GPU Snapshot: Checkpoint Offloading for GPU-Dense Systems

Kyushick Lee
University of Texas at Austin
kyushickl@utexas.edu

Michael B. Sullivan
NVIDIA
misullivan@nvidia.com

Siva Kumar Sastry Hari
NVIDIA
shari@nvidia.com

Timothy Tsai
NVIDIA
timothyt@nvidia.com

Stephen W. Keckler
NVIDIA
skeckler@nvidia.com

Mattan Erez
University of Texas at Austin
mattan.erez@utexas.edu

Abstract

Future High-Performance Computing (HPC) systems will likely be composed of accelerator-dense heterogeneous computers because accelerators are able to deliver higher performance at lower costs, socket counts and energy consumption. Such accelerator-dense nodes pose a reliability challenge because preserving a large amount of state within accelerators for checkpointing incurs significant overhead. Checkpointing multiple accelerators at the same time, which is necessary to obtain a consistent coordinated checkpoint, overwhelms the host interconnect, memory and IO bandwidths. We propose GPU Snapshot to mitigate this issue by: (1) enabling a fast logical snapshot to be taken, while actual checkpointed state is transferred asynchronously to alleviate bandwidth hot spots; (2) using incremental checkpoints that reduce the volume of data transferred; and (3) checkpoint offloading to limit accelerator complexity and effectively utilize the host. As a concrete example, we describe and evaluate the design tradeoffs of GPU Snapshot in the context of a GPU-dense multi-exascale HPC system. We demonstrate 4–40X checkpoint overhead reductions at the node level, which enables a system with GPU Snapshot to approach the performance of a system with idealized GPU checkpointing.

CCS Concepts

• **Computer systems organization** → *Processors and memory architectures.*

Keywords

Resilience, GPU, Fault Tolerance

ACM Reference Format:
Kyushick Lee, Michael B. Sullivan, Siva Kumar Sastry Hari, Timothy Tsai, Stephen W. Keckler, and Mattan Erez. 2019. GPU Snapshot: Checkpoint Offloading for GPU-Dense Systems. In *2019 International Conference on Supercomputing (ICS '19), June 26–28, 2019, Phoenix, AZ, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3330345.3330361

ICS '19, June 26–28, 2019, Phoenix, AZ, USA

ACM ISBN 978-1-4503-6079-1/19/06…$15.00
https://doi.org/10.1145/3330345.3330361

1 Introduction

We identify and mitigate a new challenge in the use of accelerator-dense systems for high-performance computing (HPC): the problem that preserving a coordinated, consistent checkpoint of an accelerator-dense node overwhelms intra-node memory bandwidth, degrading overall performance significantly. This is important because accelerator-dense systems [14, 37, 41, 45, 53]—systems with a high ratio of accelerators to CPUs—offer an attractive and affordable path to very high (multi-exaflop) performance, yet the high memory capacity of the accelerators must be preserved as part of the overall end-to-end resilience scheme.

State-of-the-art resilience schemes for HPC quickly duplicate node memory to fast-but-not-fully-reliable storage [7, 28, 31]; some of these checkpoints are later transferred to a global reliable medium. Accelerators introduce a severe problem for this hierarchical scheme. At each node checkpoint time, application state within the large memories of all accelerators within the node must be preserved, causing a burst of preservation traffic such that every accelerator can utilize only a fraction of the shared intra-node communication (e.g., PCIe or NVLink), host memory, and for some schemes also IO bandwidths. Figure 1 illustrates the organization of an accelerator-dense system, highlighting the intra-node preservation bandwidth as the main performance limiter. Unlike with current node-level checkpoint mechanisms, preservation burstiness cannot be resolved by an intra-accelerator copy because that memory is typically heavily utilized, and an additional storage layer cannot be added as with burst buffers for CPU memory [28].

We show that this burstiness problem will cost precious system performance for future multi-exaflop, GPU-dense HPC systems, degrading throughput by $5-10\%$ when projecting current HPC trends. We therefore develop *GPU Snapshot*, a host–accelerator HW/SW cooperative mechanism to reduce the time required to preserve accelerator memory contents during checkpoint. GPU Snapshot addresses the burstiness challenge by optimizing the use of the tightly-constrained host–accelerator interconnect while minimizing interruptions to both the host and accelerators. Note that this is a significant challenge as the aggregate capacity of accelerator memories is large and optimized programs avoid the frequent transfer of accelerator data to host memory because of the large discrepancy between the bandwidth within the accelerator and the host–accelerator interconnect.

At a high level, GPU Snapshot enables the incremental and non-blocking preservation of accelerator memory to reduce the checkpoint volume and to overlap checkpointing with continued GPU execution. GPU Snapshot allows for a very fast logical snapshot to be marked with the state corresponding to that snapshot then

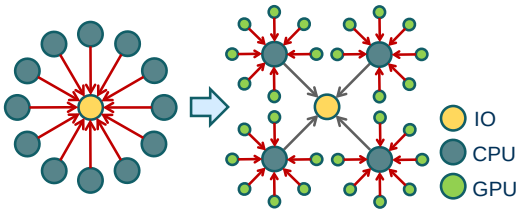


Figure 1: Illustration of bandwidth hot spots due to checkpointing in CPU-only and GPU-dense system.

asynchronously preserved to host memory. The snapshot must be logical because accelerator memory is typically highly utilized with insufficient capacity to duplicate data within the accelerator. Asynchronous non-blocking preservation is necessary to "spread out" the transfer of the data to the host without causing traffic bursts.

GPU Snapshot introduces the notion of *checkpoint offloading*, and we architect the mechanisms and interfaces necessary for enabling this offloading. With checkpoint offloading, the data preservation part of the checkpointing operation and its progress are not handled by the accelerator being checkpointed; rather in our case this is handled by the host CPU. On the GPU device, GPU Snapshot introduces new, low-cost hardware to track which memory regions need to be transferred and those that have already been transferred, possibly at a different granularity from the virtual memory system. This is important for accelerators that use coarse-grained address translation compared to the granularity at which data is written. Furthermore, this GPU Snapshot hardware optimizes how the host and the accelerator communicate to identify which regions have been safely preserved. Such offloading has not been necessary for CPUs, but we demonstrate its importance for accelerators.

Once a low-overhead accelerator snapshot has been stored to the host memory, state-of-the-art global checkpoint approaches, such as Scalable Checkpoint Restart [31] or burst buffers [7], can perform an effective and efficient global checkpoint across all nodes. We model these mechanisms in our evaluation but do not discuss them in detail as GPU Snapshot interacts similarly with any global recovery scheme. We emphasize that such mechanisms cannot be used directly by accelerators because of the inability to duplicate data within accelerator memory.

To summarize, we make the following main contributions:

- We identify that the burstiness of checkpointing traffic within an accelerator-dense node can degrade overall system performance significantly. For example, we show that a projected multi-exaflop GPU-dense system with more than 8 GPUs per CPU will perform 8% worse than a system that does not face the burstiness problem (Section 3).
- We develop GPU Snapshot hardware for logical snapshots followed by the asynchronous preservation of each snapshot to the host, overcoming the burstiness issue (Section 4).
- We introduce *checkpoint offloading*, which in our case allows the host to effectively perform the checkpoint while the accelerator continues its execution; we also discuss generalizations of this idea. In the case of GPU checkpointing, offloading is important because it does not require the GPU itself to perform IO operations (Section 4.1).
- We use a suite of HPC CUDA applications to demonstrate that GPU Snapshot reduces checkpoint overheads by $4-40\times$ and comes within 3% of the performance of a system with idealized GPU checkpointing in all but two applications studied; these two outliers are still within $\sim 5\%$ of the ideal performance (Section 6).

2 Background

2.1 Globally-Coordinated Checkpointing

Globally-coordinated checkpoint-restart is the most practical general error recovery mechanism. With global checkpoint-restart, all nodes that are running a particular application coordinate to quiesce to a consistent state, typically using a global barrier. After coordination, the globally-consistent program state is preserved to reliable storage. When a processor, node, or system failure occurs, the previous checkpoint is reloaded, the application is restarted, and error-free execution continues. So long as the time to take and restart a checkpoint is much smaller than the mean time between failures (MTBF), global checkpoint restart is very effective. Thus, an important objective of any checkpoint-restart system is to keep checkpoint times short.

One challenge with global checkpointing is that the checkpoint must be taken precisely after the globally-coordinated consistent state is established. When a large-scale application checkpoints, many nodes may attempt to simultaneously write their large checkpoints to a bandwidth-constrained global file system. Such checkpointing traffic bursts expand the checkpoint time and degrade system efficiency. While the global bandwidth is not sufficient for managing the traffic burst, the long interval between checkpoints provides for ample time to store the state of all the application processes. Hence, state-of-the-art checkpointing systems aim to spread the storing of checkpointing data from different nodes across the checkpointing interval.

There are two main ways this is achieved. The first is through distributed checkpointing protocols (e.g., [5, 13]), which are not common in practice. The second is to still globally coordinate a single checkpoint across nodes, but quickly store this checkpoint to fast, though less reliable, storage. This storage may consist of a node's own memory [31, 58], a local SSD [1], a fast network-attached remote SSD known as a burst buffer [7, 44], or even the memory of other nodes [31]. All of these systems mitigate the checkpointing-burst problem, with in-memory checkpointing offering the highest potential performance.

In addition to hiding the long checkpoint time required to write data to a reliable global file system, approaches have also been developed to reduce the volume that must be preserved. Two notable examples are: (1) incremental checkpointing, which only stores data that has been modified since the previous checkpoint [2, 4, 16, 18, 29, 39, 52, 55, 56]; and (2) compressing checkpointed data before writing a checkpoint [22, 23, 43].

