cudaCR: An In-kernel Application-level Checkpoint/Restart Scheme for CUDA-enabled GPUs

Behnam Pourghassemi
Electrical Engineering and Computer Science
University of California, Irvine, CA
bpourgha@uci.edu

Aparna Chandramowlishwaran
Electrical Engineering and Computer Science
University of California, Irvine, CA
amowli@uci.edu

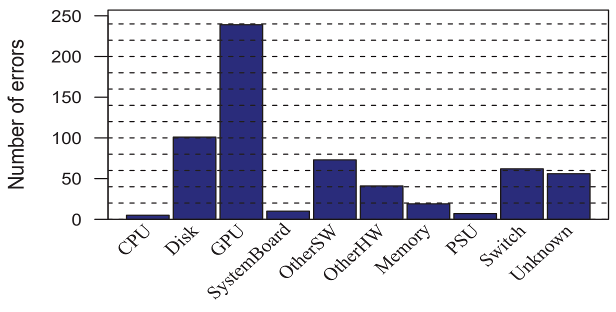
***Abstract*—Fault-tolerance is becoming increasingly important as we enter the era of exascale computing. Increasing the number of cores results in a smaller mean time between failures, and consequently, higher probability of errors. Among the different software fault tolerance techniques, checkpoint/restart is the most commonly used method in supercomputers, the defacto standard for large-scale systems. Although there exist several checkpoint/restart implementations for CPUs, only a handful have been proposed for GPUs even though more than 60 supercomputers in the TOP 500 list are heterogeneous CPU-GPU systems.**

In this paper, we propose a scalable application-level checkpoint/restart scheme, called *cudaCR* for long-running kernels on NVIDIA GPUs. Our proposed scheme is able to capture GPU state inside the kernel and roll back to the previous state within the same kernel, unlike state-of-the-art approaches. We evaluate *cudaCR* on application benchmarks with different characteristics such as dense matrix multiply, stencil computation, and k-means clustering on a Tesla K40 GPU. We observe that *cudaCR* can fully restore state with low overheads in both *time* (less than 10% in best case) and *memory* requirements after applying a number of different optimizations (storage gain: 54% for dense matrix multiply, 31% for k-means, and 4% for stencil computation). Looking forward, we identify new optimizations to further reduce the overhead to make *cudaCR* highly scalable.

***Keywords*-Fault tolerance, soft errors, checkpoint/restart, GPU, supercomputer**

I. Introduction

High-performance computing (HPC) deploys supercomputers with millions of cores for running massive-scale computationally intensive applications. As the demand for performance increases, the number of cores in the next generation of supercomputers is also expected to increase. On one hand, moving from petascale towards exascale opens new opportunities but on the other hand, it also brings new challenges in the reliability of machines. Even if there are no errors in software, there always exist failures in hardware. The Mean Time Between Failure (MTBF) for a single node is estimated to be between 10 to 1000. Contrary to this, the MTBF of a machine with hundred thousand nodes may drop to less than hour [1], [2]. Therefore, investing in a fault-tolerant system for large-scale long-running applications is inevitable. Errors in systems can be broadly classified into two categories – (a) *hard errors* which might be permanent like chip malfunction that requires hardware replacement and (b) *soft* errors which are transient like bit flips that can be recovered by rewriting correct data from another location. Checkpoint/restart is one of the most commonly used methods for fault-tolerance in supercomputers that can recover transient errors. The key idea is to periodically save the state of a system into a secondary storage. In the event of a failure, the system is recovered by restarting the execution from the last clean state from the secondary storage.



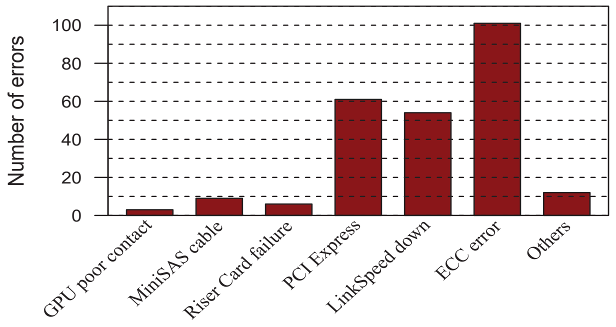


Figure 1. Break-down of failures in TSUBAME 2.5 from October 1, 2015 to October 1, 2016. Top: Failure distribution based on their source. Bottom: Break-down of GPU failures based on their reason.

