2021 IEEE International Conference on Cluster Computing (CLUSTER) | 978-1-7281-9666-4/21/$31.00 ©2021 IEEE | DOI: 10.1109/Cluster48925.2021.00065

cuZ-Checker: A GPU-Based Ultra-Fast Assessment System for Lossy Compressions

Xiaodong Yu*, Sheng Di*, Ali Murat Gok†, Dingwen Tao‡, Franck Cappello*
*Argonne National Laboratory, Lemont, IL
†Cerebras Systems, Los Altos, CA
‡Washington State University, Pullman, WA
Emails: xyu@anl.gov, sdi1@anl.gov, ali.gok@cerebras.net, dingwen.tao@wsu.edu, cappello@mcs.anl.gov

***Abstract*—Lossy compression is becoming an indispensable technique for the success of today's extreme-scale high-performance computing projects that produce vast volumes of data during scientific simulations or instrument data acquisitions. Comprehensively understanding the compression quality and performance of different lossy compressors is critical to selecting the best-fit compressors and using them properly and efficiently in practice. A few lossy compression assessment tools (e.g., Z-checker) have been developed, but none of them support the execution in a GPU environment. This is a significant gap because many recent extreme-scale applications and lossy compressors (e.g., cuSZ) can run entirely within GPUs. In this work, we develop an efficient lossy compression measuring system (called cuZ-Checker) on the GPU platform, which aims to perform the lossy compression quality and performance assessment completely within the GPU environment. Our contribution is threefold. (1) We develop a novel GPU-based lossy compression measuring framework using a computation pattern-based design approach. This approach classifies the computing-intensive metrics into three categories based on their patterns which creates large opportunities for kernel fusion and data reuse. (2) For each pattern in cuZ-Checker, we develop a CUDA kernel and provide fine-grained optimizations to boost its performance. (3) We thoroughly evaluate our cuZ-checker on a V100 GPU using four real-world scientific application datasets. Experiments show that cuZ-Checker can significantly accelerate the overall lossy compression assessment performance by 23X∼31X compared with the OpenMP-based multithreading CPU performance. To the best of our knowledge, this is the first lossy compression measuring system designed for GPU devices.**

***Index Terms*—lossy compression, GPU, performance optimization, quality evaluation, SSIM**

I. Introduction

Today's scientific applications are producing extremely large amounts of data during simulations or instrument data acquisitions. On the one hand, some scientific projects need to run the simulations on large-scale environments with millions of cores, each of which may output extremely large amounts of data. For instance, when simulating 1 billion particles in the HACC cosmology research project, one simulation run can generate 20 PB of datasets through hundreds of snapshots. On the other hand, advanced instruments such as the Advanced Photon Source [1] and Linac Coherent Light Source (LCLS) [2] can have a data acquisition rate (e.g., 250 GB/s on LCLS-II [3]) too high to store or transmit to disk.

Corresponding author: Sheng Di, Mathematics and Computer Science Division, Argonne National Laboratory, 9700 Cass Avenue, Lemont, IL 60439, USA

Error-bounded lossy compression techniques are becoming the most effective solution to resolve the big-data issue in today's extreme-scale scientific research. These compressors offer on-demand control of data distortion by allowing users to control the compression errors in a specific form, such as absolute error bound, relative error bound, and peak signal-to-noise ratio (PSNR). Moreover, unlike the lossless compressors [4]–[8] that generally suffer from very low compression ratios (around 2:1 in most of cases), error-bounded lossy compressors can generally get fairly high compression ratios (10:1, 100:1 or even higher) in most cases [9]–[12].

Although lossy compressors offer the error-bounding support for scientific data compression, scientists still need a comprehensive understanding of the reconstructed data and compression quality of lossy compressors before using them in practice. A few lossy compression assessment tools have been developed to address this issue, such as Z-checker [13] and Foresight [14]. However, none of the existing compression assessment tools or systems use GPU accelerators, leaving a significant gap in today's extreme-scale scientific simulations.

Over the last decade, GPU has been used as the mainstream accelerator due to its massive parallelism and computational power. A variety of today's extreme-scale scientific research projects, such as cosmology (e.g., EXASKY [15]), quantum chemistry (e.g., EXAALT [16], GAMESS [17]), and climate (e.g., DNN-based climate research [18]), have been successfully accelerated on GPU device. More recently, With the rapid increase of GPU capacity, the coming high-performance computing systems appear to be GPU-centric [19] to ultimately reduce the control and data movement overhead on the host CPU side.

To accommodate this GPU-centric trend, the entire lossy-compression-related workflow, including the compression quality measurement and analysis, should be incorporated into the GPU to better facilitate the GPU-based exa-scale applications. Several GPU-based lossy compressors, such as cuSZ [20] and cuZFP [21], have been proposed. However, the GPU-based lossy compression assessment system is still missing. The current assessment of GPU-based lossy compressors' compression quality is performed on CPU. It re-

978-1-7281-9666-4/21/$31.00 ©2021 IEEE
DOI 10.1109/Cluster48925.2021.00065

quires both the raw and decompressed data of the GPU-based lossy compressors to be moved from GPU memory to CPU memory for in situ analysis, resulting in huge data transfer overhead. Moreover, since the data volume could be extremely large, the CPU-based compression assessment can become a serious bottleneck for the entire simulation/analysis workflow also because of the high time complexity of some metric calculations such as the Structural Similarity Index Measure (SSIM). In fact, comprehensively assessing the GPU-based lossy compression quality is even more critical than assessing the CPU-based compression quality, since the GPU-based lossy compressors generally have to trade off compression quality for execution efficiency due to the characteristics of GPU architecture. For instance, cuZFP supports only fixed-rate mode, which suffers substantially lower compression quality than does its absolute error bound mode. As verified by [22], ZFP's fixed-rate mode could result in $2\sim3\times$ lower compression ratios than its fixed-accuracy mode, with the same level of data distortion (in terms of PSNR). Although the GPU version of the SZ library, cuSZ, supports absolute error bound in lossy compression, its compression quality is much lower than its CPU version [9], [23], [24], since it currently supports only the design of version 1.4 [23]. By comparison, the latest version 2.1 of SZ on CPU [24] has far better compression quality especially for high compression cases, because of the more advanced data prediction algorithm adopted.

Developing a GPU-supported lossy compression assessment tool or system faces several challenges. (1) The compression assessment metrics to be included in the GPU-based checker involve diverse execution patterns, which need careful investigation and classification. (2) Some advanced metrics (e.g., high-dimensional SSIM [25]) yield a heavy computational burden even for GPU devices. Their implementations need fine-grained designs and optimizations to fully leverage both the architectural and algorithmic characteristics.

In this work we present a GPU-based lossy compression assessment framework, cuZ-Checker,and optimize its performance with various strategies. The key novelties of our design are that, we propose an pattern-oriented design by classifying the metrics into three categories based on their computational patterns to allow GPU kernel fusion and data reuse. For each pattern, especially the one covering SSIM, we provide fine-grained design to optimize its GPU kernel. We also thorough evaluation our cuZ-checker using real-world scientific datasets. The detailed contributions are summarized as follows.

- We provide an in-depth investigation for all of the lossy compression assessment metrics supported by Z-checker and classify the computing-intensive metrics into three categories by their computational patterns.
- We accordingly propose a pattern-oriented design for our cuZ-Checker to maximize the CUDA kernel fusion and data reuse. Kernel fusion is an essential GPU performance optimization that can reduce kernel launch overhead as well as the data transfers and memory accesses.
- For each pattern in cuZ-Checker, we provide fine-grained optimizations for its kernel by thoroughly leveraging the advance GPU features. In particular, we design and optimize the GPU-based SSIM calculation, which is the first attempt to the best of our knowledge. SSIM is arguably one of the most important indicators for quantifying the reconstructed data quality, but it is also one of the most computationally expensive metrics. We highlight our utilization of GPU shared memory as a First-In First-Out (FIFO) buffer to maximize the data sharing of ghost regions between SSIM windows.
- We thoroughly evaluate our cuZ-Checker with various real-world scientific data. The results show that our solution can achieve 22.6–31.2 and 1.49–1.7 times speedups compared with the multithreading CPU and baseline metric-oriented GPU counterparts, respectively.

The rest of this paper is organized as follows. In Section II we introduce the background of lossy compression assessment and formulate the research problems. In Section III we present our design and optimizations for cuZ-Checker. We discuss evaluation results in Section IV and discuss related work in Section V. In Section VI we summarize our conclusions and briefly discuss future work.

II. Background and Problem Formulation

With the ever-increasing demand for lossy compressors for significantly reducing the extremely large volumes of data, Z-checker [13] was developed to comprehensively understand the lossy compression error, which is critical for users to judge compression quality accurately and thus select the best-fit compressor. Before Z-checker was developed, compressor developers and application users had to implement a battery of assessment metrics by themselves, a time-consuming and error-prone task especially for inexperienced users. In this section we provide background about Z-checker and formulate the research problem of our study.

Z-checker is a lossy compression assessment framework supporting over 20 compression-related metrics, including compression ratio, compression throughput, decompression throughput, and many metrics related to data distortion (i.e., compression error) such as PSNR and SSIM, Pearson correlation, and autocorrelation of compression errors.

