GPU-Qin: A 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
Email: {bof, karthikp, matei}@ece.ubc.ca
†AMD Research, Advanced Micro Devices, Inc.
Email: sudhanva.gurumurthi@amd.com

***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 paper makes three key contributions. First, it presents the design of a fault-injection methodology to evaluate 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, this paper characterizes the error resilience characteristics of twelve GPGPU applications.

I. 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 [1]. For example, Haque et al. reported [2] that two-thirds of more than 50,000 GPUs available on Folding@Home, a popular volunteer computing platform, exhibited ”pattern-sensitive susceptibility to soft errors in GPU memory or logic.”

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, 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 [3]. While there has been substantial work in the realm of fault injection for CPU applications [4], [5], there have been relatively few studies that have explored the reliability properties of GPGPU applications and proposed methodologies and tools to support this exploration. The major challenge for fault-injecting GPGPU applications is that, due to the massive parallelism of the platform, it is difficult to achieve a representative coverage of the application execution paths while still being time-efficient.

Prior work [6] 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 [7]. 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.

We 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 micro-architectural level injectors, our 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 paper 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 (Section III),
2) Builds a fault-injection tool, GPU-Qin, that is able to inject faults into applications running on the actual GPU hardware (Section III),
3) Demonstrates the use of the fault injector by performing an end-to-end error-resilience characterization of twelve different GPGPU applications (Section IV), and
4) Provides initial insights that explain the error resilience of these applications. (Section V)

II. Background and Fault Model

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

A. Dependability Metrics: Error Resilience and Vulnerability

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. 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 paper, because we are interested in developing and evaluating application-specific, software-based fault-tolerance mechanisms for GPGPU applications.

B. 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) [8] 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 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.

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. 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 [9]. We focus on transient faults in our study.

We consider transient faults in the functional units of the GPU processor. Examples are faults in the arithmatic 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 *d*e-facto fault model adopted in studies of transient faults [6], [10], [11]. However, our fault injector can support both single- and multiple-bit flips. We will consider multiple bit errors in future work.

D. 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.

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 it, we develop the 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.This is a different criterion than used by RTL-level and micro-architectural fault injections, as discussed in Section II.
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.

We implement our methodology based on the CUDA GPU debugging tool namely *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. *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.

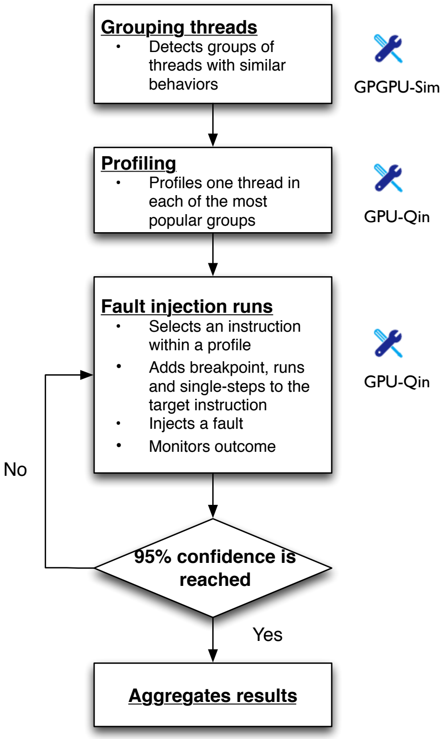
Grouping threads
Detects groups of threads with similar behaviors
GPGPU-Sim
Profiling
Profiles one thread in each of the most popular groups
GPU-Qin
Fault injection runs
Selects an instruction within a profile
Adds breakpoint, runs and single-steps to the target instruction
Injects a fault
Monitors outcome
GPU-Qin
No
95% confidence is reached
Yes
Aggregates results

Fig. 1: Overview of our fault-injection methodology

Figure 1 shows an overview of our methodology. The process consists of four main phases. In the first phase, we group threads based on similarity of their behaviors (and 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). 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 Section III-A.

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 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 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 with 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 time (in other words, we assume that all instructions take approximately the same time to execute).

