Efficient Modeling of Error Propagation in GPU Programs

by

Abdul Rehman Anwer

BEE, National University of Sciences and Technology (Pakistan), 2015

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF APPLIED SCIENCE

in

THE FACULTY OF GRADUATE AND POSTDOCTORAL STUDIES
(Electrical and Computer Engineering)

The University of British Columbia
(Vancouver)

June 2020

© Abdul Rehman Anwer, 2020
The following individuals certify that they have read, and recommend to the Faculty of Graduate and Postdoctoral Studies for acceptance, the thesis entitled: “Efficient Modeling of Error Propagation in GPU Programs” submitted by Abdul Rehman Anwer in partial fulfillment of the requirements for the degree of Master of Applied Science in Electrical and Computer Engineering.

**Examiner Committee:**
Karthik Pattabiraman, Electrical and Computer Engineering  
*Supervisor*

Tor Aamodt, Electrical and Computer Engineering  
*Supervisory Committee Member*
Abstract

Graphics Processing Units (GPUs) are popular for reliability-conscious uses in High Performance Computing (HPC), machine learning algorithms, and safety-critical applications. Fault Injection (FI) techniques are generally used to determine the reliability profiles of programs in the presence of soft errors. However, these techniques are highly resource- and time-intensive. GPU applications are highly multi-threaded and typically execute hundreds of thousands of threads, which makes it challenging to apply FI techniques. Prior research developed a model called TRIDENT to analytically predict Silent Data Corruption (SDC) (i.e., incorrect output without any indication) probabilities of single-threaded CPU applications, without requiring any FIs. Unfortunately, TRIDENT is incompatible with GPU programs, due to their high degree of parallelism and different memory architectures compared to CPU programs. The main challenge is that modeling error propagation across thousands of threads in a Graphics Processing Unit (GPU) kernel requires enormous amounts of data to be profiled and analyzed, posing a major scalability bottleneck for HPC applications. Further, there are GPU-specific behaviors that must be modeled for accuracy.

In this thesis, we propose GPU-TRIDENT, an accurate and scalable technique for modeling error propagation in GPU programs. Our key insight is that error propagation across threads can be modeled based on program execution patterns. These can be characterized by control-flow, loop iteration, data, and thread block patterns of the GPU program. We also identify two major sources of inaccuracy in building analytical models of error propagation and mitigate them to improve accuracy. We find that GPU-TRIDENT can predict the SDC probabilities of both the overall GPU programs and individual instructions accurately, and is two orders
of magnitude faster than fi-based approaches. We also demonstrate that GPU-TRIDENT can guide selective instruction duplication to protect GPU programs similar to fi. We also deploy GPU-TRIDENT to assess the input-dependence of reliability of GPU kernels and find that the SDC probability of kernels is generally insensitive to variation in inputs.
Lay Summary

Transient hardware faults are increasing due to growing system scales and progressive technology scaling. Graphics Processing Units, which efficiently process a large amount of data using their parallel structure, are susceptible to these faults, which affect their reliability. In this thesis, we analyze GPU applications and identify challenges in modeling error propagation in a scalable and accurate manner. These challenges relate to error propagation across threads and large amounts of data to be profiled due to the highly parallel nature of GPUs. We then propose heuristics to address these challenges, based on data patterns and control flow similarity across threads, thread blocks, and loop iterations in GPU applications. Next, we use the proposed techniques to guide selective instruction duplication to increase the reliability of GPU applications with low overhead. Finally, we leverage the low resource usage of our techniques to study the reliability characteristics of GPU applications with multiple inputs.
Preface

This thesis is the result of work carried out by myself, in collaboration with my advisor, Dr. Karthik Pattabiraman, Dr. Guanpeng Li (University of Illinois at Urbana-Champaign), Dr. Timothy Tsai, Dr. Siva Kumar Sastry Hari, and Dr. Michael Sullivan (Nvidia Research) All chapters, with the exception of Section 7.2, are based on work published in the 2019 IEEE Workshop on Silicon Errors in Logic - System Effects (SELSE) and another work in submission. I was responsible for conceiving the ideas, designing and conducting the experiments, compiling the results, and writing the paper. Dr. Pattabiraman was responsible for overseeing the project, providing guidance and feedback, and editing and writing parts of the paper. Gaunpeng contributed with his knowledge and expertise due to his closely related prior work. He provided insights into the problems, helped in designing experiments, and assisted with the tools used for these experiments. Timothy, Siva, and Michael provided analysis and insights throughout the course of the project.

Table of Contents

Abstract ................................................................. iii
Lay Summary ............................................................ v
Preface ................................................................. vi
Table of Contents ..................................................... vii
List of Tables .......................................................... x
List of Figures .......................................................... xi
List of Abbreviations ................................................... xiii
Acknowledgments ....................................................... xiv

1 Introduction ......................................................... 1
  1.1 Soft Errors in GPUs .......................................... 1
  1.2 Motivation ..................................................... 2
  1.3 Approach and Contributions ................................ 3

2 Background ......................................................... 7
  2.1 GPU Architecture and Programming Model ................ 7
  2.2 Fault Model ................................................... 8
  2.3 Terms and Definitions ........................................ 9
  2.4 LLVM Compiler ................................................. 10
2.5 TRIDENT Framework ........................................... 10
   2.5.1 Static-Instruction Sub-Model ($f_i$) ................. 11
   2.5.2 Control-Flow Sub-Model ($f_c$) ...................... 12
   2.5.3 Memory Sub-Model ($f_m$) ......................... 12

3 Related Work ................................................. 14
   3.1 GPU Error Resilience .................................. 14
   3.2 Modeling Error Propagation ............................ 15
   3.3 Pruning of FI Space ..................................... 16
   3.4 Resilience Analysis Across Multiple Inputs .......... 16
   3.5 Summary .................................................. 17

4 Challenges .................................................... 18
   4.1 Overview ................................................ 18
   4.2 Large Amount of Profiling Data ....................... 19
   4.3 Large Number of Threads ............................... 20
   4.4 Accumulated Inaccuracy From Individual Threads .... 21
   4.5 Summary ................................................. 21

5 Approach ..................................................... 22
   5.1 Profiling for Intra-Thread Memory Dependency ($H$-Intra) .... 22
      5.1.1 Selection Based on Thread Execution Path ($H$-Intra-Path) 23
      5.1.2 Selection Based on Loop Patterns ($H$-Intra-Loop) ........ 24
   5.2 Profiling for Inter-Thread Memory Dependencies ($H$-Inter) .... 25
   5.3 Value-Based Masking ($H$-Value) ...................... 28
      5.3.1 Multiplication by Zero ............................ 29
      5.3.2 Lucky Stores ....................................... 29
   5.4 Analysis Workflow ...................................... 30
   5.5 Summary ................................................. 31

6 Evaluation .................................................... 32
   6.1 Experimental Setup .................................... 32
      6.1.1 Workflow and Implementation of GPU-TRIDENT .... 32
      6.1.2 FI Method .......................................... 33
List of Tables

Table 5.1   Thread group details for the studied kernels ............................. 24
Table 5.2   Average Control-flow(CF) similarity percentage of threads in a
thread block .............................................................. 27
Table 6.1   Benchmarks used: 8 are from the Rodinia suite, and the other 4
are open source HPC applications (OS HPC). ........................... 34
Table 7.1   Number of threads invoked by different kernels, for multiple
inputs. ............................................................................. 47
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Workflow of Trident [35]</td>
<td>11</td>
</tr>
<tr>
<td>2.2</td>
<td>Working of TRIDENT [35]</td>
<td>11</td>
</tr>
<tr>
<td>4.1</td>
<td>A slightly modified code segment from the Circuit benchmark [1]</td>
<td>19</td>
</tr>
<tr>
<td>4.2</td>
<td>Prediction of SDC Probability by a modified GPU-TRIDENT that does not model error propagation across threads, for benchmarks that use shared memory; NW K1 means the first kernel of NW benchmark</td>
<td>21</td>
</tr>
<tr>
<td>5.1</td>
<td>Conditional branches inside a loop in the Pathfinder</td>
<td>25</td>
</tr>
<tr>
<td>5.2</td>
<td>Profiling inter-thread memory dependencies. (Values in squares represent memory addresses.)</td>
<td>26</td>
</tr>
<tr>
<td>5.3</td>
<td>Example of error masking due to multiply by 0</td>
<td>29</td>
</tr>
<tr>
<td>5.4</td>
<td>Example of Lucky stores</td>
<td>31</td>
</tr>
<tr>
<td>6.1</td>
<td>The SDC probability of GPU kernels predicted by FI and GPU-TRIDENT. Error bars are shown for the FI estimates at the 99% confidence level - they range from ±0.53% to ±1.82%</td>
<td>36</td>
</tr>
<tr>
<td>6.2</td>
<td>Computation Spent to Predict SDC Probability</td>
<td>39</td>
</tr>
<tr>
<td>6.3</td>
<td>Performance comparison of FI and GPU-TRIDENT</td>
<td>40</td>
</tr>
<tr>
<td>7.1</td>
<td>SDC Probabilities of selective instruction duplication</td>
<td>45</td>
</tr>
<tr>
<td>7.2</td>
<td>Average Protection curve obtained by FI and GPU-TRIDENT</td>
<td>45</td>
</tr>
</tbody>
</table>
Figure 7.3  Protection curves obtained by FI and GPU-TRIDENT for individual kernels, as protection overhead is varied from 0% and 100%. Blue lines indicate FI, and orange lines represent GPU-TRIDENT. 46

Figure 7.4  SDC Probabilities predicted by GPU-TRIDENT with multiple inputs. 48
List of Abbreviations

FI  Fault Injection
GPU  Graphics Processing Unit
HPC  High Performance Computing
SDC  Silent Data Corruption
I would like to express my sincere gratitude to my advisor Dr. Karthik Pattabiraman for his consistent support during my Masters. His patience, invaluable guidance, and in-depth knowledge made my stay UBC extremely rewarding and helped me grow personally as well as professionally. His enthusiasm and an optimistic outlook are quite inspiring for young researchers like me.

In addition to my advisor, I would like to thank my thesis examining committee, Dr. Tor Aamodt, and Dr. Matei Ripeanu, for their insightful feedback.

I would like to thank Dr. Gaunpeng Li and my other colleagues in the Dependable Systems Lab at UBC for their support and stimulating discussions throughout my research endeavors.

Finally, I would like to thank my family, especially my sister, for providing me with unfailing encouragement and backing throughout my years of study. My academic achievements would not have been accomplished without this support.
Chapter 1

Introduction

1.1 Soft Errors in GPUs

GPUs have been widely adopted as accelerators for scientific applications and machine learning algorithms due to their high performance and power efficiency. Their highly parallel architecture makes them ideal for efficiently parallelizing the processing of large chunks of data [2]. Recently, they are being used in safety-critical systems (e.g., self-driving cars [4]), which makes their reliability a major concern.

