ndzip-gpu: Efficient Lossless Compression of Scientific Floating-Point Data on GPUs

Fabian Knorr
fabian@dps.uibk.ac.at
University of Innsbruck
Austria

Peter Thoman
petert@dps.uibk.ac.at
University of Innsbruck
Austria

Thomas Fahringer
tf@dps.uibk.ac.at
University of Innsbruck
Austria

ABSTRACT

Lossless data compression is a promising software approach for reducing the bandwidth requirements of scientific applications on accelerator clusters without introducing approximation errors. Suitable compressors must be able to effectively compact floating-point data while saturating the system interconnect to avoid introducing unnecessary latencies.

We present ndzip-gpu, a novel, highly-efficient GPU parallelization scheme for the block compressor ndzip, which has recently set a new milestone in CPU floating-point compression speeds.

Through the combination of intra-block parallelism and efficient memory access patterns, ndzip-gpu achieves high resource utilization in decorrelating multi-dimensional data via the Integer Lorenzo Transform. We further introduce a novel, efficient warp-cooperative primitive for vertical bit packing, providing a high-throughput data reduction and expansion step.

Using a representative set of scientific data, we compare the performance of ndzip-gpu against five other, existing GPU compressors. While observing that effectiveness of any compressor strongly depends on characteristics of the dataset, we demonstrate that ndzip-gpu offers the best average compression ratio for the examined data. On Nvidia *Turing*, *Volta* and *Ampere* hardware, it achieves the highest single-precision throughput by a significant margin while maintaining a favorable trade-off between data reduction and throughput in the double-precision case.

CCS CONCEPTS

• **Computing methodologies** → **Parallel algorithms**; • **Theory of computation** → **Data compression**.

KEYWORDS

accelerator, gpgpu, data compression, floating-point

1 INTRODUCTION

On the path toward Exascale, energy efficiency is becoming the dominant driver of innovation in High Performance Computing. The rapid increase of intra-node parallelism, including the advent of GPUs as general-purpose accelerators, has lowered energy costs for compute-intensive applications considerably [8, 23]. Meanwhile, the relative time and energy cost of inter-node communication is projected to increase with overall system performance [20]. This form of data movement is already the most energy-intensive in a compute cluster, motivating the optimization of link usage in software [15], for example via communication avoidance.

Data compression with the goal of avoiding I/O bottlenecks in large-scale distributed applications has been studied extensively, primarily for checkpointing. Since the bandwidth of I/O links is comparatively low, lossy compression has allowed significant time savings in this context [5]. As checkpoints are not frequently reloaded, the loss in fidelity is acceptable for many applications.

This assumption does not hold for inter-node communication as part of a distributed computation. In simulations, data is usually exchanged at least once per time step, leading to the accumulation of compression artifacts above an acceptable level. For distributed linear algebra, kernels might not provide sufficient numerical stability under the error introduced by a lossy encoder.

Lossless data compression avoids these precision problems at the cost of a lower potential compression ratio due to incompressible noise inherent in low-order floating-point mantissa bits. Although general-purpose compressors have comparatively low throughput and parallelism due to their large state, there exist fast specialized floating-point compressors for both CPU [11] and GPU [16, 25].

Compression of communication data is only viable if compressor output and decompressor input speeds exceed the interconnect bandwidth. To avoid offsetting the gains from compressed communication, additional latencies must be as small as possible. For example, the cost of a device-to-host transfer prior to compression of results acquired on a GPU would be prohibitive.

To demonstrate the viability of data compression for the acceleration of inter-node communication, we explore how GPU compression can deliver the necessary performance. The contribution of this paper is as follows:

- ndzip-gpu, a highly-efficient GPU parallelization scheme for ndzip, a state-of-the-art lossless floating-point compressor
- A fast, warp-cooperative bit packing primitive
- An evaluation of ndzip-gpu on state-of-the-art hardware and a large set of representative test data
- A performance comparison of ndzip-gpu against five existing, publicly available lossless GPU compressors
- An open-source implementation of all of the above

The remainder of this article is structured as follows. Section 2 gives an introduction to floating-point compression and GPU implementation considerations for general-purpose as well as specialized compressors. Section 3 provides an overview of the ndzip compressor design. Section 4 discusses the architecture of the ndzip-gpu parallelization scheme in detail and derives an efficient method for vertical bit packing on GPUs. Section 5 compares ndzip-gpu against state-of-the-art GPU compressors on representative test data with current HPC hardware. Section 6 summarizes our research.

1.1 Reference System

The reference hardware for throughput evaluation in this work is the Marconi-100 supercomputer in Bologna, Italy, which holds rank 11 of the TOP500 list as of November 2020[1].

It is a cluster of 988 dual-CPU IBM POWER9 AC922 nodes with 256 GB RAM and four Nvidia Tesla V100 *Volta* GPUs each[2]. Inter-node communication is realized using Infiniband EDR with dual-channel Mellanox ConnectX5 network interface cards.

The theoretical, unidirectional peak transfer rate of Infiniband EDR is 100 Gb/s (12.5 GB/s) per channel, so each node can send and receive 25 Gigabytes per second. A suitable compression algorithm must therefore be able to deliver at least 25 GB/s in compressed throughput to avoid underutilizing the network link of this system.

2 BACKGROUND

We begin with an overview of lossless scientific-data compression and the problem of selecting and implementing efficient compressors on graphics hardware.

2.1 GPU Hardware and Programming Model

Although there are strong similarities between GPU vendors and hardware generations, high performance applications still require careful tuning for target architectures to achieve maximum performance. Our work primarily targets the Nvidia *Volta* and *Ampere* microarchitectures, released in 2018 and 2020.

Even though we implement ndzip-gpu using the SYCL programming model which follows the OpenCL naming scheme for various GPU concepts, we stick to CUDA terminology in the hope that it will be more familiar to the reader.

The defining parameter of a GPU program (kernel) is its parallel iteration space (grid), specified as the number of blocks in the grid together with the number of threads per block. Upon launch, the blocks of a kernel are distributed among the streaming multiprocessors (SMs) of the device, where multiple blocks can occupy an SM. For execution, blocks are subdivided into warps of 32 threads each. In each cycle, the schedulers of an SM each pick an eligible warp and schedule an instruction on one of the SM's execution units. Branch divergence between threads of a warp potentially reduces performance by requiring the scheduler to process the instructions of all branches sequentially. A high number of active threads per SM increases scheduler occupancy which facilitates instruction latency hiding but can lead to higher register and cache pressure.

As accessing global device memory has very high latency, each SM provides, in addition to its caches, a small area of fast shared memory that is common to all threads of a block. Shared memory has similar access latencies to registers but requires careful layout to avoid bank conflicts. An SM has 32 memory banks, each 32 bits wide, to which shared memory addresses are mapped in a modulo fashion. Memory access is conflict-free if either all simultaneous accesses within a warp map to different memory banks or all accesses that share a bank refer to the same memory address.

Kernels can issue warp-level or block-level barrier instructions which synchronize thread execution and act as a memory fence. This is required whenever threads within a thread block exchange data through shared or global memory. Barriers are inexpensive in themselves, but cause stalls when they are not reached by all participating threads simultaneously.

In addition to data exchange via shared memory, threads can make use of warp-cooperative operations such as shuffles and reductions, which provide versatile primitives for fast data accumulation and exchange without memory round-trips.

2.2 Challenges in Parallel Lossless Data Compression

Traditional lossless compressors tend to favor serial implementations because of mutable encoder / decoder state and the necessity of a variable-length output stream encoding.

Mutable Encoder / Decoder State. In the general case, lossless reduction of data volume is achieved by constructing a probability model for the input data and assigning short representations to probable inputs and longer representations to improbable ones. The decoder must have access to the encoder's probability model to reverse this mapping. Since the model is usually neither known ahead-of-time nor static for the entire length of the stream, exchanging it explicitly becomes infeasible for single-pass compressors. Instead, encoder and decoder will both construct and continuously update identical models from previously observed uncompressed symbols.

