

UNIVERSITY OF CALIFORNIA,  
IRVINE

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

THESIS

submitted in partial satisfaction of the requirements  
for the degree of

MASTER OF SCIENCE  
in Computer Engineering

by

Behnam Pourghassemi

Thesis Committee:  
Assistant Professor Aparna Chandramowlishwaran, Chair  
Associate Professor Athina Markopoulou  
Assistant Professor Mohammad Al Faruque

2017



# DEDICATION

I dedicate this thesis to my lovely parents and brothers whom I haven't seen for a long time, but they have always been there to support me, motivate me, and give me sincere advice. Undeniably, I owe all my academic achievements to them, and this dedication is the smallest gratitude that I could express to them.

# TABLE OF CONTENTS

	Page
<b>LIST OF FIGURES</b>	<b>iv</b>
<b>LIST OF TABLES</b>	<b>v</b>
<b>ACKNOWLEDGMENTS</b>	<b>vi</b>
<b>ABSTRACT OF THESIS</b>	<b>vii</b>
<b>1 Introduction</b>	<b>1</b>
<b>2 Background</b>	<b>6</b>
2.1 GPU and CUDA Programming Model . . . . .	6
2.2 Checkpoint/Restart . . . . .	8
2.3 In-kernel Checkpoint/Restart . . . . .	10
2.4 Multi-GPU In-kernel Checkpoint/Restart . . . . .	12
<b>3 Implementation</b>	<b>14</b>
3.1 Necessary and Sufficient Conditions . . . . .	14
3.2 Checkpointing Storage . . . . .	17
3.3 Transformer . . . . .	18
3.4 Incremental Checkpointing . . . . .	24
3.5 Optimization . . . . .	26
<b>4 Results and Discussion</b>	<b>28</b>
4.1 Experimental Setup . . . . .	28
4.2 Case Studies . . . . .	29
4.3 Results . . . . .	30
<b>5 Conclusion</b>	<b>35</b>
<b>Bibliography</b>	<b>36</b>

# LIST OF FIGURES

	Page
1.1 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. . . . .	3
2.1 Block diagram of GPU architecture and programming model. . . . .	7
2.2 Comparison of in-kernel vs out-kernel checkpoint/restart schemes. . . . .	9
2.3 Illustration of different CR schemes with 3 GPU nodes – (a) out-kernel CR, (b) in-kernel checkpoint without in-kernel restoration, and (c) full in-kernel CR. . . . .	12
3.1 NVCC compilation flow with cudaCR transformer . . . . .	18
4.1 Application run time of naive and CR-enabled tests for different benchmarks. The number on top of the bars denote problem sizes. . . . .	32

# LIST OF TABLES

	Page
4.1 Backup memory requirements for benchmarks with and without memory optimizations. . . . .	31

# ACKNOWLEDGMENTS

First and foremost, I would like to thank my advisor prof. A. Chandramowlishwaran for her consistent guidance from the beginning of this project up to its last moments. Her unwavering assistance, indispensable advice, and technical instructions showed me the proper direction to excel in this project. My sincere gratitude goes to her because she devotedly cooperated in our publication.

I would also thank Prof. B. Demsky for his valuable time and brilliant opinions through implementation phase which have been undoubtedly effective.

My appreciation also extends to committee members, Prof. Markopoulou and Prof. Al Faruque not only because of their priceless time and review but also because I learned many fundamentals in their courses which directly or indirectly contribute to this project.

At the end, I should give credit to all referenced publishers especially those have developed checkpoint/restart tools for GPUs. Essentially, I'm thankful for Nvidia developers because they made most of their tools and frameworks open-source and well-documented to be used by researchers. I also thank contributors of SASSI tool for their descent instrumentation tool that we have exploited as part of our implementation.

# ABSTRACT OF THE THESIS

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

By

Behnam Pourghassemi

Master of Science in Computer Engineering

University of California, Irvine, 2017

Assistant Professor Aparna Chandramowlishwaran, Chair

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 de-facto 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 work, 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. This thesis presents *cudaCR* implementation in detail and evaluate the first version of that 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.

# Chapter 1

## 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 nodes 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 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 with power4 IBM processor is estimated to be between 10 to 1000 years [9]. Assuming MTBF of a single node with newer and more reliable processor to be optimistically one century, the MTBF of a machine with hundred thousand nodes drops to 9 hours and for next generation of large-scale machines with million nodes, it will drop to less than an hour. [1, 12, 5]. 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.

Today, Graphical Processing Units (GPUs) are popular as they offer both higher peak performance and higher bandwidth compared to traditional CPU processors. Additionally, shifting trends in parallel computing have resulted in supercomputers with more number of GPU nodes. Titan, the 3rd fastest supercomputer in the Top 500 list has 18,688 compute nodes and the same number of Tesla K20 GPUs<sup>1</sup>. 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 1.1 shows the breakdown of failures by category distilled from the failure history of TSUBAME2.5<sup>2</sup> 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 1.1, 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.

---

<sup>1</sup><https://www.olcf.ornl.gov/titan>

<sup>2</sup><http://mon.g.gsic.titech.ac.jp/trouble-list/index.htm>

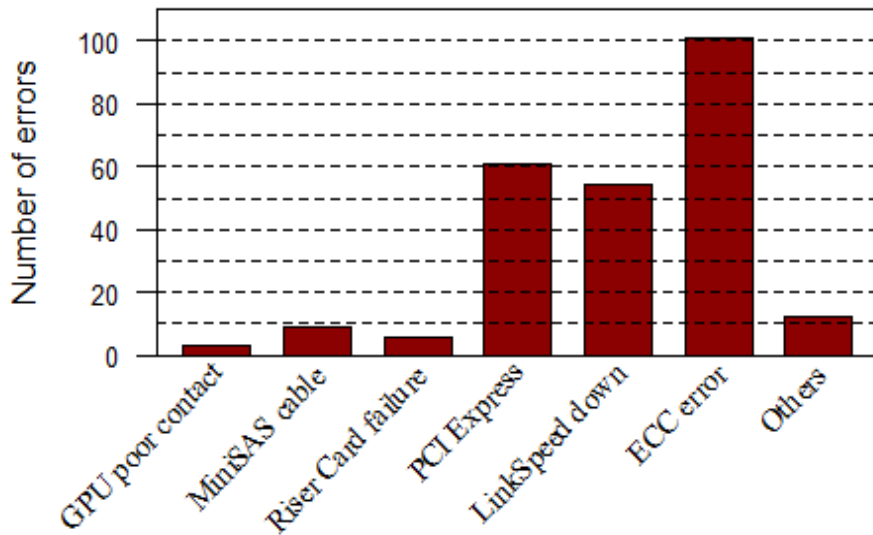
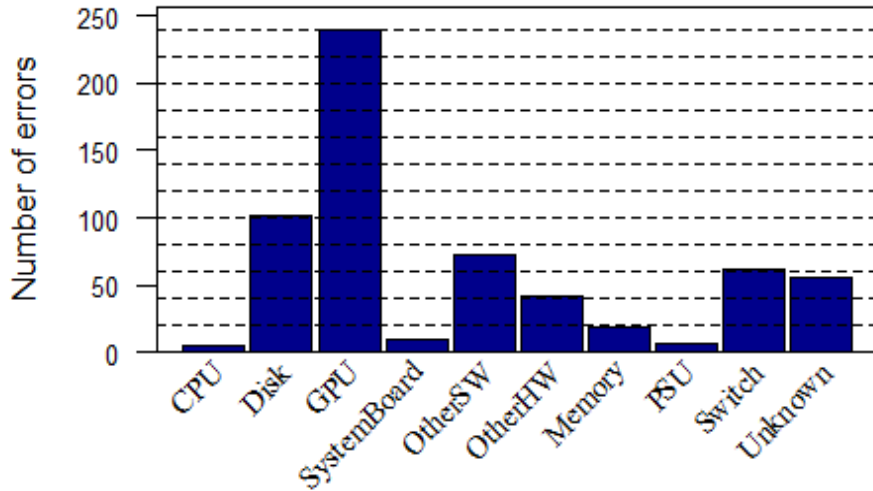