Figure 1 illustrates the design architecture of Z-checker [13], summarized as follows:

- *Data Visualization Engine* is used to plot analysis results such as visualization of reconstructed data and errors.
- *Z-server* is an engine supporting the online web visualization of compression results.
- *Output engine* is used to parse the output results (including reconstructed data and compression analysis results) for other modules such as visualization.
- *Input engine* is used to load the data in different formats, such as binary data format, HDF5, and NetCDF.
- *CPU-based execution model* is the core module in charge of the execution and coordination of the analysis kernels and other modules.
- *CPU-based analysis kernel* is the other core module; it contains the algorithms and implementation of all the

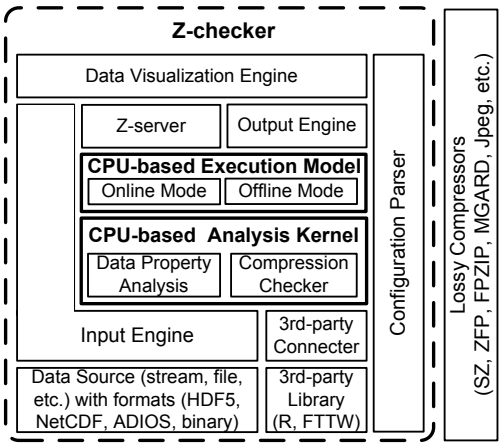


Fig. 1: The design architecture of Z-Checker.

analysis metrics including both data property analysis (such as data value range and entropy) and compression-related metrics (such as compression time, error distribution, PSNR, and SSIM).

- *Configuration parser* is used to load and parse the user's configuration, with regard to both assessment metrics and compression settings such as error bounds.

Figure 2 presents the design architecture of the cuZ-Checker. Compared with Z-checker's design, we propose GPU-compatible designs for the three critical modules—analysis kernel, module execution coordination, and configuration parser—in order to execute the assessment framework on GPU devices. Among them, the GPU analysis kernel is the core and raises the most challenges. Therefore, this paper will focus on the design and optimization of this module.

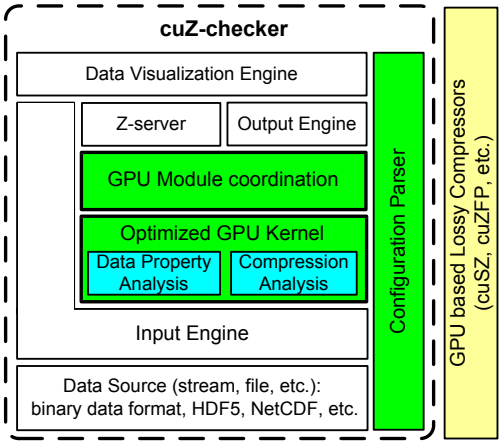


Fig. 2: The cuZ-Checker design architecture. We use different colors to highlight the modules that require GPU-compatible designs.

For the lossy compression assessment metrics (more specifically, the ones offered by Z-checker), we will explore the following topics: (1) What are the computational cores of the metrics? (2) For the computing-intensive metrics, how can we implement their computational cores on GPUs and optimize their performance? (3) How can we optimize the overall performance of cuZ-Checker, considering all the lossy compression evaluation metrics as a whole?

III. cuZ-Checker Design and Optimization

In this section we present our design of cuZ-Checker and describe how to optimize the performance of diverse patterns' GPU implementations.

A. Design Overview

Because of the substantial difference between CPU and GPU architectures, cuZ-Checker demands a new design as distinct from the CPU-based Z-checker model. The original CPU-based analysis kernel implements each analysis metric as an individual entity. Such a **metric-oriented design** is intuitive, allowing easy codebase maintenance and usage. It suffers from little data access overhead with the CPU because all the data can reside in the whole system's main memory. In contrast, applying the metric-oriented design to a GPU implementation would be inefficient because of the redundant high costs in data movement and memory access. With this feature in mind, the first critical design in our cuZ-Checker is an elaborate **(computational) pattern-oriented model**. That is, we classify all the analysis metrics into three categories based on the patterns of their computational cores, in order to ultimately fuse the GPU kernels and efficiently reuse the data during the computation. CUDA kernel fusion is crucial to the GPU performance because it can help avoid redundant kernel launches and data movements. Moreover, in the fused kernel the GPU memory access is minimized since one access can support multiple computations. For simplicity of the description, we mainly discuss the situation with three-dimensional input data in the following text, without loss of generality. In fact, all the design and optimization strategies for the 3D tenors can be easily extended to other dimensions (including 1D, 2D, and 4D).

In our design we exploit three computing patterns—*global reduction*, *stencil-like*, and *sliding window*—that cover all the computing-intensive metrics implemented by Z-checker. On the one hand, our careful performance profiling over the CPU Z-checker indicates that the expensive operations of all time-consuming metric implementations are 3D array reductions. On the other hand, we notice that many metrics have the same array-processing pattern, such that their implementations can be combined to enable CUDA kernel fusion with maximized data reuse.

The cuZ-Checker adopts a GPU module coordinator to handle the launch and execution of the metric calculations. Specifically, as cuZ-Checker is executed, the coordinator first identifies the category of the user-requested metrics and then invokes the corresponding optimized fused CUDA kernel to generate a series of compression measurement analysis data. We describe this design in the following text.

B. Pattern-Oriented Metrics Classification

Our cuZ-Checker aims to support 20+ assessment metrics, from among which 18 metrics use time-consuming

TABLE I: Pattern-oriented metrics classification

	Category I	Category II	Category III
pattern	global reduction	stencil-like	sliding window
metric	Min error, Max error, Avg error, Error PDF, Min pwr error, Max pwr error, Avg pwr error, Pwr error PDF, MSE, RMSE, NRMSE, SNR, PSNR	Derivatives, Divergence, Laplacian, Autocorre-lation	SSIM

3D array processing as their computational cores. As introduced in Section III-A, we classify these metrics into three categories—global reduction metric, stencil-like metric, and sliding window-based metric—according to their computational patterns and exhibit them in Table I. In what follows, we describe the calculations of some representative metrics in each category; other metrics in the same category follow similar styles.

1) Global Reduction Metrics: Each metric in this category performs a reduction from 3D array data to a scalar. For instance, min, max, and average errors are three such metrics (mainly calculating minimum value, maximum value, and summation, respectively), which indicate various types of differences between the original and the decompressed data. MSE calculates the mean squared error between the original and the decompressed data. It depends on a summation of the power of errors, which is a also reduction from 3D data to scalar. Figure 3 shows the GPU workflows of two illustrative

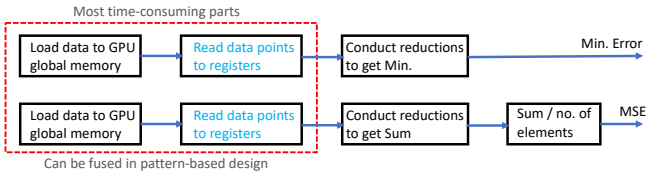


Fig. 3: Kernel fusing and data reuse for two pattern-1 metrics (Min. error and MSE). We use a red block to highlight the most time-consuming parts in their GPU workflows that can be fused.

global reduction metrics: avg. error and MSE. The metric classification allows us to process the two flows in one CUDA kernel. With this design, the most time-consuming data load and memory read (the steps in the red block in Figure 3) of two metrics can be fused, so that one data read from global memory to registers can feed two reductions.

2) Stencil-like Metrics: Derivative is a critical metric to assess how large the data varies in any position of the dataset. Such a metric can potentially cache the data changes in space and enlarge the errors introduced by lossy compression [26], making the compression developers or users understand the compression errors more easily. Our cuZ-Checker supports the computation of both the first- and second- derivatives for the original data vs. decompressed data. The first-order derivative can be calculated with the following formula:

$$Der(x,y,z) = |f(x{+}1,y,z){-}f(x{-}1,y,z)|{+}|f(x,y{+}1,z) - f(x,y-1,z)| + |f(x,y,z+1) - f(x,y,z-1)|, \quad (1)$$

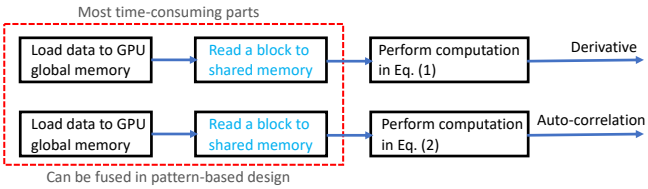


Fig. 4: Kernel fusing and data reuse for two pattern-2 metrics (derivative and autocorrelation). We highlight with a red block the fused heavy memory accesses in their GPU workflows.

where $Der(x,y,z)$ and $f(x,y,z)$ are the derivative and data value at location (x,y,z) of the 3D input data, respectively. The derivative computation is a 7-point 3D stencil, in which the computing at each data point requires the neighbor data points along all three dimensions. Besides, the Divergence and Laplacian metrics are the sums of first- and second-order derivatives, respectively.

Autocorrelation of compression errors is an important metric used to assess whether the compression errors are distributed randomly or not. This assessment metric is particularly useful for applications that require the compression errors to be uncorrelated or white noise [27]. Accordingly, our cuZ-Checker supports the autocorrelation metric for the compression errors. The autocorrelation metric can be mathematically calculated as follows:

$$AC(\tau) = \Big(\sum_{x=1}^{h-\tau}\sum_{y=1}^{w-\tau}\sum_{z=1}^{l-\tau}\big(\frac{1}{3}(e_{(x,y,z)}-\mu)((e_{(x+\tau,y,z)}-\mu)+(e_{(x,y+\tau,z)}-\mu)+(e_{(x,y,z+\tau)}-\mu))\big)\Big)/ne/\sigma^2, \quad (2)$$

where τ is a spatial gap; $e_{(x,y,z)}$ is the error value at (x,y,z); μ and σ^2 are the mean and covariance of e, respectively; (h,w,l) refers to the 3D shape of e; and $ne = (h-\tau)(w-\tau)(l-\tau)$ denotes the number of elements having been summed up. Similar to derivatives, computing the autocorrelation also follows a stencil-like pattern in which the value at any point depends on its right-hand neighbor points along all dimensions.