2.2 Checkpoint-Restart with GPUs

As accelerators increase in popularity for large-scale computing, checkpointing accelerator state becomes important. While some accelerators have little state and any computation on the accelerator can be restarted from a CPU memory checkpoint, accelerators such as GPUs have large memories and retain data in those memories for long periods without saving it to the CPU host. As such, it is

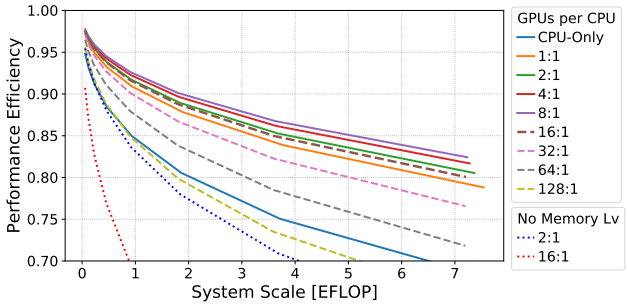


Figure 2: Performance efficiency vs. system scale and GPU density; the lower effective per-accelerator checkpointing bandwidth degrades efficiency as accelerator density (lines) and node count (x-axis) increase. Performance efficiency is the ratio between the execution time of the application without any resilience mechanisms and no failures and the total time including checkpoint and restart.

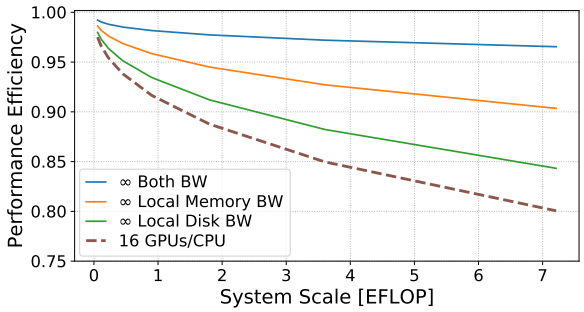


Figure 3: Limits on the performance efficiency of accelerator-dense checkpointing; improving memory-level checkpointing is the most effective option.

critical to include GPU memory state in the globally-coordinated checkpoints. With the prevalence of GPUs, checkpointing systems for CUDA applications are being explored [15, 35, 48]. The state of a CUDA program consists of two components. The first component is host-side state, such as driver memory, and open files and sockets, which are preserved by the CPU checkpointing system. The second component is the state within the GPU device memory that is required to correctly restart the application. CUDA checkpoint-restart prototypes demonstrate both the feasibility of checkpointing GPU state and that the majority of checkpointing time is from transferring data from the GPU to the host.

3 Motivation: Bursty GPU Preservation

We identify a new challenge with checkpointing multi-GPU CUDA programs on GPU-dense systems (and accelerator-dense systems by extension). This challenge is similar in nature to the checkpointing traffic burst problem discussed above for global IO, but within an accelerator-dense node. When a checkpoint is globally coordinated, all accelerator memory state must be transferred to the host CPU and then included in the global checkpoint. The host memory and interconnect bandwidth, however, are shared by all accelerators within the node such that each observes only a fraction of the total bandwidth while transferring its data to the host. Unlike the CPU-oriented checkpointing systems described earlier, the GPU does not have enough capacity to feasibly duplicate data within its memory. At the same time, even burst-buffer nodes cannot easily and economically support the burst of data from a GPU-dense node.

While distributed checkpointing systems have been demonstrated for two-sided MPI programs on CPUs, GPUs present challenges that make distributed checkpointing of GPU state prohibitively expensive. Current CUDA programming systems and GPU devices support a shared and coherent address space across the entire heterogeneous node through uniform virtual memory (UVM) [17, 36], where GPUs may communicate with the host CPUs or other GPUs in a one-sided manner. Handling such unstructured communication for distributed checkpointing requires a fine-grained, high-bandwidth, and high-capacity message logging infrastructure [6] that tracks remote memory accesses to or from each GPU in order to locally recover from errors; such infrastructure does not exist and developing it would require significant research and resources. Thus, enabling efficient global checkpoints for GPU-dense systems is important.

To demonstrate the problem of GPU checkpoint traffic bursts, we model state-of-the-art multi-level checkpointing for GPU-dense exascale systems using the methodology described in Sections 5.2 and 5.3. Figure 2 shows the efficiency of three-level checkpoint with optimal intervals, finding multi-level checkpointing to be promising yet insufficient at exascale. This organization resembles Scalable Checkpoint Restart [31] on a scaled-up version of the Coral Summit supercomputer with 3:1 GPU-to-CPU ratio at Oak Ridge National Laboratory [53]. (Scalable Checkpoint Restart is likely to be the highest performing checkpointing solution supported on the machine [60].) Our second checkpoint tier is located in node-local NVMe SSD storage [42]. Summit can also use NVMe storage in a burst buffer organization [60], but we leave the detailed analysis of this alternate organization for future work. The overhead of checkpointing the GPUs degrades overall system performance at densities of 8-to-1 and greater, in particular for very large-scale systems. Figure 3 shows the efficiency of three-level checkpoint in a 16 GPUs/CPU node with infinite intra-node preservation bandwidth, infinite local disk bandwidth, and both. This analysis shows that quickly storing GPU data to host memory is by far the most effective approach for regaining system performance. We develop GPU Snapshot for this purpose, and explain its operation below.

4 GPU Snapshot

Before detailing the various components, mechanisms, and optimizations we develop for GPU Snapshot, we first provide an overview of how GPU Snapshot is used and how its major components interact. GPU Snapshot includes three major components. First, the GPU driver running on the host manages host memory as a checkpoint space for the GPUs. Second, a *memory-zone monitor* (MZM) is integrated into each GPU memory channel controller and manages a (cached) table to track which physical memory zones are written to by the GPU (a zone is a set of contiguous per memory-channel physical addresses). Third, to improve the performance of GPU Snapshot, buffers and additional optimizations are also included.

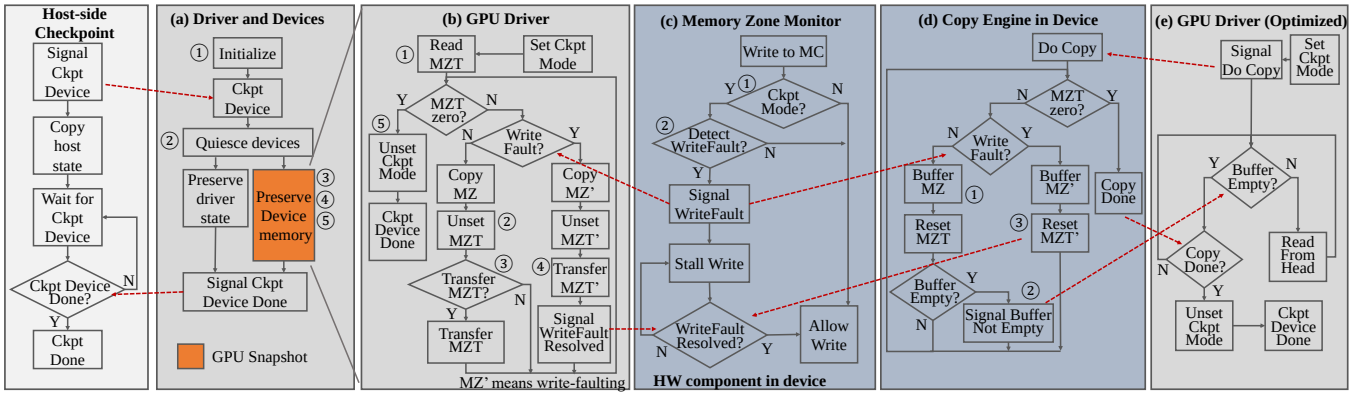


Figure 4: GPU Snapshot consists of closely-interacting driver and device components. (d), (e) illustrate the device-side optimization which simplifies the driver-side complexity of GPU Snapshot. (e) shows simpler design complexity than (b) due to offloading the checkpoint to an in-device copy engine, avoiding expensive GPU-CPU signals (red lines) on write faults.

There are several steps to using GPU Snapshot as illustrated in Figure 4. (a1) The driver allocates memory on the host for a preservation buffer as well as for metadata that associates preserved data with its original GPU physical address. The driver also allocates space in device memory for tracking the state of the memory zones. This memory zone table (MZT) is partitioned across memory channels. (a2) At a globally-coordinated time, the host CPU initiates GPU Snapshot through the GPU driver. Coordination within a node entails synchronizing all GPUs in the node to complete currently running kernels and flushing all GPU caches. This intra-node synchronization is needed due to the difficulty of (and lack of driver support for) checkpoints during kernel execution. Such synchronization is also needed for other multi-GPU checkpointing approaches, and it can be overlapped with the global inter-node barrier when bringing the system to a quiescent state [15]. We do not focus on synchronization as a source of overheads in our evaluation because it is common to all current GPU checkpointing approaches. (a3) The snapshot of current GPU memory state is logically recorded without yet transferring the data. (a4) GPU Snapshot proceeds with transferring snapshot data for the checkpoint from each GPU to the preservation buffer in the CPU. This is done using *checkpoint offloading* such that immediately following the snapshot (step (a3)), the GPU can resume normal execution. The GPU state preserved to the host becomes part of the global application state, which is later preserved in the host-local or burst buffer storage system. (a5) Checkpoint offloading proceeds with transferring the snapshot state to include in the node's checkpoint while the *memory zone monitor* (MZM) prevents now-executing kernels from overwriting snapshot state before it has been transferred to the host. This uses the MZT, which collects information on memory zones that are being modified such that an incremental checkpoint can be created at the next coordinated checkpoint time.

