Performance of lossless compression algorithms for CMS RAW data using ROOT TTree and RNTuple

Simone Rossi Tisbeni[1], Silvio Donato[2]

on behalf of the CMS collaboration

[1]CERN, INFN and University of Bologna - [2] INFN and University of Pisa

18 Nov 2025 - ROOT Users Workshop 2025

NexTGen
Next Generation Triggers

CMS HLT in Phase-2

Trigger system (HW & SW) **rejects ~99.9%** of events

By 2029 for **HL-LHC** data will likely increase by a **factor 10**

- Critical need for enhanced computing resources and techniques

The **Phase-2** event size of CMS is expected to be around **6 MB/event**.

- No way to store all events accepted from the hardware trigger (750 kHz) in raw data format (i.e. about **10 EB/year!**)

Next Generation Trigger project:

- 5-year project (2024–2028), involving **CMS**, ATLAS, CERN Theory, IT

Motivation: New Physics may be buried in **background** → current triggers discard signal

Goal: Explore innovative computing technologies to **expand trigger acceptance**

NextGen Task 3.3

The goal of **Task 3.3** is to reduce the size of events saved by CMS

- **Lossy compression**, replacing raw data with **high-level physics objects**
  (eg. muons, electrons, jets, tracks)
  - Evolution of our current **scouting** data format
  - Very strong compression (around x100), necessary to store all 750 kHz
  - Limited possibility to re-reconstruct objects with newer algorithm or calibration.
- **Lossy compression**, replacing raw data with **low-level physics objects**
  (eg. replacing tracker or calorimeter raw data with reco's hit positions and energies)
  - Extension of **RAW'** currently used in HIon collisions in CMS,
  - Limited compression (below x10),
  - Will allow to re-reconstruct high-level objects with newer algorithm or calibration
- **Lossless compression**:
  - **Will be reported in this presentation**

Lossless compression

Study performance of lossless compression algorithms
- Comparing RNTuple data format with TTree

Efficient lossless compression is necessary
- Reduce storage requirements
- Maintain accurate physics performance

This study benchmarks existing compression algorithms, evaluating:

- **Read time**
  Walltime of complete cmsRun execution

- **Compression Ratio**
  $\frac{\text{compressed size}}{\text{uncompressed size}}$ lower is better

- **Throughput**
  $\frac{\text{uncompressed size}}{\text{compression time}}$ in event/s or MB/s

- **LZMA** based on the LZ77 algorithm with huge dictionary size
  followed by a variant of arithmetic coding
  very strong compression
  very low throughput

- **ZLIB** widely used implementation of DEFLATE
  LZ77 compression
  followed by Huffman entropy coding
  moderate compression ratios
  relatively fast performance

- **ZSTD** modern compression algorithm by Meta
  fast compression and decompression
  dictionary-based with advanced entropy coding
  excellent balance between speed and compression

- **LZ4** fast compression algorithm optimized for speed
  lightweight dictionary approach
  weaker compression ratios
  suitable for high-throughput workflows.

Benchmarks with Run 3 RAW data

For Run 3 data

- *pp*-collision data file: ~ 140 s of data taken in Run 382229 of Fill 9806 (June 2024)
- Fully hadronic triggers dataset with 13.6 TeV, <PU> ~64.5

2 ROOT files with 5000 events: **uncompressed** in **TTree** ~7.5 GB, and in **RNTuple** ~7.6 GB

CMSSW `PoolOutputModule` and `RNTupleOutputModule`

- Modified to set `iWriteOptions.SetCompression(0)` for uncompressed files

`CMSSW_15_1_RNTUPLE_X` integration build with **ROOT Version: 6.35.99**

Only the **FEDRawDataCollection** object has been kept

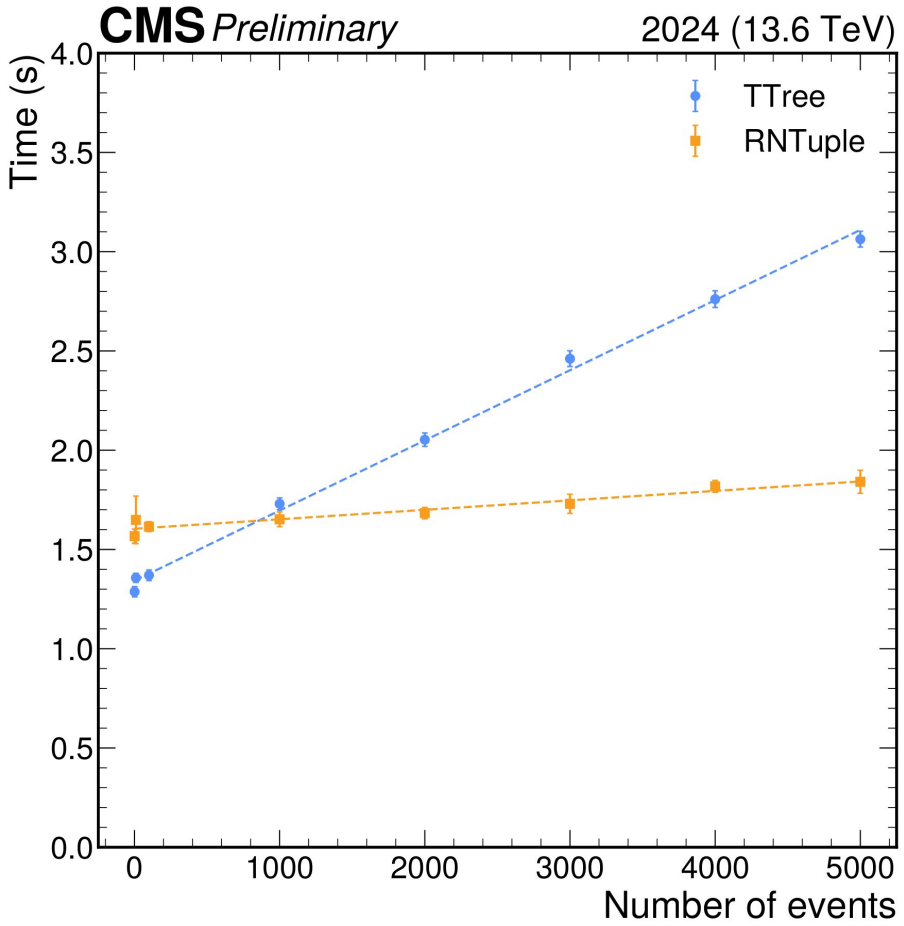
Read timing for TTree and RNTuple

Time required to read **uncompressed** root files
in **TTree** format and **RNTuple** format

