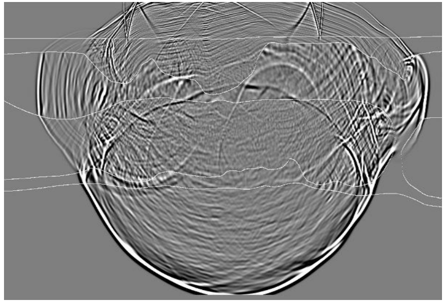
cuSZp2: A GPU Lossy Compressor with Extreme Throughput and Optimized Compression Ratio

Yafan Huang, Sheng Di*, Guanpeng Li, Franck Cappello

Big Data Issue in Modern HPC Systems

- Modern HPC systems generate **massive data** volumes at **rapid speeds**.
  - **High memory footprint**: Seismic imaging methods.
  - **Intensive data streams**: X-ray source applications.
  - **Expensive data movement overheads**: Large Language Model (LLM) training.



Reverse Time Migration[1]

2.8 PB Simulation Data

10x10x8 km3 per Snapshot

LCLS-II, 2024[2]

250 GB/s ~ 1TB/s

Data Generation Speed

LLaMA LLM, 2023[3]

2,048 x A100 GPUs

1.3 T Tokens & 65 B Parameters

[1] [Reverse Time Migration Technology] https://www.seismiccity.com/RTM.html
[2] [LCLS-II @ SLAC] https://lcls.slac.stanford.edu/lcls-ii
[3] [LLaMA @ Meta] https://llama-2.ai/llama-2-model-details/

In-situ Data Compression

- **Directly** compress/decompress data where it is generated/processed.
  - **CMP** and **DEC** here refers to “compression” and “decompression”.

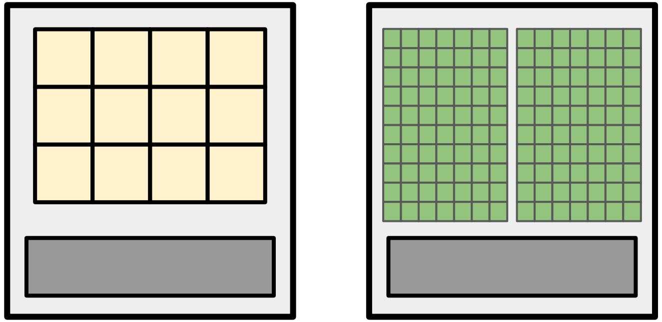
Prev. HPC Simulation or LLM Training | CMP → *Only Preserving Compressed Data in Runtime* → DEC | Succ. HPC Simulation or LLM Training

- **Two key requirements** for in-situ data compression tasks.
  - **Throughput:** compression and decompression speed, the faster the better.
  - **Compression ratio:** ori. data size/cmp. data size, the higher the better.

We need a fast and high-ratio compressor, but how?

GPU Lossy Compression

- ***GPU*** **Lossy Compression** excels in in-situ compression tasks due to:
  - ***GPU***: Entensive parallelism makes **high throughput** possible.
  - **Lossy Compression**: Offers much **higher compression ratio** than lossless ones.



CPU GPU ✔

Ori. Data Lossless Cmp. Lossy Cmp. ✔

Encoding phase of SZx[1]
~10 GB/s CPU vs ~200 GB/s GPU

Make such “lossy” acceptable to HPC by **error-bounded**.

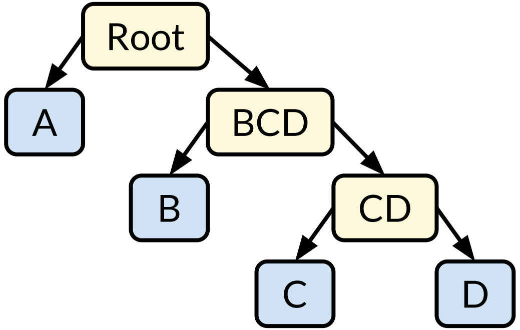
[1] [HPDC’22] Ultrafast Error-bounded Lossy Compression for Scientific Datasets

Challenge 1: Parallel Architecture Constraints

- **GPU parallelism** drastically complicates the compression algorithm designs.
  - Taking ***Huffman Encoding*** as an example. Given string array AAABBCD.

A	3
B	2
C	1
D	1

Step1:
Generating Frequency Table



Step2:
Constructing Huffman Tree

A	0
B	10
C	110
D	111

Step3:
Building Huffman Codebook

AAABBCD

↓

0001010110111111

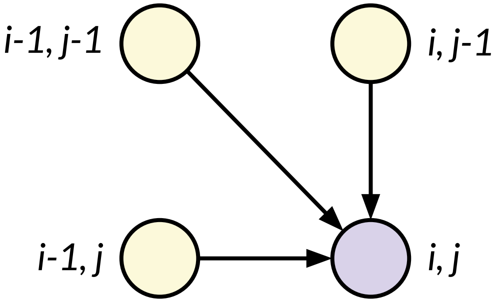
Step4:
Compression

One table/tree/codebook for entire array, requiring frequent global synchronization.

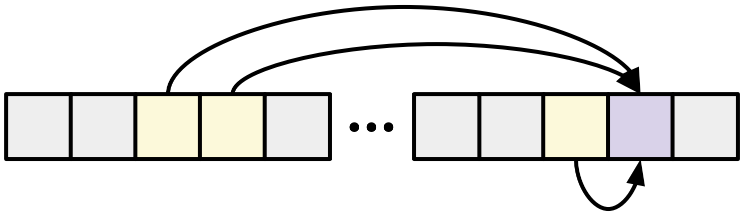
Easy to parallel, but read/write conflict exists.

Challenge 2: Complex Memory Access Patterns

- Unlike CPU, GPU is highly sensitive to **memory access** behaviors.
  - Taking a simple ***2D-Lorenzo Prediction*** as an example.



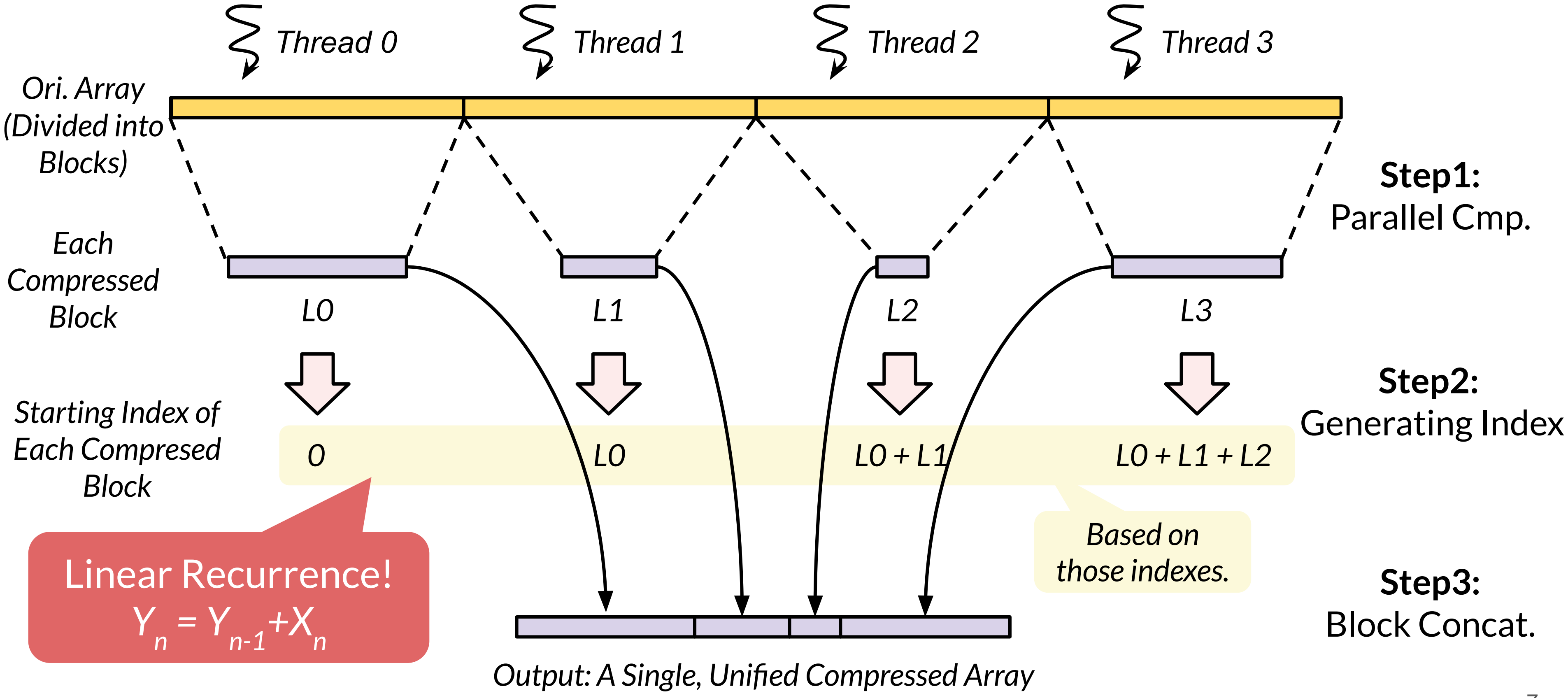
Use 3 spatially adjacent data points to predict current one.



