CRUM: Checkpoint-Restart Support for CUDA's Unified Memory

Rohan Garg*
Northeastern U.
Boston, USA
rohgarg@ccs.neu.edu

Apoorve Mohan*
Northeastern U.
Boston, USA
apoorve@ccs.neu.edu

Michael Sullivan
NVIDIA Corp.
Santa Clara, USA
misullivan@nvidia.com

Gene Cooperman*†
Northeastern U.
Boston, USA
gene@ccs.neu.edu

***Abstract*—Unified Virtual Memory (UVM) was recently introduced on recent NVIDIA GPUs. Through software and hardware support, UVM provides a coherent shared memory across the entire heterogeneous node, migrating data as appropriate. The older CUDA programming style is akin to older large-memory UNIX applications which used to directly load and unload memory segments. Newer CUDA programs have started taking advantage of UVM for the same reasons of superior programmability that UNIX applications long ago switched to assuming the presence of virtual memory. Therefore, checkpointing of UVM will become increasingly important, especially as NVIDIA CUDA continues to gain wider popularity: 87 of the top 500 supercomputers in the latest listings are GPU-accelerated, with a current trend of ten additional GPU-based supercomputers each year.**

A new scalable checkpointing mechanism, CRUM (Checkpoint-Restart for Unified Memory), is demonstrated for hybrid CUDA/MPI computations across multiple computer nodes. CRUM supports a fast, forked checkpointing, which mostly overlaps the CUDA computation with storage of the checkpoint image in stable storage. The runtime overhead of using CRUM is 6% on average, and the time for forked checkpointing is seen to be a factor of up to 40 times less than traditional, synchronous checkpointing.

***Index Terms*—CUDA, GPU, UVM, checkpoint-restart, DMTCP**

I. Introduction

The advent of virtual memory automated the task of managing a program's memory segments. Hence, for large, complex programs, the use of virtual memory becomes *more efficient in practice*, since few programmers wish to spend development time manually squeezing out the most efficient memory management. In much the same way, NVIDIA has introduced *Unified Virtual Memory* (UVM) into their recent GPUs. CUDA UVM is analogous to the virtual memory with hardware support found on traditional computers.

UVM is especially important for workloads with memory footprints that are too large to entirely fit in device memory. In this case, UVM allows the application to allocate its data within a UVM region that is directly visible to a kernel running on the device. A "working set" of memory is automatically paged into the device as needed. Furthermore, the use of a unified virtual address space enables deployment of complex data structures for GPU-based computation, with the same pointers being valid on the host as well as on the GPU.

*This work was partially supported by NSF Grants ACI-1440788 and OAC-1740218.

†This work was partially supported by Grant 2014-345 from a "Chaire d'attractivité" de l'IDEX, Université Fédérale Toulouse Midi-Pyrénées.

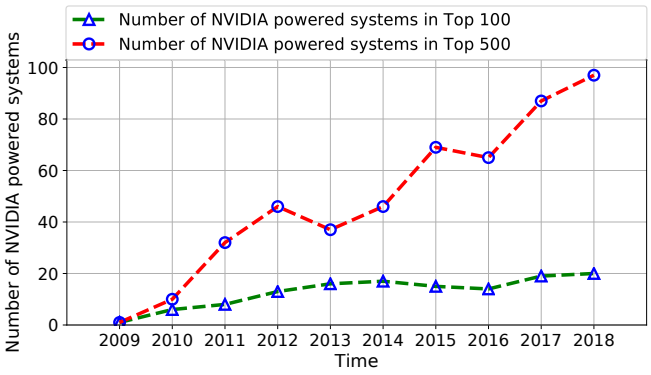


Fig. 1: NVIDIA GPUs in Top 500 list.

The use of GPUs continues to grow as seen in recent TOP-500 lists [1] (see Figure 1), and the advent of a unified shared address space is expected to further lower the entry barrier and widen the adoption of GPUs in HPC systems.

Unfortunately, GPUs have been shown to suffer from a high rate of *Detected Unrecoverable Errors* (DUEs) [2]–[7]. The mean time between failures (MTBF) is expected to become much worse as the number of compute nodes increases in the exascale generation.

Thus, efficient checkpointing for the UVM model is important for the future exascale generation. Unfortunately, previous checkpointing research [8]–[13] assumes the older (non-UVM) memory model.

A naïve approach to support checkpoint-restart would be to: (a) introspect and save the application process state (including the CUDA user-space library) and the GPU device driver; and (b) restore the process memory (including the CUDA user-space library) and restore the GPU device driver state. Unfortunately, the CUDA user-space library, which is checkpointed and restored as part of the process memory, is non-reentrant. Thus, it cannot restore the GPU device driver state.

To address these challenges, this paper proposes a novel framework, CRUM (Checkpoint-Restart for Unified Memory), which decouples the application process state from the device driver state (see Section III) by using a proxy process. Thus, CRUM can transparently checkpoint the application without involving any active driver state. (This could potentially allow a CUDA application to be checkpointed on one version of CUDA and GPU hardware, and restarted on another CUDA/GPU version.)

To optimize checkpointing of applications with large mem-

ory footprints, CRUM uses fork-based, copy-on-write mechanism. There are two phases. The first, and relatively fast, phase is the transfer of data resident on the GPU hardware to the application process through a proxy process. In the second phase, the application process disconnects from the proxy and forks a child process that writes the checkpoint data to stable storage. Meanwhile, the application process re-connects to the proxy, which resumes using the GPU for computation.

This work makes the following two novel contributions:

1) An algorithm for *shadow page synchronization* (see Algorithm 1), which ensures the isolation of an application process from the GPU device, while allowing the UVM memory regions to be shared between the two; and
2) A *forked checkpointing* model for UVM memory that overlaps writing a checkpoint image to stable storage while the application continues. This was difficult previously due to the need to share memory between the GPU device and host (UVM), and simultaneously between parent and forked child process.

Experimental results show that CRUM provides an effective and scalable approach for checkpoint-restart of real-world, high-performance computing workloads that take advantage of CUDA 8's UVM (Section IV). These hybrid CUDA/MPI applications include the DOE benchmarks HPGMG-FV and HYPRE. An average runtime overhead of 6% was observed. Further, CRUM's fast, forked checkpointing reduces the time to checkpoint up to a factor of 40 times less than a traditional checkpoint that writes out process memory to stable storage. CRUM is open source software that will be freely available.

Section II presents the background and motivation, including both the need for UVM support and the need for greater GPU reliability as we approach the exascale generation. Section III describes the design of CRUM, while Section IV presents an experimental evaluation. Section V presents an analysis of the current limitations of the current approach, and the potential impact on future generations of NVIDIA GPUs. Finally, Section VI describes the related work, and Section VII presents the conclusion.

II. Background and Motivation

A. History and Motivation for Unified Virtual Memory (UVM)

Unified Virtual Memory (UVM) and its predecessor, Unified Virtual Addressing (UVA), are major CUDA features that are incompatible with prior CUDA checkpointing approaches. Yet, UVM is an important innovation for future CUDA applications.

Through software and hardware support, UVM provides a coherent shared memory across the entire heterogeneous node [14], [15]. The use of UVM-managed memory greatly simplifies data sharing and movement among multiple GPUs. This is especially useful given that the most energy-efficient supercomputers place multiple compute accelerators per node—for instance, TSUBAME3.0 [16], Coral Summit [17], and the NVIDIA SATURNV [18] supercomputer use 4, 6, and 8 GPUs per node, respectively. The features and progression of UVM are briefly described below.

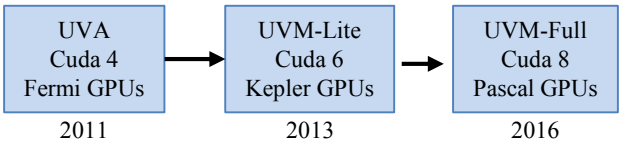


Fig. 2: The technology advancement of CUDA unified virtual memory.

Historically, in CUDA 4 (2011), Fermi-class GPUs added support for Unified Virtual Addressing (UVA) with *zero-copy memory*. UVA allows transparent zero-copy accesses to memory across a heterogeneous node using a partitioned address space. UVA never migrates data, and so non-local memory accesses suffer from less bandwidth and longer latency.