We first describe the checkpoint offload mechanism for non-blocking asynchronous GPU checkpointing (Section 4.1), as well as *incremental checkpointing* to reduce checkpointing volume (Section 4.2). We then discuss how additional hardware can mitigate some of the overheads associated with the driver and host–GPU communication (Section 4.3).

4.1 Checkpoint Offloading

State-of-the-art checkpointing systems for CPUs [7, 10, 31, 34] start by quickly duplicating memory to node-local memory or storage to allow the processor to continue executing while the checkpoint is transferred to more reliable media over a period of time (possibly until the next checkpoint is taken). This process is generally done by the CPU itself, with the responsibility of moving data across IO possibly relegated to a peripheral device [1]. This idea of duplicating memory to allow execution to continue is very effective at curbing checkpoint time. However, it is not directly applicable to GPU-dense systems; as we have explained earlier, there is insufficient memory in the GPU for duplicating state. Thus, our goal is to overlap GPU computation with transferring checkpointed data to CPU memory while avoiding expensive preservation bursts.

GPU Snapshot achieves this goal with two basic ideas: (1) enabling "live" memory to be used for temporarily storing checkpoint data until it is drained to CPU memory; and (2) enabling the effective management of this process by the CPU rather than the compute entity itself. These ideas require two key system mechanisms for their implementation. First, the GPU must identify writes to the part of the checkpointed memory that has not yet been copied to host memory and block those writes from modifying memory until the data has been transferred or duplicated. Second, checkpoint progress must be tracked to ensure that blocked writes are allowed to resume in a timely manner. We explain how GPU Snapshot implements these mechanisms below.

4.1.1 Offloading with Virtual Memory In a CPU-based system, virtual memory with the copy-on-write (CoW) approach can provide the mechanisms above. With CoW, all to-be-checkpointed virtual memory pages are marked as read-only at the initial snapshot time. When a store is issued to such a page, an exception is raised that "splits" the page by duplicating it such that one copy can be used for continued execution and the other for transferring checkpoint data. This is similar to the concept of virtual machine live migration [9, 20, 54]. However, the GPU virtual memory system cannot be used effectively for the same purpose because of four main reasons. First, GPUs (and most other accelerators) are designed for massive parallelism and throughput-computing and do not support efficient exception handling [26, 46, 49] for memory

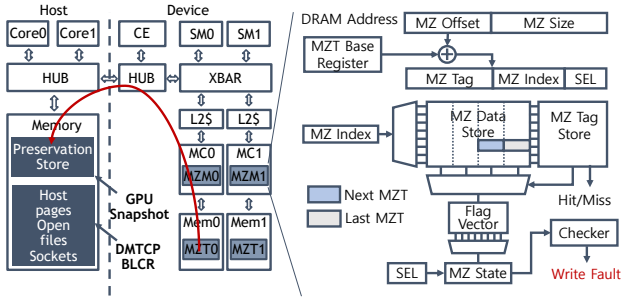


Figure 5: The microarchitecture of the memory zone monitor and memory zone table.

snapshot. Second, GPU memory (and again, that of accelerators in general) is managed by the host CPU to ensure a consistent mapping and enable effective DMAs and shared address spaces. Third, the coverage for dirty tracking is very limited with GPU L1 TLBs due to many-bit translations stored in the limited capacity and there is overlap among per-SM TLBs. Lastly, updating checkpoint progress either requires expensive TLB shootdowns or ends up with redundant preservation due to inconsistent updates. Hence, just-in-time duplication like CoW has very high latency. We find that GPU Snapshot cannot be implemented using virtual memory and that the progress rate is too low while a checkpoint is processed.

4.1.2 Offloading with Memory Zone Monitoring

We develop a new architecture that is specially designed to enabling checkpoint offloading in GPUs. The core of this architecture is the memory zone monitor (MZM). The MZM is not intended to replicate the capabilities of virtual memory. In fact, the MZM operates with physical addresses at the memory channel level because accelerators' physical memory state is being checkpointed. The MZM maintains information about physical memory zones, where each zone is a contiguous region of intra-channel physical addresses. We refer to *zones* rather than pages or frames to clarify that zones need not be at the same granularity used for virtual memory and because addresses are contiguous within a channel even though physical addresses are interleaved across channels.

The MZM architecture has two components. The first is the Memory Zone Table (MZT), which maintains the preservation status of zones; it is logically shared by the host and the device (Figure 5). The second component is the MZM hardware, which is integrated with the memory controllers and uses the MZT information to stall write operations that would otherwise prematurely modify GPU memory state.

4.1.3 Memory Zone Table and Offloading Coordination

When a checkpoint operation begins with a snapshot, the MZT contains which memory zones need to be transferred to the host. (b1) The driver can set up the table by updating the GPU memory locations that correspond to the MZT. Alternatively, we discuss later how the MZM hardware can keep the table up-to-date with only those regions that have been modified since the previous checkpoint to enable incremental checkpointing. (b2) As the checkpoint offloader copies GPU memory to the host, it updates the shared MZT. If the offloader is managed by the driver on the host, updating the MZT requires communication over PCIe (or NVLink), which incurs high latency. (b3) It is therefore desirable for the driver to only periodically update the checkpointing progress. The MZT is conservative by nature—only zones that are guaranteed to have been copied to the host are marked as safe for memory writes. Updating the MZT status during checkpoint offloading can thus be considered a performance-improving hint. We evaluate the performance impact of this tradeoff and demonstrate that providing this hint through periodic MZT updates is beneficial, even when considering the CPU–GPU communication latency.

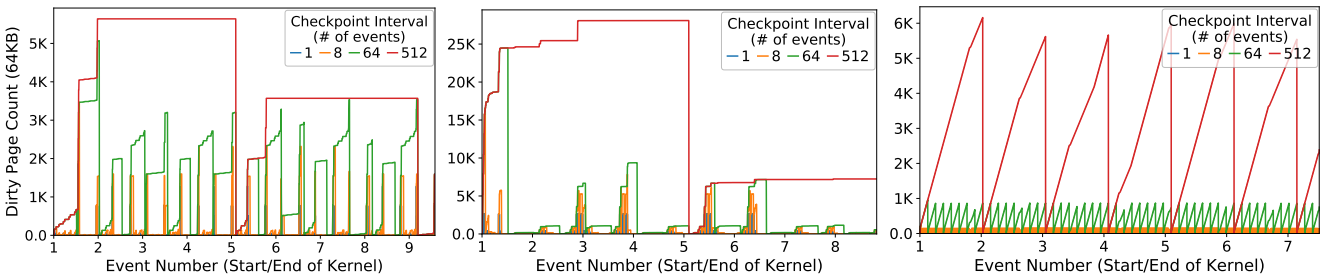
4.1.4 Memory Zone Monitor Hardware Organization

(c1) The MZM hardware checks the address of all writes. (c2) Any store that attempts to modify an address within a zone that is not yet guaranteed to have been transferred is rejected by, or buffered at, the memory controller. This ensures the snapshot is consistent without immediately stalling the GPU pipeline. The MZM is physically distributed across memory channels, with each MZM partition responsible for the physical addresses associated with the channel belonging to the MZM.

Each MZM partition includes a very simple cache for the MZT entries to minimize the extra load added to the memory channel (Figure 5). A single bit per zone identifies zones that have not yet been preserved. A cache entry therefore includes a tag and a bit vector that corresponds to a number of contiguous zones. We observe good locality in the MZT cache with a sectored cache where each 32B sector is a bit-vector that spans 16MB of DRAM (with a 64KB memory zone size). The minimum access granularity of HBM memory is 32B, which matches this sector size. Therefore, only 2048 bit-vectors are required to track 32GB device memory space. The 32 memory channels with 2KB MZT cache each lead to an aggregated 64KB of MZT cache storage with total data and tag sizes of $1.1mm^2$ and $0.056mm^2$, respectively, based on the CACTI 6.5 cache model [27]. Note that the MZT cache must be updated after the MZT is updated, though the conservative nature of the MZT and cached entries means this is, again, an optimization. To further reduce overheads, any write that is issued after the checkpoint is completely stored to the host can proceed without checking the MZT and wasting memory bandwidth. (b5) The MZM uses a global register for this purpose, which is updated when the driver completes transferring the entire snapshot memory state (Figure 5).

4.1.5 Optimizing Stalled-Write Processing

If the memory write access pattern differs significantly from the pattern at which the checkpoint offloader transfers data to the host, write operations are likely to stall for long periods and stall the GPU pipelines. Just like the CoW mechanism, GPU Snapshot mitigates the impact of long write stall times by prioritizing the transfer of zones with pending writes. (b4) This requires the MZM to send information about such zones to the offloader. In the case of the CPU acting as the offloader through the driver, this incurs a transfer over PCIe (or NVLink), which may take $25\mu s$ [49, 59]. It is therefore important to balance the size of the zone to amortize the cost of such transfers w.r.t. the time required to transfer the data of a zone. We describe a hardware mechanism for reducing this latency and further improving checkpointing performance below.



(a) CoMD(xyz49_wa_nl) (b) HPGMG(FV_UVM_opt) (c) SNAP

Figure 6: The number of dirty 64KB zones (64KB) over time with checkpoints every 1, 8, 64, 512 kernel iterations.

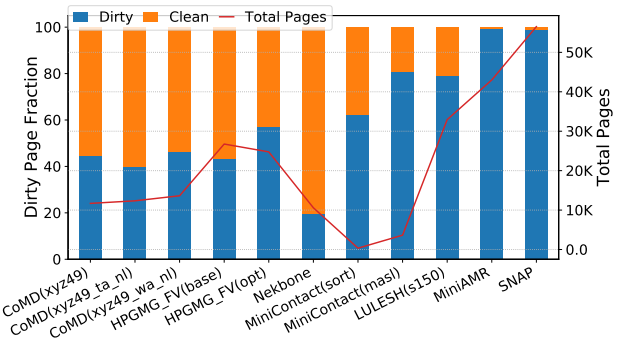


Figure 7: Statistics of 64KB dirty zones for different kernels.

