A Systematic Methodology for Evaluating the Error Resilience of GPGPU Applications

Bo Fang*, Karthik Pattabiraman*, Matei Ripeanu*, Sudhanva Gurumurthi †
*Department of Electrical and Computer Engineering, University of British Columbia
†Cloud Innovation Lab, IBM Corporation
{bof, karthikp, matei}@ece.ubc.ca; sgurumu@us.ibm.com

***Abstract*—The wide adoption of graphics processing units (GPUs) as accelerators for general-purpose applications makes the end-to-end reliability implications of their use increasingly significant. Fault injection is a widely adopted method to evaluate the resilience of applications. However, building a fault injector for general-purpose GPU applications is challenging due to their massive parallelism, which makes it difficult to achieve representativeness while being time-efficient.**

This paper makes four key contributions. First, it presents a fault-injection methodology to evaluate the end-to-end reliability properties of application kernels running on GPUs. Second, it introduces GPU-Qin, a fault-injection tool that uses real GPU hardware and offers a tunable and efficient balance between the representativeness and the cost of a fault-injection campaign. Third, it characterizes the error resilience characteristics of seventeen application kernels. Finally, it provides preliminary insights on correlations between the algorithmic properties of applications and their error resilience.

***Index Terms*—GPU, fault tolerance, error resilience**

I. Introduction

GPUs were originally designed for applications where reliability was not a primary concern (e.g., image rendering, where a few wrong pixels are not noticeable by human eyes). Today, however, GPUs are widely used to accelerate general purpose applications in areas where correctness is critical (e.g., medicine, oil and gas exploration, scientific computing). Understanding the behavior of these applications in the presence of hardware faults thus becomes important, especially as the rate of such faults increase with technology scaling [1]. Particle-induced transient faults are especially a key source of hardware faults [2].

Manufacturers have invested significant effort to improve GPU reliability. For instance, most modern GPUs such as NVIDIA Fermi^{TM}and AMD Radeon 7970^{TM}provide error correcting code (ECC) support that covers the memory space, the register file, the shared memory and caches [3], [4]. However, ECC entails area, power, and performance overheads. The computational and control datapaths are also susceptible to transient faults and more challenging to protect efficiently using ECC. Therefore, designing a reliable GPU and providing resilience for applications that run on the device is a challenging problem.

The long-term goal of our work is to develop application-specific, software-based fault-tolerance mechanisms for general purpose GPU applications. As a first step towards this goal, we have developed a fault injection methodology and the associated tool-chain to investigate the error-resilience characteristics of these applications. Fault-injection is the act of perturbing an application to emulate faults, and then studying their effect on the application outcome [5]. While there has been substantial work in the realm of fault injection for CPU applications [6], [7], 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 [8] 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 the hardware faults that occur at the granularity of microarchitectural units and instructions. To model hardware faults, the standard approaches are to inject faults into a register transfer language (RTL) model or a microarchitectural simulator [9]. While these approaches facilitate studying the impact of various types of hardware faults, as they are based on simulations they are considerably slower than executing applications on real hardware. One way to alleviate the performance bottleneck is to execute only a small section of the application. However, this limits the ability to obtain insights into the end-to-end behavior of the application under faults.

To avoid the above issues, we choose to perform fault injections at the assembly-language level using a GPU-based debugger. While not as detailed as fault injections at the microarchitectural level, this approach allows us to model the impact of faults that lead to errors in the architectural state of instructions, and thus is more precise than injecting at the high-level language level. Compared to the microarchitectural level injectors based on simulation, this approach is much more efficient and scalable as the application is executed on the actual GPU hardware. This approach serves well our final goal: we aim to understand applications' error resilience (rather than compute raw hardware FIT rates hardware) and explore software techniques to improve resilience.

This paper makes the following contributions:

1) *Proposes a methodology* to evaluate the resilience of GPU applications (Section III) and describes and quantitatively characterizes (Section IV-B) the design decisions and the corresponding trade-offs between fault injection campaign coverage and the cost required to handle the massive parallelism of GPU applications.

2) *Describes the design and implementation of a fault-injection tool*, GPU-Qin, that is able to inject faults into applications running on the actual GPU hardware (Section III).
3) *Performs an end-to-end error-resilience characterization* of 14 GPU applications (17 kernels) (Section IV-C). We find that there are significant variations in the error resilience characteristics of GPU applications. For example, the silent data corruption (SDC) rates range from 1% to 38% while the crash rates range from 6% to 69%. This further highlights the potential of application-specialized error resilience mechanisms implemented at the software level.
4) *Provides an additional set of scenarios where our methodology and tool can be used.* We characterize the causes of application crashes and crash latencies (Section IV-E). We further attempt to correlate application characteristics with their observed error resilience (Section IV-D), and discuss the use of GPU-Qin in the context of guiding code optimizations, algorithm choice or application-level configuration to maximize error resilience (Section V-B).

II. Background and Fault Model

This section offers background information on the dependability metrics associated with this work, the fault model used, and the NVIDIA GPU architecture and programming model. We chose to prototype our fault injector on an NVIDIA GPU because NVIDIA's CUDA^{TM}tool chain providers debugging capabilities that facilitate software fault injection.

A. Dependability Metrics: Error Resilience and Vulnerability

Error resilience is the *conditional* probability of a system/program not experiencing a failure given that a hardware fault has occurred. Program failures can be classified as crashes (i.e., hardware/operating system exceptions), hangs, and SDCs (i.e., incorrect output). Faults that do not cause failures are known as benign.

We focus on hardware faults that propagate to software because we are interested in building error resilient applications. Different hardware platforms usually have different fault-tolerance mechanisms. At the same time, different applications correspond to different instruction streams, which determine fault propagation. Thus, error resilience is a property of both the platform and the application. Since our evaluation is performed on one hardware platform, in our context error resilience 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) of a program. Note that vulnerability is different from error resilience as it contains two conditions: (1) the probability of the system experiencing a fault, and (2) the probability that this fault leads to a failure.

B. Characterizing Error Resilience

There are three commonly used methods to characterize error resilience:

Accelerated Testing: This method refers to the use of particle sources, such as an alpha source or neutron beams, to impinge on a device under test to induce faults [10]. The main advantage of this method is that it can be used to create faults that can mimic those that would occur under normal operating conditions at an accelerated rate. However, accelerated testing can be expensive, often limited by availability of beam-time and cost, and it is challenging to control how and on what components the faults are induced.

Fault Injection: This is a procedure to introduce faults in a systematic, controlled manner and study the resulting system's behavior. Fault injection techniques can be generally categorized into hardware-based and software-based [11], [12]. Hardware-based fault injection techniques normally require specialized hardware equipment in addition to the target systems, and they cannot be directly used to target applications or operating systems. 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. The appealing properties of these techniques are: relatively low-cost as they require no special equipment, provide higher controllability, and less constrained in terms of the number of experiments that can be done.

As mentioned before, fault injection can be performed at the RTL or microarchitectural levels. However, injecting faults at this level requires detailed RTL or microarchitectural simulators which makes end-to-end application-level resilience evaluation impractical. For this reason, we perform fault injection at a higher level, namely at the level of assembly instructions, and execute on real hardware. Our goal is to obtain sufficient coverage in terms of number of instructions executed, rather than in terms of the proportion of hardware state covered by the injections, as is typical in RTL/microarchitectural fault injections.

AVF/PVF Analysis: Mukherjee et al. [13] introduce the architecture vulnerability factor (AVF) analysis that quantifies the vulnerability of microarchitectural components to soft errors. Sridharan et al. [14] study the vulnerability of software independent of hardware by introducing the program vulnerability factor (PVF) analysis. Although AVF and PVF measure properties that are different from error resilience, they can be used for a first estimate of the error resilience of an application. The key limitation, however, is that AVF and PVF treat all failure outcomes equally, that is, they do not differentiate between crashes, SDCs and benign faults. We elaborate on AVF and PVF in Section VI.

C. 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 (e.g. stuck-at-faults) [15]. Further, transient faults are caused by external events such as cosmic rays, while permanent hardware faults are usually caused by manufacturing defects or wearout. Transient faults are one of the major sources of faults in processors [16] and are the focus of our evaluations. In this paper, we consider faults in architecturally visible registers.

Our methodology is agnostic to whether a fault arises in the register file or is propagated to the registers from elsewhere. We do not consider faults in cache, memory because we assume that they are protected by ECC. This is the case for recent server GPUs from major vendors. However, our fault injector can model memory faults and other fault models too with minimal modifications.

We use the single-bit-flip model as it is the most common SRAM transient fault mode in today's systems [2]. However, our fault injector can support both single- and multiple-bit flips by choosing corresponding fault generation functions with minimal modifications.