Computer system stores all arrays in 1D manner.
Not adjacent anymore!

Strided memory access, drastically reducing throughput.

Challenge 3: Latency in Resolving Linear Recurrence



Limitations of Existing Solutions

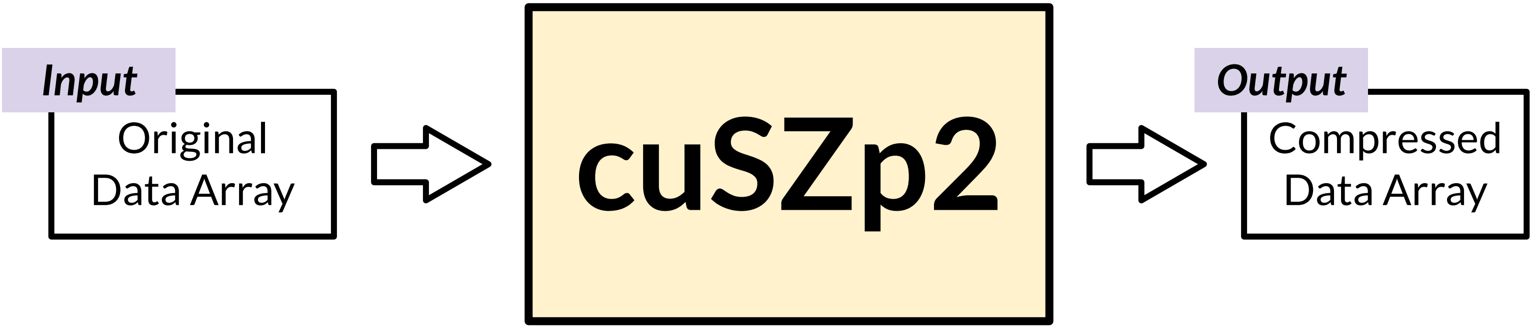
cuSZ, cuSZx, MGARD-GPU, etc.	Fail to address Challenge 1, using **CPU-GPU hybrid design** instead.
cuZFP, FZ-GPU, cuSZp, etc.	Fail to address Challenge 2, leading to **inefficient memory access** patterns.
FZ-GPU, cuSZp, etc.	Fail to address Challenge 3, resulting in **high latency** in block concatenation.

Wishlist

- Pure-GPU Design
- Efficient Memory Access
- Low Sync. Latency
- Error Control
- High CMP Ratio

Guaranteeing High Throughput

Our Solution: cuSZp2



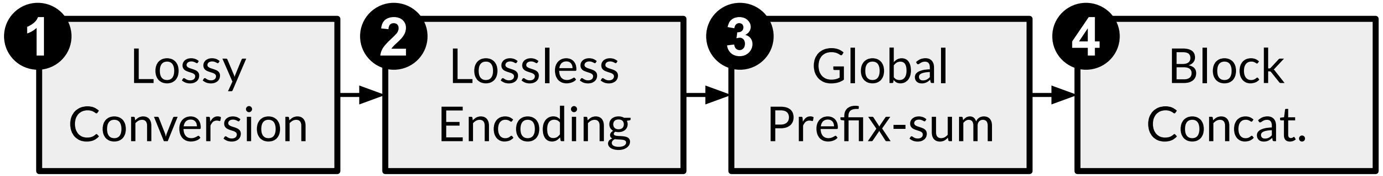
An Error-bounded GPU Lossy Compressor

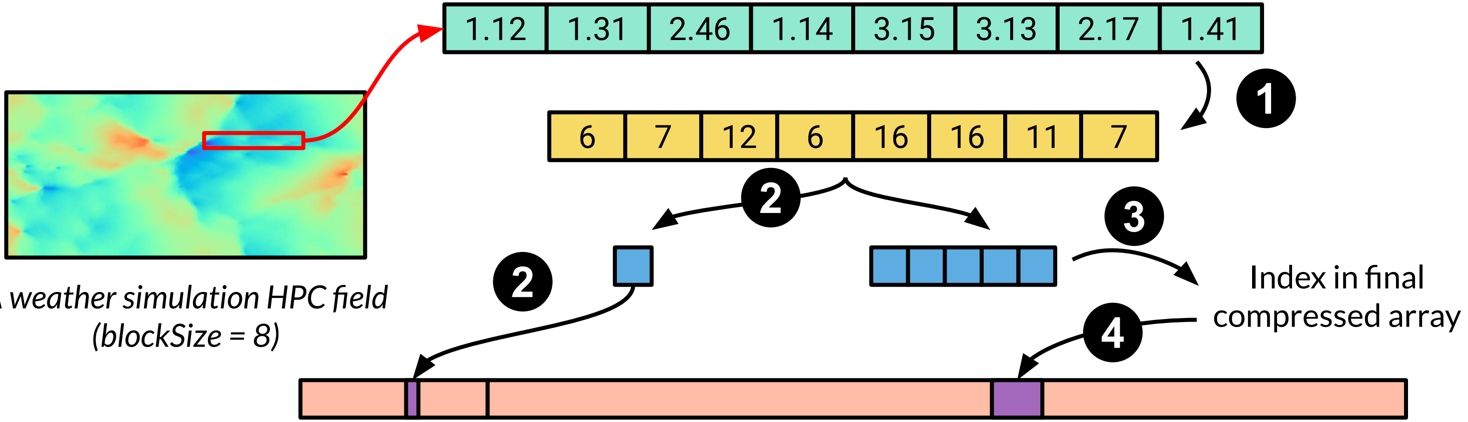
Key Features of cuSZp2

- Compression/decompression requires only **one GPU kernel function**.
- Highly efficient latency control and memory access patterns – **extreme throughput**.
- Two encoding modes, **high compression ratio** for different data patterns.
- Error-bounded lossy compression, ensuring **high reconstructed data quality**.

cuSZp2: Algorithm and Running Example

- cuSZp operates at **block granularity** and requires **four steps** to perform compression.