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

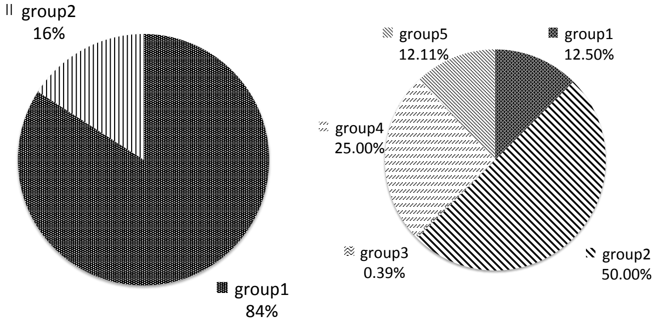

Fig. 2: Percentage of number of threads in each group to the total number of thread. *Left*: LBM *Right*: Monte Carlo

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 section presents each phase in detail.

A. Phase I: Grouping

GPU applications often have a massive number of threads [2], and it would be infeasible to obtain the execution traces for all threads in an application 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 consider instruction counts as a proxy for thread behavior.

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) [7]. 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 Section IV) can be categorized into three categories based on the results of the group identification process (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 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 heuristic: 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 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.

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

Category	**Benchmarks**	**Groups**	**Groups to profile**	**% threads in picked groups**
Category I	AES, MRI-Q, MAT, MergeSort-k0, Transpose	1	1	100%
Category II	SCAN, Stencil, Monte Carlo, SAD, LBM, HashGPU	2 - 10	1 - 4	95% - 100%
Categroy III	BFS	79	2	>60%

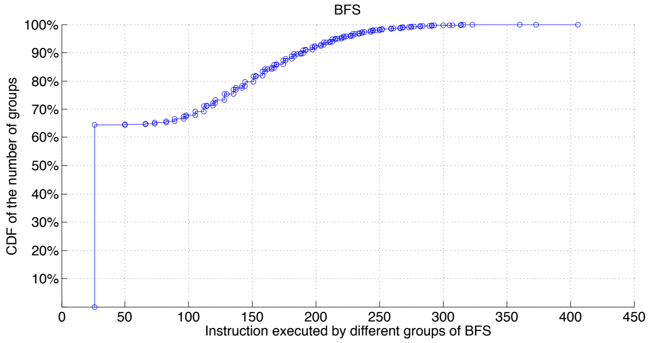

Fig. 3: Cumulative Distribution Function(CDF) of groups of BFS

For applications that have a large number of groups (in our benchmark set, only BFS (Table I)), 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), 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.

B. Phase II: Profiling

The goal of the profiling phase is to map the instructions 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 instruction we want to inject.

[2] A GPU thread is identified by a thread coordinate (blockIdx.x, blockIdx.y,blockIdx.z), (threadIdx.x, threadIdx.y, threadIdx.z).

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.

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 each trace. In each injection run, 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 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., 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.

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

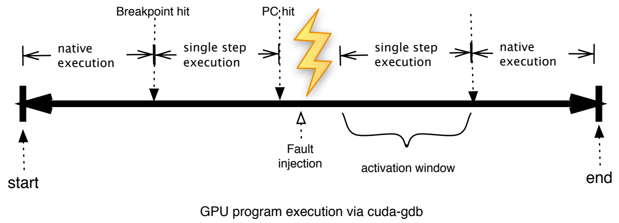

Fig. 4: Phase III - The fault-injection process

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 does the instruction occur 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 section.
2) Once the current loop iteration matches the estimated 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.

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

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.

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.

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.

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 section.

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

IV. Characterization Study

This section uses 15 GPU kernels of 12 distinct applications (presented in Section IV-A) to validate the design choices that we made (Section IV-B) and to demonstrate the use of our methodology to characterize the application's error resilience (Section IV-C). All of our experiments are conducted on NVIDIA Tesla C series GPUs.

A. Benchmarks

We use a variety of benchmarks from the Parboil benchmark suite [12], NVIDIA CUDA SDK package, Rodinia benchmark suite [13], and other well-known GPGPU applications. A short description of each benchmark is given below, along with the inputs used in our evaluation. Table II 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 MAT launches the CUDA kernel code only once, to ensure that subsequent runs do not overwrite the results. We multiply two 192*128 FP matrices.

Matrix Transpose: Matrix transpose is a common building block for many linear algebra algorithms. We use the diagonal kernel optimized for the biggest memory bandwidth. We transpose a 512*512 floating-point matrix.

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

GPUs as Storage System Accelerators (HashGPU): HashGPU [14] 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 [15] 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 [16] implements a merge-sort, representing a use case of GPUs for sorting batches of short- to mid-sized (key, value) array pairs.

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.

