Towards Efficient Cache Allocation for High-Frequency Checkpointing

Avinash Maurya, Bogdan Nicolae, M Mustafa Rafique, Amr Elsayed, Thierry Tonellot, Franck Cappello

Towards Efficient Cache Allocation for High-Frequency Checkpointing

Avinash Maurya*, Bogdan Nicolae†, M. Mustafa Rafique*, Amr M. Elsayed‡, Thierry Tonellot§, Franck Cappello†
*Rochester Institute of Technology, USA
†Argonne National Laboratory, USA
‡Brightskies Technologies, Alexandria, Egypt
§Exploration and Petroleum Engineering Advanced Research Center, Saudi Aramco, Dhahran, Saudi Arabia
Email: *{am6429, mrafique}@cs.rit.edu; †{bnicolae, cappello}@anl.gov;
‡amr.nasr@brightskiesinc.com; §thierrylaurent.tonellot@aramco.com

***Abstract*—While many HPC applications are known to have long runtimes, this is not always because of single large runs: in many cases, this is due to ensembles composed of many short runs (runtime in the order of minutes). When each such run needs to checkpoint frequently (e.g. adjoint computations using a checkpoint interval in the order of milliseconds), it is important to minimize both checkpointing overheads at each iteration, as well as initialization overheads. With the rising popularity of GPUs, minimizing both overheads simultaneously is challenging: while it is possible to take advantage of efficient asynchronous data transfers between GPU and host memory, this comes at the cost of high initialization overhead needed to allocate and pin host memory. In this paper, we contribute with an efficient technique to address this challenge. The key idea is to use an adaptive approach that delays the pinning of the host memory buffer holding the checkpoints until all memory pages are touched, which greatly reduces the overhead of registering the host memory with the CUDA driver. To this end, we use a combination of asynchronous touching of memory pages and direct writes of checkpoints to untouched and touched memory pages in order to minimize end-to-end checkpointing overheads based on performance modeling. Our evaluations show a significant improvement over a variety of alternative static allocation strategies and state-of-art approaches.**

***Index Terms*—GPU checkpointing, multi-level caching, fast initialization**

I. Introduction

High-Performance Computing (HPC) applications produce massive amounts of distributed intermediate data during their execution that needs to be checkpointed concurrently in real-time at scale. This is a fundamental I/O pattern used in a wide range of scenarios [1]: fault tolerance based on checkpoint-restart, producer-consumer patterns in workflows (e.g., offline or in-situ analytics), reproducibility (validation of intermediate states in addition to end results), etc.

One such popular scenario is the use of checkpointing for the purpose of revisiting previous states in order to advance a computation. For example, the adjoint state method is an efficient numerical method to compute the gradient that is widely employed by automatic differentiation (AD) tools [2], [3] and used in a variety of scientific applications: climate and ocean modeling [4], multi-physics [5], seismic imaging in the oil industry [6], etc. Deep learning (DL) techniques are also based on AD and often paired with stochastic gradient descent [7]. They involve two phases: a forward pass, during which a large number of intermediate checkpoints are produced, followed by a backward pass, during which the checkpoints are consumed in reverse order.

With the growing popularity of accelerators, in particular GPUs, adjoint computations make progress quickly. For example, reverse time migration (RTM), a seismic imaging application popular in the oil industry, involves large ensembles of many short-lived adjoint computations, each of which often runs for a few minutes or tens of seconds. During this time, each adjoint computation needs to capture/restore checkpoints during the forward/backward pass at high frequency, which is often in the order of milliseconds and involves large data sizes in the order of hundreds of MB. At these time scales, even one-time operations like allocating a buffer to store the checkpoints can introduce significant initialization overheads. The problem is further complicated by the need co-locate and run many such short-lived adjoint computations on powerful HPC nodes that feature a large number of GPUs. In this case, even if each adjoint computation manages its own buffer, allocations happen concurrently and introduce competition for resources, which further amplifies the initialization overhead.

Even if it was possible to make GPU memory allocation overheads negligible, GPUs have limited memory capacity, therefore it is typically not feasible to store the checkpoints in GPU memory exclusively. In this case, it is possible to use a simple solution like Unified Virtual Memory (UVM) [8], which enables applications to allocate a large host buffer that can be directly accessed both by GPUs and CPUs using transparent on-demand paging. However, despite growing popularity, such an approach not only has high initialization overheads (further amplified by competition between multiple GPUs for host memory), but it also causes significant I/O performance degradation due to on-demand paging. Another solution is to adopt state-of-art multi-level checkpointing strategies that leverage heterogeneous node-local storage tiers (e.g., a GPU memory buffer acting as a cache for a large host memory buffer) to flush and prefetch checkpoints to/from slower tiers asynchronously using explicit data movements [9]. Using this approach, a large part of the data movements between the storage tiers can be overlapped with the computations, which greatly reduces the I/O overheads associated with writing and reading checkpoints during the forward and backward pass.

However, state-of-art checkpointing approaches are not designed for short-lived jobs and incur huge initialization

overheads. For example, to enable efficient data movements between the GPU and host memory that progress at full bandwidth, the host buffer needs to be registered with the GPU driver, which in turn triggers an allocation and pinning of physical memory pages. This can be orders of magnitude slower than the read/write bandwidth to the GPU memory (e.g., for an NVIDIA DGX A100 node, host memory registration bandwidth 3 GB/s, while GPU read/write memory bandwidth is 500 GB/s). Therefore, before optimizations like asynchronous caching and prefetching can take advantage of the full I/O bandwidth, the full initialization cost needs to be paid upfront. With this cost in the order or tens of seconds, which is similar in duration to the runtime, it does not justify the benefits for short-lived jobs.

In this paper we address the problem of reducing initialization overheads for multi-level checkpointing strategies of short-lived adjoint computations. Our goal is to avoid paying a high initialization cost upfront, while at the same time maintain a high I/O bandwidth for checkpoint/restore both during the forward and backward pass. To this end, we introduce an adaptive allocation strategy that works as follows. First, it allocates the GPU cache incrementally by relying on the low-level VMM operations to overlap registration with read/write operations. This reduces the GPU cache initialization overheads. Second, it delays the registration of the host buffer (which cannot happen incrementally) and allows data transfers between the GPU and host memory to progress immediately at reduced I/O bandwidth, while concurrently touching the pages of the host buffer in an opportunistic fashion in the background, which forces an allocation of the physical pages in advance, thereby greatly reducing the subsequent registration overhead once all pages were touched.

