Practical Evaluation of Remote Adaptive Lossy Compression with Feedback

CONNOR ENNIS and JON C. CALHOUN (ADVISOR), Holcombe Department of Electrical and Computer Engineering - Clemson University, USA

When transmitting image data from a deployed edge device, a high-bandwidth connection to a cloud system cannot be guaranteed. An early-warning system for an intersection crosswalk, for instance, would have to be able to compress and transmit data with enough quality to ensure prompt detection of danger through remote image processing. Adaptive lossy compression provides a potential solution for this, although it is yet to be evaluated on actual edge hardware. Through practical adaptation and implementation of this model, we evaluate the viability of updating lossy compression parameters through detector feedback on a real cloud-edge system.

Additional Key Words and Phrases: remote image processing, lossy compression, feedback, cloud-edge communication

ACM Reference Format:
Connor Ennis and Jon C. Calhoun (Advisor). 2022. Practical Evaluation of Remote Adaptive Lossy Compression with Feedback. 1, 1 (August 2022), 5 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Edge-cloud computing architectures amplify the potential of low-power devices, making it possible to perform intensive processing remotely. Object detection at road intersections is one practical application, requiring the detection and transmission of pedestrian positions in real time. The edge device, with minimal onboard hardware, could transmit footage to and receive positional information from the cloud server to alert people and vehicles of potential danger. Due to the broad variation in environmental conditions, however, it is impossible to guarantee that edge devices will have access to high-bandwidth connections. Since image files tend to be large and must be transmitted immediately to ensure accurate safety information, this data must be compressed in a manner that effectively preserves detection accuracy, while minimizing network and compute costs. In our source solution [3], images are split and compressed by region, with tolerance values for each segment updated based on detection bounding boxes. By adjusting lossy compression from feedback, bandwidth is prioritized for regions containing crucial information. Despite the promise of this method, however, it was verified only via simulation entirely on an HPC device rather than a more realistic platform.

Thus, our solution extends this implementation in several aspects, by:

- Separating the compression and detection pipelines between client and server processes
- Improving compression ratios by up to 4.95% via a unified lossless stage
- Demonstrating client-side compression performance on an Arm-powered edge device
- Optimizing and benchmarking network performance under a range of realistic bandwidth conditions

Through these contributions, we demonstrate how the idealized model for adaptive lossy compression could be refined and implemented practically, and verify its utility for remote image detection on edge devices.

Authors' address: Connor Ennis, cjennis@clemson.edu; Jon C. Calhoun (Advisor), jonccal@clemson.edu, Holcombe Department of Electrical and Computer Engineering - Clemson University, 433 Cahoun Dr, Clemson, South Carolina, USA, 29634.

2022
XXXX-XXXX/2022/8-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

2 BACKGROUND

While object detection models have become incredibly robust in recent years, they still often exceed the capabilities of low-cost embedded hardware. GPU and tensor-enabled system-on-chips are gaining popularity, but their heightened power consumption and cost make them less feasible for mass deployment. A practical alternative is the development of cloud-edge architectures, where a compute-enabled server processes data transmitted from a low-power edge device. This split pipeline is far more practical for infrastructural applications, such as traffic surveillance systems at intersections, but such sites could vary dramatically in internet service quality. Thus, we require an appropriate method for compressing image data.

SZ, a lossy compressor specialized in floating-point data [2], possesses the advantage of basing quality preservation on specific error quantizations. It operates internally by segmenting information into chunks, each of which is analyzed and approximated with a custom function based on the assigned error metric (such as PSNR, for instance). The resulting data then undergoes a lossless pass through a compressor such as zstd [1], further minimizing output.

Cavender's theoretical solution builds upon this algorithm, adapting it to a realistic detection architecture. Rather than applying compression to the complete image, we separate the process into a pipeline. The image is segmented into split regions, each of which has its own PSNR value. This enables dynamic tuning of quality across the mapped segments through SZ. The bounding boxes generated by the Yolo v3 detector [4] can then be used for real-time refinement: quality is raised for regions intersected by boxes, and reduced for all others. With this, it is possible to dynamically optimize frame compression based on feedback from the object detector, minimizing image size as much as possible without reducing accuracy.

In place of separate server and client devices, the code developed to verify this model runs on a single HPC node, with all communications handled internally. As an evaluation prototype, this adequately demonstrates the feasibility of using feedback to adjust SZ PSNR on the fly, preserving detection accuracy while reducing data throughput. At the same time, however, this test does not accurately represent client-server deployment, both in the context of running compression on edge hardware, and in transmitting data between the two systems. Furthermore, the intermediate nature of compressed data (which is simply immediately decompressed in the simulation) does not account for additional overhead associated with network transmission, instead using only predicted timing values.