Today, Graphical Processing Units (GPUs) are popular as they offer both higher peak performance and higher bandwidth compared to traditional CPU processors. Titan, the third fastest supercomputer in the Top 500 list has 18,688 compute nodes and the same number of Tesla K20 GPUs[1]. With an increasing number of GPUs in large-scale systems, reliable GPU computing is now as important as reliable CPU computing. Moreover, GPUs exhibit more vulnerability to hardware failures compared to CPUs. The top of Figure 1 shows the breakdown of failures by category distilled from the failure history of TSUBAME2.5[2] during the period from October 1, 2015, to October 1, 2016. There was a total of 613 failures and a staggering $\approx 40\%$ are GPU failures. In the bottom of Figure 1, GPU errors are categorized based on its reason. As we can see, ECC errors (which includes single-bit and double-bit soft errors) are a significant fraction of the total GPU errors. This clearly demonstrates the high failure rate of GPUs and is the motivating factor in designing an efficient checkpoint/restart scheme for GPUs similar in spirit to CPUs.

Checkpoint/Restart (CR) schemes for CPUs are typically implemented at three levels – kernel, library, and application.

- **Kernel-level** – The operating system does checkpoint/restart using appropriate system calls and as a result, the application remains transparent to the system. In this scenario, there is no application-level modification required to ensure fault tolerance.
- **Library-level** (also known as user-level) – The application is linked with an user-library that is in charge of saving/restoring information. In this level, there is still no modification required in the application but it has to link with the CR library.
- **Application-level** – Checkpoint/rollback is done by the application and it has to be re-compiled.

GPU is an external device that is handled by drivers rather than an operating system. Additionally, there is no available API to access the computation state at the library level, so the first two approaches discussed above are infeasible on current GPUs [3]. Although there exist few schemes known as kernel-level GPU checkpointing [4], [5], they are in fact implemented on top of previous CPU checkpointing and save computational state after GPU completes the kernel execution. So, they are essentially *out-kernel* checkpoint/restart schemes. In other words, there is no checkpointing of in-kernel data and no rollback inside the GPU kernel.

This paper proposes an application-level *in-kernel checkpoint/restart scheme* for GPUs. To ensure compatibility of our proposed scheme on a vast majority of applications, we determine and set necessary conditions. Our implemented scheme makes relevant changes in both the host and device source codes at compile time and adds checkpointing modules that are invoked at runtime. This paper makes the

[1]https://www.olcf.ornl.gov/titan

[2]http://mon.g.gsic.titech.ac.jp/trouble-list/index.htm

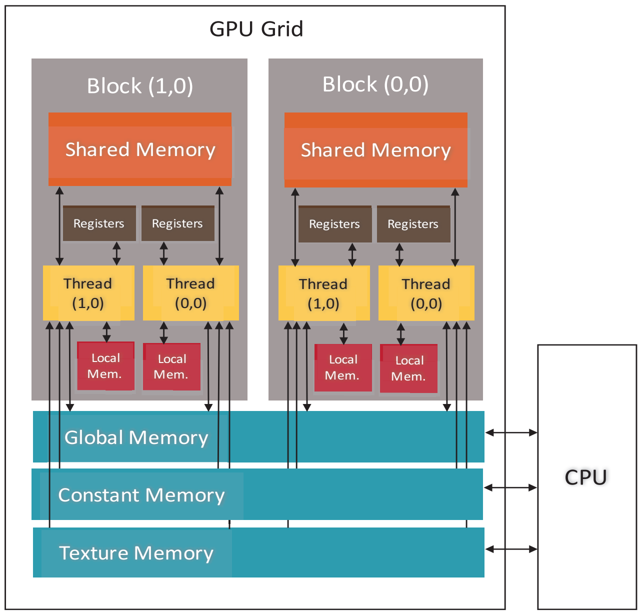


Figure 2. Block diagram of the GPU architecture and programming model.

following contributions.

- We design and implement a new scheme for in-kernel checkpointing of GPUs called *cudaCR*, where the user is only responsible for determining checkpoint locations. A pre-compiler automatically translates both the CPU and GPU codes to new codes that are capable of checkpointing and restoring the correct state (Section III).
- We discuss novel algorithms and data structures for collecting computation states asynchronously from different GPU memories and present optimizations for significantly reducing the storage of secondary data and overhead from checkpointing (Section III).
- We evaluate cudaCR on application benchmarks that exhibit different data access patterns such as dense matrix multiply, stencil computation, and k-means clustering on a Tesla K40 GPU to test both the effectiveness and efficiency of the proposed scheme. We observe that cudaCR can fully restore state with overheads less than 10% in the best case (Section IV).

II. Background

A. GPU and CUDA programming model

GPUs were originally designed for rendering images and graphics pipelines but they soon became a compelling platform for scientific and high-performance computing due to its high peak performance and memory bandwidth. In 2007, NVIDIA released CUDA (Compute Unified Device Architecture) as a new programming model to program NVIDIA GPUs in a much easier way.

