Error Resilience Evaluation on GPGPU Applications

by

Bo Fang

Master of Software Systems, The University of British Columbia, 2011

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)

August 2014

© Bo Fang 2014
Abstract

While graphics processing units (GPUs) have gained wide adoption as accelerators for general-purpose applications (GPGPU), the end-to-end reliability implications of their use have not been quantified. Fault injection is a widely used method for evaluating the reliability of applications. However, building a fault injector for GPGPU applications is challenging due to their massive parallelism, which makes it difficult to achieve representativeness while being time-efficient.

This thesis makes three key contributions. First, it presents the design of a fault-injection methodology to evaluate the end-to-end reliability properties of application kernels running on GPUs. Second, it introduces a fault-injection tool that uses real GPU hardware and offers a good balance between the representativeness and the efficiency of the fault injection experiments. Third, it characterizes the error resilience characteristics of twelve GPGPU applications. Last but not least, this thesis provides preliminary insights on correlations between algorithm properties and the measured silent data corruption rates of applications.
Preface

This thesis is based on a work conducted by myself in collaboration with Dr. Karthik Pat- tabiraman, Dr. Matei Ripeanu and Dr. Sudhanva Gurumurthi. The work was published as a conference paper in the 2014 IEEE International Symposium on Performance Analysis of Systems and Software [9]. I was responsible for coming up with the solution and validating it, evaluating the solution and analyzing the results, and writing the paper. Karthik, Matei and Sudhanva were responsible for guiding me with the solution reasoning, experiments design and results analysis, as well as editing and writing portions of the paper.
# Table of Contents

Abstract ................................................................. ii

Preface ................................................................. iii

Table of Contents ....................................................... iv

List of Tables .......................................................... vii

List of Figures .......................................................... viii

List of Acronyms ........................................................ ix

Acknowledgements ...................................................... xi

Dedication ............................................................... xii

1 Introduction ........................................................... 1

2 Background and Fault Model ......................................... 5
    2.1 Dependability Metrics: Error Resilience and Vulnerability ........ 5
    2.2 Characterizing Error Resilience ..................................... 6
    2.3 The Fault Model ..................................................... 8
    2.4 GPU Architecture and Programming Model ......................... 8
# Table of Contents

3 Methodology ................................................................. 10
   3.1 Phase I: Grouping ..................................................... 13
   3.2 Phase II: Profiling ................................................... 16
   3.3 Phase III: Fault Injection ........................................... 17

4 Characterization Study .................................................. 22
   4.1 Benchmarks ............................................................ 22
   4.2 Heuristic Validation .................................................. 24
      4.2.1 Validation of Design Decisions ........................... 24
      4.2.2 Validation of Grouping ..................................... 26
   4.3 Characterization of Error Resilience .............................. 28
   4.4 Statistical Significance of the Fault Injection .................. 31
   4.5 Crash Causes and Latency .......................................... 35
   4.6 Use Cases ............................................................... 37
      4.6.1 Scenario I: SDC Proneness of Different Code Sections .... 37
      4.6.2 Scenario II: Comparing Different Algorithms .............. 38
      4.6.3 Scenario III: Guiding Configurations ....................... 38
   4.7 Limitations .............................................................. 38
   4.8 Summary ................................................................. 39

5 Discussion ................................................................. 42
   5.1 Search-based Application ............................................ 43
   5.2 Bit-wise Operation .................................................... 45
   5.3 Averaged Out .......................................................... 46
   5.4 Graph Processing ...................................................... 47
   5.5 Linear Algebra and Grid Operation .................................. 47
List of Tables

3.1 The group identification process leads to classifying the benchmarks in three
categories. See Table 4.3 for details about the benchmarks . . . . . . . . . . 16

4.1 Fault-injection experiments information . . . . . . . . . . . . . . . . . . . . . 29
4.2 Description of CUDA hardware exceptions . . . . . . . . . . . . . . . . . . . . 36
4.3 Benchmarks properties. LOC: lines of code. Scale: number of blocks in a
grid and number of threads in a block (generally a 3D*3D space). Launch
times: the number of iterations that the kernel is launched. . . . . . . . . . 41

5.1 Benchmark categories and the mapping to the dwarfs of parallelism . . . . . . 50
# List of Figures

3.1 Overview of our fault-injection methodology including grouping, profiling, fault injection and results aggregation .................................................. 12

3.2 Percentage of number of threads in each group to the total number of thread. 
   *Left*: LBM  *Right*: monte carlo. See Table 4.3 for details about the benchmarks 14

3.3 Cumulative distribution function (CDF) of groups of BFS .......................... 15

3.4 Phase III - the fault-injection process .................................................. 17

4.1 The highest number of loop iterations executed by each benchmark kernel. 25

4.2 Comparison of SDC and crash rate for different iteration threshold.  *Left*: 
   SDC rate.  *Right*: crash rate ............................................................ 26

4.3 The instruction classification of two random threads from different groups . 28

4.4 SDC (top) and crash (bottom) rates with error bars representing 95% con- 
   fidence interval for each kernel ......................................................... 30

4.5 Instruction-level error resilience study ................................................ 32

4.6 Accumulated SDC rates for benchmarks ............................................. 33

4.7 Number of dynamic instructions executed per thread by benchmarks .... 34

4.8 Number of fault injection runs that convergence occurs for each benchmark 34

4.9 Root-cause breakdown of crashes for AES and MAT.  *Left*: AES.  *Right*: MAT 35

4.10 Crash latency analysis for AES and MAT.  *Top*: AES  *Down*: MAT ......... 40
List of Acronyms

CUDA  Compute Unified Device Architecture
GPU   Graphics Processing Unit
GPGPU General Purpose GPU
RTL   Register Transfer Language
MPI   Message Passing Interface
SDC   Silent Data Corruption
SIMT  Single Instruction Multiple Thread
SM    Streaming Multiprocessor
ALU   Arithmetic and Logic Unit
LSU   Load-Store Unit
ECC   Error-Correcting Code
PC    Program Counter
PTX   Parallel Thread Execution
ISA   Instruction Set Architecture
List of Acronyms

**SASS** Source and Assembly

**FP** Floating point
Acknowledgements

First of all, I would like to thank my advisors Dr. Matei Ripeanu and Dr. Karthik Pattabiraman for their support during the past three years. Matei and Karthik generously spent their time and energy to help me improve in many ways. Their wisdom, experience and personality build the role model for me.

I would also like to thank my labmates for their advice and suggestions on my work. I really enjoy the time in the lab and work with such great people.

Finally, special thanks to my family. Without them, it wouldn’t be possible for me to be here and accomplish what I have done.
Dedication

To my parents for their love and support for all these years. To my wife and my two daughters that bring me a lot of fun and love in my life.
Chapter 1

Introduction

GPUs were designed originally for applications that were intrinsically fault-tolerant (e.g., image rendering, in which a few wrong pixels might not be noticeable by human eyes). Today, however, GPUs are widely used to accelerate general purpose applications such as DNA sequencing and linear algebra. It therefore becomes critical to understand the behavior of these applications in the presence of hardware faults. This is especially important as the rate of hardware faults increases due to the effects of technology scaling and manufacturing variations [7]. With shrinking process technology, the primary cause of transient faults is random noise. Smaller transistor features require a lower critical charge to hold and change bits, which leads to faster microprocessors, but which also leads to higher transient fault rates. Sheaffer et. al [26] in 2006 states that current trends, expected to continue, show soft error rates increasing exponentially at a rate of 8% per technology generation, hence soft errors will soon become a driving concern for graphics architectures.

GPU manufacturers have invested significant effort to improve GPU reliability. For instance, starting with Fermi models, NVIDIA GPUs support error-correcting code (ECC) to protect register files, DRAM, cache, and on-chip memory space from transient faults. However, transient hardware faults can also occur in the computational or control data paths, and can propagate to registers and/or memory. Such faults would not be detected by ECC, because they would cause the correct ECC in registers and/or memory to be calculated on faulty data. As a result, in spite of these mechanisms, GPU applications
still can be affected by transient hardware faults. Further, hardware-protection techniques such as ECC can incur performance and energy overheads, and hence may not be enabled by users.

The long-term goal of our work is to develop application-specific, software-based fault-tolerance mechanisms for GPGPU applications. As a first step towards this goal, in this thesis we aim to investigate the error-resilience characteristics of these applications by performing fault-injection experiments. Fault-injection is the act of perturbing an application to emulate faults, then studying the effects of those faults on the application outcome [17]. While there has been substantial work in the realm of fault injection for CPU applications [1, 28], there have been relatively few studies that have explored the reliability properties of GPGPU applications and proposed methodologies and tools to support this exploration.

Prior work [35] has performed fault injections at the source-code level (i.e., mutating the source code of a program). Unfortunately, injecting faults at this level is coarse-grained, and does not represent accurately hardware faults that occur at the granularity of micro-architectural units and instructions. To inject hardware faults, the standard approaches are to inject faults into a register transfer language (RTL) model or a microarchitectural simulator [3]. However, these approaches often are considerably slower than execution on the real hardware, and can be a significant bottleneck when performing the thousands of fault-injection experiments, needed for adequate coverage. One way to alleviate the performance bottleneck is to execute only a small section of the application. However, we would not be able to obtain insights into the end-to-end behavior of the application under faults using this approach. In addition, architectural simulators often do not capture error detection and error handling features of the processor, and hence performing injections using such simulators may not be representative of the behaviour on real hardware (We
present the result of error resilience characterization of GPGPU applications using a state-of-the-art GPU simulator in Chapter 4.

