UNPUBLISHED ORIGINAL RESEARCH

USING CONFIDENTIAL DATA

Co-Design **pSZ/cuSZ** for SLAC **LCLS II**

Jiannan Tian
Ph.D. Candidate

Dr. Dingwen Tao
Associate Professor

Luddy School of Informatics, Computing, and Engineering
Indiana University Bloomington

April 27, 2024

FZ Meeting Excerpt (02/15/24)

Real-Time Data Reduction

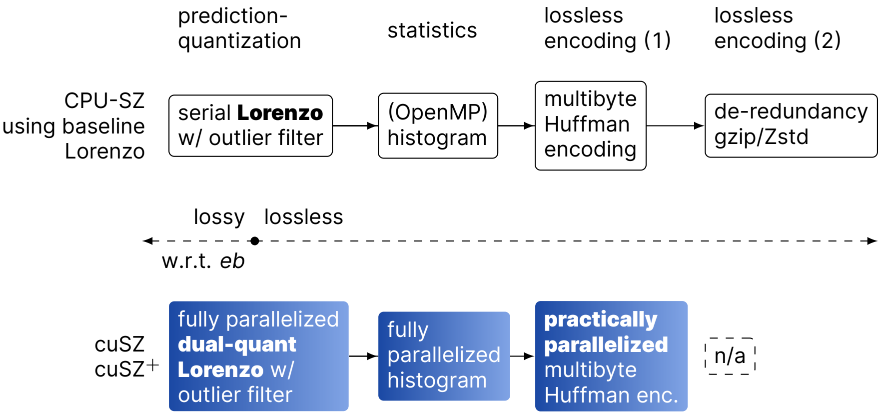
- Aim for 10x REALTIME compression (while data is being acquired)
  - Mandated by LCLS management to save $$$
  - DRP: “Data Reduction Pipeline” (part of DAQ)
- ∼**1 TB/s** raw data in 2026
- First data reduction: **high-speed-digitizer FPGA thresholding** to save peaks
  - LCLS-II-HE will use more software reduction
- In addition to reduced data we **also save a small fraction of raw data** to check effect of reduction algorithms on physics results

Performance Implications for Largest Camera

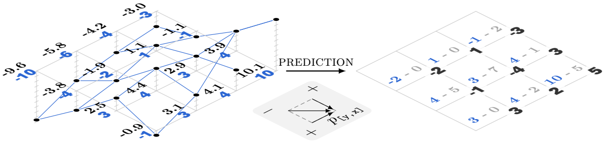
- Estimated timescale: 2026
- Simple-minded scaling to **35kHz 16Mpx EpixUHR camera** (∼1TB/s raw data, ∼2TB/s calibrated data, ∼50x more data than TXI camera) suggests ∼**1000 CPU nodes required**
  - feels unmanageable, even with some Moore's law scaling
  - investigate GPUs as an option for reducing node count
  - We will DMA data directly from FPGA to GPU using GPUDirect.
  - Hoping for **30GB/s to 50GB/s with 10x reduction in 1 GPU**
    - means we need 40 to 70 GPUs for this big camera

pSZ/cuSZ Basics

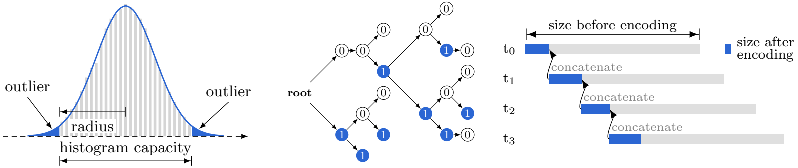
CPU-SZ vs cuSZ



cuSZ Details



(a) cuSZ uses dual-quant to quantize the input data and generate prediction “delta”.



(b) Encoding: (*left*) histogram of prediction “delta”; (*middle*) build Huffman tree/codebook based on the histogram on CPU; (*right*) encode and concatenate to output.

Figure: The system overview of cuSZ.

cuSZ Takeaways

- Predictor decides data compressibility w.r.t. *eb*, the degree to which the data can be effectively encoded to reduce size.
- The longer lossless steps tend to result in higher encoding effectiveness.
  - i.e., outcome: high compression ratio.
  - Mainly two types: ① entropy (Huffman) encoding, using fewer bits to represent more frequent symbols, used in cuSZ ② de-redundancy encoding, e.g., `0000000000000000` → `(0,16)`, not used in cuSZ.
- However, more steps slow down the processing, esp. when it results in more kernels (moving data in and out of global memory).
- Besides, CPU events can interrupt the GPU “stream” of processing, e.g., building the Huffman tree.

