Towards Efficient I/O Pipelines using Accumulated Compression

Avinash Maurya*, M. Mustafa Rafique*, Franck Cappello†, Bogdan Nicolae†,
*Rochester Institute of Technology, USA; †Argonne National Laboratory, USA
Email: *{am6429, mrafique}@cs.rit.edu; †{bnicolae, cappello}@anl.gov;

***Abstract*—High-Performance Computing (HPC) workloads generate large volumes of data at high-frequency during their execution, which needs to be captured concurrently at scale. These workloads exploit accelerators such as GPU for faster performance. However, the limited onboard high-bandwidth memory (HBM) on the GPU, and slow device-to-host memory PCIe interconnects lead to I/O overheads during application execution, thereby exacerbating their overall runtime. To overcome the aforementioned limitations, techniques such as compression and asynchronous transfers have been used by data management runtimes. However, compressing small blocks of data leads to a significant runtime penalty on the application. In this paper, we design and develop strategies to optimize the trade-off between compressing checkpoints instantly and enqueuing transfers immediately versus accumulating snapshots and delaying compression to achieve faster compression throughput. Our evaluations on synthetic and real-life workloads for different systems and workload configurations demonstrate 1.3× to 8.3× speedup compared to the existing checkpoint approaches.**

***Index Terms*—GPU compression and checkpointing, data accumulation, fast compression**

I. Introduction

A. Motivation

High-Performance Computing (HPC) applications produce massive amounts of distributed intermediate data during their execution which must be captured consistently as checkpoints in real-time under concurrency. While checkpointing have been traditionally associated with fault tolerance, it is widely used for productive scenarios as well, such as numerical and performance reproducibility [1], revisiting previous states in adjoint computations [2]–[4], posthoc analytics to study divergence, producer-consumer patterns in workflows and real-time analytics [5], capturing the lineage of DL model ancestors in transfer learning scenarios [6]. Unlike the case of fault tolerance, where the checkpoints are captured and/or restored infrequently, for productive scenarios the checkpoint frequency can be as small as tens of ms. Coupled with large checkpoint sizes, this results in a need to sustain a checkpoint throughput in the order of tens to hundreds of GB/s. While feasible on GPUs (whose memory bandwidth is in the order of TB/s), unfortunately there is not enough spare GPU memory capacity to capture all checkpoints there. Therefore, checkpoints are typically flushed to slower tiers of higher capacity (e.g., host memory, NVMe, CXL, remote storage, etc), from where they are reloaded back to GPU memory at a later time when they need to be revisited.

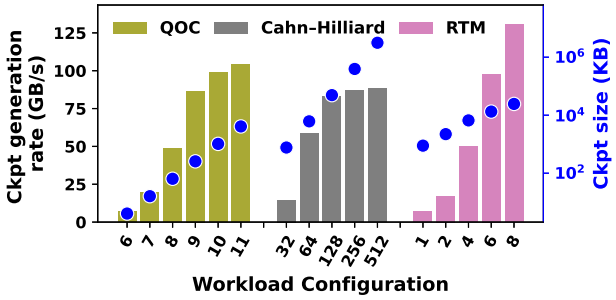


Fig. 1: Checkpoint generation rate and sizes for different workloads running with varying configurations.

As an example, consider the case of three different adjoint computations, as shown in Figure 1. The first application, Quantum-Optimal Control (QOC) [7], launches multiple simulations with different initial states of the quantum system to evaluate the optimal control parameters (e.g., external fields or pulses) to steer its evolution towards a specific target state, generating checkpoints at rates of 100 GB/s per simulation for a varying number of qubits (shown on the x-axis). The second application models spinodal decomposition in phase-field simulations using Cahn-Hilliard partial different equations [8]. The ensemble launched in the Cahn-Hilliard application explores different input parameters for studying the statistical impact of each parameter and exploring the phase space of the decomposition, generating checkpoints at rates up to 75 GB/s for different cubic lattice sizes (shown on the x-axis). Lastly, we consider the case of the Reverse-Time Migration (RTM) [2] technique which is widely used for seismic image processing in the oil and gas industry and generates checkpoints from rates of 20 GB/s to 150 GB/s for different wave field frequencies. The RTM application is launched as an ensemble in which each simulation operates on a chunk of the subsurface topology.

To reduce the overhead of flushing and restoring checkpoints to/from slower tiers, multi-level checkpoint runtimes and data movement engines such VELOC [9], ADIOS2 [10], and FTI [11] exploit asynchronous transfer techniques that overlap flushes with the computations by performing I/O operations in the background. However, as the checkpoint production rate increases, the spare GPU memory is filled faster than the checkpoints can be flushed to slower tiers over a PCIe link that can support only tens of GB/s. Therefore, asynchronous multi-level checkpointing techniques are insufficient to hide the I/O overheads, which results in longer

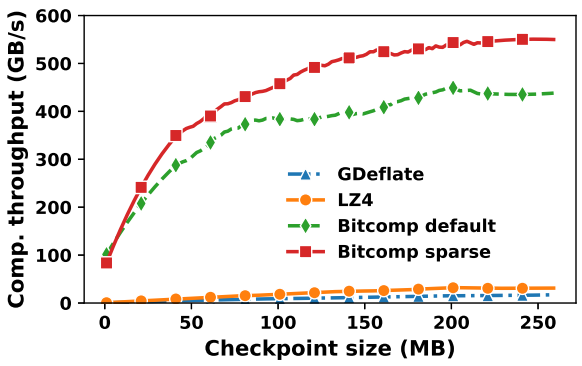


Fig. 2: Compression throughput with increasing input data size.

application runtime and GPU underutilization. The problem is further complicated by the fact that HPC applications often run ensembles that comprise multiple tasks, each of which is deployed on a separate GPU and which share and compete for the I/O bandwidth of the host memory.

To address this challenge, checkpoints can be compressed before being flushed, under the assumption that the compression reduces the checkpoint sizes enough so that asynchronous multi-level techniques are not bottlenecked by slow flushes [12]. However, compression algorithms have high computational overhead and typically block the application (i.e., they cannot be applied asynchronously like the flushes).

In this paper, we study how to apply blocking compression algorithms in order to reduce the time spent waiting for asynchronous flushes to slower tiers to finish, which typically happens when computations are faster than flushes and therefore they can only partially overlap. We aim to minimize the combined duration of compression and waits for asynchronous flushes, which minimizes the overall checkpointing overhead perceived by the application.

B. Limitations of the State-of-the-Art

HPC applications typically use various lossy and/or lossless [13] compression techniques to minimize congestion on both storage tiers and I/O interfaces. Depending on the data representation and the compression configuration, e.g., error bounds, compression throughput, and compression ratio, the state-of-the-art GPU compressors, e.g., nvCOMP [14], cuSZ [12], Mgard+ [15], and cuZFP [16], can achieve $>300\times$ compression ratio. These compression libraries feature high compression throughput for large input data sizes but they perform poorly when compressing smaller data sizes. For example, as shown in Figure 2, the state-of-art compression library from Nvidia, i.e., nvCOMP, attains the peak compression throughput of $\sim$550 GB/s only when the checkpoint sizes are $>$200 MB, when tested on Nvidia A100 GPU using 4 different compressors. Moreover, even if the nvCOMP compressors are run on a separate CUDA stream, it blocks the execution of all application kernels on the GPU, while performing compression or decompression operations, leading to longer overall application execution times. Therefore, the compression ratio does not solely speed up the checkpointing throughput or the end-to-end application execution time.

