CRAC: Checkpoint-Restart Architecture for CUDA with Streams and UVM

Twinkle Jain*
Khoury College of Computer Sciences
Northeastern University
Boston, USA
jain.t@northeastern.edu

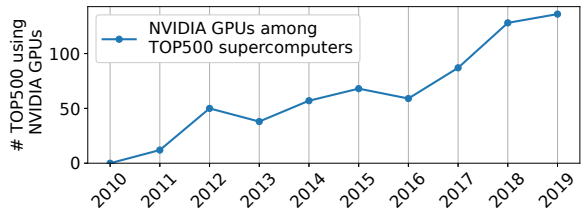
Gene Cooperman*
Khoury College of Computer Sciences
Northeastern University
Boston, USA
gene@ccs.neu.edu

***Abstract*—The share of the top 500 supercomputers with NVIDIA GPUs is now over 25% and continues to grow. While fault tolerance is a critical issue for supercomputing, there does not currently exist an efficient, scalable solution for CUDA applications on NVIDIA GPUs. CRAC (Checkpoint-Restart Architecture for CUDA) is a new checkpoint-restart solution for fault tolerance that supports the full range of CUDA applications. CRAC combines: low runtime overhead (approximately 1% or less); fast checkpoint-restart; support for scalable CUDA streams (for efficient usage of all of the thousands of GPU cores); and support for the full features of Unified Virtual Memory (eliminating the programmer's burden of migrating memory between device and host). CRAC achieves its flexible architecture by segregating application code (checkpointed) and its external GPU communication via non-reentrant CUDA libraries (not checkpointed) within a single process's memory. This eliminates the high overhead of inter-process communication in earlier approaches, and has fewer limitations.**

***Keywords*—Checkpointing, CUDA, Unified Virtual Memory, Parallel Processing, Split Processes**

I. Introduction

General-purpose GPU computing continues to become more important in supercomputers and in large- and medium-size clusters. For example, starting from zero GPUs in 2010, the number of clusters with NVIDIA GPUs has reached 136 out of 500 in the Nov., 2019 listing of the top 500 supercomputers [1], as seen in the next graph.



This work introduces CRAC (Checkpoint-Restart Architecture for CUDA) for transparently checkpointing CUDA on GPUs. Transparent checkpointing for CPUs (as opposed to GPUs) has long been important in long-running computations. Transparent checkpointing is widely available for Linux HPC applications, including MPI. Three notable examples of transparent checkpointing are DMTCP [2] (multi-host and MPI), BLCR [3] (single-host and MPI), and CRIU [4] (primarily for single-host). However, that ability to transparently checkpoint computations using GPUs has been notably lacking.

Transparent checkpointing is important in HPC for at least four reasons:

(a) long-running batch jobs that might need more time to complete than the typical 24-hour job allocation slot;
(b) fault tolerance (especially concerning GPU soft errors);
(c) backfill policies for efficient scheduling of batch queues; and
(d) process migration in the cloud, for example to exploit spot instances in the cloud for cost-effective computing [5], and for other just-in-time strategies.

The ability to checkpoint GPUs is even more pressing as clusters and supercomputers continue to scale to an increased number of GPUs. This is because of the vulnerability of GPUs to soft errors. A series of papers in the literature has highlighted the issue of fault tolerance for GPUs in the presence of soft errors [6]–[9]. In particular, NVIDIA GPUs do not have the same level of error protection of RAM as is the case for the high-end host computers used in clusters.

Finally, *transparent checkpointing* (as opposed to application-specific checkpointing) is especially important in order to relieve the application developer of the burden of coding for checkpointing. There are several anecdotes in the community of long-standing computational toolkits that "used to" have an application-specific checkpointing module, but that specialized module gradually became out-of-date as additional stateful parameters were added to a model.

Further, application-specific checkpointing typically has limitations, in that a checkpoint may be taken only at each iteration of the outermost loop. This is done in order to avoid the complication of restoring the stack as it existed at runtime. These limitations imply that application-specific checkpointing is often incompatible with *on-demand checkpointing*, which is required in the case of spot instances, or when a large high-priority job arrives and existing jobs must immediately be checkpointed.

Ironically, while the need for transparent checkpointing of GPUs has grown in the last decade, the support for transparent checkpointing of GPUs has diminished. A series of results for transparent checkpointing of GPU [10]–[14] have stopped

*This work was partially supported by National Science Foundation Grant OAC-1740218 and a grant from Intel Corporation.

SC20, November 9-19, 2020, Is Everywhere We Are
978-1-7281-9998-6/20/$31.00 ©2020 IEEE

working as of CUDA 4.0. This is because CUDA 4.0 introduced, in 2011, UVA (Unified Virtual Addressing between host and GPU device). This was later refined, with CUDA 6.0, to UVM (Unified Virtual Memory). All previous checkpointing efforts relied on the ability to save and restore the CUDA library in memory. But now that the virtual memory address space is shared between GPU device and host, any attempt to restore the checkpointed CUDA library and associated allocated memory at their original address will create inconsistencies between the host and GPU device address space.

Two more recent efforts at checkpointing (CRCUDA [15] and CRUM [16]) try to get around this problem by creating separate proxy processes. CRCUDA presents a preliminary attempt whose overhead was apparently never evaluated on real-world programs. CRCUDA's github repo [17] has not been active since 2015. CRUM presents a more complete solution, but it continues to have limitations.

The problem with both CRCUDA and CRUM is that their approach centers around passing all CUDA calls from the application process to a CUDA library resident in an independent proxy process. This requires copying buffers between the application process and proxy process before and after each CUDA library call. This has three inherent problems:

(a) Copying buffers creates a high runtime overhead. Modern CUDA applications may need to launch 1,000 CUDA kernels per second and more. (CRUM reports 6% to 12% overhead [16, (Section IV.B, figure 4(b))].)
(b) CRUM's support for UVM is incomplete. The issue is that UVM allows for hardware-supported page faults between host and device whenever one or the other updates the memory in a unified page. CRUM is limited to supporting applications that follow this pattern: CUDA-call, read from UVM, modify, write to UVM, next CUDA-call. Not all applications follow this pattern. See CUDA call [16, (Section III.B)] for details.
(c) Neither CRCUDA nor CRUM appear to have been tested in checkpointing the maximum permitted number of concurrent CUDA streams. We speculate that the reason is that both approaches sustained a significant overhead in making a CUDA call, since this required copying memory buffers (arguments to the CUDA call) to an independent proxy process. The essence of using CUDA Streams is to execute multiple CUDA kernels simultaneously (in multiple streams). This parallelism implies a higher frequency of CUDA kernel calls, placing more stress on the memory transfers to the proxy process.

In summary, this work makes three important contributions that may be summarized as (a) low runtime overhead, (b) efficient support for UVM, and (c) efficient support for many concurrent CUDA streams. More explicitly:

1) ***Low runtime overhead:*** Previous checkpointing support for CUDA 4.0 and later had unacceptably high runtime overhead (for example, CRUM's 6% to 12% [16]). The single-address space approach of this work enables more efficient, direct passing of pointers to CUDA kernels upon launch. While doing this, it retains isolation of the CUDA application program from the helper (proxy) program that "talks" to the GPU.
2) ***Efficient and complete UVM support:*** There are no compromises in the UVM support. CRUM's shadow page synchronization restricts UVM-based applications solely to a single read-modify-write cycle between CUDA kernel launches [16, Section III-B]. Further, CRUM's strategy fails when two concurrent CUDA streams write to the same memory page.
3) ***Many concurrent CUDA streams:*** The new approach scales well with many concurrent CUDA streams. The lack of previous experiments in the literature for more than two concurrent CUDA streams confirms the novelty of this work's support for many concurrent streams.

Finally, CRAC is free and open-source software. The current version of CRAC is found at: https://github.com/DMTCP-CRAC/CRAC-early-development.git. In the future, the newest version of CRAC will be included as a plugin in the mainstream DMTCP [18], which is open source.

In the remainder of this work, Section II describes the approach of three previous systems for transparent checkpointing of CUDA: CheCUDA (basic approach), and CRCUDA, and CRUM (proxy-based approaches). It also describes the deficiencies of those systems for use in HPC. Section III describes the new single address-space approach of CRAC. Section IV presents experimental results demonstrating the performance and generality of the new approach. Section V then describes the related work. Section VI presents the conclusion and future work.

II. Background

We highlight the history of CUDA and earlier approaches to transparently checkpoint CUDA applications. This highlights why older approaches stopped working with the introduction of CUDA-4.0, and a conceptually new approach was required.