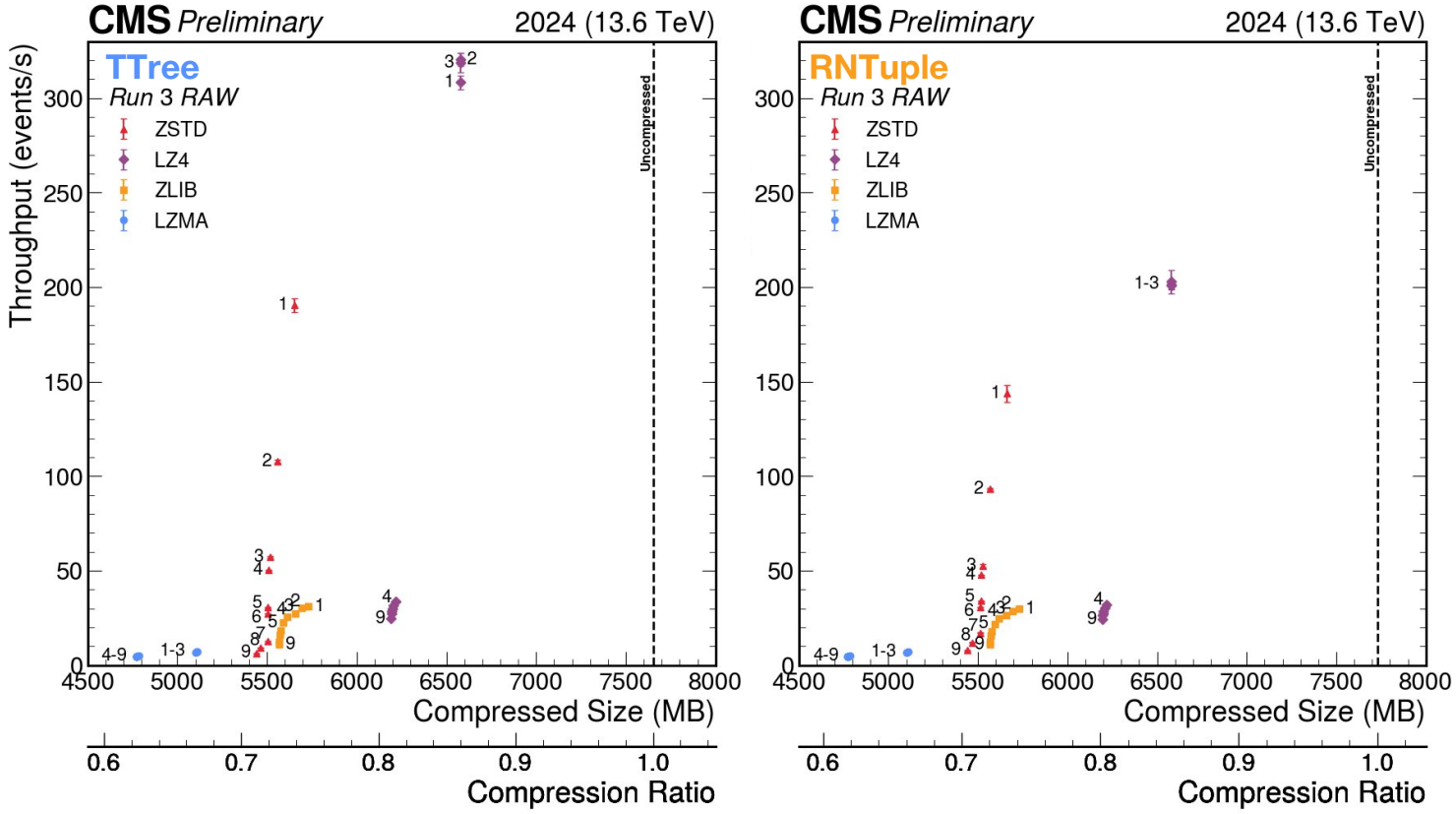
- Using the `PoolSource` and `RNTupleSource` modules together with a `GenericConsumer` Analyzer to ensure that all collections are read
- 1 thread on AMD EPYC 9534 64-cores 2.45GHz CPU, 180 GB RAM

Average timing of the entire **cmsRun** process of all events across 10 run (+3 for caching)

- **Linear** increase in reading time with the number of events
- RNTuple showing a much smaller increase
- RNTuple offers **significantly faster reading** performance for large numbers of events



Compression of Run 3 RAW data



Lossless compression comparison for Run 3 RAW

Throughput of the compression in **events/s** (5000 ev) as a function of:

- **Compressed Size** in MB
  (dashed line: uncompressed)
- **Compression Ratio**

Compressed and uncompressed: file sizes on disk

TTree uncompressed ~7.5GB

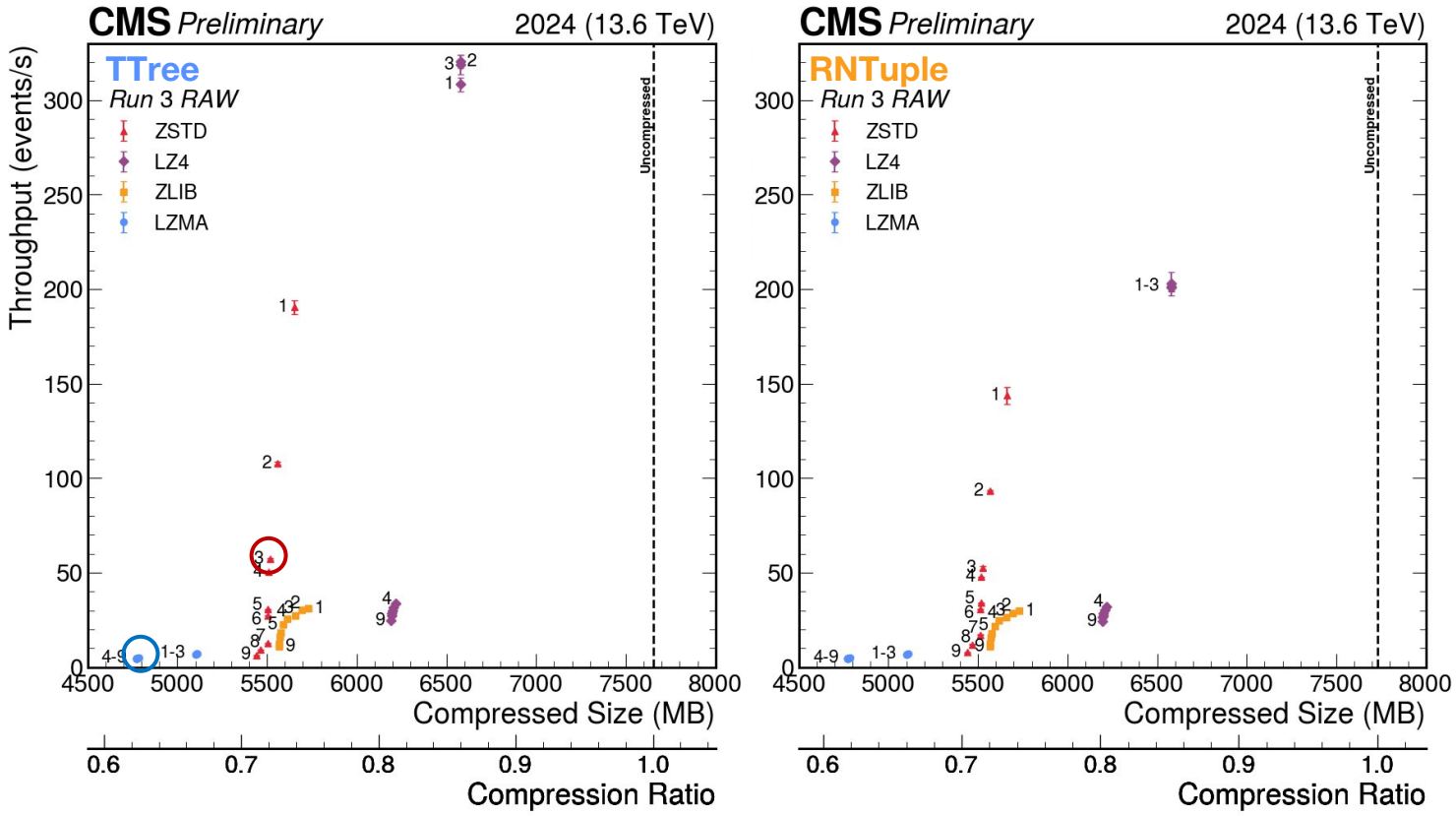
RNTuple uncompressed ~7.6GB

The labels are the compression level of each algorithm

- 9 is the strongest, 0 is the weakest

Data averaged across 5 run (+ 3 for caching)

Comparison of TTree and RNTuple



Similar trends for each format in all algorithm:

◆ **LZ4 highest throughput**, lower compression

● **LZMA** highest compression, **lowest throughput**

▲ **ZSTD** and ■ **ZLIB balanced**; ZSTD always **better** than ZLIB

TTree and **RNTuple** data formats are comparable

The thorughput is similar in all levels except for the lowest level where **TTree** show higher throughput

ZSTD 3 used for *pp* collisions in the HLT

LZMA 4 used for *PbPb* collisions in the HLT and T0 storage of RAW data.

