Parallelization of variable rate decompression through metadata

Noordsij, Lennart; Vlugt, Steven Van Der; Bamakhrama, Mohamed A.; Al-Ars, Zaid; Lindstrom, Peter

DOI
10.1109/PDP50117.2020.00045

Publication date
2020

Document Version
Accepted author manuscript

Published in
Proceedings - 2020 28th Euromicro International Conference on Parallel, Distributed and Network-Based Processing, PDP 2020

Parallelization of Variable Rate Decompression through Metadata

Lennart Noordsij*, Steven van der Vlugt*, Mohamed A. Bamakhrama†‡, Zaid Al-Ars§ and Peter Lindstrom¶
*ASML, Netherlands
†Synopsys Corporation, Netherlands
‡Leiden University, Netherlands
§TU Delft, Netherlands
¶Lawrence Livermore National Laboratory, United States of America

***Abstract*—Data movement has long been identified as the biggest challenge facing modern computer systems' designers. To tackle this challenge, many novel data compression algorithms have been developed. Often variable rate compression algorithms are favored over fixed rate. However, variable rate decompression is difficult to parallelize. Most existing algorithms adopt a single parallelization strategy suited for a particular HW platform. Such an approach fails to harness the parallelism found in diverse modern HW architectures. We propose a parallelization method for tiled variable rate compression algorithms that consists of multiple strategies that can be applied interchangeably. This allows an algorithm to apply the strategy most suitable for a specific HW platform. Our strategies are based on generating *metadata* during encoding, which is used to parallelize the decoding process. To demonstrate the effectiveness of our strategies, we implement them in a state-of-the-art compression algorithm called ZFP. We show that the strategies suited for multicore CPUs are different from the ones suited for GPUs. On a CPU, we achieve a near optimal decoding speedup and an overhead size which is consistently less than 0.04% of the compressed data size. On a GPU, we achieve average decoding rates of up to 100 GiB/s. Our strategies allow the user to make a trade-off between decoding throughput and metadata size overhead.**

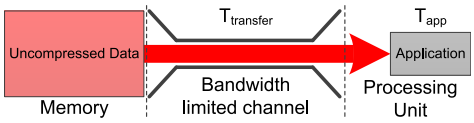
I. Introduction

Memory bandwidth and data movement have long been identified as the biggest challenge facing computer systems' designers in the current decade [1]. This challenge becomes even harder in massively data-parallel architectures such as GPUs [2]. In order to overcome the infamous "Memory Wall", a lot of research has been focusing on novel *online* data compression techniques [3]. In online compression, the data encoding and/or decoding happens at run-time as part of another application. The primary goal of online compression is to reduce the amount of data that the application has to transfer into/from memory at the expense of sacrificing extra compute power to perform the encoding/decoding steps. As the memory bandwidth gap worsens, any savings in data transfer time lead to substantial savings in the total application execution time, even with the additional overhead used to encode/decode the data. This is especially important when offloading computations to external acceleration devices, where the bandwidth of the interface to the device is often orders of magnitude lower than the internal memory bandwidth. In contrast to online compression, offline compression focuses solely on reducing, as much as possible, the compressed data size. At first look, both online and offline compression seem to share the same final goal. However, online compression focuses, in addition to reducing the compressed data size, on reducing the time overhead of encoding and decoding. To this aim, many online compression algorithms focus on utilizing modern architectural features such as vector instructions, multicore CPUs, and caches [3]. In many real-time and embedded systems, data *decompression* is in the critical path of the application execution and compression is either done offline or is done online but outside the critical path. In such applications it is important to speed up the decompression as much as possible.

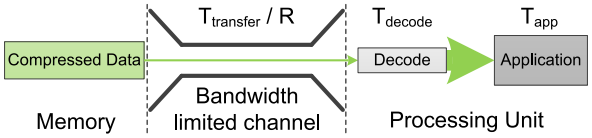
Compression algorithms can be analyzed based on many metrics. In this work, we focus on *compression ratio* and *data granularity*. An algorithm is said to be a *tiled* algorithm if it divides the input data into *tiles* or *blocks* (i.e., groups) of n values and then compresses each tile independently from other tiles. In this work we will use tiled algorithms, as the independence of tiles allows for tile-level parallel compression. Furthermore we can classify compression algorithms into: (i) *fixed rate*, and (ii) *variable rate*. In fixed rate compression, each tile in the original dataset is encoded using a *fixed* number of bits. This means that the location of each compressed tile in the compressed bitstream can be calculated once we know the *compression ratio*. In contrast, with variable rate compression, the number of bits per compressed tile *varies* across the data. This means that in variable rate decompression, finding the location of each compressed tile is much harder as we need to accumulate the bit lengths of all previous compressed tiles in the bitstream. Variable rate modes are often favored because of their generally higher compression ratios compared to fixed rate modes. Therefore, we aim to accelerate variable rate decompression.