A. The Historical Evolution of CUDA

As described in the introduction, previous mechanisms for transparent checkpointing [10]–[14] were made incompatible by the introduction of Unified Virtual Addressing (UVA) in CUDA 4.0. UVA was introduced in CUDA-4.0, and was later refined into Unified Virtual Memory (UVM) in CUDA 6.0. UVM operates in analogy with the introduction of virtual memory for UNIX. The CUDA UVM-enabled hardware and software execute on-demand paging, so that application programmers don't need to explicitly swap memory segments in and out of the GPU device. CUDA streams were introduced with CUDA-3.0 (Fermi GPUs).

B. A First Attempt at Checkpoint-restart: CheCUDA prior to CUDA 4.0

Here, we describe the architecture of CheCUDA [12], built upon CCUDA-2.2 in 2009, as representative of the general approach. The basic steps are: (a) to "drain the queue" of tasks (of pending CUDA kernels) using

`cudaDeviceSynchronize` or `cuCtxSynchronize`; (b) to copy persistent GPU state associated with resources held by the CUDA library to host memory; (c) to destroy all CUDA resources; (d) to checkpoint on the host side using BLCR [3]; and to restart by reversing these steps. Creation of CUDA resources is recorded prior to checkpoint time, and then restored during restart in a classic log-and-replay strategy.

A problem was encountered with CheCUDA and related approaches for checkpointing GPUs [10]–[14] in 2011. This is the year when NVIDIA introduced one more CUDA resource as part of the CUDA 4.0 library: the unified virtual address (UVA) facility. CUDA did not provide an API to save the state of UVA and later restore it. This was not surprising, since the UVA resource is shared between device and host, and so it would be difficult to provide a user API to restore it. Previous CUDA resources were resident solely on the GPU.

C. A Second Attempt at Checkpoint-restart: Proxy-based solutions for CUDA 4.0 and later

In 2011, CUDA 4.0 introduced UVA (Unified Virtual Addressing) [19]. CUDA 6.0 then introduced UVM (Unified Virtual Memory) in 2013 [20], exacerbating further the difficulty of saving and restoring UVA or UVM state. UVM on Pascal and later GPUs supports hardware page faulting of host pages into the GPU and vice versa.

CUDA memory allocations were then a resource that could no longer be saved and restored, since a memory allocation included a virtual memory mapping between host and device. That mapping is managed by the NVIDIA portion of the operating system, and it was not exposed to the CUDA programmer.

To overcome this, CRCUDA [15] and CRUM [16] took a proxy-based approach. But CRCUDA doesn't support UVA or UVM. CRUM supports UVM through *shadow memory* [16, Algorithm 1], but at the cost of high runtime performance, and covering only standard CUDA applications following the read-modify-cudaCall pattern.

III. The Design and Implementation of CRAC

CRAC provides the ability to save and restore the state of CUDA by first using CUDA-specific save/restore operations, and then delegating to a traditional checkpoint-restart package. Conceptually, CRAC could have used any of the three most popular systems for transparent checkpointing: BLCR [3], CRIU [4], and DMTCP [2]. However, CRIU does not support checkpointing of multiple hosts and BLCR is no longer actively maintained. In the end, the support of DMTCP for process virtualization and plugins [21] makes it easier to add modular support for CUDA without having to excessively understand details of the internals of the host checkpointing package. Further, DMTCP remains the only transparent checkpointing package to operate at petascale, as originally demonstrated in 2016 [22], when it was used to checkpoint two petascale computations: MPI-based HPCG (using 32,752 cores) and MPI-based NAMD (using 16,368 cores) [22].

The discussion of CRAC is next split into two parts: design and implementation.

A. The Design of CRAC

The problems with previous approaches to transparently checkpointing CUDA using proxies were highlighted in the introduction and Section II: high runtime overhead due to inter-process communication; and the difficulty of supporting certain newer CUDA resources (e.g., UVA/UVM and multiple CUDA streams) when the CUDA API did not expose a mechanism for easily saving and restoring those resources.

The inter-process communication bottleneck between a CUDA application process and a proxy process is an essential bottleneck of CRCUDA and CRUM. CRAC's solution is to combine the application and proxy into a single process, whose address space contains *two independent programs, each with their own text segment, data, heap, and runtime libraries*. The application program and a proxy program are loaded separately into the same address space, where the Linux kernel views them as a single process. Yet, the application and the proxy (also called a helper program) are linked to two, independent runtime libc libraries and two runtime loaders (ld.so).

The application program that was loaded into memory is linked to a dummy CUDA library that passes all CUDA calls to the proxy program that was loaded. The proxy program contains the active CUDA library, and only the code of the proxy program that communicates with the GPU. For a diagram illustrating the relationship, see Figure 1.

Checkpoint and restart then proceed more or less as described in Section II-B (CheCUDA prior to CUDA 4.0). However, there is a crucial distinction. We do not save the memory of the proxy program. Hence, we are not saving the memory of the active CUDA library that talks to the GPU. The CUDA library includes stateful memory associated with CUDA resources such as UVA/UVM-based memory.

On restart, we will load a completely new copy of the proxy program. The CUDA library of the new copy of the proxy has its original state. The stateful memory of the CUDA library is put back in its initial state. This new architecture again makes feasible the classical log-and-replay of CheCUDA and other applications. The use of log-and-replay in CRAC is described fully later in this section.

The literature describes two ways to implement this single address-space design: *split processes* [23] (two programs in the same address space) and *process-in-process* [24] (using Linux's *dlmopen* to offer independent namespaces, Using dlmopen is superficially attractive, due to the greater simplicity of this approach. Therefore, we analyze this case first.

a) Single address-space design: process-in-process: Process-in-process [24] was introduced as a mechanism to reduce the overhead in inter-process communication between two MPI ranks (processes) that coexist on the same host. By placing the two ranks within a single process by using `dlmopen`, runtime overhead was reduced. It became possible

to directly pass pointers between the two MPI ranks, instead of relying on inter-process communication techniques.

This simple approach is attractive, and it captures many of the goals of split processes, as depicted in Figure 1. However, this approach is not conducive to our requirement of tracking memory associated with the CUDA application program, versus the helper program. The NVIDIA compiler (`nvcc`) links both the CUDA application and the helper program with several libraries — in particular, the NVIDIA CUDA library and the runtime library. It becomes difficult to associate each memory region according to whether it was loaded by a library for the CUDA application or a library for the helper program.

b) Single address-space design: split processes: Split processes [23] were introduced as a mechanism to separate the MPI and network libraries from the end user's MPI application. The split-process approach is more or less the same as described for CRAC near the beginning of Section III-A. There is an important distinction in that in the case of MPI, the proxy or helper program was statically linked [23, Section 3.6]. NVIDIA encourages CUDA programs to be linked dynamically. Even when the `-static` flag is passed to NVIDIA's compiler, some NVIDIA libraries remain dynamically linked.

In this scheme, the helper program is loaded first, resulting in a new process. That process then directly loads the CUDA application into memory. It is important to track all memory allocations (all calls to mmap) by the lower half, so that they are not checkpointed. To accomplish this, a program loading mechanism is used that imitates the way in which the kernel loads an application. (The kernel first loads an ELF interpreter into memory, since the ELF interpreter is structured as a statically linked executable with text, data, and stack. The ELF interpreter then loads the dynamically linked target executable.) The loading mechanism is modified to interpose on all calls to `mmap()`. This allows our kernel loader to load each memory region (including the several NVIDIA libraries) into a restricted portion of the address space, using the `MAP_FIXED` parameter. This approach also yields the illustration in Figure 1, but it provides a mechanism for associating each memory region conceptually with an "upper-half" or "lower-half" portion of the address space.

c) Log-and-replay: Prior to CUDA-4.0, copying the persistent state of CUDA was exemplified by copying two allocation arenas: `cudaMallocHost` on the host; and `cudaMalloc` on the device, or GPU. Just prior to a checkpoint, the data on host and device was copied to a special location, and it was restored on restart. However, CUDA-4.0 and later introduced `cudaMallocManaged` for managed memory, used with UVM. CRCUDA cannot support UVM at all, and CRUM supports it imperfectly, as described in Section II-C.

Hence, copying the full persistent state at checkpoint time has become more of a challenge since CUDA-4.0. CheCuda and earlier approaches had destroyed any CUDA resources prior to checkpointing, and restored them on resume and restart. This worked because the persistent resources of the CUDA library prior to CUDA 4.0 could be logged and later restored. With the advent of UVA/UVM in CUDA-4.0 and later, the unified virtual memory is an essential resource that could not be recovered once destroyed. We infer from our own tests that the use of UVM had modified the CUDA library's state, and the restored CUDA library was then inconsistent when called after restart.

Copying the persistent state would require reverse-engineering the CUDA library, which is all but impossible, due to the closed-source nature of CUDA. But the CUDA library has internal bookkeeping information on the contents of those three allocation arenas. Upon restart, each allocation must be recreated at the original lower-half address that existed prior to checkpoint.