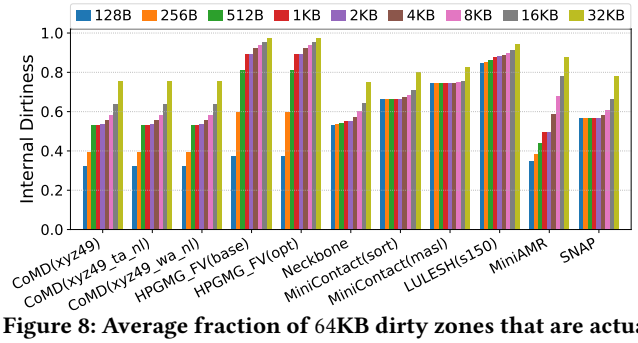


Figure 8: Average fraction of 64KB dirty zones that are actually dirty when tracked at finer granularity.

4.2 Incremental Checkpointing

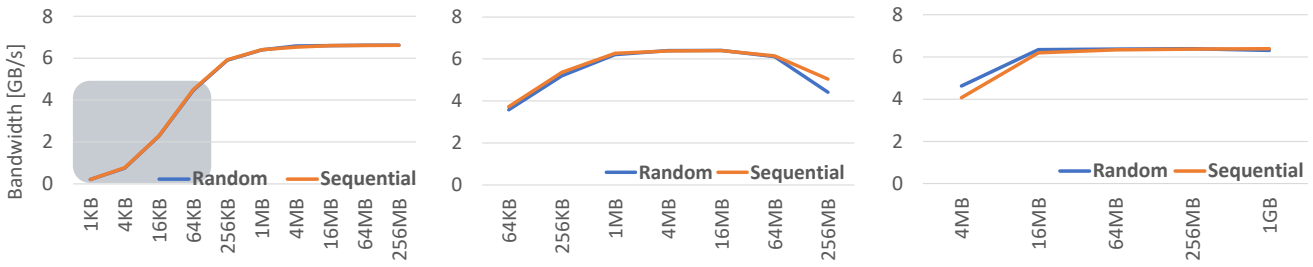
While checkpoint offloading can hide some of the checkpointing time with useful work on the GPU, incremental checkpointing can further decrease overheads by reducing the checkpoint volume. Because a checkpoint interval is long (hundreds or thousands of seconds), however, it is possible that consecutive kernel iterations simply write all of GPU memory, rendering incremental checkpointing ineffective. We quantify the application-specific impact of incremental checkpointing below.

4.2.1 Dirty Zone Count of CUDA Applications GPU Snapshot supports incremental checkpoint with the MZT as a means of tracking dirty memory zones at low cost. We study the number of memory zones (64KB granularity) that are modified by CUDA kernels for different checkpointing intervals across several HPC applications. As shown in Figure 6, every application we study (including some not shown in the figures) reaches some saturation point in terms of the number of dirty zones and does not write to all of GPU memory. While shorter checkpoint intervals will benefit more from incremental checkpointing, long intervals still exhibit substantial potential in most applications.

Figure 7 shows the total number of 64KB zones used by each of the applications we study as well as what fraction of those zones are dirty at the saturation point of each benchmark where incremental checkpointing exhibits the minimal benefit at a long node checkpoint interval. While benchmarks like SNAP and MiniAMR will not benefit from incremental checkpointing (at a dirty-zone tracking granularity of 64KB), many applications will show substantial improvement.

4.2.2 Dirty Zone Granularity Optimization The above analysis was performed with a dirty-zone tracking granularity of 64KB. However, it is possible that more fine-grained tracking will enable additional savings from incremental checkpointing. Figure 8 shows the average fraction of each 64KB zone that is dirty when tracked at finer granularity. For most kernels we studied, the finer the tracking granularity, the better incremental checkpointing would work. Very fine-grained tracking, however, adds substantial overheads to managing the MZT state and managing and coordinating transfers to the host. We later evaluate the performance and impact of incremental checkpointing using "fine-grained" 4KB tracking granularity and "coarse-grained" 64KB granularity.

4.2.3 Augmenting MZM for Incremental Checkpointing GPU Snapshot enables incremental checkpointing with the MZM by identifying dirty memory zones with the MZT. While the MZM uses the MZT to determine whether a write should stall, it can also collect information on which zones are being updated and that will require transfer to the host at the next checkpoint. We do this by extending the MZT and MZT cache to track both the state of the checkpoint that is being transferred to the host and the state being collected for the next checkpoint. The MZT thus contains two bits per zone (Figure 5). One bit corresponds to the previous checkpoint interval and indicates if the zone is guaranteed to have been transferred or did not need to be transferred because it was not modified in the previous interval. The second bit corresponds to the interval being collected and marks whether the zone has been modified by any store instructions. The MZM uses the previous-interval bit to determine whether writes must stall and updates the next-interval bit on any write operation.



(a) Device-to-Host copy at different size (b) Buffer-to-Host copy at different size (c) Buffered copy at different buffer size

Figure 9: (a) Device-to-Host copies show low efficiency with small transfers, while buffered copies achieve high bandwidth. Grey indicates lower-efficiency copy granularities. A single page is not sufficient to fully utilize the GPU-CPU interconnect. (b) Device-to-Buffer copy size=4KB, Buffer size=256MB, (c) Device-to-Buffer copy size=4KB, Buffer-to-Host copy size=2MB.

4.3 GPU Snapshot Hardware Optimizations

The previous subsection described checkpoint offloading that is managed by the GPU driver running on the CPU. The communication latency between the offloader and the accelerator is therefore high, which can potentially limit checkpointing efficiency. We discuss two hardware extensions to GPU Snapshot that integrate some of the offloading functionality into the GPU, though it is still distinct from (and asynchronous to) regular GPU execution.

4.3.1 Optimizing GPU–CPU Transfers High bandwidth to the host is critical during checkpointing and it is important to optimize the data transfers. We micro-benchmark the GPU-CPU copies of GPU Snapshot at different memory zone granularities on a system with NVIDIA Titan Xp and CUDA 8.0. Device-to-host bandwidth is maximized when the transfer granularity is large, at around 1−4MB (Figure 9). Because incremental checkpointing only transfers dirty zones and because those zones can be fragmented in memory, significant bandwidth is squandered—an 8-lane PCIe 3.0 channel, for example, only achieves 62% of maximum throughput when the transfer granularity is 64KB and even lower throughput if memory zones are smaller. Recall that smaller zones increase the benefits of incremental checkpointing.

To reclaim this bandwidth, we add a device-side double-buffer to coalesce multiple dirty zones before transferring them to the host. While one half of the double-buffer is being transferred as one large block, the other half is filled by the checkpoint offloading units. (d1) One such unit is associated with each MZM partition and uses the MZT to feed zones into the transfer buffer. To minimize synchronization and communication, each MZM partition is associated with a partition of the double-buffer. Once a buffer is ready to be transferred, the GPU initiates a DMA over the bus and then notifies the CPU. (d2) Alternatively, a signal is raised to the driver on the host, which then transfers the buffer and updates the GPU on the transfer progress.

4.3.2 In-Device Duplicate-on-Write (DoW) As discussed earlier, performance can be improved if the stall time for a write fault is kept to a minimum. While the MZM can notify the driver to prioritize the transfer of zones on which stores are stalled, the communication delay between the GPU and the host reduces the effectiveness of this idea. We therefore propose that the device itself duplicate zones when they are written—a duplicate-on-write approach that is similar to an OS copy-on-write. (d3) Once a zone is duplicated in GPU memory, the MZT is updated and stores to that zone need no longer stall.

It is impossible to duplicate all of the GPU's device memory, which can happen if stores are frequent and have little spatial locality. We, therefore, use the transfer buffer in GPU memory to also buffer prioritized transfers. The DoW hardware duplicates the now-dirty zone to the transfer buffer and records the zone's location before updating the corresponding MZM. This hardware is simple because before the snapshot process begins, the driver allocates a buffer space for each MZM. Thus, all DoW operations are local and require no synchronization. The CPU now processes data from the buffer, which is mapped to a location known to the driver. Note that the MZM only initiates a DoW on the first write-stall to a zone. When the MZT is queried for a write, if a zone is marked as not-yet-transferred and also not-dirty, a DoW is initiated. If the zone is already dirty and also not-yet-transferred, the write is stalled without initiating a redundant DoW. Note that the in-device preservation buffer absorbs the bursty GPU-CPU copies and leverages the high GPU memory bandwidth.

5 Methodology

We evaluate GPU Snapshot both in terms of how much it reduces the checkpoint time of the accelerators within a single GPU-dense node and its total impact on expected application performance on a large-scale multi-exaflop supercomputer.

5.1 Single-Node Model

We model both the hardware and driver components of GPU Snapshot using an accurate in-house simulator. We simulate the full run of seven different CUDA mini-apps (some with multiple input sets), taking 2–6 billion GPU cycles each. The simulated execution time correlates closely with performance on real hardware—the correlation coefficient is 0.957 for these workloads comparing a simulated P100 GPU to real hardware. The simulation is necessary because we modify and extend GPU hardware for *GPU Snapshot* support.

We evaluate one representative GPU within a larger node by constraining the CPU–GPU bandwidth during checkpointing. We assume that each CPU has 32 PCIe 4.0 lanes and scale down the bandwidth available for any GPU based on the ratio of GPU to CPU sockets in a node. Such bandwidth sharing can be accomplished using PCIe switches—as is done in current GPU-dense systems such as the NVIDIA DGX-2, which shares an aggregated 96 PCIe 3.0 lanes

Table 1: Organization inspired by Coral Summit; failure rates based on Titan and Blue Waters.