Given that multiple adjoint computations on different GPUs compete for host memory bandwidth, a key challenge our proposal addresses is how to decide when to allow checkpoint flushes/prefetches to untouched pages and when to touch pages in advance, such as to optimize the overall I/O throughput. We summarize our contributions as follows:

- We formulate the problem of multi-level checkpointing for short-lived adjoint computations that combine a small GPU cache and a large host memory buffer to checkpoint/restore large amounts of data at high frequency (§ II).
- We propose a series of design principles for multi-level checkpointing that avoid high initialization overheads incurred upfront through a combination of several key ideas: incremental GPU cache allocation, opportunistic touching of the memory pages of the host buffer, concurrency control to optimize competition between opportunistic touching and reads/writes to memory, delayed registration of the host buffer (§ IV-A).
- We illustrate these design principles as an extension to VELOC [9], a production-ready HPC multi-level checkpoint-restart library. In this context, we take advantage of CUDA-enabled GPUs to enable low-level GPU cache and host buffer memory management (§ IV-C).
- We evaluate our proposal in a series of experiments conducted on a multi-GPU Nvidia DGX-A100 platform. Compared with existing state-of-art multi-level checkpointing approaches and UVM, our proposal reduces the combined checkpoint/initialization overheads by orders of magnitude in a variety of synthetic scenarios and traces of a real-time HPC applications: RTM (reverse time migration), a seismic imaging application used in the oil and gas industry (§ V).

II. Problem Formulation

We formulate the problem of reducing initialization overheads for multi-level checkpointing strategies of short-lived adjoint computations as follows. Consider an HPC node consisting of N GPUs, each of which is equipped with high-bandwidth memory (HBM, also known as device memory) and main memory (host memory). N independent, short-lived adjoint computations P_i are started simultaneously, each on a dedicated GPU. For each process P_i, a fraction of the device and host memory is reserved as cache and host buffer respectively (there are N independent host buffers that are not shared between the GPUs).

Each adjoint computation takes t^i_{init} time to initialize its device buffer and host buffer. Then, it produces K checkpoints during the forward pass using a multi-level checkpointing library. This works as follows: each checkpoint is stored to the device cache (by creating a blocking copy of the GPU data structures that are part of the checkpoint), then it is flushed asynchronously from the device cache to the host buffer in the background (using the FIFO order), while the application continues running. If the device cache cannot store a new checkpoint, then the oldest checkpoints are evicted until enough room for the new checkpoint becomes available. An eviction blocks the application until the corresponding flush to the host memory is complete. During the backward pass, the checkpoints are restored in reverse order. To reduce the read latency perceived by the application, consumed checkpoints are discarded from the device cache and the free space is used to prefetch checkpoints from the host buffer to the device cache in advance. Under ideal circumstances this blocks the application only for the duration of a device-to-device copy (from the device cache to the GPU data structures).

Our goal is to minimize the time spent during initialization and the time spent blocking during checkpoint operations (forward pass) and restore operations (backward pass) for the whole group of N adjoint computations. This goal is subject to several constraints: (1) checkpoints cannot be evicted before the device cache has been fully initialized and filled, which avoids trashing during the backward pass by limiting competition between pending flushes and prefetches; (2) the host buffer cannot be registered incrementally because DMA transfers from two contiguous but separately pinned memory regions are not supported by modern GPU drivers (e.g. NVIDIA).

III. Related Work

Memory allocation libraries: Allocating large chunks of memory have been well studied in the past [10]–[17] that optimize various aspects of memory allocation such as fragmentation, allocation-latency, lightweight reclamation of free memory, physical memory consumption, lock-free allocation, and scalability. SuperMalloc [17] outperforms the state-of-the-art allocation techniques such as Hoard [14] and JEmalloc [12] by up to 3× by allocating larger chunks of 2MiB that contain homogeneous sized objects. Hoard [14] uses a global and per-process heap to bound the memory consumption for producer-consumer based applications. JEMalloc [12], SFMalloc [16], and TCMalloc [13], McRT-malloc [15] are general purpose memory allocators that outperfom the default Linux allocator for use cases such as frequent malloc/free pattern, multi-threaded applications running at scale, and transactional memory accesses. However, none of these studies focus on aggressively allocating physical memory for latency-critical applications, that are committed to using the entire memory in the future. Additionally, our approach optimizes both allocating and pinning the memory for future accesses.

HPC Checkpoint-Restart: Checkpoint-restart techniques are traditionally used in for fault-tolerance on HPC infrastructures. Transparent fault tolerance approaches (DMTCP [18], BLCR [19]) automatically capture the full state of a group of process, at the expense of generating a large checkpoint sizes that cannot be used for any other purpose than resuming the process execution. Application-level fault tolerance approaches (FTI [20], SCR [21]) rely on the application to define critical data structures, which are checkpointed and restored using multi-level checkpointing strategies. Some approaches such as VELOC [22] specialize on asynchronous multi-level checkpointing, which enables many use cases beyond resilience, including the scenario targeted in this paper. Checkpoint-restore techniques have been extended to GPUs, both for fault-tolerance [23]–[28] and workload migration [24], [29]. System-level checkpointing libraries such as NVCR [24] and CheCUDA [23], transparently record and replay the memory based CUDA APIs for checkpointing and restoring. Approaches such as CheckFreq [30], GPU-snapshot [31] and Multi-layered Buffered System [32] also exploit heterogeneous storage tiers. However, none of these approaches consider short-running jobs, for which the impact of initialization overheads is non-negligible.

Data movement engines: Data staging solutions such as Stacker [33], DataStager [34], DataSpaces [35], Data Elevator [36], and TRIO [37] have been proposed to mitigate the I/O overheads of interacting with remote storage repositories such as parallel file systems and object stores by using intermediate caching layers, both node-local (e.g., DRAM, SSDs) and remote (e.g., burst buffers). However, such data services have no support (or limited support) for GPUs and typically rely on the applications to explicitly allocate a staging buffer on the host memory and move the data to/from it.

To our best knowledge, we are the first to consider the problem of optimizing asynchronous multi-level checkpointing for short-running jobs with high-frequency checkpoint requirements for which initialization overheads cannot simply be amortized over the total runtime.

IV. System Design

A. Design principles

Incremental GPU cache allocation using low-level GPU Virtual Memory Management (VMM) to avoid GPU initialization delays: High bandwidth device memory allocation and preparation on GPUs (i.e., all steps needed to access the memory at full I/O bandwidth) is generally a lightweight, single operation. For example, NVIDIA A100 GPUs can allocate and prepare a cache in excess of 1.4 TB/s using a single API call. However, this bandwidth does not scale with an increasing number of concurrent allocation requests, even if the applications issuing them are unrelated and run on separate GPUs. This happens because of synchronization needed in the GPU driver to coordinate memory-related book-keeping across the GPUs. As a consequence, even a few concurrent allocation requests are enough to slow down the allocation bandwidth below the read/write bandwidth. To address this issue, modern GPU drivers expose a low-level GPU VMM manipulation API in the same spirit as the *mmap* Linux kernel system call: first a device memory region can be reserved in advance at negligible overhead, then parts of it can be mapped to physical memory pages later. We leverage this approach to propose an incremental device cache allocation strategy that avoids large initial allocations suffering from concurrency bottlenecks. Specifically, we reserve the whole device cache from the beginning, but then map the contiguous chunks of the device cache to physical pages asynchronously in a background. Meanwhile, any reads and writes to the GPU cache are permitted, as long as they refer to the mapped region, which in turn is atomically extended as soon as the next chunk finished being mapped in the background. Since evictions are not allowed before the device cache has been fully allocated, writes may block waiting for new chunks to be mapped. However, in practice, the reads and writes may progress at different rates on different GPUs, which means incremental allocations become staggered and therefore do not suffer from concurrency bottlenecks, which effectively reduces the overall device cache allocation overheads.

