Revisiting Huffman Coding: Toward Extreme Performance on Modern GPU Architectures

Jiannan Tian *Washington State University*
Cody Rivera *The University of Alabama*
Sheng Di *Argonne National Laboratory*
Jieyang Chen *Oak Ridge National Laboratory*
Xin Liang *Oak Ridge National Laboratory*
Dingwen Tao *Washington State University*
Franck Cappello *Argonne National Laboratory*

May 20, 2021 (Thursday) IPDPS ’21, Portland, Oregon, USA

Trend of Supercomputing Systems

The capability of compute is developed faster while those of storage and bandwidth are developed relatively slowly. There is widening gap between compute unit and storage bandwidth (PF–SB) or main memory size and storage bandwidth (MS–SB).

supercomputer	year	class	PF	MS	SB	MS/SB	PF/SB
Cray Jaguar	2008	1 PFLOPS	1.75 PFLOPS	360 TB	240 GB/s	1.5k	7.3k
Cray Blue Waters	2012	10 PFLOPS	13.3 PFLOPS	1.5 PB	1.1 TB/s	1.3k	13k
Cray CORI	2017	10 PFLOPS	30 PFLOPS	1.4 PB	1.7 TB/s •	0.8k	17k
IBM Summit	2018	100 PFLOPS	200 PFLOPS	>10 PB ••	2.5 TB/s	>4k	80k

PF: peak FLOPS MS: memory size SB: storage bandwidth

• when using burst buffer •• counting only DDR4 *Source: F. Cappello (ANL)*

Table 1: Three classes of supercomputers showing their performance, **MS** and **SB**.

Trend of Supercomputing Systems

Washington State University · Oak Ridge National Laboratory · Argonne National Laboratory

supercomputer	year	class	PF	MS	SB	MS/SB	PF/SB
Cray Jaguar	2008	1 PFLOPS	1.75 PFLOPS	360 TB	240 GB/s	1.5k	7.3k
Cray Blue Waters	2012	10 PFLOPS	13.3 PFLOPS	1.5 PB	1.1 TB/s	1.3k	13k
Cray CORI	2017	10 PFLOPS	30 PFLOPS	1.4 PB	1.7 TB/s •	0.8k	17k
IBM Summit	2018	100 PFLOPS	200 PFLOPS	>10 PB ••	2.5 TB/s	>4k	80k

PF: peak FLOPS MS: memory size SB: storage bandwidth

• when using burst buffer •• counting only DDR4 *Source: F. Cappello (ANL)*

supercomputer	year	class	PF	MS	SB	MS/SB	PF/SB
Fujitsu Fugaku	2020	"ExaScale"	537 PFLOPS •	4.85 PB	≥1.5 TB/s ••	≥3.2k	358k
Intel Aurora	future	ExaScale	≥1 EFLOPS	>10 PB	≥25 TB/s	≥0.4k	40k

• Rpeak, TOP 500 for November 2020 •• DDN Newsroom

Table 1: More classes of supercomputers showing their performance, **MS** and **SB**.

Design Compressor for HPC (1/2)

Today's scientific research is data-driven at a large scale (simulations or instruments). Compression matters when I/O, communiation, memory capacity are performance limiter.

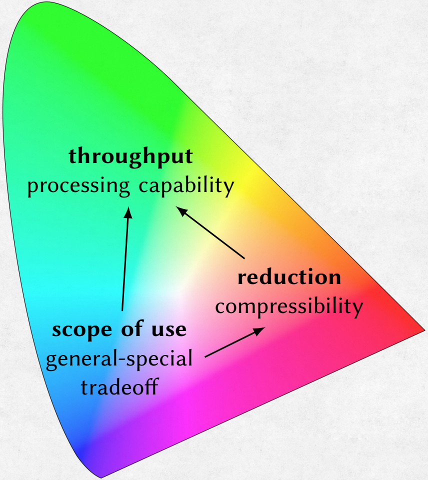
"…the rate of data that can be computed on the Summit supercomputer is five orders of magnitude greater than the bandwidth of its parallel file system. The I/O bottleneck is one driver of in situ analysis."

