Received: 30 November 2021 | Revised: 20 May 2022 | Accepted: 9 July 2022
DOI: 10.1002/cpe.7328

SPECIAL ISSUE PAPER

WILEY

Exploring the use of data compression for accelerating machine learning in the edge with remote virtual graphics processing units

Cristian Peñaranda[1] | Carlos Reaño[2] | Federico Silla[1]

[1]Departamento de Informática de Sistemas y Computadores, Universitat Politècnica de València, Valencia, Spain

[2]Departament d'Informática, Escola Tècnica Superior d'Enginyeria (ETSE-UV), Universitat de València, Valencia, Spain

Correspondence
Cristian Peñaranda, Departamento de Informática de Sistemas y Computadores, Universitat Politècnica de València, Camino de Vera, s/n., Valencia 46022, Spain.
Email: cripeace@gap.upv.es

Funding information
European Union's Horizon 2020 Research and Innovation Programme, Grant/Award Numbers: 101016577, 101017861

Summary

Internet of Things (IoT) devices are usually low performance nodes connected by low bandwidth networks. To improve performance in such scenarios, some computations could be done at the edge of the network. However, edge devices may not have enough computing power to accelerate applications such as the popular machine learning ones. Using remote virtual graphics processing units (GPUs) can address this concern by accelerating applications leveraging a GPU installed in a remote device. However, this requires exchanging data with the remote GPU across the slow network. To address the problem with the slow network, the data to be exchanged with the remote GPU could be compressed. In this article, we explore the suitability of using data compression in the context of remote GPU virtualization frameworks in edge scenarios executing machine learning applications. We use popular machine learning applications to carry out such exploration. After characterizing the GPU data transfers of these applications, we analyze the usage of existing compression libraries for compressing those data transfers to/from the remote GPU. Our exploration shows that transferring compressed data becomes more beneficial as networks get slower, reducing transfer time by up to 10 times. Our analysis also reveals that efficient integration of compression into remote GPU virtualization frameworks is strongly required.

KEYWORDS
data compression, edge computing, GPU virtualization, machine learning

1 | INTRODUCTION

In the Internet of Things (IoT) area, there is a huge amount of devices producing large quantities of data that require to be processed. IoT devices typically present low performance and consume little energy. As a result, all the data produced by them are usually sent across Internet to a cluster or to a high performance computer located in a remote data center to process it.

Edge computing has been an important advance for IoT[1-3] because data are processed closer to their source instead of traversing Internet until reaching the aforementioned data center. Furthermore, as network performance can be a limitation in IoT systems using low bandwidth networks such as WiFi or cellphone networks, edge computing can alleviate this problem by processing all or part of the data at the edge of the network.

However, edge devices may not have enough computing power to support any kind of applications. One example is machine learning applications, such as TensorFlow[4] or PyTorch,[5] which are currently boosted by using graphics processing units (GPUs) and the CUDA programming technology created by NVIDIA.[6] So the question is how to provide powerful GPUs to an edge device. This can be achieved by making use of the remote GPU virtualization mechanism,[7,8] which allows to share a GPU located in a computer among processes being executed in other computers. By using this mechanism, a CUDA application being executed in an edge device without a GPU is provided a GPU (or a part of it) located in another computer. The application is not aware of using a remote GPU because the CUDA calls performed by the application are transparently forwarded, across the network, to the remote GPU. The use of remote GPU virtualization approaches have proved to be an important advance for edge computing.[9]

The performance of applications using the remote GPU virtualization mechanism greatly depends on the throughput of the network connecting the device where the application is being executed and the remote server with the GPU. However, as mentioned before, network performance can be low in the edge computing scenario, thus reducing the benefits of using a remote GPU.

In order to reduce the overhead of using a remote GPU across a low-performance network, data exchanged with the remote GPU can be compressed on-the-fly before being transmitted to/from the remote GPU. This compression would artificially increase the available bandwidth. On-the-fly compression means that data are compressed on the sending device before being injected into the network. Data is later decompressed once it is received at other side. This compression/decompression process happens in both directions: from the edge device to the remote server with the GPU and also in the opposite direction. This compression/decompression process takes place transparently within the communication layer of the remote GPU virtualization middleware and, therefore, CUDA applications are not aware of it. As far as we know, currently there is no remote GPU virtualization framework making use of on-the-fly compression.

It is important to remark that what the compressed are data moved to/from the remote GPU. For instance, every time the CUDA application performs a call to a cudaMemcpy function, some data are moved across the network to/from the remote GPU. Notice that these data moved to/from the GPU are not necessarily the same data used as input to the application. For example, the input data to the Cifar10[10] TensorFlow sample used later in this article consist of 60,000 32×32 color images classified in 10 classes. Those images are stored in the disk and read during the execution of the Cifar10 sample. Every time one of these images is read from disk, it is appropriately transformed and the result of such transformation is inserted into the convolutional neural network (CNN) used within the Cifar10 sample. During the insertion into the CNN, several copies will be made to the GPU and data will be later copied back to main memory after processing. Data moved in those copies to the GPU memory and back to main memory are what will be compressed in a remote GPU virtualization middleware with on-the-fly compression. Notice that users do not have access to that data moved to/from the GPU because that data are part of the internals of the execution of machine learning applications. Notice also that data in those copies are only available during part of the execution of the machine learning application. Before the program begins its execution, that data do not exist. In a similar way, once the machine learning application completes its execution, that data are not available anymore.

When on-the-fly compression is leveraged by a remote GPU virtualization middleware, the compression/decompression process raises up several concerns, summarized in the question whether it is worth the effort (in terms of processing time and energy) of carrying out the compression/decompression stages. Basically, we are artificially increasing network performance by compressing the data. However, the time required by the compression/decompression process could cancel out the benefits of dealing with a smaller amount of data across the network. The reason is that overall transmission time (which now includes compression/decompression times) could be increased.

Several aspects can influence the overall benefits of using on-the-fly compression within the communication layer of a remote GPU virtualization middleware. For instance, the size of the data to be sent is a potentially important parameter, given that compressing small messages could not be as worth as compressing larger ones. Also, the exact contents of the data exchanged with the remote GPU will likely matter. The exact hardware used to compress/decompress the data could also make a difference. Additionally, slower networks could benefit from more sophisticated (and more compute-intensive) compression algorithms, achieving larger compression ratios. On the contrary, faster networks could require lighter compression algorithms with smaller compression ratios to benefit from on-the-fly compression.

In this article, we carry out an initial exploration of the suitability of using data compression in the context of remote GPU virtualization frameworks in edge scenarios using machine learning applications. As far as we know, this analysis has never been done before. Basically, we study if it is worth compressing on-the-fly the data to be exchanged with the remote GPU.

The main research contributions of this article are:

- A characterization of the internal CPU-GPU memory data copies in machine learning applications.
- An analysis of the suitability of existing compression libraries for compressing those memory data copies.
- An evaluation of the impact on performance of on-the-fly data compression in edge computing scenarios using remote GPU virtualization frameworks.
- An estimation of how an adaptive compression mechanism improves overall application performance. This adaptive mechanism is based on transmitting the compressed data (using the most beneficial algorithm) or the original one depending on the compression ratio achieved, the size of the data to be transferred and the speed of the network.

The rest of the article is organized as follows. First, Section 2 presents some background related to remote GPU virtualization, machine learning and compression libraries. Next, Section 3 describes in more detail our proposal, including a deeper description of how on-the-fly compression within remote GPU virtualization frameworks works. Then, we present our exploration of using on-the-fly compression in remote GPU virtualization frameworks and also analyze the experimental results in Section 4. Finally, Section 5 concludes the article and discusses future work.

2 | BACKGROUND

In this section, we present some background on remote GPU virtualization (Section 2.1), machine learning applications (Section 2.2) and compression libraries (Section 2.3) so that the rest of the article is better understood.

2.1 | Remote GPU virtualization

In the context of the CUDA technology developed by NVIDIA,[6] remote GPU virtualization allows a CUDA application being executed in a computer which does not have a GPU to access a GPU installed in another computer. Figure 1 depicts the architecture usually deployed by remote GPU virtualization solutions, which follow a distributed client-server approach. The client part of the middleware is a library providing the same API as CUDA. It is installed in the computer executing the application requesting GPU services. The server side of the middleware is a daemon that runs in the computer owning the actual GPU. Both client and server make use of the network to communicate.

The architecture depicted in Figure 1 is used in the following way: every time an application executes a call to a CUDA function, the client middleware intercepts that request, appropriately processes it, and forwards it to the server middleware in the remote computer. In the server side, the daemon receives the request, interprets and forwards it to the GPU, which executes the request and returns the results to the server middleware. Finally, the server side sends back the results to the client side, which forwards them to the application. At that point, the initial call to the CUDA function is completed. Notice that GPU virtualization solutions provide GPU services in a transparent way and, therefore, applications are not aware that their requests are actually serviced by a remote virtual GPU instead of a real and local GPU. In this way, the source code of applications using remote GPU virtualization solutions does not need to be modified.

Currently, the most prominent remote GPU virtualization solutions supporting CUDA are rCUDA[7] and GVirtuS.[8] Both of them present the same architecture, although rCUDA supports a wider set of CUDA functions. For this reason, in this work we make use of the rCUDA middleware.

2.2 | Machine learning applications

TensorFlow(TF)[4] is an open source library for machine learning developed by Google. It allows users to build and train machine learning models. Users can run TF over multiple CPUs as well as reducing execution time by offloading to GPUs most of the computations. Several areas are currently covered by TF, such as speech recognition,[11] image classification,[12] or text classification,[13] among others. In this work, we carry out our exploration by using the following four different TF samples where the training and the inference of different 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.

- Alexnet: uses the Alexnet[14] CNN model to evaluate the inference time of that model using images.
- Cifar10: a simple CNN is used to evaluate the image classification of the Cifar10[10] dataset. This dataset is a collection of 60,000 colored images with 32 × 32 pixels each one. 50,000 images are used to train the convolutional network whereas the other 10,000 images are used to evaluate the classification. These images are classified into 10 different classes: airplanes, automobiles, birds, cats, deer, dogs, frogs, horses, ships, and trucks.