Figure 1.1: Break-down of failures in TSUBAME 2.5 from October 1, 2015 to October 1, 2016. Top: Failure distribution based on their source. Bottom: Break-down of GPU failures based on their reason.

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

- **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 internal computation state at the library level, so the first two approaches discussed above are infeasible on current GPUs [7]. Although there exist few schemes known as kernel-level GPU checkpointing [21, 10], they are in fact implemented on top of previous CPU checkpointing and save computational states 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 thesis 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 host and device source codes at compile time and adds checkpointing modules that are invoked at runtime. This thesis makes the following contributions.

- We design and implement a new scheme for in-kernel checkpointing of GPUs called *cu-*

*daCR*, where the user is only responsible for determining checkpoint locations. A transformer automatically translates both CPU and GPU codes to new codes that are capable of checkpointing and restoring the correct state (Section 3).

- 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 3).
- 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 4).

# Chapter 2

## Background

### 2.1 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. Subsequently, applications started using graphics API such as OpenGL [16] and DirectX [14] for parallel processing. In 2007, NVIDIA released CUDA (Compute Unified Device Architecture) as a new programming model to program NVIDIA GPUs in a much easier way [4]. The CUDA platform is built on top of C/C++ language and has two sets of APIs – driver and runtime.

In the CUDA programming model, GPU is referred to as *device* since it is an external component connected to the CPU which is called the *host*. CUDA programs are typically divided into two parts – (1) the serial computation in the mainstream is executed by the CPU process (host) and (2) the data parallel computation is executed on the GPU cores (device). The NVIDIA C compiler (`nvcc`) separates host and device code during the compilation process. For C/C++ programs, the host code is further compiled with host's standard

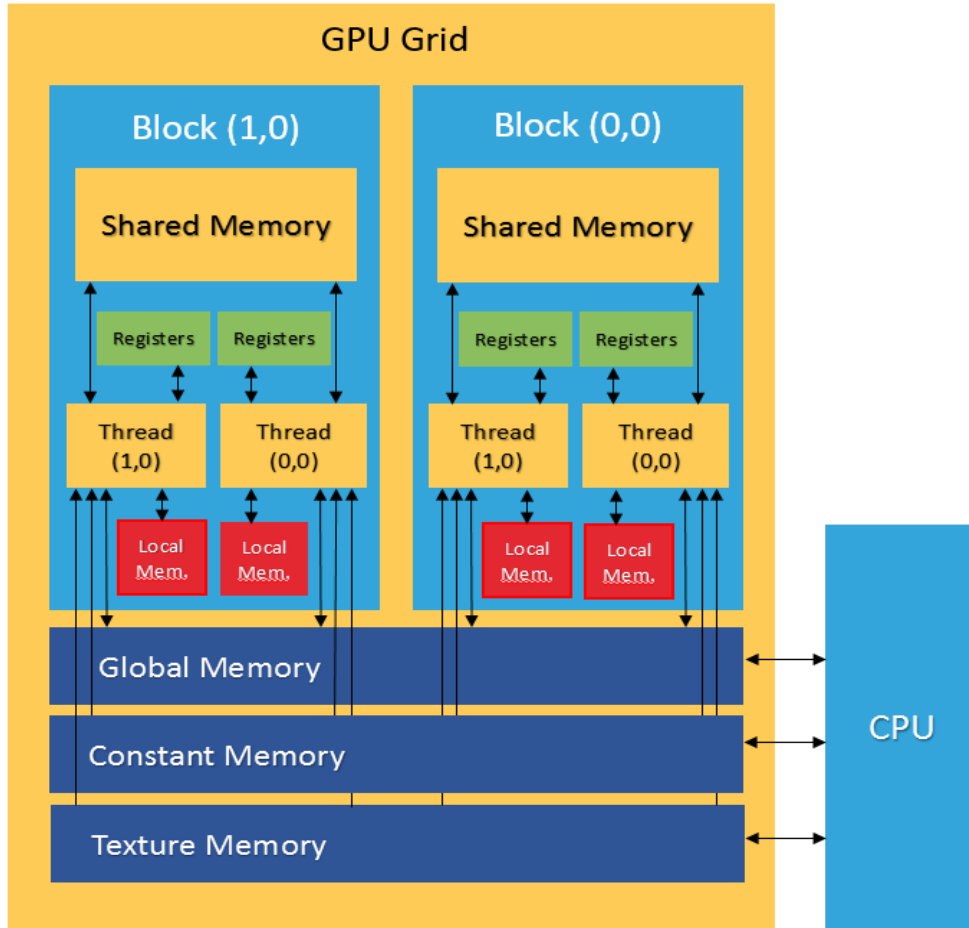


Figure 2.1: Block diagram of GPU architecture and programming model.

C/C++ compilers. The device code is written in an ANSI C extension with keywords for labeling data-parallel functions, called kernels, and then compiled using `nvcc` [4].

CUDA organizes device code using abstractions of threads, blocks, and grids. *Kernels*, functions on 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 a.k.a. 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 GPU with different access policies. *Global memory* is accessible by the entire grid (all blocks and threads within a block), *shared memory* is local to each thread block (only visible to threads inside the particular block), and *registers* are



thread-local where each thread has a limited number of them. *Local memory* is an extension of registers that resides in 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 they are read-only.

A typical CUDA program consists of four steps. Firstly, data is copied from host memory to GPU memory which is most likely the global memory. Secondly, the CPU code launches a kernel on the GPU. Thirdly, GPU threads execute the kernel in parallel on SMs. Finally, the computed results are copied back from GPU global memory to host memory. The GPU architecture and data movement between host and device along with the different memory hierarchies are illustrated in Figure 2.1.