A. Problem Statement

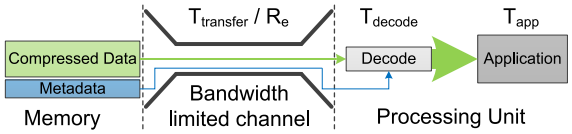
Accelerating variable rate decompression has been the focus of many studies [4]–[7]. However, most of the existing literature addresses the acceleration problem by adopting a particular HW platform and a particular execution strategy suitable for that platform. Such an approach *couples* the algorithm with the acceleration strategy. This coupling makes it difficult to achieve good performance on different HW



(a) Execution path without compression



(b) Execution path with compression



(c) Execution path with compression including metadata

Figure 1: Execution paths from Memory to Processing Unit

platforms. In this work, we alleviate this problem by *decoupling* the decompression method from the parallel execution strategy. This allows us to find a suitable parallelization strategy for each platform, without having to adjust the algorithm.

B. Paper Contributions

We propose to parallelize variable rate decompression by generating *extra information* during the encoding phase, and passing this to the decoder as *metadata*. Using this metadata, the decoder can compute very quickly the location of each tile in the compressed bitstream. We compare three different strategies to encode the metadata and show that the content and granularity have a large influence on decoding throughput across different HW platforms such as CPUs and GPUs. We also show that in many cases the size overhead introduced by the additional metadata is very small compared to the compressed data size.

II. Background

A *bandwidth-bound* application is one where the majority of the total execution time is spent in data transfers. As mentioned in Section I, compression can be used to reduce the data transfer time since a reduction in data size is also a reduction in transfer time. We will show this by comparing execution with and without compression.

Let T_{transfer} be the time needed to transfer the data over the bandwidth-limited channel, and T_{app} be the time needed to process the data once it arrives. Assume that data transfer and computation are pipelined. Then, it follows that the latency and period of the system *without* compression are given by:

$$\text{Latency} = T_{\text{transfer}} + T_{\text{app}} \tag{1}$$

$$\text{Period} = \max\left(T_{\text{transfer}}, T_{\text{app}}\right) \tag{2}$$

This situation is illustrated in Figure 1a. Now, suppose that we compress the data before sending it with a compression ratio denoted by R, as shown in Figure 1b. Our working assumption is that the data is decompressed upon arrival. It follows that the latency and period of the system *with* compression are given by:

$$\text{Latency} = \frac{T_{\text{transfer}}}{R} + T_{\text{decode}} + T_{\text{app}} \tag{3}$$

$$\text{Period} = \max\left(\frac{T_{\text{transfer}}}{R}, T_{\text{decode}} + T_{\text{app}}\right) \tag{4}$$

It is straightforward to see from Equations 3 and 4 that the data transfer time is reduced by a factor R. However, as shown in Figure 1b, a decode step is added that has its own latency (denoted by T_{decode}). Therefore, it is important to keep the decoding time as short as possible. In this work, we achieve that by *parallelizing* the decode step using different strategies that aim to maximize the utilization of different modern parallel HW platforms.

As mentioned in Section I, in this work we focus on tiled compression algorithms. There are different examples of such algorithms in the literature such as ZFP [8] and GFC [5]. We chose to use ZFP to prototype our solution and demonstrate its advantages. ZFP is a state-of-the-art compression algorithm that is able to achieve high data compression ratios for correlated floating-point or integer data. In short, the compression steps of ZFP can be described as follows [9]:

1) Divide the array into blocks (i.e., tiles)
2) Apply a decorrelation transformation on each block
3) Encode the blocks with either fixed or variable rate

The first step shows clearly that ZFP is a tiled algorithm as it divides the data in tiles, from now on referred to as blocks. The number of values in a block is set to 4^d, where d is the dimensionality of the data (i.e., 1D, 2D, 3D, etc.). An advantage of tiling, as mentioned earlier in Section I is that, after step (1), the resulting blocks are independent. This means that every subsequent operation applied to a block does not require any information from other blocks. As a result, all blocks can be encoded *in parallel*.

To decode a specific block from a compressed bitstream, we have to apply the inverse of the compression steps. This means that we need to know the bit position corresponding to the start of this compressed block in the bitstream. We call this position the *block offset* and define it as follows:

Definition 1. (Block Offset) The **block offset** of block n, denoted by $D(n)$, is the number of bits between the start of a compressed bitstream and the starting bit of the nth compressed block, with $n \geq 0$ and L_i the length of compressed block i in bits. $D(n)$ is given by:

$$D(n) = \sum_{i=0}^{n-1} L_i \tag{5}$$

Recall from Section I that in fixed rate the blocks in the compressed bitstream have the same compression ratio. This means that a fixed-rate compressed block length is the original block length divided by the compression ratio R.

In the case of ZFP, the length of an uncompressed block in bits is given by $4^d \cdot p$, where p is the precision of the input data type in number of bits. Substituting this in Equation 5 gives:

$$D(n) = \sum_{i=0}^{n-1} L_i = n \cdot \frac{4^d \cdot p}{R} \tag{6}$$

This means that the offset of any block in a fixed-rate compressed bitstream can be computed easily if the compression ratio R is known. As these parameters are constant and known at encode- and decode-time, fixed rate decompression is easy to parallelize, as is done in ZFP [10].

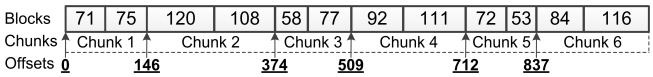
On the other hand, with variable rate compression, the compressed block length is not constant. This means that the expression from Equation 5 does not simplify like in Equation 6. It remains a summation where the offset of the current block depends on the accumulated length of all previous blocks. Hence, the challenge in parallel variable rate decompression is to find a solution to speed up the computation of this summation.

