GPGPUs: How to Combine High Computational Power with High Reliability

L. Bautista Gomez[1], F. Cappello[1], L. Carro[2], N. DeBardeleben[3], B. Fang[4], S. Gurumurthi[5], K. Pattabiraman[4], P. Rech[2], M. Sonza Reorda[6]

[1]Argonne National Laboratory
USA
{leobago, cappello}@mcs.anl.gov

[2] Institudo de Informatica, Universidade Federal do Rio Grande do Sul
Porto Alegre, RS, Brazil
{prech, carro}@inf.ufrgs.br

[3] Ultrascale Systems Research Center
Los Alamos National Laboratory
USA
ndebard@lanl.gov

[4]Electrical and Computer Engineering
University of British Columbia (UBC)
Canada
{karthikp, bof}@ece.ubc.ca

[5]AMD Research
Advanced Micro Devices, Inc.
Boxborough, MA
USA
sudhanva.gurumurthi@amd.com

[6]Dip. Automatica e Informatica
Politecnico di Torino
Torino, Italy
matteo.sonzareorda@polito.it

Abstract— **GPGPUs are used increasingly in several domains, from gaming to different kinds of computationally intensive applications. In many applications GPGPU reliability is becoming a serious issue, and several research activities are focusing on its evaluation. This paper offers an overview of some major results in the area. First, it shows and analyzes the results of some experiments assessing GPGPU reliability in HPC datacenters. Second, it provides some recent results derived from radiation experiments about the reliability of GPGPUs. Third, it describes the characteristics of an advanced fault-injection environment, allowing effective evaluation of the resiliency of applications running on GPGPUs.**

Keywords—GPGPUs, reliability, HPC, fault injection, radiation experiments

I. INTRODUCTION

In the last decade, new devices known as *general-purpose graphics processing units* (GPGPUs) made their appearance on the market. Their very high computational power combined with low cost, reduced power consumption, and flexible development platforms is pushing their adoption for graphical applications (especially in gaming) as well as high-performance computing (HPC). Moreover, GPGPUs are increasingly used in mobile computing and in some embedded system applications. In the latter case safety is sometimes a key issue (e.g., in the automotive, avionics, space and biomedical domains). As an example, the Advanced Driver Assistance Systems (ADAS), which are increasingly common in cars, make an extensive usage of the images (or radar signals) coming from external cameras and sensors to detect the occurrence of dangerous situations, and trigger the activation of proper countermeasures. The high computation power required to process these data perfectly matches the characteristics of GPGPUs, and several vendors already introduced or announced specific products in this area.

In both HPC and safety-critical embedded applications scenarios, GPGPU reliability is a serious issue. On one side, it is essential to quantitatively and independently evaluate the probability that faults (both transient and permanent) affect these devices, once the characteristics of the operating environment are known. Secondly, it is essential to evaluate the probability that hardware faults evolve into failures, and to classify the latter ones according to their severity. Finally, efficient and optimized software and architectural techniques have to be designed to harden GPGPU-based systems. Given their high degree of parallelism, we could assume that GPGPUs could intrinsically provide a good degree of fault tolerance. Moreover, when applied to graphical applications, the effects of faults are often negligible. Nevertheless, their size and complexity could make them particularly sensitive to soft errors and the usage of a huge number of GPGPUs (as it happens in HPC centers) makes the probability of a system failure significant, even if the failure probability of a single device is low.

Hence, some relevant scientific activities were triggered in the last few years, covering first the issue of how to evaluate the reliability of these devices, especially with respect to soft errors caused by radiation effects (even at terrestrial level) [1] and in HPC datacenters [11]; secondly, analytical and experimental studies (e.g., based on fault injection) have been performed to track error propagation through the device towards the outputs [2][7][8]; finally, some research activities aimed at enhancing the GPGPUs reliability [3][6] and to assess the achieved robustness [4][5][9][10]. In particular, while some improvements in the GPGPU architecture and implementation may succeed in reducing their sensitivity to faults, researchers started developing techniques to make application codes more resilient to errors, by acting both at the algorithm level and on the programming style.

This paper aims first at providing the reader with an updated view on what GPGPUs should provide not only in terms of performance and power, but also in terms of reliability when applied in specific scenarios, such as the HPC one, reporting the results of analysis performed at Los Alamos National Laboratory (LANL) and Argonne National Laboratory (ANL). Secondly, the paper summarizes some

978-3-9815370-2-4/DATE14/©2014 EDAA

major results stemming from recent research activities performed by UFRGS focused on GPGPU reliability evaluation through radiation experiments. Finally, it deals with the evaluation of the resiliency of applications running on GPGPUs, and describes a recent tool developed by UBC to perform fault injection experiments on GPU applications.

This paper is organized as follows. Section II reports an analysis of the requirements that a HPC center typically sets for GPGPUs and some experimental evaluation of how well they are matched. Section III summarizes the main results achieved so far in the evaluation of the reliability of some GPGPUs through radiation experiments. Section IV describes the characteristics that are expected from a fault injection environment for GPGPUs, and outlines a recently proposed solution for evaluating the resiliency of GPGPUs applications. Finally, section V draws some conclusions.