At HLT compression is done event-by-event (in contrast to offline and the results in this benchmark)

Benchmarks with Phase-2 simulations

For Phase-2 MC simulation

- $t\bar{t}$ *sample* with 14 TeV, <PU> ~ 200

2 ROOT files with 500 events: **uncompressed** in **TTree** ~14 GB, and in **RNTuple** ~11 GB

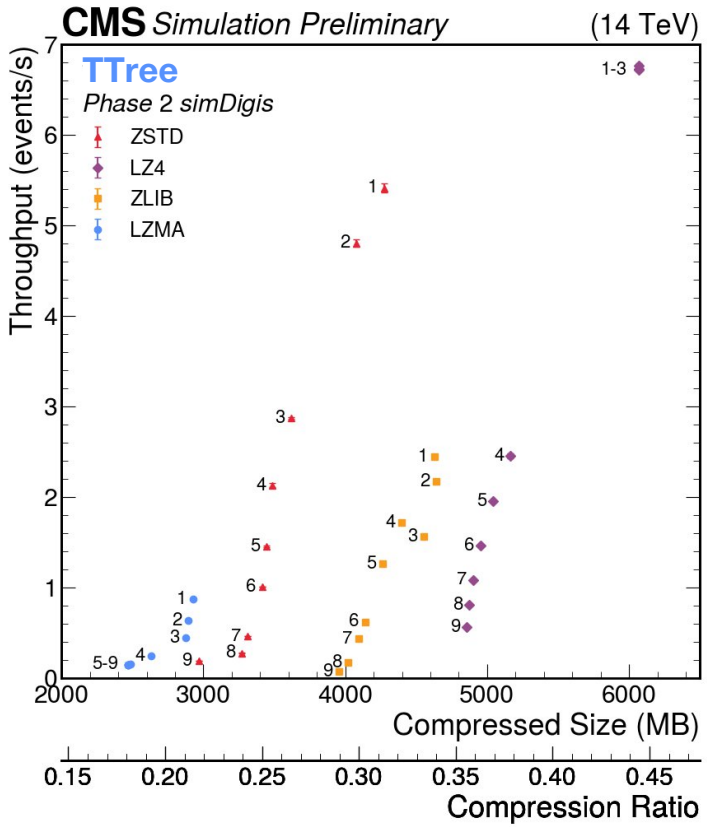
CMSSW `PoolOutputModule` and `RNTupleOutputModule`

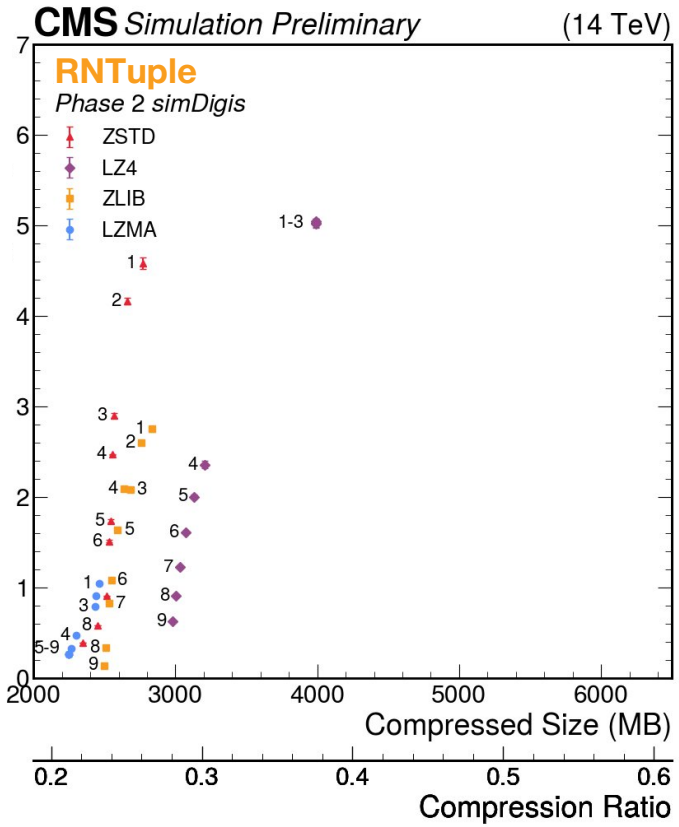
- Modified to set `iWriteOptions.SetCompression(0)` for uncompressed files

`CMSSW_15_1_RNTUPLE_X` integration build with **ROOT Version: 6.35.99**

Only the **simDigi** objects have been kept: closest approximation to RAW we have for Phase-2

Compression of Phase-2 simulation





Lossless compression comparison for Phase-2 MC

Throughput of the compression in **events/s** (500 ev) as a function of:

- **Compressed Size** in MB
- **Compression Ratio**

Compressed and uncompressed: file sizes on disk

TTree uncompressed ~14GB

RNTuple uncompressed ~11GB

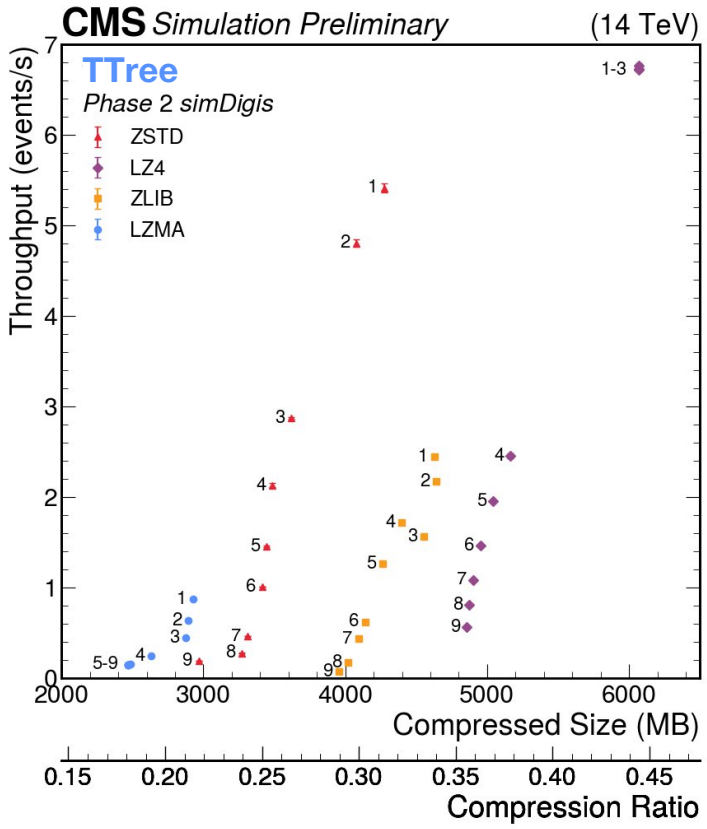
(**~20% smaller uncompressed size**)