The performance of processors has seen immense improvements due to faster transistors and increasing circuit density. This is an ongoing trend that will result in smaller and denser components in processors of the future. Transient hardware faults (i.e., soft errors) are thus predicted to increase in future processors due to these growing system scales, progressive technology scaling, and lowering of operating voltages [50]. Transient errors manifest in the form of bit flips in the system, and can be caused by high energy particle strikes (i.e., alpha particles), transistor variability, and thermal cycling. As indicated by the name, transient errors, in contrast to manufacturing faults, do not result in permanent defects in the chips; thus, they are not reproducible. In the past, such faults were handled in high-reliability systems through hardware-only solutions such as dual modular redundancy (DMR) and circuit hardening [40]. However, these techniques are challenging to deploy in commodity systems as they incur significant performance and energy overheads.
Protecting the system, against hardware faults, at the application-level, is a low-overhead alternative to hardware-based approaches.

One consequence of transient errors is incorrect program output, without any indication to the programmer or the rest of the system, which is known as silent data corruption (SDC). SDCs are challenging to detect and can have severe consequences [50] on the reliability and safety of Graphics Processing Unit (GPU) applications. Studies have shown that only a small fraction of the program state is responsible for almost all of the error propagation resulting in SDCs. So one can selectively protect the vulnerable program state from meeting a target SDC probability while incurring lower energy and performance overheads than full duplication techniques [18, 35]. Therefore, it is vital to estimate the SDC probability of a program to decide whether protection is needed and, if so, to reduce the cost of protection by selectively protecting the most SDC-prone program state (e.g., selective instruction duplication).

1.2 Motivation

For deploying any technique to protect the system from SDCs arising from soft error at the application level, it is necessary to have a comprehensive reliability profile of the application. FI is commonly employed for this purpose. FI systematically perturbs the program state during execution and checks the program output to detect failures (if any). As the injection space for FI is too large, thousands of random FI trials are required to obtain statistically significant results. Hence, FI is often very slow and challenging to deploy in practice [22, 23, 35]. Further, selective protection techniques need the SDC probability on a per-instruction basis; this requires hundreds of FIs per instruction, which is very resource-intensive. Researchers have attempted to model error propagation to evaluate SDC probabilities quickly without any FIs [18, 27, 51]. Unfortunately, these techniques (1) have significant inaccuracies [18, 51], (2) require large and difficult-to-obtain training corpuses of FI data that are representative of typical faults [27], (3) do not provide reliability profiles for individual instructions which is critical for the selective protection of programs [27, 36], or (4) the techniques proposed are only applicable to CPU applications [32, 35, 51]. Therefore, there is a need for developing a scalable
1.3 Approach and Contributions

This thesis focuses on developing an accurate and scalable model for error propagation in GPU applications, which does not require resource-intensive FI.

In recent work, Li et al. [35] proposed an automated technique called TRIDENT that analytically models error propagation in programs to predict their SDC probabilities. While TRIDENT is accurate and fast in evaluating the vulnerability of single-threaded CPU programs to transient errors, we find it is neither accurate nor scalable for GPU programs, which are highly parallel and have a very different programming model.

In particular, inter-thread data dependencies among the threads in a typical GPU program complicate error propagation [33], and not modeling them leads to large inaccuracies in predicting SDC probabilities, e.g., the mean absolute error in SDC prediction without modeling inter-thread dependencies is 17.2% with respect to FI (Section 4.3). Furthermore, when modeling GPU programs, TRIDENT takes a significant amount of time, as a lot of run-time information has to be profiled, due to a large number of threads in GPU programs, to bootstrap the model. This profiling process is extremely time-consuming and resource-intensive in real-world GPU programs. Finally, to estimate per-instruction reliability accurately, we need to consider data characteristics specific to GPU applications, which is unfortunately not supported in TRIDENT. This introduces inherent inaccuracies in the model, which, when aggregated, can result in a model that significantly deviates from the actual behavior of the program.

In this work, we propose an accurate and scalable technique to analytically model error propagation in GPU programs. Our key insight is that error propagation in GPU programs can be abstracted using program execution patterns based on memory accesses, loop iterations, and control-flow. We find that analyzing the different levels of memory dependencies in a GPU kernel is often the biggest scalability bottleneck in the model. So carefully selecting only a small subset of threads for analysis of memory dependencies in a GPU kernel, based on the characteristics mentioned earlier, can significantly reduce the time required for creating the
Our work builds on top of the TRIDENT framework \[35\], and significantly enhances it to solve the above challenges of GPU programs - therefore, we call our technique GPU-TRIDENT.

There are 3 factors that make GPU-TRIDENT practical for assessing the reliability of real-world GPU programs and protecting them. First, GPU-TRIDENT is nearly as accurate as FI but works without requiring any FIs, and is hence significantly faster than FI. Second, it does not require any training corpuses of FI data for creating the model. Finally, GPU-TRIDENT can efficiently guide the selective protection for a GPU program given a reliability target and a performance overhead budget.

*To the best of our knowledge, we are the first to efficiently model error propagation for GPU programs, using neither FI nor training data based on FI, to estimate its overall SDC probability, as well as the SDC probability of individual instructions.*

**Contributions:** The main contributions of this thesis are:

- Identify challenges in efficiently and accurately modeling error propagation for GPU applications (Chapter 4).

- Propose heuristics for pruning the state space for modeling error propagation in GPU programs. The heuristics are based on similarity among the control flow and memory access patterns of the program’s threads (Chapter 5).

- Build GPU-TRIDENT, an efficient and accurate model of error propagation for GPU programs that implements the above heuristics using the LLVM compiler \[30\] and is completely automated.

- Compare the accuracy and scalability of GPU-TRIDENT to FI when predicting the SDC probabilities of individual instructions and that of the entire GPU kernels (Chapter 6) for 17 GPU kernels belonging to 12 applications.

- Demonstrate the use of GPU-TRIDENT to guide selective instruction duplication, under a given overhead budget (Chapter 7).
• Study the input-dependence of resilience characteristics of GPU kernels leveraging the low overhead methodology of GPU-TRIDENT (Chapter 7).

Our main results are as follows:

• SDC probability predictions from GPU-TRIDENT have a high agreement (Pearson correlation coefficient of 0.88) with the FI results. Individual instructions of most kernels also have a high agreement with the FI results (average Pearson correlation coefficient of 0.83).

• GPU-TRIDENT incurs a fixed initial overhead and a small incremental overhead for each sampled instruction, while FI incurs an overhead proportional to the number of sampled instructions (i.e., injected faults). This makes GPU-TRIDENT much more efficient for large programs than FI. For example, for 5000 faults, GPU-TRIDENT is 55.6 times faster than FI. FI takes on average around 4 CPU hours for 5000 sampled instructions, while GPU-TRIDENT takes approximately 6 minutes across benchmarks. This difference is even higher for more complex applications. For example, FI takes 22.7 hrs for the Circuit benchmark, while GPU-TRIDENT takes just 8 minutes. On average, GPU-TRIDENT is about two orders of magnitude faster than FI across the benchmarks.

• Using GPU-TRIDENT to guide selective instruction duplication reduces the SDC probability of kernels by approximately 58% and 85% (at 1/3rd and 2/3rd the performance overhead of full duplication respectively) which is comparable to the results obtained using FI.

• Using GPU-TRIDENT to study the input-dependence of resilience characteristics of a GPU kernel reveals that the SDC probability of kernels is mostly insensitive to changes in the input provided to the applications.

Our work thus provides an accurate, efficient, and scalable tool for characterizing resilience profiles of GPU kernels. In addition, we demonstrate the practical use of the developed tool to guide selective instruction duplication with a very small performance overhead. Finally, we exploit the low resource usage of our proposed tool to study the variation in resilience properties of GPU applications,
which would have been highly resource-intense if traditional FI methodologies were used.

The rest of the thesis is organized as follows. We first give a brief background about the problem and tools used in Chapter 2, then we provide a brief overview of related work in Chapter 3. We identify the main challenges in Chapter 4 and present our approach for handling them in Chapter 5. We evaluate our proposed techniques in Chapter 6 and perform two case studies on the resilience of GPU kernels using GPU-TRIDENT in Chapter 7. Finally, we conclude the thesis and discuss avenues of future research in Chapter 8.
Chapter 2

Background

In this section, we first present a brief primer on GPU architecture, then define the terms we use, followed by our fault model and the infrastructure we use for FI experiments and analysis. Finally, we provide a brief overview of the TRIDENT technique [35], as it forms the basis of GPU-TRIDENT.

2.1 GPU Architecture and Programming Model

We focus on GPU applications that are implemented on the NVIDIA Compute Unified Device Architecture (CUDA), a widely adopted programming model and toolset for GPUs. In the CUDA programming model, a GPU application consists of a control program running on the CPU and a computation program called the kernel that runs in parallel on the GPU(s), in the form of multiple threads. CUDA kernels adopt the single instruction multiple thread (SIMT) execution model to exploit the massive parallelism of GPUs. From a software perspective, the CUDA programming model abstracts the SIMT model into a hierarchy of kernels, blocks, and threads. CUDA kernels consist of blocks, which consist of threads. This hierarchy allows various levels of parallelism, such as fine-grained data parallelism, coarse-grained data parallelism, and task parallelism. From a hardware perspective, blocks of threads run on streaming multiprocessors (SMs) with on-chip shared memory for threads in the same block. Within a block, threads are launched in fixed groups of 32 threads called warps. Threads in a warp execute the same sequence
of instructions but with different data values.

Each GPU has its own memory space that is distinct from the host CPU’s memory. In the CUDA programming model, there are various kinds of memory: (1) global, (2) constant, (3) texture, (4) shared, and (5) thread-local memory allocations and accesses. Global, constant, and texture memory accesses that miss in the on-chip caches are loaded from the large and comparatively-slow device memory. The shared memory space is software managed. It is much smaller and built on chip, and is hence much faster to access. Thread-local memory is typically stored in the fast register file, though compiler-inserted spill and fill operations occasionally place the thread-local state in slower areas of the storage hierarchy.

The CUDA programming model allows (1) sharing data among a subset of threads in a thread block (e.g., warp-shuffle), (2) sharing data across all the threads in a thread block using the shared-memory scratchpad, (3) sharing data across the threads in a compute kernel using device global memory, and (4) sharing across devices using unified virtual memory (UVM) [26]. This gives rise to two types of memory dependencies between instructions.

1. Intra-thread memory dependency: Static loads and stores of same thread are dependent on each other due to the same memory being accessed by them.

2. Inter-thread memory dependency: Static loads and stores in different threads are dependent on each other due to the same memory being accessed in different threads.

### 2.2 Fault Model

In this thesis, we consider transient hardware faults that occur in the computational elements of the GPU, including architectural registers and functional units, and affect the program’s execution. We assume these faults manifest as a single bit flip. Many studies [9, 10, 25, 49] have shown that there is little difference between the SDC probability of single and multiple bit flips. Moreover, previous work in this area [27, 33, 41, 55] also uses the single-bit flip model. We do not consider faults in the GPU’s control logic, nor do we consider faults in the instructions’ encoding. We also do not consider faults in the memory or caches, as we assume that these
are protected with error correction codes (ECC) - this is the case for most modern GPUs used in HPC applications [6]. However, an error can propagate to memory, if an erroneous value is stored by a store instruction into memory, resulting in subsequent loads being faulty (these faults are considered).