—ASCR Workshop on In Situ Data Management

"Novel technologies and emerging architectures provide new opportunities to address these data reduction requirements and also lead to new research challenges…new research is needed in data reduction algorithms and software stacks that can leverage their unique capabilities."

—Data Reduction for Science: Brochure from the Advanced Scientific Computing Research Workshop (Technical Report)

Design Compressor for HPC (2/2)



Under the context of huge imbalance between compute capability and data management,

scope of use A compressor can be general-purpose or data-dependent, generic or contextual.
Strategy: extending the current compressors.

reduction With context, i.e. knowing data feature, it is possible to achieve higher compressibility. In our case, Huffman coding does not exploit repeated pattern.

throughput In situ processing requires, for example,

1. time of compression + $\text{store}_{\text{DRAM}\to\text{disk}}$ < time of direct $\text{store}_{\text{DRAM}\to\text{disk}}$
2. time of $\text{load}_{\text{disk}\to\text{DRAM}}$ + decompression < time of direct $\text{load}_{\text{disk}\to\text{DRAM}}$

Lossless::Huffman::GPU & Multibyte Symbols

- workflow:
  histogram → codebook construction → encoding

	CPU	omp-CPU	GPU
histogram		√	√√
codebook	√		
encoding		√	√√

- extended general-purpose compressor
  - Pattern-finding such as LZW is non-trivial on GPU.
  - Huffman-coding-only as a solution
  - modified to enumerate all symbols
- rationale of multibyte symbols
  - Rather than combining multiple (256-symbol) singlebytes to exhaust virtually all data types, we enumerate all symbols.
  - Number of symbols is greater than 256 but far less than big number such as `INT_MAX`
  - Without pattern finding, encoded data can be dominated by 1-bit codeword from `0x00` byte.
  - demo: `(int) 512 = 0x00000200`
    multibyte: one `512`
    singlebyte: one `0x02` and three `0x00`

parallelisms

	sequential	coarse-grained	fine-grained	data-thread many-to-one	data-thread one-to-one	atomic write	reduction	prefix sum	boundary
histogram									
blockwise reduction			•	•		•	•		sync block
gridwise reduction			•	•		•	•		sync device
build codebook									
get codeword lengths		•	•	•	•	•			sync grid
get codewords			•		•	•			sync grid
canonize									
get numl array			•		•	•		•	sync grid
get first array (RAW)	•			•					sync grid
canonization (RAW)	•			•					sync grid
get reverse codebook			•						sync device
Huffman enc.									
`reduce`-merge		•	•	•			•		sync block
`shuffle`-merge		•	•		•				sync device
get blockwise code len		•	•		•			•	sync grid
coalescing copy		•	•		•				sync device

Table 2: Parallelism implemented for Huffman coding's subprocedures (kernels). "sequential" denotes that only 1 thread is used due to data dependency. "coarse-grained" denotes that data is explicitly chunked. "fine-grained" denotes that there is a data-thread mapping with little or no warp divergence.

Parallel Construction of Codebook (1/2)

- a parallel alternative to the original $\mathcal{O}(n \log n)$ Huffman codebook construction
  - directly generates codewords
  - proposed by Ostadzadeh et al.
  - Our implementation is proof-of-concept of the theoretical complexity.
- two-phase algorithm
  - `GenerateCL`: calculate the codeword length for each input symbol
  - `GenerateCW`: generate the actual codeword for each input symbol
- implementation
  - Both phases utilize fine-grain parallelism, one-thread-one-symbol/ value.
  - Both phases are implemented as single CUDA kernels with Cooperative Groups.

Parallel Construction of Codebook (2/2)

Washington State University · Oak Ridge National Laboratory · Argonne National Laboratory

- `GenerateCL`
  - input: a sorted n-symbol histogram
  - output: CL, a size n array of codeword lengths for each symbol.
  - $\mathcal{O}\left(H \cdot \log\log \frac{n}{H}\right)$ time on PRAM, where H is the longest codeword.
  - implementation: more likely $\mathcal{O}(\log n)$
  - source of parallelism: given a set of Huffman subtrees, all subtrees whose total frequencies are less than the sum of the two smallest subtree frequencies can be combined in parallel.
- `GenerateCW`
  - input: CL; output `CW`
  - generally parallel-generating canonized codebook
  - Codewords are generated by individual threads.
  - $\mathcal{O}(H)$ time per thread in the PRAM model.

