Scalable Incremental Checkpointing using GPU-Accelerated De-Duplication

Nigel Tan
University of Tennessee Knoxville

Jakob Luettgau
University of Tennessee Knoxville

Jack Marquez
University of Tennessee Knoxville

Keita Terianishi
Oak Ridge National Laboratory

Nicolas Morales
Sandia National Laboratories

Sanjukta Bhowmick
University of North Texas

Franck Cappello
Argonne National Laboratory

Michela Taufer
University of Tennessee Knoxville

Bogdan Nicolae
Argonne National Laboratory

ABSTRACT

Writing large amounts of data concurrently to stable storage is a typical I/O pattern of many HPC workflows. This pattern introduces high I/O overheads and results in increased storage space utilization especially for workflows that need to capture the evolution of data structures with high frequency as checkpoints. In this context, many applications, such as graph pattern matching, perform sparse updates to large data structures between checkpoints. For these applications, incremental checkpointing techniques that save only the differences from one checkpoint to another can dramatically reduce the checkpoint sizes, I/O bottlenecks, and storage space utilization. However, such techniques are not without challenges: it is non-trivial to transparently determine what data has changed since a previous checkpoint and assemble the differences in a compact fashion that does not result in excessive metadata. State-of-art data reduction techniques (e.g., compression and de-duplication) have significant limitations when applied to modern HPC applications that leverage GPUs: slow at detecting the differences, generate a large amount of metadata to keep track of the differences, and ignore crucial spatiotemporal checkpoint data redundancy. This paper addresses these challenges by proposing a Merkle tree-based incremental checkpointing method to exploit GPUs' high memory bandwidth and massive parallelism. Experimental results at scale show a significant reduction of the I/O overhead and space utilization of checkpointing compared with state-of-the-art incremental checkpointing and compression techniques.

CCS CONCEPTS

• **Software and its engineering** → **Checkpoint / restart**; • **Theory of computation** → **Massively parallel algorithms**.

KEYWORDS

Checkpointing, data versioning, incremental storage, de-duplication, GPU parallelization

ICPP 2023, August 07–10, 2023, Salt Lake City, UT, USA

ACM ISBN 979-8-4007-0843-5/23/08.
https://doi.org/10.1145/3605573.3605639

ACM Reference Format:
Nigel Tan, Jakob Luettgau, Jack Marquez, Keita Terianishi, Nicolas Morales, Sanjukta Bhowmick, Franck Cappello, Michela Taufer, and Bogdan Nicolae. 2023. Scalable Incremental Checkpointing using GPU-Accelerated De-Duplication. In *52nd International Conference on Parallel Processing (ICPP 2023), August 07–10, 2023, Salt Lake City, UT, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3605573.3605639

1 INTRODUCTION

One fundamentally enabling I/O pattern of HPC workflows is *checkpointing*. It involves many processes, distributed in groups over a large number of compute nodes (e.g., one process per GPU), that need to simultaneously capture important data structures at critical moments during runtime and save persistent checkpoints of these data structures durably to revisit them later. Traditionally, checkpointing has been the leading enabler of resilience for HPC workflows: applications take checkpoints periodically during runtime and restart from the latest checkpoint in case of failures to minimize the number of lost computations (at the expense of checkpointing overheads). Over time, checkpointing has found broad applicability in other scenarios: batch job preemption [7] (e.g., to make room for higher priority on-demand jobs without losing computational progress), job migration [27], adjoint computations and automated differentiation methods that generate intermediate states during a forward pass and revisit them in a backward pass [35], exploring alternative computational paths (e.g., sensitivity analysis of AI models using variations of training data starting from a common initial training) [28], and the study of reproducibility by capturing and comparing intermediate results during different runs. Under such circumstances, checkpointing is more challenging for two reasons: (1) there is a need to store the entire checkpoint record into a lineage [18] (not just the latest checkpoint); and (2) the checkpointing frequency is significantly higher than in the case of resilience (e.g., checkpoint intervals of 10ms are common in adjoint computations [13] and reproducibility, as opposed to resilience, where checkpoint intervals are correlated with the mean time between failures and are the order of hours).

The need to reduce I/O overheads and space utilization of checkpointing: With the ever-increasing computational and data processing capabilities of HPC workflows, the push towards Exascale has resulted in HPC systems made of thousands of compute nodes, each equipped with many-core CPUs and several GPUs. Such systems are complemented by a heterogeneous storage stack that includes deep local memory hierarchies (e.g., high bandwidth

memory, volatile host memory, persistent memory, NVMe-enabled flash storage) and external data repositories (e.g., parallel file systems). Traditionally, checkpointing has been performed by direct writes to the external repository, which blocks the application for the duration of checkpointing. In this case, at a large scale, many processes distributed on the compute nodes (typically one process per GPU) compete for a limited I/O bandwidth. This introduces large I/O overheads and therefore increases the end-to-end runtime. Even in the case of resilience, where the checkpointing frequency is low, I/O overheads are significant enough to warrant asynchronous multi-level checkpointing methods [21]: the processes write the checkpoints to the fastest local memory (e.g., GPU memory), then let the application continue running, while in the background they flush the checkpoints asynchronously to slower memory tiers and eventually the external repository. However, at high checkpointing frequency, such methods have two limitations: (1) there is only a limited amount of spare space available on the fastest memory tiers to cache checkpoints, so the HPC workflow may be delayed if it produces new checkpoints faster than they can be flushed to slower memory tiers; and (2) since the entire checkpoint record needs to be persisted, its accumulated size may quickly explode to large sizes and produce unacceptable resource utilization, even if performance considerations were not a concern. Thus, there is a need to simultaneously reduce both the I/O overheads and space utilization of checkpoints.

Limitations of state-of-art: One strategy that simultaneously achieves both goals is *data reduction*. The key idea is simple: if we can reduce the sizes of the checkpoints, they are both faster to flush to slower memory tiers, and they occupy less space simultaneously. In this regard, many compression techniques have been proposed, both lossy [30] and lossless [9]. They aim to solve a trade-off between fidelity compared with the original checkpoint data, compression ratio, and compression throughout. Not all compression algorithms are feasible for reducing I/O overheads. In this case, the I/O overhead is the sum between the compression and flush overhead, which means that compression reduces the I/O overhead only if it is faster than the duration of flushing the difference between the original and the compressed size. Even if compression may effectively reduce the I/O overheads and space utilization of individual checkpoints, in our case, we are interested in the entire checkpoint record. Under such circumstances, it is often the case that the checkpointed data changes only partially from one checkpoint to another. For example, graph applications use data structures that are very sparsely updated [16]. Thus, additional opportunities exist to take advantage of specialized data reduction techniques for checkpoints that evolve in time. Incremental checkpointing techniques aim to do so by means of dirty data tracking (i.e., detect what data was touched since the last checkpoint) or de-duplication [2, 6, 17, 24, 29]: save a full checkpoint initially, then save only the differences later. However, such techniques are either slow at detecting the differences, generate a large amount of metadata to keep track of the differences, or ignore important spatiotemporal checkpoint data redundancy (e.g., checkpoint data duplicated in a different checkpoint at a different position). Furthermore, the checkpointed data is typically generated on GPU memory, which has additional limitations compared with the host memory of compute nodes, and therefore limits the applicability of some incremental checkpointing techniques (e.g., based on dirty memory page tracking).

Contributions: To address the limitations mentioned above, in this paper, we propose a novel incremental checkpointing method that: (1) identifies and de-duplicates repeating patterns across the checkpoints of the entire checkpoint record; (2) extracts a compact metadata representation of these repeating patterns; (3) serializes the differences and the compact metadata representation efficiently into host memory for asynchronous transfer to other storage tiers; and (4) takes advantage of modern GPU accelerators to scale to tens of thousands of GPU cores. To this end, we propose a Merkle tree-based [15] de-duplication method that achieves a high data reduction throughput and rate, effectively reducing the I/O overheads and space utilization.