Finally, we assume that the program does not jump to arbitrary, illegal addresses due to faults during the execution, as this can be detected by control-flow checking techniques [43]. However, the program may take a faulty legal branch (the execution path is legal, but the branch direction is wrong due to faults propagating to the branch condition). Our fault model is in line with other work in the area [13, 18, 20–22, 29].

2.3 Terms and Definitions

**Fault Occurrence:** The event corresponding to the occurrence of a hardware fault in the processor. The fault may or may not result in an error.

**Fault Activation:** The event corresponding to the software manifestation of the fault, i.e., the fault becomes an error and corrupts some software state (e.g., a register or memory location). The error may or may not result in a failure.

**Crash:** The raising of a hardware trap or exception due to an error (e.g., read outside its memory segments). The program is terminated as a result by the operating system.

**Silent Data Corruption (SDC):** A mismatch between the output of a faulty program run and that of an error-free execution of the program, without any exception being thrown.

**Benign Faults:** When the program output matches that of the error-free execution even though a fault occurred, because the fault was masked or was overwritten by the program.

**Error propagation:** Error propagation means that the fault was activated and has affected some other portion of the program state, say 'X' - we say the fault has propagated to state X. We focus on faults that affect the program state, and therefore consider error propagation at the application level.

**SDC Probability:** The probability that the program had an SDC given that the fault was activated—other work uses a similar definition [18, 23, 45].
2.4 LLVM Compiler

We use the LLVM compiler [30] to perform FI experiments and do program analysis for the implementation of our model. LLVM uses a typed intermediate representation (IR) that represents source-level constructs. Moreover, it maintains variable and function names in the IR, which makes source mapping feasible, enabling us to map IR to source code and perform a fine-grained analysis of error propagation in the program. However, our methodology is not tied to LLVM. Any platform that allows us to statically analyze programs and correlate it with the FI results will suffice.

2.5 TRIDENT Framework

TRIDENT is an open-source framework that analytically models error propagation at the instruction-, control-flow-, and memory-levels to derive an estimate of the overall SDC probability and SDC probability per instruction for a given CPU program [35]. However, TRIDENT does not handle the modeling of error propagation in multi-threaded programs, as well as other GPU-specific structures. As we show in Section 4, modeling these is critical for GPU programs. Our technique is built on top of the TRIDENT framework but extends it significantly.

**Workflow:** TRIDENT takes three inputs (1) LLVM Intermediate Representation (IR) of the program, (2) program output instructions (i.e., instructions whose results determine whether an SDC occurs), and (3) input for the program [35]. It decomposes the error propagation in the dynamic execution of an application into a combination of probabilistic events. These probabilities can be calculated by plugging data profiled from a dynamic execution (i.e., instruction operands, branch direction, etc.) into certain mathematical equations. Using this information TRIDENT calculates the SDC probabilities of static instructions and aggregates them to derive the overall SDC probability of the program.

TRIDENT consists of two phases: (1) *Profiling:* In the profiling phase, TRIDENT performs static and dynamic analysis of the program such as instruction counts, data dependencies, etc., and (2) *Inferencing:* TRIDENT uses the profiled information to track error propagation, which it employs to calculate SDC probabilities.
**Figure 2.1:** Workflow of Trident [35]

(a) Code example for TRIDENT, with propagation probabilities.  
(b) A memory access trace and corresponding memory dependency graph.

**Figure 2.2:** Working of TRIDENT [35].

**Example:** Figure 2.2a shows how TRIDENT works using an example from the CPU version of the pathfinder benchmark. The instructions in the example are indexed in each basic block. As TRIDENT works at the LLVM Intermediate Representation (IR) level, these instructions correspond to LLVM instructions [30]. TRIDENT consists of three sub-models that work synergistically together to calculate error propagation probability from a given location of error occurrence to the program output instruction.

### 2.5.1 Static-Instruction Sub-Model \((f_s)\)

This sub-model computes the propagation probability of an error in a static data-dependent instruction sequence (which often appears in the same basic block). For example, if an error occurs at INDEX 1 in Figure 2.2a, \(f_s\) calculates the overall propagation probability for the error from Index 1 to Index 4. Each instruction is assigned an individual error propagation probability from that instruction to the
next data-dependent instruction in the same basic block. This probability is derived by analyzing the instruction and profiling the values of its operands. \( f_d \) computes the overall propagation probability by aggregating the individual probabilities of the data-dependent instructions in the basic block.

### 2.5.2 Control-Flow Sub-Model (\( f_c \))

This sub-model computes the probabilities of store instructions getting corrupted due to the corruption of branch instructions, which results in control-flow divergence from a fault-free run. For example, if the error propagates to Index 4, which is a branch instruction, it may modify the direction of the branch. Say that the branch F of Index 4 should be taken in a fault-free execution, and the store instruction (Index 5) is supposed to be executed. But due to the error in Index 4, the branch direction can be modified to T. Consequently, the store instruction (Index 5) is not executed correctly, and hence we say that Index 5 is corrupted. In this case, the error propagates into memory via the store instruction. TRIDENT identifies all stores that are dominated by the corrupted branch (Index 4 in the running example) and computes the probabilities of them getting corrupted.

### 2.5.3 Memory Sub-Model (\( f_m \))

The memory sub-model analyzes error propagation in the kernel via memory dependencies. Continuing with our running example, if Index 5 is corrupted, the erroneous value stored by the instruction may be loaded later by the load instruction (Index 6) in bb18. Hence, the error propagates via a memory dependency. To figure out the memory dependencies between all loads and stores, TRIDENT profiles a memory execution trace from the program. The trace (Figure 2.2b) contains all the executed store and load instructions at runtime (ordered by execution time). Each record in the trace contains the type of operation (load/store), index of the static instruction, and the address accessed by the instruction. TRIDENT leverages the memory execution trace to track error propagation in \( f_m \) by constructing a memory dependency graph for \( f_m \). The graph contains memory dependencies between the static load and store instructions, and it is weighted based on the dynamic instruction counts of these instructions. An example of the graph is shown in Figure 2.2b.
A node means an instruction (load or store), and the edges indicate possible dependencies. The weights (dependent on dynamic instruction count) are marked beside each edge. This graph is used during execution to track error propagation due to memory dependencies.
Chapter 3

Related Work

We classify related work into four broad categories. The first category includes work related to characterizing the resilience of GPU applications. It presents an overview of FI tools as well as analytical frameworks developed for GPU applications. The second category summarizes the body of work related to modeling error propagation at the application level. Next, we review some techniques suggested to overcome bottlenecks in studying the resilience characteristics of GPU applications, as this is one of the core contributions of this thesis. Finally, we give an overview of work related to the input-dependence of error resilience.

3.1 GPU Error Resilience

A significant amount of research has gone towards finding the resilience of GPU programs through FI. Yim et al. [56] developed one of the first FI tools for GPU applications to explore efficient error detectors in GPU programs. Fang et al. [16] developed GPU-Qin, which is a FI tool that operates on CUDA assembly code level using CUDA-gdb. Subsequently, Hari et al. [24] developed SASSIFI, which injects faults at the SASS assembly code level using the NVIDIA compiler (SASSI). Li et al. [33] design LLFI-GPU, which operates at the LLVM IR level, and use it to investigate error propagation in GPU kernels. The main advantage of LLFI-GPU is that it is independent of the specific GPU platform or architecture, as long as the compiler provides support for the platform. LLFI-GPU is used as the FI baseline in
this thesis. Unfortunately, FI requires significant time and resources in evaluating GPU applications. Hence, they are difficult to be integrated into the development cycles of GPU programs.

Tan et al. [53] propose an analytical framework to estimate the vulnerability of GPU micro-architectures. However, their technique does not consider the characteristics of the GPU program as it is micro-architecture dependent. Further, they do not distinguish between crashes and SDCs in their analysis; hence they cannot be used to mitigate SDCs. Wei et al. [55] study the approximation properties of soft errors and use them to guide approximate instruction duplication. However, this requires thousands of FIs and intimate knowledge of the application’s domain and functionality.

3.2 Modeling Error Propagation

Error propagation models are widely employed to estimate the resilience of CPU programs. Compared to FI, the faster execution of modeling techniques makes them ideal for incorporating in the software development cycle. Symplified [46] uses model-checking to systematically enumerate all error propagation paths in programs. Unfortunately, this technique does not scale due to state-space explosion. Shoestring [18] uses compile-time analysis and symptom-based error detection techniques to model error propagation and identify vulnerable instructions. Sridharan et al. [51] introduced an analytical model, program vulnerability factor (PVF), to capture the masking properties of the program, related to architecture-level faults. However, they do not distinguish between SDCs and crashes, leading to a loss in accuracy and limited visibility into program resilience. Fang et al. [17] introduce ePVF, which adds the ability to differentiate between crashes and SDCs in PVF analysis. However, none of these techniques model control-flow divergence or memory dependencies, and their accuracy suffers as a result. Further, none of them are targeted towards GPUs. Trident [32, 35] analytically models error propagation using static and dynamic analysis. While their technique is accurate and relatively fast, the methodology only applies to single-threaded CPU programs, and it does not apply to GPU programs. Guo et al. [21] identify naturally resilient code patterns in HPC applications but do not quantify the program’s resilience.
3.3 Pruning of FI Space

Pruning FI space based on similar execution patterns has been proposed to speed up FI. Hari et al. [22, 23] propose techniques to prune the FI space in CPU applications based on similar error propagation patterns in programs, reducing FI time.

There are two techniques for pruning the FI space of GPU programs that share the same high-level goal this thesis, namely to estimate the resilience of GPU programs with either limited or no FI. Nie et al. [41] propose heuristics to prune FI space by identifying representative states in GPU programs. While useful, their method still requires thousands of FI trials to be performed to obtain accurate SDC estimates. Running FI experiments on GPUs is very resource intensive as a GPU program has a huge FI space to explore among millions of threads. Further, unlike in CPUs, the process of FI itself in GPU is challenging to parallelize as only one GPU process is allowed per GPU card. Finally, the heuristics proposed by them are specific for the pruning of FI sites, and so they cannot be used in modeling error propagation, which is our goal. Kalra et al. [27] use statistical models to characterize program resilience using micro-architecture agnostic features of GPU programs. The main advantage is that the features are independent of the GPU micro-architecture, and hence can be used on different architectures, without requiring any fault injections in the prediction phase. However, their technique relies on a large amount of representative FI corpus in the training phase, which is used to learn the characteristics of SDCs and it is challenging to obtain sufficient, representative training data, which is critical for achieving high accuracy. Moreover, both techniques only predict the vulnerability of the overall GPU kernel, and not that of individual instructions, which is required for selective protection approaches [18, 29, 48].