Opportunistic touching of host buffer virtual pages to force allocation of physical pages in advance: Unfortunately, as mentioned in Section II, a similar incremental approach cannot be used for host buffer allocation and preparation, as this involves registering a single contiguous memory region with the GPU driver, which in turn reserves and pins physical memory pages on the host side. To circumvent this issue, we leverage the fact that the operating system reserves and maps physical memory pages to virtual memory pages as soon as they are touched, and it is unlikely for these pages to get swapped if the memory utilization is below the physical capacity. In fact, a majority of the registration overhead is related to reserving and mapping the physical memory

pages, while pinning them incurs a much smaller overhead. As a consequence, we propose to reduce the registration overhead by delaying the registration until all pages have been touched. Specifically, we start by allocating the host buffer using a regular *mmap* that requests huge pages (to reduce the number of page table entries) and that typically has a negligible overhead. Then, we allow checkpoint flushes from the device cache to the host buffer to proceed immediately without registration, which is much slower than flushes to registered host memory (e.g., 3 GB/s vs. 12 GB/s on an NVIDIA A100 GPU). However, at the same time, the pages that are not involved in flushes are touched asynchronously in the background. Touching can be achieved by writing a single byte, which has negligible overhead in itself but triggers the reservation and set up of the memory pages in advance, which under the right circumstances can be overlapped with the flushes and therefore their overhead can be masked. In addition to enabling a faster registration after all pages have been touched, this approach also speeds up flushes before the memory registration, because writes to untouched memory pages need to pay the preparation overhead on-demand. This speed-up can be significant: for example, on a NVIDIA A100 GPU, the write bandwidth increases from 3 GB/s (untouched pages) to 6 GB/s (touched pages). Thus, our approach manages to keep initial flush throughput high while at the same time eliminating a majority of the registration overhead.

Concurrency control to optimize competition between flushes and opportunistic page touching: It is important to note that asynchronous opportunistic page touching competes for host memory bandwidth with the checkpoint flushes. Thus, the touch throughput and flush throughput are both slower when they progress concurrently compared to the case when they progress sequentially. This introduces a trade-off: if we allow asynchronous touches to share the host memory bandwidth with the checkpoint flushes, then more pages will be touched during one iteration, which allows flushes in the next iteration to progress faster. At the same time, flushes in the current iteration will be slower. To address this issue, we propose two alternative strategies. First, consider a simple decoupled producer-consumer strategy in which the producer touches the pages is ascending order, while the consumer concurrently flushes to touched pages. Second, based on the observation that a checkpoint flush does not necessarily need to wait until all host pages involved in its transfer have been touched, we introduce an alternative strategy pauses touches during flushes and resumes them when the host memory bandwidth is idle (which can happen if the flushes are faster than the computations performed during each iteration). Depending on the duration of the computations, host memory throughput under concurrency, and flush pattern under concurrency from multiple GPUs (flushes may be staggered and involve different checkpoint sizes), we can select one or the other strategy. We illustrate these concurrency control techniques and resulting interactions between the device cache and host buffer in greater detail in Section IV-B.

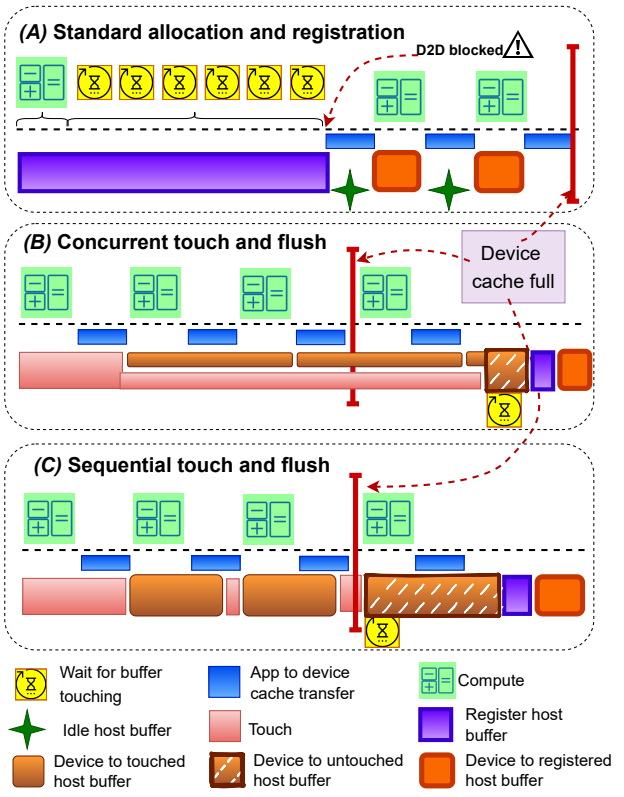


Fig. 1: Comparative illustration of the interactions between the device cache and the host buffer

Serialized registration of host buffers to avoid competition for pinning physical memory pages on the host: We run independent adjoint computations on each GPU, each of which manage their own host buffer. However, even so, when the adjoint computations exhibit similar behavior patterns (which is often the case in ensembles), it is likely that multiple host buffer registration requests happen simultaneously. Unfortunately, a registration blocks not only device-to-host transfers, but also device-to-device transfers, which essentially means neither the device cache, not the host buffer are available during host memory registration. Furthermore, competition for pinning physical memory pages leads to a dramatic slow-down of the registration. Therefore, left to compete naturally for registering host buffers, the application processes end up slowing each other down by simultaneously pausing flushes and opportunistic touching unnecessarily. As a consequence, we propose a coordinated approach that serializes the registration of the host buffers in a round-robin fashion. Specifically, only one GPU is allowed to register its host buffer at any given time, while the rest continue flushing to unregistered memory pages until their turn arrives (even if all pages have been touched). While this approach favors some GPUs over others, overall it leads to lower overheads for the whole group.

B. Interactions between the device cache and host buffer

In this section, we focus on the interactions between the device cache and the host buffer for the two concurrency control strategies introduced above, in comparison with a traditional

solution that pays the initialization overheads in advance. We assume a traditional adjoint computation in which each iteration consists of a computation followed by capturing a checkpoint (forward pass) or restoring a checkpoint (backward pass). Furthermore, we assume the device cache is not large enough to hold all checkpoints, which means it gets filled and triggers the registration of the host buffer before the forward pass is complete. Therefore, we do not study the backward pass as it operates on a registered host buffer and does not differ from a standard multi-level prefetching strategy.

