2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks

HeteroCheckpoint: Efficient Checkpointing for Accelerator-based Systems

Sudarsun Kannan, Naila Farooqui, Ada Gavrilovska, Karsten Schwan
College of Computing
Georgia Institute of Technology, Atlanta, Georgia, USA
sudarsun@gatech.edu,{naila, ada, schwan}@cc.gatech.edu

***Abstract*—Moving toward exascale, the number of GPUs in HPC machines is bound to increase, and applications will spend increasing amounts of time running on those GPU devices. While GPU usage has already led to substantial speedup for HPC codes, their failure rates due to overheating are at least 10 times higher than those seen for the CPUs now commonly used on HPC machines. This makes it increasingly important for GPUs to have robust checkpoint/restart mechanisms. This paper introduces a unified CPU-GPU checkpoint mechanism, which can efficiently checkpoint the combined GPU-CPU memory state resident on machine nodes. Efficiency is gained in part by addressing the end-to-end data movements required for checkpointing – from GPU to storage – by introducing novel pre-copy and checksum methods. These methods reduce checkpoint data movement cost seen by HPC applications, with initial measurements using different benchmark applications showing up to 60% reduced checkpoint overhead. Additional exploration of the use of next-generation storage, like NVM, show further promises of reduced checkpointing overheads.**

I. Introduction

It is a well-known fact that low overhead fault tolerance for HPC applications is critical to increasing machine sizes leading to higher failure rates. Recent studies, in fact, point to failure intervals for applications as low as once every 30 minutes, and with increasing levels of node heterogeneity, these intervals are likely to see further shrinkage. For instance, failure data from the TSUBAME2.0 [3] supercomputer with its high GPU/CPU ratio shows that GPU failure rates are as high as $7*10^{-6}$ failures/sec compared to its CPU failure rates of $0.5*10^{-6}$. At the same time, to gain higher performance, applications are spending increasing amounts of time running on GPUs vs. CPUs, one case in point being the PICON code that runs entirely on GPUs, using CPUs only for high-level orchestration tasks [1]. It is critical, therefore, to develop an efficient GPU-based fault tolerance methods on HPC machines.

This paper addresses node failures on heterogeneous high end machines, by creating and experimenting with a unified checkpoint/recovery (C/R) mechanism able to deal with both CPU and GPU failures. While such mechanisms have been well-studied for CPU-based hosts, with GPUs on heterogeneous hosts, a key issue arising for checkpointing GPU state is the lack of direct I/O access from current GPU devices. For checkpointing, GPU-resident data must be moved from the device memory (GDRAM) to CPU accessible DRAM and finally, to a nonvolatile (NV) storage. With increasing device memory capacity and application footprints, data movement needs can be substantial, compared to the available end-end bandwidth from GPUs to NV storage, with potential bottlenecks arising both at (i) the GDRAM-DRAM PCI interface and (ii) the interface to NV storage.

GPU manufacturers are aware of the limitations caused by insufficient GPU-to-host memory and subsequent storage bandwidths. NVIDIA, for instance, offers CUDA-based streaming and host-level memory pinning to reduce the bandwidth impact. At the same time, new non-volatile memory (NVM) technologies like phase change memory (PCM) promise increased bandwidths for accesses from DRAM-to-NV storage (approx. 2GB/sec) compared to (200-300 MB/sec) in SSDs. However, with continuing rapid increases in GPU capabilities and memory capacities, limited data transfer bandwidths remain a concern, opening opportunities for additional software solutions to efficiently save and restore GPU-resident data state.

Our work contributes to improving fault tolerance for heterogeneous HPC machines by development of a low overhead GPU checkpoint mechanism. Extending our previous work leveraging new NVM technologies to efficiently checkpoint host-level state [2], this unified GPU/host checkpoint/restart (C/R) mechanism directly addresses the limited bandwidth from GPUs to NV storage, as follows:

1) efficient use of NVM node-level storage via a memory-based C/R interface;
2) chunk-level data pre-copy to address limitations in end-to-end bandwidth (from device to storage);
3) techniques to eliminate redundant data, using data chunk prediction and checksums; and
4) experimental evaluation demonstrating the effectiveness of C/R with well-known GPU benchmark applications.