Encoding (1/3): of Fine Granularity

Typically, Huffman encoding is

- done on CPU, because of data dependency in codeword bit-positions
- possible to enable coarse-grained parallism, e.g. OpenMP

Encoding on GPU

- use OpenMP-procedure-like kernel
  - latency-bound kernel in general
  - But high memoy bandwidth on GPU really helps; kernel throughput can exceed CPU DRAM bandwidth.

Encoding (1/3): of Fine Granularity

Washington State University · Oak Ridge National Laboratory · Argonne National Laboratory

Typically, Huffman encoding is

- done on CPU, because of data dependency in codeword bit-positions
- possible to enable coarse-grained parallism, e.g. OpenMP

Encoding on GPU

- use OpenMP-procedure-like kernel
  - latency-bound kernel in general
  - But high memoy bandwidth on GPU really helps; kernel throughput can exceed CPU DRAM bandwidth.

- Previous fine-grained GPU method
  - partial-sum to determine positions prior to writing
  - no compressibility awareness
  - Direct writing to assigned position, however, is inefficient, considering that
    - multiprocessor registers are mostly 32-bit, while
    - the averge bitwidth is low (1 to 5 bits), the transaction time increases, and
    - variable length makes it irregular access (even coalescing accessing shmem matters).

Encoding (1/3): of Fine Granularity

Washington State University · Oak Ridge National Laboratory · Argonne National Laboratory

Typically, Huffman encoding is

- done on CPU, because of data dependency in codeword bit-positions
- possible to enable coarse-grained parallism, e.g. OpenMP

Encoding on GPU

- use OpenMP-procedure-like kernel
  - latency-bound kernel in general
  - But high memoy bandwidth on GPU really helps; kernel throughput can exceed CPU DRAM bandwidth.

- Previous fine-grained GPU method
  - partial-sum to determine positions prior to writing
  - no compressibility awareness
  - Direct writing to assigned position, however, is inefficient, considering that
    - multiprocessor registers are mostly 32-bit, while
    - the averge bitwidth is low (1 to 5 bits), the transaction time increases, and
    - variable length makes it irregular access (even coalescing accessing shmem matters).
- Our Method
  - adress two issues: register underuse (more transactions), bitwise irregular access
  - iterative merge of codewords along with reduction of bitwidths (metadata)
  - Given code-length tuples $(a, \ell)_{2k}$ and $(a, \ell)_{2k+1}$,

$$\texttt{Merge}\big((a,\ell)_{2k}, (a,\ell)_{2k+1}\big) = \big(a_{2k} \oplus a_{2k+1}, \ell_{2k} + \ell_{2k+1}\big),$$

    where $\oplus$ represents for concatenating bits of a_{2k+1} right after bits of a_{2k}.
  - note: merge is not commutative ($x \oplus y \neq y \oplus x$).

Encoding (2/3): `reduce`- and `shuffle`-merge

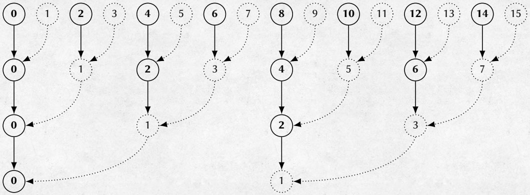


Figure 1: `reduce`-merge of 8-to-1.

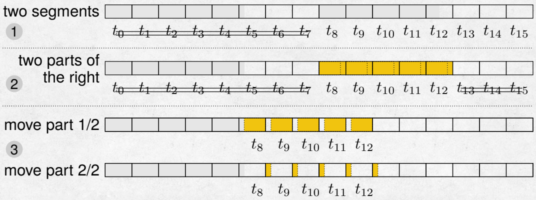


Figure 2: Two-step batch move of grouped and typed data. By batch-moving the right grouped data, warp divergence is decreased.