## 2.2 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 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 have been developed for CPUs. In CR mechanisms such as Cornell Checkpoint (pre) Compiler (C3) [13], the programmer or pre-compiler injects CR code into application source code. In this approach, the programmer is required to have a good understanding of the program. However, they are usually efficient in terms of both checkpointing time and storage as the programmer is knowledgeable as to when and what information should be checkpointed [6]. Library-level mechanisms [11] reduce the burden of checkpointing on developers. They are compiled and linked to source code as a separate library. Kernel-level CR mechanisms [8] [24] periodically save process information via the operating system or hardware. Since they don't modify the

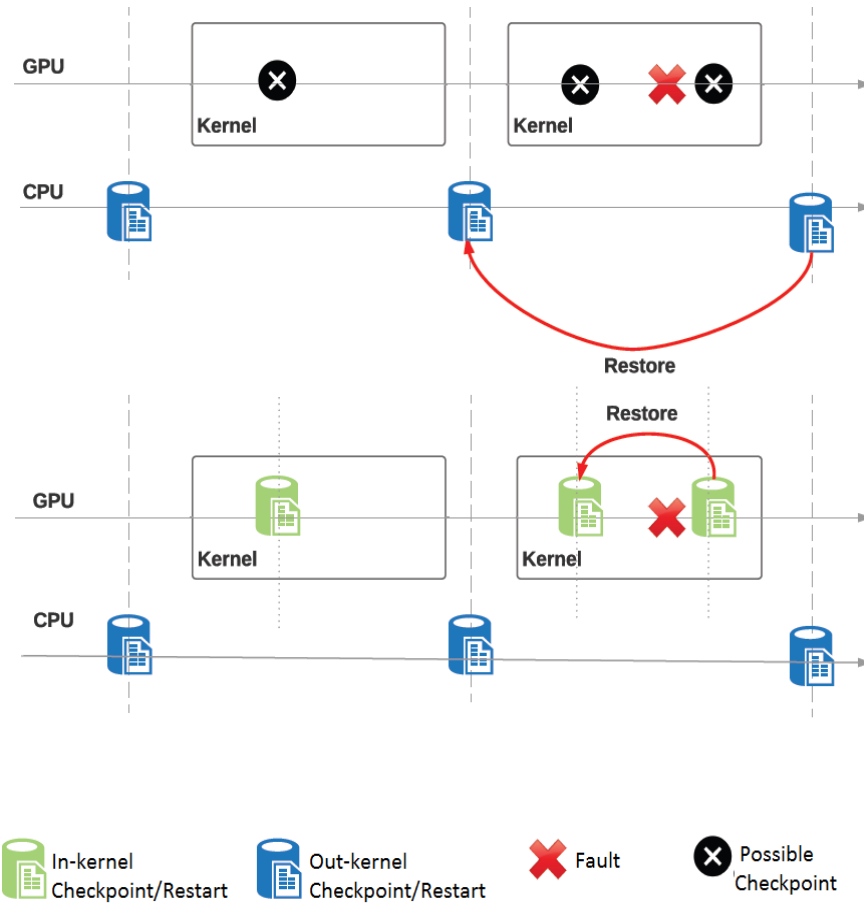


Figure 2.2: Comparison of in-kernel vs out-kernel checkpoint/restart schemes.

source code, applications remain transparent to the system.

Currently, all of the above-discussed mechanisms are not feasible on the GPUs due to the absence of particular operating system and relevant system calls. Also, we could not implement library-level mechanisms because NVIDIA’s driver and runtime APIs do not support all the necessary functions to extract computation states inside the kernel. In spite of these limitations, researchers have developed a handful of GPU CR schemes in recent years. CheCuda [21] 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) [10] that

is transparent to applications. NVCR did not make changes in CUDA context's addresses after reallocation so, it is not necessary to recompile applications. CheCL [20] is another application that follows CheCuda's approach but it was designed specifically for OpenCL applications.

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 2.2 distinguishes the two CR methodologies – in-kernel CR and out-kernel CR.

## 2.3 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 [17]. 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, [23]. Their recent measurement from epoch 2012 to August 2014 on Titan supercomputer reports more than 6 million single-bit errors. However, nearly 98% of these errors were confined to 10 cards. Even removing these faulty cards results in a high single-bit error rate of almost once per 6 minutes <sup>1</sup>. 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 [2]. In-kernel checkpoint/restart would be essential for applications whose kernel execution time is longer than SBE mean-time. Another fact is that GPU SEC-DED cannot correct multiple bit errors. Recently published

---

<sup>1</sup>GTC'15 Session S5566 GPU Errors on HPC Systems

experience report on the Titan supercomputer provides very useful information about GPU faults' types, frequency, and their locations [22]. The report 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 orders of magnitude. In that case, long-running applications, specifically non-iterative ones like CCSD in NWCHEM <sup>2</sup> 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. While HKC claims that it can recover GPU state at system-level, they use debugging API which adds additional time overhead to the system and checkpoint/restart is handled by CPU rather than GPU. They also ignore halfCTA -Cooperative Thread Array that contains running threads on SM- due to lack of corresponding APIs. Jiang et al. [7] 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. In the next iteration, this scheme does redundant computation from the beginning of the kernel up to the faulty checkpoint. Moreover, they used 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 CPU to 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 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

---

<sup>2</sup><http://www.nwchem-sw.org/index.php/Benchmarks>

respect to the thread blocks.