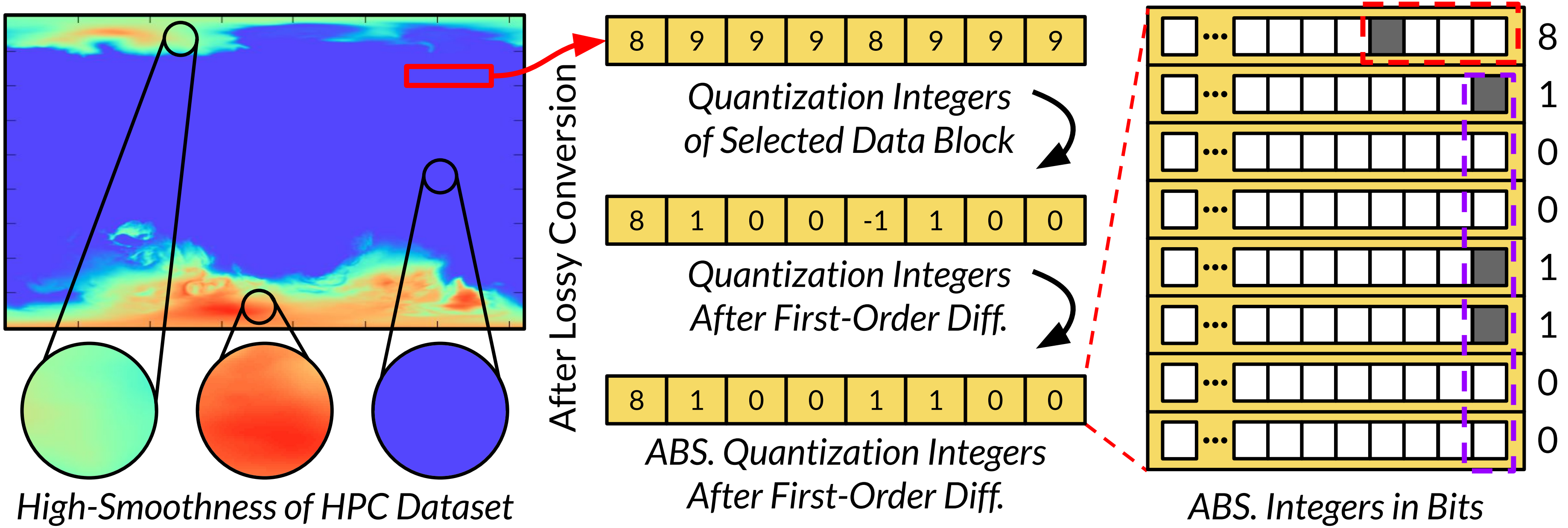


A weather simulation HPC field (blockSize = 8)

A running example to show how cuSZp2 compresses an HPC dataset.

cuSZp2: Lossless Encoding Method

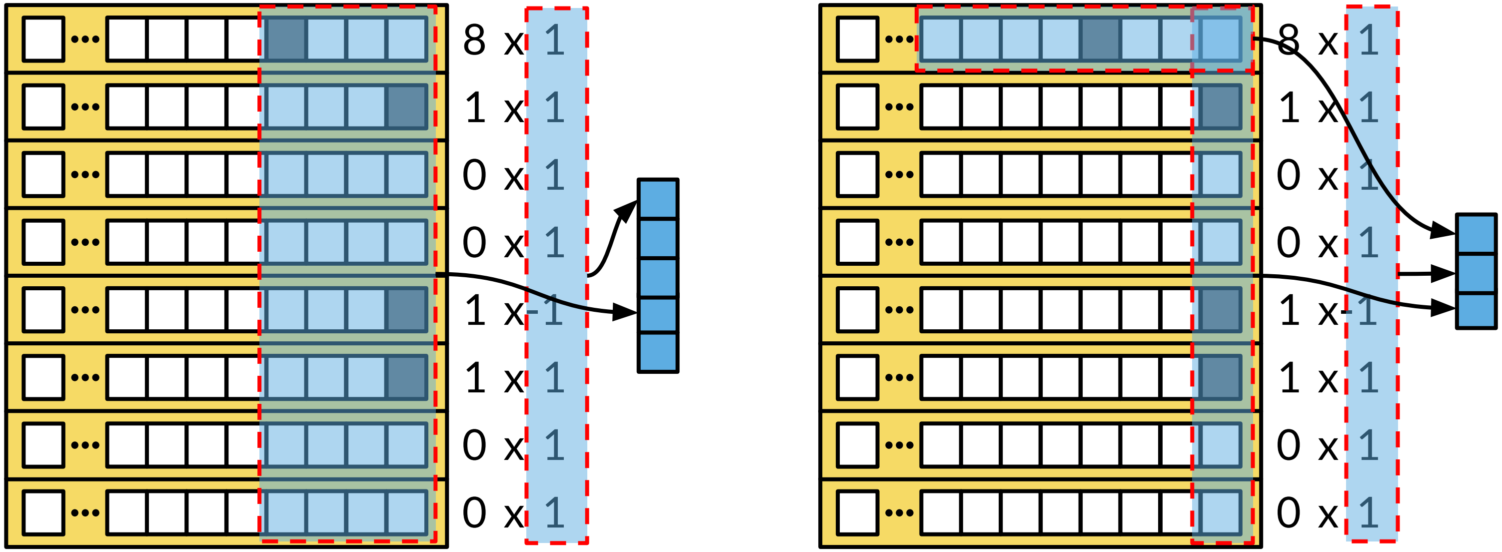
- **Motivation**: a natural **defect** in processing HPC fields by **parallel compressors**.



In blockwise parallel compressors, while processing smooth HPC fields, the first integer is usually much greater than the rest, making it an outlier.

cuSZp2: Lossless Encoding Method

- **Fixed-length encoding (FLE)** preserves the same number of bits for each integer.
  - In cuSZp2, both modes (**Plain-FLE** and **Outlier-FLE**) are preserved.



Plain-FLE
Compressed Size: ***5 bytes***

Outlier-FLE
Compressed Size: ***3 bytes***

cuSZp2: Memory Access Optimization

- **Intra-data-block**, **vectorizing** read/write operations from global memory.

```
void a_CUDA_kernel (float* array, ...) {
  for (int i=0; i<N; i++) {
    float var = array[i];
    ... // Operation for var
    array[i] = var;
  }
}
```

```
/*0080*/        LD.E R2, [R6]
...             ...
/*01a0*/        ST.E [R4], R2
```

***N* times**

vectorize

```
void a_CUDA_kernel (float4* array, ...) {
  for (int i=0; i<N/4; i++) {
    float4 var = array[i];
    ... // Operation for var.x
    ... // Operation for var.y
    ... // Operation for var.z
    ... // Operation for var.w
    array[i] = var;
  }
}
```

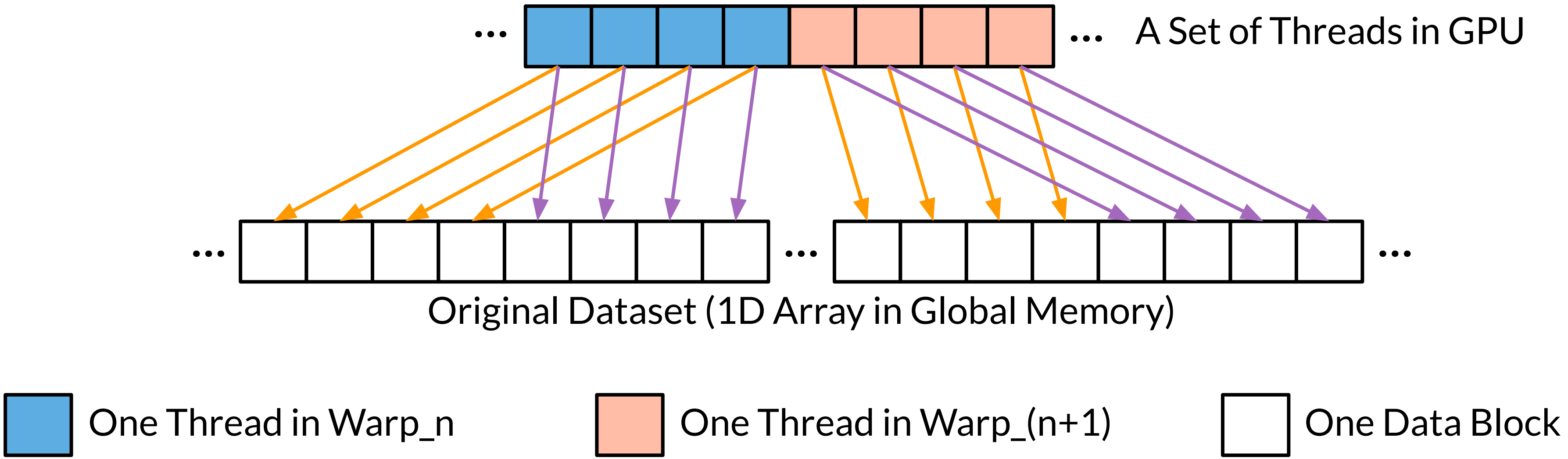
```
/*0098*/        LD.E.128 R4, [R10]
...             ...
/*04c2*/        ST.E.128 [R8], R4
```

***N/4* times**

Fixed-length encoding has balanced computation across each iteration inside a loop, making it suitable to unroll and vectorize.

cuSZp2: Memory Access Optimization

- **Inter-data-block**, enabling **coalcesing memory access** manners.