3 METHODS

Hardware	Palmetto Cluster	Raspberry Pi
CPU	Intel(R) Xeon(R) Gold 6238R x86-64 @ 2.20GHz (20× cores used)	Broadcom(R) BCM2711 Arm v8 @ 1.5GHz (4× cores used)
RAM	16GB DDR4	2-4GB LPDDR4
GPU	Nvidia Tesla V100	N/A

Table 1. Cloud and edge platform hardware

For a practical feedback compression architecture, we must designate pipeline components between client and server processes. In keeping with the project's prior iteration, we chose a GPU-enabled node on Clemson's Palmetto Cluster for our compute device, as detailed in Table 1. Our edge device, dramatically disparate from Palmetto, is a Raspberry Pi single-board computer with specifications also listed in Table 1. As with the original implementation, we use LibPressio

for compression [5], and TensorFlow for object detection. The Pi's reduced core count and speed mandated multiple pipeline modifications, such as queuing groups of frame splits when they exceed the number of MPI processes. To further minimize thread usage, some stages of the client-side pipeline (such as compression and network communication) were consolidated, also reducing MPI copy overhead. As mentioned previously, SZ applies a lossless pass to each compressed array, such that each image split undergoes both stages. As an optimization, we removed this internal step, opting for a single lossless pass of the entire frame. This reduces the memory, processing and metadata overhead of initializing multiple lossless compressors.

Realizing adaptive regional compression requires handling communication between the server and client efficiently. The actual technical details are less relevant, and many communication protocols could substitute for the TCP over SOCKS5 configuration used, but it is worth noting that this pairing provides the data security of an encrypted SSH connection combined with the resilience and ordering of TCP. Communication is bidirectional: the Raspberry Pi sends compressed image data to the Palmetto Cluster node, while the server returns updated tolerance values for future frames. To obtain consistent results, we compute new tolerances every frame in a synchronized manner. That is, at any given point in time, the client and server have the same compressor configuration. Despite introducing some idle time, this ensures that maximum accuracy is obtained through frequent tolerance updates. As a baseline, we also created uncompressed, static SZ (without updated regions) and blosc-only variations of this network implementation.

Our verification methodology was kept as faithful to the source work as possible, retaining the 362 image dataset of crosswalk footage along with baseline annotations and box data from the simulated model. Therefore, for the main pipeline, an ideal test would produce the similar accuracy results to the simulation. Benchmarking SZ on the Pi enables us to observe additional hardware-specific performance metrics not attainable via the pure-HPC implementation, so we also evaluated compression bandwidth (uncompressed size over time to compress, in MB/s, from raw image to final package). Finally, we ran all baselines and pipeline split configurations to determine their performance (in frames per second) at a series of capped communication bandwidths (256 KB/s, 512 KB/s, 1024 KB/s, 2048 KB/s and 4096 KB/s). This will provide a holistic overview of how each method compares in solving the issue of practical cloud-edge image data transmission.

4 RESULTS

To ensure this implementation remains functionally comparable to the original, our first objective is to measure its detection accuracy across the same data set. Given that the TensorFlow model and compression routine underwent minor revisions since the previous paper, the variation observed is tolerable, with a maximum pipeline accuracy loss of 0.8% for 2 and 8 splits.

As predicted, applying zstd via blosc to the entire frame in a single pass (rather than using SZ's internal lossless step per split) yielded higher compression ratios. Ranging from 1.42% for 2 regions to 4.56% for 16, the percent improvement increases with the number of splits. Compressing the entire frame simultaneously provides additional context for repetition encoding (larger segments of data across the entire file are processed, rather than each segment in isolation), while also eliminating the need for multiple headers, thus accounting for the relationship between split count and ratio improvement.

In spite of improved compression performance, benchmark results on the Pi reveal the limitations of edge hardware, as seen in Figure 2. While the overall decrease in bandwidth is unsurprising considering the difference between a Xeon processor and an Arm SoC, the inverted split performance reveals a crucial limitation of the adaptive lossy compression method itself. Having a fixed hardware thread count, and queuing segments once this is exceeded, dramatically lengthens compression time. Bandwidth is approximately halved when increasing from 4 to 16 regions, despite each split