A highly-parallel compressor must be able to break this dependency chain in order to avoid a runtime behavior dominated by synchronizing on shared state. Compressors with large state such as dictionary coders will not tolerate fine-grained subdivision of their input space without a significant drop in effectiveness. Small-state local decorrelation schemes are more robust in that regard.

Variable Length Encoding. Compression of a chunked data stream is an input-parallel problem since the compressed chunk length is not known ahead of time. Threads of a parallel compressor must synchronize in order to determine the positions of individual chunks in the output stream. There are two fundamental approaches to avoiding serialization around this dependency:

(a) Compressing k chunks in k parallel threads in fast scratch memory, deriving output positions after a barrier, and finally having each thread commit the write to the output stream.
(b) Compressing the entire stream to a sufficiently sized intermediate buffer, computing the output positions for all chunks using a prefix sum, and finalizing the output stream with a separate compaction step.

[1] https://www.top500.org/lists/top500/list/2020/11
[2] https://wiki.u-gov.it/confluence/download/attachments/358212674/redp5494.pdf

Option (a) minimizes the required global memory bandwidth at the cost of potentially expensive barrier operations, whereas (b) keeps the compressor threads fully parallel, which is useful for avoiding stalls when their run-time is not constant.

2.3 Specialized Floating-Point Compressors

Floating-point binary representations have a larger word size than assumed by byte-oriented general-purpose compressors. Also, it is unusual for floating-point data from real-world applications to exhibit bit-identical repeating values that are easily deduplicated. Therefore, the traditional dictionary coder approach is not particularly efficient on this kind of data.

However, dense grid data originating from physical simulations or sensor arrays tends to exhibit low-frequency components, making local prediction from neighboring values feasible. The higher the dimensionality of a grid, the more local correlations are expected due to the larger number of neighboring cells for each value.

Construction of a specialized floating-point compressor therefore usually includes the following three components:

(1) A **predictor** estimates data from previously-encoded points via dictionaries, hash tables or from neighboring values.
(2) A **difference operator** calculates the residual between a value and its prediction in a reversible way, for example with an XOR operation or an integer difference.
(3) A **residual coder** expresses residuals using a variable-length code favoring small-magnitude values. Algorithms usually aim to eliminate leading-zero bits through a representation such as run-length encoding or arithmetic coding.

Several notable CPU-based lossless floating-point compressors exist in addition to the ndzip algorithm described in section 3.

fpzip [14] uses the Lorenzo predictor [9] to exploit smoothness in the direct neighborhood of points within an n-dimensional grid, compacting residuals using a range coder. The scheme exhibits high compression efficiency especially for single-precision values, but is limited to single-threaded operation.

FPC [3] uses a pair of hash-table based value predictors to compress an unstructured double-precision data stream. The thread-parallel pFPC variant [4] allows further prioritization of compression throughput by processing input data in chunks.

ZFP [13] is a fixed-rate lossy compressor that uses a frequency transform to decorrelate floating-point values in a multidimensional grid. The implementation additionally features a variable-rate lossless mode. Unlike the lossy variant, lossless compression is currently not realized on GPUs.

2.4 Data Compression on GPUs

Few publicly available, lossless data compressors exist for GPUs that are suitable for floating-point data.

General-Purpose Compressors. nvCOMP[3] is a lossless data compression framework for Nvidia GPUs. It includes, among others, a high-throughput implementation of the well-known LZ4 compressor and a configurable *Cascaded* compression pipeline that is well-suited for integer data.

[3]https://developer.nvidia.com/nvcomp

cudppCompress [18] is a general-purpose byte-oriented compressor for GPUs. It parallelizes three stages of the well-known bzip2 compressor, achieving measurable speedup compared with a CPU implementation on hardware from a similar era. The corresponding decompressor is not implemented.

In related work, GPUs have been successfully used as a coprocessor to accelerate Burrows-Wheeler transform [6]. There further exist parallel implementations of the Lempel-Ziv-Welch (LZW) [7] and Lempel–Ziv–Storer–Szymanski (LZSS) [17] compressors, and GPU entropy coding has seen notable progress in the form of fast Huffman [2, 22] and Asymmetric Numeral System (ANS) coders [24].

Specialized Floating-Point Compressors. MPC [25] is a GPU compression scheme for unstructured, multivariate streams of single- or double-precision floating point data. Two-step, one-dimensional value prediction is combined with vertical bit packing, a coding scheme that maps well to the target hardware.

GFC [16] is an exceptionally fast GPU compressor for unstructured double-precision data. Residuals from a one-dimensional predictor are compacted by run-length encoding leading zero bits. Unlike all other evaluated compressors, GFC produces a fragmented compressed output that is compacted on transfer back to the host.

The authors' reference implementations of both GFC and MPC are CUDA programs targeting the Nvidia *Kepler* microarchitecture.

3 THE NDZIP ALGORITHM

We briefly summarize the established ndzip algorithm [11] before presenting the ndzip-gpu parallelization scheme.

ndzip is a state-of-the-art block compressor targeting one- to three-dimensional grids of single- or double precision floating point data. It approximates the Lorenzo predictor [9] using the Integer Lorenzo Transform, a separable in-place operation for local decorrelation of multidimensional blocks. Residuals are encoded using the vertical bit-packing scheme previously found in MPC, eliminating zero-runs in bit-positions of neighboring residuals. By operating entirely within the integer domain, the algorithm guarantees reversibility of the compression operation as well as portability.

ndzip has been shown to deliver exceptional throughput on CPUs compared to established general-purpose compressors such as Deflate and specialized algorithms like fpzip [14] or FPC [3] with an implementation leveraging both thread- and SIMD parallelism. The ndzip-gpu compressor we present in this paper reproduces the compressed format of ndzip exactly.

3.1 Block Compression

Instead of operating on the entire, arbitrarily-sized input grid at once, ndzip subdivides it into fixed-size hypercubes. Each cube consists of 4096 elements, which corresponds to block sizes of 4096^1, 64^2 and 16^3, respectively. When the grid size is not a multiple of the block size in any dimension, the remaining border is transmitted in its uncompressed form.

3.2 Integer Lorenzo Transform

The floating-point Lorenzo predictor [9] estimates the value on one corner of a length-2 hypercube within an n-dimensional space with an implicit polynomial of degree $n - 1$. It has been shown to be highly effective on multidimensional data [14]. However, the

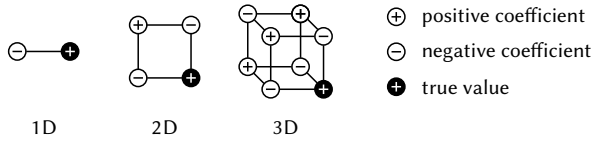

Figure 1: Decorrelation via Integer Lorenzo Transform

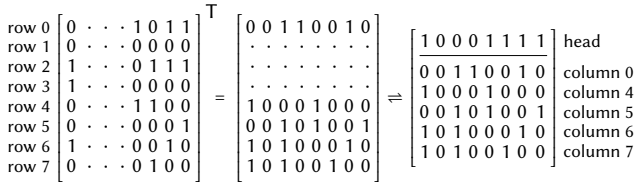

Figure 2: Residual coding through Vertical Bit Packing

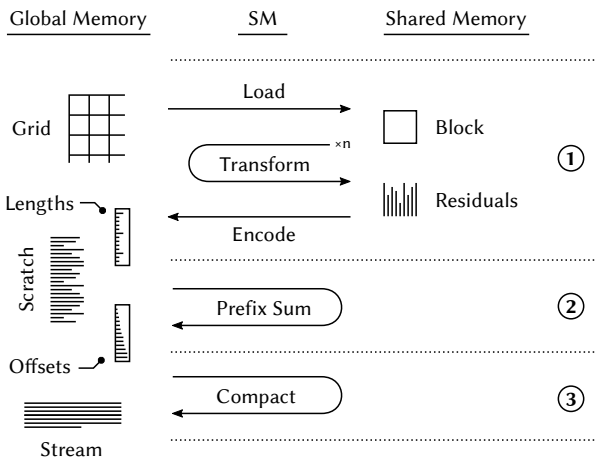

Figure 3: Three-stage compression pipeline

separation of prediction and residual computation steps requires the decompressor to reconstruct each prediction from already-decoded neighboring values, restricting parallelism.