Figure 4 shows the GPU workflows of derivative and autocorrelation calculations. Classifying them in the same categories enables the fusing of the most time-consuming memory accesses, as indicated by the red block in Figure 4. We note that in contrast to the metrics in the first category that can read data points to registers, the stencil-like metrics require the blocked data cubes to be read into GPU shared memory, which will be explained in Section III-C.

3) Sliding Window-Based Metrics: SSIM is the only metric in this category, which has the most expensive computation compared with all other metrics. SSIM measures the structural similarity between the original and the decompressed data [25]. In Figure 5, we schematically illustrate how it works. For a 3D dataset, a small 3D sliding window is used to scan both datasets along all three dimensions with a fixed stride. The window size and stride are defined by the user. At each scan position, the reductions are performed to compute the window's min, max, sum, and power sum. After that, the

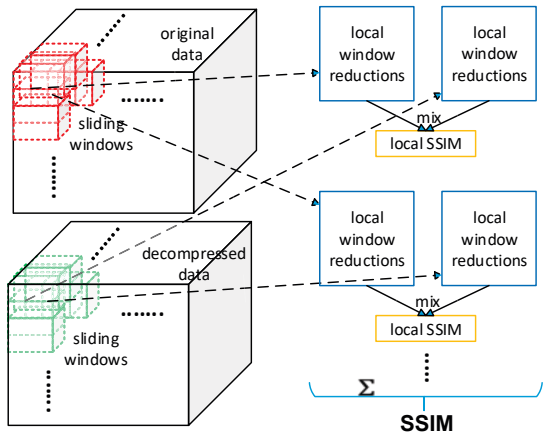


Fig. 5: Schematic illustration of 3D SSIM algorithm. The small 3D sliding window scans both original and decompressed data. At each scan position, the local reductions are performed within both windows, and then the reduction results are mixed to generate the window local SSIM. A global reduction is conducted at the end to output the final SSIM.

reduction results of the two datasets' corresponding windows are mixed (by some simple sums and multiplications) to generate the local SSIM. At the end of the entire scan, a global reduction is conducted to sum up all the local SSIMs to get the final SSIM result. For simplicity, we use scan positions and windows interchangeably in the rest of the paper.

We categorize SSIM as a unique metric since its computation conducts 3D reductions at two different levels (sliding window and global data), thus significantly increasing the computational burdens and requiring a GPU kernel that is different from the other categories. The ghost regions between neighbor windows also make SSIM distinct from other metrics, since their data sharing can largely affect the performance.

C. Pattern Implementations and Optimizations

In this subsection we describe our performance optimization strategies in detail. The fundamental idea for performance optimization is minimizing the writes and reads on both global and shared memories, since we observe that the performance of GPU-based 3D array processing is bounded by the number of memory accesses relative to the computation operations. Another optimization is on the calculation of SSIM, which demands optimized sharing of the ghost regions among the windows. To our best knowledge, no studies exist on how to accelerate the SSIM calculation over GPU. In what follows, we provide our design and optimization details of all the pattern implementations.

1) Pattern-1: global reduction: The efficient GPU global reduction heavily relies on warp-level shuffles [28]. Starting from CUDA 9, NVIDIA provides warp-level primitives including shuffle operations with explicit intra-warp thread-level synchronization [29]. In shuffles, for example, __shfl_up_sync, __shfl_down_sync, and __shfl_xor_sync, a participated thread can directly read a register of another thread in the same warp.

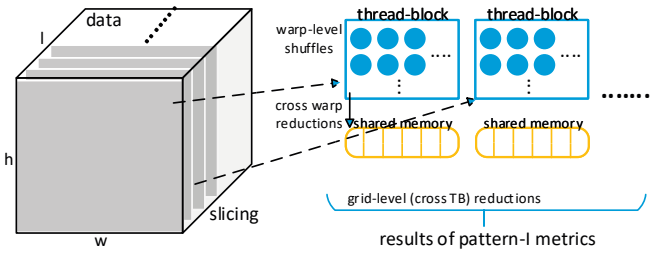


Fig. 6: GPU implementation for pattern-1 metrics. The data is divided into slices, and each slice is assigned to a thread block. The global reduction is accomplished via warp-level shuffles, cross-warp shared-memory-based reductions, and cross-thread-block grid-level reductions.

These primitives enable warp-level collective data exchanging or broadcasting without involving shared memory.

The state-of-the-art GPU-based reduction implementations, such as CUB [30], are highly optimized but not flexible enough to realize the customized reductions in our pattern-1 design. Therefore, we developed our own CUDA kernel for pattern-1 metric calculations, as illustrated in Figure 6. Assuming the 3D data shape is (h, w, l), we divide the data into 2D slices along the z-axis and assign each slice to a CUDA thread block. Hence, l thread blocks are created in the grid. The threads in each block are organized as a two-dimensional layout in which the x-dimension (blockDim.x) is set to warp size (32 in CUDA). Each block performs the warp-level and cross-warp reductions on its corresponding slice in an iterative manner along both the x-axis and y-axis. Then a grid-level reduction is performed to generate the results of pattern-1 metrics.

We present the pseudocode of the CUDA kernel for the pattern-1 calculation in Algorithm 1. As shown in the pseudocode, after the kernel is launched, each thread performs intra-thread reductions by reading the global memory data iteratively with the stride of thread-block dimensions along both axes (ln.4-6). The results are held in each thread's registers. Then, the warp-level reductions are conducted by leveraging shuffle operations (ln.7-8). The intermediate reduction results are stored in the first thread of every warp. They should be written into shared memory (ln.9) for the cross-warp reductions (ln.11-15). After that, the first thread of each thread-block has the reduction results of the corresponding slice. These results are written to the global memory (ln.16) to be used for the grid-level reduction (ln.18-22) to yield the final global reduction results (ln.23). Notice that we leverage cooperative groups (ln.2), also introduced in CUDA 9 [31], to achieve in-kernel grid synchronization (ln.17) to avoid additional kernel launch costs. The corner case handlings at the edges are straightforward; we omit the discussions of them because of space limits.

In Algorithm 1, the memory access overhead has been minimized. The $reduce()$ in Algorithm 1 can concurrently handle multiple reduction operations, thus each memory access can be used for processing all pattern-1 metrics. This memory access fusion is enabled thanks to our pattern-oriented design.

2) Pattern 2: stencil-like: In pattern 2, the processing of each data point requires its neighbor points along all three

Algorithm 1: CUDA kernel for pattern-1

```
1  Kernel PATTERN-1 (d_data, d_out)
2      grid = cg::this_grid();                          ▷ cooperative_groups
3      tidx←threadIdx.x; tidy←threadIdx.y; bidx←blockIdx.x;
       /* reductions in each individual thread. */
4      for i=tidx; i<h; i+=blockDim.x do
5          for j=tidy; j<w; j+=blockDim.y do
6              reg ⟵ reduce(d_data[i, j, bidx]);
       /* warp-level reductions. */
7      for offset=warpSize/2; offset>0; offset/=2 do
8          reg ⟵ reduce(shfl_down_sync(mask, reg, offset));
9      if tidx=0 then shared[tidy] ⟵ reg;       ▷ write to shared memory
10     __syncthreads;
       /* cross-warp reductions. */
11     if tidy=0 then
12         if tidx<blockDim.y then reg ⟵ shared[tidx];
13         mask = ballot_sync(mask, tidx<blockDim.y);
14         for offset=warpSize/2; offset>0; offset/=2 do
15             reg ⟵ reduce(shfl_down_sync(mask, reg, offset));
16         if tidx=0 then d_out[bidx] ⟵ reg;  ▷ write to global memory
17     cg::sync(grid);                                  ▷ sync the entire CUDA grid
       /* cross-threadblock reductions. */
18     if bidx=0 then
19         for i=tidx; i<l; i+=blockDim.x do
20             reg ⟵ reduce(d_out(i));
21         for offset=warpSize/2; offset>0; offset/=2 do
22             reg ⟵ reduce(shfl_down_sync(mask, reg, offset));
23         if tidx=0 then result ⟵ reg;     ▷ output final reduction result
```

dimensions. Holding the read data in registers and using shuffles are insufficient to handle this pattern, since the warp-level operations can provide neighbor access in only one dimension. Therefore, registers have to offload the read data into shared memory for data accesses in the other two dimensions.