A widely used approach to eliminate the overheads of slow compression for small-sized checkpoints is to accumulate checkpoints until they sum up to the size at which the compression can run at peak throughput. However, leveraging this approach using the nvCOMP compressor would require deferring compression until a minimum of 200 MB checkpoints are accumulated. However, such an approach has two limitations: (1) the size of the available GPU cache may not be adequate to stage the uncompressed accumulated checkpoints and/or the scratch space required by the nvCOMP compressor to compress the accumulated checkpoints; and (2) the PCIe interface would be idle while checkpoints are being accumulated, leading to application stalls due to delayed flushes. To solve these limitations, we need to solve a multi-objective optimization problem that takes into account the checkpoint generation rate, the compression throughput, the compressed sizes, and number of checkpoints in order to devise an efficient asynchronous checkpoint pipeline. Existing data-movement engines [10] and checkpointing runtimes [9], [11] do not consider such aspects simultaneously.

C. Key Insights and Contributions

In this paper, we contribute with an asynchronous checkpoint pipeline solution that determines an *optimal checkpoint schedule*, i.e., what checkpoints to leave uncompressed, what checkpoints to compress and what checkpoints to accumulate and compress in bulk in order minimize the overhead perceived by the application (i.e., total duration interruptions due to compression and waiting for flushes to finish). We assume two representative tiers: GPU memory and host memory. However, our approach is generic and can be easily adapted to other tiers with different characteristics. The key idea of our approach is a dynamic programming algorithm that builds the optimal checkpoint schedule using predictions of the compression ratio, which are obtained from past experience with similar checkpointing data generated in related ensemble runs of the application. We summarize our contributions as follows:

1) We formulate the problem of compression overheads when capturing small-sized high-frequency checkpoints from the GPU memory to the host cache (§ II).
2) We propose a series of design principles to minimize the asynchronous checkpoint flush times through multiple key ideas: profiling initial simulations to obtain checkpoint characteristics of HPC ensembles; using a mix of uncompressed, compressed, and collectively compressed checkpoints for flushing; sharing GPU cache between uncompressed, accumulated, and ready-to-flush checkpoints, and dynamic allocation and defragmentation of GPU cache (§ III).
3) We implement our design principles in the VELOC [9], a multi-level HPC checkpointing library. Although we use Nvidia-based GPUs for prototyping, our ideas are generic and can be adapted for other GPUs (§ III-C).
4) We perform a thorough evaluation to demonstrate the effectiveness of our approach using the Nvidia DGX A100 multi-GPU system. We use Intel OneAPI-based

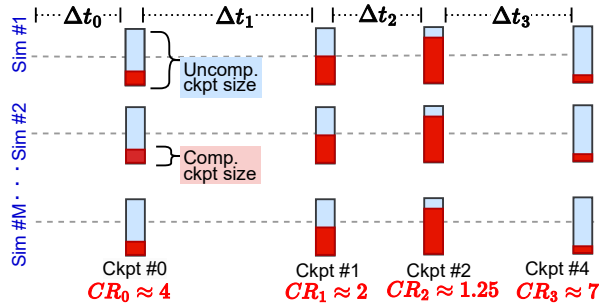


Fig. 3: M simulations running in an ensemble, each consisting of a single process producing $K = 4$ checkpoints at varying intervals (Δt) with different compression ratio (CR).

DPC++ RTM application [17] and synthetic benchmarks to study the performance of our approach for a diverse range of checkpoint characteristics (§ IV).

II. Problem Formulation

Consider an HPC ensemble consisting of M identical simulations running on an HPC system where each simulation runs N processes such that each process is exclusively mapped to a single GPU. During its execution, each process produces K checkpoints of constant size U, which are reused later. Each process i can afford to dedicate only a fraction of the GPU memory as a staging area to buffer, compress and flush its K checkpoints to the host memory. The host memory has enough capacity to collect all $K \cdot N$ checkpoints of all processes.

Each time a new checkpoint j is generated in the GPU memory, it is copied first to the staging area. If there is not enough space available on the staging area, the process blocks until enough space becomes available. After a copy of the checkpoint was made on the staging area, the process can optionally decide to further block in order to compress checkpoint j and any other predecessor checkpoints accumulated in the staging area in bulk. Then, the application continues running and overlaps with the transfers of the checkpoints (compressed or uncompressed) to host memory, which happen asynchronously.

We assume that the checkpoint intervals and checkpoint compression ratios are non-uniform and may evolve during the simulation. However, we assume the simulations in the ensemble are related and therefore produce a similar evolution (i.e., the checkpoint interval Δt_j and compression ratio CR_j of checkpoint j of process i remains similar for all simulations in the ensemble). As an example, Figure 3 shows the checkpointing pattern that we observe in real-life RTM ensemble described in § IV-C1 for a simplified case where each simulation employs a single process.

Under these circumstances, given an *oracle* that can predict the compression ratio of all checkpoints (e.g., by compressing all checkpoints of one of the simulations in the ensemble), our goal is to obtain an *optimal schedule* that decides when to flush uncompressed checkpoints vs. when to accumulate, compress in bulk and flush the compressed data such as to minimize the checkpointing overhead (interruptions due to compression and waiting for flushes) during the simulation.

III. System Design

A. Design Principles

1) The Two Oracles: Profiling Checkpoints of the Initial Simulation of the Ensemble and Empirical Compression Throughput Analysis: We consider that our HPC ensembles are composed of identical simulations, which typically lead to identical checkpoint characteristics across the simulations, as highlighted by Figure 3 and empirically characterized in § IV-C1. For example, the RTM application launches an ensemble to calculate the geological characteristics of large surface areas, and each simulation runs the RTM kernels on a smaller subsurface. Although the input data of each simulation of the ensemble is distinct, we observe that the checkpoint generation rate and compression ratios of a given timestep across different simulations remains consistent as shown in § IV-C1. We leverage this similarity between checkpoints across multiple simulations to generate a checkpoint-characteristic aware compression and flush schedule. Albeit, compression ratio prediction schemes [18] can be used to generate profiles for ensembles with heterogeneous checkpointing characteristics.

a) Oracle to Predict Compressed Checkpoint Sizes and Checkpoint Intervals: Using the formulation proposed in § II and shown in Figure 3, we build the first oracle to predict the checkpointing characteristics as follows: (1) all the checkpoints generated in the first simulation of the ensemble are compressed individually to learn about the compressed sizes of each checkpoint; and (2) the compressed sizes, along with time elapsed since the last checkpoint are recorded by the checkpointing runtime. We empirically demonstrate the similarity of checkpoints across multiple simulations and variability across a single simulation in § IV-C1.

b) Oracle to Determine the Compression Speed for Different Input Checkpoint Sizes: Next, we construct an oracle to determine the compression throughput by past compression logs or offline analysis for different data sizes. Given that the compression throughput for a large proportion of compression libraries (notably, including nvCOMP) is a function of data size and not data content, the offline compression throughputs obtained for different data sizes are applicable for all checkpoints of the ensemble, irrespective of the checkpoint content.

The two oracles, coupled with the peak PCIe transfer throughput of the system, support the following components of our algorithm to generate an optimal checkpointing schedule: $cs[K]$: compressed checkpoint sizes of all K checkpoints; $d[K]$: time interval between consecutive checkpoints of a simulation; $comp\_time(x)$: the time taken by the compressor to compress x bytes of data.