Objectives

Short-Term Objectives

The short-term improvements are backed by published papers and implemented prototypes.

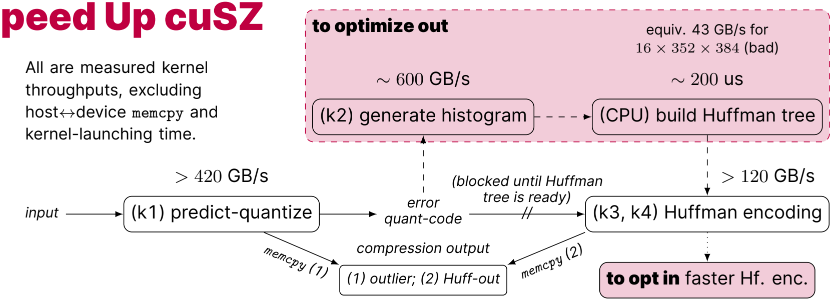
- Fixed Huffman tree that is approximated offline rather than runtime-desired
  - **Inspired** by Shah et al. (ICS ’23), *Lightweight Huffman Coding for Efficient GPU Compression*.
  - Some assumptions do not hold all the time, e.g., Gaussian/Cauchy-like distribution of prediction error.
  - **The preliminary study in the following section addresses the feasibility.**
- Faster Huffman encoding: Tian et al. (IPDPS ’21), *Revisiting Huffman*.
  - As of late April 2024 re-implementation in progress; initially integrated.
- We will discuss the long-term planning after these are delivered.

Setup

- CONFIDENTIAL DATA `epix_1000`
- Data interpretation: 1000 `float32` snapshots of $(z, y, x) = (16, 352, 384)$
- i.e., 1000 8,650,752-byte (8.7-MiB) 3D data.
- Use generic Lorenzo prediction
  - absolute-mode error bound of 3
  - Use 3D prediction internally
  - expect $\sim 10\times$ in compression ratio **(CR)**
- Expectation of **the new study**: (conditionally) use prebuilt Huffman tree to improve end-to-end performance without sacrificing compression ratio.

To Speed Up cuSZ

All are measured kernel throughputs, excluding host↔device `memcpy` and kernel-launching time.



to optimize out

equiv. 43 GB/s for
$16 \times 352 \times 384$ (bad)

~ 600 GB/s

~ 200 us

(k2) generate histogram

(CPU) build Huffman tree

> 420 GB/s

(blocked until Huffman tree is ready)

> 120 GB/s

input

(k1) predict-quantize

error quant-code

(k3, k4) Huffman encoding

compression output

`memcpy` (1)

(1) outlier; (2) Huff-out

`memcpy` (2)

to opt in faster Hf. enc.

- The prebuilt tree entails ① prebuilding a repository of trees to pick from, or ② a **aggregated** value to determine which prebuilt tree to use.
  - The aggregated value can be entropy to optimize out (CPU)-build, or
  - top-1 frequent, `hist[0]` (even simpler) to optimize out both (CPU)-build and (k2). Getting `hist[0]` can be integrated in (k1).
- Prebuilding trees results in a more thorough compressibility study.
- Can opt-in faster Huffman Encoding.

Compressibility Study for Prebuilding Huffman Tree (February 2024 Study)

Observation of Quant-Code (Prediction “Delta”)

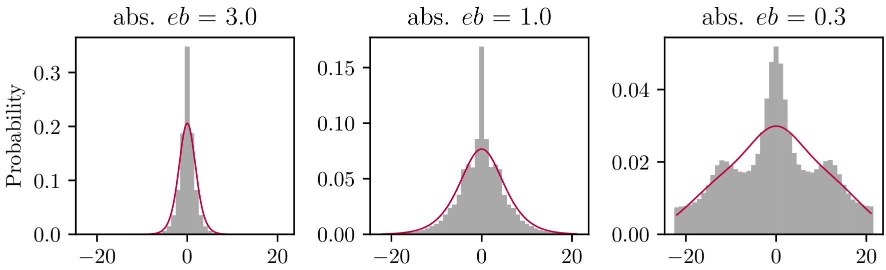


Figure: Changed prediction error distribution in response to the error bounds.