To avoid this limitation, ndzip bijectively maps the bit pattern of uncompressed floating-point input onto an integer representation that roughly preserves its monotonicity properties. The resulting block is then decorrelated using the n-dimensional Integer Lorenzo Transform. Starting at the second element of each block, the transform replaces every value with the integer difference to its predecessor. This subtraction step is repeated along each dimension, or n times in total, exploiting separability of the multi-dimensional case. This construction is visualized in Figure 1.

Since integer addition and subtraction are reversible, the Inverse Integer Lorenzo Transform is a similar in-place operation and separable in the same fashion.

3.3 Vertical Bit Packing

Small two's complement integer residuals exhibit a large number of leading redundant sign bits, which are encoded space-efficiently using vertical bit packing. This is achieved by translating residuals to a sign-magnitude representation, grouping them into sequences of 32 single-precision or 64 double-precision values, and transposing the resulting 32×32 or 64×64 bit matrix as shown in Figure 2. All resulting zero-rows are eliminated. The head, a bitmap with each position indicating whether the corresponding input column was non-zero, communicates to the decoder how to expand the stream.

4 PARALLELIZATION SCHEME

The ndzip compressor was originally designed for efficient implementation on SIMD-capable multicore CPUs, exploiting both thread parallelism between blocks and vector operations to achieve high throughput in the transform and encoding stages. The CPU reference implementation utilizes the 256-bit wide x86_64 AVX2 extension to accelerate the transform and bit packing stages. Due to the rather rigid addressing modes of AVX2, SIMD parallelism is effectively limited to adjacent values in memory.

While the block subdivision scheme maps well onto independent thread blocks of the SYCL model, an efficient GPU implementation must extract significantly more parallelism from within blocks to keep all threads of an SM occupied. Investing additional compute resources into fine-grained, conflict-free value addressing to avoid starving execution units on the GPU is both viable and necessary, and forms a core innovation of our approach.

In this section, we detail how our novel parallelization scheme ndzip-gpu is able to efficiently distribute both transform and residual coding among up to 768 threads while keeping branch divergence and serialization to a minimum.

We target compression and decompression between global memory buffers on the device. ndzip-gpu prefers coalesced loads and stores where global-memory access is necessary, but relies on fast shared-memory and warp-cooperative operations where possible.

We chose the SYCL 2020 programming model[4] for implementation, which exposes hierarchical parallelism and warp-cooperative primitives in an expressive fashion. Although our code is currently limited to devices with a warp size of 32 as is usual for Nvidia GPUs, SYCL will ensure portability to other architectures in the future.

4.1 Compression Pipeline Overview

As discussed in Section 2.2, the output-offset problem of parallel compression can be solved via whole-device synchronization in each block or multiple kernel launches and a round-trip through an intermediate global scratch buffer. ndzip-gpu employs the second variant. We would expect global barriers to partially negate the benefit of short-circuit evaluation in the compute-heavy residual coder. Also, whole-device synchronization is not supported natively by the SYCL programming model.

Figure 3 details the three-stage compression process. Kernel ① loads an uncompressed block from global into shared memory, translating floating-point values to their integer representation. The n-dimensional Integer Lorenzo Transform then computes residuals in n passes over the block data in-place. The residuals are grouped into sequences of 32 single- or 64 double-precision values and encoded via vertical bit packing, resulting in one head word and a variable number of non-zero columns.

[4] https://www.khronos.org/registry/SYCL/specs/sycl-2020/html/sycl-2020.html

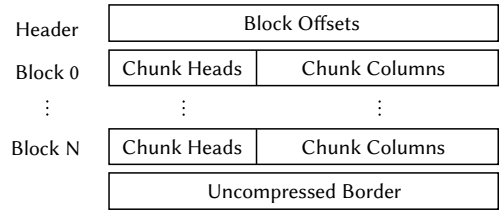

Figure 4: Compressed stream layout

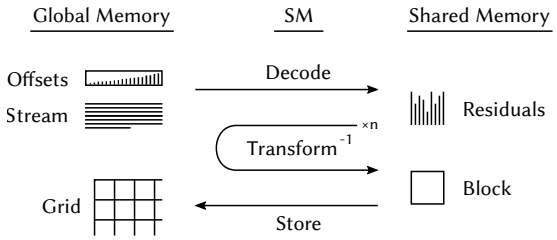

Figure 5: Single-stage decompression pipeline

A single global scratch buffer is allocated, providing enough space for the incompressible case. The index space is subdivided into *chunks*, reserving per block a single chunk for all head words followed by a smaller chunk for each sequence of bit-packed columns. All chunk offsets in the scratch buffer are known a priori from the dimensions of the input grid.

After encoding, each thread block writes their respective chunks to scratch memory and the chunk lengths to a separate buffer.

Kernel (2) computes a parallel prefix sum over the length buffer to obtain the offsets of all chunks in the compact output stream.

Finally, using the offset buffer, kernel (3) loads the chunks back from scratch memory and stores them at their final position in the output stream. The output offsets of the first chunk in each block are collected in the stream header.

The stream layout, visualized in Figure 4, purposely separates fixed-size meta-information (block offsets and chunk heads) from variable-length packed column encoding. This allows the decoder to compute absolute offsets of packed columns in parallel without requiring synchronization or multiple passes over the stream.

4.2 Decompression Pipeline Overview

Since offsets for each block in the compressed buffer can be retrieved from the stream header, decompression is output-parallel and does not need synchronization between blocks. A single kernel launch is sufficient to decode an entire stream or an arbitrary subset of blocks. Figure 5 details the decompression process of a single block.

The kernel first loads all heads from the first chunk, counts the set bits to obtain the number of non-zero columns per sequence, and finally performs a prefix sum to generate an offset table in shared memory. Bit packing of all chunks can then be reversed in parallel, expanding into the shared-memory block of residuals. The integer representation of uncompressed values is then restored by inverse Integer Lorenzo Transform. Finally, the floating point bit pattern is recovered by reversing the integer mapping, after which the block is written to the global output grid buffer.

4.3 Shared Memory Block Layout

The shared memory layout for intermediate results of the multi-pass transform step must be chosen carefully to avoid bank conflicts between all required access patterns. The hardware will split conflicting loads or stores into as many conflict-free accesses as necessary, which can dramatically increase the runtime of functions that are bound by shared-memory access, such as the Integer Lorenzo Transform. There is no obvious general solution to this problem, instead, the index space transformation must be specialized for the one-, two- and three-dimensional case as well as single- and double-precision data separately.

Padding. To ensure that consecutive indices for accessing the hypercube along all axes can map to non-overlapping banks, padding words are inserted. Since each memory bank is 32 bits wide and 64-bit loads and stores are executed as two sequential 32-bit accesses, padding in the double-precision case must still be 32 bits wide. This requires deliberately misaligned accesses to 64-bit values.

Directional Access Order. In each dimension of the transform step, iterating over the items of a lane can be modelled as a loop of fixed stride. However since each active warp processes 32 lanes simultaneously, the memory offset of the first item in each lane must be calculated explicitly. Partitioning the set of lanes must again be done carefully to avoid bank conflicts.

Table 1 shows one such possible conflict-free configuration of padding and offset computation. This is also the configuration used by our implementation of ndzip-gpu. Since forward and inverse Integer Lorenzo Transform expose different degrees of parallelism, compression and decompression require slightly different memory layouts. Both variants are compatible with grid load/store operations and the vertical bit packing scheme, so the block layouts shown are valid for the entire pipeline.