To avoid above issues, we choose to perform fault injections at the assembly-language level of GPGPU applications using a GPU-based debugger. While not as detailed as fault injections at the microarchitectural level, this approach allows us to model faults at the granularity of individual instructions, and thus is more precise than injecting at the high-level language level. Compared to the microarchitectural level injectors, this approach is much more efficient and scalable, all the more so because we natively execute the application on the GPU hardware. To the best of our knowledge, we are the first to propose an efficient instruction-level fault-injection tool, GPU-Qin, for GPGPU applications executing on actual GPU hardware.

This thesis makes the following contributions:

1. Proposes a methodology to evaluate the resilience of GPGPU applications and describes the design decisions and the corresponding trade-offs between injection coverage and efficiency to handle the massive parallelism of GPU applications. (Chapter 3),

2. Builds a fault-injection tool, GPU-Qin, that is able to inject faults into applications running on the actual GPU hardware (Chapter 3),

3. Demonstrates the use of the fault injector by performing an end-to-end error-resilience characterization of twelve different GPGPU applications (Chapter 4), and

4. Provides initial insights that explain the error resilience of these applications. (Chapter 5)

We find that there are significant variations in error resilience characteristics of GPU applications. For example, the SDC rates range from 1% to 38% and the crash rates range

\[\text{the tool is available via https://github.com/DependableSystemsLab/GPU-Injector}\]
from 5% to 70% across all benchmarks. We also observe that the main source of crash in GPU systems is the memory-related exceptions. To understand the variations in SDC rates, we find that it is beneficial to consider algorithmic characteristics of the applications.
Chapter 2

Background and Fault Model

This chapter offers background information on the dependability metrics associated with this work, the fault model used, and the NVIDIA GPU architecture and programming model.

2.1 Dependability Metrics: Error Resilience and Vulnerability

Laprie et al. [19] defined "fault-error-failure" chain as follows:

system failure occurs when the delivered service deviates from the specified service. The failure occurred because the system was erroneous: an error is that part of the system state which is liable to lead to failure. The cause in its phenomenological sense of an error is a fault.

The error resilience of a system is defined as its ability to withstand errors should they occur. An error in the program may or may not result in a failure. Errors that do not cause failures are known as benign outcomes. Program failures can be further classified into crashes (i.e., hardware exceptions), hangs, and silent data corruptions (SDCs) (i.e., incorrect outputs). In the context of our work, we define error resilience as the probability that the application does not have a failure outcome (i.e., crash, hang or SDC) after a hardware fault occurs. Different hardware platforms usually feature distinct fault tolerant
2.2. Characterizing Error Resilience

mechanisms, which manifest as exceptions that crash applications or even the whole system. Different applications correspond to different instruction executions, which determines the propagation of the faults. Thus, error resilience is both a property of the platform and the application. Since our evaluation is performed on the same hardware platform, i.e. NVIDIA GPGPUs, error resilience in our context becomes a property of the application alone.

Vulnerability, is the probability that the system experiences a fault that causes a failure (e.g., an SDC). Note that vulnerability is different from error resilience: error resilience is the conditional probability of the program not experiencing a failure given that a fault has occurred. We focus on error resilience in this thesis, because as long as hardware faults don’t propagate to the software level and cause a failure, the system and applications running on the system are unlikely to be problematic. We are interested in developing and evaluating fault-tolerance mechanisms that add minimum overhead for GPGPU applications.

2.2 Characterizing Error Resilience

There are two commonly used methods to evaluate error resilience:

**Beam Testing:** This method refers to the use of neutron source devices (i.e., neutron beams) to shower neutrons on the targets (e.g., systems, boards or components) to trigger radiation-induced faults. Targets exposed to the neutron beam experience higher rates of faults than in operation, thus enabling accelerated testing. The main advantage of this method is that it represents realistic faults. However, the costs associated are high because it requires a neutron source, and it has low controllability. Further, neutron beam time is often limited, which means that the experiment can be run only for a limited time.

**Fault Injection:** This is a procedure to introduce faults in a systematic, controlled manner and study the system’s behavior. Fault-injection techniques can be generally cat-
2.2. Characterizing Error Resilience

eriorized into hardware-based and software-based fault injection. In this thesis, we only consider software-based techniques. Software-based fault-injection techniques typically emulate the effects of hardware faults on the software by perturbing the values of selected data/instructions in the program. Fault injection’s main limitation is that it can be difficult to obtain sufficient coverage and representativeness. However, the method is relatively low-cost because it requires no special equipment. It also offers a high level of controllability and can be repeated as many times as desired. Therefore, we choose fault injection in this work.

As mentioned before, fault injection can be performed at the RTL or micro-architectural levels. However, these methods are not scalable because they require detailed RTL or micro-architectural simulators. For this reason, we perform fault injection at a higher level, namely at the level of assembly code instructions. Our goal is to obtain sufficient coverage in terms of number of instructions executed, rather than the proportion of hardware state covered by the injections, as is typical of RTL/micro-architectural fault injections.

An analogy to such characterization is political polling. Despite the technique details, the idea of opinion polling that is a survey of public opinion from a particular sample, is quite similar to our fault injection study. The whole population is like the total number of instructions executed by an application, and to survey based on a small number of people (usually 1,000 to 10,000) about a topic is like to perform fault injections on randomly-selected instructions. Polls, however, as they are based on samples of populations, are subject to sampling error which reflects the effects of chance and uncertainty in the sampling process. A margin of error is necessary to a survey, which represents the uncertainty of the sampling. In most of cases, a 3% margin of error is measured based on a group of 1,000 people with 95% confidence, and 1% margin of error is computed when the group is increased to 10,000.
2.3 The Fault Model

Hardware faults can be broadly classified as transient or permanent. Transient faults usually are "one-off" events and occur non-deterministically, while permanent faults persist at a given location. Further, transient faults are caused by external events such as cosmic rays and over-heated components, while permanent hardware faults are usually caused by manufacturing or design faults. Transient fault rates have been increasing due to diminishing noise margins, smaller voltages, and shrinking microprocessor feature sizes [5]. We focus on transient faults in this study.

We consider transient faults in the functional units of the GPU processor. Examples are faults in the arithmetic and logic unit (ALU) and the load-store unit (LSU). We do not consider faults in cache, memory, and register files because we assume that they are protected by ECC. This is the case for recent GPUs such as the NVIDIA Fermi GPU.

We use the single-bit-flip model in this study because it is the de-facto fault model adopted in studies of transient faults [13, 34, 35]. However, our fault injector can support both single- and multiple-bit flips by choosing corresponding fault generation functions at no cost.

2.4 GPU Architecture and Programming Model

We focus on GPGPU applications implemented on top of NVIDIA Compute Unified Device Architecture (CUDA), a widely adopted programming model and toolset for GPUs. The CUDA programming model defines a GPU application as a control program that runs on the host and a computation program (i.e., the kernel) that runs on GPU devices without interfering with the CPU. The kernel is implemented as a collection of functions in a language that is similar to C, but has annotations for identifying GPU code and for delineating
different types of memory spaces on the GPU.

CUDA kernels use a single instruction/multiple thread (SIMT) model that exploits the massive parallelism of GPU devices. From a software perspective, CUDA abstracts the SIMT model in the following hierarchy: kernels, blocks and threads. A CUDA kernel consists of blocks, and a block consists of threads. Fine-grained data parallelism, thread parallelism, coarse-grained data parallelism, and task parallelism can all be provided through this hierarchy. From a hardware perspective, blocks of threads run on hardware units named streaming multiprocessors (SMs) that feature a shared memory space for threads inside the same block. Inside a block, threads are scheduled in a fixed groups of 32 threads called warps. All the threads in a warp execute the same instructions, but with different data values.

In the CUDA programming model, there are four kinds of memory: (1) global, (2) constant, (3) texture, and (4) shared. Global, constant, and texture memory accesses are served from the slower large device memory. Shared memory space is a much smaller and faster on-chip software-managed cache. CUDA applications need to be aware of the memory hierarchy to access GPU memory efficiently.
Chapter 3

Methodology

This chapter outlines our methodology to characterize the error resilience of GPGPU applications and the tradeoffs we make to balance coverage and efficiency. To support our methodology, we develop GPU-Qin, a profiler and fault injector.

Any fault-injection methodology should satisfy the following three requirements:

1. **Representativeness**: The faults injected should be representative of the actual hardware faults that occur at runtime. In particular, the faults should be injected uniformly over the set of all instructions executed by the application. We assume that each dynamic instruction carries the same probability of the fault occurrence. This is a different criterion than used by RTL-level and micro-architectural fault injections, as discussed in Chapter 2.

2. **Efficiency**: Fault-injection experiments should be fast enough to allow the application to be executed to completion in reasonable time. The reason is that thousands of faults-injection experiments need to be performed to obtain statistically significant estimates of error resilience.

3. **Minimum Interference**: The tools supporting the fault-injection experiments should interfere minimally with the original application so that they do not modify its resilience characteristics. In particular, the fault injector should not change either the code or the data of the application other than for the objective of injecting the
faults themselves, and should not impose unreasonable performance overheads on the applications.

We implement our methodology based on the CUDA GPU debugging tool namely *cuda-gdb*\(^2\). The *cuda-gdb* interface provides an external method to control the application, and to trace/modify it without making any changes to the application code or data. This makes it possible to satisfy the minimum interference goal. *cuda-gdb* introduces timing delays in the application; however, we have not seen any cases in which there is considerable deviation in the behavior of the application due to such delays, because our focus is not graphics applications but general-purpose applications.