- At abs. *eb* $= 3.0$, we can treat the dist. as Gaussian-like: ① zero-mean, ② std. dev. changing in response to *eb*, and ③ `hist[abs(i)]` $>$ `hist[abs(i+1)]` (decreasing).
- It **cannot** be generalized for **all** error bounds. Because of the noise-like pattern, the prediction with smaller *eb* results in rises beyond top-*k* (e.g., 5 for *eb* $= 1.0$, 8 for *eb* $= 0.3$).

Optimize Out CPU Buiding Huffman Tree

- By studying 1000 snapshots, a repository of prebuilt trees can be determined.
- Huffman coding is known to be closely related to entropy. With the mentioned Gaussian-like dist. detected, a close approximation (theoretically) can be made.
- **caveat**: intrinsically, Huffman does not handle very-high-CR cases: extra effort to interpret when entropy < 1.0.

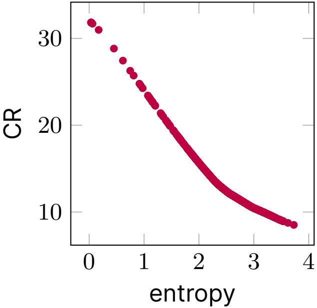


Figure: Deterministic entropy-CR relationship.

(Optional) Statistical Study

The Gaussian-like (Cauchy) dist. is more important than what is just shown in the prev. slide.

- tempting to generalize when the error bound is large, e.g., abs. $eb = 3$.
- implies the capability of approximating the precise Huffman tree.
- The most possible item is in the center, i.e., `hist[0]`, where 0 indicates most prediction errors are within $(-eb, eb)$.

We could utilize certain criteria to determine how a runtime prediction error distribution is close to a Gaussian-like.

- need to know the limitation of the following setup.

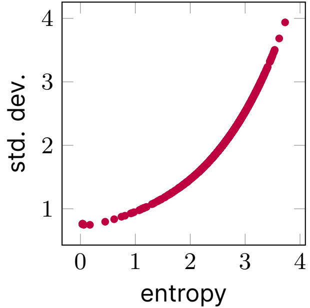


Figure: Offline study based on `epix_1000` at abs. $eb = 3$. The std. dev. The generation script is in the next slide for result replication purposes.

(Optional) Note; Result Replication

```
# +/- 20 is sufficient for abs. eb=3
bin_borders = np.arange(0-512, 1024-512)[512-20:512+20]
bin_heights = freq_array[512-20:512+20]
ax = sns.histplot(x=bin_borders, weights=bin_heights, kde=True,
    discrete=True); plt.close() # suppress plot display
kde_line = ax.lines[0]  # extract KDE line
kde_x, kde_y = kde_line.get_data()
kde_y_01 = kde_y / (np.sum(kde_y) * np.diff(kde_x[:2])) # normalize
skewness = skew(kde_y_01) # Pearson's def: fisher=False
kurtosis = kurtosis(kde_y_01, fisher=True)
# add 3 to compare with normal dist. kurtosis
print(f"(skewness,kurtosis)=({skewness},{kurtosis+3})")
# initial parameter estimates
A = np.max(kde_y)
mean = np.average(kde_x, weights=kde_y)
stddev = np.sqrt(np.average((kde_x-mean)**2, weights=kde_y))
```

Further Optimize Out Histogram Kernel

- Gaussian-like dist., $\leftrightarrow$ prebuilt trees $\leftrightarrow$ top-1 freq.
- **expected runtime**: binary search in a sorted list of $(k, v) =$ (top-1 freq, tree) to determine a tree.
- needs to count top-1 quant-code in the predict-quantize kernel
  - prototyped and initially profiled in 2023 September (`769ec5c`)
  - implies more work kernel optimization
- **caveats:** 1) we don't need 1000 trees; 2) high-CR zone needs interpolation.

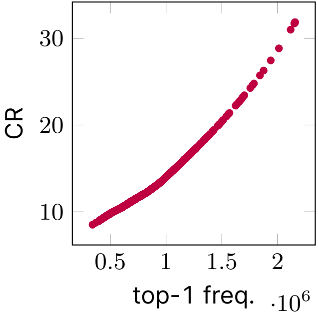


Figure: Based on abs. $eb = 3$, top-1 error quant-code. (`hist[0]`) exhibits a one-on-one mapping to CR.

