



UNIVERSITAT
POLITÈCNICA
DE VALÈNCIA

DEPARTAMENTO DE INFORMÁTICA
DE SISTEMAS Y COMPUTADORES

Using compression to enhance bandwidth on low-performance networks

*A thesis submitted in partial fulfillment of
the requirements for the degree of*

*Doctor of Philosophy
(Computer Engineering)*

Author

Cristian Peñaranda Cebrian

Advisors

Prof. Federico Silla Jiménez

Dr. Carlos Reaño González

May 2025

Agradecimientos

Descubrí el mundo de la informática gracias a mi hermano Roberto. Él fue quien despertó en mí la curiosidad por la ciencia y decidí seguir sus pasos estudiando Ingeniería Informática.

Durante la universidad tuve la suerte de unirme al Grupo de Tecnología Informática - Inteligencia Artificial (GTI-IA) donde inicié mis primeros pasos en la investigación. Gracias a mis supervisores de entonces, Vicente Julián, Javier Palanca y Soledad Valero, por haberme dado la oportunidad de adentrarme en este mundo. Pero, sin duda, lo más importante de esta etapa fueron las increíbles personas a las que conocí y que a día de hoy sigo teniendo la suerte de llamar amigos. Guillem, Jose Alemany, Jose 2, Javi y Elías: gracias por todos los momentos que hemos vivido juntos, por las risas, los cafés interminables y por hacer que la universidad se sintiera como un lugar en el que siempre había alguien con quien contar. Echo de menos aquellos días, de verdad. En especial, agradecer a Elías por haberme presentado a mi director de tesis, Fede. Se podría decir que esta tesis existe gracias a ti.

Quiero expresar también mi agradecimiento a todo el Grupo de Arquitecturas Paralelas (GAP), en especial al equipo rCUDA. Gracias a mis directores de tesis, Fede y Carlos, por haberme dado la oportunidad de realizar esta tesis bajo vuestra guía. A lo largo de esta etapa he aprendido muchísimo y os debo gran parte de ese aprendizaje. Gracias también a mis compañeros de equipo durante todos estos años, Pablo, Toni, Jaime y Javi Prades, por vuestro apoyo y por todos los momentos compartidos. Hay que retomar las rutas de senderismo del equipo rCUDA.

Por último quiero agradecer a las personas más importantes de mi vida. A mi hermano Roberto, por haber sido mi ejemplo a seguir. A mi madre Esperanza, por enseñarme que el esfuerzo y la dedicación se ven recompensados. A mi padre Patricio, por darme el valor a enfrentarme a los problemas, a pesar de que a veces no sea fácil. Y, por último, a mi pareja Cristina. Gracias por haber sido mi mayor apoyo durante la tesis, por haberme animado a continuar, a no tirar la toalla, pero sobre todo, gracias por ser la luz que ha iluminado mi camino durante estos 10 años. Gracias por ser la voz de la razón y de la esperanza. Si no hubiera sido por ti, posiblemente esta tesis no la hubiera terminado.

Muchas gracias a todos vosotros.

Contents

Abstract	ix
<i>Resumen</i>	xi
<i>Resum</i>	xiii
List of Figures	xv
List of Tables	xxiii
1 Introduction	1
1.1 Motivation	2
1.2 Objectives and contributions	6
1.3 Thesis outline	7
2 Related Work	11
2.1 Compression algorithms	12
2.1.1 <i>Huffman</i> coding	13
2.1.2 <i>Arithmetic</i> coding	14
2.1.3 <i>Run-Length Encoding</i>	16

2.1.4	<i>LZ77</i> coding	18
2.1.5	<i>LZ78</i> coding	20
2.1.6	<i>LZW</i> coding	22
2.1.7	Discussion about compression algorithms	23
2.2	Standard datasets	25
2.3	Remote GPU virtualization solutions	27
3	A new dataset for AI applications	33
3.1	Applications selected to create the AI dataset	34
3.2	Determining the appropriate dataset	36
3.3	Dataset creation from rCUDA, a remote GPU virtualization system	38
3.4	Summary	46
4	CPU-based compression libraries	49
4.1	<i>CPU-Smash</i>	50
4.1.1	Available configurations	52
4.1.2	Available compression libraries	53
4.1.3	Usage example	63
4.2	Evaluation	67
4.2.1	Synthesizing results	68
4.2.2	Evaluating standard and AI datasets	70
4.3	Summary	72
5	Introducing GPU-based compression libraries into the compression procedure	75
5.1	<i>GPU-Smash</i>	76

5.1.1	Available configurations	77
5.1.2	Available compression libraries	78
5.1.3	Usage example	80
5.2	Developing an efficient GPU-based compression library	83
5.2.1	Version 0 (V0): ideal GPU compression	85
5.2.2	Version 1 (V1): naive GPU compression	87
5.2.3	Version 2 (V2): GPU compression pipeline	88
5.2.4	Version 3 (V3): multi-stream GPU compression	94
5.2.5	Version 4 (V4): multi-GPU compression	99
5.3	A heterogeneous CPU-GPU compression library	101
5.3.1	Performance comparison of CPU- and GPU-based compression . .	101
5.3.2	<i>Hybrid-Smash</i>	104
5.4	Summary	105
6	Improving network communications using compression libraries	109
6.1	Introduction	110
6.2	Characterization of data transfers	112
6.2.1	Compression evaluation of data transfers	113
6.3	On-the-fly compression	116
6.3.1	Transfers from the edge memory to the remote GPU memory	117
6.3.2	Transfers from the remote GPU memory to the edge memory	124
6.3.3	Impact on overall application performance	127
6.4	A pipelined approach for on-the-fly compression	130
6.4.1	How the compression pipeline works	132

6.4.2	Experimental evaluation	133
6.4.3	Rationale for using CPU-based compression libraries	136
6.4.4	Optimizations focused on small data	139
6.5	An optimized parallel pipeline approach to on-the-fly compression	141
6.5.1	Experimental setup	144
6.5.2	Finding the best parameters for the parallel compression pipeline . .	147
6.5.3	Impact on the parallel compression pipeline performance	153
6.5.4	To send or not to send compressed data	160
6.6	Summary	164
7	Conclusions	167
7.1	Contributions	168
7.2	Future directions	169
7.3	Publications	170
	References	173

Abstract

The Internet of Things (IoT) has experienced significant growth in recent years, driving the need to process data on devices with limited resources efficiently. However, these devices typically have reduced computing capabilities, making it challenging to run demanding applications such as those based on artificial intelligence. A common strategy is to offload specific tasks to the edge, but the computational power of these devices remains insufficient for more complex workloads.

Remote GPU virtualization emerges as a viable solution to address these computational limitations, enabling IoT devices to leverage GPUs located on remote servers to accelerate application processing. In this way, devices can efficiently execute GPU-intensive tasks by offloading the computation to a more powerful machine. This approach significantly extends the computing capabilities of IoT systems.

However, remote GPU virtualization introduces a new challenge: network dependency. For the system to function efficiently, data must be quickly transferred between the device and the remote GPU. Data transmission can drastically affect application performance in networks with limited bandwidth, such as those commonly found in IoT environments. Therefore, reducing the amount of transmitted data and optimizing communication between the client and the server becomes a key factor.

In this thesis, several solutions have been developed to mitigate this problem through data compression within the communication layer of remote GPU virtualization. Different compression techniques have been explored, including a pipeline-based compression system to improve data transmission, and abstraction libraries that group many compression libraries: 41 CPU-based and 8 GPU-based. In addition, a specific dataset was created to evaluate the compression performance in this context. Finally, tests were carried out using AI applications on low-performance networks to analyze the impact of these solutions and propose future improvements.

Resumen

El Internet de las Cosas (IoT) ha experimentado un crecimiento significativo en los últimos años, impulsando la necesidad de procesar datos de manera eficiente en dispositivos con recursos limitados. Sin embargo, estos dispositivos suelen tener capacidades de cómputo reducidas dificultando la ejecución de aplicaciones exigentes, como las de inteligencia artificial. Una estrategia común es realizar ciertas tareas en el edge, pero la potencia de cómputo de estos dispositivos sigue siendo insuficiente para tareas más complejas.

Para abordar estas restricciones computacionales, la virtualización remota de GPU se presenta como una solución viable, con la que los dispositivos IoT pueden aprovechar las GPUs ubicadas en servidores remotos para acelerar el procesamiento de sus aplicaciones, delegando el procesamiento a servidores con mayor capacidad computacional. Este enfoque permite ampliar significativamente las capacidades de cómputo del IoT.

Sin embargo, la virtualización remota de GPU introduce un nuevo desafío: la dependencia de la red. Para que el sistema funcione de manera eficiente, los datos deben transferirse de manera rápida. En redes con ancho de banda limitado, como las que suelen encontrarse en entornos IoT, la transmisión de datos puede afectar drásticamente el rendimiento de las aplicaciones. De este modo, reducir la cantidad de datos transmitidos y optimizar la comunicación entre el cliente y el servidor se convierte en un aspecto clave.

En esta tesis, se han desarrollado varias soluciones para mitigar este problema mediante la compresión de datos dentro de la capa de comunicación de la virtualización remota de GPU. Se han explorado diferentes técnicas de compresión, incluyendo un sistema de compresión en pipeline para mejorar la transmisión de datos y bibliotecas de abstracción que agrupa un gran número de bibliotecas de compresión: 41 basadas en compresión usando la CPU y 8 usando la GPU. Además, se ha creado un dataset específico para evaluar la compresión en este contexto. Finalmente, se han llevado a cabo pruebas con aplicaciones de inteligencia artificial en redes de bajas prestaciones para analizar el impacto de estas soluciones y proponer mejoras futuras.

Resum

Internet de les Coses (IoT) ha experimentat un creixement significatiu en els darrers anys, impulsant la necessitat de processar dades de manera eficient en dispositius amb recursos limitats. No obstant això, aquests dispositius solen tindre capacitats de càlcul reduïdes, cosa que dificulta l'execució d'aplicacions exigents, com les d'intel·ligència artificial. Una estratègia habitual és dur a terme certes tasques a l'edge, però la potència de càlcul d'aquests dispositius continua sent insuficient per a tasques més complexes.

Per afrontar aquestes restriccions computacionals, la virtualització remota de GPU es presenta com una solució viable, amb la qual els dispositius IoT poden aprofitar les GPUs ubicades en servidors remots per accelerar el processament de les seues aplicacions, delegant aquest processament a servidors amb una capacitat computacional superior. Aquest enfocament permet ampliar significativament les capacitats de càlcul de l'IoT.

No obstant això, la virtualització remota de GPU introduïx un nou repte: la dependència de la xarxa. Perquè el sistema funcione de manera eficient, les dades s'han de transferir ràpidament. En xarxes amb amplada de banda limitada, com les que es troben habitualment en entorns IoT, la transmissió de dades pot afectar dràsticament el rendiment de les aplicacions. Per això, reduir la quantitat de dades transmeses i optimitzar la comunicació entre el client i el servidor esdevé un aspecte clau.

En aquesta tesi, s'han desenvolupat diverses solucions per mitigar aquest problema mitjançant la compressió de dades dins de la capa de comunicació de la virtualització remota de GPU. S'han explorat diferents tècniques de compressió, incloent-hi un sistema de compressió en pipeline per millorar la transmissió de dades i biblioteques d'abstracció que agrupen un gran nombre de biblioteques de compressió: 41 basades en compressió utilitzant la CPU i 8 utilitzant la GPU. A més, s'ha creat un conjunt de dades específic per avaluar la compressió en aquest context. Finalment, s'han dut a terme proves amb aplicacions d'intel·ligència artificial en xarxes de baixes prestacions per analitzar l'impacte d'aquestes solucions i proposar millores futures.

List of Figures

1.1	Impact of low-performance bandwidth on application performance.	4
2.1	Example of compressing the word ‘ABRACADABRA’ using <i>Huffman</i> coding.	13
2.2	Example of compressing the word ‘TEST’ using <i>Arithmetic</i> coding.	15
2.3	Example of compressing the word ‘AAAAAABBBAABBA’ using <i>Run-Length Encoding</i>	17
2.4	Example of compressing the word ‘ABRACADABRA’ using <i>LZ77</i> coding.	19
2.5	Example of compressing the word ‘ABRACADABRA’ using <i>LZ78</i> coding.	21
2.6	Example of compressing the word ‘ABRACADABRA’ using <i>LZW</i> coding.	23
2.7	General architecture of remote GPU virtualization solutions.	30
2.8	Example of using rCUDA environment variables.	31
3.1	Sample of Cifar10 dataset.	34
3.2	Sample of Flowers dataset.	35
3.3	Sample of Mnist dataset.	35
3.4	Sample of images used by Alexnet test.	36
3.5	A diagram of how the data from the Artificial Intelligence (AI) dataset was obtained.	39
3.6	A diagram illustrating where data from the AI dataset are obtained from.	40

3.7	Histogram of the data transfers when running TF applications under study. The primary Y-axis represents the number of transfers done by TF applications, while the secondary Y-axis represents the total size of data exchanged in GB.	42
3.8	Histogram of the data transfers when running TF applications under study. Notice that the Y-axis represents the frequency of data transfers with a specific size range performed by each TF application.	43
4.1	Smash benchmark execution using all the available compression libraries and the <i>dickens</i> dataset.	66
4.2	<i>LodePNG</i> library using all possible combinations of parameters.	67
4.3	Legend used in Figure 4.6, Figure 4.4 and Figure 4.5.	70
4.4	Comparison between a standard 25KB data and the AI 25KB data.	71
4.5	Comparison between a standard 100KB data and the AI 100KB data.	71
4.6	Average results for all datasets and compression libraries.	72
5.1	Comparison of the typical approach followed by CPU- and GPU-based compression libraries.	78
5.2	Legend for execution traces shown in Figures 5.3, 5.5, 5.7, 5.10, 5.11, 5.12, 5.13, 5.14, 5.17, and 5.18.	85
5.3	Execution trace for Version 0 (V0). The legend is shown in Figure 5.2. . . .	85
5.4	The compression and decompression speed achieved by Version 0 (V0). . .	86
5.5	Execution trace for Version 1 (V1). The legend is shown in Figure 5.2. . . .	87
5.6	Comparison of the compression and decompression speed achieved by Version 0 (V0) and Version 1 (V1).	88
5.7	Execution traces for compression of Version 1 (V1), Version 2 (V2), and intermediate versions. The legend is shown in Figure 5.2.	89

5.8	Exploration to find the optimal chunk size for Version 2 (V2) using the 25MB data obtained from the AI dataset.	92
5.9	Comparison of the compression and decompression speed achieved by Version 0 (V0), Version 1 (V1), and Version 2 (V2).	93
5.10	Execution trace of <i>Snappy</i> using Version 2 (V2) when compressing the 25MB data obtained from the AI dataset using a chunk size of 8MB. This trace was obtained using the NVIDIA Nsight Systems profiler. The legend is shown in Figure 5.2.	93
5.11	Execution trace of <i>Snappy</i> using Version 2 (V2) when decompressing compressed the 25MB data obtained from the AI dataset using a chunk size of 8MB. This trace was obtained using the NVIDIA Nsight Systems profiler. The legend is shown in Figure 5.2.	93
5.12	Execution trace of the synthetic test in Listing 5.2 to check if data copies overlap when using multiple streams and the same kind of transfers. This trace was obtained using the NVIDIA Nsight Systems profiler. The legend is shown in Figure 5.2.	95
5.13	Execution trace of <i>Snappy</i> using Version 3 (V3) when compressing the 25MB data obtained from the AI dataset, using a chunk size of 2MB and up to six concurrent kernels. This trace was obtained using the NVIDIA Nsight Systems profiler. The legend is shown in Figure 5.2.	96
5.14	Execution trace of <i>Snappy</i> using Version 3 (V3) when decompressing the 25MB data obtained from the AI dataset, using a chunk size of 2MB and up to six concurrent kernels. This trace was obtained using the NVIDIA Nsight Systems profiler. The legend is shown in Figure 5.2.	96
5.15	Exploration to find the optimal chunk size for Version 3 (V3) using the 25MB data obtained from the AI dataset and varying the number of kernel streams.	97
5.16	Comparison of the compression and decompression speed achieved by Version 0 (V0), Version 1 (V1), Version 2 (V2), and Version 3 (V3).	98

5.17	Execution trace of <i>Snappy</i> using Version 4 (V4) when compressing the 25MB data obtained from the AI dataset, using a chunk size of 2MB, up to six concurrent kernels and two GPUs. This trace was obtained using the NVIDIA Nsight Systems profiler. The legend is shown in Figure 5.2.	100
5.18	Execution trace of <i>Snappy</i> using Version 4 (V4) when decompressing the 25MB data obtained from the AI dataset, using a chunk size of 2MB, up to six concurrent kernels and two GPUs. This trace was obtained using the NVIDIA Nsight Systems profiler. The legend is shown in Figure 5.2.	100
5.19	Speed-up obtained by Version 4 (V4) with respect to Version 3 (V3) when compressing and decompressing the AI dataset using two and one A100 GPU respectively.	100
5.20	Comparison of the compression ratio achieved by compression libraries using their CPU- and GPU-based implementations.	102
5.21	Comparison of the compression and decompression speed achieved by CPU- and GPU-based (V3) implementations.	103
5.22	<i>Hybrid-Smash</i> workflow.	105
6.1	Scenarios explored in the experiments. Scenario A represents a real-world application in which no compression system is used. In Scenario B, CPU-based compression is used to compress data to be transferred within rCUDA.	112
6.2	Histogram of the data transfers when running TF applications under study. Notice that Y-axis represents the frequency of data transfers with a specific size range performed by each TF application.	114
6.3	Normalized size of the compressed data transferred through the network by TF applications when offloading GPU computations from the edge node to the remote GPU server. Note that a normalized size X means that the compressed data size is X times the original one.	115
6.4	On-the-fly compression workflow.	117

-
- 6.5 Normalized time of transfers done by *Alexnet* application when using on-the-fly CPU-based compression. It includes the time for compressing, sending, and decompressing the data. It has been normalized using the time of transferring the data without using on-the-fly compression. Only memory data copies from the edge node to the remote GPU are considered. 118
- 6.6 Normalized time of transfers done by *Cifar10* application when using on-the-fly CPU-based compression. It includes the time for compressing, sending, and decompressing the data. It has been normalized using the time of transferring the data without using on-the-fly compression. Only memory data copies from the edge node to the remote GPU are considered. 120
- 6.7 Normalized time of transfers done by *Mnist* application when using on-the-fly CPU-based compression. It includes the time for compressing, sending, and decompressing the data. It has been normalized using the time of transferring the data without using on-the-fly compression. Only memory data copies from the edge node to the remote GPU are considered. 122
- 6.8 Normalized time of transfers done by *Inception* application when using on-the-fly CPU-based compression. It includes the time for compressing, sending, and decompressing the data. It has been normalized using the time of transferring the data without using on-the-fly compression. Only memory data copies from the edge node to the remote GPU are considered. 123
- 6.9 Normalized time of transfers done by TF applications when using on-the-fly CPU-based compression over a 10 Mbps network. It includes the time for compressing, sending, and decompressing the data. It has been normalized using the time of transferring the data without using on-the-fly compression. Only memory data copies from the remote GPU to the edge node are considered. 125

6.10	Impact of transferring compressed data on the overall performance of the TF applications using different networks. <i>Original</i> refers to the case where no compression is used. It only requires preparing the data to be sent and performing some computations with the data (referred to as “Computation time” in the figure), and the “Transfer time”. The time when the different libraries under study are used is labeled with the name of each library. It is broken down in four parts: (i) time for preparing the data to be sent and performing some computations with the data (i.e., “Computation time”); (ii) “Compression time”, (iii) “Transfer time”, and (iv) “Decompression time”. <i>Adaptive</i> refers to our new mechanism which uses the best compression library depending on the size of the data and the network speed.	128
6.11	Compression approaches using the remote GPU virtualization solution rCUDA.	130
6.12	Comparison between pipelined and non-pipelined on-the-fly compression. .	132
6.13	Scenarios explored in the experiments. Scenario A represents a real-world application in which only the CPU of the edge device is used. In Scenario B, the real-world application runs using rCUDA. Scenario C presents the scenario where rCUDA is utilized with on-the-fly compression implemented in its pipeline communication layer to run the real-world application.	134
6.14	Results obtained when running TensorFlow applications in different scenarios: (i) offloading computations using only the CPU of the edge device, (ii) using rCUDA over a 10 Mbps network without compression, and (iii) using rCUDA over a 10 Mbps network with CPU-based pipelined on-the-fly compression employing the libraries under study.	135
6.15	Scenario with on-the-fly GPU-based compression integrated in rCUDA pipeline communication layer.	136
6.16	Scenario with an alternative approach for integrating on-the-fly GPU-based compression in rCUDA pipeline communication layer.	138
6.17	Percentage of data transfers of the most repeated 8-byte data values when running the TF applications under study.	140
6.18	A high-level diagram of the implementation of the parallel compression pipeline using n compression/decompression threads and m data chunks. . .	142

6.19	Scenarios explored in the experiments.	144
6.20	Data used in experiments.	146
6.21	Study of data chunk sizes using a 1 Gbps network and <i>Gipfeli</i> compression library. Each figure presents the bandwidth ('Bw') obtained using a different number of threads and data chunks. The horizontal line in the plots corresponds to a trade-off ('TO') equal to zero.	149
6.22	Study of the number of data chunks using a 1 Gbps network and <i>Gipfeli</i> compression library.	150
6.23	Study of the number of threads using a 1 Gbps network and <i>Gipfeli</i> compression library.	151
6.24	Results using a 1 Gbps network. Different scenarios are considered: 'No Compression' (rCUDA without using compression); ' <i>Library</i> Pipelined' (rCUDA using the parallel compression pipeline system with specific CPU-based compression library); ' <i>Library</i> Non-pipelined' (rCUDA using compression with the specific CPU-based compression library, without the pipeline system). The scenarios are evaluated with datasets of different sizes. More details about these scenarios are in Section 6.5.1.	156
6.25	Results using a 100 Mbps network. Different scenarios are considered: 'No Compression' (rCUDA without using compression); ' <i>Library</i> Pipelined' (rCUDA using the parallel compression pipeline system with specific CPU-based compression library); ' <i>Library</i> Non-pipelined' (rCUDA using compression with the specific CPU-based compression library, without the pipeline system). The scenarios are evaluated with datasets of different sizes. More details about these scenarios are in Section 6.5.1.	158
6.26	Results using a 10 Mbps network. Different scenarios are considered: 'No Compression' (rCUDA without using compression); ' <i>Library</i> Pipelined' (rCUDA using the parallel compression pipeline system with specific CPU-based compression library); ' <i>Library</i> Non-pipelined' (rCUDA using compression with the specific CPU-based compression library, without the pipeline system). The scenarios are evaluated with datasets of different sizes. More details about these scenarios are in Section 6.5.1.	159

- 6.27 Normalized bandwidth obtained over a 1 Gbps network by ‘*Gipfeli* Pipelined’ and ‘*Gipfeli* Pipelined V2’ concerning the ‘No Compression’ one when using traced data. The ‘*Gipfeli* Pipelined V2’ only sends the compressed data if its size is smaller than the original uncompressed data. 161
- 6.28 Normalized bandwidth obtained over a 100 Mbps network by ‘*Gipfeli* Pipelined’ and ‘*Gipfeli* Pipelined V2’ concerning the ‘No Compression’ one when using traced data. The ‘*Gipfeli* Pipelined V2’ only sends the compressed data if its size is smaller than the original uncompressed data. . 162
- 6.29 Normalized bandwidth obtained over a 10 Mbps network by ‘*Gipfeli* Pipelined’ and ‘*Gipfeli* Pipelined V2’ concerning the ‘No Compression’ one when using traced data. The ‘*Gipfeli* Pipelined V2’ only sends the compressed data if its size is smaller than the original uncompressed data. 163
- 6.30 Percentage of compressed chunks sent to the network with ‘*Gipfeli* Pipelined V2’ when using data with different sizes of traced data. 163

List of Tables

2.1	Data collected from different datasets that represent the standard dataset used in our experiments.	27
3.1	The compression ratio achieved by different compression libraries when compressing different kinds of data.	37
3.2	Name of files that make up in the new AI dataset created to evaluate compression libraries. The name of each file indicates its size.	44
3.3	Compression ratios achieved by compression libraries when compressing the new AI dataset.	44
3.4	Entropy values of various datasets used in compression tests.	46
4.1	Specifications of the computer used in the CPU-based compression libraries experiments.	67
4.2	All possible configurations of CPU-based compression libraries.	69
5.1	Specifications of the computer used to experiment with GPU-based compression libraries.	84
5.2	Thresholds of the data size used to decide when using the CPU or the GPU with a specific compression library.	104
6.1	Client and server node specifications used in the CPU-based compression experiments.	113

6.2	Summary of the best results achieved by CPU-based compression when evaluating on-the-fly compression.	127
6.3	Number of data transfers (i.e., memory copies) made during the execution of the TF applications under study.	129
6.4	Characterization of the 8-byte data transfers performed by the TF applications used in the experiments.	139
6.5	Specifications of the client and server node used in CPU-based compression libraries evaluation when a parallel compression pipeline communication is used.	145
6.6	Parameters used to analyze the performance of the parallel compression pipeline system.	147
6.7	Summary of the best parameters for the parallel compression pipeline system using <i>Gipfeli</i> and a 1 Gbps network for each number of threads used in the pipeline.	150
6.8	Summary of the best parameters for the parallel compression pipeline system using lineal data, different compression libraries, and networks.	152
6.9	Summary of the best parameters for the parallel compression pipeline system using random data, different compression libraries, and networks.	152
6.10	Summary of the best parameters for the parallel compression pipeline system using sparse data, different compression libraries, and networks.	154
6.11	Summary of the best parameters for the parallel compression pipeline system using traced data, different compression libraries, and networks.	154

Chapter 1

Introduction

This chapter serves as an introduction to this thesis. It lays the foundation for the following chapters. The chapter begins by delving into the primary motivation behind undertaking this research. The aim is to provide readers with a clear understanding of the driving forces behind the study.

Following the motivation, the chapter proceeds to articulate the specific objectives and contributions of the research. These objectives serve as guiding principles, outlining the intended outcomes and goals that the study endeavors to achieve.

Finally, the chapter concludes by outlining the subsequent chapters that comprise the dissertation. This outline provides readers with a roadmap of the thesis. It also previews topics that will be explored in greater detail later.

1.1 Motivation

Artificial Intelligence (AI) is pervasive and has significant societal consequences, impacting our lifestyle and the economy. On the one hand, it has recently positioned NVIDIA as a leading tech company since it manufactures key AI technologies. On the other hand, AI helps reduce costs in research and development processes. For example, it has enabled the creation of strong chipsets without using rare earth elements in a very short time, which potentially may alter geopolitics due to the industrial importance of these elements. AI is also beneficial in many other areas, and anyone with internet access can interact with AI engines. It is even integrated into our cell phones and laptops.

The increasing presence of AI in our daily lives is primarily attributed to deep learning models based on neural networks. These networks are not new. Actually, they were created more than 50 years ago. However, their computationally intensive requirements made them unavailable in the past. Nevertheless, those computationally intensive requirements have been addressed nowadays by using accelerators such as graphics processing units (GPUs). Typically, deep learning applications rely on powerful computers with last-generation GPUs [88], like the recent device created by NVIDIA, the Blackwell B200, which features 192 GB of memory and can achieve up to 20 petaflops of FP4 performance. However, there are emerging domains, like the Internet of Things (IoT), where lighter devices may be more suitable for executing deep learning models in specific scenarios, far from the mainstream use of AI widely known by the general public.

IoT refers to the network of interconnected devices that communicate and exchange data through the Internet. Examples of IoT applications include smart home devices like wearable health systems [101] or industrial sensors used in manufacturing [50] and agriculture [76]. These devices collect data from their surroundings to perform tasks, improve efficiency, and enhance user experience. The versatility of IoT applications highlights the growing integration of innovative technology into everyday life and various industries.

In IoT environments, large amounts of data are captured by devices. These data can be stored in data centers for later use and analysis or, on the contrary, can be processed to extract the most significant characteristics to avoid consuming vast amounts of storage. In this latter case, these data can be used to make high-quality decisions using deep learning models [37]. These decisions are used by the IoT devices, or other devices close to them, to carry out specific actions related to the environment where they are installed. One approach to making these high-quality decisions is to send the captured data to a powerful GPU-equipped

computer in the cloud for processing and decision-making, with the results sent back to the IoT device. Another approach is edge/fog computing [99], where computations are performed close to (or even at) the IoT device itself. This approach provides energy savings and scalability benefits because the edge/fog paradigm moves computation from large data centers to smaller computers much closer to IoT devices. This change in the locations of the computations means data does not need to be sent far away to the computing data centers, thus saving energy. Furthermore, scalability is improved because computing is distributed in small facilities instead of centralized at large data centers.

However, power constraints limit IoT device computing capabilities [14]. They often use ultra-low power solutions like ARM-based CPUs that are less suitable for executing deep learning models. Even if these ARM-based devices include an accelerator (e.g., GPU), their computational power is noticeably smaller than that of a last-generation GPU. One alternative to enhance the computing capabilities of edge devices is the remote virtualization of accelerators [46, 15, 51]. This mechanism allows edge devices to access powerful remote GPUs transparently, increasing their computational power without requiring modifications to the source code of applications. Thus, edge devices do not pay for an energy increase while they can access a large number of powerful remote GPUs.

In this way, using remote GPU virtualization could be an excellent solution to boost systems with no GPUs. That solution allows edge devices to utilize powerful GPUs hosted in remote data centers but requires a robust network connection. Effective remote GPU virtualization relies on high-bandwidth, low-latency networks to minimize performance degradation. Thus, the network must support fast data transfer between the edge device and the remote GPU. High-performance networks facilitate this data exchange, although these infrastructures are not always available in IoT environments.

Consequently, accessing remote GPUs requires network bandwidth that may exceed the limitations of IoT deployments. In this regard, remote GPU virtualization has demonstrated to work with no significant overhead in data center deployments featuring high-bandwidth interconnects. It has also shown to experience some overhead with low-bandwidth networks.

To illustrate the critical impact of data transfer overhead on application performance, Figure 1.1 shows how transmission time increases as network bandwidth decreases, from 1 Gbps to 100 Mbps and 10 Mbps, using different TensorFlow applications that will be presented in Section 3.1. As the available bandwidth decreases, the data transmission becomes a significant bottleneck, noticeably affecting the overall execution time of the application. That

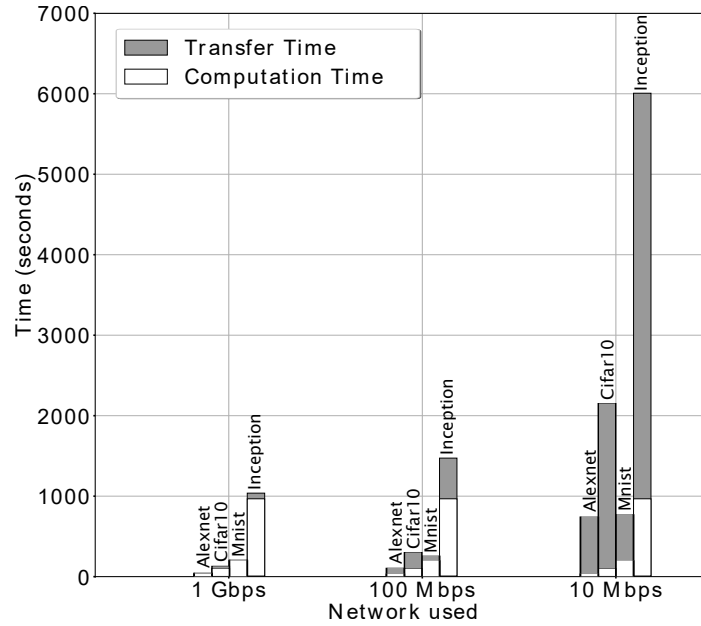


Figure 1.1: Impact of low-performance bandwidth on application performance.

highlights the necessity of implementing strategies to mitigate communication delays and sustain application performance under constrained network conditions.

In this way, network bandwidth could be artificially increased using compression techniques. Data exchanged with the remote GPU can be compressed on-the-fly before being transmitted and decompressed upon arrival. The communication layer of the remote GPU virtualization solution would carry out these compression and decompression processes transparently to the application, i.e., the application is not aware that its data is compressed during transmission. Also, if properly designed, compression could effectively reduce data size (increasing network bandwidth in practice), whereas no extra latency is added to transmissions. This would need a careful co-design of the compression and communication layers.

Nowadays, compression algorithms are widely used to reduce data size and improve the performance and efficiency of various applications that handle large volumes of data [114, 17, 52]. 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.

In general, data compression may be achieved with algorithms that lose information during the compression process or with algorithms that do not lose information. On the one hand, algorithms that lose information do not allow the recovery of the original data later because

part of that data was lost. On the other hand, algorithms that compress information without losing data allow the full recovery of the original information.

Certainly, lossy compression algorithms such as MPEG4 or H.264 are used on some specific application domains [68, 70, 64]. These algorithms noticeably reduce data size at the expense of discarding information. However, this is not a problem in areas like video streaming, where the human eye cannot detect the missing data. Contrary to lossy compression algorithms, lossless compression preserves the original data, thus making it possible to restore the original data from compressed data. Lossless compression can be helpful in scenarios like archiving. In the context of this thesis, the focus is therefore on lossless compression.

However, data compression has a computational cost that may limit its practical use in real-time or resource-constrained scenarios. In order to address this concern, GPUs can be leveraged for compression and decompression instead of CPUs. Although CPU compression algorithms have been widely studied in the past, GPU-based compression algorithms are much more novel and remain a challenge due to the GPU model itself, which imposes different difficulties compared to traditional CPU usage. In particular, GPU-based compression algorithms are very interesting for on-the-fly compression in the context of remote GPU virtualization because at least one of the communicating points presents a GPU. Nevertheless, the usage of such algorithms could not be immediate because of the intricacies of GPUs.

Thus, introducing compression techniques in real-time communications poses two important challenges: (i) reducing the size of transmitted data and (ii) not increasing communication latency. Integrating compression libraries within data transfers while hiding latency can be challenging. Pipelined designs can help hide compression/decompression latency in real-time applications, but compression libraries require careful investigation and consideration of different options. A correct design will be required.

In the context of this thesis, designing a compression-capable communication layer within a remote GPU virtualization middleware requires considering different factors, such as processing time and energy consumption. In addition, compression libraries have different compression ratios and speeds, and the choice of a specific compression library depends on the exact requirements. Moreover, inspecting the internal balance of the different stages of the communication pipeline is also essential. Last but not least, appropriate dataset selection is necessary for evaluating the compression/decompression capabilities of the new design. Thus, a proper dataset [117], representative of the traffic between the application and the remote GPU, is necessary for comprehensive analysis. This dataset will be quite different from existing ones.

Finally, several aspects can influence the benefits of on-the-fly compression within a remote GPU virtualization middleware. Factors such as the size of data to be transmitted, the contents of that data, the hardware used for compression/decompression, and network speed will play significant roles. It is expected that slower networks may benefit from more sophisticated compression algorithms that achieve larger compression ratios thanks to intensive computation. On the contrary, faster networks might require lighter compression algorithms to leverage on-the-fly compression effectively.

In summary, the usage of remote GPU virtualization in novel scenarios such as that composed of IoT devices causes new requirements. One of them is to overcome the slow network speed present in these scenarios. In this sense, on-the-fly compression inside the communications layer of the remote GPU virtualization middleware may help artificially increase network bandwidth. However, creating an efficient compression/decompression pipeline inside the communication layer of remote GPU virtualization solutions is extremely challenging because not only does the size of transmitted data need to be reduced, but increasing communications latency should be avoided.

1.2 Objectives and contributions

The main objective of this thesis is to improve bandwidth in low-performance networks through the usage of data compression when applications are accelerated by remote GPU virtualization systems. This objective is divided into more specific ones to achieve this goal. Each one seeks to address a necessary key aspect for the successful implementation and evaluation of the proposed solutions:

- **Design a representative dataset for experimental simulations.** This dataset must accurately replicate the data exchange scenarios observed in remote GPU virtualization systems.
- **Design a flexible system for the integration of different libraries and implementations.** The system will abstract the complexities of integrating new libraries, allowing users to switch or utilize different implementations with minimal changes to the overall system.
- **Optimize the communication system.** The objective is to enhance the data transmission efficiency between the client host and the remote GPU server.

Note that we focus our study on applications accelerated by virtual remote GPUs. However, the results of this research may also be helpful in other fields. The principal contributions of this thesis that have allowed us to achieve these objectives are:

- **Development of a new AI dataset.** This new dataset represents the data exchanged between the client host memory and the remote GPU server when different AI applications are executed using remote GPU virtualization.
- **Research and collection of CPU-based compression libraries.** This collection is used to develop a CPU-based compression abstraction library. This abstraction library is a higher-level framework that simplifies how to use these libraries.
- **Research and collection of GPU-based compression libraries.** Similarly, this investigation is used to create a GPU-based compression abstraction library. In this case, the included libraries are also improved. We employ a pipeline approach, which improves functionality and performance.
- **Development of a heterogeneous CPU-GPU compression library.** This library leverages CPU-based and GPU-based compression libraries. Considering previously investigated CPU and GPU compression libraries, this new library automatically selects the most suitable implementation. The selection is based on the data to be compressed and the specific compression library. This results in performance optimization.
- **Development of a communication system using compression libraries.** The purpose of this system is to improve network bandwidth. To do this, the data are compressed before transmission to reduce the amount of data transferred.
- **Research and development of a parallel compression pipeline communication system.** This system first divides data to be transferred into smaller chunks. Then, compression, transfer, and decompression of these chunks are performed concurrently in a pipelined manner. This approach hides the time required for compression and decompression in the transfer time. As a result, overall efficiency is improved.

1.3 Thesis outline

In addition to this introductory chapter, this thesis includes seven additional chapters:

- Chapter 2 provides background information to help understand the rest of the thesis. The chapter is divided into three well-differentiated parts. The first part introduces popular compression algorithms used by compression libraries. It discusses the techniques and methods employed. The second part presents the standard datasets used to evaluate these compression libraries. These datasets are commonly used to assess the performance and effectiveness of compression algorithms. The third part introduces existing remote GPU virtualization frameworks.
- Chapter 3 introduces a new dataset designed explicitly for AI. This dataset is one of the contributions of this thesis and was created to achieve the main objective of this work. It represents the transfers performed between the client host memory and the remote GPU server in a remote GPU virtualization system. The purpose of this chapter is to present the dataset, its features, and how it was created. The chapter aims to provide researchers with a valuable resource for training and testing.
- Chapter 4 presents a new compression library called CPU-Smash. This library was created by combining various CPU-based compression libraries. The purpose of CPU-Smash is to provide a convenient way for users to compare and assess different CPU-based compression libraries with minimal modifications to their existing code. This chapter also evaluates the performance of popular CPU-based compression libraries using CPU-Smash. We use the standard datasets presented in Chapter 2, and the new AI dataset developed in Chapter 3 for the evaluation. The goal is to show how CPU-based compression libraries perform depending on the dataset and the compression library. It can help users to decide the most suitable library for their dataset.
- In Chapter 5, a compression library called GPU-Smash is introduced. This library is similar to the previous library CPU-Smash presented in Chapter 4. In this regard, the GPU-Smash library also brings together several compression libraries, but in this case, the compression algorithms are GPU-based instead of CPU-based, as it was the case in Chapter 4. The chapter also evaluates these GPU-based compression libraries, and a pipeline strategy is explored to enhance compression performance. Moreover, compression libraries implemented on CPUs and GPUs are also analyzed. The performance of these implementations is compared based on two factors: speed and compression ratio. Based on the information obtained from this comparison, a new library called Hybrid-Smash is presented. This library aims to provide a more flexible and adaptable approach to compression. Based on the data to be compressed, it determines the most appropriate implementation of a specific compression library:

either CPU or GPU. Then, it automatically uses the implementation that offers the best performance.

- Chapter 6 shows how some of the previous compression libraries artificially increase the network bandwidth in real-world applications using a remote GPU virtualization framework. It introduces and evaluates a compression system for pipeline communication within the remote GPU virtualization framework. The presented approach divides the data to be transferred into different chunks. In this way, (i) one piece of data can be compressed and transferred, (ii) while the next chunk is being compressed, and (iii) the previous chunk is being decompressed. The analysis shows that the overall transfer time is reduced. The pipeline hides the time it takes to perform compression and decompression operations. In addition, a study on an efficient parallel compression pipeline is presented. The number of threads responsible for compressing and decompressing the chunks of data is evaluated. The number and size of these chunks are also considered. The goal is to determine the optimal configuration for the pipeline.
- Chapter 7 serves as a summary of the entire thesis. It provides a concise overview of the main points, arguments, and findings presented throughout the document. Moreover, the chapter discusses future work. It explores potential areas of research. Further investigation that can build on the findings and conclusions of the thesis is also discussed. Finally, the chapter lists the publications resulting from the thesis.

Chapter 2

Related Work

This chapter includes the background necessary to understand the contents of the work presented in this thesis and its significance. First, we discuss the different compression algorithms. These compression algorithms are the basis of the compression libraries later used throughout the thesis. Next, we present the datasets commonly used to evaluate these compression libraries. Finally, at the end of the chapter, we introduce an overview of remote GPU virtualization systems, focusing on rCUDA, which enables the sharing and utilization of GPU resources across multiple applications.

2.1 Compression algorithms

In this section, we focus on the compression algorithms, which are the core techniques used to reduce data size. These compression algorithms define the mathematical and procedural methods that allow data to be compressed with or without loss of information. In contrast, compression libraries refer to the practical implementations of these algorithms. While the libraries encapsulate the complexity of the algorithms, allowing for easier integration into software, it is the algorithms themselves that determine the efficiency and quality of the compression.

Compression can be carried out using two different approaches: lossy compression and lossless compression.

Lossy compression libraries obtain acceptable results with a high compression ratio and compression/decompression speed [62]. However, lossy compression techniques usually discard or lose some data during the compression process. These libraries are commonly used for multimedia files like images, audio, and video, where the minor loss of quality may not be noticeable to human perception. It should be noted that original data cannot be recovered after compression.

On the other hand, lossless compression is a technique that preserves all the original data without any loss. Lossless compression libraries are used when retaining every detail of the original data is crucial. This is important where even minor alterations or loss of information are not acceptable, such as in scientific data, text documents, or program files.

This thesis focuses on lossless compression libraries applied to data before being transmitted so that original data can be restored upon reception. These lossless compression libraries employ different algorithms or techniques to achieve compression while maintaining data integrity. Next, we outline the most significant lossless compression algorithms used.

These algorithms are *Huffman* coding, *Arithmetic* coding, *Run-Length Encoding*, *LZ77* coding, *LZ78* coding, and *LZW* coding. We have selected these algorithms because of their very different nature. Notice that this thesis does not aim to create new compression algorithms but to study the consequences of applying compression to data transmission. Thus, the focus of this thesis is on integrating existing compression algorithms in the communication pipeline of remote GPU virtualization solutions, and analyze the benefits of their use. In this regard, we aim to explore how very different compression algorithms contribute to enhancing bandwidth efficiency.

2.1.1 Huffman coding

The *Huffman* [74] algorithm is a variable-length prefix coding technique. This means that it uses a coding table based on the frequency of symbols presented in the input data, assigning shorter codes to more frequent symbols and more extended codes to less frequent ones.

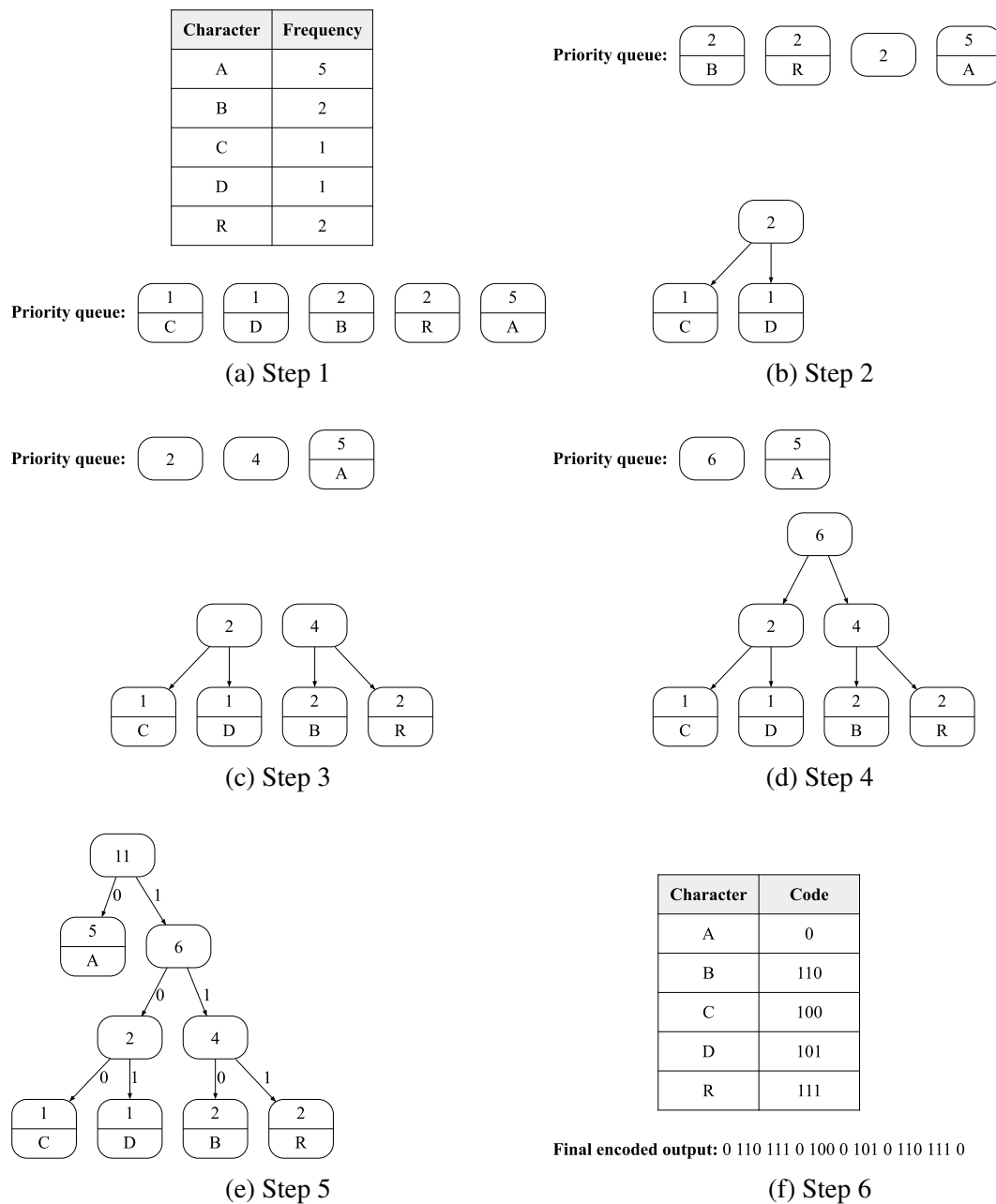


Figure 2.1: Example of compressing the word 'ABRACADABRA' using *Huffman* coding.

The basic idea is to create a binary tree, called a *Huffman* tree, where the more probable symbols will have shorter binary codes. During compression, each symbol is replaced by its corresponding binary code. In the decompression process, the *Huffman* tree information is needed to replace each code with its corresponding symbols. Therefore, compressed data must be accompanied by the *Huffman* tree information.

Figure 2.1 shows a step-by-step example where the word ‘ABRACADABRA’ is compressed using *Huffman* coding. In order to set the *Huffman* tree, the first step is to obtain the frequency of each symbol, i.e., the times the symbol appears within the data, and then order symbols in a priority queue using this frequency (Figure 2.1a). Then, the two less frequent symbols are selected to create the first sub-tree (Figure 2.1b). In our example, symbols ‘C’ and ‘D’ are selected to create this sub-tree. The union of these symbols creates a new node with the accumulated frequency. In our example, this new node contains the number ‘2’. This new node is also added to the priority queue, and the process is repeated until emptying the priority queue (Figures 2.1c, 2.1d, and 2.1e). Figure 2.1e shows the complete *Huffman* tree for our example. Once the *Huffman* tree is finished, the next step is to code each symbol. So, the left branches are encoded with 0 while the right ones are encoded with 1. Then, the code is obtained following the route to achieve the symbol. The table with the new codes for each symbol is presented in Figure 2.1f. The final step is to replace each symbol with its specific code. The word ‘ABRACADABRA’ was represented using 88 bits (eight bits per symbol). However, it is reduced to 23 bits after compressing using *Huffman* coding. To recover the original data, codes must be replaced by the correct symbol to decompress the data.

2.1.2 Arithmetic coding

The *Arithmetic* [123] algorithm uses a continuous-range coding technique. This technique encodes the entire data sequence into a single, continuously subdivided range or interval. Unlike other compression techniques that work on fixed-sized blocks or symbols, continuous-range coding processes the whole data, progressively reducing intervals to represent it.

The arithmetic algorithm assigns real numbers in the interval $[0, 1]$ to represent the input data using the frequency of symbols. In this way, the more probable symbols have larger sub-intervals. *Arithmetic* coding is more complex than *Huffman* coding but can achieve higher compression rates. The reason for this is that arithmetic coding does not require assigning fixed-length codes to symbols but represents entire data sequences as a single number within the interval.

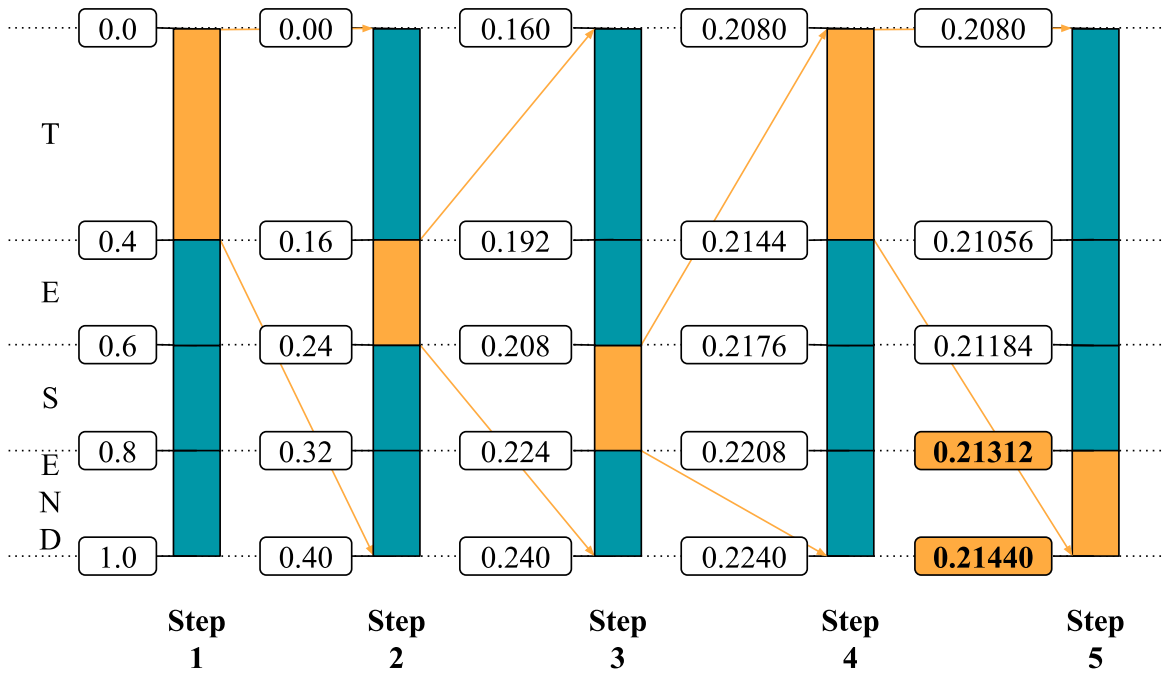


Figure 2.2: Example of compressing the word ‘TEST’ using *Arithmetic* coding.

Figure 2.2 presents the *Arithmetic* coding process when compressing the word ‘TEST’. Initially, the process begins by calculating the probability of each symbol, i.e., the times each symbol appears within the input data divided by the total number of symbols. Notice that there is a special symbol to indicate the end of the data to be compressed that we have called ‘END’. This symbol is also used to calculate the probabilities. In this example, the symbol ‘T’ has a probability of 0.4 (since it appears twice), while the symbols ‘E’, ‘S’, and ‘END’ each have a probability of 0.2 . Notice the sum of all symbol probabilities equals 1 .

Next, each symbol is assigned a range within the interval $[0, 1]$ based on this probability. In our example, ‘T’ is assigned the range $[0.0, 0.4[$, ‘E’ the range $[0.4, 0.6[$, ‘S’ the range $[0.6, 0.8[$, and ‘END’ the range $[0.8, 1[$. Once these new ranges are established, the process continues by selecting the symbol that matches the first symbol in the data (Step 1 in Figure 2.2 shows how ‘T’ is selected, the first character of the data). At this point, we know that the compressed word will fall within the interval $[0.0, 0.4[$ since this range corresponds to all words starting with ‘T’. The process must continue until the last symbol, ‘END’, is reached.

In the next step, new ranges are assigned based on the selected range ($[0.0, 0.4[$) and the probabilities of the symbols. The selected range $[0.0, 0.4[$ now acts as the new “interval”

within which the next symbol will be encoded. The same probabilities are used to subdivide this interval, just as they were used for the initial interval $[0, 1]$. This means that the interval $[0.0, 0.4]$ is split according to the probabilities: 'T' takes $[0.0, 0.16]$, 'E' takes $[0.16, 0.24]$, 'S' takes $[0.24, 0.32]$, and 'END' takes $[0.32, 0.4]$. The reason these new ranges are smaller is because we are refining the range to only focus on sequences of symbols that start with 'T'. The original interval $[0, 1]$ represented all possible sequences, but now that the first 'T' has been processed, the new range $[0.0, 0.4]$ represents only sequences that begin with 'T'. By subdividing this range, we encode the next symbol relative to the remaining possibilities. Then, the following symbol from the data, 'E', is selected (as shown in Step 2 of Figure 2.2).

