Received 19 January 2024, accepted 21 February 2024, date of publication 27 February 2024, date of current version 6 March 2024.

*Digital Object Identifier 10.1109/ACCESS.2024.3371253*

RESEARCH ARTICLE

# Hybrid-Smash: A Heterogeneous CPU-GPU Compression Library

**CRISTIAN PEÑARANDA**[1], **CARLOS REAÑO**[2], **(Member, IEEE), AND FEDERICO SILLA**[1]
[1]Departamento de Informática de Sistemas y Computadores (DISCA), Universitat Politècnica de València, 46022 Valencia, Spain
[2]Departament d'Informàtica, Escola Tècnica Superior d'Enginyeria (ETSE), Universitat de València, 46100 Burjassot, Spain

Corresponding author: Cristian Peñaranda (cripeace@gap.upv.es)

This work was supported by Spanish Ministry of Science and Innovation under Grant RTC2019-007159-5.

**ABSTRACT** Compression algorithms are widely used to reduce data size and improve application performance. Nevertheless, data compression has a computational cost which can limit its use. GPUs could be leveraged to reduce compression time. However, existing GPU-based compression libraries expect data to compress in GPU memory, although it is usually stored in CPU memory. Additionally, setup time of GPUs could be a problem when compressing small data sizes. In this paper, we implement a new GPU-based compression library. Contrary to existing ones, our library uses data located in CPU memory. Performance results show that, for the same compression algorithms, GPUs are beneficial for larger data sizes whereas smaller data sizes are compressed faster using CPUs. Therefore, we enhance our proposal with `Hybrid-Smash`: a heterogeneous CPU-GPU compression library, which transparently uses CPU or GPU compression depending on data size, thus improving compression for any data size.

**INDEX TERMS** Lossless compression, parallel computing, GPU.

## I. INTRODUCTION

Compression algorithms are widely used to reduce data size [1], [2], [3]. They also improve the performance and efficiency of various applications that handle large volumes of data. This is the case for data analysis, machine learning, visualization, image processing, encryption, pattern matching, video streaming, web browsing, cloud computing, backup and archiving, etc.

Lossy compression algorithms such as MPEG4 or H.264 may be used depending on the specific application domain [4], [5], [6]. These algorithms noticeably reduce data size at the expense of discarding information. However, this is not a problem in areas like video streaming, for instance, where the human eye cannot detect the missing data. Contrary to lossy compression algorithms, lossless compression reduces data size while making it possible to restore the original data. Lossless compression algorithms can be useful in archiving, for example. In this paper, we focus on lossless compression.

The associate editor coordinating the review of this manuscript and approving it for publication was Thomas Canhao Xu.

Data compression, however, also has a computational cost that can limit its use in real-time or resource-constrained scenarios. To address this concern, Graphics Processing Units (GPUs) could be leveraged instead of CPUs to compress and decompress data. Although CPU compression algorithms have been widely studied in the past, GPU-based compression algorithms are much more novel and still remain a challenge due to the GPU model itself, which imposes different problems compared to traditional CPU usage.

Recently, NVIDIA released the nvCOMP compression library [7], which implements several compression algorithms such as Deflate, Lz4, Zstd, or Snappy. These compression algorithms are not new, but they were previously available only for CPUs. Given that the nvCOMP library has been developed by NVIDIA, the implementation of these compression algorithms for GPUs is expected to be extremely efficient.

Using the nvCOMP library is quite simple. The data to compress (or decompress) must be located in the GPU memory. The resulting data will also be stored in the GPU memory. This simple model has some drawbacks. For instance, although the data to compress (or decompress) must be in the GPU memory, the original data is usually located in the memory of the host CPU. In a similar way, although the nvCOMP library stores resulting data in the GPU memory, this result is typically needed in the host CPU memory. Thus, to use the nvCOMP library, data must be moved from CPU memory to GPU memory and vice versa.

While moving data to and from the GPU memory is simple, efficiently integrating the use of the nvCOMP library with these data copies may not be trivial to implement if one of the design goals is to hide the latency of data copies. For example, having a low latency GPU compression mechanism could be very appealing to take advantage of data compression in real-time scenarios. This could be the case for on-the-fly data compression in communication systems, where data is transparently compressed before being sent to the network and then decompressed on receipt, before being delivered to the application on the receiver. Thus, introducing a pipelined design in that GPU compression solution may help to hide compression/decompression latency in real-time applications. Using the nvCOMP compression library in a high-performance pipeline solution is not trivial because many different options need to be investigated.

Moreover, contrary to what happens with CPUs, an important concern when integrating the nvCOMP library into an efficient pipelined compression solution is that GPUs have a much longer setup time. This could be a problem when smaller amounts of data must be processed. Therefore, there is also a need to investigate when GPUs are beneficial for data compression, and when the use of CPUs is preferred.

In this paper, we start by analyzing the nvCOMP compression library and then research how to create a hybrid CPU-GPU compression solution that completely hides the latency of data copies between the CPU and the GPU. More precisely, the main contributions of this paper are:

- We analyze the benefits that the nvCOMP library brings to data compression and decompression with respect to the usage of CPU compression. We do this by analyzing the same compression algorithms on both the CPU and the GPU.
- We implement a highly optimized compression solution based on the nvCOMP library. Contrary to the nvCOMP library, our solution uses data initially located in CPU memory and stores the resulting data in CPU memory. Our solution obtains results similar to the nvCOMP library and outperforms them when the data size is large enough. Although our solution moves data from host to device memory before compressing and from device to host memory after compression, these transfers are practically hidden by kernel executions.
- As a consequence of the performance analysis of the previous highly optimized GPU compression library, we lay the foundations for a future version. This new version is referred to as `Hybrid-Smash`, and consists of creating a heterogeneous CPU-GPU compression library. In this paper, we present an initial prototype of this new version, which transparently uses CPU or GPU compression depending on the size of the data to be compressed. This approach reduces overall latency for small, medium, and large datasets.

The rest of the paper is organized as follows. First, in Section II, we present some related work on compression libraries. Next, in Section III, we discuss our approach iteratively. We introduce experiments to confirm each decision we make until we arrive at the final version. Finally, Section IV concludes the paper.

## II. RELATED WORK

This section includes the background necessary to understand this work. First, we discuss about the different compression algorithms, the Smash abstraction library [8], and the compression libraries selected for this work. Next, we introduce the nvCOMP library [7], which implements some relevant lossless compression libraries.

### A. LOSSLESS COMPRESSION LIBRARIES

Although lossy compression libraries obtain acceptable results with a good compression ratio and compression/decompression speed [9], in this work we focus on lossless compression libraries. These libraries use different algorithms for compression. Three of the most popular algorithms used are:

- Lempel-Ziv algorithms [10], [11]. These algorithms are based on identifying repetitive patterns in the input data during compression. Once these patterns are identified, a table is created that assigns unique codes to each pattern found and their corresponding codes replace them in the table.
- Huffman coding [12]. This algorithm uses a coding table based on the frequency of symbols present in the input data. Symbols that appear more often are assigned shorter codes, while the less frequent ones are assigned longer codes. During compression, each symbol is replaced by its corresponding code in the coding table.
- Arithmetic coding [13]. This algorithm assigns a real number in the range $[0, 1)$ to each symbol in the input data corresponding to its probability based on its frequency. As the Huffman coding, a table is created where a symbol is replaced by its probability number in that range. Arithmetic coding is more complex than Huffman coding but can achieve higher compression rates.

All of these lossless compression algorithms replace frequent code patterns using smaller ones, but there is a big difference between Lempel-Ziv and both the Huffman and Arithmetic coding: the latter algorithms must process the entire input data before compressing it because their table is based on the frequency of symbols, contrary to the former algorithm, which can compress data as it reads it. As the process carried out by the Huffman and Arithmetic algorithms can be computationally expensive, some compression libraries, such as the Brotli compression library [14], use a static table or create a table with just part of the data. Notice that the table of Lempel-Ziv algorithm is built while the compression is carried out.

There are many lossless compression libraries, each using a different API. To avoid dealing with these differences, in this paper we have used the Smash abstraction library [8], [15] for CPU-based compression. It contains 41 compression libraries, allowing users to set specific parameters for each one in a simple way. In the experiments performed in this paper, we focus on the results obtained by the following four compression libraries:

- Snappy [16]. The main objective of this compression library is not a high compression ratio, but a high compression and decompression speed. It has been developed by Google and is based on LZ77 [10].
- Lz4 [17]. It is also based on the LZ77 algorithm and focuses on compression and decompressed speed.
- Zstandard (Zstd) [18]. This library is also based on LZ77 and combines a fast Finite State Entropy and Huffman coding. It has been developed by Meta.
- A library based on Deflate [19]. This library combines LZ77 and Huffman coding for compression.

We have used the above four compression libraries among the 41 libraries included in the Smash abstraction library because these four libraries are very popular and are also implemented in the GPU compression library used in this paper, which is described in the next section.

### *B. GPU LOSSLESS COMPRESSION LIBRARIES*

The high performance offered by GPUs motivates researchers to improve compression libraries by moving the computation to these devices. Patel et al. [20] introduced some techniques to compute different algorithms used by Bzip2 on the GPU, but their proposal was 2.78 times slower than the CPU one. Ozsoy and Swany [21] created a parallel version of LZSS for GPUs. This version divides the memory into different chunks and compresses each block independently. They showed how their GPU version overcame the compression time of the CPU sequential version by up to 18 times, and the CPU parallel version by up to 3 times. Chłopkowski and Walkowiak [22] developed a new library based on Deflate where CPU and GPU work together to outperform CPU results. These studies explored the possibility of increasing the number of GPUs to reduce the time spent searching for matches. This library is more than two times faster than the best compressor used in the experiments. Li et al. [23] implemented a parallel version of the C-DPCM algorithm on GPUs in order to improve the compression time for hyperspectral images. They explored different mechanisms in the GPU to reduce compression times compared to previous CPU implementations, such as using shared memory, multiple GPU streams, or multiple GPUs.

Recently, NVIDIA [24] has joined this trend and has developed a library called nvCOMP [7]. This library is a generic compression interface that facilitates the use of compression libraries on high-performance GPU applications thanks to their flexible API. The NVIDIA team has implemented eight different CPU compression libraries on the GPU and has included them in the nvCOMP library. That allows developers to use the one that best suits their needs.

## III. DEVELOPING AN EFFICIENT HETEROGENEOUS COMPRESSION SOLUTION

This section presents our approach iteratively to show how we have arrived at the final version of our heterogeneous compression solution. We discuss all the versions created during the development. In each iteration, we also introduce experiments to measure the performance and justify the decisions made. The different versions shown in this section are the following ones:

- "Version 0 (V0)" (Subsection III-A): an ideal implementation of our proposal. A direct application of the nvCOMP library, a GPU-based compression library. In this initial implementation, the data to be compressed/decompressed is located in GPU memory.
- "Version 1 (V1)" (Subsection III-B): a naive GPU-based solution where data are located in the host CPU memory.
- "Version 2 (V2)" (Subsection III-C): an implementation of a pipeline inside the compression and the decompression that overcomes the previous version.
- "Version 3 (V3)" (Subsection III-D): an enhancement of the previous version by leveraging several concurrent GPU streams within the pipeline.
- "Version 4 (V4)" (Subsection III-F): a reduction of compression and decompression times using multiple GPUs in parallel.
- "Version 5 (V5)" (Subsection III-G): final implementation of the compression solution. It automatically compresses and decompresses data on CPU or GPU depending on the knowledge gained with previous versions. We will refer to this version as "Hybrid compression" or "Hybrid-Smash".