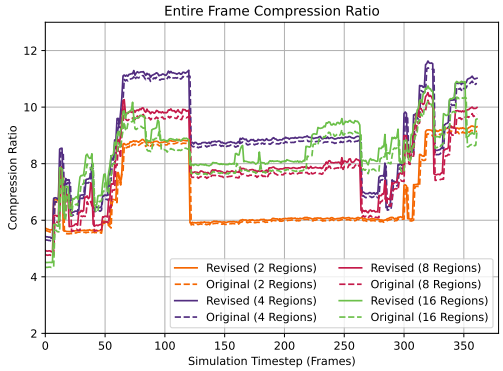
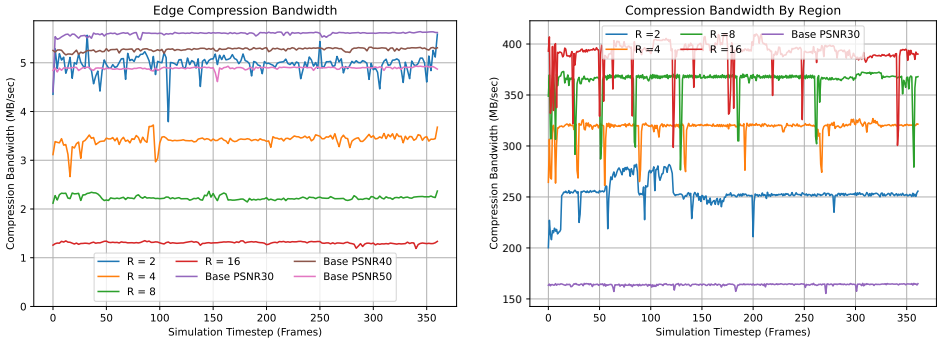


Fig. 1. Comparison of compression ratios for a combined vs. split zstd pass



(a) Revised implementation

(b) Original implementation

Fig. 2. Detection performance variation

being a quarter of the size. This suggests that, even if the input size per compressor is reduced, having to queue and configure the workers requires more processing time. Even with only two splits, achievable bandwidths are comparable to those for fixed PSNR values between 40 and 50.

With this benchmark in mind, we next consider the performance of all configurations under various network conditions in Figure 3. Unsurprisingly, the larger output size for lossless and raw data transmission resulted in a reduced framerate for low bandwidths, with raw frames becoming performant around 2 MB/s and zstd-only frames around 1 MB/s. Results for the pipelined compressors are far more intriguing, especially in the 256 KB/s region. Here, the 4 and 8 split configurations outperform 2 splits. In spite of increased processing time, the inclusion of more regions permits higher compression ratios and network performance, as theorized in the source work. Beyond 512 KB/s, and at all times for higher split counts, the trends resemble those from Figure 2a, with throttling having less of an impact. Given that even the fixed PSNR framerates plateau after 512 KB/s, it appears that edge device compression time rapidly becomes the main limiting factor for our SZ-processed data.

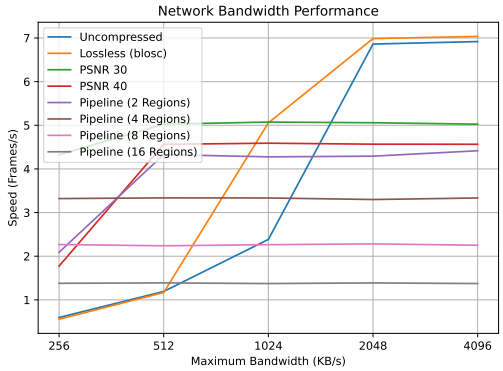


Fig. 3. Average frame processing rate for each configuration as bandwidth increases

5 CONCLUSION

For image processing and other specialized data transmission tasks, adaptive lossy compression has incredible promise for dynamic communication. Unfortunately, in its current state, it underperforms on realistic cloud-edge hardware due to client-side threading requirements. Given that image detection for pedestrian safety systems must occur in real-time, it would be most viable to transmit uncompressed or losslessly compressed data on our selected platform. These results do, however, provide invaluable insight to limiting factors and potential optimizations for devices comparable to the Pi. More specifically, we have observed that parallel processing and compression tasks impose a greater bottleneck than transmission for adaptive compression. Even if the framerates obtained were insufficient for a real-time system, we have demonstrated the useability of SZ on a system with limited resources, as well as successful optimizations for lossless compression of segmented data. Additionally, we see that the size reduction offered by adaptive compression still offers an advantage for low bandwidths, albeit in a smaller range than desired. In the future, we intend to further refine this implementation for the Pi's hardware, such as by combining client-side steps for improved thread efficiency and reduced memory transfer overhead.

ACKNOWLEDGMENTS

Clemson University is acknowledged for generous allotment of compute time on the Palmetto cluster. This material is based upon work supported by the National Science Foundation under Grant No. SHF-1910197 and SHF-1943114.

REFERENCES

[1] Yann Collet and Murray Kucherawy. 2018. Zstandard Compression and the application/zstd Media Type. RFC 8478. https://doi.org/10.17487/RFC8478

[2] Sheng Di and Franck Cappello. 2016. Fast Error-Bounded Lossy HPC Data Compression with SZ. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 730–739. https://doi.org/10.1109/IPDPS.2016.11

[3] Cavender Holt and Jon C. Calhoun. 2022. Improving Surveillance Object Detection Latency through an Adaptive Compression Feedback Loop. (April 2022).

[4] Joseph Redmon and Ali Farhadi. 2018. YOLOv3: An Incremental Improvement. arXiv:1804.02767 [cs.CV]

[5] Robert Underwood. 2021. Libpressio. https://github.com/robertu94/libpressio.