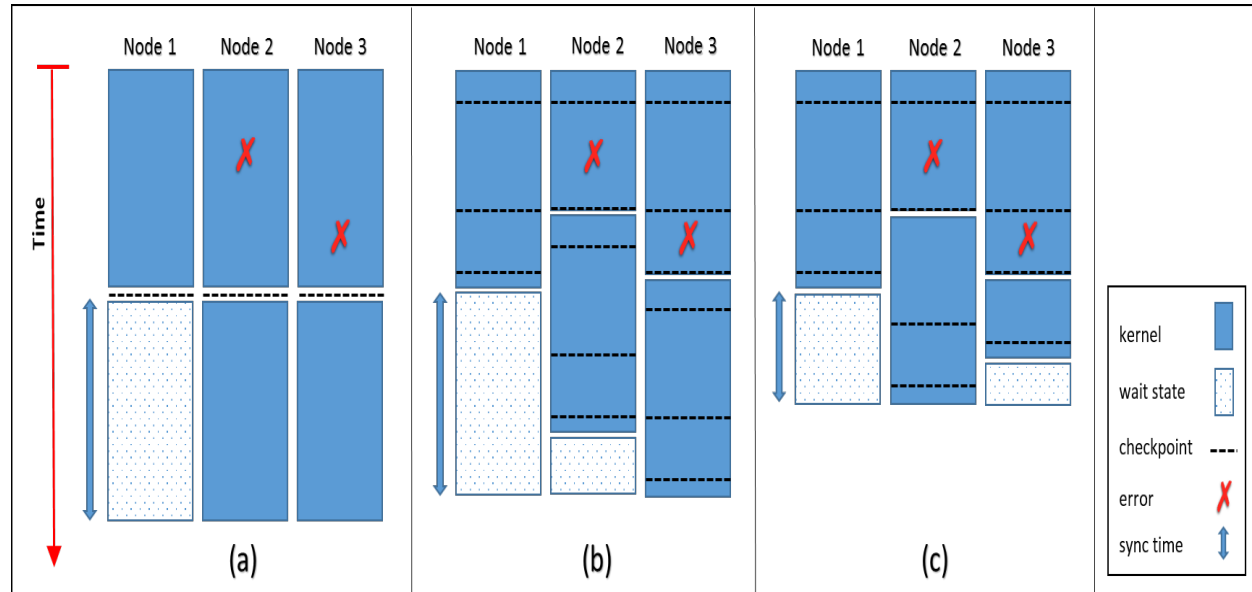


Figure 2.3: Illustration of different CR schemes with 3 GPU nodes – (a) out-kernel CR, (b) in-kernel checkpoint without in-kernel restoration, and (c) full in-kernel CR.

## 2.4 Multi-GPU In-kernel Checkpoint/Restart

In typical multi-GPU programming, we divide the application’s computations into smaller segments (grids) and distribute them among active GPU nodes. Each GPU executes the kernel on its segment of data and once they’re all done, they communicate and update shared data. This is followed by the next iteration in the case of an iterative application or a new kernel launch. Figure 2.3 illustrates different CR schemes in the context of multi-GPU programming given three GPU nodes. Here, we assume that kernels run long enough that some GPUs are likely to encounter errors within the kernels execution (or the number of GPUs is large enough that MTBF of nodes becomes comparable with the kernel lifetime). Figure 2.3 (a) shows the execution timeline of an out-kernel CR scheme where the node that successfully finished its kernel execution (*node 1*) has to wait for faulty nodes (*node 2 and 3*) to re-execute its kernel. Figures 2.3 (b) and (c) demonstrates two variants of in-kernel CR.

The former includes in-kernel checkpointing but lacks in-kernel restoration. This is similar to current state-of-the-art in-kernel CR implementations. For synchronization among GPUs, faultless nodes (*node 1*) must wait for faulty nodes (*node 2 and 3*) to restart their kernel from the beginning. Although in this mechanism, we expect less waiting time, faulty nodes compute clean sections of the code redundantly. The latter is an example of full in-kernel CR where faultless nodes (*node 1*) are held in wait state only when faulty nodes (*node 2 and 3*) re-execute corrupted computations between two checkpoints. This results in the least wait time among the three schemes.

It's evident that full in-kernel CR will have (a) less resource consumption and (b) less synchronization time. As a result, embedding in-kernel CR into long-running applications with communications will decrease the application's total run-time unless checkpointing and restoring adds significant overheads. However, implementing full in-kernel CR for GPUs is not an easy task because the grid assigned to each GPU node is split into blocks and threads which cannot synchronize at checkpointing time. Moreover, CUDA APIs are not comprehensive enough to obtain critical information such as thread's program counter, data distribution, and thread's execution stream so we have to implement the above as separate modules. Finally, in order to ensure applications compatibility with in-kernel CR, certain conditions have to be satisfied by target applications which are discussed in the next section in detail.

# Chapter 3

## Implementation

In this section, we describe the algorithm and implementation details of *cudaCR*. Specifically, we discuss the transformer that injects checkpointing code and optimizations to reduce the overhead of checkpointing.

### 3.1 Necessary and Sufficient Conditions

To ensure that full in-kernel CR works correctly, all threads have to synchronize at checkpoints and during recovery, they have to jump to the previous checkpoint simultaneously. We will show by example what might happen if there is no synchronization among threads. Code 3.1 shows a part of device code with two checkpoints. If no error is detected at either of two checkpoints, shared variable *s* will finally have a value of 6. As threads do not execute simultaneously and there is no guarantee on their execution's order in CUDA, it is possible that *thread 0* meets the second checkpoint earlier than *thread 1*. Assuming no error, *thread 0* does checkpointing and continues execution. Now, suppose an error occurs in the system. Say, for instance, bit-flips in variable *s* due to background radiation or any other

```

__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

```

Code 3.1: synchronization problem in restoration phase

hardware/software error. 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, 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 the possibility of a wrong output when there is no synchronization between threads of a block during CR. Further extending this example from threads of a block to threads of different blocks in a grid and replacing shared value *s* with a variable *g* in global memory, we can infer that a similar problem might occur if there is no synchronization among all thread blocks.

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) for two reasons. First, CUDA does not support synchronization among blocks, and second, all blocks cannot jump to a checkpoint together on recovery since all the blocks are not active during the entire execution time. This is due to the fact that the number of SMs is typically smaller than the total number of blocks in a GPU program. 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 fault in the global environment, it's not required for all blocks to redo their computations, only the block that has modified that location. Consequently, determining which part of global memory belongs to which block is a difficult task that has to be handled in the algorithm design. We discuss our methodology and implementation in detail in the next sections.

One might wonder if these conditions enforce restrictions on the user and applications but we strongly believe that our scheme is applicable to a large majority of GPU applications. We claim that following standard GPU coding practices will result in a higher likelihood of compatibility. The first condition allows the programmer to insert checkpoints anywhere in the code except inside conditional statements. Since typical and “good” GPU programs do not use many conditional statements to avoid thread divergence, programmer should be able to find many locations for inserting checkpoints. Moreover, in many GPU applications, it's not uncommon to see device code inside an if-statement (typically to prevent memory modification by threads outside of the grid's margin). The user can still insert checkpoints inside

the if-statement in these applications without violating the first condition because escaped threads do not take part in computations at all and that is why we use the terminology “**participating threads**” in our condition. Regarding the second condition, since CUDA does not provide any guarantees on the order in which the blocks execute, programmers have to write GPU code in such a way that requires no modification of global variables by different blocks to ensure the correct output (even without checkpointing). Therefore, we do not add any further restriction on the programmer and application than what is already imposed by the CUDA programming model.

## 3.2 Checkpointing Storage

For in-kernel CR, we have to store sufficient information about the kernel to make a full recovery possible. This information is distributed across the different GPU memories, precisely, in three domains – thread’s private memory (this includes registers and local memory), block shared memory, and off-chip 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 first implementation because, in the *restoration phase* and *checkpointing phase*, threads don’t have to deal with low throughput PCIe for data movement between CPU and GPU. Also, this design choice makes implementation simpler and more consistent in single-CPU multi-GPU nodes with relatively small CPU DRAM size.