System Configurations										
Peak[EFLOP]		0.45	0.90	1.8	3.6					7.2
System	GPU/CPU	16	16	16	2	4	8	16	32	16
	GPUs	57600	115200	230400	460800	460800	460800	460800	460800	921600
	CPUs	3600	7200	14400	230400	115200	57600	28800	14400	57600
	Nodes	1800	3600	7200	115200	57600	28800	14400	7200	28800
	Cabinets	100	200	400	6400	3200	1600	800	400	1600
BW[GB/s]	GPU	2.36E+08	4.72E+08	9.44E+08	1.89E+09	1.89E+09	1.89E+09	1.89E+09	1.89E+09	3.77E+09
	CPU	6.13E+05	1.23E+06	2.45E+06	3.92E+07	1.96E+07	9.81E+06	4.90E+06	2.45E+06	9.81E+06
	PCIe	2.30E+05	4.61E+05	9.22E+05	3.69E+06	3.69E+06	3.69E+06	1.84E+06	9.22E+05	3.69E+06
	Local	3.87E+03	7.74E+03	1.55E+04	2.48E+05	1.24E+05	6.19E+04	3.10E+04	1.55E+04	6.19E+04
	PFS	3.45E+03	4.49E+03	5.84E+03	7.65E+03	7.61E+03	7.60E+03	7.59E+03	7.58E+03	9.86E+03
Failures/s	GPU	6.56E-05	1.31E-04	2.62E-04	5.25E-04	5.25E-04	5.25E-04	5.25E-04	5.25E-04	1.05E-03
	CPU	1.14E-06	2.28E-06	4.56E-06	6.36E-05	3.31E-05	1.73E-05	9.13E-06	4.84E-06	1.83E-05
	Node	2.60E-07	5.20E-07	1.04E-06	1.66E-05	8.31E-06	4.16E-06	2.08E-06	1.04E-06	4.16E-06
	System	1.77E-07	1.77E-07	1.77E-07	1.77E-07	1.77E-07	1.77E-07	1.77E-07	1.77E-07	1.77E-07
	Total	6.72E-05	1.34E-04	2.68E-04	6.05E-04	5.67E-04	5.47E-04	5.36E-04	5.31E-04	1.07E-03

Table 2: Configuration of node and base system organization.

Base Node Configurations	
Perf/CPU [TFLOP]	0.384
Perf/GPU [TFLOP]	7.8
CPU/Node	2
GPU/Node	4
Core/CPU	24
Core/GPU	5120
Memory/CPU [GB]	256
Memory/GPU [GB]	32
DRAM BW/Node [GB/s]	340
PCIe/Node [GB/s]	128
SSD BW/Node [GB/s]	2.15
Total PFS BW [GB/s]	2500
Node/Cabinet	18

between 16 GPUs [37]. For each benchmark, we test a large range of possible GPU Snapshot configurations. Each configuration is denoted by which GPU Snapshot features it uses: coarse-grained incremental (I), fine-grained incremental (I-FG), non-blocking offloading (NB), driver-only checkpoint progress hints (H), full checkpoint (F), and device side double-buffering and DoW optimizations (D).

We simulate the checkpoint at the saturation point of dirty pages of each application, with the assumption that the checkpoint interval is longer than the saturation period. Note that both the advantage of incremental checkpoint and non-blocking checkpoint are amplified with shorter checkpoint interval, which means that the interval used for our evaluation is conservative. Our simulation results use 1 rank per GPU for MPI-based programs. GPU Snapshot can support multiple ranks per GPU can be supported by maintaining a memory zone table per GPU context; while we do not expect multiple MPI ranks per GPU to change the results, we leave the evaluation of this organization for future work.

5.2 System Model

We project the impact of the GPU Snapshot to a three-tiered multi-level checkpoint-restart scheme, modeled at full-machine scale. We project a system that is inspired by Coral Summit [53, 60]. We use the CPU, GPU, and GPU density parameters projected for the IBM Power9 CPU, NVIDIA V100 GPU, and varying GPU densities. The parameters we use are summarized in Table 1 and Table 2. As mentioned above, we limit each CPU to 32 PCIe 4.0 lanes, with GPU density scaling provided by switches, if necessary. For larger system configurations, we scale up the global file system bandwidth by a factor of 1.3 for every doubling of aggregate system performance, based on rough trends from planned large-scale system upgrades (Edison-to-Cori at NERSC [32, 33], Titan-to-Summit at OLCF [53], and MIRA-to-Aurora at ALCF [24]).

5.3 Failure Model

The system performance model accounts for both the multi-level checkpoint times and overheads and the recovery time associated with failures. We therefore estimate and project failure rates for the GPU-dense nodes and other system components. We use published information on large-scale HPC systems to derive scaling and baseline reliability parameters for our model. Specifically, we use statistics gathered from a 261-day period on the Blue Waters supercomputer [11]. We project the same per-component failure rates for our Summit-inspired systems as measured on Blue Waters. We also combine per-chip memory failure rates based on field measurement study in Titan [50]. The system-level failure rate in our study is lower than in prior work. For example, the total failure rate of our 0.9 EFLOP machine is 1.34E-04 (2-hour mean time between interrupts), while that of the 1 EFLOP machine in previous work [1] is 5.56E-04—4.15 times higher. The main reason for our lower failure rates is the more realistic accelerator-dense machine organization we consider: Summit-like nodes with tens or hundreds of teraflops per node allow exascale systems with fewer than $\frac{1}{27}$ the number of nodes used in prior work [1, 8].

6 Evaluation

We quantify the per-GPU checkpoint speedup of various *GPU Snapshot* organizations, then project the impact of *GPU Snapshot* on system efficiency based on these checkpoint speedups.

6.1 Single Device

6.1.1 Basic GPU Snapshot Figure 10 shows the checkpointing overhead relative to our baseline, which uses a full blocking checkpoint for GPU memory. To separate out the performance effects of our different optimizations, four organizations are shown: incremental ("I"), incremental and non-blocking ("NB+I"), full non-blocking with checkpoint progress hints ("NB+F+H"), and incremental non-blocking with checkpoint progress hints ("NB+I+H"). As shown before in Figure 7, the fraction of dirty pages in GPU memory is application dependent. Nekbone shows great checkpoint time reduction with incremental checkpointing, but SNAP and MiniAMR cannot take advantage of incremental checkpointing because SNAP and MiniAMR have few clean memory zones. A comparison of non-blocking and incremental checkpointing with and without checkpoint progress hints, NB+I+H and NB+I, respectively, highlights the benefits of coordinating the offloader and the accelerator: mini-apps such as SNAP, Nekbone, and MiniContact, for example, benefit substantially from these hints on checkpoint progress. Full non-blocking checkpoint with checkpoint progress hints (NB+F+H) helps performance substantially in all applications. Without checkpoint-progress hints, full non-blocking checkpoint alone does not always help performance and it is not shown. Incremental and non-blocking checkpoint are synergistic, and applying both (NB+I+H) performs best in all applications.

6.1.2 Optimized GPU Snapshot Figure 11 shows further performance benefits using the device-side GPU Snapshot optimizations. Four organizations are shown: incremental non-blocking

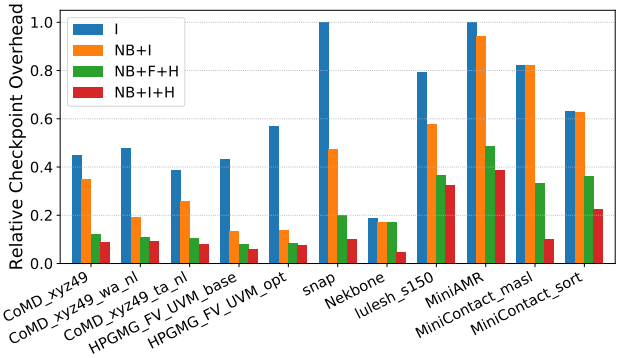


Figure 10: Relative checkpoint time of basic GPU Snapshot (no device-side optimizations) for a full checkpoint.

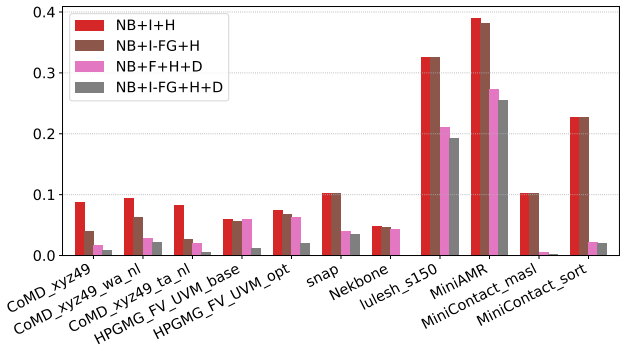


Figure 11: GPU Snapshot configurations with device-side optimizations (double-buffering and duplicate-on-write).

with checkpoint progress hints ("NB+I+H", the most aggressive organization from basic GPU Snapshot), a fine-grained variant of that organization ("NB+I-FG+H"), full non-blocking with checkpoint progress hints and device-side double-buffering and DoW optimizations ("NB+F+H+D"), and an incremental variant of that organization ("NB+I-FG+H+D"). With device-side optimizations, most applications exhibit over 20× reduction in checkpoint time. Even the worst application, MiniAMR, is improved by more than 3×. (Note that the range of the plot only goes up to 0.4 for visibility.) The optimizations utilize the device memory bandwidth for duplicate-on-write and double buffering to maximize the transfer bandwidth to the host. These two techniques not only decrease the latency of handling write faults, but also reduce the likelihood of write faults because the MZT is updated once each of the dirty zones is duplicated to the buffer. Good examples of this are MiniContact and Nekbone. MiniContact has a small working set that is cached well. Nekbone has few writes and it also caches well.