2) Mix of Uncompressed, Individually Compressed, and Accumulated Compressed Checkpoints: Depending on the time interval between consecutive checkpoints and the number of checkpoints, flushing all checkpoints in compressed or uncompressed form is faster. For instance, if the time interval between consecutive checkpoints is large enough to finish the transfer of a checkpoint in uncompressed form, compression operations should not be used because they slow down application

execution by blocking the application kernels. On the other hand, if the time interval between consecutive checkpoints is not large enough to flush uncompressed checkpoints, performing compression to reduce checkpoint sizes may lead to faster transfers depending on compression ratio, compression speed, and GPU cache size. Lastly, to accelerate compression, multiple checkpoints can be accumulated to perform collective compression. While each of the aforementioned approaches, i.e., flushing uncompressed, compress instantly and flush, and accumulate and flush, have their own limitations and advantages in minimizing the overall flushing time, in our approach, we utilize a mix of all three strategies to devise an efficient compression and flush schedule.

3) Dynamic Allocation and Defragmentation of Temporary Compression Buffers: To run compression on the GPU, compression libraries require a temporary buffer for writing the compressed data. The size of this temporary buffer is slightly larger than the size of the uncompressed input data due to two reasons: (1) it assumes the worst-case of the input data to be incompressible, i.e., the checkpoint consists of unique values and cannot be compressed even by a small degree; (2) the compressor header, which includes details about the compression algorithm and meta-data corresponding to the uncompressed checkpoint, adds to the size of the compressed data. While the size of the compressed data is the same as the size of the temporary buffer if the input data was incompressible, it can be significantly smaller than the allocated temporary buffer when the checkpoints are highly compressible.

Directly enqueuing these temporary compression buffers for flushing to the host memory leads to cache underutilization because of slow PCIe transfers. For instance, if a checkpoint was compressed by 3×, two-thirds of the temporary compression buffer is unutilized, and remains blocked until the checkpoint is flushed to the host memory. While resizing approaches can be adopted to reclaim unused space, GPU-based memory allocation techniques do not natively support resizing, and third-party memory allocation libraries such as Umpire [19], lead to uneven fragments, due to which future cache allocations are slow. Therefore, to avoid cache underutilization, before enqueueing the compressed checkpoints for flushes, we perform fast defragmentation of the temporary compression buffer as follows: (1) allocate a new buffer B on the GPU cache, whose size is equal to the compressed size of the checkpoint ($sizeof(B) \leq sizeof(temp\_buffer)$); (2) perform fast device-to-device copy (∼500 GB/s) from temporary buffer to buffer B; (3) enqueue buffer B for flushing to the main memory; and (4) free the temporary buffer space.

Existing approaches allocate one temporary compression buffer by assuming that the checkpoint sizes throughout the application execution remain constant. Reusing a single buffer leads to GPU cache underutilization, but eliminates the cost of repetitive allocation and deallocation. However, this is not suitable for accumulated checkpoints as the number of checkpoints to accumulate is variable at different timesteps, and also due to a hybrid mix of the three approaches, compression may not be required, as described in § III-A2. Therefore, statically allocating a fixed-size temporary buffer for compression is both infeasible and leads to more wastage of the expensive GPU cache. In our approach, we mitigate the GPU cache underutilization due to temporary compression buffers by on-demand allocating dynamic-sized buffers and defragmenting them after the compression is complete.

4) Shared GPU Cache for Accumulated Checkpoints, Compressor's Temporary Buffer, and Ready-to-Flush Checkpoints: The checkpoints stored on the GPU cache can be classified into three groups: (1) accumulated checkpoints: these checkpoints are stored in their uncompressed form for later compression; (2) compressor's temporary buffer: this GPU buffer is used by the compressor to write compressed checkpoint and its size is directly proportional to the size of input checkpoint; and (3) ready-to-flush checkpoints: these checkpoints are ready to be flushed to the host memory (similar to the buffering area).

Reserving a fixed fraction of the GPU cache for each of the three categories simplifies GPU cache management, but leads to underutilization because depending on the instantaneous checkpoint generation throughput, compression speed, and flush rate, the GPU cache required by each of the categories is variable. For instance, reserving a large fraction of the GPU cache for staging ready-to-flush checkpoints would be sub-optimal if the compressed checkpoints can be easily flushed between consecutive checkpoint invocations. Instead, dedicating a larger fraction of the GPU cache to accumulate checkpoints would have been a better design choice since it would allow for faster compression speeds, leading to a more optimal flush completion time. Therefore, our approach allows sharing the limited GPU cache across all three groups.

5) GPU Cache Management using Stream-Ordered CUDA Memory Pools: Sharing and managing the GPU cache efficiently across the three different classes of data (§ III-A4) is challenging due to differences in data sizes, differences in production and consumption patterns, and the difference in checkpoint lifetimes of each of these classes. Each of these differences can be attributed to dynamic compression ratios, compression speeds, and the fraction of GPU memory available for each class. To efficiently support such divergent classes of data on GPU cache, repetitive allocation, and deallocation are required. Although we can allocate memory directly on the GPU HBM at 160 GB/s on a system with 8× GPU [20], performing frequent allocations introduces significant penalties during checkpoint and compression, especially since the peak compression speed (550 GB/s) is orders of magnitude higher than the GPU memory allocation rate.

To support such high-frequency memory allocations and deallocations on the GPU cache, we leverage CUDA-based memory pools. Depending on the allocation size requested, CUDA pools achieve 773× to 9816× faster GPU memory allocation [21] as compared to the default GPU memory allocation approach.

B. Accumulated Checkpoint Compression and Flushing

In this section, we formulate the algorithm to generate the optimal checkpoint compression schedule based on the

oracles described in § III-A1, which complements our algorithm by supplying the following parameters: total number of checkpoints, n; the size of an uncompressed checkpoint, u; checkpoint generation rate, d; and compressed sizes, cs; of all checkpoints generated in the first simulation.

To generate an optimal checkpoint compression schedule, we systematically search through all combinations of compression decisions, i.e., U: flush in uncompressed form; C: compress a single checkpoint and flush; A: accumulate checkpoints on the GPU cache for later compression; and X: compress accumulated checkpoints. Using the two oracles (§ III-A1), we permute all U, C, A, X for a given checkpoint j using the recursive decomposition described in Equation 1.

$$\texttt{f(i)} = \min_{j=i+1}^{n} \begin{pmatrix} \texttt{flush(cs[j])+comp\_time(u)+f(j+1)} \\ \texttt{flush(u)+f(j+1)} \\ \min_{l=j+1}^{n} \begin{pmatrix} \texttt{flush(cs[j:l]) +} \\ \texttt{comp\_time(u*(l-j))+f(l+1)} \end{pmatrix} \end{pmatrix} \quad (1)$$

Solving Equation 1 for `f(0)` yields an optimal schedule for all simulations in the ensemble. To avoid the exponential $\mathcal{O}(3^N)$ search through approaches such as backtracking, we use a dynamic programming (DP) approach. Our proposed DP-based approach, shown in Algorithm 1, generates an optimal compression schedule in $\mathcal{O}(N \cdot T)$, where T is the checkpointing time when all checkpoints are uncompressed.