The process continues by further narrowing the range based on the next symbol until the last symbol that marks the end is reached (Step 5 in Figure 2.2). Finally, the result is a range of data, and any value inside this range represents the word compressed. In our example, all the values in the range $[0.21312, 0.21440]$ are possible solutions.

The decompression process is very similar to the compression one. Using a numerical value, we must navigate symbol by symbol until the 'END' symbol is reached. To start the decompression, we need (i) the numerical value representing the encoding and (ii) the probability of each symbol.

Using again the example illustrated in Figure 2.2, we explain next how it would be decompressed the value '0.214' representing the word 'TEST'. Each symbol is assigned a range within the interval $[0, 1]$ based on his probability. We can observe that the value '0.214' belongs to symbol 'T' whose interval is the range $[0.0, 0.4]$. So, in the first step, the symbol 'T' is selected as the first decompressed symbol. The selected range $[0.0, 0.4]$ now acts as the new "interval" within which the next symbol will be encoded. The same probabilities are used to subdivide this interval, just as they were used for the initial interval $[0, 1]$. Using the same strategy, we can see that the value '0.214' belongs to the symbol 'E' whose interval is the range $[0.16, 0.24]$. The process continues by narrowing the range based on the next symbol until the last symbol that marks the end is reached (Step 5 in Figure 2.2).

2.1.3 Run-Length Encoding

The *Run-Length Encoding (RLE)* algorithm is a simple lossless data compression technique that focuses on coding consecutive repeated symbols. The algorithm is very easy to follow. Figure 2.3 illustrates the compression and decompression of the word 'AAAAAABB-BAABBA' using *RLE*.

Codification: (length of match, symbol)

Iteration	Word	Current Symbol	Current Count	Encoded Output	Decoded Output
0	<u>A</u> AAAAABBBAAABBA	A	1	-	-
1	<u>AA</u> AAAAABBBAAABBA	A	2	-	-
2	<u>AAA</u> AAAAABBBAAABBA	A	3	-	-
3	<u>AAAA</u> AABBBAAABBA	A	4	-	-
4	<u>AAAAA</u> ABBBAAABBA	A	5	-	-
5	<u>AAAAAA</u> ABBBAAABBA	A	6	(6, A)	AAAAAA
6	AAAAAA <u>B</u> BBAAABBA	B	1	-	AAAAAA
7	AAAAAA <u>BB</u> BAABBA	B	2	-	AAAAAA
8	AAAAAA <u>BBB</u> AABBA	B	3	(3, B)	AAAAAABBB
9	AAAAAABBB <u>A</u> ABBA	A	1	-	AAAAAABBB
10	AAAAAABBB <u>AA</u> BBA	A	2	(2, A)	AAAAAABBBAA
11	AAAAAABBBAA <u>B</u> BA	B	1	-	AAAAAABBBAA
12	AAAAAABBBAA <u>BB</u> A	B	2	(2, B)	AAAAAABBBAA
13	AAAAAABBBAAAB <u>B</u> A	A	1	(1, A)	AAAAAABBBAA

Final output: (6, A) (3, B) (2, A) (2, B) (1, A)

Figure 2.3: Example of compressing the word 'AAAAAABBBAAABBA' using *Run-Length Encoding*.

In the compression, the algorithm stores the first symbol ('A' in the example shown in Figure 2.3) and sets the counter to one. Next, the algorithm runs through the word until it finds the first symbol different from the one stored. For each symbol that matches the stored one, the counter increments by one. In our example, the next symbol different from 'A' is 'B'. Then, the algorithm codifies all the consecutive repeated symbols using the number of repeated symbols and the current symbol. In this example, the algorithm detects six consecutive repeated symbols and codifies them using '(6, A)'. At this point, the algorithm stores the next symbol, sets the counter to one, and continues using the same strategy. In our example, the algorithm gets the symbol 'B' and continues with the procedure.

The decompression process is more effortless than the compression one. The algorithm reads the encoded data, which is composed of different pair values representing the number of repeated symbols and the specific symbol. In this way, the algorithm expands each pair of values, setting the specific symbol as many times as the number of repeated symbols indicates. After expanding all pairs, the algorithm obtains the complete uncompressed data.

2.1.4 LZ77 coding

The LZ77 [124] algorithm, named after its inventors Abraham Lempel and Jacob Ziv, was introduced in 1977. This lossless compression algorithm identifies and replaces repeating data sequences with previous references. It uses a sliding window technique to compress the data. That window refers to previously processed symbols, the most recent ones. The algorithm starts with an empty window, and symbols are included in the window as the compression progresses. The algorithm replaces the repeated patterns with references to previous instances of the same data. In this way, the algorithm searches the current symbol in the window. If that symbol is found in the window, the next symbol in the input data is also considered by creating a pattern. The pattern is searched again in the window. This process continues until no more symbols can be added to the existing pattern in the sliding window. Once the process of adding symbols to the pattern ends, the pattern is finally replaced by: the offset where it starts, the length of the match, and the next symbol that is part of the pattern. These two first parameters would be zero when there are no occurrences, setting only the symbol that is not found. Notice that limiting the offset and the maximum match length could be a promising strategy to obtain a favorable compression ratio. For example, we can use 2 bytes where 12 bits could represent the offset (a window size of 4KB) and 4 bits could define the match length (16 possible symbols).

Codification: (offset, length of match, symbol)

Iteration	Word	Window	Encoded Output	Decoded Output
1	<u>A</u> BRACADABRA		(0, 0, A)	A
2	A <u>B</u> BRACADABRA	A	(0, 0, B)	AB
3	AB <u>R</u> BRACADABRA	AB	(0, 0, R)	ABR
4	ABR <u>A</u> CADABRA	ABR	(3, 1, C)	ABR A C
5	ABRAC <u>A</u> DABRA	ABRAC	(2, 1, D)	ABRAC A D
6	ABRACAD <u>ABRA</u>	ABRACAD	(7, 4, -)	ABRACAD ABRA

Final output: (0, 0, A) (0, 0, B) (0, 0, R) (3, 1, C) (2, 1, D) (7, 4, -)

Figure 2.4: Example of compressing the word ‘ABRACADABRA’ using *LZ77* coding.

Figure 2.4 illustrates an example of *LZ77* compression when the word ‘ABRACADABRA’ is compressed. In the first iteration, the algorithm considers the first symbol (symbol ‘A’) and looks for a match of that pattern in the sliding window. However, the sliding window is empty, and no match is found. Therefore, the first item of codification is zero for the offset, zero for the length of the match, and ‘A’ for the symbol.

Next, the algorithm considers the next symbol of the input data, the symbol ‘B’. This is shown in the second iteration. The algorithm looks for a match of that symbol in the sliding window. However, no match is found again, and this new item of codification is zero for the offset, zero for the length of the match, and ‘B’ for the symbol. A similar process is followed with the symbol ‘R’ in the third iteration.

In the fourth iteration, the symbol ‘A’ is already in the window, but the pattern ‘AC’ does not. For that reason, the offset is three (three positions behind the current symbol), the length is one because there is only one occurrence, and the symbol is ‘C’ (the not found symbol). The same happens with the fifth iteration, with the symbol ‘D’.

Finally, in the sixth iteration, the last four symbols match perfectly with a previous occurrence. That is, the sequence ‘ABRA’ is already present in the sliding window. So, again, the algorithm indicates where the occurrence starts and the number of symbols of that occurrence.

In this case, the new item of codification is 7 for the offset, 4 for the length of the match, and the end symbol ‘-’ to indicate that there are no more data.

The final encoded output can be found in the bottom part of Figure 2.4. It can be derived that in this particular example, if 12 bits are used to represent the offset and 4 bits are used to define the match length, then no compression is achieved because the encoded output is longer than the original data. Compressing very short data can produce this effect. Nevertheless, the compression ratio is much improved for long enough input data.

In order to retrieve the original data from the compressed data, the decompression process must be carried out. To decompress, the algorithm just follows the indications set in the codification items. The first three iterations do not contain any occurrences, so just the symbol is put. For the rest of the iterations, the offset and the match length are utilized to copy the occurrence, and the symbol that does not match is put into the decompressed data.

2.1.5 LZ78 coding

The LZ78 [125] algorithm is a lossless data compression algorithm named after its inventors (Abraham Lempel and Jacob Ziv) and the year it was published (1978). It is a dictionary-based compression algorithm. It builds a dictionary of patterns encountered in the input data and represents each repeated pattern using dictionary references. The algorithm starts with an empty dictionary. For every pattern in the input data not found in the dictionary, the new pattern is added to the dictionary, and the output includes the dictionary index of the previous sequence and the new character. This process continues, with new patterns being added to the dictionary and referenced by dictionary indices. In the final output, these patterns are finally replaced by an index that refers to the match and the new symbol. A zero value in the index means no occurrence exists, and only the symbol must be used. The dictionary can be limited to reduce the maximum number of indexes and, consequently, the number of bits used to represent them. For example, indexes can be represented by 2 bytes that represent 65,536 different indexes. It is important to note that sending the dictionary with the compressed data is not necessary because it will be built from compressed data in the decompression stage.

Figure 2.5 displays the LZ78 compression example when the word ‘ABRACADABRA’ is compressed. Similar to the LZ77 algorithm, the LZ78 one cannot find any occurrence with the three first symbols.

Codification: (index, symbol)

Index	Word	Encoded Output	Decoded Output
0	ABRACADABRA	-	-
1	<u>A</u> BRACADABRA	(0, A)	A
2	AB <u>B</u> RACADABRA	(0, B)	AB
3	ABR <u>R</u> ACADABRA	(0, R)	ABR
4	ABR <u>AC</u> ADABRA	(1, C)	ABR <u>AC</u>
5	ABRAC <u>AD</u> ABRA	(1, D)	ABRAC <u>AD</u>
6	ABRACAD <u>AB</u> RA	(1, B)	ABRACAD <u>AB</u>
7	ABRACADAB <u>RA</u>	(3, A)	ABRACADAB <u>RA</u>

Final output: (0, A) (0, B) (0, R) (1, C) (1, D) (1, B) (3, A)

Figure 2.5: Example of compressing the word 'ABRACADABRA' using LZ78 coding.

First, the algorithm considers the first symbol (symbol 'A') and looks for a match in the dictionary. However, the dictionary is empty, and no match is found. Therefore, the first item of codification is zero (no pattern), and the specific symbol 'A'. The symbol 'A' is added to the dictionary with the index number 1.

Next, the algorithm considers the next symbol of the input data, the symbol 'B'. Again, the algorithm looks for a match of that symbol in the dictionary. However, no match is found again, and this new item of codification is zero (no pattern), and the symbol 'B'. The symbol 'B' is added to the dictionary with the index number 2.

A similar process is followed with the next symbol, 'R'. The symbol 'R' is also added to the dictionary with the index number 3.

The following symbol, 'A', is already in the dictionary, so the algorithm looks for the pattern 'AC' but does not find it. For that reason, this pattern is represented with index 1 (posterior decompression to symbol 'A') and symbol 'C'. The pattern 'AC' also becomes index 4.

Following this mechanism, the word is completely compressed. Notice that repeated patterns are progressively included in the dictionary with longer strings, given that longer patterns are built on top of shorter ones.

The decompression follows a similar strategy. Initially, the dictionary is empty, but as the decompression progresses, new indexes are included. Each found index is replaced using its pattern, and the symbol is put in afterward. It is important to remark that not having to attach the dictionary to the compressed output noticeably increases the overall compression ratio of the whole process.

2.1.6 *LZW* coding

Terry Welch introduced the *LZW* [119] (Lempel-Ziv-Welch) algorithm to enhance the previous *LZ78* algorithm. The algorithm begins with a dictionary containing all possible symbols from the input data. For example, if we are compressing ASCII text, the dictionary will initially include the ASCII characters (0-255). Furthermore, if we are compressing only lowercase letters from the alphabet, we can initialize the dictionary with 26 symbols ('a' to 'z'). This optimization can lead to more efficient compression, but it is not always possible. Once the initial dictionary is defined, the algorithm reads the input data one symbol at a time, forming longer sequences of symbols as it proceeds, and tries to find the longest sequence in that dictionary that matches the input. Once it finds the longest matching sequence in the dictionary, the algorithm codes it using the index used for that sequence. Next, the algorithm adds the next symbol to the current sequence and stores this new sequence in the dictionary in a new index. This mechanism allows *LZW* to learn patterns in the data as it processes more input, making it more efficient at compressing repeated sequences. Similarly to the *LZ78* algorithm, the dictionary can also be limited to reduce the maximum number of indexes and, consequently, the number of bits used to represent them. For example, indexes of 2 bytes can represent 65,536 different indexes.

Figure 2.6 presents an example where the *LZW* algorithm is used to compress the word 'ABRACADABRA'. Initially, the dictionary includes all the available single symbols (indexes from 0 to 4 locate symbols 'A', 'B', 'C', 'D', and 'R'). In the beginning, the symbol 'A' is detected, but the pattern 'AB' is not, so the symbol 'A' is coded with index 0, while the pattern 'AB' becomes index 5. After that, the symbol 'B' is detected, but again, the pattern 'BR' is not, so the symbol 'B' is coded with index 1, while the pattern 'BR' becomes index 6. The algorithm continues applying the same methodology until the rest of the word is compressed.

Codification: (index)

Index	Word	Encoded Output	Decoded Output
0	<u>A</u>	-	-
1	<u>B</u>	-	-
2	<u>C</u>	-	-
3	<u>D</u>	-	-
4	<u>R</u>	-	-
5	<u>A</u> BRACADABRA	0	A
6	A <u>B</u> BRACADABRA	1	AB
7	AB <u>R</u> ACADABRA	4	ABR
8	ABR <u>A</u> CADABRA	0	ABRA
9	ABRAC <u>C</u> ADABRA	2	ABRAC
10	ABRAC <u>A</u> DABRA	0	ABRACA
11	ABRACAD <u>D</u> ABRA	3	ABRACAD
12	ABRACAD <u>A</u> BR	5	ABRACADAB
13	ABRACADAB <u>R</u> A	7	ABRACADABRA

Final output: 0 1 4 0 2 0 3 5 7

Figure 2.6: Example of compressing the word 'ABRACADABRA' using LZW coding.

To decompress, the initial dictionary is required (in our example, indexes from 0 to 4), and it is completed with the new information gathered while the data is being decompressed.

2.1.7 Discussion about compression algorithms

All the described compression algorithms try to replace occurrences of some patterns using codes that can be encoded using fewer bits, thus reducing the overall data size. Nevertheless, there are significant differences among the described compression algorithms.

On the one hand, *Huffman* coding and *Arithmetic* coding focus on assigning optimal codes based on the statistical properties of the original data. So, they need to process the entire data before compressing it to obtain that information. As the process carried out by the *Huffman* and *Arithmetic* algorithms can be computationally expensive, they might be better suited for high-end processors than processors used in embedded systems. However, some compression libraries, such as the *Brotli* compression library [3], use a static table or create a table with just part of the data, which allows it to compress data more efficiently without the need to analyze the entire input first. By utilizing these static tables, compression libraries can skip the step of building a custom dictionary for each new dataset, which not only reduces computational overhead but also means that the dictionary does not need to be transmitted along with the compressed data, thus reducing the size of the compressed output, as no extra information about the coding scheme is required during decompression. As a result, these compression libraries achieve high compression ratios while maintaining fast compression and decompression speeds.

On the other hand, the *RLE*, *LZ77*, *LZ88*, and *LZW* algorithms focus on finding and representing repeated patterns. *RLE* is a straightforward algorithm that codes consecutive repeated symbols, while the rest of the algorithms use an adaptive dictionary that dynamically changes during the compression process. That means that two different data can have different dictionaries. In our experience, these libraries provide lower compression ratios at the expense of generating compressed data faster than the *Huffman* or *Arithmetic* coding. Thus, when deciding which compression algorithm to use, not only the achieved compression ratio must be considered but also the exact environment where that compression algorithm will be used.

Finally, note that compression libraries do not usually use only a single compression algorithm. On the contrary, they follow different strategies that combine them. An example could be the *Zstd* [71] library. This library is based on the *LZ77* algorithm and *Huffman* coding (among other strategies) to optimize its results. This compression library first applies the *LZ77* algorithm to identify and replace repetitive sequences in the data with references to previous occurrences. After the *LZ77* phase, the resulting data is passed through *Huffman* coding and other strategies. Using these algorithms, the compression library assigns shorter codes to more frequent bytes and longer codes to less frequent ones, further optimizing the data representation. Notice also that this explanation is a simplified example of how the *Zstd* compression process works. Thus, the library is more complex and integrates additional algorithms and techniques to enhance compression efficiency and speed.

2.2 Standard datasets

Compression libraries are evaluated based on their ability to compress and decompress data efficiently, typically focusing on compression ratio and speed metrics. However, the performance of these libraries can vary depending on the characteristics of the data being compressed [39]. To accurately assess the capabilities of different compression libraries, it is crucial to utilize appropriate datasets that reflect real-world scenarios.

Using the appropriate dataset to analyze performance is important because compression libraries may exhibit different behaviors when handling different data types. For example, some libraries might excel at compressing text data, while others may perform better with multimedia files or binary data. By using diverse datasets, researchers can evaluate the performance of compression libraries across various data types and identify their strengths and weaknesses. However, it is very important to use the proper dataset when focusing on a specific domain problem [117]. That is, depending on the intended use of the compression library, performance evaluation should use one dataset or another.

The evaluation of compression libraries involves measuring both compression and decompression capabilities. Compression ratio, which indicates the reduction in file size achieved by the library, is a standard metric used for comparison. It demonstrates the efficiency of the compression algorithm in reducing the data size. Similarly, compression and decompression speeds are also evaluated, as they determine how quickly the data can be compressed and subsequently decompressed, restoring the data to its original form. By assessing these factors with appropriate datasets, researchers can analyze the effectiveness of different compression libraries.

Previous studies have demonstrated that compression techniques can significantly reduce the data size, verifying the extent of data reduction achieved by different compression libraries when evaluated using available datasets [93, 120]. This helps to understand the practical implications of using these libraries for diverse applications, such as data storage or data transmission.

In this section, we present the most relevant compression datasets for compression library evaluation. They comprise a diverse collection of files representing different data types, including text, images, executable files, etc. These datasets are designed to simulate real-world scenarios and provide a standardized benchmark for comparing the performance of compression algorithms. For that reason, researchers can ensure consistent evaluation and

facilitate fair comparisons between different compression libraries when they use these datasets. These datasets are:

- **Calgary Corpus** [8, 7]. This dataset is widely recognized by researchers and practitioners in the data compression field, particularly for text compression. It includes 18 files (*bib*, *book1*, *book2*, *geo*, *news*, *obj1*, *obj2*, *paper1*, *paper2*, *paper3*, *paper4*, *paper5*, *paper6*, *pic*, *progc*, *progl*, *progp*, and *trans*).
- **Canterbury Corpus** [6, 5]. This dataset was developed as an enhanced version of the Calgary Corpus. It contains 11 different text files (*alice29.txt*, *asyoulik.txt*, *cp.html*, *fields.c*, *grammar.lsp*, *kennedy.xls*, *lcet10.txt*, *plrabn12.txt*, *ptt5*, *sum*, and *xargs.l*).
- **Wikipedia Corpus** [121]. This corpus contains *enwik8* and *enwik9* which represent the first 10^8 and 10^9 bytes of the English Wikipedia dump on March 3, 2006.
- **Protein Corpus** [79, 78]. This dataset contains four different genomes as data (*HI* for Haemophilus Influenzae, *HS* for Homo Sapiens, *MJ* for Methanococcus Jannaschii, and *SC* for Saccharomyces Cerevisiae).
- **Silesia Corpus** [23, 22]. The purpose of this dataset is to provide new data that was not covered by the Calgary Corpus and Canterbury Corpus, such as large data sets, text files in languages other than English, or medical images. There are 12 different files included in this dataset (*dickens*, *mozilla*, *mr*, *nci*, *ooffice*, *osdb*, *reymont*, *samba*, *sao*, *webster*, *x-ray*, and *xml*).
- **Snappy Corpus** [34]. The Snappy compression library includes some data to evaluate compression libraries in its repository, some of them picked up from the Canterbury Corpus. Concretely, it collects 14 different files (*alice29.txt*, *asyoulik.txt*, *baddata1.snappy*, *baddata2.snappy*, *baddata3.snappy*, *fireworks.jpeg*, *geo.protodata*, *html*, *html_x_4*, *kppkn.gtb*, *lcet10.txt*, *paper-100k.pdf*, and *plrabn12.txt*, *urls.10K*).

Due to the large amount of data available in the different datasets, in this thesis, we have selected some files of these datasets that could be interesting for our experiments. The selected files have been carefully chosen based on their data types and sizes to cover a broad range of scenarios. This selection ensures that we can thoroughly evaluate the performance of the proposed work across different use cases. By including diverse file types and varying sizes, we aim to simulate real-world conditions and ensure that the results are robust and applicable to a wide variety of data sets. Table 2.1 summarizes the data utilized in this thesis, which will be referred to as the standard dataset in the following chapters.

Name	Source	Description	Size
alice29.txt	Canterbury Corpus	English text	148.52 KB
asyoulik.txt	Canterbury Corpus	Shakespeare	122.25 KB
cp.html	Canterbury Corpus	HTML source	24.03 KB
fields.c	Canterbury Corpus	C source	10.89 KB
grammar.lsp	Canterbury Corpus	LISP source	3.63 KB
kennedy.xls	Canterbury Corpus	Excel Spreadsheet	1005.61 KB
lcet10.txt	Canterbury Corpus	Technical writing	416.75 KB
plrabn12.txt	Canterbury Corpus	Poetry	470.57 KB
ptt5	Canterbury Corpus	CCITT test set	501.19 KB
sum	Canterbury Corpus	SPARC Executable	37.34 KB
xargs.1	Canterbury Corpus	GNU manual page	4.13 KB
dickens	Silesia Corpus	Collected works of Charles Dickens	9.72 MB
mozilla	Silesia Corpus	Tarred executables of Mozilla 1.0	48.85 MB
mr	Silesia Corpus	Medical magnetic resonanse image	9.51 MB
nci	Silesia Corpus	Chemical database of structures	32.00 MB
ooffice	Silesia Corpus	A dll from Open Office.org 1.01	5.87 MB
osdb	Silesia Corpus	Sample database in MySQL format	9.62 MB
reymont	Silesia Corpus	Text from Chopi by W. Reymont	6.32 MB
samba	Silesia Corpus	Tarred source code of Samba 2-2.3	20.61 MB
sao	Silesia Corpus	The SAO star catalog	6.92 MB
webster	Silesia Corpus	1913 Webster Unabridged Dictionary	39.54 MB
x-ray	Silesia Corpus	X-ray medical picture	8.08 MB
xml	Silesia Corpus	Collected XML files	5.10 MB
enwik8	Wikipedia Corpus	First 10 ⁸ bytes of English Wikipedia	95.37 MB
fireworks.jpeg	Snappy Corpus	A JPEG image	120.21 KB
geo.protodata	Snappy Corpus	A set of Protocol Buffer data	115.81 KB
paper-100k.pdf	Snappy Corpus	A PDF	100.00 KB
urls.10K	Snappy Corpus	List of 10000 URLs	685.63 KB

Table 2.1: Data collected from different datasets that represent the standard dataset used in our experiments.

2.3 Remote GPU virtualization solutions

GPU computing has emerged as a solution for a wide range of computational tasks, significantly improving execution speed and energy efficiency. Many applications achieve significant execution time reduction by using the parallel processing power available on GPUs, particularly in fields like artificial intelligence. Besides, GPUs are an indispensable tool in optimizing performance and saving energy across different domains thanks to their

high computational processing capacity, which allows more energy-efficient execution if compared to traditional CPUs.

In general, GPUs are used in combination with the CPU. Thus, programmers identify and select computationally intensive software parts to run on the GPUs. These parts are often challenging to manage efficiently on CPUs, but they can effortlessly adapt to GPUs, benefiting from their parallel architecture. The programmer must therefore identify which parts of his software can benefit from running on the GPU. Note that there are tools and frameworks, such as the CUDA technology created by NVIDIA [82], which greatly simplify the process.

Currently, GPU computing has proven to provide significant improvements in many areas, highlighting AI, which takes huge profit from these powerful accelerators [2]. In these areas, there are different helpful tools that would have no real impact without GPUs, such as TensorFlow [1] or PyTorch [89]. These tools are typically programmed by leveraging the CUDA technology.

Similar to other devices, GPUs have been reliably virtualized. In this sense, NVIDIA has developed the vGPU [85] (virtual GPU) and the MIG [83] (Multi-Instance GPU) technologies. These technologies enable a GPU to be concurrently shared among multiple applications [80, 92]. On the one hand, vGPU technology creates virtual GPUs that can be shared across multiple virtual machines. On the other hand, MIG technology can partition the GPU into different instances, each fully isolated with its own high-bandwidth memory, cache, and compute cores.

Certainly, vGPU and MIG technologies present helpful solutions for sharing the GPU across different applications. However, these solutions can be restrictive in more complex scenarios. For instance, vGPU shares a GPU among virtual machines, but the GPU must be located in the same node, so it is limited to local systems. In a similar way, MIG allows multiple applications to run simultaneously on different instances of the GPU. However, the physical location of the GPU also limits this partitioning, and it is a problem when applications demand high memory or computation resources simultaneously. Also, due to the fact that these technologies work with local GPUs, applications cannot benefit from a distributed system composed of multiple nodes.

Consequently, the limitations described above promote the need for a remote GPU virtualization solution. This solution enables the access to GPUs across different nodes, even when these nodes do not have a physical GPU. Remote GPU virtualization provides a scalable and flexible architecture, enhancing GPU utilization and reducing hardware and energy costs by

sharing the GPU among applications that are not necessarily running on the same computer. This makes it ideal for large distributed systems.

In this way, a GPU application can run on a node without GPUs and use GPUs (or partitions of them) located in other computers. Thus, the GPU application transparently accesses the remote GPU as if it were available locally, without the application being aware of the physical location of the GPU.

Remote virtual GPUs offer several benefits. For example, in a cluster deployment, instead of installing GPUs in every node, they can be installed in a few number of nodes and shared across the rest of the system. In addition, GPUs often remain idle and underutilized while waiting for specific tasks to run. Even when they are not actively computing, GPUs still consume energy. Thus, its usage can be maximized while reducing the number of GPUs if multiple applications run simultaneously on the same GPU. Moreover, this approach has two advantages. Firstly, instead of having multiple GPUs, which would consume more energy even when idle, a single GPU can be shared among different applications, making more efficient use of its processing capabilities. The energy wasted by GPUs can therefore be significantly reduced by minimizing idle time and maximizing utilization. Secondly, since GPUs can be expensive, reducing the number of necessary GPUs by sharing them among multiple applications can lead to savings in acquisition cost. At the same time, reducing the consumption of energy also reduces overall expenses.

Remote GPU virtualization solutions can be used in many different scenarios. For instance, remote GPUs could be used by an AI application to detect whether people are wearing face masks. In this case, a small embedded hardware with a camera could take images and process them using TensorFlow, for instance, while using a remote GPU located in the cloud to increase the computing power of the embedded system.

It is important to notice that the performance of applications using remote GPU virtualization depends heavily on the network that connects the computer where the application is being executed and the remote server with the GPU. To ensure comparable performance to using local GPUs, the network throughput should be similar to that of the bus that connects the GPU to the system in a traditional local setup (e.g., PCIe).

In the context of the CUDA technology developed by NVIDIA, remote GPU virtualization allows a CUDA application being executed in a computer that does not have a physical GPU to access a GPU installed in a remote computer. This architecture follows a distributed client-server approach, as shown in Figure 2.7. The client part of the middleware is a library

that provides the same CUDA API, and it is installed on the computer where the application runs. The server side consists of a daemon running on the computer with the available GPU. Both client and server communicate over the network.

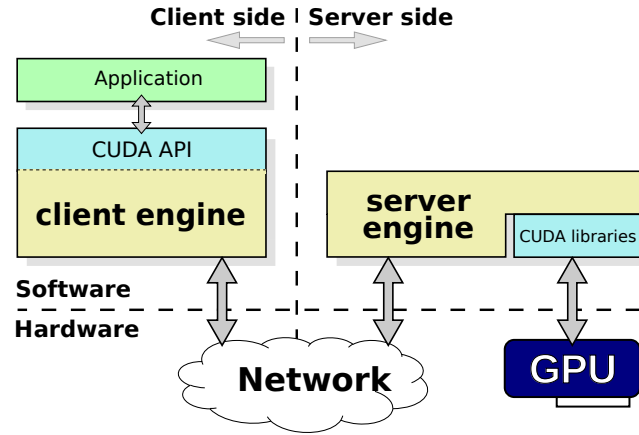


Figure 2.7: General architecture of remote GPU virtualization solutions.

The architecture depicted in Figure 2.7 is used in the following way: every time an application makes a CUDA request, the client middleware intercepts that request, appropriately processes it, and forwards it to the server middleware. On the server side, the daemon receives the request, interprets and delivers it to the GPU, which executes the request and returns the results to the server middleware. Finally, the server sends the results back to the client middleware, which forwards them to the application. Notice that GPU virtualization solutions transparently provide GPU services, and, therefore, applications are unaware that their requests are serviced by a remote virtual GPU instead of a local one. In this way, the source code of applications using remote GPU virtualization solutions does not need to be modified.

Several GPU virtualization environments have been developed over the past few years to enable efficient and scalable use of GPUs, such as DS-CUDA [86], GridCUDA [61], GViM [40], GVirtuS [31], rCUDA [104, 47], and vCUDA [103]. These solutions aim to virtualize the CUDA runtime API, which is a programming interface for GPU computing developed by NVIDIA. However, most of these existing solutions only provide partial support for outdated versions of CUDA, except for rCUDA and GVirtuS. This limitation makes these solutions less relevant for research and industry applications. Although GVirtuS supports a broader range of CUDA functions compared to the other solutions, it still supports a smaller set of functions compared to the complete CUDA runtime API. Therefore, for the purposes of the work being discussed in this thesis, the focus is specifically on rCUDA.

Using rCUDA is transparent to the application. Only a few configurations must be done before running the application. On the server side, an rCUDA daemon should be running. This

daemon enables remote access to the physical GPU. On the client side, a few environment variables must be set:

- **LD_LIBRARY_PATH.** This variable must point to the folder where the rCUDA binary is located. It is essential because it is the way to indicate that the remote GPU virtualization solution must be used.
- **RCUDA_DEVICE_COUNT.** This variable indicates the exact number of remote GPUs available that the application can access to.
- **RCUDA_DEVICE_[ID].** There must be as many variables of this type as remote GPUs available. 'ID' corresponds to a number that indicates the order used to differentiate GPUs. This variable indicates the specific location of the GPU, using the following notation: *IP:GPUID*, where *IP* refers to the IP address or name of the server, and *GPUID* corresponds to the CUDA identifier of the GPU.

```
export LD_LIBRARY_PATH=/some/path/to/rCUDA/library
export RCUDA_DEVICE_COUNT=2
export RCUDA_DEVICE_0=192.168.204.10:0
export RCUDA_DEVICE_1=192.168.204.11:3
```

Figure 2.8: Example of using rCUDA environment variables.

In Figure 2.8, we can observe an example of one possible initialization of rCUDA environment variables. Once these variables are set, the application can be run using the remote GPU virtualization solution rCUDA. For instance, when using the configuration shown in the figure, all the GPU computation is offloaded to the remote GPUs located in nodes *192.168.204.10* and *192.168.204.11*.

Chapter 3

A new dataset for AI applications

Year after year, the amount and size of data that need to be processed increases [24]. This has led to compression libraries playing an important role, resulting in a wide variety of compression libraries. These compression libraries have commonly been evaluated using the standard datasets presented in Section 2.2. As commented, those datasets include text, Excel files, programming sources, and images, among others. Different application domains require different datasets to evaluate those libraries [117]. However, AI datasets are not usually include in the tests. Furthermore, if we specifically consider data moved to/from the GPU, there are no AI datasets. Therefore, it would be interesting to test each of these available compression libraries on such datasets.

In this thesis, we focus on improving the communication of remote GPU virtualization systems using compression libraries. For that reason, the first contribution of this thesis is the development of a new dataset for AI applications. The following chapters will evaluate the compression libraries under study using this new dataset.

3.1 Applications selected to create the AI dataset

Different frameworks are available to work in machine learning and artificial intelligence areas, such as PyTorch [89] or TensorFlow (TF) [1]. We focused on the TF one, an open-source library developed by Google. Using this framework, users can build and train machine learning models. Also, multiple GPUs can be used to reduce the execution time by offloading to GPUs most of the computations. Several areas are currently covered by TF, such as Speech Recognition [49], Image Classification [44], and Text Classification [73], among others.

To create this new dataset, we have selected four different TF samples where the training and the inference of diverse Convolutional Neural Network (CNN) models are evaluated. These samples are described in the TF tutorials and are available at <https://git.dst.etit.tu-chemnitz.de/external/tf-models>:

- **Cifar10:** The image classification of the Cifar10 [54] dataset (Figure 3.1) is evaluated using a simple CNN. The Cifar10 dataset is a collection of 60,000 colored images with 32x32 pixels classified into ten classes. This test uses 50,000 images to train the convolutional network and the remaining 10,000 to evaluate the classification.

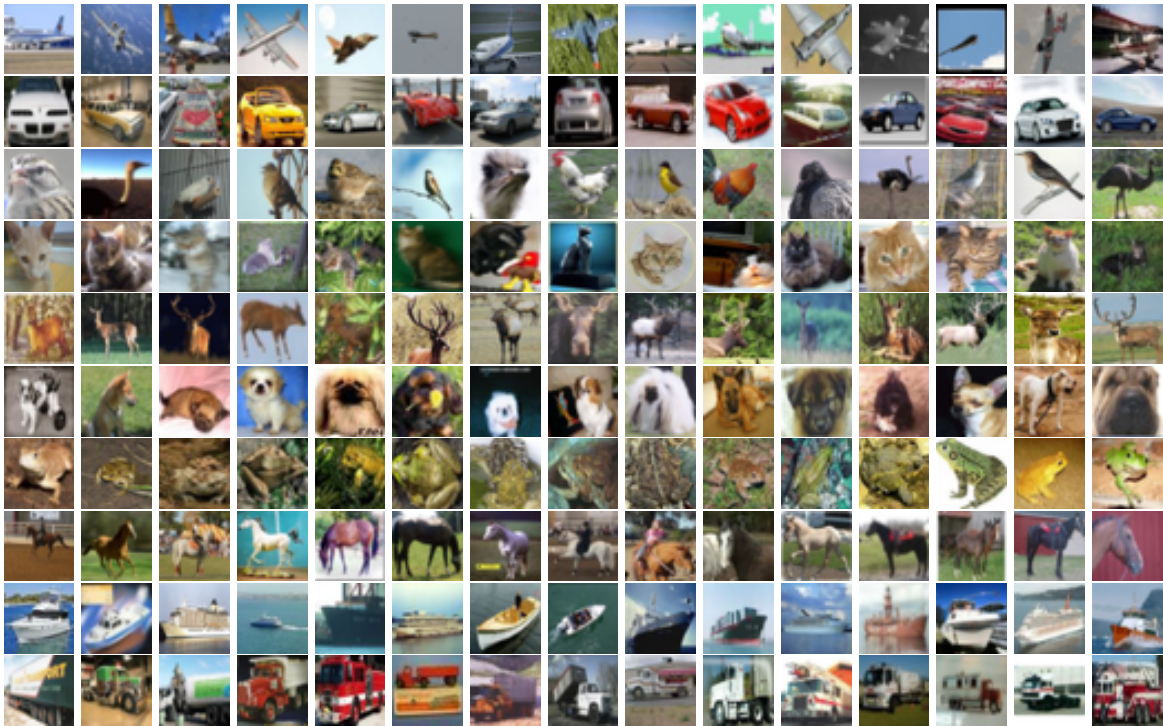


Figure 3.1: Sample of Cifar10 dataset.

- **Inception:** This test uses a CNN model widely used in image analysis and object detection called Inception-V3 [109]. It evaluates the classification of this model using the flowers [112] dataset (Figure 3.2). The flowers dataset is a collection of 3,670 colored images with different dimensions classified into five classes: daisies, dandelions, roses, sunflowers, and tulips. This test uses 2,934 images to train the model and the other 736 images to evaluate the classification.

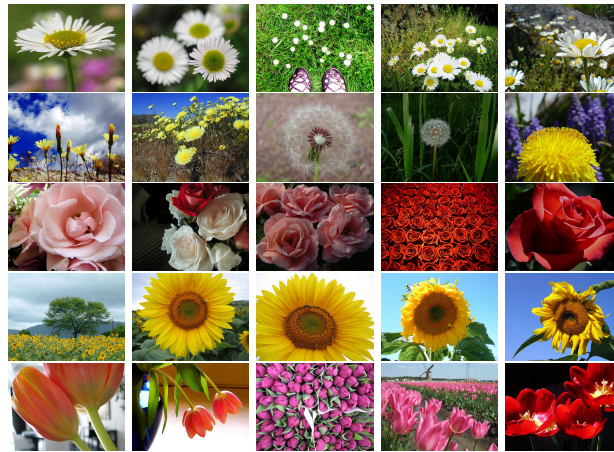


Figure 3.2: Sample of Flowers dataset.

- **Mnist:** This test is another image classification evaluation. The Mnist [57] dataset (Figure 3.3) is classified using a simple, end-to-end, LeNet-5-like CNN. This dataset contains 70,000 black-and-white images that represent handwritten digits in 28x28 pixels. This test uses 60,000 images to train the network and the remaining 10,000 to check the quality classification.

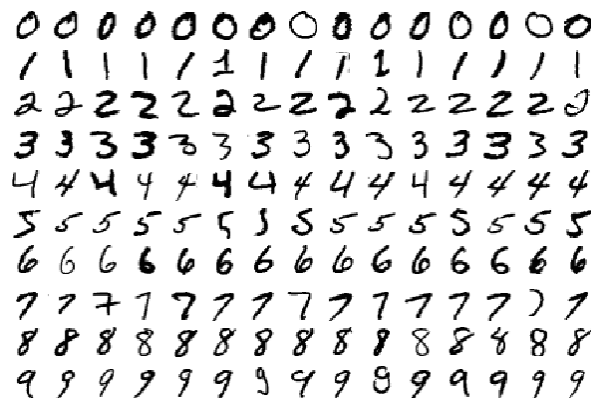


Figure 3.3: Sample of Mnist dataset.

- *Alexnet*: This test evaluates the inference time the pretrained Alexnet [55] CNN model takes using 800 grayscale images with 244x244 pixels, where each pixel has been randomly generated from a normal distribution (Figure 3.4).

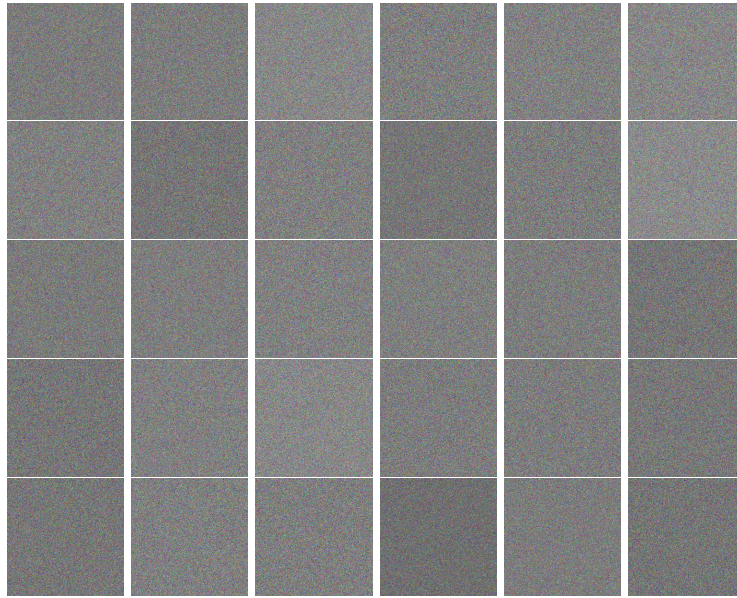


Figure 3.4: Sample of images used by Alexnet test.

We have used these tests to create a new dataset. The samples selected use different CNN and datasets. Thus, the information provided by each one is different. As a result, the new dataset is heterogeneous. The new dataset could be extended in the future by adding other tests.

3.2 Determining the appropriate dataset

Currently, a considerable number of datasets are being explored in many different pieces of research. The standard datasets, which have been presented in Section 2.2, have been highly explored using compression libraries. As we previously commented, these datasets contain many different kinds of files, such as Excel files, medical images, and binaries, among others. However, there is no AI data in these datasets because the research on compression began before the massive growth of AI that has occurred in recent years.

Table 3.1 illustrates how different compression libraries perform when compressing various data types. The data being compressed are those previously presented in Section 2.2. The

Library \ Data	alice29.txt	dickens	kennedy.xls	xml	x-ray	fireworks.jpeg
zstd [71]	2.55	2.39	8.92	7.67	2.23	1.00
z3lib [100]	2.75	2.58	5.02	7.58	1.42	1.00
snappy [35]	1.73	1.61	2.43	4.08	1.00	1.00
lzma [21]	3.14	3.60	20.96	11.79	1.89	1.00
lzham [29]	2.43	2.34	11.06	6.53	1.81	1.00
lzfse [4]	2.70	2.64	7.56	7.82	1.42	0.99
lz4 [19]	1.71	1.59	2.75	4.35	1.01	1.00
lizard [106]	1.71	1.58	2.75	4.31	1.00	1.00
libbse [36]	3.67	4.44	7.20	10.60	1.25	1.00
gipfeli [33]	2.15	2.06	2.75	4.86	1.11	1.00
bzip2 [107]	3.31	3.09	8.37	9.42	1.96	1.00
brotli [32]	1.61	1.58	3.41	2.21	1.12	1.00
brieflz [45]	2.26	2.22	3.24	4.96	1.16	0.89
Average	2.44	2.44	6.65	6.63	1.41	0.99

Table 3.1: The compression ratio achieved by different compression libraries when compressing different kinds of data.

files ‘alice29.txt’ and ‘dickens’ are text files, ‘kennedy.xls’ and ‘xml’ are XML files, and finally, ‘x-ray’ and ‘fireworks.jpeg’ are images. The table evaluates compression libraries using their compression ratio, which is essentially the original data size divided by the compressed data size, as shown in Equation 3.1. Therefore, the higher the ratio, the better the compression. Conversely, if the ratio is less than 1, the compressed data is larger than the original data.

$$\text{Compression Ratio} = \frac{\text{Size}_{\text{original}}}{\text{Size}_{\text{compressed}}} \quad (3.1)$$

As we can see in Table 3.1, the most effective compression is done with XML files, where the average compression ratio achieved by the libraries is between 6.63 and 6.65. This ratio means the data is reduced to less than one-sixth of its original size. The library with the best compression ratio achieved in this case is *lzma* which reaches compression ratios greater than 11. When using text data, libraries also provide good compression ratios, with an average of 2.44 (more than half of the original size). This time, *libbse* is the library that performs best for this kind of data. Finally, the worst compression ratios are obtained with image data. This data present an average between 0.99 and 1.41, *zstd* providing the best results.

Consequently, the compression ratio depends heavily on the type of data to be compressed and the compression library used. Thus, each compression library has different behavior depending on the data type. Therefore, creating a specific dataset for the area where the compression is to be applied is necessary to properly evaluate the performance of different compression libraries.

In the field of AI, there are different datasets, such as Cifar10 or Mnist, which were previously discussed in Section 3.1. Despite being datasets used in the area, they are very similar to the standard ones (text files, medical images, etc). However, the main objective of this thesis is to improve bandwidth in low-performance networks through the usage of data compression when AI applications are running. Thus, it is essential to identify which data-flow is sent from/to GPU during the execution of these applications. Actually, this data-flow will be the best dataset for evaluating compression libraries in the research domain of this thesis.

In summary, table 3.1 demonstrates that the impact of using different datasets during the compression libraries evaluation significantly varies depending on the executed dataset and the compression library. For this reason, we decided to create a new AI dataset composed of data transfers from/to GPU when the AI application runs. This new dataset is key for evaluating the performance of AI applications and the efficiency of data transfer and compression in remote GPU virtualization solutions. This approach allows us to study the initial transfer of data and the multiple exchanges that occur as AI models train, optimize, and evaluate neural networks. Note that it is not possible to capture this behavior using traditional static datasets, making this new dataset fundamental.

3.3 Dataset creation from rCUDA, a remote GPU virtualization system

As commented, our main objective is to improve the communications of remote GPU virtualization for AI applications. With this goal in mind, we have run the TF applications introduced in Section 3.1 using the remote GPU virtualization middleware rCUDA presented in Section 2.3. For this purpose, rCUDA has been slightly modified to gather the data moved to/from the remote GPU. That modification is necessary to create the new AI dataset.

Usually, remote GPU virtualization systems follow a client-server architecture. On the client side, the main processes send to the server side all the information to be processed by the GPU. On the server side, once data is processed, the result is copied back to the client side.

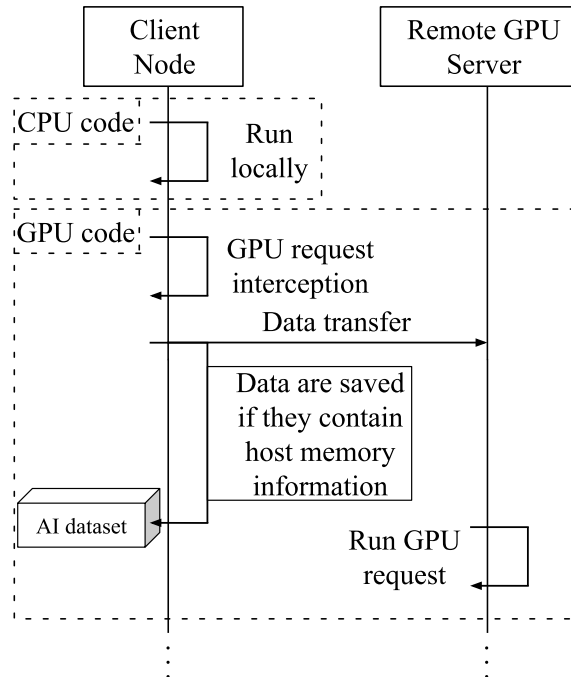


Figure 3.5: A diagram of how the data from the AI dataset was obtained.

To create the AI dataset, we have stored all the information exchanged between the client and the server side when running TF applications. Figure 3.5 illustrates a diagram where we can observe this modification on the client side of the remote GPU virtualization system.

On the one hand, the CPU code is run on the client side as a typical application. Data used by the CPU code is not interesting for our new dataset because that data is not forwarded to the network, and it will not be compressed. Our interest focuses on the network traffic between the client and the server nodes. On the other hand, the GPU code is intercepted by the remote GPU virtualization system and transferred to the remote GPU server. This functionality is one of the fundamental aspects of these systems. Once the GPU request is transferred to the remote GPU server, we have added a new code to rCUDA in order to check if that request contains host memory information (data used by the application). This information we referred to is more complex than simple parameters. It is a memory region reserved in the host, not an integer or a float. Therefore, when running the TF applications, we detected all the requests transferred done to the remote GPU server containing this kind of data, and we copied them in a new file to create the new AI dataset. In summary, the new AI dataset will contain all data exchanged between the TF application and the remote GPU. That data will be the same as if the TF application were executed without a remote GPU virtualization system.

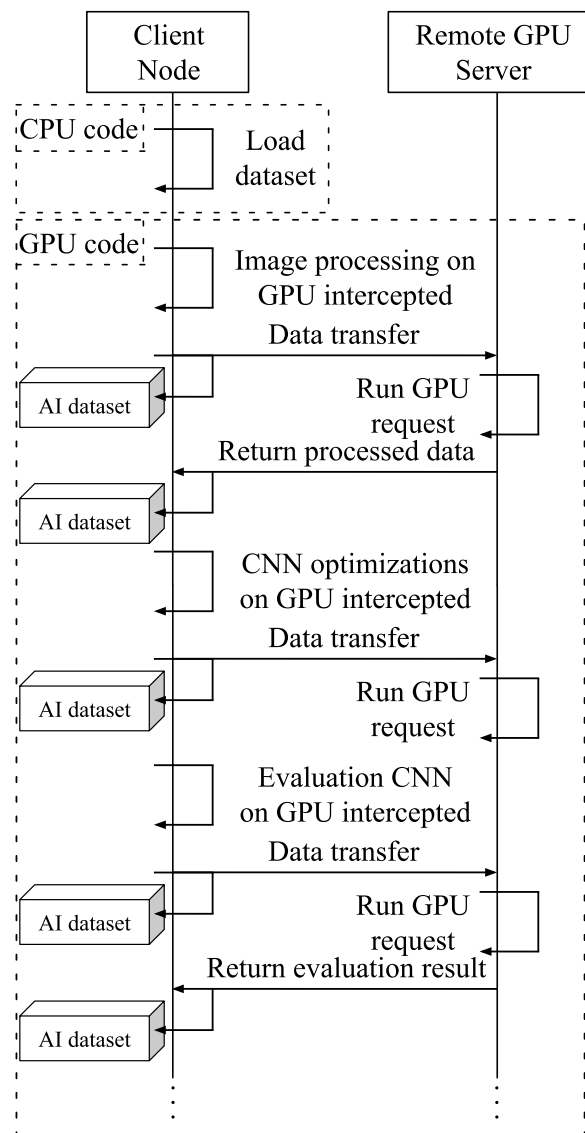


Figure 3.6: A diagram illustrating where data from the AI dataset are obtained from.

It is important to note that data moved to/from the GPU when an application is run is not necessarily the application input. To illustrate this, an example with the *Cifar10* test is provided. The input data for this test consists of 60,000 32x32 color images categorized into ten classes. These images are read during the execution of the test. When an image is read, it is appropriately transformed, and the result is inserted into the CNN utilized. During the insertion process into the CNN, multiple copies of the data are transferred to the GPU. After processing in the GPU, the data is copied back to the main memory. Many of these transfers to/from the GPU consist of internal data of the application, which is well different than the 32x32 color images used as input to the application.

In particular, these data moved to/from the GPU in these copies is the data that will be used to create the AI dataset. Figure 3.6 shows an example for creating this dataset from a specific application. First, the application loads the dataset that it will use. Then, it processes this dataset on the GPU. This processing creates several data transfers to/from the remote server. Once the dataset has been processed, the application optimizes the CNN used on the GPU. This optimization also creates several data transfers between the remote server and the client node. Finally, the CNN is evaluated by transferring, one more time, different data between client and server nodes. All these data transfers to/from the remote GPU carried out during the application lifetime are the ones that make up the new AI dataset. Thus, our modification intercepts and saves the information to create the new AI dataset.

It is worth mentioning that users do not have direct access to the data being moved to/from the GPU. This is because these data are part of the internal workings of the execution of AI applications. Furthermore, data in these copies are only available during a specific portion of the execution of AI applications. Before the application begins its execution, these data do not exist. Similarly, once the AI application completes its execution, these data are no longer accessible. Moreover, if the application returns to the operating system a memory area previously allocated, data will disappear at that exact moment.

To better clarify the nature of the data in the new dataset, it is essential to highlight that not all data exchanged with the remote GPU are application inputs or final outputs. Instead, the dataset comprises low-level memory buffers that contain intermediate representations, weights, activation maps, gradients, and other internal data structures exchanged between the client and the remote GPU while executing AI workloads. These data blocks are typically opaque to the user and are generated dynamically as the application progresses through various stages such as data preprocessing, forward and backward propagation, model optimization, and evaluation. Consequently, the dataset captures a rich and diverse set of GPU-related memory exchanges that reflect the actual internal behavior of AI applications in

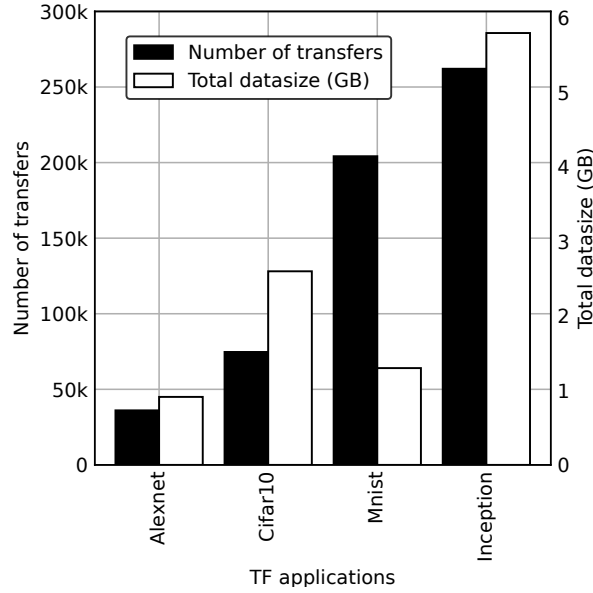


Figure 3.7: Histogram of the data transfers when running TF applications under study. The primary Y-axis represents the number of transfers done by TF applications, while the secondary Y-axis represents the total size of data exchanged in GB.

remote GPU environments, going beyond simple input/output capture and providing valuable insights into runtime data movement patterns.

Using the modified rCUDA version, we have intercepted and analyzed the data exchanged with the remote GPU during the execution of the four TF samples explained in the previous section, namely: *Alexnet*, *Cifar10*, *Mnist*, and *Inception*. We recorded the number of data transfers for each sample between the local host and the remote GPU. The total volume of data exchanged ascends to 10.6 GB and is distributed in 577,033 data transfers. Specifically, the execution of *Alexnet* involved 36,073 transfers, accounting for a total of 935 MB of data exchanged. In the case of *Cifar10*, we observed 74,698 transfers, with a total of 2.6 GB of data being transmitted. Similarly, the *Mnist* sample generated 204,194 transfers, with a total data volume of 1.3 GB, while the *Inception* model had the highest traffic, with 262,068 transfers and 5.8 GB of data exchanged. Figure 3.7 summarizes these results. The histogram illustrates the data transfers when running the TF applications under study. The primary Y-axis represents the number of transfers done by these applications and the secondary Y-axis represents the total size of data exchanged in GB.