D. GPU Architecture and Programming Model

We focus on GPGPU applications written in the NVIDIA Compute Unified Device Architecture(CUDA), a widely adopted programming model and toolchain for GPUs. The CUDA programming model defines a GPU application as a control program that runs on the host and a computation *kernel* that runs on the GPU device. CUDA kernels use a single instruction/multiple thread (SIMT) model that exploits the massive parallelism of the GPU device.

From a software perspective, CUDA abstracts the SIMT model into a hierarchy of kernels, blocks and threads. A CUDA kernel consists of blocks, and a block, in turn, consists of threads. From a hardware perspective, blocks of threads run on hardware units known as streaming multiprocessors (SMs) that feature a shared memory space for the threads inside the same block. Inside a block, threads are scheduled in a fixed group of 32 threads, known as warps. All the threads in a warp execute the same instructions, but on different data values. The kernel is implemented as a collection of functions in a language similar to C, with annotations for identifying GPU code and for delineating different types of memory spaces on the GPU.

III. METHODOLOGY

This section 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, which consists of a profiler and a 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 over the set of all instructions executed by the application. We assume that each dynamic instruction carries the same probability of fault occurrence. While this is a simplistic model, if a more realistic model is available (i.e., a model that reflects the differentiated instruction-level likelihood to experience a fault) our methodology will need only minor modifications to incorporate such a model. In particular, the change for GPU-Qin will be to assign different cycle counts (i.e. probability to experience a fault) to different instructions. However, the result of the above change to the applications' error resilience is not clear.
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 a large number (typically thousands) of faults-injection experiments need to be performed to obtain statistically significant estimates of error resilience and hence each individual run should be fast.
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.

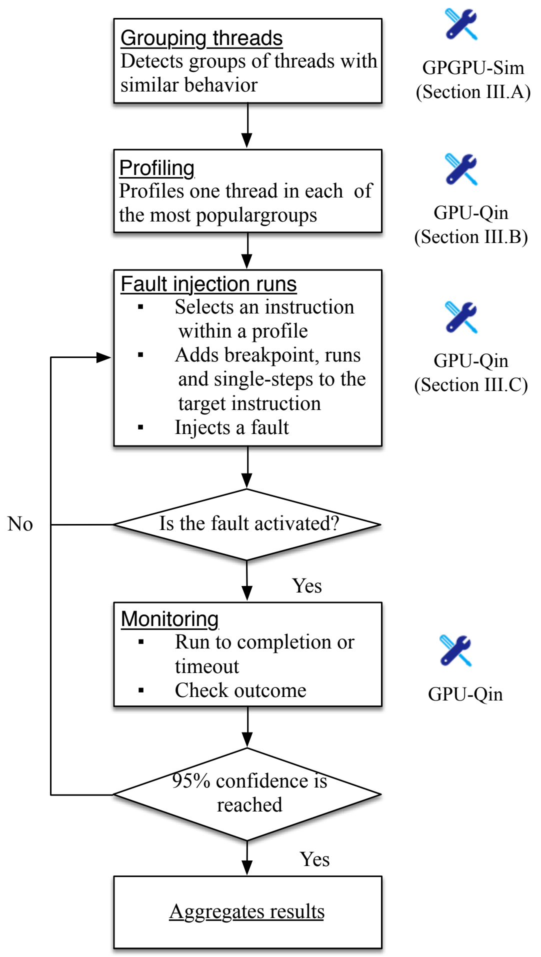

Grouping threads
Detects groups of threads with similar behavior
GPGPU-Sim (Section III.A)
Profiling
Profiles one thread in each of the most populargroups
GPU-Qin (Section III.B)
Fault injection runs
Selects an instruction within a profile
Adds breakpoint, runs and single-steps to the target instruction
Injects a fault
GPU-Qin (Section III.C)
No
Is the fault activated?
Yes
Monitoring
Run to completion or timeout
Check outcome
GPU-Qin
95% confidence is reached
Yes
Aggregates results

Fig. 1: Overview of GPU-Qin, our fault-injection methodology highlighting the main phases: grouping, profiling, fault injection, and result aggregation and the corresponding sections that describe them.

Figure 1 shows an overview of our methodology. The process consists of three main phases, which we briefly introduce here and detail in the rest of this section. In the first phase (described in Section III-A), we use the profiling information obtained by using a cycle accurate GPU simulator to group application threads based on the similarity of their behaviour. The goal is to reduce the number of threads we characterize: as GPU applications usually launch thousands or tens of thousands of threads, it would be extremely time-consuming to evaluate the error resilience of each GPU thread. Instead, we identify the representative groups of threads, and 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 Section III-A.

In the second phase (described in Section III-B), we profile the threads selected in the first phase and obtain for each of them an execution trace. This information is necessary in the next phase to locate, at runtime, the instruction at which to pause execution and inject the fault.

In the third phase (described in Section III-C), for each injection run, we randomly choose 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 thread group it represents. We also randomly pick a thread from the entire set of application run-time threads for each injection run. The choice of the instruction is done uniformly over the space of the instructions of the profile; thus, our methodology simulates the occurrence of a transient error that occurs with uniform probability over all thread lifetime. This satisfies the representativeness requirement.

We implemented our methodology based on the CUDA GPU debugging tool: *cuda-gdb* [1]. 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. In addition to cuda-gdb, we employ GPGPU-Sim [9], a cycle-accurate GPU simulator for the initial profiling of the application. This preliminary step does not distort the characteristics of a program, compared to profiling on the real GPU. *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. We automate the interactions between *cuda-gdb* and GPU-Qin.

A. Phase I: Grouping

GPU applications often have a massive number of threads (often tens of thousands) and it would be infeasible to obtain the execution traces for all threads in an application kernel for the purpose of fault injection. Therefore, a key challenge is to identify a fraction of threads that are representative of the workload behavior. To this end, we seek to separate threads into groups of threads with similar behaviour and select one thread from each group to obtain a trace. To identify groups we use dynamic instruction counts as a proxy for thread behaviour. The intuition is that, as GPUs are built on Single Instruction Multiple Threads (SIMT) [17] computation model, similar threads execute exactly the same dynamic instructions. Our grouping strategy offers a conservative representation of the thread behaviours because the threads from two groups can share the same set of most-executed instructions resulting from for example, difference in number of loop iterations executed, hence the error resilience characteristics could remain similar between threads in two groups.

[1]https://developer.nvidia.com/cuda-gdb

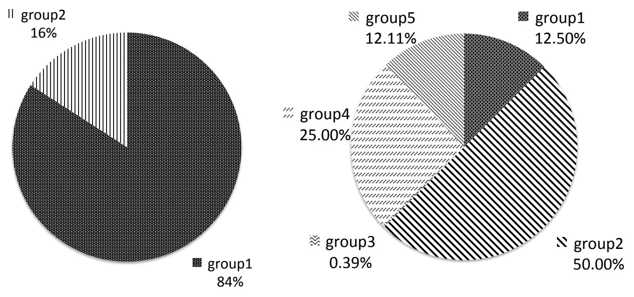


Fig. 2: Number of groups and their size in selected application kernels. Each group contains threads that have similar instruction counts. The group size is expressed as the percentage of the total number of threads of the kernel that belongs to the group. *Left*: LBM *Right*: Monte Carlo

Because the GPUs used do not have a built-in, per-thread, instruction counter, we have instrumented the state-of-the-art GPU simulator GPGPU-Sim (v3.2.0) [9] to obtain the number of dynamic instructions executed per thread. GPGPU-Sim simulates the execution of GPGPU programs from both functional and performance perspectives, and hence the number of instructions executed matches the number of instructions executed on 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 note that the instruction count includes conditionally executed instructions regardless of the condition being true or not. We have verified that, in practice, this approximation does not impact our grouping accuracy.

Based on the results of the group identification process, we find that our benchmarks (presented in detail in Section IV and Table II) can be categorized in three categories (Table I). In the first category, all threads execute the same number of instructions, and hence there is only one group. In the second category, there is a limited amount of divergence (i.e., the threads execute different instructions) 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.

In Section IV-B3 we validate the grouping methodology through an extensive fault injection experiment.

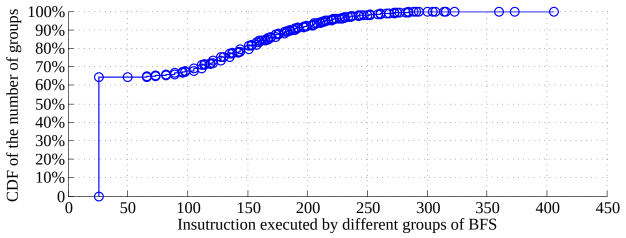


Fig. 3: Cumulative distribution function(CDF) of number of threads covered by groups in BFS. One group contains over 60% of threads, and the rest are equally popular.

TABLE I: The group identification process leads to classifying the benchmarks in three categories.