By interposing on the cudaMalloc family of CUDA calls, a log-and-replay approach is used by CRAC to copy to the upper half and later restore the memory regions, in the same order as when they were allocated. This benefits from the determinism in the implementation of CUDA internals for allocation. On restart, a fresh CUDA library in the lower half would allocate the memory regions at the same addresses as originally seen.

This traditional log-and-replay approach described in Section II is compatible with split processes only when targeting smaller CUDA applications. But this widely used log-and-replay approach is observed to fail on more complex applications. It fails for two reasons. In order to apply the approach faithfully and take advantage of determinism in the CUDA library, it would be necessary to re-execute (replay) in the original ordering all calls in the family of cudaMalloc and cudaFree. Second, this approach becomes more difficult when supporting concurrent streams, since two threads on the host may concurrently make calls to cudaMalloc, which would require an extra global lock on all calls to cudaMalloc in the lower-half library. The next section discusses the memory management approach actually used by CRAC.

B. Implementation Issues

Having chosen the split process approach for CRAC, there were several implementation issues arising for the case of CUDA that were not present in the case of MPI.

1) Implementation: Issue of library-allocated memory: The largest complexities of adapting split processes from MPI to CUDA arise from the differing conventions of allocating memory. The design of MPI assumes that calls to MPI will employ caller-allocated memory: callers to the MPI library pre-allocate buffers and pass them to MPI.

The design of CUDA assumes *callee-allocated*, or library-allocated memory: the CUDA library in the lower half may allocate its own internal buffers, and then return those buffers to the calls. A good example is `cudaMalloc` to allocate host memory for the application. This CUDA routine allocates its own memory, and potentially invokes `mmap` to do so.

One can argue that an mmap call can be intercepted, in order to do deterministic replay. However, we observe that a single cudaMalloc call can make many calls to mmap. Moreover, the

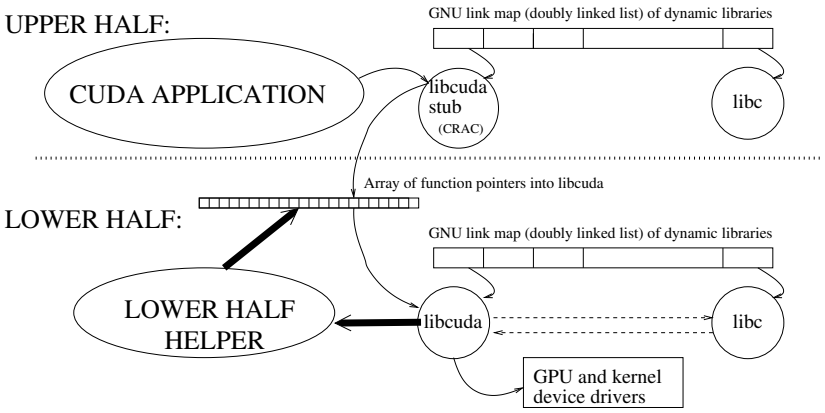


Fig. 1

Split Processes: The lower-half helper program is a tiny CUDA application that was loaded into the "lower half" of the virtual memory address space. At the time of launch, it copied the entry points of CUDA library calls from the lower-half libcuda to an array of libcuda entry addresses. When the main CUDA application was launched (in the upper half), it was launched under control of DMTCP. DMTCP arranged to create a trampoline from the upper-half libcuda to the lower-half libcuda entry point, via the libcuda addresses found in the array created by the lower-half helper program. Now, at runtime, when the end user's CUDA application makes a call to the CUDA library, the trampoline causes control to be passed to the lower-half libcuda. Later, at checkpoint time, only the memory in the upper half will be saved. At restart time, a new lower-half CUDA program is loaded into memory, and it re-initializes the array of libcuda addresses. It then restores the upper-half memory from the checkpoint image, and passes control back to the CUDA application. Note that libcuda represents the CUDA runtime library here.

first cudaMalloc creates a large CUDA malloc arena through mmap. This creation of the malloc arena may fall into the middle of several other mmap calls. Subsequent cudaMalloc calls might not call mmap at all. This produces two problems. (a) It is impractical to interpose on many mmap calls in order to identify the particular mmap calls of interest. (b) The active CUDA malloc buffers to be checkpointed are generally a small fraction of the full CUDA malloc arena.

To counter these problems, we log only the host or device pointers to buffers that were created by a call from the cudaMalloc family of APIs. This improves CRAC performance by avoiding unnecessary interceptions in the lower-half.

2) Implementation: Issue of memory overlapping: The lower- and upper-half memory regions can appear anywhere in the process address space. In DMTCP, one part of saving the state of a running process includes reading the /proc/PID/maps and saving memory regions. In /proc/PID/maps, two memory regions with the same permissions get merged after allocation. This makes it harder to decide if whole or part of a memory region belongs to the upper half and must be checkpointed. This has not been an issue in the case of MANA for MPI [23], where the lower half is compact since it is compiled as a statically linked executable.

Another issue that we observed is that when the library of the lower half allocates memory pages, it may overwrite the upper-half's existing memory pages, and indeed, it may even unmap some of the upper-half's existing memory pages. This could lead to silent memory corruption.

To counter these problems, CRAC tracks all the allocations done by the upper half and also tries to consolidate memory regions created by the upper half, as described in III-A0b.

3) Implementation: Saving the "library-allocated" arena: Since CRAC interposes on the CUDA library in the lower half, it can interpose on all calls to `mmap()`. Naively, one would assume that for each of the cudaMalloc family of calls, there is a single call to `mmap()`, which can be recorded and replayed. This does not work, since a cudaMalloc call may make multiple calls to `mmap()`. Or a single cudaMalloc call may use `mmap()` to create a large allocation arena in memory for later calls to cudaMalloc. While this is helpful for the CUDA library's memory management algorithm, it is not desirable to save the entire arena — especially, when cudaMallocs actually uses only a small portion of the allocation arena.

To counter this, CRAC does its own internal bookkeeping. Rather than saving a large allocation arena that makes the checkpoint size larger unnecessarily, CRAC only saves the memory associated with active mallocs. Active mallocs are those allocations that were allocated but not freed at the time of checkpoint. Draining and refilling device (GPU) memory at active mallocs is essential to make the device state consistent between checkpoint and restart. Note that saving the memory associated with the active mallocs is different from logging the sequence of all cudaMallocs and all cudaFrees. While only the memory associated with active mallocs is saved, we still need to replay the original sequence to get the same host and device addresses as prior to checkpoint (explained in the next section).

4) Implementation: restoring the CUDA library-allocated regions: An important implementation issue for CRAC is to restore all of CUDA's memory allocations at their original address during restart. CUDA has three primary allocation routines: cudaMalloc (on the device), cudaMallocHost/cudaHostAlloc (on the host), and cudaMallocManaged (for UVM:

unified memory on device and host). CRAC logs all CUDA calls that allocate and free memory.

In the case of cudaHostAlloc, it suffices to keep track of only the *active* memory buffers (the buffers that have not been freed at the time of checkpoint). At restart time, CRAC only needs to replay cudaHostMalloc for active memory buffers, in order to again register these buffers with the CUDA library. Note that the memory buffers are already present in the restored upper half memory.

In the cases of cudaMallocHost (on the host), cudaMalloc (on the device) and cudaManagedMalloc (for unified memory), CRAC replays all associated allocations and frees at restart time. The memory associated with these regions is saved at checkpoint time and copied back at restart time. In our experiments on real-world applications, we observed many calls to cudaMalloc and cudaManagedMalloc, but few calls to free those buffers.

CRAC replays the entire log in order to guarantee that active memory allocations are restored at the original address. CRAC relies on determinism of the CUDA library allocation. CRAC also disables address space randomization using Linux's `personality` system call. And CRAC's determinism also relies on using the same CUDA/GPU platform on restart. In the future, three possible solutions can be implemented to optimize this: virtualization of library-allocated addresses; patching applications locations containing the addresses; or a future enhancement by NVIDIA offering a MAP_FIXED flag analogous to the flag of the mmap call.

5) Implementation: Handling CUDA's internal registration of fat binaries:: At the time of launching a CUDA application, CRAC must arrange for the CUDA library in the lower half to register the CUDA kernels residing in the upper half as the active CUDA library loads before the upper half. This requires that CRAC call the lower level CUDA functions in the lower-half CUDA library: `__cudaRegisterFatBinary`, `__cudaRegister<CUDA-element>`, and `__cudaUnregisterFatBinary` (during cleanup at process exit). Here, CUDA elements are device variable, functions, texture, surface, and etc. Finally, during restart, CRAC must re-register the application kernels, since this is a fresh copy of the lower half. This may require additional patching of fat-binary-handle at restart time. This added burden never occurs in the case of MANA for MPI: As before, MANA for MPI benefits from the MPI standard, which defines an almost complete isolation of the MPI library from the MPI application.

IV. Experimental Results

This section present the comprehensive analysis of the CRAC's performance for real-world applications. The aim of this section is to demonstrate that CRAC has low runtime overhead and scales well on real-world applications.