To reduce the performance penalty of non-local zero-copy memory accesses, first-generation Unified Virtual Memory (UVM-Lite) was introduced in CUDA 6 (2013) for Kepler-class GPUs [19]. UVM-Lite shares a single memory space across a heterogeneous node, and it transparently migrates all memory pages that are attached to the CUDA streams associated with each kernel. This simplifies deep copies with pointer-based structures and it allows GPUs to transparently migrate UVM-managed memory to the device, nearly achieving the performance of CUDA programs using explicit memory management. Due to hardware restrictions, however, UVM-Lite does not allow concurrent access to the same memory from both CPU and GPU—host-side access is only allowed once all GPU-side accesses to a CUDA stream have completed. Concurrent access to UVM-managed memory from different GPUs is allowed, but data are never migrated between devices and non-local memory is accessed in a zero-copy fashion.

Second-generation UVM (UVM-Full) was introduced in CUDA 8 (2016) for Pascal-class GPUs [20]. It eliminates the concurrent-access constraints of the prior UVM generation and adds support for system-wide atomic memory operations, providing an unrestricted coherent shared memory across the heterogeneous node. On-demand data migration is supported by UVM-Full across all CPUs and GPUs in a node, with the placement of any piece of data being determined by a variety of heuristics [15].

Pascal-era UVM also adds support for memory over-subscription, meaning that UVM-managed regions that are larger than the GPU device memory can be accessed without explicit data movement. This is important for applications with large data. In particular, it greatly simplifies the programming of large-memory jobs, and avoids the need to explicitly marshal data to and from the GPU [21]. For instance, GPU-capacity-exceeding deep neural network training has been accomplished in the past through explicit data movement [22], but it can also be performed with less programmer effort by UVM over-subscription [23].

B. GPUs for Exascale: DUEs and GPU Reliability

The advantages of using GPUs for high-performance computing have been realized and a steep rise in their use in large-scale HPC systems has been observed (see Figure 1). Eighty-seven (87) systems in the Top500 list were reported to be

powered by NVIDIA GPUs in November 2017, as compared to one (1) in November 2009 [1]. Thus, it is important that both hardware and the software stack (pertaining to the use of GPUs) should be highly available and reliable to maximize large-scale HPC systems productivity.

While this makes GPUs attractive for exascale computing, the high GPU detectable-uncorrectable error rate (as compared to CPUs) remains an issue. Checkpointing plays an important role in mediating this issue. Various studies have been conducted for understanding the reliability aspects of using GPU's in large-scale HPC systems. The studies suggest that the newer generation GPU's are more reliable, as are the large-scale HPC systems using them (i.e., the observed MTBF of systems using newer GPU's is much longer than their estimated MTBF) [2]–[7].

However, one factor that motivates efficient checkpoint-restart on GPU accelerated systems is that GPU memory currently tends to have more DUEs (Detected Unrecoverable Errors) per GB than CPU memory. Memory in CPU nodes is composed of narrow 4-bit or 8-bit wide DRAM devices that are grouped together into DIMMs, meaning certain ECC codes (often called chipkill ECC) can correct the data that comes from an entire DRAM device. In contrast, GPU memory is much wider (32-bit wide for GDDR5/GDDR5X and 128-bit for HBM2) such that chipkill-level protection is not possible without a prohibitively large memory access granularity; accordingly, current GPUs use single-bit correcting SEC-DED ECC for DRAM [24], [25]. These lesser correction capabilities lead to a relative increase in detected errors. For example, a field study of the Blue Waters system [26] found that the DUE rate per GB of Kepler-era GDDR5 was roughly 5 times that of the chipkill-protected CPU memory.

Given the high rate of DUEs expected in the future exascale systems, checkpoints will be more frequent, and so it is imperative to design checkpointing mechanisms that can reduce the time that applications spend in checkpointing.

C. Checkpointing Large-memory CUDA-UVM Applications

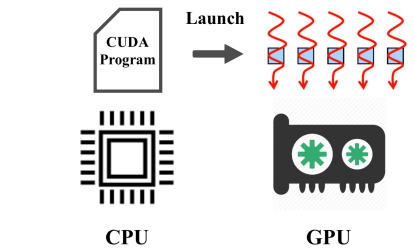
UVM acts as an enabler for easily developing large-memory CUDA applications. UVM enables a GPU to transparently access host CPU and remote GPU memory, and hence solves the problem of otherwise manually managing data transfers. All of the host CPU's memory is available, on-demand, by the GPU device. Conversely, all of the UVM memory on the GPU device is available to the CPU.

In this situation, the CUDA application may use much more memory than is present on the device. The capacity of GPU memory is currently from 16 to 32 GB for a high-end GPU, while CPU memory often ranges from 128 to 256 GB. In the past, this forced GPU application developer to choose between: scaling out to many nodes and GPUs (hence incurring communication overhead); or manually managing the data transfers on a single GPU. Later, UVM made possible a third choice: transparently transferring data on a single GPU via UVM. However, the ease of developing such large-memory CUDA-UVM applications now places a larger burden on transparent checkpointing to support this large-memory overhead.

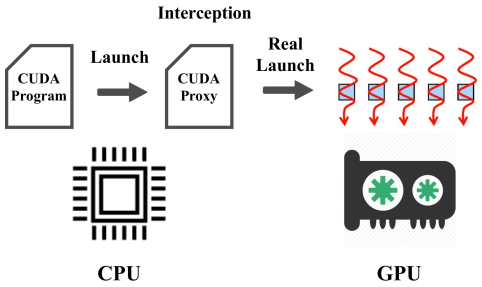
III. CRUM: Design and Implementation

To address the challenges described in Section II, this paper proposes CRUM, a novel framework that provides a checkpointing-based fault-tolerance mechanism. CRUM enables transparent, system-level checkpointing for CUDA and CUDA UVM applications.

Figure 3 shows a high-level schematic of CRUM's architecture. Note especially the organization into two processes: a CUDA program (the user's application), and a CUDA proxy (the only process that uses the CUDA library to communicate with the GPU). The flow of control is: (i) to interpose on CUDA library calls made by the application process; (ii) to forward the requests to the proxy process; (iii) which then executes the calls via its CUDA library and GPU, on behalf of the application; and (iv) finally returns the results back to the application.



(a) CUDA Original



(b) CUDA Proxy

Fig. 3: High-level architecture of CRUM

In this section, we present the key subsystems in the design of CRUM. The first research challenge is the propagation of UVM memory pages (already shared between GPU hardware and proxy process) to make them visible to the application process. Section III-B describes a shadow page scheme (summarized in Algorithm 1) for this purpose. The second research challenge is to extend this scheme to overlap checkpointing and computation for the sake of fast, forked checkpoint and future exascale needs. This is discussed in Section III-C. Finally, the implementation details of integrating CRUM with proxy processes is discussed in III-D.

A. Post-CUDA 4: The Need for a Proxy Process

Ideally, a single-process approach toward checkpointing seems simpler. But this approach for CUDA became non-viable with CUDA 4 and beyond, when NVIDIA implemented unified virtual addressing with zero-copy, an antecedent of unified memory [23]). At that point, it was no longer possible to re-initialize the CUDA library at the time of restart. We assume that this is due to the lack of clear semantics about what it means to re-initialize a CUDA library that still retains pointers to unified memory regions on host and device. One must choose either to free the host memory (thus sabotaging any CUDA application that retains a pointer to the unified memory region), or else to leave the host memory region intact (thus sabotaging any application assumptions about unification of host and device memory). Note that a fresh restart will restore all host memory, but any unification of host with device memory has already been lost.

The core issue is that the CUDA unified memory model was developed for standard CUDA applications — and naturally did not include extensions for transparent checkpointing. An alternative workaround would have been, at restart time, to overwrite the text and data memory segments of any CUDA libraries with a fresh, uninitialized CUDA library (matching a freshly booted GPU), and then to call `cudaInit()`. Unfortunately, the CUDA library/driver appeared to have additional state, which made this workaround infeasible.