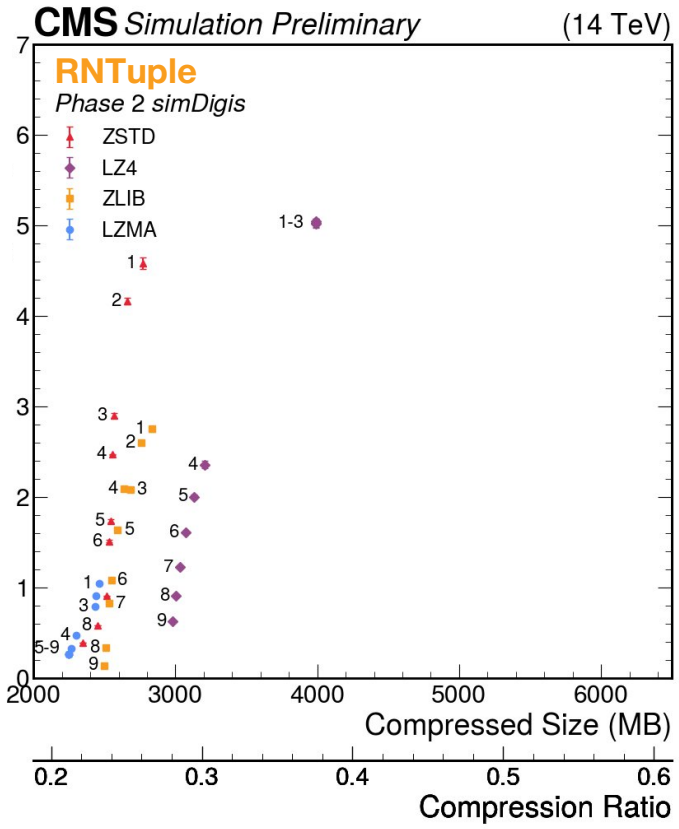
The labels are the compression level of each algorithm

- 9 is the strongest, 0 is the weakest

Data averaged across 5 run (+3 for caching)

Comparison of TTree and RNTuple





Similar trends for each format in all algorithm:

◆ **LZ4 highest throughput**, lower compression

● **LZMA** highest compression, **lowest throughput**

▲ **ZSTD** and ■ **ZLIB balanced**; ZSTD always **better** than ZLIB

TTree has higher throughput than **RNTuple** for lower levels, while is slightly lower for the higher levels

RNTuple has better compression in general

~30% smaller for **ZSTD 3**

~13% smaller for **LZMA 4**

Phase-2 *simDigi* files compress better than Run 3 *RAW* since they are **structured**
while RAW data are **bitstream**

Benchmarks of GPU-accelerated compression

Block-sorting algorithms

Encoded Input

Original order	Sorted order						
3	1	C	O	D	E	D	**E**
1	2	**D**	**E**	**C**	**O**	**D**	**E**
5	3	D	E	D	E	C	**O**
2	4	E	C	O	D	E	**D**
6	5	E	D	E	C	O	**D**
4	6	O	D	E	D	E	**C**

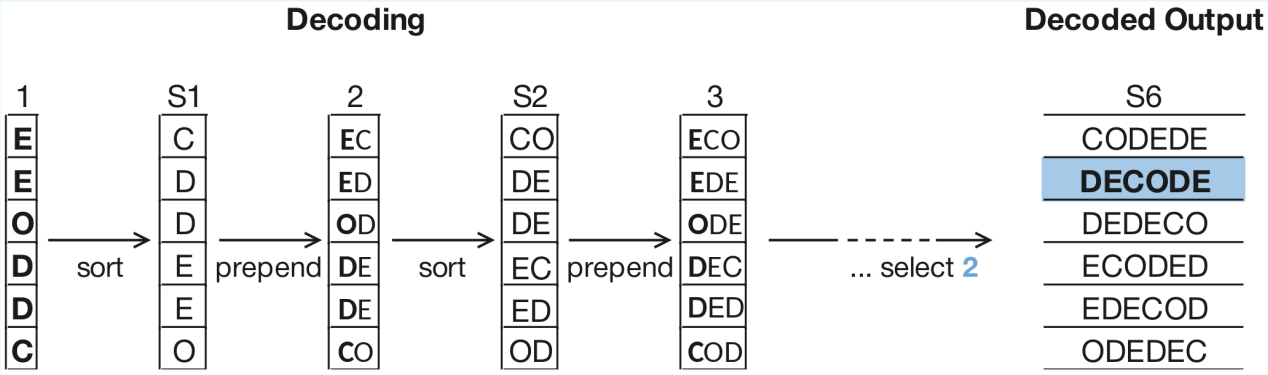
Block-sorting is **NOT** a compression algorithm

Reorders symbols to put **recurrent** symbols close together

An encoder can profit from these patterns to compress better

The output of the block sorting is:

- the last column of the matrix from all the rotations of the message sorted lexicographically
- the index of the original message in the matrix



The process is completely **invertible**

To decode, the original message can be obtained by iteratively left-inserting the encoded message and sorting the matrix.

libBSC

libBSC[1] is an **open source** compression library

- Uses different **block-sorting** algorithms and **entropy encoding** to compress the data
- Supports **multi-threading** and can use **GPU** to accelerate block-sorting

Has two options for configuring

Block-sorting (-m) (-G to enable GPU acceleration):

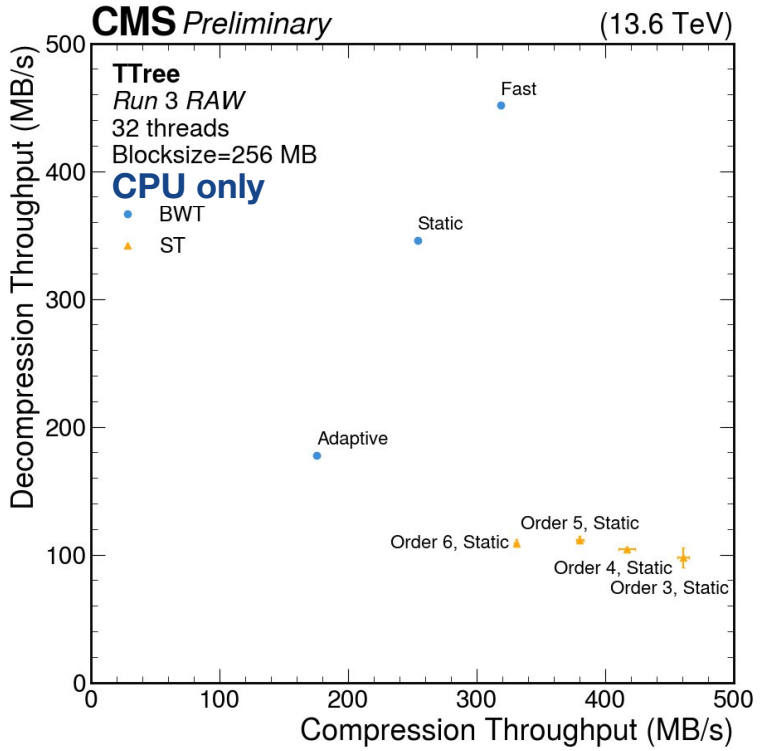
- -m0 Burrows Wheeler Transform (**BWT**): block sorting of entire data block
- -m3..8 Sort Transform (**ST**): block sorting of order N, sorts based on the first N bytes of the block