This detailed analysis of the data transfers illustrates the varying demands of the different neural network models regarding data movement between the local and remote environments. Thus, the more significant number of transfers and higher data volume observed with models

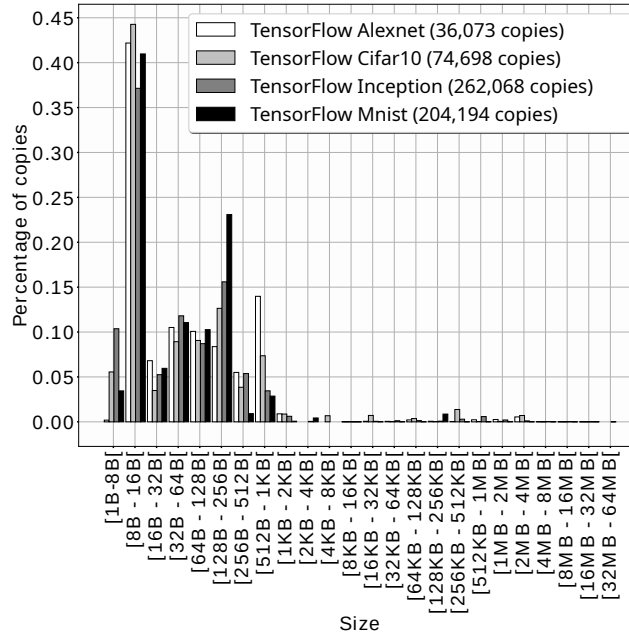


Figure 3.8: Histogram of the data transfers when running TF applications under study. Notice that the Y-axis represents the frequency of data transfers with a specific size range performed by each TF application.

like Inception suggest that complex architectures could require greater bandwidth, which would introduce more overhead in the remote execution environments focus of this thesis.

Due to the high number of data transfers observed during the execution of these samples, we decided to group the transfers into different ranges of transfer sizes, specifically into 24 ranges. Figure 3.8 illustrates a histogram of data transfers done by each TF application. The X-axis represents different ranges of sizes used by data transfers, whereas the Y-axis illustrates the frequency of data transfers with a specific size range instead of absolute numbers. As it can be observed, each TF application performs a different number of data transfers. This shows that the frequency is more beneficial than absolute numbers to compare the four applications more clearly. We can also see how the size of ranges increases exponentially. Thus, the first range contains all the data transfers with a size up to 8 B. Next ranges increment size twice compare to the previous one size. Then, the second range is from 8 B to 16 B, the third range is from 16 B to 32 B, and it continues until the last range with data transfers with a size from 32 MB to 64 MB. This way of grouping helps to manage the data complexity better and identify repetitive ones.

Once these data transfers were analyzed, we created the AI datasets using the most representative data transfers for each TF sample. The goal was to create datasets that accurately

Name
AI-5KB
AI-10KB
AI-25KB
AI-50KB
AI-100KB
AI-250KB
AI-500KB
AI-1000KB
AI-2500KB
AI-5MB
AI-10MB
AI-25MB

Table 3.2: Name of files that make up in the new AI dataset created to evaluate compression libraries. The name of each file indicates its size.

Library	Compression ratio of AI dataset
zstd	1.81
z3lib	1.79
snappy	1.51
lzma	2.04
lzham	1.77
lzfse	1.79
lz4	1.59
lizard	1.56
libbsc	1.79
gipfeli	1.59
bzip2	1.64
brotli	1.28
brieflz	1.58
Average	1.67

Table 3.3: Compression ratios achieved by compression libraries when compressing the new AI dataset.

represent real-world scenarios by reflecting the variety and volume of data exchanged during the execution of TF applications. First, we sorted the data transfers within each range based on their frequency, prioritizing the most repetitive ones and allowing us to identify the more critical patterns. Then, we selected the most repetitive data transfers from the different ranges to ensure that transfers were well-represented, capturing data with different sizes. By doing this, we guarantee that the dataset reflects the diversity of data transfers of all the TF samples. Table 3.2 shows files included in this AI dataset, which is available at <https://github.com/cpenaranda/AI-dataset>. The name of each file indicates its size, ranging from 5 KB to 25 MB. It should be noted that the dataset contains only the most representative transfers.

Using the previously presented Equation 3.1, Table 3.3 shows the average compression ratio achieved when different compression libraries compress the AI dataset presented in Table 3.2. The compression libraries used are the same as those used in Table 3.1 for evaluating compression ratios in standard datasets. As we can see, the compression library that achieves the best performance is *lzma*, which is also the best when XML files are compressed. However, we can observe that the average compression ratio achieved by all libraries is 1.67, which is better than the one achieved when using ‘fireworks.jpeg’ or ‘x-ray’, but it is worse than the 6.6 achieved for XML files or than the 2.44 achieved for text files. These results highlight how difficult it is to compress this kind of data and the relevance of having a specific dataset.

$$Entropy = - \sum_{i=1}^n p_i \log_2(p_i) \quad (3.2)$$

To finalize the creation of the new AI dataset, we analyze the entropy. In data compression, entropy can be helpful to characterize data. Equation 3.2 illustrates how the entropy is calculated. Each symbol probability of occurring on the data is multiplied by the base-2 logarithm of that probability. This result is summed across all symbols. The logarithm produces a negative value since probabilities are always less than 1. The negative sign in the formula ensures that the final entropy value is positive. Entropy helps us measure the randomness of the data. Thus, higher values of entropy indicate more randomness, while lower values denote redundancy. Consequently, higher entropy typically makes compression more difficult.

Data	Entropy
alice29.txt	4.57
dickens	4.53
kennedy.xls	3.57
XML	5.52
x-ray	6.60
fireworks.jpeg	7.97
AI dataset	6.90

Table 3.4: Entropy values of various datasets used in compression tests.

As shown in Table 3.4, the text files (i.e., ‘alice29.txt’ and ‘dickens’) have low entropy. Thus, compression is more effective in such cases because of the repetitive nature of their content. Although the XML files obtained the best compression ratio, we can observe something interesting if we analyze their entropy. In this way, ‘kennedy.xls’ shows the lowest entropy, but this is not the case of ‘XML’. In spite of this, both obtained the best compression ratios. This low entropy of ‘XML’ is due to the fact that it is a collection of XML files. In contrast, datasets like ‘fireworks.jpeg’ and ‘x-ray’, which have high entropy, are also more complex to compress because of their randomness. Finally, the AI dataset entropy is 6.90, which is higher than that of XML and text data, but slightly lower than that of ‘fireworks.jpeg’. This high entropy reflects the nature of AI workloads, where data transferred to and from the remote GPU is less redundant and, therefore, more complex to compress.

3.4 Summary

Based on the results obtained when compressing standard datasets and the entropy analysis, we have decided to create a new dataset that reflects the challenges of compression in AI workloads. The relatively high entropy of the AI dataset highlights the unique characteristics of the data being transferred between the AI application and the remote GPU in remote GPU virtualization environments. This dataset uses real-world data to evaluate compression libraries more effectively within remote GPU virtualization solutions.

To create the new AI dataset, the remote GPU virtualization system rCUDA has been modified to intercept data moved to/from the remote GPU during the execution of four popular TF applications. Remote GPU virtualization systems typically follow a client-server architecture,

where the client sends information to the server for GPU processing, and the processed results are sent back to the client. In this context, data moved to/from the GPU during application execution are not necessarily the application input itself. Besides, users do not have direct access to the data moved to/from the GPU, as it is internal to the execution of AI applications. Thus, data in these copies are only available during specific parts of the execution of applications and do not exist before the program starts or after it completes. For these reasons, this modification was key in the creation of the new AI dataset.

In particular, we have modified rCUDA to intercept the data exchanged with the remote GPU during the execution of four TF samples (i.e., *Alexnet*, *Cifar10*, *Mnist*, and *Inception*). The new AI dataset was created using representative parts of the intercepted data. The files included in this new AI dataset are available at <https://github.com/cpenaranda/AI-dataset>.

Chapter 4

CPU-based compression libraries

Compression libraries are an excellent approach to reduce the volume of data, as it is the case of studies carried out by Vega et al. [116] and Hansson and Karlsson [41]. The first study uses delta compression techniques in fusion with databases to save storage and increase bandwidth, whereas the last study utilizes the well-known compression libraries *Lzo*, *Lzfx*, *Lzw*, *Lzma*, *Bzip2*, and *Lz4* to support lossless message compression. Both studies obtain promising results. Thanks to the promising results obtained by compression libraries, they are used in many research areas. As a consequence, an enormous number of compression libraries have been created. However, one of the problems that we have observed and intend to address in this chapter is that these compression libraries do not follow any usage standards.

As discussed in previous chapters, compression libraries are built based on the compression algorithms discussed in Section 2.1 and employ different strategies to achieve their results. Despite all aiming to solve the same problem, each one utilizes different methods and parameters to optimize their results. This complicates the utilization of the libraries for users, since they have to constantly modify their code to run each new compression library. In addition, the lack of information on how to use the libraries further increases complexity. Thus, to carry out the experiments with the different compression libraries under study, we need an appropriate tool for easily comparing their behaviors using specific datasets.

To address this, in this chapter we introduce a new compression abstraction library and benchmark called *CPU-Smash*, available at <https://github.com/cpenaranda/cpu-smash>. It allows users to deploy different compression libraries with minor code changes. It is also possible to benchmark the compression libraries using the standard datasets described in Section 2.2, and also the new AI dataset presented in Chapter 3.

4.1 *CPU-Smash*

Compression libraries have proven to be an outstanding method for reducing data volume, achieving significant results in many areas where reducing the volume of data is crucial. These promising results have led to the use of these libraries in a wide range of research areas, both in academia and industry. As a result, the number of compression libraries developed has increased, each one offering different features and benefits.

These libraries follow diverse strategies, using various algorithms, parameter configurations, and optimization techniques to achieve optimal results. This diversity of approaches allows users to adapt compression libraries depending on the kind of data to be compressed. However, this diversity also adds significant challenges due to a notable lack of standardization. As a result, users are forced to make noticeable changes in their code every time they want to test different compression libraries. Moreover, some libraries provide limited or no documentation for practical use.

For that reason, comparing different compression libraries is a time-consuming task. There is a clear need for an appropriate tool that simplifies the use of these compression libraries for both, evaluating and using them. This tool would allow users to effortlessly test different libraries on specific datasets without requiring extensive code modifications, facilitating experimentation.

Our goal in this thesis is to compare several compression libraries in the context of remote GPU virtualization. To do this, we need a compression benchmark that enables us to objectively assess the behavior of the various compression libraries, making it possible to identify the most effective. In this chapter, we focus on CPU-based compression libraries because remote GPU virtualization frameworks typically do not have GPUs on the client side. With this in mind, a good benchmark should include the following features:

- **Supported libraries.** The benchmark must implement a significant number of CPU-based compression libraries. There are a large number of compression libraries, each with strategies and optimization techniques that make them unique. Therefore, no single library is universally the best for all datasets. Thus, the more compression libraries the benchmark includes, the more valuable it becomes for identifying the library that performs better for a specific scenario.

- **Evaluation metrics.** The benchmark must include different evaluation metrics, such as compression ratio or compression and decompression speed. These metrics are essential for comparing the performance of each compression library.
- **Data in RAM.** Most compression libraries only allow users to compress and decompress files. However, we plan to use them in real applications with dynamic data. Thus, the available compression libraries must work with data located in RAM memory.
- **Lossless compression.** The benchmark needs to focus on lossless compression libraries. As discussed in previous chapters, lossy compression is not contemplated in this thesis.
- **Unified interface.** As our goal is to use compression libraries in real applications, it is also important to have an abstraction library that allows users to incorporate the different compression libraries using the same API. Abstraction libraries are beneficial because they provide a higher-level interface to simplify the use of more complex systems. In this sense, we could adapt our code to employ different compression libraries with minor changes.

After an exhaustive review of different compression benchmarks and discarding those that work with lossy compression libraries or those that do not work with data allocated in RAM (i.e., only compress files), we found two promising benchmarks:

- *Squash* [77] contains 33 compression libraries and facilitates an abstraction library that allows us to adapt our code. Despite all that, this compression benchmark has not been maintained since 2018. Its last release was in 2015. Moreover, we have not been able to make it work.
- *Lzbench* [75] contains 40 compression libraries. Its last release was in 2020 and it has not been maintained since 2023. In addition, this benchmark does not include an abstraction library.

Given that the available compression benchmarks do not have all the features required, we created the compression abstraction library and benchmark *CPU-Smash*, available at <https://github.com/cripeace/cpu-smash>. It includes 41 different CPU-based compression libraries. The main features of the *CPU-Smash* benchmark are: (i) it is possible to run multiple times the same libraries with specific data showing the standard deviation to obtain

more statistically significant results, (ii) different parameters are available to configure and evaluate the compression libraries, and (iii) results can be sorted either by compression ratio, compression/decompression speed or total speed.

4.1.1 Available configurations

Each compression library supports different parameters to compress and decompress data. These parameters can enhance the compression results. Thus, keeping them is essential. *CPU-Smash* supports all the parameters of the compression libraries that it includes. Note that the values and the implementation of these parameters depend on the compression library to be used. Therefore, compression libraries may not use some parameters or none at all. They may also use the same parameters as other libraries. Next, we present a list of the different parameters supported by *CPU-Smash*:

- **Compression level.** This parameter allows for controlling the quality and speed of the compression data. It is only available during data compression and can be ignored during decompression. The available values depend on the specific library. Compression libraries use higher values to achieve a better compression ratio. However, compression libraries usually run faster using lower values.
- **Window size.** Some compression libraries use previously compressed data to compress new data. They do this by referencing previous data in what is known as a ‘sliding window’. Using this parameter, we can define the number of bits used for the size of this window. For instance, a value of 8 means that the window size is 256 bytes (2^8). I.e., data can be referenced until the most recent 256 bytes of the previously compressed data.
- **Back reference.** Some compression libraries, which use a sliding window to identify repetitive patterns, can also configure the maximum size of the pattern. Using this parameter, we can define the number of bits used to determine the maximum pattern size. For example, a value of 4 means the maximum pattern size is 16 bytes (2^4). This means that the library can search for repeated patterns in the sliding window up to this size. A more significant value allows the algorithm to detect longer patterns, potentially improving compression efficiency but increasing the compression time. Note that this value must be lower than the value set in the window size parameter because setting a maximum size higher than the size of the sliding window would not make sense.

- **Work factor.** This parameter controls the compression behavior when there are highly repetitive data. Thus, it is only available during data compression. If the compression process runs into difficulties caused by repetitive data, the library switches from the standard algorithm to a fallback algorithm. Lower values of this parameter reduce the effort that the standard algorithm will do before switching to the fallback one.
- **Number of threads.** The number of threads used by the compression library. Note that this does not mean that all libraries can run using multiple threads. It is only available for libraries that support this configuration.
- **Mode.** Compression libraries could implement different compression/decompression modes to optimize their results. Each mode uses different combinations of compression algorithms to compress and decompress data. In the following section we explain the modes available for each compression library.
- **Flags.** Flags are used to control the strategy followed by the compression library. Similar to the previous parameter, in the following section we explain the flags available for each compression library.

4.1.2 Available compression libraries

As we introduced before, *CPU-Smash* comprises a total of 41 compression libraries. Each library has its own set of configuration parameters to control various aspects of the compression process. These parameters directly affect the compression performance, such as compression speed, compression ratio, and memory usage. Next, a detailed list of these libraries is provided, along with a brief description of each one. Additionally, the specific parameters that are available for configuring each library are presented:

1. *Brieflz* [45] is a small and fast open-source Lempel-Ziv style compression algorithm implementation. The main focus is speed and code footprint, but the ratios achieved are quite good compared to similar algorithms. It supports the following configuration parameter:
 - **Compression level.** The available values range from 1 to 10.
2. *Brotli* [32] is a generic-purpose lossless compression library that compresses data using a combination of a modern variant of the *LZ77* algorithm, *Huffman* coding, and 2nd-order context modeling. It supports the following configuration parameters:

- **Compression level.** The available values range from 0 to 11.
 - **Window size.** The available values range from 10 to 24.
 - **Mode.** This parameter is only available during the compression process. There are three different modes available: (i) the generic mode used when the properties of the input are unknown, (ii) the text mode used when the input is UTF-8 formatted text, and (iii) the WOFF (Web Open Font Format) mode used when the input data contains fonts that have been compressed to be embedded in a Web page.
3. *Libbsc* [36] is a compression library that uses different strategies to process the data before compression. After applying these strategies, the compression library applies dictionary coding based on the *LZ77* algorithm. It supports the following configuration parameters:
- **Compression level.** The available values range from 1 to 3.
 - **Window size.** The available values range from 10 to 28.
 - **Back reference.** The available values range from 3 to 8.
 - **Mode.** This compression library offers different block sorters to apply before the compression: (i) the *Burrows-Wheeler Transform*, which organizes the data forming repetitive patterns; (ii) Sort Transform n (where n can take values from 3 to 8), which sorts the data to make them more predictable.
 - **Flags.** There are three different available features: (i) fast mode, where the compression library prioritizes speed over compression ratio; (ii) multithreading mode, which enables multiple threads to run the compression/decompression process; and (iii) large pages feature, which reduces the overhead of memory management improving the efficiency.
4. *Bzip2* [107] is an open-source compression library that compresses and decompresses data using the *Burrows-Wheeler Transform* and *Huffman* codification. It offers better compression ratios than other libraries based on *LZ77* and *LZ78* algorithms, but utilizes more memory and takes more time. It supports the following configuration parameters:
- **Compression level.** The available values range from 1 to 9.
 - **Work factor.** The available values range from 0 to 25.
 - **Mode.** This parameter is only available when decompressing data. There are two options available: (i) fast decompression mode and (ii) slow decompression

mode. The fast mode improves the decompression speed by incrementing the memory usage.

5. *C-blosc2* [110] is a compressor oriented to binary data. It has been designed to transmit data to the processor cache rapidly. Its main goal is to reduce the size of large datasets and accelerate memory-bound computations. It supports the following configuration parameters:

- **Compression level.** The available values range from 0 to 9.
- **Number of threads.** This compression library allows us to use multiple threads to compress and decompress different chunks of data simultaneously. The available values for this parameter range from 1 to 8.
- **Flags.** Using this parameter, the compression library can be configured to use two different types of filters before compression to improve it: (i) reorganize the data at the byte level and (ii) reorganize the data at the bit level. Reorganizing the data at the bit level is slower than at the byte level, but it can potentially achieve better compression.

6. *Csc* [27] is a compression library based on *LZ77* Sliding Window and Range Coder. It supports the following configuration parameters:

- **Compression level.** The available values range from 1 to 5.
- **Window size.** The available values range from 15 to 29.
- **Flags.** This parameter allows users to configure three different types of filters to improve the compression: (i) a delta filter that compresses data by storing the differences between successive values, enabling a more efficient representation of sequential or time-series data; (ii) a binary filter that enhances compression efficiency for binary data by reducing redundancy and exploiting the structure of binary files; and (iii) a text filter that optimizes compression for textual data by leveraging patterns and redundancies inherent in text formats. Note that more than one filter can be applied simultaneously.

7. *Density* [38] is also based on the *LZ77* algorithm. However, while other libraries consume one byte of data and then apply algorithmic processing, *Density* consumes 4 bytes and then applies its algorithmic processing. It supports the following configuration parameter:

- **Mode.** This library has three different algorithms to enhance the compression process: (i) the Chameleon algorithm, which uses a dictionary to find more frequent patterns; (ii) the Cheetah, which derives from the chameleon one and uses swapped double dictionary lookups and predictions; and (iii) the Lion algorithm, which uses multiple swapped dictionary lookups and predictions, and forms rank entropy coding.
8. *Flz* [42] is another compression library based on *LZ77*. Unlike other compression libraries, this library focuses on the compression/decompression speed instead of the compression ratio. It is a lighter library, thus, it is ideal for embedded systems where resources are limited. It supports the following configuration parameter:
- **Compression level.** The available values range from 1 to 2.
9. *Flzma2* [69] is a lossless high-ratio data compression library based on Igor Pavlov's LZMA2 codec from 7-zip. The library uses a parallel buffered radix match-finder and some optimizations to increase the speed over the default LZMA2 algorithm used in 7-zip for a slight loss in compression ratio. It supports the following configuration parameters:
- **Compression level.** The available values range from 1 to 10.
 - **Number of threads.** This library allows multi-threaded execution to parallelize the match-finding process, a crucial component. This parallelization allows the compression and decompression process to be done faster without duplicating memory-intensive data structures. The available values for this parameter range from 1 to 8.
10. *Fse* [18] (Finite State Entropy) proposes two high speed entropy coders. It supports the following configuration parameter:
- **Mode.** There are two available algorithms: (i) the *Fse* algorithm, which provides efficient compression by leveraging finite state machines to encode symbols based on their frequencies; and (ii) the *Huffman* coding method, which uses variable-length codes assigned to input symbols, optimizing the representation of more frequent symbols with shorter codes.
11. *Zlib* [48] is a general-purpose data compression library that, using the *Deflate* format, combines the *LZ77* algorithm and *Huffman* codification to achieve a fast and efficient compression. It supports the following configuration parameter:

- **Compression level.** The available values range from 0 to 9.
12. *Libdeflate* [9] is heavily optimized as a fast, whole-buffer *Deflate*-based compression and decompression library. It supports the following configuration parameters:
- **Compression level.** The available values range from 0 to 12.
 - **Mode.** This parameter allows us to configure three different formats: (i) the *Deflate* raw format, a widely-used compression algorithm that combines the *LZ77* algorithm and *Huffman* encoding; (ii) the *Zlib* format, a *Deflate* algorithm with *Zlib* wrapper (adding a lightweight header); (iii) the *gzip* format, a *Deflate* algorithm with *gzip* wrapper (adding a comprehensive header).
13. *Gipfeli* [33] is a compression library developed by Google. It is based on the *Deflate* algorithm, employing several enhancements to improve efficiency. It utilizes techniques from the *LZ77* algorithm and entropy coding, similar to those found in other compression libraries.
14. *Heatshrink* [118] is a lightweight compression library focused on embedded systems and applications with limited resources. It uses a variation of the *LZ77* compression algorithm. It supports the following configuration parameters:
- **Window size.** The available values range from 4 to 14.
 - **Back reference.** The available values range from 3 up to the value of the window size parameter.
15. *Liblzg* [28] is a minimal implementation of an *LZ77* class compression library. The main characteristic of the library is that the decoding routine is effortless, fast, and requires little memory. It supports the following configuration parameter:
- **Compression level.** The available values range from 1 to 9.
16. *Lz4* [19] combines techniques derived from the Lempel-Ziv compression algorithms. It supports the following configuration parameters:
- **Compression level.** The available values range from 1 to 12.
 - **Mode.** This compression library provides two modes: (i) a standard implementation and (ii) the *LZ4HC* implementation, which uses more memory to achieve better compression ratios than the standard one, but still retains high speeds.

17. *Lizard* [106] is based on the well-known *Lz4* library, but it uses a compression format which is not compatible with *Lz4*. It supports the following configuration parameters:
- **Compression level.** The available values range from 0 to 9.
 - **Mode.** This parameter configures the algorithms used by the compression library. There are four different modes: (i) the fastLZ4 mode, which is a high-speed variant of the *Lz4* compression library; (ii) the LIZv1 mode, an enhanced version of LZ4 with better compression efficiency; (iii) the fastLZ4 + *Huffman* mode, which combines the fastLZ4 mode with the *Huffman* encoding to further improve the compression ratio; and (iv) the LIZv1 + *Huffman* mode, which combines the LIZv1 mode with the *Huffman* encoding.
18. *LodePNG* [115] implements a PNG encoder and decoder based on the *LZ77* algorithm, the *Huffman* encoding, and prediction filters. It supports the following configuration parameters:
- **Compression level.** The available values range from 1 to 2.
 - **Window size.** The available values range from 10 to 15.
 - **Back reference.** The available values range from 1 to 8.
 - **Work factor.** The available values range from 1 to 20.
 - **Flags.** Using this parameter, the compression library can be configured to take the first match or continue looking for a better. The latter improves the compression ratio, but reduces the speed.
19. *Lzf* [58] is a fast compression library. It is ideal for applications where you want to save some space but not at the cost of speed. This compression library combines an implementation of the *LZ77* algorithm with dictionary compression techniques. It supports the following configuration parameter:
- **Compression level.** The available values range from 0 to 1.
20. *Lzfs* [4] is a compression library developed by Apple that combines the *LZ77* compression algorithm with finite state entropy coding. It was introduced as part of the Apple software ecosystem.
21. *Lzfx* [20] is a compression library focused on high compression speed, particularly for data that contains redundancy. It employs a dictionary-based compression approach. Initially, it was developed in support of the HDF5 for the Python project.

22. *Lzham* [29] is based on Lempel-Ziv compression algorithms. It is a lossless data compression codec optimized for particularly fast decompression at very high compression ratios with a *Zlib* compatible API. It supports the following configuration parameters:
- **Compression level.** The available values range from 0 to 4.
 - **Window size.** The available values range from 15 to 29.
 - **Flags.** There are three available flags: (i) the extreme parsing flag, which enhances compression ratio by allowing a more complex parse graph, significantly increasing processing time; (ii) the deterministic parsing flag, which ensures that the compression process always returns the same output; and (iii) the tradeoff decompression rate for the compression ratio flag, which allows to get higher compression ratios, significantly reducing the decompression speed.
23. *Lzjb* [11] is based on the *LZ77* algorithm. It comes from Sun/Oracle's OpenSolaris implementation with minor enhancements to make the code lighter and easy to use.
24. *Lzma* [21] compresses data by combining the *LZ77* algorithm, *Arithmetic* coding, and Markov chain modeling. It supports the following configuration parameters:
- **Mode.** There are three different configurations or modes: (i) the default mode that keeps the resource usage relatively low, (ii) the extreme mode that improves the compression ratio but increases the compression time, and (iii) the *Lzma2* mode that introduces some improvements in the compression process.
 - **Number of threads.** When compressing a large file or dataset, the data can be split into several blocks, and each block can be processed independently in a different thread. Using this variable, we can configure the compression library to work with up to 8 threads.
25. *Lzmat* [26] is based on the Lempel-Ziv family of algorithms. It seeks to be a memory-efficient compression library, making an excellent solution for embedded systems.
26. *Lzo* [66] compresses data by replacing pattern matches (using a sliding dictionary). This compression library considers long matches and long literal runs to produce good results on highly redundant data. It also deals acceptably with non-compressible data. It supports the following configuration parameters:
- **Compression level.** The available values range from 0 to 10, depending on the mode used.

- **Mode.** This compression library contains eight different implementations, each one representing an improvement of the previous one: (i) the Lzo1 implementation is the first development of this compression library; (ii) the Lzo1a implementation enhances match handling; (iii) the Lzo1b implementation improves the compression with very redundant data and large block sizes; (iv) the Lzo1c implementation focuses on improving compression, providing better ratios at the cost of some speed; (v) the Lzo1f implementation achieves good compression results with small block sizes and binary data; (vi) the Lzo1x incorporates some improvements to overcome previous implementations; (vii) the Lzo1y and (viii) the Lzo1z implementations obtain almost identical results than Lzo1x implementation, but improve the results in some cases.
27. *Lzsse* [108] is another compression library based on the Lempel-Ziv compression algorithms. However, it also employs Streaming SIMD Extensions (SSE) instructions for parallel processing, allowing multiple matches per iteration. It supports the following configuration parameters:
- **Compression level.** The available values range from 1 to 17.
 - **Mode.** This compression library implements three different variants: (i) the LZSSE2, which focuses on high compression files with small literal runs; (ii) the LZSSE4, which centers on a more balanced mix of literals and matches; and (iii) the LZSSE8, whose objective is lower compression data with longer runs of matches.
28. *Miniz* [30] is a lossless, high-performance data compression library in a single source file that implements the *Zlib* and *Deflate* compressed data format specification standards. It supports the following configuration parameters:
- **Compression level.** The available values range from 1 to 9.
 - **Window size.** The available values range from 10 to 11.
 - **Mode.** This compression library contains five different strategies to compress data: (i) the default strategy automatically selects the strategy based on the input data; (ii) the filter strategy is preferred for data with lots of redundancies; (iii) the *Huffman* strategy uses only a *Huffman* coding without *LZ77* compression; (iv) the *Run-Length Encoding* strategy to compress data with many consecutive repeating values efficiently; and (v) the fixed strategy which uses a fixed *Huffman* table for faster compression and decompression, sacrificing some compression efficiency.

29. *Ms* [12] is a compression library that contains open-source implementations of Microsoft compression algorithms based on the *LZ77* algorithm and *Huffman* coding. It supports the following configuration parameter:
- **Mode.** This compression library can configure three different modes depending on the data to compress: (i) *Lznt1* mode used for NTFS file compression, Windows 2000 hibernation file, Active Directory, among others; (ii) *Xpress* algorithm used for Windows XP and newer hibernation file, Windows Update Services, among others; (iii) *Xpress* algorithm with *Huffman* encoding used in WIM files, Distributed File System Replication, Windows 7 SuperFetch, and Windows 8 bootmgr.
30. *Snappy* [35] is a high-speed compression library developed by Google. Unlike traditional libraries, this library prioritizes compression/decompression speed over compression ratio.
31. *Pithy* [25] comes from an old version of the *Snappy* compression library developed by the Google team. It is based on the *LZ77* algorithm, and its lightweight design makes it a good choice for embedded systems. It supports the following configuration parameter:
- **Compression level.** The available values range from 0 to 9.
32. *Quicklz* [94] is another compression library based on the *LZ77* algorithm. Its lightweight design makes it easy to integrate, making it a popular choice for embedded systems.
33. *Ucl* [67] is focused on generating a competitive compression ratio and requires no memory for decompression. It is based on *LZ77* and other compression techniques like *Run-Length Encoding*. It supports the following configuration parameters:
- **Compression level.** The available values range from 1 to 10.
 - **Mode.** This parameter specifies the different optimizations introduced by this library. There are three different modes: (i) the *NRV2B*, which uses a variation of the *LZ77* algorithm; (ii) the *NRV2D*, which is similar to *NRV2B* but introduces additional optimization for specific types of data; and (iii) the *NRV2E* which incorporates features from *NRV2D* and *NRV2E*, keeping a balance of the performance.

34. *Wflz* [13] is a compression library based on *LZ77* designed for use in game engines. Their modular design and use of chunks and hash tables make it faster and more effective for many applications. It supports the following configuration parameter:
- **Compression level.** The available values range from 0 to 1.
35. *Xpack* [10] is an experimental compression format. It is intended to perform better than *Deflate* as implemented in the *Zlib* library. It is based on the *LZ77* method, incorporating enhancements such as increased sliding window size, *Fse* encoding, and separate packing of literals and matches. It supports the following configuration parameter:
- **Compression level.** The available values range from 1 to 9.
36. *Yalz77* [113] is another variation of the *LZ77* compression algorithm, designed for code simplicity and clarity. It supports the following configuration parameters:
- **Compression level.** The available values range from 1 to 12.
 - **Window size.** The available values range from 10 to 16.
37. *Z3lib* [100] primarily uses the *Deflate* algorithm, which combines *LZ77* compression and *Huffman* coding. This compression library was born as a substitute for the well-known *Zlib* compression library. It supports the following configuration parameter:
- **Flags.** This parameter allows users to configure the compression library optimization. There are two possible flags: (i) prefer longer matches, which indicates longer matches have a preference; and (ii) limit length, which checks whether coding as literal is better for codes with length three and long distance.
38. *Zlib-ng* [96] is an enhancement of the *Zlib* compression library for the next-generation systems. It supports the following configuration parameter:
- **Compression level.** The available values range from 0 to 9.
39. *Zling* [60] is an improved lightweight compression utility and library. It uses fast order-1 ROLZ (16 MB block size and 10 MB dictionary size) followed by *Huffman* encoding. It supports the following configuration parameter:
- **Compression level.** The available values range from 0 to 4.

40. *Zpaq* [65] is a data compression library that focuses on incremental compression, i.e., it allows data to be compressed efficiently and quickly by processing only the changes made since the last compression. It supports the following configuration parameter:

- **Compression level.** The available values range from 0 to 11.

41. *Zstd* [71] is a fast lossless compression library, targeting real-time compression scenarios at the *zlib*-level and better compression ratios. It is backed by a fast entropy stage provided by the *Huffman* coding and the *Fse* library. It supports the following configuration parameter:

- **Compression level.** The available values range from 1 to 22.

4.1.3 Usage example

As previously mentioned, the *CPU-Smash* benchmark presents a very flexible API, allowing us to incorporate the different compression libraries with minor changes. Moreover, this API enables users to configure the compression parameters easily. Listing 4.1 shows an example of using the *Zstd* compression library with the *CPU-Smash* API. First, our code must include the *cpu_smash* header, where the compression/decompression logic is implemented, and the *cpu_options* header, which enables the compression parameter configuration. Once these headers have been included, the code creates the *CpuOptions* and sets the different parameter configurations. Then, the data to be compressed (the *dickens* file in this example) is located in a buffer. Next, the compression library (*Zstd* in this case) is initialized and configured before the compression starts. Once the compressed data is generated, it is decompressed similarly: the decompression options are configured and the decompression begins.

```
#include <iostream>
#include <fstream>
#include <cpu_smash.hpp>
#include <cpu_options.hpp>

CpuOptions options;
// Set compression library options
options.SetCompressionLevel(1);
// options.SetWindowSize(const uint32_t &window_size);
// ...

uint64_t u_size = 0, c_size = 0, d_size = 0;
```

```

// Read file to compress
ifstream file("dickens", ios::binary | ios::ate);
u_size = file.tellg();
file.seekg(0, ios::beg);

// Initialize uncompressed data with Dickens file information
char u_data[u_size];
file.read(u_data, u_size);
file.close();

// Select the compression library to use
CpuSmash lib("zstd");

// Set compressor options
lib.SetOptionsCompressor(&options);
// Get estimated compressed data size
lib.GetCompressedDataSize(u_data, u_size, &c_size);
// Initialize compressed data
char c_data[c_size];
// Compress and get real compressed data size
lib.Compress(u_data, u_size, c_data, &c_size);

// Set decompress options
lib.SetOptionsDecompressor(&options);
// Get estimated decompressed data size
lib.GetDecompressedDataSize(c_data, c_size, &d_size);
// Initialize decompressed data
char d_data[d_size];
// Decompress and get real decompressed data size
lib.Decompress(c_data, c_size, d_data, &d_size);

```

Listing 4.1: Example of using the *Zstd* compression library with the *CPU-Smash* API.

The flexibility of the *CPU-Smash* API also allows us to use multiple compression libraries. Listing 4.2 shows an example of a pipeline compression with the *CPU-Smash* API. First, the data is compressed using the *Zstd* compression library. Then, the resulting compressed data is compressed again using the *Snappy* compression library. Finally, the resulting data is compressed a third time using the *Lz4* compressed library.

```

#include <iostream>
#include <fstream>

```

```
#include <cpu_smash.hpp>
#include <cpu_options.hpp>

uint64_t u_size = 0, c_z_size = 0, c_s_size = 0, c_l_size = 0;

// Read file to compress
ifstream file("dickens", ios::binary | ios::ate);
u_size = file.tellg();
file.seekg(0, ios::beg);

// Initialize uncompressed data with Dickens file information
char u_data[u_size];
file.read(u_data, u_size);
file.close();

// Select the compression library to use
CpuSmash lib_zstd("zstd");
CpuSmash lib_snappy("snappy");
CpuSmash lib_lz4("lz4");

// Get estimated compressed data size
lib_zstd.GetCompressedDataSize(u_data, u_size, &c_z_size);
// Initialize compressed data
char c_zstd_data[c_z_size];
// Compress data using zstd and get real compressed data size
lib_zstd.Compress(u_data, u_size, c_zstd_data, &c_z_size);

// Get estimated compressed data size
lib_snappy.GetCompressedDataSize(c_zstd_data, c_z_size, &c_s_size);
// Initialize compressed data
char c_snappy_data[c_s_size];
// Compress data using snappy and get real compressed data size
lib_snappy.Compress(u_data, u_size, c_snappy_data, &c_s_size);

// Get estimated compressed data size
lib_lz4.GetCompressedDataSize(c_snappy_data, c_s_size, &c_l_size);
// Initialize compressed data
char c_lz4_data[c_l_size];
// Compress data using lz4 and get real compressed data size
lib_lz4.Compress(u_data, u_size, c_lz4_data, &c_l_size);
```

Listing 4.2: Example of using multiple compression libraries with the *CPU-Smash* API.

```
/smash/build/bin:./smash benchmark --compression_library all --input_file dickens
```

Library	Level	Window	Mode	WF	Flags	Threads	BR	Ratio	Total
zstd	1	-----	-----	---	-----	-----	---	2.39	152.63 MB/s
zpaq	0	-----	-----	---	-----	-----	---	2.47	17.64 MB/s
zling	0	-----	-----	---	-----	-----	---	3.00	33.97 MB/s
zlib-ng	0	-----	-----	---	-----	-----	---	1.00	2090.30 MB/s
zlib	0	-----	-----	---	-----	-----	---	1.00	816.18 MB/s
z3lib	-----	-----	-----	---	None	-----	---	2.58	6.88 MB/s
yalz77	1	10	-----	---	-----	-----	---	1.44	43.42 MB/s
xpack	1	-----	-----	---	-----	-----	---	2.41	63.95 MB/s
wflz	0	-----	-----	---	-----	-----	---	1.71	0.06 MB/s
ucl	1	-----	NRV2B	---	-----	-----	---	2.10	25.65 MB/s
snappy	-----	-----	-----	---	-----	-----	---	1.61	173.17 MB/s
quicklz	-----	-----	-----	---	-----	-----	---	1.75	151.77 MB/s
pithy	0	-----	-----	---	-----	-----	---	1.52	178.55 MB/s
ms	-----	-----	Lznt1	---	-----	-----	---	1.61	32.32 MB/s
miniz	1	10	None	---	-----	-----	---	1.90	50.20 MB/s
lzsse	1	-----	LZSSE2	---	-----	-----	---	1.94	12.77 MB/s
lzo	0	-----	LZ01	---	-----	-----	---	1.61	110.73 MB/s
lzmat	-----	-----	-----	---	-----	-----	---	2.26	14.60 MB/s
lzma	-----	-----	Default	---	-----	1	---	3.60	1.53 MB/s
lzjb	-----	-----	-----	---	-----	-----	---	1.32	73.36 MB/s
lzham	0	15	-----	---	None	-----	---	2.34	2.21 MB/s
lzfx	-----	-----	-----	---	-----	-----	---	1.61	106.37 MB/s
lzfse	-----	-----	-----	---	-----	-----	---	2.64	36.90 MB/s
lzf	0	-----	-----	---	-----	-----	---	1.74	133.01 MB/s
lz4	0	-----	Fast	---	-----	-----	---	1.59	268.46 MB/s
lodepng	1	10	-----	1	Fast	-----	1	1.45	20.99 MB/s
lizard	0	-----	FastLZ4	---	-----	-----	---	1.58	191.78 MB/s
liblzg	1	-----	-----	---	-----	-----	---	1.47	34.71 MB/s
libdeflate	0	-----	Deflate	---	-----	-----	---	1.00	2818.21 MB/s
libbse	1	10	Bwt	---	None	-----	4	1.00	14.81 MB/s
heatshrink	-----	4	-----	---	-----	-----	3	0.98	13.83 MB/s
gipfeli	-----	-----	-----	---	-----	-----	---	2.06	101.03 MB/s
fse	-----	-----	FSE	---	-----	-----	---	1.76	194.92 MB/s
flzma2	1	-----	-----	---	-----	1	---	2.83	12.13 MB/s
flz	1	-----	-----	---	-----	-----	---	1.68	121.20 MB/s
density	-----	-----	Chameleon	---	-----	-----	---	1.75	577.07 MB/s
csc	1	15	-----	---	None	-----	---	2.56	10.65 MB/s
c-blosc2	0	-----	-----	---	None	1	---	1.00	2812.31 MB/s
bzip2	1	-----	Faster	0	-----	-----	---	3.09	7.85 MB/s
brotli	0	10	Generic	---	-----	-----	---	1.58	40.65 MB/s
brieflz	1	-----	-----	---	-----	-----	---	2.22	50.25 MB/s

Figure 4.1: Smash benchmark execution using all the available compression libraries and the *dickens* dataset.

Figure 4.1 shows the execution of the *CPU-Smash* benchmark using all the compression libraries to compress and decompress the *dickens* dataset with default configuration parameters. For each compression library, the benchmark shows the following information: compression library name, configuration parameters, compression ratio achieved, and total speed. Note that the compression ratio value was previously defined in Equation 3.1 as the ratio between the uncompressed data size and the compressed data size. Total speed represents the speed of the compression library for compressing and decompressing data, the *dickens* dataset in our example. The speed is formally defined in Equation 4.1 as the ratio between (i) the uncompressed data size and (ii) the time taken when the original data is compressed plus the taken when the compressed data is decompressed.

$$Speed = \frac{Size_{original}}{Time_{compression} + Time_{decompression}} \quad (4.1)$$

4.2 Evaluation

This section details the evaluation of all CPU-based compression libraries under consideration. They have been tested using our new benchmark *CPU-Smash*. All CPU-based compression libraries have been tested on the same computer to ensure consistent and significant results. The computer features are presented in Table 4.1.

Architecture	x86_64
Operating System	Ubuntu18.04.5LTS
Processor	Intel(R) Xeon(R) CPU E5-2637 v2 @ 3.50GHz
RAM	64GB

Table 4.1: Specifications of the computer used in the CPU-based compression libraries experiments.

We divided the evaluation into two different subsections. First, the compression library results are synthesized to make comparing easier. Then, they are evaluated using standard datasets and our new AI dataset.

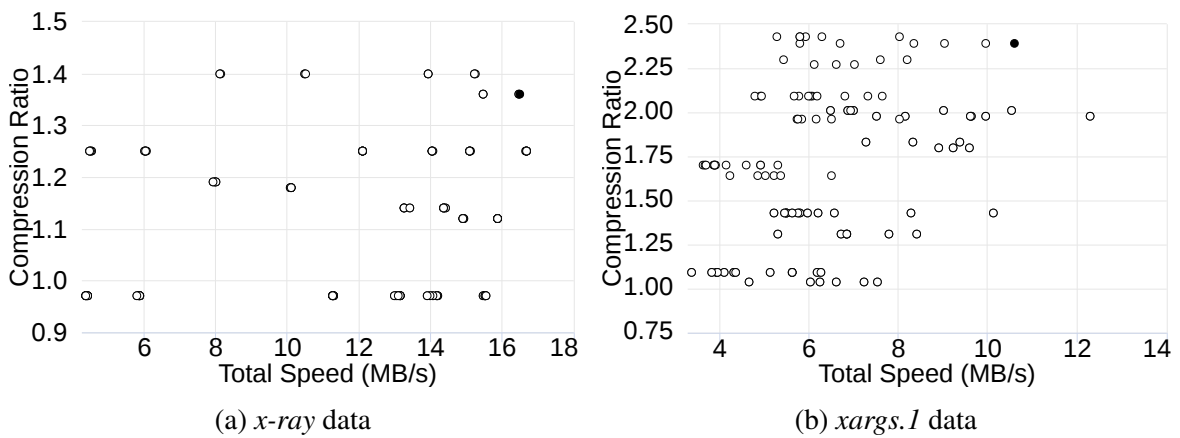


Figure 4.2: *LodePNG* library using all possible combinations of parameters.

4.2.1 Synthesizing results

As commented before, each compression library has different parameters to optimize its compression. The combination of all the possible parameters can be enormous. Figure 4.2 presents the results of compressing and decompressing two data obtained from the standard datasets, *x-ray* and *xargs.1*, using the *LodePNG* compression library and all combinations of possible parameters introduced in Section 4.1.2. In this example, *LodePNG* has two compression levels, six window sizes, eight back reference sizes, twenty work factor levels, and two flags. This results in up to 3,840 different combinations of the available configuration parameters. Certainly, there is a lot of information. Note that these results come from a single compression library. Table 4.2 summarizes all possible configurations for all the CPU-based compression libraries used in this thesis.

For clarity, it will be therefore sensible to synthesize the information presented. Thus, we have first run all the compression libraries with all data obtained from the datasets using all possible combinations of parameters, similar to Figure 4.2. Next, we have obtained the maximum compression ratio and total speed achieved by each compression library and data. Then, we have obtained the normalized values of these speeds and compression ratios using their maximum values (i.e., dividing each value by its maximum value). Finally, we have weighted the normalized compression ratio and the normalized total speed by using α and β coefficients. The sum of the α and β weights must be 1 to get a final result between 0 and 1. The configuration of the compression library with a value closer to 1 will be the best for a given data. In the experiments shown in this section, we have assumed that the compression ratio is as important as the total speed. Thus, we have set the α and β weights to 0.5 each. Equation 4.2 summarizes the above strategy.

$$f(result) = \left(\frac{Ratio_{result}}{Ratio_{max}} * \alpha \right) + \left(\frac{Speed_{result}}{Speed_{max}} * \beta \right) \quad (4.2)$$

In Figure 4.2a and Figure 4.2b, we can see the results obtained with the *LodePNG* compression library when compressing and decompressing the *x-ray* and *xargs.1* data, respectively. These data have been compressed using all possible combinations of parameters of the *LodePNG* compression library. The black dot represents the result of the best configuration calculated using Equation 4.2, while the white ones depict other possible combinations of parameters. Using this strategy, we have selected the best configurations for all the compression libraries and data, achieving a single result for each pair.

Compression library	All possible configurations
<i>Brieflz</i>	10
<i>Brotli</i>	540
<i>Bzip2</i>	468
<i>C-blosc2</i>	240
<i>Csc</i>	600
<i>Density</i>	3
<i>Flz</i>	2
<i>Flzma2</i>	80
<i>Fse</i>	2
<i>Gipfeli</i>	1
<i>Heatshrink</i>	66
<i>Libbsc</i>	18,240
<i>Libdeflate</i>	39
<i>Liblzg</i>	9
<i>Lizard</i>	40
<i>LodePNG</i>	3,840
<i>Lz4</i>	26
<i>Lzf</i>	2
<i>Lzfse</i>	1
<i>Lzfx</i>	1
<i>Lzham</i>	600
<i>Lzjb</i>	1
<i>Lzma</i>	24
<i>Lzmat</i>	1
<i>Lzo</i>	36
<i>Lzsse</i>	51
<i>Miniz</i>	90
<i>Ms</i>	3
<i>Pithy</i>	10
<i>Quicklz</i>	1
<i>Snappy</i>	1
<i>Ucl</i>	30
<i>Wflz</i>	2
<i>Xpack</i>	9
<i>Yalz77</i>	84
<i>Z3lib</i>	4
<i>Zlib</i>	10
<i>Zlib-ng</i>	10
<i>Zling</i>	5
<i>Zpaq</i>	12
<i>Zstd</i>	22

Table 4.2: All possible configurations of CPU-based compression libraries.

4.2.2 Evaluating standard and AI datasets

Evaluating standard and AI datasets aims to analyze how the *CPU-Smash* library performs with each one. As previously discussed in Section 3, our new AI dataset exhibits higher entropy, and a preliminary study conducted with a subset of compression libraries indicated that the compression ratio tended to be lower than the standard ones. In this subsection, we extend this evaluation to all CPU-based compression libraries included in *CPU-Smash*, analyzing their behavior in detail with datasets of the same size.

The legends of the following figures in this section are the same and very extensive. For that reason, we show the legend only once in Figure 4.3. The same legend is used for subsequent figures.

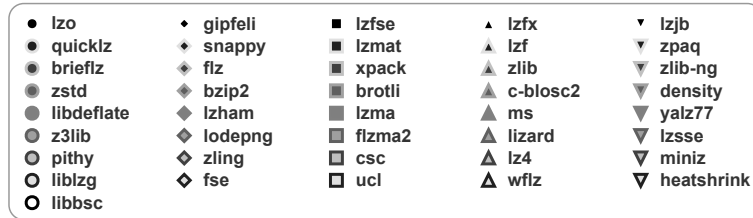


Figure 4.3: Legend used in Figure 4.6, Figure 4.4 and Figure 4.5.

If we focus on specific examples, such as *cp.html* and *AI-25KB* data, shown in Figure 4.4, we can see that both of them have a similar size, around 25KB. Some compression libraries achieve a total speed higher than 100MB/s with the *cp.html* data, while they cannot exceed that speed with the *AI-25KB* data. The complexity of the data of *AI-25KB* is higher than *cp.html*, and that makes the compression libraries need more time to compress and decompress the data obtained from the AI dataset. Furthermore, no compression library gets a compression ratio higher than 1.8 with the *AI-25KB* data, while many get values higher than 2, even 3, with *cp.html*.

Using data with a size larger than 25KB, we achieved similar conclusions. Figure 4.5a and Figure 4.5b present the best results for every CPU-based compression library when compressing and decompressing the *paper-100k.pfd* and *AI-100KB* data. These data have a size of around 100KB. This time, the total speed of compression libraries is similar in both cases. However, compression libraries achieve a minor compression ratio with the *AI-100KB* data, even lower than 1. Note that a compression ratio lower than one means the compressed data is larger than the uncompressed data.

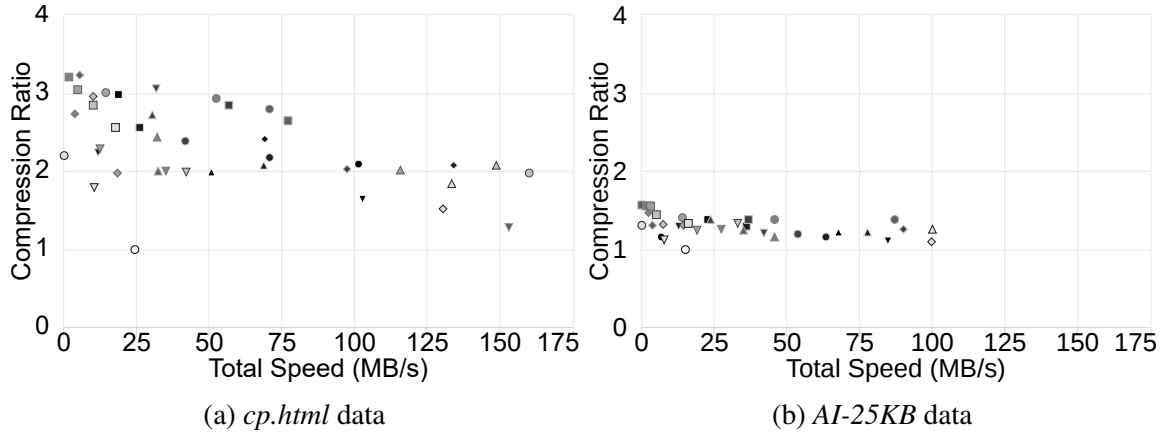


Figure 4.4: Comparison between a standard 25KB data and the AI 25KB data.

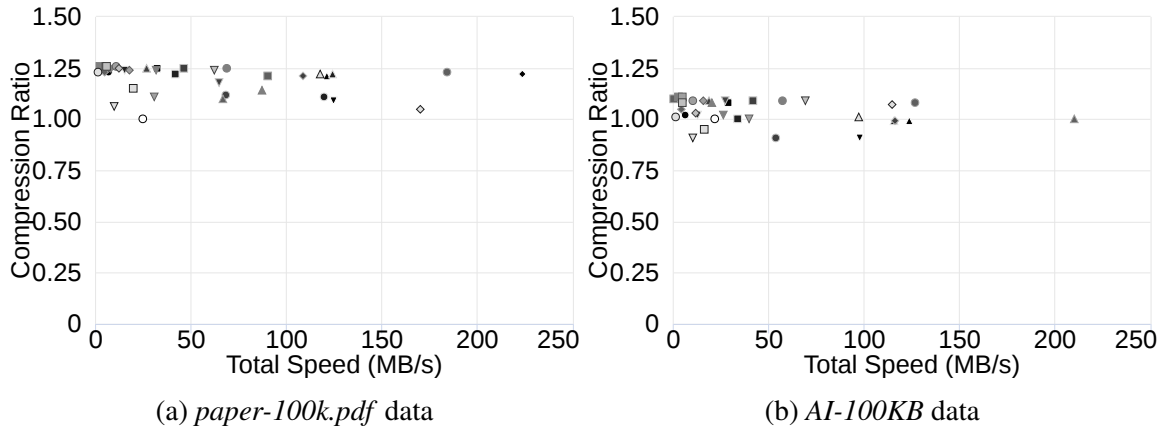


Figure 4.5: Comparison between a standard 100KB data and the AI 100KB data.

Figure 4.6 shows the average of the best results from all data for every compression library: Figure 4.6a shows results with the standard datasets presented in Section 2.2, whereas Figure 4.6b shows results for the AI dataset introduced in Chapter 3. Most compression libraries obtain an average speed between 0 and 160MB/s in both cases, but we can observe significant differences in their compression ratio. Even though most of them get a compression ratio between 2 and 5 with standard datasets, they barely achieve a compression ratio higher than 2 with the AI dataset. That makes sense because these compression libraries focus on text, image, or binary datasets instead of data transferred to/from the remote GPU when running AI applications.