B. Shadow Pages for the Support of UVM

Recall the use of a proxy process, as seen in Figure 3(b). The core research challenge in this architecture is that UVM dictates that pages are transparently shared between the GPU hardware and the proxy process, but these shared UVM pages are not visible to the application process.

The zero-copy memory of CUDA 4 implies that there are no CUDA calls on which to interpose. In direct-mapped memory, the device may read or write to the host mapped pinned memory of the proxy process at any time. But the separate application process remains unaware of modifications to memory in the proxy process. Thus, an approach using CUDA proxies is unable to support the newer and potentially more efficient zero-copy memory for UVA. To overcome this situation, a new, transparent checkpointing approach for CUDA's zero-copy memory is proposed, in which proxy and application reflect a single application with two "personalities".

The CUDA application process and the CUDA proxy process invoke the same application binary but execute two different state machines. The application process goes through three different states: CUDA call, read from device-mapped UVM memory, write to device-mapped UVM memory. Note that the state transitions are not dictated by the CRUM framework, but rather by the application logic. On the other hand, the CUDA proxy process is simply a passive listener for requests from the application process and executes the CUDA calls and the memory reads and writes as dictated by the application.

Based on these observations, we introduce the concept of "shadow UVM pages". For every CUDA UVM allocation request by the application, CRUM creates a corresponding shadow UVM region in the context of the application process. At the same time, the CUDA proxy process requests for a "real" UVM region from the device driver. The two processes, the *application* and the *proxy*, see two different views of the memory and data at any given point.

Since there are no API calls to interpose on, this opens up the requirement for tracking the changes to the application process's memory in order to keep the two sets of pages in sync. CRUM relies on the use of user-space page-fault tracking to accomplish this. There are currently two available mechanisms for page-fault tracking in Linux: `userfaultfd`; and segfault handler and page protection bits. While there are certain performance benefits with the use of `userfaultfd`, the current work uses segfault handler and page protection bits to allow for evaluation on clusters employing older Linux kernels.

The algorithm for synchronizing the data on shadow and real UVM pages is described in Algorithm 1.

Algorithm 1 Shadow page synchronization algorithm

```
upon event Page Fault do
    if addr ∈ AllShadowPages then
        if isReadFault() then
            ReadDataFromRealPage()
        else
            MarkPageAsDirty()
        end if
    end if
upon event CUDA call do
    if hasDirtyPages then
        SendDataToRealPages()
        ClearDirtyPages()
    end if
upon event CUDA Create UVM region do
    uvmAddr ← CreateUvmRegionOnProxy()
    reg ← CreateShadowPage(uvmAddr)
    AllShadowPages ← AllShadowPages ∪ reg
```

When an application process requests for a new UVM region, a new shadow UVM region is created in the process's memory (using the `mmap` system call). The shadow UVM region is given read-write permissions initially, and all the pages in the regions are marked as "dirty".

When the application makes a CUDA call where the device could potentially read or modify the UVM data (for example, a CUDA kernel launch), the data from dirty pages is "flushed" to the real UVM pages on the proxy process, the dirty flag is cleared for the UVM region, and the read-write permissions are removed (using the `mprotect` system call).

This allows CRUM to interpose on either a read or write to unified memory. Standard Linux code for segfault handlers allows CRUM to detect an attempt to read or to write, and to distinguish the two cases. In the case of a read, PROT_READ permission is set for all of the memory in the application

process corresponding to unified memory. In the case of a write, PROT_WRITE permission is set for all of the memory in the application process corresponding to unified memory. (See Section III-B1 for further discussion.)

At a later time, when the application process tries to read the results of the GPU computation back from the shadow UVM regions, a read page fault is triggered; the permissions of the shadow UVM region are changed to read-only, and the results are read in from the corresponding real UVM region on the proxy.

1) Page permissions on Linux: Note that write to shadow UVM memory region requires PROT_WRITE permission. Unfortunately, on Linux, PROT_WRITE permission implies PROT_READ permission also. Linux does not support *write-only* permission, but rather *read-write* permission instead.

This has consequences for the three-state algorithm to support unified memory in CRUM. We make the assumption here that most applications will cycle through the three states in order (possibly omitting the read-only or write-only phase). Hence, a typical cycle would be invoked: CUDA-call/read-unified-memory/write-unified-memory.

In fact, CRUM also supports overlapped execution of a CUDA call with reading and writing unified memory. The essential assumption is that read access must precede write access and a read-write cycle cannot be followed with a second read unless there is an intervening CUDA kernel. Normal CUDA calls such as `cudaMemcpy` are allowed at all times.

As discussed earlier, unfortunately, Linux's write-only permission for memory actually grants read-write permission. It is for this reason that a transition from write-unified-memory directly to read-unified-memory cannot be detected efficiently. Possible solutions are discussed at the end of this section.

This assumption has been found to hold in each of the real-world applications that we have found for testing CRUM with unified memory. Nevertheless, it is important to also build in a (possibly slower) *verified execution mode* that will test an application to see if it violates this assumed cycle of CUDA-call/read-unified-memory/write-unified-memory.

There is more than one way to implement a verified execution mode.

One of the difficulties is that a Linux segfault handler does not allow us to reset the page permission to allow only the pending read or write, and then reset the permission back to PROT_NONE. Linux's user-space page fault handling, `userfaultfd`, introduced with Linux 4.3, can fix this, but that introduces other technical difficulties. (For example, it was only with Linux 4.11 that this was extended partially to support fork and shared memory.) Another alternative is to parse the pending read or write (load or store assembly instruction), temporarily allow read-write permission to the desired memory page, and then use the parsed information to read or write the data between register and memory, and finally to restore the previous memory permission. This might be more efficient than user-space page faulting since it might have fewer transitions to a kernel system call.

Linux kernel modification to support write-only permissions for UVM shadow pages is another possibility.

C. Fast, Forked Checkpoints

UVM enables CUDA applications to use all of the host and GPU device memory transparently. This can make checkpointing, which is dominated by the time to write to the disk, prohibitively expensive. So while one could employ copy-on-write-based asynchronous checkpointing, UVM memory is incompatible with shared memory and fork on Linux.

Fortunately, CRUM's proxy-based architecture can be used to address this challenge. Note that the device state and the UVM memory regions are not directly a part of the application process's context, but rather they are associated with the proxy process. This frees up the application process to use forked checkpointing for copy-on-write-based associated checkpointing for the application process.

Forked checkpointing allows CRUM to invoke a minimal checkpointing delay in order to "drain the GPU device" of its data, after which, a child process of each MPI process can write to stable storage. This allows the system to overlap the CUDA computation with storage of the checkpoint image in stable storage.

D. Checkpoint-Restart Methodology and Integration with Proxies

Finally, for completeness, we discuss how CRUM integrates proxy concepts into the CUDA implementation requirements. Proxies have also been used by previous authors (see Section VI-d).

At checkpoint time, CRUM suspends the user application threads, and "drains" the GPU kernel queue. It issues a device synchronize call (`cudaDeviceSynchronize`) to ensure that the kernels have finished execution and the memory state is consistent. Then, for all the active CUDA-MALLOC and CUDA-UVM memory regions, data is read in from the GPU to the host. The data is first transferred from the GPU into the proxy process's memory, and then from the memory of the proxy process into the memory of the user application process. The user application process then disconnects from the proxy process. This ensures that the problem reduces to the trivial problem of checkpointing a single-process application. Finally, the state of the process is saved to a checkpoint image file on stable storage.

At the time of restart, CRUM starts a new process and recreates the user application threads. Then, the memory of the new process gets replaced by the saved state from the checkpoint image file. CRUM, then, starts a new proxy process, which starts a new GPU context. It recreates the active CUDA-MALLOC and CUDA-UVM memory regions by replaying the allocation calls. CUDA streams and events are similarly handled. (See Section V for further discussion.) Finally, CRUM transfers the data into the actual CUDA and CUDA-UVM regions through the proxy process and resumes the application threads.