Category	Benchmark	Groups	Groups to profile	Thread coverage
I	AES, MRI-Q, MAT, MergeSort-k0, Transpose	1	1	100%
II	SCAN, Stencil, Monte Carlo, SAD, LBM, HashGPU	2 - 10	1 - 4	95% - 100%
III	BFS, PageRank, SSSP	>10	2	>60%

B. Phase II: Profiling

The goal of the profiling phase is to map the Source and Assembly (SASS) instructions (the raw CUDA instructions) executed by a thread (identified during the previous phase) to their corresponding CUDA source-code line. This will enable GPU-Qin, which uses conditional breakpoints, to inject faults (phase III). The reason is that *cuda-gdb*, on which GPU-Qin is built, requires the source-code line number for setting a conditional breakpoint and a single source-code 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 (pc) value of the instruction is recorded, along with the dynamic 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.

As thread profiling a is time-consuming, to balance coverage and efficiency, we propose the following method: for applications in which there is only one group, we profile a single thread, randomly chosen. For applications with a small number of groups, we select the groups that include the majority of the threads and randomly pick one thread from each selected group to profile. Figure 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 pick both groups. However, in some cases, we ignore the less popular groups: Monte Carlo benchmark has five groups, but one of the groups is responsible only for 0.4% of total number of threads, and hence we ignore this group for profiling and statistically addressing the impact of this choice as shown in Section III - Confidence and confidence interval. For applications that have a large number of groups (i.e. Category III), we again use group popularity to make informed choices. For BFS, around 60% of the threads fall in a popular group (shown as a vertical line in Figure 3), while all the other 78 groups are equally popular; therefore, we profile random threads from the large group and other groups. For page Rank and single source shortest path, more than 90% of threads are running for processing the same amount of vertices with the virtual warp-centric programming model [18] resulting in a similar profiling strategy of BFS. At the cost of additional time, more groups can be sampled to increase coverage.

C. Phase III: Fault Injection

The third phase of the process is to inject faults into the application at runtime and monitor the outcomes. Figure 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 the trace. In each injection campaign, to ensure representativeness, the thread to inject is chosen randomly from the set of all threads used by the program, rather than only from the ones chosen during the grouping phase. GPU-Qin uses the profile from the profiling phase and uniformly selects a SASS instruction. To inject a fault, it sets up a conditional breakpoint in the program at the source code line that corresponds to that target SASS instruction using *cuda-gdb*. The conditional breakpoint is triggered 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). The application runs natively on the hardware until the breakpoint is triggered and after the fault is injected (i.e., 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 desired confidence is obtained. The rest of this section presents the details of this process.

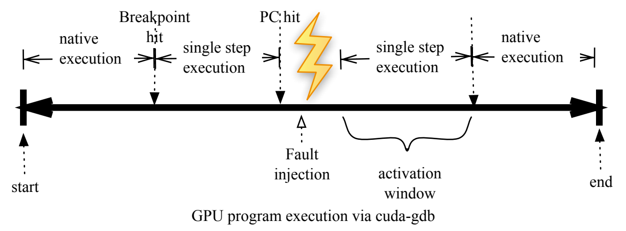


Fig. 4: Phase III - the fault-injection process

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

1) If an instruction that is the target of the fault-injection occurs in a loop, GPU-Qin estimates the loop iteration in which that dynamic instruction gets executed. 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 level the time of 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 Section IV-B.

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 over 99% source lines correspond to at most a few tens of instructions. This window's size can be configured by the user.

The location to inject: The location to inject the fault depends on the instruction executed. GPU-Qin considers three types of instructions to inject:

1) *Arithmetic instruction*: GPU-Qin injects faults into the destination register of the instruction. For vector instructions that have multiple destination registers, GPU-Qin randomly chooses a destination register to inject into.
2) *Memory instructions*: GPU-Qin injects faults into either the destination register or the address register in load/store instructions.
3) *Control-flow instruction*: NVIDIA instruction set architecture (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 register chosen as above (Section II-C discusses the fault model). 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: In rare cases a fault might not be injected in a run even when the desired 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 (recall that 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 address involved in BRA (which is a branch instruction to jump a relative address) and hence choosing this instruction leads to the run being discarded. Overall, less 1% of total fault injection runs are discarded.

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 the 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 after injection, GPU-Qin lets the program continue and consider the fault inactivated. We set the window to be 1600 instructions for our experiments. We explore the implications of this choice in Section IV-B.

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*) [2], or (4) Completes with correct output (*benign*). These four outcomes are mutually exclusive and exhaustive.

Confidence and confidence intervals: Our fault injection process can be essentially modeled with binomial distribution, as each individual fault injection trial produces either a failure (i.e., SDC, hang, or crash) or non-failure outcome. The grouping mechanism produces separate groups of threads that may have different error resilience characteristics, hence it is necessary to combine their characteristics to obtain application resilience characteristics.

We propose a uniform approach to interpret the characteristics of error resilience for the benchmarks that contain multiple groups. This approach includes two components:

(i) Estimating the failure rate for a set of groups based on sampling, i.e., fault injection experiments, conducted in each group: We use the stratified sampling technique [19]: each group of threads is considered as a stratum, while whole thread population is the full strata. Equation 1 and 2 compute the failure rate estimate and its variance. The 95% confidence interval of the failure by approximating the binomial distribution as a normal distribution (Equation 3).

(ii) Estimating the overall failure rate and confidence interval when incorporating small groups: Since the ignored group size is relatively much smaller, we can obtain conservative upper/lower bounds for the 95% confidence interval of the error rate estimate by assuming all fault injection trials (if conducted) for this small group produces a failure (either SDC or crash) and, respectively, a non failure outcome. (Equation 4 and 5).

$$\bar{p}_{strata} = \sum_{i=1}^{k} \left(\frac{N_i \bar{p}_i}{\sum_{j=1}^{k} N_j} \right) \tag{1}$$

$$Var\left(\bar{p}_{strata}\right) = \sum_{i=1}^{k} \left(\frac{N_i}{\sum_{j=1}^{k} N_j} \right)^2 \frac{p_i(1-p_i)}{n_i} \frac{N-n_i}{N-1} \tag{2}$$

$$\bar{p}_{upper/lower_bound} = \bar{p}_{strata} \pm 1.96 * \sqrt{Var\left(\bar{p}_{strata}\right)} \tag{3}$$

$$\bar{p}'_{upper\ bound} = \frac{\bar{p}_{upper\ bound} * \sum_{i=1}^{k} N_i + 1}{N} \tag{4}$$

[2]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.

$$\bar{p}'_{lower\ bound} = \frac{\bar{p}_{lower\ bound} * \sum_{i=1}^{k} N_i}{N} \tag{5}$$

Where:

p_i	= Failure (SDC, crash) rate estimate for group i
k	= Number of popular groups
N_i	= (number of threads in group i)*(number of dynamic instructions executed by a thread in group i), that is, the population size of stratum i
N	= the sum of all N_i
$\bar{p}_{strata}$	= The failure rate estimate for the set of groups
n_i	= Number of samples (i.e., fault injection trials) conducted for group i (much smaller than N_i)

IV. Characterization Study

This section uses a wide variety of applications (presented in Section IV-A) to validate GPU-Qin design choices (Section IV-B), to demonstrate the use of our methodology to characterize applications' error resilience (Section IV-C), and to explore the causes of crashes and characterize crash latency (Section IV-E). All of our experiments are conducted on NVIDIA Tesla C2075.

A. Benchmarks

Our benchmarks are drawn from the Parboil benchmark suite [20], NVIDIA CUDA SDK package, Rodinia benchmark suite [21], Totem graph processing engine [22] and applications that contain well-known GPU kernels (e.g. AES, LBM). A short description of each benchmark is given in the appendix, along with the inputs used in our evaluation. Table II summarizes the characteristics of each benchmark.

B. Validating Design Choices

This section offers empirical support for the heuristics used in Section IV. These heuristics (e.g., the grouping strategy, the size of the activation window) represent choices in the trade off space between coverage (in terms of distinct code paths profiled then used for fault injection) and efficiency (run-time characterize an application).

1) Validation of Design Choices - Limiting the Iteration Count: As mentioned in Section III, one of the design decisions we make to bound the cost of the fault injection campaigns is to limit the number of loop iterations explored. That is, if the instruction to be injected belongs to a loop iteration that exceeds a specified threshold T, we generate a random number between 0 and T and inject a fault at the corresponding loop iteration. In our experiments we use T=64. This design choice is based on the observation that GPU applications usually consist of repetitive program states, and the faults occurring in different iterations of a GPU program are likely to result in similar program states.

