GPU-Enabled Asynchronous Multi-level Checkpoint Caching and Prefetching

Avinash Maurya, M Mustafa Rafique, Thierry Tonellot, Hussain J Alsalem, Franck Cappello, Bogdan Nicolae

To cite this version:

Avinash Maurya, M Mustafa Rafique, Thierry Tonellot, Hussain J Alsalem, Franck Cappello, et al.. GPU-Enabled Asynchronous Multi-level Checkpoint Caching and Prefetching. HPDC'23: 32nd International Symposium on High-Performance Parallel and Distributed Computing, ACM; IEEE, Jun 2023, Orlando, United States. 10.1145/3588195.3592987 . hal-04119928

HAL Id: hal-04119928
https://hal.science/hal-04119928

Submitted on 7 Jun 2023

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution 4.0 International License

GPU-Enabled Asynchronous Multi-level Checkpoint Caching and Prefetching

Avinash Maurya
Rochester Institute of Technology
Rochester, NY, USA
am6429@cs.rit.edu

M. Mustafa Rafique
Rochester Institute of Technology
Rochester, NY, USA
mrafique@cs.rit.edu

Thierry Tonellot
Exploration and Petroleum Engineering Advanced Research Center, Saudi Aramco
Dhahran, Saudi Arabia
thierrylaurent.tonellot@aramco.com

Hussain J. AlSalem
Exploration and Petroleum Engineering Advanced Research Center, Saudi Aramco
Dhahran, Saudi Arabia
hussain.salim.2@aramco.com

Franck Cappello
Argonne National Laboratory
Lemont, IL, USA
cappello@anl.gov

Bogdan Nicolae
Argonne National Laboratory
Lemont, IL, USA
bnicolae@anl.gov

ABSTRACT

Checkpointing is an I/O intensive operation increasingly used by High-Performance Computing (HPC) applications to revisit previous intermediate datasets at scale. Unlike the case of resilience, where only the last checkpoint is needed for application restart and rarely accessed to recover from failures, in this scenario, it is important to optimize frequent reads and writes of an entire history of checkpoints. State-of-the-art checkpointing approaches often rely on asynchronous multi-level techniques to hide I/O overheads by writing to fast local tiers (e.g. an SSD) and asynchronously flushing to slower, potentially remote tiers (e.g. a parallel file system) in the background, while the application keeps running. However, such approaches have two limitations. First, despite the fact that HPC infrastructures routinely rely on accelerators (e.g. GPUs), and therefore a majority of the checkpoints involve GPU memory, efficient asynchronous data movement between the GPU memory and host memory is lagging behind. Second, revisiting previous data often involves predictable access patterns, which are not exploited to accelerate read operations. In this paper, we address these limitations by proposing a scalable and asynchronous multi-level checkpointing approach optimized for both reading and writing of an arbitrarily long history of checkpoints. Our approach exploits GPU memory as a first-class citizen in the multi-level storage hierarchy to enable informed caching and prefetching of checkpoints by leveraging foreknowledge about the access order passed by the application as hints. Our evaluation using a variety of scenarios under I/O concurrency shows up to 74× faster checkpoint and restore throughput as compared to the state-of-art runtime and optimized unified virtual memory (UVM) based prefetching strategies and at least 2× shorter I/O wait time for the application across various workloads and configurations.

CCS CONCEPTS

• **Computing methodologies** → **Distributed computing methodologies**; *Massively parallel and high-performance simulations*; • **Information systems** → **Data management systems**; • **Applied computing** → Physical sciences and engineering.

KEYWORDS

High-Performance Computing (HPC); Graphics Processing Unit (GPU); asynchronous multi-level checkpointing; hierarchical cache management; prefetching

ACM Reference Format:
Avinash Maurya, M. Mustafa Rafique, Thierry Tonellot, Hussain J. AlSalem, Franck Cappello, and Bogdan Nicolae. 2023. GPU-Enabled Asynchronous Multi-level Checkpoint Caching and Prefetching. In *Proceedings of the 32nd International Symposium on High-Performance Parallel and Distributed Computing (HPDC '23), June 16–23, 2023, Orlando, FL, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3588195.3592987

1 INTRODUCTION

Motivation. High-Performance Computing (HPC) applications produce massive amounts of distributed intermediate data during their execution that needs to be checkpointed concurrently by a large number of processes in real-time at scale. This is a fundamental I/O pattern used in a wide range of scenarios [35]: resilience based on checkpoint-restart, producer-consumer patterns in workflows (e.g., simulations coupled with analytics of intermediate checkpoints), reproducibility efforts (e.g., validation of intermediate checkpoints in addition to the end results), revisiting previous states to advance a computation, etc.

Some scenarios like checkpoint-restart involve frequent writes of checkpoints, but only need to retain a few latest checkpoints, which are only occasionally read back for restoring workloads in case of failures. However, many other scenarios require retaining the entire (or most of the) history of checkpoints produced during runtime

Publication rights licensed to ACM. ACM acknowledges that this contribution was authored or co-authored by an employee, contractor or affiliate of the United States government. As such, the Government retains a nonexclusive, royalty-free right to publish or reproduce this article, or to allow others to do so, for Government purposes only.

HPDC '23, June 16–23, 2023, Orlando, FL, USA

© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN 979-8-4007-0155-9/23/06…$15.00
https://doi.org/10.1145/3588195.3592987

and need to read them back with high frequency, sometimes even higher than the write frequency. For example, reproducibility efforts need to write checkpoints that represent intermediate results and/or performance measurements during the runtime of an application, then read them back in the *same order* in which they were produced to check for invariants and/or compare multiple runs to identify where they begin to diverge [33]. As another example, the adjoint state method used for automated differentiation (AD) [31], involves a forward pass that produces a large number of checkpoints and a backward pass that reads them back in *reverse order*. This is popular in a variety of scientific applications, such as climate and ocean modeling, multi-physics, and seismic imaging in the oil industry [32, 43] and deep learning [40]. Binomial checkpointing [39, 41] is another popular approach extensively used by applications such as quantum optimal control to enable execution of memory-bound AD applications [24] where the forward pass generates only a subset of checkpoints, while the backward pass recomputes missing checkpoints by triggering smaller forward passes, which themselves may generate new checkpoints. Such interleavings introduce the need to write and read checkpoints in *any predefined order*. Similarly, coupled workflows that involve producer-consumer patterns rely on intermediate checkpoints to exchange data (e.g., simulations that produce intermediate checkpoints consumed by analytics, data batches produced in real-time and consumed by ML/DL models, etc.) also often need to access the checkpoint history in a predefined order (e.g., based on priority).

These scenarios have two important aspects in common. First, the frequency of writing and reading back checkpoints is much higher than in the case of resilience. Second, there is a large number of processes, each of which needs to read and write large checkpoint sizes. For example, RTM [32] (reverse time migration) scenarios used in the oil industry need to run an entire ensemble of adjoint computation instances that run concurrently and cover a diverse spectrum of wavelength frequencies. There can be hundreds of instances, each of which needs to access a history of checkpoints that reaches up to hundreds of TBs, with a frequency of reads and writes in the order of milliseconds. For example, an instance corresponding to a wavelength frequency of 100 Hz runs on eight Nvidia V100 GPUs (one process per GPU) and generates checkpointing data at 20 GB/s per GPU, amounting to ∼200 TB of checkpointing data over 1300 seconds for all eight processes.

Limitations of State-of-the-Art. Checkpointing and I/O runtimes (e.g., VELOC [25, 26], ADIOS [11], SCR [22], and FTI [29]) often make use of multi-level strategies that leverage hierarchic node-local storage tiers (e.g., GPU memory, host memory, non-volatile memory, and SSDs) to flush the checkpoints to the slower tiers asynchronously. Specifically, the application is blocked only for the duration of the writes to the fastest tier, while other cascading transfers to the slower tier proceed concurrently in the background. However, multi-level strategies that retain high performance and scalability for both writing and reading checkpoints have received comparatively less attention.

One solution to address this gap is to complement multi-level asynchronous flushing with multi-level asynchronous prefetching, which solves the opposite problem: read the checkpoints in advance from the slowest tiers to the fastest tiers concurrently in the background to reduce the duration of read operations when the application issues them. Although prefetching strategies have been extensively studied in the context of I/O subsystems, in our case there are several important challenges. First, writing and reading checkpoints concurrently creates a complex producer-consumer interleaving, which means that independent strategies specifically optimized for checkpointing and prefetching may perform sub-optimally when combined together. Second, for a majority of HPC applications, the read order of checkpoints is deterministic for long periods of the runtime and is often fully known in advance, which can be leveraged to optimize both the prefetching decisions and eviction policies. Third, the rise in the popularity of high bandwidth memory (HBM) [9] available in accelerators, such as GPUs, opens new opportunities to leverage the spare capacity for caching but complicates the interactions with the other node-local storage tiers.

Key Insights and Contributions. To address these challenges, we propose an asynchronous multi-level strategy specifically optimized to leverage synergies between checkpointing and prefetching to minimize the I/O overheads of reading and writing checkpoints under concurrency. The novelty of our proposal is to manage the entire life cycle of the checkpoints by leveraging foreknowledge about the order in which the checkpoints will be consumed in the future using application-level hints. This enables an optimized design of caching, prefetching, and eviction strategies across multiple tiers, including the high bandwidth memory available in GPUs. We summarize our contributions as follows:

(1) We formulate the problem of asynchronous multi-level checkpointing and prefetching on hierarchic storage tiers for scenarios in which checkpoints are frequently read and written in a producer-consumer fashion under concurrency with deterministic patterns (Section 2).
(2) We propose a series of design principles and algorithms for managing a limited cache space on both individual and shared heterogeneous storage tiers of the compute nodes. Specifically, we introduce key ideas, such as checkpoint life cycle based on a state machine that combines flushing and prefetching, score-based eviction of consecutive checkpoints that are the least likely to cause I/O bottlenecks as a group in the future, management of fragmentation (Section 4.1).
(3) We illustrate these design principles and proposed algorithms as an extension to VELOC [25, 26], a production-ready HPC checkpoint-restart library. In this context, we take advantage of CUDA-enabled GPUs to provide fast node-local caching (Section 4.3).
(4) We evaluate our proposed approach in a series of experiments conducted on the Nvidia DGX-A100 [28] platform. Our experiments use two workloads consisting of uniform and variable-sized checkpoints obtain from traces of RTM [2] (reverse time migration), a real-life application used in the oil and gas industry (Section 5). These experiments show an order of magnitude higher read and write throughput compared with state-of-art approaches.