Granularity-coarsening of bit operations (`reduce`-merge)

- one-thread-multiple-data
- stop before the merged words saturate 32 bits
- performance degradation seen when set to 64 bits as the saturating bitwidth

Concurrently align the segments (`shuffle`-merge)

- address irregular bitwise access
- two-step: 1) (dtype-width − ending residue) bits, 2) ending residue bits
- mostly thread masking (`if...` without `else`)

Encoding (3/3): Compressiblity

In retrospect of our throughput-oriented design, we find that

- The performance is impacted by the intrinsic data feature.
- Specifically, (data-dependent) compressiblity.
- The compressibility is instantly known after either histograming or Huffman codebook is consructed.

Our method generally translates throughput-impacting compressibility into number of `reduce` iterations.

Encoding (3/3): Compressiblity

In retrospect of our throughput-oriented design, we find that

- The performance is impacted by the intrinsic data feature.
- Specifically, (data-dependent) compressiblity.
- The compressibility is instantly known after either histograming or Huffman codebook is consructed.

Our method generally translates throughput-impacting compressibility into number of `reduce` iterations.

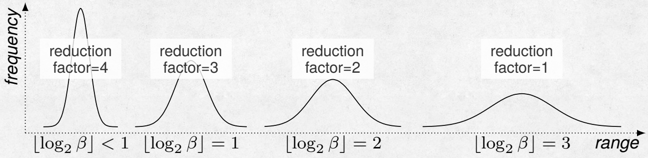


Figure 3: Average bitwidth being a consideration to decide reduction factor.

Encoding (3/3): Compressiblity

In retrospect of our throughput-oriented design, we find that

- The performance is impacted by the intrinsic data feature.
- Specifically, (data-dependent) compressiblity.
- The compressibility is instantly known after either histograming or Huffman codebook is consructed.

Our method generally translates throughput-impacting compressibility into number of `reduce` iterations.

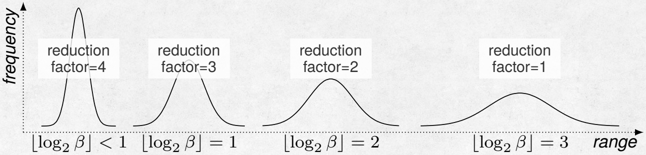


Figure 3: Average bitwidth being a consideration to decide reduction factor.

reduction factor / mag. →	Longhorn 2^{12}	Longhorn 2^{11}	Longhorn 2^{10}	Frontera 2^{12}	Frontera 2^{11}	Frontera 2^{10}	breaking
(16×) 4	227.60	274.40	291.04	110.94	124.42	133.84	0.007536%
(8×) 3	191.41	274.42	314.63	94.27	124.56	135.86	0.003277%
(4×) 2	68.32	106.87	172.54	42.70	55.53	79.45	0.000434%

Table 3: (`Nyx-Quant`, avg. bitwidth=1.027) Performance (in GB/s) of our Huffman encoding with different chunk magnitudes (mag.) and reduction factors on Longhorn and Frontera.

Evaluation Setup: Platform and Dataset

WASHINGTON STATE UNIVERSITY · OAK RIDGE National Laboratory · Argonne NATIONAL LABORATORY

- Evaluation Platforms (TACC)
  - Longhorn, NVIDIA **V100** (top-tier)
    - 16 GB HBM2 at 900 GB/s; SXM2 variant
    - 2×20-core IBM Power 9
  - Frontera, NVIDIA **RTX 5000** (PCIe 3.0) (professional/HPC-tier)
    - 16 GB GDDR6 at 448 GB/s
    - 2×28-core Intel 8280
- Comparison Baseline (multibyte codebook)

	CPU-SZ	OMP prototype	cuSZ
histogram	serial	multithread	kernel●
codebook	serial	multithread	serial
encoding	serial	multithread	kernel○

○ coarse-grained ● fine-grained

⋆ We continue using histogram kernel in cuSZ.

- Test Datasets
  - Singlebyte Based Datasets (at most 256 symbols)
    - `enwiki8` and `enwiki9`—XML-based English Wikipedia dump (*Large Text Compression Benchmark*)
    - `nci`—chemical database of structures (*Silesia Corpus*)
    - `mr`—medical magnetic resonance image sample (*Silesia Corpus*)
    - `Flan_1565`—sparse matrix in Rutherford Boeing format (*SuiteSparse Matrix Collection*)
  - Multibyte Based Datasets (beyond 256 symbols)
    - `Nyx-Quant`—integer-typed intermediate error-control code of cuSZ, with e.g. 1024 symbols
    - `gbbct1.seq`– sample DNA data from *GenBank*; every k nucleotides (k-mer) forms a symbol; $k = \{3, 4, 5\}$ are tested.