To validate the above heuristic, we count the total number of iterations executed by each loop of each kernel, and then consider the loop with the largest number of iterations (shown in Figure 5). We disregard applications that execute fewer than 64 iterations (in all loops) because they fall within the chosen threshold already. For the two applications that have loops that exceed the threshold (i.e., 128 in MAT and 512 in MRI-Q), we vary the threshold from 32 to 128 for MAT and 32, 64, 128 and no threshold for MRI-Q, and repeat the characterization experiments.

Figure 6 presents the SDC and crash rates with these thresholds. 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 limit for the number of iterations we use in our study is sufficient of our benchmark suite, it is still possible that applications with different characteristics in terms of loop iterations need different application-specific thresholds.

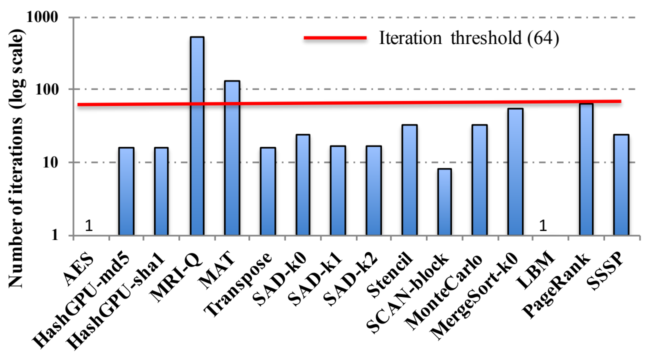


Fig. 5: The maximum number of loop iterations executed by each kernel.

2) Validation of Design Choices: Activation Window Size: As mentioned in Section III another heuristic used to improve efficiency is the following: If the injected fault is not activated within an activation window of 1,600 dynamic instructions from its injection, we consider it inactivated and drop the run.

To validate the second heuristic, 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) there are fault-injection runs in which the activation window is exceeded: 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.

3) Validation of Design Choices: Grouping: The purpose of grouping is to identify the representative threads and thus reduce the cost of profiling. As we describe in Section III-A, we use the number of dynamic instructions executed by

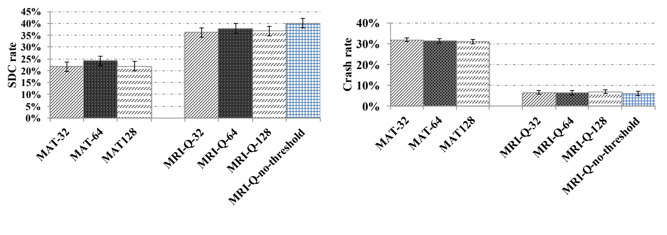


Fig. 6: SDC and crash rate estimates for different iteration thresholds. *Left:* SDC. *Right:* Crash. Varying the threshold of loop iterations does not affect the resulting SDC and crash rate estimate.

TABLE II: 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 times the kernel is launched.

Benchmark	Benchmark Suite	Kernel properties				
		Name	*Approximate LOC*	*Scale*	*Number of threads*	*Launch Times*
AES	Other [23]	aesEncrypt256	400	(257,1,1)*(256,1,1)	65,792	1
HashGPU	Other [24]	sha1_kernel_overlap	1000	(64,1,1)*(64,1,1)	4,096	1
		md5_kernel_overlap	1000	(64,1,1)*(64,1,1)	4,096	1
MRI-Q	Parboil	ComputeQ_GPU	50	(128,1,1)*(256,1,1)	32,768	3
MAT	CUDA SDK	matrixMul	110	(4,6,1)*(32,32,1)	98,304	1
Transpose[a]	CUDA SDK	transposeDiagonal	40	(64,64,1)*(16,16,1)	1,048,576	1
SAD	Parboil	mb_sad_calc	220	(44,36,1)*(61,1,1)	96,624	1
		larger_sad_calc_8	60	(44,36,1)*(61,1,1)	96,624	1
		larger_sad_calc_16	50	(11,9,1)*(32,4,1)	13,464	1
Stencil	Parboil	block2D_hybrid_coarsen_x	100	(2,32,1)*(32,4,1)	8,192	5
SCAN-block	CUDA SDK	scanExclusiveShared	70	(6656,1,1)*(256,32,1)	54,525,952	1
MONTE	CUDA SDK	MonteCarloOneBlockPerOption	40	(32,1,1)*(256,1,1)	8,192	1
MergeSort	CUDA SDK	mergeSortSharedKernel	50	(4096,1,1)*(512,1,1)	2,097,152	1
BFS	Rodinia	Kernel	20	(8,1,1)*(512,1,1)	4,096	8
PageRank	Totem [22]	page_rank_incoming_kernel	40	(9468,1,1)*(512,1,1)	4,847,616	5
SSSP	Totem [22]	sssp_vwarp_kernel	40	(9,1,1)*(512,1,1)	3,708	9
LBM[a]	Parboil	performStreamCollide	150	(120,150,1)*(120,1,1)	2,160,000	100

[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

a thread as a proxy for thread behavior, and we group a program's threads based on this metric.

This heuristic can be validated through an instruction level classification analysis. First we profile randomly chosen threads of an application and compare the grouping decision based on number of dynamic instruction with a detailed instruction-level classification analysis. While we have executed the above comparison successfully for all applications we show here only two examples applications from categories I and II, namely MAT and SAD below.

MAT: Since in MAT all the threads execute the same number of dynamic instructions (Category I), our validation aims to verify if the same number of instructions implies an identical set of instructions executed by each thread. We randomly choose 100 threads in MAT and record using GPU-Qin the instructions each thread executes. We observed that all the resulting execution traces are identical.

SAD: As we show in Table I, SAD-k0 has five 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.

To highlight the fact that our grouping strategy is conservative, i.e., threads from different groups may have similar resilience characteristics, we pick a random thread from each of the two most popular groups separately to classify the instructions. Figure 7 shows the classification of instructions executed by the two threads we pick. In total, we found 10 categories of instructions used in SAD-k0 defined in NVIDIA SASS ISA [25], and we present the break-down of the number of executed instructions in each category. The difference in terms of number of instructions between the two threads is 786, which constitutes about 2% of total number of instructions in a thread. After comparing the dynamic instructions, 95% instructions are identical. Fault injection results on those groups show approximately 1% difference in both SDC and crash rates, which matches the small variance in different groups. Thus a more aggressive grouping strategy could further reduce the cost of the fault injection campaign at the cost of a small accuracy loss.

Finally, we have validated that grouping is useful: that is, that threads in groups that differ widely in number/type of instructions do have different error resilience properties. To this end, we compare the fault-injection results of applications in categories II and III (see Table I). The crash rates vary considerably for different groups of threads in Stencil, LBM, SCAN and BFS, which is 5%, 10%, 10% and 25% respectively.

The above experiments suggest that grouping is a reasonable representation of the program's behaviour under errors, further providing evidence that using the number of dynamic instructions as the representation of a thread's behaviour is a reasonable but conservative metric.

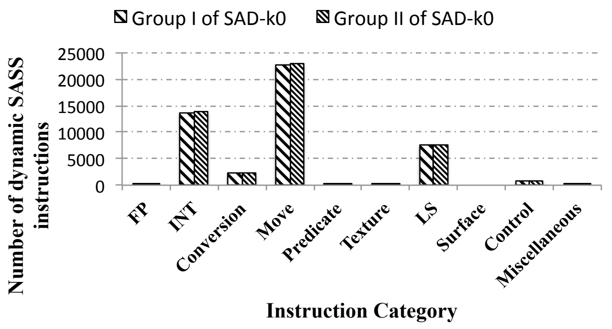


Fig. 7: The instruction classification of two random threads from different groups. Small difference in instruction classification matches the difference in SDC and crash rates.

C. Characterization of Error Resilience

We characterize the error resilience of 17 kernels from the 14 applications mentioned earlier. Table III presents, for each kernel, 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 activated, overwritten, or ignored due to it exceeding the activation window.

In Section III-C Confidence and confidence interval, we have described our an approach to estimate the combined SDC and crash rates for applications that contain multiple groups

TABLE III: Fault-injection experiments information

Kernels	Injected runs	Activated runs	Activation rate	Average runtime (s)
AES	2,351	2,042	87%	84
HashGPU-md5	2,699	2,683	99%	13
HashGPU-sha1	2,400	2,305	96%	27
MRI-Q	2,830	2,475	87%	123
MAT	2,575	2,186	85%	82
Transpose	2,395	2,160	90%	44
SAD-k0	2,671	2,435	91%	76
SAD-k1	2,208	2,195	99%	26
SAD-k2	2,627	2,618	100%	12
Stencil	2,426	2,148	89%	31
SCAN-block	1,083	1,080	99%	710
MonteCarlo	3,744	2,723	73%	66
MergeSort-k0	1,930	1,884	98%	359
BFS	2,334	2,330	100%	22
PageRank	2,039	1,740	85.37%	68
SSSP	1,845	1572	85%	14
LBM	1,895	1,845	97%	165

TABLE IV: Experimental details for FI in different groups of MONTE

Group	% of threads	# of dynamic instructions	# of activated FT trials	SDC rate	Crash rate
1	50%	2006	1,368	0.6%	23.8%
2	25%	2013	1,122	0.44%	21.8%
3	12.5%	2020	970	0.46%	21.5%
4	12.11%	2067	854	0.52%	22.6%
5	0.39%	2076	NA	NA	NA

and the small ones are not sampled. We provide an example to clarify this process. Recall that MONTE has five groups of threads (Section III-A) that include 50%, 25%, 12.5%, 12.11% and 0.39% of all threads respectively. We conduct fault injection experiments on the threads from first four groups, and Table IV shows the detailed profiling and outcome information of the experiment.