3.4 Resilience Analysis Across Multiple Inputs

Czek et al. [14] performed one of the first studies to model the variance in failure rates with multiple inputs for CPU applications. Leo et al. [15] study the variance of failure modes for different inputs. Mahmoud et al. [37] deploy software testing techniques like test case minimization to reduce the inputs required to get a representative reliability profile of CPU applications. Li et al. [32] study the variation
of SDCs across multiple inputs and present vTrident, which bounds the SDC probability of applications across multiple inputs with FI required for only one input. However, all of these techniques are designed for CPU applications.

There is limited work in studying the reliability of GPU applications across multiple inputs. Previlon et al. [47] study the impact of execution parameters (specifically input data and thread-block size) on the reliability of GPU applications. They perform extensive FIs for this analysis and find out that the vulnerability of studied applications remains unchanged across standard inputs. However, this study only looks at 6 Benchmarks, and the number of threads launched is minimal (maximum of 1024 threads are launched), which is not practical as real-world applications can launch millions of threads.

3.5 Summary

The increasing prevalence of transient hardware faults in GPUs has resulted in a glut of research in understanding the error resilience of GPUs. Random Fault injection [16, 24, 33, 56] has been traditionally used for this purpose, but they are quite resource- and time-intensive, so it is not suitable for exhaustive studies. Prior efforts have proposed reducing the overhead of FI techniques by either pruning the FI space [27, 41] or modeling error propagation [17, 18, 51], but these techniques are either limited in accuracy or only give a coarse-grained view of the reliability of GPU application. This thesis analyzes the behavior of GPU applications and presents an efficient and scalable solution for analyzing instruction-level reliability of GPU applications. Work analyzing the reliability of GPU applications across multiple inputs is quite sparse. The proposed techniques deploy FI [47], which has a high-performance overhead. Techniques proposed in this thesis can be used to characterize the reliability of GPU applications across multiple inputs efficiently.
Chapter 4

Challenges

The goal of this thesis is to create a fast and accurate model for tracking error propagation in GPU applications and hence predict the SDC probabilities of the overall kernel as well as individual instructions. In this chapter, we discuss the challenges in modeling error propagation in GPU programs. These challenges broadly relate to the large amount of data to be profiled due to massive parallelism in GPUs applications, error propagation across threads, and inaccuracies introduced due to data patterns specific to GPUs.

4.1 Overview

As described in section 2.1, GPU kernels are massively parallel, and many kernels share data across threads. Therefore, errors may propagate not only within each thread but also between the threads. Because GPU programs have a large number of threads (e.g., the Circuit application [1] has 4.25 million threads), tracking error propagation via memory dependencies incurs high overheads due to a large amount of memory access information required to be profiled. Moreover, even small inaccuracies in modeling error propagation of individual threads are exacerbated when aggregating the kernel’s overall SDC probability due to the large number of threads.

Figure 4.1 shows a code example of the cudaSolve kernel from the Circuit benchmark. This program solves a 2D circuit grid in parallel using the Jacobian
Figure 4.1: A slightly modified code segment from the Circuit benchmark [1].

```c
1. __global__ void cudasolve(double *data, double *odata)
2. {
3.        __shared__ float sdata[12][12];
4.        ...
5.        sdata[e_i][e_j] = data[index];
6.        ...
7.        __syncthreads();
8.        odata[index] = invD*(V - sdata[e_i-1][e_j])
9.        - sdata[e_i+1][e_j] - sdata[e_i][e_j-1]
10. - sdata[e_i][e_j+1]);
11. ...
12. }
```

method. For ease of explanation, we have slightly modified the code and removed the unnecessary parts. In the example, a subset of shared memory (line 4) is initialized in each thread (line 6). After all the threads have completed the initialization (line 8), the data in the shared memory is read by adjacent threads (line 10) for computation. This way, data can be efficiently transferred between threads via fast on-chip shared memory. However, this can also result in error propagation from one thread to another, and eventually to the entire thread block and the output of the kernel. Li et al. [33] have previously shown that a single fault can lead to high contamination (up to ~60%) of memory states in GPU applications, due to error propagation across threads and data flow between global and shared memory.

We identify the following three challenges in modeling error propagation for GPU programs.

### 4.2 Large Amount of Profiling Data

To track error propagation via memory dependencies, one needs to profile all memory accesses in the kernel and identify the data dependencies between each load and store. This can be done using dynamic analysis, matching the runtime memory addresses of each load and store. However, a typical GPU kernel consists of hundreds or thousands of threads, each of which may interleave dependencies between threads in their respective thread blocks. Consequently, a massive amount of memory access information needs to be collected during the profiling phase. For example, Lulesh, which is a typical HPC application [28], generates a memory
execution trace larger than 1 TB even for medium sized inputs. Processing the trace i.e., loading it into memory and traversing it (Section 2.5.3), incurs significant overheads, which makes it impractical to use TRIDENT for modeling of GPU HPC benchmarks. It is not possible to bypass this memory bottleneck by applying TRIDENT to individual threads, as the time for modeling thousands of threads runs into days or even months - see Table 6.1

4.3 Large Number of Threads

Tracing error propagation between threads in a GPU kernel requires a fine-grained error propagation model for every thread in the GPU kernel, which potentially incurs large modeling overhead. For example, if an error occurs in a thread, it first propagates within the thread. Later, the error may propagate to another thread in its thread block via a shared memory dependency. Hence, the error continues to propagate in both threads. Finally, the error may propagate to some other threads or back to the original thread, depending on the program’s execution. To track the error, one should model all the threads in the program, and trace the propagation in a lock-step fashion.

A natural question that arises is can we ignore error propagation between threads in GPU programs? To answer this question, we modify GPU-TRIDENT to stop tracing the error propagation when any shared memory access is encountered. We show the results in Figure 4.2 for the benchmarks that use shared memory in our evaluation (the experimental setup and benchmarks are described in Section 6.1). As can be seen, the SDC probabilities predicted by the modified GPU-TRIDENT are significantly lower than the FI ground truth (mean absolute difference of 17.2%). The only exception is LUD K3, which sparingly uses shared memory (only two stores are to shared memory by each thread). This result indicates that we cannot ignore inter-thread error propagation in GPU programs.

---

1We obtained these numbers by multiplying the time taken by TRIDENT for each thread of the program with the number of threads in the kernel.
Figure 4.2: Prediction of SDC Probability by a modified GPU-TRIDENT that does not model error propagation across threads, for benchmarks that use shared memory; NW K1 means the first kernel of NW benchmark.

4.4 Accumulated Inaccuracy From Individual Threads

The overall SDC probability of a GPU kernel is the aggregation of the SDC probabilities of the individual threads. Because GPU programs consist of hundreds or thousands of threads, if the model is inaccurate in each thread (even slightly), the overall SDC probability of the kernel will be inaccurate.

4.5 Summary

In this chapter, we analyze GPU application in the context of applying TRIDENT methodology on them. We find that TRIDENT can not be applied as it is on GPU kernels as the large amount of data associated with multi-threaded GPU applications posses a huge bottleneck. Moreover, inter-thread error propagation due to interleaving memory dependencies among threads renders the $f_m$ of TRIDENT inapplicable to GPUs, as it was designed for single-threaded CPU applications. Finally, we observe that minor inaccuracies introduced while modeling error propagation in GPU kernels can result in a highly inaccurate final model.
Chapter 5

Approach

In Chapter 4, we identified challenges that complicate the efficient modeling of error propagation in GPU applications. We observed that the challenges were related to a large amount of data to be profiled, inter-thread error propagation, and inaccuracies due to biased data patterns.

In this chapter, we propose heuristics to overcome these challenges and the rationale behind them. We first propose two heuristics to select a subset of threads in a GPU kernel to find the intra-thread memory dependencies within it (Section 5.1). We then propose a third heuristic to select a new subset of threads to construct inter-thread memory dependencies (Section 5.2). Finally, we identify two types of value-based masking in modeling error propagation as major sources of inaccuracies in applying TRIDENT to GPU applications, and propose heuristics to mitigate them (Section 5.3).

5.1 Profiling for Intra-Thread Memory Dependency (H-Intra)

The goal of this step is to construct the memory dependency graph for a given GPU kernel without profiling all the memory accesses in all the threads. Recall that the memory dependency graph in TRIDENT is required to establish data dependencies between static store and load instructions (Section 2.5.3). This problem has been studied in the context of performance improvements in CPUs at architecture
level [12], but our goal is to extract the memory dependency between instructions at the application level as GPU-TRIDENT models error propagation at the application level. It is possible to profile all memory addresses used by load and store instructions, and then match the addresses to build the graph. However, this can be extremely time-consuming in massively parallel GPU programs. Instead, we profile only a small subset of threads based on control-flow similarity and match loads and stores in those carefully-selected threads to build the memory dependency graph. We call this intra-dep threads, as they contain all the possible intra-thread dependencies between static load and store instructions (for that execution). The main algorithm for choosing intra-dep threads is in Algorithm 1. Note that in this section, we use the entire control-flow profile of a thread to perform the grouping of threads, rather than the number of instructions executed per thread used in prior work [41] as we found it to be more accurate.

**Algorithm 1: Selecting intra-dep threads.**

<table>
<thead>
<tr>
<th>Input</th>
<th>CF_T: Control flow of all threads</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output</td>
<td>T_p: Set of threads to profile</td>
</tr>
<tr>
<td>1 T_p = {}</td>
<td></td>
</tr>
<tr>
<td>2 for all threads T do</td>
<td></td>
</tr>
<tr>
<td>3 if conditional branches in loop then</td>
<td></td>
</tr>
<tr>
<td>4 // Remove iterations with nonunique memory accesses</td>
<td></td>
</tr>
<tr>
<td>5 CF_sim = Remove_redundant(CF_T)</td>
<td></td>
</tr>
<tr>
<td>6 else</td>
<td></td>
</tr>
<tr>
<td>7 CF_sim = CF_T</td>
<td></td>
</tr>
<tr>
<td>8 // Profile thread if simplified control-flow is unique</td>
<td></td>
</tr>
<tr>
<td>9 if CF_sim not in T_p then</td>
<td></td>
</tr>
<tr>
<td>10 T_p.insert(T)</td>
<td></td>
</tr>
</tbody>
</table>

5.1.1 Selection Based on Thread Execution Path (H-Intra-Path)

We first profile the execution path of each thread and group the threads that have identical execution paths. We then choose a single thread from each group to be part of intra-dep threads. The intuition is that threads with identical execution paths have the same static dependencies between loads and stores. This is because
only control-flow divergence can cause divergence in the memory dependencies.

Table 5.1 shows the number of threads invoked in each kernel of the benchmark applications (Section 6.1 has more details), and the number of intra-dep threads as the result of the grouping. As can be seen, most of the kernels have only a small number of intra-dep threads. The exceptions are pathfinder and NW, The two exceptions are pathfinder and NW, as they have conditional branches inside loops, and hence loops execute different number of times for different threads, which results in different control-flow paths.