4.4 Parallel Integer Lorenzo Transform

The n-dimensional Integer Lorenzo Transform, both forward and inverse, consists of n passes. In every directional pass, L lanes can be processed in parallel; these lanes are distributed among T threads of the thread block (see Table 1b).

Forward Transform. The forward transform constructs residuals in $4096/L$ iterations per lane, replacing value representations with the integer difference to their predecessor. The predecessor value is tracked in a register, so this scheme only needs to perform one load and one store per data point in shared memory.

Inverse Transform. To reconstruct value representations, each inverse transform pass must add the already-decoded predecessor to each residual. Since this introduces a dependency chain equal in size to the block side length, there can be at most $4096^{1-1/n}$ independent lanes (1 for one, 64 for two and 256 for three dimensions).

Since the inverse transform of each lane constitutes a prefix sum, serialization can be avoided by employing a parallel scan. In practice we invert the one-dimensional transform by using a fast parallel prefix sum on the contiguous block memory and accepting limited occupancy for the two- and three-dimensional cases by performing sequential summation per lane.

Transform	Bits	n	32-bit padding word inserted
forward	32	1	every 32 values
forward	64	1	every 16 values
inverse	any	1	none
forward	32	2	every 32 values
forward	64	2	every 16 values
inverse	32	2	every 64 values
inverse	64	2	every 16, skip every 64 values
any	32	3	every 32 values
any	64	3	every 16, skip every 512 values

n: Grid dimensionality

Table 1a: Conflict-free block memory layout

Transform	n	k	L	Offset of first item in lane l	Stride
forward	1	0	T	$l \cdot (4096/L)$	1
inverse	1	0	1	0	1
forward	2	0	T	$l \cdot (64^2/L)$	1
forward	2	1	T	$(l/64) \cdot ((64^2/L) \cdot 64) + l \bmod 64$	64
inverse	2	0	64	$l \cdot 64$	1
inverse	2	1	64	$l \bmod 64$	64
any	3	0	16^2	$l \cdot 16$	1
any	3	1	16^2	$2L(l/16) - 2l/L \cdot 3840 + l \bmod 16$	16
any	3	2	16^2	l	16^2

n: Grid dimensionality k: Direction L: Number of lanes
T: Number of threads per thread block (configurable)

Table 1b: Directional block memory access patterns

4.5 Warp-Cooperative Vertical Bit Packing

Vertical bit packing of fixed-width integer sequences has seen prior application in Database systems [12]. This approach to compacting bit patterns with a length indivisible by a processor's smallest addressable unit is efficiently vectorized on parallel hardware such as SIMD-capable processors [19].

Instead of operating on the contiguous bits in an integer, it can be easily adapted to compact an arbitrary subset of input bit positions, again allowing highly-efficient implementation on SIMD architectures [11]. In this form, it has previously been used in GPU floating point compression as part of the MPC compressor [25].

In the following, we refer to unpacked words as *rows* and to bits of identical position in an uncompressed sequence as *columns*.

State of the Art. For packing, the MPC encoder[5] first transposes a set of rows via warp shuffles, buffering columns in shared memory before computing a thread-block wide prefix sum to obtain output positions compacting all non-zero columns in-place. It then busy-waits for a predecessor thread block to finish compaction in order to derive the global output position for the current data.

[5]https://userweb.cs.txstate.edu/~burtscher/research/MPC/MPC_float_12.cu, l. 205ff

When unpacking, MPC calculates relative positions of non-zero columns using a prefix sum. After again busy-waiting on a predecessor block to calculate the global read offset, it expands the compact stream into shared memory and repeats the transposition.

Profiling reveals that on contemporary hardware, MPC runtime is dominated by a thread re-convergence stall after busy-waiting on the stream position update in global memory.

The novel packing scheme of ndzip-gpu improves performance beyond the MPC approach on modern GPUs significantly by

(1) short-circuit-evaluating the expensive transposition step for all-zero blocks without thread divergence
(2) allowing independent forward progress of warps by avoiding block-wide synchronization entirely
(3) during packing, avoiding serialization around output stream positions by writing compressed chunks to a global scratch buffer and employing a separate compaction kernel
(4) during unpacking, avoiding serialization around input stream positions by reading coarse-grained block offsets from the stream header and calculating fine-grained chunk offsets within blocks as a parallel prefix sum.

Packing. In the ndzip-gpu encoder, 32 threads cooperate to pack 32 32-bit or 64 64-bit rows. Listing 1 shows the mechanism for the simpler 32-bit case, where one word corresponds to one thread.

First, the head, storing positions of zero and non-zero columns, is computed using warp-cooperative reduction.

In parallel, all 32 rows are staged into contiguous shared memory to avoid repeated addressing into the complex memory layout present after the transform step (see Table 1a).

Each thread then accumulates the column identified by its thread ID in a register. To accomplish this, it iterates over each row in the sequence, collecting one bit per iteration.

Zero-columns are eliminated by having each thread calculate the position of its column in the compressed chunk via a warp-cooperative prefix sum. This scan is efficiently implemented using warp shuffle operations and requires neither thread-block-level synchronization nor additional shared memory allocation.

Each thread holding a non-zero column then writes it to the now known position in the output chunk.

As an effective performance optimization, the entire packing step can be skipped whenever the head is zero and the chunk therefore known to be empty. Since the entire warp takes the same branch at this conditional, no adverse thread divergence occurs.

Unpacking. The decoding stage employs a similar thread assignment, shown for the 32-bit case in Listing 2. First, the length of each packed chunk is determined as the population count (`popcount`) of its head. From these lengths, the offset into the packed stream is computed using a thread-block parallel prefix sum.

The same zero-head optimization seen in the encoding stage is applied. The packed chunk with a maximum of 32 entries is first staged in local memory to reduce L1 cache pressure in the transposition loop. Then the original rows are reconstructed, again using one loop iteration per bit. Since the unpacked output positions are statically known, rows can finally be stored in shared memory without requiring further synchronization.

```
in shared rows
out global lengths, column_chunks
parallel for register c in chunks {
    register head = warp_bitwise_or_reduce(rows[c][tid])
    if head != 0 {
        shared stage[32]
        stage[tid] = rows[c][31 - tid]
        warp_barrier()
        register col = 0
        for i in [0,32) {
            col |= ((stage[i] >> (31 - tid)) & 1) << i
        }
        register off = warp_exclusive_prefix_sum(col != 0)
        if col != 0 {
            column_chunks[c][off] = col
        }
        lengths[c] = popcount(head)
    } else {
        lengths[c] = 0
    }
}
```

Listing 1: Cooperative vertical bit packing

```
in global heads, columns
out shared rows
shared lengths = [popcount(h) for h in heads]
shared offsets = parallel_exclusive_prefix_sum(lengths)
parallel for register c in chunks {
    register head = heads[c]
    if (head != 0) {
        shared stage[32]
        stage[tid] = columns[offsets[c] + tid]
        warp_barrier();
        register row = 0
        register off = 0
        for i in (32,0] {
            if ((heads[c] >> i) & 1) != 0 {
                col = stage[off]
                row |= ((col >> (32 - tid)) & 1) << i
                off += 1
            }
        }
        rows[c][tid] = row
    } else {
        rows[c][tid] = 0
    }
}
```

Listing 2: Cooperative vertical bit unpacking

On contemporary GPUs, 64-bit operations tend to be slower than their 32-bit equivalents due to the wider memory operations involved. However, 64-bit vertical bit packing can be realized without a loss in throughput as no real 64-bit arithmetic is required for correctness. Instead, the 64×64 bit matrix is logically split into four 32×32 quadrants which are processed independently.