This analysis confirms that AI datasets present significantly different characteristics than standard ones. Thus, the CPU-based compressed libraries achieve lower compression ratios and reduced processing speeds with AI datasets. That makes sense due to their higher complexity and entropy. Unlike standard datasets, which often exhibit more structured and

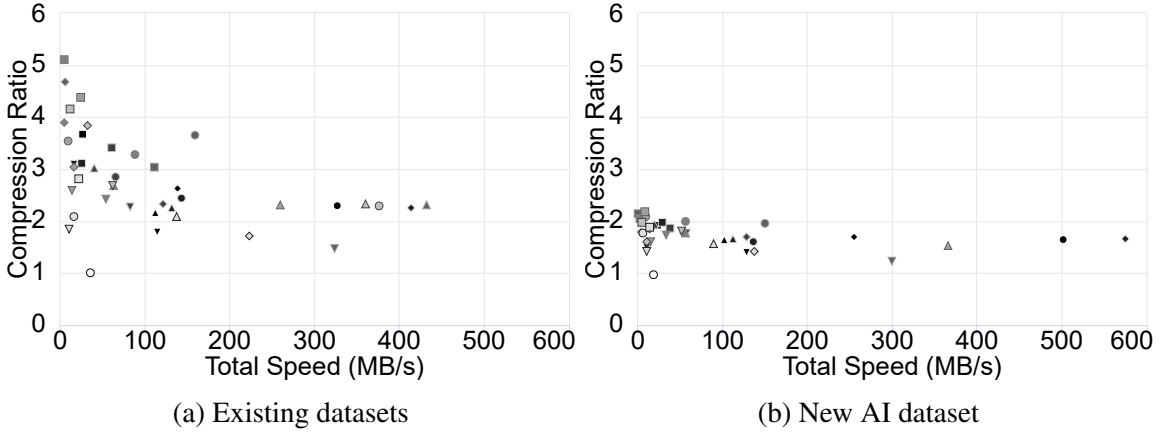


Figure 4.6: Average results for all datasets and compression libraries.

repetitive patterns, the structure of the AI datasets appears less predictable, making it more challenging for traditional compression techniques. These results emphasize the importance of selecting appropriate compression strategies, particularly in environments where efficient storage and transfer are critical.

4.3 Summary

A significant number of compression libraries have been collected to create a new benchmark called *CPU-Smash*. This new benchmark is focused on evaluating the performance of the different CPU-based compression libraries, which compress and decompress data located in RAM memory. No prior research is necessary to use *CPU-Smash*, only setting the specific parameters of the compression library to test and provide the data. A flexible API is also available to adapt our project with the CPU-based compression library and the configuration selected.

Running *CPU-Smash* with the datasets presented in Section 2.2 and Chapter 3, we confirm that using an appropriate dataset is very important to evaluate the different compression libraries. Each compression library follows different strategies. These strategies address specific types of data. Thus, to use compression in one environment, it must be evaluated with an appropriate dataset.

It is also important to recall that this thesis aims to improve the communication efficiency of a remote GPU virtualization system. These systems usually follow a client-server architecture, and it is also typical for the client side to operate without a GPU. This constraint makes

CPU-based compression essential for optimizing data transfer. The *CPU-Smash* library is fundamental in this context, providing an abstraction framework to easily integrate and evaluate a wide range of CPU-based compression libraries. By leveraging *CPU-Smash*, we can identify the most suitable compression strategies for reducing communication overhead.

Chapter 5

Introducing GPU-based compression libraries into the compression procedure

CPU-based compression libraries have been widely studied in the past. Despite that, the high computational cost of the compression and decompression processes can limit their utilization in real-time applications. GPUs can mitigate the limitation of CPU-based compression libraries by carrying out all or part of the compression computation. The CPU and the GPU can work together to compress data. Alternatively, the CPU can do another task while the GPU compresses data, lightening the CPU workload. Furthermore, highly concurrent programming can also be done on the GPU, developing strategies different from those currently existing on the CPU.

The implementation of GPU-based compression algorithms is currently under research. However, these solutions are much more novel and challenges remain due to the GPU model itself, which is different compared to the model of traditional CPUs.

In this chapter, we introduce a collection of GPU-based compression libraries. Next, we create a new GPU-based compression abstraction library and benchmark called *GPU-Smash*, which is available at <https://github.com/cpenaranda/gpu-smash>. Then, we evaluate the implemented compression libraries and improve their results using a pipeline strategy. Finally, we compare the GPU-based compression libraries, which are also implemented on the CPU, and create an initial CPU-GPU heterogeneous compression library that transparently uses the CPU- or the GPU-based compression libraries, depending on the data.

5.1 *GPU-Smash*

The high performance offered by GPUs motivates researchers in various fields to offload their application computing to these devices. In the case of compression, some researchers [90, 87] use the same CPU-based compression algorithms presented in Section 2.1, while others create new GPU-based compression libraries [16, 59] following the same strategies as their CPU-based version. More concretely, Patel et al. [90] introduce 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 Swamy [87] present a parallel version of *LZSS* for GPUs. This version divides the memory into different chunks and compresses each block independently. They show how their GPU version improves the compression time of CPU sequential and CPU parallel ones by up to 18 and 3 times, respectively. Chłopkowski and Walkowiak [16] develop a new library based on the *Deflate* one where CPU and GPU work together to improve CPU results. These studies explore 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 one used in the experiments. Li et al. [59] implement a parallel version of the *C-DPCM* algorithm on GPUs to improve the compression time for hyperspectral images. They explore 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.

NVIDIA [82] has also joined this trend and has created a new library called *nvCOMP* [84]. This library provides a generic compression interface that allows developers to use GPU-based compressors in their applications. The NVIDIA team has implemented eight different compression libraries on the GPU and has included them in the *nvCOMP* library.

In a similar way to what we did in the previous chapter, we looked for existing benchmarks for GPU-based compression libraries. However, unlike what happened with CPU-based libraries, we could not find any benchmark. For that reason, we developed the *GPU-Smash* following a methodology similar to the *CPU-Smash* one. Thus, the *GPU-Smash* benchmark comprises different GPU-based compression libraries. It also introduces an abstraction library that provides a higher-level interface to simplify using GPU-based libraries.

5.1.1 Available configurations

As CPU-based compression libraries, each GPU-based compression library presents different parameters to compress and decompress data. *GPU-Smash* implements all the parameters supported by the *nvCOMP* library by NVIDIA. If new GPU-based compression libraries were added in the future, the *GPU-Smash* library would need to be adapted to support the new parameters. The following section 5.1.2 will explain the available values that each parameter can take depending on the compression library. At this point, we list the parameters that *GPU-Smash* currently supports:

- **Compression level.** This parameter allows to control the quality and speed of the compression data. Depending on the value, the compression can be faster but with a lower compression ratio, or slower with a higher compression ratio.
- **Chunk size.** GPU-based compression libraries usually split data into chunks and compress each one independently. This parameter indicates the bytes used for storing each chunk. For example, a value of 12 means we use a chunk size of $2^{12} = 4$ Kilobytes.
- **Flags.** Flags are used to control the strategy used by the compression library.
- **Number of chunks.** This parameter is external to GPU-based compression libraries, i.e., it was specifically developed for *GPU-Smash*. It looks for enhancing the results of GPU-based compression libraries using a pipeline mechanism. The research behind this approach is presented in Section 5.2.3. It can range from 4 to 13.
- **Number of streams.** This parameter is also an external one explicitly created for *GPU-Smash*. It seeks to improve the compression by running kernels simultaneously in multiple streams. The investigation concerning this parameter is detailed in Section 5.2.4. It can range from 1 to 8.

We can observe that there is a new parameter that does not exist in the *CPU-Smash* library: the chunk size. This chunk size is a new feature of GPU-based compression libraries. These compression libraries present a different behavior than CPU-based ones. Figure 5.1 illustrates this. On the one hand, CPU-based compression libraries typically use one CPU thread to compress data. On the other hand, GPU-based compression libraries split data into different pieces (or chunks) and compress all of them on the GPU using different kernel threads simultaneously. Note that a CPU-based compression library can use multiple CPU threads to

compress data using a similar strategy to GPU-based ones. However, the high performance of the GPU enables a significant number of kernel threads to work concurrently, making it really appropriate in this case.

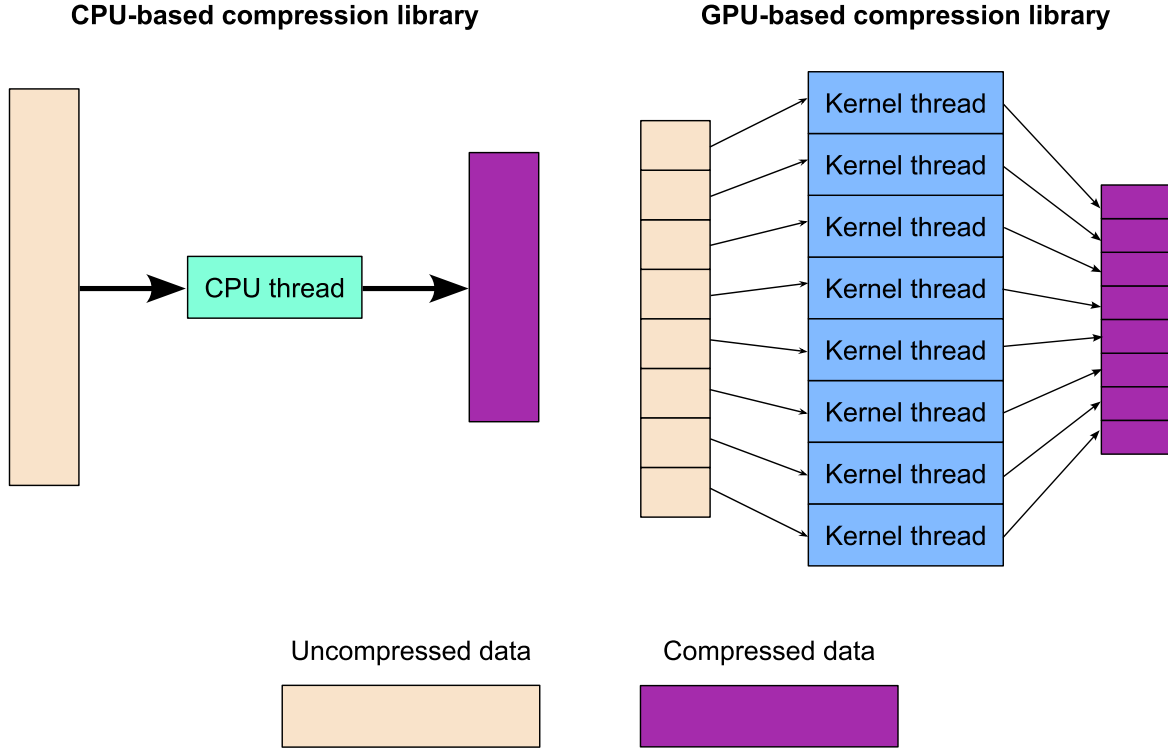


Figure 5.1: Comparison of the typical approach followed by CPU- and GPU-based compression libraries.

5.1.2 Available compression libraries

This section introduces the GPU-based compression libraries contained in our new GPU-Smash benchmark. The increasing demand for faster data processing has motivated the development of compression algorithms optimized for GPUs. This approach aims to significantly accelerate the compression/decompression processes using the parallel processing power of GPUs, enhancing the performance of traditional CPU-based libraries.

Despite having other GPU-based compression libraries, in this work we have decided to use the ones implemented by the *nvCOMP* library. This decision is based on different key factors. First, the *nvCOMP* library was designed by NVIDIA, which is widely recognized for its work in GPU technology. Its tools and libraries are synonymous with performance, efficiency, and scalability. Moreover, NVIDIA offers extensive documentation and support,

and a great community, making their products reliable and well-maintained. Lastly, libraries developed by NVIDIA are designed to ensure compatibility, performance, and reliability of its GPUs.

Given these advantages, we decided to consider only the eight different GPU-based compression libraries implemented by the *nvCOMP* library into the *GPU-Smash*. However, new compression libraries can be easily added in the future. These compression libraries are:

- *Lz4* is a general-purpose no-entropy byte-level compressor well-suited for many datasets and combines techniques derived from the Lempel-Ziv compression algorithms. It is a GPU-based implementation of the CPU-based library *Lz4*, which was presented in Section 4.1.2. It supports the following configuration parameters:
 - **Chunk size.** The available values range from 12 to 24.
 - **Flags.** This parameter is used to specify the type of symbol used to compress data. Depending on the type, the library optimizes its internal configuration to return the optimal result. The available symbols in this library are Char, Unsigned Char, Short, Unsigned Short, Int, Unsigned Int, and Bits.
- *Snappy* is a GPU-based implementation of the popular compression library developed by Google, whose CPU-based implementation was also presented in Section 4.1.2. This library prioritizes compression/decompression speed over compression ratio. It supports the following configuration parameter:
 - **Chunk size.** The available values range from 12 to 24.
- *Deflate* combines the *LZ77* algorithm and *Huffman* codification to achieve a fast and efficient compression. This library represents a GPU-based implementation of *Libdeflate*, which was presented in Section 4.1.2. It supports the following configuration parameters:
 - **Chunk size.** The available values range from 12 to 16.
 - **Compression level.** The available values range from 0 to 1.
- *Zstd* is the GPU-based implementation of the popular compression library developed by Meta. As we introduced in Section 4.1.2, it combines several algorithms (such as *Huffman*, *LZ77*, and *FSE*) to achieve different results than other libraries. It supports the following configuration parameter:

- **Chunk size.** The available values range from 12 to 16.
- *Gdeflate* uses a new compression format that closely matches the *Defalte* format and allows more efficient GPU decompression. It supports the following configuration parameters:
 - **Chunk size.** The available values range from 12 to 16.
 - **Compression level.** The available values range from 0 to 2.
- *Cascaded* is a novel high-throughput compressor ideal for analytical or structured/tabular data. It cannot configure any parameter, except for the *GPU-Smash* abstraction library internal ones, i.e., the number of streams and chunks.
- *Bitcomp* is a compressor designed for efficient GPU compression in Scientific Computing applications. It supports the following configuration parameters:
 - **Compression level.** The available values range from 0 to 1.
 - **Flags.** This parameter is used to specify the type of symbol used to compress data. Depending on the type, the library optimizes its internal configuration to return the optimal result. The available symbols in this library are Char, Unsigned Char, Short, Unsigned Short, Int, Unsigned Int, Long Long, Unsigned Long Long, and Bits.
- *Ans* is an entropy encoder based on asymmetric numeral systems (ANS). It cannot configure any parameter, except for the *GPU-Smash* abstraction library internal ones, i.e., the number of streams and chunks.

5.1.3 Usage example

The usage of the *GPU-Smash* benchmark is similar to the *CPU-Smash* one. However, the usage of the abstraction library changes slightly. CPU-based compression libraries are developed to compress data stored in main memory, while GPU-based ones are designed to compress data stored in GPU memory. Therefore, our *GPU-Smash* abstraction library enables different kinds of compression depending on the memory location. Notice that GPU-based compression libraries are not able to compress data located in main memory. For this reason, we implemented that mechanism with four different kinds of compression/decompression:

- **Device to device (D2D).** Data to be compressed/decompressed and the result of that compression/decompression is located in the GPU memory. The original *nvCOMP* compression library follows this approach.
- **Host to device (H2D).** Data to be compressed/decompressed is located in the main memory, while its result will be stored in the GPU memory.
- **Device to host (D2H).** Data to be compressed/decompressed is located in the GPU memory, while its result will be stored in the main memory.
- **Host to host (H2H).** Data to be compressed/decompressed and the result of that compression/decompression is located in the main memory. This type of compression is the one used by CPU-based compression libraries.

The *GPU-Smash* abstraction library presents a very flexible API. Listing 5.1 shows an example of using the GPU-based *Zstd* compression library. First, our code must include the *gpu_smash* header, where the compression/decompression logic is implemented, and the *gpu_options* header, which enables the compression parameter configuration. Once headers have been included, the code creates the *GpuOptions* and sets the different parameter configurations. Next, the data to be compressed (the *dickens* file) is located in a buffer. Then, the compression library (*Zstd*) is initialized, options are set to configure the compression process, and the compression starts. Once the compressed data is generated, it is decompressed in a similar way. Thus, options are set to configure the decompression process, and the decompression can begin after that.

We see that the API is similar to the previous *CPU-Smash* abstraction library if we compare it with Listing 4.1. However, there are some differences. As commented before, when we use GPU-based compression libraries the data can be stored in host memory (i.e., main memory) or in device memory (i.e., GPU memory). For that reason, the ‘GetCompressedDataSize’ call offers now two possibilities: ‘GetCompressedDataSizeFromHostMemory’ and ‘GetCompressedDataSizeFromDeviceMemory’. The same happens for the ‘GetDecompressedDataSize’ call. These calls get the compressed/uncompressed data size from the data to be compressed/decompressed. Thus, depending on where data is stored, its specific call must be run. In a similar way, we need to specify where the data is located when compressing or decompressing. In the example, data is in the main memory. For this reason, we have set ‘CompressionType::HostToHost’. Note that we have also all the options described above, which provides great flexibility.

```
#include <iostream>
#include <fstream>
#include <gpu_smash.hpp>
#include <gpu_options.hpp>

GpuOptions options;
// Set compression library options
options.SetChunkSize(14);
// options.SetCompressionLevel(const uint8_t &compression_level);
// options.SetChunkNumber(const uint8_t &window_size);
// options.SetStreamNumber(const uint8_t &mode);
// ...

uint64_t u_size = 0, c_size = 0, d_size = 0;

// Read file to compress
ifstream file("dickens", ios::binary | ios::ate);
u_size = file.tellg();
file.seekg(0, ios::beg);

// Initialize uncompressed data with Dickens file information
char u_data[u_size];
file.read(u_data, u_size);
file.close();

// Select the compression library to use
GpuSmash lib("nvcomp-zstd");

// Set compressor options
lib.SetOptionsCompressor(&options);
// Get estimated compressed data size
lib.GetCompressedDataSizeFromHostMemory(u_data, u_size, &c_size);
// Initialize compressed data
char c_data[c_size];
// Compress and get real compressed data size
lib.Compress(u_data, u_size, c_data, &c_size, CompressionType::
    HostToHost);

// Set decompress options
lib.SetOptionsDecompressor(&options);
// Get estimated decompressed data size
lib.GetDecompressedDataSizeFromHostMemory(c_data, c_size, &d_size);
// Initialize decompressed data
char d_data[d_size];
```

```
// Decompress and get real decompressed data size
lib.Decompress(c_data, c_size, d_data, &d_size, CompressionType::
    HostToHost);
```

Listing 5.1: Example of using the *Zstd* compression library with the *GPU-Smash* API.

The next section presents iteratively how we have created an efficient H2H implementation. It should be noted that we show the H2H implementation because it is directly comparable with the CPU-based compression. In each iteration of our presentation, we also introduce experiments to measure the performance and justify the decisions made. Although we focus only on H2H compression, D2H and H2D compression/decompression behave similarly. Thus, D2H is similar to H2H except that transfers to the GPU memory at the beginning are avoided. For H2D, transfers at the end from the GPU memory are the ones avoided.

5.2 Developing an efficient GPU-based compression library

When we introduced the CPU-based compression libraries, we said that the data to be compressed was already stored in the main memory. However, GPU-based compression libraries work with GPU memory to obtain their performance. Besides, GPU-based libraries usually implement their mechanisms using only data stored in GPU memory. Therefore, when using a GPU to compress or decompress data, that data must be previously copied from main memory to GPU memory and, after being processed, the resulting data must be stored back in main memory. For that reason, in the process of designing our *GPU-Smash* abstraction library, we have followed an incremental approach, enabling different kinds of compression or versions of our development:

- **Version 0 (V0):** an ideal implementation of our proposal. A direct application of the NVIDIA GPU-based compression library *nvCOMP*. In this initial implementation, the data to be compressed/decompressed is already located in GPU memory. Notice that this approach is unrealistic but should provide an upper limit for performance.
- **Version 1 (V1):** a naive GPU-based solution where data are located in host memory (i.e., main memory) and thus it must be copied to/from the GPU memory.
- **Version 2 (V2):** a pipeline implementation inside the compression and the decompression that improves the previous version by optimizing the overall data movement.

- **Version 3 (V3):** an enhancement of the previous Version 2 by leveraging multiple concurrent GPU streams within the pipeline.
- **Version 4 (V4):** a reduction of compression and decompression times using multiple GPUs in parallel.

Next, we present the development of all the previous versions. We have analyzed the performance of each version by using four of the eight GPU-based compression libraries: *Lz4*, *Snappy*, *Deflate*, and *Zstd*. We selected these compression libraries because they are also implemented on the CPU in the *CPU-Smash* benchmark, and these new results could be compared to the previous ones. Table 5.1 illustrates the features of the computer where these libraries have been tested.

Architecture	x86_64
Operating System	Ubuntu20.04.4LTS
Processor	AMD EPYC 7282 16-Core Processor
RAM	256GB
GPU	NVIDIA A100

Table 5.1: Specifications of the computer used to experiment with GPU-based compression libraries.

Similar to what happened with CPU-based compression libraries, each GPU-based library has different parameters to optimize its compression. This makes the combination of all possible parameters enormous. Therefore, it will be sensible to synthesize the information presented for clarity. For this purpose, we have used the same strategy we employed in the previous Section 4.2. Thus, we have first run all the compression libraries with all data obtained from the datasets using all possible combinations of parameters. Next, we have obtained the maximum compression ratio and total speed achieved by each compression library and data. Then, we have obtained the normalized values of these speeds and compression ratios using their maximum values (i.e., dividing each value by its maximum value). Finally, we have weighted the normalized compression ratio and the normalized total speed by using α and β coefficients. The sum of the α and β weights must be 1 to get a final result between 0 and 1. The configuration of the compression library, which obtains a value closer than other configurations to 1, will get the best results for a given data. In the experiments shown in this section, we have assumed that the compression ratio is as important as the total speed. Thus, we have set the α and β weights to 0.5. The previous Equation 4.2 summarizes the above strategy. Moreover, in Section 4.2, we have observed that using an appropriate dataset

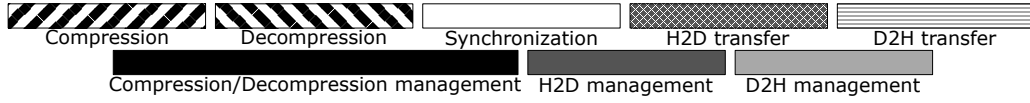


Figure 5.2: Legend for execution traces shown in Figures 5.3, 5.5, 5.7, 5.10, 5.11, 5.12, 5.13, 5.14, 5.17, and 5.18.

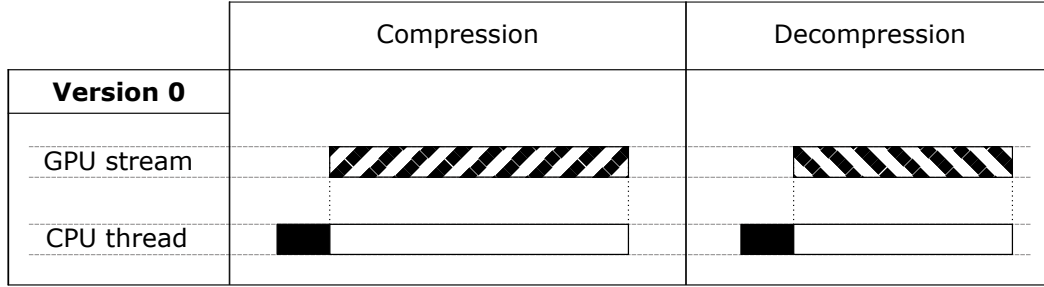


Figure 5.3: Execution trace for Version 0 (V0). The legend is shown in Figure 5.2.

to evaluate compression libraries is very important. Therefore, the experiments have been carried out using the AI dataset, presented in Chapter 3.

During the presentation and analysis of the different versions of our GPU-based compression library, we will use figures containing execution traces. Figure 5.2 illustrates the legend of the different tasks appearing in these traces. The CPU thread performs the management and synchronization tasks, while GPU streams do the others. The management tasks set asynchronous calls to the GPU stream. The compression and decompression tasks are kernels run on the GPU. The H2D transfer task refers to data copies from host memory to device GPU memory, whereas the D2H transfer task refers to copies in the opposite direction.

5.2.1 Version 0 (V0): ideal GPU compression

This initial version is a straightforward use of the NVIDIA *nvCOMP* library to implement the compression. Notice that the original data is already in GPU memory, and the data generated by the compression/decompression process is also stored in GPU memory. Therefore, this ideal version is not directly comparable with CPU compression libraries. Figure 5.3 presents the execution trace for this implementation (the legend for this figure is shown in Figure 5.2). 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 synchronizes to detect the completion of that GPU kernel. A similar execution flow is followed when decompressing data.

Using this Version 0 (V0), we have evaluated the speed achieved by the GPU-based compression libraries, i.e., the *nvCOMP* library. The speed was defined in Equation 4.1 as the ratio between (i) the uncompressed data size and (ii) the time taken when the original data is compressed plus the taken when the compressed data is decompressed. Figure 5.4 shows the speed achieved by these GPU-based compression libraries. As we can see, as the size of the data to compress increases, the speed obtained by GPU-based compression libraries increases too. The GPU setup time penalizes the GPU speed when using small data sizes. However, when big data are compressed, this time is hidden due to the longer computing time.

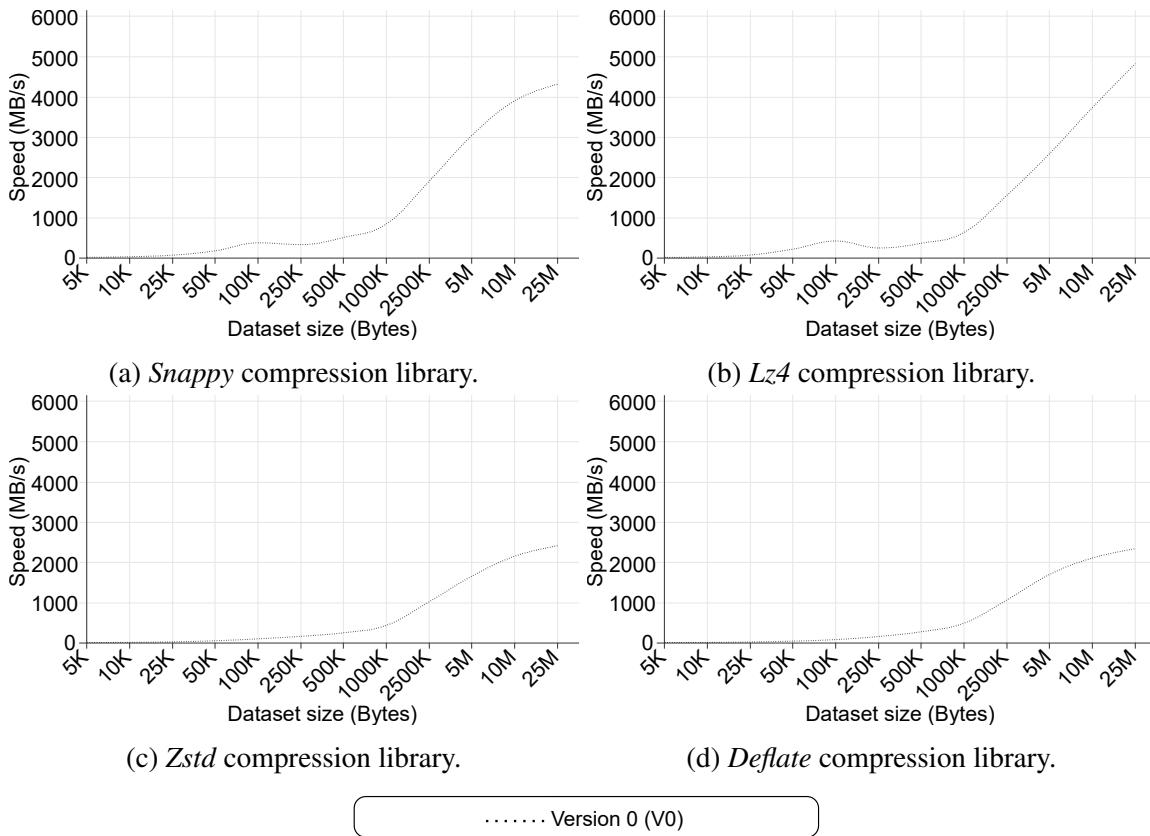


Figure 5.4: The compression and decompression speed achieved by Version 0 (V0).

5.2.2 Version 1 (V1): naive GPU compression

Previous Version 0 (V0) implements a compression where original and compressed data are located at the GPU memory, bringing to an incomplete compression/decompression process because application data typically reside on main memory. On the contrary, Version 1 (V1) performs the whole compression process, taking data from the host memory and saving the result in the host memory. This kind of compression is more common in real applications. Figure 5.5 shows the execution trace for V1. Original data are in host memory. Thus, the CPU thread must perform an asynchronous H2D memory copy before running the GPU kernel. As we can see, the kernel management is carried out while the H2D transfer is done in the GPU stream. At this point, there is a difference between the compression and the decompression. In compression, the final size of the compressed data is known once the kernel execution ends. Therefore, the CPU thread synchronizes the kernel before calling the D2H management. However, in decompression, the size of the original data is stored in the compressed data header. For that reason, no synchronization is required, and the CPU thread can run the D2H management immediately after the kernel one. Finally, the CPU thread synchronizes the GPU stream until the D2H transfer is done in the compression and the decompression.

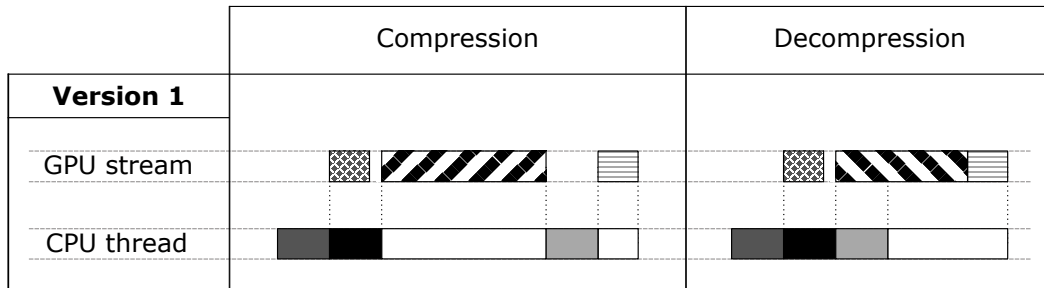


Figure 5.5: Execution trace for Version 1 (V1). The legend is shown in Figure 5.2.

Figure 5.6 shows the compression and decompression speed achieved by the previous V0 implementation and the new V1. As we can see, the new V1 implementation obtains worse results than the ideal V0. We expected this behavior because V1 transfers all data from host to device memory before kernel initialization, and from device to host memory after kernel finalization. These transfers add latency to compression and decompression executions, resulting in a reduction in speed.

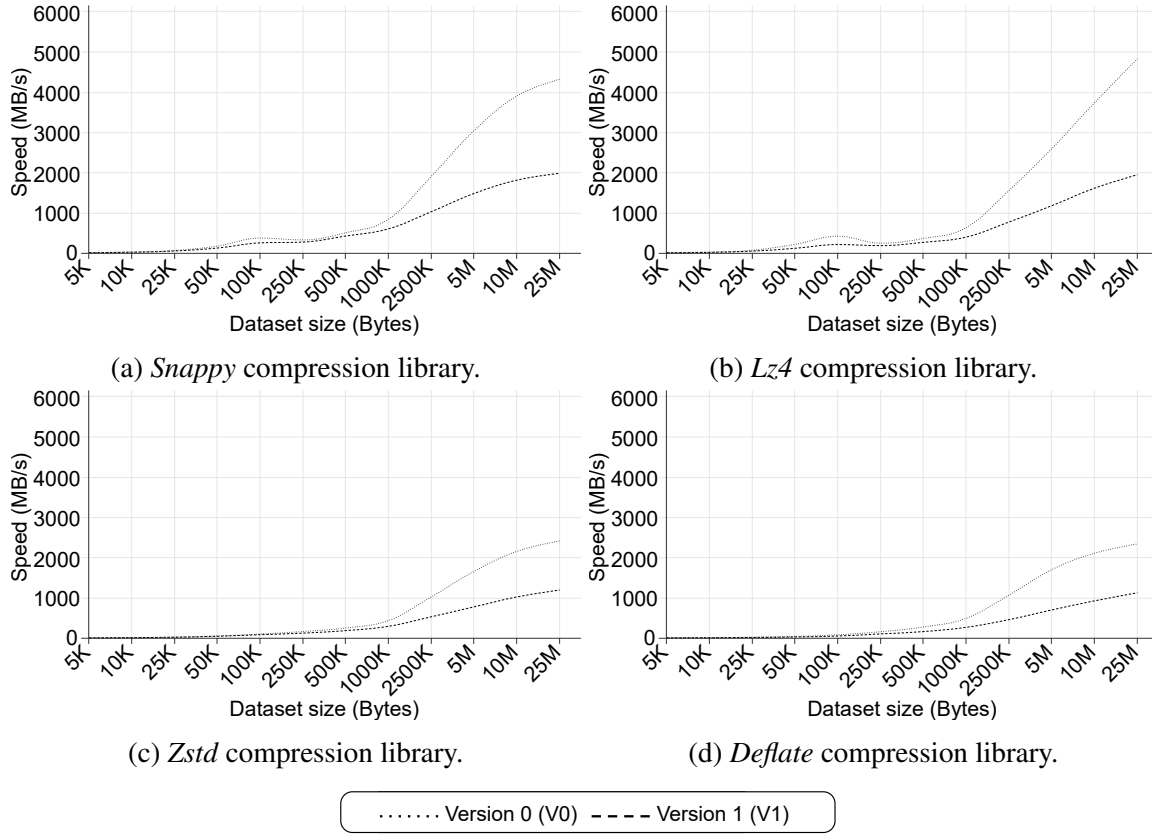


Figure 5.6: Comparison of the compression and decompression speed achieved by Version 0 (V0) and Version 1 (V1).

5.2.3 Version 2 (V2): GPU compression pipeline

The previous V1 implementation follows a naive approach. First, data to be compressed/decompressed is transferred from host memory to device memory. Then, once the entire data is located in device memory, it is compressed/decompressed using a kernel. Finally, the result is returned to the host memory. Despite using a GPU, this version does not take advantage of its high performance. GPUs are powerful devices that allow data transfers to overlap with kernel executions. Therefore, it is possible to follow a pipelined approach where: (i) copy part of the data from host 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 memory. Using this approach, data transfers may be hidden by kernel executions. However, implementing this idea increases noticeably compression and decompression complexity.

Next, we present this pipelined approach. First, we introduce the pipeline. Then, we analyze the optimal chunk size for the pipeline. Finally, we analyze the overall performance once the optimal chunk size has been configured.

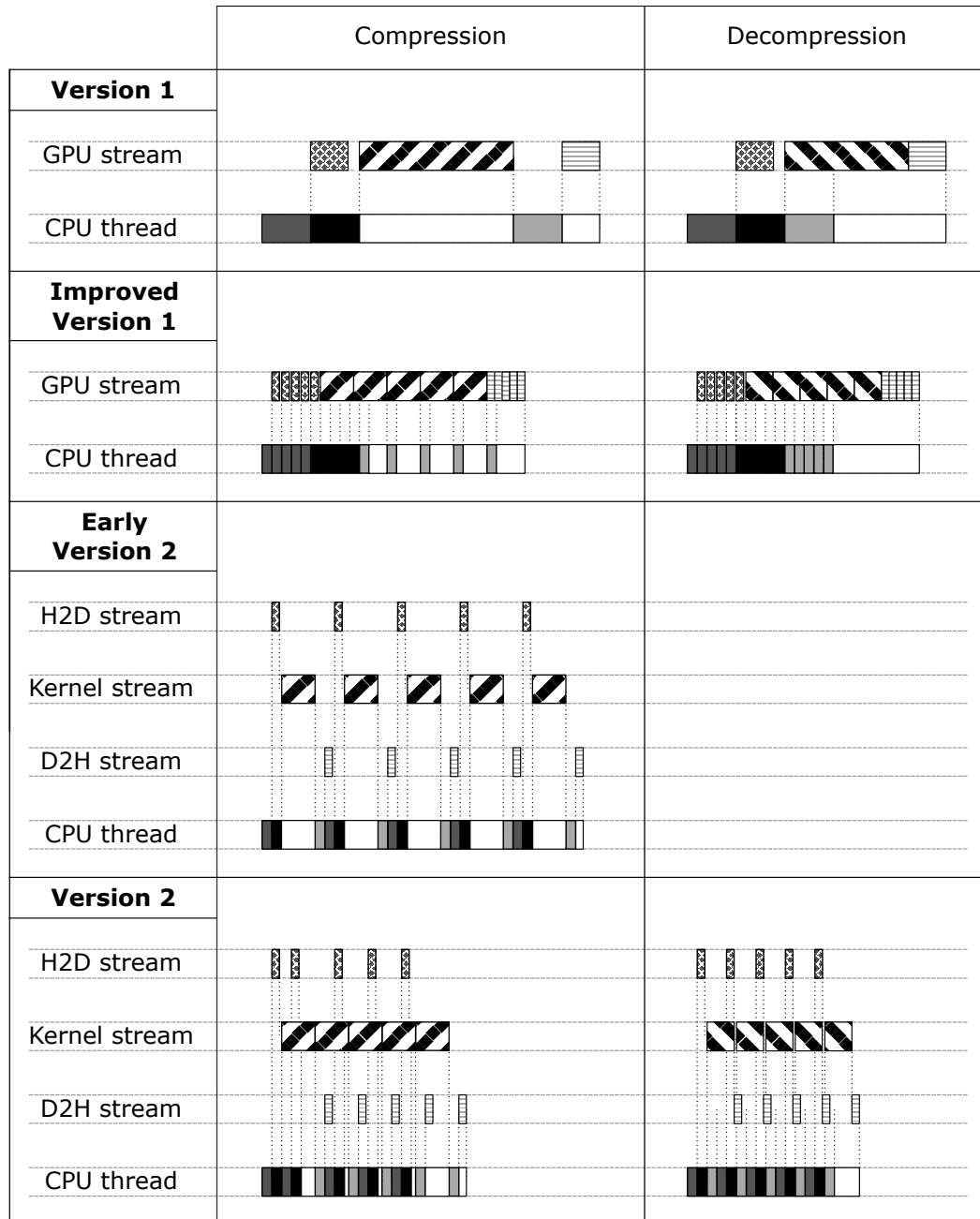


Figure 5.7: Execution traces for compression of Version 1 (V1), Version 2 (V2), and intermediate versions. The legend is shown in Figure 5.2.

Moving from a naive to a pipelined implementation

Figure 5.7 illustrates the route followed to achieve this pipelined V2 implementation from the naive V1. At the top of the figure, we show again the V1 for completeness. The first step was the “Improved Version 1”, an enhancement of the V1 implementation. Focusing on compression, this version splits the data into chunks before compressing. First, the CPU thread schedules all the H2D copies. Once the last H2D transfer is done, GPU compression kernels are set. When the last kernel has been prepared for running on the GPU stream, the CPU thread prepares D2H transfers. Note that all the operations are scheduled and run in the GPU stream sequentially. Thus, the first kernel does not start until the last H2D transfer has been completed, and the same for the first D2H transfer. Therefore, data copies and kernel executions do not overlap yet. Moreover, remember that the size of the compressed chunk is only known once the compression kernel finishes. As a result, in compression, the CPU thread must synchronize each kernel before calling its specific D2H copy. Furthermore, at the end of the process, the resulting data must include the results of all the chunks appropriately ordered.

“Early Version 2” in Figure 5.7 is the first attempt to overlap data copies with kernel executions. This implementation uses two additional streams in the GPU: the H2D stream and the D2H stream. The role of the H2D stream is to transfer the data to be processed by the GPU from host memory to device memory. On the other hand, the D2H stream is responsible for copying the processed data back to host memory. The idea is to distribute the different operations to process them simultaneously by the GPU. However, there is still a problem with this implementation. The D2H transfer cannot begin until its specific compression kernel ends. The reason for this is that the size of the compressed data is only known at that time. As a result, the CPU thread, which manages the overall execution and coordination of the GPU tasks, has to wait for the kernel to complete before it can proceed with the next chunk. This waiting period hampers the overall efficiency of the process. Despite using multiple streams in the GPU (H2D, D2H, and kernel one), data copying and kernel executions do not overlap. Thus, the system cannot fully utilize the parallelism and potential performance benefits that could be achieved by overlapping these tasks.

Finally, the desired overlap is achieved in “Version 2”, shown in Figure 5.7. The compression process starts by scheduling the compression of two chunks of data (this includes performing H2D copies and kernel executions). Once the compression of the first two chunks is set, the CPU thread responsible for managing the compression process waits for the result of the first chunk. In other words, it pauses and waits until the first kernel ends. When the first

chunk has been compressed, the CPU thread schedules its D2H transfer. Importantly, this scheduling is done while the compression of the second chunk is still ongoing. Also, these operations are run in different streams, which means that while the second chunk is being compressed, the compressed data of the first chunk is simultaneously being copied back to the host. By overlapping the D2H copy for the first chunk with the compression kernel of the second chunk, the system can make more efficient use of time and resources. Additionally, while the D2H copy for the first chunk has been set, the compression of the third chunk is scheduled. Overall, this approach aims to optimize the utilization of resources and minimize waiting time by overlapping data copies and kernel executions whenever possible.

In the case of decompression, the evolution followed is also presented in Figure 5.7. Compression and decompression are similar, although the latter does not need to synchronize after kernel completions, 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”. In addition, 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.

Optimal chunk size for the pipeline

As commented, the V2 implementation follows a strategy where the data to be compressed are split into chunks. The size of these chunks is vital for achieving high performance. On the one hand, small information is transferred to GPU faster than big ones. On the other hand, GPUs enable a significant number of kernel threads, and chunks with big sizes could benefit more from GPU computing power than small ones. Consequently, finding the best chunk size is the key to optimizing the performance of the parallel pipeline in V2.

Figure 5.8 illustrates the experiment carried out to find the optimal chunk size. We selected a large enough data to compress in order to observe the possible differences in the performance of the implemented pipeline: the 25MB data from the AI dataset. We can appreciate the speed of each GPU-based compression library when compressing and decompressing this data using different chunk sizes. The range of the chunk size used in these experiments is from 64KB to 32MB. All GPU-based libraries start with a low speed when the smallest chunk size is used. However, that speed increases as the chunk size does until achieving a performance peak with 8MB sizes for *Zstd*, *Snappy*, and *Lz4* compression libraries and 16MB sizes for the *Deflate* one. For sizes larger than those, the speed starts to decrease. Notice that the pipeline works when the data to be compressed is larger than the selected chunk sizes. Therefore, no pipeline

version will be used with smaller data. Nevertheless, despite not using the pipeline with these sizes, it works better than the V1 implementation. The V1 implementation allocates GPU memory before each compression/decompression and frees it afterward, while the new pipeline implementation only pre-allocates GPU memory to develop the pipeline. Reducing the number of calls to the system reduces the compression/decompression time.

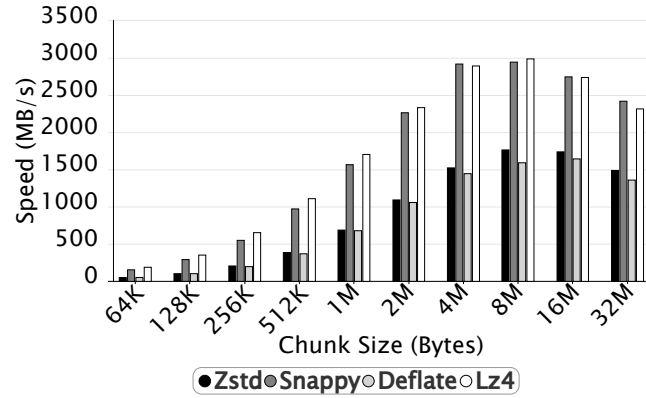


Figure 5.8: Exploration to find the optimal chunk size for Version 2 (V2) using the 25MB data obtained from the AI dataset.

Performance of the pipelined compression V2

Figure 5.9 compares the speed performance of the new version (V2) with previous versions (V1 and V0) using different compression libraries. Regardless of the compression library, V2 consistently outperforms V1 and approaches the ideal V0 performance. The V0 values represent the best possible performance without any compression overhead, i.e., without any data transfers. More specifically, when using the *Snappy*, *Lz4*, *Zstd*, and *Deflate* compression libraries, the V2 implementation performs better than V1 for data larger than particular sizes. For example, for data larger than 1000KB, V2 outperforms V1 when using the *Snappy*, and *Lz4* libraries. Similarly, for data larger than 2500KB, V2 performs better than V1 when using the *Zstd*, and *Deflate* libraries. It is worth noting that this performance improvement is not limited to data sizes, i.e., if they are equal to or larger than the chunk size used in the pipeline (16MB for *Deflate* and 8MB for *Snappy*, *Lz4*, and *Zstd*). Even for data smaller than this chunk size, V2 demonstrates better performance than V1. This improvement is possible thanks to the pre-allocated space for chunks in GPU memory which the V2 implementation does. This allocation strategy enables efficient handling of data, resulting in improved performance even for smaller data.

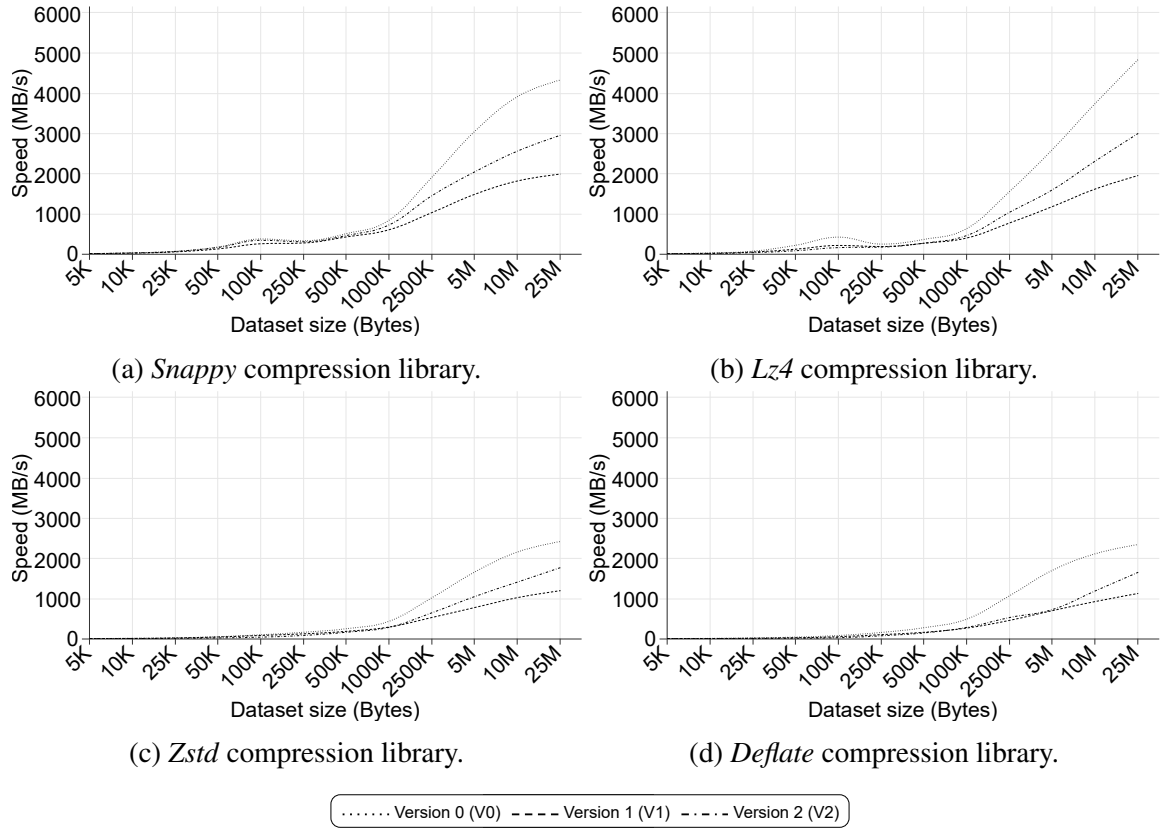


Figure 5.9: Comparison of the compression and decompression speed achieved by Version 0 (V0), Version 1 (V1), and Version 2 (V2).

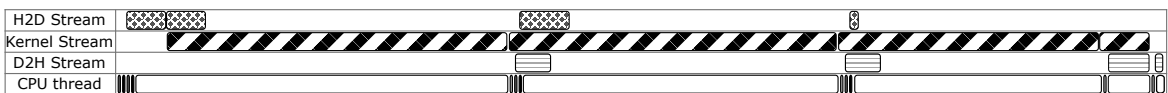


Figure 5.10: Execution trace of *Snappy* using Version 2 (V2) when **compressing** the 25MB data obtained from the AI dataset using a chunk size of 8MB. This trace was obtained using the NVIDIA Nsight Systems profiler. The legend is shown in Figure 5.2.

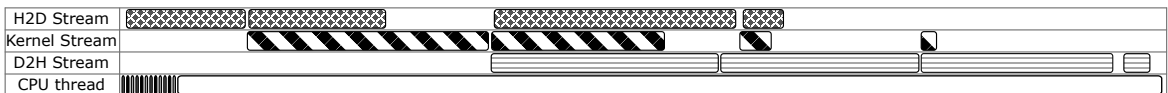


Figure 5.11: Execution trace of *Snappy* using Version 2 (V2) when **decompressing** compressed the 25MB data obtained from the AI dataset using a chunk size of 8MB. This trace was obtained using the NVIDIA Nsight Systems profiler. The legend is shown in Figure 5.2.

To conclude the analysis of the V2 implementation, we have used the NVIDIA Nsight Systems profiler [81] to analyze the execution traces of *Snappy* V2 when compressing (Figure 5.10) and decompressing (Figure 5.11) the 25MB data obtained from the AI dataset and using a chunk size of 8MB. On the compression trace, we observe how the kernel execution consumes almost all the execution time. Kernel executions run concurrently with H2D and D2H copies. Thus, we can confirm that data transfers are hidden by kernel computations. On the decompression trace, we can appreciate how the kernel execution takes a similar time to H2D and D2H copies. Nevertheless, kernel execution and transfers also overlap. Notice that decompression kernels are faster than compression ones. The entire decompression process takes about four times less than compression.

5.2.4 Version 3 (V3): multi-stream GPU compression

The V2 implementation distributes the different tasks between three streams to create the parallel pipeline: (i) one stream performs H2D transfers, (ii) another stream runs kernels, and (iii) the last stream carries out D2H transfers. These streams run their tasks concurrently. In this new Version 3 (V3) implementation, we explore the use of multiple streams to run these tasks, i.e., multiple H2D, D2H, and kernel streams. However, this approach raises several concerns about concurrency. If we use multiple streams to run these tasks, H2D transfers will start approximately simultaneously, the same for kernel executions and D2H transfers. This raises the concern about whether several transfers (H2D and D2H) can be performed concurrently if they work in the same direction. Another concern is the concurrent execution of kernels.

Regarding concerns about simultaneous data transfers, we demonstrated in V2 implementation that H2D and D2H copies can run concurrently. However, there may be a limitation on the number and kind (H2D or D2H) of data copies that can be overlapped when using multiple streams. In Listing 5.2, we present a synthetic test created to analyze the CUDA behavior when working with data transfers done in the same direction using different streams and the same GPU. First, twenty streams are created. Then, each one performs an H2D transfer. Once all streams have performed the H2D transfer, streams are synchronized. Finally, one asynchronous D2H transfer is scheduled in each stream.

```
int number_of_streams = 20;
int data_size = (32*1024*1024); // Each stream will copy 32MB
cudaStream_t cuda_streams[number_of_streams];
```

```

char *device_memories[number_of_streams];

// Allocate pinned host memory using the cudaHostAlloc function
char *host_source_memories[number_of_streams];
char *host_destination_memories[number_of_streams];

// Launch all the H2D copies simultaneously
for (int i = 0; i < number_of_streams; ++i) {
    cudaMemcpyAsync(device_memories[i], host_source_memories[i],
        data_size, cudaMemcpyHostToDevice, streams[i]);
}
cudaDeviceSynchronize(); //Wait for the completion of all copies

// Launch all the D2H copies simultaneously
for (int i = 0; i < number_of_streams; ++i) {
    cudaMemcpyAsync(host_destination_memories[i], device_memories[i],
        data_size, cudaMemcpyDeviceToHost, streams[i]);
}
cudaDeviceSynchronize(); // Wait for the completion of all copies

```

Listing 5.2: 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.

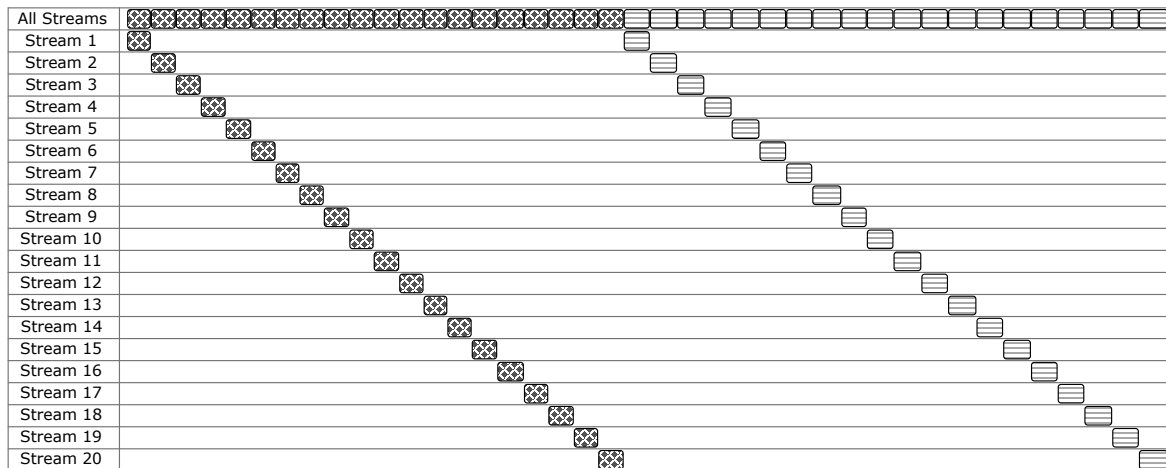


Figure 5.12: Execution trace of the synthetic test in Listing 5.2 to check if data copies overlap when using multiple streams and the same kind of transfers. This trace was obtained using the NVIDIA Nsight Systems profiler. The legend is shown in Figure 5.2.