Limitations of the Proposed Approach. Our approach assumes that the checkpoints are monolithic objects for which partial writes

and reads are not allowed. Furthermore, we assume that each checkpoint remains immutable after a write request at least until it is read back. While these assumptions limit the scope of our proposal compared with generic, system-level caching and prefetching solutions, they also enable significant optimization opportunities.

The rest of the paper is organized as follows. In Section 2, we describe the problem of checkpoint and restore of GPU-resident data structures using asynchronous multi-tier I/O operations. In Section 3, we summarize existing efforts and approaches that are related to the scope of this paper. In Section 4, we explain our proposed system design, describe our design principles and gap-aware scoring-based eviction algorithm for efficient checkpoint and prefetch overlaps, and provide the implementation details. In Section 5, we describe our evaluations using diverse applications and system configurations and show a significant improvement in the application performance as compared to the state-of-the-art techniques. Finally, in Section 6, we conclude this paper.

2 PROBLEM FORMULATION

Consider an HPC application that runs on N compute nodes, each of which is equipped with multiple GPUs and features a hierarchy of node-local storage (high-bandwidth GPU memory, host memory, SSDs) and access to remote storage (e.g., parallel file systems [36], object stores [17]). The HPC application is composed of P processes, each of which is assigned to a single GPU of each compute node and is allocated a fixed cache size on each of the node-local storage tiers. While the GPU caches are used separately by the processes, the rest of the caches are shared, competing for the I/O bandwidth at either node level (host memory, SSDs) or globally (remote storage). We assume that the SSD has enough capacity to hold all checkpoints of a single node, while the remote storage has enough capacity to hold all checkpoints of all nodes.

The processes perform computations either independently or in a tightly coupled fashion. Each process produces a history of K intermediate datasets in the GPU memory assigned to it. These intermediate datasets need to be checkpointed and revisited later by the same process, by using the node-local storage hierarchy to swap in and out the checkpoint data as needed. To revisit an intermediate dataset, we need to "restore" it from a checkpoint. The restore order is known to a large extent in advance and specified at runtime by the application. However, the order is simply considered as a hint and the application is allowed to deviate from it at the expense of a potential performance penalty. Once a checkpoint is restored, it is considered *consumed* by the application and will either be discarded or will not be accessed again before any other unconsumed checkpoint in the history.

The following conditions apply: (1) during a checkpoint request, a process blocks until the checkpoint has been fully copied to the GPU cache, after which control is returned back to the application. Meanwhile, the checkpoint is asynchronously flushed to the slower tiers; (2) a process is allowed to read a checkpoint it has previously written even if the asynchronous flushes to slower tiers are still pending; (3) to speed up restore requests, the checkpointing runtime is allowed to prefetch checkpoints based on the restore order specified by each process; (4) to avoid trashing, once a checkpoint has been prefetched on the GPU cache, it can only be evicted after it was consumed; and (5) if a checkpoint was consumed and can be discarded, any of its pending flushes to other tiers are not required to complete successfully.

We aim to design and implement an asynchronous multi-level strategy that minimizes the I/O overheads of both reading and writing checkpoints from the application perspective while satisfying the aforementioned conditions.

3 RELATED WORK

3.1 HPC Checkpoint-Restart

Checkpoint-Restart techniques are traditionally used to provide resilience for HPC applications. Transparent fault tolerance approaches (BLCR [12]) automatically capture the full state of a group of processes, at the expense of generating large checkpoint sizes that cannot be used for any other purpose than resuming the process execution. Application-level fault tolerance approaches (FTI [29], SCR [22], VELOC [25], Canary [5]) require the user to define the critical data structures and reconstruct a consistent state from them on restart. This enables many use cases beyond resilience, including the scenario targeted in this paper. However, such approaches typically focus on checkpointing performance, with comparatively less attention dedicated to restart performance. Both approaches can be complemented by asynchronous multi-level strategies to ensure checkpoint durability. They include optimized flushes from faster to slower tiers [25], but also complementary strategies such as partner replication and erasure coding.

3.2 Workflow Engines

To enable efficient coupling between different HPC application components, several workflow engines have been extensively explored [7, 21, 23]. In addition to addressing the placement and scheduling of application tasks, the increasing data-centric nature of HPC workloads has brought attention to several aspects: complex data redistribution and block-level parallelism (e.g. DIY [23] and Decaf [7]), data staging and publish-subscribe (e.g. DataSpaces [13]), in-situ engines (e.g. ADIOS2 [11]), coroutines (e.g., Henson [21]). While they provide flexible data pipelines, they are typically optimized for coarse-grained I/O between different tasks with different access patterns and caching requirements.

3.3 GPU Checkpointing

Checkpoint-restore techniques have been extended to GPUs, both for fault-tolerance [27, 29] and workload migration [27, 42]. System-level checkpointing libraries such as NVCR [27], transparently record and replay the memory-based CUDA APIs for checkpointing and restoring. Approaches such as CheckFreq [20] and Multi-layered Buffered System [3] also exploit heterogeneous storage tiers. However, none of these efforts consider the case of dedicated cache regions on the GPU and host memory. Furthermore, they typically focus on caching and prefetching of single snapshots rather than collections of snapshots.

3.4 Caching and Prefetching

Cache management for many-core processors and deep hierarchies has been explored in various contexts, e.g., location awareness [19,

30], locality-aware data access control [37], software-defined cache hierarchies [38], and urgency-based prioritization of I/O across caching hierarchies [14]. Several approaches specifically focus on the problem of prefetching for HPC using multi-tiered storage, notably eliminating cache pollution and redundancy [6] and user-level access pattern modeling [34]. Of particular relevance are data pipelines in the context of machine learning. These abstractions (e.g. Nvidia DALI [44]) are used by AI runtimes (e.g., Tensorflow, PyTorch) during model training for sampling the training data. This is often achieved using pseudo-random number generators, whose seed can be exploited to enable foreknowledge about the access order, and, subsequently, optimized prefetching based on reuse distance [8, 16]. Such approaches treat the problem of caching and prefetching separately, without exploiting synergies between the interleaving of producer-consumer checkpointing patterns. While this limits the performance and scalability of such approaches in the scenarios we consider, several ideas are complementary to our proposal and could be used to enhance our proposal.

To our best knowledge, we are the first to consider the case of combined asynchronous multi-level strategies for flushing and prefetching of checkpoints that exploit foreknowledge about the access order.

4 SYSTEM DESIGN

4.1 Design Principles

4.1.1 Dynamic Foreknowledge of Read Order using Prefetching Hints. As mentioned in Section 2, we assume that a large part of the checkpoint read order is known in advance, which enables optimized prefetching. In the ideal case, the order is fixed from the beginning and remains static (e.g., adjoints such as RTM [32] and quantum optimal control [24] produce a fixed number of checkpoints revisited in reverse order). However, many applications have partial foreknowledge of the checkpoint read order, which may evolve based on various runtime conditions. To accommodate the need for dynamic foreknowledge, we propose to maintain a separate restore order queue for each process. The queue can be used by each process to programmatically enqueue as many hints about the future read requests as it desires (potentially covering the whole checkpoint history from the beginning) at runtime. These application hints can be enqueued at any time and can be interleaved with checkpoint and restore requests. For simplicity, we assume that once enqueued, hints cannot be revoked, i.e., the application is not allowed to "change its mind". Note that the application does not need to provide the hints directly; they can also be provided by higher-level I/O middleware, e.g., by using predictors [6]. Furthermore, since the hints have an advisory role, i.e., the read requests can deviate from the hints, our proposal is not limited in its applicability to the general case.

4.1.2 Shared Multi-level Cache Hierarchy for Flush/Prefetch. Asynchronous flushing of checkpoints over a multi-level cache hierarchy involves cascading data transfers from the fastest (GPU memory) tier to the slowest (SSD or parallel file system) tier, while asynchronous prefetching of checkpoints follows the opposite path. A naive strategy could simply manage a separate space on each tier to handle the flushing and prefetching strategies separately. Such an approach is simple to design and implement because it requires minimal coordination between flushing and prefetching. However, it has two major disadvantages. First, it leads to high under-utilization of the scarce cache space (especially GPU memory). Second, it leads to uncontrolled competition for shared I/O bandwidth between the flushing and prefetching. Both of these aspects result in significant degradation of I/O performance and scalability. Therefore, a key design choice in our approach is to maximize the cache sharing and control the competition between flushing and prefetching across as many processes as possible, allowing the application to run prefetches in parallel with the flushes.

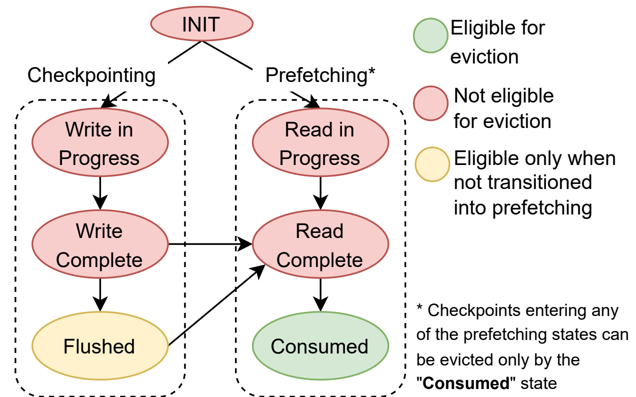


Figure 1: Life cycle of each checkpoint represented as a finite-state machine.