II. Background and Related Work

Checkpointing methods. Checkpointing/Restart is a well-known fault tolerance technique where the consistent application state is stored in a NV storage at frequent intervals. The frequency depends on the mean time to failure (MTTF) of the execution environment. Checkpoints can be application transparent [3], [4] or application [5] initiated. Transparent checkpoints do not require application changes, but the entire memory state is saved. Incremental approaches [6], [4] are beneficial only when application state does not

978-1-4799-2233-8/14 $31.00 © 2014 IEEE
DOI 10.1109/DSN.2014.76

change substantially and are captured by setting the dirty bit for memory pages that are modified across checkpoint iterations. Unlike CPU, most of the virtual memory interfaces for GPU device memory are not accessible due to closed device drivers. Further, achieving asynchrony among GPU threads is difficult as the global memory is accessible by all the threads. Therefore, before saving device memory, all threads need to synchronize. Finally, for HPC applications [7] with large memory footprints (in several GBs, greater than the PCI or the NV storage bandwidth), data movement costs can be very high. Checkpoints can be to local storage in compute nodes (NVM/SSD) or remote storage which can be buddy node or a parallel file system (PFS). The local and remote checkpoint intervals are guided by soft errors and hard errors respectively. Soft errors require only application restart (80% of failures [8]) unlike hard errors due to failure of a system component/s. Further, local checkpoints are generally coordinated (with compute phase), whereas remote checkpoints can be asynchronous with compute phase.

GPU checkpointing. While the need for GPU fault tolerance and the methods of improving checkpoint performance is in a nascent stage of research, there are quite a few proposals [9], [10], [11], [12] that have been inspired from well studied CPU checkpointing methods. Xu et al. [11] was one of the first to discuss the need for fault tolerance and proposed a basic checkpoint mechanism. Laosooksathit. S et al. [10] proposed a two level GPU checkpointing approach, i.e., local and remote checkpoint methods and also provided extensive modeling. Bautista-Gomez et al. [13] proposed a topology-aware Reed-Solomon encoding in a three-level checkpoint scheme for hybrid systems with GPUs. CheCUDA [11] proposed a technique of overlapping computation with data movement. One key difference between our work and others are that, our work stresses on the fact that usage of end-end bandwidth by the checkpoint process needs to be improved and also provides techniques that can improve the overall performance of applications. Further, our work also considers the use of next generation storage technologies for GPU checkpoint and points out that just replacing disks with NVM is not sufficient for addressing end-end bandwidth issues. Agarwal. S [14] discuss incremental checkpointing for CPUs by accessing page protection data, but such techniques are hindered on the GPU due to non-availability of open source drivers. In our current work, we use a data chunk checksum based checkpoints. We also discuss a possible solution for using dynamic instrumentation that captures data modifications and save only modified application state. Other work like parallel compression checkpointing focus reducing the overall checkpoint size, but our chunk-based scheme is more focused towards reducing the impact of bandwidth limitations.

NVM for GPU-CPU checkpoints. Future NVMs like PCM and Memristors [15], offer hardware-based persistence with 100x faster access rates compared to existing HDDs and SSDs. Faster access speeds combined with byte addressability, page-based virtual memory support, and higher storage density makes NVMs a promising candidate for reducing the impact of DRAM scalability issues in sub-45nm technology nodes, and to provide persistence for fault tolerance. However, NVMs have certain hardware limitations, including slow writes, low die bandwidth, and high write costs, write latencies are 10x higher and bandwidth is 4x lower compared to DRAM, and other major limitations include 10^8 write durability compared to 10^{16} for DRAM and 40 times higher write energy/ bit.