A. Hardware

The experiments presented in this section are performed on the *PSG* cluster of NVIDIA. Each node runs CentOS 7.7 release (kernel version 3.10.0), with 4 NVIDIA Tesla V100 (compute capability 7.0), each with 32 GB of RAM. Each Haswell node is running two 16-core Intel Xeon E5-2698 v3 (2.30 GHz) processors with a total of 256 GB of RAM.

A local, NVIDIA Quadro K600 node with 1 GB of RAM was used only in Section IV-D6. This section includes a special set of experiments to analyze any runtime improvement using the FSGSBASE patch to the Linux kernel [25]. The FSGSBASE patch is under active review for inclusion in the mainline Linux kernel [26], [27]. A custom Linux kernel version 5.0.6 was built from the official Ubuntu git repository[1] on Ubuntu 18.04.3 LTS (Bionic Beaver).

B. Software

Each GPU runs NVIDIA CUDA version 10.0 with driver 440.33.01. We use NVCC to compile the application and use gcc/g++ version 7.3.0 for the linking with libc version 2.17. We use MPICH version 3.3.2 for the MPI-based applications.

The experiments use CRAC, a DMTCP plugin [21], developed to specifically checkpoint and restart CUDA applications using the novel split-process and user-space program loading mechanism (Section III-A0b). DMTCP [2] is an open-source tool for transparent checkpoint-restart for distributed and multi-threaded applications. We use DMTCP version 3.0 [18] (master branch).

C. Terminology

We define the following terminology and formulas that will be used for the rest of the paper.

(a) Runtime overhead: we use the standard formula to calculate the runtime overhead where E_{CRAC} is the execution time of an application under CRAC and $E_{\overline{CRAC}}$ is the native execution time (not under CRAC).

$$\text{Runtime Overhead } \% = \frac{E_{CRAC} - E_{\overline{CRAC}}}{E_{\overline{CRAC}}} \times 100 \quad (1)$$

(b) CUDA calls-per-second (CPS): CUDA API calls are calculated by NVIDIA's profiler `nvprof`. We are interested only in the number of calls from upper half to lower half, for the sake of analyzing their overhead. A simple script extracted just those calls from the upper half (i.e., to the lower-half CUDA runtime library), and not the calls to the CUDA device library (made directly from the lower-half CUDA runtime library). There are three additional CUDA calls that the upper half can make: `cudaLaunchKernel` (reported by `nvprof`), along with two undocumented internal APIs, `__cudaPushCallConfiguration` and `__cudaPopCallConfiguration`. The CUDA compiler generates all three calls for one CUDA kernel launch. So, the formula for total CUDA calls is as follows:

$$\text{Total CUDA calls} = 3 \times count(\text{cudaLaunchKernel}) + count(\text{rest of CUDA runtime API})$$

[1] git://kernel.ubuntu.com/ubuntu/ubuntu-disco.git

The CUDA "calls per second" (CPS) is defined as:

$$CPS = \frac{\text{Total CUDA calls}}{E_{\overline{CRAC}}} \quad (2)$$

D. Application benchmarks

CRAC is analyzed using six CUDA applications. Four of them are standard benchmark suites or real-world applications, and the rest are taken from the official NVIDIA CUDA reference code suite[2]. These applications are chosen to cover a wide range of CUDA features, including Unified Virtual Memory (UVM) and CUDA Streams.

Application	UVM	Streams	CPS	# streams
Rodinia	✗	✗	38K–132K	—
Lulesh	✗	✓	2.5K	2–32
simpleStreams	✗	✓	10K	4–128
UnifiedMemory Streams	✓	✓	4.4K	4–128
HPGMG-FV	✓	✗	35K	—
HYPRE	✓	✓	0.6K	1–10
GROMACS	✗	✓	6K–58K	2

TABLE I: Application benchmarks characterization

Table I characterizes the applications used here. The table includes four columns: UVM and Streams are checked if the application uses the respective CUDA feature. The CUDA calls per second (CPS) are calculated using equation 2. Lastly, for applications using CUDA streams, the range of the number of CUDA streams used by the application is shown.

The Rodinia benchmark suite [28] provides a wide range of applications with a varying CPS. Also, two stream-oriented codes from the NVIDIA CUDA toolkit are used: simpleStreams and UnifiedMemoryStreams [29]. The two applications are chosen because: they exclusively demonstrate the streams feature; and they can be configured easily to use the maximum number of streams on a given GPU (128 streams in our case).

To evaluate CRAC's performance on real-world applications, we use three benchmarks from the DOE (Department Of Energy): LULESH: Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics version 2 [30]; HYPRE: Scalable Linear Solvers and Multigrid Methods library version 2.13.0 [31]; and HPGMG: High-Performance Geometric MultiGrid (using the Github repository's master branch [32]). We also include a very widely used real-world application: GROMACS [33] (using two variants of the ADH_cubic benchmark [34] for the Alcohol Dehydrogenase protein).

*1) **Rodinia Benchmark Suite**:* Rodinia [35], [36] is a commonly used benchmark suite for CUDA. Version 3.1 covers a diverse range of 23 CUDA applications using basic CUDA features, and compatible with all CUDA versions starting from CUDA version 2.x.

We use 14 of the applications from the Rodinia benchmark suite for this work. The other 9 applications were omitted either because they were too short (completing within one second), or because they are similar to benchmarks already included in terms of the total number of CUDA API calls, or because the total number of CUDA API calls was too few.

Rodinia's applications can be scaled by adjusting the input. We use the command line arguments given in Table II for the respective Rodinia benchmark applications.

Application	Command-line argument(s)
BFS	graph1MW_6.txt
CFD	fvcorr.domn.193K
DWT2D	rgb.bmp -d 1024x1024 -f -5 -l 100000
Gaussian	-s 8192 -q
Heartwall	test.avi 104
Hotspot	temp_512 power_512 output.out
Hotspot3D	512 8 1000 power_512x8 temp_512x8 output.out
Kmeans	kdd_cup -l 1000
LUD	-s 2048 -v
Leukocyte	testfile.avi 500
Myocyte	500 1 0
NW	40960 10
Particlefinder	-x 128 -y 128 -z 10 -np 100000
SRAD	2048 2048 0 127 0 127 0.5 1000
Streamcluster	10 20 256 65536 65536 1000 none output.txt 1
LULESH	-s 150

TABLE II: Command-line arguments for Rodinia benchmarks

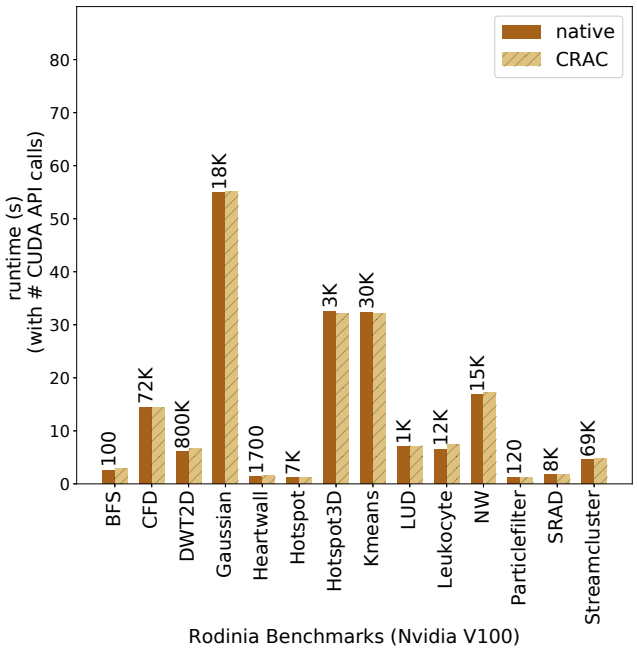


Fig. 2: Runtimes of 14 Rodinia benchmarks with total CUDA API calls made by each benchmark (rounded off)

a) Runtime overhead: Figure 2 shows the runtimes of Rodinia benchmarks without CRAC (native) and with CRAC. We ran 10 iterations of each benchmark and calculated the mean for the each runtime. In almost every case, the 10 iterations had a standard deviation of approximately 0.1 seconds.