<table>
<thead>
<tr>
<th>Kernel</th>
<th>Total threads</th>
<th>After H-Intra-Path</th>
<th>After H-Intra-Loop</th>
<th>After H-Inter</th>
</tr>
</thead>
<tbody>
<tr>
<td>NW K1</td>
<td>48</td>
<td>46</td>
<td>25</td>
<td>48</td>
</tr>
<tr>
<td>NW K2</td>
<td>16</td>
<td>16</td>
<td>13</td>
<td>16</td>
</tr>
<tr>
<td>Pathfinder</td>
<td>592,640</td>
<td>563,606</td>
<td>15</td>
<td>3840</td>
</tr>
<tr>
<td>BFS K1</td>
<td>32,768</td>
<td>781</td>
<td>8</td>
<td>-</td>
</tr>
<tr>
<td>BFS K2</td>
<td>32,768</td>
<td>2</td>
<td>2</td>
<td>-</td>
</tr>
<tr>
<td>Gaussian</td>
<td>3,840</td>
<td>4</td>
<td>4</td>
<td>-</td>
</tr>
<tr>
<td>HotSpot</td>
<td>473,344</td>
<td>250</td>
<td>234</td>
<td>35,328</td>
</tr>
<tr>
<td>Particlefilter</td>
<td>9,216</td>
<td>38</td>
<td>7</td>
<td>-</td>
</tr>
<tr>
<td>LUD K1</td>
<td>64</td>
<td>16</td>
<td>16</td>
<td>64</td>
</tr>
<tr>
<td>LUD K2</td>
<td>192</td>
<td>2</td>
<td>2</td>
<td>64</td>
</tr>
<tr>
<td>LUD K3</td>
<td>3,584</td>
<td>1</td>
<td>1</td>
<td>256</td>
</tr>
<tr>
<td>CudaBenchMarking</td>
<td>8,192,000</td>
<td>1</td>
<td>1</td>
<td>-</td>
</tr>
<tr>
<td>SRAD K1</td>
<td>32,768</td>
<td>144</td>
<td>144</td>
<td>16,384</td>
</tr>
<tr>
<td>SRAD K2</td>
<td>32,768</td>
<td>16</td>
<td>16</td>
<td>4,096</td>
</tr>
<tr>
<td>Circuit</td>
<td>4,156,416</td>
<td>18</td>
<td>18</td>
<td>2,592</td>
</tr>
<tr>
<td>Lulesh</td>
<td>66,816</td>
<td>2</td>
<td>2</td>
<td>-</td>
</tr>
<tr>
<td>Blur</td>
<td>236,544</td>
<td>7</td>
<td>7</td>
<td>-</td>
</tr>
</tbody>
</table>

Table 5.1: Thread group details for the studied kernels

5.1.2 Selection Based on Loop Patterns (H-Intra-Loop)

To further reduce the number of intra-dep threads, we pick threads that have unique loop patterns. We use a code example from the Pathfinder benchmark to explain this heuristic. Figure 5.1a shows a kernel that has three conditional branches inside a loop. Pathfinder has a high number of intra-dep threads, as different threads take different branches in each iteration, and hence have a different number of iterations resulting in diverging execution paths. We profile the execution paths inside each
loop iteration and list a subset of them in Figure 5.1b. Numbers in the figure represent the comparison instruction, while $T$ and $F$ represent the branch direction taken. We find that many iterations contain repeated branch patterns, which results in the same memory dependency from the profiling. As seen from the figure, the first three iterations in Thread 1 and Thread 3 are identical. Therefore, we only need to profile one of them to obtain the inter-thread dependencies. Similarly, the order of the repeated patterns within a thread does not matter, as it would yield similar dependencies. For example, Thread 3 and Thread 4 follow the same branch patterns but in different orders of iterations, yet yield the same memory dependency. Therefore, we only need to profile one of the two threads.

Table 5.1 also shows the number of intra-dep threads after applying the above loop selection heuristic. As seen, Pathfinder and BFS K1 see a significant reduction in the number of threads, but not the other kernels as the threads do not have conditional branches with divergent control-flow within loops.

### 5.2 Profiling for Inter-Thread Memory Dependencies (H-Inter)

Intra-thread memory dependencies can be established based on the thread selection method in the previous section. However, inter-thread dependencies may still exist between threads with differing control-flows and we need to model these.
As before, we propose a heuristic to reduce the number of loads and stores that must be profiled to capture the inter-thread dependencies. Our heuristic is to first select the thread blocks that each of intra-dep threads belongs to and then profile the shared memory accesses in each thread block. This is based on the observation that there is often a high ratio of threads to thread blocks, as well-designed GPU kernels attempt to avoid control-flow divergence within each thread block to avoid stalls. Hence, we find that selecting thread blocks based on unique control-flow patterns (i.e., thread blocks containing intra-dep threads) will capture most of the unique shared memory access patterns (see Table 5.2).

Figure 5.2 shows the executions of two kernels as examples. Each row represents a possible execution of an instruction, while each column indicates a thread. For simplicity, we only show instructions accessing shared memory. The value inside each square is the memory address that the instruction uses. Having the same address means that the load and the store have an inter-thread dependency, which we need to identify.

In Figure 5.2a, there are 3 thread blocks (TBs) as shown. Each TB contains two threads. T1, T2, T5, and T6 have the same execution path, whereas T3 and T4 follow a different execution path. Note that the control-flow paths of threads are identical within each thread block in this example. T1 and T3 are selected as intra-dep threads based on their execution paths (Section 5.1). According to our heuristic, we profile all the shared memory accesses in the thread blocks that T1 and T3 belong to. In other words, we profile the shared memory in T1, T2, T3, and
T4. This way, we establish 2 inter-thread memory dependencies, namely (Index 5 and Index 9), and (Index 7 and Index 9), based on the addresses. Since the ratio of control-flow similarity across thread blocks is high (100%), our heuristic identifies all the inter-thread dependencies.

The example in Figure 5.2b also has 6 threads in 3 thread blocks. Threads in TB1 follow the same execution path, as do the threads in TB3. However, threads in TB2 (i.e., T3 and T4) follow different paths. In this case, the ratio of the similarity of the control-flow of threads inside thread blocks is only 66.67% ((100%+0%+100%)/3). Since threads T1 and T4 are selected as *intra-dep threads*, all the threads in TB1 and TB2 are considered for profiling, and the inter-thread memory dependencies (Index 5 and Index 9), and (Index 5 and Index 11) are identified. However, since the control-flow of T4 is different from T3 in TB2, the selection of T4 does not include the profiling of the threads in TB3, thereby missing another inter-thread dependency (Index 7 and Index 11) in T5 and T6.

Therefore, our heuristic achieves high coverage only when the control-flow similarity ratio is high in a thread block. So we ask the question *what are the ratios of control-flow similarity in thread blocks in GPU programs?* Table 5.2 shows the results, for the kernels that have inter-thread memory dependencies. On average, about 75% of the threads in a thread block exhibit control-flow similarity, with the highest being 100% in LUD K3, and the lowest being 0% in LUD K1. Therefore, our proposed heuristic (*H-inter*) can identify most of the inter-thread memory dependencies in practice.

<table>
<thead>
<tr>
<th>Kernel</th>
<th>CF Similarity</th>
<th>Kernel</th>
<th>CF Similarity</th>
</tr>
</thead>
<tbody>
<tr>
<td>NW K1</td>
<td>54.16%</td>
<td>NW K2</td>
<td>25%</td>
</tr>
<tr>
<td>HotSpot</td>
<td>98.31%</td>
<td>LUD K1</td>
<td>0%</td>
</tr>
<tr>
<td>LUD K2</td>
<td>70.32%</td>
<td>LUD K3</td>
<td>100%</td>
</tr>
<tr>
<td>Circuit</td>
<td>99.35%</td>
<td>Pathfinder</td>
<td>96.61%</td>
</tr>
<tr>
<td>SRAD K1</td>
<td>98.28%</td>
<td>SRAD K2</td>
<td>99.6%</td>
</tr>
</tbody>
</table>

**Table 5.2:** Average Control-flow(CF) similarity percentage of threads in a thread block

Table 5.1 shows the number of threads chosen for profiling inter-thread memory dependency (*inter-dep threads*). On average, we only select ~40% of the total threads as *inter-dep threads*. It can be be seen that kernels that have a lower num-
ber of total threads do not show much reduction (0% for NW K1, NW K2), while kernels that have a large number of threads show a much higher reduction in the number of threads chosen (~99.3% for Circuit). Note that only load and store instructions to shared memory are profiled for these threads. The algorithm for choosing inter-dep threads is in Algorithm 2.

Algorithm 2: Selecting inter-dep threads.

Input : $T_P$: Threads from H-Intra
Output: $T_{PS}$: Threads to profile

1 $T_{PS} = \{\}$
2 $T_{BP} = \{\}$
3 for all threads $T$ in $T_P$ do
4    $tb = \text{find\_threadblock}(T)$
5    if $tb$ not in $T_{BP}$ then
6        $T_{BP}.\text{insert}(tb)$
7        // Profile all threads of this thread block
8        $T_{PS}.\text{insert}($all\_threads$(tb))$
9 end

We construct the memory dependency graph for a given GPU kernel applying the above heuristic. We then follow the method of TRIDENT to weight the edges of the graph based on the dynamic instruction count of each static load and store and then trace error propagation using the graph. Note that $f_m$ does not need the memory addresses (i.e., memory execution trace) anymore for extracting the dependency. Instead, it reads it directly from the memory dependency graph, that we created for this purpose, and aggregates the propagation probabilities based on the edge weights (Section 2.5.3).

5.3 Value-Based Masking (H-Value)

As mentioned, it is imperative to have a high accuracy in modeling error propagation in GPU threads, due to the potential of aggregating small errors into large ones. We identify two sources of inaccuracies in TRIDENT and propose the $H$-Value heuristic to mitigate them - this leads to an average accuracy improvement of approximately 1.9%. The sources of inaccuracy and heuristics proposed to over-
come them are as follows.

5.3.1 Multiplication by Zero

If a target operand of an instruction is multiplied by zero, the result will always be zero, regardless of errors present in the target operand. If this is frequently seen in a program, and not taken into account while developing the model, this will overestimate the error propagation since the model does not consider this source of error masking. We find that there is a non-negligible amount of multiplication-by-zero in GPU kernels (i.e., as high as 56.6% in Lulesh, and on average 11.2%), hence this masking must be considered in GPU-TRIDENT. To account for this, we profile the operands of multiplication instructions and calculate the frequency of zero operands of \texttt{mul} instructions. We then weight the propagation and masking probabilities in the relevant instruction tuple (Section 2.5.1) accordingly.

Consider the code snippet shown in Figure 5.3 in which two values are loaded from memory and multiplied. Now suppose a fault occurs at INDEX 1 due to which its result ($\texttt{1}$) is corrupted. This fault will not propagate beyond INDEX 3 ($\texttt{3}$) if the other operand of multiply instruction i.e., $\texttt{2}$ (the result of INDEX 2) is 0, which will result in error masking, which TRIDENT does not take into account. In the example, we get the probability of $\texttt{1}$ and $\texttt{2}$ being zero, which is 0% and 40% respectively, and use it to modify the default propagation probability (100%) of INDEX 3. As each operand contributes 50% to the overall propagation probability, the new propagation probability would be \((1.0 - 0.5*0.4 - 0.5*0)\times100\% = 80\%\).