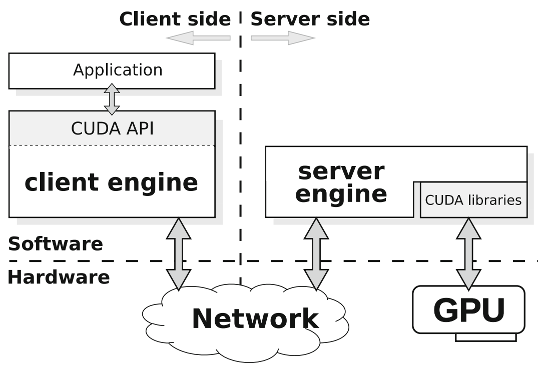

FIGURE 1 General architecture of remote GPU virtualization solutions.

- Mnist: a simple, end-to-end, LeNet-5-like CNN is used to classify the Mnist[15] dataset. The dataset has 70,000 black and white images, where 60,000 images are used to train the network and the other 10,000 images are used to check the quality of such training. These images represent handwritten digits in 28×28 pixels.
- Inception: a CNN model widely used in image analysis and object detection called Inception-V3[16] is leveraged by this sample. Using this model, the classification of a flowers dataset[17] is evaluated. This dataset was created by the TF team and has 3670 colored images with 180×180 pixels each one, where 2934 images are used to train the model and the other 736 images are used to test the training. These images are classified into 5 different classes: daisies, dandelions, roses, sunflowers, and tulips.

2.3 Compression libraries

There are different compression libraries, each one uses different compression algorithms. In this article we make use of some of the most popular ones. The selected libraries allow to compress data allocated in RAM. Almost all of them provide different compression levels to make trade-offs between compression ratio and computation time. The compression libraries used in this work are the following ones:

- BZIP2.[18] This library combines different algorithms to obtain a high compression ratio. First, it runs the Burrows-Wheeler block-shorting algorithm to order a specific block. After that, it performs a Huffman coding. The library has nine different compression levels to indicate the block size to use.
- LZ4.[19] This library uses an algorithm based on LZ77. It is a library focused on fast compression and decompression. It has 12 different compression levels.
- ZLIB.[20] This library uses the deflate method of compression: a combination of LZ77 and Huffman coding. We can chose between nine different compression levels.
- Zstandard (ZSTD).[21] This library is based on LZ77 with a combination of a fast finite state entropy and Huffman coding. This solution, created by Facebook, has 22 different compression levels.
- SNAPPY.[22] This library, created by Google, is based on the LZ77 algorithm. It is oriented to obtain a short computation time instead of a higher compression ratio. The library does not provide any parameter to select the compression level.
- GIPFELI.[23] Similarly to SNAPPY, this library has been developed by Google and is based on the LZ77 algorithm. The library targets slightly higher compression ratios than other compression libraries. GIPFELI does not allow any configuration.
- LZO.[24] This compression library is a LZ77 derivative. LZO provides eight different modes to set it up. Moreover, it has different compression levels depending on the mode used.
- FastLZ (FLZ).[25] An implementation of the LZ77 algorithm for lossless data compression. The library allows us to use two different compression levels to get faster compression or better compression ratio.

The above libraries present some limitations that are relevant to our work. BZIP2, LZO, and FLZ libraries are not able to calculate in advance the size of the new compressed data. Therefore, it is necessary to reserve a memory buffer large enough to compress the information. Moreover, BZIP2, LZ4, ZLIB, GIPFELI, LZO, and FLZ cannot calculate the original data size (non-compressed data) from the compressed data, thus making necessary to know the original data size before decompressing.

There are other software compression solutions available, such as GZIP, PIGZ, PBZIP2, or LBZIP2. The main concern with these ones is that they compress files. On the contrary, the approach proposed in this article is to compress data allocated in RAM memory before sending it to the network. Copying the data to be compressed into a file is not feasible in our edge computing scenario because that would introduce a large overhead.

Finally, an important concern is why we have selected for this study the eight compression libraries mentioned above. Among the many different compression libraries available in the literature, we found the `Squash` compression benchmark[*] which includes 33 different compression libraries. Unfortunately, this benchmark is not maintained since 2018, being its last release in 2015. Also, we have not been able to make it work. For this reason, we decided to create a new compression benchmark named `Smash`, which is available at https://github.com/cpenaranda/smash. This new benchmark includes 39 compression libraries (the same as Squash plus 6 additional ones). Using this new benchmark, we have analyzed in this article all the 39 compression libraries included in Smash and later we have selected a set of 8 compression libraries to generate clearer figures (figures with 39 lines were not easy to interpret). To reduce the results in Section 4 from 39 compression libraries down to 8 libraries, we have followed two different criteria: (1) on the one hand, we have selected the compression libraries providing the best performance and (2) on the other hand, we have included in the selection compression libraries that presented interesting results for our analysis. We will go into more detail in Section 4.

3 | USING DATA COMPRESSION WITHIN REMOTE GPU VIRTUALIZATION SOLUTIONS

Applying the remote GPU virtualization mechanism to an edge device is straightforward. As shown in Figure 2, the application runs in the edge device and GPU computations are offloaded to the remote GPU server. In the remote GPU server, CUDA is used in the same way as it would be used to offload computations to the local GPU in a normal GPU scenario. It is important to remark that the application is not modified to use the remote GPU. Actually, it is not aware of using a remote GPU. The remote GPU virtualization middleware makes that this offloading process to the remote GPU happen transparently.

When a remote GPU virtualization solution is used, the performance of the network connecting the node executing the CUDA application and the remote GPU server plays an important role to achieve good performance. However, in edge computing environments, networks usually present low bandwidth. As shown in Figure 2, the approach proposed in this article to address this problem is compressing the data exchanged with the remote GPU. In this regard, as depicted in Figure 3, whenever the application performs a call to the CUDA library, the remote GPU virtualization middleware intercepts that call to forward it to the remote GPU server, as explained in Section 2.1. However, before the middleware injects the data associated to the CUDA call into the network, such data is compressed to reduce its size. After compression, the new contents are sent to the network. Once they are received at the server side, data are decompressed to retrieve the original data managed by the CUDA application. Data are also copied in the opposite direction, from the remote GPU server to the edge device following a similar approach. Previous studies on compression have shown that data size can be significantly reduced.[26,27] Thus, reducing the size of transmitted data could be a good approach to improve the results obtained with IoT low performance networks.

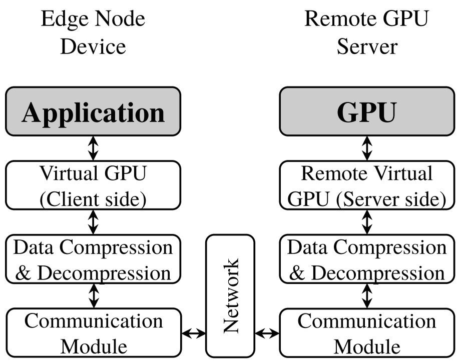
Edge Node Device
Remote GPU Server
Application
GPU
Virtual GPU (Client side)
Remote Virtual GPU (Server side)
Data Compression & Decompression
Data Compression & Decompression
Communication Module
Network
Communication Module

FIGURE 2 Architecture of remote GPU virtualization solutions when on-the-fly data compression is integrated into them

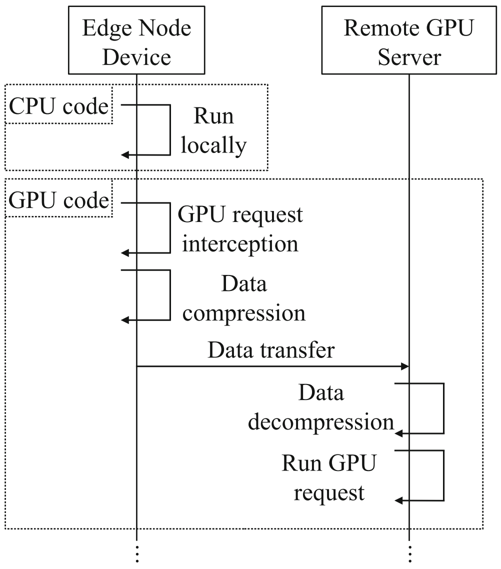

FIGURE 3 Sequence diagram of remote GPU virtualization solutions using on-the-fly data compression

In this work we make use of the rCUDA[7,28] remote GPU virtualization framework to provide edge nodes with GPU capabilities. rCUDA features the architecture depicted in Figure 1. We have slightly modified the communications layer within the rCUDA middleware to gather the memory data copies carried out during the execution of the machine learning applications studied in this article.

It is important to remark that in this article we explore the compression/decompression of memory data copies to/from the remote GPU internally made within rCUDA. That is, we focus on copies between (i) the local host memory of the edge node and (ii) the remote GPU memory of the server. The compression/decompression mechanism is not carried out by the application but by the remote GPU virtualization middleware. Thus, the application remains unchanged and is not aware of using a remote GPU neither it is aware that data exchanged with the remote GPU is compressed. In summary, we propose to artificially improve network performance by reducing the amount of data to be transferred thanks to compression. Notice that our proposal is intended for remote GPU virtualization frameworks.

As commented before, the final goal is to accelerate machine learning applications in edge computing environments. For that purpose, in next sections we explore the suitability of using data compression with the TF applications mentioned in previous sections. First, we will characterize memory data transfers to/from the remote GPU. Next, we will apply the compression libraries previously described, varying also the speed of the network, to assess how both of them affect the performance of the applications.

4 | EXPERIMENTAL RESULTS

In this section, we present and analyze the experimental results. After describing the experimental setup, we characterize memory data copies between CPU and GPU during the execution of the four TF applications under study. Then we compare the different compression libraries. Finally, we analyze the impact on transfer time of transferring the compressed data both from the host memory of the edge node to the GPU memory in the remote server and also in the other way, that is, transferring from the remote server to the edge node. Moreover, different network speeds are used to better analyze the influence of the network.

4.1 | Experimental setup

To assess the benefits of compressing the data to be sent through the network, we will consider the two scenarios shown in Figure 4. On the one hand, we have scenario A, where the remote GPU virtualization framework does not compress the information to be sent. On the other hand, scenario B compresses the information before sending it, and decompresses it upon reception. For the experiments, we will use the TF applications presented in Section 2.2. In addition, when considering scenario B, we will use the different compression libraries described in Section 2.3.