Specifically, our algorithm uses the oracle functions to model the timesteps of the main application thread t^a and asynchronous flushing thread t^f based on the amount of GPU cache available g. We construct a vector of maps for memoizing the intermediate results of optimal flush strategies for i^{th} checkpoint at the i^{th} index of the vector. The map entries in the vector are of the form $\langle t^f : (t^a, ug) \rangle$, where t^f, t^a, and ug represent the timestamps of flush completion, application execution, and used GPU cache, respectively. Similarly, for storing the intermediate solutions, vector s stores a map of t^f and a string of compression plan, which is composed of characters (U, C, A, X), indicating the corresponding compression strategy to be adopted at the given index of the string.

Using the decomposition from Equation 1, the minimum checkpointing time for i^{th} checkpoint, when paired with previous combinations from $dp[\min(i-1, 0)]$, is given as the lesser of three approaches: (1) individually compress and flush (Lines 10-15); (2) flush without compression (Lines 16-21); or (3) accumulate the next k checkpoints to compress them collectively before flushing (Line 22-29). The output string is composed of characters $\{U, C, A, X\}$ and is indicative of the optimal checkpoint compression strategy for all the remainder simulations of the ensemble.

Algorithm 1: Algorithm to generate checkpoint accumulation and compression schedule.

Input : n: total number of checkpoints, u: uncompressed checkpoint size, g: GPU cache capacity, $\langle cs, d \rangle$: tuples of compressed sizes and checkpoint intervals from the oracle. P: const. peak PCIe bandwidth.

Output: S: a string of the optimal compression plan, where $S[i] \in (U, C, A, X)$ denotes the compression plan to be adopted for the i^{th} checkpoint.

1 **Function** `get_compression_schedule(`n, u, cs, d`)`**:**
2 $dp \leftarrow [\{0 : (0, 0)\}]$ // $\{t^{flush} : (t^{app}, used\_gpu)\}$
3 $s \leftarrow [\{0 : \text{""}\}]$ // $\{t^{flush} : plan\}$
4 $r \leftarrow$ `get_temp_buffer(`u`)` // $r = u +$ comp_header
5 **if** $u + r > g \vee$ `d2h(`u`)` $< max(d)$ **then**
6 $S \leftarrow$ 'U' $* n$; **return** S;
7 **for** $i \leftarrow 0$ **to** n **do**
8 $j \leftarrow min(i-1, 0)$
9 **for** $\{t^f, (t^a, ug)\} \in dp[j]$ **do**
// Individually compress ckpt. i
10 $t^a_c \leftarrow t^a +$ `comp_time(`u`)` $+ d[i]$
11 $ug_c \leftarrow u + r + cs[i] + min(0, ug - P \times (t^a_c - t^a))$
12 $t^f_c \leftarrow$ `d2h(`$cs[i]$`)` $+ max(t^f, t^a_c)$
13 **if** $(a[i][t^f_c][0] > t^a_c)$ **then**
14 $dp[i] \leftarrow dp[j] \cup \{t^f_c, (t^a_c, ug_c - u - r)\}$
15 $s[i][t^f_c] \leftarrow s[j][t^f] \cup$ 'C'
// Do not compress ckpt. i
16 $t^a_u \leftarrow t^a + d[i]$
17 $ug_u \leftarrow u + min(0, ug - P \times (t^a_u - t^a))$
18 $t^f_u \leftarrow$ `d2h(`u`)` $+ max(t^f, t^a_u)$
19 **if** $a[i][t^f_u][0] > t^a_u$ **then**
20 $dp[i] \leftarrow dp[j] \cup \{t^f_u, (t^a_u, ug_u)\}$
21 $s[i][t^f_u] \leftarrow s[j][t^f] \cup$ 'U'
// Accumulate and compress ckpts. $i \cdots i+k$
22 **for** $k \leftarrow 2$ **to** $min(n-i, \lfloor g/(u+r) \rfloor)$ **do**
23 $l \leftarrow i + k - 1$
24 $t^a_a \leftarrow t^a +$ `comp_time(`$k * u$`)` $+ \Sigma(d[i:l])$
25 $r' \leftarrow$ `get_temp_buffer(`$k * u$`)`
26 $ug_a \leftarrow k * u + r' + \Sigma(cs[i:l]) + min(0, ug - P * (t^a_a - t^a))$
27 $t^f_a \leftarrow$ `d2h(`$\Sigma(cs[i:l])$`)` $+ max(t^f, t^a_a)$
28 **if** $(a[l][t^f_a][0] > t^a_a)$ **then**
29 $dp[i] \leftarrow dp[j] \cup \{t^f_a, (t^a_a, ug_a - k*u - r')\}$
30 $s[i][t^f_a] \leftarrow s[j][t^f] \cup$ 'A' $* (k-1) \cup$ 'X'
31 **return** $s[n-1][min(dp[n-1].keys())]$

C. Implementation

We implement our approach in VELOC [9], which is a production-ready multi-level user-space checkpointing library. VELOC is optimized for CUDA-enabled GPU, and provides parameterized configuration of GPU and host-cache space, fast GPU and pinned host cache initialization, asynchronous DMA-enabled device-to-host transfers, and state-machine-enabled efficient prefetching [4], [20]. We integrate Nvidia's nvCOMP compression library [14] with VELOC to compress checkpoints using multiple compressors, e.g., GDeflate, LZ4, and BitComp [12], [14]. We improve the compression and cache management performance by adopting two optimizations: (1) proactively preloading nvCOMP library on the GPU at initialization to eliminate runtime penalties due to lazy loading of the library, and (2) pre-faulting CUDA memory pool for faster cache allocations.

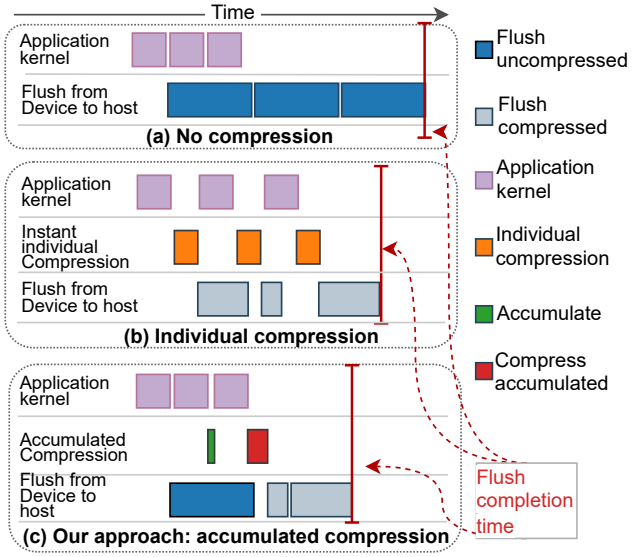


Fig. 4: Flow of different checkpointing approaches.