5.3.2 Lucky Stores

Recall that if an error modifies a branch direction, the store instructions dominated by the corrupted branch will be not be executed correctly (Section 2.5.2). In this
case, TRIDENT assumes that the error always propagates to the store instructions. However, if a store instruction was supposed to overwrite the value in memory, which was already there, errors will not propagate even though the instruction is skipped due to a corrupted branch. This is similar to the phenomenon of silent stores (dynamic store instructions that do not change the state of a system) [31], which has been studied in the context of performance improvement in computer architecture. We call such store instructions **lucky stores**, similar to lucky loads found in prior studies related to the reliability of CPUs [13, 35].

To understand this, consider Figure 5.4. Suppose that the store in \( bb2 \) was to be executed in a fault-free run, but due to a fault the result of INDEX 1 comparison instruction was flipped and the \( bb2 \) was not executed. The model will conservatively assume that the store is corrupted and propagates the error. However, if a condition arises such that zero was to be stored (that is missed due to a fault), and that memory location already contained a zero, the fault will have no effect on memory contents. In such a scenario, the outcome will be benign, and we will end up overestimating the SDC probability of kernel. Identifying all the lucky stores requires recording every operand of store instructions in the memory execution trace, which can be extremely time-consuming. We observe that zero values are dominant in the operands of output stores of GPU kernels as most initializations of kernel memory use zeros. Therefore, we record only the frequencies of output stores that have a zero operand at run-time, and weight the propagation probabilities in \( f_c \) accordingly. This way, we keep the profiling overhead reasonably low. Our experiments show that an average of 7.6% and a maximum of 40% output stores are lucky stores, for the kernels used in this paper.

### 5.4 Analysis Workflow

Based on the above heuristics, the analysis workflow of GPU-TRIDENT consists of the following steps:

1. **H-Value** is applied to get the \( f_s \) and \( f_c \) sub-models based on the dynamic data profiled from the threads.

2. **H-Intra** is used to select a sub-set of threads to get the intra-thread memory dependencies, whose memory access instructions are then profiled.
3. **H-Inter** is used to select a sub-set of threads to get the inter-thread memory dependencies, whose memory access instructions are then profiled.

4. $f_m$ model is created from the memory dependency graph, based on the profiled memory access instructions.

5. The sub-models ($f_c$, $f_s$, and $f_m$) are used to get the SDC probability of individual instructions and the kernel.

### 5.5 Summary

In this chapter, we present heuristics for profiling, different levels of memory dependencies in a GPU kernel. We find that a subset of threads selected based on the similarity of execution patterns of threads, loop iterations, and shared memory accesses within a thread-block can be used to recover all the inter-thread as well as intra-thread memory dependencies of a kernel. This information is then be used to construct the *memory dependency graph* of a kernel, which is used for tracking error propagation within as well as between threads. Finally, we identify two common data patterns in GPU kernel executions and use them to improve the accuracy of our modeling methodology.
Chapter 6

Evaluation

In Chapter 5, we introduced heuristics to overcome challenges in modeling error propagation in GPU applications. In this chapter, we evaluate the effectiveness of our proposed techniques. We first describe the experimental setup used (Section 6.1), giving an overview of the workflow of GPU-TRIDENT, FI method used as the baseline, and the benchmarks we use for this evaluation. We then present our results for the accuracy and scalability of GPU-TRIDENT at the kernel and instruction level. Finally, we discuss the sources of inaccuracies in our proposed techniques.

6.1 Experimental Setup

6.1.1 Workflow and Implementation of GPU-TRIDENT

We implement GPU-TRIDENT as a set of LLVM compiler passes that are integrated into NVIDIA’s NVCC compiler [42]. Although NVCC is based on LLVM, it does not expose its IR representation to external tools. Therefore, we attach a dynamic library [39] to insert the LLVM passes of GPU-TRIDENT in the toolchain. GPU-TRIDENT needs the application code (with the kernel under test annotated), and an input required to execute it. We also need to identify the static store instructions used by the kernel for transferring data to the host and designate them as output instructions. Using these inputs, GPU-TRIDENT profiles the program, and
then estimates the SDC probability of individual instructions as well as that of the complete kernel without performing any FIs.

### 6.1.2 FI Method

Recall that GPU-TRIDENT aims to predict SDC probabilities, which are usually measured using FI. Therefore, we use FI as the baseline to establish our ground truth when evaluating GPU-TRIDENT. We use an open-source injector, LLFI-GPU [33] to perform FIs. LLFI-GPU aids comparison with GPU-TRIDENT, as both are implemented using LLVM. Therefore, we can map GPU-TRIDENT predictions to the FI result of individual instructions to examine the accuracy of GPU-TRIDENT on a per-instruction basis.

As described in the fault model, we consider faults in the computational elements of the GPU (Section 2.2), we inject faults into the destination registers of executed instructions. Only one fault is injected per program execution. In each FI trial, the application is executed from the beginning. We uniformly choose an instruction at random from the set of executed instructions in the kernel. We then flip a single bit in its destination register and execute the program to completion. This method has been shown to be accurate for estimating SDC probabilities [45].

To determine if an SDC occurred, the entire array in device memory that is written by the kernel (to transfer data to the host code) is compared to its contents in a fault-free execution (i.e., golden run). This is because we only study error propagation within the GPU kernel, and hence cannot use the application output as is usually done for CPU programs [19].

### 6.1.3 Benchmarks

We choose a total of 17 kernels from 12 applications belonging to different domains, 8 of which are from the Rodinia benchmark suite [11], and 4 are open source HPC applications [1], [28], [3], [5]. These are listed in Table 6.1. They range in size from 16 to 511 static LLVM IR instructions and from 16 to 8,192,000 in terms of total threads launched. There is no standard set of benchmarks or application classification for reliability analysis. Parallel applications are usually classified into 13 categories (called dwarfs) based on their computation and communication
Benchmark | Suite | LLVM Insts. | Kernel ID | Uses shared memory | Total threads
---|---|---|---|---|---
Needleman-Wunsch | Rodinia | 248 | NW K1 | Yes | 48
249 | NW K2 | Yes | 16
Pathfinder | Rodinia | 132 | Pathfinder | Yes | 592,640
BFS | Rodinia | 47 | BFS K1 | No | 32,768
20 | BFS K2 | No | 32,768
Gaussian | Rodinia | 59 | Gaussian | No | 3,840
HotSpot | Rodinia | 259 | HotSpot | Yes | 473,344
Particlefilter | Rodinia | 39 | Particlefilter | Yes | 9,216
LU Decomposition | Rodinia | 142 | LUD K1 | Yes | 64
238 | LUD K2 | Yes | 192
78 | LUD K3 | Yes | 3,584
SRAD | Rodinia | 511 | SRAD K1 | Yes | 32,768
193 | SRAD K2 | Yes | 32,768
Lulesh | OS HPC [28] | 29 | Lulesh | No | 66,816
Circuit | OS HPC [1] | 167 | Circuit | Yes | 4,156,416
Perf Benchmark | OS HPC [5] | 16 | Perf_BM | No | 8,192,000
HPCCUDA | OS HPC [3] | 397 | HPCCUDA | No | 236,544

Table 6.1: Benchmarks used: 8 are from the Rodinia suite, and the other 4 are open source HPC applications (OS HPC).

methods [7]. The benchmarks we use in this thesis capture 6 of these 13 dwarfs. The choice of benchmarks was governed by whether they can be compiled with the LLVM-based infrastructure of LLFI-GPU and GPU-TRIDENT, and whether FI can be completed in a reasonable amount of time. We removed all sources of randomness in the benchmarks (e.g., changed random numbers to constant values), to get reproducible results for the experiments.

We use the smaller inputs that come with each benchmark. We have also tested GPU-TRIDENT with bigger inputs for a subset of the same kernels (Section 7.2), and find that the prediction results closely match up with FI (though they take much longer). However, to keep the time for the detailed FI experiments manageable, we choose to use the smaller inputs in this evaluation. Prior work [27, 33, 41, 53, 55] has also used similar input sizes.

Table 6.1 also shows the times TRIDENT and GPU-TRIDENT take for each of the kernels (as mentioned earlier, TRIDENT times are estimates). We consider the time taken for both systems to obtain the overall SDC probability and per-instruction SDC probability. As can be seen, GPU-TRIDENT takes less than about 10 minutes for all the kernels. In contrast, TRIDENT takes days or even months to
analyze the kernels.

It should be noted that a subset of these benchmarks was used to identify (5 benchmarks) the challenges and for devising heuristics (7 benchmarks) to avoid over-fitting. The rest of the benchmarks were just used to test the proposed techniques.

6.2 Accuracy

In this section, we evaluate the accuracy of GPU-TRIDENT in predicting SDC probabilities of the kernel and individual instructions.

6.2.1 Overall SDC Probability

We use GPU-TRIDENT to predict the SDC probability of a given kernel (and input), and then measure its SDC probability (under the same input) using random FI. We perform 5000 FI experiments per kernel - the error bars for the SDC measurements range from ±0.53% to ±1.82% at the 99% confidence level.

Figure 6.1 shows the result of our experiments. As can be seen, GPU-TRIDENT provides reasonably accurate predictions. The average difference (mean absolute error) between FI and GPU-TRIDENT is 5.7% (in comparison, TRIDENT had an average difference of 4.75% [35] for CPU programs). This number is inflated by the results of Lulesh, BFS K1 and Pathfinder, which have a difference of ~24%, ~15% and ~16% respectively (the difference is 2.97% if we remove these kernels). We explain the reasons in Section 6.4.

Pearson Correlation Coefficient

We also calculate the Pearson correlation coefficient between the FI results and GPU-TRIDENT predictions for all the kernels. The correlation coefficient is 0.88, showing a high agreement between them. The correlation coefficient increases to 0.99, if we ignore the three outliers described earlier.

Paired T-test

We also use a paired t-test to examine if the predictions are statistically different from the FI measurements, in line with TRIDENT’s method [35]. We first
checked that the differences between the predictions and FI measurements are approximately normally distributed, as required by the t-test. In the t-test, our null-hypothesis is that there is no statistically significant difference between the results from FIs and the predicted SDC probabilities by GPU-TRIDENT for the 17 kernels. We find that the t-test yields a p-value of 0.36, which is much higher than the customary threshold of 0.05 [52], and hence we fail to reject the null hypothesis.