Figure 3.1 shows an overview of our methodology. The process consists of four main phases, which we briefly describe here and detail in the rest of the this chapter. In the first phase, we group threads based on similarity of their behaviors (we use the number of instructions executed as a proxy, because threads executing a different number of instructions likely execute different control-flow paths, and hence have divergent behaviors) by running applications with GPGPU-Sim \(^3\). This is because GPU applications usually launch thousands or tens of thousands of threads and it is extremely time-consuming to evaluate the error resilience of each GPU thread. Instead, we only consider representative groups to study. We then choose one thread from each group to profile in the next phase.

To balance coverage and efficiency, in some cases we use only the most popular groups, as we detail in Chapter 3.1.

In the second phase, GPU-Qin profiles the threads selected in the first phase and obtains the execution trace of the GPU portion of the application. This information is used to map the source code lines to the executed assembly instructions. This information is necessary in the next phase to locate at runtime the instruction at which to stop execution and inject

\(^2\)https://developer.nvidia.com/cuda-gdb
Chapter 3. Methodology

Figure 3.1: Overview of our fault-injection methodology including grouping, profiling, fault injection and results aggregation.

The fault.

In the third phase, for each injection run, GPU-Qin randomly chooses one executed instruction from one of the traces obtained in the second phase. The choice of the trace is biased proportionally based on the popularity of the group it represents. The choice of the instruction is done uniformly over the space of the instructions of the profile; thus, GPU-Qin simulates the occurrence of a transient error that occurs uniformly over instruction (in other words, we assume that all instructions take approximately the same time to execute). GPU-Qin also randomly picks a thread from the entire set of application run-time threads for each injection run. This satisfies the representativeness requirement.
Finally, the last phase aggregates the results. The rest of this chapter presents each phase in detail.

3.1 Phase I: Grouping

GPU applications often have a massive number of threads, and it would be infeasible to obtain the execution traces for all threads in an application kernel for the purpose of fault injection. Therefore, the main challenge is to identify a fraction of threads that are representative of the workload behavior for tracing. To this end, we separate threads into groups to find representative threads from the groups. We identify the groups based on the behaviours of the threads and consider instruction counts as a proxy for thread behaviour. That is to say, threads in the same group should execute the same number of instructions.

Because GPUs don’t have built-in instruction counters, we gather the instruction counts of all threads in a benchmark by executing the program in an instruction-level GPU simulator, GPGPU-Sim (version 3.2.0). GPGPU-Sim simulates the execution of GPGPU programs from both functional and performance perspectives, and hence the number of instructions executed by it matches the number of instructions executed in the real hardware. We perform the group identification operation only once per application, so it is acceptable for this phase to be slower than the fault-injection phase, which is performed thousands of times. We then group the threads executing the same number of dynamic instructions.

We find that our benchmarks (presented in detail in Chapter 4 and Table 4.3) can be categorized into three categories based on the results of the group identification process (Table 3.1). In the first category, all threads execute the same number of instructions,
3.1. Phase I: Grouping

Figure 3.2: Percentage of number of threads in each group to the total number of thread. Left: LBM Right: monte carlo. See Table 4.3 for details about the benchmarks

and hence there is only one group. In the second category, there is a limited amount of divergence among the threads, which leads to only a few groups (2 to 10). Finally, in the third category, there is significant divergence leading to tens of groups or more.

Because profiling a thread is time-consuming, to balance coverage and efficiency, we propose the following method: for applications in which there is only one group, we randomly choose a single thread in the group to profile; for applications with a small number of groups, we select the groups that constitute the majority of the threads and randomly pick one thread from each selected group to profile. Figure 3.2 shows two examples of how we pick such major groups. For example, LBM has two groups: one has 84% and the other has 16%, of the total number of threads. To satisfy the representativeness requirement, we need to pick both groups. However, in other cases, we ignore some less popular groups. For example, Monte Carlo has five groups, but one of the groups is responsible only for 0.4% of total number of threads, and hence we ignore that group; for applications that have a large number of groups (in our benchmark set, only BFS (Table 3.1)), we again use group popularity to make informed choices. For BFS, around 60% of threads fall into the same
3.1. Phase I: Grouping

group (shown as a vertical line in Figure 3.3), while all the other 78 groups are equally
popular; therefore, we pick a random thread from the large group and another random
thread from the other groups. Given enough resources, more groups can be sampled to
increase coverage. In Chapter 4.2.2, we report the result of a random fault injection experi-
ment to validate the grouping. For applications that have a large number of groups (in our

![Figure 3.3: Cumulative distribution function (CDF) of groups of BFS](image)

benchmark set, only BFS (Table 3.1), we again use group popularity to make informed
choices. For BFS, around 60% of threads fall into the same group (shown as a vertical
line in Figure 3.3), while all the other 78 groups are equally popular; therefore, we pick a
random thread from the large group and another random thread from the other groups.
Given enough resources, more groups can be sampled to increase coverage.
Table 3.1: The group identification process leads to classifying the benchmarks in three categories. See Table 4.3 for details about the benchmarks

<table>
<thead>
<tr>
<th>Category</th>
<th>Benchmarks</th>
<th>Groups</th>
<th>Groups to profile</th>
<th>% threads in picked groups</th>
</tr>
</thead>
<tbody>
<tr>
<td>Category I (one group)</td>
<td>AES, MRI-Q, MAT, MergeSort-k0, Transpose</td>
<td>1</td>
<td>1</td>
<td>100%</td>
</tr>
<tr>
<td>Category II (2-10 groups)</td>
<td>SCAN, Stencil, Monte Carlo, SAD, LBM, HashGPU</td>
<td>2 - 10</td>
<td>1 - 4</td>
<td>95% - 100%</td>
</tr>
<tr>
<td>Category III (&gt;10 groups)</td>
<td>BFS</td>
<td>79</td>
<td>2</td>
<td>&gt;60%</td>
</tr>
</tbody>
</table>

3.2 Phase II: Profiling

The goal of the profiling phase is to map the assembly-level instructions (i.e. SASS) executed by a thread (chosen during the grouping phase) to their corresponding CUDA source-code line. This will enable GPU-Qin which uses conditional breakpoints to inject faults. The reason is that *cuda-gdb*, on which GPU-Qin is built, requires the source line number for setting a conditional breakpoint. Mapping a source line to assembly instructions is one-to-many. (i.e., a single source line may correspond to multiple instructions). We will explain later how GPU-Qin locates the specific assembly instruction to inject to.

The profiling phase consists of single-stepping the program using *cuda-gdb* for the thread(s) selected in the first phase. At each step, the program counter value of the instruction is recorded, along with the instructions corresponding to the source line. The output of the profiling step is an instruction trace consisting of the program counter values and the source line associated with each instruction.
3.3 Phase III: Fault Injection

The third phase of the process is to inject faults into the application at runtime and monitor the outcomes. Figure 3.4 briefly illustrates this process. GPU-Qin has instruction traces from the second phase and it obtains the associated source code line for each instruction from each trace. In each injection campaign, GPU-Qin chooses a profile from the profiling phase and uniformly chooses an instruction; to inject a fault, it sets up a conditional breakpoint in the program at the source code line corresponds to that instruction using cuda-gdb. The conditional breakpoint is triggered only when the chosen thread reaches the chosen source line. When the breakpoint is triggered and the chosen instruction is reached, a fault is injected into the application. The application is then monitored to determine if the fault is activated (i.e., the modified state is read by the application). To ensure representativeness, the thread coordinate is chosen randomly from the set of all threads used by the program, rather than only from the ones chosen during the grouping phase. The application runs natively on the hardware until the breakpoint is triggered and after the fault is injected (except for a short window of time when it is single-stepped to monitor fault activation). This satisfies the efficiency requirement. The fault injection is repeated until the 95% confidence interval is reached for the results with the error bar that is between 1% to 2%.

The rest of this chapter presents the details of this process.

![Figure 3.4: Phase III - the fault-injection process](image-url)
3.3. Phase III: Fault Injection

Reaching the target instruction: After the breakpoint is set, the program is launched under cuda-gdb, and it runs natively until the conditional breakpoint is hit. Because multiple dynamic instructions can map to the same source line, the breakpoint being hit does not mean that the target instruction is reached. To reach the target instruction, GPU-Qin performs two steps:

1. GPU-Qin estimates in which iteration of a loop the target instruction occurs in (if it belongs to a loop). It can perform this estimate based on the information gathered in the profiling phase. If the current loop iteration is less than the estimated iteration, GPU-Qin increments the iteration count and continues the program natively until the next time the conditional breakpoint is reached. To optimize the injection process, GPU-Qin bounds the loop iteration estimate at 64. In other words, if the iteration that needs to be injected exceeds 64, GPU-Qin generates a random number between 0 and 64 and injects a fault at the corresponding loop iteration. We examine the implications of this heuristic in the next chapter.

2. Once the current loop iteration matches the target iteration, GPU-Qin single-steps the program from the breakpoint until the program counter matches the instruction we want to inject. For performance reasons, GPU-Qin uses a fixed window to limit the number of times the single-stepping is invoked. If this window has been exceeded and the target instruction has not been reached, GPU-Qin abandons the run. Currently, GPU-Qin uses 300 instructions as the window size because we find that most source lines correspond to at most a few tens of instructions. This window’s size can be configured by the user.