The figure also shows that 9 out of 14 benchmarks namely, BFS, DWT2D, Heartwall, Hotspot, LUD, Leukocyte, Parti-

[2]https://docs.nvidia.com/cuda/cuda-samples/index.html

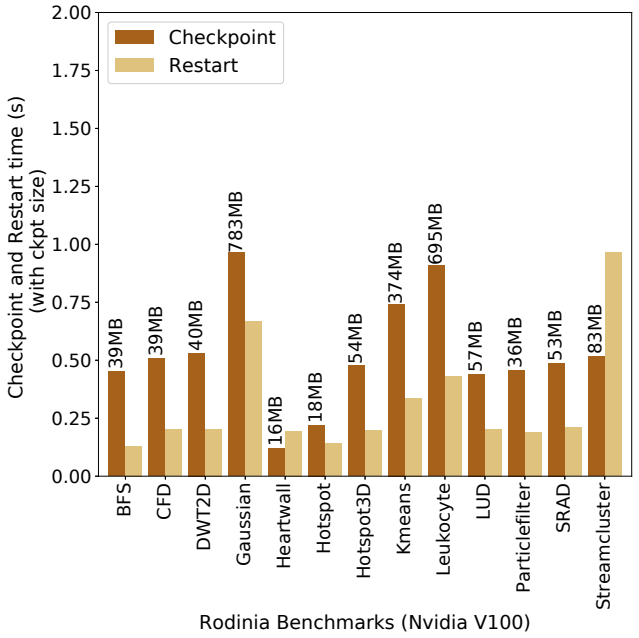


Fig. 3: Checkpoint and restart times of 14 Rodinia benchmarks with checkpoint image sizes

clefilter, SRAD, and Streamcluster, ran in less than 7 seconds. With these benchmarks, the runtime overhead varies between 1% and 14%. The 11% runtime overhead of DWT2D is explained by the abnormal 133,000 CUDA calls per second. There are two common reasons for a higher overhead: first, with short-running applications DMTCP's startup time becomes significant (2.5 second runtime of BFS with 14% overhead, and 1.4 second runtime of Heartwall with 11% overhead); and second, with short-running tasks, the small standard deviation of 0.1 seconds becomes significant compared to the running time, and statistically leads to a higher overhead.

On the other hand, the remaining Rodinia benchmarks run for more than 10 seconds, and we observe there a 0–2% overhead. Interestingly, Hotspot3D, and Kmeans even have a negative overhead. We suspect that this is a result of caching. Finally, CFD and Gaussian have less than 1% overhead, while LUD and NW have less than 2% overhead.

b) Checkpoint overhead: For checkpoint and restart, we disabled DMTCP's default `gzip` compression and triggered checkpoint at random times during an entire run of an application. Figure 3 shows that the checkpoint-restart time is fairly small for CRAC and completes within one second for almost all cases. Checkpoint time is usually smaller than restart time, but there are two outliers (Streamcluster and Heartwall) for which restart takes more time than the checkpoint time. Further investigation showed that these two benchmarks specifically do many CUDA mallocs and CUDA frees. We log CUDA mallocs and frees to make the CUDA library's state consistent, and later replay those APIs on restart. We log the API when a user application calls the CUDA APIs that need to be logged. So, at checkpoint time, no extra work is needed, but at restart, those allocation and free calls were replayed. Note that even then the restart time is still less than 1 second.

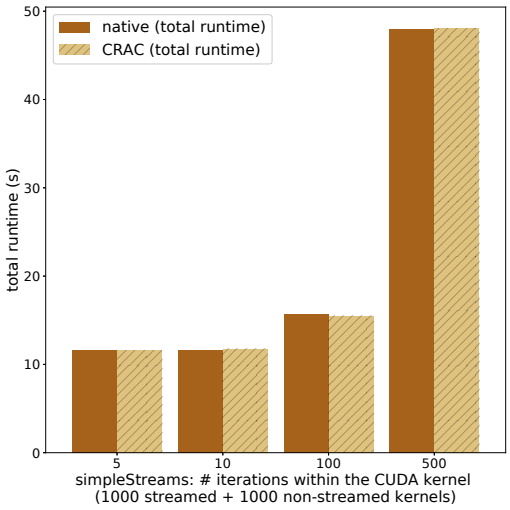
*2) **Stream-oriented benchmarks**:*

a) simpleStreams: SimpleStreams is one of the two code samples we took from NVIDIA's official CUDA code samples. We quote from the code's documentation that simpleStream illustrates the usage of CUDA streams for overlapping kernel execution with device/host memcpy (memory copy). The kernel is used to initialize an array to a specific value, after which the array is copied to the host (CPU) memory. To increase performance, multiple kernel/memcopy pairs are launched asynchronously, with each pair in its own stream. Kernels are serialized. Thus, if n pairs are launched, a streamed approach can reduce the memcopy cost to $(1/n)$th of a single copy of the entire data set.

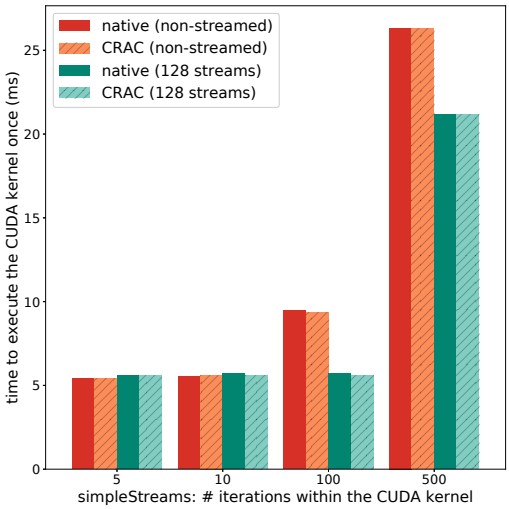
b) Configuration and runtime overhead: We reconfigured the number of streams from 4 (default) to 128. For a NVIDIA V100 GPU with its compute capability of 7.0, 128 is the maximum concurrent kernel limit [37]. The application fails if the stream count is increased beyond the max limit. `nreps` is the number of times each experiment is repeated. For better accuracy, we changed it from its default value of 10 to 1000. `niterations` is the number of iterations for the loop inside the kernel. We have varied this `niterations` variable with values 5, 10, 100, and 500. We use the default Blocking Sync Event synchronization method. The benchmark reports the time to execute one CUDA kernel with streams and without streams (i.e., non-streamed).

Figure 4a shows how the overall runtime of simpleStreams varies with the number of iteration increments. CRAC still maintains less than 1% overhead in each case. Figure 4b (plot on the right) shows the impact of CUDA streams. As `niterations` increases (see previous paragraph) the time to run the CUDA kernel increases. Figure 4b shows that the streamed version becomes significantly faster, compared to the non-streamed version, as `niterations` increases. Yet CRAC continues to perform with low overhead even for the faster streamed version. For the same reasons, CUDA streams is widely used over regular non-streamed kernel launches. Note that CRAC incurs no overhead; neither in non-streamed CUDA kernel execution time nor in the one with streams. This shows that even after increasing the concurrency level to the max (128 streams), CRAC handles it well as compared to previous solutions. Figure 5a shows the runtime with the same configuration (128 streams, 1000 repetitions, and 500 iterations).

c) UnifiedMemoryStreams(UMS): UnifiedMemoryStreams (UMS) is taken from NVIDIA's code samples and illustrates the usage of streams with Unified Memory. UnifiedMemoryStreams implements a simple task consumer using threads and streams with all data in Unified Memory, and tasks consumed by both host and device. The application randomizes task sizes for a total of 40 tasks with 4 streams. Based on the task size, the application decides at run time



(a) Runtimes of simpleStreams without and with CRAC while increasing the iterations within the CUDA kernel



(b) Time for one CUDA kernel execution time without and with streams in simpleStreams. (More iterations imply a longer-running kernel.)

Fig. 4: Experiments with simpleStreams from the NVIDIA CUDA code sample

whether the task should be run on the host or the device. Note that both the device and host are using same unified memory. **Configuration and runtime overhead:** We configured the application to use 128 streams with a total of 1280 tasks. Since we needed to run the experiments 10 times for the average runtime, we set the seed to a random number 12701 to get consistent task allocations. We measured the execution time by elapsed wall-clock time. Figure 5a shows the average runtime without and with CRAC. We observed an overhead of 1.5%.

d) LULESH: Version 2.0 GPU model of LULESH is specifically implemented for NVIDIA's GPUs. LULESH is a scientific real-world application developed by Lawrence Livermore National Laboratory [30] that solves the Shock Hydronomics Challenge Problem. LULESH provides two options, one with an unstructured grid and the other with a structured grid. In our case, we use a structured grid with 150 edge elements, which makes the problem size $150\times150\times150$, and which uses nearly 2 GB of memory.

Runtime overhead: LULESH calls 210K CUDA calls in 80 seconds of its execution time that means around 2.5K CUDA calls per second. We saw that with maximum streams in simpleStreams and UnifiedMemoryStreams, CRAC still incurs low-overhead. With the real-world application that makes 65K cudaLaunchKernel calls. Figure 5a shows that LULESH's performance is still the same with CRAC, with an overhead slightly less than 2%.