IV. Experimental Evaluation

The goal of this section is to present a detailed analysis of the performance of CRUM. In particular, this section answers the following questions:

Q1 *What's the overhead of running a CUDA (or a CUDA UVM) application under CRUM?*

Q2 *Does CRUM provide the ability to checkpoint CUDA (and CUDA UVM) applications?*

Q3 *Can CRUM improve a CUDA UVM based application's throughput by reducing the checkpointing overhead?*

Q4 *Is the approach scalable?*

A. Setup

To answer the above questions, we first briefly describe our experimental setup and methodology.

1) Hardware: The experiments were run on a local cluster with 4 nodes. Each node is equipped with 4 NVIDIA PCIe-attached Tesla P100 GPU devices, each with 16 GB of RAM. The host machine is running a 16-core Intel Xeon E5-2698 v3 (2.30 GHz) processor with 256 GB of RAM. Each node runs CentOS-7.3 with Linux kernel version 3.10.

2) Software: Each GPU runs NVIDIA CUDA version 8.0.44 with driver 396.26. Experiments use DMTCP [27] version 3.0. We developed a CRUM-specific DMTCP plugin [28] for checkpoint-restart of NVIDIA CUDA UVM applications.

The DMTCP CRUM plugin (referred to as the CRUM plugin from here onwards) interposes on the CUDA calls made by the application. The interposition code is generated in a semi-automated way, where a user specifies the prototype of a CUDA function, and whether the call needs to be logged. This not only allows us to cover the extensive CUDA API, but also allows for ease of maintainability and for future CUDA extensions.

The plugin forwards the requests, over a SysV shared memory region, to a proxy process running on the same node. The forwarded request is then executed by the proxy process, which then returns the results back to the application. To improve the performance, we use well-studied concepts from pipelining of requests, to allow the application to send requests without blocking. Blocking requests, such as, `cudaDeviceSynchronize`, result in a pipeline flush. For data transfers (both for UVM shadow page data and for `cudaMalloc` data) we use Linux's Cross Memory Attach (CMA) to allow for data transfers using a single copy operation.

3) Application Benchmarks: We use Rodinia 3.1 [29] benchmarks for evaluating CRUM for CUDA applications. Note that the Rodinia benchmarks do *not* use UVM, and can be run even with CUDA 2.x. They are included here to show comparability of the new approach with the older work from 2011 and earlier using CUDA 2.x [10], [12], [30].

We note that CheCUDA [10] does not work for modern CUDA (i.e., CUDA version 4 and above) because it relies on a single-process checkpoint-restart approach. CheCL [30] only supports OpenCL and does not work with CUDA. We tried compiling the CRCUDA [13] version available online [31], but it failed to compile with CUDA version 8. It didn't work for the benchmarks used in our experiments, after applying our compilation fixes.

To evaluate CRUM using UVM-managed memory allocation, we run a GPU-accelerated build of two DOE benchmarks: a high-performance geometric multigrid proxy application (HPGMG-FV [32]), and a test application using a production linear system solver library (HYPRE [33]). For the HYPRE library, we run the test driver for an unstructured matrix interface using the AMG-PCG solver. For HPGMG-FV, we evaluate two versions: the standard HPGMG-FV benchmark with one grid (the *master* branch, as described in [34]), and an AMR proxy modification with multiple AMR levels (the *amr_proxy* branch, as described in [21]).

We focus on HPGMG-FV and HYPRE because they are scientific applications and libraries with potential importance in future exascale computing [35], and they have publicly available ports to UVM-enabled multi-GPU CUDA. HPGMG-FV has also been used as a benchmark for ranking the speeds of the top supercomputers [36].

To evaluate the relative performance of HPGMG-FV runs, we quote its throughput in degrees-of-freedom per second — the same metric used to rank supercomputer speeds [36]. Thus, larger numbers indicate higher performance. To evaluate the relative performance of HYPRE runs, we measure the wall clock time taken by each program execution.

B. Runtime Overhead

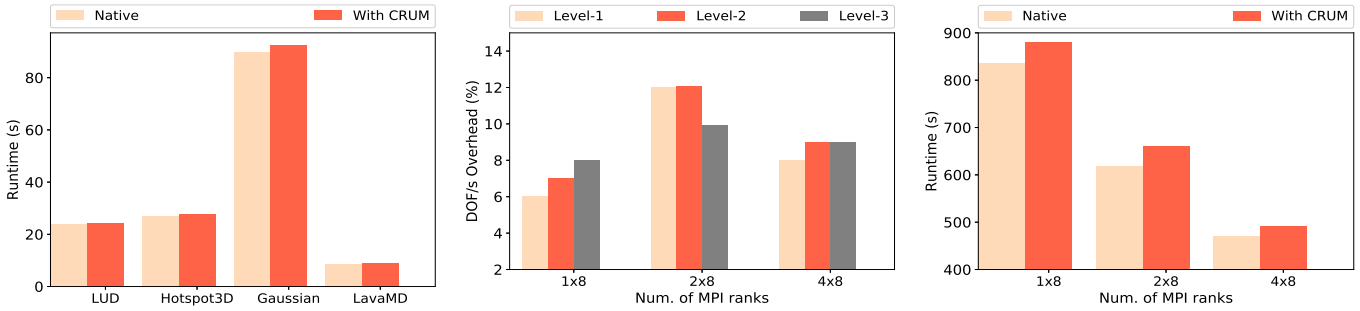
While the ability to checkpoint is important for improving the throughput of an application on a system with frequent failures, a checkpointing system that imposes excessive runtime overhead can render the framework ineffective, and in the worst case, reduce the throughput. We, therefore, benchmark and analyze the sources of runtime overhead. For these experiments, no checkpoint or restart was invoked during the run of the application.

The results demonstrate that CRUM is able to run the CUDA application with a worst case overhead of 12%, and a 6% overhead on average. We note that this is a prototype implementation and a production system could incorporate many optimizations to further reduce the overhead.

TABLE I: Runtime parameters for Rodinia applications.

Application	Configuration Parameter
LUD	"-s 2048 -v"
Hotspot3D	"512 8 1000 power_512x8 temp_512x8"
Gaussian	"-s 8192"
LavaMD	"-boxes1d 40"

Figure 4(a) shows the runtimes for four applications (LUD, Hotspot3D, Gaussian, and LavaMD) from the Rodinia benchmark suite with and without CRUM. The applications mostly use the CUDA API's from CUDA 2.x: `cudaMalloc`, `cudaMemcpy`, and `cudaLaunch`. Table I shows the configuration parameters used for the experiments. We observe that the runtime overhead varies from 1% (for LUD) to 3% (in the case of LavaMD). The runtime overhead is dominated by the



(a) Rodinia Benchmark. (b) HPGMG-FV Benchmark. (c) HYPRE Benchmark.

Fig. 4: Runtime overheads for different benchmarks under CRUM.

cost of data transfers from the application process to the proxy process. In a different experiment, using Unix domain sockets for data transfer, we observed overheads varying from 1.5% to 16.5%. The use of CMA reduces the overhead significantly.

Figure 4(b) shows the runtime results for the HPGMG-FV benchmark with increasing number of nodes and MPI ranks. As noted in Section IV-A3, we use the HPGMG-FV throughput metric DOF/s as a proxy for performance. We note that the DOF/s reported by the application running under CRUM are less than the native numbers by 6% to 12%. We present a more in-depth analysis below.

In our experiments, we observed that a single MPI rank of the HPGMG-FV benchmark runs about 9 million CUDA kernels during its runtime of 3 minutes. This implies that each CUDA kernel runs for approximately 20 microseconds on average. Note that the cost of executing a `cudaLaunch` call itself can be up to 5 microseconds. The program allocates many CUDA UVM regions, sets up the data, and runs a series of kernels to operate on the data. Each MPI rank then exchanges the results with its neighbors. While the size of the UVM regions vary from 12 KB to 128 KB, the frequent reads and writes the application process, stresses the CRUM framework in two dimensions: (a) frequent interrupts and data transfer; and (b) frequent context switches and the need to synchronize with proxy process (because of the many CUDA calls that need to be executed).