B. Design Decision Validation

This section offers empirical support for the three heuristics used in Section III. All these heuristics 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). The heuristics are:

1) Threads are partitioned into different groups, then profiling and fault injection is based on the most popular groups.
2) 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.
3) 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 compare the fault-injection results of applications in categories II and III (see Table I). We find that crash rates vary considerably for different groups of threads in Stencil, LBM, SCAN and BFS. The differences

[3] 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.

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 iterations that the kernel is launched.

Benchmark	**Benchmark Suite**	**Kernel properties**			
		Name	*Approximate LOC*	*Scale*	*Launch Times*
SAD	Parboil	mb_sad_calc larger_sad_calc_8 larger_sad_calc_16	220 60 50	(44,36,1)*(61,1,1) (44,36,1)*(61,1,1) (11,9,1)*(32,4,1)	1 1 1
Stencil	Parboil	block2D_hybrid_coarsen_x	100	(2,32,1)*(32,4,1)	5
MRI-Q	Parboil	ComputeQ_GPU	50	(128,1,1)*(256,1,1)	3
LBM[a]	Parboil	performStreamCollide	150	(120,150,1)*(120,1,1)	100
MAT	CUDA SDK	matrixMul	110	(4,6,1)*(32,32,1)	1
SCAN-block	CUDA SDK	scanExclusiveShared	70	(6656,1,1)*(256,32,1)	1
MONTE	CUDA SDK	MonteCarloOneBlockPerOption	40	(32,1,1)*(256,1,1)	1
Transpose[a]	CUDA SDK	transposeDiagonal	40	(64,64,1)*(16,16,1)	1
MergeSort	CUDA SDK	mergeSortSharedKernel	50	(4096,1,1)*(512,1,1)	1
BFS	Rodinia	Kernel Kernel2	20 15	(8,1,1)*(512,1,1) (8,1,1)*(512,1,1)	8 8
AES	Other [17]	aesEncrypt256	400	(257,1,1)*(256,1,1)	1
HashGPU	Other [14]	sha1_kernel_overlap md5_kernel_overlap	1000 1000	(64,1,1)*(64,1,1) (64,1,1)*(64,1,1)	1 1

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

in Stencil, SCAN-block and BFS are 5%, 10%, and 25% respectively. This shows the potential impact of this design decision and demonstrates the value of considering different groups in our profiling strategy.

To validate the second heuristic, we first measure the total number of iterations executed by each loop of each kernel, and then consider the loop with the maximum iterations. The results are shown in Figure 5. 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 6 presents the SDC rates and crash rates for MAT and MRI-Q for 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 our choice does not affect the overall error resilience estimation.

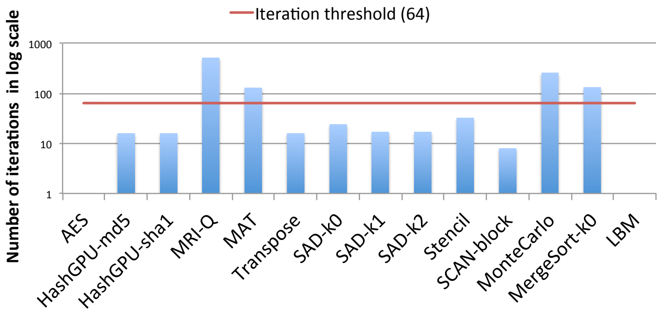

Fig. 5: The number of loop iterations executed by each benchmark kernel.

To validate the third 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: 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. Thus our choice of the activation window size leads to only minimal inaccuracy in evaluating error resilience.

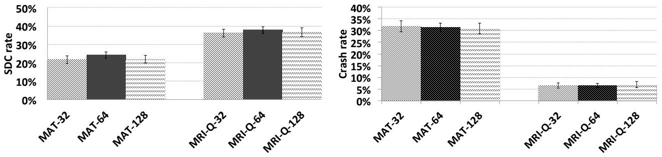

Fig. 6: Comparison of SDC and crash rate for different iteration threshold. *Left:* SDC. *Right:* Crash

C. 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 III 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 II). 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 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 7 presents the SDC rate and crash rate of the

TABLE III: Fault-injection experiments information

Kernels	Injected runs	Activated runs	Activation rate	Average time per run (seconds)
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
LBM	1,895	1,845	97%	165