Codebook Construction (OMP vs. CPU) 2/2

	#sym.	serial	1 core	2 cores	4 cores	6 cores	8 cores
Nyx-Quant	1024	**0.045**	0.219	0.469	0.622	0.700	0.840
3-mer	2048	**0.208**	0.361	0.691	1.101	1.122	1.303
4-mer	4096	0.695	**0.626**	1.006	1.309	1.456	1.707
5-mer	8192	1.806	**1.167**	1.513	1.657	1.836	2.158
Synthetic	16384	3.671	**1.683**	1.796	1.705	2.055	2.222
Synthetic	32768	5.783	2.974	2.858	**2.626**	2.873	3.139
Synthetic	65536	7.641	5.221	4.850	**4.411**	4.952	5.713

Table 4: Performance (in milliseconds) of multi-thread codebook construc-tion with different numbers of input symbols. The length of the bar under thenumber reflects the execution time.

- sythetic data: normally distributed histograms with 16k to 65k symbols
- Serial construction excels when symbol number is small.
- OpenMP overhead is overcome beyond 32k symbols.

Codebook Construction (2/2): GPU Old vs. New

		#sym.	ref. CPU serial	gen. codebook (TU)	gen. codebook (V)	canonize (TU)	canonize (V)	total time (TU)	total time (V)
cuSZ serial	Nyx-Quant	1024	0.045	3.051	3.689	0.095	0.115	3.416	3.804
	3-mer	2048	0.208	8.381	9.760	0.242	0.284	8.623	10.044
	4-mer	4096	0.695	20.148	24.684	0.519	0.663	20.667	25.347
	5-mer	8192	1.806	61.748	59.092	1.453	1.449	63.201	60.541
		#sym.	**serial**	**gen. CL (TU)**	**gen. CL (V)**	**gen. CW (TU)**	**gen. CW (V)**	**total time (TU)**	**total time (V)**
Ours parallel	Nyx-Quant	1024	0.045	0.315	0.383	0.134	0.161	0.449	0.544
	3-mer	2048	0.208	0.494	0.570	0.180	0.209	0.674	0.779
	4-mer	4096	0.695	0.633	0.682	0.173	0.185	0.806	0.867
	5-mer	8192	1.806	1.330	1.145	0.154	0.187	1.484	1.332

Table 5: Breakdown comparison of Huffman codebook construction time(in milliseconds) on RTX 5000 and V100 with different numbers of symbols.

- Unlike CPU, GPU parallel construction can always yield a speedup over serial construction in our tested cases.
- Ours exhibits more dramatic speedups over cuSZ's when using more input symbols, consistent with our theoretical analysis and performing up to 45.5× faster when creating a codebook for 8192 symbols.
- Note that ours is no faster than the CPU serial construction when the number of symbols is below 8192.

Performance Evaluation: GPU vs. OpenMP

cores	1	2	4	8	16	32	56	64	TU	V
hist. (GB/s)	2.24	4.42	8.83	17.61	34.97	63.59	61.47	63.14	74.80	197.60
par. efficiency	1.00	0.99	0.98	0.98	0.97	0.89	0.49	0.44		
codebook (ms)				0.22					0.45	0.54
enc. (GB/s)	1.22	2.43	4.83	9.64	19.16	37.85	55.71	29.33	145.20	314.60
par. efficiency	1.00	0.99	0.99	0.99	0.98	0.97	0.81	0.37		
hist+enc (GB/s)	0.79	1.57	3.12	6.23	12.38	23.73	29.22	20.03	45.35	96.01

Table 6: Performance of multi-thread Huffman encoder on `Nyx-Quant`.