We explain the details of the fault injection experiment in this context:
3.3. Phase III: Fault Injection

The locations to inject: The locations to inject depend on the instruction executed. GPU-Qin considers three types of instructions:

1. Arithmetic instruction: GPU-Qin injects faults into the destination register of instructions to simulate an error in the ALU and floating-point (FP) unit. For vector instructions that have multiple destination registers, GPU-Qin randomly chooses a destination register to inject.

2. Memory instructions: GPU-Qin simulates faults in the LSU by injecting faults into either the destination register or the address register in LD/ST instructions.

3. Control-flow instruction: NVIDIA ISA uses predicate registers to control the branches of the program. Instructions such as "ISETP" are used to set values to the predicate registers and an optional predicate guard is used to control the conditional execution. Unfortunately, cuda-gdb does not let us modify the predicate registers, so GPU-Qin injects faults into the source operands of the control-flow instructions, instead of directly manipulating the predicate registers.

The fault: A fault is injected by flipping a randomly chosen single bit in the result of the instruction’s destination register. Only one fault is injected in each run because hardware faults are relatively rare events compared to the execution time of a typical application.

Successful fault injections: A fault might not be injected in a run even when the instruction is reached. This can occur either because cuda-gdb will not allow us to modify the instruction, or because the thread GPU-Qin randomly picks does not execute the corresponding instruction (because choosing the thread for injection is based on all threads but the profile comes from a particular group of threads). GPU-Qin discards the executions that do not lead to fault injections. For example, GPU-Qin is not allowed to change the
3.3. Phase III: Fault Injection

address involved in BRA (which is a branch instruction to jump a relative address).

Activated fault: Once a fault is injected, GPU-Qin checks if the faulty location is read by the program (and not overwritten). Such faults are said to be activated. Only activated faults are considered in the evaluation because our goal is to measure the application’s resilience (the conditional probability that given a fault, the program is able to work correctly). To track the activation of a fault, GPU-Qin single-steps the program after injection to check if there is another instruction that reads registers modified by the fault. To ensure that this process terminates in a reasonable amount of time, GPU-Qin picks a threshold: the activation window. If the fault is not activated within the activation window instructions after injection, GPU-Qin lets the program continue and consider the fault unactivated. We set the window to be 1600 instructions for our experiments. We explore the implications of this choice in the next chapter.

Execution Outcome: If the fault is activated, the application execution has one of the following outcomes: (1) Throws an exception (crash), (2) Times out by going into an infinite loop (hang), (3) Completes with incorrect output (SDCs) or (4) Completes with correct output (benign). These four outcomes are mutually exclusive and exhaustive.

Error Bars: the formula we use for calculating the error bars within a confidence interval, when we do not know the mean and standard deviation is as follows:

\[ p \pm \frac{z \times \sqrt{p \times (1-p)}}{n} \] (3.1)

where 'p' is the coverage for target, e.g. SDC rate or crash rate and n is the activated fault injection runs. The 'z' value is what is called the normal score. A 'z' value of 0.95 means that 95% of the area under a normal curve lies within roughly 1.96 standard deviations from the mean.

\[ ^{4}\text{We define an SDC as an outcome that fails the correctness check of the benchmark (if one is provided), or output mismatch between fault-free and fault-injected runs if a correctness check is not provided. Thus, we take application-specific characteristics into account in our definition of an SDC.} \]
3.3. Phase III: Fault Injection

deviations of the mean, and due to the central limit theorem, this number is therefore used in the construction of approximate 95% confidence intervals [25]. We can also tell from equation 3.1 that when \( p \) is equal to 10,000 the error bar is less than 1%. As long as we use random selection on the instructions to inject, estimations drawn on a small number of fault injection runs represents the characteristics of a whole program.

In summary, this chapter introduces the design principles our fault injection methodology, and describes the three phases of the methodology including grouping, profiling and fault injection to characterize the GPGPU applications. We also explain types of faults, type of instructions and important details of experiments.
Chapter 4

Characterization Study

This chapter uses a wide variety of applications (presented in Chapter 4.1) to validate the design choices that we made (Chapter 4.2) and to demonstrate the use of our methodology to characterize the application’s error resilience (Chapter 4.3). All of our experiments are conducted on NVIDIA Tesla C series GPUs.

4.1 Benchmarks

We use a variety of benchmarks from the Parboil benchmark suite [29], NVIDIA CUDA SDK package, Rodinia benchmark suite [6], and other well-known GPGPU applications. A short description of each benchmark is given below, along with the inputs used in our evaluation. Table 4.3 summarizes the characteristics of each benchmark and its kernels.

**AES encryption (AES):** AES supports both encryption and decryption. We encrypt a 256-KB file with a 256-bit key.

**Matrix Multiplication (MAT):** Matrix multiplication is a common building block widely used in many linear algebra algorithms. We modify the code so that MAT launches the CUDA kernel code only once, to ensure that subsequent runs do not overwrite the results. We multiply two 192*128 floating-point matrices.

**Matrix Transpose:** Matrix transpose is a common building block for many linear algebra algorithms. We use the diagonal kernel optimized for the highest memory bandwidth.
4.1. Benchmarks

We transpose a 512*512 floating-point matrix.

**Monte Carlo (MONTE):** MONTE simulates the price of an underlying asset using the Monte Carlo method. We let it simulate 262,144 paths for 256 options.

**GPUs as Storage System Accelerators (HashGPU):** HashGPU [18] is a library that accelerates a number of hash-based primitives. We use both SHA1 and MD5.

**Breadth-First Search (BFS):** BFS applies a breadth-first search on a graph. We perform BFS on a random graph with 4096 nodes.

**Magnetic Resonance Imaging - Q (MRI-Q):** MRI-Q computes a matrix, representing the scanner configuration for calibration, used in a 3D MRI reconstruction algorithms in non-Cartesian space. We use 32*32*32 as the size of the 3D matrix.

**3-D Stencil Operation (Stencil):** Stencil performs an iterative Jacobi stencil operation on a regular 3-D grid. We use a 128*128*32 3D FP matrix and iterate the operation five times to make it converge.

**Sum of Absolute Differences (SAD):** SAD computes the sum of absolute differences, used in MPEG video encoders. It is based on a full-pixel motion-estimation algorithm found in the JM reference H.264 video encoder. There are three kernels in this benchmark and each kernel uses the previous kernel’s output. We use the default data frame as the initial input.

**CUDA Parallel Prefix Sum (SCAN):** SCAN [16] demonstrates an efficient CUDA implementation of a parallel prefix sum. Given an array of numbers, SCAN computes a new array in which each element is the sum of all the elements before it in the input array. We include SCAN-block, which works with any length of arrays.

**Merge Sort (MS):** MergeSort [24] implements a merge-sort, representing a use case of GPUs for sorting batches of short- to mid-sized (key, value) array pairs.
4.2 Heuristic Validation

Lattice-Boltzman Method Simulation (LBM): LBM implements a solution of the system of partial differential equations for fluid simulation, which can be derived for the propagation and collision of fictitious particles. The input file is a discrete representation of immobile flow obstructions (120,120,150) in the simulated volume.

4.2 Heuristic Validation

This chapter offers empirical support for the heuristics used in Chapter 4. All these heuristics (including the grouping strategy and design decisions we make) represent choices in the trade-off space between coverage (either in terms of distinct code paths profiled or used for fault injection) and efficiency (run-time to execute an application characterization).

4.2.1 Validation of Design Decisions

There are two design decisions we make to ensure the efficiency of the fault injector, namely:

1. To control runtime, we limit the number of loop iterations explored. That is, if the instruction to be injected belongs to a loop iteration that exceeds a threshold of 64, we generate a random number between 0 and 64 and inject a fault at the corresponding loop iteration.

2. If the injected fault is not activated within an activation window of 1,600 dynamic instructions, we consider it unactivated.

To validate the first heuristic, we first count the total number of iterations executed by each loop of each kernel, and then consider the loop with the largest number of iterations. The results are shown in Figure 4.1. We disregard applications that execute fewer than 64 iterations (in all loops) because they fall within the chosen threshold already. Among the four applications that have loops that exceed the threshold, we pick MRI-Q, which has the
largest number of iterations, and MAT, which has the smallest number of iterations still
greater than the threshold, and vary the threshold from 64 to 32 and 128 and repeat the
characterization experiments.

Figure 4.2 presents the SDC rates and crash rates for MAT and MRI-Q for max-iteration
threshold values of 32, 64, and 128. We find that varying this threshold does not affect the
resulting SDC rate and crash rate for these benchmarks. This indicates that limiting the
number of iterations does not affect the overall error resilience estimation. Although the
number of iterations we limit in our study is representative for our benchmark suite, it is
still possible that applications which have different sets of characteristics in terms of loop
iterations may need further investigation and to get adjusted accordingly.

Figure 4.1: The highest number of loop iterations executed by each benchmark kernel.

To validate the second decision, we count the number of instances when the activation
window threshold is exceeded. We find that for only three benchmarks (HashGPU-sha1,
MAT and MRI-Q) are there fault-injection runs in which the activation window is exceeded:
4.2 Heuristic Validation

Figure 4.2: Comparison of SDC and crash rate for different iteration threshold. *Left:* SDC rate. *Right:* crash rate

two cases in HashGPU-sha1, 36 in MAT, and 29 in MRI-Q. However, the proportion of these is negligible, compared to the thousands of fault-injected runs executed for each benchmark. Thus our choice of the activation window size leads to only minimal inaccuracy (about 1% for three benchmarks) in evaluating error resilience.