### 6.2.2 Instruction SDC Probability

We also evaluate the prediction accuracy of per-instruction SDC probabilities for each kernel. We use GPU-TRIDENT to predict the SDC probabilities of all static instructions, and then compare the predicted values with the FI results. In FI, we inject 100 random faults in each static instruction of the kernel (a random dynamic instance of the instruction is chosen for FI in each run).
**Pearson Correlation Coefficient**

As before, we calculate the Pearson correlation coefficient between the SDC contribution by each static instruction found using FI and by GPU-TRIDENT for all kernels. The average correlation coefficient was 0.83, excluding the outliers mentioned in Section 6.2.1 and *Gaussian*. Gaussian has a low coefficient which is reflected in the predicted SDC probability being more than double the measured SDC probability, although the percentage difference is ~5%. The correlation analysis shows that the per-instruction SDC probability obtained by GPU-TRIDENT has a high agreement with FI results.

**Paired T-test**

We also perform paired T-test for SDC probability of individual instructions. We first check if the distribution of the differences between the prediction and the FI measurement is approximately normally distributed. The normality does not hold for *SRAD K1, SRAD K2, Circuit* and *Particlefilter*, and so we exclude them from this experiment (however, the predicted SDC probabilities for these kernels are close to the FI results (Figure 6.1)). We perform paired t-test experiments in the remaining 13 kernels. The number of paired data in the t-test for each kernel is the number of its static instructions. As before, our null hypothesis is that there is no difference between the FI measurement and the predicted SDC probability of each (static) instruction by GPU-TRIDENT. The p-values are higher than 0.05 in 11 out of the 13 kernels (all except *Lulesh* and *NW K1*), and hence we cannot reject the null hypothesis.

**6.3 Scalability**

In this section, we assess the scalability of GPU-TRIDENT with respect to FI. We evaluate the scalability both in terms of the overall SDC probabilities of kernels as well as those of individual instructions. As mentioned earlier, our accuracy experiment considered 5,000 trials and obtained error bars at the 99% confidence interval. However, if one is prepared to accept lower confidence intervals or looser error bars, it is possible to decrease the number of FI experiments (i.e., samples). To evaluate the scalability of GPU-TRIDENT against FI, we compare the time taken...
by GPU-TRIDENT and FI when using 500 to 5,000 samples. This method is in line with the number of samples used in other studies [18, 27, 33].

As mentioned earlier (Section 2.5), executing GPU-TRIDENT can be divided into two phases. (1) Profiling phase, which requires multiple executions of the program to collect data, and the (2) inference phase, which uses the information obtained from the profiling phase to calculate the SDC probabilities for static instructions, and requires no program executions. We implement the inference phase of GPU-TRIDENT in (1) single-threaded mode and (2) multi-threaded mode. However, we do not parallelize the profiling phase, to keep it consistent with FI, which is not parallelized either. Note that parallelizing the profiling phase is non-trivial as it requires either multiple GPUs or the ability to run multiple applications simultaneously on a single GPU.

6.3.1 Kernel SDC Probability

Figure 6.2a shows the average time taken by FI and both implementations of GPU-TRIDENT to predict the SDC probability of CUDA kernels. Due to space constraints, we show the average time across all the kernels. It can be seen that the differences between the times taken by GPU-TRIDENT and FI increase sharply as the number of samples is increased. For example, for 500 samples, GPU-TRIDENT is 2.2 and 5.7 times faster than FI (for single and multi-threaded implementations respectively). This increases to 12.7 and 33.4 times for 3,000 samples, and 21.1 and 55.6 times for 5,000 samples. This is because FI has negligible fixed cost at startup, but every FI trial requires a complete program execution. So the time required for FI experiments increases linearly with the number of FI samples. In contrast, GPU-TRIDENT has an initial fixed cost for building the model. Once the model is built, calculating the SDC probability of individual instructions has a negligible cost, as it only involves a table lookup. This results in predominantly flat lines for GPU-TRIDENT in Figure 6.2a. It can be seen that multi-threaded GPU-TRIDENT is on average around 2.5X faster (on an 8-core CPU) than its single-threaded counterpart.
6.3.2 Instruction SDC Probability

Figure 6.2b compares the average time taken by GPU-TRIDENT and FI, for getting instruction wise SDC probability, for different numbers of static instructions in the kernel, with different numbers of faults injected (100, 500 and 1,000) per instruction. The time shown in the figure is projected based on the per instruction times recorded for kernels that we use in our evaluation. The number of samples per instruction is appended as suffixes in Figure 6.2b. For example, FI-100 means that 100 faults are injected into each static instruction. Because GPU-TRIDENT does not need to sample individual instructions, the time taken by it remains constant; therefore, it has only one curve for each implementation. It can be seen that as the number of instructions in a kernel increases, the difference between the time taken by FI and GPU-TRIDENT also increases. For example, at 50 instructions, FI-100 takes around 4 hours more than GPU-TRIDENT (averaged for both implementations). This difference increases to around 84 hours for 1,000 instructions (more than 20X increase).

Figure 6.3 shows the wall-clock time taken by both implementations of GPU-TRIDENT and FI (100 faults injected per instructions), for obtaining instruction wise SDC probability, for each kernel. There is a wide variation in times taken by GPU-TRIDENT and FI for different kernels. For example, BFS K2 takes 0.16 hours, while Circuit takes 63.75 hours for FI, so we use a logarithmic scale in the figure on the y-axis. From the figure, we can see that on average, single-threaded
GPU-TRIDENT is faster than FI by more than one order of magnitude (~38X), while multi-threaded GPU-TRIDENT is faster than FI by about two orders of magnitude on average. Further, higher the complexity of the application, greater the improvement, e.g., Circuit and Lulesh in Figure 6.3.

![Figure 6.3: Performance comparison of FI and GPU-TRIDENT](image)

### 6.4 Sources of Inaccuracies in GPU-TRIDENT

In this section, we discuss the three sources of inaccuracies in GPU-TRIDENT.

#### 6.4.1 Validity of Faulty Store Addresses

GPU-TRIDENT assumes that if a wrong memory location is accessed, it will either result in a crash if the memory location is invalid, or an SDC if the memory location is valid. However, in some cases, the accessed wrong memory location is valid, but is out of the scope of the kernel. This results in a benign outcome.

On the other hand, because FI records the kernel’s output in the host code, it has the knowledge about the memory that should be written by GPU-Kernel in order to cause an SDC - therefore, it can obtain the exact SDC probability. This is one of the major reasons for SDC overestimation by GPU-TRIDENT for both Luelsh and Pathfinder. This inaccuracy can be mitigated by incorporating the information about the ratio of the memory used by the kernel that causes an SDC into GPU-TRIDENT. For example, in the case of Luelsh, if we integrate the information about the ratio of total memory used by FI to detect SDC (0.5 in this case, obtained through analysis of the host code) into GPU-TRIDENT, the absolute error in prediction is reduced from 24% to just 0.31%.
6.4.2 Conservativeness in Determining Memory Corruption

In Section 5.3.2, we discuss the phenomenon of lucky stores and introduce a heuristic to identify lucky output store instructions. However, lucky stores can also occur in the intermediate stores of the kernel, e.g., when a store to local or shared memory dominated by a faulty branch is missed or incorrectly executed, but the value written happens to be the same as the correct value. GPU-TRIDENT assumes that any fault propagating to a store instruction results in an erroneous memory value, and hence over-estimates the SDC probability. This is another reason for the inaccuracy in Pathfinder (in addition to the first).

Similarly, GPU-TRIDENT assumes that a fault in the address operand of a load instruction propagates to the instruction’s output. However, if the incorrect memory location has the same contents as the fault-free memory (i.e., lucky load [13]), it will result in a benign outcome. This is the major reason for inaccuracy for BFS $K1$ (especially in the earlier kernel invocations), where most of the device memory allocated initially contains zeroes. Therefore, when a load instruction reads from an incorrect memory address, it is likely to load a zero, which is the same value it would obtain in a fault-free run. Using FI, we find that around 15% of load instructions in BFS $K1$ are affected by this phenomenon, resulting in over-estimation of SDC probability by GPU-TRIDENT.

6.4.3 Heuristics About Error Propagation

The original TRIDENT considers only three instruction types for logical masking, namely comparisons, logical, and cast operations. GPU-TRIDENT also includes multiply instructions based on dynamic profiling (Section 5.3.1). However, other instructions can also mask errors. For example, the $fdiv$ instruction can mask errors while averaging the corrupted bits in the mantissa. This results in GPU-TRIDENT over-estimating SDCs.

*H-Inter Heuristic:* A kernel which has control-flow divergence between threads in a thread block (Figure 5.2b) can result in GPU-TRIDENT missing some inter-thread memory dependencies in the memory dependency graph, which can result in inaccuracies (both under- and over-estimates are possible).
6.5 Limitations

In this section, we describe some of the main limitations of the evaluation described in this thesis.

6.5.1 FI Methodology

We use LLFI-GPU, which injects single bit flips faults at the LLVM IR level. This method has been shown to be accurate for estimating SDC probabilities [15, 33, 45], but its effectiveness for other types of failures is an open question. That said, we focus primarily on SDCs in this thesis, which makes LLFI-GPU an appropriate method for our purposes.

6.5.2 Evaluation Kernels

We choose 17 kernels from 12 benchmarks applications belonging to different domains and suites. However, some of the benchmark kernels are quite small. For example, BFS K2 has only 20 static instructions. We have included benchmarks (Rodinia) and real applications (Circuit, Lulesh, HPCCUDA) that are much larger. Other work in this domain has also used a similar approach [27, 33, 41].

6.6 Summary

In this chapter, we evaluate the accuracy and performance of GPU-TRIDENT against traditional FI methodologies. We observe that the predicted SDC probability closely follow the values obtained from FI. The mean absolute error is 5.7%, with a Pearson correlation coefficient of 0.88. If we do not consider the outliers (3 out of 17 kernels), then this error is reduced to 2.97%, while the new Pearson correlation coefficient is 0.99. GPU-TRIDENT is also found to be accurate in predicting SDC probabilities for individual instructions of kernels. We also find that GPU-TRIDENT has minimal performance overheads compared to FI. It is, on average, 55 times faster than FI in evaluating kernel SDC probability and two orders of magnitude faster than FI for finding the SDC probability of individual instructions.
Chapter 7

Case Studies

In Chapter 6 we evaluated the accuracy and scalability of GPU-TRIDENT against FI. We found that GPU-TRIDENT is highly accurate for most applications, and has a very low performance overhead. This opens up many avenues in reliability analysis, which can not be explored in detail due to the resource-intensiveness of FI.

In this chapter, we use GPU-TRIDENT to study two such applications. The first application is selective instruction duplication guided by GPU-TRIDENT. Selective instruction duplication requires SDC probability of every instruction in a kernel, obtaining them can be very time consuming using FI, as it requires a lot of trials for statistically significant results. As GPU-TRIDENT intrinsically provides per instruction SDC probability, it is a low overhead alternative to FI. Second, we use GPU-TRIDENT to study the variation in SDC probability of GPU kernels across multiple inputs. GPU-TRIDENT provides an efficient alternative to FI for this study, as for larger inputs, FI quickly becomes intractable.