III. Related Work

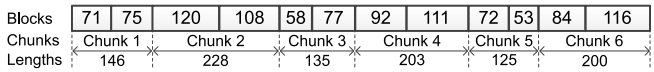
In [4], the author implemented a parallel variable length encoding (PVLE) scheme suited for integer data sets. The scheme is based on saving the length of each *codeword* and then performing a parallel *prefix sum* [11] on the array of lengths. Our work differs from [4] in two main aspects. First, we explore different schemes for generating the metadata needed to parallelize the decompression process and show that the lengths based approach used in [4] is not the most suitable one for modern GPUs. Second, we target different parallel architectures (multicore CPUs and GPUs) and show that our approach allows us to use different parallel execution strategies based on the architecture.

In [5], the authors proposed an algorithm, called GFC, for parallel decompression on GPUs. GFC shares many similarities with the ZFP algorithm used in our work. Both algorithms are tiled compression algorithms and support floating-point data. However, compared to GFC, our work differs in decoupling the generation of the metadata (needed for parallel decoding) from the underlying algorithm. This decoupling enables us to try different execution policies for different HW platforms. Our execution policies can, in principle, be applied to GFC as well.

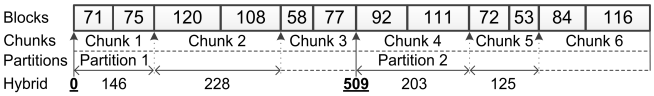
In [6] an approach to parallelize JPEG is proposed. It is based on placing restart markers in the compressed stream, which creates blocks that can be decompressed independently. Then the bit locations of these restart markers with respect to the start of the bitstream, the offsets, are stored and used to parallelize the decoding. This is very similar to our offsets strategy. The difference with our approach is that this method requires insertion of restart markers into the compressed bitstream, which means the parallelization strategy and algorithm are not completely decoupled. This is needed because in JPEG the blocks are coupled. We only target algorithms where the blocks are independent. Furthermore, we provide different methods of encoding the



(a) Strategy 1, bold underlined values are encoded offsets



(b) Strategy 2, values under the chunks are encoded lengths



(c) Strategy 3, with $P = 3$, bold underlined values are chunk offsets, others values are chunk lengths

Figure 2: Metadata under different strategies for 12 blocks of the Brain benchmark [12] that was compressed using ZFP with a fixed accuracy of 0.05. For all strategies we used $C = 2$, values inside the blocks are the block lengths in bits

information besides simply storing the offsets. This means that our proposed framework is more flexible, as it allows the user to trade off compression ratio for decoding speed.

In [7], the authors proposed a new approach to parallel decoding based on *speculative execution*. They developed a prediction algorithm that tries to predict block offsets in the compressed bitstream. These predictions are fed into a parallel decoding stage. For the correctly predicted offsets the decoded blocks are merged into the decompressed result, while mis-predictions are discarded. This process continues until all blocks are correctly decoded. The approach in [7] does not incur extra metadata overhead. However due to the mis-prediction penalty their speedup is limited when scaling-up the number of cores used. They report maximum speedups of 8.53, 6.42 and 1.20 for bzip2, H.264 and zlib respectively on a 36 core platform. Also, they show that the maximum speedup in these cases is achieved when using 14, 18 and 3 cores. This shows that their approach is not scalable into the thousands of cores and therefore not efficient on GPUs. For applications where decode latency should be minimized, speculative decoding is not suitable.

IV. Proposed Solution

As mentioned in Section II, in order to decode blocks in parallel, the block offsets have to be known before decoding starts. These block offsets can be stored directly or can be computed with the sum in Equation 5. If one has only the compressed bitstream, neither the block offsets nor the block lengths needed for the sum are known at the decoder. Therefore our solution is to generate *metadata*, containing this information, during encoding. Figure 1c illustrates the concept by depicting the execution path with compression and the addition of the metadata.

Sending extra data over a bandwidth-limited communication channel seems counter-intuitive, since it will increase data transfer time. However, it allows us to reduce the decoding time massively. There is a trade-off between the decoder speedup and the introduced metadata size overhead.

The metadata can be compressed itself to reduce its size further, for example using Binary Interpolative Coding [13]. However, for the sake of simplicity, in the rest of this paper we assume that it is sent as is. We start introducing our solution by defining an *overhead factor* and an *effective compression ratio* as shown in Definition 2 and Definition 3.

Definition 2. (Overhead Factor) Let S be the total size of the original uncompressed data in bits, R be the compression ratio and I be the total number of bits required to encode the metadata. The **overhead factor**, denoted by f, is given by:

$$f = \frac{I}{S/R} \tag{7}$$

Definition 3. (Effective Compression Ratio) The **effective compression ratio**, denoted by R_e, is the original uncompressed data size divided by the total amount of data transferred over the communication channel. R_e is given by:

$$R_e = \frac{S}{S/R + I} = \frac{R}{1+f} \tag{8}$$

It can be seen from Equation 8 that the effective compression ratio depends merely on two factors: (i) the average compression ratio R, and (ii) the overhead factor introduced in Definition 2. To minimize the impact of the metadata on the transfer time, one would like to minimize the metadata size I with respect to the original data size S, as this in turn minimizes f. To explain our strategies which minimize I and still allow for a large decoding speedup, we will introduce a number of definitions. Given a block with length L, we define the number of bits required to encode L as I_L and it is given by:

$$I_L = \lceil \log_2(L) \rceil \tag{9}$$

Similarly, given a block with block offset $D(n)$, then the number of bits required to encode $D(n)$ is denoted by $I_{D(n)}$ for $n > 0$ and it is given by:

$$I_{D(n)} = \left\lceil \log_2 \left(\sum_{i=0}^{n-1} L_i \right) \right\rceil \tag{10}$$

Finally, we define I_D to be the maximum number of bits required to encode an offset for any block in a compressed bitstream of N blocks with an average data compression ratio of R. As an offset is a cumulative sum of lengths, the largest offset is the one of the last compressed block in the bitstream. Then I_D is given by:

$$I_D = \left\lceil \log_2 \left(\sum_{i=0}^{N-1} L_i \right) \right\rceil \leq \left\lceil \log_2 \left(\frac{S}{R} \right) \right\rceil \tag{11}$$

Now, we are ready to detail **what** and **how much** extra information we send to compute the block offsets before we start decoding. In the following sub-sections we introduce our strategies for encoding the metadata.

A. Strategy 1: Offset Encoding

The first and simplest strategy is to plainly transfer the block offsets. This means that the metadata can be used without any preprocessing step during decoding. However, from Equation 11, it can be seen that the number of bits required to encode offsets grows with the original data size S, resulting in significant transfer size overhead. To reduce this, we introduce the concept of *chunks*.

Definition 4. (Chunk) A **chunk** is a set of C consecutive compressed blocks to be decoded by a single thread.

We group the blocks in chunks of granularity C and encode one offset per chunk rather than one offset per block. As a result, during decompression, every thread decodes a chunk instead of a block. This reduces the metadata size by a factor C. However, the increase in blocks per thread introduces the following two issues:

1) A decoding dependency between blocks in a chunk, which introduces a control loop
2) The access granularity becomes more coarse by a factor C

The impact of these issues differs per HW architecture and will be explored later.

Figure 2a gives a visual impression of blocks and chunks for a selection of 12 compressed blocks from the Brain benchmark [12].

B. Strategy 2: Length Encoding

From Equation 5, it can be seen that it is also possible to compute any offset as a cumulative sum of the lengths of the previous blocks. Recall that for an uncompressed ZFP block, L is constant and is equal to $4^d \cdot p$. Assume $R \geq 1$ for each block, meaning the length of any compressed block is smaller than or equal to the length of an uncompressed block. Then, from Equation 9 it follows that for compressed ZFP blocks the number of bits required is given by:

$$I_{L,\text{ZFP}} = \left\lceil \log_2 \left((4^d \cdot p)/R \right) \right\rceil \leq 2d + \log_2(p) \tag{12}$$

Equation 12 shows that, contrary to offset encoding, the number of bits required to encode the length of a ZFP block does not scale with array size and has an upper bound based on the data precision and dimensionality. Therefore, instead of transferring block offsets, we propose to transfer block lengths and introduce a *preprocessing* step at the decoder. The preprocessing step computes the required block offsets through a cumulative sum of all the block lengths. Such a sum introduces a dependency between blocks, where the block offset of block n is dependent on the block lengths of the previous $n-1$ blocks. This cumulative sum is also known as Exclusive Scan or Prefix Sum [11]. There are work-efficient parallel algorithms to perform Prefix Sum in $\mathcal{O}(N)$ operations in $\mathcal{O}\left(\log_2(N)\right)$ parallel steps [14].

Similar to Strategy 1, we might create chunks of blocks and encode the length of a chunk rather than a block. This reduces the number of elements in the Prefix Sum by a factor C. However the number of bits needed to encode a chunk

length is larger than the number of bits required to encode a block length, as the maximum length of a chunk of C consecutive blocks in bits is C times larger than the length of a single block. This effect can be quantified by substituting L with $L \cdot C$ in Equation 9 to see that it increases the required number of metadata bits per chunk by $\log_2(C)$.

Figure 2b shows the chunk lengths that would be encoded on the example bitstream. Here, it can be seen that contrary to chunk offsets, the chunk lengths do not accumulate. Hence the number of bits needed to store them does not scale with the compressed data size.

If we compare Strategy 1 to Strategy 2, we see that Strategy 2 has an advantage in the sense that the required information is encoded more efficiently. Therefore the overhead size of Strategy 2 is smaller than for Strategy 1. However in Strategy 2 a preprocessing step is needed to reconstruct the chunk offsets. The main issue with this preprocessing step is that even in a work efficient parallel implementation, its duration scales with the number of blocks. In order to address the issues associated with both strategies, we propose a third *hybrid* strategy that combines the advantages of strategies 1 and 2 in order to minimize both size and computation overhead.

C. Strategy 3: Hybrid Encoding

The core concept of this strategy is to divide the decoding work over independent partitions. This means that the complexity of any preprocessing step is bounded by the partition size. We define a partition as follows:

Definition 5. (Partition) A **partition** is a set of P consecutive chunks, hence a partition contains $P \cdot C$ consecutive blocks.

For a partition of cardinality P, the first element is encoded as a single chunk offset and the remaining $P-1$ elements will be encoded as chunk lengths. Using an offset as first element decouples the partition from other partitions, which decouples the complexity of the preprocessing step from the array size. As a result, the preprocessing will consist of P small steps rather than a single large step. Figure 2c shows Strategy 3 applied to the example bitstream.

The chunk lengths within a partition are used to compute the chunk offsets using Prefix Sum. The overhead size for large partitions is comparable to that of Strategy 2 as for every P blocks we store $P-1$ chunk lengths and only a single chunk offset. Thus, the overhead per partition is defined by:

$$I_H = I_D + (P-1) \cdot I_L \tag{13}$$

D. Overhead Analysis and Implementation Considerations

To compare the overhead size of the different strategies we use Table I. The values for f are computed by substituting the expression of I in Equation 7 for every strategy. This means that these values are the maximum achievable f, as they assume a size-optimal implementation, meaning that the information is stored in the absolute minimum number of bits required. However, there are several practical considerations and hardware limitations that prevent us from using the minimum number of bits. First, most modern computer architectures work most efficiently with data types that are multiples of bytes. Therefore, we choose to implement schemes where lengths and offsets are encoded using standard data types. Second, in the case of ZFP, the number of bits required to encode chunk lengths in Strategy 2 and 3 is dependent on the dimensionality of the array. In order to make one general solution that is applicable to all modes, one would have to "over-size" the type used for storing the chunk lengths. Third, the number of bits used to encode chunk offsets places an upper bound on the compressed file size. In this work, we have a design requirement to support compressed files with sizes up to 1 TiB. Such a file size means that the chunk offset must be encoded with at least 43 bits. In order to fit a 43 bit value within standard data types, we choose to encode chunk offsets as 64 bit integers.

Table I: Overhead factor f for the metadata encoding strategies, where $Y = R/(C \cdot L)$

Strategy	f
1: Offsets	$Y \cdot \lceil \log_2(S/R) \rceil$
2: Lengths	$Y \cdot \lceil \log_2(C \cdot L) \rceil$
3: Hybrid	$(Y \cdot \lceil \log_2(S/R) \rceil + (P-1) \cdot \lceil \log_2(C \cdot L) \rceil)/P$

Table II: The setup used for evaluating the strategies

Item	Value
CPU	Intel Xeon Bronze 3106 (dual socket, 6 cores/socket)
GPU	NVIDIA Tesla V100 32 GiB (PCI-E Gen3 16 lanes)
RAM	256 GiB DDR4 2666 EEC
Storage	1.92 TiB SSD SATA disk
OS	CentOS 7.5.1804 kernel 3.10.0-957 (64 bit)
Compiler	GNU GCC 4.8.5
CUDA	CUDA version 10.0.130

V. Evaluation and Results

In this section, we present the results of evaluating our three proposed strategies in Section IV. We start by describing our experimental setup used for evaluation.

A. Experimental Setup

We evaluate the proposed strategies on both a CPU with an **OpenMP** implementation and a GPU with a **CUDA** implementation. We use a server whose configuration is outlined in Table II.

We implement the strategies on top of ZFP version 0.5.4 released on October 1, 2018 [10]. ZFP v0.5.4 is the first official ZFP release to support CUDA encoding and decoding. However, its CUDA support is limited to fixed rate. We first introduce the concept of chunks and then implement the calculation of the chunk offsets using metadata for variable rate modes. The metadata is encoded using the strategies as outlined in Section IV. ZFP release v0.5.4 does not support OpenMP decoding at all. However, it does support OpenMP encoding as well as serial decoding. We implement the

Table III: The used datasets. More extensive descriptions can be found in the referenced sources

Name	Description	Dimensions	Precision	Size (MiB)	R	Source
NYX	Cosmological hydrodynamics simulation	3D: 512x512x512 (6 fields)	Single	3072.0	4.94	[15]
ISABEL	Hurricane simulation	3D: 100x500x500 (13 fields)	Single	1239.8	3.22	[15]
CESM-ATM	Climate simulation	2D: 3600x1800 (79 fields)	Single	1952.9	5.60	[15]
BRAIN	Brain impact simulation	2D: 17730 x 1000	Double	135.3	7.45	[12]
BROWN	Synthetic Brown data	1D: 8388609	Double	64.0	5.75	[15]
PLASMA	Plasma temperature simulation	1D: 4386200	Single	16.7	1.88	[12]

Require: A benchmark to be evaluated
1: $\mathbf{R} = \emptyset$ {List of mean compression ratios}
2: $\mathbf{T} = \emptyset$ {List of mean execution times}
3: $\mathbf{I} = \emptyset$ {List of metadata sizes}
4: **for** $i = 0$ **to** 12 **do**
5: $C = 4^i$
6: $\Gamma = \emptyset$
7: $\Lambda = \emptyset$
8: **for** $j = 8$ **to** 24 (incremented by 4) **do**
9: Invoke ZFP fixed precision mode on the benchmark with precision j
10: Append the resulting compression ratio to Γ
11: Append the execution time in Λ
12: **end for**
13: Compute the harmonic mean of Γ and append it to $\mathbf{R}$
14: Compute the standard mean of Λ and append it to $\mathbf{T}$
15: Compute the metadata size I and append it to $\mathbf{I}$
16: **end for**
17: **return** Harmonic mean compression ratio, mean overhead, and mean execution time of the benchmark under every chunk granularity C

Figure 3: Evaluation methodology for the CPU implementation

parallel decoding by using the metadata encoding strategies to compute chunk offsets and from there launch threads that decode the chunks in parallel. We use static OpenMP scheduling. Our implementations can be found on [16].

We choose to use open source floating-point datasets used in research on compression algorithms in order to get representative results. They are listed in Table III. For all benchmarks on a CPU, we use the evaluation methodology outlined in Figure 3. The decoding throughput is defined as the uncompressed array size divided by the decoding time, so S/T_{decode}. In ZFP, the decoding throughput is highly dependent on array dimensionality, data type and compression ratio. We use multiple datasets which have different dimensionalities and data types. For every dataset we use 5 ZFP precision values and we compute the harmonic mean of the compression ratios and mean of the execution times. The rationale to average over these parameters is that we aim to analyze the impact of our proposed parallelization strategies on a high level, rather than the maximum ZFP performance using optimal settings. When optimizing for a specific dataset, we are able to achieve significantly higher decoding throughput. On a GPU we use the same methodology with $C = \{1, 2, 4, 8, 16\}$.

B. CPU Results

We start by evaluating the OpenMP implementation and showing the impact of chunk granularity C on the decoding throughput and size overhead. Then, we show the speedup obtained.

Figure 4 shows the decoding throughput of the OpenMP

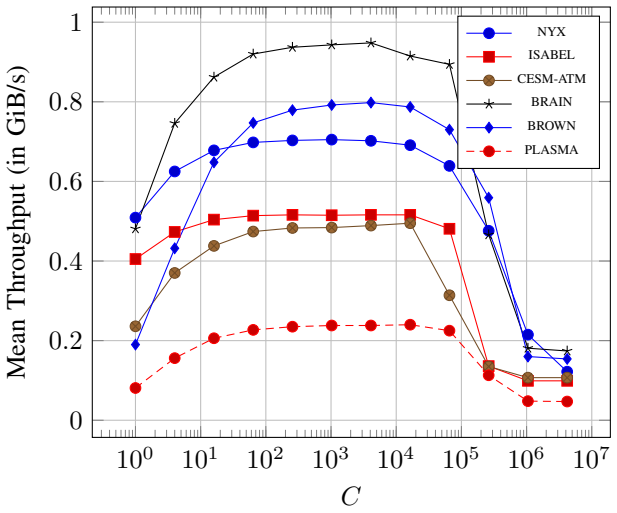


Figure 4: CPU decoding throughput, under Strategy 1, as a function of C

implementation as a function of C. We observe that as we increase C the decoding throughput increases until a certain threshold. Then, it is roughly constant up until we reach a next threshold, from where it start to degrade. Increasing the chunk granularity leads to more coarse-grain parallelism, as we parallelize per chunk of C blocks rather than per block. For large C, this leads to insufficient load balancing. For the considered benchmarks, one can conclude that a chunk granularity in the range $[10^3, 10^4]$ would correspond to near-optimal decoding throughput. The exact value is data and platform dependent, but is not of interest in this paper as our goal is to show the trend and design trade-offs to consider when using this approach.

Figure 5 shows the size overhead introduced by Strategy 1 as a function of C. When $f = 1$ the introduced overhead is as large as the compressed data itself. We observe that Strategy 1 suffers from very large size overhead when C is small. However, as the chunk granularity increases, the overhead drops dramatically and becomes less than 0.0004 for $C = 4096$.

Figure 6 shows the speedup achieved with the OpenMP implementation on a 6-core CPU for $C = 4096$. We see that the implementation has a near-optimal speedup, as it is nearly linear with the number of threads. We observe similar results when performing the same test on a Xeon Gold 6126 using up to 24 threads. If we combine the findings from

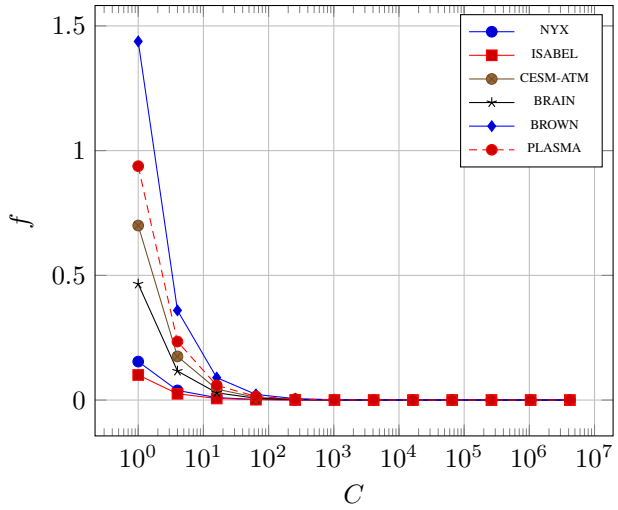


Figure 5: CPU overhead factor f, under Strategy 1, as a function of C

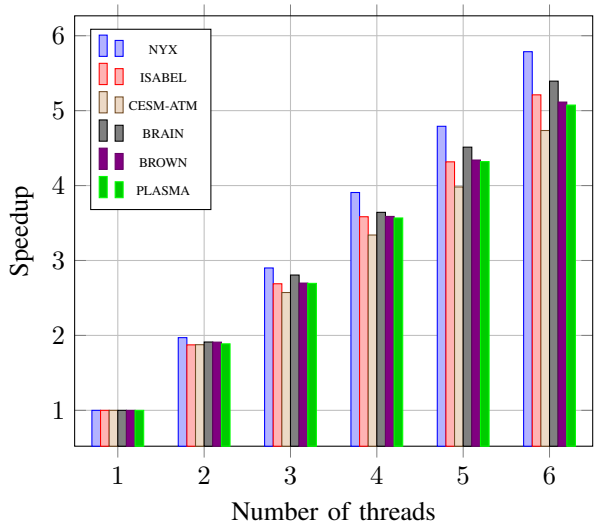


Figure 6: CPU decoding speedup, under Strategy 1, as a function of the number of threads with $C = 4096$