4.6 Parameter Tuning

Since the ndzip format mandates a fixed block size, the most important tunable parameter is the number of threads per block. This number can be chosen independently from the rest of the implementation and allows trading cache locality for higher occupancy, which improves the ability to hide instruction latencies.

5 EVALUATION

Our implementation of ndzip-gpu is realized as a C++17 library using a custom version of hipSYCL [1], a cross-vendor CPU/GPU implementation of the SYCL 2020 programming model. We compare its performance against five competing GPU compressors with publicly available source code.

MPC 1.2[6] and GFC 2.2[7] are available as stand-alone CUDA programs. The source of both compressors was updated to CUDA 11, augmented with an interface for in-memory compression and instrumented to provide accurate kernel timings.

nvCOMP 2.0 is provided as a library[8] with an interface similar to the CUDA runtime API. We chose its LZ4 and Cascaded modes for comparison, which are both available for all evaluated platforms and produce a single compact compressed stream.

cudppCompress is part of the larger CUDPP 2.3 parallel primitives library[9]. In addition to syntactic fixes for compatibility with Clang and an update to CUDA 11, the sequence of compression kernel launches was instrumented to record GPU timings.

FPC 1.1[10] is provided as a file compressor and was adapted to allow in-memory compression.

Neither the code of liblzma 5.2.5[11], implementing the well-known LZMA compressor, nor fpzip 1.3.0[12] or ZFP 0.5.5[13] required any modifications for benchmarking.

The C++, SYCL and CUDA source code of ndzip-gpu, MPC, GFC, cudppCompress, FPC and fpzip was compiled for Linux on each host architecture using Clang 10 and the `-O3` optimization flag. nvCOMP, currently being incompatible with Clang, was compiled using NVCC while keeping the remaining configuration identical.

5.1 Test Data

Compressor performance and the achieved compression ratios were evaluated on data of varying dimensionality from real-world applications [10], shown in Table 2. Datasets are available in single precision, double precision, or both.

`msg_sppm` and `msg_sweep3d` are messages sent by a cluster node running ASCI Purple solvers. `snd_thunder` is a 32-bit float PCM audio recording. `ts_gas` is a time series of gas sensor readings. `ts_wesad` is a time series of physiological and motion sensor readings. `hdr_night` and `hdr_palermo` are luminance components of HDR photographs. `hubble` is an image taken by the Hubble space telescope. `rsim` is a radiosity field from room response simulation for time-of-flight imaging. `spitzer_fls_irac`, `spitzer_fls_vla` and

[6] https://userweb.cs.txstate.edu/~burtscher/research/MPC
[7] https://userweb.cs.txstate.edu/~burtscher/GFC
[8] https://github.com/NVIDIA/nvcomp/
[9] https://github.com/cudpp/cudpp
[10] https://userweb.cs.txstate.edu/~burtscher/research/FPC
[11] https://tukaani.org/xz
[12] https://github.com/LLNL/fpzip/releases/tag/1.3.0
[13] https://github.com/LLNL/zfp

`spitzer_frontier` are images from the Spitzer telescope. `asteroid` is the pressure component in an asteroid impact simulation. `astro_mhd` is the temperature component of a magnetohydrodynamic solar wind simulation. `astro_pt` is one velocity component of a particle transport simulation. `hurricane` is the precipitation component of a hurricane simulation. `magrecon` is one time step from a simulation of magnetic reconnection. `redsea` is the salt content of a sea eddy simulation. `sma_disk` is observational data from the Submillimeter Array. `turbulence` is one time step of a turbulent flow simulation. `wave` are multiple time steps of a wave propagation function.

5.2 Environment

All compressors discussed so far were evaluated on the following four GPU-accelerated systems:

- One node of the Marconi-100 supercomputer, featuring dual POWER9 AC922 CPUs with 256 GB RAM and four Nvidia **Tesla V100** *Volta* HPC GPUs (Compute Capability 7.0).
- One AMD Ryzen 9 3900X desktop system with 64 GB RAM and one Nvidia **RTX 2070 SUPER** mid-range *Turing* consumer GPU (Compute Capability 7.5)
- One Nvidia DGX A100[14] node featuring dual AMD EPYC 7742 CPUs with 1 TB RAM and eight Nvidia **A100** 40GB *Ampere* HPC GPUs (Compute Capability 8.0)
- One dual-socket AMD EPYC 7282 node with 256 GB RAM and four Nvidia **RTX 3090** high-end *Ampere* consumer GPUs (Compute Capability 8.6)

We did not specialize ndzip-gpu to exploit functionality present of the *Ampere* card that is not available on *Volta* or *Turing*. For our use case, the GPUs differ mainly in their shared memory size and maximum thread / warp allocation per SM[15].

5.3 Methodology

We define the compression ratio of a dataset as compressed size divided by uncompressed size in bytes, with lower ratios indicating better compression. This definition allows meaningful analysis of expected compression ratios from a set of observations using the unweighted arithmetic mean.

Compressor erformance was evaluated by measuring device execution time from the start of the first kernel to the end of the last kernel. Buffer allocation as well as host-device memory transfers are excluded from the measurements. We report the throughput of uncompressed bytes per second, which translates to compression input and decompression output bandwidth. Measurements for each algorithm–dataset pair were repeated until the total runtime exceeded one second, but at least five times.

nvCOMP Cascaded requires configuration of the compression pipeline ahead of time. Results vary with the GPU architecture, but parameters `num_RLEs=1`, `num_deltas=0` and `use_bp=1` result in the best average compression ratio for our test data in all situations, so this configuration was used for benchmarking.

All CPU algorithms were benchmarked by measuring execution time excluding all memory allocations that could be performed ahead of time. Since not all compressors can make use of multiple threads, we only evaluated single-threaded configurations. The CPU implementation of ndzip requires an x86_64 processor, so instead of the POWER9 Marconi-100, the Ryzen 9 3900X system was chosen for its high single-threaded performance.

[14] https://docs.nvidia.com/dgx/pdf/dgxa100-user-guide.pdf
[15] https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#features-and-technical-specifications

dataset	data type	dimensions	extent
`msg_sppm`	single, double	1	34,874,483
`msg_sweep3d`	single, double	1	15,716,403
`snd_thunder`	single	1	7,898,672
`ts_gas`	single	1	4,208,261
`ts_wesad`	single	1	4,588,553
`hdr_night`	single	2	8,192 × 16,384
`hdr_palermo`	single	2	10,268 × 20,536
`hubble`	single	2	6,036 × 6,014
`rsim`	single, double	2	2,048 × 11,509
`spitzer_fls_irac`	single	2	6,456 × 6,389
`spitzer_fls_vla`	single	2	8,192 × 8,192
`spitzer_frontier`	single	2	3,874 × 2,694
`asteroid`	single	3	500 × 500 × 500
`astro_mhd`	single	3	128 × 512 × 1024
`astro_mhd`	double	3	130 × 514 × 1026
`astro_pt`	single, double	3	512 × 256 × 640
`hurricane`	single	3	100 × 500 × 500
`magrecon`	single	3	512 × 512 × 512
`redsea`	single, double	3	50 × 500 × 500
`sma_disk`	single	3	301 × 369 × 369
`turbulence`	single	3	256 × 256 × 256
`wave`	single, double	3	512 × 512 × 512

Table 2: Scientific sample datasets

5.4 Discussion

Figure 6 and Table 3 demonstrate the superior trade-off between throughput and compression ratio offered by ndzip-gpu. On the evaluated test data, our novel parallelization scheme simultaneously delivers both the best average compression ratio and the highest throughput of all examined compressors in the single precision case. For double precision, the GFC and nvCOMP Cascaded schemes manage to exceed ndzip-gpu's speed on the RTX 2070 SUPER and A100 GPUs, albeit at a worse compression ratio.

Unlike both GFC and MPC, ndzip-gpu shows a notable discrepancy between compression and decompression speeds. This can be explained with the multi-stage architecture of the compressor, which requires a full global-memory round-trip for compaction.