4.1.3 Unified Flush/Prefetch Support using Finite-State Machine Life Cycle. Even with improved cache utilization due to maximized sharing across the cache tiers, the available capacity on each tier is typically limited and therefore evictions to slower tiers are unavoidable. However, unlike the case of a traditional cache where the benefits of not evicting an object are well-understood (e.g., evict the least frequently used object), the interleaving of flushes and prefetches mandates a new model to uniquely identify the validity of evictions across the cache tiers for a producer-consumer pattern. To this end, we introduce a life cycle for every checkpoint instance on all cache tiers. The life cycle is illustrated in Figure 1 and is based on a finite-state machine. Specifically, every new checkpoint is born into the *INIT* state and transitions into either the *Checkpointing* path or the *Prefetching* path, depending on whether it serves a checkpoint or restore request. In the *Checkpointing* path, the checkpoint waits in the *Write in Progress* state until the pending transfers from the faster (or the application buffer) to the slower cache tier have finished. Then, it transitions into the *Write Complete* state. If no pending restore or prefetch request exists for the checkpoint, then it transitions further into the *Flushed* state, which makes it eligible for eviction. If the checkpoint was successfully flushed and evicted from the cache, then a subsequent prefetch hint will trigger the *Prefetching* path. In this case, the checkpoint waits in the *Read in Progress* state until the pending transfers from the slower to the faster cache tier are complete, which transitions it into the *Read Complete* state. Otherwise, the checkpoint is either in the process

of being flushed or was already flushed but not evicted yet. In this case, the checkpoint transitions directly into the *Read Complete* state. Regardless of how this state was reached, the checkpoint will not be evicted until the restore request has finished copying the data to the application buffer, which triggers a final transition into the *Consumed* state, at which point the checkpoint is again eligible for eviction. Using this approach, individual checkpoints are aware of any concurrent flushes and prefetches that affect them across all cache tiers, thereby mitigating the competition and improving the caching utilization for the combined checkpoint and restore request interleaving.

4.1.4 Contiguous Cache Buffer Pre-allocation and Pinning. Allocating memory on the cache tiers on-demand to hold the checkpoints is expensive. For some types of cache tiers, the allocation cost is higher than the transfer cost. For example, on Nvidia A100 GPU HBM, memory allocation speed and GPU-to-GPU transfer speed are about 1 TB/s. However, pinned memory can be allocated on the host cache at about 4 GB/s, whereas the GPU cache-to-host cache transfer speed is about 25 GB/s. Therefore, in our design, we opt to pay the expensive allocation cost (and, where applicable, the pinning cost) upfront at the initialization time by using a single contiguous cache buffer on each node-local storage tier per process, which is reused for the whole application runtime. It is worthwhile to note that such a reuse strategy can also be complemented by techniques to hide the overhead of memory pinning asynchronously, as described in [18]. However, this is outside the scope of this paper.

4.1.5 Fragmentation Mitigation of Cache Buffers. When the checkpoint sizes are identical, the management of the cache buffer is straightforward: each eviction creates a gap that is large enough to accommodate a new checkpoint. However, when the checkpoint sizes are not identical (e.g., because they are compressed) it is difficult to avoid fragmentation. For example, compressed checkpoint sizes are difficult to predict in advance, and even if it were possible to predict them, the interleaving of producing and consuming checkpoints under concurrency will lead to many small contiguous gaps in the cache buffer that cannot be individually used to hold a new checkpoint, even if their combined size would be sufficient to do so. To solve this problem, we propose a fragmentation mitigation approach that is closely coupled with the eviction strategy: we take into consideration not only what checkpoints are the best candidates for eviction, but also how these checkpoints are bordering the gaps around them. This leads to interesting trade-offs. For example, a small checkpoint may not be a good candidate for eviction by itself but becomes so if it is surrounded by large gaps. Furthermore, evictions of single checkpoints may not be sufficient to accommodate new writes of large checkpoints. Thus, we have to determine the value of evicting an entire contiguous region in the cache tier that is potentially made of interleaving of gaps and checkpoints. To this end, we introduce a gap-aware eviction strategy that coalesces gaps and checkpoints during evictions to form a single contiguous gap where we can fit a new checkpoint.

4.1.6 Gap-aware Eviction Policy based on Sliding-window Scoring. We propose a strategy to identify the value of contiguous regions considered for evictions that works as follows: we start by assigning each checkpoint a score that indicates the estimated overhead on the future restore requests it causes if it were allowed to be evicted. This score is derived from several aspects. First, we obviously prefer consumed checkpoints, followed by flushed checkpoints (which are immediately evictable). If there is a flush pending on a checkpoint and no prefetch in progress, then we prefer the checkpoint whose estimated flush completion time is the smallest based on its size and the bandwidth between the cache tiers. This minimizes the wait time until the transition into the *Flushed* state makes the checkpoint evictable. Third, we prefer to evict a checkpoint for which there is either no prefetching hint available, or otherwise, its prefetching hint is the farthest away from the head of the restore order queue, i.e., it will be restored last. This excludes any checkpoint for which a prefetch has already started and thus, according to its life cycle, it is not evictable. For the purpose of scoring, gaps are treated like regular checkpoints and have the highest eviction priority. Once we obtained a score for each checkpoint, we combine the scores of consecutive gaps and checkpoints until a window is obtained that is large enough to accommodate a new checkpoint. Then, we slide this window of variable size over the entire cache buffer and retain the best performer. This process is detailed in Section 4.2.

```
1  VELOC_Init(MPI_World, config_file);
2  for (int ver=num_ckpts-1; ver >= 0; ver--)
3      VELOC_Prefetch_enqueue(ver);
4  VELOC_Mem_protect(1, ptr, size);
5  for (int ver=0; ver < num_ckpts; ver++) {
6      compute_and_compress<<<nb, nt>>>(ptr);
7      VELOC_Checkpoint(ckpt_name, ver);
8  }
9  VELOC_Prefetch_start();
10 for (int ver=num_ckpts-1; ver >= 0; ver--) {
11     size_t size = VELOC_Recover_size(ver, 1);
12     VELOC_Mem_protect(1, ptr, size);
13     VELOC_Restart(ver);
14     uncompress_and_compute<<<nb, nt>>>(ptr);
15 }
```

Listing 1: Example of prefetching API usage: Application reading back checkpoints in reverse order (new APIs highlighted with blue background).

4.2 Gap-aware Eviction Policy

In this section, we propose an algorithm for the gap-aware eviction strategy introduced in Section 4.1. As outlined in Algorithm 1, this strategy aims to identify the set of consecutive fragments (represented either by checkpoints or gaps in the cache buffer) such that: (1) they form a gap in the cache buffer that is large enough to write a new checkpoint $ckpt_new$ of size $size_new$; (2) the impact, measured as blocking duration, on subsequent restore requests is minimized. Note that writing a checkpoint into the cache buffer can be triggered either by a checkpoint request or a prefetch operation.

We assume that the cache buffer keeps an allocation table A for its checkpoints with tuples of the form $\langle ckpt_id, offset, size, state_ts \rangle$, where $ckpt_id$ denotes its unique id, $offset$ and $size$ mark segments of the cache buffer occupied by the checkpoint, and $state_ts$ is the estimated time left at which the checkpoint will reach an evictable state in its life cycle (either *Flushed* or *Completed*). We also assume that we have access to the restore order queue, denoted P.

Algorithm 1: Score-based look-ahead cache eviction.

Input : $\langle ckpt_new, size_new \rangle$: checkpoint to written to the cache buffer and its size; A: the allocation table of the cache buffer, holding tuples $\langle ckpt_id, offset, size, state_ts \rangle$; P: restore order

Output: index of $ckpt_target$ in A

```
1  Function score_eviction(⟨ckpt_new, size_new⟩, A, P):
2      n = |A|; j = 0; window = 0
3      p_score = 0; s_score = 0
4      min_p_score = ∞ ; max_s_score = 0
5      r_start ← 0; r_end ← 0
6      for i ← 0 to n and j < n do
7          if i > 0 then
8              p_score− = A[i − 1].state_ts
9              s_score− = P[A[i − 1].ckpt_id]
10             window− = A[i − 1].size
11         while window < size_target and j < n do
12             p_score+ = A[j].state_ts
13             s_score+ = P[A[j].ckpt_id]
14             window+ = A[j].size
15             j++;
16         if window ≥ size_new and
            (p_score < min_p_score or (p_score ==
            min_p_score and s_score > max_s_score)) then
17             min_p_score = p_score
18             max_s_score = s_score
19             r_start = i
20             r_end = j
21     start = A[r_start].offset
22     free_size = 0
23     for i ← r_start to r_end do
24         while A[i].state_ts ≠ 0 do
25             wait until A[i] evictable
26         free_size+ = A.erase(i)
27     if free_size > size_new then
28         insert_gap(start+size_new, free_size-size_new)
29     state_ts = predict_evictable(A, size_new, B)
30     return A.insert(⟨ckpt_new, start, size_new, state_ts⟩)
```

The algorithm uses a sliding window approach to iterate over all fragments of the cache buffer. Specifically, a *window* is a sequence of consecutive fragments in A, with the first denoted i and the last denoted j. Each window is assigned two scores: (1) the p_score, which measures the estimated total blocking duration until all fragments in the window are evictable; (2) the s_score, which measures the sum of the *prefetch distances* of each checkpoint belonging to the window. The prefetch distance is the number of elements in P between the position of a checkpoint j in P and the head of P and measures how far a checkpoint is from being prefetched. Ideally, we want to minimize (1) and maximize (2) simultaneously. However, if this is not possible, we prioritize (1) with the assumption that waiting and doing nothing while evictions become eligible causes a more negative impact than suboptimal decisions for (2).