We summarize our contributions as follows. First, we introduce a series of design principles that are at the core of our method: (1) identify repeating data chunks at fine granularity (hundreds of bytes) through hashing that leverages spatial and temporal redundancy across the entire checkpoint record; (2) coalesce such contiguous chunks into a hierarchic set of repeating non-overlapping patterns that are matched against the entire checkpoint record to obtain a compact metadata representation; (3) collect and assemble the compact metadata and unique chunks into a separate contiguous GPU buffer that is optimized for transfer to the host memory; and (4) leverage fused GPU kernels that feature massive parallelism and low latency and synchronization overheads in order to achieve high scalability (Section 2.1). Second, we propose highly parallel algorithms based on the above design principles. In particular, we zoom on the aspect of how to coalesce contiguous chunks into a hierarchic set of repeating overlapping patterns efficiently in parallel by leveraging the structure of Merkle trees (Section 2.2). Third, we illustrate how to implement our algorithms by proposing a research prototype that leverages Kokkos performance-portable abstractions, which generalize our method to various GPU accelerators (Section 2.4). We demonstrate the benefits of our solution for a real-life graph application using extensive experiments at scale for a variety of graphs. The results show our solution reduces the I/O overhead and space utilization of checkpointing by up to orders of magnitude compared with state-of-art incremental checkpointing and compression techniques (Section 3).

2 SYSTEM DESIGN

This section introduces our approach. A high-level overview is depicted in Figure 1.

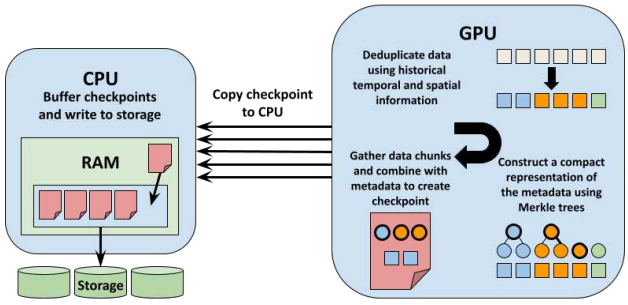


Figure 1: Our method in a nutshell: each process performs de-duplication on its own GPU.

2.1 Design Principles

De-duplication of data chunks using fine-grain hashing that is spatiotemporal agnostic: We assume an HPC workflow in which processes are co-located on the same compute node, each assigned to a dedicated GPU. In this case, each process produces a checkpoint record with high frequency directly into the GPU memory. Since the spare GPU memory available for checkpointing is limited, we cannot afford to hold the entire checkpoint record in the GPU memory, even if we apply an incremental checkpointing technique that stores only the differences. Furthermore, even if we had enough free space on the GPU memory to hold the entire checkpoint record, it is not feasible to compare a new checkpoint with all previous checkpoints in the historical record to identify the differences. Thus, we propose a hash-based method that splits a new checkpoint into fine-grain chunks (in the order of tens or hundreds of bytes), then hashes each chunk to produce a set of unique chunk hashes, which can be compared with the accumulated set of unique chunk hashes of the checkpoint record to identify the data chunks that are unique to the new checkpoint. Using this method, a chunk must be stored only the first time it is encountered, regardless of how many times it appears again in the same checkpoint (spatial duplication) or future checkpoints (temporal duplication). Although such methods have been proposed before in the context of incremental block storage [14], we operate directly in the memory of GPU accelerators, which introduces additional challenges. First, unlike storage systems, we cannot afford to access a separate metadata repository with a historical record of unique hashes due to high I/O latency. Therefore, we propose to keep a distinct record for each process in the GPU memory. Second, as the historical record of unique hashes grows over time, there is a need for efficient indexing and lookup techniques that are GPU-optimized. To this end, we leverage specialized hash tables, as detailed in Section 2.4.

Compact hierarchic representation of contiguous repeating patterns using Merkle tree-based metadata: A GPU-optimized data structure that holds the entire checkpoint record of unique chunk hashes enables us to identify a minimal set of chunks representing the difference between new and older checkpoints. However, this set may be very large since we are using fine-grain chunk sizes. Consequently, storing naive metadata about the chunks (e.g., an entry for each chunk that identifies the checkpoint ID and offset where it first occurred) quickly leads to an explosion of the metadata size, which may dramatically reduce the space savings, even if the difference is negligible. On the other hand, it is essential to note that the chunks may form large contiguous regions that repeat both within the same checkpoint and across past checkpoints. Therefore, there is an opportunity to reduce the metadata sizes by identifying and leveraging such contiguous regions directly. To this end, we propose a hierarchical Merkle tree-based [15] method that stores hashes corresponding to different arrangements of non-overlapping adjacent regions in the historical record of unique hashes (bound by up to two times more hashes than the naive method) to identify a close to a minimum number of contiguous regions that cover the difference. Using this method, the metadata size can be dramatically reduced under the right circumstances, as we only need to store the difference between the checkpoint ID and offset where an entire region appeared the first time. To this end, we introduce a specialized algorithm detailed in Section 2.2.

Efficient combined serialization of metadata and unique chunks as a consolidated difference optimized for transfer to host memory: Once we have identified the smallest set of regions and unique chunks that make up the difference, we need to consolidate the data and metadata into the host memory to obtain a single checkpoint object that can be flushed asynchronously further down the storage hierarchy. However, these unique chunks may be scattered all over the GPU memory, especially if the de-duplication is effective and the difference is negligible. Therefore, a naive strategy that initiates transfers of individual chunks from the GPU memory to the host memory suffers significant I/O bandwidth degradation since it involves non-trivial latencies to set up the transfer, not to mention cache misses. Therefore, we propose serializing the metadata and the unique chunks into a consolidated difference directly on the GPU memory. Then, for the consolidated difference, initiating a single device-to-host data transfer is enough, which can take advantage of the full PCIe bandwidth that links the GPU memory with the host memory. Even if the consolidation leverages high GPU-to-GPU memory bandwidth, it is non-trivial because it needs to consider coalesced memory accesses that occur when concurrently running threads access memory close to each other. Doing so allows the hardware to predict and retrieve data ahead of time. Cache hierarchies are also better utilized. To this end, we design a specialized serialization method that pre-calculates offsets in the consolidated difference and assigns GPU threads to parallelize the data transfers by the observations above.

Fused GPU kernels for optimal massive GPU parallelism: Without taking advantage of the massive parallelism of GPUs, even simple steps in our method, such as hashing the data chunks or performing GPU memory accesses and GPU-to-GPU data transfers, are time-consuming and lead to large overheads. Therefore, our method needs to scale to a large number of GPU cores, which are on the order of ten thousand on modern GPUs. However, this is non-trivial because hashing of data chunks, indexing, and lookup in the hash historical record, generating compact metadata representations, and consolidating checkpoint differences are complex operations with tight dependencies. Therefore, it is not enough to reason about these aspects as independent steps that can be parallelized using separate GPU kernels, as such a naive method would introduce unacceptable latencies associated with submitting and executing new kernels. To address this issue, we propose to use a single fused kernel that takes advantage of containers and abstractions offered by performance portability abstractions such as Kokkos to parallelize the execution into related "waves" that ensure the work is evenly distributed and maximizes coalesced memory accesses without delays between waves, as detailed in Section 2.4.