In the experiments, we will use a Raspberry Pi 4 Model B with Quad core ARM Cortex-A72 64-bit 1.5GHz as the edge node. Raspberry Pi is one of the most popular devices[29,30] in edge computing due to its low cost and good performance. Regarding the GPU server used to offload computations from the edge node, it features an Intel(R) Xeon(R) CPU E5-2637 v2 3.50GHz with an NVIDIA V100 GPU.

For connecting the edge node with the GPU server, we will use a 1 Gbps Ethernet wired network. In addition, we will use the Linux traffic shaper (i.e., `tc`) 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 will reduce network bandwidth down to 100 and 10 Mbps.

4.2 | Characterization of data transfers between CPU and GPU

As mentioned before, the solution we propose to accelerate applications running in the edge increases network traffic because CUDA calls are forwarded to the remote GPU. Edge environments usually feature low performance networks, and this can hinder the potential benefits of offloading computations to a remote GPU server.

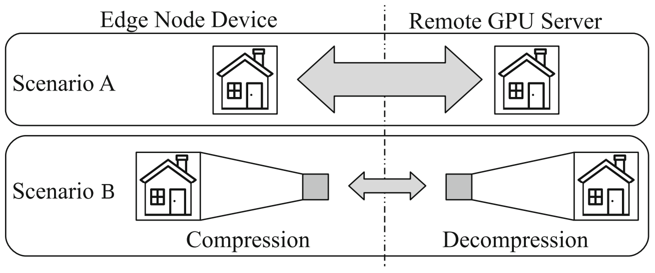

FIGURE 4 Scenarios used in the experimental evaluation

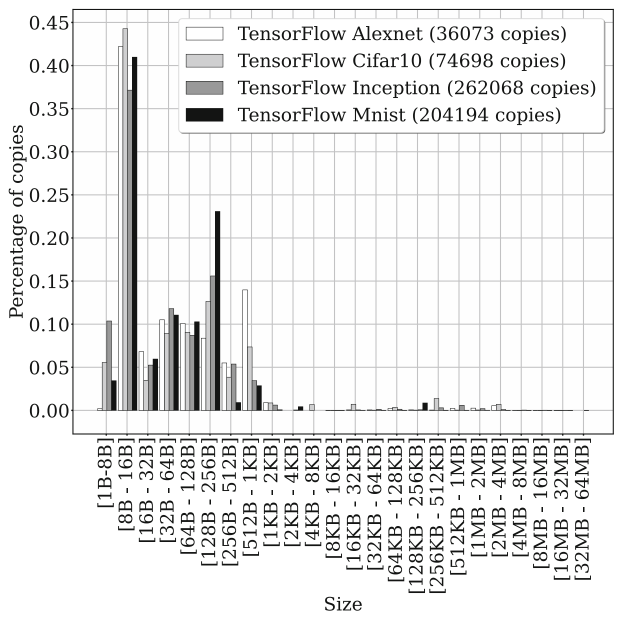

FIGURE 5 Histogram of the data transfers (i.e., memory copies) carried out during the execution of the TF applications under study, that is, Alexnet, Cifar10, Inception and Mnist. Notice that y-axis represents the percentage of memory copies with respect to the total amount of copies performed by each of the TF applications (regardless of the size of the copied data)

To better understand the impact on the network of our approach, we have characterized the size of memory data copies between CPU and GPU during the execution of the four TF samples described in Section 2.2. Notice that data transfers between CPU and GPU of TF applications are not necessarily the input to the applications. In this regard, TF applications usually have their input stored on disk and they read it during their execution. Data read by TF applications are transformed and inserted into the CNN used. During this insertion, different data transfers to/from the GPU are done. In our experiments, TF applications are using the rCUDA middleware to provide the edge device with a powerful GPU. Therefore, to get access to the internal memory data copies carried out by the application to/from the GPU, the rCUDA middleware has been modified to store on disk the data that are sent to the network. In this way, every data transfer to/from the remote GPU done by the TF application creates a new file on disk. At the end, all the data transfers carried out during the execution of the TF samples under study will be stored on disk. These files have been classified into different groups depending on the size of the copied data and have been later used to test the different compression libraries. Notice also that before our modification to rCUDA, the information of these copies was only available during the execution of TF applications. Now, thanks to that modification to the rCUDA communications layer, we can analyze those data transfers without having to execute the TF applications every time. It should be recalled that we are carrying out an initial exploration to assess whether using on-the-fly data compression may be beneficial in the context of edge devices using remote GPU virtualization.

Figure 5 shows a histogram of data transfers (i.e., memory copies) when running TF with the applications under study, that is, Alexnet, Cifar10, Inception and Mnist. In the x-axis we show several ranges of data copy sizes. For instance, the range labeled as '[128B-256B[' includes all the data copies carried out during the execution of an application with sizes ranged between 128 and 255 bytes, including both the lower and the upper limits. It should be noted that Figure 5 is a histogram, therefore, it shows the frequency for each of the data copy size ranges. In this regard, as we are analyzing four different TF applications in the histogram, each of them performing a different amount of data copies, instead of using absolute numbers in the y-axis of the histogram, we use "Percentage of copies". This allows the four applications to be compared more clearly.

As we can observe in Figure 5, 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 2KB. This characteristic will have a direct impact on the performance of compression libraries because typically compression behaves better with larger data sizes.

4.3 | Compressing data transfers between CPU and GPU

In this section, we evaluate the compression ratio that can be achieved by the libraries under study. We compress the data transferred through the network by TF applications when offloading GPU computations from the edge node to the remote GPU server. As commented before, the libraries that we analyze are: BZIP2, LZ4, ZLIB, ZSTD, SNAPPY, GIPFELI, FLZ, and LZO. Some of these libraries allow different settings. Results shown were obtained using the settings that provided the best results for each library.

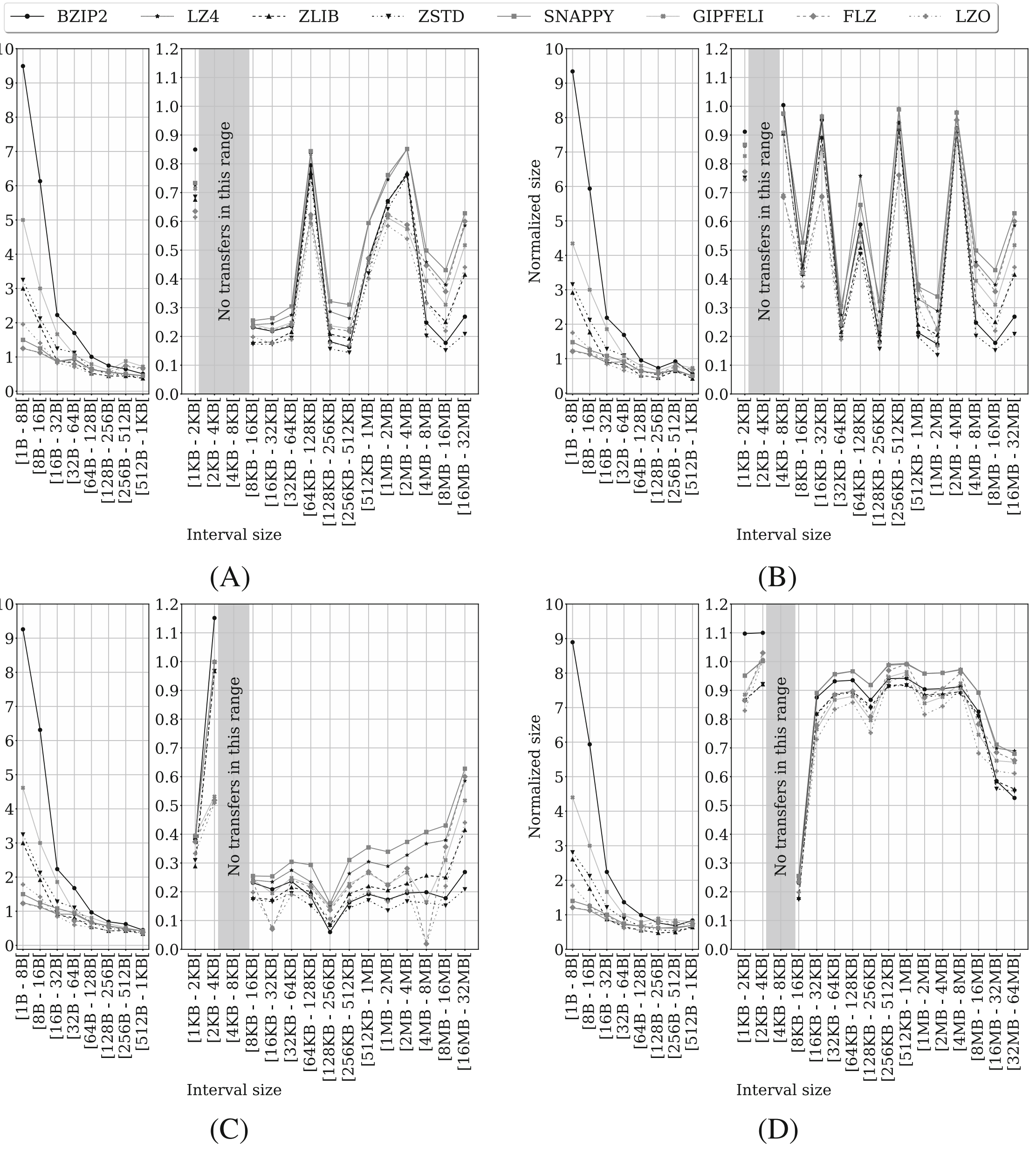

FIGURE 6 Normalized size of the compressed data transferred through the network by TF applications when offloading GPU computations from the edge node to the remote GPU server. Different compression libraries and configurations are used. (A) Alexnet application; (B) Cifar10 application; (C) Mnist application; (D) Inception application

In the next figures, we present the normalized size obtained when each library compresses the data to be transferred through the network. The normalized size has been calculated dividing the size of the compressed data by the size of the original data. Thus, compressing the data are beneficial when the normalized size is below 1. On the left side of the figures we show small size ranges used in data transfers, whereas on the right side we show large size ranges.

Figure 6A shows the results obtained by the different compression libraries when running the Alexnet application. As we can see on the left side of the figure, no library obtains good performance for smallest data sizes. For instance, compressing the data with the BZIP2 library results in data ten times larger than the original one for sizes between 1 and 8 bytes. This is the expected behavior, because these compression libraries include additional information in the compressed data to restore the original data. As we can observe, some compression libraries reduce the size of the original data for sizes over 16 bytes. Over this data size FLZ, FLO, and SNAPPY are the libraries providing better results for small data sizes.