The performance evaluation presented in the experiments in this section has been carried out using the compression libraries detailed in Section II. The experiments have been run over an AMD EPYC 7282 16-Core Processor with NVIDIA A100 GPUs. Regarding the datasets used for the experiments, Canterbury [25] and Silesia [26] Corpus are typically used to evaluate compression libraries. However, there is a growing popularity of artificial intelligence (AI) applications, especially those that use GPUs. In addition, we plan to apply the research presented in this paper to remote GPU virtualization systems in the future, specifically to rCUDA [27], [28]. These systems present a client-server architecture and allow the applications with GPU computing to be run on the client side by sending the information to be processed to the server where the GPU is physically installed. Once that information has been processed, the result is returned to the client from the server. For these reasons, we have decided to use an AI dataset [8]. This dataset has

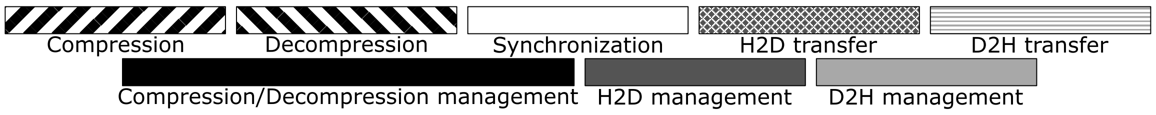

**FIGURE 1.** Legend for execution traces shown in Figures 2, 5, 7, 10, 11, 12, 13, 14, 19, and 20.

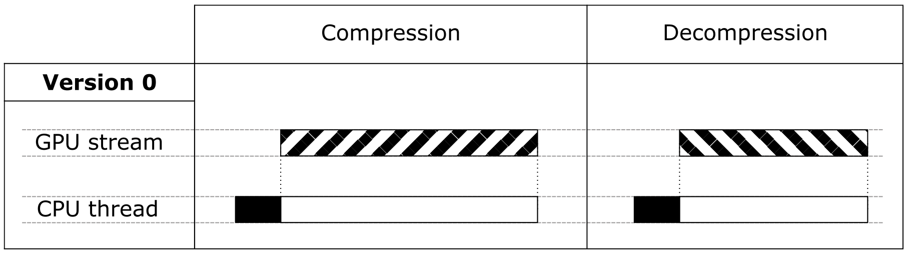

**FIGURE 2.** Illustrative execution trace for version 0 (V0). The legend is shown in Figure 1.

been created using the transfers made by rCUDA when four AI applications are executed, so it belongs to the domain we want to address. In addition, CPU-based compression libraries have shown that they work well with this type of data [29], artificially increasing the bandwidth when applied in the communications layer.

### A. VERSION 0 (V0): IDEAL GPU COMPRESSION

This initial version of our proposal is a straight-forward use of the nvCOMP library. Basically, we have created the API for our solution. This API will be used later in subsequent versions. It internally prepares and makes the required calls to the nvCOMP library.

Figure 2 presents an illustrative execution trace for this implementation (the legend for this figure is shown in Figure 1), so the time taken by actions is used to explain Version 0 and is not an authentic representation. As can be seen, the CPU thread first performs a "compression management" task before compressing data. This management task prepares the data and calls the GPU kernel. After that, the GPU compression kernel runs asynchronously to the CPU thread. Later, the CPU thread performs a synchronization to detect the completion of that GPU kernel. A similar execution flow is followed when decompressing data.

Using this Version 0, we have evaluated the performance of GPU compression libraries, i.e. the nvCOMP library. We have also evaluated the same compression libraries but running on the CPU. We have compared their compression ratio and compression/decompression speeds.

Figure 3 shows the compression ratio achieved by the compression libraries when running on the GPU and the CPU. Although CPU and GPU use the same compression algorithms, the compressed data obtained is not exactly the same. The reason is that CPU compresses the entire bunch of data, whereas GPU splits data into batches that are independently compressed. For that reason, the compression ratio obtained is different, but we can observe that they follow the same trend. Differences are more noticeable when the Lz4 compression library is used (Figure 3b) because the compression library is run over the CPU with a specific flag to reduce the compression ratio and increase the compression and decompression speed. We have used this flag because we prioritize speed in this paper.

Figure 4 shows the speed obtained by the CPU and the GPU compression libraries. The speed is defined in Equation 1 as the ratio between (i) the size of the original data and (ii) the time taken to compress the original data plus decompress the compressed data:

$$Speed = \frac{Size_{original}}{Time_{compression} + Time_{decompression}} \tag{1}$$

As we can see, for small data sizes, all CPU compression libraries perform better than GPU ones. GPU setup time penalizes GPU speed for these data sizes. However, as data size increases, GPU setup time is compensated by the shorter computing time, so GPU outperforms CPU for larger data sizes. Specifically, the GPU starts to outperform the CPU at $100KB$ using the Deflate compression library, at $1000KB$ using Snappy and Zstd, and at $10MB$ using the Lz4. On average, the CPU implementation of Lz4 is the fastest one.

Experiments in this section have been carried out with data stored in device memory for the GPU compression libraries, whereas data was stored in host memory for the CPU ones. The reason is because CPU libraries were initially developed to work with host memory. Note that if we want to compare GPU and CPU libraries, having the data to be compressed initially located in different places makes a significant difference. This is because the time taken to move the data is not considered, and it would be an unfair comparison. Thus, in the next sections we compare GPU and CPU libraries using data located in the host memory in both cases. Notice also that typically data would be expected to be in host CPU memory (after reading it from disk, for instance). Therefore, GPU compression libraries should copy data from the host CPU memory to the device GPU memory

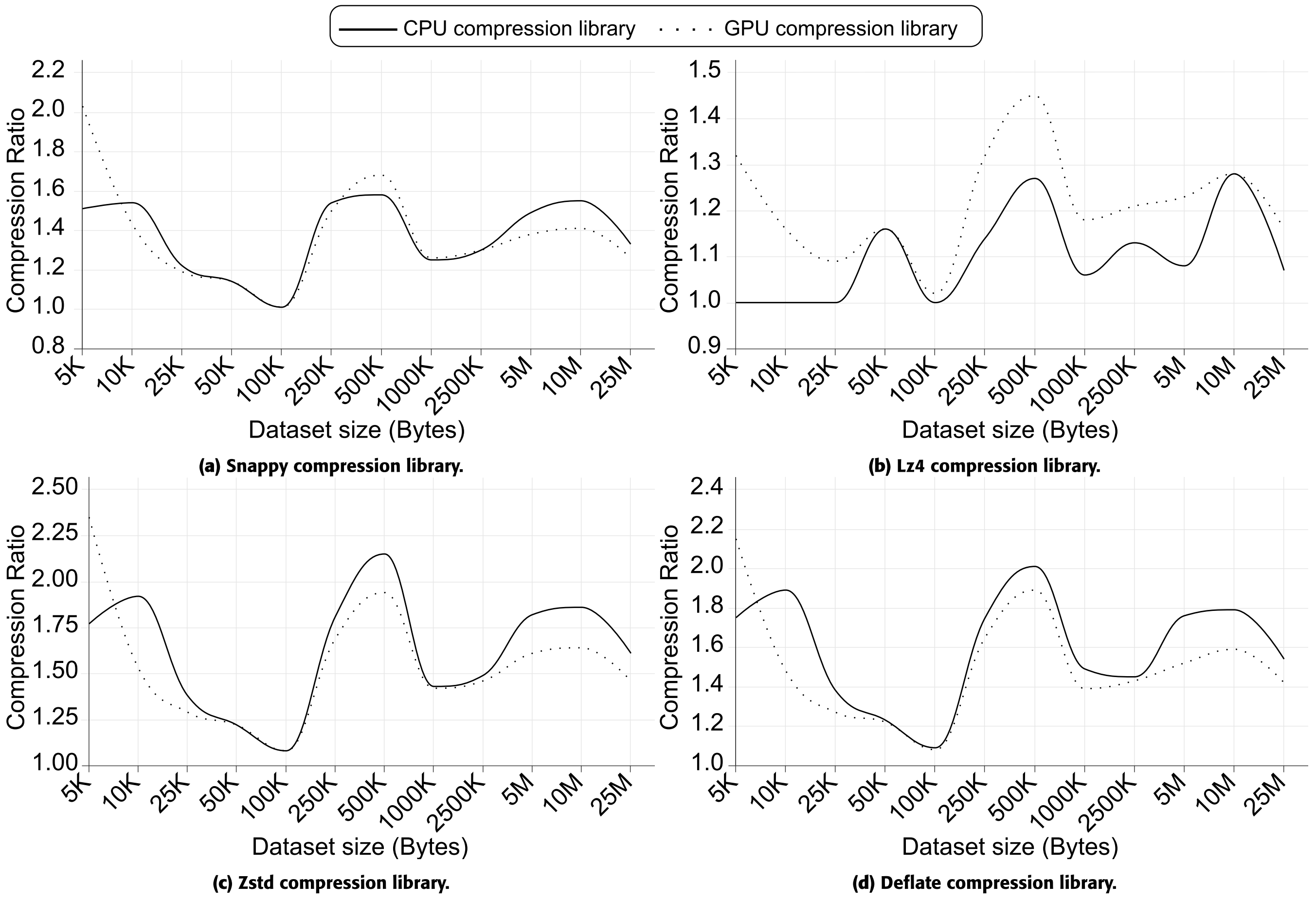

**FIGURE 3.** **Comparison of the compression ratio achieved by the CPU compression library, and the GPU version V0 (ideal).**

(H2D copy) prior to starting compression or decompression. In addition, after completing the GPU kernel, the result should be copied back from the device GPU memory to the host CPU memory (D2H copy). NVIDIA provides mechanisms to efficiently transfer data from host/device to host/device, however, the nvCOMP library assumes that data to be compressed is already in device memory. Thus, we call ideal GPU compression to this Version 0 (V0) because data is directly located in device memory, and no transfers to/from host memory are done. In the next section, we present Version 1 (V1) implementation, where we enable the possibility of using the GPU compression libraries from data located in host memory.

In future figures, V0 results are shown to evaluate the room for improvement of new GPU compression implementations. As commented, V0 are ideal results where data is located in device memory. Thus, it will be unfair to compare them with results where data is initially located in host memory and transfers to/from device memory are required.

### *B. VERSION 1 (V1): NAIVE GPU COMPRESSION*

Previous Version 0 (V0) compresses and decompresses data located in the device GPU memory. On the contrary, this Version 1 (V1) takes data from host CPU memory, similarly to CPU implementations. Therefore, the performance comparisons in this section are fairer and closer to reality than in the previous section.

Figure 5 shows an illustrative execution trace for V1. Again, the time taken by actions is used to explain Version 1 and is not an authentic representation. As we can see, in this version the CPU thread performs an asynchronous H2D memory copy before running the GPU kernel. Once the kernel is completed, the CPU thread performs a D2H copy. However, notice that there is an important difference between compression and decompression. When compressing, the final size of the compressed data depends on the internals of the compression algorithm. Therefore, the CPU thread must wait (i.e. first "Synchronization" part shown in the trace) until the compression kernel is completed to know this size. At that point, it can set the asynchronous D2H copy. On the other hand, the original size is stored in the compressed data header and used later when decompressing. For that reason, the final size of the decompressed data is known in advance when decompressing. Thus, the D2H copy can be configured immediately after configuring the decompression GPU kernel.

Figure 6 shows a comparison of the compression and decompression speed achieved by the CPU compression