To estimate the overall SDC rate of MONTE and compute its 95% confidence interval, we take the approach described in Section III-C and calculate the upper and lower bound of the 95% confidence interval (1.1%, 0.3%).

Figure 8 presents the SDC and crash rate for all kernels. We do not show the hang rates as they are uniformly lower than 1%. Fault injections in CPUs exhibit similar hang rates [26]. A first observation is that both the SDC rate and the crash rate vary widely across benchmarks. In particular, the SDC rate ranges from 0.7% to nearly 38%. This observation suggests that it is important to take into account the inherent error resilience characteristics of an application when designing mechanisms to protect it from SDC-causing errors. For example, the SDC rate for MONTE is less than 1%, likely because the results of each simulation path in the Monte-Carlo simulation will eventually be aggregated, which mitigates the effect of the incorrect output of a simulation path if a fault occurs on one of them. We note that similar applications in terms of application behaviour, 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%. In total, across all benchmarks, failure rates (crash+SDC+hang) range from 24% (MONTE) to 91% (SCAN), and the mean failure rate is 65% across the applications. We explore the possible reasons behind these variations in the next section.

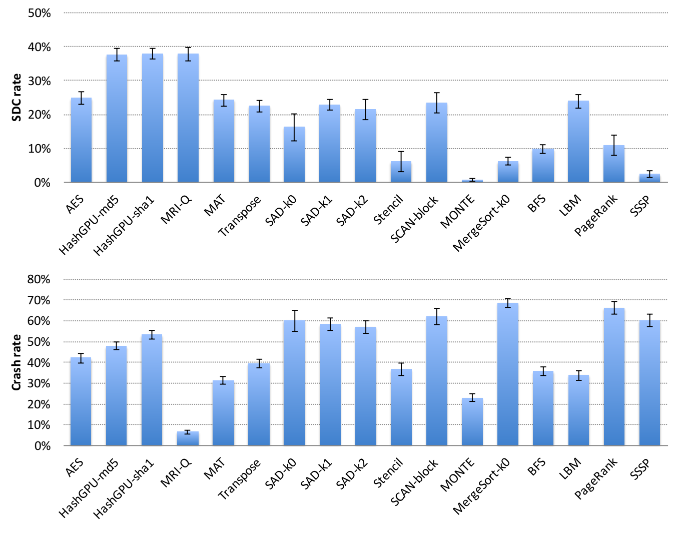


Fig. 8: SDC (top) and crash (bottom) rates with error bars representing 95% confidence interval for each kernel.

D. Exploring the Correlation between Error Resilience and Application Characteristics

The characterization study presented in the previous section shows that the SDC rate varies widely across kernels. For example, Monte Carlo has nearly no SDCs while HashGPU-sha1 and HashGPU-md5 have SDC rates of about 40%. In this section, we explore whether there are instruction-level (Section IV-D1) or algorithm-level (Section IV-D2) characteristics that correlate well with the observed SDC rates.

1) A Negative Result: SDC rates are only weakly correlated with application's instruction-mix characteristics: Prior studies have shown that errors have different effects on different types of instructions. For example, Thaker et. al. [27] have found that on CPU applications ”computations involving control are more sensitive to inaccuracy than data and address operations”. The reason is that errors involved in address calculations are likely to cause crashes: accessing invalid addresses will lead to paging or segmentation faults [26].

Since the SDC rate reflects the impact of transient faults on program's internal state, we hypothesized that it is possible to correlate the SDC rate of an application and its instruction-level characteristics.

We used GPU-Qin to characterize the 17 GPU kernels used previously in terms of instruction-level characteristics: we captured dynamic instructions of each application and counted the instructions of each type (including ALU, FP, memory and control flow) to obtain their relative frequency.

We then attempted to find correlations: we delineated instruction frequency as explanatory variables and the SDC rate as the dependent variable, and applied multiple types of regression analysis. However, even after extensive attempts, we were unable to find a satisfactory model that successfully predicted the estimated SDC rate based on instruction-level characteristics. Note that we did not confine ourselves to linear combinations. For example, we tried non-linear combinations such as log or square root exhaustively on each variable. The best model we were able to obtain is a linear model, which had an R^2 around 0.5 and the p-value about 0.04, which indicates a weak correlation between the SDC rate and the variables we used.

While, of course, our negative experience does not constitute a proof of the absence of such a model, it indicates that explanatory models that use only instruction-level features may not be sufficient for SDC rate prediction. A key factor, we believe, is that instruction classification does not capture the semantics of the sequence of the instructions, which is a determining factor in the error resilience of applications. This inspired us to explore, in the next section, the impact of algorithmic application characteristics.

2) Algorithmic-Level Properties: Does algorithmic structure correlate with observed SDC rates?: Asanovic et al. [28] describe the ”thirteen dwarfs of parallelism” to design and evaluate templates for parallel computing applications. Each of these dwarfs captures a pattern of computation common to a class of parallel applications. We grouped our benchmarks according to their structure as defined by the ”dwarfs” (Table V). As showed in Table V, applications in different dwarfs exhibit diverse error resilience characteristics. We can attribute this to micro-level code patterns as we describe below. We highlight the two operations that are correlated with fault resilience.

1) *Averaging-Out:* This operation 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 averaging operations over these intermediary states. If a fault happens in one of the temporary states, it is likely that it would be averaged out in the final state given application-specific precision requirements. Examples of this class of applications are stencil computations and Monte-Carlo simulations.
2) *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, search would be considered to return the solutions that are either accurate or optimal. Search-based applications usually contain frequent comparison operations, and those operations are much more error resilient than other operations. One example from our benchmarks is MergeSort, which implements parallel sorting based on binary search [29], and we observe that it has a relatively low SDC rate (6%).

E. Characterizing Crashes: Causes and Latency

GPU-Qin can be used to gain a deeper understanding of the error-resilience characteristics of GPGPU applications. For example, it can be used 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.

TABLE VI: Description of CUDA hardware exceptions

Exception type	Description
Lane user stack overflow	Occurs when a thread exceeds its stack memory limit
Warp out-of-range address	Occurs when a thread within a warp accesses an out-of-bounds local or shared memory address
Warp misaligned address	Occurs when a thread within a warp accesses an incorrectly aligned local or shared memory address
Device illegal address	Occurs when a thread accesses an out-of-bounds global memory address

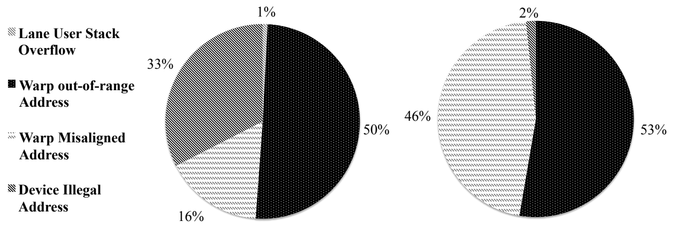


Fig. 9: Root-cause breakdown of crashes for AES (*left:*) and MAT (*Right:*). Out-of-bounds memory addressing is the leading cause of crashes.

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* (Table VI).

We report results for only two benchmarks, AES and MAT, however, the observations generalize to all benchmarks. Figure 9 shows the relative frequency of various root causes for crashes in these two applications. The two most common causes are *warp out-of-range addresses* and *device illegal address*. *Warp misaligned address* also plays an important role in crashes for the MAT benchmark.

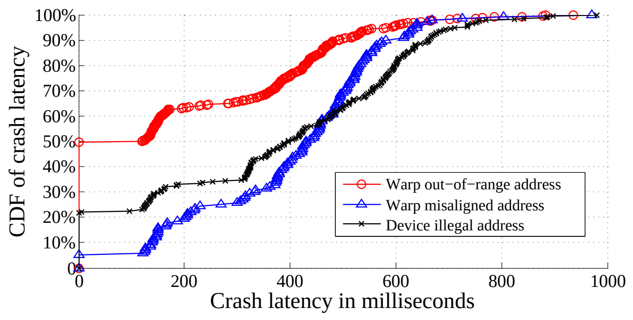
Crash latency measures the time interval between the moment a fault is activated and the moment a crash occurs. Our measurements do not include the time for single-stepping to the target instruction. We measure crash latency for each exception type above, to understand how quickly the crash is detected. Figure 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 occur, 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.