No off-the-shelf GPU autocorrelation is available; and the existing GPU derivatives, such as [32], are not flexible enough to support our fused kernel. Thus, we implement our pattern-2 computation on the GPU as described in Figure 7 and Algorithm 2. We leverage the blocking approach used in GPU stencil implementations [33] and divide the data of pattern-2 into 3D cubes (or 3D blocks), as shown in Figure 7. Such a blockwise scheme can leverage data locality in pattern 2 better than does the slicing (which is utilized in pattern 1). As indicated in Fig. 7, different planes along the z-axis are assigned to different thread-blocks. In each plane, the cubes are iteratively read into GPU shared memory and then processed by the corresponding thread-block. Algorithm 2 presents the CUDA kernel for pattern 2. The input $stride$ is the distance between the current data point and its neighbor point. It can represent both the derivative's order and the autocorrelation's spatial gap. It is also the overlap length of the adjacent cubes. In each iteration (ln.4-5), a cube is loaded by the thread-block from global memory to shared memory (ln.6-8). Subsequently, each thread of the thread-block (except the ones in the $stride$-wide borders) (ln.10) first computes the derivative at its location (ln.11-16), then calculates the autocorrelation of its corresponding data point (ln.17-18), both by reading neighbor data along all three dimensions from shared memory. We note that a reduction of intermediate results at all data points is required to yield the final autocorrelation and divergence/Laplacian (if opted

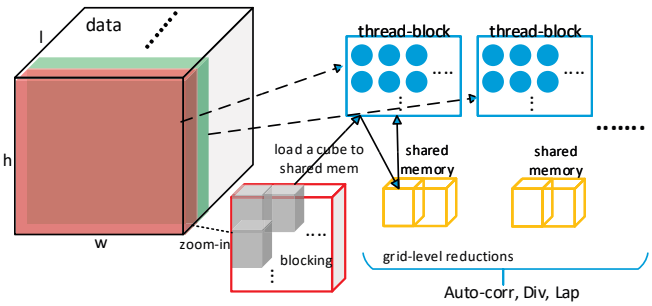


Fig. 7: GPU implementation for pattern-2 metrics. Since the processing of current data points requires the neighbors along all three dimensions, the data is blocked to small cubes that can be read into GPU shared memory.

in) (ln.18). The $reduce$ operation here includes warp-level, cross-warp, and cross-block reductions. We omit the detailed description of how the corner cases at the edges along all axes are processed, since they are straightforward to deal with.

Algorithm 2: CUDA Kernel for Pattern-2

```
1  Kernel PATTERN-2 (d_data, μ, ssize, stride) ▷ ssize is the side length
       of the cube; stride is the distance between current point and its neighbor
2      tidx←threadIdx.x; tidy←threadIdx.y; bidx←blockIdx.x;
3      ssize' = ssize - stride; k = bidx * ssize';
       /* in each iteration of the second loop, a cube is processed. */
4      for i=0; i<h-stride; i+=ssize' do
5          for j=0; j<w-stride; j+=ssize' do
               /* the corresponding cube is load to shared memory. */
6              for s=0; s<ssize; s++ do
7                  shared[tidx, tidy, k] ← d_data[i+tidx, j+tidy, k+s];
8              __syncthreads;
               /* read data from shared memory for computings. */
9              for s=0; s<ssize'; s++ do
10                 if tidx and tidy < ssize' then
                       /* compute derivatives. */
11                     x ← tidx+stride/2; y ← tidy+stride/2;
12                     z ← s+stride/2;
13                     dx ⟵ (shared[x+stride/2, y, z] -
                         shared[x-stride/2, y, z]) / 2;
14                     dy ⟵ (shared[x, y+stride/2, z] - shared[x,
                         y-stride/2, z]) / 2;
15                     dz ⟵ (shared[x, y, z+stride/2] - shared[x, y,
                         z-stride/2]) / 2;
16                     Der[i+tidx, j+tidy, k+s] ⟵ sqrt(dx², dy²,
                         dz²);          ▷ output derivative of current point
                       /* compute auto-correlations. */
17                     sum ⟵ (shared[tidx+stride, tidy+stride,
                         s+stride] - μ) ∗ (shared[tidx, tidy, s] - μ);
18     Div,Lap,corr ← reduce(Der,sum); ▷ required to yield final autocorr
```

To summarize, the kernel fusion and data reuse are enabled in pattern 2 because of our pattern-oriented design. This CUDA kernel implementation allows computing multiple metrics with a single kernel launch. Furthermore, in the kernel, one loading of a data point from global memory to shared memory can serve the calculations of all pattern-2 metrics.

3) Pattern 3: sliding window: Compared with the first two patterns, pattern-3's algorithm exposes significantly heavier computation because of the local reductions in each SSIM window. Moreover, since the sliding step length usually is small (mostly just 1), substantial overlaps exist between adjacent windows. Hence, data-sharing efficiency among the windows

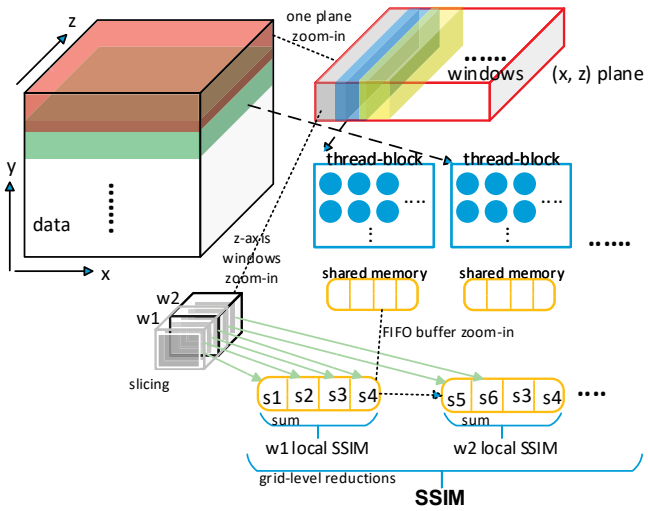


Fig. 8: GPU implementation for pattern-3 metric (SSIM). In this design, each (x,z) plane is assigned to a thread block. In a plane, the windows along the z-axis are further sliced to utilize shared memory as FIFO buffers for ultimately avoiding redundant ghost region processings.

can significantly affect the performance of pattern 3.

To the best of our knowledge, no previous work exists about GPU-based SSIM calculation. Thus, we develop a new GPU kernel to calculate the SSIM and also optimize its performance. As demonstrated in Figure 5, the calculation of multidimensional SSIM needs to simultaneously slide the windows in both the original and decompressed datasets. At each sliding step, the key operation is the local window reduction which is algorithmically identical for both datasets' windows; performing mix to get the local SSIM thereafter is simple. Therefore, to be concise, in the following text we mainly describe the GPU local window reduction design for one (original) dataset. The local window reductions calculation for the decompressed dataset is exactly the same.

Figure 8 illustrates our design using only the original dataset as an example. In our design we assign one (x,z) plane to a thread block. Each plane contains several windows along the y-axis and all windows along the x- and z-axes. In a thread-block, each thread handles the processing of one window, so the ghost regions between adjacent windows along the x-axis can be ultimately shared by using warp-level shuffles. We note that shuffle operations allow stride thread access; thus this design supports arbitrary sliding step length. Furthermore, in a plane, we slice the windows along the z-axis and then utilize the GPU shared memory as a **FIFO buffer** to maximize data sharing between windows along this dimension, as shown in Fig. 8. This FIFO buffer design allows the current window to reuse the intermediate ghost region results of the preceding window along the z-axis. Thus each slice needs to be read and processed only once.

Algorithm 3 presents the pseudocode of the CUDA kernel for our pattern-3 implementation. In each thread-block, the corresponding plane is further divided into slices along the z-axis that will be iteratively processed (ln.6). In an iteration, each thread reads one data point of the current slice from global memory to its register. Then, the threads within the $mask$ (ln.9) handle one window per thread, so a total of *xNum·yNum* windows can be processed simultaneously. These threads perform window local reductions first along the x-axis by leveraging warp-level shuffles to read the registers of neighbor threads and then along the y-axis through shared memory (ln.10-12; see warp-level and cross-warp reduction details in Algorithm 1). These intermediate reduction results of the current slice are subsequently stored into the FIFO buffer (ln.15) before the kernel moves to the processing of the next slice. After the $wsize$th ($wsize$ is the window's side length along z-axis) slices are processed (ln.16), the results in the FIFO buffer are reduced and mixed to yield the local SSIMs of the first *xNum·yNum* 3D windows (ln.17-19). Thereafter, the kernel processes the ($wsize$+1)th slice likewise and stores the intermediate results in the FIFO buffer by overwriting the first slice's buffered data (ln.15). The kernel keeps iterating until the ($wsize$+$step$)th slice (ln.16) has been processed. After that, the results in the FIFO buffer are reduced and mixed again to generate the second set of *xNum·yNum* windows' local SSIMs (ln.17-19). At the end of the entire loop, a grid-level reduction of all local SSIMs is conducted to get the final SSIM result (ln.20). Notice that corner cases at the edges happen only along the x- and y-axes and are straightforward to be handled.

Algorithm 3: CUDA kernel for pattern-3

```
1  Kernel PATTERN-3 (d_data1, d_data2, wsize, step)   ▷ wsize is the
     length of a window side while step is the sliding step length
2    tidx←threadIdx.x; tidy←threadIdx.y; bidx←blockIdx.x;
3    j = bidx * yNum;   ▷ j is the base index of data and yNum is the
       number of windows, both along y-axis and for current threadblock
4    xNum = warpSize - wsize + step;
     /* i iterates along x-axis while k iterates along z-axis. */
5    for i=0; i<h; i+=xNum do
6      for k=0; k<l; k++ do
7        d1 ← d_data1[i+tidx, j+tidy, k];
8        d2 ← d_data2[i+tidx, j+tidy, k];
9        mask = ballot_sync(mask, tidx<xNum);
         /* window local reductions on the current slice. */
10       for offset=1; offset<wsize; offset++ do
11         w1 ←reduce(shfl_down_sync(mask,d1,offset));
12         w2 ←reduce(shfl_down_sync(mask,d2,offset));
13       if tidx%step = 0 and tidx < xNum and tidx%step = 0
           and tidy < yNum then
14         wx = tidx/step; wy = tidy/step;
           /* store current slice's results into FIFO buffer. */
15         shared[wx, wy, k%wsize] ⟵ w1, w2;
           /* if the current slice is the end of windows, reduce the
              entire buffer data to yield local window SSIMs. */
16         if (k+1-wsize) % step = 0 then
17           for s=0; s<wsize; s++ do
18             w1', w2' ← reduce(shared[wx, wy, s]);
19           wssim[i+wx, j+wy, (k+1-wsize)/step] ⟵
               calw(w1', w2');   ▷ get local window SSIMs
20   SSIM ⟵ reduce(wssim); ▷ reduce window results to get final SSIM
```

To summarize, our implementation achieves perfect data sharing along the z-axis, thanks to our FIFO buffer design. With this design, each slice only needs to be read from global memory and processed once. Our implementation also

ultimately leverages the x-axis data sharing because of the utilizing of warp-level shuffles. Most data points along the x-axis need to be processed just once; only slightly repeated data point processings exist between the x-axis iterations (ln.5). The data sharings along the y-axis are also partially leveraged since we assign multiple windows on this dimension to a thread-block. The y-dimensional data processing repetitions only happen between adjacent thread blocks. Generally speaking, with our GPU kernel design, for most data points, the pattern-3 (SSIM) calculation reads them from global memory and processes them just once. The shared memory is only minimally used by the reductions along the y-axis (cross-warp reductions) and z-axis (FIFO buffer).