Once we identify the window with the best score (delimited by r_start and r_end), we wait for its checkpoints to become evictable, then we remove them from A, and we insert a new tuple in A corresponding to the new checkpoint (which overwrites the window). The estimated time until the new checkpoint becomes evictable in the future is based on its state in the life cycle: (1) if the checkpoint is being flushed, then we estimate how long this will take; (2) if the checkpoint is being prefetched, then we estimate how long it will take until the checkpoint will be consumed. The estimation is based on several factors: the size of the checkpoint $size_new$, bandwidth B between the caching tiers, and other enqueued flushes and prefetches in A that compete for bandwidth. It is computed by $predict_evictable$.

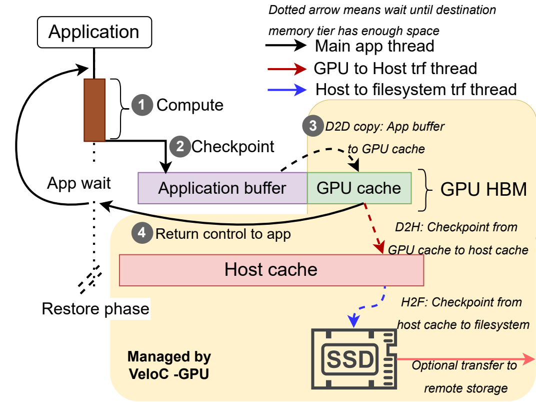


Figure 2: Workflow of checkpoint operations.

By adjusting both scores every time we increment i and j (rather than recalculating the score for each window), we can achieve an optimal $O(N)$ complexity, where N is the number of checkpoints cached on the given memory tier. This aspect is important, because a long response time may delay the data transfer and thus lead to lower overall throughput, especially when fast tiers are involved.

4.3 Implementation

We implemented our approach in VELOC [25], a production-ready, multi-level checkpointing framework. We added support for GPU-aware multi-level flushing and prefetching with the design principles noted in Section 4.1. The default VELOC API exposes basic primitives to declare the memory regions that must be checkpointed (`VELOC_Mem_protect`), write the defined memory regions into a new checkpoint ID (`VELOC_Checkpoint`) and read the memory regions from a given checkpoint ID (`VELOC_Restart`). We extend the VELOC API with two new primitives: (1) `VELOC_Prefetch_enqueue`, which enables a process to provide a hint about the next checkpoint ID it intends to restore in the future; and (2) `VELOC_Prefetch_start`, which allows the application to indicate when to start prefetching. The latter is helpful (but optional) in scenarios when it is desirable to delay the asynchronous prefetches in order to avoid unnecessary interference with the asynchronous flushes. For example, this would be the case when the application writes the checkpoints in a forward pass, then reads them back in a backward pass, as illustrated in Listing 1.

4.3.1 Asynchronous Flushing and Prefetching. We developed a multi-threaded approach that serves checkpoint and restore requests

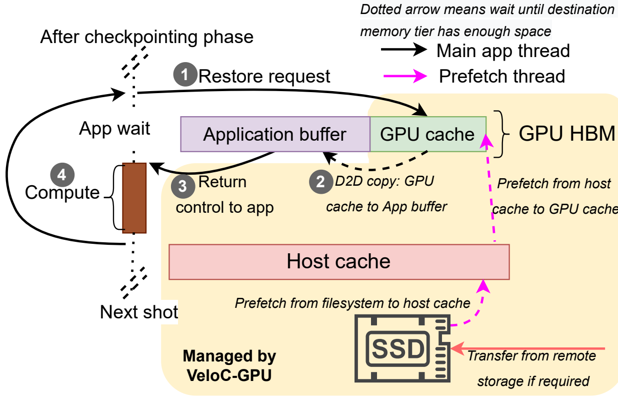


Figure 3: Workflow of prefetch and restore operations.

issued in the main application thread (T_{main}) using dedicated background threads for asynchronous I/O (GPU to host cache flushes: T_{D2H}, host cache to local SSD flushes: T_{H2F}, prefetches on all tiers: T_{PF}). Additionally, since the application processes may use the default CUDA stream, our implementation creates a separate set of dedicated CUDA streams, which are then used by the background threads. Using this approach, transfers between different memory layers, such as GPU-to-GPU, GPU-to-host, and host-to-GPU can be efficiently overlapped thanks to dedicated copy engines employed by the Nvidia GPUs and the CUDA driver.

4.3.2 Workflows. Next, we detail the workflows of the checkpoint/flush and prefetch/restore requests in Figure 2 and Figure 3, respectively. We start by enqueuing the prefetching hints (Line 3 of Listing 1) after completing the initialization phase. Then, the memory regions that need to be checkpointed as protected using `VELOC_Mem_protect` (Line 4). During the checkpointing phase (Lines 5-8), `VELOC_Checkpoint` (Line 7) blocks until the protected memory regions are copied to the GPU cache using the main thread T_{main}, at which point T_{D2H} is notified to begin flushing. In turn, immediately after flushing finished to the host cache, T_{H2F} is notified to flush to the local SSD (and further to a parallel file system if persistence is required). Prefetching is initiated by the T_{PF} thread, which respects the finite-state machine described in Figure 1 for each tier. Specifically, if the requested checkpoint is present on the GPU cache, `VELOC_Restart` (Line 13) restores the declared memory regions from the checkpoint and marks its state as *Read Consumed*. Otherwise, `VELOC_Restart` blocks until T_{PF} promotes the checkpoint to the faster tiers as follows: space is reserved for it on the faster tier (by waiting for evictions if necessary) and marked as *Read in Progress*, then the checkpoint is copied to the reserved space on the faster tier and marked as *Read Completed*, while the original is marked as *Read Consumed*.

5 EVALUATION

5.1 Experimental Setup

We evaluated our approach on the ThetaGPU HPC testbed [4], which consists of 24 Nvidia DGX A100 nodes [28]. 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 per drive), and eight Nvidia A100 Tensor core GPUs (for a total of 320 GB HBM2e memory). The nodes have access to a 10 PB Lustre parallel file system [36] that is accessible through a POSIX mount point. On each node, the eight A100 GPUs are interconnected using 6 NVSwitches. Furthermore, they can access the host memory through a PCIe Gen 4 interface. The peak unidirectional device-to-device and pinned device-to-host (and vice versa) bandwidths on each GPU are 1 TB/s and 25 GB/s, respectively. Two GPUs share the same PCIe 4 link to the host memory, which has two implications. First, only four of the eight NUMA domains are directly accessible from the GPUs. Second, there is a contention for the device-to-host memory bandwidth. We use up to four nodes, i.e., 32 GPUs, for our experiments.

5.2 Compared Approaches

5.2.1 Adaptable Input Output System version 2 (ADIOS2). This is a data management runtime for HPC applications that aims to provide scalable I/O on HPC infrastructures [11]. It provides a set of reusable and extendable components for managing data presentation and transport engines. In our comparison, we use the `BP5` transport engine, which enables deferred (asynchronous) I/O to the fast Gen 4 NVMe storage by first buffering in the main memory. We use the `adios2::MemorySpace::CUDA` feature, which instructs ADIOS2 that the data to be checkpointed resides on the GPU memory. Unlike our approach, ADIOS2 does not use the spare GPU capacity for caching and prefetching and instead performs the on-demand movement of data between the GPU memory and host memory. We choose ADIOS2 because it is representative of a multi-level asynchronous I/O framework that handles checkpointing of GPU data structures on-demand, which is a widely used approach in the production settings of the HPC domain.

5.2.2 Nvidia Unified Virtual Memory (UVM). The UVM is a popular memory management technique introduced by Nvidia to transparently manage data movement across GPUs and host memory tiers in a cache-coherent fashion. Data movement across these tiers is governed by various heuristics, such as hardware-based counters, memory access pattern, amount of available device memory, prefetching and eviction policies, page-fault replay policy, and memory device configured by the application [1, 10]. In addition to transparent data movement, UVM provides several other benefits, such as on-demand memory allocation, dynamic expansion and reduction of UVM memory allocated on the device HBM, and zero-copy data access to achieve low latency while eliminating the overheads of copying data between memory tiers. We choose UVM because it is representative of a system-level transparent caching and prefetching solution that adopts a combination of best practices introduced by state-of-the-art over time.

Note that just like our proposal, UVM enables the application to provide hints about the memory access pattern. We provide these hints by exploiting the `cudaMemAdvise` and `cudaMemPrefetchAsync` primitives. Specifically, we make use of two flags that are relevant for `cudaMemAdvise`: (1) `cudaMemAdviseSetPreferredLocation`, which advises the CUDA driver to move the checkpoint to either the host memory (in case of flushing) or to the GPU memory (in case of prefetching). Once a checkpoint is read by the application,

Table 1: Compared approaches using different number of prefetching hints.

Notation	Prefetch Hints
`No hints, ADIOS2`	0
`No hints, UVM`	0
`No hints, Score`	0
`Single hint, UVM`	1
`Single hint, Score`	1
`All hints, UVM`	All
`All hints, Score`	All

we again set its preferred location to the host memory that enables the CUDA driver to immediately evict consumed checkpoints, which otherwise would have been retained due to the default LRU eviction policy; (2) `cudaMemAdviseSetAccessedBy`, which enables faster access for specific processes. In addition, we initiate asynchronous prefetching using `cudaMemPrefetchAsync` in order to make sure that the checkpoints that are already present on the device memory but not yet consumed by the application are not evicted prematurely. Since UVM does not expose a mechanism to block further prefetch operations until the previously prefetched memory regions on the device are consumed by the application, we explicitly control the prefetch request by tracking the amount of device memory consumed (when prefetching operations load checkpoint data for application consumption) and released (when application consumes the prefetched checkpoints). During the restore phase, if the amount of memory consumed by the prefetch operations exceeds the amount of the UVM device cache, the prefetching thread waits until the application consumes checkpoints from the UVM device cache. In addition to avoiding page thrashing, this also allows efficient use of the limited host-to-device interconnect bandwidth. Thanks to these optimizations, we aim for a fair comparison that informs the CUDA driver about the foreknowledge of the access pattern of the checkpoint data.