Kernel Runtime Allocation. Figure 7 breaks down the average time spent per kernel on *Volta* hardware. The combined transform and encoding / decoding kernels of both pipelines show the largest contribution to total runtime. Performance is not identical between the two pipelines because although the underlying computations are similar, different memory access patterns and resulting optimization strategies make them not entirely comparable.

The compression pipeline has the compaction kernel as a second major contributor. Runtime of the remaining kernels is negligible; also border compaction can be overlapped with chunk compaction and border expansion with block decoding and transformation.

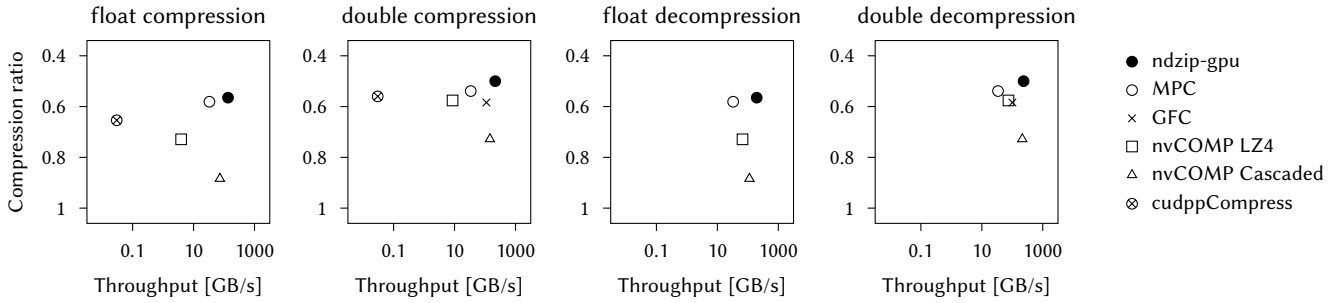

Figure 6: Average compression ratio compared to compressor / decompressor throughput on Tesla V100

Algorithm	Ratio	Compress GB/s				Decompress GB/s			
		V100	2070	A100	3090	V100	2070	A100	3090
single precision									
ndzip-gpu	**.565**	**135**	**81**	**162**	**177**	**196**	**107**	**273**	**241**
MPC	.581	33	38	24	40	33	39	24	41
nvC. LZ4	.729	3.9	2.8	8.6	5.6	68	48	81	80
nvC. Casc.	.884	73	45	91	92	113	58	169	138
cudppCo.	.654	.03	.04	.01	.03		—		
double precision									
ndzip-gpu	**.500**	**216**	98	**259**	**236**	**235**	**111**	324	**278**
MPC	.539	34	40	25	42	35	40	25	42
GFC	.584	111*	**126***	105*	136*	101	110	100	127
nvC. LZ4	.576	8.6	6.0	14	11	75	57	106	94
nvC. Casc.	.728	145	89	201	189	213	**111**	**359**	261
cudppCo.	.560	.03	.04	.01	.03		—		

* Excluding compaction of the compressed blocks

Table 3: GPU compressor benchmark results. Values report the arithmetic mean over the analyzed datasets.

Algorithm	*single precision*			*double precision*		
	Ratio	GB/s		Ratio	GB/s	
ndzip (CPU)	.565	**2.9**	**2.3**	.500	**3.0**	**2.6**
ZFP lossless	.693	.25	.22	.654	.32	.27
fpzip	**.475**	.15	.12	.416	.30	.23
FPC -15	—			.464	1.2	1.3
LZMA -9	.481	.01	.10	**.336**	.01	.20

Serial compression / decompression on AMD Ryzen 9 3900X

Table 4: Reference serial CPU benchmark results. Values report the arithmetic mean over the analyzed datasets.

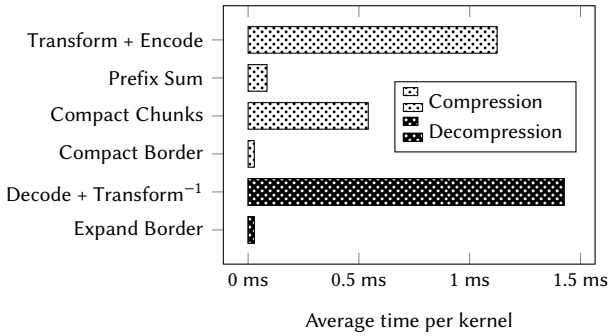

Figure 7: Average ndzip-gpu kernel runtimes on Tesla V100

Performance of Existing GPU Compressors. As a dictionary coder, the nvCOMP implementation of LZ4 shows a strong asymmetry between encoding and decoding speeds.

While the nvCOMP Cascaded scheme achieves exceptional compression and decompression throughput, its compression ratio is much worse than that of all other alternatives. As such, it is not well suited for compressing most discussed floating-point datasets.

The general-purpose cudppCompress is orders of magnitude slower than all specialized competitors. Since it does not provide a GPU-based decoder, only encoding performance is reported.

Despite its algorithmic similarities to ndzip-gpu, MPC achieves significantly lower throughput and an unusually consistent performance between systems. This matches our observations regarding the limitations of MPC's packing scheme (Section 4.5).

The compression throughput shown for GFC underestimates its total cost, since unlike all other studied compressors, GFC does not produce a contiguous compressed stream on the device. Instead, it relies on a chunked transfer back to the host for compaction. Similar to MPC, an unusual performance consistency between systems suggests that GFC would benefit from an updated implementation for newer microarchitectures.

Comparison with CPU-Based Compressors. Although the throughput numbers of most GPU compressors are out of reach for their CPU counterparts due to limited main memory bandwidth, a comparison is still appropriate to analyze the trade-offs involved in the choice between CPU and GPU compression.

The CPU variant of ndzip produces an identical compressed stream to ndzip-gpu. Following the comparison in Table 4, the