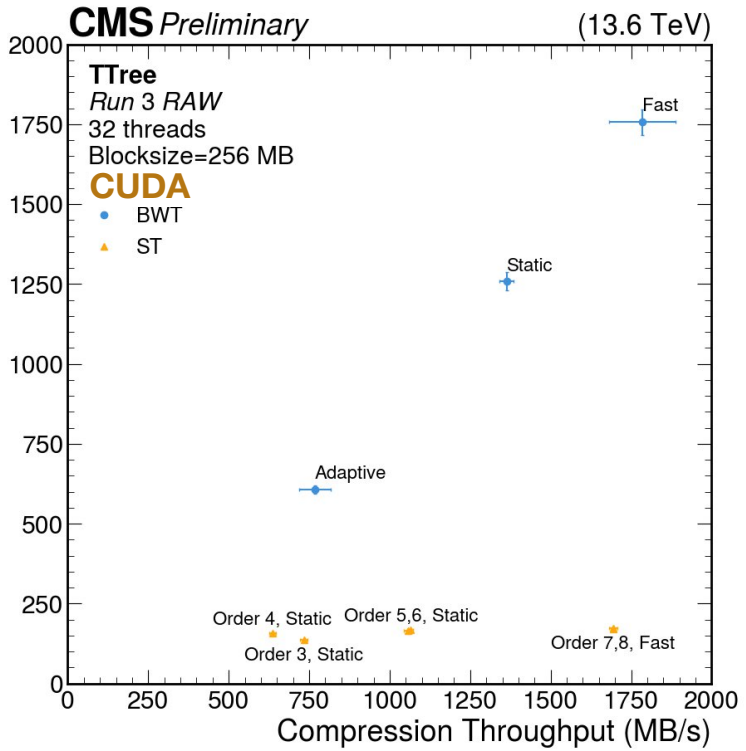
Encoding (-e):

- -e0 **Fast** Quantized Local Frequency Coding (QLFC): optimized for speed, uses a LUT with a simple probability model
- -e1 **Static** QLFC: evaluate once the frequency of the symbols in the message, balanced
- -e2 **Adaptive** QLFC: dynamically evaluate the frequency while encoding to adjust the weights, highest compression

[1] github.com/IlyaGrebnov/libbsc - program and library for lossless, block-sorting data compression

Compression and Decompression throughput





Compression vs Decompression throughput (MB/s) for different configurations (outside of ROOT)

- **BWT** algorithm for the three encoding
- ▲ **ST** algorithm for different orders and encodings

Timed[1] averaging 10 runs on

32 threads, blocksize 256 MB, on AMD EPYC 9654 96-core 3.7GHz CPU, 180 GB RAM, NVIDIA H100-NVL

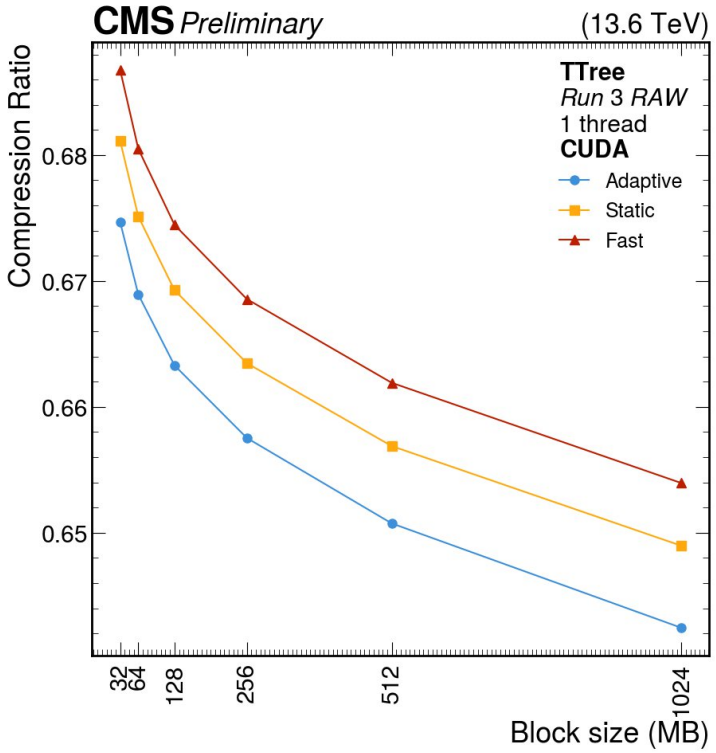
↓

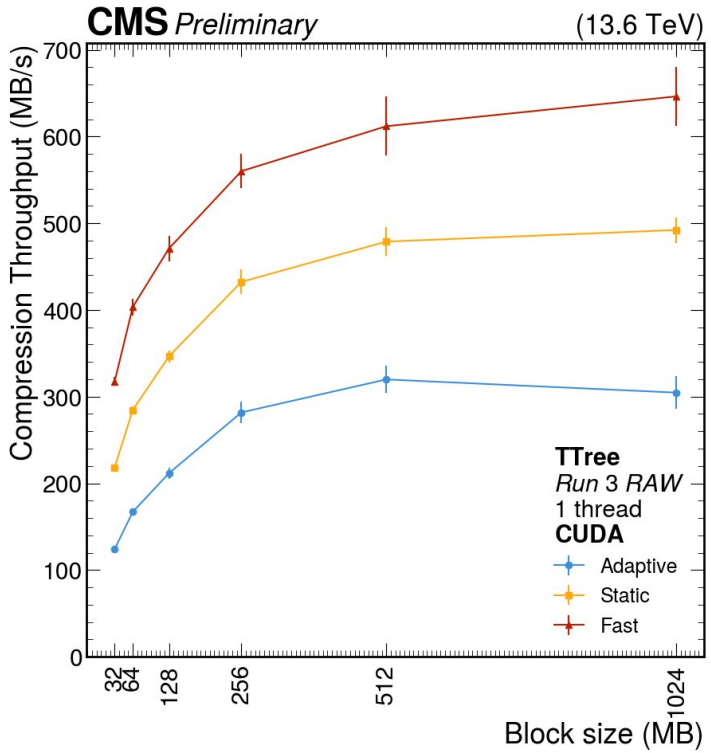
- ST faster in **CPU only**, **slower** in decompression
- BWT faster with **CUDA**, good in **both direction**

Since for RAW we want
compress once, decompress multiple times
BWT CUDA is used from the rest of the benchmarks

[1] github.com/inikep/lzbench - library for in-memory benchmarking of open source compression algorithms

Performance by Blocksize





The three lines show the behavior for the different encoding configurations of BWT

(Fast, Static, Adaptive)

- **Larger** blocksize → **stronger** compression
- Each block is compressed **independently**
- Compression throughput increases with the blocksize as the **sorting is parallelized** on GPU

GPU Comparison

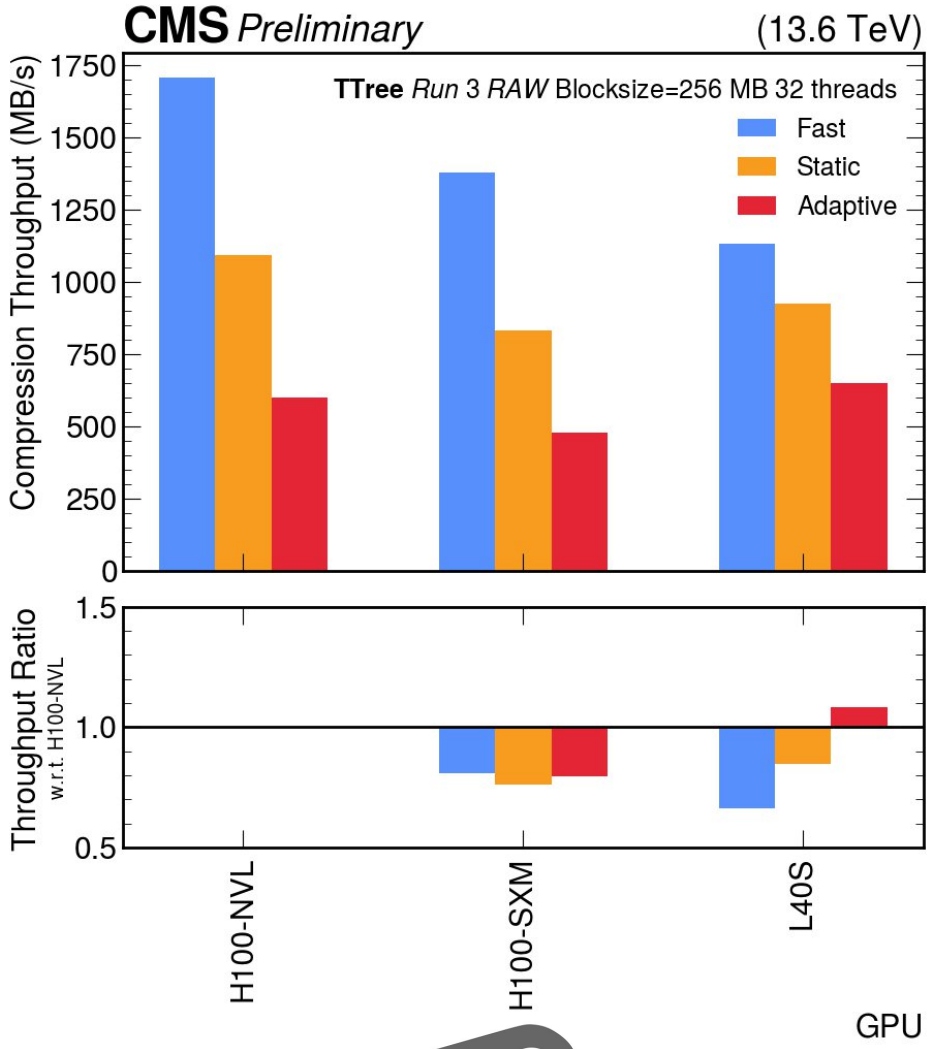
BWT running on different machines with 32 threads 256 MB blocksize:

- AMD EPYC 9654 96-Core Processor with 3.7GHz CPU
  180 GB RAM, 1x Nvidia **H100-NVL** GPU
- INTEL XEON GOLD 6530 64-Core Processor with 4 GHz CPU
  240 GB RAM, 1x Nvidia **H100-SXM** GPU
- AMD EPYC 9534 64-Core Processor with 3.7GHz CPU
  360 GB RAM, 1x Nvidia **L40S** GPU

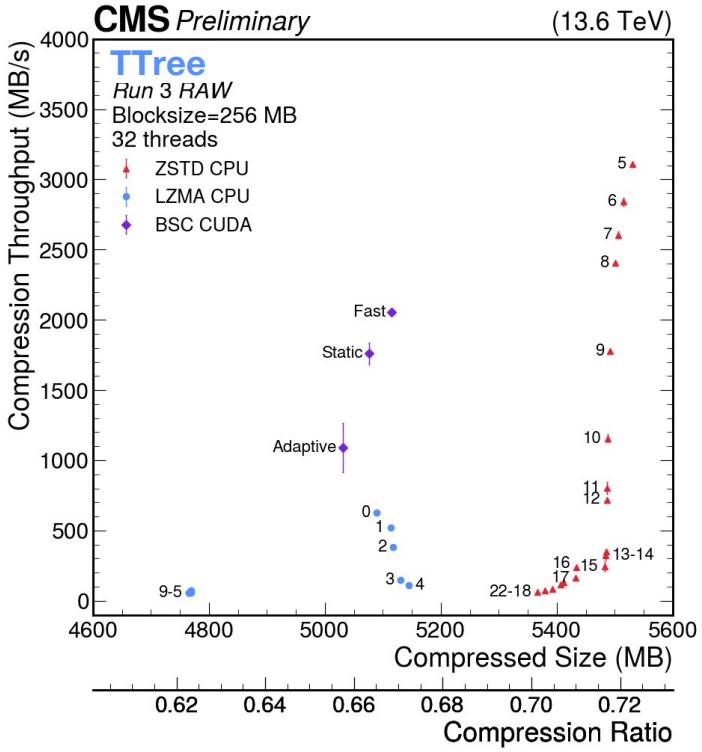
Compression throughput ratio w.r.t. H100-NVL GPU.

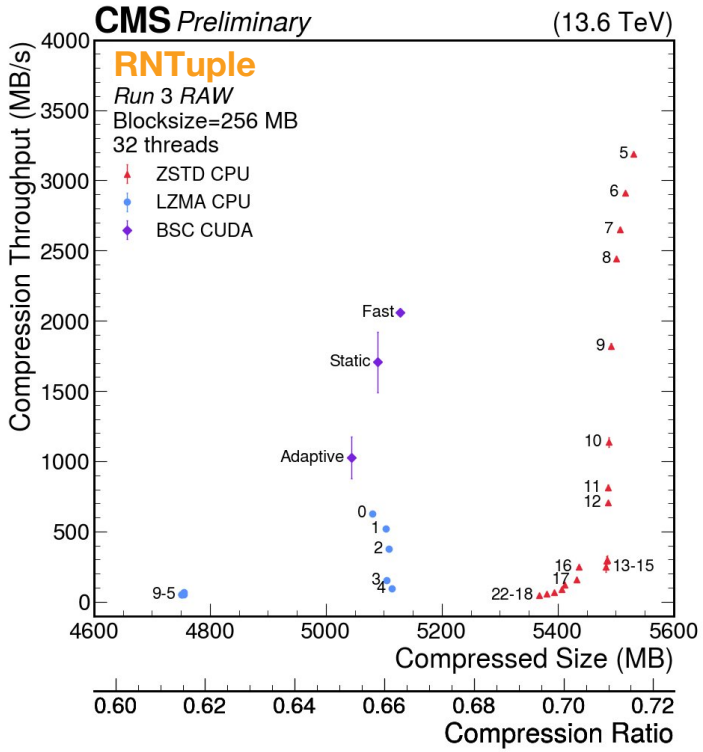
Run 3 RAW in TTree (5000 ev) uncompressed size ~7.5GB

Higher performance with the H100-NVL GPU, with the SXM slower due to the **lower bandwith** (3.9 TB/s vs. 3.35 TB/s)



Comparison of TTree and RNTuple





Lossless compression comparison for Run 3 RAW

Throughput of the compression in **events/s** (5000 ev) as a function of:

- **Compressed Size** in MB
- **Compression Ratio**

Compressed and uncompressed: file sizes on disk

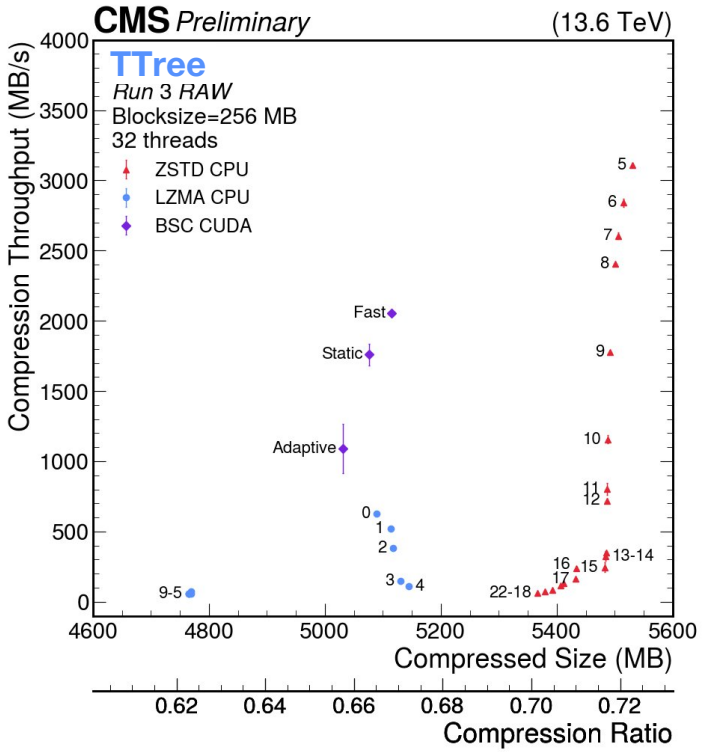
TTree uncompressed ~7.5GB

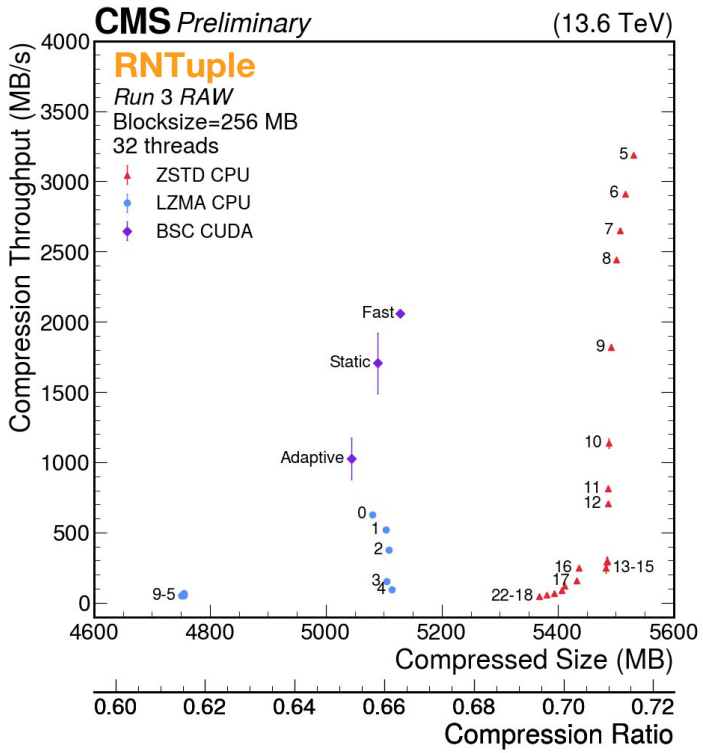
RNTuple uncompressed ~7.6GB

The labels are the configuration of each algorithm

Data averaged across 10 run

Comparison of TTree and RNTuple





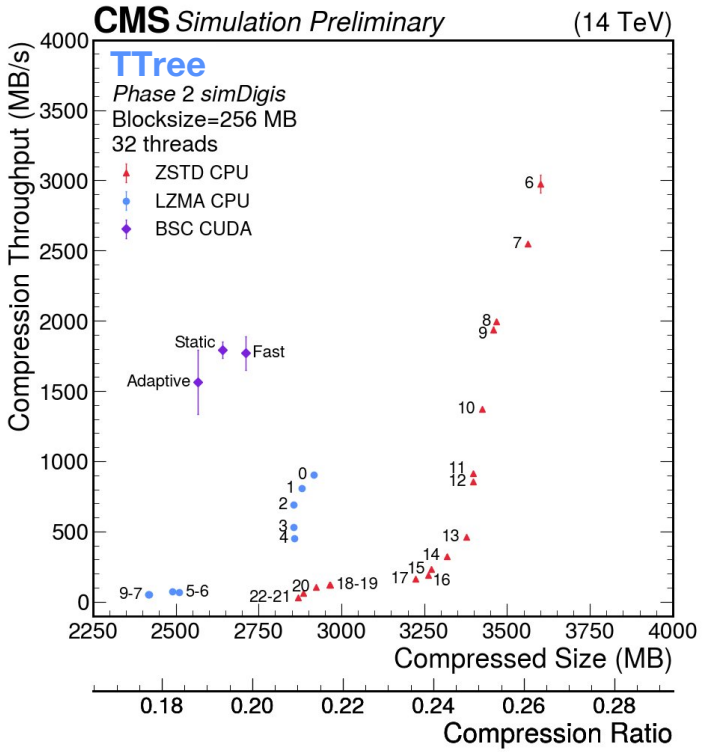
◆ **BSC** algorithm achieves a **better compression** ratio than ▲ **ZSTD** across all configurations

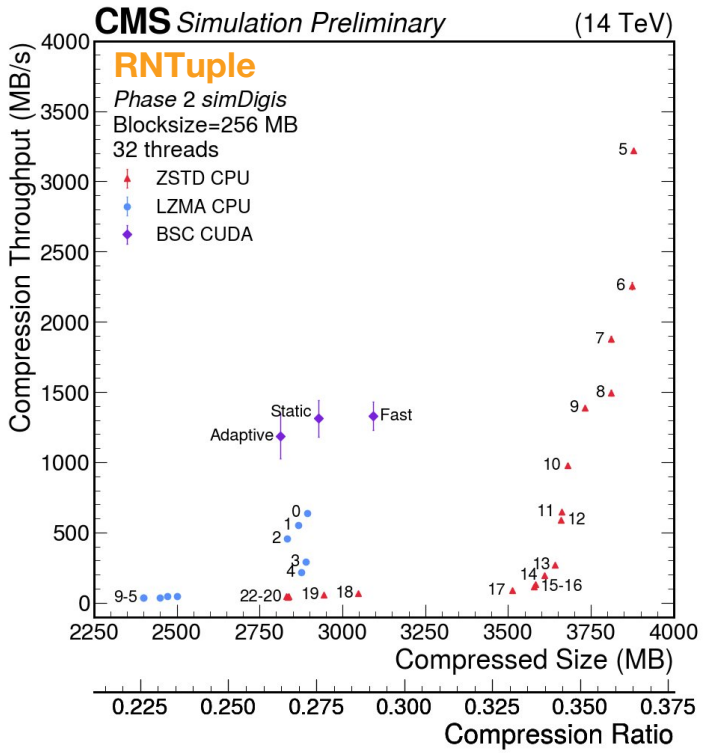
Higher throughput at the strongest compression levels

The strongest ◆ **BSC** compression setting achieves **higher throughput** than all ● **LZMA** settings

better compression ratio than some ● **LZMA** settings

Comparison in Phase-2 simulations





Lossless compression comparison for Phase-2 MC

Throughput of the compression in **events/s** (500 ev) as a function of:

- **Compressed Size** in MB
- **Compression Ratio**

Compressed and uncompressed: file sizes on disk

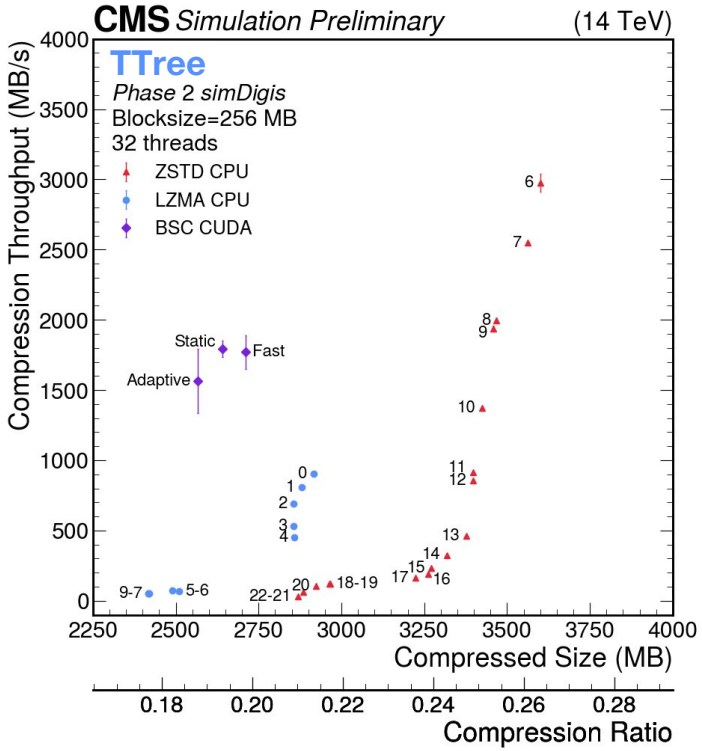
TTree uncompressed ~14GB

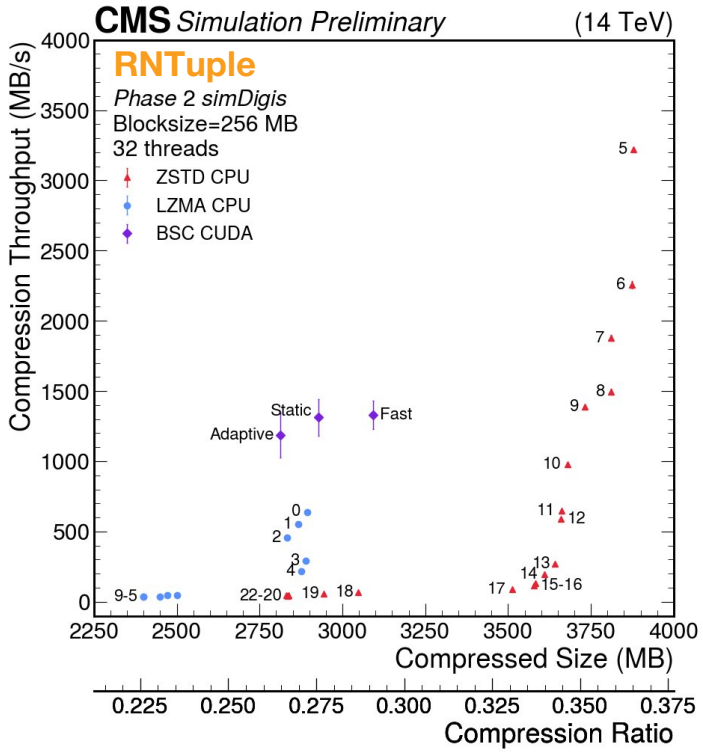
RNTuple uncompressed ~11GB

The labels are the configuration of each algorithm

Data averaged across 10 run

Comparison in Phase-2 simulations





◆ **BSC** algorithm shows a **better compression** ratio than ▲ **ZSTD** across all configurations

Better compression than the lowest ● **LZMA** levels

Higher throughput than the strongest ▲ **ZSTD** compression settings
and **all** ● **LZMA** compression levels.

References & acknowledgments

Next Generation Triggers, *“NGT Evaluation Report - 2024”*, doi: 10.17181/nkwjc-7pa06

CMS Collaboration, *"Performance Comparison of Lossless Compression Algorithms on CMS Data using ROOT TTree and RNTuple"*, https://cds.cern.ch/record/2941436

Michael Burrows and D. J. Wheeler, *"A Block-sorting Lossless Data Compression Algorithm"*, https://www.cs.jhu.edu/~langmea/resources/burrows_wheeler.pdf

M. Schindler, *"A fast block-sorting algorithm for lossless data compression"*, doi: 10.1109/DCC.1997.582137

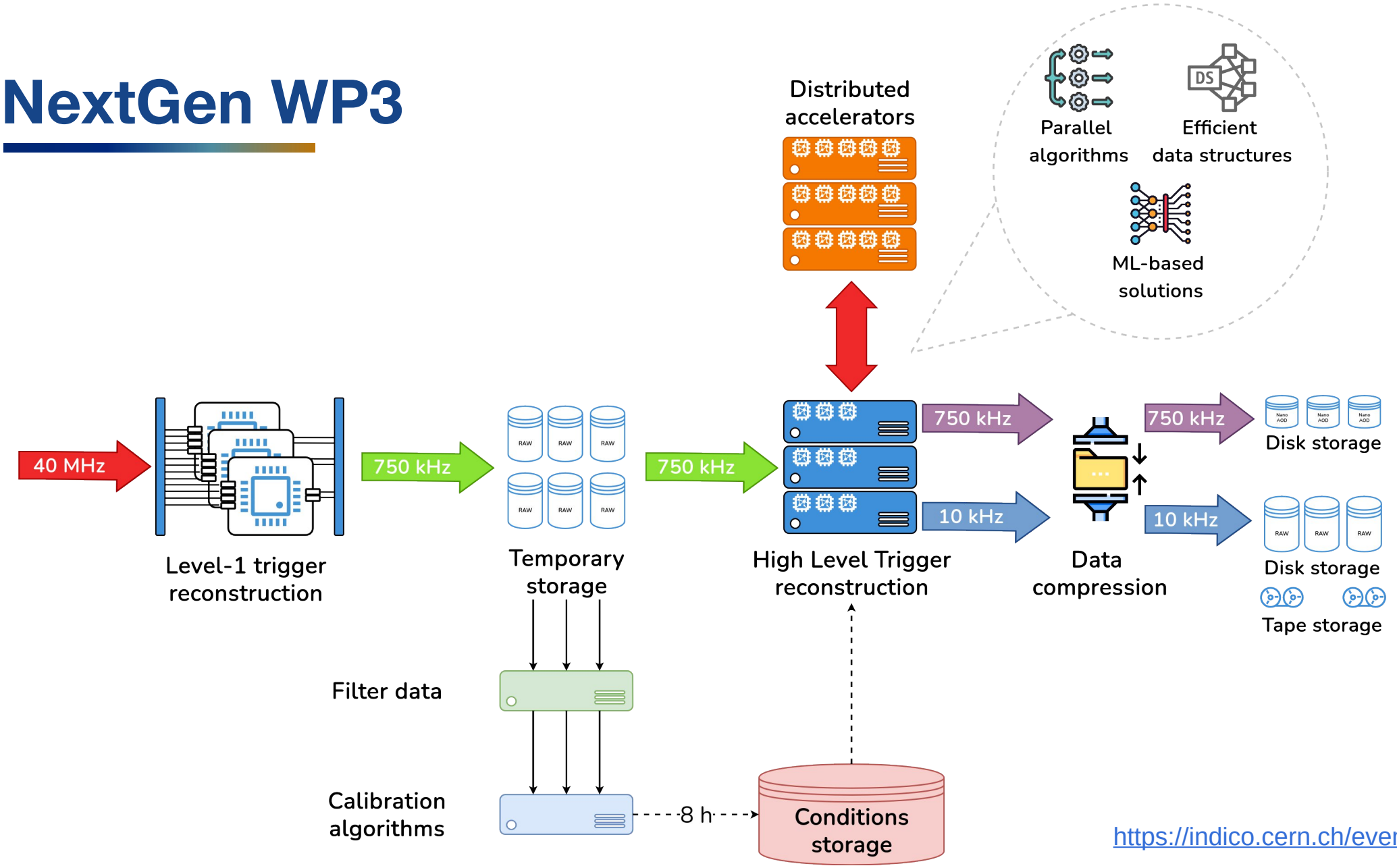
CMS Collaboration, *“Performance of GPU-accelerated lossless compression for CMS Data using ROOT TTree and RNTuple”*, https://cds.cern.ch/record/2948320

This work has been partially funded by the Eric & Wendy Schmidt Fund for Strategic Innovation through the CERN Next Generation Triggers project under grant agreement number SIF-2023-004.

Thank you


NexTGen
Next Generation Triggers


NextGen WP3



https://indico.cern.ch/event/1550919/

ROOT compression vs libraries

	Run 3 RAW RNTuple	Phase-2 simDigis RNTuple
LZMA 4 (ROOT)	4786 MB	2303 MB
LZMA 5 (lzbench)	4754 MB	2501 MB
ZSTD 3 (ROOT)	5525 MB	2566 MB
ZSTD 6 (lzbench)	5516 MB	3873 MB
BSC Adaptive (lzbench)	5031 MB	2566 MB

Test with ROOT 6.36

CMS approval process is slower than ROOT and CMSSW Core developers :)

Tests with **CMSSW_16_0_RNTUPLE_X**

ROOT version: 6.36.05

↓

~ Same results in compression and uncompressed file size