Specifically, Figure 1 depicts three approaches: *(A) standard allocation and pinning*, which corresponds to a traditional solution: in this case, at the end of the first iteration, when the first checkpoint is requested, the application blocks until the device cache and host buffer is fully allocated and prepared (i.e., host pages are mapped and pinned). Later, all device-to-host flushes proceed at full speed. Next, *(B) concurrent touch and flush* corresponds to the decoupled producer-consumer strategy that allows opportunistic touches to compete with the flushes for host memory bandwidth. In this case, no initialization overhead is paid in advance. Flushes take longer than in the case of the standard approach, both because they involve writes to touched but unregistered memory pages and because they are subject to competition with concurrent touching of the memory pages. Eventually, after the device cache is fully touched, the host buffer is registered significantly faster as this operation only has to pin the physical pages. Finally, *(C) sequential touch and flush* alternates between flushing and touching pages to address the situation when competition between touching and flushing is counterproductive. Specifically, the memory pages are touched only during "idle" intervals that start when the flushing of the checkpoint corresponding to the previous iteration finished and that end when the current iteration finished. Eventually this may lead to an accumulation of untouched, unregistered pages that are directly written to.

C. Implementation details

We implemented the mentioned design principles and performance model as an extension of VELOC [9], a production-ready, multi-level checkpointing user-space runtime. As depicted in Figure 2 during the initialization, VELOC_Init, step, child threads are spawned for initializing device and host buffer, and performing asynchronous transfers between the device and the host buffer. The device cache thread first allocates the entire cache using `cuMemAddressReserve`, and then incrementally maps the virtual allocation of the cache to physical HBM memory by using `cuMemMap` calls. Finally, the access to the mapped regions can be enabled for a given to peer GPUs (if required) using `cuMemSetAccess` API. Similarly, the host buffer allocation thread first allocates the virtual memory for the cache using `malloc` call, and later incrementally touches the cache, either using `memset` or setting the first bit of every page, which is enough to generate a physical mapping for the entire page. Once the entire host

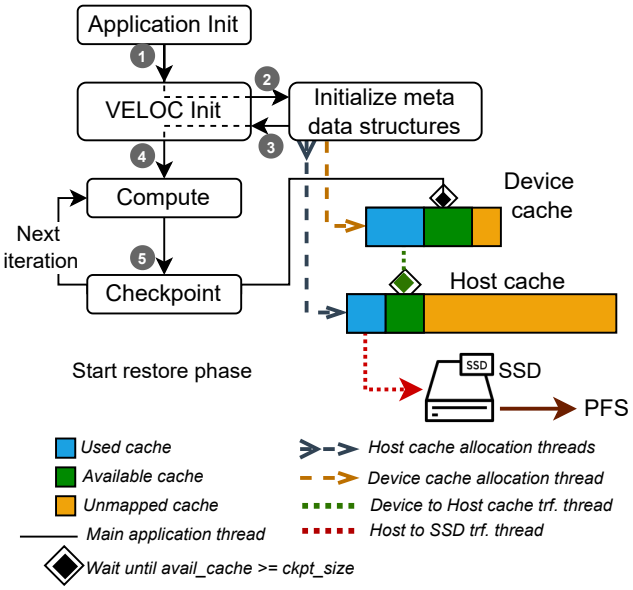


Fig. 2: VELOC memory mapping and checkpointing workflow

buffer is touched, it is pinned using `cudaHostRegister`, which then allow for faster DMA transfers.

Until the host buffer is fully touched, checkpoints can be written to the touched or untouched regions as per the performance model described in Section IV-B. However, unlike the host buffer, checkpoints cannot be written to un-mapped regions of the device cache, thus checkpoints to the device cache are blocked until the required amount of memory is mapped by the device-allocation thread. Finally, if any of the cache tiers are full with checkpoints, we evict one or more of the existing checkpoints that have been flushed to a slower memory tier using a first in first out (FIFO) schedule to accommodate the next checkpoint.

V. Evaluation

A. Experimental Setup

We conduct our experiments on the ThetaGPU HPC testbed, 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 Gen4 NVME drives (4 GB/s) and 8 Nvidia A100 Tensor core GPUs (aggregating to a total of 320 GB HBM memory). The nodes have access to a 10 PB Lustre [38] PFS that is accessible through a POSIX mount point. On each node, the 8 A100 GPUs are interconnected with each other using 6 NVSwitches and with the host memory through a SXM4 interface. The peak unidirectional Device-to-Device, pinned, and unpinned Device-to-Host (and vice versa) bandwidths on each GPU are 500 GB/s, 25 GB/s and 12 GB/s, respectively. Two GPUs share the same interconnect to host buffer, which effectively reduces their Device-To-Host bandwidths during concurrent access, and also allows for only 4 out of the 8 available NUMA domains to be directly accessible from the GPUs. To emphasize the cost of node-local cache initialization, we use a single node consisting of 8 GPUs in our experiments.

B. Compared Approaches

Throughout our evaluations, we compare the following approaches for initializing the device and the host buffers.

1) Device buffer: We compare the following two approaches for initializing the device buffer:

Direct allocation: Uses the `cudaMalloc` API, and blocks the application's I/O operations until the reserved buffer on the device is fully allocated and mapped. While state-of-the-art HPC checkpoint-restore runtimes such as FTI [27] rely on checkpointing directly to the host buffer, frameworks such as PyTorch [3] initialize the device buffer for checkpointing and/or staging data using this approach. Therefore, we consider this approach as the baseline for comparison against our proposed approach.

Our approach: As outlined in Section IV-A, this approach uses CUDAs virtual memory management (VMM) technique that first reserves virtual address space on the device and later incrementally maps it in chunks to physical HBM memory. Incrementing in chunks smaller than 1 GB results in slower device memory allocation rates. Therefore we increment the VMM buffer in chunks of 1 GB.

2) Host buffer: For initializing the host buffer, we compare the following approaches. The notations used in the evaluation graphs are abbreviated in bold font.

Direct pin *(standard allocation and registration):* This approach refers to the standard technique of allocating pinned memory on the host buffer in using the CUDA API `cudaHostMalloc`. Checkpoint-restore runtimes such as FTI [27] allocate the host buffer using this approach, and thus we consider it as the baseline representing the current state-of-the-art host cache initialization approach.

UVM *(Unified Virtual Memory):* In this approach, we allocate the collective device and host buffer using a single UVM call on each process. Since the UVM presents a general purpose multi-tier memory management approach adopted by CUDA drivers, it implicitly balances trade-off such as on-demand page touching, evictions, locality-aware data placement, and transparent interaction between host and device buffers, and is thus relevant to our use-case.