In contrast, on the right side of Figure 6A, we can see that all the libraries obtain good results when compressing large data sizes. As shown in the figure, there are some peaks where compression ratio is lower. This is due to the nature of the application data. For large data sizes, the best compression library is ZSTD followed by BZIP2 and LZO. On the contrary, SNAPPY, LZ4, and FLZ are the worst performing libraries.

Figure 6B presents the results obtained with the Cifar10 application. The results for small data sizes, shown on the left side of the figure, are similar to the ones obtained with Alexnet. However, for large data sizes, on the right part of the figure, there are many peaks where results are closer to 1. These means that the data transferred in this case is more difficult to compress. Similarly to what happened in the Alexnet application, SNAPPY, FLZ, and LZ4 present the worst results for large data transfers, while ZSTD provides the best performance, followed by BZIP2 and LZO.

Figure 6C shows the results for the Mnist application. The results for small data sizes (left side of the figure) are similar to the previous ones. For large data sizes (right side of the figure), a big compression ratio is achieved for sizes between 128 and 256 KB. Thus, compressed data are over 10 times smaller than the original data in the best case. From 256 KB, the compression ratio tends to decrease as the size of data increases. As in previous applications, on average, the best compression library is the ZSTD, followed by BZIP2. SNAPPY and LZ4 are again the worst ones.

Finally, Figure 6D presents the results for the Inception application. For small data sizes, results are in general similar to previous ones. However, we observe a different behavior in the case of large data sizes. The compression ratio tends to be close to 1, but improves with data sizes greater than 16 MB. In the latter case, LZO and GIPFELI present the best results, while SNAPPY and LZ4 the worst ones.

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, that is, the kind of data transferred. For instance, Inception and Cifar10 applications use color images of 180 × 180 and 32 × 32 pixels, respectively. According to our experiments, this kind of data seems to be more difficult to compress. On the other hand, Mnist application uses black and white images of 28 × 28 pixels and yields better compression ratios. In summary, the best compression libraries are ZSTD and BZIP2, while SNAPPY and LZ4 are the worst ones.

4.4 | Impact of transferring compressed data on transfer time

First, in Section 4.2, we have characterized the data transfers made by different applications. Then, in Section 4.3, we have evaluated the compression ratio that compression libraries can achieve. In this section, we evaluate the impact on the transfer time of the applications of transferring compressed data from the host memory of the edge node to the GPU memory in the remote server (the opposite direction will be analyzed in next section). We use different network speeds to study the influence of the interconnect.

The time shown in the next figures has been calculated as the sum of (see Equation 1): (i) time spent compressing the data, (ii) time spent transferring the data through the network, and (iii) time spent decompressing the data. Also, the compression ratio is defined in Equation (2) as the ratio between the size of the original data and size of the compressed data.

$$Time_{total} = Time_{compress} + Time_{transfer} + Time_{decompress}, \tag{1}$$

$$Ratio = \frac{Size_{original}}{Size_{compressed}}. \tag{2}$$

As mentioned, the focus of the article is on accelerating applications in edge computing environments by leveraging remote GPUs. In these environments the speed of the network is usually low. For that reason, we carry out our exploration by using three different network speeds: 1 Gbps, 100, and 10 Mbps.

In Figures 7–10 we maintain a similar structure for showing the data than in the previous figures. Thus, results corresponding to small data sizes are shown on the left side of the figures, whereas results for large data sizes are shown on the right side. Moreover, times shown in the figures are normalized with respect to the transmission time without using compression, that is, the time of transferring the original data through the network without compressing and decompressing it. Thus, using compression is beneficial when the normalized time is below 1.

Figure 7 shows the results for the Alexnet application. As we can see, when using the 1 Gbps network (Figure 7A), our approach does not reduce total transfer time for small data sizes in general. This was to be expected given the results of the previous section, in which compression ratios for smaller messages were not noticeable. Now, when compression and decompression times are also considered, the overall result is that using compression increases total time. On the other hand, for larger data sizes, however, LZ4 and SNAPPY present a significant reduction. They can reduce over a quarter of the time when data size is between 16 KB and 2 MB.

The curve for the BZIP2 compression algorithm on the right side of Figure 7A presents much larger total times than the rest of algorithms. This avoids a clear presentation of the results for the other algorithms. Thus, in Figure 7B the results for 1 Gbps have been zoomed to better show the differences among the compression libraries in the bottom part of Figure 7A. It can be clearly seen on the right side of Figure 7B that three of the compression algorithms provide benefits up to size ranges equal to 4 MB.

Figure 7C shows the results for Alexnet using a 100 Mbps network. Similarly to what happened with the 1 Gbps network, compression has no benefits for small data sizes. Results are even worse, the reason being that compressed data are larger than original data in most cases for these smaller data ranges and, therefore, when using a slower network, total time is increased. However, for larger data sizes, the reduction in total time is even greater than with the 1 Gbps network, where time is reduced by up to 4 times in some data size ranges. The best compression libraries in this

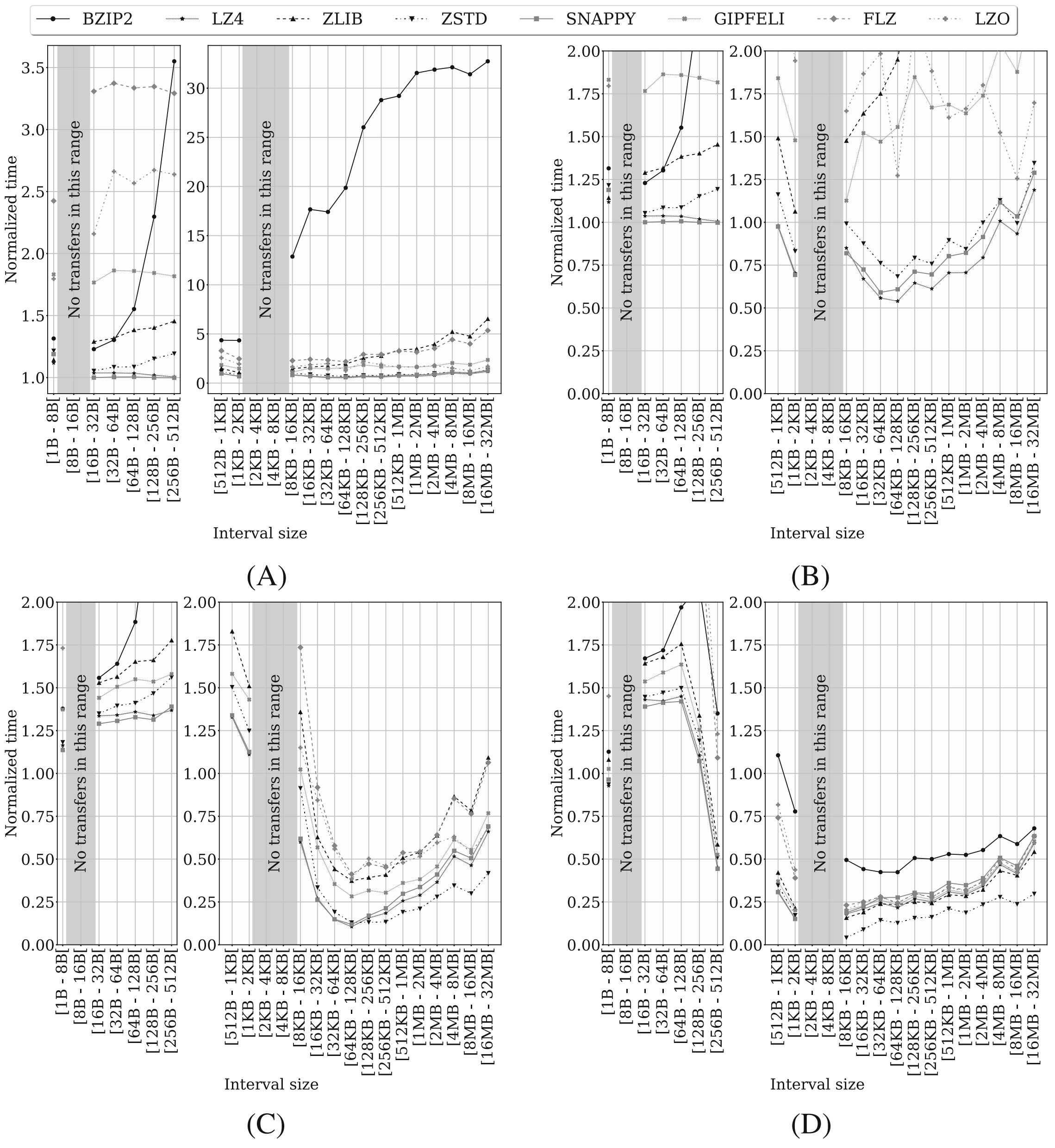

FIGURE 7 Impact on total transfer time when compressing data for the Alexnet application using different networks. Normalized time includes the time for compressing, sending, and decompressing the data. Only memory data copies from the edge node to the remote GPU are considered. (A) Using a 1 Gbps network; (B) using a 1 Gbps network (zoom); (C) using a 100 Mbps network; (D) using a 10 Mbps network

experiment are ZSTD, LZ4, and SNAPPY. The reason for the relative improvement over the 1 Gbps results is that compression in these cases reduces data size and, as network bandwidth is smaller, the relative benefit of transmitting a smaller amount of information is larger than for faster networks.

As shown in Figure 7D, our approach is highly beneficial for Alexnet when using a 10 Mbps network. Almost all the algorithms are beneficial for transfers over 256 bytes. The reason is that the network is so slow that as soon as compressed data sizes are slightly smaller than the original size, total time is noticeably reduced. This is the case, for instance, of the 256 B range, which achieved no benefit with the other network speeds. In some cases, the transfer time is reduced by up to 10 times. In this experiment, the most effective libraries are ZSTD, ZLIB, and SNAPPY.