We can compare crash latency for CPUs and GPUs: A fault injection study by Gu et al. [26] found on CPUs that crashes usually happen within thousands of cycles after the fault injection, which translates to a few micro-seconds at most on their architecture. In contrast, on GPUs, crashes happen in milliseconds. This could result from both the more aggressive CPU hardware and the OS checking mechanisms and from

TABLE V: Benchmark representative operations and the mapping to the dwarfs of parallelism

Dwarf	Kernel	Measured SDC rate	Key Operation as explanation
Linear Algebra	Transpose, MAT, LBM, SCAN-block, LBM, SAD-k0, SAD-k1, SAD-k2	$15 \sim 25\%$	Not Avaliable
Structured Grids	Stencil	6%	Average-out Effect
Graph Traversal	BFS, PageRank, SSSP	$2.5 \sim 11\%$	Overwrite
Monte Carlo	MONTE	1%	Average-out Effect
Combinational Logic	HashGPU-md5, HashGPU-sha1, AES	$25 \sim 38\%$	Bit-wise Operation
Backtrack and Branch+Bound	MergeSort-k0	6%	Search-based



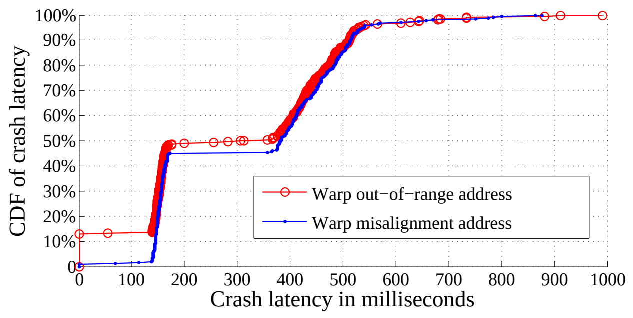


Fig. 10: Crash latency analysis for AES and MAT. *Top:* AES *Down:* MAT. Warp out-of-range address exceptions occur faster than other exceptions.

GPUs execution model that round-robins over all execution warps. Systems that have longer crash latency may allow faults to propagate to more states, and faults have higher chance to affect states beyond the current context via shared memory, disk or network. Recovery in this scenario is more complicated because the whole system-wide state rebuilding maybe required.

V. Discussion

This section presents supporting data that justifies for our choice to base GPU-Qin methodology on executions on real hardware (as opposed to microarchitectural simulations, Section V-A), presents a number of additional usage scenarios where our methodology can be used (Section V-B), and discusses its limitations (Section V-C).

A. Running Time for Fault Injection

The average runtime of each fault-injection run varies across kernels from 11 seconds to 710 seconds, and is directly proportional to the block size (i.e., number of threads used by the kernel, shown in Table II). We observe that our worst-case kernel, SCAN, which takes 710 seconds on average, is 10x faster with GPU-Qin than running with GPGPU-Sim. Other benchmarks show speedups as high as 100x. 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 compared to architectural level simulation based fault injection techniques.

B. QPU-Qin Usage Scenarios

GPU-Qin can be used to evaluate error resilience characteristics of general purpose GPU applications in a number of contexts. We provide three usage scenarios to show how our tool can be used.

1) Scenario I: Evaluating SDC Vulnerability of Different Code Sections: The key problem that selective fault tolerance mechanisms need to solve is to identify which parts of a program are more "important" than others, and protect these parts such that an increased fault tolerance can be achieved with minimal overhead [30]–[32]. In our context, selective mechanisms need to pinpoint the code sections of a program that have high probability to cause SDCs. These code sections are different across different applications. GPU-Qin can be used to analyze the fault injection results that lead to SDCs, and identify the source code statements/instructions 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 prior work on this topic [33].

2) Scenario II: Comparing Error Resilience of Different Algorithms: As reliability becomes increasingly critical to computing systems, applications now need to choose algorithms by also taking into account their error resilience characteristics. In this scenario, GPU-Qin can be used to evaluate error resilience of different algorithms solving the same problem. For example, sorting is a popular operation in many application domains. Using GPU-Qin for characterizing the resilience of different sorting algorithms like quicksort can suggest which algorithm to choose to target a desired reliability level and also maintain acceptable performance and power consumption. Further, even for the same algorithm, GPU-Qin can also be used to decide which version/implementation of the algorithm to choose.

3) Scenario III: Guiding Configurations: High performance computing (HPC) applications usually have complex system-wide configurations. GPU-Qin can be used to build on the understanding of the error resilience characteristics of the system users and help them better configure its reliability options, e.g. setting up appropriate checkpointing intervals, turning on ECC. It can also allow users to test the error resilience of an application under different configurations.

C. Limitations

Our evaluation study is subject to four main limitations:

1) Experimentally derived configuration parameters such as the limit on 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, we found no difference in terms of SDC rate for different upper bound values. These configurations are based on empirical experience; thus they may also need to be adjusted for new sets of applications.

2) As CUDA GPU-Qin relies on *cuda-gdb* applications need to be compiled in debug mode, some compiler optimizations will be disabled. This choice, however is in line with prior studies such as NFTAPE, FERRARI and Xception that implement debugger-based fault injection infrastructures focusing on the dependability of CPU-based systems. Past work by Sangchoolie et al. [34] has found that turning on/off standard optimization flags for gcc (i.e. O1, O2 and O3) had only marginal impact on the SDC rates. This study was conducted for CPU applications, however. The impact of disabling compiler optimizations on error resilience characteristics for GPU applications remains unclear, and we will investigate this in our future work.

3) *cuda-gdb* allows access only to a subset of the GPU state for injecting faults (i.e., the state available for register file or memory, but not for structures like warp scheduler, cache, etc). Studying the impact of faults in other structures requires either access to lower-level tools or hardware fault injection support. In addition, *cuda-gdb* does not allow directly altering the opcode of an instruction. Adding this feature would require significant effort to extend the current GPU-Qin.

To provide an understanding of how much state can be injected into by GPU-Qin, we compute the ratio of the number of bits in the register files that can be injected (i.e. data bits) to the total number bits including instruction registers, flags (i.e. instruction bits) and register files (i.e. data bits) in appendix B. In particular, in the NVIDIA Fermi architecture, each instruction register contains 64 bits and each register used in SASS instructions contain 32 bits. As GPUs use the Single Instruction Multiple Thread (SIMT) model, a single instruction may be shared by all threads in a warp, and hence this ratio is also a function of the level of warp divergence in a benchmark. The average value of the ratio across our benchmarks is 94.23%, with the highest 96.30% for MAT, and lowest 85.40% for SCAN. This shows that the GPU-Qin is able to cover most of the GPU program states in our benchmarks.

4) Many application attributes whether they are performance- or dependability-related (e.g., run-time, crash latency, vulnerability, etc) will vary with the input (size and characteristics). We do not explore the possible variation in application characteristics when varying the input size. In our experiments, for benchmarks from the Parboil suite, we use small/medium input size. For other benchmarks, we use the default input inputs that are representative of the applications' real-world usage, based on our own judgement. The goal of this paper is to present a methodology and a practical tool to explore different resilience aspects of a diverse set of applications. We are not aiming for an definitive characterization of the error resilience GPGPU applications.

VI. Related Work

This section provides an overview of related work in the areas of software-based error resilience techniques and GPU vulnerability studies, and positions our work vis-a-vis these related projects.

Fault injection. Fault injection has been well-explored on CPUs using run-time debuggers (e.g., GOOFI [7] and NFTAPE [6]) or dynamic binary instrumentation (PIN framework [35]). However, neither of these injectors work on GPUs. More importantly, they do not consider multi-threaded programs, nor do they concern themselves with choosing representative parts of the program for injection. In very recent work, Hari et al. [36] propose a compile-time fault injection tool for GPGPU applications called SASSIFI. SASSIFI is similar to GPU-Qin in many respects. In addition, it also allows faults to be injected into predicate and condition code registers. The authors of SASSIFI do not consider techniques to reduce the fault injection space like GPU-QIn does, and leave it to the user to determine the fault injection sites.

AVF and PVF. A common way to estimate vulnerability is through the architectural vulnerability factor (AVF) [37], which analyzes the vulnerability of specific microarchitectural structures to soft errors. The main idea is to track the execution of a program through the processor, typically by executing it in a microarchitectural simulator, and identifying the ACE bits (Architecturally Correct Execution bits): that is the bits that would cause faulty execution if flipped. The total number of ACE bits in a microarchitectural structure is an estimate of its vulnerability. Several studies [13], [38] have attempted to characterize the vulnerability of different microarchitectural structures in GPUs. For example, Tan et al. [13] characterized GPU instructions (CUDA PTX) based on whether the execution of an instruction impacts 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.