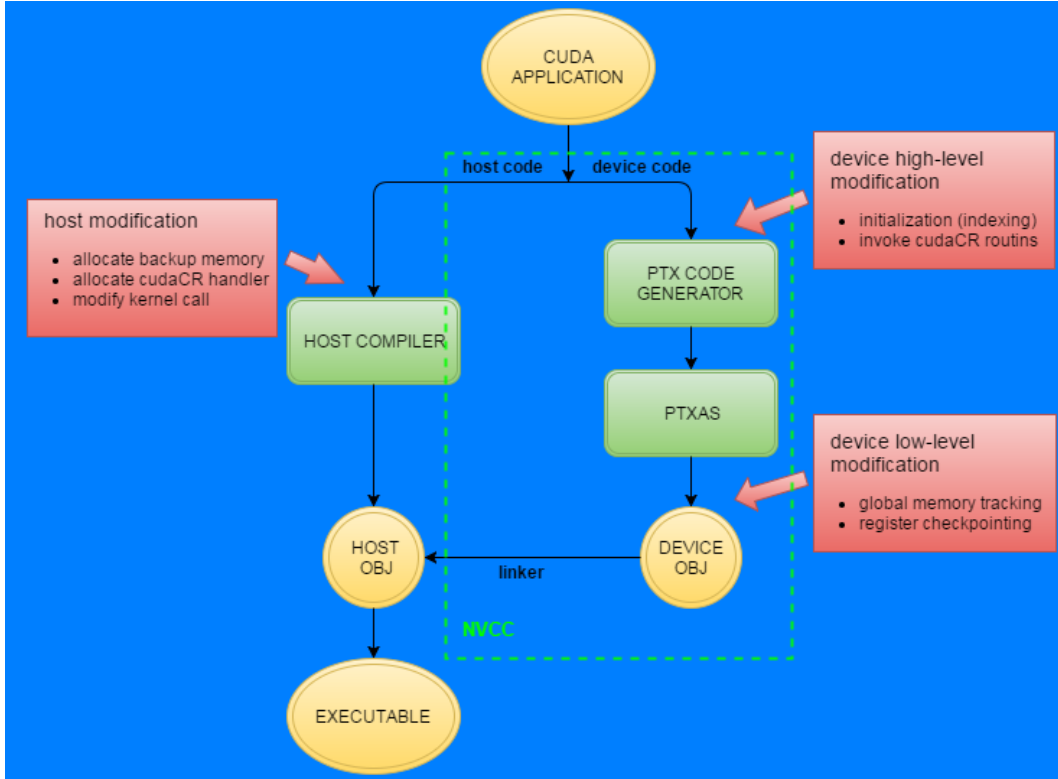


Figure 3.1: NVCC compilation flow with cudaCR transformer

### 3.3 Transformer

Once checkpoint storage and essential information for backup are determined, the user has to insert checkpoints based on the first condition outlined in Section 3.1. This is the only task the user has to perform to enable *cudaCR*. After that, a *transformer* injects code and makes changes to both host and device original source code as part of application-level checkpoint/restart procedure. Flowchart 3.1 demonstrates *cudaCR* code injection spots within CUDA-application compilation flow. At the beginning of compilation, Nvidia CUDA compiler (NVCC) separates host code from CUDA-application and passes it to the CPU compiler (e.g. GCC). The device code is then optimized and compiled to a low-level virtual machine and ISA by Nvidia PTX code generator. Nvidia assembler (PTXAS) gets cross-platform PTX ISA and generates architecture-specific machine code which can be represented by shader assembly language (SASS). The final device binary is a fat-file that may contain

one or more device-specific binary image. Later, it links with host object file and is embedded inside the host executable. Compiling with *cudaCR* causes its transformer to modify both host and device codes. These modifications are indicated by red boxes in the flowchart.

The required changes on the host side are minimal. It is essentially allocating memory for backup and instantiating data structure for CR. These modifications are all applied to the high-level source code before compilation. We classify these transformations into three categories based on the data.

- Initialize and allocate memory for *cudaCR* handler.
- For **global data**, once it is allocated in device memory, transformer allocates the same amount of memory as a backup and saves its pointer and its size into a list (*CR\_gList*). The *CR\_gList* is passed to device threads along with other kernel arguments.
- Since **shared and private data** are defined inside the kernel, we could allocate their backup on the device side. Using `cudaMalloc()` to allocate backup memory on the device side is time-consuming; so we transfer shared and private memory allocations from device to host. For shared memory, *cudaCR* transformer allocates memory of size `number_of_blocks × maximum_shared_memory_per_blocks`. It creates a structure out of private data and allocates `number_of_threads × sizeof(struct)`. Threads and blocks are linearly indexed so they know which part of backup memories belong to them.

Since CUDA allows programmers to allocate shared memory dynamically, we could not statically allocate all expected backup memory at compile-time unless transformer reserves maximum anticipated shared memory in advance. Subsequently, by shifting memory allocation from device to host, it is likely that some parts of shared backup memory remain unused. Dynamic memory allocation after kernel launch (on device side) could prevent unused storage but unfortunately, we experienced long allocation time. Therefore, we favored host static allocation to device dynamic allocation.

On the device side, transformer invokes a setup function immediately after launching the kernel. This function includes threads/blocks flattening according to their situation in the grid. This indexing requires accessing a number of special registers at PTX level but we do inline them in high-level source code. Transformer also synchronizes the threads and runs error-detection module at user specified locations. Based on error checking outcome, threads invoke either of the checkpoint or restart modules of cudaCR. Other important parts of device code modification are tracking and registers' transfer which are essentially handled at low-level SASS code. We will discuss them in the next section.

Code 3.2: Host code before transformation

```
int main(){
    dtype* d_A;
    cudaMalloc((void*)&dA, A_size);
    cudaMemcpy(d_A, h_A, A_size, cudaMemcpyHTD);
    foo<<<grid,block>>>(d_A, local_var);
    cudaMemcpy(h_A, d_A, A_size, cudaMemcpyDTH);
    return 0;
}
```

Code 3.3: Device code before transformation

```
__global__ void foo(dtype* d_A, dtype local_var)
{
    //some computation

    cudaCR(); //user injects CR

    //some computation
}
```

After transformer applied all required changes in both host and device code, the new codes are linked with our checkpoint/restart library. At run-time, when execution stream meets a checkpoint, it synchronizes threads in all blocks and checks for faults. If only one thread detects an error, the entire block goes to restoration phase. Otherwise, it goes to checkpointing phase.

1. In *restoration phase*, all threads perform a synchronous jump to the previous checkpoint label. Then, each thread invokes a module in the checkpoint/restart library to copy thread's private information from its backup to its original locations. Shared memory is also reloaded from its backup located in device memory. To parallelize shared memory copy, data movement is split among 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 *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.