CUDA organizes the device code using abstractions of threads, blocks, and grids. *Kernels*, functions on the device side, are executed by *threads*. Groups of threads are organized into *blocks* and threads within a block can synchronize with one another. CUDA assigns each block to one GPU computing box aka Streaming Multiprocessor (SM). Blocks cannot synchronize among one another and form a *grid* that denotes application scope. There is also a hierarchy of memories in GPUs with different access policies. The *global memory* is accessible by the entire grid (all blocks and threads within a block), the *shared memory* is local to each thread block (only visible to threads inside the particular block), and the *registers* are thread-local where each thread has a limited number of them. The *local memory* is an extension of registers that reside in the global memory. GPUs also have other types of memories known as *texture* and *constant* that have the same access policy as the global memory but are read-only. The GPU architecture and data movement between host and device along with the different memory hierarchies are illustrated in Figure 2.

B. Checkpoint/Restart

The most popular fault-tolerance technique for long-running applications is checkpoint/restart (CR). CR applications periodically take a snapshot of the system and save it into secondary storage (*checkpointing phase*). In the event of a failure or migration, the current state is replaced with the previously stored state and execution resumes from the last checkpoint (*restoration phase*). Several CR mechanisms at different levels (kernel, library, application) have been developed for CPUs [6] [7] [8].

Currently, all of the above-discussed mechanisms are not feasible on NVIDIA GPUs due to the absence of particular operating system and relevant driver/runtime APIs to extract computation state inside the kernel. In spite of these limitations, researchers have developed a handful of GPU CR schemes in recent years. CheCuda [4] was the first GPU CR scheme implemented in 2009. It uses Berkeley Lab Checkpoint/Restart (BLCR) library [8] for checkpointing system state. But, because this library does not support CUDA contexts, it backups and destroys CUDA contexts before checkpointing, then runs BLCR checkpointing and finally reallocates all destroyed GPU contexts. In 2011, Nukada et al. developed a new CUDA CR library (NVCR) [5] that is transparent to applications. NVCR does not make changes in CUDA context's addresses after reallocation so, it is not necessary to recompile applications. CheCL [9] is another application that follows CheCuda's approach but it was designed specifically for OpenCL applications. Another technique for OpenCL-accelerated applications is VOCL-FT [10] which protects co-processor's memory from silent data corruption using virtualization layer APIs. It validates device data by monitoring GPU ECC output from host side and efficiently recovers bit-flips by re-playing faulty epoch.

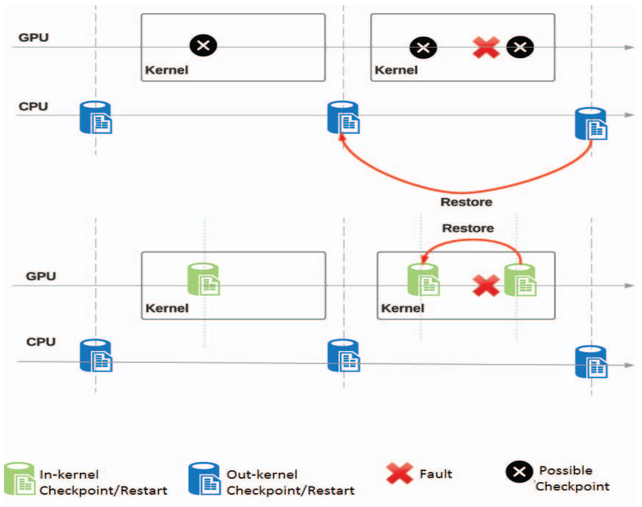


Figure 3. Comparison of in-kernel vs out-kernel checkpoint/restart schemes.

In the event of failure, all of the above-discussed CR techniques reload GPU state and re-launch kernels from the beginning. In other words, they are not able to reload thread's computing state inside the kernel. We refer to these approaches as *out-kernel* CR schemes. Figure 3 distinguishes the two CR methodologies – in-kernel and out-kernel CR.

C. In-kernel Checkpoint/Restart

Although newer NVIDIA GPUs are equipped with Single-bit Error Correction - Double-bit Error Detection (SEC-DED) ECC, a recently published paper showed that this is not well suited for modern DRAM subsystems and there still exists a noticeable amount of undetected errors in large-scale systems [11]. Additionally, D.Tiwari et al. observed that SEC-DED causes non-negligible Silent Data Corruption (SDC) in GPU clusters because it cannot protect all GPU memories such as queues, scheduler, flip-flops, etc, [12]. Hopefully, ECC can recover these errors, but there are some real-world applications such as molecular dynamics that prefer not to enable ECC because it has side effects on performance, storage, and power [13]. Another fact is that GPU SEC-DED cannot correct multiple bit errors. Recently published experience report on the Titan supercomputer [14] shows that MTBF of double-bit errors is about 160 hours but assuming next generation of supercomputers with more number of nodes, this time is expected to drop by an order of magnitude. In that case, long-running applications, specifically non-iterative ones like CCSD in NWCHEM [3] will benefit significantly from an in-kernel recovery system.