- encoding: 32-core throughput at 56 GB/s while GPU achieves 314.6 GB/s on V100 (5.6×)
- overall: 32-core throughput at 29.22 GB/s vs. GPU's at 96.01 GB/s (3.3×)

Performance Evaluation

WASHINGTON STATE UNIVERSITY · OAK RIDGE National Laboratory · Argonne NATIONAL LABORATORY

			avg. bits	#reduce	breaking	hist. GB/s TU	hist. GB/s V	codebook MS TU	codebook MS V	encode GB/s TU	encode GB/s V	hist+enc GB/s TU	hist+enc GB/s V
cuSZ	enwik8	95 MB	5.1639	-	-	102.5	252.4	1.375	1.635	10.1	12.2	8.2	9.8
	enwik9	954 MB	5.2124	-	-	108.2	259.6	1.382	1.640	7.2	11.3	6.8	10.8
	mr	9.5 MB	4.0165	-	-	36.2	86.5	1.565	1.831	9.6	15.2	3.5	3.8
	nci	32 MB	2.7307	-	-	66.1	150.6	0.706	1.027	8.6	14.9	6.6	9.6
	Flan_1565	1.4 GB	4.1428	-	-	104.2	256.6	0.758	0.950	8.5	10.7	7.8	10.2
	Nyx-Quant	256 MB	1.0272	-	-	74.8	197.7	3.416	3.804	17.7	29.7	12.1	18.9
Ours	enwik8	95 MB	5.1639	2 (4×)	0.034915%	102.8	252.0	0.594	0.707	42.2	94.0	25.4	46.1
	enwik9	954 MB	5.2124	2 (4×)	0.021747%	108.1	276.1	0.626	0.666	49.7	94.6	34.0	70.6
	mr	9.5 MB	4.0165	2 (4×)	0.000174%	36.2	99.0	0.300	0.312	42.0	76.8	12.3	18.4
	nci	32 MB	2.7307	3 (8×)	0.152880%	56.4	169.1	0.507	0.514	63.7	154.8	20.6	36.1
	Flan_1565	1.4 GB	4.1428	2 (4×)	nearly 0%	103.5	274.7	0.314	0.327	50.0	94.9	33.5	69.5
	Nyx-Quant	256 MB	1.0272	3 (8×)	0.003277%	74.8	197.6	0.449	0.544	145.2	314.6	45.4	96.0

Table 7: reakdown comparison of Huffman performance on tested datasets. Gathering time is excluded.

- mostly 4-plus bits (vs. uncompressed 8 bits), leading to a relatively low compression ratio.
- `nci` and `Nyx-Quant` can use $r = 3 \rightarrow$ over 100 GB/s. Small `nci` is difficult to saturate memory bandwidth. Higher-compression-ratio `Nyx-Quant` (2.66×) has less writing effort, reaches 314.6 GB/s.
- Comparing to coarse-grained encoder, there is 3.1× to 5.0× on RTX 5000, and 3.8× to 6.8× on V100.

Conclusion

WASHINGTON STATE UNIVERSITY · OAK RIDGE National Laboratory · Argonne NATIONAL LABORATORY

In this work,

- We propose and implement an efficient Huffman encoder for NVIDIA GPU architectures, including
- an efficient parallel codebook construction and a novel reduction based encoding scheme,
- and we implemented a multithread Huffman encoder for a fair comparison.
- Our solution can improve the parallel encoding performance up to 5.0× on RTX 5000, 6.8×on V100, and 3.3× on CPUs.

Future work,

- tune the performance for low-compression-ratio data
- explore more efficient gathering methods
- explore how intrinsic data feature affects the compression ratio and the throughput

Acknowledgement (ECP)

This R&D was supported by the Exascale Computing Project (ECP), Project Number: 17-SC-20-SC, a collaborative effort of two DOE organizations—the Office of Science and the National Nuclear Security Administration, responsible for the planning and preparation of a capable exascale ecosystem. This repository was based upon work supported by the U.S. Department of Energy, Office of Science, under contract DE-AC02-06CH11357, and also supported by the National Science Foundation under Grants SHF-1617488, SHF-1619253, OAC-2003709, OAC-1948447/2034169, and OAC-2003624.

THANK YOU ‖

ANY QUESTION? 🧐

github.com/szcompressor/huffre (to be updated)

contact us Jiannan Tian jiannan.tian@wsu.edu
Dingwen Tao dingwen.tao@wsu.edu

EXASCALE COMPUTING PROJECT

NSF

WASHINGTON STATE UNIVERSITY

OAK RIDGE National Laboratory

Argonne NATIONAL LABORATORY