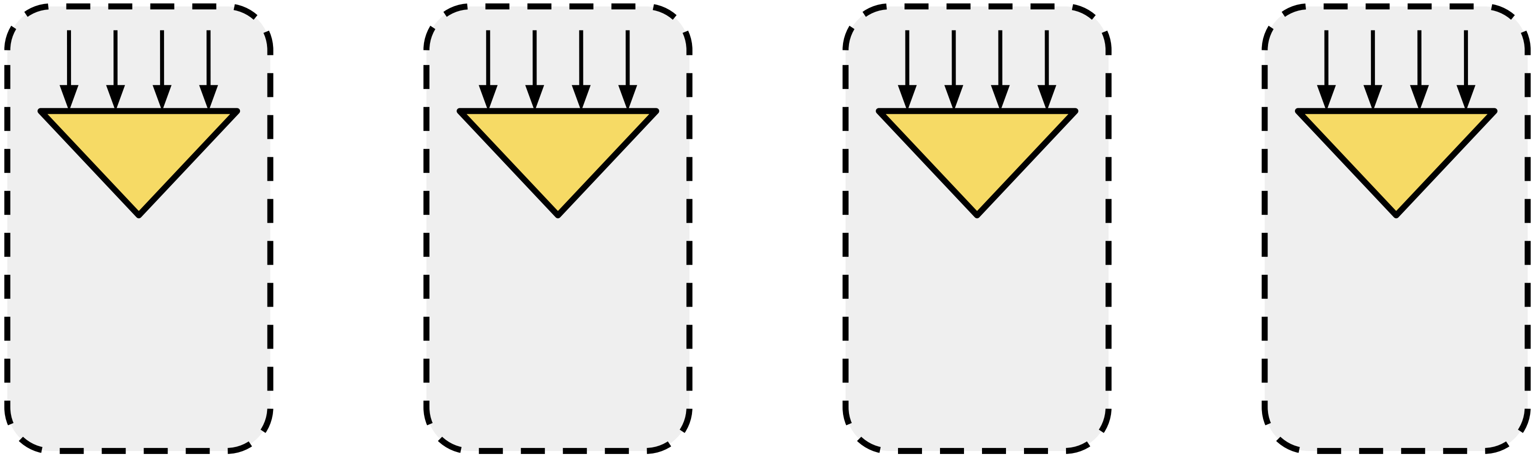


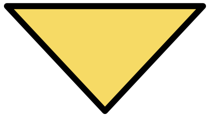
No matter how many data blocks one thread process, always enabling coalescing memory access!

cuSZp2: Synchronization Latency Control

- **Motivation**: High synchronization latency caused by **Serial chained-scan**.

Timestep: 0



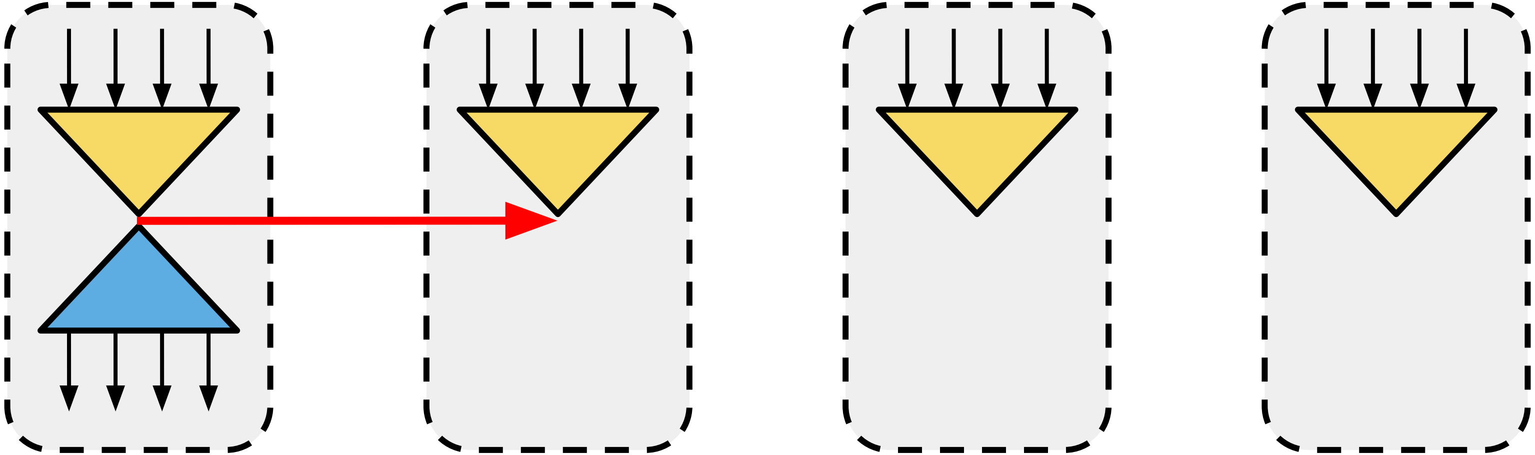


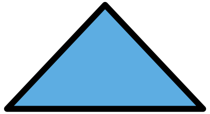
This is a reduce operation (i.e. add up all compressed block lengths) within a thread block

cuSZp2: Synchronization Latency Control

- **Motivation**: High synchronization latency caused by **Serial chained-scan**.

Timestep: 1



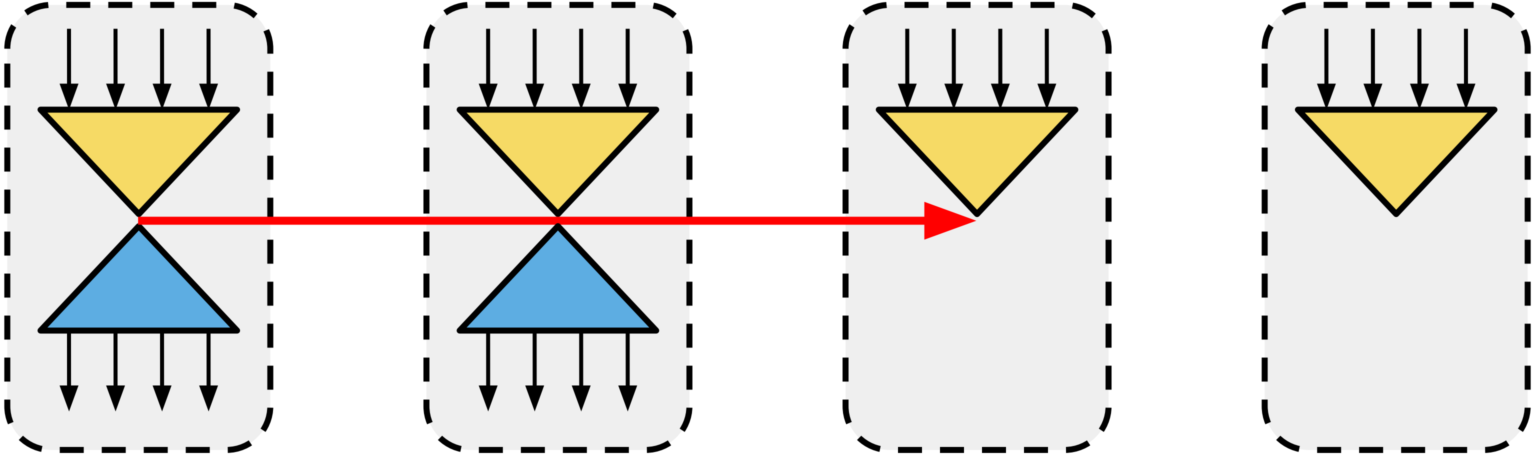


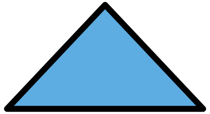
This is a scan operation (i.e. distribute its global location to each data block) within a thread block.

cuSZp2: Synchronization Latency Control

- **Motivation**: High synchronization latency caused by **Serial chained-scan**.