End-end checkpoint bandwidth. Checkpoint time can be loosely defined as the time taken to move checkpoint data from the GPU device to a memory-based NV storage. The GPU-NVM bandwidth plays a key role in reducing the checkpoint time. As discussed earlier, NVMs have relatively lower bandwidth (2GB/sec) compared to the device-host (PCI) bandwidth, and hence the checkpoint speeds are dominated by $min(PCI, NVMbandwidth)$. While several studies [9], [10], [11], [12] have considered a checkpoint approach of first moving data to DRAM through PCI and from overlapping DRAM-storage device with computation, the main issue with such approaches is that, when running multiple instance of an application across several nodes, if a failure of application checkpoint happens in one node, all the checkpoint data need to be erased across all the nodes and restart from previous consistent state if one exists. A checkpoint is more like a transactional commit, and hence a complete end-end checkpoint is required to improve failure reliability. Further, moving all GPU data to DRAM and not immediately flushing them from DRAM to NV storage before next compute iteration would lead to substantial DRAM pressure. With DRAM scalability bottlenecks already well-known, such approach does not seem to be feasible. When DRAM is not a constraint, then checkpoint bandwidth is dependent on PCI bandwidth. Figure 1 below characterizes both approaches.

Using NVM as virtual memory. The benefits of using NVM as virtual memory [2], [16] include i) ability to exploit VM paging and protection, ii) hiding high write latencies using processor cache, iii) byte addressability and hence avoiding serialization and iv) ability to use NVM not only for storage, but also as heap for processing. Further, current file systems are not optimized for NVMs and require redesign. We extend our prior work on using NVM to support CPU-based checkpoint [2], [17] to support GPU checkpoints. Our NVM design is based on the hybrid memory model where NVM is connected to the memory controller and can be accessed by load/store instructions. GPU applications use specialized NVM interfaces, in a similar manner as with explicit application-initiated HPC checkpointing. During computation, application data remain in GDRAM instead of directly writing to NVM, to avoid a substantial application slowdown (up to 25% [16] for certain classes of write-intensive HPC codes). HeteroCheckpoint takes advantage of NVM byte addressability and NVM use as a persistent heap by checkpointing in granularity of memory objects called chunks. To deal with bandwidth limitations and slow writes to NVM, it uses mechanisms like pre-copy and shadow buffering of chunks. More details about NVM OS level support can be found in [2], [17].

Basic Checkpoint Model. To solve these bandwidth issues,

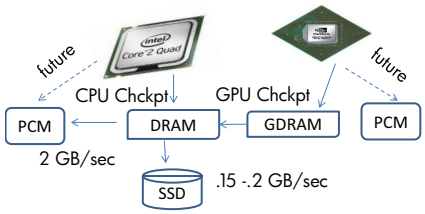

Fig. 1. GPU-Based Checkpoint Design

Functionality	Interface
Allocation	chkCudaAlloc("varname")
Commit	chkCommit(void)
Recover/restart	chkCudaAllocRestart("varname")
Pre-Copy	chkHint(ptrlist)
Checksum	chkValidateChksm(void)

TABLE I
GPU CHECKPOINT INTERFACE

we use a simple model and then discuss the checkpointing methods. The model quantifies the checkpoint time in terms of PCI and storage bandwidth and shows that, for a low overhead checkpoint, overcoming the bandwidth limitation is a key.

$Ttotal = T_{compute} + T_{checkpoint} + T_{restart} + T_{recovery}$

$T_{checkpoint} = N * (chckpt.size/B_{PCI} + chckpt.size/B_{storage})$

$N = fn(MTTF) e.g., Young's model$

where T_{total} is total run time, N - No. of checkpoints

$B_{PCI} = PCIB/W$,

$B_{storage}$- NVM B/W, MTTF - Mean time to failure

III. DESIGN

We next discuss the key design principles of our checkpointing mechanism and propose our novel techniques that reduce the overall end-end bandwidth limitations.

Storage oblivious interfaces using shadowing. During checkpoint, the data from GPU memory can be moved to DRAM or NVM. But the application interfaces for checkpoint need to be oblivious of the underlying storage device and their bandwidth. Further, for optimizing checkpoints, it is important to gain information on each memory allocated variable which changes over time. To enable variable related information, applications are provided with a custom allocation wrapper, shown in Table I, to specify the application variables that need to be checkpointed. We call each such variable *a chunk*. Chunk size can vary between a few bytes to pages and can be multidimensional. The checkpoint library keeps track of each chunk. For all GPU allocated chunks, an equivalent destination chunk is created in NVM. An application can have several such chunks (variables). The key principle of using chunks creates independence between different application variables. This facilitates asynchronous data checkpoints across GPU threads. Note that, asynchrony is restricted within a checkpoint interval and does not affect correctness.