Figure 5.12 shows 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 transfers of the same kind (H2D or D2H) do not overlap. Therefore,

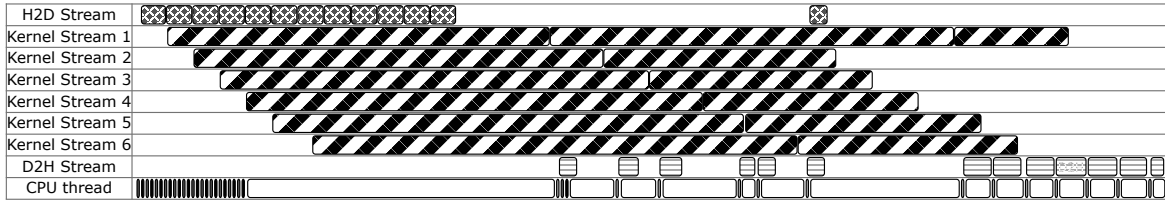


Figure 5.13: Execution trace of *Snappy* using Version 3 (V3) when **compressing** the 25MB data obtained from the AI dataset, using a chunk size of 2MB and up to six concurrent kernels. This trace was obtained using the NVIDIA Nsight Systems profiler. The legend is shown in Figure 5.2.

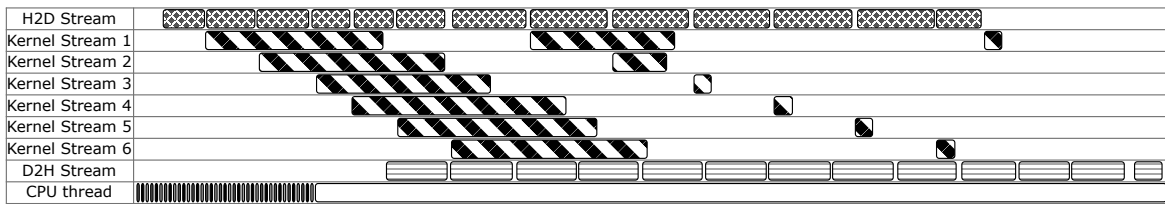


Figure 5.14: Execution trace of *Snappy* using Version 3 (V3) when **decompressing** the 25MB data obtained from the AI dataset, using a chunk size of 2MB and up to six concurrent kernels. This trace was obtained using the NVIDIA Nsight Systems profiler. The legend is shown in Figure 5.2.

increasing the number of H2D and D2H streams to create a new V3 will not present any improvement with respect to V2 implementation.

We have demonstrated that multiple transfers cannot perform simultaneously if they run in the same direction. However, kernel executions take longer than data transfers as we found in the V2 experiments. Furthermore, these kernels do not usually take full advantage of the high computing power of modern GPUs. Thus, running multiple kernels can reduce the overall execution time by increasing the GPU utilization. Unlike the previous study of copies, we have improved V2 by creating multiple streams for running multiple kernels simultaneously. Thus, the new parallel pipeline in “Version 3 (V3)” follows the next strategy: (i) one stream performs H2D transfers, (ii) multiple streams run kernels concurrently, and (iii) the last stream does D2H transfers.

Figure 5.13 shows the execution trace of this enhanced pipeline when running *Snappy* for compressing the 25MB data obtained from the AI dataset, using a chunk size of 2MB. Six GPU streams were created to launch up to 6 compression kernels in parallel. The compression kernels from different streams overlap as expected. As a result, the total kernel execution is reduced, which also reduces overall compression time. Figure 5.14 presents a similar trace when the *Snappy* library is used for decompression where decompression kernels also

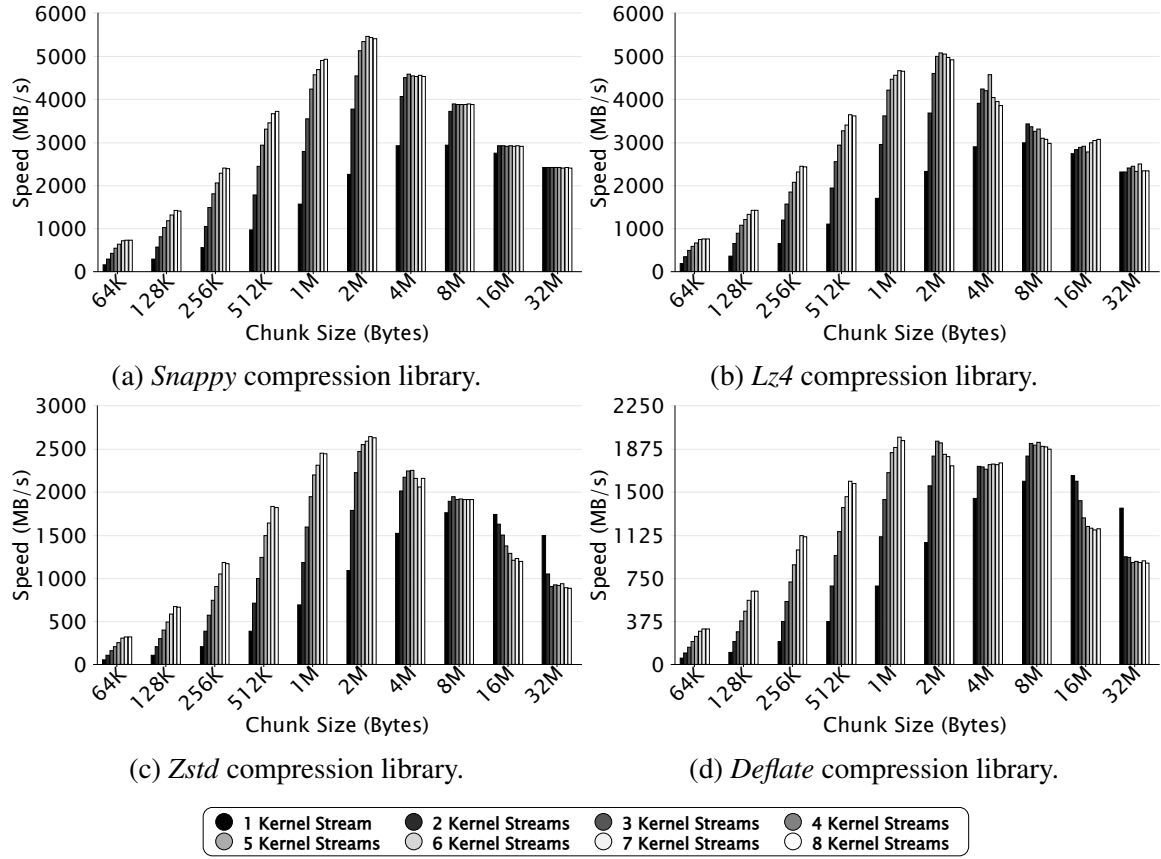


Figure 5.15: Exploration to find the optimal chunk size for Version 3 (V3) using the 25MB data obtained from the AI dataset and varying the number of kernel streams.

overlap. However, the improvement is not as noticeable as when compressing because kernel execution time is shorter.

Due to the increment in the number of streams used to compress/decompress data, the optimal chunk size used in the V3 implementation could not be the same as in the V2 implementation. For this reason, we have used again the 25MB data obtained from the AI dataset using all the GPU-based compression libraries and different chunk sizes. Figure 5.15 shows the speed achieved by using the V3 implementation 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 shown in the figure (the lighter the color, the more kernels running concurrently). Figure 5.15a 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 5.15b, the best result is obtained with five kernel streams and a chunk size of 2MB. *Zstd* (Figure 5.15c) gets the best performance using seven streams and a



Figure 5.16: Comparison of the compression and decompression speed achieved by Version 0 (V0), Version 1 (V1), Version 2 (V2), and Version 3 (V3).

chunk size of 2MB. *Deflate* (Figure 5.15d) 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*.

Finally, Figure 5.16 presents the speed results obtained with the V3 implementation. As expected, V3 demonstrates significant improvements over the previous V1 and V2 implementations. Specifically, when using the *Snappy*, *Lz4*, and *Zstd* compression libraries, the performance of the new V3 implementation surpasses even the ideal V0 for some data sizes. For the *Deflate* compression library, the performance of V3 is very close to the ideal V0. Even though the idea was not to improve the results obtained by the ideal version, V3 does because it utilizes the computing power of the GPU more efficiently by running multiple concurrent kernels. This means that V3 can execute multiple compression tasks simultaneously, taking full advantage of the parallel processing capabilities of the GPU. As a result, V3 outperforms

previous versions and even achieves better performance than the ideal V0 in some cases, hiding completely data transfers.

5.2.5 Version 4 (V4): multi-GPU compression

Previous versions have been enhanced using only one GPU. However, many computers in data centers today have multiple GPUs. Thus, we have developed a new “Version 4 (V4)”, which distributes compression/decompression computations across all the available GPUs in the system. The introduction of multiple GPUs offers two significant advantages. Firstly, an increase in the total computing power available for compression and decompression tasks. By utilizing several GPUs simultaneously, V4 can use the combined processing capabilities of these GPUs, resulting in improved performance. Secondly, each GPU can independently perform data copies simultaneously in the same direction (H2D or D2H). This means that the data to be compressed/decompressed can be copied faster to kernels. Moreover, the resulting data can be copied back to the host memory from the computation kernels faster. This concurrent data copying capability enhances the overall efficiency of the compression process. However, supporting multiple GPUs also introduces complexity. Data chunks that need to be compressed/decompressed must be distributed among the available GPUs to enable parallel processing. In addition, the resulting data must be stored in the host memory in the correct order and a contiguous memory region once kernels have completed their computations. This requires careful orchestration by the CPU thread before scheduling the D2H transfer. Thus, the CPU thread must be improved to manage the distribution of data chunks and handle the complexities associated with multiple GPUs.

Figure 5.17 and Figure 5.18 show the execution trace of *Snappy* V4 when compressing and decompressing the 25MB data obtained from the AI dataset, using a chunk size of 2MB, 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 5.18 have fewer overlaps because the execution time of these kernels is shorter than that of compression ones.

Figure 5.19 illustrates the speed-up obtained when the V4 implementation is used instead of the V3 one. The main difference between these two implementations is the number of GPUs used. While the V3 implementation uses one A100 GPU, the V4 one uses two A100 GPUs. Note that the optimal number of kernel streams (used to run kernels concurrently)

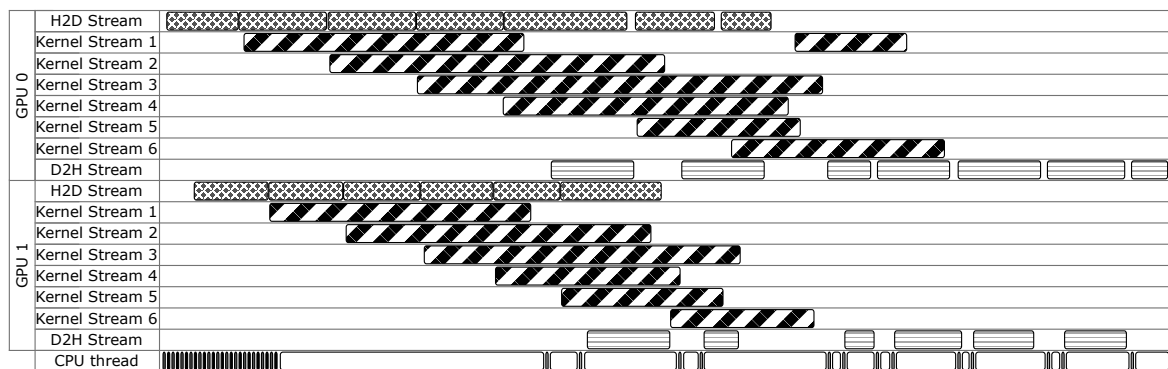


Figure 5.17: Execution trace of *Snappy* using Version 4 (V4) when **compressing** the 25MB data obtained from the AI dataset, using a chunk size of 2MB, up to six concurrent kernels and two GPUs. This trace was obtained using the NVIDIA Nsight Systems profiler. The legend is shown in Figure 5.2.

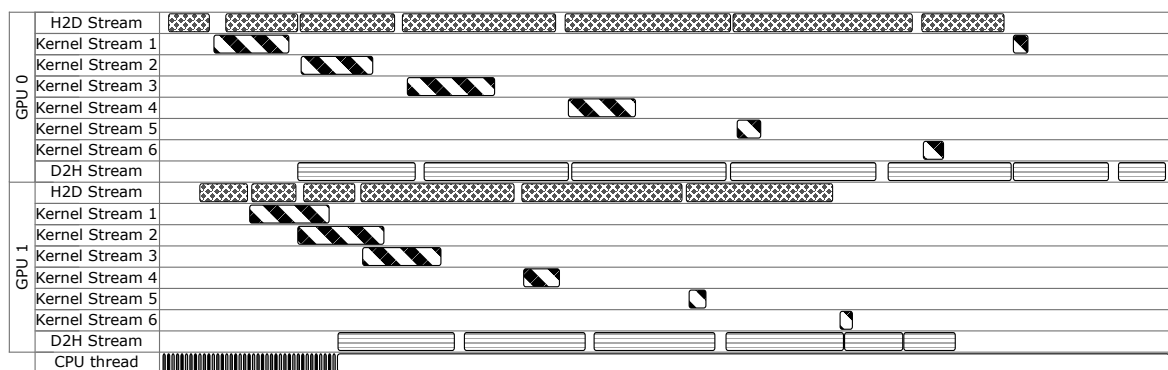


Figure 5.18: Execution trace of *Snappy* using Version 4 (V4) when **decompressing** the 25MB data obtained from the AI dataset, using a chunk size of 2MB, up to six concurrent kernels and two GPUs. This trace was obtained using the NVIDIA Nsight Systems profiler. The legend is shown in Figure 5.2.

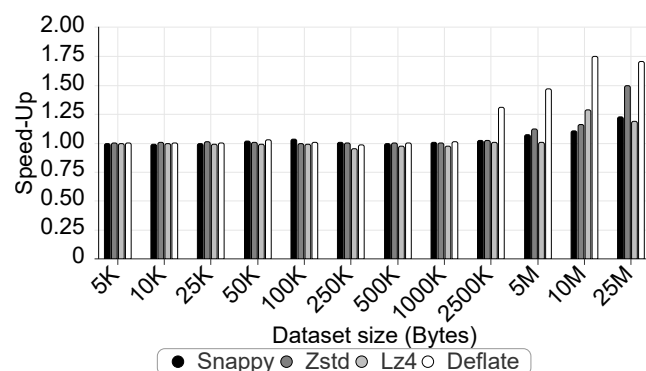


Figure 5.19: Speed-up obtained by Version 4 (V4) with respect to Version 3 (V3) when compressing and decompressing the AI dataset using two and one A100 GPU respectively.

and the optimal chunk used are the same for these two versions and were selected in the previous study when the V3 implementation was presented. 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 improves V3 for data larger than 2MB. This makes sense because these compression libraries use a chunk size of 2MB. Thus, multiple GPUs are used for data sizes over that chunk size. *Deflate* presents a similar behavior but for data larger than 1MB, which is the chunk size used for this library. The best speed-up (up to 1.7x) is obtained by *Deflate*, followed by *Zstd* (up to 1.5x). *Snappy* and *Lz4* present the lower speed-up (up to 1.2x and 1.3x, respectively). The theoretical maximum speed-up would be 2x because we use two GPUs. However, the complexity of this new version and the GPU-based compression library behaviors make it difficult to obtain this speed-up.

5.3 A heterogeneous CPU-GPU compression library

In previous sections we have analyzed the performance of CPU- and GPU-based compression library independently. Given that some libraries are implemented in both CPU and GPU, in this section we compare these different implementations by studying both implementations in more depth. Using these results, we introduce a heterogeneous CPU-GPU compression library that dynamically selects the best implementation depending on the data.

5.3.1 Performance comparison of CPU- and GPU-based compression

Next we compare the compression libraries that have already been used in previous experiments and that are implemented on both CPU and GPU. These libraries are the following ones: (i) *Lz4*, (ii) *Snappy*, (iii) *Deflate*, and (iv) *Zstd*. For the GPU-based compression libraries, we used the Version 3 (V3) implementation presented in Section 5.2.4. This version uses only one GPU, which is the most typical scenario. We use one A100 NVIDIA GPU for the experiments and the AI datasets presented in Chapter 3.

Notice that although the same compression libraries are used for CPU and GPU implementations, the compressed data obtained is not the same. During the compression/decompression, CPU-based libraries usually process all the data together. However, GPU-based libraries split data into chunks and compress/decompress them independently using GPU kernels. This has already been explained in Figure 5.1, which compares the typical approaches followed by

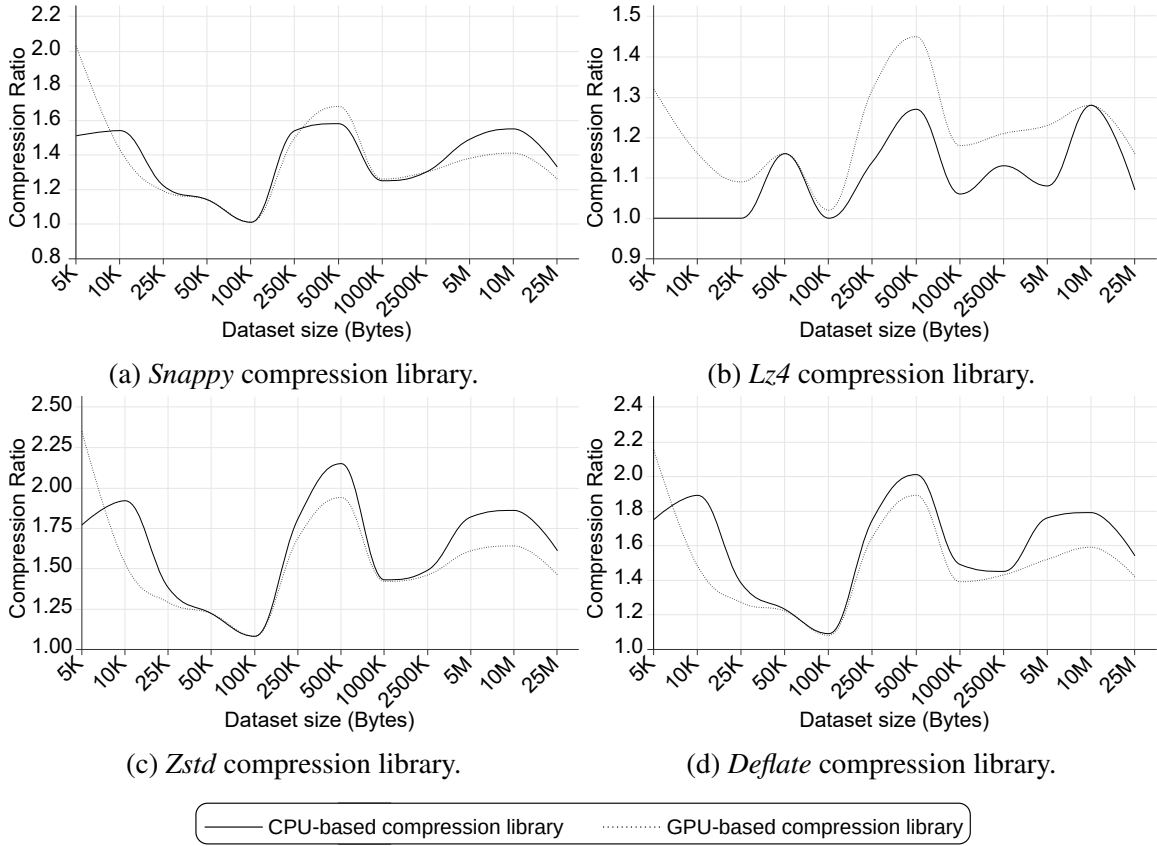


Figure 5.20: Comparison of the compression ratio achieved by compression libraries using their CPU- and GPU-based implementations.

CPU- and GPU-based implementations. The immediate consequence of splitting data into chunks is that the source data for the compression process are not the same, and thus, the result obtained is different.

In Figure 5.20, we can observe the compression ratio achieved by the compression libraries when running on CPU and GPU. This ratio was formally defined in Equation 3.1 as the ratio between the uncompressed and compressed data sizes. As expected, the compression ratio obtained differs, but the trend followed is similar. The results show that the CPU-based implementations obtain a better compression ratio than the GPU-based ones for almost all libraries. The CPU-based implementation results are created with the information of the whole data. However, the GPU-based implementations result contain different independent compressions subsequently joined together. Each compression utilizes the information of the chunk of data used at that moment. Thus, it is expected for CPU-based compression libraries to obtain better compression ratios than GPU-based ones. However, this is not the case for the *Lz4* compression library (see Figure 5.20b). In that case, the GPU-based implementation

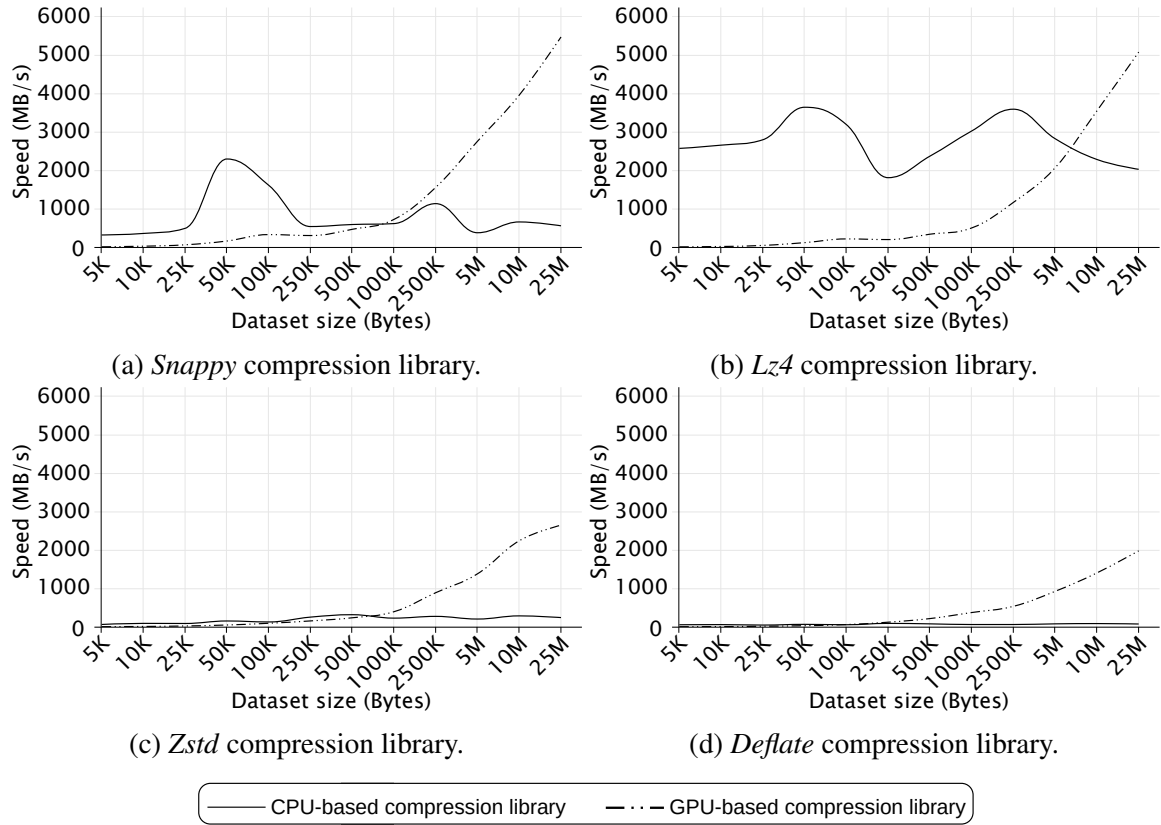


Figure 5.21: Comparison of the compression and decompression speed achieved by CPU- and GPU-based (V3) implementations.

obtains a better compression ratio. The reason is that the CPU-based compression library runs with a specific flag set to increase the speed at the expense of the compression ratio. We used this flag to prioritize the compression/decompression speed because we wanted to compare it with the high performance of the GPU. Note also that a library implemented on CPU and GPU does not necessarily have the same configuration parameters. For example, the *Lz4* compression library has this flag to increase speed performance in its CPU implementation, but it is not available in the GPU version.

Figure 5.21 shows the speed obtained by CPU- and GPU-based compression libraries. The speed was defined in Equation 4.1 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. We can see that CPU-based implementations present better results than GPU-based ones when small data sizes are used. The longer GPU setup time penalizes compression for small data sizes. However, as the data size increases, the GPU setup time is compensated by the shorter computing time of GPU-based implementations. Thus, GPU-based implementations outperform CPU-based ones for large data sizes. More specifically, the GPU is beneficial

when data to compress is greater than 1000KB, 10MB, 1000KB, and 100KB, for *Snappy*, *Lz4*, *Zstd*, and *Deflate* compression libraries, respectively.

These results reveal a minimal effect on compression ratio when using CPU- or GPU-based compression libraries. On the contrary, there are noticeable differences in the compression and decompression speed. Thus, when the data to compress is small, CPU-based compression libraries usually achieve higher speed. However, GPU-based compression libraries gradually improve as the data size increases and end up delivering better performance. Specifically, Table 5.2 presents the thresholds of the data size used by each compression library from which the GPU-based implementation achieves better results than the CPU-based ones.

Compression Library	Data Size Threshold
<i>Snappy</i>	500KB
<i>Lz4</i>	5MB
<i>Zstd</i>	500KB
<i>Deflate</i>	50KB

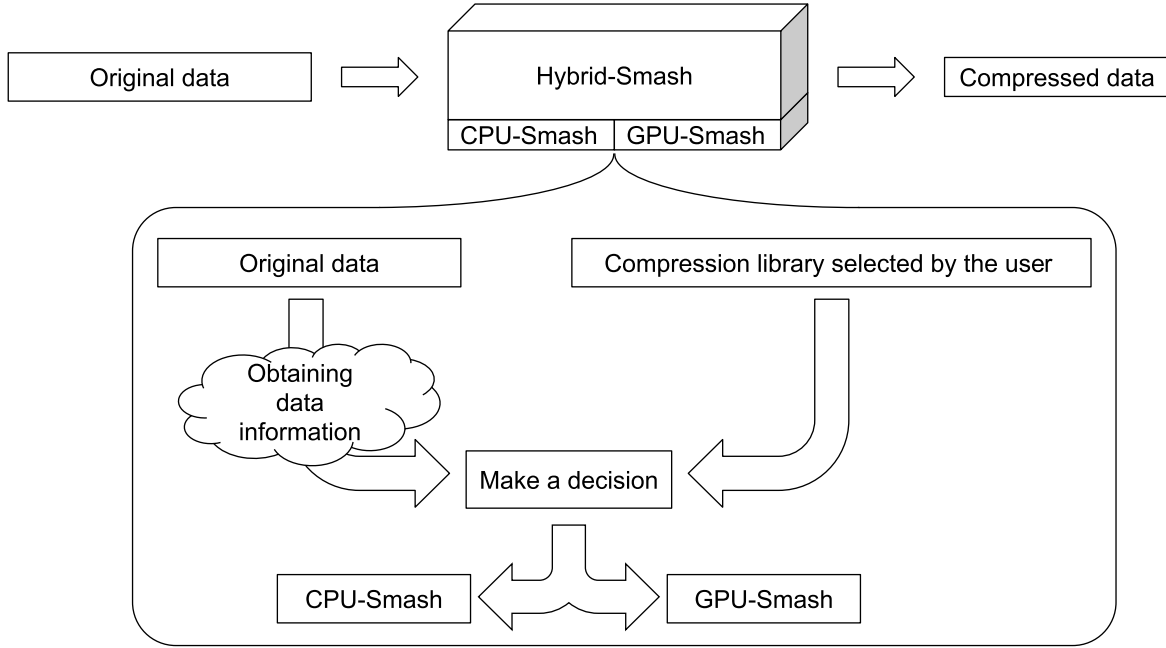
Table 5.2: Thresholds of the data size used to decide when using the CPU or the GPU with a specific compression library.

5.3.2 Hybrid-Smash

With the conclusions drawn from the previous performance comparison, we have created a new heterogeneous CPU-GPU abstraction compression library called *Hybrid-Smash*. It uses the CPU or the GPU version of the compression libraries depending on which one achieves the best performance. Similar to previous libraries, this library offers a flexible API to enable easy integration of real-world applications. This API also allows users to specify the specific configuration parameters required by each compression library.

The compression libraries included in this new abstraction library are the ones implemented on both CPU and GPU, namely (*Lz4*, *Snappy*, *Deflate*, and *Zstd*). The configuration parameters for each compression library are the same as the ones described in Section 4.1.2 and Section 5.1.2 when presenting the compression libraries used by the *CPU-Smash* and *GPU-Smash* abstraction libraries.

Using the *Hybrid-Smash* abstraction compression library, the user only needs to select the compression library and the configuration parameters. Figure 5.22 illustrates the workflow of this new abstraction compression library. The decision to use the *CPU-Smash* or the *GPU-Smash* abstraction compression library depends on two factors: (i) the size of the

Figure 5.22: *Hybrid-Smash* workflow.

data to compress, and (ii) the selected compression library. The abstraction compression library makes that decision transparently to the user. Thus, the user does not need to be concerned about the particular configuration parameters of each compression library. Internally, the *Hybrid-Smash* abstraction compression library decides whether to use the GPU-based implementation or the CPU-based one depending on the data size and the thresholds previously presented in Table 5.2 for each compression library.

5.4 Summary

We have created a new benchmark called *GPU-Smash*. This benchmark includes the GPU-based compression libraries developed by NVIDIA. It also provides an API for users where different kinds of compression can be deployed depending on (i) where the data to be compressed/decompressed is located, and (ii) where the resulting data will be stored. The available kinds of compression are: host-to-host (H2H), host-to-device (H2D), device-to-host (D2H), and device-to-device (D2D).

In principle, existing GPU-based compression libraries only work using device memory (D2D compression). To address this concern, we have introduced and evaluated different versions to develop an efficient H2H compression approach. Although we have only focused

on H2H compression, D2H and H2D compression/decompression behave similarly. Thus, in the D2H one, the transfers to the GPU memory at the beginning are avoided, while in the H2D one, the ones from the GPU memory at the end.

The first version presented (V0) was the ideal one. This version is a D2D compression and is the result of using the *nvCOMP* library without any improvement. Starting from this V0 implementation, we created a naive V1 one, where all data is transferred to the GPU. Then, once the data are in the device memory, the kernel compresses/decompresses these data. Finally, the resulting data is copied back to the host memory after the kernel execution ends. As expected, the V1 performance is worse than the V0 one because it contains transfers before and after the kernel execution. This makes the overall execution take longer.

The next version (V2) was a pipelined implementation. This version uses different streams to divide the tasks done by compression/decompression processes. We create three different streams: (i) the H2D stream, which focuses on H2D transfers; (ii) the kernel stream, which manages kernel executions; and (iii) the D2H stream, which handles D2H transfers. This V2 implementation enhances the previous V1 thanks to the pipeline created, achieving a speedup of over 1.4x.

Then, we created a multi-stream implementation (V3). This version is similar to the previous one V2, but we increase the number of streams used to run kernels. This increase allows GPU-based compression libraries to run multiple kernels concurrently, improving previous results by more than 2.5x. It is worth noting that it even improves on the ideal results of V0. Certainly, running multiple kernels concurrently allows libraries to completely hide the transfers done from/to GPU. Moreover, it also allows other kernel executions.

Finally, we developed an implementation that uses multiple GPUs (V4). In this version, transfers done in the same direction (H2D or D2H) can be done simultaneously if they are done from/to different GPUs. This allows for more concurrency than in previous versions, resulting in an improvement of up to 1.7 times compared to V3.

Depending on the GPU-based compression library, the configuration used by these implementations can be different. Since each library follows different strategies, it is necessary to investigate each compression library to obtain the best results. Thus, we have demonstrated how to get the best configuration for each specific GPU-based compression library, increasing the speed by more than 2.5x. Moreover, a multi-GPU version was also presented, achieving an additional speedup of 1.7x.

After evaluating the different GPU-based implementations, we compared their performance with their corresponding CPU-based versions. The result was that CPU-based implementations tend to obtain a better compression ratio than GPU-based ones. This happens because GPU-based libraries split data into chunks and then compress/decompress each chunk independently. These chunks contain less information than if all the data were processed together. For this reason, the compression ratio is usually lower for GPU-based versions. However, GPU-based implementations performed better than CPU-based ones when data to compress were larger than a threshold size. More specifically, GPU-based libraries performed better than CPU-based ones when the data to compress are greater than 1000KB, 10MB, 1000KB, and 100KB for *Snappy*, *Lz4*, *Zstd*, and *Deflate* compression libraries, respectively. Therefore, the high performance of the GPU does not seem to be beneficial when the data to compress/decompress have smaller sizes because the GPU execution introduces some latency.

Finally, based on the knowledge acquired during this study, we have created the *Hybrid-Smash* abstraction compression library. *Hybrid-Smash* is a heterogeneous CPU-GPU compression library that transparently uses CPU or GPU compression depending on the size of the data to be compressed and the compression library to be used.

Chapter 6

Improving network communications using compression libraries

AI has marked a beginning and an end in computer science, becoming part of our daily life, with tools such as voice assistants or recommendation systems. These applications are tied to powerful computers equipped with last-generation GPUs [88]. Nevertheless, although this execution environment is acceptable for many application domains, there are other domains where lighter devices could be more advisable, where there is no GPU or the CPU is not that powerful. There are different approaches to increase the computing capabilities of these domains. In this thesis, we focus on the remote virtualization of accelerators [46, 15, 51]. This mechanism transparently provides GPUs located at remote nodes to applications that run in a node without GPUs.

In particular, this chapter focuses on real-world AI applications using rCUDA and compression libraries to boost client-server communication in IoT environments. To that purpose, we implement an approach to perform on-the-fly compression. It requires adapting the existing communication pipeline used in rCUDA to leverage compression libraries. This hides the compression time with the GPU transfer time by splitting the data into chunks and working with them separately. The final result is an optimized parallel pipeline to perform on-the-fly compression.

6.1 Introduction

IoT environments usually involve a large number of devices generating significant amounts of data that need to be processed. These devices are typically low-performance and energy-efficient. Consequently, the data they produce is often sent to a remote data center for processing.

Edge computing has emerged as a significant advancement for IoT [98, 105, 95] by enabling data processing closer to the data source instead of relying on communication over the network to reach a remote data center. This approach addresses the limitations of low-bandwidth networks in IoT systems, such as WiFi or cellular networks, by processing data at the network edge.

However, edge devices may lack sufficient computing power to support some applications, including machine learning applications that rely on GPUs and CUDA technology [82]. The challenge lies in providing powerful GPUs to edge devices. Remote GPU virtualization [104, 31] is a mechanism that addresses this challenge by allowing the sharing of a GPU located in one computer with processes executing on other computers. With this mechanism, a CUDA application running on an edge device without a GPU can access a GPU located in another computer without being aware of its remote nature. This technology has been a significant advancement in edge computing [51].

The performance of applications using remote GPU virtualization systems depends heavily on the network throughput between the node where the application is run and the remote server where the GPU is located. However, the network performance limitation in edge computing scenarios can reduce the benefits of using a remote GPU.

On-the-fly compression can be applied to the data exchanged with the remote GPU to overcome the overhead of using a remote GPU over a low-performance network. On-the-fly compression involves compressing the data before transmitting and decompressing it upon receipt. This compression and decompression process occurs transparently within the communication layer of the remote GPU virtualization system without the CUDA applications being aware of it. Currently, there is no remote GPU virtualization framework that incorporates on-the-fly compression. However, some experiments have focused on improving network bandwidth.

Liang and Li [63] and Uthayakumar et al. [114] focus on wireless sensor networks. Due to the limitations of these devices on energy, bandwidth, memory, and processing capabilities,

Liang and Li [63] compare a new proposal with popular lossless compression algorithms analyzing power and robustness. Alternatively, Uthayakumar et al. [114] introduce a highly reliable and low-complexity compression scheme using a neighborhood correlation sequence algorithm.

Welton et al. [120] investigate compression services that compress and decompress data before transferring it over the network. Their exploration uses *Bzip2*, *Zlib*, and *Lzo*. Results show the relevance of the data used to evaluate compression services. Also, Wiseman [122] proposes a transmission system able to select the intensity of the compression algorithm depending on the CPU cycles and the bandwidth available. Routray et al. [97] enumerate different compression algorithms used in IoT environments over low bandwidth networks, such as Narrowband Internet of Things and Long-Term Evolution Machine Type Communication. They argue in favor of using compression systems to reduce resources and data transfers in these networks, but they do not provide experimental results.

On-the-fly compression in a remote GPU virtualization system raises concerns regarding the trade-off between the processing time and power required for compression and decompression and the resulting benefits of reducing the data transmitted over the network. The overall transmission time, including compression and decompression, could potentially offset the benefits of transferring a smaller amount of data. Various factors can influence the effectiveness of on-the-fly compression, such as the size of the data being sent, the specific contents of the data, the hardware used for compression and decompression, and the network bandwidth.

In this chapter, we explore the suitability of data compression in remote GPU virtualization systems for edge scenarios using machine learning applications. The objective is to determine whether compressing the data on-the-fly for exchange with the remote GPU is worth the associated processing overhead. Thus, we evaluate our system by utilizing different bandwidth networks to simulate a real IoT scenario and use the four Tensorflow (TF) applications introduced in Section 3.1: *Alexnet*, *Cifar10*, *Mnist*, and *Inception*. Figure 6.1 illustrates the scenarios explored. On the one hand, Scenario A represents a real-world application in which no compression is used. On the other hand, in Scenario B, CPU-based compression is used to compress data to be transferred within rCUDA.

We use a Raspberry Pi 4 Model B as the edge node. Raspberry Pi is one of the most popular devices [102, 72] in edge computing due to its low cost and good performance. The GPU server used to offload computations from the edge node contains an NVIDIA V100 GPU. Both node specifications are presented in Table 6.1. These nodes are connected by a 1 Gbps

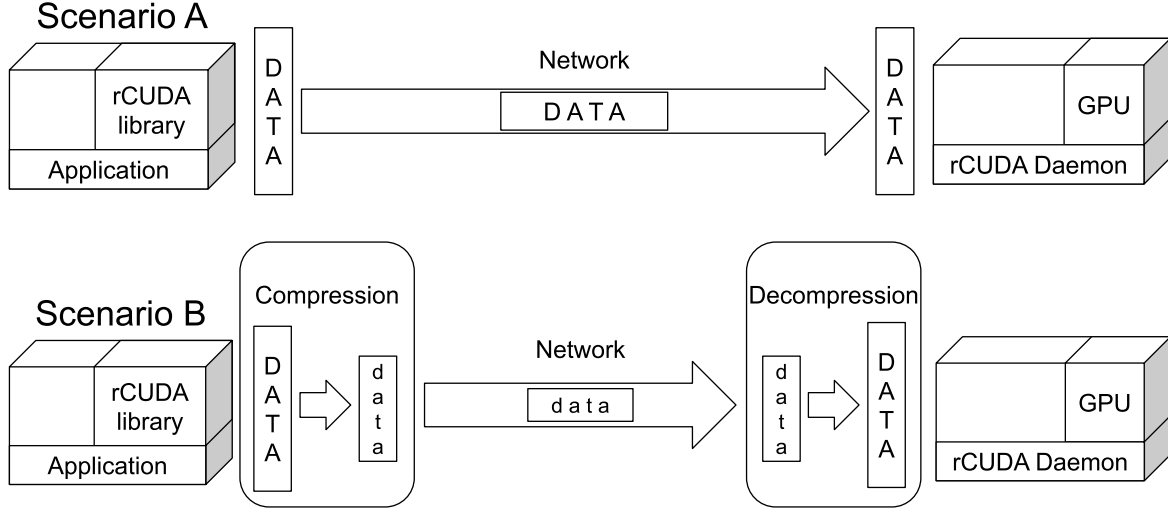


Figure 6.1: Scenarios explored in the experiments. Scenario A represents a real-world application in which no compression system is used. In Scenario B, CPU-based compression is used to compress data to be transferred within rCUDA.

Ethernet wired network. In addition, we use the Linux traffic shaper (i.e., *tc* [56]) to reduce the bandwidth of the network to speeds comparable to the ones achieved when using a wireless network (i.e., WiFi), which are more common in edge computing. In particular, we reduce network bandwidth to 100 Mbps and 10 Mbps.

6.2 Characterization of data transfers

As described in previous chapters, using the rCUDA middleware, we can access the internal data transfers that TF applications carry out from/to the GPU. In the previous Section 3.3, we have characterized the size of these data transfers. Thanks to this characterization, we can better understand what applications are transferring across the network. Notice that, as commented in previous chapters, data transfers done are not necessarily the input of applications. Transfers done by TF applications have been classified into different groups depending on the size of the copied data from host memory to GPU memory.

Figure 6.2 reintroduces the histogram of data transfers (presented in Section 3.3) when running the TF applications under study, i.e., *Alexnet*, *Cifar10*, *Inception*, and *Mnist*. X-axis represents different ranges of sizes used by data transfers. For example, the range labeled as “[128B – 256B]” includes all the data transfers carried out during the execution of an

Client Node (Raspberry Pi 4 Model B)	
Architecture	aarch64
Operating System	Ubuntu20.04.1LTS
Processor	Quad core ARM Cortex-A72 64-bit 1.5GHz
RAM	8GB
Server Node	
Architecture	x86_64
Operating System	Ubuntu20.04.3LTS
Processor	Intel(R) Xeon(R) CPU E5-2637 v2 3.50GHz
RAM	64GB
GPU	NVIDIA V100

Table 6.1: Client and server node specifications used in the CPU-based compression experiments.

application with sizes between 128 bytes and 255 bytes, including both the lower and the upper limits. The histogram illustrates the frequency of data transfers with a specific size range instead of absolute numbers in the Y-axis because four different TF applications have been analyzed. Each TF application performs a different number of data transfers, thus, showing frequency is more beneficial than showing absolute numbers to compare the four applications more clearly.

As we can observe in Figure 6.2, data transfers vary from application to application. However, the size ranges with higher frequency (i.e., the most common ones) are similar for all the applications: between 1B and 1KB. It will directly impact the performance of compression libraries because we observed compression typically behaves better with bigger data sizes in previous chapters.

6.2.1 Compression evaluation of data transfers

In this section, we aim to evaluate the effectiveness of CPU-based compression libraries in reducing data size and identify which libraries achieve the best bandwidth. We present the results of our experiments, highlighting the normalized size of the compressed data across different applications. By examining both small and large data sizes, we aim to understand how compression efficiency varies depending on the characteristics of the transferred data.

This study has been carried out with eight CPU-based compression libraries: *Bzip2*, *Lz4*, *Zlib*, *Zstd*, *Snappy*, *Gipfeli*, *Flz*, and *Lzo*. We have run the experiments with all their

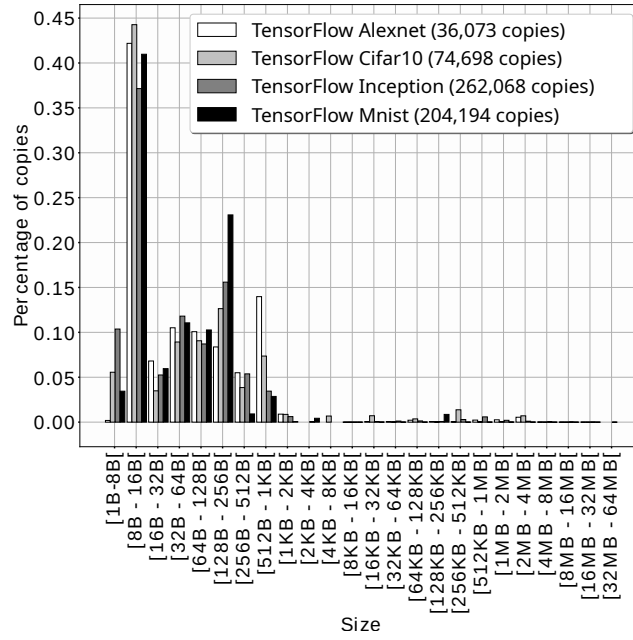


Figure 6.2: Histogram of the data transfers when running TF applications under study. Notice that Y-axis represents the frequency of data transfers with a specific size range performed by each TF application.

configuration parameters. For brevity, we only show the results that obtain the lowest compression and decompression time. Figure 6.3 illustrates the normalized size for each CPU-based compression library when compressing the data transferred through the network. The normalized size has been calculated by dividing the compressed data size by the original data size. Thus, compressing the data is beneficial when the normalized size is below 1. On the left side of the figures, we show small size ranges used in data transfers that correspond with higher frequency sizes used in transfers. On the right side, we display large size ranges.

Results for *Alexnet*, *Cifar10*, *Mnist*, and *Inception* applications are shown in Figure 6.3a, Figure 6.3b, Figure 6.3c, and Figure 6.3d, respectively. As we can observe, CPU-based compression libraries follow the same trend for small sizes. Among the different libraries, the *Bzip2* library stands out for obtaining compressed data with a size almost ten times higher than the original one for data sizes between 1 and 7 Bytes. Notice that compression libraries include additional information in the compressed data to restore the original data, thus, bad compression is expected for small data sizes. We can also see that from 16 Bytes data sizes, some compression libraries start to reduce the original size, such as *Lzo*, *Flz*, *Zlib*, and *Lz4*. In summary, regarding the results for small data sizes, shown on the left of the figures, the libraries that achieve better performance are *Zlib*, *Zstd*, *Lz4*, and *Snappy*.

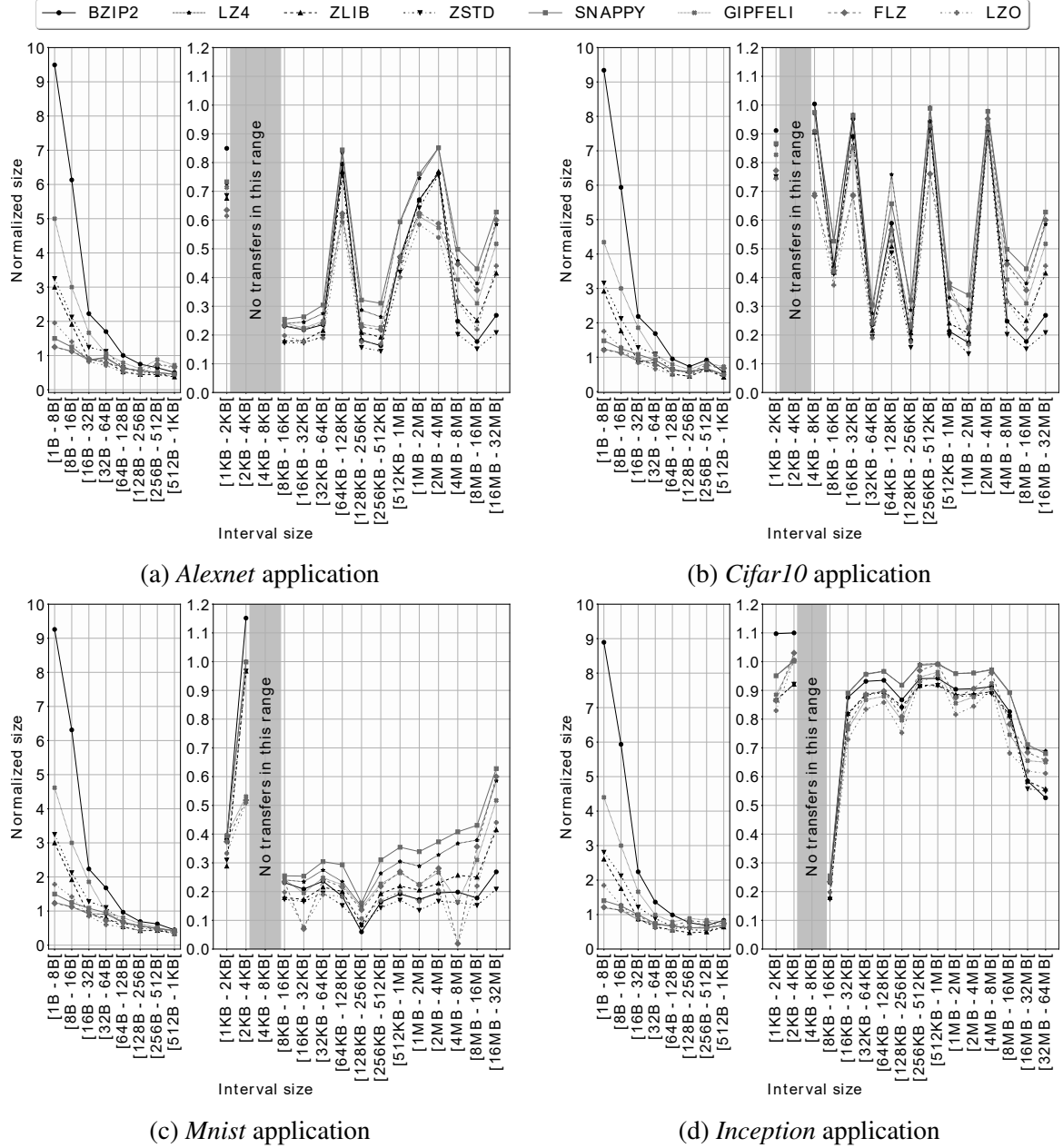


Figure 6.3: Normalized size of the compressed data transferred through the network by TF applications when offloading GPU computations from the edge node to the remote GPU server. Note that a normalized size X means that the compressed data size is X times the original one.

However, if we focus on larger data sizes, shown on the right side of the figures, the conclusions are different. Thus, in the case of *Alexnet* (right side of Figure 6.3a), all libraries achieve, in general, good compression. There are some peaks where the compression data

size is close to the uncompressed one. This is due to the nature of the data because all compression libraries follow the same trend.

Cifar10, in the right side of Figure 6.3b, present similar results. However, in this case we observe more peaks close to one. This increment happens because data transferred by the *Cifar10* application is more difficult to compress than the *Alexnet* ones.

Figure 6.3c shows the results for *Mnist* application. For large data sizes (right side of the figure), only a peak is detected for sizes between 2KB and 4KB. Results show how the original data size used in transfers tends to reduce. We also notice the big compression ratio achieved for data sizes between 128KB and 256KB, where compressed data are over ten times smaller than the original data in the best case. From 256KB, the compression ratio decreases as the data size increases.

Finally, Figure 6.3d presents the results for big data sizes obtained for *Inception*. This time, the nature of the data makes compression difficult for most data sizes. Thus, the normalized size tends to be close to one. However, results improve at data sizes of 8MB and larger.

As we have seen, the compression libraries analyzed present similar results for small data sizes and compression starts being beneficial for sizes over 16 bytes. For large data sizes, however, results vary depending on the application used, i.e., the kind of data transferred. For example, the *Inception* application uses a complex dataset that includes color images with different dimensions, while the *Mnist* application uses a simpler one that contains black and white 28x28 images. Although the input of applications is not necessarily the data used in the transfers between the rCUDA client and the rCUDA server, the input affects these transfers, obtaining worse results with the *Inception* application. In summary, the best compression libraries are *Zstd* and *Bzip2*, while the worst ones are *Snappy* and *Lz4*.

6.3 On-the-fly compression

In previous sections, we have characterized the data transfers of different applications and evaluated the compression ratio achieved by CPU-based compression libraries. In this section, we study the impact on the transfer time of the applications when transferring compressed data. As Figure 6.4 shows, this approach consists on compressing the data exchanged between the rCUDA client and server, i.e., compressing all outgoing data before transmission and decompressing it upon reception. By doing this, we aim to reduce the amount of data transferred over the network, potentially improving bandwidth efficiency.

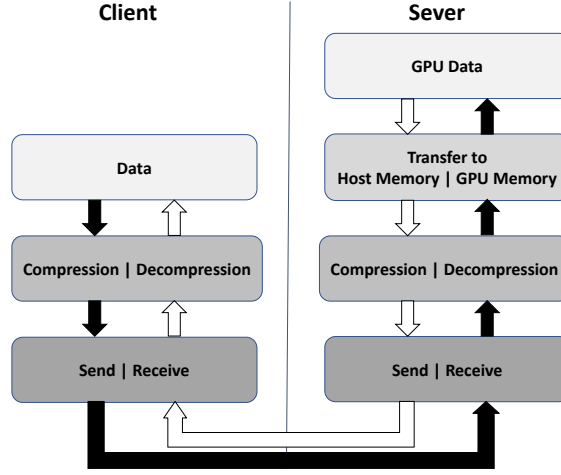


Figure 6.4: On-the-fly compression workflow.

First, we will evaluate the transfers from the edge memory to the GPU memory located in a remote server. Next, the opposite direction will be analyzed. Given that IoT environments usually involve low-performance networks, we use different network speeds to study the influence of the interconnect. Thus, we carry out our exploration by using three different network speeds: 1 Gbps, 100 Mbps, and 10 Mbps.

The time shown in the figures presented in this section has been calculated as the sum of (see Equation 6.1): (i) time spent compressing the data, (ii) time spent transferring the compressed data through the network, and (iii) time spent decompressing the compressed data.

$$Time_{total} = Time_{compress} + Time_{transfer} + Time_{decompress} \quad (6.1)$$

The figures presented in this section maintain a similar structure to the previous figures. Thus, results corresponding to small data sizes are shown on the left side, whereas results for large data sizes are shown on the right. Moreover, the time presented in these figures is normalized with respect to the transfer time without the compression system, i.e., the time of sending the original data through the network without compressing and decompressing it. Therefore, the compression results are acceptable when the normalized time is below one.