While the use of CMA (cross-memory attach) reduces the cost of data transfers, interestingly, we observed a lot of variability in the cost of a single CMA operation for the same data transfer size. The cost of a single page transfer varies from 1 microsecond to 1 millisecond, a difference of three orders of magnitude. We attribute this to two sources: (a) O/S jitter; (b) the pre-fetching algorithm employed by the UVM driver. In many cases, reading a UVM page is slowed down because of a previous read on a large UVM region, spanning several pages, because the driver gets busy pre-fetching the data for the large UVM region.

To address the second source of overhead, we optimized the CRUM implementation to: (a) use a lock-free, inter-process synchronization mechanism over shared-memory; and (b) pipeline non-blocking CUDA calls from the application. A CUDA call, such has `cudaLaunch`, `cudaMemsetAsync`, is pipelined and the application is allowed to move ahead in its execution, while the proxy finishes servicing the request. At a synchronization point, like `cudaDeviceSynchronize`, the application must wait for a pipeline flush, i.e., for the pending requests to be completed.

Figure 4(c) shows the runtimes for the HYPRE benchmark for a different number of MPI ranks running on a varying number of nodes. The benchmark observes up to 6.6% overhead when running under CRUM compared to native execution.

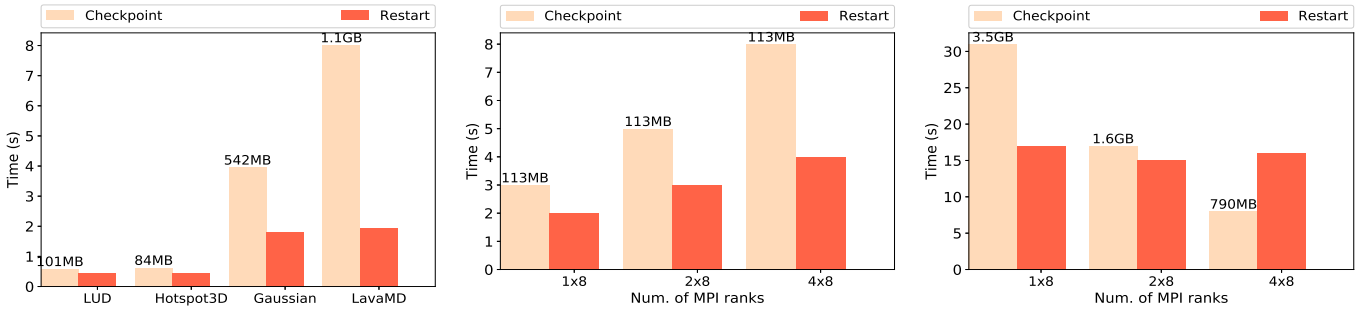
The HYPRE benchmark presents different checkpointing challenges than HPGMG-FV. While the HYPRE benchmark invokes only about 100 CUDA kernels per second (10 milliseconds on average per kernel) during its execution, it uses many large UVM regions (up to 900 MB). Thus, the overhead is dominated by the cost of data transfers between the application process and the proxy.

In addition to CMA, CRUM employs a simple heuristic to help reduce the data transfer overhead. For small shadow UVM regions, it reads in all of the data from the real UVM pages on the proxy. However, for a read fault on a large shadow UVM region, it starts off by only reading the data for just one page containing the faulting address. On subsequent read faults on the same region, while in the read phase (see Section III), we exponentially increase (by powers of 2) the number of pages read in from the real UVM region on the proxy. This heuristic relies on the spatial and temporal locality of accesses. While there will be pathological cases where an application does "seemingly" random reads from different UVM regions, we have found this assumption to be valid in the two applications we tested.

C. Checkpointing CUDA Applications: Rodinia and MPI

Next, we evaluate the ability of CRUM to provide fault tolerance for CUDA and CUDA UVM applications using checkpoint-restart.

Figure 5(a) shows the checkpoint times, restart times, and the checkpoint image sizes for the four applications from the Rodinia benchmark suite. The checkpointing overhead is dependent on the time to transfer the data from the device memory to the host memory, then transferring it from the



(a) Rodinia Benchmark. (b) HPGMG-FV Benchmark. (c) HYPRE Benchmark.

Fig. 5: Checkpoint-restart times and checkpoint image sizes for different benchmarks under CRUM.

proxy process to the application process using CMA, and then finally writing to the disk. We observe that the time to write dominates the checkpointing time.

Figure 5(b) shows the checkpoint times, restart times, and the checkpoint image sizes for HPGMG. The results are shown with increasing number of MPI ranks (and the number the nodes). We observe that as the total amount of checkpointing data increases from 904 MB (8×113 MB) to 3.6 GB (32×113 MB), the checkpoint time increases from 3 seconds to 8 seconds. We attribute the small checkpoint times to the buffer cache on Linux. We observed that forcing the files to be synced (by using an explicit call to `fsync` increased the checkpoint times by up to 3 times.

The results for HYPRE are shown in Figure 5(c). The application divides a fixed amount of data (approx. 28 GB in total) equally among its ranks. So, we observe that the checkpoint image size reduces by almost half every time we double the number of ranks. This helps improve the checkpoint cost especially with smaller process sizes, as the Linux buffer caches the writes, and the checkpoint times reduce from 31 seconds (for 8 ranks on 1 node) to 8 seconds (for 32 ranks over 4 nodes).

D. Reducing the Checkpointing Overhead: A Synthetic Benchmark for a Single GPU

To showcase the benefits of using CRUM to reduce checkpointing overhead for CUDA UVM applications, we develop a CUDA UVM synthetic benchmark. The synthetic benchmark allocates two vectors of 2^{32} 4-byte floating point numbers (32 GB in total) and computes the dot product of the two vectors. The floating point numbers are generated at random. Note that the total memory requirements are double of what is available on the GPU device (16 GB). However, UVM allows an application to use more than the available memory on the GPU device. The host memory, in this case, acts as "swap storage" for the device and the pages are migrated to the device or to the host on demand.

Table II shows the checkpoint times for three different cases: (a) using a naïve checkpointing approach; (b) using three different compression schemes, Gzip, Parallel Gzip, and LZ4, before writing to the disk; and (c) using CRUM's forked checkpointing approach. The first two approaches, naïve and compression, use CRUM's CUDA UVM checkpointing framework. The third approach adds the forked checkpointing optimization to the base CUDA UVM checkpointing framework. The three compression schemes use Gzip's lowest compression level (`-1` flag). While parallel Gzip uses the same compression algorithm as Gzip, it launches as many threads as the number of cores on a node to compress input data.

TABLE II: Checkpoint times using different strategies for the synthetic benchmark.

Strategy	Ckpt Time	Ckpt Size	Data Migration Time
Naïve	45 s	33 GB (100% random)	4 s
Gzip	1296 s	29 GB (100% random)	4 s
Parallel gzip	86 s	29 GB (100% random)	4 s
LZ4	62 s	33 GB (100% random)	4 s
Forked Ckpting	4.1 s	32 GB (100% random)	4 s
Gzip	749 s	15 GB (50% random)	4 s
Parallel gzip	56 s	15 GB (50% random)	4 s
LZ4	45 s	17 GB (50% random)	4 s

We observe that the forked checkpointing approach outperforms the other two approaches by up to three orders of magnitude. Since the program uses random floating point numbers, compression is ineffective at reducing the size of the checkpointing data (Table II). We note that the time taken by the compression algorithm is also correlated with the randomness of data. As an experiment, we introduced redundancy in the two input vectors to improve the "compressibility". Of the 2^{32} floating point elements in a vector, only half (2^{16}) of the elements were generated randomly and the rest were assigned the same floating point number. This improves the compression time and reduces the checkpoint time to 749 seconds and the checkpoint image size is reduced to 15 GB by using the Gzip-based strategy.

Note that parallel Gzip may not be a practical option in many HPC scenarios, where an application often uses one MPI rank per core on a node. On the other hand, LZ4 provides a computationally fast compression algorithm at the cost of a lower compression ratio.

E. Reducing the Checkpoint Overhead: Real-world MPI Applications

Finally, we present the results from using CRUM with the forked checkpointing optimization for the real-world CUDA UVM application benchmarks. The results reported here correspond to the largest scale of 4 CPU nodes, with 16 GPU devices, running 8 MPI ranks per node (32 processes in total).