CUDA streams for checkpoint: CUDA streams are the channels which define the order of data movement between a GPU and CPU and the CUDA kernel execution. By default, all kernel memory and movement operations are in one stream and as a result sequential. To enable parallel data movement, each application chunk (i.e., CUDA variable) has its own stream. When application decides to checkpoint,

- All GPU threads are synchronized using a barrier
- On a checkpoint call, all GPU chunks are moved to NVM
- Moving chunks from GPU to NVM can start even before an application call for a coordinated checkpoint

Applications are provided with a custom allocation interface, along with an interface to start checkpoint and restart, and to provide hints for pre-copy operations (see Table I). Figure 2 shows the pseudocode of the proposed checkpoint mechanism.

Reducing the impact of bandwidth. The main contribution of this work is to reduce the bandwidth impact of checkpointing by employing software techniques that are suited for GPU-based checkpoints. We will discuss next two such optimization techniques suited for HPC applications. While we focus specifically on bandwidth issues in moving checkpoint data from GDRAM to NVM , our techniques are applicable for addressing bandwidth issues in GDRAM-DRAM checkpoints also (as discussed in Section II).

Chunk level data pre-copy (PCP). Most HPC applications are long running, with multiple iterations and can have multiple GPU kernels (e.g., molecular dynamics). Also, an application can have many heap variables (chunks) that are modified across different kernels. Some chunks are modified across all iterations until a coordinated local checkpoint is taken. Identifying chunks that are not always modified across kernels and can be pre-copied even before a synchronous checkpoint is started. Figure. 2 shows an application with four kernels and variables one to seven. Not all variables are modified across all kernels. In fact, variables two and three are last modified by second kernel. If the code has reached an iteration count corresponding to the checkpoint, variable 1,2,3 can be checkpointed (pre-copied) in parallel with computation of kernel three and four, even before a coordinated checkpoint is actually started. This can reduce the total data moved exactly at checkpoint. We refer to this technique as chunk pre-copy (PCP) based checkpoints. Note that, there could be multiple threads trying to move data across to NVM. Starting this movement earlier, reduces the GDRAM-NVM interface traffic, reducing the impact of the NVM bandwidth limits. These principles are applicable to any other storage medium like SSD or even DRAM, constrained by interface bandwidth. We believe such methods are important for increasing threads per node in future exascale machines, reducing the per-thread effective bandwidth to storage/NVM.

Figure 3 shows a corresponding timing diagram to compare the regular sequential approach with the pre-copy approach. Block Ci indicates the application execution interval and Li indicated application local checkpoint interval. The upper half shows an execution pattern of a regular checkpoint approach where application execution and checkpoints are sequential. The lower half shows the pre-copy approach where the blocks L1, L2, L3 start even before the completion of the respective execution blocks. As a result, the pre-copy method reduces application runtime, as shown in Section V.

To enable PCP support in the GPU, we need to rely on developer hints for identifying variables not modified any further across the iteration. In the Figure 2, the pre-copy keyword with corresponding variables var1, var2, var3, provides hints

```
chk_validate_chksm();

for ( i = 0; i < nsteps; i++ ) {
        Kernel1<<<…var1, var2>>>
        Kernel2<<<…var2, var3>>>
  if (i == checkpoint_step)
        #PRECOPY Var 1,Var2,Var
         Kernel3 <<<…var4,var5>>
         Kernel4 <<<…var4, var7>>
  if (i == checkpoint_step)
        synchronize;
}
```

Fig. 2. GPU Checkpoint Psuedocode

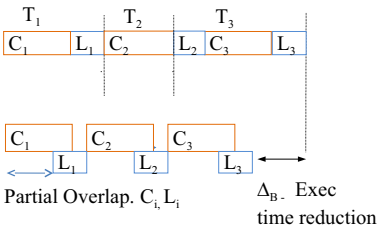

Fig. 3. HeteroCheckpoint timing diagram

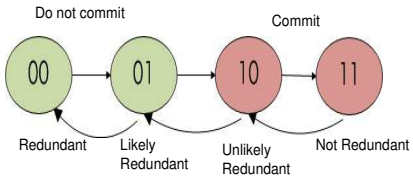

Fig. 4. Redundant data elimination by prediction

to the checkpoint mechanism for enabling pre-copy. In case of CPU-only checkpointing using DRAM such developer hints are not required [2], since using the page protection features application pages can be 'write protected' and changes can be easily tracked. In current GPUs, given the lack of access to page level information, we rely on developer hints. In our future work, we plan to develop compiler-based capture [18].