HKC [15] is a hybrid CPU/GPU checkpoint/restart which was proposed as an in-kernel GPU checkpointing scheme.

[3]http://www.nwchem-sw.org/index.php/Benchmarks

While HKC claims that it can recover GPU state at system-level, it uses debugging API which adds additional time overhead to the system and checkpoint/restart is handled by the CPU rather than the GPU. Jiang et al. [3] present a data structure and mechanism to save/restore computation state that is located in different GPU memories. Their contribution was a stepping stone in *cudaCR* development. However, their application-level implementation seems to be restricted to iterative applications. Once a failure is detected inside the kernel, it breaks the kernel and restoration phase happens in the next iteration of that kernel. They use kernel break as a way to synchronize blocks at checkpointing time.

In the next section, we describe our fully *in-kernel* CR scheme, *cudaCR*, that addresses the above deficiencies. We shift checkpoint/restart from the CPU to the GPU. Our application-level scheme is able to save computation state, which is distributed over the entire GPU memory hierarchy, anywhere inside the kernel efficiently. In event of a failure, it can restore in-kernel state and resume the execution from the last checkpoint somewhere inside the GPU kernel. Also, by defining a new algorithm and data structure, we make cudaCR asynchronous with respect to the thread blocks.

III. Implementation

In this section, we describe the algorithm and implementation details of *cudaCR*. Specifically, we discuss the pre-compiler transformations to inject checkpoints and optimizations to reduce the overhead of checkpointing.

A. Necessary and sufficient conditions for in-kernel CR

To ensure that full in-kernel CR works correctly, all threads have to synchronize at checkpoints and during recovery. We will show by example what might happen if there is no synchronization among threads. Figure 4 shows a segment of device code with two checkpoints. If no error is detected at the two checkpoints, shared variable s will finally have a value of 6. Since threads may not execute simultaneously and there is no guarantee on their execution order in CUDA, it is possible that ***thread 0*** meets the second checkpoint earlier than ***thread 1***. Assuming there is no error, ***thread 0*** does checkpointing and continues execution. Now, suppose an error occurs in the system, e.g. bit-flip in variable s. Then, ***thread 1*** detects an error as part of its checkpointing and jumps to the first checkpoint and reloads s with value 3 and continues execution. In the absence of ***thread 0***, since ***thread 0*** has already finished its computation, the final value of s would be 5. If this situation is reversed, i.e. ***thread 1*** meets the second checkpoint before ***thread 0***, the final value would be 4. This example illustrates code vulnerability due to the lack of synchronization between threads of a block in CR. Further extending this example from threads of a block to threads of different blocks in a grid and replacing shared variable s with a global variable g, we can infer that a similar problem might occur if there is no synchronization among all thread blocks.

```
__shared s;
if ( threadIdx.x == 0 )
    s = 3;
__syncthreads();
checkpoint();       //first checkpoint
if ( threadIdx.x == 0 )
    s = s + 1;
__syncthreads();
if ( threadIdx.x == 1 )
    s = s + 2;
checkpoint();       //second checkpoint
```

Figure 4. An example of the synchronization problem in the restoration phase.

Since CUDA supports synchronization between threads within a block, we can use `__syncthreads()` before checkpoint modules but it's mandatory for all threads in that block to participate in checkpointing/restoration. So, our *first condition* is as follows.

1) *Checkpoints must be seen by all participating threads.*

In other words, the user is not allowed to insert checkpoints in a conditional statement or code fragments unreachable by all threads.

Unfortunately, we cannot expand the same idea for all threads (threads in different blocks) because CUDA does not support synchronization among blocks. Therefore, our second constraint is defined as follows.

2) *Threads of a block are not allowed to modify a global memory location that has been accessed by another block.*

This condition simply states that applications cannot have block interference in write transactions. By implicitly separating blocks, we can achieve asynchronous checkpointing/restoration. Asynchronous blocks' checkpoint/restart also has an advantage – in the event of a failure in the global environment, it is not required for all the blocks to redo their computations, only the block that has modified that location has to redo its computation.

B. Checkpointing storage

For in-kernel CR, we have to store sufficient information about the kernel for a full recovery. This information is distributed across the different GPU memories, precisely, in three domains – thread's private memory (this includes registers and local memory), shared memory, and global memory. It is essential to allocate sufficient temporary storage to backup the data across these three memory domains. If the system demands more memory or the data is not modified for a sufficiently long time, the backup data might shift to hard disk as permanent storage. CPU main memory and GPU global memory are two alternatives for temporary checkpoint storage. We chose GPU global memory in our implementation because, in the *restoration phase* and *checkpointing phase*, we don't have to deal with low throughput PCIe for data movement between CPU and GPU. Also, this design choice moves CR completely to the GPU and it can be handled independently of the CPU.