Figures 4, 5, and 6, then one can conclude that when C is large Strategy 1 is a very good choice for CPUs as it provides near-optimal speedup for negligible overhead.

Recall from Section IV that the advantage of strategies 2 and 3 is a reduction of the size of the metadata, at the cost of decoding throughput due to an added preprocessing step. These strategies are useful in cases where increasing the chunk granularity is not favorable. We observe that increasing the chunk granularity is feasible with our datasets and test setup and therefore we do not further investigate strategies 2 and 3 on a CPU.

C. GPU Results

On the GPU, we start by implementing Strategy 1 and investigating differences with the CPU implementation.

1) Strategy 1: Offsets: Figure 7 shows the decoding throughput of the CUDA implementation of Strategy 1. If

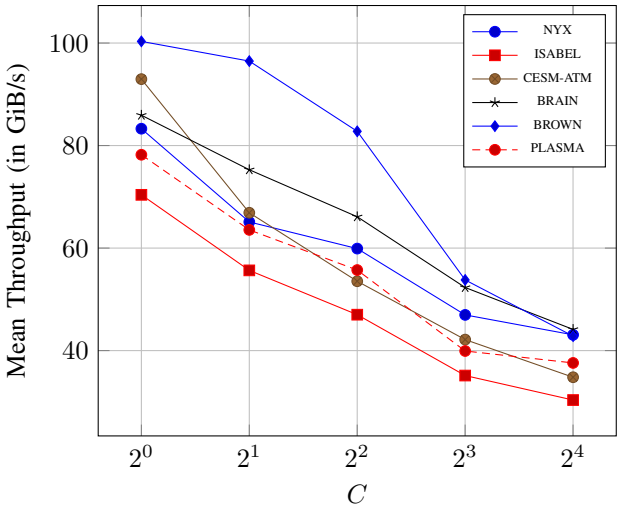


Figure 7: GPU decoding throughput, under Strategy 1, as a function of C

we compare Figure 7 to Figure 4 we observe that CUDA decoding throughput decreases if we increase C. This is in contrast to OpenMP, where increasing C would increase decoding throughput up to a threshold. The reasons for this decrease are the control loop and access granularity issues introduced in Section IV-A. Since a GPU has many more cores than a CPU we observe the load balancing issues at a lower chunk granularity compared to the CPU. Furthermore the introduction of a control loop has a larger impact on a GPU than on a CPU. This is due to the fact that after each loop iteration all threads in a warp need to synchronize. Finally the overhead associated with launching many threads is less on a GPU.

Table IV lists the overhead factor of the datasets for strategies 1 and 3 with a chunk granularity of 1. Due to the large decrease in decoding throughput, it is not feasible to increase C to the same extent as in the OpenMP implementation. Therefore, when comparing to the overhead for the OpenMP implementation (Figure 5), we see that the CUDA implementation suffers from a much larger overhead factor. We will implement strategies 2 and 3 in CUDA with the goal to reduce this overhead factor without significant loss of decoding throughput.

2) Strategy 2: Lengths: In order to evaluate Strategy 2 we used a work efficient parallel implementation of Prefix Sum as described in [14]. We found that for large arrays the time needed to perform the Prefix Sum is the dominant factor in the decoder execution time. As a result, in Strategy 2 decoding throughput is more than halved compared to Strategy 1. This is consistent with observations of other authors who also used Prefix Sum for parallel decompression on GPUs and reached the conclusion that it is a significant portion of their execution time [4]. Since our primary objective is maximizing the decoding throughput, we conclude that Strategy 2 is not promising for GPUs.

3) Strategy 3: Hybrid: A dominant factor in the Prefix Sum execution time for large arrays is data dependencies

Table IV: f for CUDA implementations of strategies 1 and 3 for various datasets with $C = 1$

Dataset	Strategy 1	Strategy 3
NYX	0.15	0.04
ISABEL	0.10	0.03
CESM-ATM	0.70	0.20
BRAIN	0.47	0.13
BROWN	1.44	0.40
PLASMA	0.94	0.26

between warps. In order to eliminate these inter-warp dependencies, we propose a partition size of 32, which is the warp size for modern NVIDIA GPUs [17]. This minimizes the preprocessing time needed per block, and also leads to a significantly lower overhead size compared to Strategy 1.