Redundant data elimination by prediction. One key insight is that not all data chunks change across a given checkpoint interval. There are some chunks which do not get modified across iterations (read-only). Finding the checksum of such chunks avoids an unnecessary copy from GDRAM to NVM, and can reduce checkpoint overhead substantially. This technique is similar to the incremental checkpoint approach, but we apply it to GPU-based checkpointing where incremental data modifications are not easy to capture due to absence of virtual memory page protection or page level dirty tracking techniques [3]. In our current implementation, we delegate checksum calculation to the CPU, but we also propose a more optimal GPU checkpointing approach, using dynamic instrumentation, which obviates the need to perform checksums altogether. Also, we do not apply checksums to memory page granularity, but to a variable sized application chunks.

Delegating checksum calculation to CPU. For identifying application chunk checksums, one approach is to use a GPU kernel during a coordinated checkpoint time. But checksum calculation is a time-consuming operation. Calculating checksums of each variable across checkpoint intervals can severely impact the performance of the application. To avoid checksum calculation from the application execution flow, and to overlap with actual computation, the checkpoint calculation is delegated to an idle CPU core. The CPU core after every checkpoint step calculates the checksum of the chunk data content. To reduce the checksum overheads further, a two bit prediction mechanism is used to predict if checksum calculation and chunk copy needs to be done.

Two bit prediction-based on checksums. Initially, all checkpoint variables are assumed to be non-redundant (see figure 4). Every chunk has a two bit prediction field which indicates a prediction value if the chunk is redundant or not. After every checkpoint is complete, a background thread first extracts the checksum of a chunk. Then, it compares the checksum with the previous checksum value. If the values are different, then it indicates the chunk has been modified, and the predicted value is incremented from redundant to likely redundant state. With further modification, the chunk state is incremented to unlikely redundant and finally not redundant. This two-bit prediction field is incremented/decremented for every checkpoint step. Only chunks which are in committing stages and unlikely redundant are committed. The algorithm III describes the steps.

Algorithm 1 Algorithm for eliminating redundancy

```
                                  ▷ %Main application thread:%
if current iteration == checkpoint step then
    Synchronize
    for i = 0 to GPUChunks do
        if (chunk is not committed due to pre-copy)
(chunk.state != redundant or != likely redundant) then
            Move data from GDRAM to DRAM to NVM
        end if
    end for
end if
                                  ▷ %Background application thread:%
for i = 0 to GPUChunks do
    Get the DRAM shadow pointer of each chunk
    Get the previous content hash
    Generate MD5 hash and compare checksums
    If checksums same, decrement chunk state bit
end for
```

Avoiding checksum calculation via dynamic instrumentation. An alternative to performing the checksum calculation on the CPU is to use dynamic instrumentation of GPU kernels to identify variables that get modified in each iteration, and mark them as "dirty". GPU Lynx [18], [19] is a dynamic instrumentation engine for GPGPU applications, which enables the creation of customized, user-defined instrumentation routines that can be applied transparently at run-time for a variety of purposes, including the facilitation of incremental checkpointing for GPUs. With GPU Lynx, it is possible to write instrumentations that are applied at the basic-block and/or instruction-level of a GPU kernel.

For GPU checkpointing, an instruction-level instrumentation can be provided such that on every global memory store operation, the dirty bit for the corresponding variable is updated. In the case of GPU kernels, all global memory load and store instructions correspond to specific kernel arguments passed into the function. Prior to executing the GPU kernel, a data-flow analysis pass can be performed to link each global memory store instruction with its corresponding kernel parameter, which represents the variable modified by that operation. The dirty bit identifier of the corresponding variable is then set to one on reaching a global memory store instruction. After kernel execution, the dirty bit vector is read from the GPU, and is checked to determine which variables have been modified in the current iteration.