IV. Performance Evaluation

A. Experimental Setup

We conduct our experiments on ALCF's ThetaGPU HPC testbed [1] that consists of 24 Nvidia DGX A100 nodes. Each node is equipped with 1 TB DDR4 memory (20 GB/s, 8 NUMA domains), two 64-core AMD Rome CPUs (256 threads), four 3.84 TB Gen 4 NVMe drives (4 GB/s) and 8 Nvidia A100 GPUs aggregating to a total of 320 GB HBM memory. On each node, 8 A100 GPUs are connected with each other using 6 NVSwitches and with the host memory through a PCIe Gen 4 interface. The peak unidirectional Device-to-Device (D2D), and pinned Device-to-Host (D2H) (and vice versa) bandwidths on each GPU are 500 GB/s and 25 GB/s, respectively. Two GPUs share the same PCIe interconnect via PCIe switch to the host buffer, which effectively reduces their D2H bandwidths during concurrent access, and also allows for only 4 out of the 8 available NUMA domains to be directly accessible from the GPUs. All the nodes run Nvidia CUDA v11.4.152, nvCOMP v2.6.1, OneAPI/DPC++ v2023.0.0, and OpenMPI v4.1.0 on top of the Ubuntu 20.04.6 OS. In our experiments, we use a single node consisting of 8 GPUs to study the impact of PCIe interface contention and DRAM write bandwidth under node-local concurrency.

B. Compared Approaches

We compare the approaches shown in Figure 4 as follows:

1) No Compression: This is the first approach in Figure 4 and represents the typical state-of-the-art strategy employed by data movement engines and checkpointing runtimes, i.e., individual transfers of the uncompressed checkpoints from the GPU memory to the host memory. Although many approaches perform the transfers synchronously (e.g., ADIOS2 [10]), we compare with an asynchronous approach [4] that overlaps the transfers with the computations by using the spare GPU memory as a buffer. This approach was shown to be significantly faster than synchronous approaches, hence a better baseline. We denote this approach *No compress* in our evaluations.

2) On-the-fly Individual Checkpoint Compression: In this approach, the checkpoints are compressed one at a time at their source, i.e., the GPU HBM in our case, by the checkpointing runtime. This approach is widely used for accelerating data transfer for out-of-core stencil computations [22], reverse-mode adjoint computations [23], and reducing data-stream intensity from scientific equipment, e.g., Advanced Photon source [24]. Therefore, we consider this approach as representative of state-of-the-art GPU-compression-enabled data movement techniques. To this end, we configure the GPU-enabled VELOC checkpointing runtime to perform data compression before storing the checkpoints on the GPU cache. This approach is labeled as *Compress single* in our evaluations and is illustrated as the second approach in Figure 4.

3) Our Approach: Finally, we compare the aforementioned approaches with our proposal, which is based on the design principles listed in § III-A. As described in § III-C, this approach represents the optimized checkpoint accumulation and compression scheme using the dynamic-programming formulation listed in Algorithm III-B and is illustrated as the third approach in Figure 4.

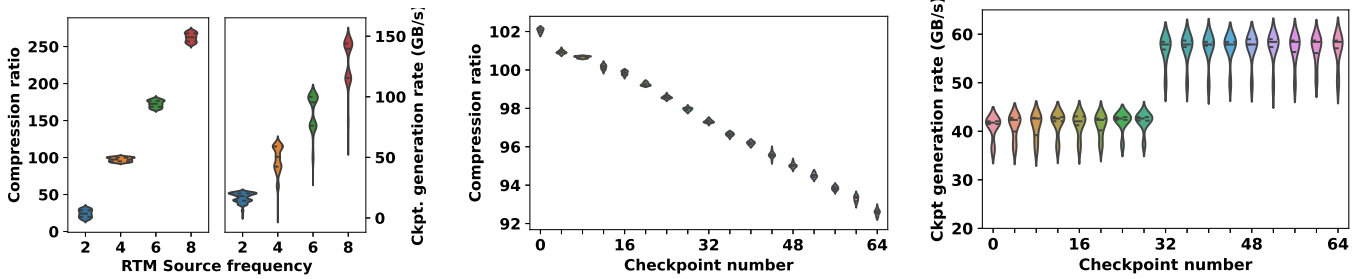
C. Evaluation Methodology

We evaluate each of the above-mentioned approaches using the following metrics and scenarios:

1) Real-life Application – RTM: As an application example, we consider the case of seismic imaging and characterization of geological substructures using the Reverse-Time Migration (RTM) technique. RTM is a widely used technique in the oil and gas industry for the exploration of reservoirs and aids geophysicists in understanding geology formation, faults, and fractures. In terms of checkpointing, the RTM application can be characterized as an adjoint computation, in which the checkpoints produced during the forward pass are later consumed (in reverse order) by the backward pass to cross-correlate and form the subsurface image. However, for the cases of debugging, exploring alternate models for cross-correlation, and post hoc analysis, the checkpoints produced in the forward mode are consumed by different agents/jobs of the workflow ensemble.

a) RTM Configuration: In our evaluations, we run an open-source version of the RTM application, written in DPC++, for various source frequencies of the wavefield. While lower source frequencies are used by RTM to compute the coarse-grained topology of the subsurface, higher frequencies are used to study the finer details of the underlying substructures. In practice, the high-frequency RTM simulations are usually run in distributed mode on several GPUs using domain decomposition and produce the same checkpoint sizes at the same intervals, as those generated by running low-frequency simulations on a single GPU. Therefore, without

[1] https://www.alcf.anl.gov/alcf-resources/theta



(a) Compression ratios and checkpoint generation rate for different RTM source frequencies.

(b) Variations in compression ratios of RTM app. at every checkpoint in an ensemble of 20 simulations.

(c) Variation in checkpoint generation rates across 20 shots for 66 checkpoints.

Fig. 5: Checkpointing characteristics for varying RTM source frequencies (Figure 5a); Variations in compression ratios (Figure 5b) and checkpoint generation rates (Figure 5c) of 20 RTM simulations for RTM source frequency=4.

loss of generality, we restrict our configuration options to multiple low-frequency $(2 \ldots 8)$ RTM simulations. The number of checkpoints produced during the execution of a single seismic shot in the RTM application, in addition to other configuration options, depends on the source frequency of the simulation. Since the varying number of checkpoints would add another dimension of variability to our evaluations, we tune the `dt-relax` parameter, such that all simulations produce 66 checkpoints (maximum number of checkpoints produced by the lowest frequency) to evaluate a consistent number of timesteps (checkpoints) for different source frequencies.

b) RTM Dataset: We use the open-source 2004 BP velocity benchmark dataset [25], which is widely used for benchmarking depth migration in geology. The dataset consists of 1348 seismic shots, out of which we use 20 representative shots (601-620) in our evaluations to limit the amount of execution time for different sets of configurations and approaches.

c) Analysis of RTM Checkpointing Characteristics: We study the checkpoint generation and compression characteristics of the RTM application across 20 representative seismic shots to understand the I/O and compute patterns that can be exploited to build efficient checkpoint compression schedules.

We first analyze the compression ratio of the RTM application running these 20 seismic shots. Figure 5a shows a statistical overview of the distribution of compression ratio and checkpoint generation rates for different source frequencies across the 20 representative seismic shots. Here, the compression ratio and checkpoint generation rates are both a function of source frequency and show some degrees of variance in compression ratio and checkpoint generation. We further investigate these variations by considering the case of a source frequency of 4 Hz. Figure 5b shows a violin plot representing the variance in compression ratio of the 66 checkpoints, across 20 different seismic shots. In each shot across 66 checkpoints, the size of the uncompressed buffer is constant, but their compressed sizes differ based on the contents of the checkpoint. We observe that during the initial checkpoints, the checkpoints are highly compressible, i.e., show high compression ratios, and gradually become less compressible. This is because, during the later timesteps of the simulation, the wave field becomes more complex due to interference resulting in checkpoints with more unique data points and low compression ratios. We highlight and reiterate two important observations from this graph: (1) the compression ratio of checkpoints while processing a single seismic shot is not consistent; and (2) the compression ratio, for a given timestep (checkpoint number), shows at most 10% variation across the 20 different seismic shots. From the perspective of checkpointing, observation (1) leads to uneven GPU cache utilization which results in fragmentation, and variability in the amount of compressed checkpoints flushed to the host memory across the slow PCIe interface. The second observation with similarity in compression ratio reveals that the checkpoint-compression schedules generated based on the compressed sizes of the first seismic shot would be nearly optimal for the remainder of simulations in the ensemble.