C. Algorithm

Once checkpoint storage and essential information for backup are determined, the user has to insert checkpoints based on the first condition outlined in the previous section. This is the only task the user has to perform to enable cudaCR. After that, a *precompiler* makes changes to both the host and device codes as part of the application-level checkpoint/restart procedure. The required changes on the host side are minimal. It is essentially allocating memory for backup and instantiating data structures for CR. Most of the changes happen on the device side. We classify these transformations into two categories based on the data.

- For **global data**, once it is allocated in the device memory, the precompiler allocates the same amount of memory as backup and saves its beginning address and its size in a global backup list (`CR_gList`). The `CR_gList` is passed to device's threads along with other kernel arguments.
- Since **shared and private data** are initialized inside the kernel, we can also allocate their backup on the device side. Using `cudaMalloc()` to allocate backup memory is too time-consuming; so we transfer shared and private memory allocations from the device to the host. For shared data, we allocate a memory of size `number_of_blocks` × `max_shared_memory_per_block`. The precompiler also creates a structure out of private data and allocates a vector of this struct with the size of `number_of_threads` × `sizeof(struct)`.

After the precompiler transforms both the host and device codes for final compilation, these new codes are linked with our checkpoint/restart library. At run-time, when the execution stream meets a checkpoint, it synchronizes the threads in all the blocks and checks for faults. If at least one thread detects an error, the entire block goes to the restoration phase. Otherwise, it goes to the checkpointing phase.

1) In the *restoration phase*, all the threads perform a synchronous jump to the previous checkpoint label. Then, each thread invokes a module in the checkpoint/restart library to copy the thread's private information from its backup to its original location. The shared memory is also reloaded from its backup located in the device memory. To parallelize the shared memory copy, data movement is split among the threads in a block. For global memory restoration, only those global locations that belong to the faulty block are reloaded. Finally, the faulty block resumes its execution.
2) In the *checkpointing phase*, we perform all the data movements discussed above but in the reverse direction. Thread's private memory, block's shared memory, and the associated global information are checkpointed into their temporary backup. After checkpointing, execution resumes with no branch.

D. Incremental checkpointing

When application's execution meets checkpoints, it has to save or restore information. The best-case scenario is to save/restore only those sections that have been modified since the previous checkpoint. This method, also known as incremental checkpointing, reduces CR time significantly but requires us to continuously keep tracking and registering data into the corresponding data structure. By *registering*, we mean recording what variables are created and where they are located. This information helps CR applications to quickly find data instead of scanning the entire memory. All registered data need not be copied during checkpoints so we need another function called tracking. *Tracking* here means determining which data has been modified from the previous checkpoint. It is similar in spirit to the idea of dirty bits in caches.

For private data registration, the precompiler scans the device code and finds all local data initializations (including variables, pointers, arrays, structs and so on) and creates a structure to save this information. To find out which data are shared between threads in a block, the precompiler scans the device code for the `__shared__` directive. For global data, the registering phase takes place on the host side. All global arrays and their backups are registered in lists (`CR_gList`) and passed as arguments to kernels. This structure contains the beginning address of the original and backup arrays along with their sizes. In the device code, the precompiler finds every global write transaction. This information is recorded into a data structure called `CR_gAddr`. Every thread has one instantiation of `CR_gAddr`. During the checkpointing phase, each thread scans its `CR_gAddr`. By using these data structures, each thread is able to find the backup location of its accesses. Then it copies the data from the original location to the relevant backup location. The same procedure happens during the restoration phase but with the data movement in the reverse direction.

E. Optimization

One drawback of the current implementation is when an application expands and launches a kernel with more threads and blocks, the amount of backup memory required for shared and private data increases. For example, if an application with 100 million threads launches a kernel with a block size of 32×32, it will allocate nearly 5 GBs of shared memory backup (assuming 48 KB shared memory per block). To make *cudaCR* scalable, we apply a space optimization. The main idea behind this optimization is to reuse the backup locations that have already been allocated for threads that have finished execution. Therefore, instead of allocating `number_of_blocks` $\times$ `max_shared_memory_per_block` for shared memory backup, the precompiler only allocates `number_of_SMs` $\times$ `max_blocks_per_SM` $\times$ `max_shared_memory_per_block`. At run-time, once a block finishes its task, its backup data is no longer required since it has passed all checkpoints successfully. So, the next block will use the shared memory backup of the corresponding SM. To implement this optimization, *cudaCR* has to know which active block is running on which SM. Unfortunately, CUDA run-time API does not have access to this information. Therefore, we inline low-level PTX instructions to dynamically assign backup locations to active blocks. We apply a similar approach to private backup memory where the precompiler allocates `number_of_SMs` $\times$ `max_threads_per_SM` $\times$ `sizeof(struct)` bytes for private memory backup instead of `number_of_threads` $\times$ `sizeof(struct)`. By indexing threads in blocks, the precompiler can uniquely assign a pre-used (but not-in-use) chunk of private memory backup to each active thread. In summary, with this optimization, the amount of backup storage for shared and private memories does not depend on the application's size but only on the GPU being used.