Timestep: 2



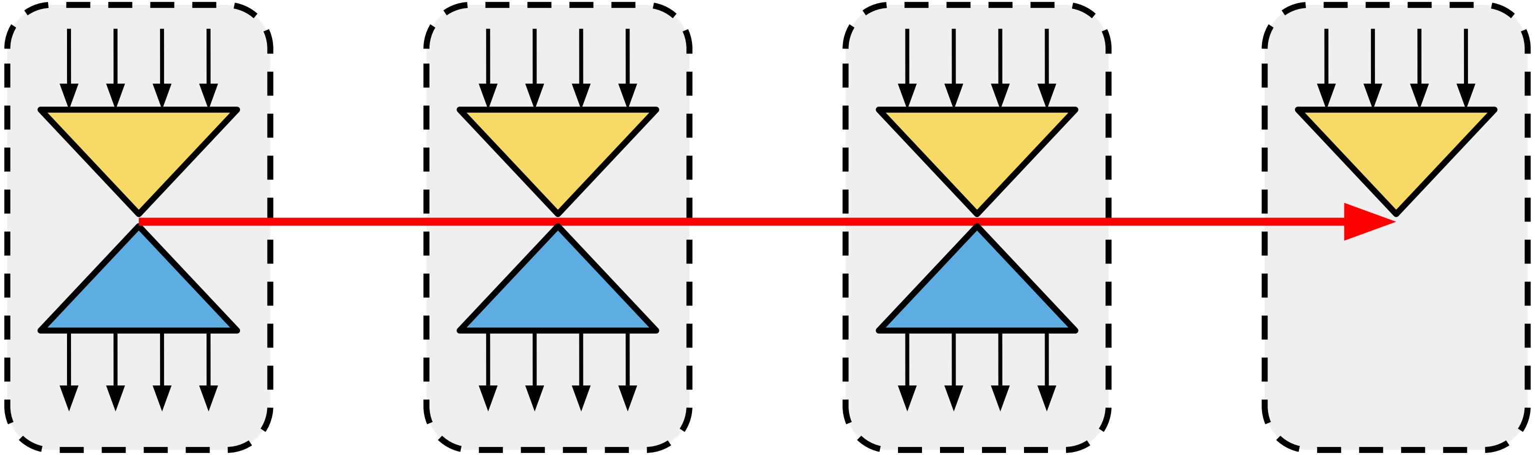


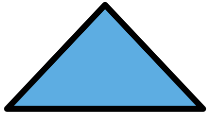
This is a scan operation (i.e. distribute its global location to each data block) within a thread block.

cuSZp2: Synchronization Latency Control

- **Motivation**: High synchronization latency caused by **Serial chained-scan**.

Timestep: 3



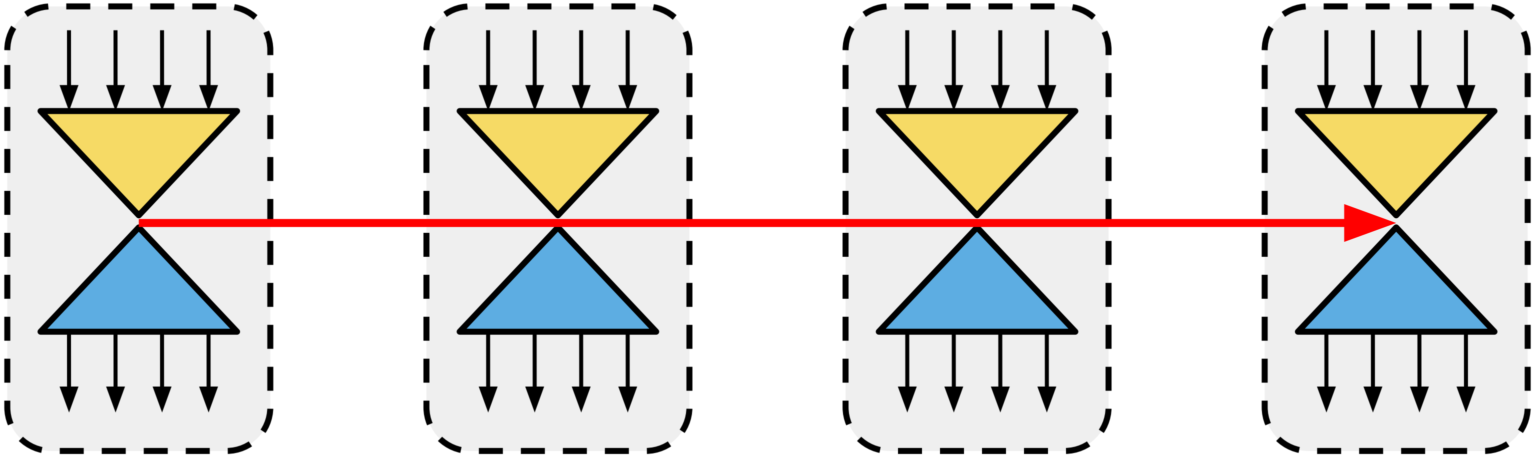


This is a scan operation (i.e. distribute its global location to each data block) within a thread block.

cuSZp2: Synchronization Latency Control

- **Motivation**: High synchronization latency caused by **Serial chained-scan**.

Timestep: 4

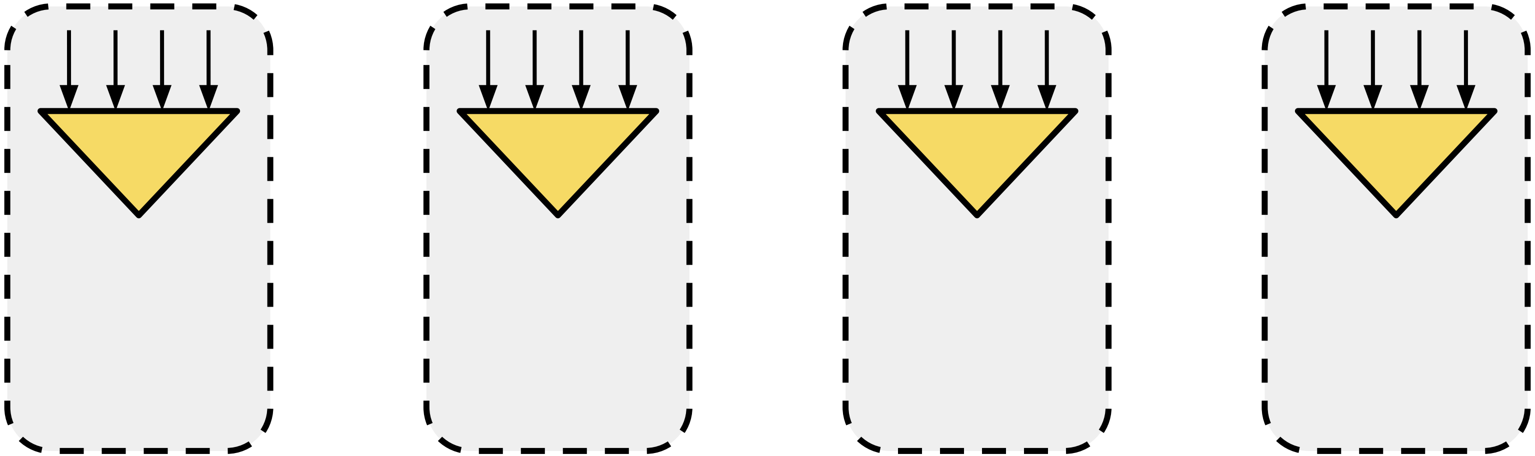


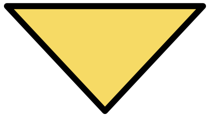
Every thread block must wait until its predecessor is finished!

cuSZp2: Synchronization Latency Control

- In cuSZp2, we control such latency by **decoupling the serial chained-scan**[1].