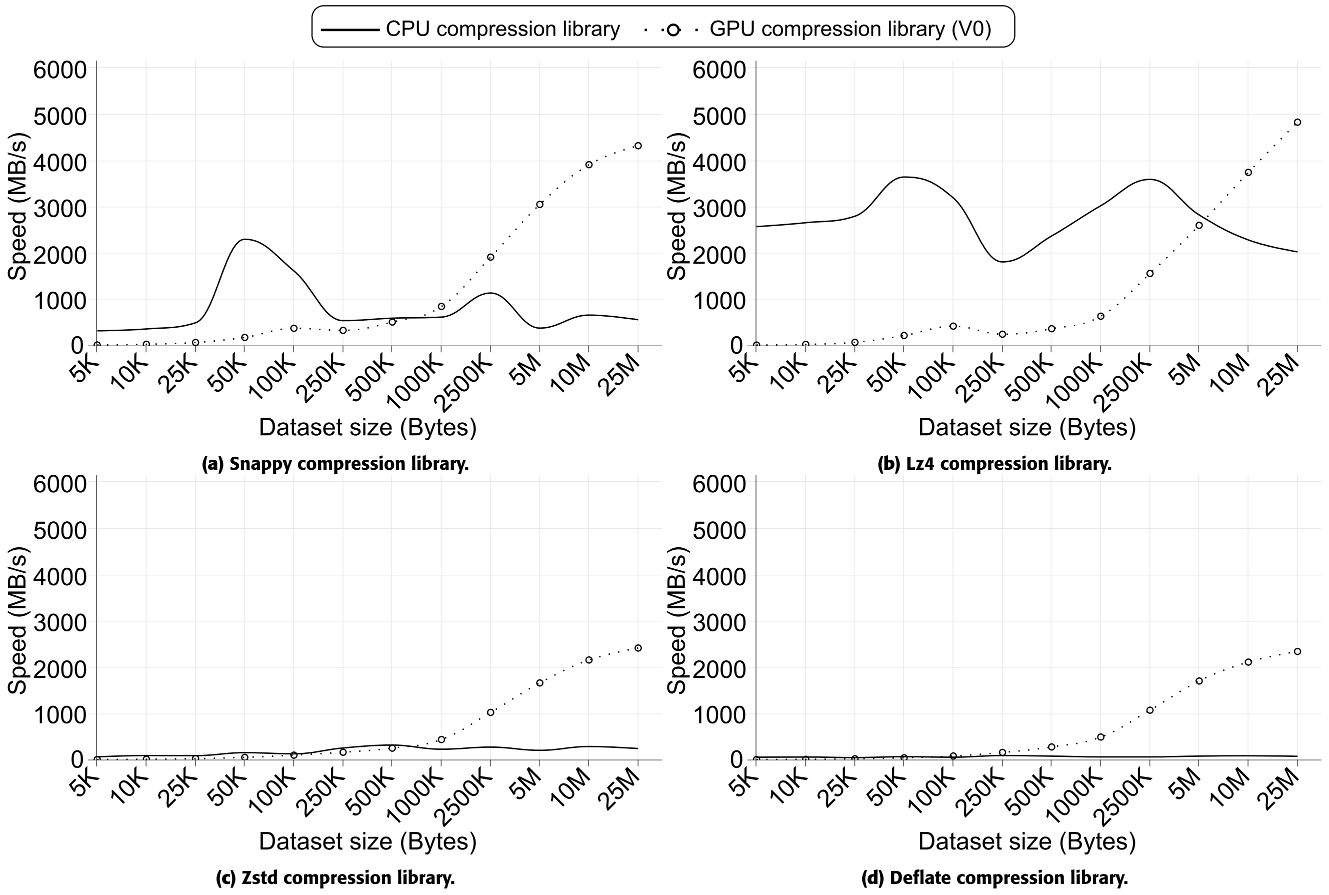

**FIGURE 4.** **Comparison of the compression and decompression speed achieved by the CPU compression library and the GPU version V0 (ideal).**

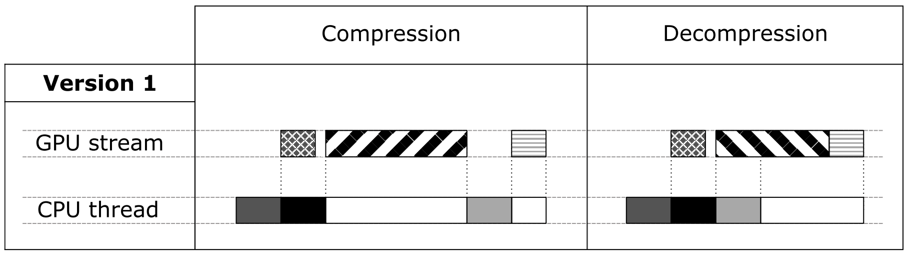

**FIGURE 5.** **Illustrative execution trace for version 1 (V1). The legend is shown in Figure 1.**

libraries, the previous V0 implementation, and the new V1. As we can see, the new V1 implementation obtains worse results than the ideal V0. That was expected because V1 transfers all data from host to device memory before running kernels, and from device to host memory after kernels are completed. Despite that, the V1 implementation of Deflate, Zstd, and Snappy still outperforms CPU for datasets larger than 250*KB*, 1000*KB*, and 5*MB*, respectively. In the case of Lz4, V1 achieves similar performance for 25*MB* datasets. Snappy and Lz4 are known to be fast. For this reason, V1 is less beneficial in those cases.

## C. VERSION 2 (V2): GPU COMPRESSION PIPELINE

The V1 implementation has a naive behavior: (i) it copies the data to compress/decompress from the host memory to the device memory, (ii) compresses (or decompresses) it using that device memory, and (iii) copies the result back from the device memory to the host memory. However, GPUs are powerful devices that allow data transfers to overlap with kernel executions. Therefore, it is possible to follow a pipelined approach: (i) copy data from host CPU memory to device GPU memory, (ii) at the same time run the kernel in the GPU for compressing (or decompressing) data, and (iii) in parallel, copy back compressed data from device GPU memory to host CPU memory. With such an approach, data copies may be hidden by kernel executions. Notice, however, that implementing this idea increases overall complexity.

### 1) MOVING FROM A NAIVE TO A PIPELINED IMPLEMENTATION

Figure 7 shows the iterations followed to go from the naive V1 to the pipelined V2 implementation. It shows illustrative

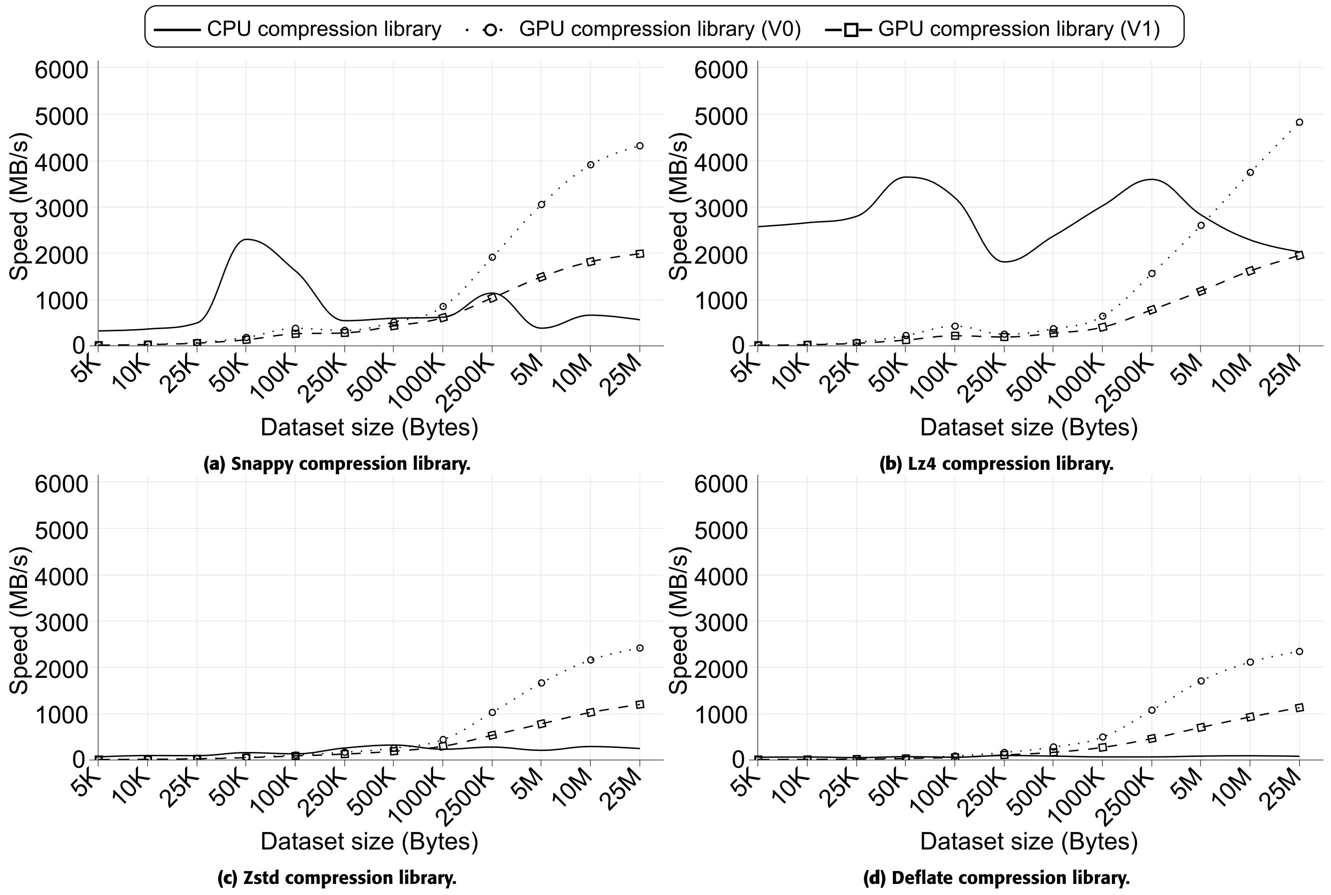

**FIGURE 6. Comparison of compression and decompression speed achieved by the CPU compression library, and the GPU versions V0 (ideal) and V1 (naive).**

execution traces where the time taken by actions is used to explain the different versions and is not an authentic representation. At the top of the figure, we show V1 again for completeness. Next, we show an improved version of V1 ("Improved Version 1"). This version splits the data to compress/decompress into chunks. Then, the CPU thread schedules one H2D copy per chunk. Next, the CPU thread launches one GPU compression kernel per chunk. Notice that the compression kernels are executed sequentially after the H2D copies end. After each kernel completion, the CPU thread configures the D2H copy of the compressed chunk. Again, as the D2H data copies are scheduled on the GPU stream after launching all the kernels, D2H data copies will not begin until the last kernel is completed. Thus, data copies and kernel executions do not overlap yet.

As commented, remember that the size of the compressed chunk is only known once the kernel finishes. So, the CPU thread synchronizes each kernel before calling its specific D2H copy. Furthermore, at the end of the process, the resulting compressed data must include all compressed chunks properly ordered. As a result, if the compressed data is stored in a single contiguous host memory region (which is the usual case), it is not possible to know the exact location of a compressed chunk until the previous chunk has been compressed.

"Early Version 2" in Figure 7 shows a first attempt to overlap data copies with kernel executions. In this implementation, two additional streams are created in the GPU: the H2D stream and the D2H stream. These two streams perform the H2D and D2H data copies, respectively. However, there is still a problem: the D2H data copy cannot be done until the kernel completes the compression because the size of the compressed data is only known once the kernel ends. Thus, the CPU thread has to wait for kernel completion before proceeding with the next chunk of data to compress. Despite using several streams in the GPU, data copies and kernel executions do not overlap.

"Version 2" in Figure 7 finally achieves the desired overlap. The solution to the synchronization problem mentioned above has been addressed by initially launching the compression of two chunks (this includes the associated H2D data copies and kernel executions). Once the compression of the first two chunks is launched by the CPU thread, it will wait for the result of the compression of the first chunk. At that point, the D2H data copy for that chunk is scheduled. This data copy overlaps with the compression of the second chunk.

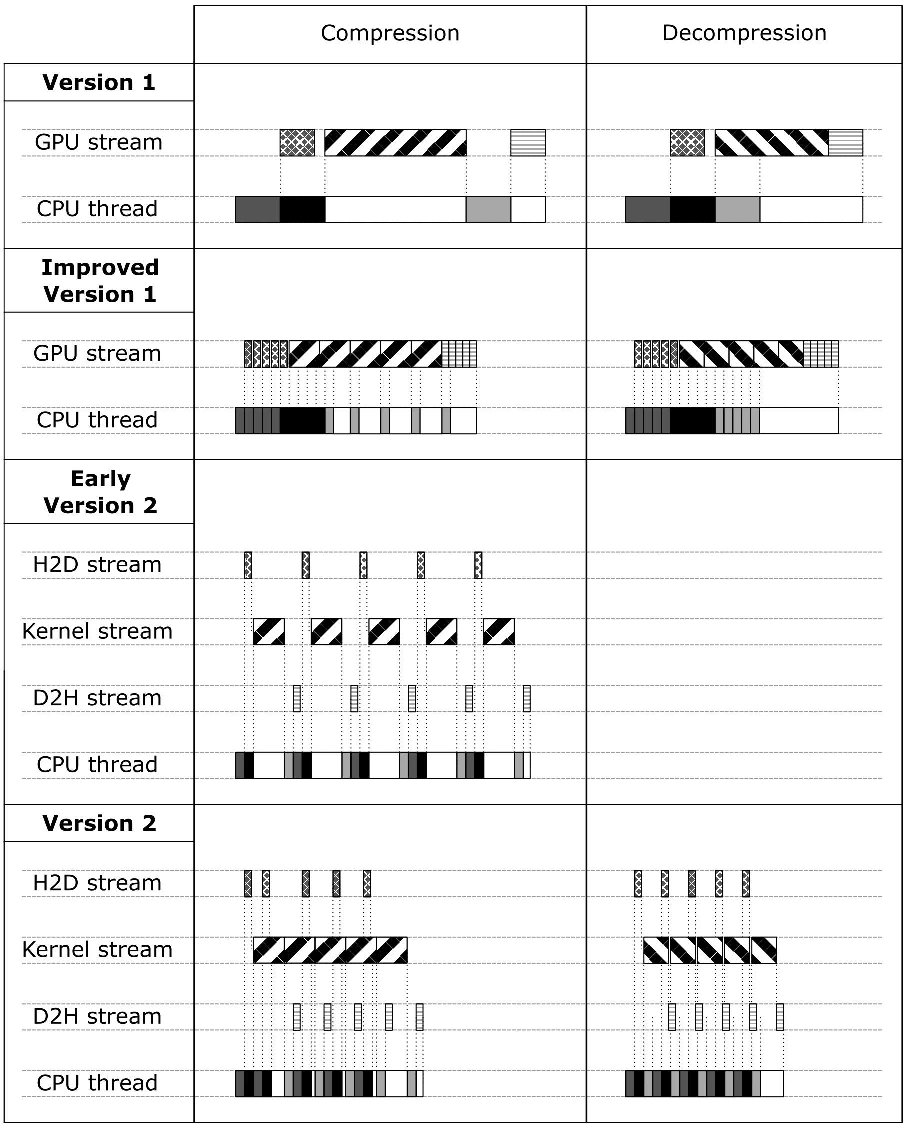

**FIGURE 7.** **Illustrative execution traces for GPU compression versions V1 (naive), V2 (pipeline) and intermediate versions. The legend is shown in Figure 1.**

Also at that very moment, the compression of the third chunk is launched (H2D copy and kernel execution). In this way, (1) the CPU thread will synchronize and perform the D2H copy for chunk $i - 1$, (2) while chunk $i$ is being compressed, (3) at the same time that the H2D copy and kernel execution for chunk $i + 1$ are launched. This allows us to effectively create the pipeline and completely overlap data copies with kernel executions. Notice that GPU streams must be properly synchronized using GPU events to avoid race conditions.