It is worth paying attention to an interesting behavior related to the BZIP2 compression library in Figures 6A and 7. As shown in Figure 6A, the BZIP2 library is one of the best performing libraries for data copies larger than 512 KB. However, Figure 7A shows that the BZIP2 library provides the worst normalized time, specially for data copies larger than 512 KB. The reason for this different behavior in both figures is compression and decompression times. That is, the BZIP2 library achieves an extraordinary compression ratio, as shown in Figure 6A, at the cost of spending considerable computational effort. It will be better shown later in the article in Figure 12. This increased computational effort to achieve a very good

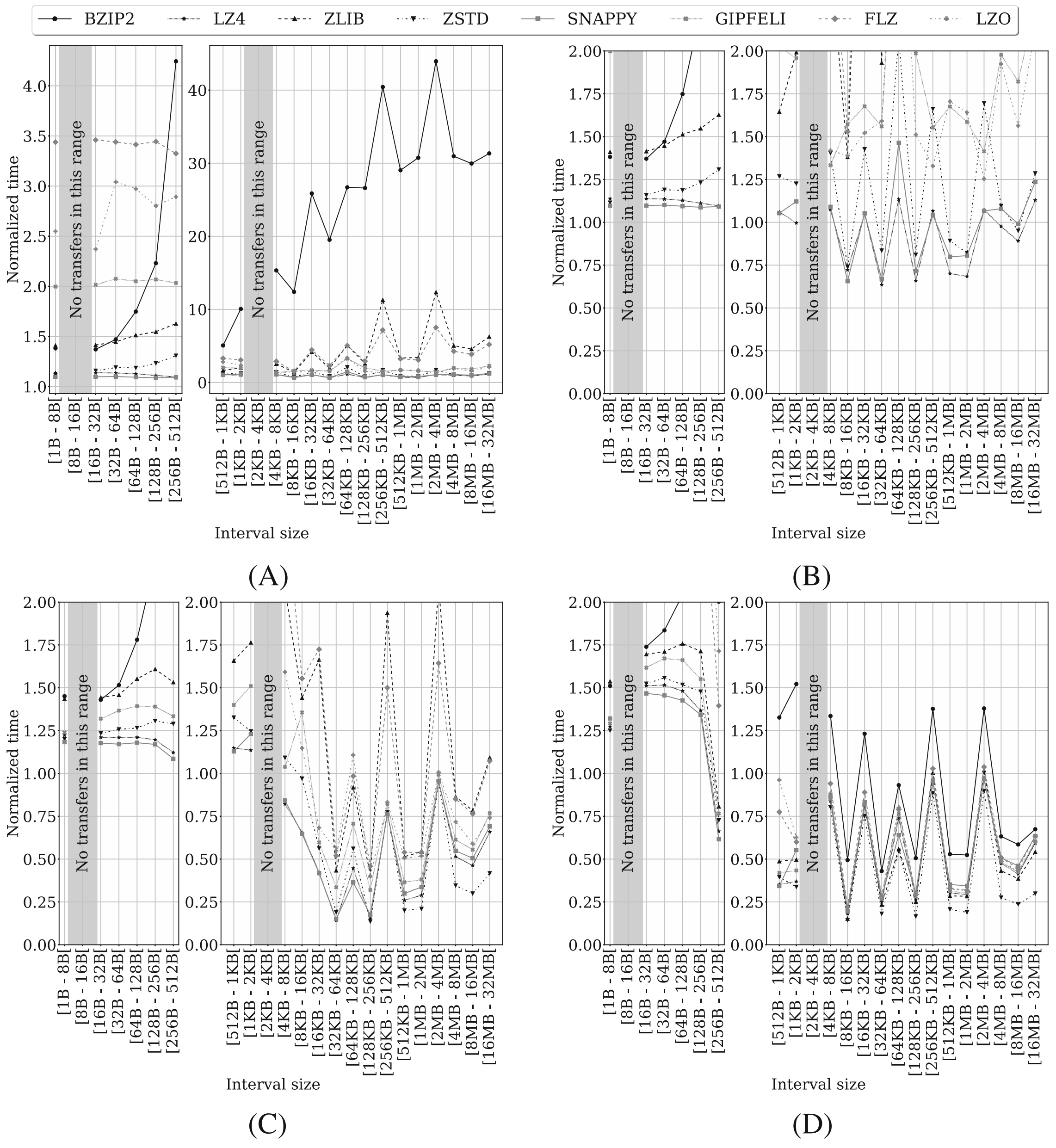

FIGURE 8 Impact on total transfer time when compressing data for the Cifar10 application using different networks. Normalized time includes the time for compressing, sending and decompressing the data. Only memory data copies from the edge node to the remote GPU are considered. (A) Using a 1 Gbps network; (B) using a 1 Gbps network (zoom); (C) using a 100 Mbps network; (D) using a 10 Mbps network

compression ratio does not compensate the reduction in data size when the network is relatively fast, as the 1 Gbps fabric. For this network speed it is more convenient to use a compression library with a poorer compression ratio but with shorter computation times (i.e., less computational effort). Figure 7D provides additional information regarding this behavior. It can be seen in Figure 7D that the performance of the BZIP2 library approaches that of the rest of considered libraries. The reason is that for this very slow network, the reduction in data size partially compensates the huge computational time of BZIP2. Notice, however, that the BZIP2 library still presents the worst behavior for 10 Mbps networks. Thus, its long compression and decompression times will potentially be worth only for extremely slow networks.

Figure 8 shows the results for the Cifar10 application. When using the 1 Gbps network (Figure 8A,B), the compression is not beneficial for small data sizes. For large data sizes, however, some compression libraries reduce the data for almost any size range. LZ4 and SNAPPY obtain the best transfer time. In some cases, they reduce transfer time by a quarter.

Figure 8C presents the results for Cifar10 using a 100 Mbps network. For small data sizes, no library reduces time. On the contrary, as in the case for Alexnet, for large data sizes the compression benefits are more relevant, with ZSTD, LZ4, and SNAPPY performing better than the rest

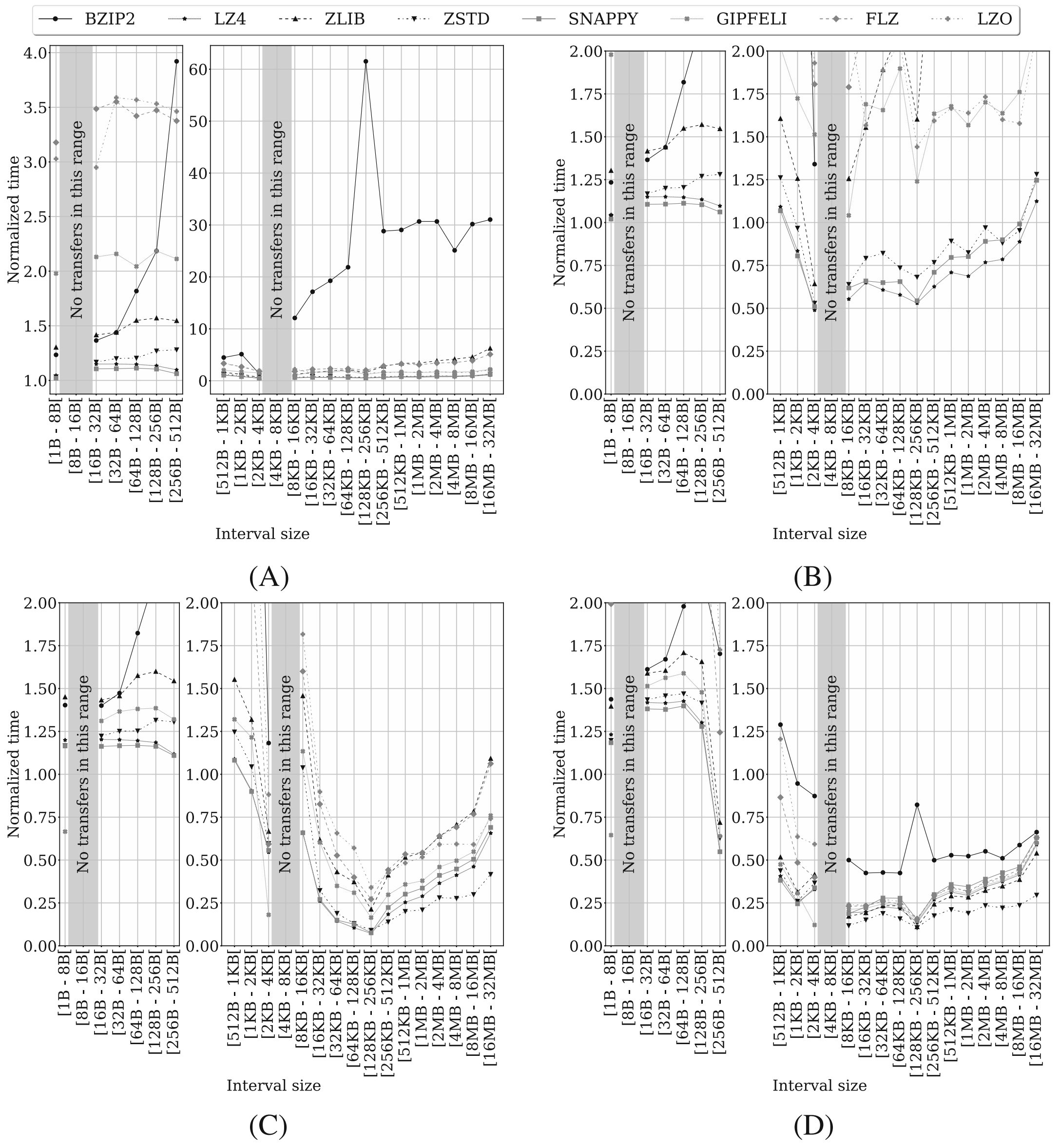

FIGURE 9 Impact on total transfer time when compressing data for the Mnist application using different networks. Normalized time includes the time for compressing, sending and decompressing the data. Only memory data copies from the edge node to the remote GPU are considered. (A) Using a 1 Gbps network; (B) Using a 1 Gbps network (zoom); (C) Using a 100 Mbps network; (D) Using a 10 Mbps network

of compression libraries. However, in this case results are worse than with Alexnet (see Figure 7C). There are some peaks indicating that data are difficult to compress. The reason is that the compression ratio for the data used in Cifar10 is lower, as shown in Figure 6B.

Again, the best scenario for our approach is when using a 10 Mbps network, shown in Figure 8D. For small data sizes, using compression leads to reductions of over 30% for sizes over 256 bytes. For large data sizes, almost all the compression libraries reduce total transfer time. The only exception is for size ranges where data compression ratio is very low. Again, in some cases the transfer time is reduced by up to 10 times. In this experiment, ZSTD, ZLIB, and SNAPPY present the best results.