4.2.2 Validation of Grouping

The purpose of grouping is to identify the representative threads from a large amount of threads of an GPU application. In Chapters 3.1 we state that we use the number of dynamic instructions executed by a thread as a representation of the behaviours, and we group a program’s threads based on their behaviours. This heuristic can be validated through an instruction level classification analysis. Therefore, we analyze two example applications from category I and II, namely MAT and SAD to check if threads in the same group would execute different sets of instructions. We don’t consider category III because in this context there is no fundamental difference between applications in category II and III as they both contain multiple groups. The details of the validation are described below:

*MAT:* Since we found that in MAT all the threads execute the same number of instructions, we want to understand if the same number of instructions implies identical instruc-
4.2. Heuristic Validation

tions executed by each thread. By using GPU-Qin, we randomly choose 100 threads in MAT and record instructions of each thread to get 100 profiles. Then, we examine every profile with each other and discover that all of them are identical. This shows that our grouping strategy for applications like MAT is effective.

SAD: As we show in Table 3.1, SAD-k0 contains 5 groups, each of which contains threads that execute 1989, 1856, 1948, 1995 and 950 PTX instructions. To see how many deviations of the threads in different groups, we pick a random thread from two most popular groups (1956 and 1948) separately to classify the instructions. Figure 4.3 shows the classification of instructions executed by the two threads we pick. In total, there are 11 categories of instructions in Fermi SASS ISA, and we present the break-down of the number of executed instructions in each category. The amount of instructions differs between two threads is 786, which constitutes of about 2% of total instructions. Most of the differences come from the integer and move instructions (INT and MOV). To show if the grouping works for applications that have multiple groups, we randomly profile two threads from each group of SAD-k0 and compare the dynamic instructions they execute, and find that they are also identical. We can use such low-level instruction classification to reason about the behaviours of threads in different groups as well as the error resilience characteristics in the future.

In summary, we show that using number of instructions as the representation of the thread behaviours is valid because the threads in the same group execute the same stream of instructions. It is also inferred that threads in different groups would show different error resilience characteristics. To validate this point, we compare the fault-injection results of applications in categories II and III (see Table 3.1). The crash rates vary considerably for different groups of threads in Stencil, LBM, SCAN and BFS, which is 5%, 10%, and 25% respectively. This demonstrates the value of considering grouping that is a key strategy in
4.3. Characterization of Error Resilience

We characterize the error resilience of the 15 kernels mentioned. We run enough experiments to obtain 95% confidence, with a 1% to 2% (depending on the benchmark) confidence interval for the SDC rate and crash rate.

Table 4.1 presents, for each benchmark, the total number of injected runs, the overall activation rate, and the average time for a fault-injection run. The total number of injected runs includes runs when the fault was injected successfully and was either: activated, overwritten, or ignored by exceeding the activation window.

The average time of each fault-injection run varies across benchmarks from 11 seconds to 710 seconds, and is directly proportional to the scale of the block size of the benchmark (shown in Table 4.3). We observe that our worst-case benchmark SCAN, which takes 710 seconds on average, is still 10X faster with GPU-Qin than running with GPGPU-Sim. Other benchmarks show speedups as high as 100x. Moreover, the simulator needs days to

Figure 4.3: The instruction classification of two random threads from different groups

4.3 Characterization of Error Resilience

We characterize the error resilience of the 15 kernels mentioned. We run enough experiments to obtain 95% confidence, with a 1% to 2% (depending on the benchmark) confidence interval for the SDC rate and crash rate.

Table 4.1 presents, for each benchmark, the total number of injected runs, the overall activation rate, and the average time for a fault-injection run. The total number of injected runs includes runs when the fault was injected successfully and was either: activated, overwritten, or ignored by exceeding the activation window.

The average time of each fault-injection run varies across benchmarks from 11 seconds to 710 seconds, and is directly proportional to the scale of the block size of the benchmark (shown in Table 4.3). We observe that our worst-case benchmark SCAN, which takes 710 seconds on average, is still 10X faster with GPU-Qin than running with GPGPU-Sim. Other benchmarks show speedups as high as 100x. Moreover, the simulator needs days to
### 4.3. Characterization of Error Resilience

Table 4.1: Fault-injection experiments information

<table>
<thead>
<tr>
<th>Kernels</th>
<th>Injected runs</th>
<th>Activated runs</th>
<th>Activation rate</th>
<th>Average time per run (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>AES</td>
<td>2,351</td>
<td>2,042</td>
<td>87%</td>
<td>84</td>
</tr>
<tr>
<td>HashGPU-md5</td>
<td>2,699</td>
<td>2,683</td>
<td>99%</td>
<td>13</td>
</tr>
<tr>
<td>HashGPU-sha1</td>
<td>2,400</td>
<td>2,305</td>
<td>96%</td>
<td>27</td>
</tr>
<tr>
<td>MRI-Q</td>
<td>2,830</td>
<td>2,475</td>
<td>87%</td>
<td>123</td>
</tr>
<tr>
<td>MAT</td>
<td>2,575</td>
<td>2,186</td>
<td>85%</td>
<td>82</td>
</tr>
<tr>
<td>Transpose</td>
<td>2,395</td>
<td>2,160</td>
<td>90%</td>
<td>44</td>
</tr>
<tr>
<td>SAD-k0</td>
<td>2,671</td>
<td>2,435</td>
<td>91%</td>
<td>76</td>
</tr>
<tr>
<td>SAD-k1</td>
<td>2,208</td>
<td>2,195</td>
<td>99%</td>
<td>26</td>
</tr>
<tr>
<td>SAD-k2</td>
<td>2,627</td>
<td>2,618</td>
<td>100%</td>
<td>12</td>
</tr>
<tr>
<td>Stencil</td>
<td>2,426</td>
<td>2,148</td>
<td>89%</td>
<td>31</td>
</tr>
<tr>
<td>SCAN-block</td>
<td>1,083</td>
<td>1,080</td>
<td>99%</td>
<td>710</td>
</tr>
<tr>
<td>MonteCarlo</td>
<td>3,744</td>
<td>2,723</td>
<td>73%</td>
<td>66</td>
</tr>
<tr>
<td>MergeSort-k0</td>
<td>1,930</td>
<td>1,884</td>
<td>98%</td>
<td>359</td>
</tr>
<tr>
<td>BFS</td>
<td>2,334</td>
<td>2,330</td>
<td>100%</td>
<td>22</td>
</tr>
<tr>
<td>LBM</td>
<td>1,895</td>
<td>1,845</td>
<td>97%</td>
<td>165</td>
</tr>
</tbody>
</table>

finish for some applications and hence the speedups for those applications are definitely greater than 100x; however, we did not measure these speedups. The average speedup across benchmarks (that the simulator is able to finish within a couple of hours) is 22x. This demonstrates the efficiency of GPU-Qin.

Figure 4.4 presents the SDC rate and crash rate of the benchmark kernels. We do not show the hang rates because they are uniformly lower than 1%. Fault injections in CPUs exhibit similar hang rates [13] because hangs occur when the number of loop iterations is increased so significantly that the benchmark times out. This case is relatively uncommon in practice.

At a first glance, both the SDC rate and the crash rate vary widely across benchmarks. In particular, the SDC rate ranges from 0.5% to nearly 38%. This observation suggests that it is important to take into account the inherent error resilience characteristics of an
4.3. Characterization of Error Resilience

Figure 4.4: SDC (top) and crash (bottom) rates with error bars representing 95% confidence interval for each kernel
4.4. Statistical Significance of the Fault Injection

application when protecting it from SDC-causing errors. For example, the SDC rate for MONTE is less than 1%, likely because the result of simulating each path will eventually be aggregated, which potentially mitigates the effect of faults. We note that similar applications in terms of application behavior, (e.g., HashGPU-sha1 and HashGPU-md5 as well as SAD-k1 and SAD-k2) exhibit similar SDC rates. On the other hand, crash rates vary even more than the SDC rates, from 6% to 71%. We discuss the possible reasons behind these variations in the next chapter. In total, across all benchmarks, failure rates (crash+SDC+hang) range from 24% (MONTE) to 93% (SCAN), and the average failure rate is 67%.

Understanding the error-proneness of instructions is a important aspect of the error resilience characterization because it can help determine which portions/parts of the program need to be protected. NVIDIA Fermi SASS instructions can be generally classified into the following classes: FP, ALU, memory and control. For all activated fault injections, we investigate which class the fault-activated instruction belongs. Figure 4.5 presents the SDC rates for each class of instruction for all benchmarks. The overall SDC rates are displayed for further reference. We find that across all benchmarks, no categories have significant correlations with the overall SDC rates. Our result suggests that a specific SDC rate of a GPGPU program is not simply determined by any single factor (i.e. type of instructions) but a combination of various of factors.

4.4 Statistical Significance of the Fault Injection

Figure 4.4 shows that we confine the error bar of our fault injection experiment within 2%. However, the measured SDC rate is necessarily significant based on a sample of instructions of applications. In our fault injection experiment, we inject about 2000 to 3000 faults for an

\footnote{Our categorization is based on a general classification of the purposes of the instructions}
4.4. Statistical Significance of the Fault Injection

Figure 4.5: Instruction-level error resilience study

Application by randomly selecting an instruction from all dynamic instructions. Therefore, the number of fault injection runs does not depend on the number of dynamic instructions an application usually executes. This situation raises a question: does the number of activated fault injection runs depend on the size of the programs?

To answer this question, we use an empirical method as follows. We calculate the SDC rate for different size of samples. For example, we calculate the SDC rate for the first 200 injection runs and we repeat it for the first 400 injection runs, and so on. In Figure 4.6, we illustrate the trend of SDC rates for each benchmark for different ample sizes. We find that SDC rates fluctuate a little in most of benchmarks (even the maximum variation is only 6% in HASHGPU-sha1), but all of them stabilize on constant values near the end where the fluctuations are within the error bars.