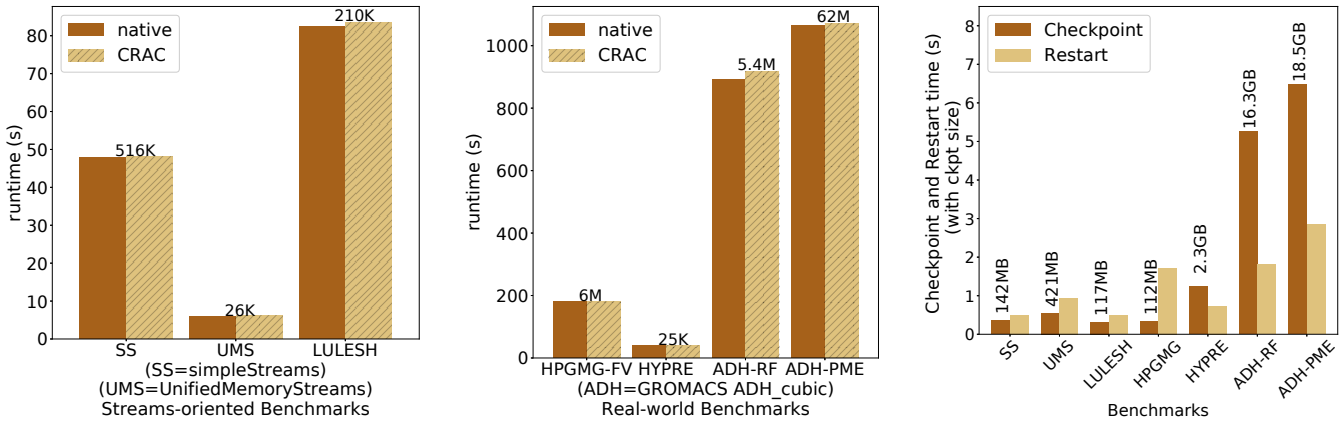
e) Checkpoint overhead: Figure 5c shows that the checkpoint overhead is very low as compared to the overall runtime of each stream-oriented application. CRAC needs to recreate streams and make the CUDA library's state consistent. So, the time is slightly more than the checkpoint time. However, both checkpoint and restart finish within one second in each stream-oriented application.

*3) **Real-world applications (HPGMG-FV, HYPRE, and GROMACS)**:* HPGMG is a high-performance geometric multigrid application. It is one of the benchmarks used for ranking speeds of the top supercomputers [38]. We use HPGMG-FV (Finite Volume) for our experiments. HPGMG-FV can be scaled further with MPI over multiple nodes. However, it suffices for our purposes to run HPGMG-FV over a single MPI rank. This already provides a real-world scale since this configuration of HPGMG-FV results in 2 million CUDA calls per minute (35,000 CUDA calls per second). This scale is already representative of real-world high-performance applications.

HYPRE is a linear system solver library that makes only 600 CUDA calls per second. However, HYPRE creates large UVM regions, and employs long-running kernels. One MPI-rank can create UVM regions of up to 1 GB, and the host and the device both work simultaneously on UVM regions via CUDA streams and textures. Therefore, HYPRE incurs a higher memory footprint than HPGMG-FV.

GROMACS (GROningen MAchine for Chemical Simulation) is widely used molecular dynamics simulation package. It is primarily designed for biochemical molecules like proteins, lipids and nucleic acids. A test benchmark (ADH_cubic) for one such protein, Alcohol Dehydrogenase is provided by the GROMACS developers [34]. The ADH cubic benchmark contains two use cases — namely RF (Reaction-Field) and PME (Particle Mesh Ewald). GROMACS has been run with one MPI rank and 32 OpenMP threads.

The following table shows the command-line arguments needed to run these four real-world applications.



(a) (b) (c)

Fig. 5: (a) Runtimes of stream-oriented benchmarks; (b) Runtimes of real-world benchmarks; and (c) Checkpoint and Restart times using CRAC. The numbers above the bars in parts (a) and (b) report the total number of CUDA calls. The numbers above the bars in part (c) report the checkpoint image size. Note that GROMACS ADH cubic PME exhibits as many as 58,000 CUDA calls per second from upper to lower half. Even under this very high stress, CRAC's runtime overhead remains small.

Application	Command-line arguments
HPGMG-FV	7 8
HYPRE	ij -solver 1 -rlx 18 -ns 2 -CF 0 -hmis -interptype 6 -Pmx 4 -keepT 1 -tol 1.e-8 -agg_nl 1 -n 250 250 250 250
GROMACS ADH_cubic-rf	-nb gpu -nsteps 600000 -s rf.tpr
GROMACS ADH_cubic-pme	-nb gpu -nsteps 600000 -s pme.tpr

Runtime overhead: Figure 5b shows native and CRAC runtimes for HPGMG-FV and HYPRE, and for two variants of GROMACS. The times in Figure 5b for HPGMG-FV and HYPRE are averaged using 10 native runs and 10 runs with CRAC. The two variants of GROMACS have been run with 600,000 steps instead of the default 10,000 steps, in order to avoid some high statistical variances seen with the deault option. The GROMACS runs are averaged over 3 native runs and 3 runs with CRAC, and each individual time is within about 1% of the overall average. CRAC manages a runtime overhead of about 2% for HPGMG and 3% for HYPRE. For GROMACS, the runtime overhead was less than 3% for ADH cubic-RF and less than 1% for ADH cubic-PME.

Checkpoint overhead: Figure 5c shows that with HPGMG, CRAC needs to replay many CUDA calls in relation to its checkpoint size. Therefore, CRAC takes nearly 1.75 seconds to restart HPGMG. In contrast, HYPRE's checkpoint size is 2.3GB, but it takes less time to restart. Thus, while the runtime overhead and time to checkpoint are low, the time to restart can be longer, depending on how many CUDA calls CRAC needs to replay to restore the state of the new CUDA library. We observe that even though the PME case of GROMACS incurs 58,000 CUDA calls per second, the restart time is still relatively short. So, we conclude that the restart time of the GROMACS variants are dominated by restoring the image by DMTCP.

*4) **Split processes for cuBLAS**:* By placing the NVIDIA cuBLAS library in the lower half, the support for BLAS [39] (Basic Linear Algebra Subprograms) further reduces the runtime overhead. This works by reducing the number of calls from upper to lower half. (See column 4 of Table III.)

We ran three types of programs: cublasSdot (inner product), cublasSgemv (matrix-vector product), and cublasSgemm (matrix-matrix product). The dimension was chosen so that the matrix (or vector, for cublasSdot) had data size 1 MB, 10 MB, or 100 MB. A timing loop of 10,000 calls to the cuBLAS routine was used for accuracy, and times are reported for just a single iteration. When processing larger data (matrices and vectors of size 100 MB), CRAC (column 4) shows a runtime overhead varying from 0.5% to -0.8%. (The negative overhead is presumed due to cache locality.)

CUDA Call	Data size	Native (ms)	CRAC(ms) (% overhead)	CMA/IPC(ms) (% overhead)
cublasSdot	1MB	0.026	0.027 (3.9)	0.21 (698)
cublasSdot	10MB	0.049	0.050 (3.3)	2.56 (5142)
cublasSdot	100MB	0.282	0.284 (0.5)	50.4 (17766)
cublasSgemv	1MB	0.012	0.012 (1.9)	0.082(577)
cublasSgemv	10MB	0.036	0.037 (0.7)	1.25 (3329)
cublasSgemv	100MB	0.142	0.142(-0.1)	25.5 (17812)
cublasSgemm	1MB	0.202	0.207 (2.4)	0.49 (142)
cublasSgemm	10MB	1.806	1.816 (0.6)	9.03 (400)
cublasSgemm	100MB	32.373	32.107 (-0.8)	100.34 (209)

TABLE III: Comparison of native both: (a) to CRAC with cuBLAS in lower half; and (b) to use of CMA/IPC to simulate the approaches of CRCUDA and CRUM

*5) **Comparison of CRAC to earlier proxy-based approaches: The cost of IPC**:* As described at the beginning of Section III-A, the starting point for CRAC was the observation that the existing proxy-based checkpointing approaches (e.g., CRCUDA and CRUM) rely on expensive inter-process communication (IPC) between CUDA application and the proxy. The authors of CRUM measured the runtime overhead on real-