Figure 9 shows the results for the Mnist application. When using the 1 Gbps network (Figure 9A,B), the compression is not beneficial for small data sizes in general, as with previous applications. For large data sizes, however, LZ4 and SNAPPY reduce time by about half in some cases. It should be noted that in this case the results worsen as data size increases. We can see in Figure 6C that this behavior is due to the compression ratio decreasing as data size increases.

Figure 9C presents the results for Mnist using a 100 Mbps network. Compression libraries do not present any reduction for smaller data sizes. However, for larger data sizes, SNAPPY, ZSTD, and LZ4 get reductions over 4 times in several size ranges. Also, they reduce transfer time by up to 10 times with data sizes between 128 and 256 KB.

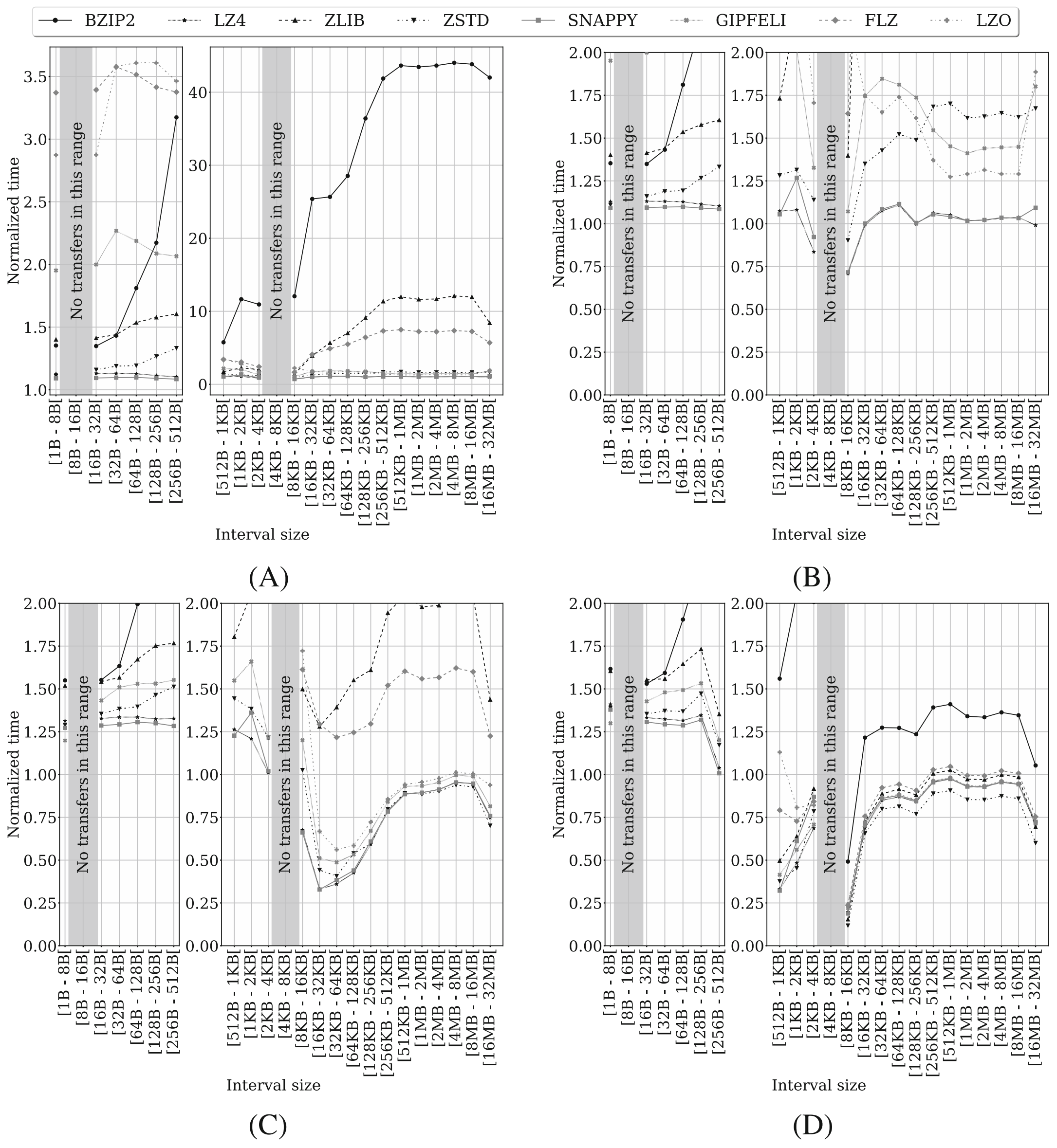

FIGURE 10 Impact on total transfer time when compressing data for the Inception application using different networks. Normalized time includes the time for compressing, sending and decompressing the data. Only memory data copies from the edge node to the remote GPU are considered. (A) Using a 1 Gbps network; (B) using a 1 Gbps network (zoom); (C) using a 100 Mbps network; (D) using a 10 Mbps network

As in previous experiments, the best scenario for our approach is when using a 10 Mbps network, shown in Figure 9D. For 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 is able to reduce the time up to almost half. For large data sizes, all the compression libraries reduce the transfer time. ZSTD presents the best results, reducing transfer time by up to 10 times.

Figure 10 shows the results for the Inception application. When using the 1 Gbps network (Figure 10A,B), results differ from previous experiments. LZ4 and SNAPPY are the only compression libraries presenting a significant reduction. Unlike the previous applications, data used in Inception are larger and more complex. This makes it more difficult to compress, as shown in Figure 6D. For that reason, our approach does not present good results in this case.

Figure 10C 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 8 KB. ZSTD, SNAPPY, and LZ4 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 10D. As we can see, ZSTD, ZLIB, LZ4, SNAPPY, and GIPFELI reduce the time for all data size ranges from 256 bytes.

TABLE 1 Summary of compression libraries presenting the best results in the experiments shown in Section 4.4

Network →	Small data sizes ([1B-512B[)			Large data sizes ([512B-32MB[)		
Application↓	**1 Gbps**	**100 Mbps**	**10 Mbps**	**1 Gbps**	**100 Mbps**	**10 Mbps**
Alexnet	SNAPPY	SNAPPY	SNAPPY	LZ4	ZSTD	ZSTD
Cifar10	SNAPPY	SNAPPY	SNAPPY	LZ4	ZSTD	ZSTD
Mnist	SNAPPY	SNAPPY	SNAPPY	LZ4	ZSTD	ZSTD
Inception	SNAPPY	SNAPPY	SNAPPY	LZ4	ZSTD	ZSTD

In summary, in this section we have seen that transferring compressed data provides benefits depending on the network speed and also on the exact size of the data to be exchanged with the remote GPU. As expected, the slower the network is, the better results are. Thus, when using 100 and 10 Mbps networks, reduction in total time can be up to 10 times. In our experiments, LZ4, ZSTD, and SNAPPY were the compression libraries performing the best in general, as shown in Table 1.

4.5 Impact of computational power on data compression

In the previous section we have studied transfers from the host memory of the edge node to the GPU memory of the remote server (i.e., from left to right in the scenarios shown in Figure 4). Thus, the edge node compresses data while the remote server decompresses data. In general, compression is more complex than decompression and therefore requires more computational power. As mentioned, the computational power of the edge node is lower than the one of the remote server.

Data transfers can also happen in the opposite direction, where data are copied from the GPU memory of the remote server to the host memory of the edge node (i.e., from right to left in the scenarios shown in Figure 4). To study the impact of computational power on the performance of transferring compressed data, in this section we show results for these transfers.

We have repeated all the experiments in the previous section, but with data transfers done from the remote server to the edge node. Due to space limitations, we only present the results that better show the impact of the computational power on the performance, that is, results when using a 10 Mbps network. However, similar conclusions can be drawn for the rest of networks speeds (1 Gbps and 100 Mbps).

Figure 11 shows the results for the different applications. If we compare these results to the ones in previous section, we can observe that almost all the algorithms improve: (i) Figure 11A compared to Figure 7D, (ii) Figure 11B compared to Figure 8D, (iii) Figure 11C compared to Figure 9D and (iv) Figure 11D compared to Figure 10D

As mentioned, the reason for this improvement is that the computational power of the remote server is higher than the one of the edge node. Thus, compression in the remote server is done faster and decompression in the edge node is done slower. As compression requires more computational power than decompression, the total transfer time is reduced when compared to the opposite memory transfer direction because now compression is carried out in a better CPU. Notice that results in Figure 11 do not consider performing the compression in the GPU installed in the remote server (i.e., compression is done in the CPU). This is considered as future work, as explained in Section 5.

Similarly to what happens in previous section, it should be noted that with faster networks speeds (i.e., 1 Gbps and 100 Mbps), the improvements are lower. In general, the compression libraries performing the best in these experiments are the same ones as in previous sections, that is, LZ4, ZSTD, and SNAPPY.

4.6 Impact on overall application performance

In the previous sections, we have separately studied the impact of transferring compressed data to the GPU and from the GPU. We have presented the normalized time for each of those cases. In this section, we evaluate the impact on the overall performance of the applications under study. Again, we use different network speeds to study the influence of the network.

In Figure 12 we can see the total time for each application without compressing data transfers, labeled as "Original", and also the total time when compressing data transfers with the different libraries under study. As explained in Section 4.4, the total time is composed of three parts: (i) time spent compressing the data, (ii) time spent transferring the data through the network, and (iii) time spent decompressing the data (see Equation 1). The total time shown in the figure is breakdown based on these three parts.