II. GPGPUs in High Performance Computing

GPGPUs offer great potential to the field of high performance computing (HPC) / supercomputing and they have been already adopted by many datacenters. These devices offer extreme performance improvements over general purpose CPUs for applications that can adapt to a new programming paradigm. Luckily, there are many important applications in HPC that indeed do fit this paradigm and major performance improvements have already been realized. Furthermore, coupled with GPUs original audience of visualization, many researchers have capitalized on in-situ analysis to unlock interesting new modes of computing that hold a great deal of potential.

HPC applications are by their very nature distributed. That is, they run across multiple (in many cases, very many) devices in a datacenter. Contemporary HPC systems have between hundreds of thousands to millions of compute cores. The current fastest GPU system (Titan at Oak Ridge National Laboratory) has nearly 19 thousand NVIDIA Kepler GPUs. As of November 2013, Titan is the 2nd fastest supercomputer in the world as ranked on the top500 list (http://www.top500.org), so there is no question that GPUs are gaining prominence in the HPC field. However, two areas still hold concern for the HPC community: GPGPU performance variability and GPU reliability.

Due to the way HPC applications are structured, work is done in parallel on different devices. Periods of synchronization in the application are costly and a great deal of effort is put into keeping applications from exchanging data with other parts of the supercomputer. In many ways, the ultimate performance of the application is limited by the amount of synchronization. Therefore, it is of utmost importance that when devices synchronize to exchange data these periods are as closely aligned as possible. This is why performance variation is so damaging on supercomputers and why there has been much research into system noise to reduce "jitter". As such, one feature of importance to the HPC field is predictable, regular GPU performance.

A series of experiments have been conducted by researchers at Los Alamos National Laboratory (LANL) and Argonne National Laboratory (ANL) on NVIDIA GPGPUs. DeBardeleben, et al., in [11] showed a large degree of variability on NVIDIA M2090 Tesla cards at LANL when using the High Performance Linpack (HPL) and SHOC benchmarks. Cappello and Bautista Gomez at ANL performed a similar analysis on NVIDIA GPUs and presented this work at the SC'13 Critically Missing Pieces in Heterogeneous Accelerator Computing BOF. Results from these experiments as well as a follow-on collaboration set of experiments between LANL and ANL follows.

Moonlight is a GPU production cluster at LANL consisting of 308 compute nodes, each with 2 NVIDIA M2090 Tesla cards. This machine has been in production since 2011, during which time many experiments have been done to evaluate the GPGPUs. A great deal of performance variation is present on this machine but in general it appears to be linked with the GPU number in each node. Experiments were conducted using a variety of software packages including High Performance Linpack (HPL), the SHOC benchmark, and other benchmarks [11]. Interestingly, while performance variation is attached to specific applications, not all applications exhibited this problem. It is likely that this effect is due to external, environmental effects such as temperature and throttling. However, on the M2090 cards, measuring these values is not possible.

In an experiment done by Argonne National Laboratory (ANL) on a cluster with 209 compute nodes, each with 2 NVIDIA M2050, large performance variations where discovered. The experiment consisted in almost 9000 short HPL runs (about 5 minutes each), for a total of more than 550 hours of HPL execution. We used the hpl-2.0_FERMI_v15 code, which can be downloaded from the NVIDIA website [12]. The results, depicted in Figure 1, show that the performance of HPL in these devices vary greatly, between 100Gigaflops and 500 Gigaflops, and only a third of the runs showed the expected performance of ~450 Gigaflops.

As a way to control this experiment, ANL developed a test program that can be used to check for performance variations and silent data corruption. The test computes the product of a matrix A and a matrix B (using cublasSgemm) thousands of times and checks that the results are bit-by-bit identical each time. LANL extended this tool to also include testing of many internal GPU characteristics and counters. Figure 2 shows a study of over 100,000 individual runs of this application on the Moonlight cluster. One can clearly see that the two GPUs on each node behave differently. In fact, through further analysis it was discovered that GPU 0 is running at reduced clock rates and reduced power states. Again, without the ability to accurately measure temperature and power, we were unable to precisely determine why this was occurring. Instead, we note that these devices are likely to exhibit performance fluctuation that reduces their usefulness in HPC environments.

Recent testing of the NVIDIA next generation, Kepler, product line has shown that performance is varying considerably less than it was in the Fermi line. More experiments need to be conducted as more of Kepler HPC systems become available for system testing.

Reliability and resilience are key areas of interest in HPC as most HPC applications are used to set policy or make national security decisions. Example uses of HPC include climate prediction, combustion, nuclear stockpile stewardship, etc. In

these applications, often small data corruptions can lead to incorrect output through error propagation. Therefore, system users and administrators closely observe error rates of devices in supercomputers.