7.1 Selective Instruction Duplication

In this section, we use GPU-TRIDENT to guide selective instruction duplication, which is a standard protection technique to detect SDCs [18, 29, 48]. Selective instruction duplication can provide high error coverage with low-performance overhead. It duplicates most vulnerable instructions and compares their outputs at run-
time to detect any differences as errors [18, 29, 35]. Typically, it is assumed that the maximum performance overhead for the protection is fixed to a budget value. The main question while applying selective instruction duplication is: Which static instructions should be duplicated to reduce the overall program SDC, given a performance overhead budget?. The predominant way to answer this question for GPU applications today is through FI, which is very time consuming, as it requires per instruction SDC probabilities. In contrast, we study the usefulness of GPU-Trident in replacing FI to answer this question, by accurately predicting the SDC percentages of individual instructions.

GPU-Trident gives us the SDC probability of instructions, while the dynamic count for each instruction can be used as a proxy for its overhead. Using this information, we can frame the original question as a classical 0-1 knapsack problem [38]. In this case, maximum allowed overhead is the sack capacity, instructions are the objects while their SDC probability is their associated profit. For simplicity, we consider instruction SDC probabilities to be independent of each other, which is a conservative assumption that has also been made by the previous work [35]. We use GPU-Trident to estimate the SDC probability of each instruction, and use the dynamic instruction count as a proxy of the instruction’s performance overhead (measuring the exact overhead is tedious).

We consider two protection overhead levels, which correspond to one-third and two-thirds performance overhead of duplicating all the instructions in a given GPU kernel. After identifying the instructions to duplicate, for each protection level, we select the instructions to be duplicated, and then we project the results of duplicating these instructions, from actual FI experiments.

Figure 7.1 shows the SDC probabilities of kernels after applying selective duplication. Without protection, the average SDC probability across benchmarks is 33.73%. With one-third protection overhead, the average SDC probability is reduced to 14.2%, which is a reduction of about 58%. With a two-thirds protection overhead, the average SDC probability is further reduced to 5.27%, which is a reduction of about 85%. Note that FI is only used to measure the SDC probability after the protection to validate the predictions made by GPU-Trident.

Figure 7.2 shows the average SDC coverage, for all benchmarks, provided by instruction duplication when FI and GPU-Trident guide it at different overhead
levels. We can see that the protection curve of GPU-TRIDENT closely follows that of FI throughout the range. The maximum difference between these two curves is about 13.34%, which translates to an absolute difference of only 4.4% in terms of SDC percentages. Figure 7.3 gives an overview of the protection curve of selective instruction duplication obtained using FI and GPU-Trident, for individual kernels. We see that protection curves of individual kernels also show a trend similar to Figure 7.2 as the protection provided by GPU-TRIDENT closely follows protection provided by FI for most of the kernels. Thus, GPU-TRIDENT is an efficient and accurate replacement for FI in guiding selective instruction duplication.

7.2 Characterizing SDCs Across Multiple Inputs

As stated earlier, FI is traditionally used to analyze the reliability of GPU applications and is time-consuming as GPU applications can consist of millions of threads. This problem is exacerbated in the case of multiple inputs, as the whole FI process
Figure 7.3: Protection curves obtained by FI and GPU-TRIDENT for individual kernels, as protection overhead is varied from 0% and 100%. Blue lines indicate FI, and orange lines represent GPU-TRIDENT.
Kernel ID | Input 1  | Input 2  | Input 3  
---|---|---|---
BFS K2   | 32,768 | 655,360 | 12,005,376
Hotspot | 473,344 | 9,216 | 1,893,376
Particlefilter | 9,216 | 18,432 | 27,648
LUD K1  | 64 | 256 | 512
LUD K2  | 192 | 3,840 | 15,872
LUD K3  | 3,584 | 317,440 | 2,666,496
SRAD K1 | 32,768 | 524,288 | 2,097,152
SRAD K2 | 32,768 | 524,288 | 2,097,152
NW K1 | 48 | 576 | 2,176
NW K2 | 16 | 448 | 1,920
HPCCUDA | 236,554 | 1,182,720 | 3,548,160
Perf_BM | 8,192,000 | 16,384,000 | 24,576,000

Table 7.1: Number of threads invoked by different kernels, for multiple inputs.

has to be repeated for each input. Further, the fault space of FI can become huge for bigger inputs. In this thesis, we have presented GPU-TRIDENT, which is an efficient alternative to FI. This performance and resource efficiency of GPU-TRIDENT can be used to characterize the reliability of a GPU kernel across multiple inputs.

In this section, we use GPU-TRIDENT to analyze the input-dependence of the error resilience of GPU kernels. We select a subset of benchmarks from Section 6.1.3 for this case study, that give accurate results with GPU-TRIDENT (absolute error < 5% from Section 6.2.1). We also ensure that the selected benchmarks have standard input available from the source of the benchmark, to avoid any biased inputs (e.g., all zeros), which do not represent the real-world resilience characteristics of an application [47]. Table 7.1 shows the selected benchmarks and the number of threads launched by different inputs. Input 1, is the input used in Section 6.1.3, while Input 2 and Input 3 are new inputs.

For this study, we purposefully select new inputs that invoke a relatively large number of threads (up to 24.5 million), as the main benefit of GPU-TRIDENT is that it has very small overhead even for applications with large numbers of threads, in contrast to FI, whose practicality is limited in case of bigger inputs due to high overheads associated with it. In this way, we can study the reliability of GPU
applications under bigger inputs.

Figure 7.4 shows the SDC probability predicted by GPU-TRIDENT for all the benchmarks. We observe from our experiments that the SDC probability of a kernel exhibits little variation across different inputs of different sizes. The average variation across inputs of all benchmarks is 0.56%. The maximum variation in predicted SDC probability is 5.66% between Input 1 and Input 3 for NW K2. LUD K1, LUD K3, HPCCUDA and Perf_BM show no variation across all the inputs. This result is in line with previous studies of the variation in the reliability of GPU kernels across multiple inputs that use FI [47].

We also perform FI experiments to test that accuracy results from the earlier experiments (Section 6.2.1) hold for newer inputs as well. FI has a high execution time for kernels that launch a large number of threads. So we only select kernels that launch a limited number of threads for this experiment and perform FI experiments with Input 2 and Input 3, as we already have the results for Input 1 from Section 6.2.1. The four chosen kernels are LUD K1, LUD K2, LUD K3 and Particlefilter.

We perform 3,000 random FI experiments with the four kernels. We observe
that the average variation of SDC probabilities predicted by GPU-TRIDENT with that of FI is 1.07%, while the maximum variation is 2.24% (for Particlefilter between Input 1 and Input 2). The error bars for these SDC predictions from FI range from 1.76% to 1.78% at 95% confident interval. This shows that predictions by GPU-TRIDENT are in line with the FI results.

The results from this case study thus show that the reliability of GPU kernels does not vary widely across different standard inputs of varying sizes. Therefore, we can use a small set of valid inputs to evaluate the reliability of GPU applications instead of exploring a large input space.
Chapter 8

Conclusion and Future Work

8.1 Summary

GPUs are commonly used to accelerate scientific and safety-critical applications today. Transient hardware faults are increasing in frequency due to progressive technology scaling, which affects the reliability of GPUs.

FI techniques have been traditionally used to assess the reliability of GPU applications, but due to the increasing scale of applications, this has become impractical. This thesis analyzes the reliability characteristics of GPU kernels in a scalable and efficient manner without performing FI.

To get the reliability profile of the GPU applications, we introduce GPU-TRIDENT, which models soft error propagation in GPU kernels without performing any FI. Due to the typically large number of threads in a GPU kernel, it is challenging to model error propagation accurately and scalably. We observe that error propagation in GPU programs can be modeled by carefully analyzing the execution patterns of GPU threads pertaining to data repetition, memory accesses, loop iterations, and control-flow. We find that insufficient details about different levels of memory dependencies in a GPU kernel can cause a significant detrimental effect on accurately modeling the error propagation. On the other hand, accurately getting this information is often the biggest scalability bottleneck in the model.

To address these challenges, we propose several heuristics to prune the error propagation space by selecting only a small subset of threads for analysis of
memory dependencies, based on the characteristics mentioned above, which significantly improves the time required for creating the model. We implemented GPU-TRIDENT as LLVM compiler passes, and evaluated it on 17 GPU Kernels. Our results show that the accuracy of GPU-TRIDENT is comparable to FI both for the kernel as a whole (Pearson correlation coefficient is 0.88) and for individual instructions. Further, GPU-TRIDENT is two orders of magnitude faster, and much more scalable than FI for finding the SDC probabilities of kernels as a whole, as well as individual instructions.

We evaluate the use of GPU-TRIDENT in two applications. First, we found that GPU-TRIDENT can be used to guide selective instruction duplication with comparable accuracy as FI, reducing the SDC probability by approximately 58% and 85% for 1/3rd and 2/3rd overhead, respectively. Second, we deploy GPU-TRIDENT to study the variance of reliability of GPU applications across multiple inputs and find that their SDC probability is generally insensitive to the change in input data.

In conclusion, this thesis presents an accurate and scalable method for modeling error propagation GPU kernels, which can be integrated into the development cycle of real-world applications.

8.2 Future Work

There are three potential areas in which this work can be extended.

8.2.1 Extending GPU-TRIDENT

There are certain sources of inaccuracies in GPU-TRIDENT identified in Section 6.4, these can be analyzed in detail, and their mitigation can be integrated into the proposed techniques to improve their accuracy. Moreover, the GPU-TRIDENT can be extended to analyze the behavior of applications during crashes to predict the crash probabilities of individual instructions. This updated analysis method can then guide Checkpoint/restart (C/R), which is one of the most popular methods, to recover from faults that cause a crash in programs [8, 54]. This method involves identifying the most crash-prone sites in the program (which can be done using GPU-TRIDENT) and saving the program state to resume execution from there if
the application crashes.

8.2.2 End to End Reliability Analysis of GPU Applications

GPU-TRIDENT provides an accurate framework for modeling error propagation within a GPU kernel. A typical GPU application can consist of two parts: the host code that runs on CPU, and device code consisting of all the GPU kernels present in the application. During the execution of the application, data is transferred between these two parts, which can result in error propagating from device code to host code and vice versa. Due to this, the fault activated in one kernel can propagate to other kernels as well. So, to get a complete overview of the reliability of a GPU application, error propagation due to all these interactions needs to be modeled. The techniques proposed in this thesis can be combined with previous ones like TRIDENT [35] to get an end-to-end resilience estimate.

8.2.3 Error Propagation Modeling in Accelerator Platforms

The recent rise of resource-intensive machine learning applications has boosted the interest of academia and industry in domain-specific accelerators (DSA) for machine learning as they provide enhanced performance with significantly less energy budget compared to general-purpose computers. As these accelerators are usually deployed in safety-critical tasks such as autonomous vehicles and business analytics, their reliability warrants an investigation. Although some studies have used fault Injectors for studying the resilience characteristics of accelerators [34, 44], these efforts are bound to run into the same scalability issues of FI, that we have discussed in this thesis. Extending the framework and heuristics proposed in this thesis to these accelerator platforms is a potential direction of future work.
Bibliography


57


58