5.2.3 Our Proposal (Score). We compare the state-of-the-art approaches mentioned above with our proposal, whose key design principles and implementation details are outlined in Section 4.1 and Section 4.3, respectively. We denote our proposal `Score`, after the gap-aware eviction policy based on the sliding-window scoring algorithm introduced in Section 4.2.

5.2.4 Number of Prefetching Hints. We vary the degree of foreknowledge by varying the amount of prefetching hints provided by the application as follows: unlimited (the restore order is fully known in advance); one prefetching hint at a time (the application specifies what checkpoint to prefetch for the next iteration at the beginning of the current iteration) and no prefetching hints (corresponding to direct read requests where foreknowledge is not available). By varying the degree of foreknowledge, we study the impact of our design principles even in the case of the scarcity or absence of prefetching hints. The combination of all the compared approaches is summarized in Table 1.

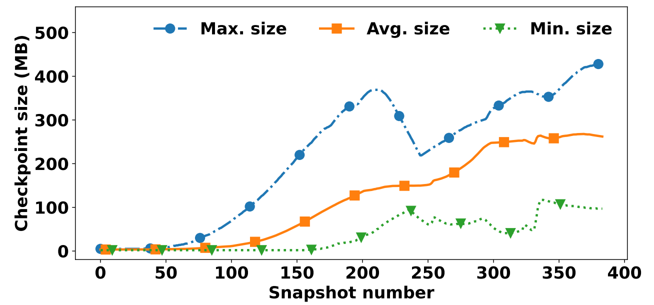


Figure 4: Size distribution of 32 RTM snapshots.

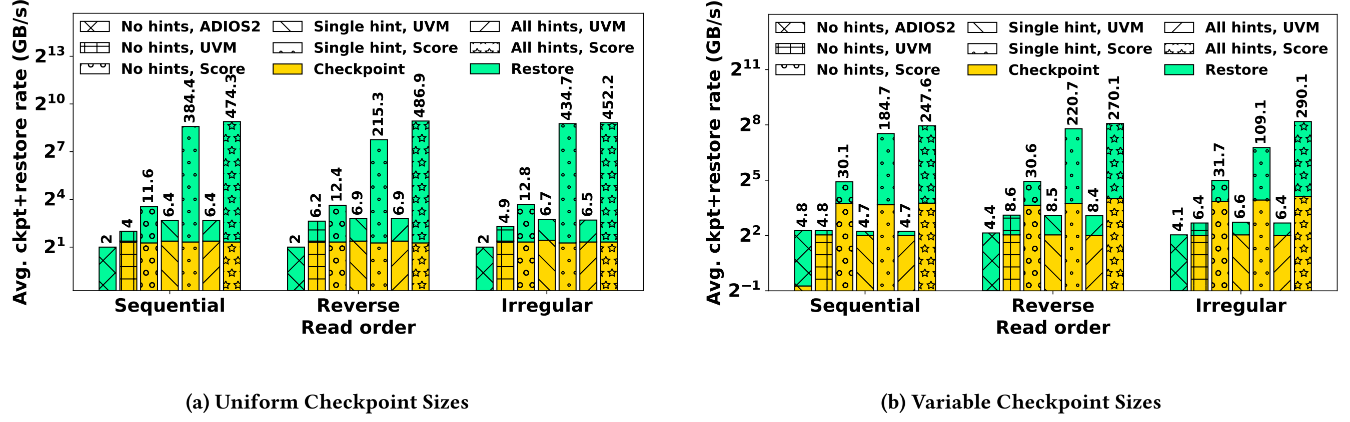
5.3 Evaluation Methodology

We evaluate each approach used in our comparison using the scenarios and metrics described below.

5.3.1 Application Benchmark – RTM. As an application, we consider the case of Reverse Time Migration (RTM) [15], an HPC adjoint computation commonly used in the oil and gas industry. Specifically, RTM is used to generate subsurface images from seismic data. This process involves wave fields generated in the forward pass, which are checkpointed and consumed in a predefined order during the backward pass to cross-correlate and form the subsurface image. RTM can run either in embarrassingly parallel mode (many shots in parallel, each on a different GPU) or tightly coupled mode (one shot across multiple GPUs with synchronization steps at each iteration). In this work, due to a large number of compared configurations, we designed a series of benchmarks that emulate the behavior of RTM based on traces obtained during large-scale production runs that record the checkpoint sizes. Specifically, our benchmarks run trivial iterations, by sleeping to simulate computations between consecutive checkpoints or restore operations, but generate the exact same checkpoint sizes as in the RTM traces.

5.3.2 Restore Order Patterns. We consider a set of distributed processes, each of which performs a single forward pass that writes a checkpoint at each iteration, then starts the prefetching, and then performs a backward pass that reads a checkpoint at each iteration. This corresponds to the example used in Listing 1. We refer to such a run as a *shot*. In this context, we evaluate all three restore order patterns discussed in Section 1: `Sequential` (the backward pass consumes the checkpoints in the same order they were generated during the forward pass), `Reverse` (the backward pass consumes the checkpoints in the reverse order than they were generated during the forward pass), and `Irregular` (the backward pass consumes the checkpoints in a random but predetermined order).

5.3.3 Traces and Distribution of Checkpoint Sizes. In a typical shot, RTM compresses the image checkpoints by default during the forward pass, which leads to variable checkpoint sizes not only across different iterations but within the same iteration among different processes. The average compression ratio of these checkpoints is ~30×. From over 1600 shot traces, 32 representative shots were chosen for benchmarks on up to 32 GPUs (4 compute nodes). Figure 4 shows the minimum, maximum, and average sizes per snapshot for 384 snapshots. The aggregated size of checkpoints on each shot



(a) Uniform Checkpoint Sizes

(b) Variable Checkpoint Sizes

Figure 5: Average checkpoint+restore throughput across 8 GPUs for uniform (Fig. 5a) and variable (Fig. 5b) sized checkpoints when we WAIT for checkpoints to be flushed from the caches before the restore phase. Higher is better. Y-axis is in log scale.

ranges between 38 GB to 50 GB. From a caching perspective, variable checkpoint sizes are difficult to handle because they introduce fragmentation. We complement this scenario with another that uses uniform checkpoint sizes: 128 MB per checkpoint, which roughly corresponds to the 50 percentile distribution of the 1600 RTM shot traces, leading to 384 checkpoints amounting to 48 GB per GPU. This enables us to study the effectiveness of our approach in the absence of cache fragmentation as well.

5.3.4 Multi-level Cache Tiers. Unless noted otherwise, we use the following cache configurations: for each process, we reserve 4 GB GPU memory (i.e., 10% of the capacity) as a device-level cache and 32 GB main memory (i.e., 25% of the total capacity consumed by 8 processes) as a pinned host cache. Additionally, we use a node-local SSD as the slowest tier, which is shared by all processes co-located on the same compute node. The SSD can accommodate all checkpoints produced on the same compute node during the entire forward pass of RTM. If this is not enough for other applications, an external storage tier (e.g., parallel file system) shared by all compute nodes can be used to store the checkpoint data.

5.3.5 Performance Metrics. We use the following performance metrics in our evaluation:

(1) Write and read throughput observed by the application for two cases. First, wait for checkpoints to be flushed from the caches before starting the restore phase. Second, the restore phase starts immediately after the checkpointing phase.
(2) Read throughput observed by the application at different iterations.
(3) Prefetch distance, i.e., the number of successive checkpoints successfully prefetched in advance at the moment when a given checkpoint is restored to measure the effectiveness of the prefetch strategy.

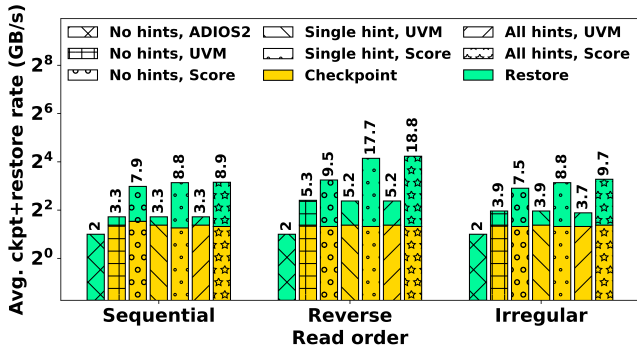
These metrics are obtained in a variety of settings, i.e., single node vs. multiple nodes, tightly coupled (barrier after each iteration) and embarrassingly parallel (no synchronization after each iteration) scenarios, and variable compute complexity (simulated sleep duration) of each iteration and variable GPU cache sizes.

5.4 Performance Results

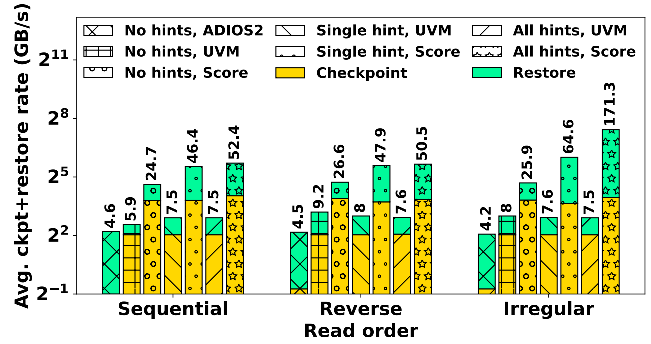
5.4.1 Checkpointing Throughput. Our first set of experiments measures the average checkpointing and restoring throughput as observed by the application (total checkpoint size divided by blocking time of checkpoint and restore operations respectively) over the entire shot. Note that the data movement (flush and prefetch) across slower tiers is asynchronous, therefore blocking time only includes the duration of writes or reads from the GPU/UVM cache and any delays due to evictions or prefetches, which is most relevant from the application perspective. We run our benchmarks on a single node (8 GPUs) in embarrassingly parallel mode and fix the checkpoint frequency to 10 ms to be consistent with the RTM behavior.