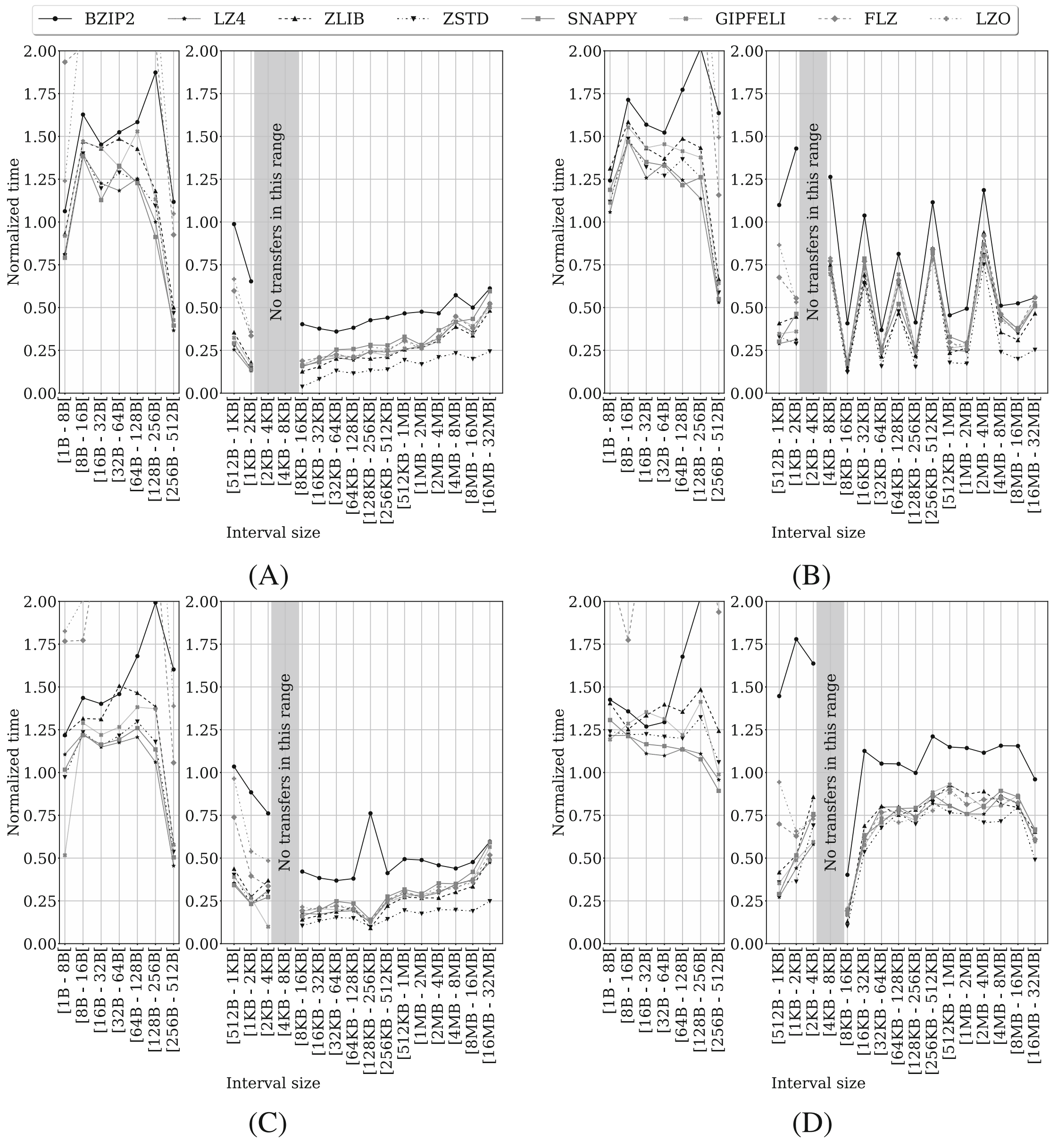

FIGURE 11 Impact on total time when compressing data for the different applications using a 10 Mbps network. In this case, copies are performed from the GPU memory of the remote server to the host memory of the edge node. Normalized time includes the time for compressing, sending and decompressing the data. (A) Alexnet application; (B) Cifar10 application; (C) Mnist application; (D) Inception application

A new compression mechanism, referred to as "Adaptive", has been created in this section based on the results obtained in Sections 4.4 and 4.5. In those sections we have seen that, in general, the behavior of compression libraries has been similar for each of the applications under study: (1) on the one hand, we can see that there are smaller data size ranges where compression is not beneficial because it adds some latency due to a bad compression ratio, and (2) on the other hand, for every data size range, we can find a compression library that performs better than the others. Considering this, we have created the new mechanism called "Adaptive". It uses the best possible compression library depending on the exact size of the data to be transmitted. Furthermore, in case the size of the data to be transmitted is too small to provide any benefit, the original data is sent uncompressed.

Figure 12 shows very interesting results. For the four TF applications under study, when using a 1 Gbps network, Adaptive compression offers similar results than when not using compression. This is not surprising because for this network speed compression is in general not beneficial and adaptive mechanism avoids using compression. In any case, this is an important result because it shows that our proposal self adjusts depending on the characteristics of the underlying network fabric. On the other hand, it is interesting to notice that for many of the compression libraries under study, compression requires more time than decompression, showing that compression is computationally heavier.

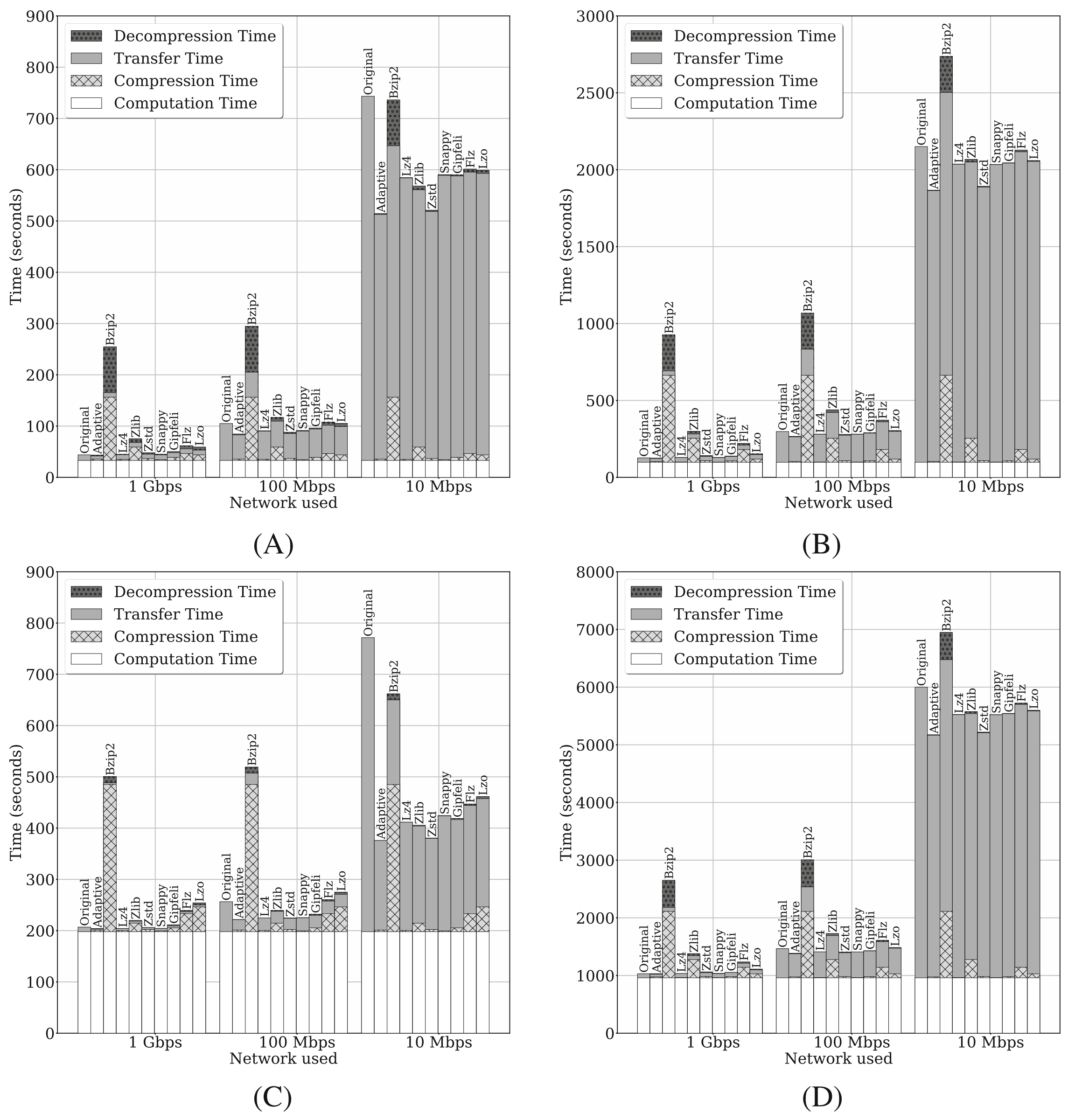

FIGURE 12 Impact of transferring compressed data on the overall performance of the applications using different networks. Time shows the total application time without using compression, labeled as "Original", and compressing data transfers with different algorithms. "Adaptive" refers to using the best compression library depending on the size of the data transfer and the network speed. (A) Alexnet; (B) Cifar10; (C) Mnist; (D) Inception

When using a 100 Mbps network, Adaptive compression is the one that performs the best. Thus, the time of Alexnet, Cifar10, Mnist and Inception applications is reduced by 20, 32, 35, and 80 s, respectively. Finally, Adaptive compression obtains again the best results when a 10 Mbps network is used. In this case, the execution time of Alexnet application is reduced by almost 4 min, Cifar10 by over 4 min, Mnist by 6.6 min and Inception by almost 14 min.

In summary, from the results in this section we can draw similar conclusions than in previous sections: as network speed is reduced, transferring compressed data provides more benefits, reducing total execution time by up to almost 14 min in the best case. Notice, however, that these results can potentially be improved, as it will be discussed in the future work (Section 5).

A final consideration in this section is related to the fact that compression takes longer than decompression, as shown in Figure 12. It should be recalled that we are using two different devices: (i) the edge node, which executes the TF application, and (ii) the remote GPU server, which is computationally more powerful. In this scenario, one might ask whether the actual reason for the compression times being larger is simply because there are much more data copies from the edge device to the remote GPU than in the opposite direction. Thus, most data is compressed in the edge node, which features a less powerful CPU.