Algorithm	asteroid.f32	astro_mhd_temp.f32	astro_mhd_temp.f64	astro_pt_vx.f32	astro_pt_vx.f64	hdr_night.f32	hdr_palermo.f32	hubble.f32	hurricane.f32	magrecon.f32	msg_sppm.f32	msg_sppm.f64	msg_sweep3d.f32	msg_sweep3d.f64	redsea.f32	redsea.f64	rsim.f32	rsim.f64	sma_disk.f32	snd_thunder.f32	spitzer_fls_irac.f32	spitzer_fls_vla.f32	spitzer_frontier.f32	ts_gas.f32	ts_wesad.f32	turbulence.f32	wave.f32	wave.f64
Compression ratio																												
ndzip-gpu	**.29**	.07	.08	**.47**	**.70**	.87	.72	.41	**.79**	.83	.32	.39	.75	.83	.14	.15	**.51**	**.63**	.92	.83	.77	.42	.29	.57	.60	**.81**	**.50**	**.72**
MPC	.62	.09	.12	.64	.79	.81	**.67**	**.38**	.81	**.78**	.32	.38	.72	.81	.11	.16	.65	.71	**.90**	**.82**	**.74**	**.41**	**.27**	**.55**	**.42**	.83	.66	.79
GFC	—	—	.11	—	.91	—	—	—	—	—	—	.39	—	.89	—	.15	—	.73	—	—	—	—	—	—	—	—	—	.92
nvCOMP Casc.	1.0	**.04**	**.04**	1.0	1.0	1.0	1.0	.69	1.0	.88	.45	.81	1.0	1.0	.12	.10	1.0	.90	1.0	1.0	1.0	.74	.44	.81	.78	.91	1.0	1.0
nvCOMP LZ4	.46	**.04**	**.04**	1.0	1.0	.93	.91	.48	1.0	1.0	**.12**	**.23**	.98	.98	**.09**	**.07**	.76	.76	1.0	1.0	.81	.44	.42	.99	.91	1.0	.97	.94
cudppCompress	.51	.15	.16	.91	.98	**.78**	.68	.50	.98	.89	.22	.25	**.41**	**.78**	.17	.16	.75	.77	.97	.98	.85	.53	.37	.86	.51	.96	.73	.82
Compression throughput (GB/s)																												
ndzip-gpu	**184**	264	386	**143**	**155**	**132**	**137**	**183**	**100**	**128**	**159**	**203**	**121**	**148**	153	**300**	**111**	**169**	**118**	**95**	**138**	**178**	**125**	**69**	**74**	**80**	**146**	**155**
MPC	32	32	34	33	33	33	32	33	33	34	33	34	31	33	32	34	33	34	32	31	34	33	32	32	32	33	32	34
GFC	—	—	143	—	93	—	—	—	—	—	—	108	—	93	—	139	—	95	—	—	—	—	—	—	—	—	—	103
nvCOMP Casc.	65	**288**	**387**	47	72	48	38	87	45	44	126	102	47	69	**156**	238	56	84	47	40	58	96	91	34	36	43	48	65
nvCOMP LZ4	4.3	19	33	1.7	1.7	1.7	2.0	3.7	1.7	1.7	18	11	2.1	1.7	4.3	8.6	2.5	2.2	1.6	2.5	2.0	3.9	2.5	2.1	1.6	1.9	1.6	1.8
cudppCompress	.03	.03	.03	.03	.04	.03	.03	.03	.03	.04	.03	.03	.03	.03	.03	.03	.03	.03	.04	.04	.03	.03	.03	.03	.03	.04	.03	.03
Decompression throughput (GB/s)																												
ndzip-gpu	**248**	**507**	411	**187**	**171**	**172**	**174**	**224**	**147**	**185**	**266**	**195**	**139**	**161**	172	365	**163**	**170**	**163**	**122**	**175**	**266**	**213**	**123**	**126**	**162**	**189**	**172**
MPC	35	34	35	34	35	34	33	34	34	35	32	35	32	34	33	33	33	35	33	32	33	35	34	31	31	32	35	35
GFC	—	—	135	—	89	—	—	—	—	—	—	89	—	86	—	133	—	88	—	—	—	—	—	—	—	—	—	89
nvCOMP Casc.	87	438	**549**	74	105	69	69	128	65	68	181	137	68	94	**296**	**387**	80	109	73	70	83	120	143	66	59	73	71	107
nvCOMP LZ4	90	60	102	119	114	62	77	58	85	117	76	54	24	52	13	26	19	67	135	46	112	129	8.3	13	7.1	76	100	108
cudppCompress	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—

Table 5: Per-dataset compression ratio and throughput achieved by each GPU compressor on Tesla V100

lossless mode of ZFP achieves only moderate volume reduction at a much lower speed. fpzip, FPC and LZMA all achieve better compression ratios than ndzip, with fpzip delivering the best single-precision- and LZMA the best double-precision results.

Compressor Efficiency per Dataset. Table 5 lists the compression ratio and throughput achieved per dataset by each compressor. While ndzip-gpu achieves the best data reduction and highest throughput on average, some datasets can be compressed more effectively or quicker by competitor algorithms.

Ratios for ndzip and MPC are very close for most inputs since both algorithms share the same residual coding algorithm. The largest differences are visible for `asteroid.f32` in favor of ndzip-gpu and `ts_wesad.f32` in favor of MPC. A likely explanation is that the three-dimensional `asteroid.f32` is strongly correlated in multiple dimensions while the `ts_wesad.f32` time series interacts favorably with the second prediction step of MPC.

In a variety of other datasets, MPC holds a minute, but consistent advantage in compression ratio of around two percentage points. This is expected for data that is not strongly correlated in multiple dimensions, where ndzip's block subdivision can limit effectiveness by leaving some border elements uncompressed.

GFC never manages to outperform all other evaluated compressors in terms of compression ratio.

Both nvCOMP schemes are unique in that they compress a few select datasets exceptionally well (`astro_mhd_temp.*`, `redsea.*`) while being completely ineffective for many other datasets.

Although cudppCompress exhibits the lowest average compression ratio, it performs best for some of the unstructured `msg_*` datasets. This is likely due to other compressors' assumptions about data smoothness being violated for these streams.

With the exception of MPC and cudppCompress, compressor throughput of all implementations shows an inverse relationship to compression ratio: Highly compressible data consumes less global memory bandwidth on output and, in the case of ndzip-gpu, allows the compressor to short-circuit part of its computation.

The same relationship exists in the decompressor pipelines of ndzip-gpu, GFC and nvCOMP Cascaded, but is notably absent in nvCOMP LZ4, for which compressor and decompressor throughput appear to be uncorrelated.

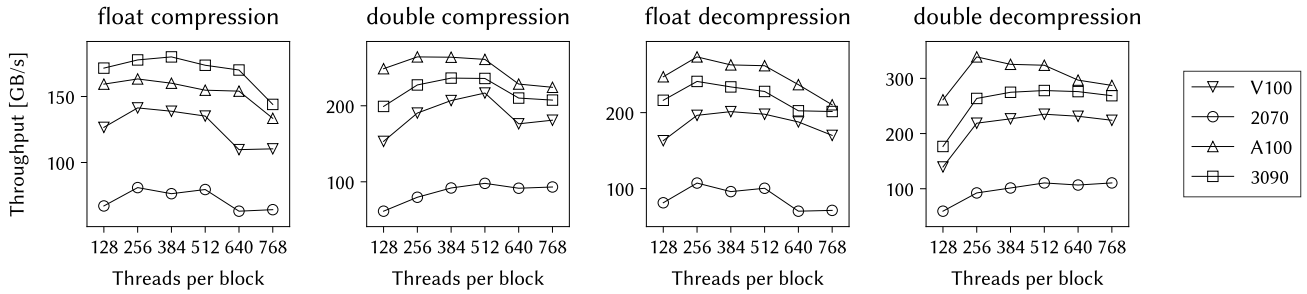

Figure 8: Effect of thread block size on compressor and decompressor throughput

Parameter Tuning Results. Figure 8 shows the effect of varying the thread block size parameter, sampled at multiples of 128. For small block sizes, occupancy is limited by the high shared memory requirements of the transform kernel. A higher thread count will increase occupancy within a thread block but will limit the number of blocks resident per SM. Since shared memory and thread count limits per SM vary between architectures, the optimal configuration is hardware dependent. To obtain close-to-optimal performance on all hardware studied in this paper, our implementation chooses a default block size of 256 threads for single-precision and 512 for double-precision pipelines.

Suitability for Network Communication. To avoid underutilizing a system's network link, throughput of compressed data (as opposed to uncompressed data throughput, which was discussed so far) must exceed the link bandwidth. Since all high-efficiency GPU compressors are memory bound, this becomes increasingly difficult as compression effectiveness rises.

For our reference system Marconi-100, we follow the throughput measurement in Table 5, assuming the minimum of compressor and decompressor throughput.

Multiplying with the compression ratio per dataset, we conclude that a single Tesla V100 GPU is able to saturate the 25 GB/s Infiniband EDR interconnect for all except two single-precision and all double-precision datasets studied. The two exceptions, `redsea.f32` and `astro_mhd_temp.f32`, occupy a theoretical bandwidth of 18 and 21 GB/s respectively, slightly underutilizing the link.

Saturation is thus easily achievable in the average case.

6 CONCLUSION

We presented ndzip-gpu, an efficient parallelization scheme for the state-of-the art ndzip compressor, targeting multi-dimensional dense grids of floating-point data. Through the combination of a block transform step optimized for data locality and an efficient warp-cooperative vertical bit packing primitive, ndzip-gpu makes excellent use of modern GPU hardware.

By comparing ndzip-gpu against five other, existing lossless GPU compressors, we observe that compressibility strongly depends on interactions between the decorrelation strategy of the algorithm and the structure of the dataset in question. For instance, a multi-dimensional grid with strong correlation along more than one dimension will be compressed very effectively by ndzip-gpu, whereas some unstructured datasets can be decorrelated more effectively by stream compressors like MPC or dictionary coders such as LZ4.