2.2 Zoom on Merkle Tree-based Compact Metadata

We propose a heuristic algorithm to identify a close to a minimal number of non-overlapping contiguous regions that fully describe the difference between a new checkpoint and all previous checkpoints in the checkpoint record. Using this method, in addition to the unique chunks encountered in the latest checkpoint, we only need to store a small amount of enough metadata to restore the checkpoint later fully.

Specifically, we assume the fine-grain chunks are the leaves of a (potentially incomplete) binary tree. Then, we compute the hashes not only for the leaves but also in a bottom-up fashion for the intermediate tree nodes by hashing the left child's hash with the right child's hash, similar to Merkle trees. We only go one level up if the hash of the left and right child were found in the historical record of unique hashes and represent a contiguous region that can be consolidated. If this is the case, the new region is added to the historical record of unique hashes. Then, we collect the intermediate nodes that were touched by this process and are the closest to the tree's root. This yields a compact representation of the checkpoint difference, both with respect to the new chunks encountered the first time and the reused chunks from previous checkpoints.

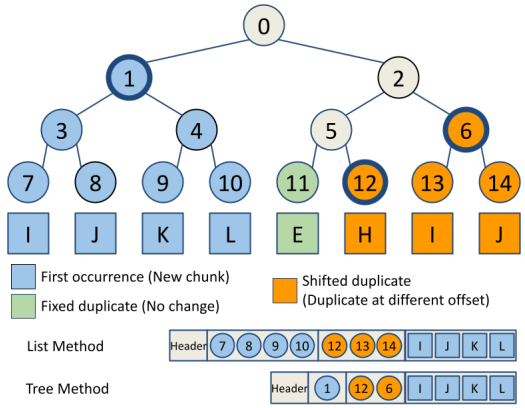


Figure 2: Example of compact metadata representation. Our method reduces the metadata amount from 7 entries to 3 entries compared with a naive method.

To understand why the method works, consider an example depicted in Figure 2. The checkpoint historical record includes a single first checkpoint that was taken fully and the historical record of unique hashes consists of all possible non-overlapping regions corresponding to the intermediate nodes 0-14. Then, a second checkpoint is taken, and the new chunk 11 is identical to the previous chunk 11 (which we refer to as *fixed* duplicate). In contrast, chunk 12 is identical to another previous chunk from the first checkpoint other than 12 (which we refer to as *shifted duplicate*). All other chunks between the first and second checkpoints are different.

The algorithm works as follows. First, we hash the chunks of the second checkpoint and insert any new hashes into the historical record of unique hashes, which results in four inserts I, J, K, L, referred to as *first-time occurrences*. Note that chunks 13 and 14 are identical to chunks 7 and 8. Even though they belong to the second checkpoint, they are still marked as shifted duplicates, just like chunk 12. Chunks 11 and 12 are the only ones that form two non-contiguous regions that cannot be consolidated. Therefore, we stop. For all other leaves, we go one level up. We consolidate chunks 7 and 8 into region 3 and chunks 9 and 10 into region 4. Both region 3 and region 4 are added to the historical record of unique hashes. Then we consolidate chunks 13 and 14 into region 6. Since the hash of region 6 is identical to the hash of region 3, which already is in the historical record of unique hashes, the entire region 6 is marked as a shifted duplicate. The process continues only for regions 3 and 4, which can now be consolidated into region 1, also inserted in the historical record of unique hashes. We obtain the compact metadata representation of the difference as a set of three non-overlapping regions: 1, 12, 6. We can omit chunk 11 from the difference since it remains unchanged from the first checkpoint. Finally, we obtain a mix of metadata describing the first-time occurrences and shifted duplicates, followed by the chunk content corresponding only to the first-time occurrences. Using this method, we save only three metadata entries compared with the naive method that saves a metadata entry for each non-fixed duplicate chunk (referred to as the List method). Now the compact metadata and new unique chunks are ready to be serialized and transferred to the host memory.

Algorithm 1 De-duplication using compact metadata.

Input: $Chunks, Tree, Leaves, Labels, Map$
1: **for all** $leaf \in Leaves$ **do in parallel**
2: $digest \leftarrow$ Hash($Chunk(leaf)$)
3: **if** $digest == Tree(leaf)$ **then**
4: $Labels(leaf) \leftarrow$ FIXED_DUPL
5: **else**
6: $entry \leftarrow (leaf, chkptID)$
7: $success \leftarrow Map$.insert($digest$,$entry$)
8: **if** $success$ **then**
9: $Labels(leaf) \leftarrow$ FIRST_OCUR
10: **else if** not $success$ **then**
11: $(leaf_{old}, chkptID_{old}) \leftarrow Map[digest]$
12: $sameID \leftarrow chkptID_{old} == chkptID$
13: **if** $leaf < leaf_{old}$ && $sameID$ **then**
14: $Labels(leaf_{old}) \leftarrow$ SHIFT_DUPL
15: $Labels(leaf) \leftarrow$ FIRST_OCUR
16: $leaf_{old} \leftarrow leaf$
17: **else**
18: $Labels(leaf) \leftarrow$ SHIFT_DUPL
19: **end if**
20: **end if**
21: $Tree(leaf) \leftarrow digest$
22: **end if**
23: **end for**
24: **for** $level \in Tree$ **do**
25: **for all** $node \in level$ **do in parallel**
26: **if** $child_l$ and $child_r$ are FIRST_OCUR **then**
27: $Labels(node) \leftarrow$ FIRST_OCUR
28: $Tree(node) \leftarrow$ Hash($(Tree(child_l, child_r))$
29: $Map[Tree(node)] \leftarrow (node, chkptID)$
30: **end if**
31: **end for**
32: **end for**
33: **for** $level \in Tree$ **do**
34: **for all** $node \in level$ **do in parallel**
35: **if** $Labels(child_l) \neq Labels(child_r)$ **then**
36: save roots $child_l$ and $child_r$
37: **else if** $child_l$ and $child_r$ are SHIFT_DUPL **then**
38: $Tree(node) \leftarrow$ Hash($Tree(child_l), tree(child_r)$)
39: **if** $Tree(node) \in Map$ **then**
40: $Labels(node) \leftarrow$ SHIFT_DUPL
41: **else**
42: save roots $child_l$ and $child\_r$
43: **end if**
44: **end if**
45: **end for**
46: **end for**

Algorithm 1 lists the pseudocode corresponding to our metadata compaction method in greater detail. The historical record of unique hashes is denoted $Map[hash]$, while $Tree[node]$ and $Label[node]$ are temporary data structures that hold the hashes of leaves (intermediate nodes) and, respectively the type of the region covered by each leaf (intermediate node). Identical labels mean a region can be consolidated.

According to the design principle that aims at leveraging the massive parallelism offered by GPUs, we parallelize this algorithm level-by-level. However, to avoid a situation where shifted duplicates are hashed faster than first-time occurrences (which leads to a missing entry in the historical record of unique hashes and therefore missed de-duplication opportunities), we perform the parallelization in two stages: first, we process the sub-trees corresponding to the first-time occurrences, then we process the sub-trees corresponding to the shifted duplicates.

To restore a checkpoint from the differences, it is enough to start from the first-time occurrences, then fill the fixed duplicates and finally assemble the shifted duplicates from the corresponding checkpoint ID (which can be a previous checkpoint or the current checkpoint to be restored).

2.3 Architecture

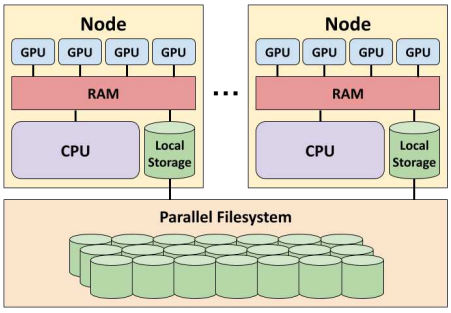


Figure 3: Architecture of multi-level asynchronous checkpointing that integrates our GPU-accelerated de-duplication.