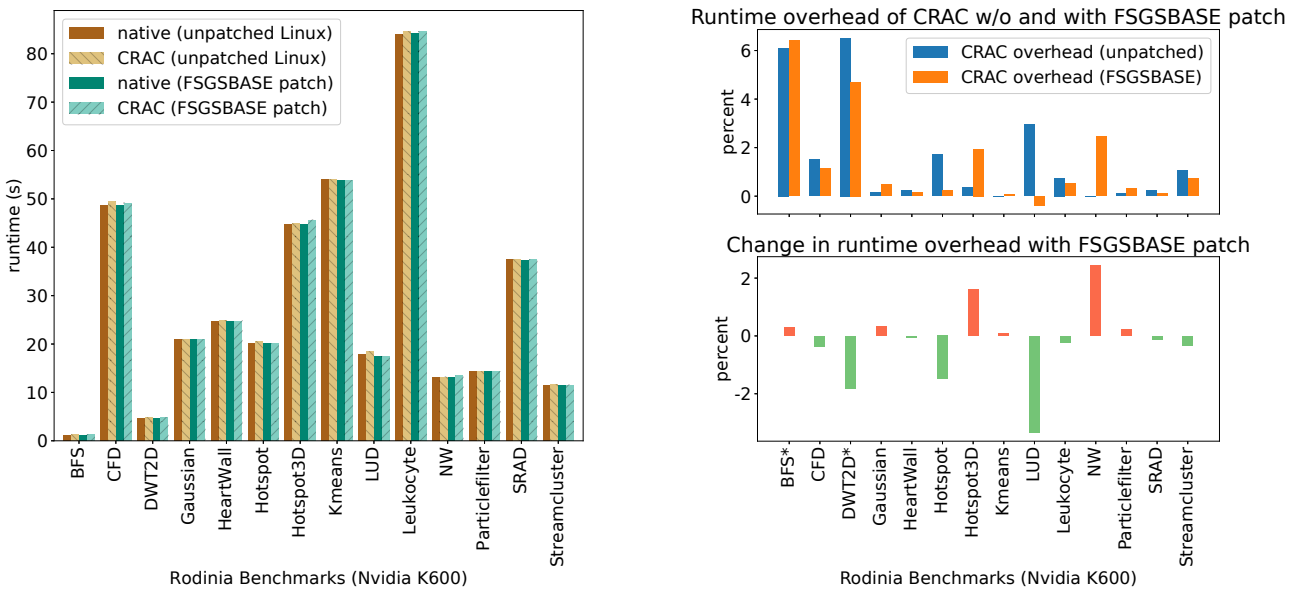


Fig. 6: (left) Runtimes of Rodinia benchmarks without and with CRAC on both unpatched and patched (FSGSBASE) Linux; (right, top) Runtime overhead of CRAC on both unpatched and patched (FSGSBASE) Linux; (right, bottom) and percentage difference observed with CRAC's runtime overhead with patched Linux as compared to unpatched Linux (lower is better).

world benchmarks at from 6% to 12%.

Here we present a synthetic IPC benchmark (CMA/IPC: column 5 in Table III) to simulate the proxy-based approach of CRCUDA and CRUM. CMA is Cross-Memory Attach (i.e., the Linux syscalls process_vm_readv and process_vm_readv). We assume that cuBLAS is in the proxy process, to reduce the IPC communication. CMA is used to copy the application to a proxy process (which executes the cuBLAS routine), and the result is copied back to the application.

The overhead using CMA [40] (Cross Memory Attach) for IPC varies from 142% to 17,812%. The overhead is huge, as expected. As an example, one iteration of cublasSdot for 1 MB data runs natively in 0.026 ms. This implies 1/(0.026 ms) = 38,000 calls per second. So, CMA must be invoked 76,000 times per second for the round trip of the cublasSdot call.

*6) **Runtime overhead improvement using Linux's upcoming FSGSBASE patch**:* A small experiment was also performed to see if there was significant benefit to using the upcoming FSGSBASE patch to the Linux kernel [25]. In the current Linux, switching to a new thread (or to the lower-half program in our case) requires a kernel call to set the corresponding x86-64 "fs" register for that thread. A kernel call may require a millisecond. If done frequently, this can be expensive. At least in the case of MPI applications, it was previously observed in [23] that the expense of the kernel calls was significant when calling lower-half routines.

Hence, we wished to see if CRAC's already small runtime overhead could be further reduced by using the FSGSBASE patch to directly set the "fs" register, instead of setting "fs" through kernel calls. As we shall see, the added advantage of the FSGSBASE patch is small, and often nearly zero.

We next analyze whether the FSGSBASE patch can further reduce CRAC's runtime overhead. CRAC needs to get and set the "fs" register when it makes a call from the upper half to the lower half. (This is analogous to the use of the "fs" register in context switches among threads in Linux.) Setting the "fs" register is expensive due to the kernel call.

The experiments of Figure 6 were run on a local node: an older NVIDIA Quadro K600 GPU. It was not possible to install a patched Linux kernel on the public, production nodes used for the experiments in the other figures. This also explains why the same Rodinia benchmarks mostly ran for at least 10 seconds in this experiment.

Figure 6 presents two columns of graphs. On the left, the original 14 Rodinia benchmarks are plotted. Each benchmark shows the native runtime and the CRAC runtimes, both with and without the FSGSBASE patch. The runtimes with FSGSBASE were taken using the FSGSBASE/v9 kernel patches [25].

The two graphs on the right in Figure 6 present the same data, but they express the data as percentage differences, to more clearly contrast two cases: the runtime overhead of CRAC (with and without FSGSBASE); and the change in runtime overhead of CRAC when using the FSGSBASE patch. Lower is better in both cases.

V. Related Work

Much of the work targeting transparent checkpointing of CUDA was already covered in Section II, as part of the

motivation for a fresh approach in CRAC. See Section II for more details.

To summarize, several techniques [10]–[14] were explored prior to CUDA 4.0 (in 2011 and earlier). Unified memory between device and host was later introduced to CUDA in two increments: Unified Virtual Addressing (UVA) in CUDA 4.0; and Unified Virtual Memory (UVM [41]) in CUDA 6.0. This made CUDA incompatible with the previous techniques.

Since then, two newer checkpointing approaches appeared: CRCUDA [15] and CRUM [23]. The limitations of these two approaches were already described: high runtime overhead, incomplete UVM support, and untested scaling of concurrent streams. (See the second page of Section I for details.)

It remains to describe four techniques from the literature that are related to the implementation of CRAC: proxies in CRUM; proxies in the wider literature; split processes; and process-in-process.

a) Use of proxy processes in CRUM: The previous work of CRUM in checkpointing CUDA has an unacceptably high overhead of 6% or more. This occurs at two extremes.

Case I: *Many short-lived kernels.* This incurred overhead because of the need to frequently marshal and un-marshal the parameters for communication between the application and the proxy process when invoking CudaLaunchKernel. For example, HPGMG-FV has a high frequency of CUDA calls.

Case II: *Kernel and host access many UVM memory pages frequently.* This requiring frequent calls to `mprotect` and userfault_fd (a recent Linux utility serving the same purpose as segfault handlers). This interacted particularly badly with NVIDIA UVM.

b) Proxy processes: Proxy processes are a well-known concept that is widely used in systems. In an early example, Zandy et al. [42] 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. [43] use a proxy process for isolation of heap accesses by a process and for containment of attacks to the heap. CheCL [44] has employed proxy processes already in 2010, for the closely related OpenCL language [45] for GPUs. CRCUDA [15] and CRUM also employ proxy processes.

c) Split processes: Split processes are described in Figure 1 in Section III-A0b. MPI for MANA [23] adopted split processes in the context of checkpoint-restart for MPI. Upon checkpoint, only the upper half memory is saved. On restart, a small bootstrap program in the lower half restores the upper half memory, and the upper half then replays any persistent state associated with a physical device. In the case of MANA, that physical device would be the network, and/or sockets communicating with a central MPI coordinator. In the case of the current work (CRAC), the physical device is the GPU.

There are several antecedents to the idea of combining two programs in a single process. Here we note McKernel and shadow device drivers, both devised for the Linux kernel.

McKernel [46] runs a "lightweight" kernel along with a full-fledged Linux kernel. The HPC application runs on the lightweight kernel, which implements time-critical system calls. The rest of the functionality is offloaded to a proxy process running on the Linux kernel. The proxy process is mapped in the address space of the main application, similar to MANA's concept of a lower half, to minimize the overhead of "call forwarding" (argument marshalling/un-marshalling).

Swift et al. [47] developed the idea of a "shadow device driver". The lower half corresponds to the actual device driver, and the upper half corresponds to a shadow device driver that mirrors (or "logs") all transactions to the lower half. If the lower-half device driver crashes, then it is re-initialized and a long-and-replay approach is used to re-initialize it.

d) Process-in-process: an approach related to split processes: Process-in-process [24] is related to split process in that that both approaches load multiple programs into a single address space. However, the goal of process-in-process was intra-node communication optimization, and not checkpoint-restart. Given two MPI ranks (processes) co-located on a single computer node, the two ranks were loaded into a single address space, to make copying of messages between the two MPI ranks more efficient.

Unlike split processes, process-in-process loads *all* MPI ranks co-located on the same node as separate *threads* within a single process, but in different logical "namespaces", in the sense of the `dlmopen` namespaces in Linux.

VI. Conclusion and Future Work

Transparent checkpointing of CUDA with low runtime overhead has been demonstrated. This is important, since: earlier approaches (prior to CUDA 4.0) are incompatible with the versions of CUDA-4.0 and later; and two later approaches [15], [23] suffer from high runtime overhead, incomplete UVM support, and untested scaling of concurrent streams. The new CRAC approach using split processes demonstrates low runtime overhead (about 1%), and support for the aforementioned CUDA features.