Although by definition dynamic instrumentation incurs run-

time overheads, this is overall a low overhead instrumentation, requiring only a global memory update to a single bit-vector on every global memory store instruction. The data required to store the results from this instrumentation is also minimal, possibly no more than a single integer, depending on the number of variables that need to be monitored. We believe that the dynamic instrumentation approach will significantly reduce overall checkpointing overheads by obviating the need for checksum calculation altogether, albeit at the cost of slightly increasing kernel runtime overheads due to the instrumented code. We have not implemented this scheme, but plan on adding it as part of our future work.

IV. Implementation

The key goal of this work is to develop a GPU checkpointing library that understands CUDA-based GPU applications, provides a flexible interface for GPU applications and performs GPU specific optimization. For supporting GPU-based checkpointing, we extend our own NVMCheckpoint library [2] for CPUs to support fault tolerance for GPU-CPU-based application using NVM as a memory device.

Emulating NVM. Next generation NVMs like PCM are not currently available, and hence need to be emulated. To emulate NVM, we use one of the DRAM sockets in a dual socket machine. We use our NVM user and kernel manager designed for checkpoints from our prior work. Briefly, user-level manager provides heap-based interfaces by using a persistent allocator and manages application chunk information. The persistent user-level allocator is extended from the scalable JEMalloc allocator. The allocator is responsible for interacting with the NVM kernel manager using a virtual memory-based $mmap()$ interface. The kernel manager is responsible for managing all NVM pages in a socket, keeping track of pages allocated by an application and recovering the pages across restarts. The kernel internally maintains process level persistence metadata in a red-black tree loaded into kernel after restart/recovery. Our kernel design does not use the current filesystem implementation. The kernel implementation provides session level persistence by locking pages to avoid pages getting reclaimed after a process exists, and for restarts across failures, both user and kernel structures are stored to a SSD. We next describe the GPU checkpoint interface and implementation details. More details about NVM library can be found in [2] and [17].

GPU checkpoint interfaces. Table I shows the set of interfaces for GPU specific checkpoints. Applications are modified to use $chk\_CudaAlloc$ for checkpointing device chunks that are needed when restarting from failure. A variation of our implementation supports interposing all current CUDA allocation methods without requiring to use custom allocation methods, at the cost of checkpointing all GPU allocated variables that may not be used for restart. Each allocation method uses an additional 'varname' parameter for naming and identifying the chunk during failure recovery. The library creates and performs bookkeeping of all such chunk structures. Applications use the $chckpt\_all()$ interface that requires no arguments as a hint to begin a checkpoint.

Benchmark	Chekpt Size(MB)	Num Chunks	Pre-Copy Chunks	Unmodified chunks
BarnesHut	1096.02	14	4	1
Rodinia/particle filter	903.04	13	4	5
Rodinia/SRAD	896.112	12	2	6

TABLE II
Benchmark characteristics

The underlying checkpoint library identifies all checkpointing chunks, copies them from GPU to CPU, and then from CPU to NV storage. The $chckpt\_validate\_chksm()$ interface starts a background checksum validation thread and $chckpt\_hint()$ interface provides support for pre-copy (requires developer-provided hints of variables that can be pre-copied).

V. Evaluation

The overall goals of our evaluations are to understand the benefits and implications of various GPU checkpointing optimizations that we discussed previously. We use a mix of different standard GPU benchmarks and evaluate them to

1) Understand the impact of checkpointing in application performance and analyze the end-end checkpoint bandwidth constraints.
2) Discuss benefits of our pre-copy and redundant data elimination techniques.
3) What-if-analysis: We discuss the implications of our optimizations when NVM bandwidth is the same as the GPU to host data movement bandwidth.

Experimental setup and applications. All the experiments were conducted on a cluster with TeslaM2090 "Fermi" GPU cards. The peak GDRAM-DRAM bandwidth is 6-7 GB/sec when measured using the CUDA benchmarks. The experiments use one node, each consisting of 23 2.8 GHz Intel Xeon cores, 48 GB of DRAM memory. We model our experiments using a NVMs bandwidth of 2 GB/sec. We use the LANL memory copy benchmark and add corresponding delays to model NVM bandwidth for checkpoints (delays in copying checkpoint data from GDRAM to NVM). We use three benchmark applications to evaluate the proposed mechanism. The applications were consciously selected from different benchmark suites because our optimizations like data pre-copy and redundant data elimination are application centric. The application runtime and checkpoint data size is almost similar as summarized in Table II.