CoMD and HPGMG show great reduction of checkpoint volume with fine-grained dirty tracking (I-FG). While fine-grained tracking does not substantially decrease the checkpoint volume of most other applications, there could still be system-wide benefits to reducing the GPU checkpoint size. Smaller GPU checkpoints mean smaller node checkpoints, and smaller node checkpoints improve overall system performance measurably in many cases, as we explain in Section 6.2.

Note that the overhead of NB results from non-overlapped stalled-write processing. If the write faults are not balanced over MZM partitions (memory channels), buffering resources will be exhausted and the pipeline stalled. However, this is not a major concern because physically-contiguous regions are typically balanced across channels by a hardware address swizzle to maximize the effective memory bandwidth. *GPU Snapshot* benefits from this same address swizzle because it spreads write faults between channels, avoiding write fault camping and the associated pipeline stalls.

6.2 System-Level Impact

We project the relative checkpoint overhead measured at the GPU level to the system level. The baseline is again full checkpointing with three-tier multi-level checkpointing. Each dot in Figure 12 represents a different combination of GPU Snapshot features for each mini-app studied. The top facet takes into account the memory footprint of each application and the bottom facet shows efficiency if each program is scaled to full GPU memory utilization. The figure also shows expected system performance with the baseline checkpointing scheme and a theoretical limit that checkpoints the GPUs to host-side memory and local disk with zero overhead.

The theoretical line with infinite intra-node preservation bandwidth is not at 100% performance efficiency because system-level failures must still be tolerated. Likewise, zero GPU–CPU transfer time (black line) exhibits 92.7% in performance efficiency because of recovering node and network failures with the checkpoint written to node-local disk. Note that GPUs within a node share the same node-local disk. The small footprint of applications such as MiniContact results in efficient resilience with three-level checkpointing system even without GPU Snapshot. However, when programs use a large portion of GPU memory, or multiple MPI ranks fill up all

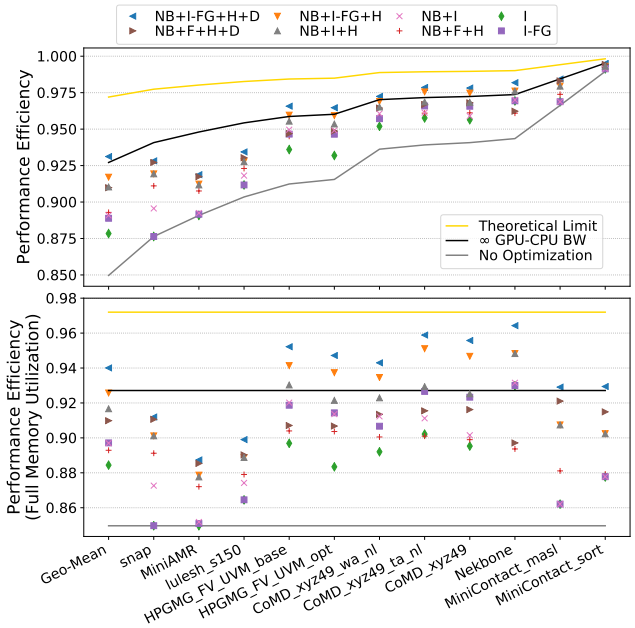


Figure 12: The system-level impact of the GPU Snapshot variants at 3.6 exaflops and 16 GPUs/CPU. The top facet shows the efficiency when memory footprints are taken into account, and the bottom facet is scaled to estimate efficiency with full memory utilization.

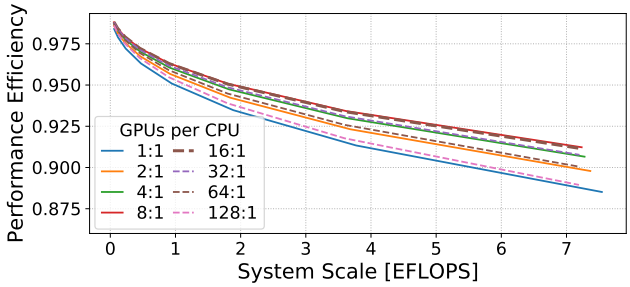


Figure 13: The scalability of different GPU densities with GPU Snapshot.

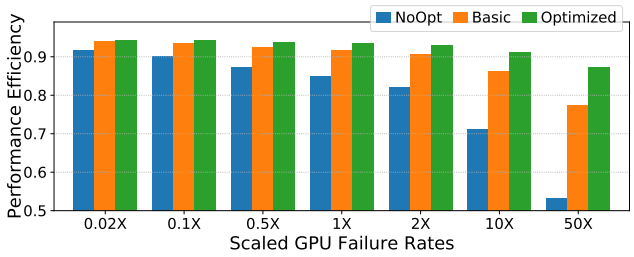


Figure 14: GPU failure rate scaling in a three-level checkpointing system with 3.6 exaflops and 16 GPUs/CPU.

of GPU memory, GPU Snapshot greatly improves performance. The device-optimized GPU Snapshot variants, in particular, come within 3% of the ideal performance with zero GPU—CPU transfer time on average considering full GPU memory occupancy.

The points above the black line with infinite GPU–CPU preservation bandwidth in Figure 12 also demonstrate the effect of incremental checkpointing on overall system efficiency. GPU Snapshot preserves only modified state from GPUs, where most computation happens, to storage media. As a result, the reduced checkpoint volume of *GPU Snapshot* also improves disk-level checkpoint time by reducing the total application state. This indicates that combining incremental and non-blocking checkpoint with *GPU Snapshot* leads to a synergistic effect that further improves performance efficiency than optimizing memory-level checkpoint only (black line) at large system scales.

6.2.1 GPU Density Scaling with GPU Snapshot As shown in Figure 2, dense GPU nodes suffer from node-local resource congestion for fast memory-level checkpoint, preventing high performance efficiency at system scale. GPU Snapshot significantly reduces checkpoint overhead and enables efficient scaling. Figure 13 projects the effectiveness of *optimized GPU Snapshot* in GPU-dense exaflop systems. GPU Snapshot alleviates the resource congestion for memory-level checkpoint, and allows GPU density greater than 16 GPUs/CPU to be scaled at multi-exaflop system. This is a drastic change from multi-level checkpointing and current intra-node preservation mechanisms (presented earlier in Figure 2), where scaling beyond 16 GPUs/CPU becomes increasingly expensive.

6.2.2 Sensitivity to GPU Failure Rate Accelerated memory-level checkpointing by GPU Snapshot shows robust efficiency even at high GPU failure rates. Figure 14 shows the sensitivity of performance efficiency of three-level checkpointing system at 3.6 exaflops with 16 GPUs/CPU. Without GPU Snapshot, the efficiency of three-level checkpointing is sensitive to GPU failure rates, decreasing to less than 54% at 50X rates. However, *optimized* GPU Snapshot sustains 90% in performance efficiency at highly-scaled GPU failure rates. On the other hand, 0.1X GPU failure rates allows multi-level checkpoint without GPU Snapshot to achieve 90% efficiency for the 3.6 exaflop system. This indicates that GPU-dense systems must either be equipped with highly reliable GPU devices or an efficient resilience mechanism at the node level to pave the road for efficient exascale computation.

7 Related Work

7.1 Multi-Level Checkpoint-Restart

Multi-level checkpoint and restart [31] mitigates frequent bursty IO accesses by efficiently utilizing the bandwidth at each level of the storage hierarchy. Checkpointing to node-local storage such as local disk or memory is fast and sufficient to tolerate node failures, but is not capable of handling rare-yet-severe global failures. Multi-level checkpoint-restart improves overall system-level efficiency by optimizing the checkpoint interval for each level of storage based on the bandwidth and failure rate associated with that level. We show in Section 3 that multi-level checkpoints, alone, are insufficient for providing high efficiency for large-scale GPU-dense systems. *GPU Snapshot* is an orthogonal optimization technique for GPU-dense systems to accelerate global checkpointing.

7.2 Overlapping Global Disk Access

There are prior efforts to reduce the cost of bursty IO in large systems by better managing disk bandwidth—for instance, the burst buffer is an example of an IO-attached appliance that is able to overlap global IO writes with continuing execution [28]. Alternatively, instead of a burst buffer, Agrawal et al. propose a node-local non-volatile memory with a hardware controller that is capable of trickling memory-level checkpoints from local nodes to global disk [3]. *GPU Snapshot* optimizes top-level intra-node preservation, which we show in Figure 3 to be the major efficiency bottleneck in GPU-dense systems. Burst buffers operate as a transparent IO accelerator for node checkpointers. GPU Snapshot stores serialized GPU state and metadata in host memory. This serialized GPU state can thus be integrated with the node checkpoint and transferred to the burst buffer.

7.3 Hardware-Based Checkpoint-Restart

SafetyNet [47], ReVive [40], and Clank [19] provide fast CPU checkpointing using hardware extensions. They log main memory operations and perform checkpoint-on-write to maintain a snapshot of main memory. SafetyNet copies each evicted cache line to another cache-level storage with limited capacity. Therefore, it cannot checkpoint applications at long intervals, but preservation overheads are low due to the speed of the checkpoint storage. ReVive supports a longer interval than SafetyNet by checkpointing to main memory. It extends a directory-based cache coherence protocol to support hardware checkpoint in a cache-coherent multi-processor. Clank logs memory operations, maintains a checkpoint in non-volatile memory, and recovers from power loss in an energy-harvesting system with frequent checkpoints. *GPU Snapshot* is similar to these prior efforts in that it provides hardware-accelerated checkpoint

and restart. The goal of *GPU Snapshot* differs from these works, however, in that it attempts to provide checkpoint-restart for resilience against detectable-uncorrectable errors (where the optimal checkpoint intervals are much longer), it targets inter-kernel checkpoint and restart, and it provides hardware-software collaborative acceleration through checkpoint offloading.

