalties for increased compression ratio \( \approx \) decreased error-bound

- **Reason**
  - With high compression ratios, each thread writes more data
  - Also, there is a larger stride between threads, an even worse access pattern
Shared Memory Optimization Details

- **The technique**
  - Each thread writes into the block-local shared memory
  - The shared memory is **cooperatively** written out to global memory

- **Allocating shared memory**
  - Proportional to the **compression ratio** of the data
  - Different portions of the data need different amounts of shared memory
  - Use **multiple kernel launches** to efficiently decompress different portions of the data

---

**Algorithm 1: Decoding and writing using a shared memory buffer.**

```
DecWrite — decode and write using shared memory

1. sharedBuffer[n] — the shared memory buffer of size n
2. si <- outIndex[blockIdx.x * blockDim.x]
3. ei <- outIndex[(blockIdx.x + 1) * blockDim.x]
4. gid <- threadIdx.x + blockDim.x * threadIdx.x
5. tempEnd <- ei
6. while si < ei do
7.   start <- outIndex[gid] - si, end <- outIndex[gid + 1]
8.   if si <= start and end <= si + n then
9.     outArray[start ... end] = DECODE(inArray, startPoint[gid])
   > if symbols can fit into the buffer, decode them
10. else if start < si + n and end >= si + n then
11.    tempEnd <- outIndex[gid]
   > Executed by one thread if buffer is not large enough; results in another iteration
12. end if
13. outArray[s] ... tempEnd = sharedBuffer[0 ... tempEnd - si]
   > This write is performed cooperatively by threads in the block
14. si <- tempEnd
15. end while
```
Evaluation

- **Experimental Setup**
  - **Datasets**: Multidimensional data from a variety of scientific domains; data sources include the Scientific Data Reduction Benchmark [15], in addition to some other sources
  - **Platform**: 2 Xeon Gold 6428 “Cascade Lake” CPUs, 20 cores; 8 Nvidia Tesla V100-32GB SXM2 GPUs (only 1 GPU was used for evaluation)

<table>
<thead>
<tr>
<th>datasets</th>
<th>dimensions</th>
<th>examples(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>cosmology</td>
<td>1,071.75 MB</td>
<td>6 in total</td>
</tr>
<tr>
<td>HACC</td>
<td>280,953,867 MB</td>
<td>xx, vx</td>
</tr>
<tr>
<td>molecular dynamics</td>
<td>951.73 MB</td>
<td>6 in total</td>
</tr>
<tr>
<td>EXAALT</td>
<td>2338×106711 MB</td>
<td>dataset2.x</td>
</tr>
<tr>
<td>climate</td>
<td>642.70 MB</td>
<td>33 in total</td>
</tr>
<tr>
<td>CESM-ATM</td>
<td>26×1800×3,600 MB</td>
<td>CLDICE, RELHUM</td>
</tr>
<tr>
<td>cosmology</td>
<td>512 MB</td>
<td>6 in total</td>
</tr>
<tr>
<td>Nyx</td>
<td>512×512×512 MB</td>
<td>baryon_density</td>
</tr>
<tr>
<td>climate</td>
<td>381.47 MB</td>
<td>13 in total</td>
</tr>
<tr>
<td>Hurricane</td>
<td>4×100×500×500 MB</td>
<td>CLDICE, QRAIN</td>
</tr>
<tr>
<td>quantum circuits</td>
<td>601.52 MB</td>
<td>2 in total</td>
</tr>
<tr>
<td>QMCPack</td>
<td>115×69×69×288 MB</td>
<td>einspline, einspline.pre</td>
</tr>
<tr>
<td>petroleum exploration</td>
<td>180.73 MB</td>
<td>1 in total (3600 snapshots) snapshot-1000</td>
</tr>
<tr>
<td>RTM</td>
<td>449×449×235 MB</td>
<td></td>
</tr>
<tr>
<td>quantum chemistry</td>
<td>306.19 MB</td>
<td>3 in total</td>
</tr>
<tr>
<td>GAMESS</td>
<td>80,265,168 MB</td>
<td>dddd, fddd, ffff</td>
</tr>
</tbody>
</table>

Our tested datasets

Bridges2 cluster at Pittsburg Supercomputing Center [14]
Evaluation

- Evaluation on Decoding Alone
  - Outperforms coarse-grained cuSZ decoder
    - Average 2.74x for self-synchronization
    - Average 3.64x for gap arrays
  - Predictably, gap arrays are faster than self-synchronization

---

Performance of Optimized Huffman Decoding

- cuSZ baseline
- Opt. Self Sync
- Opt. Gap Array

Throughput (GB/s)
Evaluation

- **Evaluation on cuSZ’s decompression, overall**
  - Considering decoding took up 83% of cuSZ’s time, not surprising to see a speedup
  - Average 2.08x for self-synchronization
  - Average 2.43x for gap arrays
Discussion & Conclusion

**Discussion**

- **Self-Synchronization**
  - **Benefits:** Huffman encoder and decoder are decoupled
  - **Drawbacks:** Redundant computations

- **Gap Arrays**
  - **Benefits:** Higher performance in decoding
  - **Drawbacks:** Encoder and decoder must be coupled; higher overhead on encoder

**Conclusion**

- Apply **two algorithms** for finer-grained parallel Huffman decoding to significantly speed up cuSZ’s decompression
- **Optimize** these algorithms to more effectively take advantage of GPU architectures
- Future work to be done
  - **Incorporate** optimizations into a Huffman coding library
  - **Extend** work to other applications of Huffman decoding
Thank you!
All the questions and ideas are welcomed

Contact: Dingwen Tao: dingwen.tao@wsu.edu
Cody Rivera: cjrivers1@crimson.ua.edu