Timestep: 0





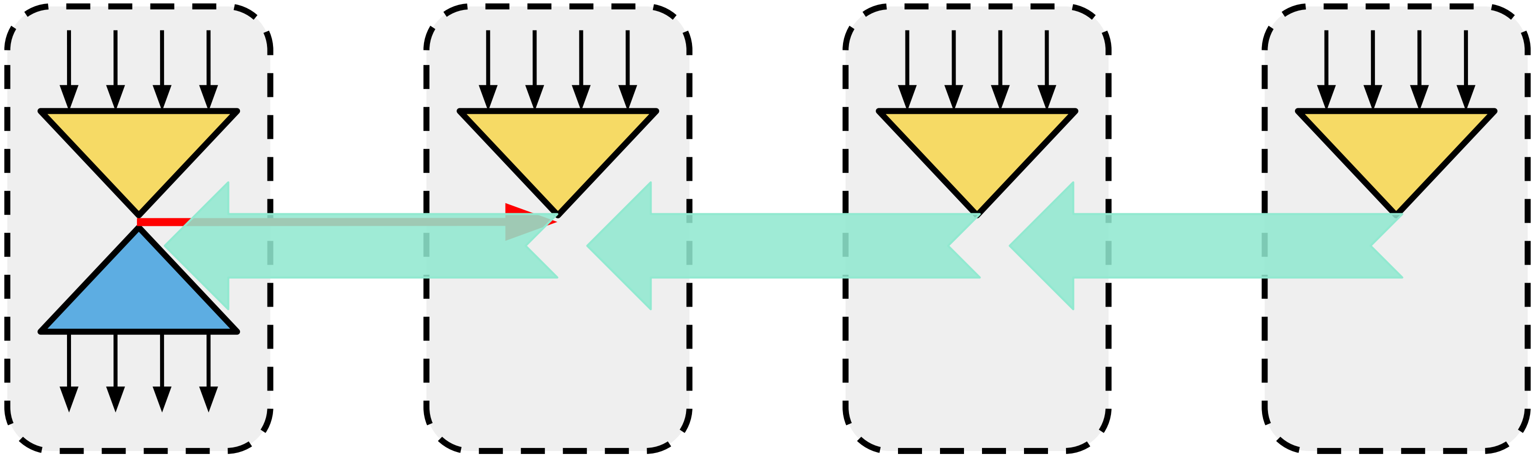
This is a reduce operation (i.e. add up all compressed block lengths) within a thread block

[1] [Nvidia Tech Report'16] Single-pass Parallel Prefix Scan with Decoupled Look-back

cuSZp2: Synchronization Latency Control

- In cuSZp2, we control such latency by **decoupling the serial chained-scan**.

Timestep: 1

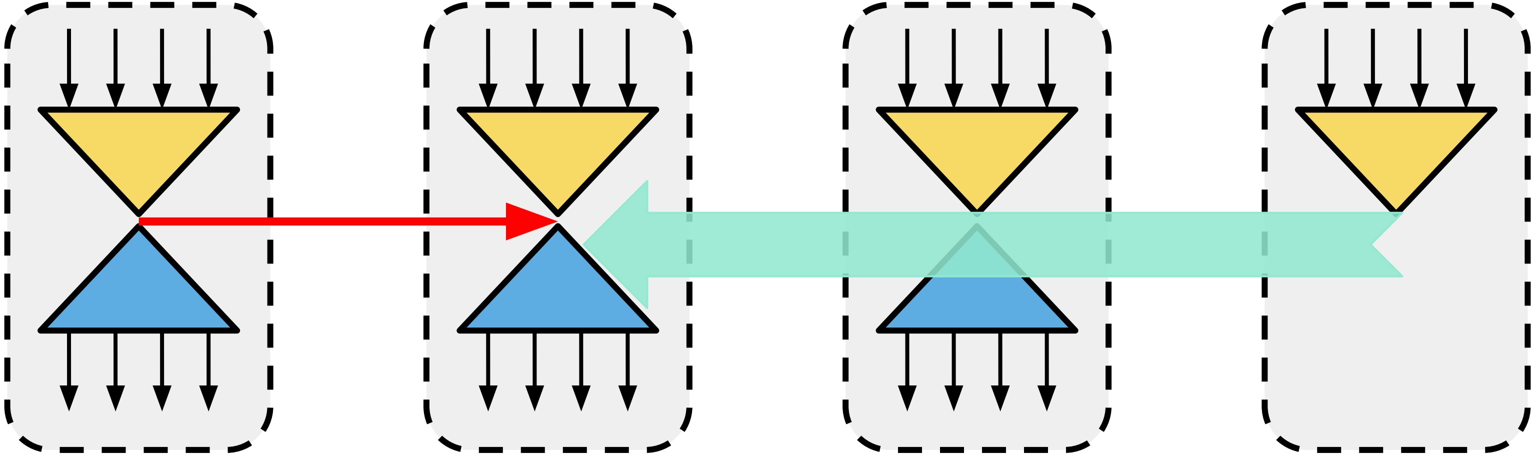


This is a lookback operation. When serial chained-scan not reached, every thread block aggregates its predecessors.

cuSZp2: Synchronization Latency Control

- In cuSZp2, we control such latency by **decoupling the serial chained-scan**.

Timestep: 2

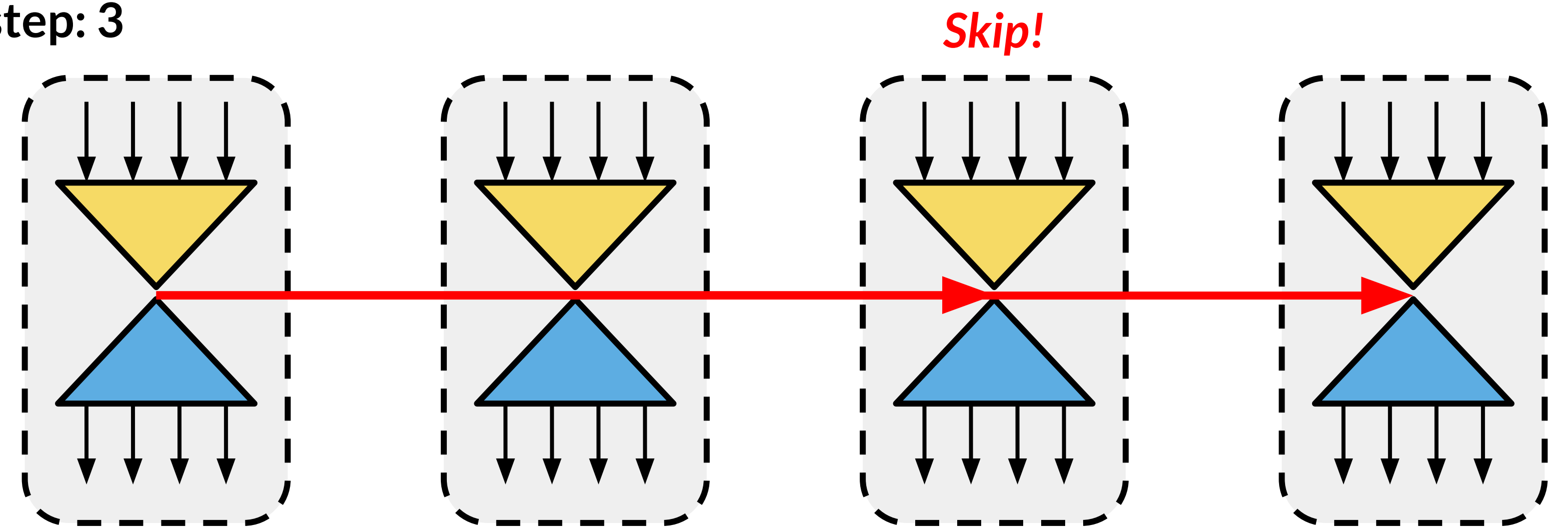


This is a lookback operation. When serial chained-scan not reached, every thread block aggregates its predecessors.

cuSZp2: Synchronization Latency Control

- In cuSZp2, we control such latency by **decoupling the serial chained-scan**.