Code 3.4: Host code after transformation

```

int main(){
    dtype* d_A;
    cudaMalloc((void**)&dA, A_size);
    cudaMemcpy(d_A, h_A, A_size, cudaMemcpyHTD);
    //define global memory backup handler
    GlobalList CR_gList;
    CR_gList.gPtr[0] = d_A;
    //allocate global memory backup
    dtype* CR_d_A;
    cudaMalloc(&CR_d_A, A_size);
    //registering backup
    CR_gList.gCRPtr[0] = CR_d_A;
    //copy original to backup
    cudaMemcpy(CR_d_A, d_A, A_size, cudaMemcpyDTD);
    // allocate backup mem for local vars with size of
    // number of threads * sizeof(Vals)
    Vals* CR_vals;
    cudaMalloc((void**)&CR_vals, ... );
    // allocate backup mem for shared with size of
    // number of blocks * SHARED_SIZE
    void* CR_shared;
    cudaMalloc((void**)&CR_shared, ... );
    //launch kernel with CR arguments
    foo<<<grid,block>>>(d_A, local_var,
        CR_gList, CR_vals, CR_shared);
    cudaMemcpy(h_A, d_A, A_size, cudaMemcpyDTH);
    return 0;
}

```

Code 3.5: Device code after transformation

```

__global__ void foo(dtype* d_A, dtype local_var,
GlobalList CR_gList, Vals CR_vals, void* CR_shared){
    int CR_BIndex = ... // block number in grid
    int CR_TBIndex= ... // thread number in block
    int CR_TIndex = ... // thread number in grid
    __shared__ int CR_flag; // error detection flag
    sharedList CR_shList; // shared memory handler
    globalAddress CR_gAddr; //used for tracking
    CR_flag = 0;
    //reset of code in setup function
    __syncthreads();

    //some computation

    error_detection(&CR_flag, CR_gAddr);
    __syncthreads();
    if (CR_flag){ //restoration
        restore_local(local_var, CR_vals[CR_TIndex]);
        restore_shared(CR_shList, CR_BIndex, ...);
        restore_global(CR_gAddr, CR_gList, CR_TIndex);
        CR_flag = 0;
        __syncthreads();
        goto previous_CR;}
    else{ //storing
        save_local(local_var, CR_vals[CR_TIndex]);
        save_shared(CR_shList, CR_BIndex, CR_TBIndex, ...);
        save_global(CR_gAddr, CR_gList, CR_TIndex);
        __syncthreads();}

    //some computation
}

```



## 3.4 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 *cudaCR* to continuously keep track of thread’s memory footprint. In our first version of scheme, we applied tracking and registering to the high-level source code. By *registering*, we mean recording what variables are created and where they are located (initialization). However, 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 since the previous checkpoint. It is similar in spirit to the idea of dirty bits in caches.

For private data registration, transformer scans through device code and finds all local data initializations (including variables, pointers, arrays, structs and so on) and creates a structure out of them for checkpointing. To determine which data are shared between threads in a block, the transformer scans device code for `__shared__` directive. We checkpoint all private and shared memory regardless of incremental checkpointing. The reason why we remove incremental checkpointing from this part is that incremental checkpointing requires constant monitoring of thread’s accesses to these locations. This tracking likely takes more time if we checkpoint all of them because threads access to these memories many times in typical applications. Besides, they usually comprise a small portion of application’s data. For example, to understand which registers have been modified since the previous checkpoint, recording register writes is obviously time-consuming especially when we have a loop that updates one register (loop counter) numerous times. So, we prefer to copy all live registers at checkpointing time. Shared memory also is accessed more often and has a smaller size in comparison with global data. To further improve data-movement overhead, we make shared memory bounded via two registers namely, `min_address` and `max_address` which contain

the beginning and final addresses of used shared memory respectively. Once transformer finds shared variables declaration or shared array declarations, it checks their beginning and final addresses; if they were beyond `min_address` and `max_address`, the transformer updates these registers to completely envelop declared shared data. Consequently, instead of the entire available shared memory, *cudaCR* only copies portions of that.

For global data, registering phase takes place on the host side implicitly. All global arrays and their backups are registered in a list (*CR\_gList*) and passed as arguments to kernels. This structure contains the beginning address of original and backup arrays plus their size. In the device code, transformer finds every global write transaction. This information is recorded into a data structure called `CR_gAddr`. Every thread has one instantiation of the `CR_gAddr`. The size of `CR_gAddr` depends on the number of global transactions each thread performs. During 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 data from the original location to the relative backup location. The same procedure happens during restoration phase but with the data movement in the reverse direction. Pseudocode segments in Code 3.2 and 3.3 demonstrate some of the above transformations on an example code in Code 3.4 and 3.5.

We experience tracking memory footprint at high-level source code gets sophisticated for large and complex codes. Particularly, when the user operates with pointers or calls many nested device functions, transformer might not detect global memory transactions. So, in our second version of implementation, we shift tracking task to the lower level of source code “SASS”. We exploit a recent developed SASS instrumentation tool, SASSI [18], to inject our tracking module into SASS code. We embed our tracking module in SASSI handler which is invoked after every device memory transaction. This module records every fine-grained *1 – byte*, *2 – byte*, *4 – byte*, *8 – byte* and *16 – byte* global memory writes into a corresponding container. During checkpoint or restart, *cudaCR* uses same width memory transaction for

data movement. Although SASSI runs at hardware speed, its current implementation still has significant run-time overhead. The main factor is because SASSI uses stack to pass their objects so it's ABI-compliance and register-spilling has poor performance. Especially, in our case that tracking module doesn't have much time-overhead by itself but needs to invoke many times.

### 3.5 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  $32 \times 32$ , assuming 48KB shared memory per block, it will allocate nearly 5GB shared memory backup and probably will need more for their private data. To make our scheme scalable, we apply a space optimization to our first version. The main idea behind this optimization is to reuse the backup spaces that have already been allocated for accomplished threads. To fulfill this idea, instead of allocating `number_of_blocks`  $\times$  `maximum_shared_memory_per_blocks` for shared memory backup, transformer allocates `maximum_number_of_blocks_per_SM`  $\times$  `number_of_SMs`  $\times$  `max_shared_memory_per_block`. In other words, it allocates memory equivalent to the total amount of available shared memory on a chip. At run-time, once a block finishes its task, we don't require its backup data since it passes all checkpoints successfully. So, the next block that is replaced with an accomplished block will use the shared memory backup of the corresponding SM. The transformer uses *smid* special register (with inline PTX instructions) and device locks to uniquely enumerates active blocks. We apply a similar approach to private backup memory. The transformer allocates `number_of_SMs`  $\times$  `maximum_number_of_threads_per_SM`  $\times$  `sizeof(struct)` bytes for private memory backup instead of `number_of_threads`  $\times$  `sizeof(struct)`. By index-

ing threads in blocks and knowledge of the executing SM's ID, the transformer 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 the GPU architecture (such as the number of SMs and the maximum number of threads per block). For example, if a kernel with millions of threads is running on GTX980 (Maxwell, compute capability = 5.2, maximum shared memory per SM = 96KB, SMs = 16, maximum number of threads per block = 2048, max number of 32-bit registers = 255), it doesn't need more than 35MB backup for both shared and local registers which is trivial compared to its 4GB device memory. Neglecting this small amount of memory, applications can roughly use half of device memory for their global data (the other half is used for global backup).