End-End checkpoint performance. First, we evaluate the end to end checkpoint performance. We use all three application described earlier. All benchmarks are compared against the baseline no-checkpoint case. As it can be seen in Figure 5, overall our pre-copy and checksum methods perform well.

As indicated in Table II, for SRAD almost half of the chunks do not get modified at all across checkpoints. These are mostly read only chunks which include chunks used as input. While the pre-copy (PCP) mechanism performs better than the no pre-copy case (No PCP), the checksum method actively detects unmodified chunks with prediction and avoids checkpointing in addition to the PCP. This leads to a direct reduction of checkpoint time. To verify, we keep the frequency of checkpoints constant (approx. 15 sec) and increase the checkpoint size per interval. Increasing data size increases the

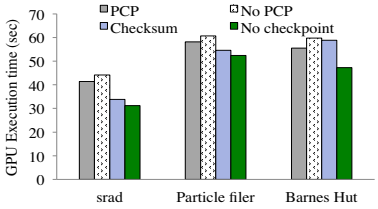

Fig. 5. Checkpoint end-end Performance

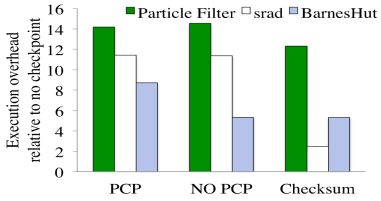

Fig. 6. What-if-analysis: GDRAM- DRAM performance

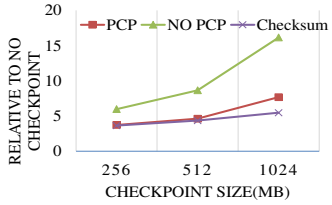

Fig. 7. Particle filter-checkpoint size vs. runtime

checksum gains by avoiding checkpoints for redundant chunks and hence checkpoint overhead is less than 5%. Overall, SRAD showed more than 60% benefits. We believe, with binary instrumentation, we can further reduce the checksum cost. Similar benefits of checksum were observed for the particle filter application. Just using pre-copy provided only a small benefit. Figure 7 compares the gains of checksum computation for different data sizes. For data size 256MB and 512MB, both PCP and checksum provided relatively same benefits over no pre-copy, whereas for larger data size, checksum provided close to 12% reduction in checkpoint overhead. While in this work we analyze a single node checkpoint performance, the impact of such optimizations can be substantial in a large scale experiment running across several nodes for a coordinated checkpoint. Our future work would focus on understanding the implications in a larger scale. The last benchmark 'BarnesHut' shows that, while the pre-copy approach performs well, the checksum approach performs poorly and provides almost no benefits. For BarnesHut (see Table II), almost all checkpoint variables change across every checkpoint interval, whereas chunks which can be pre-copied are relatively high. Using a pre-copy + checksum increased the overall overhead, in part due to the synchronization operations on accesses to shared memory between the main application thread and the checksum thread. The cost of such synchronization is higher compared to the actual checksum benefits, and leads to performance degradation.

What-if-Analysis: GDRAM-DRAM checkpoint performance. Figure 6 evaluates the performance of our optimizations if NVM and DRAM bandwidths are same (6-7 GB/sec). We modify the checksum mechanism such that, only for the first few iterations, checksum of variables is calculated. We assume a static application behavior across checkpoint intervals and stop checkpointing if the variables are redundant. As seen in the figure, using 'pre-copy' only approach is not always beneficial for our scale as checkpoint size is comparatively lesser compared to the peak bandwidth (less than 10%). When the checkpoint size is larger than the available bandwidth, pre-copy would perform better. When using pre-copy with checksum, the performance improves for the SRAD (around 5x), BarnesHut (1.5x), and in the case of the particle filter, there is not much benefit.