IV. Performance Evaluation

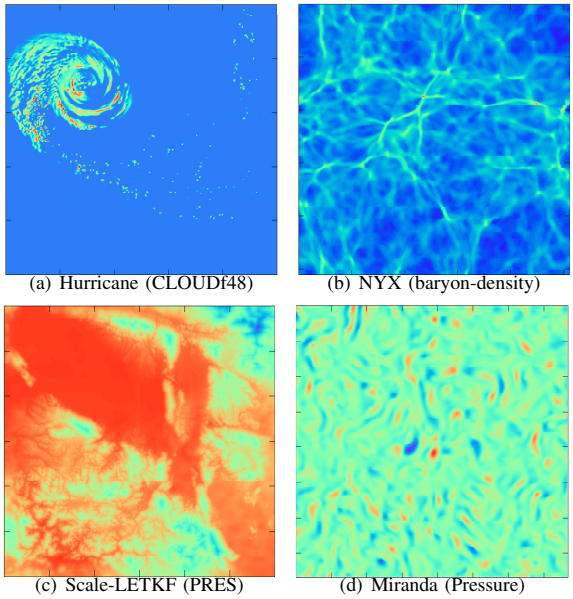
In this section we thoroughly evaluate our cuZ-Checker using a GPU node of Argonne National laboratory's Lambda cluster[1]. This machine is configured with 512 GB of system main memory, Intel Xeon Gold 6148 CPU, and NVIDIA Tesla V100 GPU with CUDA driver version 11.2. Xeon Gold 6148 CPU has 20 physical cores and a 27.5 MB L3 cache, running at 2.40 GHz base frequency. V100 GPU is built on NVIDIA's Volta microarchitecture. It has 80 streaming multiprocessors (SMs) and 32 GB GPU global memory. Each SM has 64 CUDA cores (total of 5,120 cores device-wide), 48 KB on-chip shared memory, and 64K registers.

A. Datasets and Their Characteristics

We evaluate our cuZ-Checker by using it to measure the cuSZ [20] lossy compressor based on the simulation datasets generated by four well-known real-world scientific applications, including (a) Hurricane ISABEL simulation [34], which simulates the most intense hurricane in the 2003 Atlantic hurricane season; (b) NYX cosmology simulation [35], which solves equations of compressible hydrodynamics flow describing the evolution of baryonic gas coupled with an N-body treatment of the dark matter in an expending universe; (c) Scale-LETKF weather simulation [36], which performs real-time, high-resolution, short-term prediction of heavy rainfall systems; and (d) Miranda turbulence simulation [37], a radiation hydrodynamics code designed for large-eddy simulation of multicomponent flows with turbulent mixing.

Each dataset contains different data patterns and features. Specifically, (a) Hurricane consists of 13 data fields (e.g., QCLOUD and temperature); each field is a $100\times500\times500$ 3D data. (b) NYX includes six fields, such as dark matter density and baryon density, and each field is a 512^3 3D array. (c) Scale-LETKF has six data fields with $98\times1200\times1200$ elements per field. (d) Miranda contains severn different fields, each being a $256\times384\times384$ 3D vector. All data fields are stored in single precision. One illustrative field of each dataset is visualized in Figure 9. All these scientific datasets can be downloaded from the SDRBench [38], [39].

[1]https://collab.cels.anl.gov/display/LCD



(a) Hurricane (CLOUDf48) (b) NYX (baryon-density)
(c) Scale-LETKF (PRES) (d) Miranda (Pressure)

Fig. 9: Visualization of diverse application datasets.

B. Overall Performance Analysis

We have verified that our cuZ-Checker has the correct calculation on all assessment metrics by comparing it with the Z-checker's output. For example, with first field of the Hurricane dataset, both cuZ-Checker and the CPU-based Z-checker yield 2.8×10^{-9} as the first-order derivative result.

We compare the performance of our cuZ-Checker (abbreviated as cuZC) with an OpenMP-based CPU counterpart (abbreviated as ompZC) and a metric-oriented GPU counterpart (abbreviated as moZC). OmpZC is the multithreading version of the original Z-checker [13]. MoZC is our straightforward CUDA implementation of Z-checker following the conventional metric-oriented design principle. We develop moZC mainly for demonstrating the advantage of our patten-oriented design. MoZC implements each metric as an individual CUDA kernel. In moZC, we leverage NVIDIA's CUB library [30] to achieve reductions for each pattern-1 metric. We note that we do not utilize the CUB library in our cuZC since its APIs are not flexible enough to express user-defined reductions [40] (e.g., the fused reductions in our pattern-1 implementation). We separately implement autocorrelation and derivative metrics following NVIDIA's approach [32]. Since SSIM has no GPU implementation in the literature, in order to evaluate our cuZC's data-sharing efficiency, the counterpart implementation in moZC adopts an approach similar to that introduced in Section III-C3 but without the FIFO buffer.

Figure 10 shows cuZC's overall performance improvements compared with that of both ompZC and moZC. Its x-axis represents different datasets while the y-axis displays the speedups (in log scale) of cuZC relative to both counterparts. In this experiment we measure the entire executions of the assessment systems with all metrics enabled. For the derivative metric, we calculate both first-order and second-order derivatives. For autocorrelation, we allow the spatial gap to be up

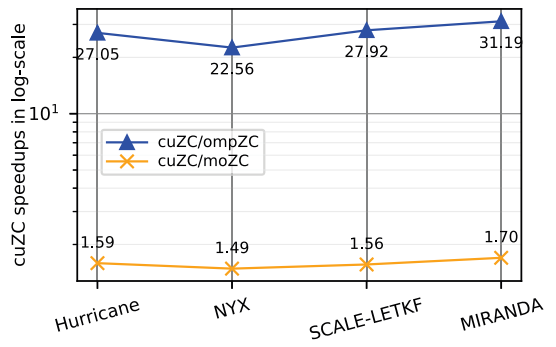


Fig. 10: Overall performance comparisons of cuZC with both ompZC and moZC. The x-axis represents different datasets while the y-axis indicates cuZC's speedups in log scale.

to 10. For SSIM, we set the sliding-window size to 8 on each side and the step length to 1. Each dataset involves many fields (e.g., 13 fields for Hurricane and 6 fields for NYX). We show the average performance calculated over all fields for each dataset in Figure 10. As shown in the figure, compared with multithreading CPU-based ompZC, our cuZC can achieve at least 22.6× and up to 31.2× speedups. Moreover, it exhibits 1.5–1.7× better performance than the does the metric-oriented GPU-based moZC. The overall performance of our cuZC is dominated by the most computing-intensive metrics (e.g., SSIM). If we need our framework to assess only part of the supported metrics, the performance improvement potentially could be larger, as we detail in the next subsection.

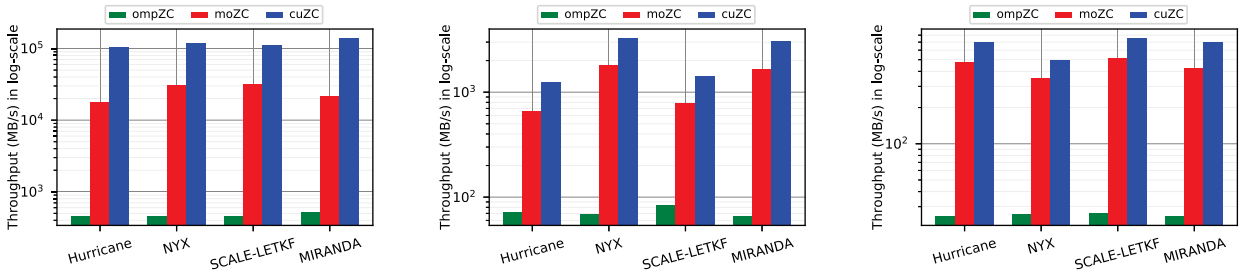
C. Performance Breakdown Analysis

In this section we perform a breakdown performance analysis for the time-consuming pattern metrics listed in Table I. Figure 11 presents the throughput of all three frameworks (ompZC, moZC, and cuZC) with only pattern-1 metrics (Figure 11(a)), pattern-2 metrics (Figure 11(b)), or pattern-3 metrics (Figure 11(c)) enabled, respectively. From these figures, we can observe that our cuZC always yields much higher throughput than the other two frameworks do. We also observe that from among the three patterns, the throughputs on pattern-1 are significantly higher than the throughputs on pattern-2 and pattern-3. Specifically, as the dataset changes, cuZC can achieve a throughput of 103–137 GB/s while moZC can reach only 17–31 GB/s and ompZC can reach only 0.44–0.51 GB/s. The reason for the high throughput of pattern 1 is that pattern-1 requires much fewer computations than the other patterns do, as described in Section III. On the other hand, we observe that all three frameworks exhibit the lowest throughputs on pattern 3 (i.e., SSIM), since it yields the heaviest computation burden. For this pattern, our cuZC has 497–758 MB/s throughput while moZC and ompZC have 351–514 and 24.8–26.6 MB/s throughputs, respectively. The throughput on pattern 3 clearly dominates the overall framework performance.