The Program Vulnerability Factor (PVF) is a metric proposed by Sridharan et al. [14] to separate the architectural and microarchitectural components of the AVF and isolate the impact of the program on vulnerability. While this metric takes application properties into account, it still does not consider the end-to-end impact of faults on the application (i.e., it does not separate crashes, hangs, SDCs and benign faults).

Typically, AVF/PVF 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 [39]. In contrast, our work is from the applications' perspective, and focuses on understanding the behaviors of GPU applications under errors through fault injection.

Generic fault tolerance techniques. Dimitrov et al. [40] proposed three approaches for GPU application 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 GPU applications, nor do they develop application-specific mechanisms. Wadden et al. [41] designed three compiler-implemented redundant multi-threading algorithms to protect GPU applications from hardware faults. Their methodology allows automatic transformation for GPU kernels with the respect to performance and power overheads. Their evaluation shows that the performance of redundant multi-threading depends on the behavior of kernels and the required sphere of reliability.

Shoestring [30] aims to reduce SDCs by selectively protecting program instructions that potentially lead to SDCs. It identifies high value instructions that write to global memory or produce function call arguments in the program and apply vulnerability analysis heuristics to the program instruction level for selective duplication. At a high-level, Shoestring attempts to correlated program level characteristics with SDC-proneness in CPU applications. Lu et al. [31] proposed SDCtune,a empirical model to predict the SDC proneness of a programs data. However, we have not observed similar trends for GPU applications.

Application specific fault tolerance. Some studies have attempted to establish correlations between SDCs and program characteristics. For example, Thaker et al. [27] observes that errors in control-data are more likely to lead to SDCs and catastrophic failures in multimedia applications. Thomas et al. [42] find that errors in data affecting a large amount of computation are likely to lead to egregious outcomes (what they call EDCs) for soft computations.

These observations and findings are only applied to some categories of applications and they can not be used to explain a particular SDC rate exclusively.

Hari et al. [43] present a low-cost, program-level fault detection mechanism for reducing SDCs in CPU applications. They use their own prior work, Relyzer [44] 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 compared to CPU applications.

Finally, Yim et al. [8] proposed a technique to detect errors through data duplication at the programming-language level (loop code and non-loop code) for GPU 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 easily modelled at the source code level, our injections are more representative of hardware faults.

VII. Conclusion

This paper presents a systematic methodology and a tool-set 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. Then, we provide empirical support for design choices we made thoroughly for validation. Using GPU-Qin, we study the error resilience characteristics of twelve GPGPU applications comprised of fifteen kernels. The investigation showed that 1% to 38% of the faults result in SDCs and 6% to 70% 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, which is useful for guiding the design of application-specific fault tolerance techniques. We also find that algorithmic characteristics of the application (such as dwarfs) can help us understand the variation in the SDC rates among applications.

Acknowledgment

The authors would like to thank Wilson Fung, Lauro Beltrao Costa, Elizeu Santos-Neto, Jiesheng Wei, Anna Thomas and Abdullah Gharaibeh for their suggestions on the different phases of this project. This work was funded in part by an NSERC Discovery grant, an equipment donation from NVIDIA, and a research gift from AMD corporation. We also thank the anonymous reviewers of ISPASS 2014 for their feedback to improve the paper. GPU-Qin is publicly available at **https://github.com/DependableSystemsLab/GPU-Injector**.

References

[1] C. Constantinescu, "Trends and challenges in vlsi circuit reliability," in *IEEE MICRO*, 2003.

[2] V. Sridharan, N. DeBardeleben, a. K. F. S. Blanchard, J. Stearley, J. Shalf, and S. Gurumurthi, "Memory errors in modern systems: The good, the bad, and the ugly," in *Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems*, 2015.

[3] NVIDIA. (2009) Nvidia fermi whitepaper. [Online]. Available: http://www.nvidia.ca/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf

[4] M. Mantor. (2012) Amd radeon hd 7970 with graphcis core next (gcn) architecture. [Online]. Available: http://www.hotchips.org/wp-content/uploads/hc_archives/hc24/HC24-3-ManyCore/HC24.28.315-AMD.GCN.mantor_v1.pdf

[5] M.-C. Hsueh, T. Tsai, and R. Iyer, "Fault injection techniques and tools," *Computer*, vol. 30, no. 4, pp. 75–82, 1997.

[6] D. Stott, B. Floering, D. Burke, Z. Kalbarczpk, and R. Iyer, "Nftape: a framework for assessing dependability in distributed systems with lightweight fault injectors," in *IPDPS 2000*, 2000, pp. 91 –100.

[7] J. Aidemark, J. Vinter, P. Folkesson, and J. Karlsson, "Goofi: generic object-oriented fault injection tool," in *Dependable Systems and Networks, 2001 International Conference on*, 2001, pp. 83–88.

[8] K. S. Y. et al., "Hauberk: Lightweight silent data corruption error detector for gpgpu," in *IEEE International Parallel and Distributed Processing Symposium*, 2011.

[9] A. Bakhoda, G. Yuan, W. Fung, H. Wong, and T. Aamodt, "Analyzing cuda workloads using a detailed gpu simulator," in *Performance Analysis of Systems and Software. ISPASS 2009.*

[10] T. Gaitonde, S.-J. Wen, R. Wong, and M. Warriner, "Component failure analysis using neutron beam test," in *Physical and Failure Analysis of Integrated Circuits, 2010 17th IEEE International Symposium on the.*