Moreover, in our second version, we used unified memory and unified virtual addressing that is available in newer GPU architectures (compute capability 3.0 and more) to address global memory limitation. In this case, (1) devices can use their memory entirely, (2) managing backup memory in mutli-gpu nodes is simpler (because devices share the same virtual memory address), (3) host side can use backup memory while kernel is running, (4) backup memory could shift to host side and driver abstracts this transfer. So, in this case, it's possible to recover data in the event of GPU failure that needs to reboot the device.

# Chapter 4

## Results and Discussion

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

### 4.1 Experimental Setup

We evaluate our implementation on NVIDIA Tesla K40 GPU with 15 streaming multiprocessors. The shared memory size is 48KB 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).

## 4.2 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.

- **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 is highly compute bound. This benchmark is from the Parboil GPU benchmark suite [19].
- **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 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 [3].

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. Designing error-detection module for our scheme was not our contribution in this work however we simulate this in two separate ways based on different error-detection approaches. Generally, error-detection can be applied at software level or hardware level. At the software level, error-detection routine usually uses an application-level checksum to detect errors, for example, it checks parity and reports data corruption. To simulate this, we change the value of a number of variables (with constant probability in different locations including local, shared, and global memory) for randomly selected threads immediately before selected checkpoints. Then, in the error-detection routine, we set the

`CR_flag` (refer to code in Code 3.5) of that block automatically. In the other methodology, the system uses hardware error detector and reports its source. For example, GPU ECC can detect double-bit errors and Nvidia driver writes its address into the InfoRom. To simulate this, we launch a tiny fault-injection kernel on a different stream simultaneously. We control the kernel time to be approximately similar to the benchmark kernel’s time. Then we change some value in the global locations (with constant probability in original kernel’s domain) and report their address in a global list. For error-detection, if each thread accesses either of these locations, we set the `CR_flag` of that block. By triggering errors’ threshold and manipulating their location, we are able to determine which threads and when they encounter an error.

For each benchmark, we conducted three tests. (1) We run the benchmark in an error-free mode and save the output. This is represented as *naive* in our results. (2) We save the output of the error-polluted kernel without any checkpointing. (3) We apply *cudaCR* by making precompiler changes and inserting sample checkpoints inside the kernel. Then we inject faults and output the result.

## 4.3 Results

By comparing outputs of aforementioned tests, we observed differences in the output between the first test (naive case) and second test (faulty kernel without CR) as expected while there was no difference between the outputs of the first test (naive case) and third test (faulty kernel with CR). This experiment shows that *cudaCR* can fully recover state on all three benchmarks. Also, slightly longer run-time of the third test with respect to the second test shows that *cudaCR* re-computed some code fragments by few 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

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

Table 4.1: Backup memory requirements for benchmarks with and without memory optimizations.

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. Intitively, 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 any CR,  $t_{setup}$  is the time needed for backup memory allocation/initialization, constructing structures, and indexing threads/blocks.  $t_r$  is the time spent on re-computing. It also depends on what fraction of blocks go to restoration phase and how much parallilization we could have.  $n$  is the number of checkpoints,  $t_d$  is error detection time and  $M_g$  and  $M_{sh,p}$  are the amount of memory that has to be checkpointed in global and shared-private memories respectively. Since we didn't track 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) and recovers it during CR; hence we expect a much greater coefficient for global data ( $\alpha_g \gg \alpha_{sh,p}$ ). Hopefully, caching in global memory reduces the global data coefficient,  $\alpha_g$ , but it still contributes significantly to *cudaCR* performance.



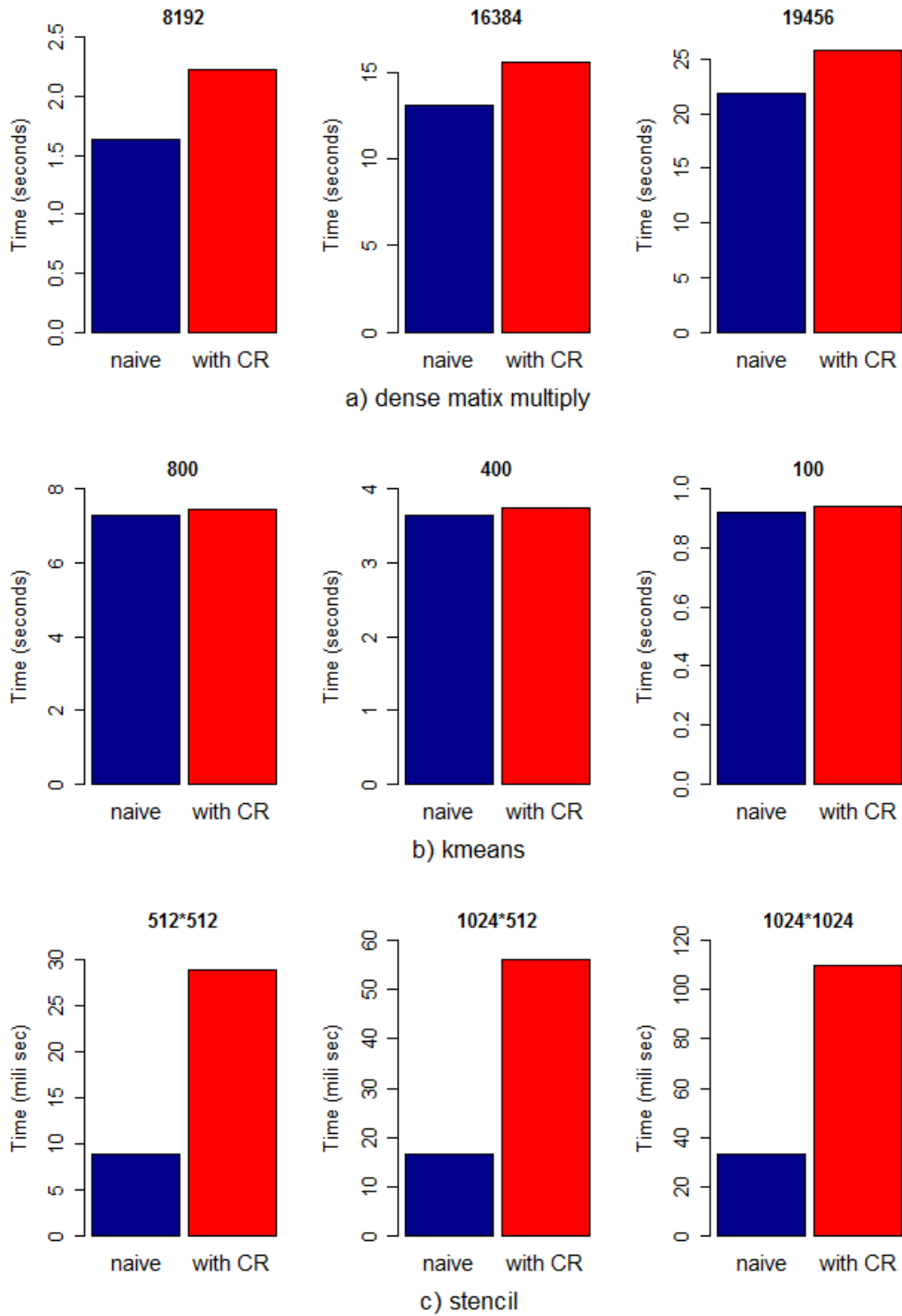


Figure 4.1: Application run time of naive and CR-enabled tests for different benchmarks. The number on top of the bars denote problem sizes.