To better demonstrate cuZC's performance improvement on different patterns, we also plot the speedups of cuZC over ompZC and moZC in Figure 12, which clearly shows that our cuZC always exhibits best performance. In Figure 12(a), we observe that cuZC can achieve 227×-268× speedups to ompZC on pattern 1. These speedups are remarkably higher than the overall speedups in Figure 10, especially because pattern-1's implementation in cuZC requires the fewest operations in thread iterations compared with the other two patterns. Moreover, it is capable of thoroughly leveraging advanced GPU features, such as warp-level primitives, to optimize the performance. We can also see that cuZC achieves a speedup of 3.49×-6.38× over moZC on pattern 1. We note that in Table I, RMSE's and NRMSE's computational cores essentially are MSE while PSNR's core is SNR. Therefore, moZC contains 10 CUDA kernels for pattern 1, and cuZC's speedup upper bound is 10. However, the additional resource usages and branching overhead lower the actual speedup. Figure 12(b) shows that on pattern 2 our cuZC can achieve 17.1×-47.4× and 1.79×-1.86× speedups over ompZC and moZC, respectively. Since Divergence and Laplacian in Table I are the summations of order-1 and order-2 derivatives, respectively, moZC implements two CUDA kernels for pattern 2. cuZC's speedups over moZC on pattern 2 are close to 2, indicating that our kernel fusion and data reuse are efficient with only slight branching overhead. Figure 12(c) shows that our cuZC attains 19.2×–28.5× and 1.42×–1.63× speedups to ompZC and moZC on pattern 3 (SSIM). It indicates our FIFO buffer design in cuZC can successfully improve the performance by around 50%.

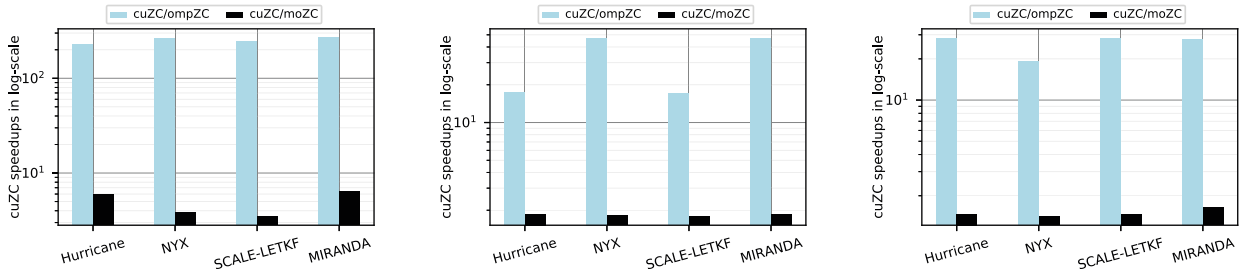
Takeaway 1: Our cuZC shows different speedups with different categories of metrics. For pattern 1, the speedups are significantly higher than the overall speedups since its kernel fully utilizes advanced GPU features and computes 14 metrics within one kernel launch. For pattern 2, cuZC shows a nearly twofold speedup compared with moZC, indicating that our kernel fusion is efficient with tiny overhead. For pattern 3, our FIFO buffer design achieves ∼50% performance improvement.

To gain insights about how different datasets influence the assessment performance, we profile our cuZC during run time and display the results in Table II. For each pattern, we present the register usage per thread block (Regs/TB), the shared-memory usage per thread block (SMem/TB), the total iterations in each thread (Iters/thread), and the number of thread-blocks assigned to each streaming multiprocessor along with the number of thread-blocks among them that can be concurrently handled (TB(cncr.)/SM), with different datasets. During the execution, an active thread-block (TB) will exclusively reserve a portion of its corresponding SM's computation resources, including registers and shared memory, based on the request. Once a SM has enough idle resources available, the TBs assigned to it can be scheduled to execute concurrently. The SM occupancy will be maximized when all assigned TBs concurrently execute. Therefore, the amount of resources requested by a TB largely affects the overall performance. Furthermore, Iters/thread represents the sequentially executed workloads in each thread, which can also impact the GPU performance. Regs/TB and SMem/TB are determined by the computation pattern, while Iters/thread



(a) Pattern-1 (b) Pattern-2 (c) Pattern-3

Fig. 11: Throughput of cuZC, moZC, and ompZC performing (a) pattern-1 metrics, (b) pattern-2 metrics, and (c) pattern-3 metrics with all four datasets.



(a) pattern-1 (b) pattern-2 (c) pattern-3

Fig. 12: Speedups (in log scale) of our cuZC over ompZC and moZC when executing the three groups of metrics separately with all four datasets.

and TB(cncr.)/SM can be affected by the datasets. Several remarkable performance varying with the dataset change can be observed in Figure 12. (i) In Figure 12(a), cuZC shows significantly less speedup with NYX and Scale-LETKF to moZC. The reason is that these two datasets have larger data sizes resulting in more TBs per SM (seven) or more iterations per thread (6.3k), as shown in Table II. Since cuZC processes 14 pattern-1 metrics in one kernel, the register usage of a TB is big, which limits the concurrent TBs in a SM to at most four (64k/14k). Consequently, with NYX, a SM needs two rounds of execution to process all associated thread-blocks. (ii) In Figure 12(b), cuZC shows much less speedup than does ompZC with Hurricane and Scale-LETKF. The reason is that they yield at most one thread-block on each SM. The number of TBs in pattern 2 is decided by the z-axis size; both two datasets have short lengths along this dimension (100 and 98) and thus have small numbers of TBs. With only one thread block on a SM, there is no space to hide the latency, leading to less efficient GPU utilization. (iii) In Figure 12(c), compared to cuZC with other datasets, cuZC with NYX achieves less speedup than does ompZC. This is because in our FIFO buffer design, the data length along the z-axis determines Iters/thread in the pattern-3 (SSIM) calculation. Since NYX has the biggest length along the z-axis (512), it requires more iterations in each thread resulting in less speedup than the other datasets. We emphasize that Table II intends to demonstrate dataset impacts on cuZC's performance with the same pattern. Any cross-pattern comparisons of Iters/thread or TBs/SM are not meaningful because, with different patterns, an iteration has quite different instructions and workloads.

Takeaway 2: With the same pattern, changing datasets can affect our cuZC's speedup. Specifically, the size and shape of a dataset determine the number of thread-blocks per multiprocessor and the number of iterations per thread, which can potentially impact GPU performance. With pattern 1, small datasets exhibit higher cuZC speedup compared with ompZC. With pattern 2, a more balanced data shape (cube vs. cuboid) can lead to a better cuZC performance. With pattern 3, a dataset having shorter length along the z-axis can make cuZC achieve higher speedup.

TABLE II: cuZC Runtime Profiling

	Regs/TB	SMem/TB	Iters/thread	TB(cncr.)/SM
			Pattern-1	
Hurricane	14k	0.4KB	977	2(2)
NYX	14k	0.4KB	1k	7(4)
SCALE-LETKF	14k	0.4KB	6.3k	2(2)
MIRANDA	14k	0.4KB	576	4(4)
			Pattern-2	
	Regs/TB	SMem/TB	Iters/thread	TB(cncr.)/SM
Hurricane	2.3k	17KB	205	1(1)
NYX	2.3k	17KB	205	2(2)
SCALE-LETKF	2.3k	17KB	1.1k	1(1)
MIRANDA	2.3k	17KB	89	2(2)
			Pattern-3	
	Regs/TB	SMem/TB	Iters/thread	TB(cncr.)/SM
Hurricane	11k	16KB	1.8k	2(2)
NYX	11k	16KB	8.7k	2(2)
SCALE-LETKF	11k	16KB	3.4k	2(2)
MIRANDA	11k	16KB	2.9k	2(2)

V. Related Work

In this section we first discuss the existing lossy compression assessment systems and then describe current GPU-based implementations and optimization techniques related to the three patterns in our cuZ-Checker.

To the best of our knowledge, none of the existing lossy compression assessment platforms or libraries support GPU accelerators, which leaves a significant gap for comprehensively assessing the emerging GPU-based lossy compressors such as cuSZ [20] and cuZFP [21]. Z-checker [13] is the first scientific lossy compression assessment toolkit/library, which includes many compression-related metrics such as distribution of errors, MSE, PSNR, derivatives, autocorrelation, and SSIM. QCAT [41] is a lightweight compression analysis toolkit, which focuses on easy-to-use compression data analysis. Foresight [14] is a generic framework for analysis and visualization of lossy compression errors. It does not provide as compression assessment metrics as comprehensive as Z-checker requires, while it offers better visualization of simulation data and compression errors.

CUDA warp-level shuffles with either implicit (before CUDA 9) or explicit warp synchronization have been broadly used to improve the performance for a variety of applications, including sequence alignments [42], sparse matrix computations [43]–[47], string matchings [48]–[52], graph traversals [53]–[55], and data scans [56]–[59]. Some works also propose advanced optimizations for GPU reductions. Reddy et al. [40] design language constructs allowing arbitrary reductions to be easily expressed on user-defined data types. Jradi et al. [60] employ techniques such as loop unrolling and persistent threads to implement the generic GPU reductions. De Gonzalo et al. [61] provide a new set of high-level APIs for domain-specific languages (DSLs) to easily generate warp shuffle instructions and atomic instructions. Navarro et al. [62] levarage GPU tensor cores to boost the reduction performance.