[11] M.-C. Hsueh, T. Tsai, and R. Iyer, “Fault injection techniques and tools,” vol. 30, no. 4, apr 1997, pp. 75 –82.
[12] J. Arlat, Y. Crouzet, and J.-C. Laprie, “Fault injection for dependability validation of fault-tolerant computing systems,” in *Fault-Tolerant Computing, 1989. FTCS-19. Digest of Papers., Nineteenth International Symposium on*, June 1989, pp. 348–355.
[13] J. Tan, N. Goswami, T. Li, and X. Fu, “Analyzing soft-error vulnerability on gpgpu microarchitecture,” in *IEEE International Symposium on Workload Characterization (IISWC)*, 2011, pp. 226–235.
[14] V. Sridharan and D. Kaeli, “Eliminating microarchitectural dependency from architectural vulnerability,” in *High Performance Computer Architecture, 2009. HPCA IEEE 15th International Symposium on*.
[15] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr, “Basic concepts and taxonomy of dependable and secure computing,” *Dependable and Secure Computing, IEEE Transactions on*.
[16] S. Borkar, “Designing reliable systems from unreliable components: The challenges of transistor variability and degradation,” *IEEE Micro*, vol. 25, no. 6, pp. 10–16, Nov. 2005.
[17] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, “Nvidia tesla: A unified graphics and computing architecture,” *Micro, IEEE*, vol. 28, no. 2, pp. 39–55, March 2008.
[18] S. Hong, S. K. Kim, T. Oguntebi, and K. Olukotun, “Accelerating cuda graph algorithms at maximum warp,” in *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*.
[19] W. G. Cochran, *Sampling techniques*. John Wiley & Sons, 2007.
[20] J. A. S. et al., “Parboil: A revised benchmark suite for scientic and commercial throughput computing,” in *IMPACT Technical Report*, 2012.
[21] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, “Rodinia: A benchmark suite for heterogeneous computing,” in *IISWC 2019*.
[22] A. Gharaibeh, L. B. Costa, E. Santos-Neto, and M. Ripeanu, “On graphs, gpus, and blind dating: A workload to processor matchmaking quest,” in *IPDPS 2013*.
[23] S. A. Manavski, “Cuda compatible gpu as an efcient hardware accelerator for aes cryptography,” in *IEEE Intl Conf. on Signal Processing and Communication*, 2007, pp. 65–68.
[24] S. A. Kiswany, A. Gharaibeh, E. S. Neto, G. Yuan, and M. Ripeanu, “StoreGPU: exploiting graphics processing units to accelerate distributed storage systems,” in *HPDC ’08: Proceedings of the 17th international symposium on High performance distributed computing*.
[25] NVIDIA. (2014) Cuda binary utitilies. [Online]. Available: http://docs.nvidia.com/cuda/cuda-binary-utilities/index.html
[26] W. Gu, Z. Kalbarczyk, and R. Iyer, “Error sensitivity of the linux kernel executing on powerpc g4 and pentium 4 processors,” in *Dependable Systems and Networks, 2004 International Conference on*, 2004.
[27] D. Thaker, D. Franklin, J. Oliver, S. Biswas, D. Lockhart, T. Metodi, and F. Chong, “Characterization of error-tolerant applications when protecting control data,” in *Proc. IISWC*, 2006, pp. 142–149.
[28] K. e. a. Asanovic, “The landscape of parallel computing research: A view from berkeley,” EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2006-183, Dec 2006.
[29] N. Satish, M. Harris, and M. Garland, “Designing efficient sorting algorithms for manycore gpus,” NVIDIA Corporation, NVIDIA Technical Report NVR-2008-001, Sep. 2008.
[30] S. Feng, S. Gupta, A. Ansari, and S. Mahlke, “Shoestring: probabilistic soft error reliability on the cheap,” in *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, 2010.
[31] Q. Lu, K. Pattabiraman, M. S. Gupta, and J. A. Rivers, “Sdctune: A model for predicting the sdc proneness of an application for configurable protection,” in *CASES 2014*.
[32] A. Thomas and K. Pattabiraman, “Error detector placement for soft computation,” in *Dependable Systems and Networks (DSN), 2013 43rd Annual IEEE/IFIP International Conference on*, June 2013, pp. 1–12.
[33] B. Fang, J. Wei, K. Pattabiraman, and M. Ripeanu, “Towards building error resilient gpgpu applications,” in *3rd Workshop on Resilient Architecture (WRA) in conjunction with MICRO*, 2012.
[34] B. Sangchoolie, F. Ayatolahi, R. Johansson, and J. Karlsson, “A study of the impact of bit-flip errors on programs compiled with different optimization levels,” in *Dependable Computing Conference 2014 (EDCC),*.
[35] D. Li, J. Vetter, and W. Yu, “Classifying soft error vulnerabilities in extreme-scale scientific applications using a binary instrumentation tool,” in *SC 12*.
[36] S. K. S. Hari, T. Tsai, M. Stephenson, S. W. Keckler, and J. Emer, “Sassifi: Evaluating resilience of gpu applications,” in *Proceeding of the 11th Workshop on Silicon Errors in Logic - System Effects*.
[37] S. Mukherjee, C. Weaver, J. Emer, S. Reinhardt, and T. Austin, “Measuring architectural vulnerability factors,” in *IEEE MICRO*, 2003.
[38] R. U. N. Farazman and and D. Kaeli, “Statistical fault injection-based avf analysis of a gpu architecure,” in *IEEE Workshop on Silicon Errors in Logic*.
[39] N. J. Wang, A. Mahesri, and S. J. Patel, “Examining ace analysis reliability estimates using fault-injection,” in *Proceedings of the 34th annual international symposium on Computer architecture*.
[40] M. Dimitrov, M. Mantor, and H. Zhou, “Understanding software approaches for gpgpu reliability,” in *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, 2009.
[41] J. Wadden, A. Lyashevsky, S. Gurumurthi, V. Sridharan, and K. Skadron, “Real-world design and evaluation of compiler-managed gpu redundant multithreading,” in *Proceeding of the 41st Annual International Symposium on Computer Architecture*, ser. ISCA ’14.
[42] A. Thomas and K. Pattabiraman, “Error detector placement for soft computation,” in *Dependable Systems and Networks (DSN), 2013 43rd Annual IEEE/IFIP International Conference on*.
[43] S. K. S. Hari, S. V. Adve, and H. Naeimi, “Low-cost program-level detectors for reducing silent data corruptions,” in *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2012.
[44] S. K. S. Hari, S. V. Adve, H. Naeimi, and P. Ramachandran, “Relyzer: exploiting application-level fault equivalence to analyze application resiliency to transient faults,” in *ACM ASPLOS*, 2012.
[45] M. Harris, S. Sengupta, and J. D. Owens, “Parallel prefix sum (scan) with CUDA,” in *GPU Gems 3*, H. Nguyen, Ed. Addison Wesley, August 2007, ch. 39, pp. 851–876.

Bo Fang Bo Fang received his bachelor’s from the Wuhan university in China in year 2006 and Master of Applied Science from the University of British Columbia in 2014. He joined the Ph.D program in the electrical and computer engineering department at UBC since 2014. Bo’s research interests include error resilience characterization and fault tolerance techniques from the perspective of applications.

Karthik Pattabirman Karthik Pattabiraman received his M.S and PhD. degrees from the University of Illinois at Urbana-Champaign (UIUC) in 2004 and 2009 respectively. After a post-doctoral stint at Microsoft Research (Redmond), Karthik joined the University of British Columbia (UBC) as an assistant professor of electrical and computer engineering. Karthik’s research interests include programming languages, compilers and computer architecture for building error resilient software systems.

Matei Ripeanu Matei Ripeanu is an Associate Professor at the University of British Columbia. Matei is broadly interested in experimental parallel and distributed systems research with a focus on massively parallel accelerators, data analytics, and storage systems. The Networked Systems Laboratory website (netsyslab.ece.ubc.ca) offers an up-to-date overview of the projects he works on together with a fantastic group of students.

Sudhanva Gurumurthi Sudhanva Gurumurthi is a Senior Data Center Engineer and Manager at the IBM Cloud Innovation Lab and a Visiting Associate Professor in the Computer Science Department at the University of Virginia. Prior to joining IBM, he was a Senior Member of the Technical at AMD where he directed projects on resiliency and reliability. He used to be a tenured Associate Professor at the University of Virginia. Sudhanva’s interests are in the field of computer architecture and systems. He is a Senior Member of the IEEE and the ACM.

Appendix A

We provide a short description for our benchmark applications and their application inputs below.

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

HashGPU: HashGPU [24] is a library that accelerates hash-based primitives. We use SHA1 and MD5 on a 32KB dataset.

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 3D matrix.

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. We transpose a 512*512 floating-point matrix.

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.

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.

CUDA Parallel Prefix Sum (SCAN): SCAN [45] 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 given length of arrays.

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.

Merge Sort (MS): MergeSort [29] implements a merge-sort algorithm to sort batches of short- to mid-sized (key, value) array pairs. We sort 32,768 key-value pairs.

Breadth-First Search (BFS): Breadth-first search on a graph. We use a random graph with 4096 nodes.

Page Rank (PageRank): PageRank counts the links and quality of those links of webpages to estimate how important those webpages are. We run Page Rank on a R-MAT graph with 1 million vertices.

Single Source Shortest Path (SSSP): SSSP finds the shortest path to reach every vertex from the source vertex.

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.

Appendix B

We can measure the number of data bits in a program, and also the number of instruction bits for each thread. Using this, we can calculate the ratio of the number of data bits to the total number of bits for each warp. Let γ denote the ratio of the number of data bits to the total number of bits for a single thread (i.e., sum of data bits and instruction bits), and let n denote the warp divergence observed. For simplicity, we assume that all threads in the program have the same numbers of instruction and data bits. Equation 6 shows the formula for calculating the ratios of the number of data bits to the total number of bits for the entire warp. Instruction bits are multiplied by n as each warp has a separate instance of the instruction. Equation 7 shows how to express the ratio based on the values of γ and n. The values of γ and n for each benchmark are shown in Table VII. We obtain these values through measurements and manual inspection. We use Equation 7 to calculate the ratio for each program in Table VII.

$$Ratio_{warp} = \frac{32 * Data\ bits}{32 * Data\ bits + n * Instruction\ bits} \tag{6}$$

$$Ratio_{warp} = \frac{1}{1 + \frac{n*(1-\gamma)}{32*\gamma}} \tag{7}$$

TABLE VII: The ratio of data bits to the total bits (data bits + instruction bits) both in a single thread and within a warp, for our benchmarks. For benchmarks that have multiple groups, we show the average ratio of threads from different groups. The level of warp divergence is presented as the number of divergent groups of threads inside a warp.

Benchmark	Ratio for a single thread (γ)	Level of warp divergence (n)	Ratio for a warp
SAD-k0	(47.84+/-0.01)%	2 (31:1)	93.60%
SAD-k1	45.81%	2 (31:1)	93.10%
SAD-k2	45.78%	2 (31:1)	93.10%
Stencil	(47.80+/-0.01)%	2 (31:1)	93.58%
MRI-Q	46.57%	0	95.88%
LBM	(45.25+/-0.1)%	2 (31:1)	92.97%
MAT	45.91%	0	96.30%
MONTE	(47.82+/-0.02)%	0	95.98%
HashGPU-md5	(49.52+/-0.01)%	0	96.12%
HashGPU-sha1	(49.40+/-0.01)%	0	96.11%
MergeSort-k0	46.78%	0	95.90%
Transpose	47.29%	0	95.94%
Scan	(47.76+/-0.1)%	2 (28:4)	85.40%
BFS	52.99%	4 (29:1:1:1)	90.07%
AES	47.41%	0	95.95%
PageRank	48.90%	0	96.07%
SSSP	46.00%	0	95.83%
Average	47.58%	N/A	94.23%