6.3.1 Transfers from the edge memory to the remote GPU memory

Figure 6.5 shows the results achieved by CPU-based compression libraries when the *Alexnet* application is executed using different network bandwidths. Figure 6.5a and Figure 6.5b show

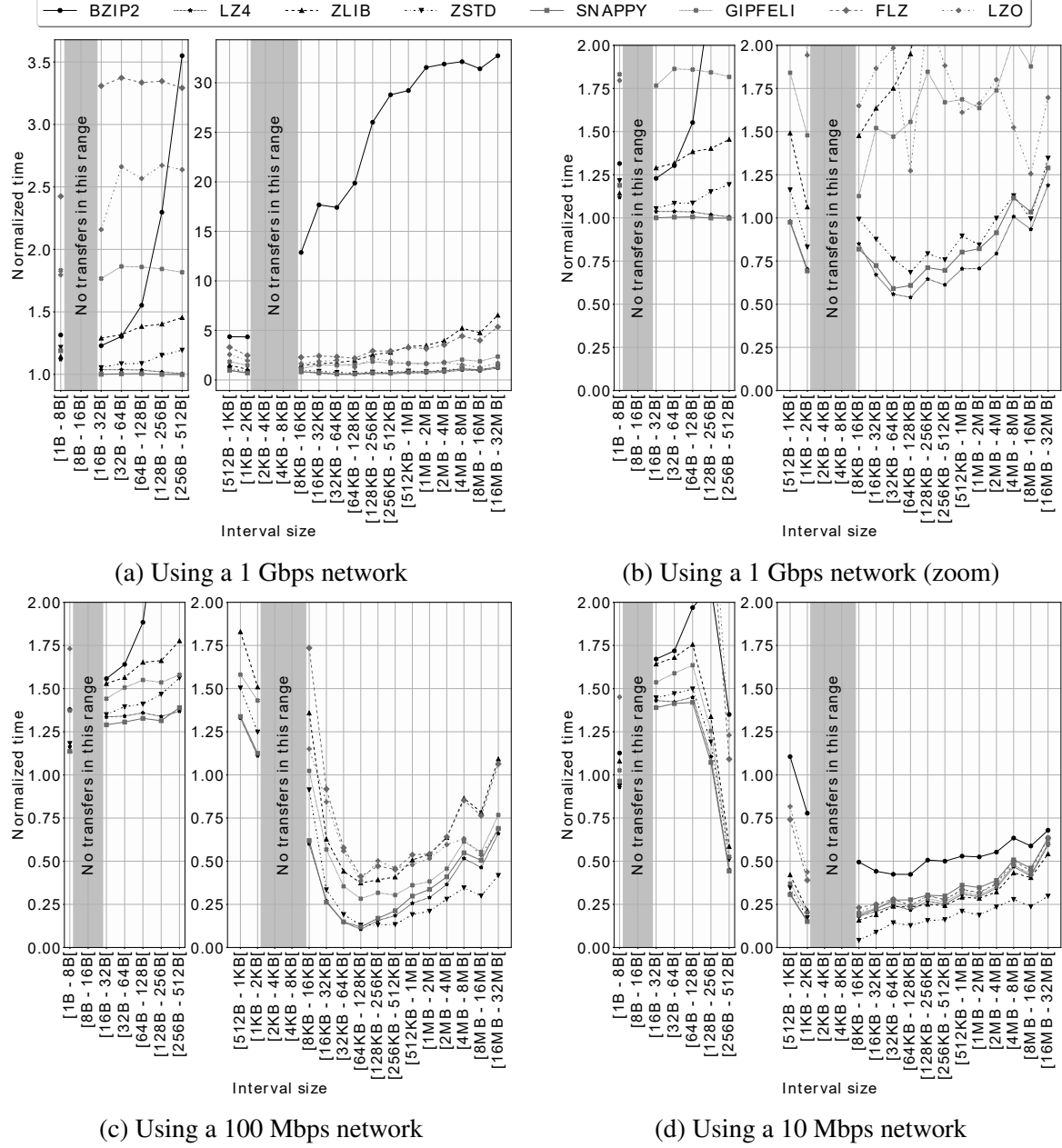


Figure 6.5: Normalized time of transfers done by *Alexnet* application when using on-the-fly CPU-based compression. It includes the time for compressing, sending, and decompressing the data. It has been normalized using the time of transferring the data without using on-the-fly compression. Only memory data copies from the edge node to the remote GPU are considered.

1 Gbps results. The last figure presents the same results amplified for a better analysis. On the left side, we observe that libraries do not reduce the transfer time when using small data sizes. If we analyze again the previous results obtained for the normalized size in Figure 6.3a, we can observe that compression ratios for smaller messages were not noticeable. Besides, although small data reduces their size using compression libraries and the transfer time has been reduced too, the overall time can be higher than without a compression system because it includes the compression and decompression time. On the right side, using big data sizes, some CPU-based compression libraries present significant results, such as *Snappy*, *Lz4*, and *Zstd*. These libraries enhance their results from 8KB data sizes by reducing it to more than a quarter of the transfer time.

The results for the *Alexnet* application using 100 Mbps are presented in Figure 6.5c. Similarly to what happened with the 1 Gbps network, compression has no benefits for small data sizes. Results are even worse because compressed data are larger than original data in most cases for these smaller data ranges, and the total time increases because the transfer time increases due to slow network. However, if we analyze the transfers using big data sizes, the reduction achieved in total time is even greater than with the 1 Gbps network, where time is reduced by up to four times in some data size ranges. Again, *Zstd*, *Lz4*, and *Snappy* are the CPU-based compression libraries that obtain better performance.

As shown in Figure 6.5d, our approach is highly beneficial for *Alexnet* when using a 10 Mbps network, the slowest network. Thus, almost all the libraries enhance the transfer time when using data sizes greater than 256B. The reason is that the network bandwidth is so slow that as soon as compressed data sizes are slightly smaller than the original size, total time is noticeably reduced. For example, this is the case of the range between 256 and 511 bytes, which achieved no benefit with higher network bandwidths. However, using a 10 Mbps network, the transfer time is reduced by up to ten times. The most effective libraries are *Zstd*, *Lz4*, and *Zlib*.

Figure 6.6 illustrates the results of the *Cifar10* application. The CPU-based compression libraries cannot enhance the results when using the 1 Gbps network for small data sizes (Figure 6.6a and Figure 6.6b). However, some compression libraries reduce the data for many size ranges when using large data sizes. *Lz4*, *Snappy*, and *Zstd* obtain the best transfer time. In some cases, they reduce transfer time by a quarter.

Figure 6.6c presents the results for *Cifar10* using a 100 Mbps network. Using small data sizes, no library reduces time. On the contrary, as in the case of *Alexnet*, the compression benefits are more relevant when using large data sizes. *Zstd*, *Lz4*, and *Snappy* compression

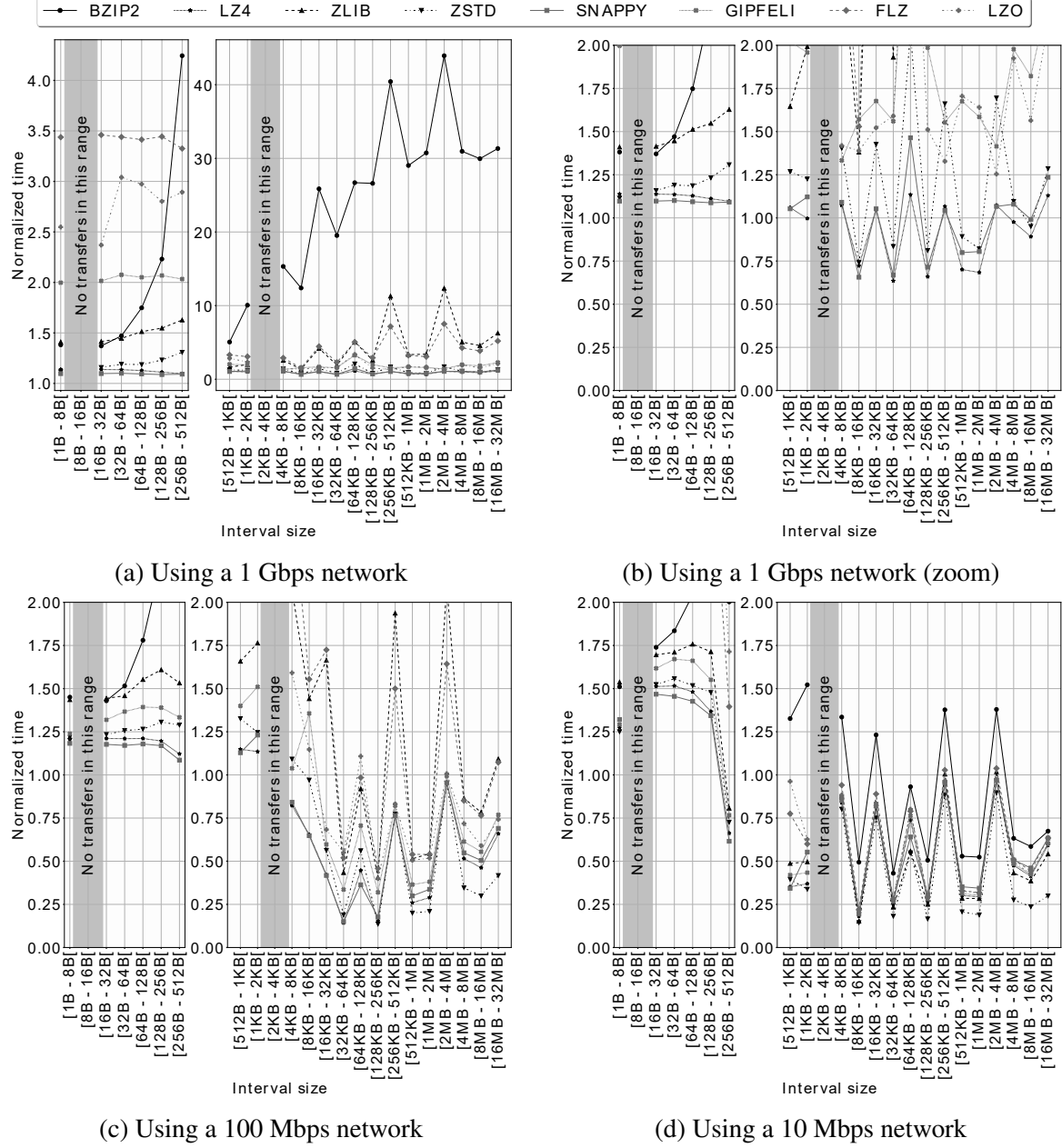


Figure 6.6: Normalized time of transfers done by *Cifar10* application when using on-the-fly CPU-based compression. It includes the time for compressing, sending, and decompressing the data. It has been normalized using the time of transferring the data without using on-the-fly compression. Only memory data copies from the edge node to the remote GPU are considered.

libraries perform better than the rest of the compression libraries. However, in this case, the results are worse than with the *Alexnet* ones (see Figure 6.5c). Some peaks indicate that data are difficult to compress. The reason is that the compression ratios obtained in these data size ranges for *Cifar10* are lower, as shown in Figure 6.3b. In some cases, however, CPU-based compression libraries reduce the transfer time by up to four times in some data size ranges.

Again, the best scenario for our approach is when using a 10 Mbps network, shown in Figure 6.6d. Our system achieves transfer time reductions of over 30% when using data sizes greater than 256 bytes. In addition, almost all the compression libraries reduce total transfer time when using large data sizes. However, some libraries have problems with data size ranges whose compression ratio is low. Again, the transfer time is reduced by up to four times for some data size ranges. In this experiment, *Zstd*, *Zlib*, and *Lz4* present the best results.

Figure 6.7 shows the results achieved by the *Mnist* application. Similar to what happened with previous applications, when using the 1 Gbps network (Figure 6.7a and Figure 6.7b), the compression is not beneficial for small data sizes. When using big data sizes, however, *Lz4* and *Snappy* compression libraries reduce their time by almost half. It should be noted that results worsen as data size increases. We can see in Figure 6.3c that this behavior is due to the compression ratio decreasing as data size increases.

Figure 6.7c presents the results for *Mnist* using a 100 Mbps network. Compression libraries do not offer any reduction for smaller data sizes. However, *Snappy*, *Zstd*, and *Lz4* libraries continue reducing the transfer time when using large data sizes. They get reductions over four times in several size ranges. Also, they reduce transfer time by up to ten times with data sizes between 128KB and 255KB.

As in previous experiments, the best scenario for our approach is when using a 10 Mbps network, shown in Figure 6.7d. Using small data sizes, reductions become relevant for sizes over 256 bytes. Almost all the compression algorithms provide some improvement. The best one is *Snappy*, which can nearly reduce the time by up to half. All the CPU-based compression libraries reduce the transfer time when using big data sizes. *Zstd* presents the best results, reducing transfer time by up to ten times in some data size ranges.

Figure 6.8 illustrates the results of the *Inception* application. When using the 1 Gbps network (Figure 6.8a and Figure 6.8b), CPU-based compression libraries achieved different results from previous experiments. *Lz4* and *Snappy* are the only compression libraries presenting some reduction. Unlike prior TF applications, data used in *Inception* are larger and more

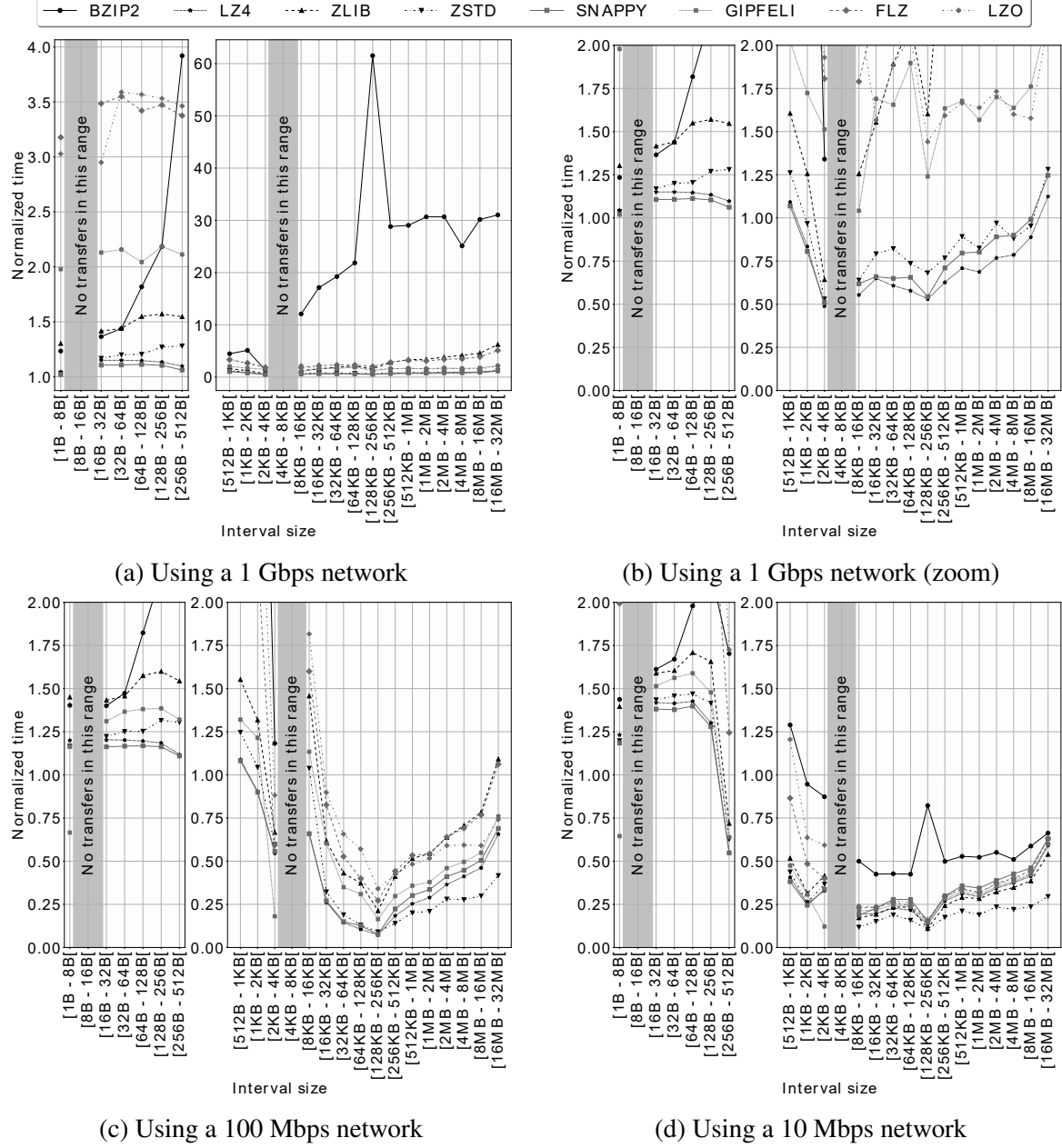


Figure 6.7: Normalized time of transfers done by *Mnist* application when using on-the-fly CPU-based compression. It includes the time for compressing, sending, and decompressing the data. It has been normalized using the time of transferring the data without using on-the-fly compression. Only memory data copies from the edge node to the remote GPU are considered.

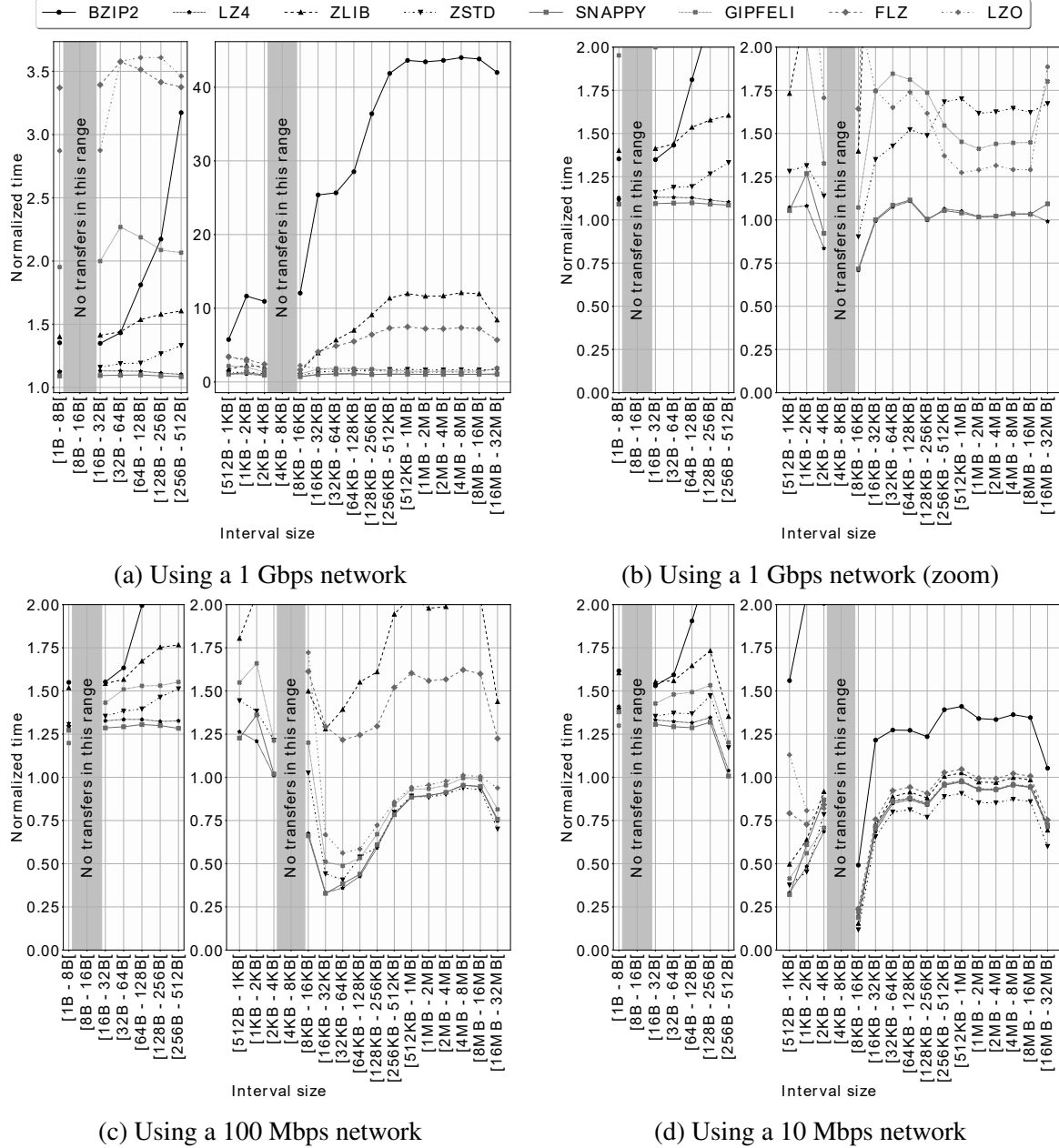


Figure 6.8: Normalized time of transfers done by *Inception* application when using on-the-fly CPU-based compression. It includes the time for compressing, sending, and decompressing the data. It has been normalized using the time of transferring the data without using on-the-fly compression. Only memory data copies from the edge node to the remote GPU are considered.

complex. This makes it more difficult to compress, as shown in Figure 6.3d. Thus, our approach does not present good results in this case.

Figure 6.8c presents the results for *Inception* using a 100 Mbps network. Once again, compression libraries obtain better results as the network is slower. In this case, compression becomes beneficial over 8KB. *Zstd*, *Snappy*, and *Lz4* compression libraries present the best performance, reducing the time between 10% and 60%.

Results obtained for the *Inception* application when using a 10 Mbps network are shown in Figure 6.8d. As we can see, all CPU-based compression libraries, except the *Bzip2* one, reduce the time for big data sizes.

6.3.2 Transfers from the remote GPU memory to the edge memory

In many edge-cloud scenarios, particularly those involving GPU acceleration, data transfer from the remote GPU back to the edge device is a crucial step. Although edge devices are often limited in processing capabilities, they frequently participate in a hybrid processing model where lightweight tasks (such as postprocessing or decision-making) are handled locally. In contrast, computationally intensive operations (such as inference or batch processing) are offloaded to remote GPUs. In this context, the edge device must retrieve intermediate or final results from the GPU to complete the application workflow. For instance, raw sensor data might be offloaded to a remote GPU for anomaly detection in industrial IoT settings. Still, the edge device remains responsible for real-time actions, such as triggering alarms or initiating control responses based on the results. Therefore, we have also studied transfers done in the opposite direction, from the remote GPU memory to the edge node.

Although we have run the experiments using 1 Gbps, 100 Mbps, and 10 Mbps networks, we only present the results that better show the impact of the computational power on the performance, i.e., results when using a 10 Mbps network. However, similar conclusions can be drawn for the rest of the networks (1 Gbps and 100 Mbps). Figure 6.9 shows the results for the different applications. We can observe that almost all the algorithms improve if we compare these results to previous ones: (i) Figure 6.9a compared to Figure 6.5d, (ii) Figure 6.9b compared to Figure 6.6d, (iii) Figure 6.9c compared to Figure 6.7d and (iv) Figure 6.9d compared to Figure 6.8d

As mentioned, the computational power of the remote server is higher than that of the edge node. This makes compression faster. However, the decompression is slower because it is

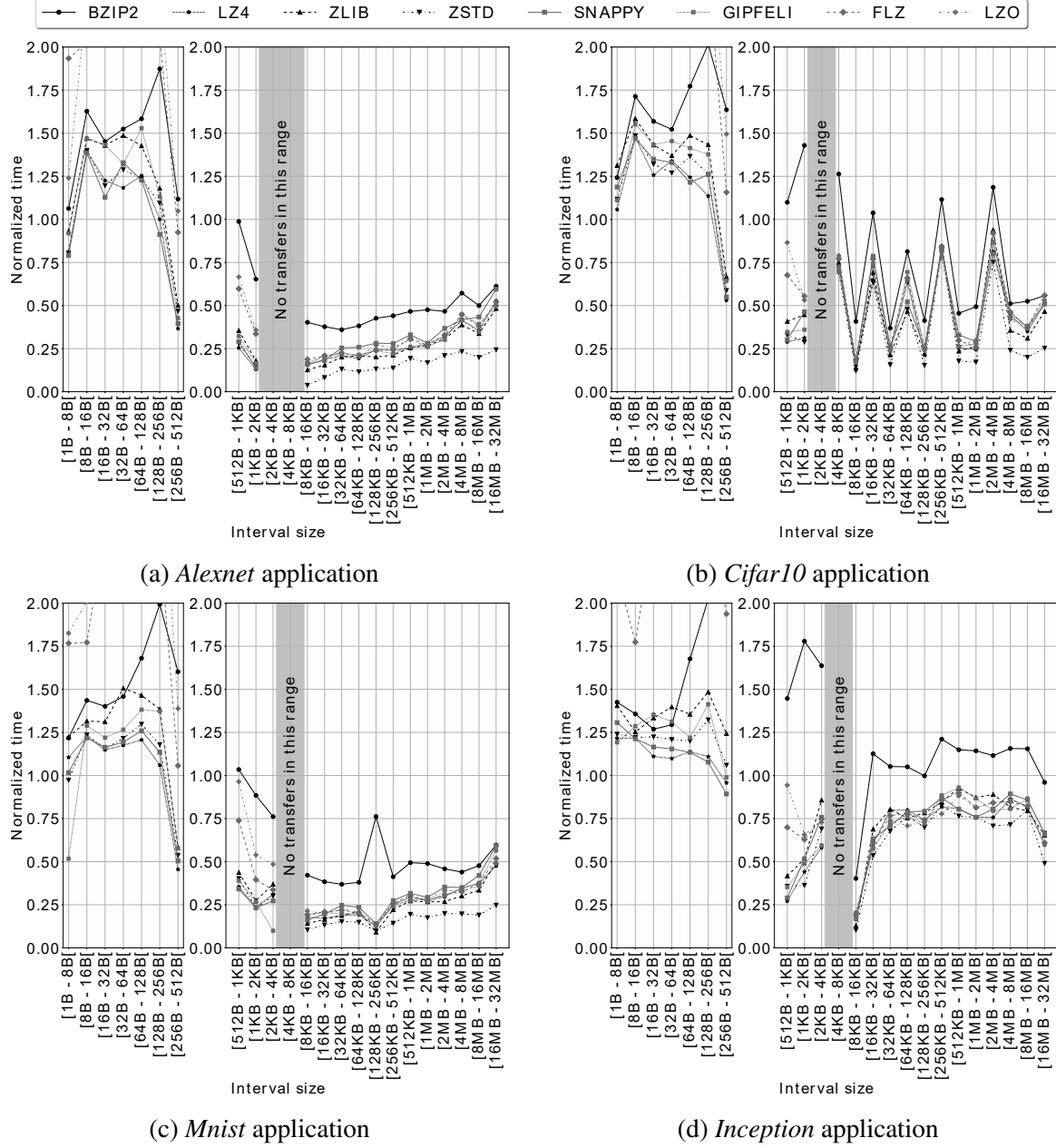


Figure 6.9: Normalized time of transfers done by TF applications when using on-the-fly CPU-based compression over a 10 Mbps network. It includes the time for compressing, sending, and decompressing the data. It has been normalized using the time of transferring the data without using on-the-fly compression. Only memory data copies from the remote GPU to the edge node are considered.

done on the edge node. In general, the compression requires more computational power than decompression, and running it on the remote server significantly reduces the transfer time. This reduction means that the increment time produced by doing the decompression on the edge node does not worsen the total time. Notice that the results presented in this section have not utilized the GPU installed in the remote server (i.e., compression and decompression are done in the CPU). Finally, as we observe with transfers done in the opposite direction, CPU-based compression libraries obtain lower performance when using faster network speeds (i.e., 1 Gbps and 100 Mbps). Also, the compression libraries that get better performance are the same ones, i.e., *Lz4*, *Zstd*, and *Snappy*.

It is worth paying attention to an interesting behavior related to the CPU-based compression libraries, such as *Bzip2* and *Lz4*. The *Bzip2* library got one of the best compression ratios in the experiments, while the *Lz4* library got one of the worse, as shown in Figure 6.3. A good compression ratio means that the transfer time is reduced because the data to be sent is smaller than the original. However, we observe how the *Bzip2* library obtains the worst results when we analyze the transfer time, whereas the *Lz4* library gets one of the bests in Figure 6.5, Figure 6.6, Figure 6.7, and Figure 6.8. This is due to the compression and decompression time of these compression libraries. *Bzip2* achieves an extraordinary compression ratio but it requires considerable computational effort. This effort is not compensated when the network is relatively fast, such as the 1 Gbps one. Thus, with faster networks, a compression library with a poorer compression ratio but with shorter computation time (i.e., less computational effort), such as the *Lz4* library, is more convenient. As the network bandwidth decreases, *Bzip2* results approach the rest of the compression libraries. This happens because the reduction in data size compensates for the high computational time required for compression and decompression. However, *Bzip2* still presents the worst behavior for 10 Mbps networks. Therefore, its long compression and decompression times will only be worthwhile for extremely slow networks.

Summarizing this section, we have seen that transferring compressed data provides benefits depending on two factors: (i) the network speed and (ii) the data size exchanged with the remote GPU. As expected, the slower the network is, the better the results are. Thus, when using 100 Mbps and 10 Mbps networks, TF applications can reduce their total time up to ten times. In our experiments, *Lz4*, *Zstd*, and *Snappy* compression libraries were the ones with the best performance in general, as shown in Table 6.2.

		Small Data Sizes ([1B - 512B])			Large Data Sizes ([512B - 32MB])		
App ↓	Net →	1 Gbps	100 Mbps	10 Mbps	1 Gbps	100 Mbps	10 Mbps
<i>Alexnet</i>		<i>Snappy</i>	<i>Snappy</i>	<i>Snappy</i>	<i>Lz4</i>	<i>Zstd</i>	<i>Zstd</i>
<i>Cifar10</i>		<i>Snappy</i>	<i>Snappy</i>	<i>Snappy</i>	<i>Lz4</i>	<i>Zstd</i>	<i>Zstd</i>
<i>Mnist</i>		<i>Snappy</i>	<i>Snappy</i>	<i>Snappy</i>	<i>Lz4</i>	<i>Zstd</i>	<i>Zstd</i>
<i>Inception</i>		<i>Snappy</i>	<i>Snappy</i>	<i>Snappy</i>	<i>Lz4</i>	<i>Zstd</i>	<i>Zstd</i>

Table 6.2: Summary of the best results achieved by CPU-based compression when evaluating on-the-fly compression.

6.3.3 Impact on overall application performance

We have separately studied the impact of transferring compressed data from/to the GPU using CPU-based compression libraries to compress data used by TF applications before sending it. In this section, we evaluate the impact on the overall performance of the applications. Again, we use different network speeds to study the influence of the network.

Figure 6.10 compares the total time of each TF application in different cases. *Original* refers to the case where no compression is used. Thus, it only requires preparing the data to be sent and performing some computations with the data (referred to as “Computation time” in the figure), and the “Transfer time”. The figure also includes the time when the different libraries under study are used, labeled with the name of each library. In this latter case, the time is broken down in four parts: (i) time for preparing the data to be sent and performing some computations with the data (i.e., “Computation time”); (ii) time spent compressing the data (i.e., “Compression time”), (iii) time spent transferring the compressed data through the network (i.e., “Transfer time”), and (iv) time spent decompressing the data (i.e., “Decompression time”).

A new compression mechanism has also been developed in this section, referred to as *Adaptive*. This new mechanism is based on previous results where we have seen that the behavior of compression libraries has been similar for each of the applications under study: (i) there are smaller data size ranges where compression is not beneficial because it adds some latency due to a bad compression ratio, and (ii) some compression libraries perform better than others depending on the data size range. Considering this, we have created a new mechanism called *Adaptive*. It uses the best possible compression library depending on the size of the data to be transmitted. Furthermore, if the size of the data is too small to provide any benefit, the original data will be sent uncompressed.

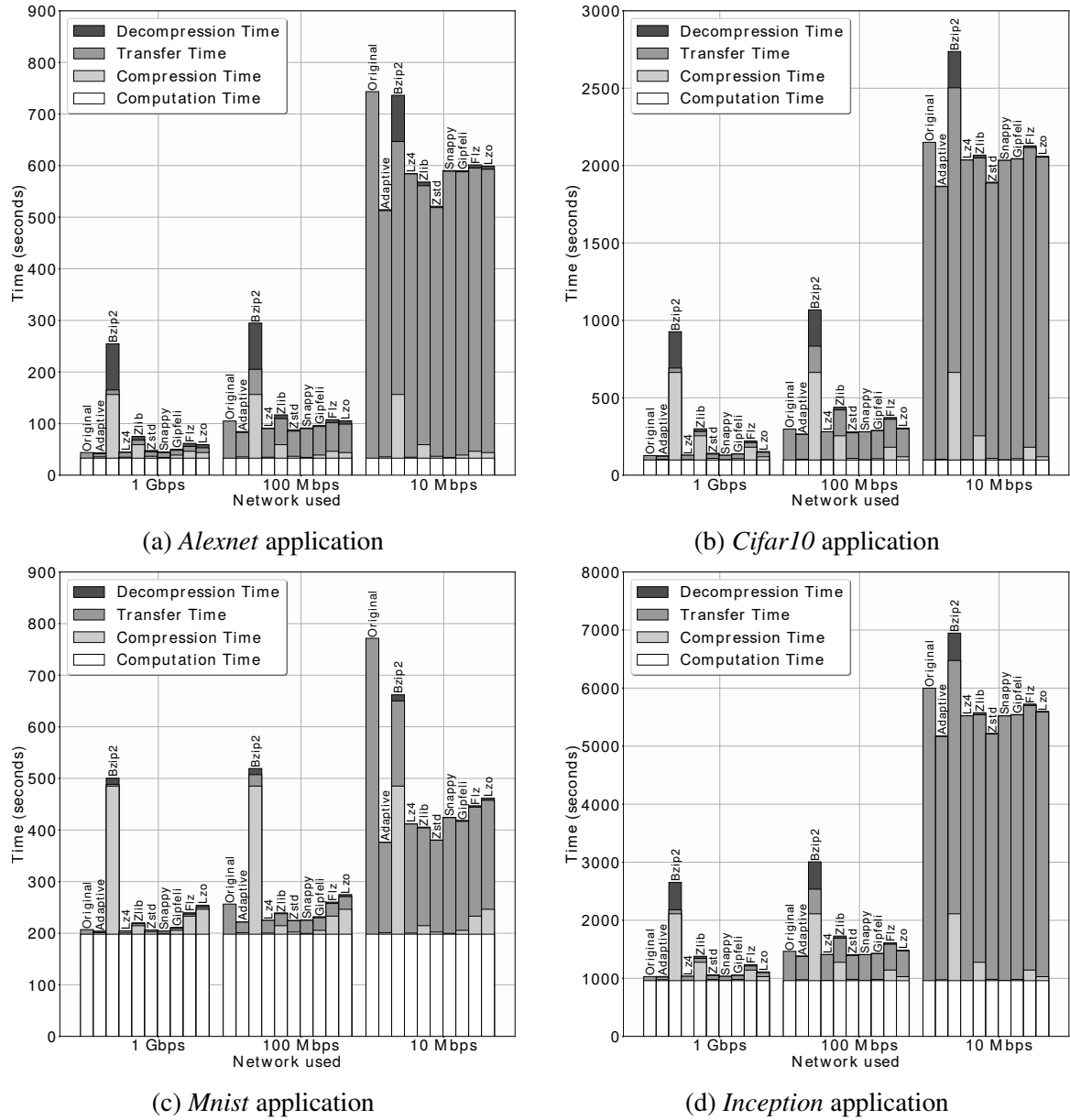


Figure 6.10: Impact of transferring compressed data on the overall performance of the TF applications using different networks. *Original* refers to the case where no compression is used. It only requires preparing the data to be sent and performing some computations with the data (referred to as “Computation time” in the figure), and the “Transfer time”. The time when the different libraries under study are used is labeled with the name of each library. It is broken down in four parts: (i) time for preparing the data to be sent and performing some computations with the data (i.e., “Computation time”); (ii) “Compression time”, (iii) “Transfer time”, and (iv) “Decompression time”. *Adaptive* refers to our new mechanism which uses the best compression library depending on the size of the data and the network speed.

Application	Number of transfers done	
	From edge node to remote GPU	From remote GPU to edge node
<i>Alexnet</i>	20,112	15,961
<i>Cifar10</i>	34,918	39,780
<i>Mnist</i>	112,616	91,578
<i>Inception</i>	131,933	130,135

Table 6.3: Number of data transfers (i.e., memory copies) made during the execution of the TF applications under study.

Figure 6.10 shows the results achieved when using this new mechanism. As we can see, using a 1 Gbps network, *Lz4*, *Snappy*, *Zstd*, and *Gipfeli* achieve a total time close to not using compression. The *Adaptive* mechanism even improves those ones. This was expected because this mechanism uses only CPU-based compression libraries when the transfer time can be reduced. We can also appreciate how compression libraries require more time for compressing than decompressing, as commented in previous sections.

When using a 100 Mbps network, the *Adaptive* compression mechanism is the one that performs the best. Thus, the time of *Alexnet*, *Cifar10*, *Mnist*, and *Inception* applications is reduced by 20 seconds, 32 seconds, 35 seconds, and 80 seconds, respectively. Moreover, *Lz4*, *Snappy*, *Zstd*, and *Gipfeli* compression libraries can also reduce the execution time of the TF applications.

Finally, the *Adaptive* mechanism obtains the best results once more when a 10 Mbps network is used. In this case, the execution time of the *Alexnet* application is reduced by almost 4 minutes, *Cifar10* by over 4 minutes, *Mnist* by 6.6 minutes, and *Inception* by 14 minutes. Using this network, all compression libraries reduce the TF application time. However, the *Bzip2* library cannot reduce this time with the *Cifar10* and the *Inception* applications due to the *Bzip2* library compression process seeming to be expensive.

A final consideration in this section is related to the fact that compression takes longer than decompression, as shown in Figure 6.10. The experiments have been carried out using two nodes: (i) the edge node, which executes the TF application; and (ii) the node where the remote GPU is stored, which is computationally more powerful. Thus, a possible reason could be simply because there are many more data copies from the edge device to the remote GPU than in the opposite direction. If this were the case, then most data would be compressed in the edge node, which features a less powerful CPU. To answer this question, we have collected all the transfers done by the TF applications under study. In addition, we divided

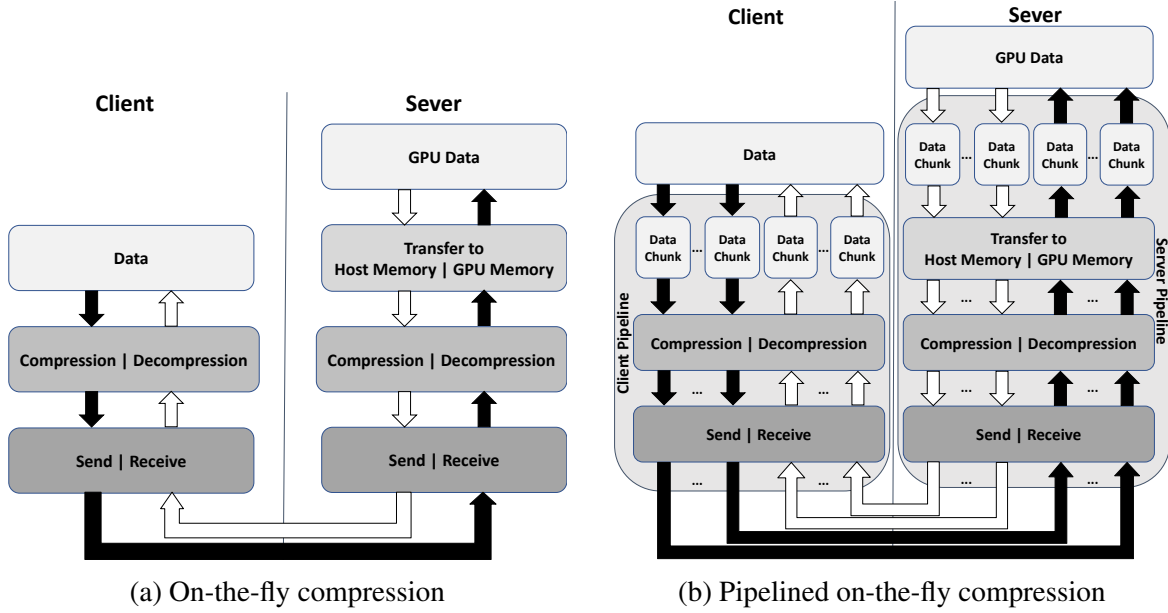


Figure 6.11: Compression approaches using the remote GPU virtualization solution rCUDA.

each transfer depending on its direction, i.e., from the edge node to the remote GPU or vice-versa. Table 6.3 shows the number of data transfers done in the four TF applications. As shown in the table, *Alexnet*, *Mnist*, and *Inception* present more data copies in the direction from the edge node to the remote GPU server. However, *Cifar10* contains more data copies in the opposite direction, from the remote GPU server to the edge node. In any case, the differences in the number of transfers do not seem to be the reason for the differences between compression and decompression times. Thus, compression must be computationally heavier than decompression.

6.4 A pipelined approach for on-the-fly compression

In the previous sections, on-the-fly compression was deployed in the communication layer of the remote GPU virtualization middleware rCUDA. In this way, data sent to/from the remote GPU is compressed using CPU-based compression libraries. Thus, every time an application performs a CUDA data copy, the client middleware intercepts the data to be copied. Then, data is compressed on-the-fly and sent to the server side, as shown in Figure 6.11a. Once the compressed data is received on the server side, the remote GPU virtualization middleware decompresses it before forwarding it to the GPU. In the opposite direction, a similar process is followed. There is no overlap between compression, transmission, and decompression.

In addition, we must also take into account that implementing on-the-fly data compression within a remote GPU virtualization middleware must be done in such a way that the overhead of the compression and decompression does not cancel the benefits of transmitting a smaller amount of data.

Considering all that, a pipelined approach can significantly enhance the implementation developed in Section 6.1. This approach has been successfully utilized in other studies in the past, such as the ones carried out by Hu [43] and Krintz and Sucu [53]. On the one hand, Hu [43] studies the possibility of compressing the data sent over the network based on the available bandwidth in a 10 Mbps network. The author proposes a simple pipeline that splits the data into different chunks and calculates if these chunks must be compressed before sending it, depending on the available network bandwidth. Although the pipeline stages do not overlap, the results are promising. On the other hand, Krintz and Sucu [53] present an adaptive on-the-fly compression system to transmit data over 100 Mbps and 1.7 Mbps networks. The compression system splits the data into chunks of 32KB, checks the available network bandwidth, and determines whether these chunks must be compressed (using *Lzo*, *Zlib*, and *Bzip2*) before sending it. Again, although data is split, the stages of the system do not overlap.

Other studies present interesting pipeline systems where stages overlap, such as the ones by Peterson and Reiher [91], Chowdhury et al. [17], and Kim et al. [52]. Peterson and Reiher [91] present a system that decides whether to compress or not a specific chunk of data using *Lzo*, *Zlib*, *Bzip2*, and *Xz* compression libraries. Data is split into 32KB pieces, and a parallel pipeline compression system is used, hiding the compression and decompression time behind the transfer time. However, despite the acceptable performance, the study does not compare the results obtained with a compression system without a pipeline, nor does it evaluate different pipeline configurations. Chowdhury et al. [17] create a novel two-step compression scheme that exploits temporal correlation in the individual streams to be transmitted in order to increase bandwidth savings. This study obtains a significant reduction in the network resources required at the expense of being a lossy solution. Kim et al. [52] introduce a selective data compression scheme based on data prediction. In this study, the authors apply *LZ77* encoding and dynamic *Huffman* coding to compress data. The prediction method reduces wasted computing resources by avoiding coding data when necessary. Using this selective compression, they can choose a dynamic or static *Huffman* codification to code the result of the *LZ77* reduction. The authors also show a compression pipeline in which *LZ77* encoding and *Huffman* coding overlap. Although the study focuses on using *Huffman* codification, there is no research on *LZ77* compression. Both algorithms

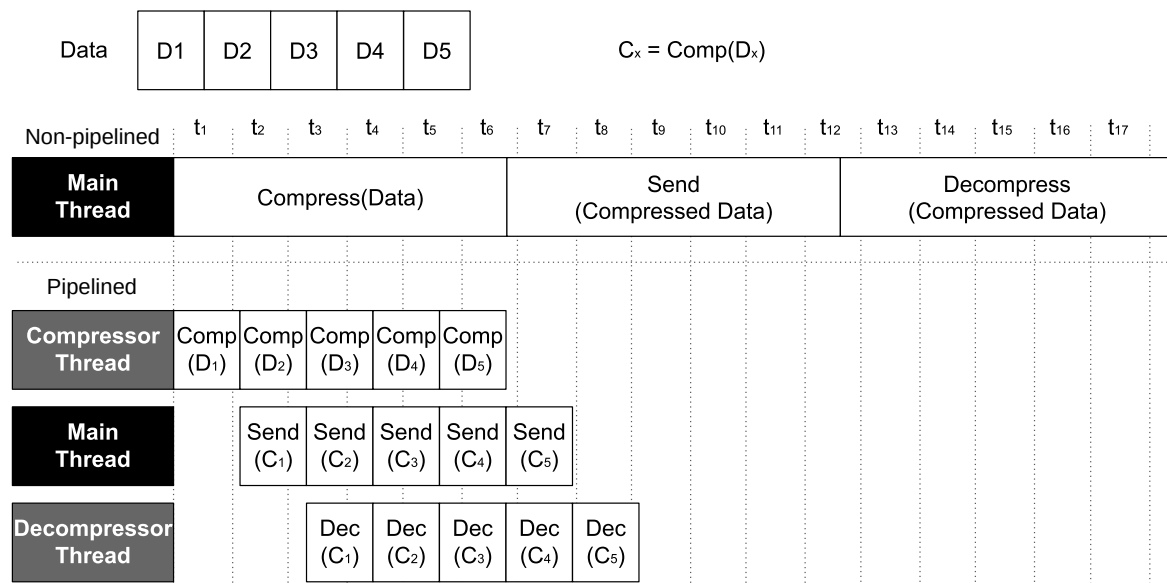


Figure 6.12: Comparison between pipelined and non-pipelined on-the-fly compression.

should be considered; otherwise, the resulting data might be larger than the input data, and computing resources would have been wasted.

These studies highlight the importance of implementing an efficient compression pipeline, which should aim to overlap and execute steps simultaneously to maximize performance. The latency can be significantly reduced by processing different stages in parallel (e.g., compressing the data, transmitting the data through the network, and decompressing the data once received).

This section presents an implementation of an on-the-fly data compression pipeline within a remote GPU virtualization middleware. More precisely, we have modified the internal communication layer of the remote GPU virtualization framework rCUDA to perform a pipelined on-the-fly data compression. This communication layer already implements a pipeline approach to enhance network speed. We have taken advantage of this existing pipeline to introduce our CPU-based compression approach. The reasons for using these compression libraries instead of GPU-based ones are also discussed.

6.4.1 How the compression pipeline works

As shown in Figure 6.12, when applying a pipeline strategy in the rCUDA remote GPU virtualization system, the client middleware intercepts the data of each application request

and splits it into chunks. Then, the client middleware compresses the first chunk. Once it is compressed, it begins compressing the second chunk. At the same time as the second chunk is being compressed, the client middleware sends the first compressed chunk to the server.

This process continues with the rest of the chunks. In this way, compression and network transfers overlap on the client side of the middleware. A similar approach is followed on the server side, overlapping decompressions and transfers to GPU memory. Thus, while the server decompresses the chunk $i + 1$, in parallel, it copies the chunk i to the GPU. At the same time, the network transfers the chunk $i + 2$ to the server while the chunk $i + 3$ is compressed on the client side.

Figure 6.11b shows this pipelined approach on both sides of the middleware, the client and server one. Applying the well-known pipeline technique could significantly improve performance for two reasons: (1) the total amount of transmitted data is reduced (as it was with the non-pipelined approach), and (2) compression and decompression stages are overlapped with transmission, hiding the latency of these stages.

6.4.2 Experimental evaluation

As in Section 6.1, we study the pipelined approach focusing on edge environments. Thus, we will use the same nodes: (i) a Raspberry Pi 4 Model B as the edge node and (ii) a node with an NVIDIA V100 GPU. Both node specifications were presented in Table 6.1. These nodes are connected by a 1 Gbps Ethernet wired network. We use the Linux traffic shaper (i.e., *tc*) to limit the network bandwidth to 10 Mbps, simulating the low-performance networks used in edge environments.

To evaluate the benefits of implementing compression within the rCUDA communication layer, we have run the TF applications (*Alexnet*, *Cifar10*, *Mnist*, and *Inception* presented in Section 3.1). We consider the scenarios shown in Figure 6.13: offloading computations using only the CPU of the edge device (Scenario A), using rCUDA over a 10 Mbps network without any compression system (Scenario B), and using rCUDA over a 10 Mbps network with on-the-fly compression implemented in rCUDA pipeline communication layer (Scenario C). The last scenario has been carried out with six CPU-based compression libraries: *Lz4*, *Zlib*, *Zstd*, *Snappy*, *Gipfeli*, and *Flz*. Notice that Scenario A has been possible thanks to TensorFlow Lite [111], a mobile library for deploying models on mobile, microcontrollers, and other edge devices.

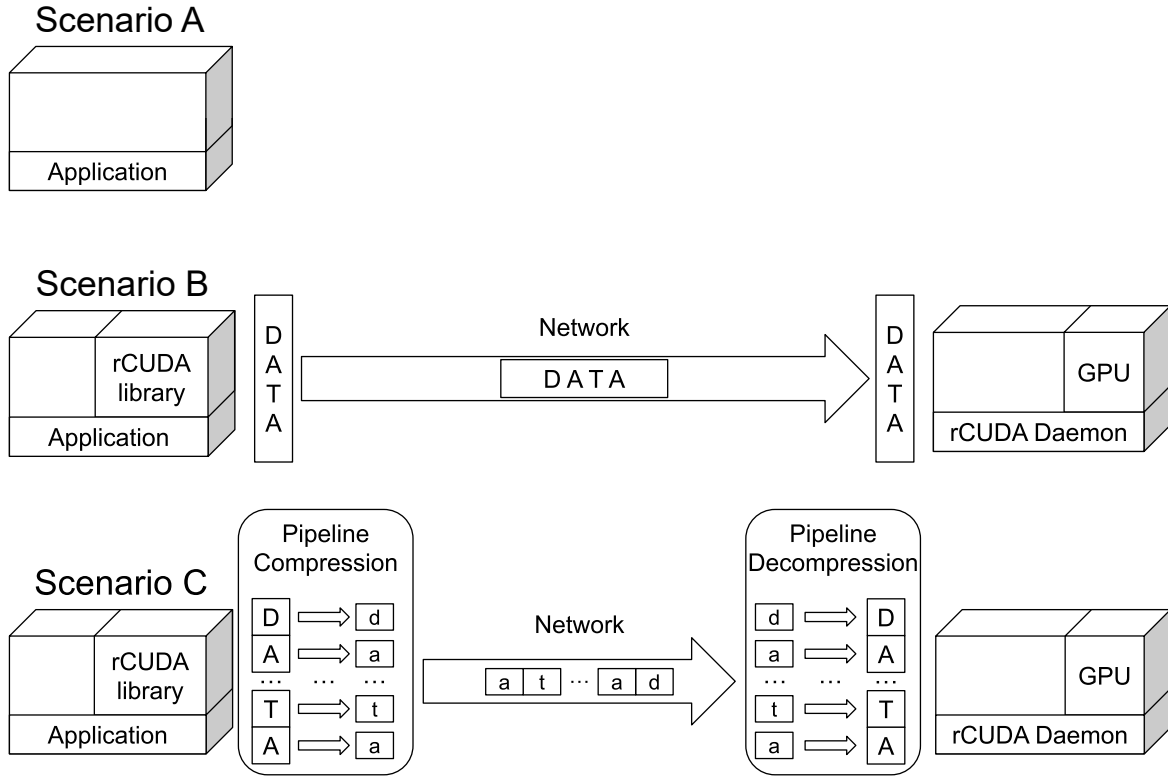


Figure 6.13: Scenarios explored in the experiments. Scenario A represents a real-world application in which only the CPU of the edge device is used. In Scenario B, the real-world application runs using rCUDA. Scenario C presents the scenario where rCUDA is utilized with on-the-fly compression implemented in its pipeline communication layer to run the real-world application.

Figure 6.14 presents the results obtained when TF applications run in these scenarios. We can observe that using rCUDA without compressing data transfers increases execution time between 1.4 and 25 times more than using only the edge device CPU. These results clearly show that optimizing data transfers is essential for achieving better performance with rCUDA. One such optimization is the proposed on-the-fly data compression. As it can also be seen, using rCUDA with pipelined on-the-fly compression improves the results in all the cases when compared to using rCUDA without compression. However, if we compare the results to the ones using only the edge device CPU, we observe that only the *Mnist* application performs better with rCUDA.

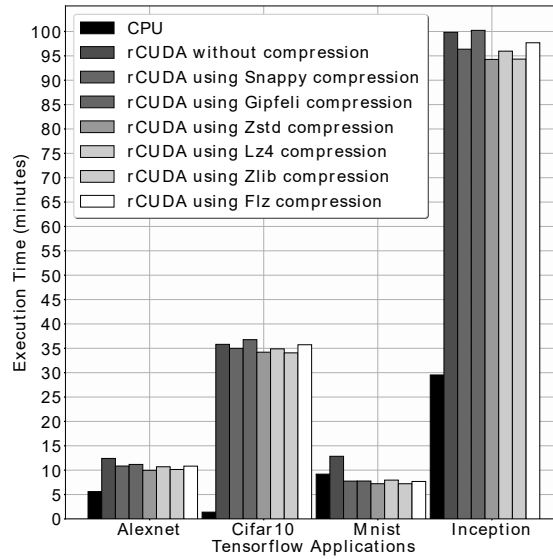


Figure 6.14: Results obtained when running TensorFlow applications in different scenarios: (i) offloading computations using only the CPU of the edge device, (ii) using rCUDA over a 10 Mbps network without compression, and (iii) using rCUDA over a 10 Mbps network with CPU-based pipelined on-the-fly compression employing the libraries under study.

TensorFlow Lite is designed to run lightweight, optimized models on resource-constrained devices. These models benefit from aggressive optimizations such as quantization and operator fusion, enabling faster inference with significantly lower computational and memory overhead. Using a 10 Mbps bandwidth network constitutes a significant bottleneck in the system, leading to faster execution times on the local CPU.

The offloading process requires transmitting input data to the remote GPU and receiving processed results back. Given the limited bandwidth and inherent latency, the communication overhead can outweigh the computational acceleration benefits of the remote GPU. This network limitation clearly constrains the practical gains of remote execution in the current setup.