Incremental memset: This approach corresponds to the concurrent touch and flush technique outlined in Figure 1. Default Linux pages of 4 KB are used during allocation, and the touch operation sets the entire contents of page range to a default value (0).

Our approach: Lastly, we compare the approach implemented using our proposed design principles (§ IV-A) as described in Section IV-B. Our optimized approach uses transparent hugepages of 2 MB, and we incrementally touch only the first bit of the allocated pages which is enough to create a physical mapping of entire pages.

C. Evaluation Methodology

For each of the aforementioned approaches (§ V-B), we evaluate the total checkpoint and restore overheads observed by the application across all the processes. In our evaluations, we report the time for which the application was blocked during checkpointing, and blocked for total I/O (checkpoint+restart) during its complete execution cycle, i.e. the aggregated checkpointing and restoring time.

We consider the case of two complementary applications to evaluate our proposal that generate uniform and variable sized checkpoints respectively. Applications that checkpoint for revisiting previous states or fault-tolerance, typically produce checkpoints of homogeneous sizes, primarily because the same set of critical data structures, or intermediate buffer is checkpointed. On the other hand, applications that run data-reduction techniques such as compression and decimation before checkpointing generate checkpoints of varying sizes.

To evaluate the case of variable-sized checkpoints, we consider a real-life HPC adjoint computation workload- Reverse Time Migration (RTM) [39], that is extensively used in the oil and gas industry, specifically for generating subsurface images from seismic data. In RTM, the forward and backward propagated wave fields are first calculated for a known propagation model, after which the two wave fields are cross-correlated in time to form the subsurface image. Since the wave fields need to be combined at identical propagation times, one of the wave fields is reversed in time using checkpoint-restore techniques [40]. The wave fields to be checkpointed can amount to several terabytes for large-scale production runs, because of which compression is crucial to mitigate I/O overheads during transfers.

To study factors such as compute time between consecutive checkpoints or restore operations, number of seismic images (shots) processed per node, and the amount of host buffer required per process, we design a set of benchmarks that emulates the behaviour of RTM application based on the traces (§ V-D) collected from full-scale production run. While our benchmark performs trivial iterations by sleeping for the specified compute interval, it generates checkpoints of the same sizes as observed in the traces, and hence is representative for demonstrating the effects of our proposed approach for real-world settings.

Considering the restoration of checkpoints done in reversed order by the RTM application, we adopt a reverse restore pattern for both uniform and variable sized checkpoint benchmarks, such that the checkpoints written during the end of the checkpoint cycle will be restored and consumed first by the application. Similar to the state-of-the-art checkpoint-restore runtimes [30], we enable prefetching in our benchmarks, that enable the checkpoints to be restored in the near future to be staged on the device buffer for faster Device-To-Device transfer when required by the application. However, since the `UVM` approach transparently manages prefetching using CUDA's density based prefetcher, we do not enable explicit prefetching in this approach.

We evaluate each of the aforementioned approaches against two configurations- the first in which the device buffer is large enough to accommodate all the checkpoints of a given process, and the second in which the host-cache is large enough to hold all checkpoints generated by its process. These configurations are abbreviated as `large-device` and `large-host` re-

spectively. For the case of `large-device`, 32 GB of device memory (80% of total capacity) is allocated as device buffer, and since it can contain all the checkpoints, the host buffer becomes redundant in this configuration. In the `large-host` configuration, we allocate 4 GB of device memory (10% of its capacity) as device buffer and 32 GB of main memory per process to be used as the host-cache, which when aggregated across 8 processes constitutes to 25% of total host memory. Both the `large-device` and `large-host` configurations are representative of the spare memory available during execution cycle of the RTM application for low and high-frequency seismic simulations respectively. Note that VELOC support asynchronous data movements further from host-cache to local and remote storage, but we do not consider them under the scope for our study that aims to ephemerally store the intermediate checkpoints for short-lived computations. Unless otherwise noted, the device buffer for `large-device` and `large-host` are 32 GB and 4 GB respectively, and the host buffer for `large-host` is set to 32 GB.

D. Shot Traces

In this section, we describe the distribution of checkpoints sizes and compute intervals for both uniform and variable-sized checkpoints.

Variable checkpoint sizes: The RTM application compresses the intermediate data generated during the forward pass before checkpointing it. Since it uses lossy compression techniques, that yield as high as $\sim 30\times$ compression ratio, the checkpoint sizes differ not only across different processes, but also within same process for successive checkpoint operations. Figure 3a depicts the checkpoint size distribution per process for 8 representative shots. The aggregated size of each shot ranges between 20 GB to 24 GB.

Instead of capturing checkpoints after every timestep, to reduce the I/O, they can be captured after certain number of timesteps and can be interpolated later when required in restore phase. Capturing checkpoints after a large number of timesteps minimizes the required I/O but also reduces the image quality. However, depending on the size and frequency of the input shot, the checkpoint capture interval can range from checkpointing after succesive timesteps to checkpointing after tens of timesteps have been evaluated. Therefore, we evaluate for varying capture intervals which are represented in terms of compute time between consecutive checkpoints.

Uniform checkpoint sizes: Next, we consider the case when the HPC application performs an even domain decomposition and generates checkpoints of homogeneous sizes, either by capturing checkpoints in raw form of applying fixed size output data reduction techniques such as truncating, padding, etc. Unlike the case of variable-sized checkpoints, in this case all processes attempt to capture checkpoints at simultaneously, that leads to contention on device-host memory interconnect (two GPUs share the same SXM), and the writes made on the host buffer. We derive the checkpoint size for this class of applications from the RTM traces, where the size of every checkpoint is 128 MB, which roughly corresponds to 50 percentile distribution of 1600 RTM shots. We capture 256 checkpoints on each process, which amounts to 32 GB of checkpoints per GPU.

E. Results

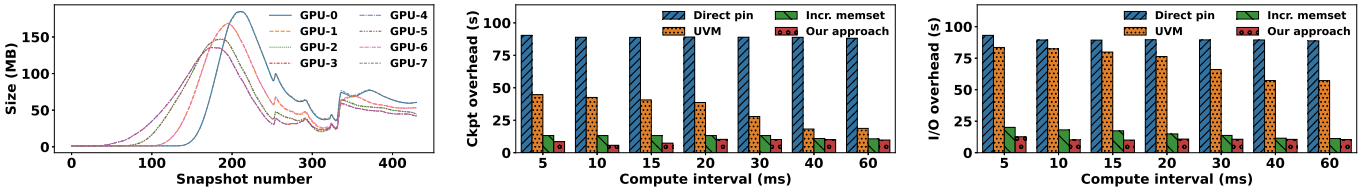
Cache initialization throughput: We first evaluate the initialization throughput of the device memory by measuring the time taken to allocate, map and pin 32 GB of device buffer (80% of HBM capacity) per GPU. We launch multiple processes such that each GPU is exclusively associated with a single process. All processes then concurrently initialize their entire device buffer, which resembles the with concurrent initialization pattern of real-world adjoint computations. As observed in Figure 4a, the allocation throughput decreases with an increasing number of processes. More specifically, we observe up to 1.2 TB/s initialization rate for a single process, which drops down to 160 GB/s for 8 processes. Although every process independently handles the memory allocation of its local GPU, the CUDA drivers by default enables peer access of the allocated device buffer, due to which it needs to sequentially register this memory in all of its peers' CUDA contexts for inter-GPU memory access. Thus, we observe that the allocation rate of the device buffer is inversely proportional to the number of processes.