In the case of decompression, the evolution followed is also presented in Figure 7. Compression and decompression are similar, although the latter does not need to synchronize after kernel completions, as explained in the previous section, because compressed data size is known in advance. This means that the CPU thread does not need to stop sending operations to the GPU stream in "Improved Version 1". Also, given that the kernel synchronization done by the CPU thread is not required, "Early Version 2" and "Version 2 (V2)" are actually the same. For that reason, "Early Version 2" has been omitted for decompression.

### 2) OPTIMAL CHUNK SIZE FOR THE PIPELINE

As commented, the V2 pipeline is based on splitting the data to be compressed into chunks. Thus, it is necessary to select the optimal chunk size. On the one hand, larger chunks are transferred to the GPU slower than smaller ones. On the other hand, kernels compress and decompress chunks using a considerable amount of threads within the GPU. Therefore, larger chunks could benefit more from GPU computing power than smaller ones. Consequently, finding the best chunk size is key to optimizing the performance of the parallel pipeline in V2.

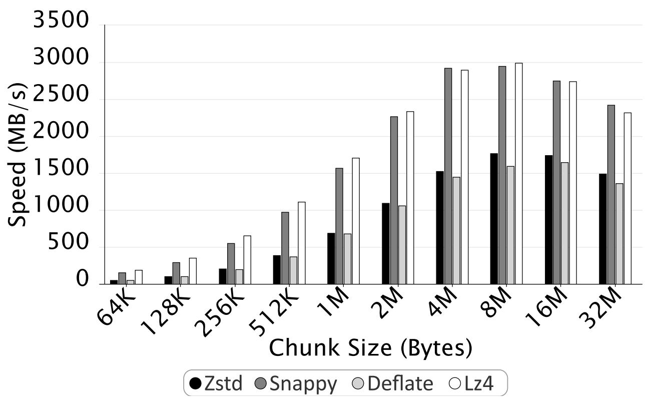

**FIGURE 8. Exploration to find the optimal chunk size for V2 pipeline using a 25MB dataset.**

In the experiments for finding the optimal chunk size, we have used the 25MB dataset because it is large enough to appreciate the differences in the performance of the pipeline. Figure 8 shows the experiments performed to find the optimal chunk size. The figure shows the speed of each GPU compression library when compressing and decompressing the dataset using V2. We have considered chunk sizes from 64$KB$ to 32$MB$. As we can see, all GPU compression libraries start with a low speed when using the smallest chunk size. The speed increases as chunk size increases. Zstd, Snappy, and Lz4 compression libraries achieve a performance peak with a chunk of 8$MB$, while the Deflate achieves its best speed with 16$MB$. For sizes larger than those, the speed starts to decrease. Notice that using a chunk size equal to 8$MB$ or 16$MB$ means that the pipeline will only be used for data sizes larger than these chunk sizes. Thus, for compressing data smaller than 8$MB$ or 16$MB$, the data will not be split into chunks, and the pipeline will not be fully leveraged. Nevertheless, it will still benefit from the space pre-allocated for chunks in GPU memory, avoiding allocating GPU memory before each compression/decompression and freeing it afterward.

#### 3) PERFORMANCE OF THE V2 PIPELINED COMPRESSION

Figure 9 compares the speed of V2 with previous versions. As we can see, regardless of the compression library used, V2 performs better than V1 and is closer to the ideal V0. Specifically, the V2 implementation of Snappy, Lz4, Zstd and Deflate outperforms V1 for datasets larger than 1000$KB$, 1000$KB$, 2500$KB$ and 2500$KB$ respectively. Notice that this also happens for data smaller than chunk size (16$MB$ for Deflate and 8$MB$ for Snappy, Lz4, and Zstd). As explained before, this is thanks to the space pre-allocated for chunks in GPU memory.

When comparing V2 with the CPU implementation, the V2 implementations of Snappy, Lz4, Zstd and Deflate perform better than CPU ones for datasets larger than 1000$KB$, 25$MB$, 1000$KB$, and 1000$KB$, respectively. Again, the CPU implementation of Lz4 is the one performing better on average.

To conclude the analysis of V2, we have used the NVIDIA Nsight Systems profiler [30] to get the execution traces of Snappy V2 when compressing (Figure 10) and decompressing (Figure 11) a 25MB dataset using a chunk size of 8$MB$. As we can see, kernel executions consume almost all the compression execution time and overlap completely with H2D and D2H copies. This confirms that data copies are hidden by kernel computations. We can also observe that decompression kernels are faster than data copies. Thus, the entire decompression process (i.e. H2Ds + kernels + D2Hs) takes about four times less than compression.

### D. VERSION 3 (V3): MULTI-STREAM GPU COMPRESSION

In V2 we divided specific tasks into three different streams to create the parallel pipeline: (i) one stream to perform H2D copies, (ii) another stream to execute kernels, and (iii) a third stream to perform D2H copies. In V3, we replicate these three streams to create multiple concurrent pipelines to further improve the performance of V2. Thus, data to compress or decompress is split into several pieces and each of the pieces is managed by one of the concurrent pipelines. However, this approach raises concerns about concurrency. If we create several concurrent pipelines at the beginning of the compression, all H2D copies will occur approximately at the same time. The same will happen with kernel executions and with D2H copies. This raises the concern about whether several H2D (or several D2H) data copies can be performed at the same time. Similarly, concurrent kernel execution is another concern.

**Algorithm 1** Synthetic Test to Check if There Are Any Limitations on the Number and Kind (H2D or D2H) of Data Copies That Can Be Overlapped Using Multiple Streams

```
$number\_of\_streams \leftarrow 20$;
$data\_size \leftarrow (32 * 1024 * 1024)$; // Each stream will
    copy 32MB
$cuda\_streams[number\_of\_streams]$;
$device\_memories[number\_of\_streams]$;
// Allocate pinned host memory using the
    cudaHostAlloc function
$host\_source\_memories[number\_of\_streams]$;
$host\_destination\_memories[number\_of\_streams]$;
// Launch all the H2D copies simultaneously
for $i \leftarrow 1$ to $number\_of\_streams$ do
    $cudaMemcpyAsync(device\_memories[i],$
      $host\_source\_memories[i], data\_size,$
      $cudaMemcpyHostToDevice, streams[i])$;
end
$cudaDeviceSynchronize()$; // Wait for completion of
    all copies
// Launch all the D2H copies simultaneously
for $i \leftarrow 1$ to $number\_of\_streams$ do
    $cudaMemcpyAsync(host\_destination\_memories[i],$
      $device\_memories[i], data\_size, cudaMemcpyDeviceToHost,$
      $streams[i])$;
end
$cudaDeviceSynchronize()$; // Wait for completion of
    all copies
```

Regarding the concern about concurrent data copies, we have observed in the previous section that H2D and D2H copies can overlap. However, there may be a limitation on

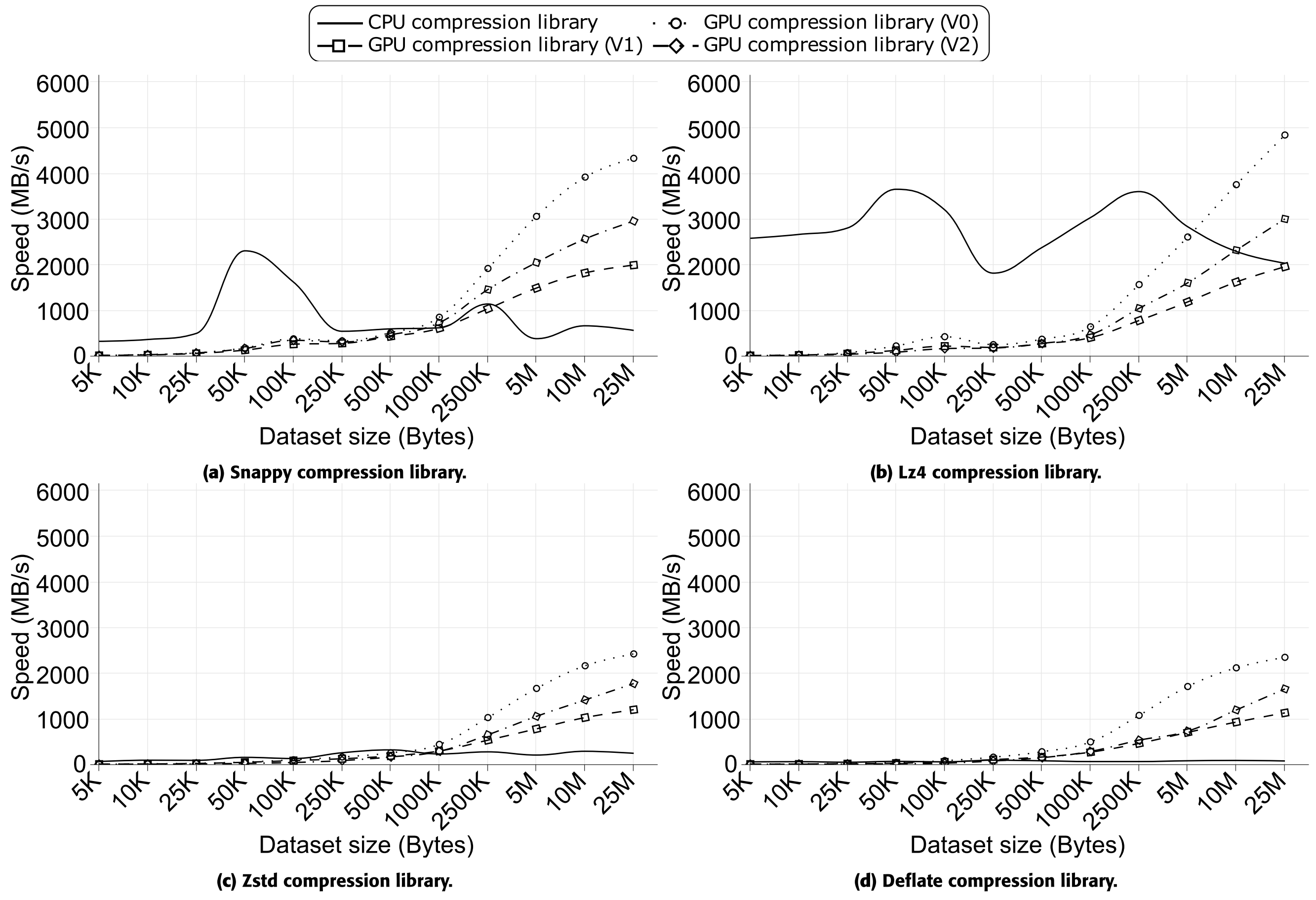

**FIGURE 9.** Comparison of the compression and decompression speed achieved by the CPU compression library, and the GPU versions V0 (ideal), V1 (naive) and V2 (pipeline).