Figure 3 depicts the general architecture of an asynchronous checkpointing that integrates with our GPU-accelerated method. Each compute node features multiple GPUs. A large number of compute nodes compete for the I/O bandwidth of an external repository, typically a parallel file system. Each application process uses a dedicated GPU and stops during checkpointing to perform the de-duplication on the GPU according to the steps illustrated in Figure 1. Then, it transfers the consolidated difference to the host memory and resumes the computations. From this point forward, an asynchronous multi-level checkpointing runtime flushes the differences stored in the host memory down the storage hierarchy (local storage, parallel file system). Since each process maintains its own record of unique hashes on its GPU, the only bottleneck is the competition for PCIe bandwidth between the GPUs when transferring the differences. This competition is nevertheless much lower compared with the case when the full checkpoints are transferred to the host memory, in which case much larger sizes are involved. Furthermore, intermediate storage tiers like host memory and local SSDs are filling up slower, which helps amortize the competition for the I/O bandwidth of the parallel file system. Ultimately, this leads to better utilization of the entire storage hierarchy and lower I/O overheads, in addition to storage space savings.

2.4 Implementation

Our implementation builds on performance portable abstractions that enable efficient massive parallelization into fused kernels while offering optimized implementations of GPU-aware hash tables (used to maintain the historical record of unique hashes).

Parallelization with Kokkos: We implement our method using the Kokkos performance portability framework [32] for parallel execution on CPUs and GPUs. Kokkos includes several execution and data structure abstractions for developing scalable applications. We use the UnorderedMap provided by Kokkos. The UnorderedMap is designed to handle thousands of concurrent insertions. The map is lock-free and minimizes the use of atomic operations. This performance-focused design is important for calculating and identifying differences between thousands of hashes. Kokkos provides flexible parallel execution constructs that allow direct control of work division. The ability to control which chunks are assigned to which threads and in what order greatly impacts performance, as outlined in the subsequent sections. Merkle trees are complete binary trees, so we store them in a flattened array and identify parent-child relationships using simple formulas based on the offset in the array. This simplifies tree search and management on the GPU, as the array format does not waste space on unused pointers.

Efficient hash calculation using GPUs: We use the 128-bit Murmur3 [1] hash function for comparing chunks. Murmur3 is a common non-cryptographic hash function used for hash tables. A fast hash function such as Murmur3 is necessary to maximize de-duplication throughput. Slow cryptographic hash functions such as MD5 [26] would introduce a bottleneck. To efficiently leverage the GPUs hardware, we structure the code such that successive threads compute hashes for successive chunks. By doing so, we ensure that the stride between memory accesses is reduced. Reducing the stride decreases the number of memory accesses to global memory, speeding up the hash calculation. Optimized memory access patterns are necessary to utilize the GPU's bandwidth and computational capabilities fully. This is particularly true for hashing data since the computational cost of Murmur3 is already low. Our method may compute and store up to twice the number of hashes in the historical record of unique hashes since the number of intermediate nodes is equal to the number of leaves minus one. However, this is a worst-case scenario encountered only when the checkpointed data fully changes during the checkpoint interval. This can be easily detected, and incremental checkpointing can be deactivated accordingly. Conversely, when the checkpointed data remains unchanged between checkpoints, our method may calculate intermediate nodes unnecessarily. This may be mitigated by adopting a top-down method. Another important aspect is the size of the chunks, which needs to be larger than double the size of the hash values. In our case, each Murmur3 hash digest is 16 bytes: so long as the chunk size exceeds 32 bytes, the cost of computing an inner node is lower than that of a leaf node. This represents a trade-off: hashing smaller chunks of data improves the memory access pattern by reducing the stride between memory accesses, but

increases the number of inner nodes. Thanks to compact metadata representation, the latter aspect is of lesser concern. Finally, we did not consider hash collisions. If hash collisions are a concern, they can be mitigated by using a cache of chunks that can be directly compared in parallel with the metadata compaction.

High throughput serialization of scattered chunks: As with hash calculation, optimizing memory accesses for gathering scattered chunks is important in order to obtain high GPU-to-GPU transfer throughput for the serialization of the consolidated checkpoint difference. Rather than having each thread copy a different chunk, we use a team of threads to copy each chunk into the contiguous buffer. This ensures that memory accesses coalesce and memory bandwidth utilization is maximized. Without such optimizations, even if the transfer from the GPU to the host memory is accelerated due to contiguity, this benefit would be negated by poor serialization throughput.

3 EVALUATION

3.1 Setup

We perform our tests on two supercomputers at Argonne National Laboratory's Leadership Computing Facility (ALCF): ThetaGPU and Polaris. Theta GPU is a DGX A100-based system comprised of 24 NVIDIA DGX A100 nodes, each with eight NVIDIA A100 Tensor Core GPUs and two AMD EPYC 7742 64-core CPUs. Memory-wise, each node is equipped with 1 TB of DDR4 and 320 GB GPU memory for 24 TB DDR4 and 7.6 TB GPU memory. The nodes are interconnected using 20 Mellanox QM9700 HDR200 40-port switches wired in a fat-tree topology. External storage is provided by a Lustre parallel file system, which is mounted using POSIX and provides an aggregated I/O bandwidth of 250 GB/s. Polaris is a 560-node HPE Apollo 6500 Gen 10+ system. Each node consists of an AMD 32-core EPYC 7543P CPU, 4 A100 GPUs, 512 GB of DDR4, two 1.6 TB SSDs, and 4 A100 GPUs. The nodes are connected using the Slingshot 10 network. The number of processes tested ranges from a single process to 64 processes. Each process has its own GPU and is isolated from the other processes.

3.2 Methodology

This section outlines the experimental procedures for evaluating our work. The use case, state-of-the-art, performance metrics, experimental scenarios, and input data are shown in detail.

Application use case: Our driver application for the experimental evaluation is ORbit ANd Graphlet Enumeration at Scale (ORANGES), a parallel graph application that takes a graph as an input and computes each vertex's graphlet degree vector (GDV) in order to enable graph matching. A graphlet is an induced subgraph of a small number of vertices. A graphlet degree vector can be seen as a generalization of the degree concept [31]. The degree of a vertex represents the number of times a vertex is part of an edge. The graphlet degree vector represents the number of times a vertex is part of different graphlets, with one entry for each graphlet. GDVs are used for graph-matching applications, such as in comparing phylogenetic networks in bioinformatics and comparing event graphs in large-scale HPC applications. We create the GDV on all two to five vertices graphlets. Each vertex in the graph is associated with a vector of size 72, representing the different positions (or orbits) of the vertex in the 30 possible graphlets (more details on the graphlets and orbits are given in [10]). For each vertex in the graph, we identify the graphlets associated with it and its different orbits in the graphlets. Based on this calculation, we increase the count of each orbit in GDV associated with the vertex. If the graph is sparse, then the GDV is also sparse, as not all graphlets are formed. For example, triangles are rarely formed in event graphs representing communications in HPC simulations and in almost planar road graphs. Due to this, only 10 possible 30 graphlets are formed frequently, and the remaining 20 very rarely, if at all. Graphs will also have repeated substructures which can result in some GDVs being similar to others. The updated pattern depends on the input graph, simplifying the exploration of different patterns. These characteristics make ORANGES a good candidate to showcase the benefits of our work.