For the case of host-cache, we observe no difference in the allocation and touching throughput on increasing the number of processes to a maximum of 8 processes (total number of GPUs on our system). This is due to the fact that we have a total of 8 NUMA domains, each having their independent PCIe and cache access lines thereby making allocations and touches scalable for our system. However, similar to the case of device buffer the rate of pinning the host buffer is inversely proportional to the number of processes, that effectively reduces the overall cache initialization rate for an increasing number of processes.

As mentioned in Section IV-A, the rate of mapping virtual to physical memory is 3 GB/s, which is done either during cache initialization or direct transfers. Therefore, for writing checkpoints of 128 MB, we do not experience any transfer stalls if the time between consecutive checkpoints is greater than 43 ms. Thanks to our incremental device based cache that acts as an intermediary buffer between application and host cache, our system enables minimum transfer overheads even when consecutive checkpoints are issued significantly faster than the host cache allocation rate.

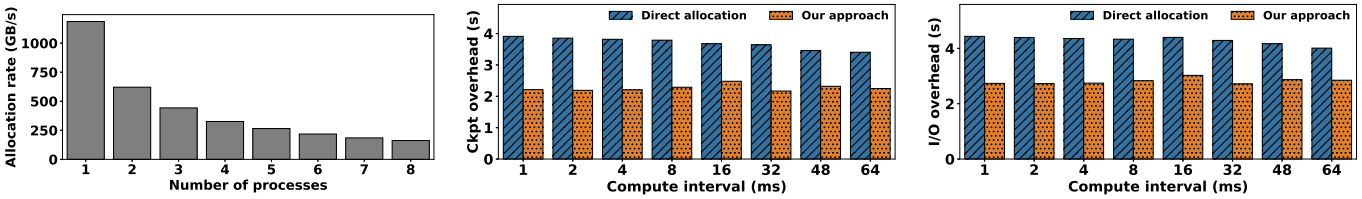
Checkpointing and overall blocking time of the application: Our next set of experiments evaluates the blocking time per process of the application during checkpoint and restore operations. The blocking time per process is reported as the (total blocking time of all processes divided by the total number of processes).

We first evaluate the case of uniform-sized checkpoints for the `large-device` configuration. As observed in Figure 4b, we observe up to 76% faster checkpointing when device buffer is allocated using our proposed incremental CUDA virtual-memory management based approach. Since the



(a) RTM checkpoint sizes **(b)** Checkpoint overhead **(c)** Ckpt+restore overhead

Fig. 3: Checkpoint sizes per GPU of RTM application running on 8 GPUs (Fig. 3a). Total checkpoint (Fig. 3b) and checkpoint+restore overhead (Fig. 3c) for variable-sized checkpoints for the large-host configuration.



(a) Device buffer allocation rate **(b)** Checkpoint overhead **(c)** Ckpt+restore overhead

Fig. 4: Device allocation rate with an increasing number of processes per node (Fig. 4a). Total checkpoint (Fig. 4b) and checkpoint+restore (Fig. 4c) overhead for uniform-sized checkpoints for large-device configuration.

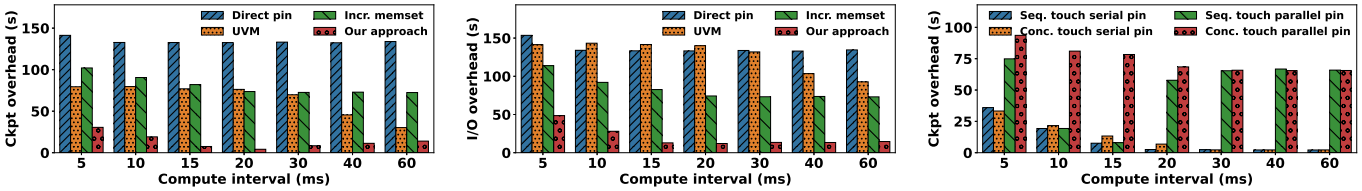
checkpoints consume the entire device buffer, our incremental cache approach initializes the entire cache before the restore phase begins, thereby performing restore operations at peak Device-To-Device bandwidths without any interference from cache allocation. Additionally, as seen in Figure 4c, the total application overhead is reduced by up to 63% when using our approach as compared to initializing device buffer using the baseline (Direct allocation).

Next, we consider the case of uniform-sized checkpoints by using the `large-host` configuration. Figure 5a and Figure 5b depict the overheads associated with checkpointing and overall I/O of the application for different compute intervals. On comparing the performance of `Direct pin`, `UVM` and `Incr. memset`, we observe that the UVM approach outperforms the other two approaches during checkpoint cycle, primarily due to on-demand cache initialization as opposed to the proactive cache initialization adopted by the other approaches. However, when comparing the overall I/O overhead, the cost of proactive cost initialization pays off for the `Incr. memset` approach, while the UVM still pays more, if not the same overall I/O overhead as the Direct pin approach for short compute times. The on-demand overhead of UVM is amortized by larger compute intervals, which leads to smaller I/O overheads. Nonetheless, our proposed approach outperforms the baseline (Direct pin) by up to 12.5× and 7.8× respectively for checkpointing and checkpoint+restore cycles respectively. We observe a similar behaviour of the compared approaches for the case of variable-sized checkpoints as depicted in Figure 3b and Figure 3c. For variable sized checkpoints, the total checkpointing and checkpoint+restart overhead is reduced by 16.7× and 9.3× respectively when using our approach as compared to the baseline.

Comparison of sequential vs. concurrent cache initialization and serial vs. parallel pinning for host buffer: We next evaluate the impact of concurrency control performance model by comparing the sequential initialization approach with the concurrent initialization approach for the case of uniform-sized checkpoints. We also study the performance implications of performing serial pinning of the host buffer as compared to parallel pinning. As outlined in Figure 5c, the sequential pinning approach performs on average 6.4×, and a maximum of 8× faster as compared to parallel pinning approach, which supports our design principle of serially pinning the host-cache.