TABLE III: Checkpoint times using different strategies for real-world CUDA UVM applications. The numbers reported corresponds to running 32 MPI ranks over 4 nodes. The checkpoint size reported is for each MPI rank. The checkpoint times are normalized to the time for the naïve checkpointing approach (1x).

App.	Strategy	Ckpt Time	Ckpt Size
HPGMG-FV	Gzip	0.78x	14 MB
HPGMG-FV	Parallel gzip	0.60x	14 MB
HPGMG-FV	LZ4	0.30x	16 MB
HPGMG-FV	Forked ckpting	0.025x	113 MB
HYPRE	Gzip	2x	176 MB
HYPRE	Parallel gzip	1x	176 MB
HYPRE	LZ4	1x	296 MB
HYPRE	Forked ckpting	0.032x	868 MB

Table III shows the results for checkpointing time (and checkpoint image sizes) normalized to the checkpointing time using the naïve checkpointing approach (as shown in Figures 5(b) and 5(c)). The results are shown for HPGMG-FV and HYPRE.

We observe trends similar to the synthetic benchmark case. While in the naïve checkpointing approach, the checkpointing overhead is dominated by the cost of I/O, i.e., writing the data to the disk, under forked checkpointing, the overhead is dominated by the cost of in-memory data transfers: from the GPU to the proxy process, and from the proxy process's address space to the application process's address space. Further, the cost of quiescing the application process, quiescing the network (for MPI), and "draining" and saving the in-flight network messages is 0.01% of the total cost.

However, unlike the synthetic benchmark, using in-memory compression to reduce the size of data for writing is better in this case for both HPGMG and HYPRE. This indicates that the compression algorithm is able to efficiently reduce the size of the data, which helps lower the I/O overhead. Note that this is still worse than using forked checkpointing by an order of magnitude.

V. Discussion

Driver support for restart: In order to restart a computation, CRUM must re-allocate memory in the same locations as during the original execution—otherwise the correctness of pointer-based code cannot be guaranteed during re-execution. The current CRUM prototype relies on deterministic CUDA memory allocation, which we verify to work with the CUDA driver libraries via experimentation (for both explicit device memory and UVM-managed memory allocation). The assumption of deterministic memory re-allocation is shared by previous GPU checkpointing efforts [12].

Memory Overhead: In a CUDA program with large data resident on the host, the memory overhead due to an additional proxy process could be a concern. In the special case of asynchronous checkpointing, the overhead could be even higher, although copy-on-write does prevent it from going too high. This could be ameliorated by future support for shared memory UVM pages between application and a proxy running CUDA.

Advanced CUDA language features: Dynamic parallelism allows CUDA kernels to recurse and launch nested work; it is supported by CRUM without change. Device-side memory allocation allows kernel code to allocate and de-allocate memory. It is partially supported by CRUM, with one important distinction—no live device-side allocations are allowed at a checkpoint time. Thus, device-side memory allocations are to be freed before the system is considered quiesced and ready for a checkpoint. We do not anticipate this constraint to be particularly difficult to satisfy, since device-side mallocs tend to be used to store temporary thread-local state within a single kernel, whereas host-side CUDA memory allocation (which is supported by CRUM without restriction) is more often used for persistent storage.

Using `mprotect`: Currently, in a Linux kernel, `PROT_WRITE` protection for a memory region implies read-write memory permission rather than write-only memory permission. Because of this, some compromises were needed in the implementation. This work has demonstrated the practical advantages of a write-only memory permission for ordinary Linux virtual memory. It is hoped that in the future, the kernel developers at NVIDIA will be encouraged to support write-only memory permission for this purpose.

Another issue with an mprotect-based approach is that when kernel-space code page faults on a read/write protected page, it returns an error code to the user, `EFAULT`, rather than a segfault. This forces the implementation to be extended to handle such failures; the implementation cannot rely solely on a segfault handler [37]–[40].

Other APIs and Languages: This work provides checkpoint-restart capabilities for programs written in C/C++ with the CUDA runtime library. In our experience, the CRUM prototype should support the majority of GPU-accelerated HPC workloads; however, there are other APIs to that may be valuable for some users. Given the current framework of code auto-generation for CRUM, we believe that it will be straightforward to extend the implementation to support other APIs, such as OpenACC. The ability of CRUM to support UVM-managed memory would be especially useful for OpenACC programs, as PGI's OpenACC compiler provides native and transparent support for high-performance UVM-managed programs, making UVM-accelerated OpenACC programs a low-design-effort route to performant GPU acceleration [41].

Future Versions of CUDA: Just as prior checkpointing methods for GPUs were unable to cope with versions of CUDA since CUDA 4 (released in 2011), it is likely that CRUM will need to be updated to support language features after CUDA 8. One such development is Heterogeneous

Memory Management (HMM) [42], which is a kernel feature introduced in Linux 4.14 that removes the need for explicit cudaMallocManaged calls (or use of the __managed__ keyword) to denote UVM-managed data. Rather, with HMM the GPU is able to access any program state, including the entire stack and heap. Because the current CRUM prototype relies on wrapping cudaMallocManaged calls, it will need to be redesigned to support HMM.

VI. Related Work

a) Use of proxy process: Zandy et al. [43] demonstrated the use of a "shadow" process for checkpointing currently running application processes that were not originally linked with a checkpointing library. This allows the application process to continue to access its kernel resources, such as open files, via RPC calls with the shadow process.

Kharbutli et al. [44] use a proxy process for isolation of heap accesses by a process and for containment of attacks to the heap.

b) GPU virtualization: A large number of previous HPC studies have focused on virtualizing the access to the GPU [8]–[10], [12], [13], [30], [45], [46]. Here we describe some of those studies, with an emphasis on the use for GPU checkpointing and GPU-as-a-Service in the cloud and HPC environments.

Lagar-Cavilla et al. [45], Shi et al. [8], Gupta et al. [9], and Giunta et al. [46] focus on providing access to the GPU for processes running in a virtual machine (VM), as an alternative to PCI pass-through. The access is provided by forwarding GPU calls to a proxy process that runs outside the VM and has direct access to the GPU.

c) GPU-as-a-Service: Two other efforts, DS-CUDA [47] and rCUDA [48], have focused on providing access to a remote GPU for the purposes of GPU-as-a-Service [49]–[55]. They also rely on a proxy process. Using the proxy process is similar to the one described in this work; however, the focus is on efficient remote access by using the InfiniBand's RDMA API. To the best of our knowledge, none of the previous studies solve the problem of efficient checkpointing of modern CUDA applications that use UVM. We note that the optimizations described in these works can be used in conjunction with CRUM for providing efficient access to remote GPUs.

d) GPU Checkpointing: Early work on virtualizing or checkpointing GPUs was based on CUDA 2.2 and earlier [8]–[12]. Those approaches stopped working with CUDA 4 (introduced in 2011), which introduced Unified Virtual Addressing (UVA). Presumably, it is the introduction of UVA that made it impossible to re-initialize CUDA 4.

In 2016, CRCUDA [13], employed a proxy-based approach, similar to the 2011 approach of CheCL [30] that targeted OpenCL [56] (as opposed to CUDA) for GPUs. OpenCL does not support unified memory, and so CheCL and CRCUDA do not support NVIDIA's unified memory [23] targeted here.

VOCL-FT [57] aims to provide resilience against soft errors. VOCL-FT leverages the OpenCL programming model to reduce the amount of data movement: both to/from the device from/to the host, and to/from the disk. This allows them to do fast checkpointing and recovery.

HiAL-Ckpt [58], HeteroCheckpoint [59], and cudaCR [60] use application-specific approaches for providing GPU checkpointing.

None of the approaches described above work for CUDA UVM. CRUM focuses on providing efficient runtime and checkpointing support for CUDA and CUDA-UVM based programs. We note that the techniques described in above approaches are complementary to CRUM and can be used to further optimize the runtime and checkpointing overheads.