TABLE 2 Number of data transfers (i.e., memory copies) made during the execution of the four TF applications under study

Transfer Direction→ Application↓	From edge node to remote GPU	From remote GPU to edge node
Alexnet	20,112	15,961
Cifar10	34,918	39,780
Mnist	112,616	91,578
Inception	131,933	130,135

To answer this question, Table 2 shows the number of data transfers (i.e., memory copies) made in each of the directions for all the four TF applications under study. 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. On the contrary, Cifar10 presents more data copies in the opposite direction, form the remote GPU server to the edge node. In any case, the differences in the number of copies does not seem to be the reason for the differences between compression and decompression times. Thus, compression must be computationally heavier than the decompression. This becomes especially evident for BZIP2 library, where the differences are huge between compression and decompression times for the four TF applications. This happens regardless of which copy direction presents more number of copies.

5 | CONCLUSIONS AND FUTURE WORK

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 processing all or part of the computations at the edge of the network. However, edge devices may not have enough computing power to support any kind of applications, such as the popular machine learning ones. The use of remote virtual GPUs provided by remote GPU virtualization frameworks can address this problem. This approach enables applications to be accelerated using a virtual GPU physically installed in a remote server. However, this involves sending data through the network and, as mentioned, the network performance can be low in this kind of scenarios.

To address the slow-network problem, data sent/received to/from the remote virtual GPU could be compressed. Notice that this compression would be carried out by the remote GPU virtualization middleware and, therefore, applications do not need to be modified, as they would not be neither aware of using a remote GPU nor aware that data is being compressed before transferred.

However, using compression in the context of remote GPU virtualization frameworks is not trivial. As far as we know, no proposal has been previously made in this regard. When leveraging compression within a remote GPU virtualization middleware, the benefits of reducing the size of data transfers could be canceled out by the overhead of the compression and decompression stages, as well as by the type of the data exchanged with the remote GPU. For that reason, in this article we have carried out an initial exploration, using different compression algorithms and the data generated by several machine learning applications.

We have used popular machine learning applications to explore data compression. After characterizing the GPU data transfers of these applications, we have analyzed the suitability of existing compression libraries for compressing those data transfers. From the experiments, we can conclude that, as network speed is reduced, transferring compressed data become more beneficial, reducing transfer time by up to 10 times in some data size ranges, and total execution time by up to almost 14 min in the applications analyzed. In conclusion, we can state that our initial exploration has reported that using data compression in remote GPU virtualization frameworks in the edge is a feasible and appealing field of research that is worth exploring further.

As for future research, notice that in this article we have carried out a naive implementation of data compression. In this regard, data to be sent are first compressed in the CPU, then sent, and afterwards decompressed using the CPU at the receiver. This mechanism is too simple and it can be significantly improved in future work by concurrently applying the following three different approaches:

- Implementing a compression pipeline. Instead of compressing all data before sending them, a compression pipeline can be implemented. That is, the data to be copied as part of a single CUDA call could be split into several smaller data chunks. Notice that splitting that data into smaller chunks would transparently happen inside the communications layer of the remote GPU virtualization framework and, therefore, the application source code would remain unchanged. Then, after splitting the data into chunks, once a data chunk is compressed, it is sent through the network while the next data chunk is being compressed. Thus, compression and network transfers would progress in parallel, hiding part of the overhead generated by the compression stage. Furthermore, if the receiving peer is considered, compression will happen in parallel to transmission and decompression. Notice that the compression libraries used in our exploration might not be compatible with this pipeline, so a new implementation of the compression libraries might be required.

- Using the GPU for compression. At the server side where a GPU is available, we have compressed/decompressed the data information using libraries which do not leverage the GPU. However, it should be possible to use the GPU in the compression/decompression process. For that reason, another piece of future work could be using algorithms that take advantage of the GPU. However, notice that because the remote GPU will be used by the TF application, this solution might increment application execution time. Thus, how to implement this improvement must be carefully studied. Also note that this GPU implementation is compatible with the pipeline implementation described above.
- Asymmetric compression. As shown in Figure 12, compression and decompression times are usually different for most compression libraries. This allows an additional optimization in the context of on-the-fly compression within remote GPU virtualization frameworks. The additional improvement would be the following. Once the GPU in the remote server is used to execute the compression library, it would be possible to leverage a computationally intensive compression library, such as the BZIP2 one for instance, to noticeably reduce data size without increasing compression time because now compression is carried out on the GPU. Furthermore, compression time would be hindered inside the aforementioned pipeline. On the other peer, at the edge node, decompressing received data would have to be done by the CPU. However, as shown in Figure 12, decompression is a much lighter computational task. On the other hand, for the data copied from the edge node to the remote GPU server, a different compression library should be used. That compression library should require a small computational effort because compression will take place at the CPU of the edge device.

By applying the improvements mentioned above we expect that data compression will be beneficial in many cases. In addition, the implementation of those improvements in collaboration with the remote GPU virtualization framework could even lead to better results.

FUNDING INFORMATION

This work was supported in part by the project "AI in Secure Privacy-Preserving Computing Continuum (AI-SPRINT)" through the European Union's Horizon 2020 Research and Innovation Programme under Grant 101016577, and in part by the European Union's Horizon 2020 research and innovation programme under Grant agreement No. 101017861.

CONFLICT OF INTEREST

The authors declare no conflict-of-interests.

DATA AVAILABILITY STATEMENT

The data that support the findings of this study are available on request from the corresponding author. The data are not publicly available due to privacy or ethical restrictions.

ENDNOTE

*https://quixdb.github.io/squash-benchmark/

ORCID

Cristian Peñaranda https://orcid.org/0000-0003-3805-4584
Carlos Reaño https://orcid.org/0000-0001-7871-9152
Federico Silla https://orcid.org/0000-0002-6435-1200

REFERENCES

1. Salman O, Elhajj I, Kayssi A, Chehab A. Edge computing enabling the Internet of Things. Proceedings of the 2015 IEEE 2nd World Forum on Internet of Things (WF-IoT); 2015:603-608.
2. Sittón-Candanedo I, Alonso RS, García O, Muñoz L, Rodríguez-González S. Edge computing, IoT and social computing in smart energy scenarios. *Sensors*. 2019;19(15): 3353. doi:10.3390/s19153353
3. Ren J, Pan Y, Goscinski A, Beyah RA. Edge computing for the Internet of Things. *IEEE Netw*. 2018;32(1):6-7. doi:10.1109/MNET.2018.8270624
4. Abadi M, Barham P, Chen J, et al. TensorFlow: a system for large-scale machine learning. OSDI'16; 2016:265-283.
5. Paszke A. PyTorch: an imperative style, high-performance deep learning library; 2019:8024-8035.
6. NVIDIA Corporation. CUDA (Compute Unified Device Architecture); 2022. https://developer.nvidia.com/cuda-toolkit
7. Silla F, Iserte S, Reaño C, Prades J. On the benefits of the remote GPU virtualization mechanism: the rCUDA case. *Concurr Comput Pract Exp*. 2017;29(13):e4072. doi:10.1002/cpe.4072
8. Giunta G. A GPGPU transparent virtualization component for high performance computing clouds. *Euro-Par Parallel Process*. 2010; 6271:379-391.
9. Kennedy J, Varghese B, Reaño C. AVEC: accelerator virtualization in cloud-edge computing for deep learning libraries. Proceedings 2021 IEEE 5th International Conference on Fog and Edge Computing (ICFEC); 2021:37-44.
10. Krizhevsky A, Hinton G. Learning multiple layers of features from tiny images; 2009.

11. Jorge J, Giménez A, Iranzo-Sánchez J, Civera J, Sanchis A, Juan A. Real-time one-pass decoder for speech recognition using LSTM language models. INTERSPEECH; 2019:3820-3824.
12. Hussain M, Bird JJ, Faria DR. A study on CNN transfer learning for image classification. Proceedings of the UK Workshop on computational Intelligence; 2018:191-202.
13. Miyato T, Dai AM, Goodfellow I. Adversarial training methods for semi-supervised text classification. arXiv preprint arXiv:1605.07725; 2016.
14. Krizhevsky A, Sutskever I, Hinton GE. Imagenet classification with deep convolutional neural networks. *Adv Neural Inf Process Syst*. 2012;25:1097-1105.
15. LeCun Y, Cortes C, Burges C. MNIST handwritten digit database. ATT Labs [Online]; 2010:2. http://yann.lecun.com/exdb/mnist
16. Szegedy C, Vanhoucke V, Ioffe S, Shlens J, Wojna Z. Rethinking the inception architecture for computer vision. Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition; 2016:2818-2826.
17. TensorFlow. Flowers; 2019. http://download.tensorflow.org/example_images/flower_photos.tgz
18. Seward J. BZIP2 website. Accessed 20th July, 2021. https://www.sourceware.org/bzip2/
19. LZ4. LZ4 website. Accessed 20th July, 2021. https://lz4.github.io/lz4/
20. Gailly J, Adler M. ZLIB website. Accessed 20th July, 2021. https://zlib.net/
21. Facebook. ZSTANDARD website. Accessed 20th July, 2021. https://facebook.github.io/zstd/
22. Google. SNAPPY - a fast compressor/decompressor; 2021. https://github.com/google/snappy
23. Google. Gipfeli, a high-speed compression library. https://github.com/google/gipfeli
24. Oberhumer MF. LZO website. Accessed 20th April, 2022. http://www.oberhumer.com/opensource/lzo/
25. Ariya FastLZ, Small & portable byte-aligned LZ77 compression; 2022. https://github.com/ariya/FastLZ
26. Promberger L, Schwemmer R, Fröning H. Assessing the overhead of offloading compression tasks. Proceedings of the 49th International Conference on Parallel Processing-ICPP: Workshops; 2020:1-10.
27. Welton B, Kimpe D, Cope J, Patrick CM, Iskra K, Ross R. Improving I/O forwarding throughput with data compression. Proceedings of the 2011 IEEE International Conference on Cluster Computing; 2011:438-445.
28. Iserte S, Prades J, Reaño C, Silla F. Increasing the performance of data centers by combining remote GPU virtualization with Slurm. Proceedings of the 2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid); 2016:98-101; ACM.
29. Sforzin A, Mármol FG, Conti M, Bohli JM. Rpids: Raspberry pi ids—A fruitful intrusion detection system for IoT; 2016:440-448; IEEE.
30. Miori L, Sanin J, Helmer S. A platform for edge computing based on Raspberry Pi clusters; 2017:153-159; Springer.

How to cite this article: Peñaranda C, Reaño C, Silla F. Exploring the use of data compression for accelerating machine learning in the edge with remote virtual graphics processing units. *Concurrency Computat Pract Exper*. 2022;e7328. doi: 10.1002/cpe.7328