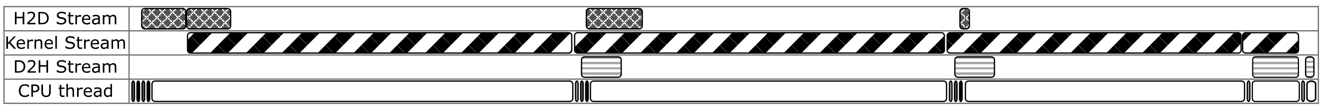

**FIGURE 10.** Execution trace of Snappy V2 (pipeline) when compressing a 25*MB* dataset using a chunk size of 8*MB*. This trace was obtained using the NVIDIA Nsight Systems profiler. The legend is shown in Figure 1.

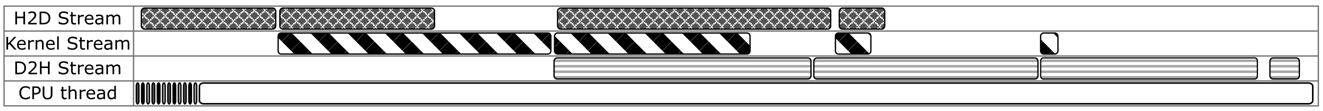

**FIGURE 11.** Execution trace of Snappy V2 (pipeline) when decompressing a compressed 25*MB* dataset using a chunk size of 8*MB*. This trace was obtained using the NVIDIA Nsight Systems profiler. The legend is shown in Figure 1.

the number and kind (H2D or D2H) of data copies that can be overlapped when using multiple streams. To analyze this, we have created the synthetic test shown in Algorithm 1. It first creates twenty different GPU streams. Then, one asynchronous H2D transfer is performed by each stream. Next, all streams are synchronized and, finally, one asynchronous D2H transfer is scheduled in each stream.

Figure 12 presents the execution trace of the synthetic test obtained using the NVIDIA Nsight Systems profiler. Although the streams work independently with different host and device memory ranges, data copies of the same kind (H2D or D2H) do not overlap. Therefore, replicating the data copy streams (H2D and D2H) used in V2 to multiple streams will not provide any improvement.

Regarding the concern about concurrent kernel execution, in theory, current GPUs support that. Moreover, kernels often do not take full advantage of the huge computing power of modern GPUs. Thus, launching multiple concurrent kernels may increase GPU utilization at the same time that overall kernel execution time is reduced. Furthermore, in V2

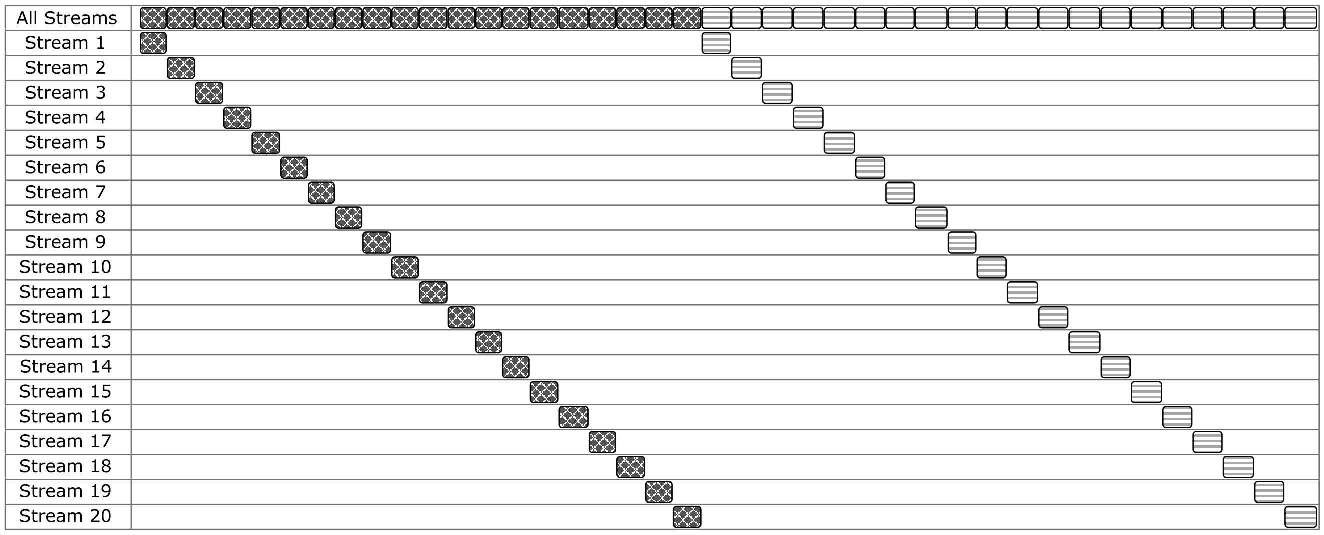

**FIGURE 12.** Execution trace of the synthetic test in Algorithm 1 to check if data copies overlap when using multiple streams. This trace was obtained using the NVIDIA Nsight Systems profiler. The legend is shown in Figure 1.

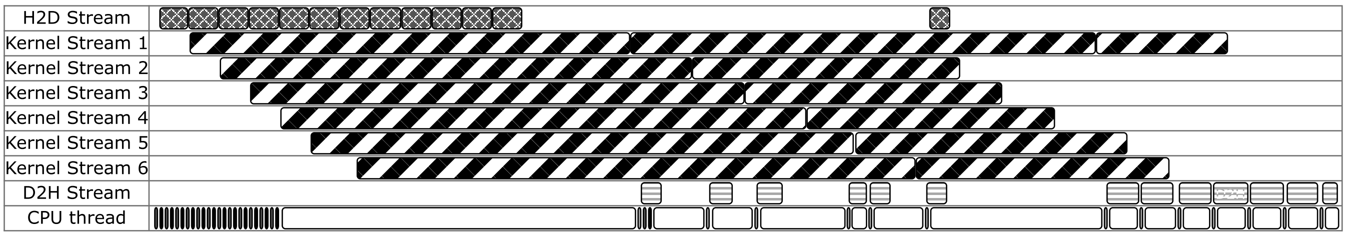

**FIGURE 13.** Execution trace of Snappy V3 (multi-stream) when compressing a 25*MB* dataset, using a chunk size of 2*MB* and up to six concurrent kernels. This trace was obtained using the NVIDIA Nsight Systems profiler. The legend is shown in Figure 1.

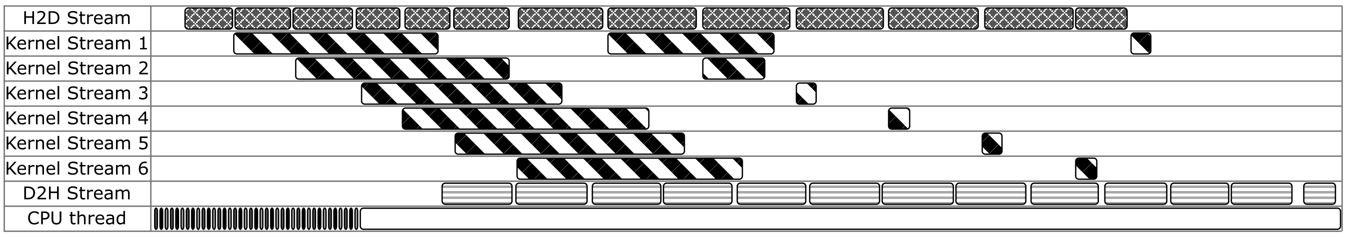

**FIGURE 14.** Execution trace of Snappy V3 (multi-stream) when decompressing a 25*MB* dataset, using a chunk size of 2*MB* and up to six concurrent kernels. This trace was obtained using the NVIDIA Nsight Systems profiler. The legend is shown in Figure 1.

experiments we found that the kernel execution time for compressing a data chunk takes longer than moving the chunk from/to CPU memory to/from GPU memory. Therefore, it seems a good approach to launch several compression (or decompression) kernels in parallel.

To analyze the effect of executing concurrent kernels, we have enhanced V2 implementation by creating multiple GPU streams for launching multiple kernels. Thus, the new parallel pipeline in "Version 3 (V3)" has the following stages: (i) one stream to perform H2D copies, (ii) multiple "kernel streams" (streams to execute multiple kernels that will run in parallel), and (iii) one stream to perform D2H copies.

Figure 13 shows the execution trace of this enhanced pipeline when executing the Snappy for compressing a 25*MB* dataset, using a chunk size of 2*MB*. Six GPU streams were created to launch up to 6 compression kernels in parallel. As we can see, the compression kernels from different streams overlap. As a result, the total kernel execution is reduced, which also reduces overall compression time.

Figure 14 shows a similar trace when the Snappy library is used for decompression. Similarly to compression kernels, decompression kernels also overlap. However, the improvement is not as noticeable as when compressing because kernel execution time is shorter.

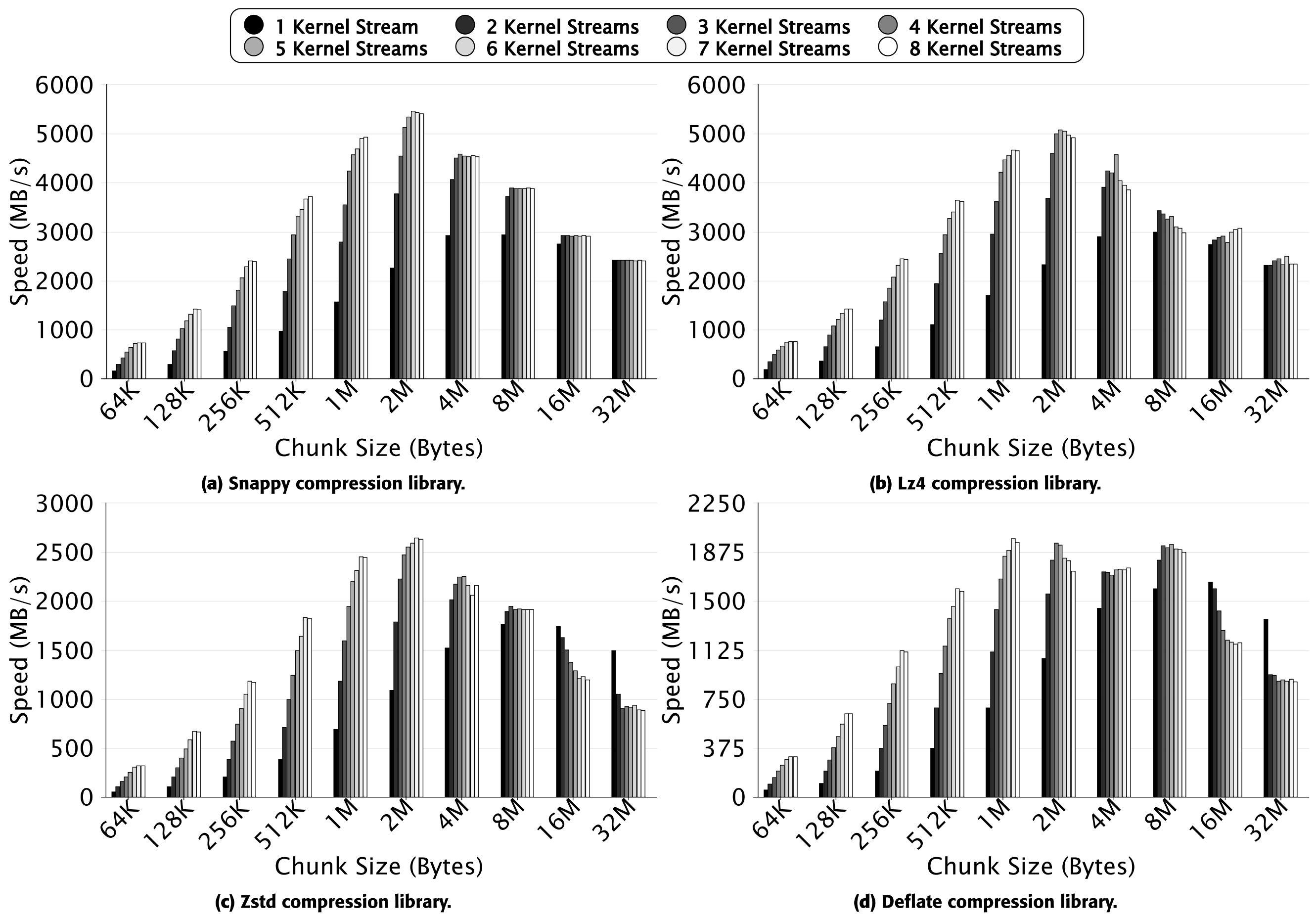

**FIGURE 15.** Exploration to find the optimal chunk size for V3 pipeline using a 25MB dataset and varying the number of kernel streams.