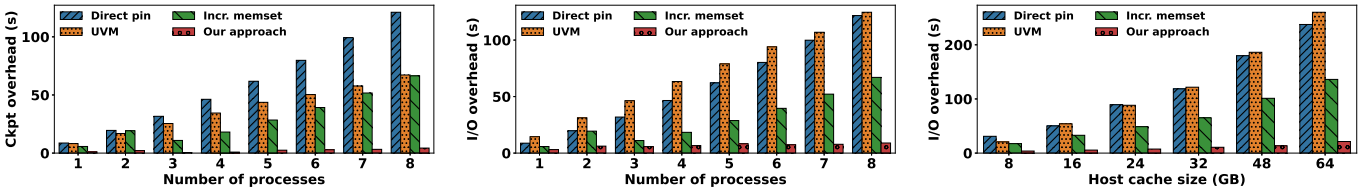
Next, we observe that the sequential touching and checkpoint writes approach outperforms the concurrent approach for all compute intervals except the smallest interval of 5ms when the host buffer is pinned serially. This is because for small intervals during sequential touching, the forerunner threads do not get enough time-slices in between checkpoint writes to touch enough pages for the next checkpoint, due to which the next checkpoint resort to writing on untouched pages. As compared to this case of sequential touching, for smaller compute intervals the concurrent touch and write approach outperforms the sequential approach by 10%. However, for larger compute intervals, the sequential touch approach proves to be on average 43% and a maximum of 2.7× faster as compared to the concurrent memory touching approach. These results demonstrate the effectiveness of our proposed performance model that dynamically chooses between the different optimization techniques to yield the lowest checkpointing and overall I/O overhead for the application.

Scalability: In our next set of experiments, we study the scalability of our proposed allocation techniques by increasing the number of processes on a given node from 1...8 (max-



(a) Checkpoint overhead **(b)** Ckpt+restore overhead **(c)** Sequential vs concurrent checkpoint overhead

Fig. 5: Total checkpoint (Fig. 5a) and checkpoint+restore (Fig. 5b) overhead for uniform-sized checkpoints for large-host configuration. Impact of sequential vs concurrent touching and serial vs parallel pinning of host buffer for different compute intervals (Fig. 5c).



(a) Total checkpoint overhead for increasing processes **(b)** Total ckpt+restore overhead for increasing processes **(c)** Total ckpt+restore overhead for increasing host-cache

Fig. 6: Weak scaling for the total checkpoint (Fig. 6a) and checkpoint+restore overhead (Fig. 6b). Total application blocking time during checkpoint+restore on increasing the host buffer (Fig. 6c).

imum number of GPUs) for uniform sized checkpoints. For this experiment, we select a compute interval of 20ms (that represents the mean checkpoint interval of RTM traces with similar checkpoint sizes) on the `large-host` configuration. Figure 6a and Figure 6b represent the total amount of time a process process was blocked on checkpointing and the total I/O (checkpoint+restore). For a single process, our approach yields 7.6× and 4.7× lower checkpointing and total I/O overheads respectively as compared to the baseline approach. As we increase the number of processes to 8, we observe 27.8× and 13.6× lower checkpoint and total I/O overheads with our approach relative to the baseline. With more number of processes, our optimizations such as concurrency control to optimize competition between writes and eager touching, and lazy sequential pinning contribute more significantly towards reducing the I/O wait time of the application. This demonstrates that our approach achieves significant performance improvements as compared to the baseline even at scale.

Lastly, we evaluate how does our approach performs for an increasing host buffer per process. As seen in Figure 6c, our approach outperforms the baseline approach from 4.8× to 12.4× for various host buffer size per process ranging from 8...64 GB The results of this experiment indicate that larger host buffer provide more opportunity for overlapping cache touching and registration with ongoing checkpointing; thereby illustrating the effectiveness of our proposed approach for minimizing the I/O overheads of the targeted set of short-running applications performing checkpoint-restore at high-frequencies.

VI. Conclusions

In this paper, we address the problem of reducing initialization overheads for multi-level checkpointing strategies of short-lived adjoint computations running on multiple GPUs co-located on the same HPC node. Unlike state of art approaches that pay a high initialization cost upfront (map and pin host memory pages), we propose a new approach that combines GPU cache optimizations (incremental allocation) with host cache optimizations (asynchronous opportunistic touching of the memory pages to reduce pinning overheads) We implemented a prototype of our proposal based on VELOC, a production-ready multi-level HPC checkpoint-restart runtime. We ran extensive experiments that compare state of art approaches with our proposal. Our results show an average of ∼12× faster checkpointing during the forward pass (considering and amortized initialization cost) and ∼8× lower overall I/O overhead when combining the checkpointing overhead during the forward pass with the restore overhead during the backward pass. Encouraged by these results, in future work we plan to explore two complementary directions: (1) a globally shared host buffer across all GPUs that improves the overall host memory utilization by reducing fragmentation (at the cost of higher synchronization overheads); (2) support for Nvidia GPUDirect storage that enables the use of a third tier (NVMe based SSDs) in addition to the host buffer.

Acknowledgments

This work is supported in part by the ARAMCO Services Company and the U.S. Department of Energy (DOE), Office of Science, Office of Advanced Scientific Computing Research and Argonne National Laboratory, under contract numbers PRJ1008127, Argonne: 0F-60169/DOE: DE-AC02-06CH11357. The initial versions of results were obtained using the Chameleon and CloudLab testbeds supported by the National Science Foundations.

REFERENCES

[1] R. Ross, L. Ward, G. Grider, S. Klasky, G. Lockwood, K. Mohror, and B. Settlemyer, "Storage systems and i/o: Organizing, storing, and accessing data for scientific discovery," Department of Energy, Office of Science, Tech. Rep., 2019.

[2] J. Utke, U. Naumann, M. Fagan, N. Tallent, M. Strout, P. Heimbach, C. Hill, and C. Wunsch, "Openad/f: A modular open-source tool for automatic differentiation of fortran codes," vol. 34, no. 4, jul 2008. [Online]. Available: https://doi.org/10.1145/1377596.1377598

[3] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer, "Automatic differentiation in pytorch," in *NIPS 2017 Workshop on Autodiff*, Long Beach, USA, 2017.

[4] R. M. Errico, "What is an adjoint model?" *Bulletin of the American Meteorological Society*, vol. 78, no. 11, pp. 2577–2591, 1997.

[5] A. Lindsay, R. Stogner, D. Gaston, D. Schwen, C. Matthews, W. Jiang, L. K. Aagesen, R. Carlsen, F. Kong, A. Slaughter *et al.*, "Automatic differentiation in metaphysicl and its applications in moose," *Nuclear Technology*, pp. 1–18, 2021.

[6] 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 *HiPC'19: The IEEE 26th International Conference on High Performance Computing, Data, and Analytics*, 2019, pp. 312–322.

[7] H. Wan, "Deep learning: Neural network, optimizing method and libraries review," in *ICRIS'19: The 2019 International Conference on Robots and Intelligent System*, 2019, pp. 497–500.