Figure 4.7 shows how many dynamic instructions are executed per thread in each benchmark. For benchmarks in category I that is described in Chapter 3, we simply
4.4. Statistical Significance of the Fault Injection

Figure 4.6: Accumulated SDC rates for benchmarks

report the number of instructions executed by a random thread because all the threads execute the same number of instructions. For benchmarks in category II, we report the number of instructions of a thread that belong to the most popular group. For example, we pick a thread in group 2 (Figure 3.2) of LBM as it is the most popular group. For the benchmark in category III (i.e. BFS), we randomly pick one thread to report as what we did in the grouping phase. However, a wide spectrum of dynamic instructions of these benchmarks shown in Figure 4.7 along with the number of runs which convergence occurs for each benchmark (shown in Figure 4.8) in fact suggests that the number of needed fault injections for a statistically significant estimation on the error resilience of applications (i.e. SDC rate) does not depend on the number of executed instructions of the programs. For instance, MRI-Q and SAD-k0 execute the largest number of instructions, but the SDC rates for them converge around 2200 runs. In contrast, LBM executes only a small number
of instructions, but the SDC rate still varies by 1% from 2600 runs to 2800 runs.

![Number of dynamic instructions executed per thread by benchmarks](image1)

Figure 4.7: Number of dynamic instructions executed per thread by benchmarks

![Number of fault injection runs that convergence occurs for each benchmark](image2)

Figure 4.8: Number of fault injection runs that convergence occurs for each benchmark
4.5 Crash Causes and Latency

GPU-Qin can be used to gain a deeper understanding of the error-resilience characteristics of GPGPU applications. In this chapter, we attempt to understand the reasons for the crashes observed in the characterization study, and characterize the crash latency. This investigation is important for two reasons. First, crashes are a form of error detection performed by the GPU hardware and CUDA run-time, and understanding the reasons for crashes can help understand the effectiveness of the existing error-detection mechanisms. Second, it is important to detect the crashes early to contain the errors. We report results for only two benchmarks, AES and MAT; however, the observations generalize to all the benchmarks.

When a hardware exception occurs, the application crashes and the crash cause is reported to cuda-gdb. GPU-Qin traps these exceptions and logs them. Overall, we observe four types of hardware exceptions: lane user stack overflow, warp out-of-range address, warp misaligned address and device illegal address. The exceptions and their causes are presented in Table 4.2.

Figure 4.9 shows the root causes for crashes in the applications. The two most common causes are warp out-of-range addresses and device illegal address. We find that warp misaligned address also plays an important role in crashes in the MAT benchmark.
4.5. Crash Causes and Latency

Table 4.2: Description of CUDA hardware exceptions

<table>
<thead>
<tr>
<th>Exception type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lane user stack overflow</td>
<td>Occurs when a thread exceeds its stack memory limit</td>
</tr>
<tr>
<td>Warp out-of-range address</td>
<td>Occurs when a thread within a warp accesses an out-of-bounds local or shared memory address</td>
</tr>
<tr>
<td>Warp misaligned address</td>
<td>Occurs when a thread within a warp accesses an incorrectly aligned local or shared memory address</td>
</tr>
<tr>
<td>Device illegal address</td>
<td>Occurs when a thread accesses an out-of-bounds global memory address</td>
</tr>
</tbody>
</table>

Crash latency measures the time interval between the moment a fault is activated and the moment a crash occurs. We measure crash latency for each exception type above, to understand how quickly the crash is detected. Figure 4.10 shows the crash latency for each exception type for AES and MAT. In AES, 90% of the warp out-of-range address exceptions occur within around 500 milliseconds, compared to 70% of warp misaligned address exceptions and 60% of device illegal address. In MAT, warp out-of-range address exceptions occur faster compared to warp misaligned address exceptions. Only in the Stencil benchmark, does the device illegal address exception occurs and it occurs faster than the other three exception types. In all other benchmarks, the warp out-of-range address exceptions have lower crash latency than the other three exception types. Comparing the crash latency for CPU and GPU, Gu et al. [13] reported that on CPUs crashes usually happens in thousands of cycles after the fault injection, whereas on GPUs crashes happen in milliseconds. This could result from both the hardware check and the OS checking mechanisms. Systems that have longer crash latency may allow faults to propagate to
4.6. Use Cases

more states, and also have higher chance to affect states beyond the current context via shared memory, disk or network. Recovery in this scenario could become complicated because whole system-wise state rebuilding maybe required. Application-specific checkpointing/recovery techniques can be designed and configured based on different level of crash latency.

4.6 Use Cases

GPU-Qin can be used to evaluate error resilience characteristics of GPGPU applications for various purposes. In this section, we provide three scenarios to show how our tool can be used.

4.6.1 Scenario I: SDC Proneness of Different Code Sections

The key problem that selective fault tolerance mechanisms need to solve is to identify which parts of a program is more "important" than others for minimum overhead. In our context, selective mechanisms need to pinpoint the code sections of a program that have high probability to cause SDCs. These code sections, as discussed in the prior chapters, are not necessarily the same across different applications. GPU-Qin can be used to backtrack the fault injection results that lead to SDCs, and find out which source code statements/instructions are where the fault gets activated. For example, our preliminary observation is that some code patterns associated with CUDA programming model (e.g. computations involving thread IDs) have higher probability to lead to SDCs. A detailed use case can be found in our previous paper [10].
4.6.2 Scenario II: Comparing Different Algorithms

As reliability becomes more and more critical to computing systems, applications now need to choose algorithms by also taking into account error resilience. In this scenario, GPU-Qin can be used to evaluate error resilience of different algorithms solving the same problem. For example, sorting is a very popular operation in many application domains. Using GPU-Qin to perform characterization study for different sorting algorithms like quick sort, merge sort or heap sort etc, can suggest which algorithm to choose for necessary reliability requirement but also maintain acceptable performance and power consumption. Further, even for the same algorithm, GPU-Qin can also be used to tell which version/implementation of the algorithm to choose.

4.6.3 Scenario III: Guiding Configurations

HPC applications usually have complex system and application-wide configurations. GPU-Qin can provide the understanding of the error resilience characteristics to system users and help them better configure systems, e.g. setting up appropriate check-pointing intervals. It also allows users to test the error resilience of an application under different parameter combinations.

4.7 Limitations

Our evaluation study and analysis is subject to three limitations:

1. Experimental configurations such as the limit of the number of iterations of a loop and the size of the activation window may be specific to our benchmark suite. For example, based on our evaluation on the two benchmarks that contain the largest and smallest number of loop iterations, there is no difference in terms of SDC rate
for different upper bound values. We simply use 64 as the upper bound of number of iterations to explore. However, it can be adjusted for different sets of applications. Similarly, we use 1600 instructions as an activation window to limit the number of single-steps. For most of benchmarks this size is sufficient to have a high activation rate. These configurations are from empirical experience, thus they may also need to be justified for new sets of applications.

2. Another limitation is that CUDA applications need to be compiled in debug mode to use GPU-Qin. It is required because cuda-gdb cannot link source code to instructions when the debugging information is absent. Applications cannot get optimized by the compiler when the debugging information is present.

3. In terms of input selection, it is not possible for us to exercise all inputs for an application. We use inputs that are representative to the applications.