Compared state-of-the-art methods: We compare our method (henceforth denoted *Tree*) with a baseline checkpointing method that always stores a full checkpoint (denoted *Full*). Additionally, we implemented a *Basic* incremental checkpointing method that breaks the checkpoint into chunks, hashes the chunks, then builds a bitmap to indicate what chunks are new and what chunks remain unchanged. It saves the bitmap and the new chunks. Furthermore, we implemented a *List* method that is identical to our method except for the metadata compaction, which is omitted. Instead, a full list of all first-time occurrences and shifted duplicates is stored along the new chunks. For fairness, both the Basic and List methods benefit from the same massive parallelization optimizations introduced by our method. Furthermore, we use several lossless compression algorithms included with the open-source *nvCOMP* [22] library provided by NVIDIA. Since our application counts graphlets and needs an exact output for correctness, lossy compression is not applicable.

Metrics: We focus on two metrics when evaluating our work; data de-duplication ratio and de-duplication throughput. The de-duplication ratio is measured as the size of the full checkpoints divided by the size of the de-duplicated checkpoints. Higher ratios indicate greater space savings. Throughput is calculated as the size of the original data divided by the time it takes to create and copy the incremental checkpoint from the GPU to the host memory. In the case of Full, this measures the GPU to host flush throughput. In the case of the other methods, the de-duplication throughput includes both the overhead of compression/de-duplication and the overhead of GPU-to-host transfers.

Experimental scenarios: We present three scenarios that examine different factors affecting our method versus existing methods. The first scenario studies the impact of chunk size on de-duplication performance. Chunk size determines the de-duplication granularity, which directly affects checkpoint size reduction and the computational overhead of checkpointing. We vary the chunk size from 32 to 512 bytes and compare our method with the Full, Basic, and List method in terms of data de-duplication ratio and de-duplication throughput. This scenario leverages a single GPU. The second scenario investigates how the benefits of de-duplication accumulate as the checkpointing frequency increases. Specifically, we capture a full initial checkpoint, then another $N-1$ incremental checkpoint that is evenly distributed during the runtime R (i.e., we use a fixed

checkpoint interval R/N). We vary N from 5 to 20 and aggregate the metrics for all captured checkpoints (excluding the first checkpoint). Using this method, we can compare the results for different input graphs that produce different runtimes (in the order of minutes). This scenario again leverages a single GPU. The last scenario performs strong scaling tests. We vary the number of GPUs from 1 to 64 and take checkpoints every 10 minutes. At scale, for larger dense graphs, the number of iterations rapidly increases, hence the longer checkpoint interval. We focus on the comparison between our Tree method and the Full method.

Input data: For the scenarios mentioned above, we use a set of graphs with different complexities in terms of vertices and edges (Tab. 1). Message Race, Unstructured Mesh, Asia OSM, and Hugebubbles are used for the single process tests, and Delaunay is used for the scaling test. The Message Race and Unstructured Mesh scenarios are event graphs representing communication patterns in HPC benchmarks. Asia OSM, Hugebubbles, and Delaunay N24 are graphs from the SuiteSparse collection [3]. The event graphs are more sparse than the graphs from SuiteSparse, with fewer dense subgraphs. As noted later, this leads to improved results for the event graphs.

Graph	$\|\|V\|\|$	$\|\|E\|\|$	GDV size
Message Race	11,174,336	16,761,248	3.26 GB
Unstructured Mesh	14,418,368	21,627,296	4.21 GB
Asia OSM	11,950,757	25,423,206	3.49 GB
Hugebubbles	18,318,143	54,940,162	5.35 GB
Delaunay N24	16,777,216	100,663,202	4.9 GB

Table 1: Input graphs used for our tests. Delaunay N24 is used for the scaling test.

Before running ORANGES, we pre-process the graphs by reordering the vertices using Gorder [36]. Gorder is a graph reordering application that relabels a graph's vertices to improve cache performance while maintaining the graph's topological structure. Gorder uses an approximate greedy algorithm with a priority queue to find a graph ordering where connected vertices are stored close together. Keeping the vertices close together improves cache reuse when operating on graphs and is a typical optimization applied by the graph community.

3.3 Results

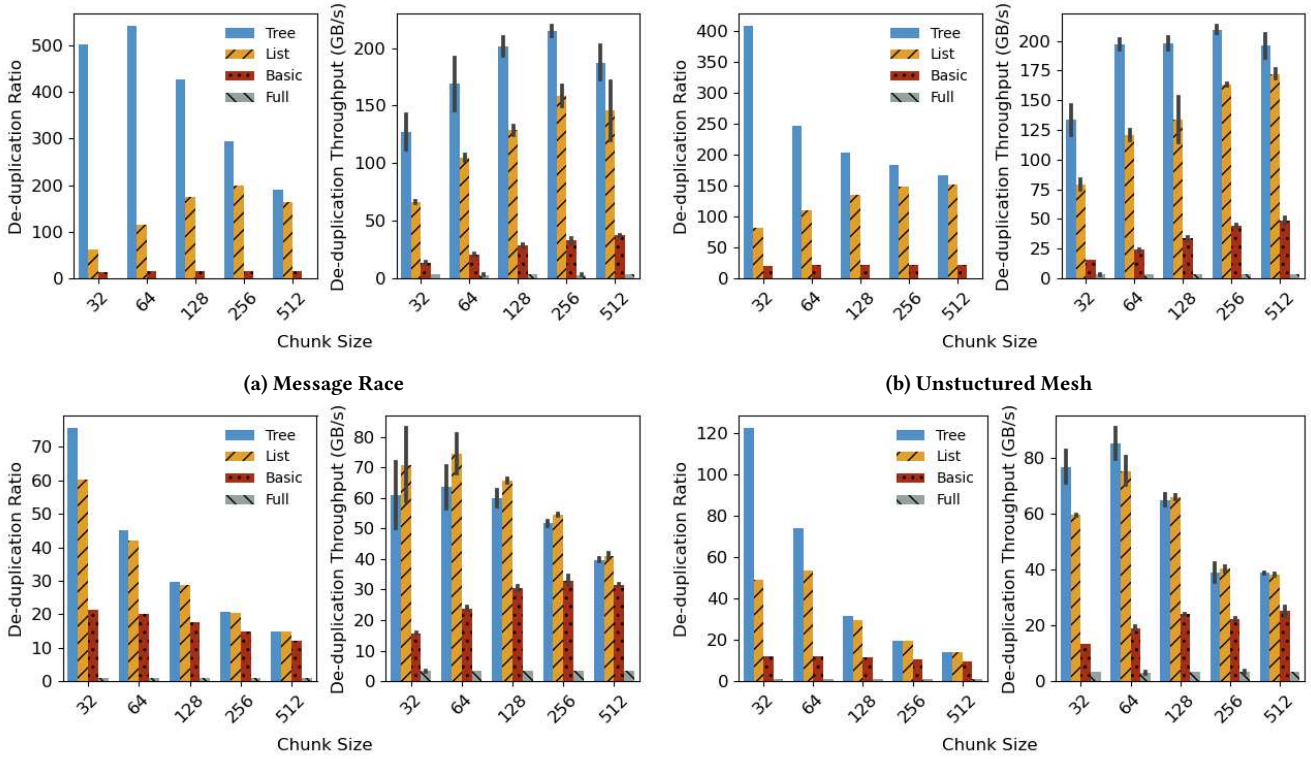
Figure 4 shows the impact of chunk size on the de-duplication ratio and throughput for our Tree method versus Full, Basic, and List.