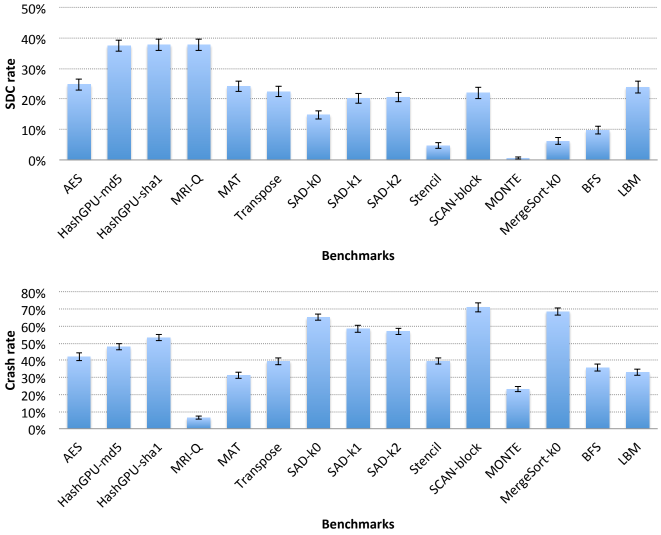

Fig. 7: SDC (top) and crash (bottom) rates with error bars representing 95% confidence interval for each kernel

benchmark kernels. We do not show the hang rates because they are uniformly lower than 1%. Fault injections in CPUs exhibit similar hang rates [10] 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 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 behaivor, (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 section. In total, across all benchmarks, failure rates (crash+SDC+hang) range from 24% (MONTE) to 93% (SCAN), and the average failure rate is 67%.

TABLE IV: Description of CUDA hardware exceptions

Exception type	Descrption
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

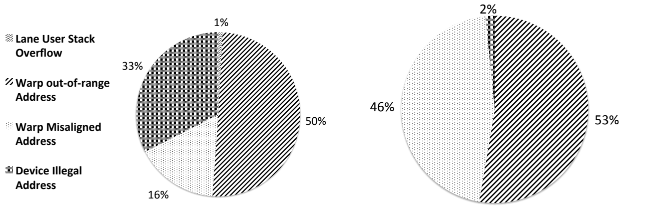

Fig. 8: Root-cause breakdown of crashes for AES and MAT. *Left:* AES. *Right:* MAT.

D. Crash Causes and Latency

GPU-Qin can be used to gain a deeper understanding of the error-resilience characteristics of GPGPU applications. Here, we attempt to understand the reasons for the crashes observed in the characterization study, and characterize the *crash latency*. These metrics are important for two reasons. First, crashes are a form of error detection performed by the GPU, 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. Due to space constraints, 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*. These exceptions and their causes are presented in Table IV.

Figure 8 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.

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 9 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

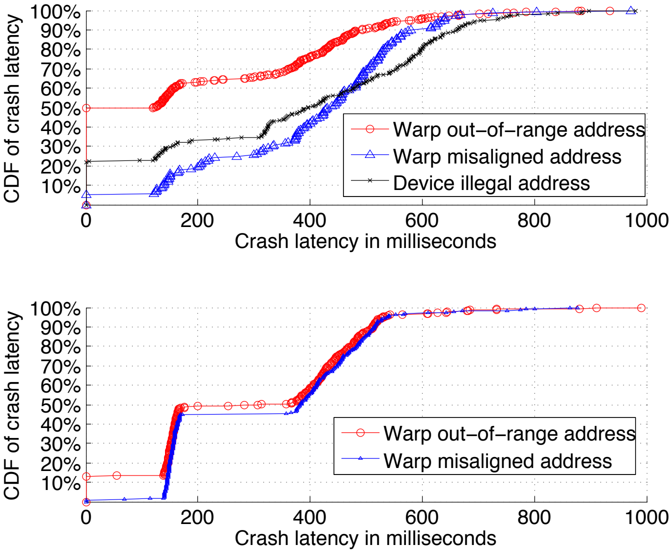

Fig. 9: Crash latency analysis for AES and MAT. *Top:* AES *Down:* MAT

exceptions occur faster compared to warp misaligned address exceptions. Only in the Stencil benchmark does the *device illegal address* exception occur 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.

V. Discussion

The fault-injection study presented in the previous section 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 section, 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.

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 in CUDA SDK implements parallel sorting based on binary search [16], and we observe a relatively low SDC rate (6%).

Another type of applications that has a low SDC rate is what we call an "average out" algorithm, such as Stencil (SDC rate: 5%) and MONTE (SDC rate: 1%). These include computations in which the final state is a 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. If a fault happens in one of the temporary states, it is likely that it would be averaged out in the final state.