Figure 15 shows the speed achieved by GPU V3 compression libraries when varying the number of kernel streams (streams to execute kernels that will run in parallel). The number of streams varies from one, the darkest bar in the figure, to eight, the lightest bar in the figure (i.e. the lighter the color, the more kernels running concurrently). We also use different chunk sizes to find the optimal chunk size.

Figure 15a shows the results for the Snappy compression library. As we can see, the speed increases with the number of streams until it converges. The highest speed is achieved using six kernel streams and a chunk size of $2MB$. When using Lz4, see Figure 15b, the best result is obtained with five kernel streams and a chunk size of $2MB$. Zstd (Figure 15c) gets the best performance using seven streams and a chunk size of $2MB$. Deflate (Figure 15d) achieves the highest speed using seven streams and a chunk size of $1MB$. Compared to the optimal chunk size obtained for V2, which only used one kernel stream, the optimal chunk size for V3 has been reduced from $8MB$ to $2MB$ for Snappy, Lz4, and Zstd, and from $16MB$ to $1MB$ for Deflate.

Figure 16 shows the speed results obtained using V3. As expected, this new version overcomes previous versions V1 and V2. In the case of Snappy, Lz4, and Zstd, the new implementation outperforms even the ideal V0 for some data sizes. For Deflate, V3 results are closer to the ideal V0 ones than the results of previous versions. Even though the idea was not to improve the results obtained by the ideal version, V3 does it because it runs multiple concurrent kernels and takes full advantage of the computing power of the GPU. However, for small datasets, the CPU compression is still faster. GPU compression is beneficial for datasets larger than $1000KB$, $10MB$, $1000KB$, and $500KB$ when using Snappy, Lz4, Zstd, and Deflate compression libraries respectively. These results indicate that compression and decompression must be done on the CPU or the GPU depending on (i) the data size to compress and (ii) the compression library used.

## E. COMPARING COMPRESSION AND DECOMPRESSION SPEED OF IMPLEMENTED GPU VERSIONS

In the previous sections, we have compared the total speed of compression libraries when compressing and decompressing specific datasets using the GPU. In this section, we deepen the study by analyzing how the different versions behave in compression and decompression separately.

Figure 17 shows the compression time in milliseconds (ms) achieved by compression libraries using the different GPU versions already presented. We can see in the figure that as

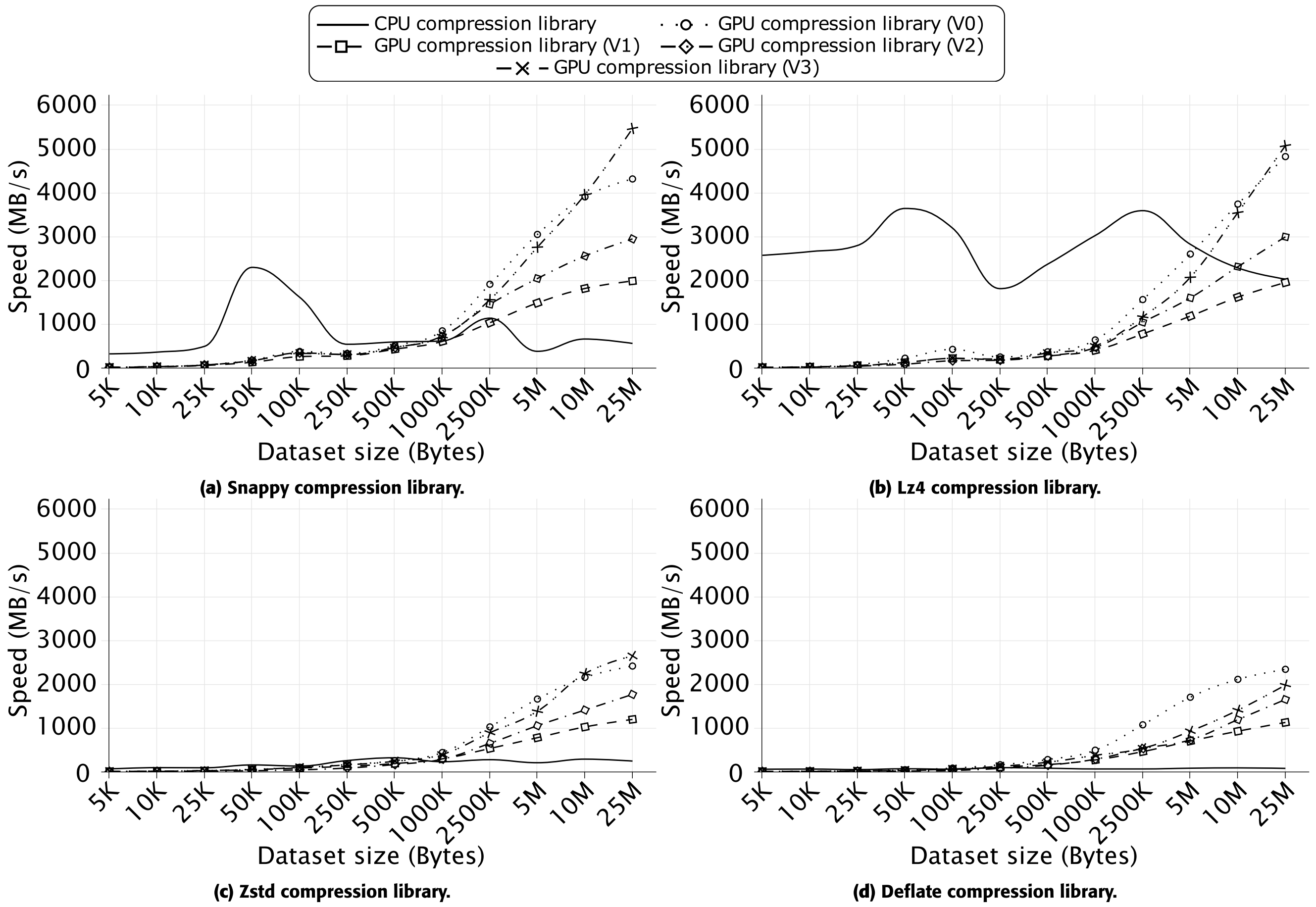

**FIGURE 16.** Comparison of the compression and decompression speed achieved by CPU libraries, and GPU versions V0 (ideal), V1 (naive), V2 (pipeline) and V3 (multi-stream).

the size increases, all compression libraries present the same behavior:

- V1 compression time becomes lower than the CPU from 5*MB*, 100*KB*, and 100*KB* dataset sizes using Snappy, Zstd, and Deflate compression libraries, respectively. However, the CPU implementation of Lz4 is faster and V1 cannot improve it. V2 does it from 25*MB* dataset sizes.
- V2 implementation gets better results using fast compression libraries. This version enhances V1 compression using Snappy and Lz4 regardless of the dataset used. For Zstd and Deflate, improvement is achieved from 500*KB* and 25*MB* dataset sizes, respectively. The improvement in compression is more noticeable when the data set is large enough.
- Using Zstd and Deflate, V3 improves V2 compression regardless of the dataset sizes. For the rest of the libraries, the improvement is achieved from 2500*KB* dataset sizes. The compression improvement achieved by V3 seems more noticeable with slow libraries.

From these results, we conclude that by using sufficiently large data in compression, V1 improves CPU results despite being a simple version. Moreover, the pipeline implemented in V2 achieves better compression results for all libraries. V2 works better with fast compression libraries because it overlaps copies. However, compression kernels are essential in compression time, being the bottleneck. Thus, the slower the kernels are, the less noticeable the improvement of V2 is. The improved version V3 further reduces compression time, being more notable with slow libraries. V3 overlaps compression kernels, this is why it is more noticeable in slow compression libraries, where kernels take a more critical role. Finally, it should be noted that although the objective of showing the compression time achieved by the ideal V0 was just for reference, V3 achieves better results for some datasets thanks to overloading the GPU. Thus, V3 overcomes the ideal V0 with the Deflate compression library from 2500*KB* dataset sizes, while it does it for the rest of the libraries from 5*MB* dataset sizes.

Figure 18 presents results for decompression using the implemented GPU versions. Comparing these results with the compression ones, we can observe similar conclusions. However, decompression times are faster than compression ones, as observed in the previously presented execution traces using the NVIDIA Nsight Systems profiler. In general, decompression requires less computation than compression. In decompression, the implemented

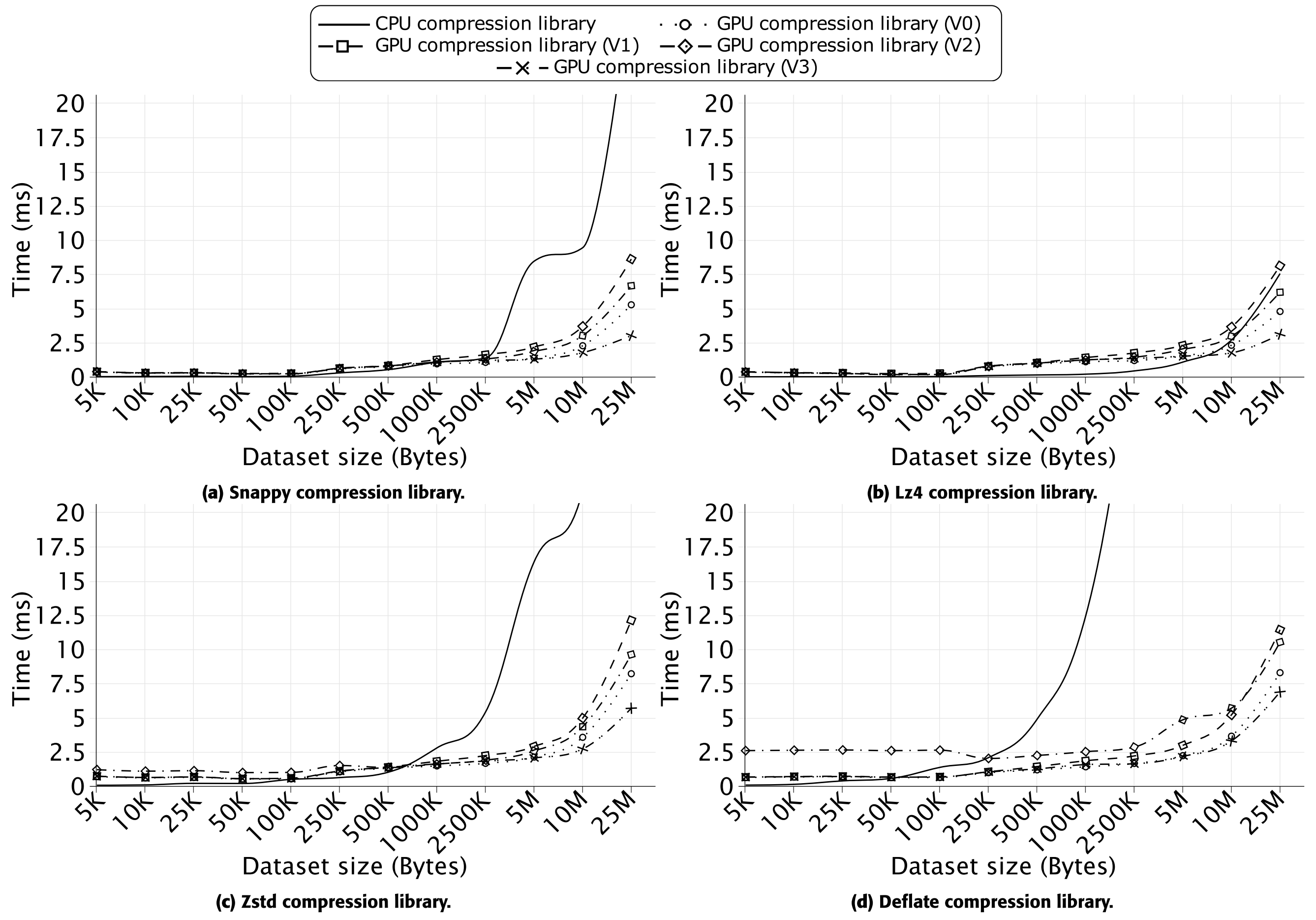

**FIGURE 17.** Comparison of the compression time achieved by CPU libraries, and GPU versions V0 (ideal), V1 (naive), V2 (pipeline) and V3 (multi-stream).

versions behave similarly to compression except for two exceptions:

- V3 enhances V2 for any dataset size and compression library. In the compression stage, this only happens when the compression library is slow enough.
- V3 does not get better results than the ideal V0. This is because the decompression computation performed on the GPU is considerably lower than that performed in compression. Thus, in this case the bottleneck are the copies made to/from the GPU (H2D and D2H) instead of the kernel computation.