Figure 4a highlights the trade-offs when deciding chunk size for the Message Race graph. With 64-byte chunks, our method achieves a 5 times better de-duplication ratio than List (the best among the compared methods). The List method sees a decline in ratio with chunks smaller than 256 bytes—large amounts of metadata cause the decrease. As the chunk size shrinks, more of the checkpoint comprises metadata for tracking duplicate chunks. Our method compacts the metadata, which allows smaller chunk sizes without performance degradation. The throughput values suggest that significant reductions in checkpoint sizes can overcome the additional overheads of identifying compact metadata. Specifically, our method shows superior throughput, matching the improved de-duplication ratio performance. Throughput performance starts to degrade with chunks smaller than 256 bytes, where the additional overhead exceeds the benefits of smaller checkpoints. This behavior is typical of all compared methods. Our Tree method benefits the most from small chunk sizes, allowing smaller chunks without decreasing throughput performance. Figure 4b shows similar performance characteristics for the Unstructured Mesh graph as the Message Race graph.

Figure 4c shows that it is more challenging to de-duplicate the checkpoints for Asia OSM than Message Race or Unstructured Mesh. The de-duplication ratio is lower for all methods. Our Tree method only shows notable improvements with 32-byte chunks, outperforming the other methods. Figure 4d shows the same metrics for Hugebubbles. Similar to Asia OSM, the Hugebubbles graph is more challenging to de-duplicate. Despite the difficulty, we see significant improvements with our Tree method with chunk sizes of less than 128 bytes. With 64-byte chunks and smaller ones, we see a 37% improvement in the de-duplication ratio and a 13% increase in throughput.

Figure 5 shows how checkpoint frequency affects our method's de-duplication ratio and throughput compared with the state-of-art techniques. We capture 5, 10, or 20 checkpoints at moments evenly distributed during the runtime. The de-duplication ratio results in Figures 5a- 5c demonstrate the benefits of using temporal information for de-duplicating data. Increasing the checkpoint frequency reduces the number of updates at each checkpoint. Fewer updates and frequent checkpoints increase the temporal information our method can leverage. However, this does not always compare favorably with compression. For example, our method has worse de-duplication ratios than Zstd for all four graphs. Increasing the number of checkpoints to 20 allows our Tree method to outperform Zstd for all input data, even with the more difficult Asia OSM and Hugebubbles graphs. This behavior is expected, as the compression techniques are limited to individual checkpoints. We are using our Tree method; taking 20 checkpoints results in a smaller total checkpoint size for all graphs except Asia OSM, which sees only a 2% Figures 5d- 5f show that throughput performance is less impacted by increasing checkpoint frequency. Throughput increases for our Tree method as well as for the List, and Basic methods, while the compression techniques are unaffected. The improvements to the Tree method range from 1.37× with the Unstructured Mesh to 2.77× with the Hugebubbles graph.

Figure 6 shows the strong scaling results of the Full method compared with our Tree method. The Delaunay N24 graph is the input, and the number of GPUs varies from 1 to 64. Each process checkpoints independently, but multiple GPUs copying data to a shared CPU can impact performance. We measure the sum of the first ten checkpoints for all processes. Throughput is measured by taking the sum of 10 checkpoints and dividing it by the maximum runtime spent on de-duplication across all processes. Figure 6a shows the sum of total checkpoint sizes for ORANGES running the Delaunay N24 graph. As the number of processes increases exponentially, so does the checkpointed data. At 64 processes, we see a 215× reduction in total checkpoint size compared with Full: 4.33 TB of checkpoints is reduced to 20 GB. The de-duplication throughput is shown in Figure 6b. This indicates that our Tree method has greater throughput than Full, and the throughput maintains or improves as the number of processes increases. Since ORANGES



(a) Message Race

(b) Unstuctured Mesh

(c) Asia OSM

(d) Hugebubbles

Figure 4: Impact of chunk size on de-duplication ratio and throughput for our method (Tree) vs. other incremental methods (i.e., Full, Basic, and List) for the Message Race, Unstructured Mesh, Asia OSM, and Hugebubbles input graphs. Chunk size ranges from 32 to 512 bytes.

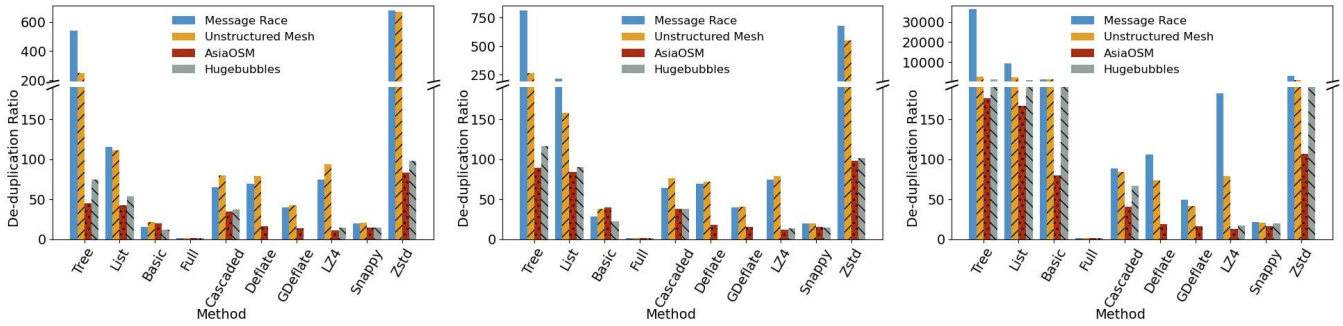
is embarrassingly parallel (it finishes with a reduction of the independently obtained results on each GPU), we did not perform a full comparison at scale between all methods. We anticipate similar trends compared with the single GPU experiments.

4 RELATED WORK

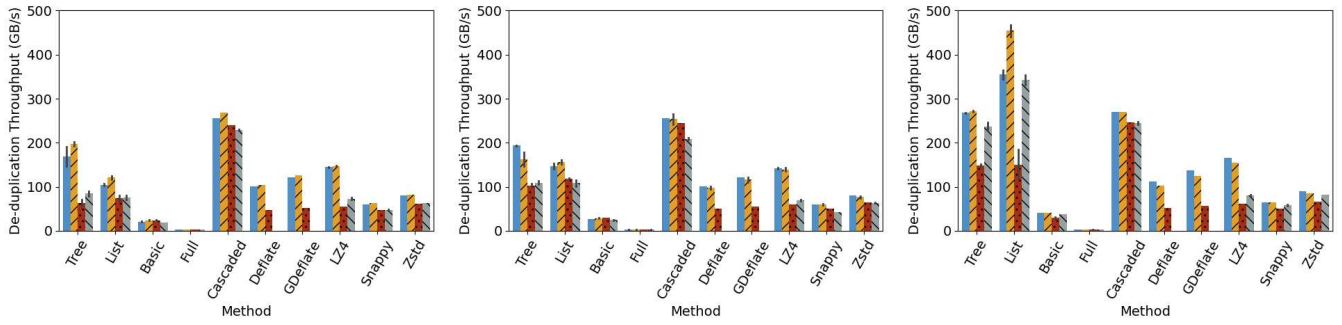
Incremental storage is a well-known technique to accelerate I/O and to reduce storage space utilization. In addition to checkpointing, it is applied in many other scenarios: versioning and incremental snapshots of file systems, virtual machine images, and block devices. They use either dirty page (block) tracking or de-duplication [6, 17, 33, 34] to identify incremental differences. Dirty page tracking for host memory can be accelerated by the OS and hardware using techniques such as user-level faults that avoid expensive context switches triggered by more conventional methods that trap the SEGFAULT signal. However, dirty-page tracking requires specialized kernel support and is unavailable on all platforms. Furthermore, they are typically limited to memory page granularity (e.g., 4 KB), which limits their de-duplication potential (e.g., single-byte changes or writing identical data to the same address can mark an entire page dirty). Such techniques are unavailable on GPUs because the GPU drivers handle the memory virtualization fully transparently [11, 12]. Complementary to incremental storage is the problem of how to re-assemble checkpoints from differences, which involves metadata organization, indexing and search techniques [19, 20]. Several works focus on enabling checkpointing for GPU applications [5, 8, 23, 25]. However, each has drawbacks. Some works [8, 23] only perform essential temporal de-duplication. Others [5, 25] only perform de-duplication at the page or variable level. Methods such as libhashckpt [4] use a hybrid method with multiple change detection systems to reduce the checkpoint size. Another alternative to incremental checkpointing is checkpoint compression, both lossless [9] and lossy [30]. Typical compression algorithms prioritize decompression performance, assuming that compression is a relatively infrequent operation, which does not hold in our high-frequency checkpointing scenario. Furthermore, many compression algorithms cannot leverage the temporal redundancy of data that evolves in time.