IV. Results and Discussion

In this section, we briefly describe the various benchmarks used for evaluating cudaCR and their access patterns. Then, we present time and storage overhead of checkpoint/restart using cudaCR on these test cases.

A. Experimental setup

We evaluate our implementation on NVIDIA Tesla K40 GPU with 15 streaming multiprocessors. The shared memory size is 48 KB per block and the global memory bandwidth is 288 GB/s. The host-side CPU is a dual-socket Intel Xeon E5-2630 v3 with 8 cores per socket for a total of 16 cores. We use CUDA `nvcc` version 7.5 as the GPU compiler with full optimization enabled (`-O3` flag).

B. Case studies

To validate the integrity and measure the overhead of our proposed scheme, we test *cudaCR* on three different CUDA benchmarks detailed below with different data access patterns.

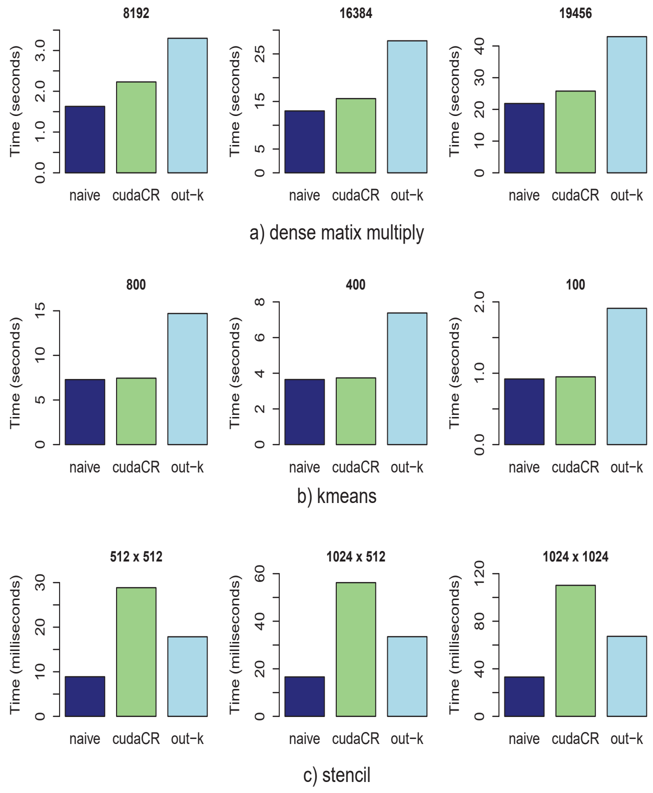


Figure 5. Application run time of naive and CR-enabled tests for different benchmarks. Light blue bar denotes zero-overhead out-kernel CR in the error-polluted test. The numbers on top of the bars denote problem sizes.

- **3-D Stencil computation** – An iterative Jacobi stencil operation on a regular 3-D grid. We chose three grids of sizes $512 \times 512 \times 400$, $1024 \times 512 \times 400$, and $1024 \times 1024 \times 400$ for our experiments. This is a memory-bound computation with a low flop:byte ratio. This benchmark is from the Parboil GPU benchmark suite [16].
- **Dense matrix multiply** (`sgemm`) – This benchmark performs a dense matrix multiplication using the standard BLAS format on matrices of sizes 8192, 16384, and 19456. Matrix multiply is one of the most popular and well-studied algorithms that are highly compute bound. This benchmark is also from the Parboil GPU test suite.
- **k-means clustering** – This is a popular iterative clustering algorithm used extensively in data-mining. We choose k to be 100, 400, and 800 for our tests. This benchmark is from Rodinia GPU test suite [17].

Since the duration of the kernels in the above benchmarks is far from a realistic system's MTBF, it's unlikely for them to encounter any failure during run-time. So, we artificially inject errors into the system. This is done by modifying select variables with constant probability in different locations including local, shared, and global data for randomly selected threads before checkpoints. For each benchmark, we conducted three tests. (1) Run the benchmark in an error-free mode. This is represented as *naive* in our results. (2)

	Backup memory without optimization (MB)				Backup memory with space optimization (MB)				
Benchmark	private	shared	global	**total**	private	shared	global	**total**	gain
stencil	18	98	3300	3416	3	< 1	3300	3303	4%
k-means	36	64	220	320	3	< 1	220	223	31%
`sgemm`	234	262	402	898	9	< 1	402	412	54%

Table I
BACKUP MEMORY REQUIREMENTS FOR BENCHMARKS WITH AND WITHOUT MEMORY OPTIMIZATIONS.