5.4.2 Restore Phase Waits for Checkpoint Phase. We first consider the case when the checkpoint and restore phases are sequential, i.e., the restore phase waits for the checkpoint phase to finish flushing to the slowest tier. This scenario captures the cases where checkpoints from the forward pass are persisted for reproducibility, post-hoc, or in-transit analysis. Figure 5a and Figure 5b show the average checkpoint and restore overheads associated with uniform and variable-sized checkpoints respectively. As expected, the performance of the ADIOS2 approach across all restore patterns for uniform and variable checkpoint sizes is the slowest due to the lack of a dedicated device cache tier.

As compared to the optimized UVM approach, our proposed score-based approach shows 2×–74× and 1.6×–52× faster checkpoint+restore rates for uniform and variable-sized checkpoints, respectively. Note that even without any restore-order hints, our approach outperforms the optimized UVM approach by at least 2×, thanks to directly evicting the consumed checkpoints from the device cache (if already consistently captured on slower memory tiers), as opposed to UVM's approach of migrating the checkpoints before eviction. We observe similar checkpointing throughput for UVM and score-based approaches due to– (1) the problem of slow host cache initialization (mapping and registration of host cache for faster direct-memory transfers) as studied in [18]; and (2) the shot contains 48 GB worth of checkpoints, which does not fit in 4 GB



(a) Uniform Checkpoint Sizes



(b) Variable Checkpoint Sizes

Figure 6: Average checkpoint+restore throughput across 8 GPUs for uniform (Fig. 6a) and variable (Fig. 6b) sized checkpoints when we DO NOT WAIT for checkpoints to be flushed before the restore phase. Higher is better. Y-axis is in log scale.

GPU cache and 32 GB host cache, triggering waits for evictions to the SSD towards the end of the shot.

An interesting observation when waiting for checkpoints to get flushed before restoring is that all checkpoints residing in either of the cache tiers transition to the *Flushed* or the *Read Complete* state, thereby, minimizing waits due to evictions. However, this approach incurs an additional overhead of waiting for cache flushes, which amounts to 70 seconds per rank (effective flush rate per rank = 48 GB/70s ≈ 685 MB/s). Since the cache flush rate is uniform for all approaches, we do not show this additional flush time in the graphs to better highlight the impact of our approach on checkpoint and restore operations exclusively.

We also observe that the average checkpoint rate of variable-sized checkpoints (Figure 5b) is higher than the corresponding uniform-sized checkpoints (Figure 5a) for all read-orders in the score-based approach. This is because smaller-sized checkpoints at the beginning of the shot (Figure 4) allow for faster evictions by flushing smaller checkpoints between consecutive iterations (of 10ms), and the disparity in checkpoint sizes across multiple processes reduces competition on shared resources such as PCIe lane, host memory allocated on the same NUMA domain, and the file system. On the contrary, we observe that the restore rates of uniform-sized checkpoints are faster compared to the variable-sized scenario. This is because uniform-sized checkpoints do not cause fragmentation on either of the cache tiers, but the variable-sized checkpoints undergo fragmentation and adopt our proposed gap-aware eviction policy based on sliding-window scoring (Section 4.1) which may block the prefetch/restore operation until a contiguous region consisting of multiple checkpoints becomes *evictable*. Nonetheless, we note that the restore rate of our proposed score-based approach is at least 2× faster than the optimized UVM-based approach and state-of-art ADIOS2 framework (note the representation of the y-axis in log scale).

5.4.3 Restore Phase Immediately Follows Checkpoint Phase. Next, we consider the case when the restore phase begins immediately after the checkpointing phase. This illustrates scenarios such as adjoint computations where the overall job runtime needs to be

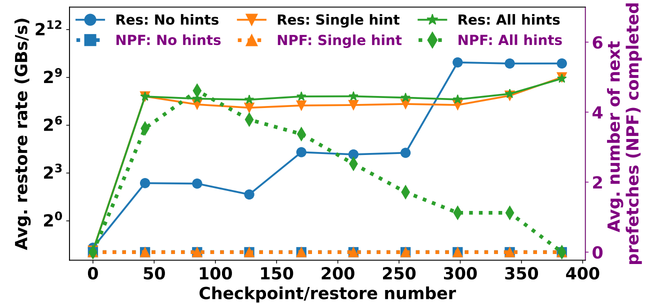
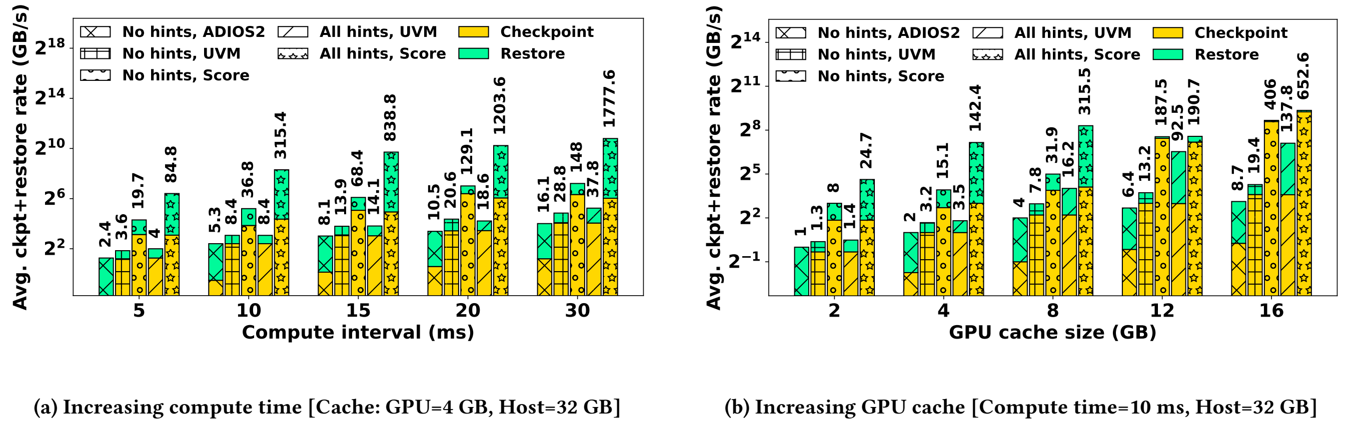


Figure 7: Restore rate and number of next prefetches completed for different timesteps for uniform sizes checkpoints using score-based approach.

minimized and the checkpoints need not be persisted. Similar to the case of waiting for flushes after the checkpoint phase, the ADIOS2 approach is the slowest across all approaches for all read orders and checkpoint sizes.

As observed in Figure 6, our score-based approach outperforms the optimized UVM-based approach by 1.8×–3.6× and 2.9×-22.8× for uniform (Figure 6a) and variable (Figure 6b) checkpoint sizes, respectively. Again, we observe consistent checkpoint throughput across all approaches for various restore order patterns because of slow cache initialization and waiting for evictions (flush to the file system) when both cache tiers have been fully utilized. Note that in this case, we only wait for flushes to the file system to capture the checkpoints that exceed the combined memory capacity of the device and host cache tiers.

Unlike the previous case of waiting for checkpoints to get flushed before starting the restore phase where checkpoints residing on the cache belong to either *Flushed* or *Read Complete* states, in this case, the checkpoints residing on the cache tiers can belong to any state of the checkpoint life cycle (Figure 1). Transitioning through the state life cycle leads to longer cache eviction times in the restore phase, due to which the restore rate across various read orders and



(a) Increasing compute time [Cache: GPU=4 GB, Host=32 GB]

(b) Increasing GPU cache [Compute time=10 ms, Host=32 GB]

Figure 8: Impact of varying compute interval (Fig. 8a) and GPU cache size (Fig. 8b) on I/O throughput for variable-sized checkpoints and irregular read order on 8 GPUs. Higher is better. Y-axis is in log scale.

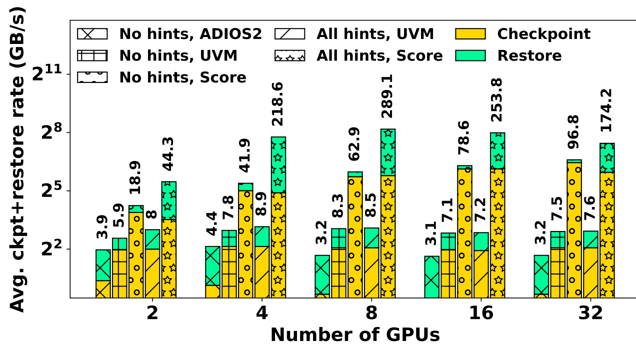
a varying number of hints are magnitudes of order slower than the corresponding restore rates observed for the case of waiting for checkpoints to be flushed before starting the restore phase. However, this approach achieves faster overall execution time by eliminating the additional wait time (~70s) to flush checkpoints in between the checkpoint and restore phases.

Experiments with variable-sized checkpoints (Figure 6b) show higher restore rates as compared to uniform-sized checkpoints (Figure 6a) due to faster evictions for smaller checkpoint sizes and lower congestion on shared resources because of differences in checkpoint sizes across multiple processes. The case of irregular read order is the most challenging scenario across all executions because of– (1) large fragmentation management overheads due to variable sizes of checkpoints, (2) complex transitions in the state life cycle due to overlapping checkpoint and restore operations across various cache tiers, and (3) non spatio-temporal read-access due to irregular read order. However, even with these complexities, our proposed score-based approach achieves 3.7× faster checkpoint and 38× faster restore rates as compared to the UVM approach.