5 CONCLUSIONS

This paper presents a scalable GPU-aware Merkle tree-based incremental checkpointing method leveraging de-duplication to reduce the checkpoint sizes and increase the checkpointing throughput simultaneously. To this end, we identify contiguous repeating patterns across the entire checkpoint record, for which we eliminate the redundancy both at the data and metadata levels. We use these fundamental ideas to improve the de-duplication ratio and de-duplication throughput for graph-matching applications by significant margins (up to orders of magnitude) compared with

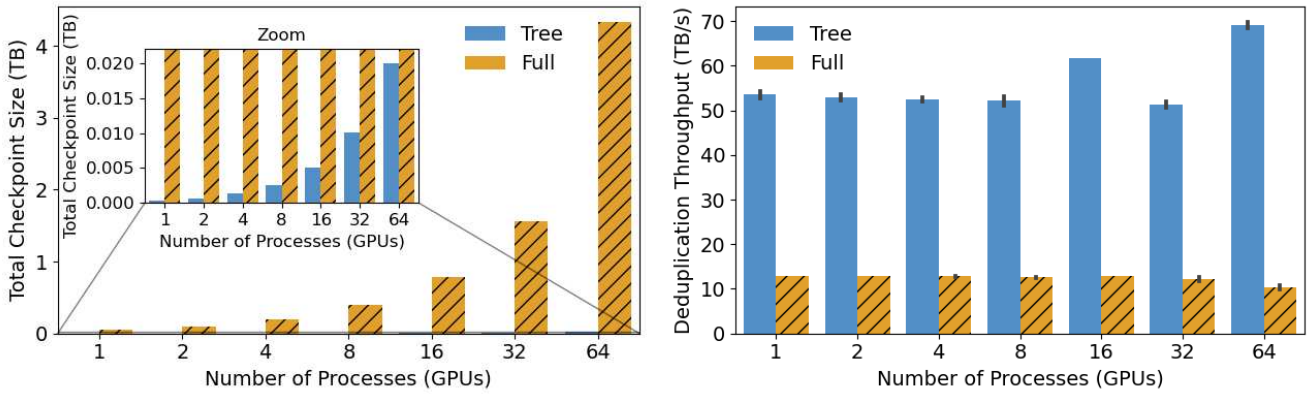


(a) De-duplication ratio for 5 checkpoints (b) De-duplication ratio for 10 checkpoints (c) De-duplication ratio for 20 checkpoints



(d) Throughput for five checkpoints (e) Throughput for 10 checkpoints (f) Throughput for 20 checkpoints

Figure 5: Impact of checkpoint frequency on de-duplication ratio and throughput for our method (Tree) vs. state-of-art (i.e., Full, Basic, and List) and several *nvCOMP* compression algorithms. Results are shown for $N = 5, 10, 20$ checkpoints evenly distributed during the runtime (which varies for each input graph).



(a) Total checkpoint size (b) De-duplication throughput

Figure 6: Strong scaling results with up to 64 GPUs using the Delaunay N24 input graph. Our method (Tree) achieves over two orders of magnitude reduction in checkpoint size compared with Full and retains a high throughput at scale.

other incremental checkpointing methods. Unlike high-throughput compression techniques, our method improves the de-duplication ratio and throughput for an increasing checkpointing frequency, resulting in a size of the checkpoint record up to 67× smaller. Furthermore, our method shows excellent scalability for large graphs, reducing the checkpoint sizes by two orders of magnitude and increasing the checkpointing throughput by almost an order of magnitude compared with full checkpoints. Such benefits are significantly impacted in non-resilience scenarios where incremental checkpoints are used to analyze intermediate results (e.g., reproducibility efforts) or make progress (e.g., adjoint computations).

Encouraged by these results, in future work, we plan to address several directions: evaluating the benefits of our method for other classes of applications, such as adjoint computations; combining our method with compression techniques to further reduce the checkpoint sizes and increase the data reduction throughput (e.g., by compressing the first-time occurrences in the difference); streaming methods that overlap de-duplication with transfers to the host

memory; and scalable reconstruction techniques that efficiently collect scattered compact regions from multiple previous checkpoints.

ACKNOWLEDGMENTS

This material is based upon work supported by: the U.S. Department of Energy (DOE), Office of Science, Office of Advanced Scientific Computing Research, under Contract DE-AC02-06CH11357; the National Science Foundation under Grants #1900888 and #1900765; and the IBM Shared University Research Award at the University of Tennessee. This manuscript has been authored by UT-Battelle LLC under contract DE-AC05-00OR22725 with the US Department of Energy (DOE). Sandia National Laboratories is a multi-mission laboratory managed and operated by National Technology & Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

REFERENCES

[1] Austin Appleby. 2012. SMHasher & MurmurHash. Retrieved from https://code.google.com/p/smhasher.

[2] Iván Cores, Gabriel Rodríguez, Mará J Martín, Patricia González, and Roberto R Osorio. 2013. Improving Scalability of Application-Level Checkpoint-Recovery by Reducing Checkpoint Sizes. *New Generation Computing* 31 (2013), 163–185.

[3] Timothy A Davis and Yifan Hu. 2011. The University of Florida Sparse Matrix Collection. *ACM Transactions on Mathematical Software (TOMS)* 38, 1 (2011), 1–25. https://doi.org/10.1145/2049662.2049663

[4] Kurt B Ferreira, Rolf Riesen, Ron Brighwell, Patrick Bridges, and Dorian Arnold. 2011. libhashckpt: Hash-based Incremental Checkpointing Using GPU's. In *European MPI Users' Group Meeting*. Springer, Santorini, Greece, 272–281.

[5] Rohan Garg, Apoorve Mohan, Michael Sullivan, and Gene Cooperman. 2018. CRUM: Checkpoint-Restart Support for CUDA's Unified Memory. In *2018 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, Belfast, United Kingdom, 302–313.

[6] Roberto Gioiosa, Jose Carlos Sancho, Song Jiang, and Fabrizio Petrini. 2005. Transparent, Incremental Checkpointing at Kernel Level: a Foundation for Fault Tolerance for Parallel Computers. In *SC'05: Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*. IEEE, Seattle, WA, USA, 9–9.

[7] Paul H Hargrove and Jason C Duell. 2006. Berkeley Lab Checkpoint/Restart (BLCR) for Linux Clusters. In *Journal of Physics: Conference Series*, Vol. 46. IOP Publishing, Denver, USA, 494.

[8] Sudarsun Kannan, Naila Farooqui, Ada Gavrilovska, and Karsten Schwan. 2014. Heterocheckpoint: Efficient Checkpointing for Accelerator-based Systems. In *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. IEEE, Atlanta, USA, 738–743.

[9] Fabian Knorr, Peter Thoman, and Thomas Fahringer. 2021. ndzip: A High-Throughput Parallel Lossless Compressor for Scientific Data. In *2021 Data Compression Conference (DCC)*. IEEE, Snowbird, USA, 103–112.