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 V. Asanovic et al. [18] 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 V shows. We did not start out trying to find such a mapping, and hence may not cover all dwarfs in our application categories. We will explore this mapping systematically in future work.

VI. Related Work

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

Fault injection has been well-explored on CPUs using runtime debuggers. Examples are GOOFI [5] and NFTAPE [4]. 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 [19] 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.

Several studies [20], [21] have attempted to characterize the vulnerability of different micro-architectural structures in GPUs to soft errors through architecture vulnerability factor (AVF) analysis [22]. However, 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. In contrast, our work is from the applications' perspective, and focuses on understanding the behavior of GPGPU applications under errors. Program Vulnerability Factor (PVF) is a metric proposed by Sridharan et al. [23] to apply AVF analysis at the application layer. While this takes application properties into account, it still does not consider the end-to-end impact of faults on the application.

Dimitrov et al. [24] 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.

Finally, Yim et al. [6] 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. Further, they perform fault injections at the source-code level, and it is unclear how representative of hardware faults are their injections.

TABLE V: Benchmark categories and the mapping to the dwarfs of parallelism

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

VII. Summary

This paper 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. Our fault injector enables the opportunity to study various reliability characteristics of applications, such as crash latency. Finally, we find that algorithmic characteristics of the application can help us understand the variation in the SDC rates among different 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 - please contact the authors if you are interested.

References

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

[2] I. S. Haque and V. S. Pande, “Hard data on soft errors: A large-scale assessment of real-world error rates in gpgpu,” in *Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, ser. CCGRID ’10, 2010.

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

[4] 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.

[5] 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.

[6] K. S. Yim, “Hauberk: Lightweight silent data corruption error detector for gpgpu,” in *IEEE International Parallel and Distributed Processing Symposium*, 2011.

[7] 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, 2009. ISPASS 2009. IEEE International Symposium on*, April 2009, pp. 163–174.

[8] 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 (IPFA), 2010 17th IEEE International Symposium on the*, 2010, pp. 1–5.

[9] 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.

[10] 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, pp. 887–896.

[11] J. Wei and K. Pattabiraman, “BLOCKWATCH: Leveraging similarity in parallel programs for error detection,” in *Proceedings of the IEEE International Conference on Dependable Systems and Networks (DSN)*, 2012.

[12] J. A. Stratton, C. Rodrigues, I.-J. Sung, N. Obeid, L.-W. Chang, N. Anssari, G. D. Liu, and W. mei W. Hwu, “Parboil: A revised benchmark suite for scientific and commercial throughput computing,” in *IMPACT Technical Report*, 2012.

[13] 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 *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC)*, ser. IISWC ’09, 2009, pp. 44–54. [Online]. Available: http://dx.doi.org/10.1109/IISWC.2009.5306797

[14] 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*. New York, NY, USA: ACM, 2008, pp. 165–174. [Online]. Available: http://dx.doi.org/10.1145/1383422.1383443

[15] 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.

[16] N. Satish, M. Harris, and M. Garland, “Designing efficient sorting algorithms for manycore gpus,” NVIDIA Corporation, NVIDIA Technical Report NVR-2008-001, Sep. 2008.

[17] S. A. Manavski, “Cuda compatible gpu as an efficient hardware accelerator for aes cryptography,” in *IEEE Intl Conf. on Signal Processing and Communication*, 2007, pp. 65–68.

[18] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick, “The landscape of parallel computing research: A view from berkeley,” EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2006-183, Dec 2006. [Online]. Available: http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.html

[19] D. Li, J. Vetter, and W. Yu, “Classifying soft error vulnerabilities in extreme-scale scientific applications using a binary instrumentation tool,” in *High Performance Computing, Networking, Storage and Analysis (SC), 2012 International Conference for*, 2012, pp. 1–11.

[20] 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.

[21] 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*, 2012.

[22] S. Mukherjee, C. Weaver, J. Emer, S. Reinhardt, and T. Austin, “Measuring architectural vulnerability factors,” in *IEEE MICRO*, vol. 23, no. 6, 2003, pp. 70–75.

[23] V. Sridharan and D. Kaeli, “Eliminating microarchitectural dependency from architectural vulnerability,” in *High Performance Computer Architecture, 2009. HPCA 2009. IEEE 15th International Symposium on*, 2009, pp. 117–128.

[24] 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, pp. 94–104.