Timestep: 3

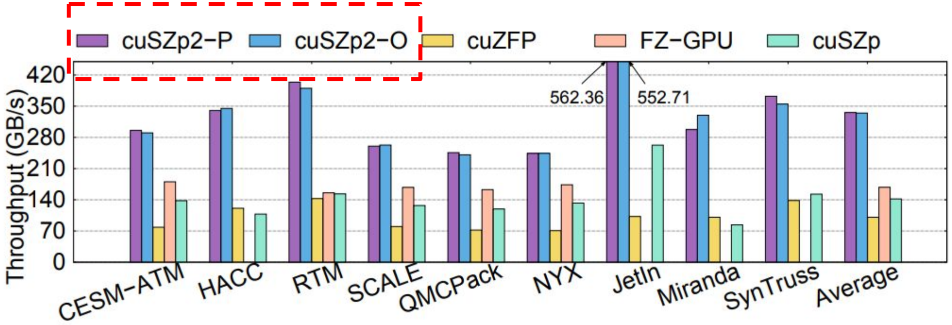


Hiding latency by making every thread block as busy as possible.

Evaluation: Single-Precision Datasets

- 9 real-world HPC datasets on one NVIDIA A100 GPU.

Datasets	Suite	Dims per Field	# Fields	Total Size
CESM-ATM [38]	SDRBench	3600×1800×26	33	20.71 GB
HACC [13]	SDRBench	1,073,726,487	6	23.99 GB
RTM [43]	SDRBench	1008×1008×352	3	3.99 GB
SCALE [60]	SDRBench	1200×1200×98	12	6.31 GB
QMCPack [39]	SDRBench	69×69×33120	2	1.17 GB
NYX [61]	SDRBench	512×512×512	6	3.00 GB
JetIn [62]	Open-SciVis	1408×1080×1100	1	6.23 GB
Miranda [63]	Open-SciVis	1024×1024×1024	1	4.00 GB
SynTruss [64]	Open-SciVis	1200×1200×1200	1	6.42 GB



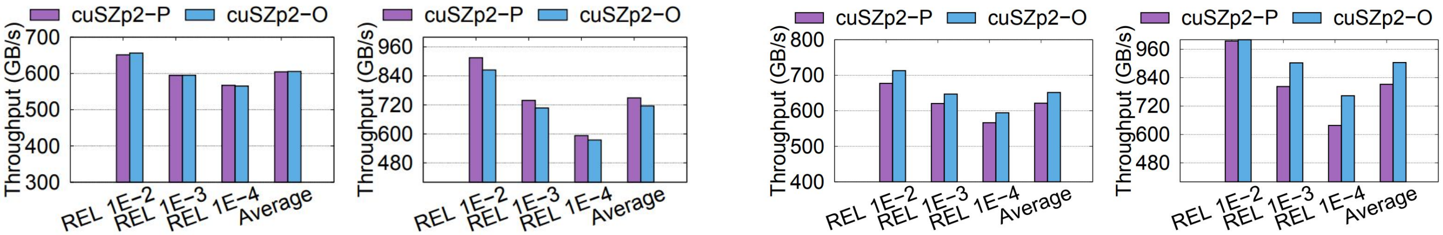
(c) Compression throughput with REL 1E-3 (Fixed-Rate 8 for cuZFP).

- On average, **~300 GB/s** and **~500 GB/s** for compression and decompression.
- **~2x** throughput than pure-GPU compressors, **~200x** throughput than hybrid ones.
- cuSZp2-O exhibit **the highest compression ratio** in almost all cases (24/27).
- This observation is consistent in other lower-end GPUs, such as RTX 3080/3090.

Evaluation: Double-Precision Datasets

- 2 real-world HPC datasets on one NVIDIA A100 GPU.

Datasets	Suite	Dims per Field	# Fields	Total Size
S3D [67]	SDRBench	$11 \times 500 \times 500 \times 500$	5	51.22 GB
NWChem [68]	SDRBench	801,098,891	1	5.96 GB

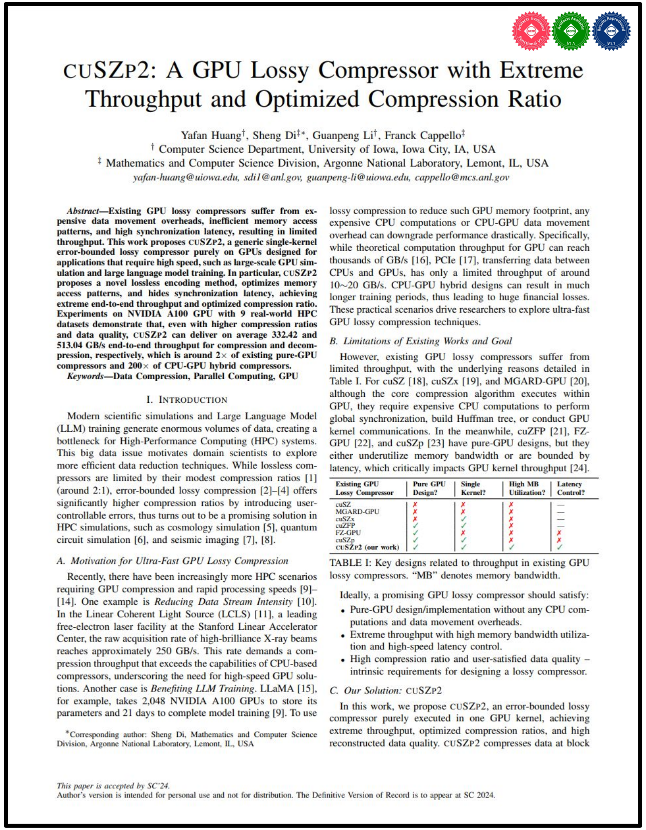


(a) NWChem compression (b) NWChem decompression (c) S3D compression (d) S3D decompression

- On average, **~500 GB/s** and **~700 GB/s** for compression and decompression.
- **Highest compression ratio** compared with all existing GPU lossy compressors.
- This observation is consistent in other lower-end GPUs, such as RTX 3080/3090.

More Design/Evaluation Details: Check Our Paper

- Ratio Profiling in Outlier-FLE.
- Compression ratio results.
- Memory utilization results.
- Global synchronization results.
- Data quality evaluation.
- cuSZp2-P vs cuSZp2-O.
- Other optimizations.



cuSZp2: A GPU Lossy Compressor with Extreme Throughput and Optimized Compression Ratio

Yafan Huang[†], Sheng Di[‡*], Guanpeng Li[†], Franck Cappello[‡]
[†] Computer Science Department, University of Iowa, Iowa City, IA, USA
[‡] Mathematics and Computer Science Division, Argonne National Laboratory, Lemont, IL, USA
yafan-huang@uiowa.edu, sdi1@anl.gov, guanpeng-li@uiowa.edu, cappello@mcs.anl.gov

***Abstract*—Existing GPU lossy compressors suffer from expensive data movement overheads, inefficient memory access patterns, and high synchronization latency, resulting in limited throughput. This work proposes cuSZp2, a generic single-kernel error-bounded lossy compressor purely on GPUs designed for applications that require high speed, such as large-scale GPU simulation and large language model training. In particular, cuSZp2 proposes a novel lossless encoding method, optimizes memory access patterns, and hides synchronization latency, achieving extreme end-to-end throughput and optimized compression ratio. Experiments on NVIDIA A100 GPU with 9 real-world HPC datasets demonstrate that, even with higher compression ratios and data quality, cuSZp2 can deliver on average 332.42 and 513.04 GB/s end-to-end throughput for compression and decompression, respectively, which is around 2× of existing pure-GPU compressors and 200× of CPU-GPU hybrid compressors.**

***Keywords*—Data Compression, Parallel Computing, GPU**

I. Introduction

Modern scientific simulations and Large Language Model (LLM) training generate enormous volumes of data, creating a bottleneck for High-Performance Computing (HPC) systems. This big data issue motivates domain scientists to explore more efficient data reduction techniques. While lossless compressors are limited by their modest compression ratios [1] (around 2:1), error-bounded lossy compression [2]–[4] offers significantly higher compression ratios by introducing user-controllable errors, thus turns out to be a promising solution in HPC simulations, such as cosmology simulation [5], quantum circuit simulation [6], and seismic imaging [7], [8].