4.8 Summary

In summary, the result of error resilience characterization on GPGPU applications shows that measured SDC rates and crash rates vary significantly across benchmarks, and the average failure rate for all benchmarks is 67%. In contrast, difference in SDC rates for CPU applications is bounded by 5% to 15% [13]. Application-specific fault tolerance techniques hence are needed by GPGPU applications because generic techniques are likely to cause both unnecessary performance and power overhead without considering the implicit resilience of applications.
Figure 4.10: Crash latency analysis for AES and MAT. Top: AES Down: MAT
Table 4.3: Benchmarks properties. *LOC*: lines of code. *Scale*: number of blocks in a grid and number of threads in a block (generally a 3D*3D space). *Launch times*: the number of iterations that the kernel is launched.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Benchmark Suite</th>
<th>Kernel properties</th>
<th>Approximate LOC</th>
<th>Scale</th>
<th>Number of threads</th>
<th>Launch Times</th>
</tr>
</thead>
<tbody>
<tr>
<td>SAD</td>
<td>Parboil</td>
<td>mb_sad_calc</td>
<td>220</td>
<td>(44,36,1)*61,1,1</td>
<td>96624</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>larger_sad_calc_8</td>
<td>60</td>
<td>(44,36,1)*61,1,1</td>
<td>96624</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>larger_sad_calc_16</td>
<td>50</td>
<td>(11,3,1)*32,4,1</td>
<td>13464</td>
<td>1</td>
</tr>
<tr>
<td>Stencil</td>
<td>Parboil</td>
<td>block2D_hybrid_coarsen_x</td>
<td>100</td>
<td>(2,32,1)*32,4,1</td>
<td>8192</td>
<td>5</td>
</tr>
<tr>
<td>MRI-Q</td>
<td>Parboil</td>
<td>ComputeQ_GPU</td>
<td>50</td>
<td>(128,1,1)*256,1,1</td>
<td>32768</td>
<td>3</td>
</tr>
<tr>
<td>LBM</td>
<td>Parboil</td>
<td>performStreamCollide</td>
<td>150</td>
<td>(120,150,1)*120,1,1</td>
<td>2160000</td>
<td>100</td>
</tr>
<tr>
<td>MAT</td>
<td>CUDA SDK</td>
<td>matrixMul</td>
<td>110</td>
<td>(46,1)*32,32,1</td>
<td>98304</td>
<td>1</td>
</tr>
<tr>
<td>SCAN-block</td>
<td>CUDA SDK</td>
<td>scanExclusiveShared</td>
<td>70</td>
<td>(6656,1,1)*256,32,1</td>
<td>5452952</td>
<td>1</td>
</tr>
<tr>
<td>MONTE</td>
<td>CUDA SDK</td>
<td>MonteCarloOneBlockPerOption</td>
<td>40</td>
<td>(32,1,1)*256,1,1</td>
<td>8192</td>
<td>1</td>
</tr>
<tr>
<td>Transpose</td>
<td>CUDA SDK</td>
<td>transposeDiagonal</td>
<td>40</td>
<td>(64,64,1)*16,16,1</td>
<td>1048576</td>
<td>1</td>
</tr>
<tr>
<td>MergeSort</td>
<td>CUDA SDK</td>
<td>mergeSortSharedKernel</td>
<td>50</td>
<td>(4096,1,1)*512,1,1</td>
<td>2097152</td>
<td>1</td>
</tr>
<tr>
<td>BFS</td>
<td>Rodinia</td>
<td>Kernel</td>
<td>20</td>
<td>(8,1,1)*512,1,1</td>
<td>4096</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Kernel2</td>
<td>15</td>
<td>(8,1,1)*512,1,1</td>
<td>4096</td>
<td>8</td>
</tr>
<tr>
<td>AES</td>
<td>Other [21]</td>
<td>aesEncrypt256</td>
<td>400</td>
<td>(257,1,1)*256,1,1</td>
<td>65792</td>
<td>1</td>
</tr>
<tr>
<td>HashGPU</td>
<td>Other [18]</td>
<td>sha1_kernel_overlap</td>
<td>1000</td>
<td>(64,1,1)*64,1,1</td>
<td>4096</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>md5_kernel_overlap</td>
<td>1000</td>
<td>(64,1,1)*64,1,1</td>
<td>4096</td>
<td>1</td>
</tr>
</tbody>
</table>

a Randomly picking blocks to inject faults takes too long for LBM and Transpose because *cuda-gdb* launches the application block-by-block; thus, in practice, we only inject into the first 256 blocks of them
Chapter 5

Discussion

The fault-injection study presented in the previous chapter finds that the SDC rate varies widely across different benchmarks. For example, Monte Carlo has nearly no SDCs while HashGPU-sha1 and HashGPU-md5 have SDC rates of about 40%. In this chapter, we ask if there are fundamental reasons that some applications experience fewer SDCs than others. We focus on SDCs as these are considered the most severe failures: when an SDC occurs, there is no indication that something went wrong, yet an application produces incorrect output.

Intuitively, we attempted to find correlations between instruction-level characteristics of programs and the SDC rates, by trying to build a linear regression model based on the instruction classification to predict the SDC rates. We are not able to create such model because the sample size is small (for each application, there is only one observation) and with such small dataset, the correlation is low. This inspires us to explore the problem from a different perspective.

We believe that the reason for the variability in the SDC rate is related to the applications’ characteristics. For instance, applications based on search algorithms are likely to have lower SDCs than applications that perform computations such as linear algebra. This is because a fault affecting the search in a part of the space that will not lead to a match is unlikely to produce an incorrect result and the result will still be a mismatch. MergeSort
5.1 Search-based Application

Search is a subset of the class of computations in the dwarf "Branch and bound algorithms". The core computation pattern is that the search space is divided into segments and queries are searched in parallel in each segment. Depending on the actual search criteria, searching would be considered to return the solutions that are either accurate or optimal.

**Merge Sort** Merge sort consists of two major steps. The first step is to break the original input array into small blocks and sorts each block in parallel. The second step is the parallel merge. This step consists of three procedures (kernels) which would be executed iteratively together as merging small blocks into the final output array. The first kernel picks samples from the sorted blocks - it chooses multiple elements with which to split the blocks and computes the ranks of all sample elements in the even/odd pair of

---

6MergeSort-k0 divides the input into equal-sized tiles, and sort all tiles in parallel. Sorting in this kernel uses an implementation of Batcher's odd-even merge sort, which is based on comparison [4]
blocks. The second kernel takes the rank list of each block as the input and merges the rank lists, which essentially reorders the sub-blocks of the block. The third kernel performs the merge in parallel. Each element is assigned its own thread which performs a binary search over all elements of the other array to find the rank of its assigned element. During our characterization we only focus on the first kernel, but it is straightforward to include the rest of the kernels.

There are two factors affecting the end-to-end correctness of merge sort. The first factor is that binary search dominates the execution time of either locating, ranking and merging the internal blocks, or the actual merge of the final output. This explains the low SDC rates for all of the kernels in the sense that the decision of picking "left" or "right" elements is unlikely to change due to a fault as it is the result of a comparison operation, which is fairly error resilient. Secondly, the first kernel is only producing internal data that may not be critical to the final sorted output, while the last kernel directly changes the output. Therefore, a fault in the first kernel is not likely to have as much of an impact as a fault in the third kernel. These two factors explain the relatively low SDC rate of merge sort.

The following code segment taken from CUDA SDK shows the core computation of MergeSort-k0, which represents the behaviours that involve searching for correct positions to sort. The binary search is the major operation performed. We ignore the definition of binarySearchInclusive as it closely resembles the exclusive one.

```c
for(uint stride = 1; stride < arrayLength; stride <<= 1){
    // determine positions
    uint posA =
        binarySearchExclusive <sortDir>(keyA, baseKey + stride, stride, stride) + lPos;
    uint posB =
        binarySearchInclusive <sortDir>(keyB, baseKey + 0, stride, stride) + lPos;
__syncthreads();
```
5.2. Bit-wise Operation

SHA-1, MD5 and AES are popular industrial encryption standards for cryptography. These benchmarks (HashGPU-sha1, HashGPU-md5 and AES) have a similar behaviour in terms of data manipulation. The input data gets segmented into blocks and iterative operations are performed on each block to produce the final output. The SDC rates of the three benchmarks are also close, which are 28%, 31% and 28% respectively. Based on the correctness checks provided by these applications, even a single bit error would be considered as a failure, and hence one would expect these applications to have very high SDC rates.
However, they have some operations that mask faults, for example: (1) ANDing a flipped bit with a 0, or ORing it with 1, and (2) shifting a flipped bit out of a word. These operations provide a mitigating influence on the SDC rate.

### 5.3 Averaged Out

This category includes computations in which the final state is a converging product of multiple temporary states, either in space or time. The core pattern here is that the product of all states is likely to be obtained via operations that average those states.

**3D stencil**: This application represents a Jacobi solver of a heat equation on a 3D stencil. The blocks of threads are assigned to handle each element on XY plane and gathers 6 neighbours of that element to feed in the equation, which averages the result and replaces the original value with the new value. This process is repeated till all the values converge and there are no more changes. This makes the application capable of masking faults that affect a single element (the final values may be slightly different from the correct values, but the correctness checks of the application will accept them).

**Monte Carlo**: This application performs Monte Carlo simulation for American option pricing, in which multiple paths are explored at different time steps using random numbers. The final result is obtained by averaging the results of individual paths. Therefore, even if the fault affects a single path and causes it to compute the wrong value, it would make only a small difference in the result. In reality though, this application would explore thousands of paths, so the effect of a fault is likely to be even less pronounced, and hence the SDC rates are likely to be even lower.
5.4 Graph Processing

**BFS** This BFS implementation calculates the cost of each vertex from the source vertex on the level basis. The first kernel of BFS traverses vertices from the source vertex and assigns one thread to take one vertex in that level for the cost calculation. The cost of each vertex on the same level would be updated by this kernel. In a case of a vertex that has multiple parents, different threads could reach the vertex from different paths and end up with overwriting the cost for each other. The correctness is not affected since those threads are from the same level, so the cost would be overwritten by the same value. The second kernel deals with the race condition. It makes sure that every internal state of the vertex is synced before the launch of the first kernel. So if a fault affects the global states in the first kernel (not the updating mask array or the real cost), the second kernel would reset those states to what they are supposed to be.

5.5 Linear Algebra and Grid Operation

This category contains applications that involve regular data structures (e.g. matrix or grid). The computing task of these applications is usually distributed evenly across all computing units (threads in GPU) and the data access is also uniformly distributed. There are two major types of computation performed by applications in this category: (1) index calculation, and (2) value calculation. Index calculations involve calculating memory addresses of array elements, and value calculations involve the values that are written into the array. Of these two types of computations, errors in index calculations are likely to result in crashes as they can cause a pointer to point outside the bounds of the array, resulting in a hardware exception. Errors in value calculations are likely to result in SDCs. We find that the SDC rates of the applications in this category can be explained based on
5.5. Linear Algebra and Grid Operation

the relative proportion of index calculations and value calculations in them.

**Transpose and MAT** Matrix transpose and matrix multiplication are both very common operations in linear algebra. On GPUs, the matrices are tiled so that multiple threads can operate on them in parallel, with synchronization code inserted if needed. For matrix transpose most of the computations are index calculations, while for matrix multiplication, most of the computations are value calculations.