When we implement Strategy 3 with a partition size of 32, we observe that the execution time of the preprocessing step is minimal. The difference in decoding throughput compared to Strategy 1 is $< 3\%$ for every C and every dataset. If we consider the size overhead of the CUDA implementation of Strategy 3 as shown in Table IV, then we see the clear advantage of this strategy. Under Strategy 1, the overhead is 64 bits per chunk, while under Strategy 3 the overhead is $64 + 32 \cdot 16$ bits per partition, which is 18 bits per chunk. This is an overhead reduction with a factor of 3.55 for a loss in decoding throughput that is consistently lower than 3%. This shows clearly that Strategy 3 is well suited for GPUs. Reducing the overhead size further involves a trade-off as increasing the chunk granularity impacts the decoding throughput significantly. This trade-off is data-dependent in the case of ZFP and it requires consideration of the application requirements.

VI. Conclusions

In this paper, we presented an approach to parallelize variable rate decompression in a tiled compression algorithm. This approach is to transfer metadata, in addition to the compressed data, to allow parallelization. This metadata introduces a storage overhead, but also greatly increases the decoding throughput. We presented three strategies for generating and storing the metadata and showed that the optimal strategy depends on the hardware platform used. On a CPU, we are able to achieve a speedup in decoding throughput that is near linear with the number of cores, with a storage overhead of less than 0.04% of the compressed data size. On a GPU, we were able to achieve a decoding throughput of up to 100 GiB/s. However, the storage overhead in this case is almost 40% of the compressed data size. Strategy 1 (offsets) is an effective strategy in cases where it is possible to use a coarse chunk granularity. On a CPU this is often a good approach as this will also lead to a high decoding throughput. On a GPU this is often not feasible, as it will result in a low decoding throughput due to load balancing issues. Here, Strategy 3 (hybrid) is more efficient since it maps well to the HW, for example when using a partition size equal to the warp size on NVIDIA GPUs. Finding an optimal chunk granularity is dependent on the input data and application requirements. Our method can be applied to other tiled compression algorithms, since it decouples the parallelization strategy from the compression. This allows optimizing for different architectures and applications.

Acknowledgment

The authors would like to thank Matthew Larsen from Lawrence Livermore National Laboratory (LLNL) for his support with ZFP. We would also like to thank all of our reviewers. Furthermore we would like to thank the DOE NNSA ECP project and ECP CODAR project for providing the Scientific Data Reduction benchmarks. This work was partially performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344.

References

[1] J. Shalf, S. Dosanjh, and J. Morrison, "Exascale Computing Technology Challenges," in *Proc. of VECPAR*, J. M. L. M. Palma *et al.*, Eds. Springer Berlin Heidelberg, 2011, pp. 1–25.

[2] S. W. Keckler *et al.*, "GPUs and the Future of Parallel Computing," *IEEE Micro*, vol. 31, no. 5, pp. 7–17, 2011.

[3] S. Mittal and J. S. Vetter, "A Survey Of Architectural Approaches for Data Compression in Cache and Main Memory Systems," *IEEE Trans. Parallel Distrib. Syst.*, vol. 27, no. 5, pp. 1524–1536, 2016.

[4] A. Balevic, "Parallel Variable-Length Encoding on GPGPUs," in *Proc. of Euro-Par*. Springer Berlin Heidelberg, 2010, pp. 26–35.

[5] M. A. O'Neil and M. Burtscher, "Floating-point Data Compression at 75 Gb/s on a GPU," in *Proc. of GPGPU*. ACM, 2011, pp. 7:1–7:7.

[6] K. Zhu and J.-M. Kim, "Method and apparatus for generating JPEG files suitable for parallel decoding," Sep 2013, US Patent 8,538,180.

[7] H. Jang, C. Kim, and J. W. Lee, "Practical Speculative Parallelization of Variable-length Decompression Algorithms," in *Proc. of LCTES*. ACM, 2013, pp. 55–64.

[8] P. Lindstrom, "Fixed-Rate Compressed Floating-Point Arrays," *IEEE Trans. Vis. Comput. Graphics*, vol. 20, no. 12, pp. 2674–2683, 2014.

[9] (2019) zfp Documentation. Release 0.5.4. [Online]. Available: https://buildmedia.readthedocs.org/media/pdf/zfp/release0.5.4/zfp.pdf

[10] (2019) zfp v0.5.4. [Online]. Available: https://github.com/LLNL/zfp/

[11] G. E. Blelloch, "Prefix Sums and Their Applications," School of Computer Science, Carnegie Mellon University, Tech. Rep. CMU-CS-90-190, November 1990.

[12] M. Burtscher and P. Ratanaworabhan, "FPC: A High-Speed Compressor for Double-Precision Floating-Point Data," *IEEE Trans. Comput.*, vol. 58, no. 1, pp. 18–31, 2009.

[13] A. Moffat and L. Stuiver, "Binary Interpolative Coding for Effective Index Compression," *Inform. Retrieval*, vol. 3, no. 1, pp. 25–47, 2000.

[14] M. Harris, S. Sengupta, and J. D. Owens, "Parallel Prefix Sum (scan) with CUDA," *GPU gems*, vol. 3, no. 39, pp. 851–876, 2007.

[15] (2019) Scientific Data Reduction Benchmarks. Last accessed on: March 4, 2019. [Online]. Available: https://sdrbench.github.io/

[16] (2019) ZFP Parallel decompression. [Online]. Available: https://github.com/LennartNoordsij/zfp/tree/zfp_parallel

[17] *NVIDIA CUDA C Programming Guide*, NVIDIA Corporation, last accessed on: March 5, 2019, v10.0. [Online]. Available: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html