(Very Coarse) Preliminary Study in CR Change

- If staying in the comfort zone (e.g., similar data, abs. $eb = 3$), just use the database of prebuilt trees.
  - non-parametric, only smoothening the existing trees
  - (This work is not machine-learning.) like using training data to verify.
- If not staying, we can prebuild trees based on Gaussian-like distribution (parametric) with stricter criteria.

```
(estimated vs precise) CR error,   0-precentile:  0.000262%
(estimated vs precise) CR error,   1-precentile:  0.000535%
(estimated vs precise) CR error,  25-precentile:  0.001380%
(estimated vs precise) CR error,  50-precentile:  0.002107%
(estimated vs precise) CR error,  75-precentile:  0.005045%
(estimated vs precise) CR error,  99-precentile:  0.016773%
(estimated vs precise) CR error, 100-precentile:  0.020878%
```

Conclusion: the specialized study **implies** a minimum loss in CR by using prebuilt trees.

WIP

- implementation, mostly focusing on handling outlier quantization codes of statistically low occurrence
- planned interface: `--import-prebuilt-tree /path/to/trees` (if CLI)
- fuse partial histogram (Top-K/Top-1) into predict-quantize kernel
  - integrate the existing `769ec5c`
- end-to-end performance profiling
  - There are unidentified reasons for the longer kernel-launching time in the new prototype.

Performance Profiling

End-To-End Performance (March 2024 Study)

- The end-to-end data processing is measured using CPU timer quickly.
  - `std::chrono::high_resolution_clock`
  - typically accurate or longer than the most accurate GPU timer (onerous to set up).
  - excl. memory allocation (considered as one-time).
- Larger batch size results in higher processing throughput until saturated.
  - a single snapshot is $384 \times 352 \times 16$, or 8.25 MiB, hard to saturate the hardware resource.
  - As a reference, NVIDIA A100 GPU has 40 MiB L2 cache.

End-To-End Performance (March 2024 Study)

- The end-to-end data processing is measured using CPU timer quickly.
- Larger batch size results in higher processing throughput until saturated.

batch size input size (`float32`)	3 $z = 48$	6 $z = 96$	12 $z = 192$	25 $z = 400$	50 $z = 800$	100 $z = 1600$
input MiB	24.75	49.50	99.00	206.25	412.50	825.00
Lorenzo pred-quant	644.42	674.78	749.32	904.98	1221.43	1859.81
histogram	25.56	21.82	21.76	22.96	23.19	24.23
enc (include CPU)	1138.51	1543.47	2335.95	4077.52	8202.68	15353.70
Huff CPU time	221.75	207.55	209.52	216.45	214.99	223.98
archive/`memcpy` on device	111.36	108.46	110.09	110.91	111.69	126.19
end-to-end μs	1919.86	2348.53	3217.13	5116.37	9559.00	17364.00
end-to-end GiB/s	12.59	20.58	30.05	39.37	42.14	46.40

Table: Initial end-to-end performance on `epix_1000`, whcih contains 1000 snapshots of size $(x, y, z) = (384, 352, 16)$. Larger batch sizes result in higher processing throughput until saturated.

Quick Takeaway

- Eliminating Huffman encoding at the current time will not significantly increase the end-to-end data processing throughput.
- Because Huffman encoding (coarsely parallelized)is the current bottleneck.
- Tian et al. (2021) IPDPS proposed faster Huffman encoding.
    - need re-implementing and integrating
- (Estimating based on small input sizes is not accurate.)

batch size	12	25	50	100
input MiB	99.00	206.25	412.50	825.00
e2e GiB/s, measured	30.05	39.37	42.14	46.40
e2e GiB/s, if excl. Huff CPU	32.15	41.11	43.11	47.00
e2e GiB/s, if double Huff enc	49.72	67.83	75.29	84.14

Table: If simply taking out CPU Huffman time, the end-to-end performance will not be significantly imporived.

(To Be Updated) Initial Integration of Faster Hf-Enc.

(Need to stabilize and verify the integration of faster Huffman encoding.)

(Extended Topic) Process Multiple Snapshot at Once

- However, an unlimitedly large batch size does not fit the postanalysis pipeline.
  - Decompression overhead + hard to select the desired snapshot.
- Overall, we need a holistic solution.
  - Make compressed data archive queryable
    → more metadata to enable random access to arbitrary snapshots
- 4/26/24 In one application, 5k-by-5k (16M pixel, 64 MB) can be right in the comfort zone.