Over the representative set of single- and multi-dimensional test data considered in this paper however, ndzip-gpu delivers the most effective average data reduction of all evaluated algorithms on both single- and double-precision data.

On the Tesla V100 GPUs of the Marconi-100 supercomputer, our method achieves average compression speeds of 135 and 216 GB/s and decompression speeds of 196 and 235 GB/s for single and double precision data respectively, significantly outperforming all compared state-of-the-art compressors. Performance results vary with the compute power and memory bandwidth of the device, and on other GPUs, two competing algorithms are able to exceed the throughput of ndzip-gpu in the double precision case at the cost of a worse compression ratio.

For most evaluated datasets, the compressed data stream form a single GPU running ndzip-gpu is sufficient for exceeding the system interconnect bandwidth, promising benefits for communication-bound applications in the future.

The source code of our ndzip-gpu implementation is publicly available on GitHub[16].

6.1 Future Work

We are currently evaluating the integration of ndzip-gpu in Celerity [21], a distributed-memory runtime for accelerator clusters that is especially well-suited for dense-grid algorithms. As all data movement is managed transparently by the runtime, Celerity can automatically decide which data transfers will profit from lossless compression by ndzip-gpu.

Once a future version of the SYCL standard exposes whole-device synchronization, we will be able to further increase compressor performance by avoiding the global-memory round-trip currently necessary for compaction. This will require intelligent load balancing to avoid negating the benefit of zero-head short-circuit evaluation in the residual coder.

6.2 Acknowledgement

This research is supported by the D–A–CH project CELERITY, funded by FWF project I3388, and the European High-Performance Computing Joint Undertaking (JU) project LIGATE under grant agreement No 956137.

[16] https://github.com/fknorr/ndzip

REFERENCES

[1] Aksel Alpay and Vincent Heuveline. 2020. SYCL beyond OpenCL: The architecture, current state and future direction of hipSYCL. In *Proceedings of the International Workshop on OpenCL*. 1–1.

[2] Ana Balevic. 2009. Parallel Variable-Length Encoding on GPGPUs. In *European Conference on Parallel Processing*. Springer, 26–35.

[3] Martin Burtscher and Paruj Ratanaworabhan. 2008. FPC: A high-speed compressor for double-precision floating-point data. *IEEE Trans. Comput.* 58, 1 (2008), 18–31.

[4] M. Burtscher and P. Ratanaworabhan. 2009. pFPC: A parallel compressor for floating-point data. In *2009 Data Compression Conference*. IEEE, 43–52.

[5] Franck Cappello, Sheng Di, Sihuan Li, Xin Liang, Ali Murat Gok, Dingwen Tao, Chun Hong Yoon, Xin-Chuan Wu, Yuri Alexeev, and Frederic T Chong. 2019. Use cases of lossy compression for floating-point data in scientific data sets. *The International Journal of HPC Applications* 33, 6 (2019), 1201–1220.

[6] Aditya Deshpande and PJ Narayanan. 2015. Fast burrows wheeler compression using all-cores. In *2015 IEEE International Parallel and Distributed Processing Symposium Workshop*. IEEE, 628–636.

[7] Shunji Funasaka, Koji Nakano, and Yasuaki Ito. 2015. A parallel algorithm for LZW decompression, with GPU implementation. In *International conference on parallel processing and applied mathematics*. Springer, 228–237.

[8] Song Huang, Shucai Xiao, and Wu-chun Feng. 2009. On the Energy Efficiency of Graphics Processing Units for Scientific Computing. In *2009 IEEE International Symposium on Parallel & Distributed Processing*. IEEE, 1–8.

[9] Lawrence Ibarria, Peter Lindstrom, Jarek Rossignac, and Andrzej Szymczak. 2003. Out-of-core compression and decompression of large n-dimensional scalar fields. In *Computer Graphics Forum*, Vol. 22. Wiley Online Library, 343–348.

[10] Fabian Knorr, Peter Thoman, and Thomas Fahringer. 2020. Datasets for Benchmarking Floating-Point Compressors. *arXiv e-prints*, Article arXiv:2011.02849 (Nov. 2020), arXiv:2011.02849 pages. arXiv:2011.02849 [cs.DC]

[11] Fabian Knorr, Peter Thoman, and Thomas Fahringer. 2021. ndzip: A High-Throughput Parallel Lossless Compressor for Scientific Data. In *2021 Data Compression Conference*. IEEE. https://dps.uibk.ac.at/~fabian/publications/2021-ndzip-a-high-throughput-parallel-lossless-compressor-for-scientific-data.pdf

[12] Yinan Li and Jignesh M Patel. 2013. Bitweaving: Fast scans for main memory data processing. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. 289–300.

[13] Peter Lindstrom. 2014. Fixed-rate compressed floating-point arrays. *IEEE transactions on visualization and computer graphics* 20, 12 (2014), 2674–2683.

[14] Peter Lindstrom and Martin Isenburg. 2006. Fast and efficient compression of floating-point data. *IEEE Transactions on Visualization and Computer graphics* 12, 5 (2006), 1245–1250.

[15] Robert Lucas, James Ang, Keren Bergman, Shekhar Borkar, William Carlson, Laura Carrington, George Chiu, Robert Colwell, William Dally, Jack Dongarra, et al. 2014. *DOE advanced scientific computing advisory subcommittee (ASCAC) report: Top ten exascale research challenges.* Technical Report. USDOE Office of Science (SC)(United States).

[16] Molly A O'Neil and Martin Burtscher. 2011. Floating-point data compression at 75 Gb/s on a GPU. In *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units*. 1–7.

[17] Adnan Ozsoy and Martin Swany. 2011. CULZSS: LZSS lossless data compression on CUDA. In *2011 IEEE International Conference on Cluster Computing*. IEEE, 403–411.

[18] Ritesh A Patel, Yao Zhang, Jason Mak, Andrew Davidson, and John D Owens. 2012. *Parallel lossless data compression on the GPU.* IEEE.

[19] Orestis Polychroniou and Kenneth A Ross. 2015. Efficient lightweight compression alongside fast scans. In *Proceedings of the 11th International Workshop on Data Management on New Hardware*. 1–6.

[20] Seung Woo Son, Zhengzhang Chen, William Hendrix, Ankit Agrawal, Wei-keng Liao, and Alok Choudhary. 2014. Data compression for the exascale computing era-survey. *Supercomputing frontiers and innovations* 1, 2 (2014), 76–88.

[21] Peter Thoman, Philip Salzmann, Biagio Cosenza, and Thomas Fahringer. 2019. Celerity: High-Level C++ for Accelerator Clusters. In *European Conference on Parallel Processing*. Springer, 291–303.

[22] Jiannan Tian, Cody Rivera, Sheng Di, Jieyang Chen, Xin Liang, Dingwen Tao, and Franck Cappello. 2020. Revisiting Huffman Coding: Toward Extreme Performance on Modern GPU Architectures. *arXiv preprint arXiv:2010.10039* (2020).

[23] Oreste Villa, Daniel R Johnson, Mike Oconnor, Evgeny Bolotin, David Nellans, Justin Luitjens, Nikolai Sakharnykh, Peng Wang, Paulius Micikevicius, Anthony Scudiero, et al. 2014. Scaling the power wall: a path to exascale. In *SC'14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 830–841.

[24] André Weißenberger and Bertil Schmidt. 2019. Massively Parallel ANS Decoding on GPUs. In *Proceedings of the 48th Int. Conference on Parallel Processing*. 1–10.

[25] Annie Yang, Hari Mukka, Farbod Hesaaraki, and Martin Burtscher. 2015. MPC: a massively parallel compression algorithm for scientific data. In *2015 IEEE International Conference on Cluster Computing*. IEEE, 381–389.