VI. Conclusions and Future Work

In this work we discuss mainly reducing the end-end bandwidth limitation of GPU checkpoint and use of persistent memory-specific optimizations to improve such performance. While our techniques show promising benefits for the benchmark applications, we are yet to evaluate our methods for a large scale applications like LAMMPS to understand the real benefits and implications. Further, we would like to study the impact of our techniques for MPI-based CUDA applications and multi-GPU applications, where kernel migrations are common. With a dynamic GPU instrumentation framework [18], [19], we could characterize the entire GPU kernel behavior and detect precisely the data that has been modified by the kernel, applying our pre-copy and redundant data elimination techniques on top of such an infrastructure. We plan to investigate this further next.

Acknowledgment

This research is supported in part by an Intel award for research on non-volatile memory and by the DOE Exact Center for Exascale Simulation.

References

[1] "PIConGPU: Many-GPGPU Particle-in-Cell Code," picongpu.hzdr.de/.
[2] S. Kannan, A. Gavrilovska, K. Schwan, and D. Milojicic, "Optimizing checkpoints using nvm as virtual memory," ser. IPDPS '13.
[3] J. Duell, "The design and implementation of berkeley labs linux checkpoint/restart," Tech. Rep., 2003.
[4] R. Gioiosa, J. C. Sancho, S. Jiang, F. Petrini, and K. Davis, "Transparent, incremental checkpointing at kernel level: a foundation for fault tolerance for parallel computers," in *SC '05*.
[5] A. Moody, G. Bronevetsky, K. Mohror, and B. R. d. Supinski, "Design, modeling, and evaluation of a scalable multi-level checkpointing system," ser. SC '10.
[6] K. B. Ferreira, R. Riesen, R. Brighwell, P. Bridges, and D. Arnold, "Libhashckpt: Hash-based incremental checkpointing using gpu's," ser. EuroMPI'11.
[7] "Lammps benchmark," http://lammps.sandia.gov/.
[8] X. Dong, Y. Xie, N. Muralimanohar, and N. P. Jouppi, "Hybrid checkpointing using emerging nonvolatile memories for future exascale systems," *ACM Trans. Archit. Code Optim.*
[9] X. Xu, Y. Lin, T. Tang, and Y. Lin, "Hial-ckpt: A hierarchical application-level checkpointing for cpu-gpu systems," in *ICCSE '10*.
[10] S. Laosooksathit, N. Naksinehaboon, and C. Leangsuksan, "Two-level checkpoint/restart modeling for gpgpu," ser. AICCSA '11.
[11] H. Takizawa, K. Sato, K. Komatsu, and H. Kobayashi, "Checuda: A checkpoint/restart tool for cuda applications," ser. PDCAT '09.
[12] L. Solano-Quinde, B. Bode, and A. Somani, "Coarse grain computation-communication overlap for efficient application-level checkpointing for gpus," ser. EIT '10.
[13] L. Bautista-Gomez, S. Tsuboi, D. Komatitsch, F. Cappello, N. Maruyama, and S. Matsuoka, "Fti: high performance fault tolerance interface for hybrid systems," ser. SC '11.
[14] S. Agarwal, R. Garg, M. S. Gupta, and J. E. Moreira, "Adaptive incremental checkpointing for massively parallel systems," ser. ICS '04.
[15] X. Dong, N. Muralimanohar, N. Jouppi, R. Kaufmann, and Y. Xie, "Leveraging 3d pcram technologies to reduce checkpoint overhead for future exascale systems," ser. SC '09.
[16] D. Li, J. S. Vetter, G. Marin, C. McCurdy, C. Cira, Z. Liu, and W. Yu, "Identifying opportunities for byte-addressable non-volatile memory in extreme-scale scientic applications," ser. IPDPS '12.
[17] S. Kannan, A. Gavrilovska, and K. Schwan, "Reducing the cost of persistence for nonvolatile heaps in end user devices," ser. HPCA '14.
[18] N. Farooqui, A. Kerr, G. Diamos, S. Yalamanchili, and K. Schwan, "A framework for dynamically instrumenting gpu compute applications within gpu ocelot," ser. GPGPU-4, 2011.
[19] Farooqui, Naila, "Lynx Library," https://code.google.com/p/gpulynx/.