7.4 Live Migration of Virtual Machines

In virtual machine live migration, a memory snapshot of a virtual machine is copied to a different host, and that virtual machine continues to execute during the copy [9]. A similar idea has also been studied for MPI process migration [54]. Like live migration, *GPU Snapshot* allows the GPU to continue executing while a previous snapshot is transferred, but to the host CPU memory rather than to a remote node. The post-copy variant of live migration [20] is most similar to *GPU Snapshot*. With post copy, the snapshot is first recorded and pages are then transferred to the destination node. Specifically, *active pushing* and *prepaging* copy dirty move pages proactively, avoiding page faults. *GPU Snapshot* is similar in concept, but the implementation is significantly different because *GPU Snapshot* targets compute accelerators and therefore it cannot rely on OS virtual-memory exceptions.

7.5 Uniform Virtual Memory (UVM)

Unified Virtual Memory (UVM) uses a page-fault-based data migration system on x86 machines, which is similar in concept to our write-fault-based non-blocking checkpoint [17]. Relying on the UVM mechanisms for *GPU Snapshot* is not practical, as it would incur significant write-fault overheads and prevent *GPU Snapshot* from overlapping GPU preservation with computation. One fundamental reason for this is the expensive ~25μs round-trip latency needed to synchronize UVM-related TLB state [49, 59]. Furthermore, the GPU TLB coverage is orders of magnitude lower than that of the *GPU Snapshot* MZT cache. The *GPU Snapshot* MZT tracks physical memory within each memory controller, allowing highly-cacheable accesses and updates to MZT state without indirection or coherence mechanisms. Finally, *GPU Snapshot* has fine-grained and buffered back-pressure that is inherently tolerated by the GPU pipeline (similar to in-flight memory accesses), whereas UVM faults stall SMs, or at least CTAs, until they are fully resolved [59].

8 Discussion

8.1 Wider Applicability

Rapid memory snapshots are applicable to many domains, and the usefulness of GPU Snapshot is not limited to high-reliability systems. First, profiling tools such as *nvprof* use a form of checkpoint and replay to profile more performance counters than the hardware supports concurrently [38]. The ability to preserve state and restart execution is useful for advanced debugging techniques [21], or exception support in GPU applications [25, 30]. GPU Snapshot could also be used to accelerate control speculation for dynamic parallelization [12] or speculative loop parallelization [57] for GPUs. These techniques speculatively execute code that is predicted to be highly likely to execute, but requires rollback in the case of mis-speculation. Finally, the software-hardware mechanism for non-blocking checkpointing can be applied to accelerate the live migration of virtual machines (VMs). Live migration of virtualized GPU resources may be essential for load balancing, maintenance, and energy efficiency in accelerator-dense data centers.

8.2 Uncoordinated GPU Checkpoint-Restart

GPU Snapshot offloads and accelerates GPU state preservation to reduce the performance penalty of bursty checkpointing traffic within an accelerator-dense node. An alternative approach for avoiding bursts of preservation has traditionally been uncoordinated [13] or staggered [51] checkpoints. However, uncoordinated checkpointing between GPUs is problematic due to remote GPU–GPU and GPU–CPU memory accesses that require new logging mechanisms. Current CUDA programming systems and GPU devices support a shared and coherent address space across the entire heterogeneous node through UVM [17, 36]. Therefore, uncoordinated checkpoint-restart between GPUs must somehow track remote memory accesses to or from each GPU in order to locally recover from errors, introducing system design complexity and logging overheads [6]. To avoid these additional complexities, we model and optimize a globally coordinated checkpoint. Future work may also explore the combination of GPU Snapshot with an uncoordinated checkpoint between nodes while relying on a coordinated checkpoint within each node.

9 Conclusions

GPU Snapshot is a hardware-software collaborative mechanism to accelerate GPU state preservation. We identify the intra-node bandwidth to be the efficiency bottleneck for multi-level checkpoint and restart in GPU-dense systems, and introduce GPU Snapshot based on the *checkpoint offload* model to amplify the GPU state preservation bandwidth. Keeping most of the complexity in the driver, a simple GPU Snapshot organization is able to reduce the time required to take a memory-level checkpoint by 3–20X. Adding in device-side optimizations increases the minimum speedup to 4X and nearly eliminates the non-overlapped latency of a memory-level checkpoint in many programs.

We project the impact of GPU Snapshot to system-level performance efficiency at exascale. Based on an analytical model of multi-level checkpoint-restart, the average system-level efficiency is improved from 84.9% to 90.9% with the driver-intensive GPU Snapshot organization. Device-optimized GPU Snapshot further improves the average efficiency to 94.0%, nearing the theoretical maximum efficiency of 97.2%.

Acknowledgement

We acknowledge support from the Advanced Simulation and Computing program of the U.S. Department of Energy via the PSAAP II Center under award numbers DE-NA0002373 and an internship at NVIDIA Research.

References

[1] J. Tuck A. Agrawal, G. Loh. 2017. Leveraging Near Data Processing for High-Performance Checkpoint/Restart. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*.

[2] Saurabh Agarwal, Rahul Garg, Meeta S Gupta, and Jose E Moreira. 2004. Adaptive incremental checkpointing for massively parallel systems. In *Proceedings of the International Supercomputing Conference (ISC)*. ACM.

[3] Abhinav Agrawal, Gabriel H. Loh, and James Tuck. 2017. Leveraging Near Data Processing for High Performance Checkpoint/Restart. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*. IEEE.

[4] Samer Al-Kiswany, Matei Ripeanu, Sudharshan S Vazhkudai, and Abdullah Gharaibeh. 2008. stdchk: A checkpoint storage system for desktop grid computing. In *Proceedings of the International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 613–624.

[5] Lorenzo Alvisi and Keith Marzullo. 1998. Message logging: Pessimistic, optimistic, causal, and optimal. *IEEE Transactions on Software Engineering* 24, 2 (1998), 149–159.

[6] Maciej Besta and Torsten Hoefler. 2014. Fault tolerance for remote memory access programming models. In *Proceedings of the International Symposium on High-Performance Parallel and Distributed Computing (HPDC)*. ACM, 37–48.

[7] Wahid Bhimji, Debbie Bard, Melissa Romanus, David Paul, Andrey Ovsyannikov, Brian Friesen, Matt Bryson, Joaquin Correa, Glenn K Lockwood, Vakho Tsulaia, et al. 2016. Accelerating science with the NERSC burst buffer early user program. *CUG2016 Proceedings* (2016).

[8] Yong Chen. 2011. Towards Scalable I/O Architecture for Exascale Systems. In *Proceedings of the Workshop on Many Task Computing on Grids and Supercomputers (MTAGS)*. 43–48.

[9] Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. 2005. Live migration of virtual machines. In *Proceedings of the 2nd Conference on Symposium on Networked Systems Design & Implementation-Volume 2*. USENIX Association, 273–286.

[10] C. Coti, T. Herault, P. Lemarinier, L. Pilard, A. Rezmerita, E. Rodriguezb, and F. Cappello. 2006. Blocking vs. Non-Blocking Coordinated Checkpointing for Large-Scale Fault Tolerant MPI. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*. 18–18. https://doi.org/10.1109/SC.2006.15

[11] Catello Di Martino, Zbigniew Kalbarczyk, Ravishankar K Iyer, Fabio Baccanico, Joseph Fullop, and William Kramer. 2014. Lessons learned from the analysis of system failures at Petascale: The case of Blue Waters. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*. IEEE.

[12] Gregory Diamos and Sudhakar Yalamanchili. 2010. Speculative execution on multi-GPU systems. In *Proceedings of the International Symposium on Parallel and Distributed Processing (IPDPS)*. IEEE, 1–12.

[13] E. N. (Mootaz) Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David B. Johnson. 2002. A Survey of Rollback-recovery Protocols in Message-passing Systems. *Comput. Surveys* 34, 3 (Sept. 2002), 375–408. https://doi.org/10.1145/568522.568525

[14] Fernanda Foertter. 2017. Preparing GPU-Accelerated Applications for the Summit Supercomputer. http://on-demand.gputechconf.com/gtc/2017/presentation/s7642-fernanda-foertter-preparing-gpu-accelerated-app.pdf. http://on-demand.gputechconf.com/gtc/2017/presentation/s7642-fernanda-foertter-preparing-gpu-accelerated-app.pdf GPU Technology Conference (GTC).

[15] Rohan Garg, Apoorve Mohan, Michael Sullivan, and Gene Cooperman. 2018. CRUM: Checkpoint-Restart Support for CUDA's Unified Memory. In *Proceedings of International Conference on Cluster Computing (CLUSTER)*.

[16] Roberto Gioiosa, Jose Carlos Sancho, Song Jiang, Fabrizio Petrini, and Kei Davis. 2005. Transparent, incremental checkpointing at kernel level: A foundation for fault tolerance for parallel computers. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*. IEEE Computer Society.

[17] Mark Harris. 2016. Unified Memory for CUDA Beginners. https://devblogs.nvidia.com/unified-memory-cuda-beginners/. [Online; accessed 18-Jan-2018].

[18] Junyoung Heo, Sangho Yi, Yookun Cho, Jiman Hong, and Sung Y Shin. 2005. Space-efficient page-level incremental checkpointing. In *Proceedings of the Symposium on Applied Computing (SAC)*. ACM, 1558–1562.

[19] Matthew Hicks. 2017. Clank: Architectural Support for Intermittent Computation. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*. 228–240.

[20] Michael R Hines, Umesh Deshpande, and Kartik Gopalan. 2009. Post-copy live migration of virtual machines. *ACM SIGOPS operating systems review* 43, 3 (2009), 14–26.

[21] Joshua Hursey, Chris January, Mark O'Connor, Paul Hargrove, David Lecomber, Jeffrey M Squyres, and Andrew Lumsdaine. 2010. Checkpoint/Restart-Enabled Parallel Debugging. In *Proceedings of the 24th European MPI Users' Group Meeting (EuroMPI)*. 219–228.

[22] Dewan Ibtesham, Dorian C Arnold, Kurt B Ferreira, and Patrick G Bridges. 2011. On the Viability of Checkpoint Compression for Extreme Scale Fault Tolerance.. In *Euro-Par Workshops (2)*. 302–311.

[23] Dewan Ibtesham, Kurt B Ferreira, and Dorian Arnold. 2015. A checkpoint compression study for high-performance computing systems. *The International Journal of High Performance Computing Applications* 29, 4 (2015), 387–402.

[24] Intel. 2015. Ushering in a New Era. https://www.intel.com/content/dam/www/public/us/en/documents/presentation/intel-argonne-aurora-announcement-presentation.pdf. https://www.intel.com/content/dam/www/public/us/en/documents/presentation/intel-argonne-aurora-announcement-presentation.pdf

[25] Hyesoon Kim. 2012. Supporting Virtual Memory in GPGPU Without Supporting Precise Exceptions. In *Proceedings of the ACM SIGPLAN Workshop on Memory Systems Performance and Correctness*. 70–71.

[26] Sangman Kim, Seonggu Huh, Xinya Zhang, Yige Hu, Amir Wated, Emmett Witchel, and Mark Silberstein. 2014. GPUnet: Networking Abstractions for GPU Programs. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

[27] Sheng Li, Ke Chen, Jung Ho Ahn, Jay B. Brockman, and Norman P. Jouppi. 2011. CACTI-P: Architecture-level Modeling for SRAM-based Structures with Advanced Leakage Reduction Techniques. In *Proceedings of Computer-Aided Design (ICCAD)*. Piscataway, NJ, USA.

[28] Ning Liu, Jason Cope, Philip Carns, Christopher Carothers, Robert Ross, Gary Grider, Adam Crume, and Carlos Maltzahn. 2012. On the role of burst buffers in leadership-class storage systems. In *Mass Storage Systems and Technologies (MSST), 2012 IEEE 28th Symposium on*. IEEE, 1–11.

[29] John Mehnert-Spahn, Eugen Feller, and Michael Schoettner. 2009. Incremental checkpointing for grids. In *Linux Symposium*, Vol. 120.

[30] Jaikrishnan Menon, Marc De Kruijf, and Karthikeyan Sankaralingam. 2012. iGPU: Exception Support and Speculative Execution on GPUs. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*. 72–83.

[31] Adam Moody, Greg Bronevetsky, Kathryn Mohror, and Bronis R de Supinski. 2010. Design, modeling, and evaluation of a scalable multi-level checkpointing system. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*. IEEE Computer Society, 1–11.

[32] NERSC. 2013. Edison Storage and IO. http://www.nersc.gov/users/computational-systems/edison/file-storage-and-i-o/.

[33] NERSC. 2017. Cori Storage and IO. http://www.nersc.gov/users/computational-systems/cori/file-storage-and-i-o/.

[34] Xiang Ni, Esteban Meneses, and Laxmikant V Kalé. 2012. Hiding checkpoint overhead in HPC applications with a semi-blocking algorithm. In *Proceedings of International Conference on Cluster Computing (CLUSTER)*. IEEE, 364–372.

[35] Akira Nukada, Hiroyuki Takizawa, and Satoshi Matsuoka. 2011. NVCR: A transparent checkpoint-restart library for NVIDIA CUDA. In *Proceedings of the International Symposium on Parallel and Distributed Processing (IPDPS) Workshops*. IEEE, 104–113.

[36] NVIDIA. 2017. CUDA C Programming Guide, Appendix K: Unified Memory Programming. http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf. , 267–286 pages. PG-02829-001_v9.1 [Online; accessed 17-Jan-2018].

[37] NVIDIA. 2018. NVIDIA DGX-2: The world's most powerful AI system for the most complex AI challenges. https://www.nvidia.com/en-us/data-center/dgx-2/.

[38] NVIDIA. 2019. The NVIDIA profiling tool (nvprof). http://docs.nvidia.com/cuda/profiler-users-guide/.

[39] James S Plank, Micah Beck, Gerry Kingsley, and Kai Li. 1994. *Libckpt: Transparent checkpointing under unix*. Computer Science Department.

[40] Milos Prvulovic, Zheng Zhang, and Josep Torrellas. 2002. ReVive: cost-effective architectural support for rollback recovery in shared-memory multiprocessors. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, Vol. 30. IEEE Computer Society, 111–122.

[41] Roy Kim. 2016. NVIDIA DGX SATURNV Ranked World's Most Efficient Supercomputer by Wide Margin. https://blogs.nvidia.com/blog/2016/11/14/dgx-saturnv/. https://blogs.nvidia.com/blog/2016/11/14/dgx-saturnv/ NVIDIA Blog.

[42] Samsung. 2018. Samsung PM1725a NVMe SSD. https://www.samsung.com/semiconductor/global.semi.static/Samsung_PM1725a_NVMe_SSD-0.pdf.

[43] Naoto Sasaki, Kento Sato, Toshio Endo, and Satoshi Matsuoka. 2015. Exploration of lossy compression for application-level checkpoint/restart. In *Proceedings of the International Symposium on Parallel and Distributed Processing (IPDPS)*. IEEE, 914–922.

[44] K. Sato, N. Maruyama, K. Mohror, A. Moody, T. Gamblin, B. R. de Supinski, and S. Matsuoka. 2012. Design and modeling of a non-blocking checkpointing system. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*.

[45] Akira Nukada Shinichi Miura Akihiro Nomura Hitoshi Sato Hideyuki Jitsumoto Aleksandr Drozd Satoshi Matsuoka, Toshio Endo. 2017. Overview of TSUBAME3.0, Green Cloud Supercomputer for Convergence of HPC, AI and Big-Data. https://www.titech.ac.jp/news/pdf/news_17675_2.pdf.

[46] Mark Silberstein, Bryan Ford, Idit Keidar, and Emmett Witchel. 2013. GPUfs: Integrating a file system with GPUs. In *ACM SIGPLAN Notices*, Vol. 48. ACM, 485–498.

[47] Daniel J Sorin, Milo MK Martin, Mark D Hill, and David A Wood. 2002. SafetyNet: improving the availability of shared memory multiprocessors with global checkpoint/recovery. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*. IEEE, 123–134.

[48] Hiroyuki Takizawa, Katsuto Sato, Kazuhiko Komatsu, and Hiroaki Kobayashi. 2009. CheCUDA: A checkpoint/restart tool for CUDA applications. In *Proceedings of the International Conference on Parallel and Distributed Computing, Applications and Technologies*. IEEE, 408–413.

[49] Ivan Tanasic, Isaac Gelado, Marc Jorda, Eduard Ayguade, and Nacho Navarro. 2017. Efficient exception handling support for GPUs. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*. ACM, 109–122.

[50] Devesh Tiwari, Saurabh Gupta, et al. 2015. Understanding GPU errors on large-scale HPC systems and the implications for system design and operation. In *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 331–342.

[51] Nitin H Vaidya. 1996. On staggered checkpointing. In *Symposium on Proceedings of Parallel and Distributed Processing (SPDP)*. IEEE, 572–580.

[52] Manav Vasavada, Frank Mueller, Paul H Hargrove, and Eric Roman. 2011. Comparing different approaches for incremental checkpointing: The showdown. In *Linux Symposium*. 69.

[53] Sudharshan S Vazhkudai, Bronis R de Supinski, Arthur S Bland, Al Geist, James Sexton, Jim Kahle, Christopher J Zimmer, Scott Atchley, Sarp Oral, Don E Maxwell, et al. 2018. The design, deployment, and evaluation of the CORAL pre-exascale systems. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*. 52.

[54] Chao Wang, Frank Mueller, Christian Engelmann, and Stephen L Scott. 2008. Proactive process-level live migration in HPC environments. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*. IEEE Press, 43.

[55] Chao Wang, Frank Mueller, Christian Engelmann, and Stephen L Scott. 2011. Hybrid full/incremental checkpoint/restart for MPI jobs in HPC environments. In *Proceedings of the International Symposium on Parallel and Distributed Processing (IPDPS)*.

[56] Sangho Yi, Junyoung Heo, Yookun Cho, and Jiman Hong. 2006. Adaptive page-level incremental checkpointing based on expected recovery time. In *Proceedings of the Symposium on Applied Computing (SAC)*. ACM, 1472–1476.

[57] Chenggang Zhang, Guodong Han, and Cho-Li Wang. 2013. GPU-TLS: An efficient runtime for speculative loop parallelization on gpus. In *IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*. IEEE, 120–127.

[58] Gengbin Zheng, Xiang Ni, and Laxmikant V Kalé. 2012. A scalable double in-memory checkpoint and restart scheme towards Exascale. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*. IEEE, 1–6.

[59] T. Zheng, D. Nellans, A. Zulfiqar, M. Stephenson, and S. W. Keckler. 2016. Towards high performance paged memory for GPUs. In *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)*. 345–357.

[60] Chris Zimmer. 2018. Summit Burst Buffer. https://www.olcf.ornl.gov/wp-content/uploads/2018/05/Intro_Summit_Burst-Buffer-Webinar.pdf.