We run the error-polluted kernel without any checkpointing. (3) We apply *cudaCR* by making the precompiler changes and inserting sample checkpoints inside the kernel. Then we inject faults and output the result.

C. Results

By comparing the outputs of the aforementioned tests, we observed differences between the first test (naive case) and second test (faulty kernel without CR) as expected while the outputs of the first test (naive case) and third test (faulty kernel with CR) were identical. This experiment shows that *cudaCR* can fully recover state on all three benchmarks. Also, a longer runtime of the third test with respect to the second test shows that *cudaCR* re-computed some code fragments by some blocks for error recovery.

It is almost impossible to report a single value for time overhead and storage requirement because they depend on several factors such as the number of checkpoints, number of blocks that have encountered an error, amount of memory that has to be checkpointed, number of global memory accesses, block and grid sizes, and so on. Intuitively, we can approximate the time of an application with *cudaCR* by the following formula.

$$t_{CR} \simeq t_0 + t_{setup} + n * (\alpha_g M_g + \alpha_{sh,p} M_{sh,p} + t_d) + t_r$$

where t_0 is the application time without *cudaCR*, t_{setup} is the time needed for backup memory allocation/initialization, constructing structures, and indexing threads/blocks. t_r is the time spent on re-computing which depends on what fraction of the blocks go to the restoration phase and degree of parallelization. n is the number of checkpoints, t_d is the error detection time and M_g and $M_{sh,p}$ are the amount of memory that has to be checkpointed in the global and shared-private memories respectively. Since we did not use tracking for shared and private data, the required time to checkpoint specific amount of such data are about the same, so we merge them into a single coefficient, $\alpha_{sh,p}$. However, for global data, we do tracking that takes considerable time. In fact, for every global write, *cudaCR* registers its addresses into a list in local memory (that may reside in global memory); hence we expect a larger coefficient for global data ($\alpha_g >> \alpha_{sh,p}$).

In order to make the test cases comparable, we fix the number of checkpoints to 2 and ensure the same SM occupancy for different experiments. We also remove error detection, re-computation, and fault-injection times to calculate pure checkpointing overhead. Figure 5 shows application run-time for the naive code (dark blue bars) and CR-enabled code (green bars). Unfortunately, none of the out-kernel checkpoint/restart tools [4], [5] were open-source or compatible with newer NVIDIA architectures. So, to compare the efficiency of in-kernel CR, we simulate ideal (zero-overhead) out-kernel CR in an error-polluted test by adding kernel re-execution time and asynchronous memory copy between the device and host (checkpointing) to the naive test. This is shown by out-kernel (light blue bars) in Figure 5. The plots in the first row are for `sgemm` benchmarks with matrix dimension of 19456, 16384, and 8192 respectively. In `sgemm`, threads have limited global accesses (32 writes per thread in a 32 $\times$ 32 tile), so we observe small overheads of 17.9%, 19.2%, and 37% respectively. As the size of the matrices increases, the time for computation (t_0) increases while global accesses (M_g) remains almost constant. Hence the checkpointing overhead reduces. The second-row shows the results for *k-means clustering* with varying number of clusters (800, 400, and 100). We observe extremely small overheads of 2-3% for this benchmark because it has few global modifications per thread and more importantly, we insert checkpoints in the best locations that require minimal data for checkpointing/restoration. Unlike *k-means* and `sgemm`, *stencil* performs significantly more global memory modifications. We increase the dimension of the grid to enable more than 800 writes per thread for this benchmark to really stress *cudaCR*. The third row presents results for grid sizes $512 \times 512 \times 400$, $1024 \times 512 \times 400$, and $1024 \times 1024 \times 400$. As one might expect, due to a large number of global accesses, we observe larger overheads (nearly 200%). Comparing the overhead of cudaCR with out-kernel, it is preferable to re-start the kernel in the case of *stencil*. The time overheads demonstrate how effective *cudaCR* can be if checkpoints are inserted at appropriate locations (like *k-means clustering*) or on compute-bound applications (such as `sgemm`). However, it might not deliver good performance on applications that spend the majority of their time on global memory accesses rather than computation (bandwidth applications like *stencil*).

Table I shows the the amount of backup memory used by the three benchmarks before and after the memory optimization. For *stencil* (grid size of $1024 \times 1024 \times 400$ with a block size of 16×16), the naive code requires 3416 MB but after space optimization, the backup memory reduces to 3303 MB. Similarly, *k-means* (1 million data points with 34 double-precision features, and 400 clusters) and `sgemm` (with matrix size of 4096×4096 and block size of 8×64) use 31% and 54% less backup memory respectively. Therefore, *cudaCR* shows lower storage requirement after the memory optimization.