As discussed, V3 achieves better decompression results for all dataset sizes and libraries, while compression does not. To better understand this behavior, we analyze compression and decompression traces again when using the V3 implementation (Figures 13 and 14). As we can see, in the compression the CPU must synchronize the compression system to store data in contiguous memory. However, this is not necessary for the decompression because we know the position where the data will be located. This synchronization penalizes fast compression kernels. Therefore, it penalizes fast compression libraries.

### F. VERSION 4 (V4): MULTI-GPU COMPRESSION

Previous versions of our compression solution only use one GPU. However, many computers in data centers today have multiple GPUs. In this section, we introduce "Version 4 (V4)", an improved V3 implementation that leverages multiple GPUs. This new version distributes the computations over all the available GPUs in the system. This presents two main benefits. First, more computing power is available for compression and decompression. Second, multiple data copies (one per GPU) of the same kind (H2D or D2H) can be performed concurrently. The latter is very important because having multiple H2D parallel copies means that data to compress/decompress is copied faster to the compression/decompression kernels. The result is also copied back faster from the computation kernels to host memory. However, supporting multiple GPUs also has the disadvantage of increasing complexity. Data chunks must be distributed among the available GPUs to perform compression in parallel. Moreover, when kernels complete execution and the corresponding D2H memory copies finish, the resulting compressed data (i) must be stored in host memory, (ii) in the correct order, and (iii) in a contiguous memory region. Thus, the CPU thread must be improved

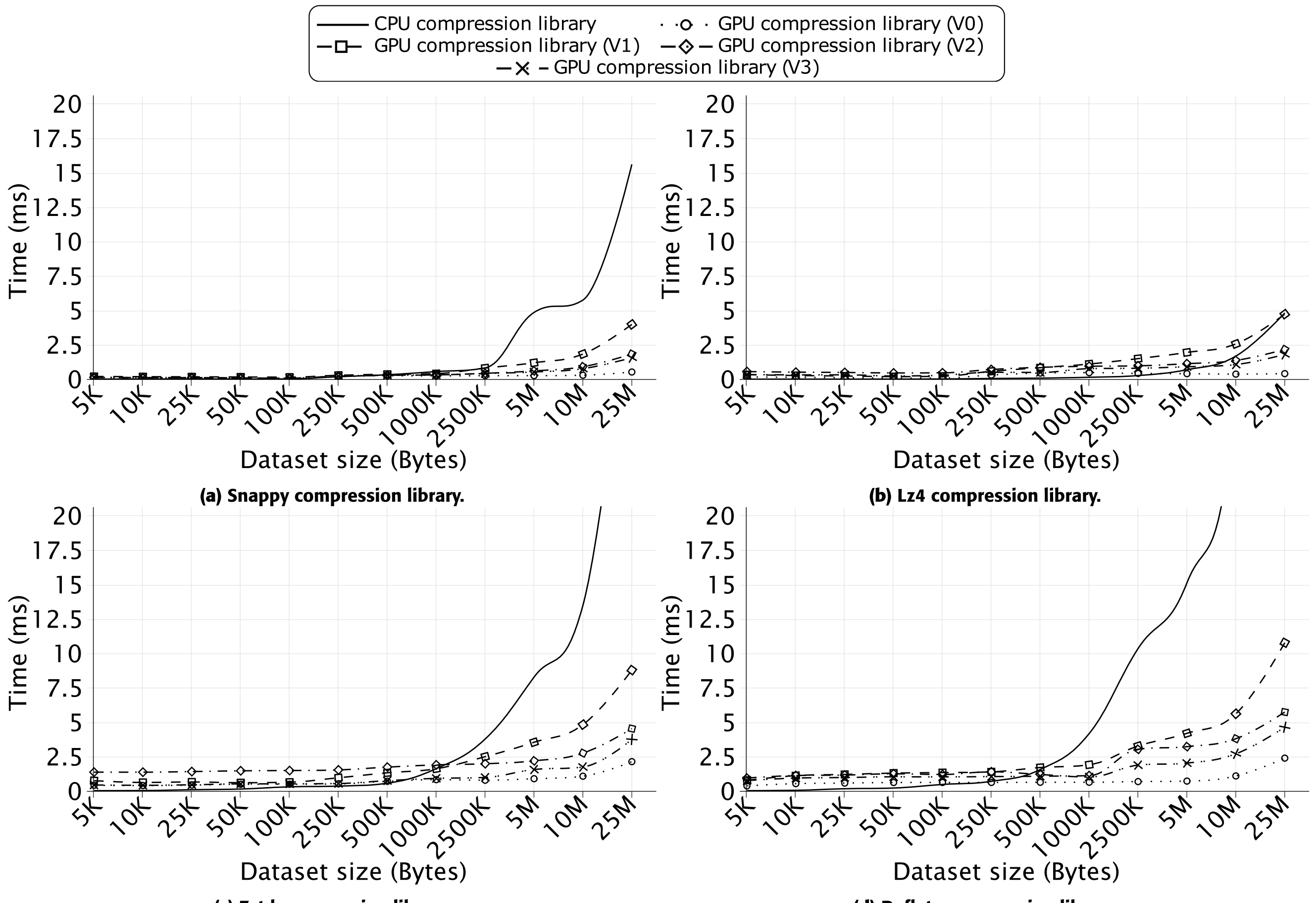

**FIGURE 18.** Comparison of the decompression time achieved by CPU libraries, and GPU versions V0 (ideal), V1 (naive), V2 (pipeline) and V3 (multi-stream).

to properly orchestrate the entire compression process, synchronizing the work done by the various GPUs available in the system.

Figure 19 and Figure 20 show the execution trace of Snappy V4 when compressing and decompressing a 25*MB* dataset, using a chunk size of 2*MB*, up to six concurrent kernels and two GPUs. As we can see, data copies (H2D and D2H) on different GPUs overlap. In addition, kernels running on different GPUs also overlap. Similarly to what happened in V3, kernels also overlap within the same GPU. Notice that decompression kernels in Figure 20 have fewer overlaps because the execution time of these kernels is shorter than that of compression ones.

Figure 21 presents the speed-up achieved by the V4 implementation with respect to V3. Data is compressed and decompressed using two GPUs and the same compression libraries as in previous sections. The optimal number of kernel streams and the optimal chunk size used in V4 is the same as for V3. These values are optimal per GPU, regardless of the number of GPUs. As we can see, in the case of Snappy, Lz4, and Zstd, V4 overcomes V3 for datasets larger than 2*MB*. This makes sense because these compression libraries use a chunk size of 2*MB*. Thus, multiple GPUs are used for data sizes over that chunk size. Deflate presents a similar behavior but for datasets larger than 1*MB*, which is the chunk size used for this library. The best speed-up (up to 1.7*x*) is obtained by Deflate, followed by Zstd (up to 1.5*x*). Snappy and Lz4 present the lower speed-up (up to 1.2*x* and 1.3*x*, respectively).

As we can observe, despite the increment of GPUs from one to two, no compression library has achieved a speed-up of two. It would be the ideal result. The best speed-up is achieved by Deflate, 1.7*x*. In this version, devices work independently. However, it also has the disadvantage of increasing complexity, as explained previously. Once compression kernels finish, results must be transferred to host memory in the correct order and in a contiguous memory region. These synchronization points increase the complexity of multi-GPU compression. This is more noticeable when using fast compression libraries, such as Snappy or Lz4 are used.

## G. VERSION 5 (V5): HYBRID-SMASH (HYBRID CPU-GPU COMPRESSION)

Performance results in previous sections have shown that CPU compression performs better than GPU compression for small data sizes. GPU setup time penalizes GPU performance for these data sizes. As data size increases, however, GPU

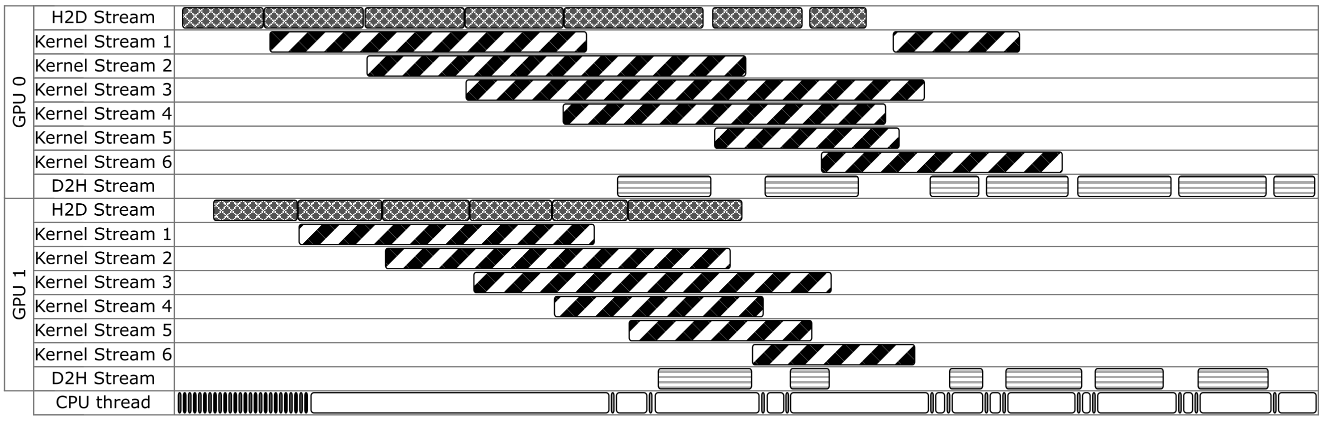

**FIGURE 19.** Execution trace of Snappy V4 (multi-GPU) when compressing a 25*MB* dataset, using a chunk size of 2*MB*, up to six concurrent kernels and two GPUs. This trace was obtained using the NVIDIA Nsight Systems profiler. The legend is shown in Figure 1.

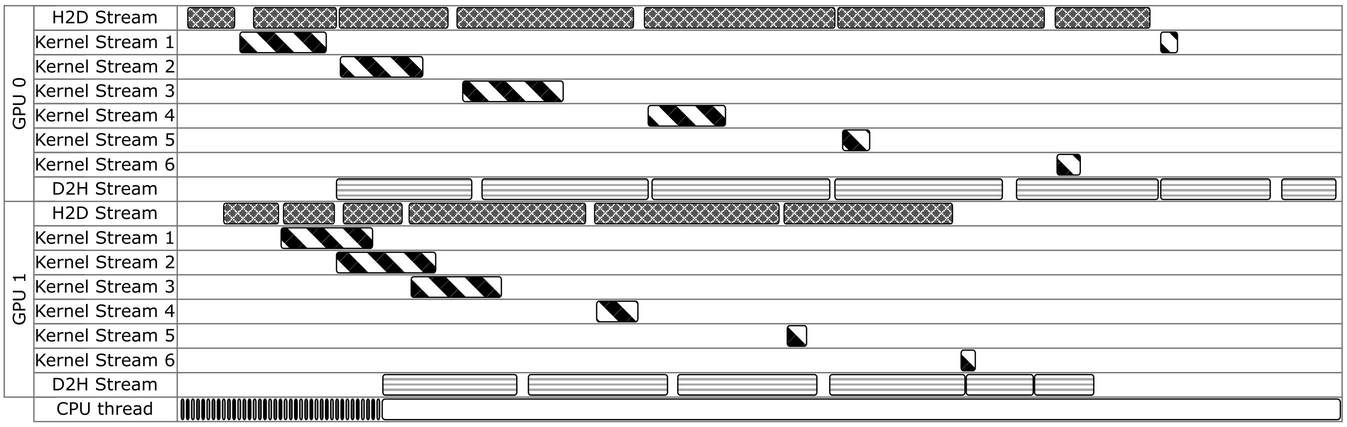

**FIGURE 20.** Execution trace of Snappy V4 (multi-GPU) when decompressing a 25*MB* dataset, using a chunk size of 2*MB*, up to six concurrent kernels and two GPUs. This trace was obtained using the NVIDIA Nsight Systems profiler. The legend is shown in Figure 1.