A. Motivation for Ultra-Fast GPU Lossy Compression

Recently, there have been increasingly more HPC scenarios requiring GPU compression and rapid processing speeds [9]–[14]. One example is *Reducing Data Stream Intensity* [10]. In the Linear Coherent Light Source (LCLS) [11], a leading free-electron laser facility at the Stanford Linear Accelerator Center, the raw acquisition rate of high-brilliance X-ray beams reaches approximately 250 GB/s. This rate demands a compression throughput that exceeds the capabilities of CPU-based compressors, underscoring the need for high-speed GPU solutions. Another case is *Benefiting LLM Training*. LLaMA [15], for example, takes 2,048 NVIDIA A100 GPUs to store its parameters and 21 days to complete model training [9]. To use lossy compression to reduce such GPU memory footprint, any expensive CPU computations or CPU-GPU data movement overhead can downgrade performance drastically. Specifically, while theoretical computation throughput for GPU can reach thousands of GB/s [16], PCIe [17], transferring data between CPUs and GPUs, has only a limited throughput of around 10~20 GB/s. CPU-GPU hybrid designs can result in much longer training periods, thus leading to huge financial losses. These practical scenarios drive researchers to explore ultra-fast GPU lossy compression techniques.

B. Limitations of Existing Works and Goal

However, existing GPU lossy compressors suffer from limited throughput, with the underlying reasons detailed in Table I. For cuSZ [18], cuSZx [19], and MGARD-GPU [20], although the core compression algorithm executes within GPU, they require expensive CPU computations to perform global synchronization, build Huffman tree, or conduct GPU kernel communications. In the meanwhile, cuZFP [21], FZ-GPU [22], and cuSZp [23] have pure-GPU designs, but they either underutilize memory bandwidth or are bounded by latency, which critically impacts GPU kernel throughput [24].

Existing GPU Lossy Compressor	Pure GPU Design?	Single Kernel?	High MB Utilization?	Latency Control?
cuSZ	✗	✗	✗	—
MGARD-GPU	✗	✗	✗	—
cuSZx	✗	✓	✗	—
cuZFP	✓	✓	✗	—
FZ-GPU	✓	✗	✗	✗
cuSZp	✓	✓	✗	✗
cuSZp2 (our work)	✓	✓	✓	✓

TABLE I: Key designs related to throughput in existing GPU lossy compressors. "MB" denotes memory bandwidth.

Ideally, a promising GPU lossy compressor should satisfy:

- Pure-GPU design/implementation without any CPU computations and data movement overheads.
- Extreme throughput with high memory bandwidth utilization and high-speed latency control.
- High compression ratio and user-satisfied data quality – intrinsic requirements for designing a lossy compressor.

C. Our Solution: cuSZp2

In this work, we propose cuSZp2, an error-bounded lossy compressor purely executed in one GPU kernel, achieving extreme throughput, optimized compression ratios, and high reconstructed data quality. cuSZp2 compresses data at block

*Corresponding author: Sheng Di, Mathematics and Computer Science Division, Argonne National Laboratory, Lemont, IL, USA

This paper is accepted by SC'24.

To Use cuSZp API: C/C++

```
#include <cuSZp.h> // the only header needed

// For measuring the end-to-end throughput.
TimingGPU timer_GPU;

cuszp_type_t dataType = CUSZP_TYPE_FLOAT;           // or CUSZP_TYPE_DOUBLE
cuszp_mode_t encodingMode = CUSZP_MODE_PLAIN;       // or CUSZP_MODE_OUTLIER

// cuSZp compression.
timer_GPU.StartCounter(); // set timer
cuSZp_compress(d_oriData, d_cmpBytes, nbEle, &cmpSize, errorBound, dataType, encodingMode, stream);
float cmpTime = timer_GPU.GetCounter();

// cuSZp decompression.
timer_GPU.StartCounter(); // set timer
cuSZp_decompress(d_decData, d_cmpBytes, nbEle, cmpSize, errorBound, dataType, encodingMode, stream);
float decTime = timer_GPU.GetCounter();
```

- A unified API for float/double GPU array with different encoding modes.
- More specified APIs and some intrinsic features are also provided.

To Use cuSZp API: Python

```python
from pycuSZp import cuSZp

compressor = cuSZp()
# cuSZp compression.
start_time = time.time()                        # set cuSZp timer start
compressed_size = compressor.compress(
    ctypes.c_void_p(data.data_ptr()),           # Input data pointer on GPU
    ctypes.c_void_p(int(d_cmpBytes)),           # Output buffer on GPU
    data.numel(),                               # Number of elements
    1E-2,                                       # Set 1E-2 as error bound.
    data_type=0,                                # float 32, 1 for float64 (i.e. double)
    mode=0                                      # Plain mode, 1 for outlier mode
)
compression_time = time.time() - start_time     # set cuSZp timer end
```

- Slight performance degradation but still very fast in Python end-to-end usages.
- Both **Numpy Array** and **Torch Tensor** examples are provided in cuSZp repo.

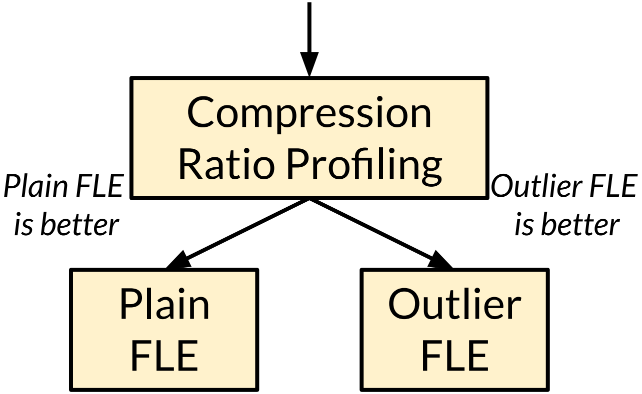
Summary

- cuSZp2 is open source at https://github.com/szcompressor/cuSZp
- **~300 GB/s** and **~500 GB/s** cmp/dec throughput for **single-precision** datasets.
- **~500 GB/s** and **~700 GB/s** cmp/dec throughput for **double-precision** datasets.
- Two lossless encoding modes supported, **high compression ratio** for different data.
- Efficient implementation on both high-end and low-end NVIDIA GPUs.
- Easy-to-use C/C++ and Python APIs are provided.

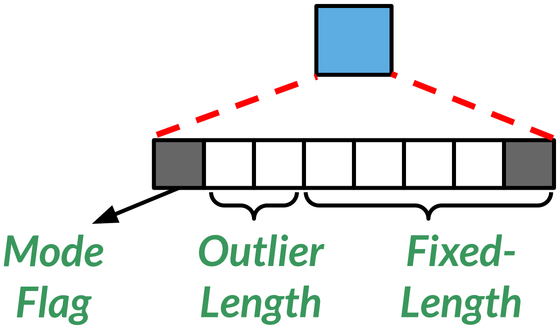
Yafan Huang
University of Iowa
yafan-huang@uiowa.edu
https://hyfshishen.github.io

Backup: Ratio Profiling in Outlier-FLE

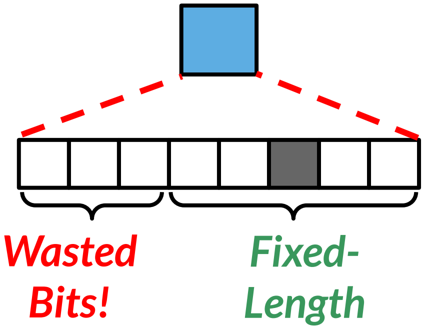
- In Outlier-FLE: the outlier processing is only adopted **when it has benefit over plain-FLE**. This is achieved by a **ratio profiling** phase.



cuSZp-O lossless encoding mode explanation
<u>Only storing outlier when it has benefits</u>



Offset in Outlier-FLE mode.



Offset in Plain-FLE mode.

In another word, in an HPC field, if ratio profiling always telling "Plain-FLE is better", Outlier-FLE will be downgraded into Plain-FLE.