CRAC's runtime overhead of 1% is dominated by the cost of calls from the upper half to the lower half. This can be further minimized by placing intermediate-level libraries in the lower half. As an example, cuBLAS (column 4 of Table III) shows close to zero runtime overhead when processing matrices or vectors of size 100 MB. Further, a proof of principle was demonstrated for checkpointing of hybrid MPI+CUDA on a single node. In future work, this proof of principle for transparent checkpointing of MPI+CUDA will be extended to full support for MPI on multiple nodes, using ideas from MANA [23].

Acknowledgment

We thank Michael Sullivan of NVIDIA for a careful reading and comments, and also for the use of computer resources at NVIDIA. We also thank Rohan Garg for conversations describing his earlier design of CRUM for CUDA.

REFERENCES

[1] TOP500, "TOP500 supercomputer sites," https://www.top500.org/, Nov. 2019.

[2] 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, pp. 1–12.

[3] P. H. Hargrove and J. C. Duell, "Berkeley Lab Checkpoint/Restart (BLCR) for Linux clusters," *Journal of Physics: Conference Series*, vol. 46, no. 1, p. 494, 2006.

[4] CRIU team, "CRIU," accessed Dec., 2019, http://criu.org/.

[5] S. Yi, A. Andrzejak, and D. Kondo, "Monetary cost-aware checkpointing and migration on Amazon cloud spot instances," *IEEE Transactions on Services Computing*, vol. 5, no. 4, pp. 512–524, 2011.

[6] B. Nie, D. Tiwari, S. Gupta, E. Smirni, and J. H. Rogers, "A large-scale study of soft-errors on GPUs in the field," in *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2016, pp. 519–530.

[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, pp. 297–310. [Online]. Available: http://doi.acm.org/10.1145/2694344.2694348

[8] 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, pp. 38:1–38:12. [Online]. Available: http://doi.acm.org/10.1145/2807591.2807666

[9] 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, pp. 331–342.

[10] 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, pp. 1–11.

[11] 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, pp. 17–24.

[12] 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, pp. 408–413.

[13] L. B. Gomez, A. Nukada, N. Maruyama, F. Cappello, and S. Matsuoka, "Transparent low-overhead checkpoint for GPU-accelerated clusters," 2010. [Online]. Available: https://wiki.ncsa.illinois.edu/download/attachments/17630761/INRIA-UIUC-WS4-lbautista.pdf

[14] 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, pp. 104–113.

[15] T. Suzuki, A. Nukada, and S. Matsuoka, "Transparent checkpoint and restart technology for CUDA applications," in *GPU Technology Conference (GTC'16)*, 2016. [Online]. Available: https://on-demand.gputechconf.com/gtc/2016/presentation/s6429-akira-nukada-transparen-checkpoint-restart-technology-cuda-applications.pdf

[16] R. Garg, A. Mohan, M. Sullivan, and G. Cooperman, "CRUM: Checkpoint-restart support for CUDA's unified memory," in *IEEE Int. Conf. on Cluster Computing (CLUSTER'18)*. IEEE Press, 2018, pp. 302–313.

[17] T. Suzuki, A. Nukada, and S. Matsuoka, "CRCUDA source," 2015. [Online]. Available: https://github.com/tbrand/CRCUDA

[18] DMTCP, "dmtcp/dmtcp," Mar 2020. [Online]. Available: https://github.com/dmtcp/dmtcp.git

[19] T. C. Schroeder, "Peer-to-peer & Unified Virtual Addressing," NVIDIA webinar, 2011. [Online]. Available: https://developer.download.nvidia.com/CUDA/training/cuda_webinars_GPUDirect_uva.pdf

[20] M. Harris, "Unified memory in CUDA 6," NVIDIA Blog, 2013. [Online]. Available: https://devblogs.nvidia.com/parallelforall/unified-memory-in-cuda-6/

[21] 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, pp. 402–412.

[22] J. Cao, K. Arya, R. Garg, S. Matott, D. K. Panda, H. Subramoni, J. Vienne, and G. Cooperman, "System-level scalable checkpoint-restart for petascale computing," in *22nd IEEE Int. Conf. on Parallel and Distributed Systems (ICPADS'16)*. IEEE Press, 2016, pp. 932–941, also, technical report available as: arXiv preprint arXiv:1607.07995.

[23] R. Garg, G. Price, and G. Cooperman, "MANA for MPI: MPI-agnostic network-agnostic transparent checkpointing," in *Proc. of the 28th Int. Symp. on High-Performance Parallel and Distributed Computing*. ACM, 2019, pp. 49–60.

[24] A. Hori, M. Si, B. Gerofi, M. Takagi, J. Dayal, P. Balaji, and Y. Ishikawa, "Process-in-process: Techniques for practical address-space sharing," in *Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing*, 2018, pp. 131–143.

[25] C. S. Bae, "[PATCH v9 00/17] Enable FSGSBASE instructions," Oct. 4, 2019, https://lkml.org/lkml/2019/10/4/725.

[26] J. Corbet, "A possible end to the FSGSBASE saga," June 1, 2020. [Online]. Available: https://lwn.net/Articles/821723/

[27] S. Levin, "[PATCH v13 00/16] Enable FSGSBASE instructions," May 28, 2020, https://lkml.org/lkml/2020/5/28/1358.

[28] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *2009 IEEE International Symposium on Workload Characterization (IISWC'09)*. IEEE, 2009, pp. 44–54.

[29] "CUDA samples." [Online]. Available: https://docs.nvidia.com/cuda/cuda-samples/index.html

[30] I. Karlin, J. Keasler, and R. Neely, "Lulesh 2.0 updates and changes," Lawrence Livermore National Laboratory, Tech. Rep. LLNL-TR-641973, August 2013.

[31] Lawrence Livermore National Laboratory (LLNL), "HYPRE: Scalable linear solvers and multigrid methods," 2017. [Online]. Available: https://computing.llnl.gov/projects/hypre-scalable-linear-solvers-multigrid-methods

[32] Lawrence Berkeley National Laboratory (LBL), "HPGMG: High-Performance Geometric Multigrid," 2017. [Online]. Available: https://bitbucket.org/nsakharnykh/hpgmg-cuda

[33] D. Van Der Spoel, E. Lindahl, B. Hess, G. Groenhof, A. E. Mark, and H. J. Berendsen, "GROMACS: Fast, flexible, and free," *Journal of Computational Chemistry*, vol. 26, no. 16, pp. 1701–1718, 2005.

[34] Gromacs team, "Gromacs benchmarks," Aug 2015. [Online]. Available: http://ftp.gromacs.org/pub/benchmarks/

[35] 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, pp. 44–54.

[36] "Rodinia: Accelerating compute-intensive applications with accelerators." [Online]. Available: http://rodinia.cs.virginia.edu/doku.php

[37] "CUDA C programming guide." [Online]. Available: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#features-and-technical-specifications__technical-specifications-per-compute-capability

[38] HPGMG team, "High-Performance Geometric multigrid, an HPC benchmark and supercomputing ranking metric," 2016. [Online]. Available: https://hpgmg.org

[39] L. S. Blackford, A. Petitet, R. Pozo, K. Remington, R. C. Whaley, J. Demmel, J. Dongarra, I. Duff, S. Hammarling, G. Henry *et al.*, "An updated set of basic linear algebra subprograms (BLAS)," *ACM Transactions on Mathematical Software*, vol. 28, no. 2, pp. 135–151, 2002.

[40] J. Vienne, "Benefits of cross memory attach for MPI libraries on HPC clusters," in *Proceedings of the 2014 Annual Conference on Extreme Science and Engineering Discovery Environment*, 2014, pp. 1–6.

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

[42] 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, pp. 177–184.

[43] M. Kharbutli, X. Jiang, Y. Solihin, G. Venkataramani, and M. Prvulovic, "Comprehensively and Efficiently Protecting the Heap," *ACM Sigplan Notices*, vol. 41, no. 11, pp. 207–218, 2006.

[44] 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, pp. 864–876.

[45] J. E. Stone, D. Gohara, and G. Shi, "OpenCL: A parallel programming standard for heterogeneous computing systems," *Computing in Science and Engineering*, vol. 12, no. 3, pp. 66–73, 2010.

[46] B. Gerofi, M. Takagi, A. Hori, G. Nakamura, T. Shirasawa, and Y. Ishikawa, "On the scalability, performance isolation and device driver transparency of the IHK/McKernel hybrid lightweight kernel," in *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2016, pp. 1041–1050.

[47] M. M. Swift, M. Annamalai, B. N. Bershad, and H. M. Levy, "Recovering device drivers," *ACM Transactions on Computer Systems (TOCS)*, vol. 24, no. 4, pp. 333–360, 2006.