Next, we evaluate the checkpoint generation rates across the 20 representative shots. As observed in Figure 5c, the checkpoint generation rate across different checkpoints, even for a single shot, is variable. Since the size of all checkpoints in uncompressed format is the same, varying checkpoint generation rates suggest that the time interval between consecutive checkpoints of a single shot is variable. Specifically, for a given seismic shot, the initial checkpoints are generated at $\sim$40 GB/s until timestep 32, after which they are generated at the rate of $\sim$57 GB/s. However, as seen in the violin plot in Figure 5c, the checkpoint generation rate at a given timestep across all 20 seismic shots is near similar, i.e., irrespective of the input dataset, the time taken by the RTM kernel to run consecutive timesteps is similar across all 20 seismic shots.

2) Synthetic Benchmarks: We develop a series of synthetic benchmarks to evaluate the performance of various checkpoint compression approaches on varying application characteristics. These benchmarks generate synthetic checkpoints for which the compression ratio, checkpoint interval, number of checkpoints, and scalability can be parameterized. Unlike the case of RTM, which has a variable compression ratio and checkpointing interval while processing different timesteps in a single simulation (seismic shot), the synthetic benchmarks produce all checkpoints at uniform intervals and constant compression ratios, similar to those generated by the QOC and Cahn-Hilliard [1].

This benchmark enables us to evaluate a wide range of application characteristics and configurations in our experi-

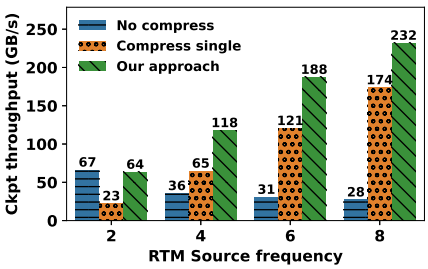


Fig. 6: Checkpointing throughput for varying RTM source frequencies.

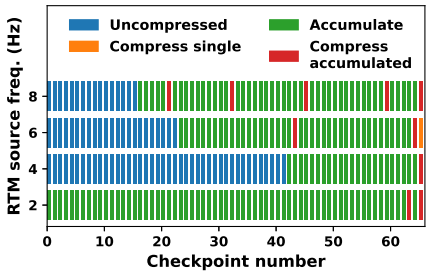


Fig. 7: Compression schedule decisions for varying RTM source frequencies.

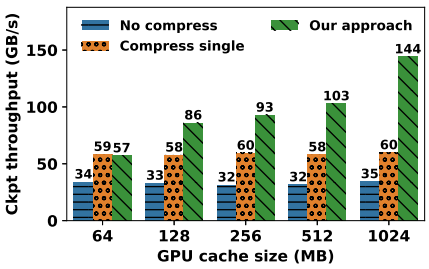


Fig. 8: Checkpointing throughput for varying GPU cache sizes for the RTM ensemble.

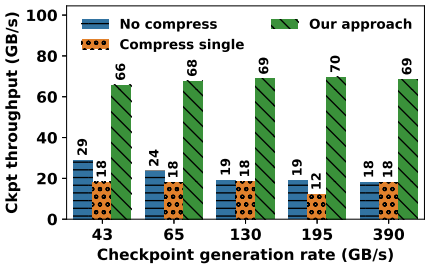


Fig. 9: Checkpointing throughput for varying checkpoint rates of synthetic benchmark.

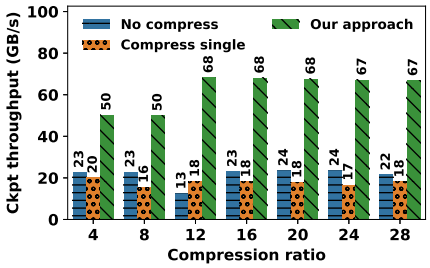


Fig. 10: Checkpointing throughput for varying comp. ratios of synthetic benchmark.

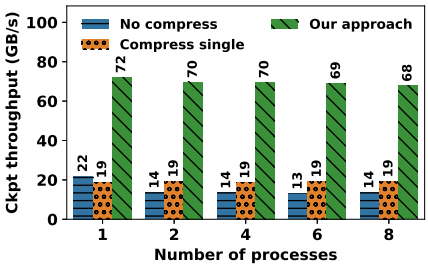


Fig. 11: Weak scaling for concurrent checkpointing using synthetic benchmark.

ments. To control the compression ratio of the checkpoint, we set $(cr-1)/cr$ elements of the floating point buffer to be checkpointed as `0.0F`, where cr represents the supplied compression ratio. Similar to the case of RTM, the synthetic benchmark ensembles consist of 20 simulations, but instead of producing 66 checkpointing per simulation, we generate 100 checkpoints of 2 MB each per simulation to more extensively evaluate our proposal for a larger number of checkpoints. The checkpoint generation rate is controlled by simulating 'sleep cycles' between consecutive checkpoints.

3) Performance Metrics and Multi-level Cache Configurations: In our evaluations, we measure the checkpointing throughput (GB/s) perceived by the ensemble of RTM application and synthetic benchmarks. Additionally, we study the sequence of checkpoint compression schedules generated by our approach for varying RTM source frequencies to understand the composition of uncompressed, singly compressed, and cumulatively compressed checkpoints. We consider that the host cache is large enough to store all checkpoints generated by the ensemble in uncompressed form, which can be extended to local or remote storage (PFS), if required.

Depending upon the source frequency of RTM simulation, the total size of all checkpoints generated by a single seismic shot can vary from 600 MB to 1.6 GB. Unless otherwise noted, we set the available spare GPU HBM available as GPU cache to 128 MB. A small GPU cache size allows the application to consume a major fraction of the expensive low-capacity GPU HBM for latency-critical kernels and creates challenging scenarios for evaluating our approach since lower cache sizes limit the number of checkpoints that can be accumulated.

D. Performance Results

1) Checkpointing Throughput: In our first set of experiments, we evaluate the checkpointing throughput as observed by the application. This is an important metric because it measures the amount of overheads incurred by the application as a result of checkpoint operations. A higher checkpoint throughput indicates faster completion of flushes and minimal blocking time for the application, indicating higher is better.