However, it is essential to highlight that the models deployed on TensorFlow Lite are often simplified or compressed versions of the full-scale models used in rCUDA execution. It typically comes at the cost of reduced accuracy or representational power. In contrast, rCUDA allows running more complex and computationally intensive models on a powerful GPU, which can deliver higher accuracy or handle more sophisticated tasks that would be prohibitive on the Raspberry Pi.

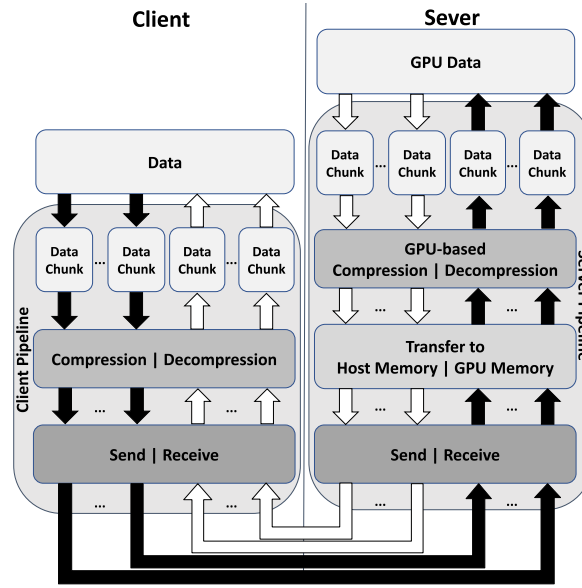


Figure 6.15: Scenario with on-the-fly GPU-based compression integrated in rCUDA pipeline communication layer.

At first glance, since we use CPU-based compression libraries, one possible approach to improve the results could be leveraging GPU-based compression. However, this raises an important consideration: the nature of the transferred data and the inherent characteristics of GPUs may prevent an efficient integration in the pipeline. Moreover, as mentioned in Section 6.1, over 90% of data transfers are between 1B and 1KB. However, the pipeline communication used in rCUDA is implemented with chunks of 1KB. This explains the results obtained in Figure 6.14. The experiments do not evaluate the pipelined compression correctly. CPU-based compression libraries are not taking advantage of this communication pipeline because most of the data transferred is smaller than the chunk size utilized, i.e., there is no pipeline. In addition, it should be noted that compression libraries usually perform better with big data sizes because the more data, the more information is repeated. Sections 6.4.3 and 6.4.4 further analyze these challenges and explore some optimizations. Finally, Section 6.5 uses a more helpful study to evaluate pipelined approaches for on-the-fly compression and improve the approach presented in this section.

6.4.3 Rationale for using CPU-based compression libraries

At this point, exploring GPU-based compression libraries could be interesting. GPUs are well-suited for parallel workloads, making them an attractive option for accelerating compression,

as they can process multiple chunks of data simultaneously, potentially reducing latency and increasing throughput. To integrate these libraries, the server-side data flow must be adapted. Data received is sent to GPU memory before decompressing it. Thus, data located in GPU memory is compressed before copying it to host memory to be sent through the network. Figure 6.15 visually represents this new data flow, illustrating how the GPU-based compression libraries can be integrated into the server workflow.

There are several reasons why GPU-based compression has not been addressed in this thesis to enhance remote GPU virtualization systems like rCUDA: (i) the size of the data used by applications, (ii) the incompatibility of the GPU with the processing pipeline, and (iii) the impact of the GPU overhead.

The size of the data used by applications

As experiments demonstrated in Section 5.3, the potential advantages of GPU-based compression may not overcome the overhead introduced for relatively small data. Specifically, GPU-based libraries became relevant with data sizes greater than *50KB* for the *Deflate* library, *500KB* for the *Snappy* and *Zstd* libraries, and *5MB* for the *Lz4* library.

When we analyzed the histogram of data transfers for running the TF applications under study (see Section 6.1 and Figure 6.2), we observed that the size of data used in transfers with higher frequency (i.e., the most common ones) are similar for all the applications: between *1B* and *1KB*. More than 90% of data transfers are in this range.

Thus, the data size plays a critical role, and applications use tiny data sizes. Therefore, even if we used GPU-based compression libraries, the results would not be as good as those of CPU-based compression libraries.

The incompatibility of the GPU with the processing pipeline

Using compression in pipelined communication seeks to increase the network bandwidth by hiding compression and decompression time with the compressed data transfer time. However, the way to work with GPUs is to divide the task into small chunks. Then, these chunks are executed in parallel by GPU threads, and the task is only considered complete once all threads have finished processing their respective portions.

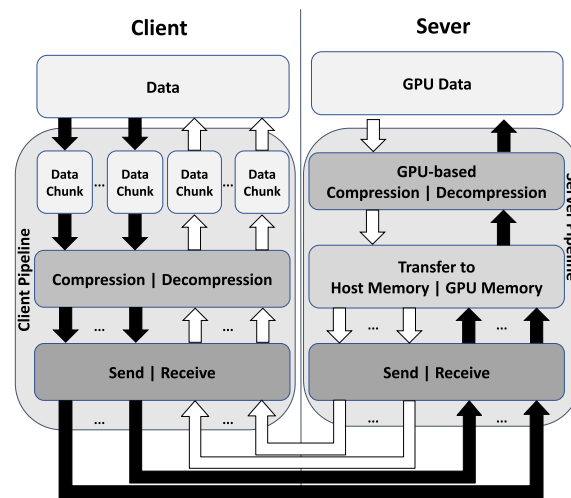


Figure 6.16: Scenario with an alternative approach for integrating on-the-fly GPU-based compression in rCUDA pipeline communication layer.

Individual GPU threads are not as computationally powerful as CPU ones, but the GPU compensates this with a significantly larger number of threads that can work simultaneously. This massive parallelism to process data is where the GPU takes advantage. GPU-based compression libraries also work in this way. Thus, data is divided into small chunks that are compressed/decompressed in parallel. Once all chunks are processed, the compression/decompression task is considered complete.

If data were compressed/decompressed as it is received, as shown in Figure 6.15, the GPU potential would be wasted because only one GPU thread would be used. In this scenario, compression and decompression would become a bottleneck, notably limiting performance.

An alternative approach would be to wait until all the data is available before performing compression/decompression, as shown in Figure 6.16. While this method ensures that the GPU can operate more efficiently on larger data chunks, it introduces another issue: it disrupts the pipeline. Specifically, this approach does not effectively hide the compression and decompression time on the server side, which is a core objective of using a compression pipeline in the first place. Therefore, none of the options analyzed achieves the desired balance between maximizing GPU efficiency and maintaining the performance benefits of the pipeline.

TensorFlow Application	Number of data transfers	Percentage of data transfers	Number of data transfers with different data values
<i>Alexnet</i>	15,218	42.19%	2,820
<i>Cifar10</i>	33,067	44.27%	10,479
<i>Mnist</i>	83,665	31.92%	15,855
<i>Inception</i>	97,346	47.67%	25,530

Table 6.4: Characterization of the 8-byte data transfers performed by the TF applications used in the experiments.

The impact of the GPU overhead

In remote GPU virtualization systems, there are typically two key components: (i) applications aiming to boost their performance by offloading part of their computation to remote GPUs, and (ii) servers with physical GPUs that process the request sent by these applications.

Given that these servers constantly process requests on their physical GPUs, the GPUs are expected to have a high workload. Thus, if the GPU also handled compression and decompression tasks, the GPU workload would increase, directly affecting the application requests and reducing the performance of compression/decompression.

On the contrary, the CPU is usually not as overloaded as the GPU in these systems. Therefore, delegating compression/decompression tasks to the CPU could be more efficient because it allows GPUs to focus exclusively on critical workloads, such as application requests performing intensive computations.

6.4.4 Optimizations focused on small data

We observed that the size of data transfers in the range of ‘[8B-16B]’ represents more than 30% of all the data transfers done by TF applications. Moreover, we discovered that all data transfers have a size of 8 bytes within that range. This means that data can have up to 2^{64} different values. In Table 6.4, we show more information regarding these data transfers. Despite the large number of 8 byte data transfers, TF applications use less than 65,535 (2^{16}) different data values. Thus, these data could be represented by 2 bytes instead of 8. This optimization would improve the results of our approach. For example, the Inception application makes 97,346 data transfers. If the data size were 2 bytes instead of 8 bytes, it would reduce these transfers by 570.39KB.

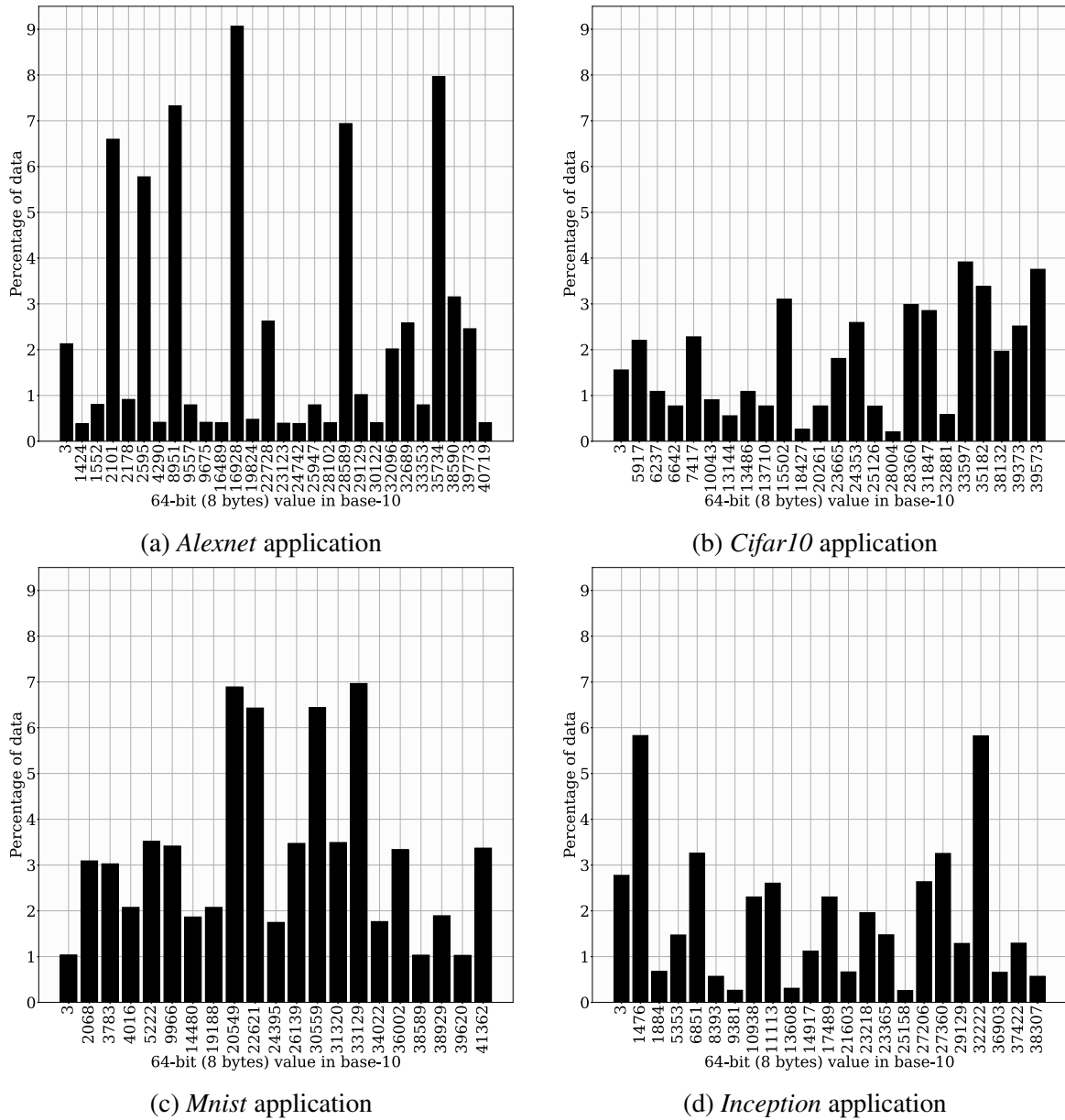


Figure 6.17: Percentage of data transfers of the most repeated 8-byte data values when running the TF applications under study.

Although there are several 8-byte data transfers with different data values, some are more repetitive than others. Figure 6.17 illustrates the frequency of the most repeated ones when running the TF applications, discarding the transferred data with a frequency below 0.2%. As we can see, the most repeated 8-byte data depend on the TF application. The aggregate frequency of data transfers for these data represents between 42.69% and 67.98% of all 8-byte data transfers. Thus, we can improve even more the performance if we handle these data values differently from the rest of the 8-byte data values. In this way, we would use only 1 byte for these specific data values because there are fewer than 256 data values (around 20 different values per application). The rest 8-byte data can continue to use the 2 bytes concluded from the previous optimization. Considering that these data values represent a high percentage of the transfers, this optimization would considerably reduce the size of the data sent over the slow network.

Our analysis has demonstrated that a significant portion of the transfers consist of only 8-byte values and that the number of unique transferred values is limited. However, no experimental results have been conducted to evaluate the impact of reducing their representation size. The main reason is that implementing and integrating such an improvement requires deep study and is beyond the scope of this study. In addition, despite reducing the data size for these specific applications, we see that their 8-byte data differs from the rest of the TF applications. The main objective of this thesis is to improve the bandwidth networks using compression on remote GPU virtualization systems in a general way. However, the application of this solution would only be particularly beneficial for these four TF applications.

6.5 An optimized parallel pipeline approach to on-the-fly compression

Section 6.4 presented on-the-fly pipelined compression implemented within the rCUDA communication layer, where compression, transmission, and decompression tasks overlap. This approach was primarily designed to evaluate the impact of compression on the existing rCUDA communication pipeline without modifications. However, we have not appreciated relevant enhancements concerning the non-pipelined on-the-fly compression approach in our experiments. The reason was that the pipeline uses a chunk size of 1KB, and more than 90% of data transfers done by TF applications have a size below this size. Thus, the pipeline mechanism was not leveraged in these experiments. It should be noted that the primary

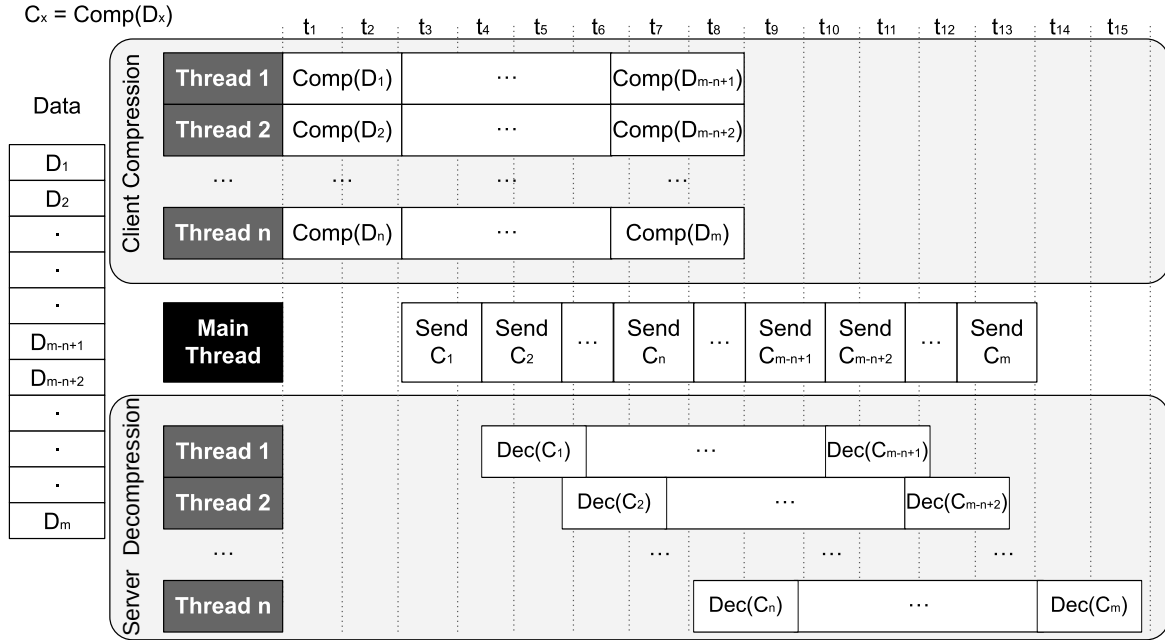


Figure 6.18: A high-level diagram of the implementation of the parallel compression pipeline using n compression/decompression threads and m data chunks.

objective was to assess performance within the current framework rather than to optimize or redesign the pipeline.

As explained in Section 6.4, developing an efficient compression pipeline is essential to maximize performance. To do this, the pipeline aims to overlap and execute steps simultaneously. The latency can be significantly reduced by processing different stages in parallel (compressing the data, transmitting through the network, and decompressing once received). In this section, we research an optimized parallel compression pipeline seeking the maximum overlap between the compression/decompression stages and the data transfers. Moreover, we evaluate it using a more sophisticated analysis, where specific data is used with data sizes from 1B up to 16MB.

Figure 6.18 presents a high-level implementation of the proposed parallel compression pipeline. The proposal shows how n threads work on m data chunks. On the client side, n threads work in parallel compressing data chunks in order for the main thread. The main thread sends these compressed data chunks over the network. There is always a new data chunk ready to be sent. On the server side, n threads work in parallel, decompressing data chunks as they arrive. In this example, a compression thread takes $2t$ to compress a piece of data, while the main thread takes $1.5t$ to transfer this compressed piece. If only one compression thread were used instead of n compression threads, the main thread

would have to wait for $0.5t$ between the current transfer and the following one. Having several decompression threads is also necessary to improve performance. Similar to what happens with compression threads, in this example the decompression thread takes $1.8t$ to decompress the compressed chunk. If only one decompression thread were used instead of n decompression threads, a latency of $0.3t$ would be added to the decompression time.

In Figure 6.18, we have seen the importance of appropriately optimizing the different stages of the pipeline. In particular, there are some parameters to consider for implementing the most efficient possible pipeline:

- **Number of compression/decompression threads.** Compression and decompression times vary depending on the utilized data. In an ideal scenario, the main thread must transfer compressed chunks over the network all the time without being idle. Thus, there should always be a piece of data compressed and ready to be sent. This could be achieved by increasing the number of compression/decompression threads.
- **Number of data chunks.** The result of compressing a data chunk must be saved until the main thread has sent it over the network. Increasing the number of available compressed data chunks allows compression/decompression threads not to be idle. Moreover, it allows the main thread to have more pieces of data compressed and ready to be sent. Thus, an appropriate number of data chunks must be selected so that the threads always have work to do.
- **Size of data chunks.** Choosing the best data chunk size is also very important. Compression takes longer with a large data chunk than with a smaller one. However, the compression rate is usually better for a large data chunk than for a smaller one. In addition, better compression could also lead to a faster transfer.

Of course, the ideal scenario provides the pipeline with all the data chunks and all the threads it needs. However, that is not always possible due to resource limitations in the real world. Therefore, finding a trade-off between the resources used and the performance achieved is necessary. For this reason, next we analyze the influence on the performance of the number of threads, the number of compressed data chunks, and the size of the data chunks.

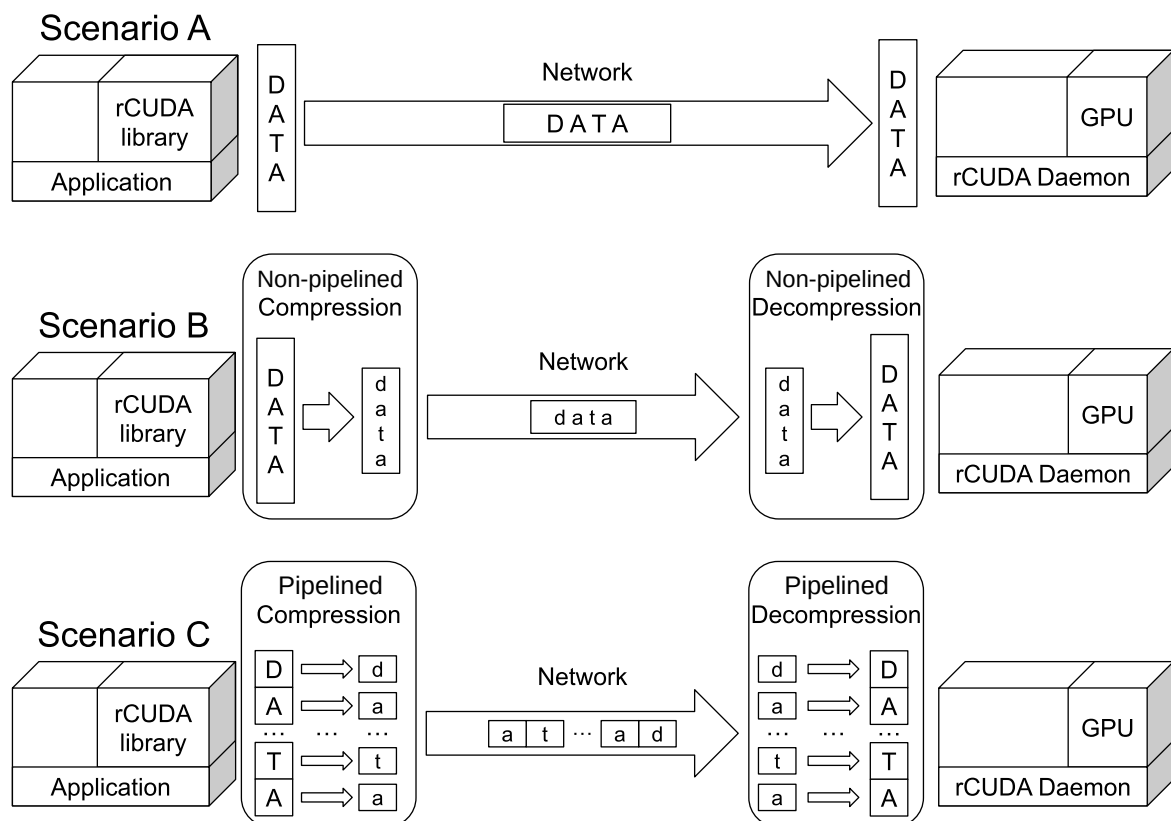


Figure 6.19: Scenarios explored in the experiments.

6.5.1 Experimental setup

In this study, we consider the scenarios shown in Figure 6.19 to explore the benefits of using a parallel compression pipeline system:

- Scenario A represents the initial scenario where the remote GPU virtualization solution rCUDA is used without compression.
- Scenario B shows an improvement over the previous scenario. Transferred data is compressed by rCUDA. This compression has been previously presented in Section 6.1.
- Scenario C uses the optimized parallel compression pipeline. Compression and decompression stages are overlapped with data transfers by rCUDA.

Notice that previous experiments compared the results with tests performed locally. However, the current tests aim to develop an optimized parallel pipeline solution to enhance client-server

Client Node	
Architecture	x86_64
Operating System	Ubuntu18.04.5LTS
Processor	Intel(R) Xeon(R) CPU E5-2637 v2 @ 3.50GHz
RAM	64GB

Server Node	
Architecture	x86_64
Operating System	Ubuntu20.04.4LTS
Processor	AMD EPYC 7282 16-Core Processor
RAM	256GB
GPU	NVIDIA A100

Table 6.5: Specifications of the client and server node used in CPU-based compression libraries evaluation when a parallel compression pipeline communication is used.

communication. Since this communication does not exist in local tests, direct comparisons are not relevant in this context. Note also that the compression and decompression stages shown in scenarios B and C are implemented inside the communication layer of rCUDA. Therefore, their use is transparent to the rest of the system.

Previous experiments were done in an edge environment using a Raspberry Pi 4 B. This time, we want to explore the possibilities of a parallel compression pipeline system. For this reason, we have used an Intel(R) Xeon(R) CPU E5-2637 v2 3.50GHz as the client node and an AMD EPYC 7282 16-Core Processor with an NVIDIA A100 GPU as the server node in the experiments. That decision is driven by the need to evaluate first the feasibility of our solution under controlled conditions. The primary objective is to analyze the performance and effectiveness of the parallel pipeline compression communication system without being constrained by the limited computational power of the Raspberry Pi 4 B. Using a more powerful machine ensures that any observed performance trends result from the system rather than hardware limitations. We can observe the complete specifications of the experimental setup in Table 6.5. The connection between the client node and the server node was a 1 Gbps Ethernet wired network. As in previous experiments, the Linux traffic shaper (i.e., *tc*) was used to reduce the network bandwidth to that is usually available in edge environments, such as WiFi or cell phone networks. By doing this, we were able to evaluate networks with the following bandwidths: 1 Gbps, 100 Mbps, and 10 Mbps.

In the experiments, we have used the compression libraries provided by the *CPU-Smash* compression abstraction library. In particular, after examining all the compression libraries

Lineal	0	1	2	3	4	5	6	...	252	253	254	255	0	1	2	...
Random	246	65	104	196	250	44	150	43	153	52	181	45	170	231	24	...
Sparse	0	0	0	246	0	65	104	0	0	0	0	0	196	250	44	...
Traced	24	8	6	2	4	3	3	3	6	1	5	9	7	7	2	...

Figure 6.20: Data used in experiments.

that *CPU-Smash* provides, we have selected the following four libraries: *Snappy*, *Gipfeli*, *Lz4*, and *Lzo*. The reason for choosing these CPU-based compression libraries is twofold: (1) they provide the best performance, and (2) despite being the fastest libraries, they present different features. For instance, *Lzo* is much more computationally intense than *Gipfeli*. This will provide helpful information about the parallel compression pipeline communication system.

Previous experiments presented in Section 6.1 and Section 6.4 have demonstrated that CPU-based compression libraries can artificially enhance the bandwidth of networks. However, the applications utilized do not facilitate the compression system. More than 90% of data transfers done by these applications use data sizes lower than 1KB, but the enhancements of compression systems are more noticeable with higher data sizes. In order to focus our analysis on the benefits of using a pipelined on-the-fly compression approach to artificially increase network bandwidth, we have used the bandwidth test developed by NVIDIA, which can be obtained when downloading the CUDA software. Using this bandwidth test, we can evaluate our system depending on the data sizes instead of the application executions. Given that the properties of compression algorithms greatly depend on the exact data being compressed, we have modified the bandwidth test program to consider the following four data types, all of them represented in Figure 6.20:

- **Lineal data.** Data start from 0 and increase one by one up to 255. Once the value 255 is reached, it starts again from 0.
- **Random data.** Data are composed of random numbers between 0 and 255.
- **Sparse data.** Data contain a random set of 0s followed by a random set of numbers between 0 and 255.
- **Traced data.** Real traces obtained from TF applications: *Alexnet*, *Cifar10*, *Mnist*, and *Inception*. This data has been presented in Chapter 3. This data represents the data

exchanged between the host and the GPU memory during the execution of the different TF applications used in experiments shown in Section 6.1 and Section 6.4.

On the one hand, random data are the worst scenario because the information does not follow any pattern. We expect very low compression ratios and, therefore, bad results in general when using this data type. On the other hand, Lineal and Sparse data contain repetitive information and follow patterns, which can help compression libraries perform well. These three data types were generated synthetically, and their results can be helpful to get insights from our study. However, using traced data seems to be more representative of the environment targeted by this study. In the following sections, we present results for these four data types.

6.5.2 Finding the best parameters for the parallel compression pipeline

The performance of the parallel compression pipeline system depends on: (i) the number of concurrent threads used for compressing and decompressing data, (ii) the amount of data chunks that are concurrently being processed within the pipeline, and (iii) the size of these chunks. The design space can be considerable, thus, we only consider the values shown in Table 6.6 for each parameter. We analyze the performance of the pipeline by varying the number of threads, the number of data chunks, and the size of those data chunks according to the values in the table.

Notice that if we do not limit the usage of resources (i.e., memory and CPU usage), the parallel compression pipeline system will usually obtain better performance as the amount of available resources increases. For example, we can expect that eight threads provide the best performance. However, the performance improvement is not strictly related to the resource

Number of Threads	Number of Chunks	Size of Chunks
	2	512B
1	4	1KB
2	8	2KB
4	16	4KB
8	32	32KB
	64	128KB

Table 6.6: Parameters used to analyze the performance of the parallel compression pipeline system.

increment, i.e., using twice as many threads or memory does not double the benefits. An elaborated parallel compression pipeline system, which saves resources and obtains a good performance, is essential. Therefore, we need to find a trade-off between the number of available resources and the performance achieved.

In a real scenario, we are limited by the available resources and should preferably use as few resources as possible. Considering this, we have developed an equation to evaluate the achieved performance, considering the number of resources used to obtain that performance. Equation 6.3 represents the trade-off between the bandwidth and the resources for a given parameter configuration. Note that Bw represents the bandwidth, $Size$ is defined as the chunk size times the number of chunks (as shown in Equation 6.2), and Th is the number of threads.

$$Size = ChunkSize \times NumberChunks \quad (6.2)$$

$$Trade-Off(x) = \left(\frac{Bw_x}{Bw_{min}} \times \alpha \right) - \left(\frac{Size_x}{Size_{min}} \times \beta \right) - \left(\frac{Th_x}{Th_{min}} \times \delta \right) \quad (6.3)$$

In Equation 6.3, we first normalize the bandwidth for each value by dividing it by the minimum bandwidth obtained. The size and the number of threads are normalized in a similar way. Finally, we weighted these normalized values with the user-defined constants: α , β , and δ . The sum of those constants must be one. If the α value is close to one, the user gives more importance (i.e., prioritizes) to bandwidth. On the contrary, the β or δ values are near one when the user is more concerned about resource utilization.

Notice that the bandwidth value is positive because higher bandwidth is desirable. This term is added positively to favor configurations that achieve better performance. However, the size value and the number of threads are negative because a larger size indicates higher resource consumption. For this reason, they are subtracted to penalize configurations that increase data size or the number of threads.

In this study, we have selected the values 0.60, 0.15, and 0.25 for α , β , and δ , respectively. We give more weight to bandwidth (α is the highest value) because the performance of our approach greatly depends on it. In addition, computing nodes have limited computational power (i.e., number of threads) than memory (i.e., size of the data chunks) in our experimental setup. Thus, the δ weight is higher than the β one. Using this trade-off value, we decide

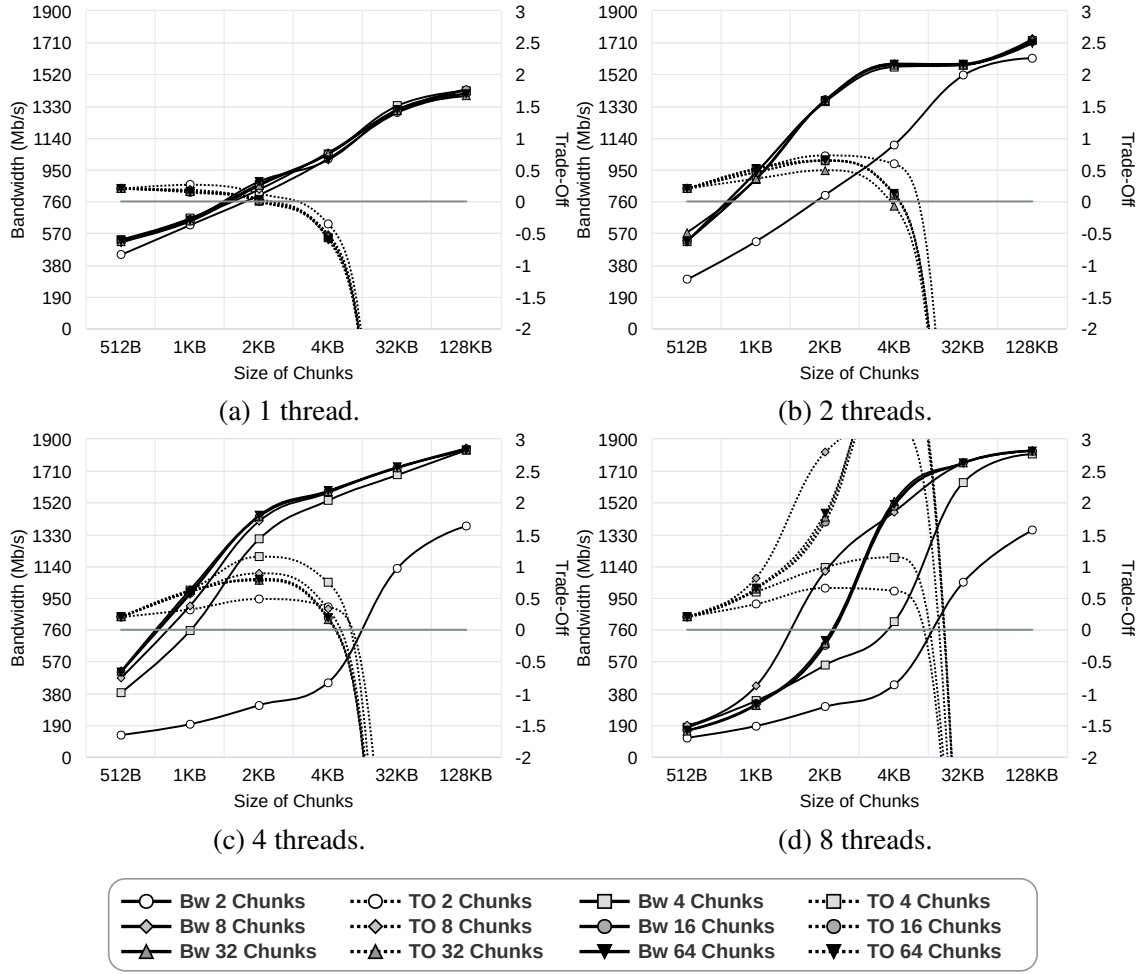


Figure 6.21: Study of data chunk sizes using a 1 Gbps network and *Gipfeli* compression library. Each figure presents the bandwidth ('Bw') obtained using a different number of threads and data chunks. The horizontal line in the plots corresponds to a trade-off ('TO') equal to zero.

whether the achieved bandwidth is good. If the trade-off value is greater than 0, we can consider this setting, but if the value is lower than 0, it is discarded. Finally, the best configuration will be the one that gets the higher bandwidth, and its trade-off value is greater than 0.

Considering all the combinations of parameters shown in Table 6.6 is not feasible. Thus, to reduce the exploration, we first focus on analyzing the impact of the size of data chunks. Then, by each number of threads and data chunks, we select the best chunk size using the trade-off detailed in Equation 6.3. In this way, we further reduce the exploration by analyzing the impact of the number of chunks per thread. Finally, we use those conclusions to study how the number of threads impacts performance. We carry out this analysis considering

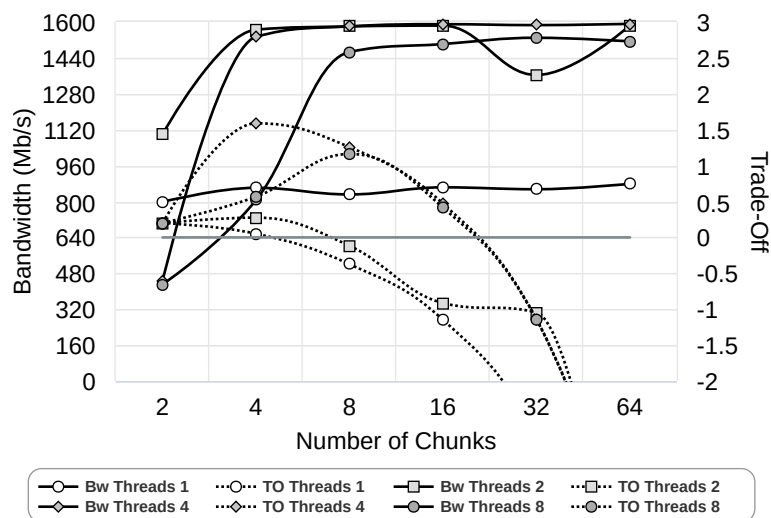


Figure 6.22: Study of the number of data chunks using a 1 Gbps network and *Gipfeli* compression library.

Number of Threads	Number of Chunks	Size of Chunks
1	4	2KB
2	4	4KB
4	16	4KB
8	16	4KB

Table 6.7: Summary of the best parameters for the parallel compression pipeline system using *Gipfeli* and a 1 Gbps network for each number of threads used in the pipeline.

only messages with 8MB of data. We have chosen this message size because it is large enough to keep the parallel compression pipeline working while measuring performance. In the next section, we show the performance of the entire range of message sizes. Notice that in this section we aim to show how the best features are selected depending on the compression library and data type. As an example, we only show results for the *Gipfeli* compression library when using the 1 Gbps network and the traced data. The results for the rest of compression libraries, network bandwidths, and data types have been obtained following the same strategy.

In the first step, we investigated the best data chunk size depending on the number of threads and data chunks. Figure 6.21 presents the results obtained by the *Gipfeli* compression library using a 1 Gbps network. As expected, the bandwidth (denoted as “Bw” in the plots) increases as more resources are dedicated to the pipeline in all scenarios. However, the trade-off between resources and bandwidth (denoted as “TO” in the plots) worsens as the memory used increases. The reason is that the bandwidth does not improve enough to compensate

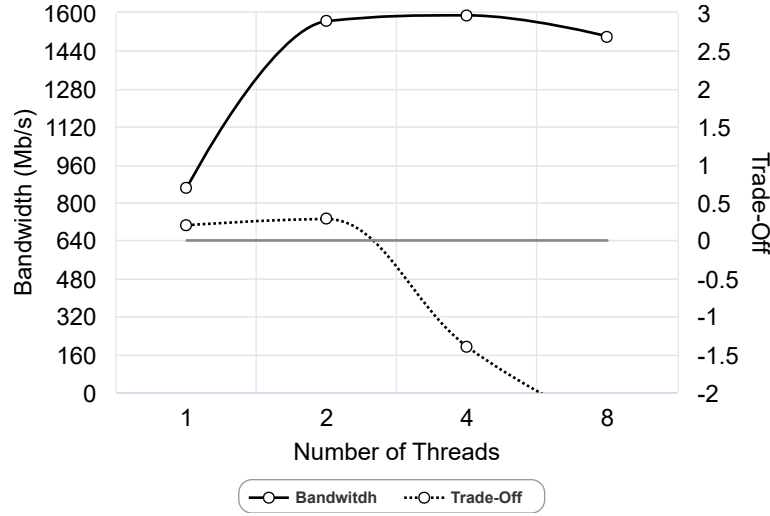


Figure 6.23: Study of the number of threads using a 1 Gbps network and *Gipfeli* compression library.

for the memory increment. The chunk size that achieves the best performance using one thread is *2KB*, regardless of the number of chunks, as shown in Figure 6.21a. Figure 6.21b presents a similar behavior when two threads are used. The optimal size is *4KB* for almost any number of chunks, except for 32 chunks, where the best chunk size is *2KB*. Finally, when using four and eight threads, regardless of the number of chunks, the best performance is obtained when using chunk sizes of *4KB*, as illustrated in Figure 6.21c and Figure 6.21d.

Once data chunk sizes have been investigated, the number of data chunks is the following parameter to be studied. Figure 6.22 presents this analysis, where we have set the best size according to previous results shown in Figure 6.21 for each number of threads (1, 2, 4, and 8). We can appreciate how the trade-off between resources and bandwidth worsens as the amount of chunks increases. The reason is that the total size used increases exponentially. Table 6.7 summarizes the results showing the best configurations. We selected the configuration that obtains better bandwidth.

Finally, we study the optimal number of threads when the chunk size and the number of chunks are set according to Figure 6.21 and Figure 6.22. Again, we use our equation to obtain the best trade-off between bandwidth and resources used. In Figure 6.23, we can observe that the bandwidth increases as the number of threads increases from one to two threads and from two to four threads. In the case of eight threads, bandwidth is reduced. However, the best trade-off is achieved with two threads. After having studied the three parameters, we can conclude that for the *Gipfeli* using a 1 Gbps network, the best configuration is two threads, with four chunks, and a chunk size of *4KB*.

Network	Library	Threads	Chunks	Size (KB)	Ratio	Compression + Decompression Time (μ s)	Bandwidth (Mb/s)
1 Gbps	<i>Snappy</i>	2	2	32	18.29	18.46	8830.11
	<i>Gipfeli</i>	1	4	4	8.94	6.96	3863.49
	<i>Lz4</i>	1	2	32	83.17	24.86	10245.63
	<i>Lzo</i>	4	8	4	14.52	55.32	2697.30
100 Mbps	<i>Snappy</i>	1	4	4	9.16	5.01	885.22
	<i>Gipfeli</i>	1	4	4	8.94	6.96	854.13
	<i>Lz4</i>	1	2	32	83.17	24.86	7423.55
	<i>Lzo</i>	2	4	4	14.52	55.32	1288.82
10 Mbps	<i>Snappy</i>	1	4	4	9.16	5.01	88.79
	<i>Gipfeli</i>	1	2	4	8.94	6.96	86.63
	<i>Lz4</i>	1	2	128	168.26	67.77	1574.96
	<i>Lzo</i>	1	2	32	75.16	352.52	698.11

Table 6.8: Summary of the best parameters for the parallel compression pipeline system using lineal data, different compression libraries, and networks.

Network	Library	Threads	Chunks	Size (KB)	Ratio	Compression + Decompression Time (μ s)	Bandwidth (Mb/s)
1 Gbps	<i>Snappy</i>	1	2	2	0.998	6.82	919.43
	<i>Gipfeli</i>	1	2	2	0.992	6.39	920.18
	<i>Lz4</i>	2	4	1	0.995	26.37	895.98
	<i>Lzo</i>	8	8	4	0.995	359.52	878.50
100 Mbps	<i>Snappy</i>	1	2	1	0.995	5.33	97.00
	<i>Gipfeli</i>	1	2	1	0.981	5.66	95.57
	<i>Lz4</i>	1	2	1	0.995	26.37	97.00
	<i>Lzo</i>	1	4	1	0.992	80.45	96.56
10 Mbps	<i>Snappy</i>	2	2	1	0.995	5.33	9.71
	<i>Gipfeli</i>	1	2	1	0.981	5.66	9.57
	<i>Lz4</i>	1	2	1	0.995	26.37	9.71
	<i>Lzo</i>	2	4	1	0.992	80.45	9.68

Table 6.9: Summary of the best parameters for the parallel compression pipeline system using random data, different compression libraries, and networks.

We have only shown plots for the *Gipfeli* compression library and 1 Gbps network when compressing 8MB of traced data. However, Table 6.8, Table 6.9, Table 6.10, and Table 6.11 summarize the best configurations obtained for all the compression libraries used when compressing lineal, random, sparse, and traced data, respectively.

Using lineal data (Table 6.8), the *Lz4* library obtains the best performance. Due to its high compression ratio, it can artificially increase the network speed ten times with a 1 Gbps network, 74 times with a 100 Mbps network, and 157 times with a 10 Mbps network. The rest of the compression libraries obtain a high performance, but there is a big difference with the results obtained by the *Lz4* one

Random data (Table 6.9) are difficult to compress, as we can observe by the compression ratio achieved by CPU-based compression libraries. That makes that no compression library can increase the network bandwidth with this data, independently of the resources used. Despite that, results are close to the bandwidth without using a compression system.

With sparse data (Table 6.10), fast compression libraries (such as *Gipfeli* and *Snappy*) obtain the best network bandwidth with 1 Gbps and 100 Mbps networks, increasing it by 3 and 19 times, respectively. However, despite the slow compression and decompression speed obtained by the *Lz4* compression library, it can considerably reduce the size of the transferred data, increasing by more than 65 times the bandwidth with a 10 Mbps network.

Finally, with traced data (Table 6.11), *Gipfeli* gets better bandwidth than others using 1 Gbps, despite being *Lzo* the compression library with the best compression ratio. This is because the compression and decompression time is the bottleneck in an environment with 1 Gbps networks, and *Lzo* takes more time to compress and decompress than *Gipfeli*. However, we can see the importance of obtaining a good compression ratio by reducing the network bandwidth. Thus, when using 100 Mbps and 10 Mbps networks, *Lzo* performs better than the other compression libraries because transfer time compensates for the extra time spent in compression and decompression. This can also be seen in the resources. All compression libraries require fewer resources as network bandwidth decreases. With traced data, the bandwidth has been increased by CPU-based compression libraries around 1.5 times.

6.5.3 Impact on the parallel compression pipeline performance

In the previous section, we selected the best configuration parameters for our parallel compression pipeline system when 8MB messages were used. In this section, we evaluate

Network	Library	Threads	Chunks	Size (KB)	Ratio	Compression + Decompression Time (μs)	Bandwidth (Mb/s)
1 Gbps	<i>Snappy</i>	1	2	4	20.50	7.12	3401.36
	<i>Gipfeli</i>	1	4	4	18.63	7.86	3445.01
	<i>Lz4</i>	1	2	1	69.37	27.83	385.93
	<i>Lzo</i>	2	8	1	80.02	22.77	833.96
100 Mbps	<i>Snappy</i>	1	2	1	19.35	2.22	1889.86
	<i>Gipfeli</i>	1	2	4	18.63	7.86	1727.83
	<i>Lz4</i>	2	4	1	69.37	27.83	750.15
	<i>Lzo</i>	2	8	1	80.02	22.77	828.20
10 Mbps	<i>Snappy</i>	1	2	4	20.50	7.12	174.77
	<i>Gipfeli</i>	1	4	1	15.41	2.49	141.61
	<i>Lz4</i>	2	4	2	104.31	57.15	657.80
	<i>Lzo</i>	2	4	1	80.02	22.77	531.81

Table 6.10: Summary of the best parameters for the parallel compression pipeline system using sparse data, different compression libraries, and networks.

Network	Library	Threads	Chunks	Size (KB)	Ratio	Compression + Decompression Time (μs)	Bandwidth (Mb/s)
1 Gbps	<i>Snappy</i>	1	2	4	2.25	17.81	1345.68
	<i>Gipfeli</i>	2	4	4	2.36	55.39	1565.04
	<i>Lz4</i>	4	8	2	2.24	74.26	1190.93
	<i>Lzo</i>	8	8	2	2.52	159.54	763.29
100 Mbps	<i>Snappy</i>	1	4	1	1.49	7.96	127.41
	<i>Gipfeli</i>	1	4	1	1.54	21.84	135.02
	<i>Lz4</i>	1	4	1	1.73	40.43	134.24
	<i>Lzo</i>	2	4	1	1.94	91.56	148.42
10 Mbps	<i>Snappy</i>	1	4	1	1.49	7.96	12.74
	<i>Gipfeli</i>	1	4	1	1.54	21.84	13.51
	<i>Lz4</i>	1	2	1	1.73	40.43	13.48
	<i>Lzo</i>	1	2	1	1.94	91.56	14.87

Table 6.11: Summary of the best parameters for the parallel compression pipeline system using traced data, different compression libraries, and networks.

the performance of the scenarios described in Section 6.5.1 for the entire range of message sizes. Figure 6.24, Figure 6.25, and Figure 6.26 illustrate the results obtained using 1 Gbps, 100 Mbps, and 10 Mbps network speeds, respectively. These figures present the results for three different cases: (i) rCUDA without using compression, referred to as ‘No Compression’, (ii) rCUDA using the parallel compression pipeline system with the CPU-based compression libraries, labeled as ‘*Library Pipelined*’, and (iii) rCUDA using compression with the CPU-based compression libraries but without pipeline, marked as ‘*Library Non-pipelined*’. These different cases have been evaluated with lineal, random, sparse, and traced data. Note that the pipeline cases use the best configuration that depends on the network bandwidth and the kind of data utilized. These configuration have presented in Tables 6.8, 6.10, 6.9, and 6.11.

Using a 1 Gbps network and lineal data (Figure 6.24a), the non-pipelined version of *Lz4* and *Snappy* obtains the best results. The ‘*Lz4 Non-pipelined*’ can enhance the network bandwidth achieved by ‘No Compression’ close to 7 times at its peak. However, the bandwidth achieved by the ‘*Lz4 Pipelined*’ only improves this bandwidth by 1.5 times. The reason is that the non-pipelined version uses all the data, and this kind of data is easy to compress. Thus, this version achieves a compression ratio higher than the pipeline one.

On the other hand, no compression library improves the bandwidth obtained by ‘No Compression’ with random data and a 1 Gbps network (Figure 6.24b). However, the performance of pipelined versions is usually better than the non-pipelined ones because pipelined versions start to copy the data through the network earlier.

Using a 1 Gbps network and the sparse data (Figure 6.24c), the best results are obtained by *Gipfeli* and *Snappy* when the pipeline compression is utilized, improving the ‘No Compression’ bandwidth by up to 3.5 times at some points. With this kind of data, non-pipelined versions get worse results than the pipelined ones.

Finally, we can observe the results of using a 1 Gbps network and the traced data in Figure 6.24d. The pipeline version of *Gipfeli* and *Snappy* achieves a bandwidth of over 1500 Mbps for big data, which is 1.5 times the ‘No Compression’ bandwidth. The benefit of using a parallel compression pipeline system becomes more noticeable compared to the non-pipelined version, which uses compression without the pipeline. In that case, not using compression presents better results (near 1000 Mbps).

In the results using a 1 Gbps network shown in Figure 6.24, we observe that the non-pipelined version of CPU-based compression libraries can obtain better bandwidth than a no-compression and a pipelined compression system, depending on the data used. As

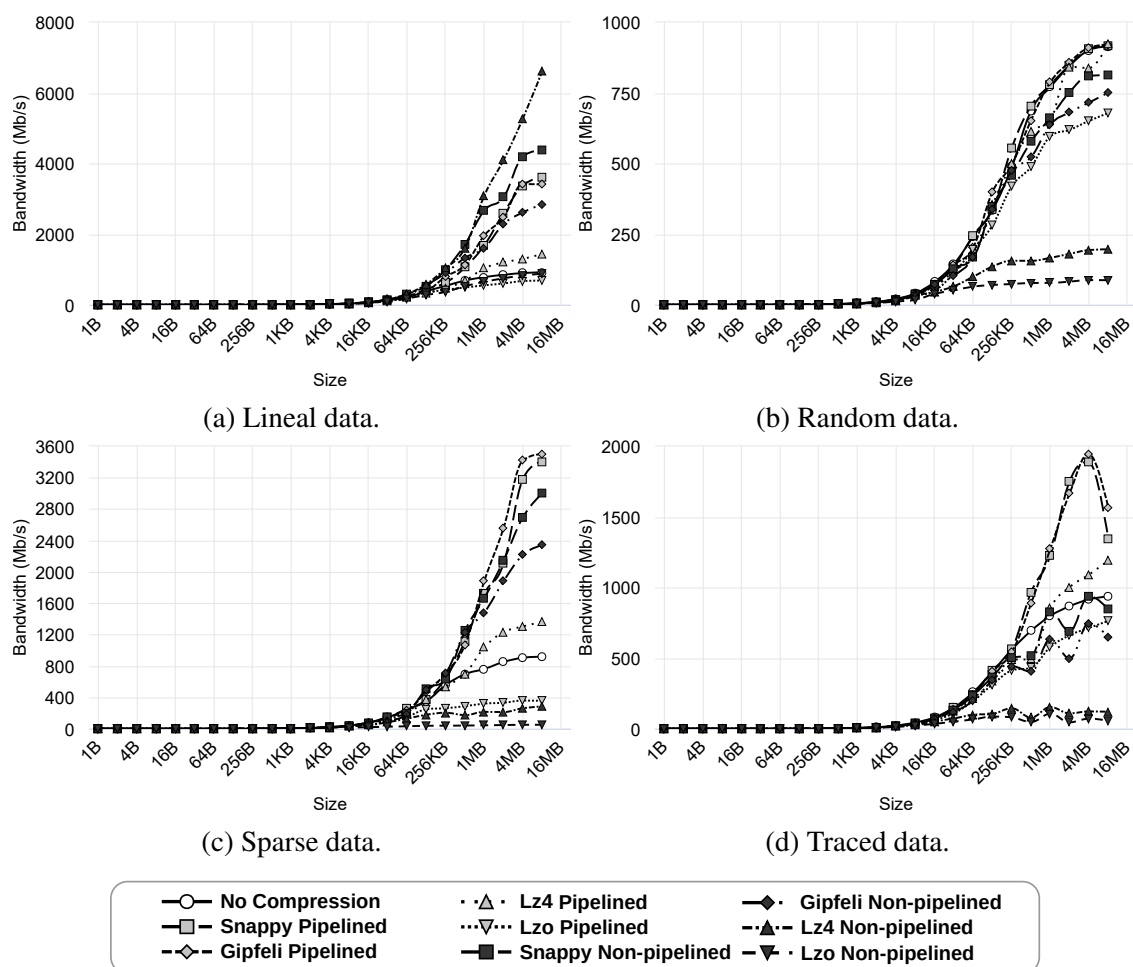


Figure 6.24: Results using a 1 Gbps network. Different scenarios are considered: ‘No Compression’ (rCUDA without using compression); ‘Library Pipelined’ (rCUDA using the parallel compression pipeline system with specific CPU-based compression library); ‘Library Non-pipelined’ (rCUDA using compression with the specific CPU-based compression library, without the pipeline system). The scenarios are evaluated with datasets of different sizes. More details about these scenarios are in Section 6.5.1.

the lineal data results illustrated in Figure 6.24a, no pipeline system is required if data are compressible enough. Moreover, we can appreciate how the *Lzo* compression library cannot improve the results obtained by not using compression on any kind of data due to the high time it takes to compress and decompress.

Figure 6.25 presents the results using a 100 Mbps network. Using lineal data (Figure 6.25a), the *Lz4* compression library continues to obtain the best results using its non-pipelined version. This version can improve the ‘No Compression’ bandwidth by near 70 times at its peak. We can also observe that all the non-pipelined versions of the CPU-based compression libraries work better than their pipelined ones because the compression ratio achieved by non-pipelined versions is higher, and the bandwidth network becomes the bottleneck. Again, with random data (Figure 6.25b), no CPU-based compression library can enhance the ‘No Compression’ bandwidth. One more time, using sparse data (Figure 6.25c), *Snappy* and *Gipfeli* compression libraries get the best performance. However, although their pipelined versions obtain high bandwidth, their non-pipelined versions improve them. The ‘No Compression’ bandwidth is enhanced by more than 15 times at some points. Finally, similar results are achieved using traced data (Figure 6.25d). *Gipfeli* and *Snappy* libraries are the best compression libraries. They attain a maximum bandwidth of over 300 Mbps using their non-pipelined version, 30 times more than the ‘No Compression’ bandwidth.