[10] Oleksii Kuchaiev, Tijana Milenković, Vesna Memišević, Wayne Hayes, and Nataša Pržulj. 2010. Topological network alignment uncovers biological function and phylogeny. *Journal of the Royal Society Interface* 7, 50 (2010), 1341–1354.

[11] 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 *Proceedings of the ACM International Conference on Supercomputing*. Association for Computing Machinery, Phoenix, AZ, USA, 171–183.

[12] Jiacheng Ma, Xiao Zheng, Yaozu Dong, Wentai Li, Zhengwei Qi, Bingsheng He, and Haibing Guan. 2018. gMig: Efficient GPU Live Migration Optimized by Software Dirty Page for Full Virtualization. In *Proceedings of the 14th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. Association for Computing Machinery, New York, NY, USA, 31–44.

[13] Avinash Maurya, Mustafa Rafique, Thierry Tonellot, Hussain AlSalem, Franck Cappello, and Bogdan Nicolae. 2023. GPU-Enabled Asynchronous Multi-level Checkpoint Caching and Prefetching. In *HPDC'23: The 32nd International Symposium on High-Performance Parallel and Distributed Computing*. Association for Computing Machinery, Orlando, FL, USA.

[14] Dirk Meister, Jurgen Kaiser, Andre Brinkmann, Toni Cortes, Michael Kuhn, and Julian Kunkel. 2012. A Study on Data Deduplication in HPC Storage Systems. In *SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE, Salt Lake City, UT, USA, 1–11.

[15] Ralph C Merkle. 1988. A Digital Signature Based on a Conventional Encryption Function. In *Advances in Cryptology—CRYPTO'87: Proceedings 7*. Springer, Santa Barbara, CA, USA, 369–378.

[16] Lifeng Nai, Yinglong Xia, Ilie G Tanase, Hyesoon Kim, and Ching-Yung Lin. 2015. GraphBIG: Understanding Graph Computing in the Context of Industrial Solutions. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. Association for Computing Machinery, Austin, TX, USA, 1–12.

[17] Bogdan Nicolae. 2013. Towards Scalable Checkpoint Restart: A Collective Inline Memory Contents Deduplication Proposal. In *IPDPS '13: The 27th IEEE International Parallel and Distributed Processing Symposium*. Association for Computing Machinery, Boston, USA, 19–28.

[18] Bogdan Nicolae. 2020. DataStates: Towards Lightweight Data Models for Deep Learning. In *SMC'20: The 2020 Smoky Mountains Computational Sciences and Engineering Conference*. Springer, Nashville, United States, 117–129. https://doi.org/10.1007/978-3-030-63393-6_8

[19] Bogdan Nicolae. 2022. Scalable Multi-Versioning Ordered Key-Value Stores with Persistent Memory Support. In *IPDPS 2022: The 36th IEEE International Parallel and Distributed Processing Symposium*. IEEE, Lyon, France, 93–103.

[20] Bogdan Nicolae, Gabriel Antoniu, Luc Bouge, Diana Moise, and Alexandra Carpen-Amarie. 2011. BlobSeer: Next-generation data management for large scale infrastructures. *J. Parallel Distrib. Comput.* 71, 2 (2011), 169–184.

[21] Bogdan Nicolae, Adam Moody, Elsa Gonsiorowski, Kathryn Mohror, and Franck Cappello. 2019. VeloC: Towards High Performance Adaptive Asynchronous Checkpointing at Large Scale. In *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, Rio de Janiero, Brazil, 911–920.

[22] NVIDIA. 2023. *nvCOMP: A library for fast lossless compression/decompression on the GPU*. Nvidia. https://developer.nvidia.com/nvcomp

[23] Konstantinos Parasyris, Kai Keller, Leonardo Bautista-Gomez, and Osman Unsal. 2020. Checkpoint Restart Support for Heterogeneous HPC Applications. In *2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*. IEEE, Melbourne, Australia, 242–251.

[24] James S Plank, Jian Xu, and Robert HB Netzer. 1995. *Compressed Differences: An Algorithm for Fast Incremental Checkpointing*. Technical Report. Citeseer.

[25] Behnam Pourghassemi and Aparna Chandramowlishwaran. 2017. cudaCR: An In-kernel Application-level Checkpoint/Restart Scheme for CUDA-enabled GPUs. In *2017 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, Honolulu, USA, 725–732.

[26] Ronald L. Rivest. 1992. The MD5 Message-Digest Algorithm. RFC 1321. https://doi.org/10.17487/RFC1321

[27] Manuel Rodríguez-Pascual, Jiajun Cao, José A Moríñigo, Gene Cooperman, and Rafael Mayo-García. 2019. Job Migration in HPC Clusters by Means of Checkpoint/Restart. *The Journal of Supercomputing* 75 (2019), 6517–6541.

[28] Elvis Rojas, Diego Pérez, Jon C Calhoun, Leonardo Bautista Gomez, Terry Jones, and Esteban Meneses. 2021. Understanding Soft Error Sensitivity of Deep Learning Models and Frameworks through Checkpoint Alteration. In *2021 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, Portland, OR, USA, 492–503.

[29] Jose Carlos Sancho, Fabrizio Petrini, Greg Johnson, and Eitan Frachtenberg. 2004. On the Feasibility of Incremental Checkpointing for Scientific Computing. In *18th International Parallel and Distributed Processing Symposium, 2004. Proceedings.* IEEE, Santa Fe, NM, USA, 58.

[30] Naoto Sasaki, Kento Sato, Toshio Endo, and Satoshi Matsuoka. 2015. Exploration of Lossy Compression for Application-level Checkpoint/Restart. In *2015 IEEE International Parallel and Distributed Processing Symposium*. IEEE, Hyderabad, India, 914–922.

[31] Nino Shervashidze, SVN Vishwanathan, Tobias Petri, Kurt Mehlhorn, and Karsten Borgwardt. 2009. Efficient Graphlet Kernels for Large Graph Comparison. In *12th International Conference on Artificial Intelligence and Statistics (AISTATS), Society for Artificial Intelligence and Statistics, 488-495 (2009)*. Proceedings of Machine Learning Research, Clearwater beach, USA, 488–495.

[32] Christian R Trott, Damien Lebrun-Grandié, Daniel Arndt, Jan Ciesko, Vinh Dang, Nathan Ellingwood, Rahulkumar Gayatri, Evan Harvey, Daisy S Hollman, Dan Ibanez, et al. 2021. Kokkos 3: Programming Model Extensions for the Exascale Era. *IEEE Transactions on Parallel and Distributed Systems* 33, 4 (2021), 805–817.

[33] Manav Vasavada, Frank Mueller, Paul H Hargrove, and Eric Roman. 2011. Comparing Different Approaches for Incremental Checkpointing: The Showdown. In *Linux Symposium*, Vol. 69. Ottowa, Canada, 69–80.

[34] Dirk Vogt, Armando Miraglia, Georgios Portokalidis, Herbert Bos, Andy Tanenbaum, and Cristiano Giuffrida. 2015. Speculative Memory Checkpointing. In *Proceedings of the 16th Annual Middleware Conference*. Association for Computing Machinery, New York, NY, USA, 197–209.

[35] Qiqi Wang, Parviz Moin, and Gianluca Iaccarino. 2009. Minimal Repetition Dynamic Checkpointing Algorithm for Unsteady Adjoint Calculation. *SIAM Journal on Scientific Computing* 31, 4 (2009), 2549–2567.

[36] Hao Wei, Jeffrey Xu Yu, Can Lu, and Xuemin Lin. 2016. Speedup Graph Processing by Graph Ordering. In *Proceedings of the 2016 International Conference on Management of Data*. Association for Computing Machinery, New York, NY, USA, 1813–1828.