We first measure the checkpointing throughput for the RTM application for variable simulation source frequency. As observed in Figure 6, the checkpoint throughput for a low source frequency of 2 Hz using *compress single* checkpoint at a time approach is 2.8× slower than the case in which we flush without compression or our approach. This is because, at lower source frequencies, the checkpoint generation rate (17 GB/s) is lower than the peak PCIe bandwidth (25 GB/s), therefore flushing checkpoints in uncompressed form is more efficient as compared to flushing in compressed form. However, our approach shows slightly lower checkpointing throughput as compared to the case of *No-compress*. This is because the oracle described in Section III-A1 performs compression of all checkpoints during the first simulation to generate an optimal checkpoint schedule. Due to a very slow checkpoint generation rate and less number of seismic snapshots in the ensemble, our approach could not offset the cost of the profiling phase required to run the algorithm. However, starting with higher source frequencies, which are more widely used in production, we observe that our approach outperforms *No compress* and *Compress single* approaches by 3.2×–8.3× and 1.3×–1.8×, respectively. At higher source frequencies, since the checkpoint size are larger, and the checkpoint generation frequency is high, the *compress single* approach shows higher checkpointing throughput. However, even at higher source frequencies, our approach outperforms the *compress single* case by at least 1.3×.

Next, we evaluate the checkpoint compression and flush schedule generated by our approach at various RTM source

frequencies to understand the proportion of uncompressed, individually compressed, and collectively compressed checkpoints. Figure 7, shows the decision made for each checkpoint number (simulation timestep). For lower source frequencies (2 Hz), the checkpointing throughput (17 GB/s) is lower than the peak PCIe bandwidth of 25 GB/s, due to which our approach favors flushing checkpoints in uncompressed form. However, we observe that as the checkpoint generation rate and checkpoint sizes increase for higher source frequencies, our approach accumulates more number of checkpoint for faster compression and flushing, resulting in up to 58 GB/s higher checkpointing throughput as compared to the approach of compressing individually.

Given that the application is composed of multiple computations, communication, and I/O operations in addition to checkpointing, the speedup observed in checkpointing throughput is not directly translated to the end-to-end application speedup. Based on the set of RTM configurations we evaluated, our proposed accumulated checkpointing approach minimizes the overall application runtime by at least 28% and up to 3.4× as compared to compressing individually and flushing without compression strategies.

2) Impact of Variable GPU Cache Size: We next evaluate the checkpointing throughput as a function of the amount of GPU cache available for checkpointing. Given that the GPU can concurrently run multiple applications using GPU sharing techniques, such as CUDA MPS or MIG, limiting the amount of GPU HBM for caching allows us to understand the performance of our proposal under more constrained setups. Although the total checkpoint size for synthetic benchmarks in uncompressed form is 200 MB, we vary the GPU cache from 64 MB, which is typically the amount of GPU memory unused by application in practice, to 1 GB to study how effectively can our approach utilize larger than available caches. As shown in Figure 8, our approach incrementally performs better with increasing GPU cache sizes, showing up to 4.1× and 2.4× speedup as compared to *no compression* and *individual compression* approaches, respectively. For the GPU cache size of 64 MB, our approach shows ∼3% slower checkpointing throughput as compared to the case of individual compression because compressing individually is the optimal decision for the given cache size, and our oracle profiling could not offset the overheads within the small ensemble runtime.

3) Impact of Variable Checkpoint Generation Rates: We now evaluate the checkpointing throughput as a function of the checkpoint generation rate by varying the checkpoint generation rate for synthetic benchmarks from 40 GB/s (more than the PCIe peak transfer bandwidth) to 400 GB/s (to capture extreme scenarios beyond those observed in Figure 1). As observed in Figure 9, for varying checkpoint generation rates, our approach outperforms the other approaches by at least 2.27× and demonstrates a near-consistent checkpointing throughput. Both the other approaches, however, show lower checkpointing throughput with increasing checkpointing generation pressure because they have lower opportunities to overlap flushes with application kernels.

4) Impact of Variable Compression Ratios: Our next set of experiments evaluates the performance of checkpoint throughput for an increasing compression ratio. This study helps us understand the performance of various compression and flushing approaches for different variations of sparse and dense checkpoints. We vary the compression ratio from 4× to 28×, which imposes higher compressed sizes and therefore more pressure at checkpoint flushing than those observed in the RTM application (Figure 5b). Figure 10 shows that our approach outperforms the other approaches by 3.7× for different compression ratios.

5) Scalability Analysis: We evaluate the performance of checkpointing throughput for an increasing number of processes on the node. When operating at scale, all GPUs compete for both PCIe bandwidth multiplexed through PCIe switches to GPU pairs and the writes on the host memory. Figure 11 shows a weak scalability analysis showing that increasing the concurrent number of processes that capture checkpoints in uncompressed form reduces the checkpointing throughput by up to 57%. However, our checkpoint accumulation-based approach demonstrates excellent scalability by outperforming the no compression and single compression approaches by 3.6× and 4.9× at scale, respectively.

V. Related Work

A. GPU-based Compression Libraries

GPU-based compression libraries are widely used across broad range of applications ranging from scientific simulations to deep learning [3]. Several lossy and lossless compression efforts such as nvCOMP [14], cuSZ [12], Mgard+ [15], and cuZFP [16] focus on accelerating compression throughput and compression ratio by parameterizing the compressor based on various use-cases. However, none of these techniques focus on optimizing compression and/or decompression throughputs for small-sized input datasets.

B. HPC Checkpointing and Data Movement Runtimes

Application-level checkpointing runtimes such as FTI [11], VELOC [9], and SCR [26] require the application to define critical data structures for checkpointing, which are then flushed asynchronously to lower memory and storage tiers. ADIOS2 [10], a unified high-performance I/O framework, is widely used in HPC for efficient data movement across storage tiers. Various other efforts have focused on optimizing collaborative GPU-cache sharing [27], fast cache initialization [20], and modern CXL memory support [28], [29]. While these approaches focus on optimizing I/O through asynchronous multi-level transfers and data lifecycle, they do not support efficient accumulation-based GPU compression for faster throughput. CupCake [30], a recent effort, aims at maximizing compression throughput by fusing tensors during gradient synchronization but is specifically designed for multi-GPU all-gather communication and assumes a consistent compression ratio and input data arrival (checkpoint generation) rate, due to which it cannot be applied to a broader class of applications that we consider in this paper.

VI. Conclusions

In this paper, we address the problem of inefficient asynchronous checkpointing due to the slow compression of small-sized high-frequency checkpoints in simulations of HPC ensembles generated on the GPU. Existing checkpointing runtimes and data movement engines either lack support for GPU-enabled compression or adopt a simplistic approach of instantly compressing checkpoints regardless of the compression overheads. We demonstrate, using three HPC ensembles, that the checkpoint data rate is significantly higher than the PCIe interface, warranting compression, but the small sizes of checkpoints incur huge overheads on application runtime, even when using state-of-the-art compressors such as nvCOMP. To solve this challenge, we designed and developed a checkpoint accumulation approach that leverages the spare capacity of GPU high-bandwidth memory to minimize the overall checkpointing time of simulations in an ensemble. Our approach shows up to 8× higher checkpointing throughput as compared to existing asynchronous checkpointing approaches. Encouraged by these results, in the future, we plan to explore partial compression, remote-GPU-based compression, asynchronous multi-level persistent flushing, and coupled checkpoint production-consumption across workflows.

Acknowledgments

This work is supported in part by the U.S. Department of Energy (DOE), Office of Science, Office of Advanced Scientific Computing Research (under contract DEAC02-06CH11357/0F-60169) and the National Science Foundation (award no. 2106634/2106635).

References

[1] R. S. Canon and A. Younge, "A case for portability and reproducibility of hpc containers," in *Proc. of IEEE/ACM Intl. Workshop on Containers and New Orchestration Paradigms for Isolated Environments in HPC (CANOPIE-HPC)*, 2019.