VII. Conclusion

This paper introduced CRUM, a novel framework for checkpoint-restart for CUDA's unified memory. The framework employs a proxy-based architecture along with a novel shadow page synchronization mechanism to efficiently run and checkpoint CUDA UVM applications. Furthermore, the architecture enables fast, copy-on-write-based, asynchronous checkpointing for large-memory CUDA UVM applications. Evaluation results with a prototype implementation show that average runtime overhead imposed is less than 6%, while improving the checkpointing overhead by up to 40 times.

Acknowledgment

We would like to thank the reviewers for their constructive feedback. We thank Nikolay Sakharnykh from NVIDIA for sharing his knowledge of UVM programming, and the NVIDIA PSG Cluster for compute time. We also thank Kapil Arya for insightful discussions and his feedback on an earlier draft of the paper. And we wish to thank Onesphore Ndayishimiye for his early prototype of an auto-generator for communication between application and proxy process.

References

[1] TOP500, "TOP500 supercomputer sites," https://www.top500.org/, 2018.

[2] I. S. Haque and V. S. Pande, "Hard Data on Soft Errors: A Large-Scale Assessment of Real-World Error Rates in GPGPU," in *CCGRID*, May 2010.

[3] J. Y. Shi, M. Taifi, A. Khreishah, and J. Wu, "Sustainable GPU Computing at Scale," in *2011 14th IEEE International Conference on Computational Science and Engineering*, Aug 2011.

[4] N. DeBardeleben, S. Blanchard, L. Monroe, P. Romero, D. Grunau, C. Idler, and C. Wright, "GPU Behavior on a Large HPC Cluster," in *Euro-Par 2013: Parallel Processing Workshops*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014.

[5] D. Tiwari, S. Gupta, G. Gallarno, J. Rogers, and D. Maxwell, "Reliability Lessons Learned from GPU Experience with the Titan Supercomputer at Oak Ridge Leadership Computing Facility," in *SC*. New York, NY, USA: ACM, 2015. [Online]. Available: http://doi.acm.org/10.1145/2807591.2807666

[6] D. Tiwari, S. Gupta, J. Rogers, D. Maxwell, P. Rech, S. Vazhkudai, D. Oliveira, D. Londo, N. DeBardeleben, P. Navaux *et al.*, "Understanding GPU Errors on Large-scale HPC Systems and the Implications for System Design and Operation," in *HPCA*. IEEE, 2015.

[7] V. Sridharan, N. DeBardeleben, S. Blanchard, K. B. Ferreira, J. Stearley, J. Shalf, and S. Gurumurthi, "Memory Errors in Modern Systems: The Good, The Bad, and The Ugly," in *ASPLOS*. New York, NY, USA: ACM, 2015. [Online]. Available: http://doi.acm.org/10.1145/2694344.2694348

[8] L. Shi, H. Chen, and J. Sun, "vCUDA: GPU-accelerated High Performance Computing in Virtual Machines," in *Proceedings of the International Symposium on Parallel and Distributed Processing (IPDPS)*. IEEE, 2009.

[9] V. Gupta, A. Gavrilovska, K. Schwan, H. Kharche, N. Tolia, V. Talwar, and P. Ranganathan, "GViM: GPU-accelerated Virtual Machines," in *Proc. of the 3rd ACM Workshop on System-level Virtualization for High Performance Computing*. ACM, 2009.

[10] H. Takizawa, K. Sato, K. Komatsu, and H. Kobayashi, "CheCUDA: A Checkpoint/Restart Tool for CUDA Applications," in *Proceedings of the International Symposium on Parallel and Distributed Processing (IPDPS)*. IEEE, 2009.

[11] L. B. Gomez, A. Nukada, N. Maruyama, F. Cappello, and S. Matsuoka, "Transparent Low-overhead Checkpoint for GPU-accelerated Clusters," 2010, [Online; accessed 16-Mar-2018]. [Online]. Available: https://wiki.ncsa.illinois.edu/download/attachments/17630761/INRIA-UIUC-WS4-lbautista.pdf

[12] A. Nukada, H. Takizawa, and S. Matsuoka, "NVCR: A Transparent Checkpoint-Restart Library for NVIDIA CUDA," in *Proceedings of the International Symposium on Parallel and Distributed Processing Workshops and PhD Forum*. IEEE, 2011.

[13] T. Suzuki, A. Nukada, and S. Matsuoka, "Transparent Checkpoint and Restart Technology for CUDA Applications," GPU Technology Conference (GTC), 2016, [Online; accessed 17-Jan-2018]. [Online]. Available: https://tinyurl.com/ycb7y8xw

[14] NVIDIA, "CUDA C programming guide, appendix k: Unified memory programming," NVIDIA Developer Zone, 2017, pG-02829-001_v9.1 [Online; accessed 17-Jan-2018]. [Online]. Available: http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf

[15] M. Harris, "Unified memory for CUDA beginners," NVIDIA Blog, 2016, [Online; accessed 18-Jan-2018]. [Online]. Available: https://devblogs.nvidia.com/unified-memory-cuda-beginners/

[16] T. Trader, "TSUBAME3.0 points to future HPE Pascal-NVLink-OPA server," HPC Wire, 2017. [Online]. Available: https://www.hpcwire.com/2017/02/17/tsubame3-0-points-future-hpe-pascal-nvlink-opa-server/

[17] F. Foertter, "Preparing GPU-accelerated applications for the Summit supercomputer," GPU Technology Conference (GTC), May 2017. [Online]. Available: http://on-demand.gputechconf.com/gtc/2017/presentation/s7642-fernanda-foertter-preparing-gpu-accelerated-app.pdf

[18] R. Kim, "NVIDIA DGX SATURNV ranked world's most efficient supercomputer by wide margin," NVIDIA Blog, 2016. [Online]. Available: https://blogs.nvidia.com/blog/2016/11/14/dgx-saturnv/

[19] M. Harris, "Unified memory in CUDA 6," NVIDIA Blog, 2013, [Online; accessed 17-Jan-2018]. [Online]. Available: https://devblogs.nvidia.com/parallelforall/unified-memory-in-cuda-6/

[20] M. Harris, "CUDA 8 features revealed," NVIDIA Blog, 2016, [Online; accessed 17-Jan-2018]. [Online]. Available: https://devblogs.nvidia.com/parallelforall/cuda-8-features-revealed/

[21] N. Sakharnykh, "Beyond GPU memory limits with unified memory on Pascal," NVIDIA Blog, 2016, [Online; accessed 17-Jan-2018]. [Online]. Available: https://devblogs.nvidia.com/parallelforall/beyond-gpu-memory-limits-unified-memory-pascal/

[22] M. Rhu, N. Gimelshein, J. Clemons, A. Zulfiqar, and S. W. Keckler, "vDNN: Virtualized Deep Neural Networks for Scalable, Memory-Efficient Neural Network Design," in *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2016.

[23] N. Sakharnykh, "Unified memory on Pascal and Volta," GPU Technology Conference (GTC), 2017, [Online; accessed 17-Jan-2018]. [Online]. Available: http://on-demand.gputechconf.com/gtc/2017/presentation/s7285-nikolay-sakharnykh-unified-memory-on-pascal-and-volta.pdf

[24] "NVIDIA Tesla P100—the most advanced data center accelerator ever built," http://www.nvidia.com/object/pascal-architecture-whitepaper.html, 2016.

[25] D. A. Oliveira, P. Rech, L. L. Pilla, P. O. Navaux, and L. Carro, "GPGPUs ECC Efficiency and Efficacy," in *Proceedings of the International Symposium on Defect and Fault Tolerance in VLSI Systems (DFT)*, Oct. 2014.

[26] C. Di Martino, Z. Kalbarczyk, R. K. Iyer, F. Baccanico, J. Fullop, and W. Kramer, "Lessons Learned from the Analysis of System Failures at Petascale: The case of Blue Waters," in *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2014.

[27] J. Ansel, K. Arya, and G. Cooperman, "DMTCP: Transparent Checkpointing for Cluster Computations and the Desktop," in *Proceedings of the International Symposium on Parallel and Distributed Processing (IPDPS)*. IEEE, 2009.