GPU-based stencil computations have been intensively studied. Zhang et al. [63] develop autogeneration and optimization techniques to autotune 3D stencil computations on GPUs. Rawat et al. [64] discuss a DSL that can generate effective tiled code for GPUs stencils. Anjum et al. [65] present a new GPU-based technique that uses 2D caching to efficiently implement 3D stencil computations. Matsumura et al. [66] propose AN5D, an automated framework that can transform C stencil code to optimized CUDA stencil code. Oh et al. [67] present GOPipe, a programming framework for efficient pipelined stencil executions on GPUs that can automatically find task granularity and dynamically schedule tasks of it.

Although no existing work studied GPU-based 3D SSIM, the sliding-window algorithm has been implemented on GPUs for some other applications. Green et al. [68] propose a GPU-based algorithm for merging two sorted arrays that requires window sliding. Krizhevsky [69] implements CNN on GPUs with direct convolution approach that utilizes the filter as a sliding window to scan the input feature map. De Matteis et al. [70] present an autotunable general sliding-window operator for streaming systems. Luo et al. [71] utilize sliding-window exponentiation to optimize a GPU side-channel timing attack of the RSA cryptosystem. Cooke et al. [72] optimize the sliding-window algorithm in general on various platforms including GPUs.

VI. Conclusion and Future Work

This work presents cuZ-Checker, a GPU-based assessment system to measure GPU-based lossy compressors' compression quality and performance. In our design, we proose a patten-based approach to maximizes the opportunity for GPU kernel fusion and data reuse. For each pattern, we provide fine-grained design and optimization to its CUDA kernel by leveraging various GPU features. We comprehensively evaluate our cuZ-Checker with four real-world scientific datasets and obtain the following key findings:

- Our cuZ-Checker (cuZC) can achieve 22.6–31.2× overall speedup over the OpenMP-based multithreading CPU baseline (ompZC) and 1.49–1.7× overall speedup compared with a GPU counterpart designed using a metric-oriented approach (moZC).
- Our cuZC exhibits different performances when executing metrics in different categories. With pattern-1 metrics, cuZC shows more than 227× and 3.5× speedups over ompZC and moZC, respectively. With pattern-2 metrics, cuZC shows nearly 2× speedup over moZC. With pattern-3 metrics, FIFO buffer design in cuZC improves the performance by around 50%.
- With each pattern, the dataset's size and shape can also affect cuZC's performance, since they can determine the GPU's number of thread block per multiprocessor and number of iterations per thread.

In our future work, we will extend our cuZ-Checker to a multi-node multi-GPU environment, in terms of the fundamental single-GPU performance optimization that has been solved in this paper. Many scientific applications can yield exascale datasets that far beyond the capacity of any single GPU. Accordingly, both multi-GPU lossy compressor and assessment system are desired. Similar to other multi-GPU implementations [73], [74], the multi-GPU version of cuZ-Checker needs fine-grained design of inter-GPU synchronization and communication to optimize the performance. We also plan to incorporate cuZ-Checker with cuSZ to make the assessment more seamless.

Acknowledgments

This research was supported by the Exascale Computing Project (ECP), Project Number: 17-SC-20-SC, a collaborative effort of two DOE organizations – the Office of Science and the National Nuclear Security Administration, responsible for the planning and preparation of a capable exascale ecosystem, including software, applications, hardware, advanced system engineering and early testbed platforms, to support the nation's exascale computing imperative. The material was supported by the U.S. Department of Energy, Office of Science, under contract DE-AC02-06CH11357, and supported by the National Science Foundation under Grant OAC-2003709, OAC-2042084, OAC-2104023, and OAC-2104024. We acknowledge the computing resources provided on Bebop (operated by Laboratory Computing Resource Center at Argonne) and on Theta operated by Argonne Leadership Computing Facility.

REFERENCES

[1] T. E. Fornek, "Advanced Photon Source Upgrade Project preliminary design report," 9 2017.

[2] SLAC National Accelerator Laboratory, "Linac Coherent Light Source (LCLS-II)," https://lcls.slac.stanford.edu/, 2017, online.

[3] F. Cappello, S. Di, S. Li, X. Liang, G. M. Ali, D. Tao, C. Yoon Hong, X.-c. Wu, Y. Alexeev, and T. F. Chong, "Use cases of lossy compression for floating-point data in scientific datasets," *International Journal of High Performance Computing Applications (IJHPCA)*, vol. 33, pp. 1201–1220, 2019.

[4] M. Burtscher and P. Ratanaworabhan, "FPC: A high-speed compressor for double-precision floating-point data," *IEEE Transactions on Computers*, vol. 58, no. 1, pp. 18–31, Jan 2009.

[5] Zlib, https://www.zlib.net/, online.

[6] L. P. Deutsch, "GZIP file format specification version 4.3," 1996.

[7] BlosC compressor, http://blosc.org/, online.

[8] Y. Collet, "Zstandard – real-time data compression algorithm," *http://facebook.github.io/zstd/*, 2015.

[9] S. Di and F. Cappello, "Fast error-bounded lossy HPC data compression with SZ," in *IEEE International Parallel and Distributed Processing Symposium*, 2016, pp. 730–739.

[10] P. Lindstrom, "Fixed-rate compressed floating-point arrays," *IEEE Transactions on Visualization and Computer graphics*, vol. 20, no. 12, pp. 2674–2683, 2014.

[11] N. Sasaki, K. Sato, T. Endo, and S. Matsuoka, "Exploration of lossy compression for application-level checkpoint/restart," in *IPDPS 2015*, 2015, pp. 914–922.

[12] A. H. Baker, H. Xu, J. M. Dennis, M. N. Levy, D. Nychka, S. A. Mickelson, J. Edwards, M. Vertenstein, and A. Wegener, "A methodology for evaluating the impact of data compression on climate simulation data," in *Proceedings of the 23rd International Symposium on High-performance Parallel and Distributed Computing*, ser. HPDC '14. New York, NY, USA: ACM, 2014, pp. 203–214. [Online]. Available: http://doi.acm.org/10.1145/2600212.2600217

[13] D. Tao, S. Di, H. Guo, Z. Chen, and F. Cappello, "Z-checker: A framework for assessing lossy compression of scientific data," *The International Journal of High Performance Computing Applications*, vol. 33, no. 2, pp. 285–303, 2019.

[14] Los Alamos National Laboratory, "VizAly-Foresight: A compression benchmark suite for visualization and analysis of simulation data," https://github.com/lanl/VizAly-Foresight, 2018, online.

[15] "EXASKY: Computing The Sky at Extreme Scales," https://www.exascaleproject.org/wp-content/uploads/2019/10/ExaSky.pdf, 2020, online.

[16] "EXAALT: Malecular dynamics at the exascale," https://www.exascaleproject.org/wp-content/uploads/2019/10/EXAALT.pdf, 2020, online.

[17] "GAMESS: Enabling GAMESS for exascale computing in chemistry and materials."

[18] T. Kurth, S. Treichler, J. Romero, M. Mudigonda, N. Luehr, E. Phillips, A. Mahesh, M. Matheson, J. Deslippe, M. Fatica, Prabhat, and M. Houston, "Exascale deep learning for climate analytics," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, ser. SC '18. IEEE Press, 2018.

[19] A. Li, O. Subasi, X. Yang, and S. Krishnamoorthy, "Density matrix quantum circuit simulation via the BSP machine on modern GPU clusters," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2020, pp. 1–15.

[20] J. Tian *et al.*, "CuSZ: An efficient gpu-based error-bounded lossy compression framework for scientific data," in *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '20, 2020, p. 3–15.

[21] cuZFP, https://github.com/LLNL/zfp/tree/develop/src/cuda_zfp, 2020, online.

[22] R. Underwood, S. Di, J. C. Calhoun, and F. Cappello, "FRaZ: A generic high-fidelity fixed-ratio lossy compression framework for scientific floating-point data," https://arxiv.org/abs/2001.06139, 2020, online.

[23] D. Tao, S. Di, Z. Chen, and F. Cappello, "Significantly improving lossy compression for scientific data sets based on multidimensional prediction and error-controlled quantization," in *2017 IEEE International Parallel and Distributed Processing Symposium*. IEEE, 2017, pp. 1129–1139.

[24] X. Liang, S. Di, D. Tao, S. Li, S. Li, H. Guo, Z. Chen, and F. Cappello, "Error-controlled lossy compression optimized for high compression ratios of scientific datasets," in *2018 IEEE International Conference on Big Data*. IEEE, 2018.

[25] Z. Wang, A. C. Bovik, H. R. Sheikh, and E. P. Simoncelli, "Image quality assessment: from error visibility to structural similarity," *IEEE Transactions on Image Processing*, vol. 13, no. 4, pp. 600–612, 2004.

[26] Lawrence Livermore National Laboratory, "zfp and Derivatives," https://computing.llnl.gov/projects/zfp/zfp-and-derivatives, 2021, online.

[27] Z. Wu and N. E. Huang, "A study of the characteristics of white noise using the empirical mode decomposition method," *Proceedings of the Royal Society of London. Series A: Mathematical, Physical and Engineering Sciences*, vol. 460, no. 2046, pp. 1597–1611, 2004.

[28] Justin Luitjens, "Faster parallel reductions on Kepler," https://developer.nvidia.com/blog/faster-parallel-reductions-kepler, 2014, online.