[8] T. Allen and R. Ge, "In-depth analyses of unified virtual memory system for gpu accelerated computing," in *SC '21: International Conference for High Performance Computing, Networking, Storage and Analysis*, St. Louis, USA, 2021.

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

[10] M. Aigner, C. M. Kirsch, M. Lippautz, and A. Sokolova, "Fast, multicore-scalable, low-fragmentation memory allocation through large virtual memory and global data structures," in *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA 2015. New York, NY, USA: Association for Computing Machinery, 2015, p. 451–469. [Online]. Available: https://doi.org/10.1145/2814270.2814294

[11] A. Pi, J. Zhao, S. Wang, and X. Zhou, "Memory at your service: fast memory allocation for latency-critical services," in *Proceedings of the 22nd International Middleware Conference*, 2021, pp. 185–197.

[12] J. Evans, "A scalable concurrent malloc (3) implementation for freebsd," in *Proc. of the bsdcan conference, ottawa, canada*, 2006.

[13] "Tcmalloc : Thread-caching malloc," https://gperftools.github.io/gperftools/tcmalloc.html.

[14] E. D. Berger, K. S. McKinley, R. D. Blumofe, and P. R. Wilson, "Hoard: A scalable memory allocator for multithreaded applications," *ACM Sigplan Notices*, vol. 35, no. 11, pp. 117–128, 2000.

[15] R. L. Hudson, B. Saha, A.-R. Adl-Tabatabai, and B. C. Hertzberg, "Mcrt-malloc: A scalable transactional memory allocator," in *Proceedings of the 5th international symposium on Memory management*, 2006, pp. 74–83.

[16] S. Seo, J. Kim, and J. Lee, "Sfmalloc: A lock-free and mostly synchronization-free dynamic memory allocator for manycores," in *2011 International Conference on Parallel Architectures and Compilation Techniques*. IEEE, 2011, pp. 253–263.

[17] B. C. Kuszmaul, "Supermalloc: A super fast multithreaded malloc for 64-bit machines," in *Proceedings of the 2015 International Symposium on Memory Management*, ser. ISMM '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 41–55. [Online]. Available: https://doi.org/10.1145/2754169.2754178

[18] J. Cao, K. Arya, R. Garg, S. Matott, D. K. Panda, H. Subramoni, J. Vienne, and G. Cooperman, "System-level scalable checkpoint-restart for petascale computing," in *ICPADS'16: The 22nd IEEE Int. Conf. on Parallel and Distributed Systems*. IEEE Press, 2016, pp. 932–941.

[19] P. H. Hargrove and J. C. Duell, "Berkeley lab checkpoint/restart (BLCR) for linux clusters," in *Journal of Physics: Conference Series*, vol. 46, no. 1. IOP Publishing, 2006, p. 067.

[20] L. Bautista-Gomez, S. Tsuboi, D. Komatitsch, F. Cappello, N. Maruyama, and S. Matsuoka, "FTI: High performance Fault Tolerance Interface for hybrid systems," in *Proc. ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2011.

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

[22] B. Nicolae, A. Moody, G. Kosinovsky, K. Mohror, and F. Cappello, "VELOC: very low overhead checkpointing in the age of exascale," *CoRR*, vol. abs/2103.02131, 2021.

[23] H. Takizawa, K. Sato, K. Komatsu, and H. Kobayashi, "Checuda: A checkpoint/restart tool for cuda applications," in *Proc. International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT)*, 2009, pp. 408–413.

[24] A. Nukada, H. Takizawa, and S. Matsuoka, "Nvcr: A transparent checkpoint-restart library for nvidia cuda," in *Proc. IEEE International Symposium on Parallel and Distributed Processing Workshops and PhD Forum (IPDPSW)*, 2011.

[25] T. Suzuki, A. Nukada, and S. Matsuoka, "Transparent checkpoint and restart technology for cuda applications," in *Proc. GPU Technology Conference (GTC)*, 2016.

[26] R. Garg, A. Mohan, M. Sullivan, and G. Cooperman, "CRUM: Checkpoint-restart support for CUDA's unified memory," in *Proc. IEEE International Conference on Cluster Computing (CLUSTER)*, 2018.

[27] K. Parasyris, K. Keller, L. Bautista-Gomez, and O. Unsal, "Checkpoint restart support for heterogeneous hpc applications," in *Proc. IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*, 2020.

[28] M. Siavvas and E. Gelenbe, "Optimum checkpoints for programs with loops," *Simulation Modelling Practice and Theory*, 2019.

[29] S. Xiao, P. Balaji, J. Dinan, Q. Zhu, R. Thakur, S. Coghlan *et al.*, "Transparent accelerator migration in a virtualized gpu environment," in *Proc. IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, 2012.

[30] J. Mohan, A. Phanishayee, and V. Chidambaram, "Checkfreq: Frequent, fine-grained DNN checkpointing," in *Proc. USENIX Conference on File and Storage Technologies (FAST)*, 2021.

[31] K. Lee, M. B. Sullivan, S. K. S. Hari, T. Tsai, S. W. Keckler, and M. Erez, "Gpu snapshot: checkpoint offloading for gpu-dense systems," in *Proc. ACM International Conference on Supercomputing (ICS)*, 2019.

[32] 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. IEEE International Conference on High Performance Computing, Data, and Analytics (HiPC)*, 2019.

[33] P. Subedi, P. Davis, S. Duan, S. Klasky, H. Kolla, and M. Parashar, "Stacker: an autonomic data movement engine for extreme-scale data staging-based in-situ workflows," in *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2018, pp. 920–930.

[34] H. Abbasi, M. Wolf, G. Eisenhauer, S. Klasky, K. Schwan, and F. Zheng, "Datastager: scalable data staging services for petascale applications," *Cluster Computing*, vol. 13, no. 3, 2010.

[35] C. Docan, M. Parashar, and S. Klasky, "Dataspaces: an interaction and coordination framework for coupled simulation workflows," *Cluster Computing*, vol. 15, no. 2, pp. 163–181, 2012.

[36] B. Dong, S. Byna, K. Wu, H. Johansen, J. N. Johnson, N. Keen *et al.*, "Data elevator: Low-contention data movement in hierarchical storage system," in *2016 IEEE 23rd International Conference on High Performance Computing (HiPC)*. IEEE, 2016, pp. 152–161.

[37] T. Wang, S. Oral, M. Pritchard, B. Wang, and W. Yu, "Trio: Burst buffer based i/o orchestration," in *2015 IEEE International Conference on Cluster Computing*. IEEE, 2015, pp. 194–203.

[38] "Lustre: A parallel file system that supports many requirements of leadership class hpc simulation environments," https://www.lustre.org/.

[39] H.-W. Zhou, H. Hu, Z. Zou, Y. Wo, and O. Youn, "Reverse time migration: A prospect of seismic imaging methodology," *Earth-Science Reviews*, vol. 179, pp. 207–227, 2018.

[40] 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.