In order to make test cases more comparable, we fix the number of checkpoints to 2 and ensure same SM occupancy for different experiments. We also remove error detection, re-computation portion, memory allocation (because it’s done once), and fault-injection from time overhead to calculate pure checkpointing overhead. Figure 4.1 shows application run-time for naive code (blue bars) and application run-time after applying first version of *cudaCR* to code (red bars).

The plots in the first row are for *sgemm* benchmarks with matrix dimension of 19456, 16384, and 8192. In *sgemm*, threads have limited global accesses (32 writes per thread in tile  $32 \times 32$ ) but have plenty of floating-point operations, so we observe small overheads of 17.9%, 19.2%, and 37% respectively. As the size of matrices increases, the time for computation increases ( $t_0$ ) while global writes remains almost constant ( $M_g$ ) and hence the checkpointing overhead reduces a little. Second-row shows results for *k-means clustering* with varying number of clusters (800, 400, and 100). We get an impressive little overhead of 2-3% for this benchmark because it has a few global modifications per thread and more importantly we insert checkpoints in the best locations that need minimal data for store/restoration. Unlike *k-means* and *sgemm*, *stencil* performs much more global memory modification. We increase the dimension of the grid to have more than 800 writes per thread for this benchmark to really stress *cudaCR*. The third row presents results from grid sizes of  $512 \times 512 \times 400$ ,  $1024 \times 512 \times 400$ , and  $1024 \times 1024 \times 400$ . As one might expect, due to a large number of global accesses, we observe larger overheads (near 200%). Reported time overhead demonstrates how effective *cudaCR* will be if checkpoints are inserted in 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*).

To evaluate our discussed memory optimization, we apply it on all three benchmarks. Table 4.1 shows the amount of backup memory used in the three benchmarks. For *stencil* (grid

size of  $1024 \times 1024 \times 400$  with a block size of  $16 \times 16$ ), naive code requires 3416 MB but after space optimization, backup memory reduces to 3303 MB. Similarly, *k-means* (1 million data point with 34 double-precision features, and 400 clusters) and `sgemm` (with matrix size of  $4096 \times 4096$  and block size of  $8 \times 64$ ) use 31% and 54% less backup memory respectively. Therefore, *cudaCR* shows extremely low storage requirement after memory optimization.

# Chapter 5

## 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 everywhere 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 in the multi-GPU environment.

Our current 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. Also, the current version of *cudaCR* still has few limitations regarding performance and register data-movement in big codes that need to be addressed in the next version.

# Bibliography

- [1] K. Bergman, S. Borkar, D. Campbell, W. Carlson, W. Dally, M. Denneau, P. Franzon, W. Harrod, K. Hill, J. Hiller, et al. Exascale computing study: Technology challenges in achieving exascale systems. *Defense Advanced Research Projects Agency Information Processing Techniques Office (DARPA IPTO), Tech. Rep*, 15, 2008.
- [2] R. M. Betz, N. A. DeBardleben, 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*, 26(13):2134–2140, 2014.
- [3] 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*, pages 44–54. IEEE, 2009.
- [4] S. Cook. *CUDA programming: a developer’s guide to parallel computing with GPUs*. Newnes, 2012.
- [5] J. Dongarra, T. Herault, and Y. Robert. Fault tolerance techniques for high-performance computing. In *Fault-Tolerance Techniques for High-Performance Computing*, pages 3–85. Springer International Publishing, 2015.
- [6] I. P. Egwutuoha, D. Levy, B. Selic, and S. Chen. A survey of fault tolerance mechanisms and checkpoint/restart implementations for high performance computing systems. *The Journal of Supercomputing*, 65(3):1302–1326, 2013.
- [7] 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*, pages 247–252. IEEE, 2013.
- [8] P. H. Hargrove and J. C. Duell. Berkeley lab checkpoint/restart (blcr) for linux clusters. In *Journal of Physics: Conference Series*, volume 46, page 494. IOP Publishing, 2006.
- [9] S. S. Mukherjee, J. Emer, T. Fossum, and S. K. Reinhardt. Cache scrubbing in microprocessors: Myth or necessity? In *Dependable Computing, 2004. Proceedings. 10th IEEE Pacific Rim International Symposium on*, pages 37–42. IEEE, 2004.

- [10] 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*, pages 104–113. IEEE, 2011.
- [11] J. S. Plank, M. Beck, G. Kingsley, and K. Li. *Libckpt: Transparent checkpointing under unix*. Computer Science Department, 1994.
- [12] 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.
- [13] 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*, page 38. IEEE Computer Society, 2004.
- [14] A. Sherrod. *Beginning DirectX 11 game programming*. Cengage Learning, 2011.
- [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*, pages 470–481. Springer, 2013.
- [16] D. Shreiner, B. T. K. O. A. W. Group, et al. *OpenGL programming guide: the official guide to learning OpenGL, versions 3.0 and 3.1*. Pearson Education, 2009.
- [17] V. Sridharan, N. DeBardleben, S. Blanchard, K. B. Ferreira, J. Stearley, J. Shalf, and S. Gurusurthi. Memory errors in modern systems: The good, the bad, and the ugly. In *ACM SIGPLAN Notices*, volume 50, pages 297–310. ACM, 2015.
- [18] M. Stephenson, S. K. Sastry Hari, Y. Lee, E. Ebrahimi, D. R. Johnson, D. Nellans, M. O’Connor, and S. W. Keckler. Flexible software profiling of gpu architectures. In *ACM SIGARCH Computer Architecture News*, volume 43, pages 185–197. ACM, 2015.
- [19] 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*, 127, 2012.
- [20] H. Takizawa, K. Koyama, K. Sato, K. Komatsu, and H. Kobayashi. Checkl: Transparent checkpointing and process migration of opencl applications. In *Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International*, pages 864–876. IEEE, 2011.
- [21] 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*, pages 408–413. IEEE, 2009.
- [22] 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*, page 38. ACM, 2015.

- [23] D. Tiwari, S. Gupta, J. Rogers, D. Maxwell, P. Rech, S. Vazhkudai, D. Oliveira, D. Londo, N. DeBardleben, 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*, pages 331–342. IEEE, 2015.
- [24] H. Zhong and J. Nieh. Crak: Linux checkpoint/restart as a kernel module. Technical report, Technical Report CUCS-014-01, Department of Computer Science, Columbia University, 2001.