[28] K. Arya, R. Garg, A. Y. Polyakov, and G. Cooperman, "Design and Implementation for Checkpointing of Distributed Resources using Process-level Virtualization," in *Proceedings of International Conference on Cluster Computing (CLUSTER)*. IEEE, 2016.

[29] S. Che, M. Boyer, J. Meng, D. Tarjan, J. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A Benchmark Suite for Heterogeneous Computing," in *Proceedings of the International Symposium on Workload Characterization*, 2009.

[30] H. Takizawa, K. Koyama, K. Sato, K. Komatsu, and H. Kobayashi, "CheCL: Transparent Checkpointing and Process Migration of OpenCL Applications," in *Proceedings of the International Symposium on Parallel and Distributed Processing (IPDPS)*. IEEE, 2011.

[31] T. Suzuki, A. Nukada, and S. Matsuoka, "CRCUDA Source," 2015, [Online; accessed 17-Jan-2018]. [Online]. Available: https://github.com/tbrand/CRCUDA

[32] L. B. N. L. (LBL), "HPGMG: High-Performance Geometric Multigrid," 2017, [Online, accessed 17-Jan-2018]. [Online]. Available: https://bitbucket.org/nsakharnykh/hpgmg-cuda

[33] L. L. N. L. (LLNL), "HYPRE: Scalable Linear Solvers and Multigrid Methods," 2017, [Online, accessed 17-Jan-2018]. [Online]. Available: https://github.com/LLNL/hypre

[34] N. Sakharnykh, "High-Performance Geometric Multi-Grid with GPU Acceleration," NVIDIA Blog, 2016, [Online; accessed 21-Jan-2018]. [Online]. Available: https://devblogs.nvidia.com/high-performance-geometric-multi-grid-gpu-acceleration/

[35] Kothe, Douglas B., "Exascale Applications: Opportunities and Challenges," Presentation to the Advanced Scientific Computing Advisory Committee (ASCAC), Sep. 2016, [Online; accessed 28-Mar-2018]. [Online]. Available: https://science.energy.gov/~/media/ascr/ascac/pdf/meetings/201609/Kothe_ExaApps_ASCAC_2016-09-20.pdf

[36] "High-performance Geometric Multigrid, an HPC Benchmark and Supercomputing Ranking Metric," 2016, [Online; accessed 28-Mar-2018]. [Online]. Available: https://hpgmg.org

[37] J. S. Plank, M. Beck, G. Kingsley, and K. Li, "Libckpt: Transparent Checkpointing Under Unix," in *Proceedings of the USENIX 1995 Technical Conference Proceedings*, ser. TCON'95. Berkeley, CA, USA: USENIX Association, 1995. [Online]. Available: http://dl.acm.org/citation.cfm?id=1267411.1267429

[38] J. F. Ruscio, M. A. Heffner, and S. Varadarajan, "DejaVu: Transparent User-Level Checkpointing, Migration, and Recovery for Distributed Systems," in *IPDPS*. IEEE, 2007.

[39] E. Bugnion, V. Chipounov, and G. Candea, "Lightweight Snapshots and System-Level Backtracking," in *Proceedings of the 14th Workshop on Hot Topics on Operating Systems*, no. EPFL-CONF-185945. USENIX, 2013.

[40] D. Vogt, C. Giuffrida, H. Bos, and A. S. Tanenbaum, "Lightweight Memory Checkpointing," in *DSN*. IEEE, 2015.

[41] N. Sakharnykh, "Combine OpenACC and Unified Memory for Productivity and Performance," NVIDIA Blog, 2015, [Online; accessed 21-Jan-2018]. [Online]. Available: https://devblogs.nvidia.com/combine-openacc-unified-memory-productivity-performance/

[42] J. Hubbard, "Using HMM to Blur the Lines between CPU and GPU Programming," GPU Technology Conference (GTC), May 2017. [Online]. Available: http://on-demand.gputechconf.com/gtc/2017/presentation/s7764_john-hubbardgpus-using-hmm-blur-the-lines-between-cpu-and-gpu.pdf

[43] V. C. Zandy, B. P. Miller, and M. Livny, "Process Hijacking," in *Proceedings of the International Symposium on High-Performance Parallel and Distributed Computing (HPDC)*. IEEE, 1999.

[44] M. Kharbutli, X. Jiang, Y. Solihin, G. Venkataramani, and M. Prvulovic, "Comprehensively and Efficiently Protecting the Heap," in *ACM Sigplan Notices*, vol. 41, no. 11. ACM, 2006.

[45] H. A. Lagar-Cavilla, N. Tolia, M. Satyanarayanan, and E. De Lara, "VMM-independent Graphics Acceleration," in *Proc. of the 3rd Int. Conf. on Virtual Execution Environments*. ACM, 2007.

[46] G. Giunta, R. Montella, G. Agrillo, and G. Coviello, "A GPGPU Transparent Virtualization Component for High Performance Computing Clouds," in *EUROPAR*. Springer, 2010.

[47] M. Oikawa, A. Kawai, K. Nomura, K. Yasuoka, K. Yoshikawa, and T. Narumi, "DS-CUDA: A Middleware to Use Many GPUs in the Cloud Environment," in *High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion:*. IEEE, 2012.

[48] J. Duato, A. J. Pea, F. Silla, R. Mayo, and E. S. Quintana-Ort, "rCUDA: Reducing the Number of GPU-based Accelerators in High Performance Clusters," in *2010 International Conference on High Performance Computing Simulation*, June 2010.

[49] C. Reaño, F. Silla, G. Shainer, and S. Schultz, "Local and Remote GPUs Perform Similar with EDR 100G InfiniBand," in *Proceedings of the Industrial Track of the 16th International Middleware Conference*. ACM, 2015.

[50] C. Reaño and F. Silla, "A Performance Comparison of CUDA Remote GPU Virtualization Frameworks," in *CLUSTER*. IEEE, 2015.

[51] B. Varghese, J. Prades, C. Reaño, and F. Silla, "Acceleration-as-a-Service: Exploiting Virtualised GPUs for a Financial Application," in *e-Science (e-Science), 2015 IEEE 11th International Conference on*. IEEE, 2015.

[52] F. Silla, J. Prades, S. Iserte, and C. Reaño, "Remote GPU Virtualization: Is It Useful?" in *High-Performance Interconnection Networks in the Exascale and Big-Data Era (HiPINEB), 2016 2nd IEEE International Workshop on*. IEEE, 2016.

[53] C. Reaño, F. Silla, D. Nikolopoulos, and B. Varghese, "Intra-node Memory Safe GPU Co-Scheduling," *IEEE Transactions on Parallel and Distributed Systems*, 2017.

[54] C. Reaño, F. Silla, and J. Duato, "Enhancing the rCUDA Remote GPU Virtualization Framework: From a Prototype to a Production Solution," in *CCGRID*, May 2017.

[55] J. Prades and F. Silla, "Turning GPUs into Floating Devices over the Cluster: The Beauty of GPU Migration," in *2017 46th International Conference on Parallel Processing Workshops (ICPPW)*, Aug 2017.

[56] J. E. Stone, D. Gohara, and G. Shi, "OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems," *Computing in Science and Engineering*, vol. 12, 2010.

[57] A. J. Peña, W. Bland, and P. Balaji, "VOCL-FT: Introducing Techniques for Efficient Soft Error Coprocessor Recovery," in *SC*, Nov 2015.

[58] X. Xu, Y. Lin, T. Tang, and Y. Lin, "Hial-Ckpt: A Hierarchical Application-level Checkpointing for CPU-GPU Hybrid Systems," in *2010 5th International Conference on Computer Science Education*, Aug 2010.

[59] S. Kannan, N. Farooqui, A. Gavrilovska, and K. Schwan, "HeteroCheckpoint: Efficient Checkpointing for Accelerator-based Systems," in *DSN*. IEEE, 2014.

[60] B. Pourghassemi and A. Chandramowlishwaran, "cudaCR: An In-Kernel Application-Level Checkpoint/Restart Scheme for CUDA-Enabled GPUs," in *CLUSTER*, Sept 2017.