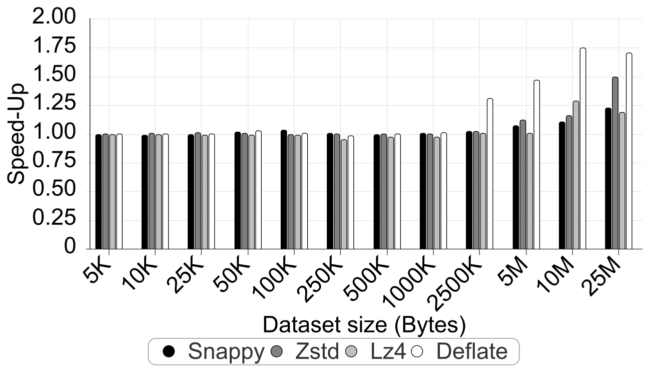

**FIGURE 21.** Speed-up obtained by V4 (multi-GPU) with respect to V3 (multi-stream) when compressing and decompressing using two GPUs.

setup time is compensated by the longer computing time, and GPU compression performs better than CPU. For this reason, in this section, we present "Version 5 (V5)", a heterogeneous CPU-GPU compression library. It uses the CPU or the GPU depending on (i) the size of the data to compress/decompress and (ii) the compression library used. We also referred to this final version as "Hybrid-Smash". Table 1 shows the data size thresholds used for each compression library to decide whether to use the CPU or the GPU. If the size of the data to compress is below the threshold, it will be compressed using the CPU; otherwise, the GPU will be used. The thresholds have been chosen taking into account the results of the experiments in the previous sections.

We would like to emphasize that our implementation encompasses the culmination of extensive research outlined in the paper. This version has been developed thanks to the iterations done in the different versions of GPU-based compression libraries implemented, and the research of the best configuration for each of them.

**TABLE 1.** Data size thresholds used for each compression library to decide whether to use the CPU or the GPU. If the size of the data to compress is below the threshold, the CPU is used; otherwise, the GPU is used.

| Compression Library | Data Size Threshold |
|---|---|
| Snappy | 1000KB |
| Lz4 | 10MB |
| Zstd | 1000KB |
| Deflate | 500KB |

## IV. CONCLUSION AND FUTURE WORK

In this paper, we have investigated when GPUs are beneficial for data compression and when CPUs are preferred. We have done this by comparing the performance of the same compression algorithms on both the CPU and the GPU. Based on these results, we have presented our new abstraction compression library and we explain in an iterative manner how we have arrived at this version. Thus, we have discussed the different versions created and the improvements introduced in each one: the initial naive GPU compression (V1), implementing a GPU compression pipeline (V2), using multiple GPU streams (V3), leveraging multiple GPUs (V4), and the final hybrid CPU-GPU compression (V5).

During the development of this final version, our developed GPU compression library outperformed the nvCOMP provided by NVIDIA for some compression libraries when the data size was large enough. Furthermore, contrary to this one, our GPU library uses data initially located in the CPU memory and stores the resulting data in the CPU memory.

Finally, notice that V5 (referred to as `Hybrid-Smash`) is based on the knowledge acquired during all the evolution process followed in the paper. `Hybrid-Smash` is a heterogeneous CPU-GPU compression library. Our research shows that CPU compression performs better than GPU compression for small data sizes. For this reason, Hybrid-Smash transparently uses CPU or GPU compression depending on the size of the data to be compressed and the compression library to be used.

The research carried out in this paper provides several opportunities for future work, among which we highlight the following:

- Collaborative version based on CPU and GPU. In this paper, we have created a basic heterogeneous CPU-GPU compression library that uses a CPU or a GPU implementation depending on the data size and the specific compression library. However, developing a more complex version that decides which implementation to use and even a collaboration between both implementations depending on factors such as the system load would be interesting. In this way, this future version would appropriately distribute the chunks to be (de)compressed between the CPU and GPU implementations presented in this paper.
- On-the-fly CPU-GPU communication system. As commented in previous sections, we plan to apply this research in new challenging scenarios, such as the rCUDA remote GPU virtualization framework. This type of solutions present a client-server architecture with GPUs installed only on the server side. Thus, this new communication system will use CPU-based implementation on the client side, because there is no GPU there. However, the server side is more complex because there is a GPU and both GPU and CPU implementations can be used. Thus, on the server side it would be possible to locate data to (de)compress on the device or on the host memory, while in this paper we have only considered the case where the data is located on the host. All that would lead to further research to find the best configuration.

## REFERENCES

[1] J. Uthayakumar, M. Elhoseny, and K. Shankar, "Highly reliable and low-complexity image compression scheme using neighborhood correlation sequence algorithm in WSN," *IEEE Trans. Rel.*, vol. 69, no. 4, pp. 1398–1423, Dec. 2020.

[2] M. R. Chowdhury, S. Tripathi, and S. De, "Adaptive multivariate data compression in smart metering Internet of Things," *IEEE Trans. Ind. Informat.*, vol. 17, no. 2, pp. 1287–1297, Feb. 2021.

[3] Y. Kim, S. Choi, D. Lee, J. Jeong, J. Kwak, J. Lee, G. Lee, S. Lee, K. Park, J. Jeong, W. Kexin, and Y. H. Song, "Low-overhead compressibility prediction for high-performance lossless data compression," *IEEE Access*, vol. 8, pp. 37105–37123, 2020.

[4] D. Marpe, T. Wiegand, and G. J. Sullivan, "The H.264/MPEG4 advanced video coding standard and its applications," *IEEE Commun. Mag.*, vol. 44, no. 8, pp. 134–143, Aug. 2006.

[5] P. Merkle, K. Müller, A. Smolic, and T. Wiegand, "Efficient compression of multi-view video exploiting inter-view dependencies based on H.264/MPEG4-AVC," in *Proc. IEEE Int. Conf. Multimedia Expo.*, Jul. 2006, pp. 1717–1720.

[6] G. Liebl, T. Schierl, T. Wiegand, and T. Stockhammer, "Advancedwireless multiuser video streaming using the scalable video coding extensions of H.264/MPEG4-AVC," in *Proc. IEEE Int. Conf. Multimedia Expo.*, Jul. 2006, pp. 625–628.

[7] NVIDIA Corporation. (2023). *The NvCOMP Library*. Accessed: Apr. 18, 2023. [Online]. Available: https://developer.nvidia.com/nvcomp

[8] C. Peñaranda, C. Reaño, and F. Silla, "Smash: A compression benchmark with AI datasets from remote GPU virtualization systems," in *Proc. Int. Conf. Hybrid Artif. Intell. Syst. (HAIS)*. Salamanca, Spain: Springer, 2022, pp. 236–248.

[9] X. Liang, S. Di, D. Tao, S. Li, S. Li, H. Guo, Z. Chen, and F. Cappello, "Error-controlled lossy compression optimized for high compression ratios of scientific datasets," in *Proc. IEEE Int. Conf. Big Data (Big Data)*, Dec. 2018, pp. 438–447.

[10] J. Ziv and A. Lempel, "A universal algorithm for sequential data compression," *IEEE Trans. Inf. Theory*, vol. IT-23, no. 3, pp. 337–343, May 1977.

[11] J. Ziv and A. Lempel, "Compression of individual sequences via variable-rate coding," *IEEE Trans. Inf. Theory*, vol. IT-24, no. 5, pp. 530–536, Sep. 1978.

[12] A. Moffat, "Huffman coding," *ACM Comput. Surv.*, vol. 52, no. 4, pp. 1–35, 2019.

[13] I. H. Witten, R. M. Neal, and J. G. Cleary, "Arithmetic coding for data compression," *Commun. ACM*, vol. 30, no. 6, pp. 520–540, 1987.

[14] J. Alakuijala, A. Farruggia, P. Ferragina, E. Kliuchnikov, R. Obryk, Z. Szabadka, and L. Vandevenne, "Brotli: A general-purpose data compressor," *ACM Trans. Inf. Syst.*, vol. 37, no. 1, pp. 1–30, Jan. 2019.

[15] C. Peñaranda. *SMASH: Compression Abstraction Library*. Accessed: May 20, 2023. [Online]. Available: https://github.com/cpenaranda/smash

[16] Google. *Snappy GitHub Repository*. Accessed: Apr. 18, 2023. [Online]. Available: https://github.com/google/snappy

[17] Y. Collet. *Lz4 GitHub Repository*. Accessed: Apr. 18, 2023. [Online]. Available: https://github.com/lz4/lz4

[18] Y. Collet. *Zstandard GitHub Repository*. Accessed: Apr. 18, 2023. [Online]. Available: https://github.com/facebook/zstd

[19] E. Biggers. (2023). *Deflate-Based GitHub Repository*. Accessed: Apr. 18, 2023. [Online]. Available: https://github.com/ebiggers/libdeflate

[20] R. A. Patel, Y. Zhang, J. Mak, A. Davidson, and J. D. Owens, *Parallel Lossless Data Compression on the GPU*. San Jose, CA, USA: IEEE, 2012.

[21] A. Ozsoy and M. Swany, "CULZSS: LZSS lossless data compression on CUDA," in *Proc. IEEE Int. Conf. Cluster Comput.*, Sep. 2011, pp. 403–411.

[22] M. Chłopkowski and R. Walkowiak, "A general purpose lossless data compression method for GPU," *J. Parallel Distrib. Comput.*, vol. 75, pp. 40–52, Jan. 2015.

[23] J. Li, J. Wu, and G. Jeon, "GPU acceleration of clustered DPCM for lossless compression of hyperspectral images," *IEEE Trans. Ind. Informat.*, vol. 16, no. 5, pp. 2906–2916, May 2020.

[24] NVIDIA Corporation. (2023). *CUDA (Compute Unified Device Architecture)*. [Online]. Available: https://developer.nvidia.com/cuda-toolkit

[25] R. Arnold and T. Bell, "A corpus for the evaluation of lossless compression algorithms," in *Proc. Data Compress. Conf.*, Mar. 1997, pp. 201–210.
[26] S. Deorowicz, "Universal lossless data compression algorithms," Ph.D. dissertation, Dept. Autom. Control, Electron. Comput. Sci., Inst. Comput. Sci., Silesian Univ. Technol., Gliwice, Poland, 2003, p. 38.
[27] S. Iserte, J. Prades, C. Reaño, and F. Silla, "Improving the management efficiency of GPU workloads in data centers through GPU virtualization," *Concurrency Comput., Pract. Exper.*, vol. 33, no. 2, pp. 1–16, Jan. 2021, doi: 10.1002/cpe.5275.
[28] C. Reaño and F. Silla, "Redesigning the rCUDA communication layer for a better adaptation to the underlying hardware," *Concurrency Comput., Pract. Exper.*, vol. 33, no. 14, pp. 1–17, Jul. 2021, doi: 10.1002/cpe.5481.
[29] C. Peñaranda, C. Reaño, and F. Silla, "Exploring the use of data compression for accelerating machine learning in the edge with remote virtual graphics processing units," *Concurrency Comput., Pract. Exper.*, vol. 35, no. 20, p. e7328, Sep. 2023.
[30] Nvidia. *Nsight Systems 2023*. Accessed: May 15, 2023. [Online]. Available: https://developer.nvidia.com/nsight-systems

**CRISTIAN PEÑARANDA** received the B.S. degree in informatics engineering and the M.S. degree in artificial intelligence, pattern recognition and digital imaging from Universitat Politècnica de València, Valencia, Spain, in 2015 and 2016, respectively, where he is currently pursuing the Ph.D. degree in computer science.

**CARLOS REAÑO** (Member, IEEE) is currently an Assistant Professor in computer architecture with Universitat de València, Spain. He is also an external collaborator of the rCUDA project, in which he worked, from 2011 to 2018. He has published several papers in peer-reviewed conferences and journals. He is involved in different ways in several conferences and journals. His research interests include the virtualization of GPU-accelerators in both HPC clusters and cloud/edge computing systems. For more information, see https://mural.uv.es/caregon.

**FEDERICO SILLA** received the M.S. and Ph.D. degrees in computer engineering from Universitat Politècnica de València, Spain, in 1995 and 1999, respectively. He is currently a Full Professor with the Department of Computer Engineering, Universitat Politècnica de València. He worked for two years at Intel Corporation developing on-chip networks. He has published 30 JCR-indexed journal articles, more than 100 contributions to international conferences, ten book chapters, 43 invited talks, and 12 tutorials, providing an H-index impact factor equal to 28 according to Google Scholar. He has been a coordinator of the rCUDA remote GPU virtualization project, since 2008. His research interests include high-performance on-chip and off-chip interconnection networks and remote GPU virtualization mechanisms. He is an Associate Editor of the *Journal of Parallel and Distributed Computing*. For more information, see http://www.disca.upv.es/fsilla.