V. Conclusion

Due to the lack of full in-kernel GPU checkpoint/restart at the application-level, we design and implement a new scheme for NVIDIA GPUs and CUDA programming model called *cudaCR*. The proposed scheme is able to capture data universally inside the kernel at checkpoints under a specific condition. Unlike previous GPU checkpoint/restart implementations, *cudaCR* moves checkpointing/restoration task from CPU to GPU. Experiments across different benchmarks show that our scheme has low overheads in time and backup storage, especially for real and standard GPU applications. Our in-kernel checkpoint/restart is suitable for long-running kernels. Currently, our application-level CR requires the user to inject checkpoints in the code. Future work will address the challenge of designing a smart compiler that can identify the best (or approximate) locations for checkpointing.

References

[1] J. Dongarra, T. Herault, and Y. Robert, "Fault tolerance techniques for high-performance computing," in *Fault-Tolerance Techniques for High-Performance Computing*. Springer International Publishing, 2015, pp. 3–85.

[2] R. Riesen, K. Ferreira, J. Stearley, R. Oldfield, J. H. Laros III, K. Pedretti, R. Brightwell *et al.*, "Redundant computing for exascale systems," *Sandia National Laboratories*, 2010.

[3] X. Guo, H. Jiang, and K.-C. Li, "A checkpoint/restart scheme for cuda applications with complex memory hierarchy," in *Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD), 2013 14th ACIS International Conference on*. IEEE, 2013, pp. 247–252.

[4] H. Takizawa, K. Sato, K. Komatsu, and H. Kobayashi, "Checuda: A checkpoint/restart tool for cuda applications," in *2009 International Conference on Parallel and Distributed Computing, Applications and Technologies*. IEEE, 2009, pp. 408–413.

[5] A. Nukada, H. Takizawa, and S. Matsuoka, "Nvcr: A transparent checkpoint-restart library for nvidia cuda," in *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on*. IEEE, 2011, pp. 104–113.

[6] M. Schulz, G. Bronevetsky, R. Fernandes, D. Marques, K. Pingali, and P. Stodghill, "Implementation and evaluation of a scalable application-level checkpoint-recovery scheme for mpi programs," in *Proceedings of the 2004 ACM/IEEE conference on Supercomputing*. IEEE Computer Society, 2004, p. 38.

[7] J. S. Plank, M. Beck, G. Kingsley, and K. Li, *Libckpt: Transparent checkpointing under unix*. Computer Science Department, 1994.

[8] P. H. Hargrove and J. C. Duell, "Berkeley lab checkpoint/restart (blcr) for linux clusters," in *Journal of Physics: Conference Series*, vol. 46, no. 1. IOP Publishing, 2006, p. 494.

[9] H. Takizawa, K. Koyama, K. Sato, K. Komatsu, and H. Kobayashi, "Checl: Transparent checkpointing and process migration of opencl applications," in *Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International*. IEEE, 2011, pp. 864–876.

[10] A. J. Peña, W. Bland, and P. Balaji, "Vocl-ft: introducing techniques for efficient soft error coprocessor recovery," in *High Performance Computing, Networking, Storage and Analysis, 2015 SC-International Conference for*. IEEE, 2015, pp. 1–12.

[11] 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 *ACM SIGPLAN Notices*, vol. 50, no. 4. ACM, 2015, pp. 297–310.

[12] 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 *High Performance Computer Architecture (HPCA), 2015 IEEE 21st International Symposium on*. IEEE, 2015, pp. 331–342.

[13] R. M. Betz, N. A. DeBardeleben, and R. C. Walker, "An investigation of the effects of hard and soft errors on graphics processing unit-accelerated molecular dynamics simulations," *Concurrency and Computation: Practice and Experience*, vol. 26, no. 13, pp. 2134–2140, 2014.

[14] 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 *Proceedings of the international conference for high performance computing, networking, storage and analysis*. ACM, 2015, p. 38.

[15] L. Shi, H. Chen, and T. Li, "Hybrid cpu/gpu checkpoint for gpu-based heterogeneous systems," in *International Conference on Parallel Computing in Fluid Dynamics*. Springer, 2013, pp. 470–481.

[16] J. A. Stratton, C. Rodrigues, I.-J. Sung, N. Obeid, L.-W. Chang, N. Anssari, G. D. Liu, and W.-m. W. Hwu, "Parboil: A revised benchmark suite for scientific and commercial throughput computing," *Center for Reliable and High-Performance Computing*, vol. 127, 2012.

[17] 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 *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*. IEEE, 2009, pp. 44–54.