[29] Yuan Lin and Vinod Grover, "Using CUDA warp-level primitives," https://developer.nvidia.com/blog/using-cuda-warp-level-primitives, 2018, online.

[30] NVIDIA Research, "CUB documentation," https://nvlabs.github.io/cub/index.html, accessed 2021, online.

[31] Mark Harris and Kyrylo Perelygin, "Cooperative groups: Flexible CUDA thread programming."

[32] Mark Harris, "Finite difference methods in CUDA C/C++, Part 1," https://developer.nvidia.com/blog/finite-difference-methods-cuda-cc-part-1, 2013, online.

[33] K. Hou, H. Wang, and W.-c. Feng, "Gpu-uniache: Automatic code generation of spatial blocking for stencils on gpus," in *Proceedings of the computing frontiers conference*, 2017, pp. 107–116.

[34] "Hurricane ISABELA simulation dataset in IEEE Visualization 2004 Test," http://vis.computer.org/vis2004contest/data.html, online.

[35] "NYX simulation," https://amrex-astro.github.io/Nyx, online.

[36] "The local ensemble transform Kalman filter (letkf) data assimilation package for the scale-rm weather model," https://github.com/gylien/scale-letkf, online.

[37] "Miranda turbulence simulation," https://wci.llnl.gov/simulation/computer-codes/miranda, online.

[38] "Scientific data reduction benchmark," https://sdrbench.github.io/, online.

[39] K. Zhao, S. Di, X. Lian, S. Li, D. Tao, J. Bessac, Z. Chen, and F. Cappello, "SDRBench: Scientific data reduction benchmark for lossy compressors," in *2020 IEEE International Conference on Big Data (Big Data)*, 2020, pp. 2716–2724.

[40] C. Reddy, M. Kruse, and A. Cohen, "Reduction drawing: Language constructs and polyhedral compilation for reductions on GPU," in *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation*, 2016, pp. 87–97.

[41] "Quick Compression Analysis Toolkit (QCAT)," https://github.com/szcompressor/qcat.

[42] J. Wang, X. Xie, and J. Cong, "Communication optimization on GPU: A case study of sequence alignment algorithms," in *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2017, pp. 72–81.

[43] X. Yu, H. Wang, W.-c. Feng, H. Gong, and G. Cao, "cuART: Fine-Grained Algebraic Reconstruction Technique for Computed Tomography Images on GPUs," in *2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, 2016.

[44] Y. Tao and H. Zhi-Bin, "Shuffle reduction based sparse matrix-vector multiplication on Kepler GPU," *International Journal of Grid and Distributed Computing*, vol. 9, no. 10, pp. 99–106, 2016.

[45] X. Yu, H. Wang, W.-c. Feng, H. Gong, and G. Cao, "An enhanced image reconstruction tool for computed tomography on GPUs," in *Proceedings of the Computing Frontiers Conference*, ser. CF'17. ACM, 2017.

[46] C. Yang, A. Buluç, and J. D. Owens, "Design principles for sparse matrix multiplication on the GPU," in *European Conference on Parallel Processing*. Springer, 2018, pp. 672–687.

[47] X. Yu, H. Wang, W.-c. Feng, H. Gong, and G. Cao, "GPU-based iterative medical CT image reconstructions," *Journal of Signal Processing Systems*, vol. 91, no. 3-4, pp. 321–338, 2019.

[48] X. Yu and M. Becchi, "Exploring Different Automata Representations for Efficient Regular Expression Matching on GPUs," *SIGPLAN Not.*, 2013.

[49] ——, "GPU Acceleration of Regular Expression Matching for Large Datasets: Exploring the Implementation Space," in *Proceedings of the*

ACM International Conference on Computing Frontiers, ser. CF '13. New York, NY, USA: ACM, 2013, pp. 18:1–18:10.

[50] S. Koibuchi, K. Ikeuchi, S. Ishida, and H. Nishi, "GPU-based string matching method using warp shuffle instructions for network intrusion detection system on routers," *IEICE Technical Report; IEICE Tech. Rep.*, vol. 114, no. 155, pp. 113–118, 2014.

[51] X. Yu, *Deep packet inspection on large datasets: algorithmic and parallelization techniques for accelerating regular expression matching on many-core processors*. University of Missouri-Columbia, 2013.

[52] T. Ho, S.-R. Oh, and H. Kim, "A parallel approximate string matching under Levenshtein distance on graphics processing units using warp-shuffle operations," *PloS one*, vol. 12, no. 10, p. e0186251, 2017.

[53] F. Busato and N. Bombieri, "BFS-4K: an efficient implementation of BFS for kepler GPU architectures," *IEEE Transactions on Parallel and Distributed Systems*, vol. 26, no. 7, pp. 1826–1838, 2014.

[54] K. Shirahata, H. Sato, and S. Matsuoka, "Out-of-core GPU memory management for MapReduce-based large-scale graph processing," in *2014 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 2014, pp. 221–229.

[55] M. Nourian, X. Wang, X. Yu, W.-c. Feng, and M. Becchi, "Demystifying automata processing: Gpus, fpgas or micron's ap?" in *Proceedings of the International Conference on Supercomputing*, 2017, pp. 1–11.

[56] A. P. Diéguez, M. Amor, and R. Doallo, "Efficient scan operator methods on a GPU," in *2014 IEEE 26th International Symposium on Computer Architecture and High Performance Computing*. IEEE, 2014, pp. 190–197.

[57] K. Hou, W. Liu, H. Wang, and W.-c. Feng, "Fast segmented sort on GPUs," in *Proceedings of the International Conference on Supercomputing*, 2017, pp. 1–10.

[58] X. Yu, F. Wei, X. Ou, M. Becchi, T. Bicer, and D. D. Yao, "GPU-based static data-flow analysis for fast and scalable android app vetting," in *The 34th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2020.

[59] X. Yu, "Algorithms and frameworks for accelerating security applications on HPC platforms," Ph.D. dissertation, Virginia Tech, 2019.

[60] W. A. R. Jradi, H. A. D. do Nascimento, and W. S. Martins, "A fast and generic GPU-based parallel reduction implementation," in *2018 Symposium on High Performance Computing Systems (WSCAD)*. IEEE, 2018, pp. 16–22.

[61] S. G. De Gonzalo, S. Huang, J. Gómez-Luna, S. Hammond, O. Mutlu, and W.-m. Hwu, "Automatic generation of warp-level primitives and atomic instructions for fast and portable parallel reduction on GPUs," in *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2019, pp. 73–84.

[62] C. A. Navarro, R. Carrasco, R. J. Barrientos, J. A. Riquelme, and R. Vega, "GPU tensor cores for fast arithmetic reductions," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 1, pp. 72–84, 2020.

[63] Y. Zhang and F. Mueller, "Auto-generation and auto-tuning of 3D stencil codes on GPU clusters," in *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, 2012, pp. 155–164.

[64] P. S. Rawat, M. Vaidya, A. Sukumaran-Rajam, M. Ravishankar, V. Grover, A. Rountev, L.-N. Pouchet, and P. Sadayappan, "Domain-specific optimization and generation of high-performance GPU code for stencil computations," *Proceedings of the IEEE*, vol. 106, no. 11, pp. 1902–1920, 2018.

[65] O. Anjum, G. de Gonzalo Simon, M. Hidayetoglu, and W.-M. Hwu, "An efficient GPU implementation technique for higher-order 3D stencils," in *2019 IEEE 21st International Conference on High Performance Computing and Communications; IEEE 17th International Conference on Smart City; IEEE 5th International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*. IEEE, 2019, pp. 552–561.

[66] K. Matsumura, H. R. Zohouri, M. Wahib, T. Endo, and S. Matsuoka, "An5d: automated stencil framework for high-degree temporal blocking on GPUs," in *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization*, 2020, pp. 199–211.

[67] C. Oh, Z. Zheng, X. Shen, J. Zhai, and Y. Yi, "GOPipe: a granularity-oblivious programming framework for pipelined stencil executions on GPU," in *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques*, 2020, pp. 43–54.

[68] O. Green, R. McColl, and D. A. Bader, "GPU merge path: a GPU merging algorithm," in *Proceedings of the 26th ACM international conference on Supercomputing*, 2012, pp. 331–340.

[69] A. Krizhevsky, "One weird trick for parallelizing convolutional neural networks," *arXiv preprint arXiv:1404.5997*, 2014.

[70] T. De Matteis, G. Mencagli, D. De Sensi, M. Torquati, and M. Danelutto, "Gasser: An auto-tunable system for general sliding-window streaming operators on GPUs," *IEEE Access*, vol. 7, pp. 48 753–48 769, 2019.

[71] C. Luo, Y. Fei, and D. Kaeli, "Side-channel timing attack of RSA on a GPU," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 16, no. 3, pp. 1–18, 2019.

[72] P. Cooke, J. Fowers, G. Brown, and G. Stitt, "A tradeoff analysis of FPGAs, GPUs, and multicores for sliding-window applications," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 8, no. 1, pp. 1–24, 2015.

[73] X. Yu, V. Nikitin, D. J. Ching, S. Aslan, D. Gursoy, and T. Bicer, "Scalable and accurate multi-gpu based image reconstruction of large-scale ptychography data," *arXiv preprint arXiv:2106.07575*, 2021.

[74] X. Yu, T. Bicer, R. Kettimuthu, and I. Foster, "Topology-aware optimizations for multi-gpu ptychographic image reconstruction," in *Proceedings of the ACM International Conference on Supercomputing*, 2021, pp. 354–366.