The aim of implementing the parallel compression pipeline system is to hide the compression and decompression time by overlapping it with the transfer time. Nevertheless, the compression system without a pipeline performed better when the bandwidth was reduced to 100 Mbps. The network becomes the bottleneck in our communication system. That makes the version that achieves a better compression ratio gets better results. In this way, non-pipelined versions obtain a better compression ratio than pipelined ones because they compress the whole data using all the information of that data, such as the *Gipfeli* compression library. However, we can observe slower CPU-based compression libraries, such as the *Lzo* one, where the pipelined version improves the non-pipelined version bandwidth. Let us compare these two compression libraries (*Gipfeli* and *Lzo*) with traced data and 1 Gbps (Figure 6.24d) and 100 Mbps (Figure 6.25d) networks. We can observe how *Gipfeli* versions get better bandwidth than *Lzo* ones with 1 Gbps. The ‘*Gipfeli* Non-pipelined’ bandwidth is close to the ‘*Lzo* Pipelined’ one. Using this bandwidth network, pipelined versions work better than their non-pipelined versions. However, using a 100 Mbps network, we appreciate that the ‘*Gipfeli* Non-pipelined’ improves all compression libraries, including its pipelined version, whereas the ‘*Lzo* Non-pipelined’ bandwidth is the worst, even worse than without

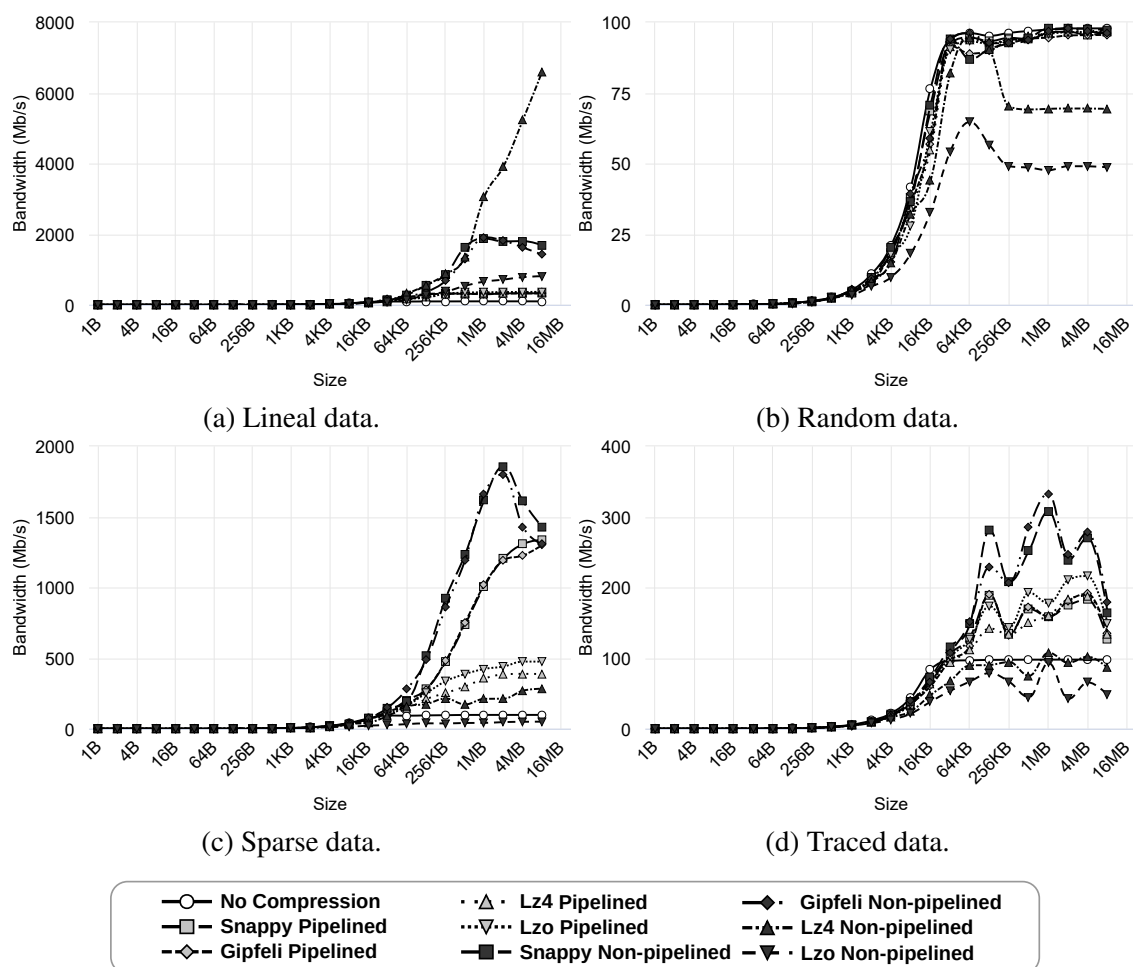


Figure 6.25: Results using a 100 Mbps network. Different scenarios are considered: ‘No Compression’ (rCUDA without using compression); ‘Library Pipelined’ (rCUDA using the parallel compression pipeline system with specific CPU-based compression library); ‘Library Non-pipelined’ (rCUDA using compression with the specific CPU-based compression library, without the pipeline system). The scenarios are evaluated with datasets of different sizes. More details about these scenarios are in Section 6.5.1.

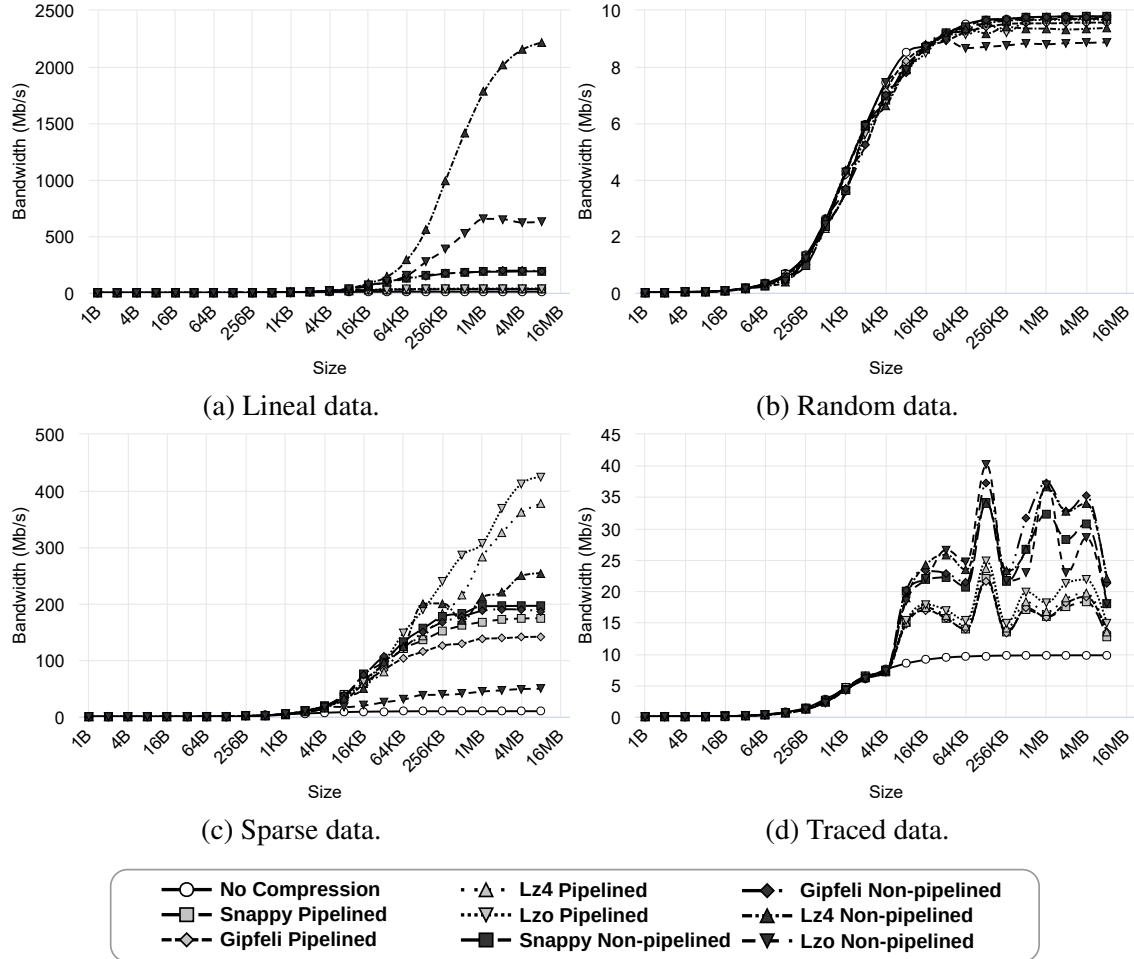


Figure 6.26: Results using a 10 Mbps network. Different scenarios are considered: ‘No Compression’ (rCUDA without using compression); ‘Library Pipelined’ (rCUDA using the parallel compression pipeline system with specific CPU-based compression library); ‘Library Non-pipelined’ (rCUDA using compression with the specific CPU-based compression library, without the pipeline system). The scenarios are evaluated with datasets of different sizes. More details about these scenarios are in Section 6.5.1.

compression. On the other hand, the ‘*Lzo* Pipelined’ improves its bandwidth broadly with respect to using a 1 Gbps network. It improves even the ‘*Gipfeli Pipeline*’ bandwidth.

Finally, Figure 6.26 presents the results using a 10 Mbps network. We obtain similar results to using a 100 Mbps network with lineal (Figure 6.26a) and random data (Figure 6.26b). With the first ones, non-pipelined versions outperform pipelined ones. The *Lz4* compression library obtains the best bandwidth, enhancing the ‘No Compression’ one by more than 200 times at some points. With the last ones, no compression library enhances the ‘No Compression’ bandwidth. However, pipelined versions of *Lzo* and *Lz4* greatly enhance the ‘No Compression’ bandwidth of sparse data (Figure 6.26c), being the ones which the best bandwidth. They can improve this bandwidth about 40 times at some points. If we compare these results with 1 Gbps (Figure 6.24c) and 100 Mbps (Figure 6.25c) networks, we can observe how ‘*Lzo* Pipelined’ and ‘*Lz4* Pipelined’ bandwidths have increased as the bandwidth network has been reduced. As presented in Table 6.10, these CPU-based compression libraries obtain the best compression ratio but also get the worst compression and decompression speed. However, using a low-performance scenario as a 10 Mbps network, this compression ratio compensates for the extra time of compressing and decompressing. Finally, with traced data (Figure 6.26d), all versions of CPU-based compression libraries enhance the ‘No Compression’ bandwidth. Non-pipelined versions perform better than pipelined versions. The ‘*Gipfeli* Non-pipelined’ achieves the best bandwidth with some values near 35 Mbps, 3.5 times more than the ‘No Compression’ bandwidth.

6.5.4 To send or not to send compressed data

As commented, the parallel pipeline compression system presented in this paper compresses the data on-the-fly before sending it over the network. However, depending on the exact data contents, the resulting compressed data size can be larger than the original uncompressed data. This happens more often for small messages. For that reason, in this section, we evaluate the improvement of checking the compressed data size to decide whether to send the compressed or uncompressed data. Note that the latter will also avoid the time needed to decompress the data in the server.

We evaluate this improvement using the *Gipfeli* compression library, the traced data, and 1 Gbps, 100 Mbps, and 10 Mbps networks. Figures 6.27, 6.28, and 6.29 show the normalized bandwidths obtained by: (i) ‘No Compression’, where no compression libraries are used, (ii) ‘*Gipfeli* Pipelined’, where always the compressed chunk is sent, and (iii) ‘*Gipfeli* Pipelined

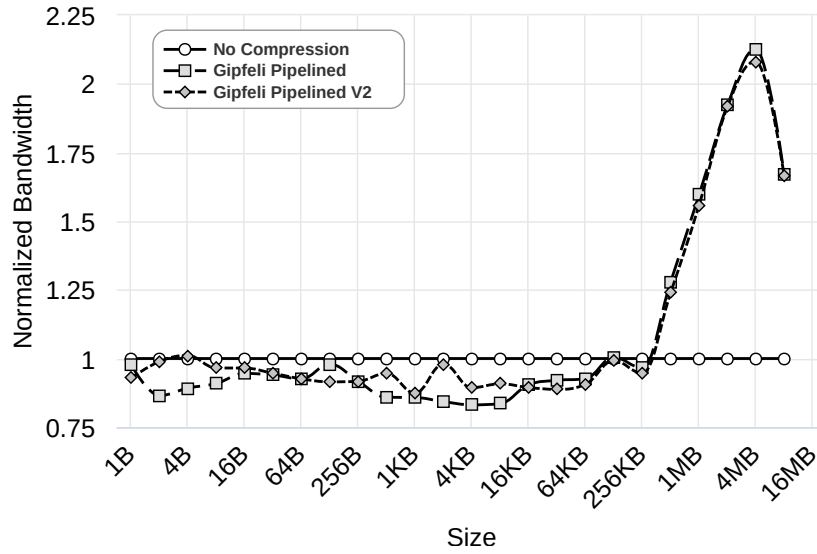


Figure 6.27: Normalized bandwidth obtained over a 1 Gbps network by ‘*Gipfeli Pipelined*’ and ‘*Gipfeli Pipelined V2*’ concerning the ‘No Compression’ one when using traced data. The ‘*Gipfeli Pipelined V2*’ only sends the compressed data if its size is smaller than the original uncompressed data.

V2’, where the parallel compression pipeline decides to send the uncompressed chunks or the compressed ones. These bandwidths have been normalized to the ‘No Compression’ ones. Thus, ‘No Compression’ values are always 1, while ‘*Gipfeli Pipelined*’ and ‘*Gipfeli Pipelined V2*’ will obtain normalized values greater than 1 when they improve the ‘No Compression’ bandwidth.

Figure 6.27 illustrates the results obtained with a 1 Gbps network. As we can see, ‘*Gipfeli Pipelined V2*’ achieves better bandwidth than ‘*Gipfeli Pipelined*’ when using data sizes lower than 16KB. Both get similar bandwidths for bigger data sizes, being the ‘*Gipfeli Pipelined*’ slightly better. Both approaches improve the bandwidth obtained by ‘No Compression’ when data sizes are greater than 256KB.

Similar results are obtained using a 100 Mbps network, as shown in Figure 6.28. Again, ‘*Gipfeli Pipelined V2*’ only achieves better values than ‘*Gipfeli Pipelined*’ for data sizes smaller than 16KB. Both outperform ‘No Compression’ for data sizes greater than 32KB, where the ‘*Gipfeli Pipelined*’ bandwidth slightly improves the ‘*Gipfeli Pipelined V2*’ one. Notice that using a 1 Gbps network, *Gipfeli* outperformed the ‘No Compression’ bandwidth for data higher than 256KB with 1 Gbps. However, using a 100 Mbps network, this threshold is reduced to 32KB.

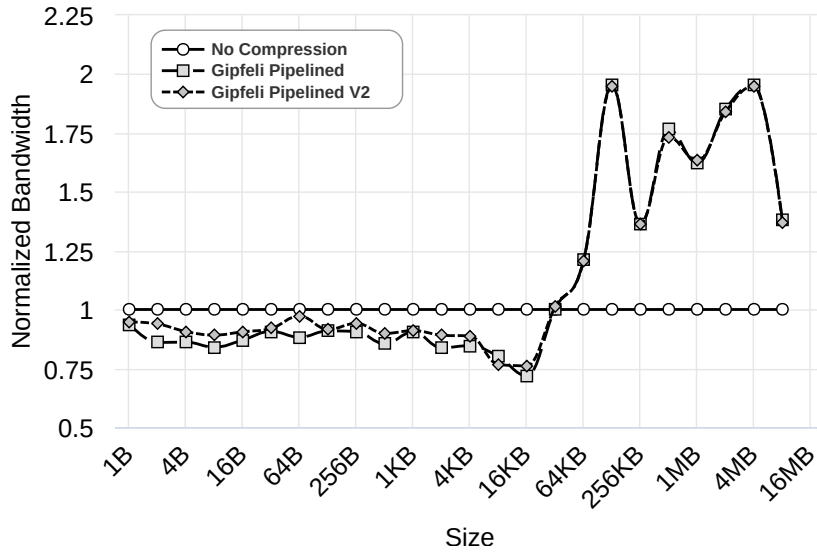


Figure 6.28: Normalized bandwidth obtained over a 100 Mbps network by ‘*Gipfeli Pipelined*’ and ‘*Gipfeli Pipelined V2*’ concerning the ‘No Compression’ one when using traced data. The ‘*Gipfeli Pipelined V2*’ only sends the compressed data if its size is smaller than the original uncompressed data.

Finally, the plot shown in Figure 6.29 presents the results obtained using a 10 Mbps network. The differences between ‘*Gipfeli Pipelined V2*’ and ‘*Gipfeli Pipelined*’ become lower as we reduce the network bandwidth. With data sizes lower than 16B, ‘*Gipfeli Pipelined V2*’ achieves better bandwidth than ‘*Gipfeli Pipelined*’. For values over 16B, there is a lot of variability. In this scenario, ‘*Gipfeli Pipelined V2*’ and ‘*Gipfeli Pipelined*’ outperform ‘No Compression’ for data sizes larger than 4KB. It should be noted that, again, this threshold from which the pipelined versions outperform the approach of not using compression decreases as the network bandwidth decreases. Thus, as expected, a compression system becomes more relevant as the network is slower.

To better understand the previous results, Figure 6.30 shows the percentage of compressed chunks used by ‘*Gipfeli Pipelined V2*’ when using data with different sizes. A 0 value means no compressed data has been sent because it is larger than the original uncompressed data. On the other hand, a 100 value indicates all data sent over the network has been compressed before being transmitted. As we can observe, the system does not compress the data transferred over the network until the data size is equal to or larger than 32B. From that value, the compression is relevant. We can detect lower percentages of compressed chunks for some data sizes, such as 2KB. In general, the number of compressed chunks which are sent increases as the data size increases.

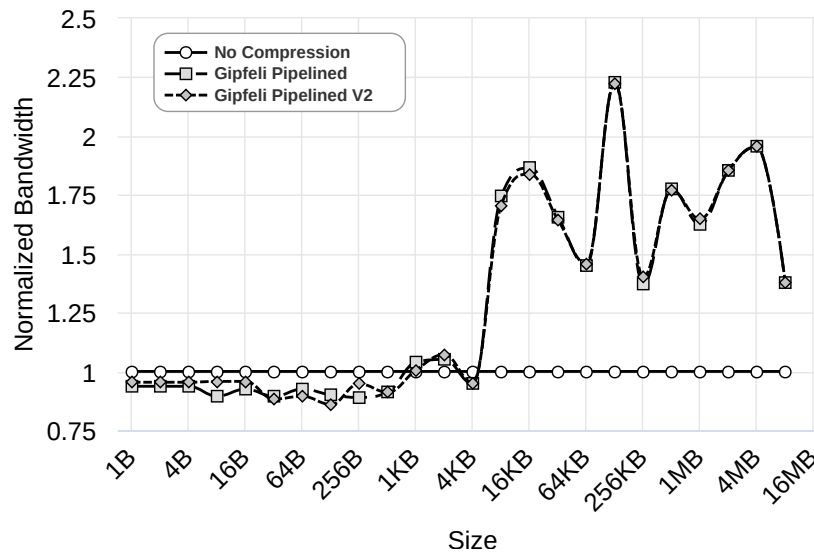


Figure 6.29: Normalized bandwidth obtained over a 10 Mbps network by ‘*Gipfeli Pipelined*’ and ‘*Gipfeli Pipelined V2*’ concerning the ‘No Compression’ one when using traced data. The ‘*Gipfeli Pipelined V2*’ only sends the compressed data if its size is smaller than the original uncompressed data.

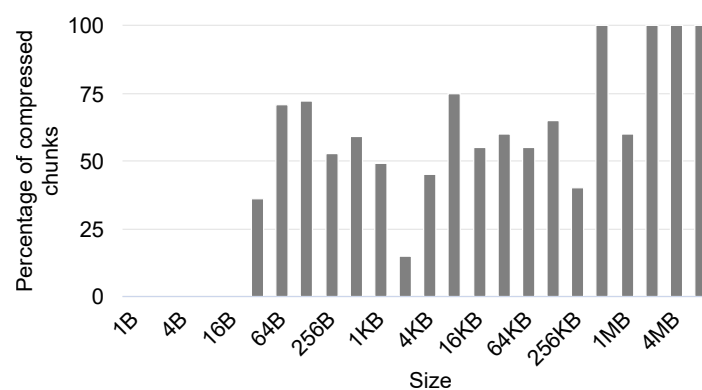


Figure 6.30: Percentage of compressed chunks sent to the network with ‘*Gipfeli Pipelined V2*’ when using data with different sizes of traced data.

6.6 Summary

IoT devices are usually low-performance nodes interconnected by a low-performance network. One possible way to improve the computational power of IoT devices is to process all or part of the computations at the edge of the network. However, edge devices may not have enough computing power to support high-power applications, such as the popular machine learning ones. The use of remote virtual GPUs provided by remote GPU virtualization frameworks can address this problem. Leveraging a virtual GPU physically installed in a remote server can boost these applications. However, this involves sending data through the network and, as mentioned, the network performance can be low in this kind of scenario.

To address the slow-network problem, data sent/received to/from the remote virtual GPU could be compressed. Notice the remote GPU virtualization middleware would carry out this compression, and applications do not need to be modified. Thus, they would not be aware of using a remote GPU or that data are compressed before being transferred.

In this chapter, we have used four popular TF applications to explore the use of a compression system within the communication layer of the remote GPU virtualization system rCUDA. After characterizing the GPU data transfers of these applications, we have analyzed the suitability of existing CPU-based compression libraries for compressing those data transfers on-the-fly. These libraries have been chosen using the CPU-Smash library. The initial implementation compresses the whole data before transmitting them over the network. This first version shows that transfer time is reduced by up to 10 times for some data size ranges, and total execution time is reduced by up to near 14 minutes in the applications analyzed.

Once the first implementation was evaluated, we used a pipelined approach to improve on-the-fly data compression. The rCUDA middleware already uses a pipeline in its communication layer. Thus, we introduced on-the-fly compression within this pipeline. At this point, the GPU-based compression libraries have been discarded because (i) the size of the data used by applications is not big enough, (ii) the GPU is incompatible with the processing pipeline, and (iii) the possible impact on the workload of the GPU. This analysis shows that our pipelined approach hides the overhead of compressing and decompressing the transferred data. However, performance still needs to be further improved. The pipeline mechanism splits data into chunks with a size of 1KB, but more than 90% of data used by TF applications are below this size.

Finally, we have presented the implementation and performance evaluation of a parallel compression pipeline intended to be used within the communication layer of remote GPU

virtualization solutions such as rCUDA. This time, we have decided to use the bandwidth test created by NVIDIA to evaluate the parallel compression pipeline. This test has been updated to work with four kinds of data: lineal, random, sparse, and traced. The last one contains transfers done by TF applications between the host memory and the remote GPU.

The analysis of the parallel compression pipeline raises several interesting conclusions. The first one is that the new compression system can artificially increase the network bandwidth up to 2, 3, and 3.5 times when data is large enough using data traced and 1 Gbps, 100 Mbps, and 10 Mbps networks, respectively. On the contrary, it reduces the network performance with small data sizes. However, we have seen that as networks get slower, the benefits of using compression become more noticeable. Despite improving the results of not using compression, the pipelined approach presents worse results than the non-pipelined approach for slow networks and traced data. This occurs because the transfer time is the bottleneck in this scenario. Thus, compressing the whole data obtains a better compression ratio and, therefore, a better transfer time than pipeline compression.

Chapter 7

Conclusions

This chapter encapsulates the essence of this thesis, highlighting its overall impact on the field of compression in communication systems and providing a roadmap for continued exploration and advancement. In addition to assessing the contributions, this chapter also looks toward the future, outlining potential directions for further exploration and emphasizing the ongoing nature of the academic inquiry. Lastly, this chapter acknowledges the publications that have emerged from the research presented in this thesis. Their inclusion in this chapter serves to recognize the scholarly achievements resulting from this doctoral work and highlights their contribution to the academic community.

7.1 Contributions

This thesis has proposed a new compression mechanism to enhance bandwidth networks through several contributions. The first contribution was the novel AI dataset presented in Chapter 3. Standard datasets are commonly used to evaluate compression systems, but an appropriate dataset was needed to assess our specific scenario. Thus, this dataset has been created by intercepting the data moved to/from the remote GPU when the rCUDA remote GPU virtualization system executes four popular TensorFlow applications (*Alexnet*, *Cifar10*, *Mnist*, and *Inception*). This AI dataset has been utilized in the rest of the chapters to evaluate our proposal.

The second contribution was presented in Chapter 4, where several CPU-based compression libraries were collected to create a new abstraction compression library called *CPU-Smash*. This abstraction library contains a flexible API that enables the adaptation of a compression system in any project with minor changes. In addition, we validated that CPU-based compression libraries have different strategies when used to compress standard and AI datasets. These strategies address specific data types. Thus, creating the AI dataset in the previous Chapter 3 was very useful.

The third contribution corresponds to Chapter 5. Similar to previous Chapter 4, we developed a new abstraction compression library called *GPU-Smash* that includes different GPU-based compression libraries. Existing GPU-based compression libraries only work using device memory (D2D compression). Thus, we also introduced and evaluated different versions to create an efficient H2H compression, which works using host memory. We demonstrated how to obtain the best configuration for specific GPU-based compression libraries, increasing its speed by more than 2.5x. Moreover, a multi-GPU version was presented, achieving an additional speedup of 1.7x.

A heterogeneous CPU-GPU compression library was also introduced in Chapter 5. By analyzing the compression libraries collected in Chapter 4 and Chapter 5, we discovered that some libraries are implemented for the CPU and also for the GPU. Thus, we decided to join them together, creating the *Hybrid-Smash* abstraction compression library. We find out that GPU-based implementations perform better or worse than CPU-based ones depending on the data size and the compression library. Therefore, the *Hybrid-Smash* abstraction compression library transparently uses the CPU or GPU implementation using the information collected in these experiments.

The last contribution was covered in Chapter 6. There, the challenges of computational power in low-performance IoT devices were discussed, and the use of the rCUDA remote GPU virtualization system with a CPU-based compression mechanism in its communication layer was proposed to address these challenges. Our approximation demonstrated that bandwidth could increase from 2 to 3.5 times depending on the network bandwidth. That was possible thanks to our pipelined approach, which hides the latency introduced by the compression stages within the transmission.

7.2 Future directions

Despite the significant progress made in this thesis, several opportunities provide this work for future improvements related to remote GPU virtualization systems:

- When we compare the results obtained with standard and AI datasets, we appreciate that existing compression libraries are focused on standard datasets, and, therefore, they get better results. Creating specific CPU- and GPU-based compression libraries to work with AI datasets could be interesting. These libraries could considerably increase the achieved compression ratio. Moreover, these libraries could also be developed considering the network pipeline.
- The proposed compression network solution only uses CPU-based compression libraries. However, despite deciding to discard GPU-based compression libraries, exploring them more exhaustively to research if they can enhance the transfer time even more would be worthwhile. Remote GPU virtualization systems have a powerful GPU on the server side that can be exploited to enhance even more the transfer time.
- Some compression libraries require more computational power than others, and the compression ratio achieved could also differ. This makes the network, CPU, and GPU utilization vary depending on the library. Sometimes, the network bandwidth, the CPU power, and the GPU power available are limited because they are busy, and changing the compression library to use is preferable. Therefore, an adaptive compression system that changes the compression library to use during the execution depending on the current capabilities of the node could be an interesting area to research.
- In this thesis, we have evaluated how compression libraries affect the application execution time. One crucial characteristic of remote GPU virtualization systems is

reducing power energy by sharing GPUs between several applications. Introducing a compression mechanism in its communication layer can directly affect this power energy consumption. Therefore, it would also be interesting to evaluate how the developed compression system affects the power energy consumed by the client and the server nodes.

- Although this work focuses on bandwidth optimization, transmitting data to remote GPU servers raises significant privacy concerns, especially in sensitive or personal information scenarios. A practical first step to address these concerns is the integration of standard security protocols such as Transport Layer Security (TLS), widely used in web browsers and other secure communication systems. TLS provides a robust and well-established foundation for protecting data in transit without requiring custom implementations. In addition to TLS, future research could explore ad hoc security mechanisms tailored to improve or facilitate compression, aiming to strike a balance between privacy, security, and transmission efficiency. These enhancements would enable the proposed system to be safely deployed in sensitive contexts such as healthcare, smart cities, or industrial monitoring, without significantly compromising performance.

7.3 Publications

In the course of this Ph.D. thesis, the following papers closely linked to the research conducted have been successfully published in various international journals and conferences:

Journals

- 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 and Computation: Practice and Experience* (2023), vol. 35, no 20, p. e7328.
- C. Peñaranda, C. Reaño, and F. Silla. Hybrid-Smash: a heterogeneous CPU-GPU compression library. *IEEE Access* (2024), vol. 12, pp. 32706-32723.
- C. Peñaranda, C. Reaño, and F. Silla. A Parallel Compression Pipeline for Improving GPU Virtualization Data Transfers. *Sensors* (2024), vol. 24, no 14, p. 4649.

Conferences and Workshops

- C. Peñaranda, C. Reaño, and F. Silla. Smash: A Compression Benchmark with AI Datasets from Remote GPU Virtualization Systems. In *Hybrid Artificial Intelligent Systems: 17th International Conference (HAIS)*, pages 236-248. Salamanca, Spain, 2022.
- C. Peñaranda, C. Reaño, and F. Silla. Pipelined Compression in Remote GPU Virtualization Systems using rCUDA: Early Experiences. In *Workshop Proceedings of the 51st International Conference on Parallel Processing (ICPP Workshops '22)*, Article 7, 1–7.

In addition, one related paper has been published in a domestic conference:

- C. Peñaranda, C. Reaño, and F. Silla. Smash: Un Benchmark de Compresión con Datasets de IA Extraídos de Sistemas de Virtualización Remota de GPUs. In *Actas de las VI Jornadas de Computación Empotrada y Reconfigurable (JCER)*, pages 773-782, Alicante, Spain, 2022.

During this thesis, the rCUDA framework has also been exhibited at the Mellanox¹ Technologies booth in the following international industrial/research exhibitions:

- International Conference for High Performance Computing, Networking, Storage and Analysis (Supercomputing, SC), Denver (United States), 2019.
- International Supercomputing Conference (ISC), Frankfurt (United States), 2019.
- International Conference for High Performance Computing, Networking, Storage and Analysis (Supercomputing, SC), Dallas, Texas (United States), 2018.
- International Supercomputing Conference (ISC), Frankfurt (United States), 2018.

Finally, the following book chapter indirectly related to this dissertation was also published:

- F. Silla, C. Peñaranda, J. Prades, C. Reaño, F. E. Serrano, J. Sierra: rCUDA (remote CUDA). In *In Programación de GPUs Usando Compute Unified Device Architecture (CUDA)*. RA-MA, 2020.

¹www.mellanox.com

References

- [1] Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., Kudlur, M., Levenberg, J., Mané, D., Monga, R., Moore, S., Murray, D., Olah, C., Schuster, M., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P., Vanhoucke, V., Vasudevan, V., Viégas, F., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y., and Zheng, X. (2015). TensorFlow: Large-scale machine learning on heterogeneous systems. <https://www.tensorflow.org/>. Software available from tensorflow.org.
- [2] AbuTalib, M., Majzoub, S., Nasir, Q., and Hejji, D. J. (2021). A systematic literature review on hardware implementation of artificial intelligence algorithms. *J. Supercomput.*, 77(2):1897–1938.
- [3] Alakuijala, J., Farruggia, A., Ferragina, P., Kliuchnikov, E., Obryk, R., Szabadka, Z., and Vandevenne, L. (2018). Brotli: A general-purpose data compressor. *ACM Transactions on Information Systems (TOIS)*, 37(1):1–30.
- [4] Apple (Accessed 7th July, 2023). Lzfse Repository. <https://github.com/lzfse/lzfse>.
- [5] Arnold, R. and Bell, T. (1997). A corpus for the evaluation of lossless compression algorithms. In *Proceedings DCC'97. Data Compression Conference*, pages 201–210. IEEE.
- [6] Arnold, R. and Bell, T. (Accessed 7th July, 2023). Canterbury Corpus. <https://corpus.canterbury.ac.nz/>.
- [7] Bell, T., Witten, I. H., and Cleary, J. G. (1989). Modeling for text compression. *ACM Computing Surveys (CSUR)*, 21(4):557–591.
- [8] Bell, T., Witten, I. H., and Cleary, J. G. (Accessed 7th July, 2023). Calgary Corpus. <http://www.data-compression.info/Corpora/CalgaryCorpus/index.html>.
- [9] Biggers, E. (Accessed 7th July, 2023a). Libdeflate Repository. <https://github.com/ebiggers/libdeflate>.
- [10] Biggers, E. (Accessed 7th July, 2023b). Xpack Repository. <https://github.com/ebiggers/xpack>.
- [11] Bonwick, J. (Accessed 7th July, 2023). Lzjb Repository. <https://github.com/nemequ/lzjb>.

- [12] Bush, J. (Accessed 7th July, 2023). Ms Repository. <https://github.com/coderforlife/ms-compress>.
- [13] Calimlim, S. (Accessed 7th July, 2023). Wflz Repository. <https://github.com/ShaneYCG/wflz>.
- [14] Capra, M., Peloso, R., Masera, G., Ruo Roch, M., and Martina, M. (2019). Edge Computing: A Survey on the Hardware Requirements in the Internet of Things World. *Future Internet*, 11(4):100.
- [15] Cecilia, J. M., Morales-García, J., Imbernón, B., Prades, J., Cano, J.-C., and Silla, F. (2023). Using Remote GPU Virtualization Techniques to Enhance Edge Computing Devices. *Future Generation Computer Systems (FGCS)*, 142:14–24.
- [16] Chłopkowski, M. and Walkowiak, R. (2015). A general purpose lossless data compression method for GPU. *Journal of Parallel and Distributed Computing*, 75:40–52.
- [17] Chowdhury, M. R., Tripathi, S., and De, S. (2020). Adaptive Multivariate Data Compression in Smart Metering Internet of Things. *IEEE Transactions on Industrial Informatics*, 17(2):1287–1297.
- [18] Collet, Y. (Accessed 7th July, 2023a). Fse Repository. <https://github.com/Cyan4973/FiniteStateEntropy>.
- [19] Collet, Y. (Accessed 7th July, 2023b). Lz4 Repository. <https://github.com/lz4/lz4>.
- [20] Collette, A. (Accessed 7th July, 2023). Lzfx Repository. <https://code.google.com/archive/p/lzfx/>.
- [21] Collin, L. (Accessed 7th July, 2023). Lzma Repository. <https://git.tukaani.org/?p=lzma.git>.
- [22] Deorowicz, S. (2003). Universal lossless data compression algorithms. *Philosophy Dissertation Thesis, Gliwice*, page 38.
- [23] Deorowicz, S. et al. (Accessed 7th July, 2023). Silesia Corpus. <https://sun.aei.polsl.pl/~sdeor/index.php?page=silesia>.
- [24] Duranton, M., De Bosschere, K., Gamrat, C., Maebe, J., Munk, H., and Zendra, O. (2017). The HiPEAC Vision 2017.
- [25] Engelhart, J. (Accessed 7th July, 2023). Pithy Repository. <https://github.com/johnejang/pithy>.
- [26] Foundation, F. S. (Accessed 7th July, 2023). Lzmat Repository. https://www.autohotkey.com/mpress/lzmat_web.htm.
- [27] Fu, S. (Accessed 7th July, 2023). Csc Repository. <https://github.com/fusiyuan2010/CSC>.
- [28] Geelnard, M. (Accessed 7th July, 2023). Liblzg Repository. <https://github.com/mbitsnbites/liblzg>.

- [29] Geldreich, R. (Accessed 7th July, 2023a). Lzham Repository. https://github.com/richgel999/lzham_codec_devel.
- [30] Geldreich, R. (Accessed 7th July, 2023b). Miniz Repository. <https://code.google.com/archive/p/miniz/>.
- [31] Giunta, G., Montella, R., Agrillo, G., and Coviello, G. (2010). A GPGPU Transparent Virtualization Component for High Performance Computing Clouds. Euro-Par 2010.
- [32] Google (Accessed 7th July, 2023a). Brotli Repository. <https://github.com/google/brotli>.
- [33] Google (Accessed 7th July, 2023b). Gifpeli Repository. <https://github.com/google/gifpeli>.
- [34] Google (Accessed 7th July, 2023c). Snappy Corpus. <https://github.com/google/snappy/tree/main/testdata>.
- [35] Google (Accessed 7th July, 2023d). Snappy Repository. <https://github.com/google/snappy>.
- [36] Grebnov, I. (Accessed 7th July, 2023). Libbsc Repository. <https://github.com/IlyaGrebnov/libbsc>.
- [37] Gubbi, J., Buyya, R., Marusic, S., and Palaniswami, M. (2013). Internet of Things (IoT): A Vision, Architectural Elements, and Future Directions. *Future Generation Computer Systems (FGCS)*, 29(7):1645–1660.
- [38] Guillaume (Accessed 7th July, 2023). Density Repository. <https://github.com/k0dai/density>.
- [39] Gupta, A., Bansal, A., and Khanduja, V. (2017). Modern lossless compression techniques: Review, comparison and analysis. In *2017 Second International Conference on Electrical, Computer and Communication Technologies (ICECCT)*, pages 1–8. IEEE.
- [40] Gupta, V., Gavrilovska, A., Schwan, K., Kharche, H., Tolia, N., Talwar, V., and Ranganathan, P. (2009). GVIM: GPU-accelerated virtual machines. In *Proceedings of the 3rd ACM Workshop on System-level virtualization for High Performance Computing*, pages 17–24.
- [41] Hansson, E. and Karlsson, S. (2013). Lossless Message Compression.
- [42] Hidayat, A. (Accessed 7th July, 2023). Flz Repository. <https://github.com/svn2github/fastlz>.
- [43] Hu, N. (2001). Network Aware Data Transmission with Compression. *Selected Papers from the Proceedings of the Fourth Student Symposium on Computer Systems (SOCS-4)*, page 33.
- [44] Hussain, M., Bird, J. J., and Faria, D. R. (2018). A study on cnn transfer learning for image classification. pages 191–202. UK Workshop on computational Intelligence.
- [45] Ibsen, J. (Accessed 7th July, 2023). Brieflz Repository. <https://github.com/jibsen/brieflz>.

- [46] Imbernón, B., Prades, J., Gimenez, D., Cecilia, J. M., and Silla, F. (2018). Enhancing Large-Scale Docking Simulation on Heterogeneous Systems: An MPI vs rCUDA Study. *Future Generation Computer Systems (FGCS)*, 79:26–37.
- [47] Iserte, S., Prades, J., Reaño, C., and Silla, F. (2016). Increasing the performance of data centers by combining remote GPU virtualization with Slurm. pages 98–101. 2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid).
- [48] Jean-loup Gailly, M. A. (Accessed 7th July, 2023). Zlib Repository. <http://zlib.net/>.
- [49] Jorge, J., Giménez, A., Iranzo-Sánchez, J., Civera, J., Sanchis, A., and Juan, A. (2019). Real-Time One-Pass Decoder for Speech Recognition Using LSTM Language Models. pages 3820–3824. INTERSPEECH.
- [50] Kalsoom, T., Ahmed, S., Rafi-ul Shan, P. M., Azmat, M., Akhtar, P., Pervez, Z., Imran, M. A., and Ur-Rehman, M. (2021). Impact of iot on manufacturing industry 4.0: A new triangular systematic review. *Sustainability*, 13(22):12506.
- [51] Kennedy, J., Varghese, B., and Reaño, C. (2021). AVEC: Accelerator Virtualization in Cloud-Edge Computing for Deep Learning Libraries. pages 37–44. 2021 IEEE 5th International Conference on Fog and Edge Computing (ICFEC).
- [52] Kim, Y., Choi, S., Lee, D., Jeong, J., Kwak, J., Lee, J., Lee, G., Lee, S., Park, K., Jeong, J., et al. (2020). Low-Overhead Compressibility Prediction for High-Performance Lossless Data Compression. *IEEE Access*, 8:37105–37123.
- [53] Krintz, C. and Sucu, S. (2005). Adaptive On-the-Fly Compression. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 17(1):15–24.
- [54] Krizhevsky, A., Hinton, G., et al. (2009). Learning multiple layers of features from tiny images.
- [55] Krizhevsky, A., Sutskever, I., and Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems*, 25:1097–1105.
- [56] Kuznetsov, A. N. (Accessed 7th July, 2023). TC-TBF website. <https://man7.org/linux/man-pages/man8/tc-tbf.8.html>.
- [57] LeCun, Y., Cortes, C., and Burges, C. (2010). MNIST handwritten digit database. *ATT Labs [Online]*. Available: <http://yann.lecun.com/exdb/mnist>, 2.
- [58] Lehmann, M. A. (Accessed 7th July, 2023). Lzf Repository. <https://github.com/nemequ/liblzf>.
- [59] Li, J., Wu, J., and Jeon, G. (2019). GPU acceleration of clustered DPCM for lossless compression of hyperspectral images. *IEEE Transactions on Industrial Informatics*, 16(5):2906–2916.
- [60] Li, Z. (Accessed 7th July, 2023). Zling Repository. <https://github.com/richox/libzling>.

- [61] Liang, T.-Y. and Chang, Y.-W. (2011). GridCuda: a grid-enabled CUDA programming toolkit. In *2011 IEEE Workshops of International Conference on Advanced Information Networking and Applications*, pages 141–146. IEEE.
- [62] Liang, X., Di, S., Tao, D., Li, S., Li, S., Guo, H., Chen, Z., and Cappello, F. (2018). Error-controlled lossy compression optimized for high compression ratios of scientific datasets. In *2018 IEEE International Conference on Big Data (Big Data)*, pages 438–447. IEEE.
- [63] Liang, Y. and Li, Y. (2014). An Efficient and Robust Data Compression Algorithm in Wireless Sensor Networks. *IEEE Communications Letters*, 18(3):439–442.
- [64] Liebl, G., Schierl, T., Wiegand, T., and Stockhammer, T. (2006). Advancedwireless multiuser video streaming using the scalable video coding extensions of h. 264/mpeg4-avc. In *2006 IEEE International Conference on Multimedia and Expo*, pages 625–628. IEEE.
- [65] Mahoney, M. (Accessed 7th July, 2023). Zpaq Repository. <http://mattmahoney.net/dc/zpaq.html>.
- [66] Markus (Accessed 7th July, 2023a). Lzo Repository. <http://www.oberhumer.com/opensource/lzo/>.
- [67] Markus (Accessed 7th July, 2023b). Ucl Repository. <http://www.oberhumer.com/opensource/ucl/>.
- [68] Marpe, D., Wiegand, T., and Sullivan, G. J. (2006). The H. 264/MPEG4 advanced video coding standard and its applications. *IEEE communications magazine*, 44(8):134–143.
- [69] McCarthy, C. (Accessed 7th July, 2023). Flzma2 Repository. <https://github.com/conor42/fast-lzma2>.
- [70] Merkle, P., Muller, K., Smolic, A., and Wiegand, T. (2006). Efficient compression of multi-view video exploiting inter-view dependencies based on H. 264/MPEG4-AVC. In *2006 IEEE International Conference on Multimedia and Expo*, pages 1717–1720. IEEE.
- [71] Meta (Accessed 7th July, 2023). Zstd Repository. <https://facebook.github.io/zstd/>.
- [72] Miori, L., Sanin, J., and Helmer, S. (2017). A platform for edge computing based on Raspberry Pi clusters. In *British International Conference on Databases*, pages 153–159. Springer.
- [73] Miyato, T., Dai, A. M., and Goodfellow, I. (2016). Adversarial training methods for semi-supervised text classification. *arXiv preprint arXiv:1605.07725*.
- [74] Moffat, A. (2019). Huffman coding. *ACM Computing Surveys (CSUR)*, 52(4):1–35.
- [75] Morotti (Accessed 7th July, 2023). Lzbench Compression Benchmark. <https://morotti.github.io/lzbench-web/>.
- [76] Naresh, M. and Munaswamy, P. (2019). Smart agriculture system using iot technology. *International journal of recent technology and engineering*, 7(5):98–102.

- [77] Nemerson, E. (Accessed 7th July, 2023). Squash Compression Benchmark. <https://quixdb.github.io/squash-benchmark/>.
- [78] Nevill-Manning, C. G. and Witten, I. H. (1999). Protein is incompressible. In *Proceedings DCC'99 Data Compression Conference (Cat. No. PR00096)*, pages 257–266. IEEE.
- [79] Nevill-Manning, C. G. and Witten, I. H. (Accessed 7th July, 2023). Protein Corpus. <http://www.data-compression.info/Corpora/ProteinCorpus/index.html>.
- [80] Nvidia (2020). Multi-Instance GPU and Virtual Compute Server. Technical brief.
- [81] Nvidia (Accessed 15th May, 2023b). Nsight Systems 2023.2. <https://developer.nvidia.com/nsight-systems>.
- [82] Nvidia (Accessed 7th July, 2023). CUDA (Compute Unified Device Architecture). <https://developer.nvidia.com/cuda-toolkit>.
- [83] Nvidia (Accessed 7th July, 2023a). Multi-Instance GPU User Guide. <https://docs.nvidia.com/datacenter/tesla/mig-user-guide/index.html>.
- [84] Nvidia (Accessed 7th July, 2023). The nvCOMP library. <https://developer.nvidia.com/nvcomp>.
- [85] Nvidia (Accessed 7th July, 2023). Virtual GPU Software User Guide. <https://docs.nvidia.com/grid/latest/grid-vgpu-user-guide/index.html>.
- [86] Oikawa, M., Kawai, A., Nomura, K., Yasuoka, K., Yoshikawa, K., and Narumi, T. (2012). DS-CUDA: a middleware to use many GPUs in the cloud environment. In *2012 SC Companion: High Performance Computing, Networking Storage and Analysis*, pages 1207–1214. IEEE.
- [87] Ozsoy, A. and Swamy, M. (2011). CULZSS: LZSS lossless data compression on CUDA. In *2011 IEEE International Conference on Cluster Computing*, pages 403–411. IEEE.
- [88] Papadokostaki, K., Mastorakis, G., Panagiotakis, S., Mavromoustakis, C. X., Dobre, C., and Batalla, J. M. (2017). Handling Big Data in the Era of Internet of Things (IoT). In *Advances in Mobile Cloud Computing and Big Data in the 5G Era*, pages 3–22. Springer.
- [89] Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., and Chintala, S. (2019). PyTorch: An Imperative Style, High-Performance Deep Learning Library. In Wallach, H., Larochelle, H., Beygelzimer, A., d'Alché-Buc, F., Fox, E., and Garnett, R., editors, *Advances in Neural Information Processing Systems* 32, pages 8024–8035. Curran Associates, Inc.
- [90] Patel, R. A., Zhang, Y., Mak, J., Davidson, A., and Owens, J. D. (2012). *Parallel lossless data compression on the GPU*. IEEE.
- [91] Peterson, P. A. and Reiher, P. L. (2016). Datacomp: Locally Independent Adaptive Compression for Real-World Systems. In *2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS)*, pages 211–220. IEEE.

- [92] Prades, J. and Silla, F. (2019). GPU-Job Migration: The rCUDA Case. *IEEE Transactions on Parallel and Distributed Systems*, 30(12):2718–2729.
- [93] Promberger, L., Schwemmer, R., and Fröning, H. (2020). Assessing the Overhead of Offloading Compression Tasks. pages 1–10. 49th International Conference on Parallel Processing-ICPP: Workshops.
- [94] Reinhold, L. M. (Accessed 7th July, 2023). Quicklz Repository. <https://github.com/RT-Thread-packages/quicklz>.
- [95] Ren, J., Pan, Y., Goscinski, A., and Beyah, R. A. (2018). Edge Computing for the Internet of Things. *IEEE Network*, 32(1):6–7.
- [96] Rosbach, H. K. (Accessed 7th July, 2023). Zlib-ng Repository. <https://github.com/Dead2/zlib-ng>.
- [97] Routray, S. K., Javali, A., Sharmila, K., Semunigus, W., Pappa, M., and Ghosh, A. D. (2020). Lossless Compression Techniques for Low Bandwidth Networks. In *2020 3rd International Conference on Intelligent Sustainable Systems (ICISS)*, pages 823–828. IEEE.
- [98] Salman, O., Elhajj, I., Kayssi, A., and Chehab, A. (2015). Edge computing enabling the Internet of Things. pages 603–608. 2015 IEEE 2nd World Forum on Internet of Things (WF-IoT).
- [99] Satyanarayanan, M. (2017). The Emergence of Edge Computing. *Computer*, 50(1):30–39.
- [100] Schirmer, O. (Accessed 7th July, 2023). Z3lib Repository. <https://scara.com/~schirmer/o/z3lib/>.
- [101] Selvaraj, S. and Sundaravaradhan, S. (2020). Challenges and opportunities in iot healthcare systems: a systematic review. *SN Applied Sciences*, 2(1):139.
- [102] Sforzin, A., Mármol, F. G., Conti, M., and Bohli, J.-M. (2016). Rpiids: Raspberry pi ids—a fruitful intrusion detection system for iot. In *2016 Intl IEEE Conferences on Ubiquitous Intelligence & Computing, Advanced and Trusted Computing, Scalable Computing and Communications, Cloud and Big Data Computing, Internet of People, and Smart World Congress (UIC/ATC/ScalCom/CBDCom/IoP/SmartWorld)*, pages 440–448. IEEE.
- [103] Shi, L., Chen, H., Sun, J., and Li, K. (2011). vCUDA: GPU-accelerated high-performance computing in virtual machines. *IEEE Transactions on Computers*, 61(6):804–816.
- [104] Silla, F., Iserte, S., Reaño, C., and Prades, J. (2017). On the benefits of the remote GPU virtualization mechanism: The rCUDA case. *Concurrency and Computation: Practice and Experience*, 29(13):e4072.
- [105] Sittón-Candanedo, I., Alonso, R. S., García, Ó., Muñoz, L., and Rodríguez-González, S. (2019). Edge computing, iot and social computing in smart energy scenarios. *Sensors*, 19(15):3353.

- [106] Skibinski, P. (Accessed 7th July, 2023). Lizard Repository. <https://github.com/inikep/lizard>.
- [107] Snyder, M. (Accessed 7th July, 2023). Bzip2 Repository. <https://gitlab.com/bzip2/bzip2>.
- [108] Stokes, C. (Accessed 7th July, 2023). Lzsse Repository. <https://github.com/ConorStokes/LZSSE>.
- [109] Szegedy, C., Vanhoucke, V., Ioffe, S., Shlens, J., and Wojna, Z. (2016). Rethinking the inception architecture for computer vision. pages 2818–2826. IEEE conference on computer vision and pattern recognition.
- [110] Team, T. B. D. (Accessed 7th July, 2023). C-blosc2 Repository. <https://github.com/Blosc/c-blosc2>.
- [111] TensorFlow (Accessed 7th July, 2023). TensorFlow Lite. <https://www.tensorflow.org/lite/>.
- [112] TensorFlow (Accessed 7th July, 2023). Flowers. http://download.tensorflow.org/example_images/flower_photos.tgz.
- [113] Tkatchev, I. (Accessed 7th July, 2023). Yalzh77 Repository. <https://bitbucket.org/tkatchev/yalzh77/src/master/>.
- [114] Uthayakumar, J., Elhoseny, M., and Shankar, K. (2020). Highly Reliable and Low-Complexity Image Compression Scheme Using Neighborhood Correlation Sequence Algorithm in WSN. *IEEE Transactions on Reliability*, 69(4):1398–1423.
- [115] Vandevenne, L. (Accessed 7th July, 2023). LodePNG Repository. <https://github.com/lvandeve/lodepng>.
- [116] Vega, J., Ruiz, M., Sánchez, E., Pereira, A., Portas, A., and Barrera, E. (2007). Real-Time Lossless Data Compression Techniques for Long-Pulse Operation. *Fusion Engineering and Design*, 82(5-14):1301–1307.
- [117] Venu, D., Mayuri, A., Neelakandan, S., Murthy, G., Arulkumar, N., and Shelke, N. (2022). An efficient low complexity compression based optimal homomorphic encryption for secure fiber optic communication. *Optik*, 252:168545.
- [118] Vokes, S. (Accessed 7th July, 2023). Heatshrink Repository. <https://github.com/atomicobject/heatshrink>.
- [119] Welch, T. A. (1984). A technique for high-performance data compression. *Computer*, 17(06):8–19.
- [120] Welton, B., Kimpe, D., Cope, J., Patrick, C. M., Iskra, K., and Ross, R. (2011). Improving i/o forwarding throughput with data compression. pages 438–445. 2011 IEEE International Conference on Cluster Computing.
- [121] Wikipedia (Accessed 7th July, 2023). English Wikipedia Corpus. <http://mattmahoney.net/dc/textdata.html>.

-
- [122] Wiseman, Y. (2016). Unlimited and Protected Memory for Flight Data Recorders. *Aircraft Engineering and Aerospace Technology (AEAT)*.
 - [123] Witten, I. H., Neal, R. M., and Cleary, J. G. (1987). Arithmetic coding for data compression. *Communications of the ACM*, 30(6):520–540.
 - [124] Ziv, J. and Lempel, A. (1977). A universal algorithm for sequential data compression. *IEEE Transactions on information theory*, 23(3):337–343.
 - [125] Ziv, J. and Lempel, A. (1978). Compression of individual sequences via variable-rate coding. *IEEE transactions on Information Theory*, 24(5):530–536.