[2] R.-E. Plessix, "A review of the adjoint-state method for computing the gradient of a functional with geophysical applications," *Geophysical Journal International*, vol. 167, no. 2, pp. 495–503, 2006.

[3] J. Shen, Y. Wu, M. Okita, and F. Ino, "Accelerating gpu-based out-of-core stencil computation with on-the-fly compression," in *Proc. of Intl. Conf. on Parallel and Distributed Computing, Applications and Technologies (PDCAT)*, 2021.

[4] A. Maurya, M. Rafique, T. Tonellot, H. AlSalem, F. Cappello, and B. Nicolae, "Gpu-enabled asynchronous multi-level checkpoint caching and prefetching," in *Prof. of Intl. Symposium on High-Performance Parallel and Distributed Computing (HPDC)*, 2023.

[5] K. Haskins, Q. Wofford, and P. G. Bridges, "Workflows for performance predictable and reproducible hpc applications," in *Proc. of Intl. Conf. on Cluster Computing (CLUSTER)*, 2019.

[6] M. Madhyastha, R. Underwood, R. Burns, and B. Nicolae, "Dstore: A lightweight scalable learning model repository with fine-grained tensor-level access," in *Proc. of Intl. Conf. on Supercomputing (ICS)*, 2023.

[7] S. H. K. Narayanan, T. Propson, M. Bongarti, J. Hückelheim, and P. Hovland, "Reducing memory requirements of quantum optimal control," in *Proc. of Intl. Conf. on Computational Science (ICCS)*, 2022.

[8] M. Yousefi, "Cahn hilliard phase-field simulation by using cuda gpu: Exascale computational materials science," https://github.com/myousefi2016/Cahn-Hilliard-CUDA, 2018, GitHub Repository.

[9] B. Nicolae, A. Moody, E. Gonsiorowski, K. Mohror, and F. Cappello, "VeloC: Towards High Performance Adaptive Asynchronous Checkpointing at Large Scale," in *Proc. of IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2019.

[10] R. Tchoua, J. Y. Choi, S. Klasky, Q. Liu, J. Logan, K. Moreland, J. Mu, M. Parashar, N. Podhorszki, D. Pugmire, and M. Wolf, "ADIOS visualization schema: A first step towards improving interdisciplinary collaboration in high performance computing," in *Proc. of IEEE Intl. Conf. on eScience (eScience)*, 2013.

[11] K. Parasyris, K. Keller, L. Bautista-Gomez, and O. Unsal, "Checkpoint Restart Support for Heterogeneous HPC Applications," in *International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*, 2020.

[12] J. Tian, S. Di, K. Zhao, C. Rivera, M. H. Fulp, R. Underwood, S. Jin, X. Liang, J. Calhoun, D. Tao *et al.*, "Cusz: An efficient gpu-based error-bounded lossy compression framework for scientific data," in *Proc. of Intl. Conf. on Parallel Architectures and Compilation Techniques (PACT)*, 2020.

[13] X. Delaunay, A. Courtois, and F. Gouillon, "Evaluation of lossless and lossy algorithms for the compression of scientific datasets in netcdf-4 or hdf5 files," *Geoscientific Model Development*, vol. 12, no. 9, pp. 4099–4113, 2019.

[14] NVIDIA, "nvCOMP: A CUDA library that features generic compression interfaces." [Online]. Available: https://github.com/NVIDIA/nvcomp

[15] X. Liang, B. Whitney, J. Chen, L. Wan, Q. Liu, D. Tao, J. Kress, D. Pugmire, M. Wolf, N. Podhorszki, and S. Klasky, "Mgard+: Optimizing multilevel methods for error-bounded scientific data reduction," *IEEE Transactions on Computers*, vol. 71, no. 7, pp. 1522–1536, 2022.

[16] Y. Wu, J. Shen, M. Okita, and F. Ino, "Accelerating a lossy compression method with fine-grained parallelism on a gpu," in *Intl. Symposium on Parallel Architectures, Algorithms and Programming (PAAP)*, 2021.

[17] "Reverse time migration," https://github.com/brightskiesinc/Reverse_Time_Migration, GitHub Repository.

[18] R. Underwood, J. Bessac, D. Krasowska, J. C. Calhoun, S. Di, and F. Cappello, "Black-box statistical prediction of lossy compression ratios for scientific data," *The International Journal of High Performance Computing Applications*.

[19] D. A. Beckingsale, M. J. McFadden, J. P. Dahm, R. Pankajakshan, and R. D. Hornung, "Umpire: Application-focused management and coordination of complex hierarchical memory," *IBM Journal of Research and Development*, vol. 64, no. 3/4, pp. 00–1, 2019.

[20] A. Maurya, B. Nicolae, M. M. Rafique, A. M. Elsayed, T. Tonellot, and F. Cappello, "Towards Efficient Cache Allocation for High-Frequency Checkpointing," in *Proc. of Intl. Conf. on High Performance Computing, Data, and Analytics (HiPC)*, 2022, pp. 262–271.

[21] "Using CUDA Stream-Ordered Memory Allocator," https://developer.nvidia.com/blog/using-cuda-stream-ordered-memory-allocator-part-1/.

[22] J. Shen, L. Long, X. Deng, M. Okita, and F. Ino, "A compression-based memory-efficient optimization for out-of-core gpu stencil computation," *The Journal of Supercomputing*, pp. 1–23, 2023.

[23] T. Alturkestani, T. Tonellot, H. Ltaief, R. Abdelkhalak, V. Etienne, and D. Keyes, "MLBS: Transparent Data Caching in Hierarchical Storage for Out-of-Core HPC Applications," in *Proc. of Intl. Conf. on High Performance Computing, Data, and Analytics (HiPC)*, 2019.

[24] Z. Liu, T. Bicer, R. Kettimuthu, and I. Foster, "Deep learning accelerated light source experiments," in *Proc. of Third Workshop on Deep Learning on Supercomputers (DLS)*, 2019, pp. 20–28.

[25] F. Billette and S. Brandsberg-Dahl, "The 2004 bp velocity benchmark," in *EAGE Conference & Exhibition*. European Association of Geoscientists & Engineers, 2005.

[26] A. Moody, G. Bronevetsky, K. Mohror, and B. R. d. Supinski, "Design, modeling, and evaluation of a scalable multi-level checkpointing system," in *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2010.

[27] A. Maurya, B. Nicolae, M. M. Rafique, T. Tonellot, and F. Cappello, "Towards Efficient I/O Scheduling for Collaborative Multi-Level Checkpointing," in *Proc. of Intl. Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, 2021.

[28] M. Arif, A. Maurya, and M. M. Rafique, "Accelerating performance of gpu-based workloads using cxl," in *Proc. of Workshop on AI and Scientific Computing at Scale Using Flexible Computing (FlexScience)*, ser. FlexScience '23. ACM, 2023, p. 27–31.

[29] M. Arif, K. Assogba, M. M. Rafique, and S. Vazhkudai, "Exploiting cxl-based memory for distributed deep learning," in *Proc. of Intl. Conf. on Parallel Processing (ICPP)*, ser. ICPP '22. ACM, 2023.

[30] Z. Wang, X. Wu, Z. Xu, and T. Ng, "Cupcake: A compression scheduler for scalable communication-efficient distributed training," *Machine Learning and Systems (MLSys)*, vol. 5, 2023.