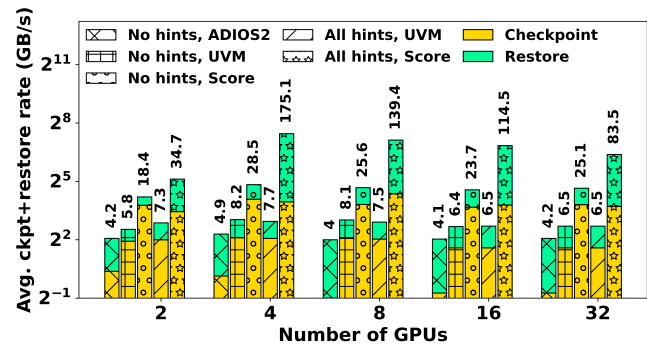
Since this case of starting the restore phase immediately after the checkpoint phase introduces performance challenges due to fragmentation and complex transitions in the state life cycle, we run the remainder of the experiments without waiting for checkpoints to complete before starting the restore cycle.

5.4.4 Prefetch Distance and Restore Rate. To explain the observed restore throughput better, we study the effectiveness of the prefetches by measuring the prefetch distance, i.e., the number of successor checkpoints successfully prefetched based on the hints at the moment of a read request. The intuition behind this metric is as follows: the more successor checkpoints are available on the GPU cache, the higher the likelihood of a cache hit in the near future, which translates to a higher restore throughput. Figure 7 shows the average prefetch distance and corresponding restore throughput achieved by our score-based approach at every restore operation for uniform-sized checkpoints for the case of *sequential* read order. As can be observed, the number of checkpoints prefetched in advance is significant (up to 4 checkpoints per process), which provides enough reserve to amortize unexpected or slow prefetches that delay the restore operations. We also note that irrespective of zero consecutive prefetches completed for the case of *Single hint*, our proposed approach is able to achieve ~6.3 GB/s restore rate. For the case of *No hints*, the score-based approach performs the best towards the end of the restore phase because it retains all checkpoints written in the last iterations on the cache. For *All hints*, the number of next prefetches completed grows steeply until the maximum GPU cache size (4 GB accommodates 32 uniform-sized checkpoints of 128 MB) after which it restores checkpoints up to iteration 96 from the host cache. Once all the checkpoints from the GPU and host cache are consumed, we observe low prefetching distance for further iterations due to slow reads from the file system.

5.4.5 Impact of Variable Checkpoint Interval and GPU Cache Size. Next, we evaluate the impact of variable checkpoint interval and cache sizes on the compared approaches. Again, we run 8 processes (one per GPU) in embarrassingly parallel mode on a single compute node. We focus on the irregular read order since it is the most problematic to handle. The *Single hint* scenario has a similar speedup as in Figure 5b, thus we omit it from further evaluations and focus on the extreme cases of *No hints* and *All hints*. Figure 8a shows similar checkpoint throughput for all approaches, except for ADIOS2, which exhibits a significantly lower checkpoint throughput (consistent with the previous experiments) regardless of the checkpoint interval. Meanwhile, the restore throughput increases gradually with an increasing checkpoint interval, reaching the peak device-to-device bandwidth at a 30 ms checkpoint interval. This can be explained by the fact that a larger checkpoint interval allows more and/or larger prefetches to finish in time until the next read request is issued. Next, we study the impact of varying GPU cache sizes for a checkpoint interval of 10 ms. As expected, the checkpointing throughput increases with larger GPU cache sizes, because evictions are delayed. Figure 8b shows up to 2.6× faster checkpoint and restore throughput of our approach as compared to the optimized UVM approach. Since ADIOS2 does not feature a GPU cache, its throughput remains unchanged, but we include it for reference.



(a) Tightly coupled (barrier at each iteration)



(b) Embarrassingly parallel (no synchronization between GPUs)

Figure 9: Scalability of checkpoint and restore throughput for variable checkpoint sizes and irregular restore order. Higher is better. Note that Y-axis is log scale.

5.4.6 Scalability Study. We study the scalability of the compared approaches using variable-sized checkpoints. Figure 9 shows the stacked checkpoint and restore throughputs per process for an increasing number of GPUs. At scale, synchronization can introduce significant overheads, therefore we study both the case of embarrassingly parallel and tightly coupled runs (as explained in Section 5.3). In the former case, the competition for shared resources (host memory, links between GPUs and host memory, SSD) due to asynchronous flushes and prefetches is increasing at scale. In the latter case, the competition for resources is reduced due to synchronization at each iteration. Despite increasing competition for resources, both the embarrassingly parallel and tightly coupled cases maintain a relatively stable throughput for an increasing number of GPUs regardless of the considered approach. This is an important observation because it shows that our approach can maintain its advantage over the other approaches at scale. Even at scale, our approach still maintains 22.8× and 13× advantage over the optimized UVM approach for the *No hints* and *Single hint* cases, respectively. It is interesting to note that ADIOS2 suffers a drop in checkpointing throughput for an increasing number of processes and thus demonstrates limited scalability, while the opposite effect can be observed for other approaches.

6 CONCLUSIONS

In this paper, we address the problem of managing large-scale intermediate data generated by the HPC applications that must be checkpointed and restored with high frequency, typically in the order of milliseconds. State-of-the-art solutions have limited support for such scenarios in which the access pattern of restoring data from the saved checkpoint is often known in advance. Specifically, checkpointing frameworks focus on minimizing I/O overhead through asynchronous multi-level flushing techniques, while transparent system-level memory management techniques, such as UVM, can only make limited use of hints about the access pattern. To address this gap, we designed and developed a checkpointing strategy that co-optimizes asynchronous multi-level strategies for flushing and prefetching of checkpoints by exploiting foreknowledge about the access order. We conduct extensive evaluation with different settings, such as checkpoint sizes, restore order, number of prefetch hints, varying compute interval and cache sizes, all of which are applicable to a wide range of application scenarios. Our proposed approach shows up to an order of magnitude speedup compared with state-of-the-art I/O runtimes, such as ADIOS2, as well as transparent system-level memory management, such as UVM. We demonstrated these benefits both for synchronized and interleaved checkpoint and restore phase at scale.

In the future, we plan to explore how to share the host cache across different processes and nodes to load balance variable-sized checkpoints and incorporate support for Nvidia GPUDirect storage.

ACKNOWLEDGMENTS

This work is supported in part by the ARAMCO Services Company, 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). Results presented in this paper are obtained using Argonne's ALCF HPC systems.

REFERENCES

[1] Tyler Allen and Rong Ge. 2021. Demystifying GPU UVM Cost with Deep Runtime and Workload Analysis. In *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 141–150. https://doi.org/10.1109/IPDPS49936.2021.00023

[2] Tariq Alturhkestani, Hatem Ltaief, and David Keyes. 2020. Maximizing I/O Bandwidth for Reverse Time Migration on Heterogeneous Large-Scale Systems. In *Euro-Par 2020: Parallel Processing*, Maciej Malawski and Krzysztof Rzadca (Eds.). Springer International Publishing, Cham, 263–278.

[3] Tariq Alturkestani, Thierry Tonellot, Hatem Ltaief, Rached Abdelkhalak, Vincent Etienne, and David Keyes. 2019. MLBS: Transparent Data Caching in Hierarchical Storage for Out-of-Core HPC Applications. In *2019 IEEE 26th International Conference on High Performance Computing, Data, and Analytics (HiPC)*. 312–322. https://doi.org/10.1109/HiPC.2019.00046

[4] Argonne Leadership Computing Facility. n.d.. Theta GPU. https://www.alcf.anl.gov/alcf-resources/theta. Accessed: May 9, 2023.

[5] M. Arif, K. Assogba, and M. Rafique. 2022. Canary: Fault-Tolerant FaaS for Stateful Time-Sensitive Applications. In *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE Computer Society, Los Alamitos, CA, USA, 1–16. https://doi.org/10.1109/SC41404.2022.00046

[6] Hariharan Devarajan, Anthony Kougkas, and Xian-He Sun. 2020. HFetch: Hierarchical Data Prefetching for Scientific Workflows in Multi-Tiered Storage

Environments. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 62–72. https://doi.org/10.1109/IPDPS47924.2020.00017

[7] Matthieu Dreher and Tom Peterka. 2016. Bredala: Semantic Data Redistribution for In Situ Applications. In *2016 IEEE International Conference on Cluster Computing (CLUSTER)*. 279–288. https://doi.org/10.1109/CLUSTER.2016.30

[8] Nikoli Dryden, Roman Böhringer, Tal Ben-Nun, and Torsten Hoefler. 2021. Clairvoyant Prefetching for Distributed Machine Learning I/O. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (St. Louis, Missouri) *(SC '21)*. Association for Computing Machinery, New York, NY, USA, Article 92, 15 pages. https://doi.org/10.1145/3458817.3476181

[9] Denis Foley and John Danskin. 2017. Ultra-Performance Pascal GPU and NVLink Interconnect. *IEEE Micro* 37, 2 (2017), 7–17. https://doi.org/10.1109/MM.2017.37

[10] Debashis Ganguly, Ziyu Zhang, Jun Yang, and Rami Melhem. 2019. Interplay between Hardware Prefetcher and Page Eviction Policy in CPU-GPU Unified Virtual Memory. In *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*. 224–235.

[11] William F. Godoy, Norbert Podhorszki, Ruonan Wang, Chuck Atkins, Greg Eisenhauer, Junmin Gu, Philip Davis, Jong Choi, Kai Germaschewski, Kevin Huck, Axel Huebl, Mark Kim, James Kress, Tahsin Kurc, Qing Liu, Jeremy Logan, Kshitij Mehta, George Ostrouchov, Manish Parashar, Franz Poeschel, David Pugmire, Eric Suchyta, Keichi Takahashi, Nick Thompson, Seiji Tsutsumi, Lipeng Wan, Matthew Wolf, Kesheng Wu, and Scott Klasky. 2020. ADIOS 2: The Adaptable Input Output System. A framework for high-performance data management. *SoftwareX* 12 (2020), 100561. https://doi.org/10.1016/j.softx.2020.100561

[12] Paul H Hargrove and Jason C Duell. 2006. Berkeley lab checkpoint/restart (BLCR) for Linux clusters. *Journal of Physics: Conference Series* 46, 1 (sep 2006), 494. https://doi.org/10.1088/1742-6596/46/1/067

[13] Tong Jin, Fan Zhang, Qian Sun, Hoang Bui, Melissa Romanus, Norbert Podhorszki, Scott Klasky, Hemanth Kolla, Jacqueline Chen, Robert Hager, Choong-Seock Chang, and Manish Parashar. 2015. Exploring Data Staging Across Deep Memory Hierarchies for Coupled Data Intensive Simulation Workflows. In *2015 IEEE International Parallel and Distributed Processing Symposium*. 1033–1042. https://doi.org/10.1109/IPDPS.2015.50

[14] Mahmut Kandemir, Taylan Yemliha, Ramya Prabhakar, and Myoungsoo Jung. 2012. On Urgency of I/O Operations. In *2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (ccgrid 2012)*. 188–195. https://doi.org/10.1109/CCGrid.2012.40

[15] Suha N. Kayum, Thierry Tonellot, Vincent Etienne, Ali Momin, Ghada Sindi, Maxim Dmitriev, and Hussain Salim. 2020. GeoDRIVE - a high performance computing flexible platform for seismic applications. *First Break* 38, 2 (2020), 97–100. https://doi.org/10.3997/1365-2397.fb2020015

[16] Jie Liu, Bogdan Nicolae, and Dong Li. 2023. Lobster: Load Balance-Aware I/O for Distributed DNN Training. In *Proceedings of the 51st International Conference on Parallel Processing* (Bordeaux, France) *(ICPP '22)*. ACM, NY, USA, Article 26, 11 pages. https://doi.org/10.1145/3545008.3545090

[17] Jay Lofstead, Ivo Jimenez, Carlos Maltzahn, Quincey Koziol, John Bent, and Eric Barton. 2016. DAOS and Friends: A Proposal for an Exascale Storage System. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Salt Lake City, Utah) *(SC '16)*. IEEE Press, Article 50, 12 pages.

[18] Avinash Maurya, Bogdan Nicolae, M. Mustafa Rafique, Amr M. Elsayed, Thierry Tonellot, and Franck Cappello. 2022. Towards Efficient Cache Allocation for High-Frequency Checkpointing. In *2022 IEEE 29th International Conference on High Performance Computing, Data, and Analytics (HiPC)*. 262–271. https://doi.org/10.1109/HiPC56025.2022.00043

[19] Avinash Maurya, Bogdan Nicolae, M. Mustafa Rafique, Thierry Tonellot, and Franck Cappello. 2021. Towards Efficient I/O Scheduling for Collaborative Multi-Level Checkpointing. In *2021 29th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*. 1–8. https://doi.org/10.1109/MASCOTS53633.2021.9614284

[20] Jayashree Mohan, Amar Phanishayee, and Vijay Chidambaram. 2021. CheckFreq: Frequent, Fine-Grained DNN Checkpointing. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*. USENIX Association, 203–216. https://www.usenix.org/conference/fast21/presentation/mohan

[21] Dmitriy Monozov and Zarija Lukie. 2016. Henson v1.0. https://www.osti.gov/servlets/purl/1312559. https://www.osti.gov/biblio/1312559 HENSON; 004707WKSTN00.

[22] Adam Moody, Greg Bronevetsky, Kathryn Mohror, and Bronis R. de Supinski. 2010. Design, Modeling, and Evaluation of a Scalable Multi-level Checkpointing System. In *SC '10: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–11. https://doi.org/10.1109/SC.2010.18

[23] Dmitriy Morozov and Tom Peterka. 2016. Block-parallel data analysis with DIY2. In *2016 IEEE 6th Symposium on Large Data Analysis and Visualization (LDAV)*. 29–36. https://doi.org/10.1109/LDAV.2016.7874307

[24] Sri Hari Krishna Narayanan, Thomas Propson, Marcelo Bongarti, Jan Hückelheim, and Paul Hovland. 2022. Reducing Memory Requirements of Quantum Optimal Control. In *Computational Science – ICCS 2022*, Derek Groen, Clélia de Mulatier, Maciej Paszynski, Valeria V. Krzhizhanovskaya, Jack J. Dongarra, and Peter M. A. Sloot (Eds.). Springer International Publishing, Cham, 129–142.

[25] Bogdan Nicolae, Adam Moody, Elsa Gonsiorowski, Kathryn Mohror, and Franck Cappello. 2019. VeloC: Towards High Performance Adaptive Asynchronous Checkpointing at Large Scale. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 911–920. https://doi.org/10.1109/IPDPS.2019.00099

[26] Bogdan Nicolae, Adam Moody, Gregory Kosinovsky, Kathryn Mohror, and Franck Cappello. 2021. VELOC: VEry Low Overhead Checkpointing in the Age of Exascale. *CoRR* abs/2103.02131 (2021).

[27] Akira Nukada, Hiroyuki Takizawa, and Satoshi Matsuoka. 2011. NVCR: A Transparent Checkpoint-Restart Library for NVIDIA CUDA. In *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum*. 104–113. https://doi.org/10.1109/IPDPS.2011.131

[28] NVIDIA. n.d.. NVIDIA DGX A100 System Architecture. https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/dgx-a100/dgxa100-system-architecture-white-paper.pdf. Accessed: May 6, 2023.

[29] Konstantinos Parasyris, Kai Keller, Leonardo Bautista-Gomez, and Osman Unsal. 2020. Checkpoint Restart Support for Heterogeneous HPC Applications. In *2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*. 242–251. https://doi.org/10.1109/CCGrid49817.2020.00-69

[30] Jongsoo Park, Richard M. Yoo, Daya S. Khudia, Christopher J. Hughes, and Daehyun Kim. 2013. Location-aware cache management for many-core processors with deep cache hierarchy. In *SC '13: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. 1–12. https://doi.org/10.1145/2503210.2503224

[31] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. 2017. Automatic Differentiation in PyTorch. In *NIPS Workshop on Autodiff*. USA.

[32] R.-E. Plessix. 2006. A review of the adjoint-state method for computing the gradient of a functional with geophysical applications. *Geophysical Journal International* 167, 2 (2006). https://doi.org/10.1111/j.1365-246X.2006.02978.x

[33] Line C. Pouchard, Tanzima Z. Islam, Bogdan Nicolae, and Robert Ross. 2022. A (meta)data framework for reproducing hybrid workflows with FAIR. In *WORKS'22: 17th Workshop on Workflows in Support of Large-Scale Science (in conjunction with SC'22)*. Dallas, USA.

[34] Yubo Qin, Ivan Rodero, Anthony Simonet, Charles Meertens, Daniel Reiner, James Riley, and Manish Parashar. 2021. Leveraging user access patterns and advanced cyberinfrastructure to accelerate data delivery from shared-use scientific observatories. *Future Generation Computer Systems* 122 (2021), 14–27. https://doi.org/10.1016/j.future.2021.03.004

[35] Robert Ross, Lee Ward, Philip Carns, Gary Grider, Scott Klasky, Quincey Koziol, Glenn K. Lockwood, Kathryn Mohror, Bradley Settlemyer, and Matthew Wolf. 2018. *Storage Systems and Input/Output: Organizing, Storing, and Accessing Data for Scientific Discovery. Report for the DOE ASCR Workshop on Storage Systems and I/O. [Full Workshop Report]*. Technical Report DOE/SC-0196. US Department of Energy. https://doi.org/10.2172/1491994 Conference: Storage Systems and I/O: Organizing, Storing, and Accessing Data for Scientific Discovery, Gaithersburg, MD, 19-20 Sep 2018.

[36] Philip Schwan et al. 2003. Lustre: Building a file system for 1000-node clusters. In *Proceedings of the 2003 Linux symposium*, Vol. 2003. 380–386.

[37] Qingchuan Shi, George Kurian, Farrukh Hijaz, Srinivas Devadas, and Omer Khan. 2016. LDAC: Locality-Aware Data Access Control for Large-Scale Multicore Cache Hierarchies. 13, 4, Article 37 (nov 2016), 28 pages. https://doi.org/10.1145/2983632

[38] Po-An Tsai, Nathan Beckmann, and Daniel Sanchez. 2017. Jenga: Software-defined cache hierarchies. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*. 652–665. https://doi.org/10.1145/3079856.3080214

[39] Andrea Walther and Sri Hari Krishna Narayanan. 2016. *Extending the Binomial Checkpointing Technique for Resilience*. Technical Report. Albuquerque, NM, US. https://doi.org/10.1137/1.9781611974693.13

[40] Hanqin Wan. 2019. Deep Learning:Neural Network, Optimizing Method and Libraries Review. In *2019 International Conference on Robots and Intelligent System (ICRIS)*. 497–500. https://doi.org/10.1109/ICRIS.2019.00128

[41] Xian Wang, Paul Kairys, Sri Hari Krishna Narayanan, Jan Hückelheim, and Paul Hovland. 2022. Memory-Efficient Differentiable Programming for Quantum Optimal Control of Discrete Lattices. In *2022 IEEE/ACM Third International Workshop on Quantum Computing Software (QCS)*. 94–99. https://doi.org/10.1109/QCS56647.2022.00016

[42] Shucai Xiao, Pavan Balaji, James Dinan, Qian Zhu, Rajeev Thakur, Susan Coghlan, Heshan Lin, Gaojin Wen, Jue Hong, and Wu-chun Feng. 2012. Transparent Accelerator Migration in a Virtualized GPU Environment. In *2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (ccgrid 2012)*. 124–131. https://doi.org/10.1109/CCGrid.2012.26

[43] Hua-Wei Zhou, Hao Hu, Zhihui Zou, Yukai Wo, and Oong Youn. 2018. Reverse time migration: A prospect of seismic imaging methodology. *Earth-Science Reviews* 179 (2018), 207–227. https://doi.org/10.1016/j.earscirev.2018.02.008

[44] Mahdi Zolnouri, Xinlin Li, and Vahid Partovi Nia. 2020. Importance of data loading pipeline in training DNNs. *arXiv preprint arXiv:2005.02130* (2020).