**MRI-Q** MRI-Q computes the Q matrix which in MRI image reconstruction is a pre-computable value based on the sampling trajectory, the plan of how points in k-space will be sampled. The algorithm examines a large set of input representing the intended MRI scanning trajectory and the points that will be sampled. Each element of the Q matrix is computed by a summation of contributions from all trajectory sample points. MRI-Q simply collects corresponding elements from 3-D data set and computes a number of element in Q. As a result, MRI-Q has only value computations, and not index computations.

**SCAN** SCAN is a simple implementation of the prefix-sum for GPUs. It assigns threads to fetch elements adding them up with previous elements for logN rounds and double-buffers the shared memory for storing the temporary sum of the array. It also consists of a combination of index and value calculations.

**SAD** SAD stands for "Sum of Absolute Differences", and consists of two major steps. The first step is to compute SAD for 4*4 blocks from reference pixel and the frame pixels. There are two levels of index calculations and a SAD computation involved in the first step. The SAD computation for each 4*4 block pair is a simple 2D stencil like operation. The second step is to merge the basic 4*4 pixel blocks into larger blocks. It takes the output vector of the first step as the input and performs additions on the vector to form results for different types of blocks such as 4*8, 8*4, 16*8, etc. The output of the second step is the SADs for all 7 block sizes. This step is also comprised of a large amount of index
5.6 Summary

In general, the linear algebra and grid operation applications are low error-resilient, as every single element and operation involved in the computation are likely to evenly contribute to the final output. Within this category, however, we observe that a wide range of SDC rates for benchmarks. Our hypothesis is that index calculation and value calculation make a difference in SDC rate as index calculations would likely to lead to crashes. For example, the benchmarks that contain index calculations, namely Transpose, SCAN and SAD, have above 40% crash rate, as opposed to those that do have fewer address calculations in intuition, e.g. MAT and MRI-Q. Interestingly, applications that combine index and value calculations have higher SDC rates than those that have only one predominant type of calculation. We fill validate our hypothesis in the future work.

These observations suggest that it might be useful to cluster the benchmarks based on both the SDC rate and the high-level operations they perform. We categorize the benchmarks into five resilience categories, shown in Table 5.1. Asanovic et al. [2] defines ”thirteen dwarfs of parallelism” to design and evaluate the parallel computing applications. Each of these dwarfs captures a pattern of computation common to a class of parallel applications. We find that the resilience categories we consider map well to one or more of the dwarfs, as Table 5.1 shows. We did not start out trying to find such a mapping, and
Table 5.1: Benchmark categories and the mapping to the dwarfs of parallelism

<table>
<thead>
<tr>
<th>Resilience Category</th>
<th>Benchmarks</th>
<th>Measured SDC</th>
<th>Dwarfs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Search-based</td>
<td>MergeSort</td>
<td>6%</td>
<td>Backtrack and Branch+Bound</td>
</tr>
<tr>
<td>Bit-wise Operation</td>
<td>HashGPU, AES</td>
<td>25 ~ 37%</td>
<td>Combinational Logic</td>
</tr>
<tr>
<td>Average-out Effect</td>
<td>Stencil, MONTE</td>
<td>1% ~ 5%</td>
<td>Structured Grids, Monte Carlo</td>
</tr>
<tr>
<td>Graph Processing</td>
<td>BFS</td>
<td>10%</td>
<td>Graph Traversal</td>
</tr>
<tr>
<td>Linear Algebra</td>
<td>Transpose, MAT, MRI-Q, SCAN-block, LBM, SAD</td>
<td>15% ~ 25%</td>
<td>Dense Linear Algebra, Sparse Linear Algebra, Structured Grids</td>
</tr>
</tbody>
</table>

hence may not cover all dwarfs in our application categories. We will explore this mapping systematically in future work.
Chapter 6

Related Work

This chapter provides an overview of related work in the areas of software-based error resilience techniques and GPU vulnerability studies, and how our work differs.

Fault injector Fault injection has been well-explored on CPUs using run-time debuggers. Examples are GOOFI [1] and NFTAPE [28]. However, neither of these injectors work on GPUs. Further, they do not consider multi-threaded programs, nor do they concern themselves with choosing representative parts of the program for injection. Other work [20] attempted to inject faults in scientific applications using the PIN tool from Intel, a dynamic binary instrumentation framework. However, this work has not been applied on GPUs to the best of our knowledge.

AVF and PVF A common way to estimate vulnerability is through the architectural vulnerability factor (AVF) [22], which analyzes the vulnerability of specific micro-architectural structures to soft errors. The main idea is to track the execution of a program through the processor, typically by executing it in a simulator, and identifying certain bits as ACE bits (Architecturally Correct Execution) bits. The total number of ACE bits in a microarchitectural structure is an estimate of its vulnerability. AVF-based methods have two disadvantages over fault injection based methods. First, they rarely consider the end-to-end behaviour of the application under faults, and hence end up approximating the set of bits that need to be marked ACE. For example, Wang et al. [33] found that AVF estimates are significantly less accurate than fault injection based results. Although the
original formulation of architectural vulnerability factor considers the outcome of the pro-
gram in deciding if a specific bit is an ACE-bit, practical implementations of the technique
count every bit that can potentially affect a program (within a certain window) as an ACE
bit. This conservative estimation can grossly overestimate the observed vulnerability com-
pared to a fault-injection [33]. Second, as AVF studies consider specific microarchitectural
structures, microprocessor simulators are usually required to execute experiments, which
means that they are significantly slower than executing the program on native hardware.

Several studies [23, 30] have attempted to characterize the vulnerability of different
micro-architectural structures in GPUs. For example, Tan et al. [30] characterized GPU
instructions (CUDA PTX) based on whether the execution of an instruction affects the
final output of the application, and hence determines the AVF by the quantity of ACE
instructions per cycle and their residency time within the hardware structures. Program
Vulnerability Factor (PVF) is a metric proposed by Sridharan et al. [27] to apply AVF
analysis at the application layer. While this takes application properties into account, it
does not consider the end-to-end impact of faults on the application. These approaches
do not consider the end-to-end impact of faults in applications, nor do they attempt to
understand the behavior of the application under errors. Moreover, AVF analysis has been
shown to have significant inaccuracies compared to fault-injection based approaches [33].
In contrast, our work is from the applications’ perspective, and focuses on understanding
the behaviors of GPGPU applications under errors.

Generic fault tolerance techniques Dimitrov et al. [8] proposed three approaches
for GPGPU reliability that leverage both instruction-level parallelism and thread-level
parallelism to replicate the application code. Their approach incurs performance overheads
of 85 to 100%, and they conclude that understanding both the application characteristics
and the hardware platform is necessary for efficient protection. They do not characterize
the reliability of GPGPU applications however.

**Application specific fault tolerance** Some studies have attempted to establish correlations between SDCs and program characteristics. Thaker et al. [31] observes that errors in control-data are more likely to lead to SDCs and catastrophic failures in multimedia applications. Thomas et a. [32] find that errors in data affecting a large amount of computation are likely to lead to egregious outcomes (what they call EDCs). Shoestring [11] finds that errors in high value instructions that write to global memory or produce function call arguments in the program are likely to lead to undetected SDCs. These observations and findings are incomprehensive in the sense that they are only applied to some categories of applications and they can not explain the SDC rates exclusively.

Hari et al. [14] presents a low-cost, program-level fault detection mechanism for reducing SDCs in CPU applications. They use their prior work, Relyzer [15] to profile applications and select a small portion of the program fault site to identify static instructions that are responsible for SDCs. Then by placing program level error detectors on those SDC-causing sections, they can achieve high SDC coverage at low cost. It is noteworthy that application-specific behaviours are major contributors of SDCs for half of their benchmarks, which makes it difficult to extend their technique to other applications, especially GPU applications which have different behaviours from CPU applications.

Finally, Yim et al. [35] proposed a technique to detect errors through data duplication at the programming-language level (loop code and non-loop code) for GPGPU applications. This is different from our focus which is to understand the inherent error-resilience characteristics of an application in order to find the most efficient protection. They perform fault injections at the source code level, while we do so at the executable code level. Because many hardware faults cannot be modelled at the source code level, our injections are more representative of hardware faults.
Chapter 7

Conclusion and Future Work

This thesis presents a methodology to investigate the end-to-end error resilience characteristics of GPGPU applications through fault injection. One of the main challenges in building a fault injector for GPGPU applications is balancing representativeness with time efficiency, due to their massive parallelism. We first build a fault-injection tool, GPU-Qin, to efficiently inject faults on real GPU hardware, while maintaining representativeness of the faults injected. Using GPU-Qin, we study the error resilience characteristics of twelve GPGPU applications comprised of fifteen kernels. The investigation showed that 0.3% to 38% of the faults result in SDCs and 6% to 71% of the results in crashes, which suggests that application-specific fault tolerance mechanisms are needed to deal with such variety of levels of error resilience. Our fault injector enables the opportunity to study various reliability characteristics of applications, such as instruction-level error resilience and crash latency. It also exposes the impact of faults in different bit-positions on the error resilience of different applications. All of these can be used to guide the design of application-specific fault tolerance techniques. Finally, we find that algorithmic characteristics of the application can help us understand the variation in the SDC rates among different applications.

In the future, we plan to proceed in the following directions:

1. To understand the variance of SDC rates within the same resilience category. For example, in the category of linear algebra and grid computations, SDC rates vary from 15% to 25%, and 25% to 40% in the category of bit-wise operations.
2. To perform error resilience characterization on CPU applications, and see if the similar categorization can be observer.

3. Compare CPU and GPU applications in terms of error resilience, and come up with specific fault tolerance mechanisms for each platform.
Bibliography


[6] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha