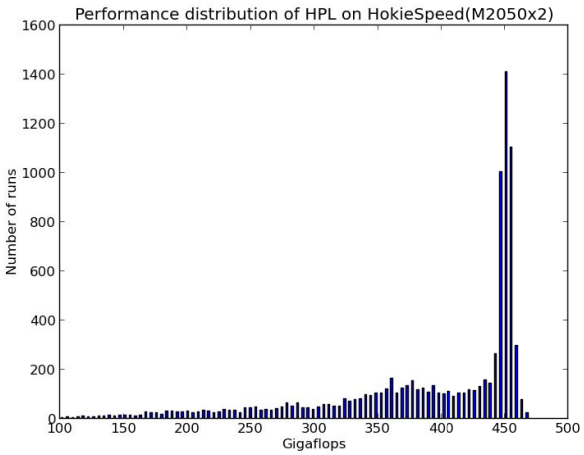

Figure 1 - Performance variation of HPL on M2050x2 GPGPUs on the HookieSpeed Cluster

In experiments conducted at LANL on the HPL application, 4% of the runs resulted in bad residuals – a signal that data corruption had occurred. Upon closer inspection, it was determined that round-off error propagation had accounted for at least some of these. 0.07% of these runs had associated double bit errors in the system logs but the error counters on these devices did not match this number. Further experiments have shown that mismatch is common between the aggregate and volatile error counters as well as with the errors reported to system administrators.

During a four month time period in 2013 the Moonlight M2090 GPGPUs were under heavy utilization at LANL. In this time 135 double bit errors were reported by the GPUs into the system logs. These were spread out across the machine and so not attributable to one or two bad cards. This means that Moonlight saw 1.32 double bit errors per day during this period. In comparison, the Cielo supercomputer at LANL experienced one uncorrectable DRAM error every 11-12 days. However, it is not fair to directly compare these machines as Cielo is a 9,000 node Cray XE6 with 286 terabytes of DDR3 and Moonlight only has 308 nodes, each with 2 M2090 NVIDIA GPGPUs and 6GB GDDR5. Still, in this comparison, the double bit error rate of the M2090s seen on Moonlight are over 4 orders of magnitude higher than Cielo when compared on per gigabyte of memory. This number seems excessive, even when accounting for the inherent reliability differences between SECDED ECC (on Moonlight's M2090 GDDR5) and single Chipkill correct (on Cielo's DDR3). When considered from a user's perspective, uncorrectable errors result in application interrupt or potentially data corruption if not checked in the case of the GPGPU. Therefore, comparing the rates between the two is interesting from the effects they have on users.

Small scale experiments conducted so far on NVIDIA Keplers have seen the performance variation disappear between GPUs. However, many more experiments need to be conducted and much more data needs to be collected before a final conclusion can be reached. Furthermore, application hangs are not uncommon. In one experiment of 500 runs of the above mentioned matrix multiplication application, 23% of the runs hung. After further analysis, it was discovered that these errors occurred only while using CUDA5.0. The problem disappeared after moving to CUDA5.5. On the M2090 Moonlight cluster around 15% of GPU applications hung during a similar experiment.

Like any new piece of hardware, there are complexities involved with integration in extreme scale, leadership-class computing systems. It is apparent that improvements are being made but there is still room for improvement. Users of these devices are cautioned to develop applications that are able to check for correctness. Additionally, users test for performance variation and carefully consider developing applications that are more tolerant of inter-process latency if they hope to harness the full power these GPGPUs have to offer.

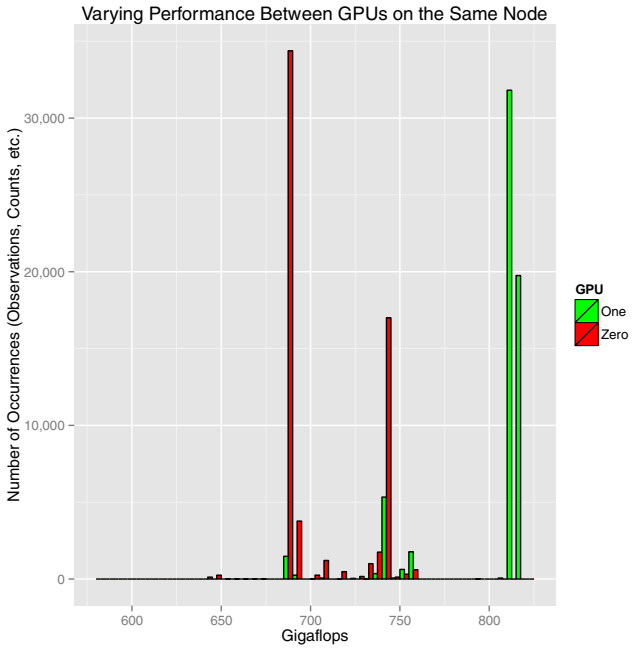

Figure 2 - Performance variation on M2090 GPGPUs on the Moonlight cluster at LANL

III. GPU RELIABILITY ASSESSMENT AND ENHANCEMENT

The newest GPU core processors are built with cutting-edge technology and thus are potentially very prone to experience radiation-induced errors [13][14], even on terrestrial applications, where neutrons are among the main responsible for failures. Atmospheric neutrons may corrupt both memory and logic resources of a GPGPU. The radiation fault could be masked or propagate to the output generating single/multiple silent data corruptions. Finally, radiation may cause functional interruptions or system hangs.

GPGPUs are typically divided into several compute units that share caches. A code to be executed on a GPGPU is normally composed of several parallel threads. A scheduler and a dispatcher are in charge of assigning a thread to a compute unit, and synchronizing the computation [15]. When exposed to radiation, the compute units are isolated such that a single

radiation-induced event in one computing unit will only corrupt the thread assigned to it. Threads that follow the corrupted one or threads assigned to compute units near the struck one will not be affected. Nevertheless, the corruption of shared resources, like caches, or critical resources, like the scheduler, may affect the execution of several threads, generating multiple output error [16][17].

In this scenario, an experimental characterization of GPGPUs radiation sensitivity becomes essential to precisely evaluate GPGPUs behaviors when exposed to radiation, to estimate an application error rate, and to detect possible output error patterns caused by shared or critical resources corruption. Such an experimental analysis avoids errors underestimation or the introduction of useless overhead, which is fundamental in HPC applications. In the following, an overview of the experimental results obtained on NVDIA GPGPUs is provided, highlighting the shrewdness necessary to design and conduct a successful test and the impact of radiation data analysis on the design of specific correcting procedures.

A. Radiation Experiments Setup

The presented analysis is based on commercial-off-the-shelf GeForce GTX480 and Tesla C2050 GPUs manufactured by NVIDIA in a 40nm technology node. For the GTX480 GPU, 15 blocks of threads can be executed in parallel with a maximum of 32 threads in each block for a total of 480 threads while for the C2050, 14 blocks of 32 threads each can be executed in parallel, for a total of 448 threads.

While the tested devices are neither used in HPC nor in safety-critical applications, their architecture and parallelism management are very similar to the one of more powerful devices (like the Tesla GPUs used in HPC applications) or of low-power devices (like the Tegra System on Chip, used in automotive and aerospace applications). The proposed test methodology and the achieved results will be easily extendable to other GPUs families, and give novel insight on the response to radiation of generic parallel computing systems. The C2050 is equipped with the same ECC mechanism used in devices voted to HPC applications like the K20 or K40. Such mechanism reliability will be addressed in the following.

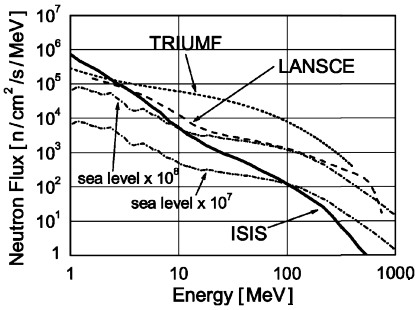

Figure 3 - LANSCE and ISIS neutrons spectra compared to those of TRIUMF facilities and the terrestrial one at sea level multiplied by 10^7 and 10^8 [18].

Experiments were performed by UFRGS researcher at Los Alamos Neutron Science Center (LANSCE), LANL, Los Alamos, NM, USA, and in the VESUVIO beam line in ISIS, Rutherford Appleton Laboratories, Didcot, UK. As shown in Figure 3, both of these facilities provide a white neutron source that emulates the energy spectrum of the atmospheric neutron flux [18].

The available neutron flux was approximately $1x10^6$n/(cm^2·s) in LANSCE and $4x10^4$n/(cm^2·s) in ISIS for energies above 10MeV. It is worth noting that even if the used neutron fluxes are several orders of magnitude higher than the natural one (which is of about 13n/(cm^2·s) at sea level [19]), the experiments were carefully tuned to make negligible the probability of having more than one neutron generating a failure in a single code execution (observed error rates were actually lower than 10^{-2} errors/execution). This allows the scaling of experimental data in the natural radioactive environment without introducing artificial behaviors.

The beam was focused on a spot with a diameter of 2 inches plus 1 inch of penumbra, which provided uniform irradiation of the GPU chip without directly affecting the nearby board power-control circuitry and the DDR memory. Even if the beam is collimated, scattering neutrons may be found outside the beam spot; therefore, to ensure that the DDR content was consistent during our experience, we periodically checked it during experiments, and no error has been observed. As input and output data were stored in the DDR, the errors reported in the following sections were only caused by the corruption of the GPU core resources.

A desktop PC controlled the GPGPU under test through a 20cm long PCI-Express bus extension with fuses to prevent latchups from compromising the DUT and PC functionalities. No latchup was observed during the overall test experience. Irradiation was performed at room temperature with normal incidence and nominal voltages.

B. Experimental Results

The first benchmark tested under radiation performs the multiplication of two 2048x2048 random matrices (A and B) instantiating 2048x2048 parallel threads, each in charge of calculating a single element of the resulting matrix M. Matrix multiplication is an application particularly suitable to be efficiently executed on a GPGPU, as it can be fully parallelized.

As Rech et al. reported in [16], the experimentally obtained cross section for matrix multiplication, computed by dividing the number of faulty executions per time unit by the flux, is $2.66 \cdot 10^{-6}$ cm^2 As the available spectrum of neutron energies at ISIS and LANSCE resembles the atmospheric one [18], the cross section obtained at ISIS is also the probability for a neutron in the natural environment to corrupt the matrix multiplication execution on the GPGPU. Multiplying the cross section and the natural neutron flux (which is of about 13 n/cm^2·h at sea level [19]), one can obtain the radiation-induced error rate of matrix multiplication when executed on a GPU during real applications, which is $3.46 \cdot 10^4$ Failures In Time (FIT) [16].

We can further study the experimental data by analyzing the characteristics of the corrupted resulting matrix. Table I reports and Figure 4 shows graphically the percentage of faulty executions in which a single error or multiple errors were detected on the output matrix (details can be found in [16]). As it can be seen, single output errors are detected in less than 43% of the cases. This result is of extreme importance as it demonstrates that for modern GPGPUs the accredited

assumption of having just single radiation-induced output errors is no longer valid. Figure 4 shows the different error patterns we detected when multiple errors affect the output matrix. In most of the cases, multiple errors are distributed on a single row or column, while just in 8% of the cases errors are randomly distributed.

TABLE I
SINGLE VS MULTIPLE ERRORS

Single Error [%]	42.29
Errors in a Row [%]	22.86
Errors in a Column [%]	26.85
Random Errors [%]	8.00

As said, experiments were tune so to avoid more than one neutron to corrupt the GPU during the same benchmark execution. Moreover, as demonstrated in [13], the Multiple Cell Upsets cross section is normally one order of magnitude lower than the SEU one. Thus, the observed multiple output errors are caused by neither multiple neutrons nor single neutron generating multiple errors. Multiple errors on the same column/row are actually caused by the corruption of caches while randomly distributed errors are caused by scheduler corruption [16][17].

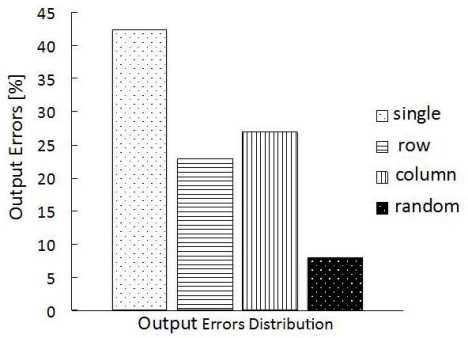

Figure 4 - Percentage of corrupted output matrices affected by single and multiple output errors.

The second benchmark tested by UFRGS researchers under radiation implements 512x512 1D-FFTs of 64-points each. The FFT input is composed of a 64x512x512 double precision floating-point matrix for the real part and a 64x512x512 matrix for the imaginary part. We choose to test relatively small FFTs (64-points) to limit the number of iterations and ease the study of error propagation, while having 512x512 1D-FFTs eases the gathering of a statistically significant amount of errors.

A thread acts like a butterfly module updating the values of two floating-point elements in the complex matrix using the values of two elements computed in the previous iteration as inputs. The implemented algorithm is based on the FT kernel of the NAS Parallel Benchmarks [20] implemented in C and ported to the GPGPU architecture using CUDA. Each 64-points 1D FFT kernel is composed of 6 sequential iterations ($\log_2 64=6$) of a variant of the Stockham FFT algorithm [21]. The FFT algorithm is then composed of threads that are not independent, since a thread uses the output of previously executed threads to update the real and imaginary part of two complex elements. It is then very likely that a radiation-induced event affecting a thread in the early stages of the FFT execution will spread, affecting various bits of the output.

As detailed in [17], the experimentally measured cross section for the FFT benchmark executed on the C2050 (ECC disabled) is $3.69 \cdot 10^{-6}$ cm^2, which equals to $4.80 \cdot 10^4$ FIT at sea level. What is impressive in the case of FFT is the percentage of corrupted executions that present multiple output errors. As reported in Tab. II, more than 95% of faulty executions were affected by multiple errors. As said, this behavior is mainly caused by the characteristic data dependence inherit in the FFT algorithm. Further details are available in [17].

TABLE II
512X512 64-POINTS FFTS SINGLE AND MULTIPLE ERRORS

	FFT Real		FFT Imaginary	
	Single	Multiple	Single	Multiple
Percentage	1.61%	98.39%	4.00%	96.00%
FIT	$7.33 \cdot 10^2$	$4.48 \cdot 10^4$	$1.85 \cdot 10^3$	$4.43 \cdot 10^4$

C. *Experimentally-Tuned Hardening Strategy*

GPGPUs for the HPC market are equipped with an ECC mechanism able to correct single errors and detect double errors in caches, shared memory, and internal registers [22]. When the ECC mechanism is enabled, the output error rate of both matrix multiplication and FFT benchmarks is reduced by at least one order of magnitude as experimentally proved by Rech et al. in [16] and [17]. In the particular case of matrix multiplication, all the multiple errors on the same row/column are corrected, as caches are well protected.

Software-based hardening techniques could be designed analyzing experimental data and the algorithm structure, following the Algorithm-Based Fault Tolerance philosophy [23]. ABFT has been proved to be more efficient than traditional duplication with comparison or TMR in GPGPUs [16]; however, it requires the design of specific correcting procedures to correct the experimentally observed errors.

For matrix multiplication, an efficient ABFT was proposed in [24]. However, the technique is only capable to correct single errors, which correspond to less than 43% of the cases (see Table I). Experimental data presented here were fruitfully used to extend such a technique to correct also multiple errors [16]. The FFT characteristic data dependence, on the other side, requires a prompt error detection to avoid propagation. The ABFT for FFT proposed in [25] should then be extended, dividing the FFT into smaller sub-problems, using the well-known propriety that a N-point FFT can be decomposed into N1 FFTs on N2 points and N2 FFTs on N1 points, where $N1 \cdot N2=N$ [17]. Nevertheless, such efficient error correction procedures are algorithm and code dependent and can hardly be generalized.

IV. EVALUATING THE ROBUSTNESS OF GPU APPLICATIONS

GPUs were originally used for applications that were error tolerant, such as gaming and video applications. However, today GPUs are used to accelerate a wide variety of general-purpose applications (GPGPU applications), whose error tolerance is unclear. Further, due to the effects of technology scaling and power consumption limits, hardware faults are increasing in GPUs [26]. GPGPUs are also increasingly used in large scale systems such as supercomputers (e.g., the Titan supercomputer at ORNL) in which reliability is critical. Thus, there is a compelling need to make GPGPU applications

resilient to hardware errors through software-based techniques and to evaluate and understand the error resilience of GPGPU applications. A common way to evaluate and analyze the error resilience of an application is through fault-injection [27]. Fault injection is the act of perturbing an application's data or instructions to emulate the effect of hardware faults and observing the effects.

A major challenge in fault injection is balancing the efficiency of the fault-injection process with its coverage (i.e., injecting sufficient number of faults to obtain statistically significant results). Because faults typically are injected one at a time in each execution of the application, the application needs to be executed thousands of times to get statistically significant results, which is very expensive in terms of computational time. This problem is exacerbated for GPU applications that often consist of hundreds or even thousands of threads, and hence require many thousands of injections to get reasonable coverage.

Another challenge that arises in using fault injection is that the application needs to be executed to completion in order to understand its resilience characteristics. This is because faults may be masked during the execution of the application, and hence not every injected fault will propagate to the application's output. Further, many applications are tolerant of mild deviations in the output (e.g., physics-based animations), and hence even if a fault propagates to the output of the application, it may not cause a failure [28].

We build a fault injector for GPGPU applications, which balances coverage with efficiency, by pre-analyzing the application and injecting faults into representative elements of the application. Further, our injector works on real GPU hardware (i.e., not an architectural simulator), and can hence execute applications to completion, thereby obtaining insights on their end-to-end error resilience.

In the rest of this section, we first present our fault model and introduce the generic notion of error resilience. We then survey related work on GPU error resilience characterization. Finally, we present the constraints that any fault injection approach should satisfy, and present our fault injection approach that satisfies the constraints.

A. Fault model

Hardware faults can be broadly classified as *transient* or *permanent*. Transient faults are usually "one-off" events, and occur non-deterministically, while permanent faults persist at a given location. Transient fault rates have been increasing due to issues such as diminishing noise margins and shrinking microprocessor feature sizes [29].

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 ALU and the load-store unit. We do not consider faults in cache, memory and register files, as we assume that these elements are protected by Error Correcting Codes (ECC). This is the case for recent GPUs such as the NVIDIA Fermi GPU™. We use the single-bit flip model to emulate transient faults, as this is the de-facto fault model adopted in other studies on transient faults [30][31].

B. Error Resilience and Vulnerability

The *error resilience* of a system is defined as its ability to withstand errors should they occur. Note that 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. 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. Resilience is predominantly a property of the application.

Vulnerability is the probability that the system experiences a fault that causes a failure (e.g., an SDC). This is a property of both the microarchitecture *and* the application. Note that vulnerability is different from error resilience, as error resilience is the *conditional* probability of the program not experiencing a failure given that a fault has occurred. We focus on error resilience, as we are ultimately interested in developing and evaluating application-specific, software-based fault-tolerance mechanisms for GPGPU applications.

C. Related Work

Fault injection has been studied for CPUs using run-time debuggers. Examples are GOOFI [33] and NFTAPE [32]. Neither of these injectors are suitable for GPUs applications. Further, they do not consider multi-threaded programs, nor do they concern themselves with choosing representative parts of the program for injection. Other work [34] has attempted to inject faults in scientific applications using the PIN™ tool from Intel, a dynamic binary instrumentation framework. However, these tools have not been applied on GPU applications.

Architecture vulnerability factor (AVF) analysis [37] has been used to study the vulnerability of several hardware structures in a GPU [35][36]. AVF analysis estimates vulnerability by identifying the bits, which when flipped, can lead to an error in the architecturally visible state of the machine. However, these approaches do not consider the end-to-end impact of faults in applications, nor do they attempt to understand the behaviour of the application under errors. In contrast, our work is from the applications' perspective, and focuses on understanding the behaviour of GPGPU applications under errors.

Program Vulnerability Factor (PVF) is a metric proposed by Sridharan et al [38] to decouple the microarchitectural and architectural components of vulnerability to provide an application-layer abstraction of vulnerability. 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. [39] propose three approaches for GPGPU reliability that leverage both instruction-level parallelism and thread-level parallelism to replicate the application code. Despite these optimizations, 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. [40] propose 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 GPGPU applications. Further, they perform fault injections at the source code level, and it is unclear how representative of hardware faults are their injections, as hardware faults occur at much lower levels.

D. Constraints of Fault-Injection Approaches

In this section, we outline the constraints that any fault injection approach (including ours) should satisfy:

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, and should cover all elements of the application's data (i.e., registers, memory locations and instruction pointer).

2) **Efficiency**: Fault injection experiments should be fast enough to allow the application to be executed to completion in a reasonable amount of 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.

E. Fault Injection Methodology

Our fault-injection methodology satisfies the constraints identified above, as we show in this section.

We implement our methodology based on the CUDA GPU debugging tool, namely *cuda-gdb*. 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. Note however that *cuda-gdb* does introduce timing delays in the application; however, we have not seen any cases where there is considerable deviation in the behaviour of the application due to such delays, as our focus is on general purpose applications. Using *cuda-gdb* thus satisfies the *minimum interference* constraint.

Figure 5 shows an overview of our fault injection methodology. The process consists of four main phases. The first phase is performed in a popular micro-architectural GPGPU simulator, GPGPUSim [41], while the next three phases are performed by our fault injection tool, GPU-Qin.

The first phase consists of grouping together the threads that have similar behavior, in order to choose a subset of threads to inject the fault into. Because GPU programs have hundreds or even thousands of threads, it is important to choose the threads that best represent the behavior of the application. We use the number of instructions executed by each thread as a proxy for its behavior. In other words, we assume that similar threads are likely to execute similar numbers of instructions as they are likely to execute similar paths in the program. We obtain the number of instructions executed by each thread using GPGPUSim. We then group threads into similarity classes based on the number of instructions they execute. Finally, we choose one thread from each group (or the most popular groups) to profile in the next phase. This satisfies the *efficiency* constraint.

In the second phase, we obtain the execution traces of the threads identified in the first phase, and map the trace back to the application source code. This is because *cuda-gdb* requires us to specify the instruction to inject into based on the source line of the program, and hence we need a mapping from the executed instructions to the source lines. This information is used in the next phase to choose the instruction to inject.

In the third phase, we first choose a random instruction uniformly from the set of all dynamic instructions executed by the application to inject into. We also choose the thread to inject into based proportionally on the popularity of the group it represents. Once the instruction and thread to inject into has been chosen, we translate the instruction coordinates to a source line based on the trace gathered in Phase 2. We then set a conditional breakpoint in *cuda-gdb* at this line to be triggered by the chosen thread. When the conditional breakpoint is reached, we single step the program until the chosen instruction is encountered. Once the instruction is reached, a single bit in the destination register of the instruction is flipped to emulate the occurrence of a transient fault, and the program is monitored. Thus, the fault is injected uniformly over the space of executed instructions across all threads. This satisfies the *representativeness criteria.*

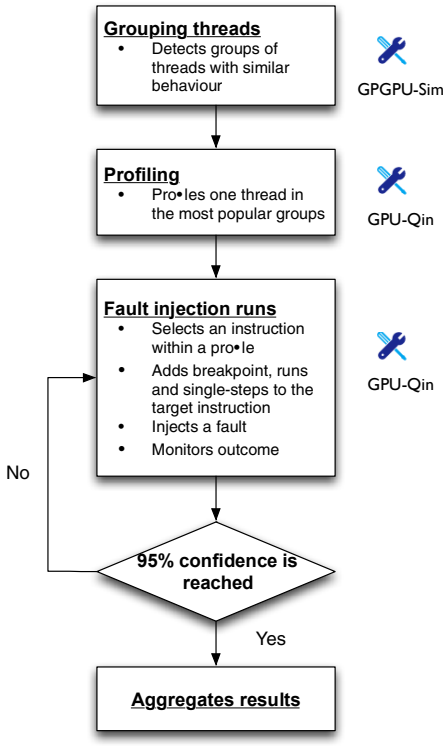

Figure 5 - Our fault injection methodology for GPGPU Applications

Finally, the last phase aggregates the results, after classifying the results of the injection into *correct, crashes, hangs* and *SDCs*. We detect crashes by monitoring the application for exceptions or assertion failures. We use a watchdog timer to detect hangs. We classify outcomes as

SDCs if the output produced by the application does not satisfy the correctness check provided by the developer. Note that this is different from the traditional classification of SDCs where *any* deviation from the correct outcome of the benchmark is considered as data corruption, and takes into account benchmark-specific correctness criteria.
Note that we take into account only *activated* faults, or faults that are actually read by the application. This is because we are interested in evaluating the error resilience of applications, which is defined as the ability to prevent an error from becoming a failure, given that the error has occurred in it. Therefore, we disregard hardware-level masking of errors.

More details about the fault-injection methodology may be found in our recent paper on this topic [42].

F. Resilience Characterization Results

We choose twelve programs drawn from three different benchmark suites (Parboil [43], Rodinia [44] and the CUDA SDK [45]) to evaluate our technique. These applications consist of a total of 15 kernels. For each kernel, we perform sufficient fault injection experiments (typically thousands) to obtain 95% confidence, with an error bar of 1-2%.

Hang rates are less than 1% across all benchmarks. The crash rates vary from 6% to 71 %, while SDC rates vary from 0.5 % to 38 %, across benchmarks. More details about these experiments may be found in our paper [42].

The variations in both crash and SDC rates are higher than what is typically seen for CPU-based applications [30][31], suggesting that application-specific resilience techniques are extremely important for protecting GPGPU applications. We are investigating such techniques as part of our ongoing work.

V. CONCLUSIONS

GPGPUs are already used in several domains, and their usage is still expected to grow in the next years, covering several domains including some ones, where reliability is a key issue. Hence, data about the reliability of GPGPUs and of applications running on them are crucial, as well as techniques for effectively assessing the resilience of related code.

This paper reported an overview of some recent results in the area, with special emphasis on the usage of GPGPUs in HPC centers, on the evaluation of their sensitiveness to radiation, and on the most effective techniques for assessing the resilience of applications they run.

The authors wish that this paper and the presentations of the connected embedded tutorial may be useful to trigger further research activities in the field.

REFERENCES

[1] P. Rech, C. Aguiar, R. Ferreira, C. Frost, L. Carro, "Neutron radiation test of graphic processing units", 2012 IEEE 18th International On-Line Testing Symposium (IOLTS), pp. 55 – 60

[2] I.S. Haque, V.S. Pande, "Hard Data on Soft Errors: A Large-Scale Assessment of Real-World Error Rates in GPGPU", 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing (CCGrid), 2010 , pp. 691 – 696

[3] S. Di Carlo, G. Gambardella, I. Martella, P. Prinetto, D. Rolfo, P. Trotta, "Fault mitigation strategies for CUDA GPUs", 2013 IEEE International Test Conference (ITC), pp. 1 - 8

[4] D. Sabena, M. Sonza Reorda, L. Sterpone, P. Rech, L. Carro, "On the evaluation of soft-errors detection techniques for GPGPUs", 2013 8th IEEE International Design & Test Symposium, to be published

[5] P. Rech, C. Aguiar, C. Frost and L. Carro, "Experimental Evaluation of Thread Distribution Effects on Multiple Output Errors in GPUs", IEEE European Test Symposium, pp. 27 - 32, May 2013

[6] H. J. Wunderlich, C. Braun, S. Halder, "Efficacy and Efficiency of Algorithm-Based Fault-Tolerance on GPUs," 2013 IEEE 19th International On-Line Testing Symposium (IOLTS), pp. 240 – 243, July 2013.

[7] B. Fang, J. Wei, K. Pattabiraman, M. Ripeanu, "Evaluating the Error Resilience of GPGPU Applications", 2012 SC Companion: High Performance Computing, Networking Storage and Analysis, pp. 1504

[8] J. Tan, N. Goswami, T. Li, X. Fu, "Analyzing Soft-Error Vulnerability on GPGPU Microarchitecture", 2011 IEEE IISWC, pp. 226 – 235

[9] W.W.L. Fung, I. Sham, G. Yuan, T.M. Aamodt, "Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow", 40th Annual IEEE/ACM International Symposium on Microarchitecture, 2007 (MICRO 2007), pp. 407-420

[10] C. Ding, C. Karlsson, H. Liu, T. Davies, Z. Chen, "Matrix Multiplication on GPUs with On-line Fault Tolerance," 2011 IEEE 9th International Symposium on Parallel and Distributed Processing with Applications (ISPA), pp. 311 – 317

[11] N. DeBardeleben, S. Blanchard, L. Monroe, P. Romero, D. Grunau, C. Idler, C. Wright, "GPU Behavior on a Large HPC Cluster", 6th Workshop on Resiliency in High Performance Computing (Resilience) in Clusters, Clouds, and Grids, 2013

[12] https://developer.nvidia.com/rdp/assets/cuda-accelerated-linpack-linux64

[13] N. Seifert, Z. Xiaowei, and L. W. Massengill, "Impact of Scaling on Soft-Error Rates in Commercial Microprocessors", IEEE Trans. Nucl. Sci, vol. 46, no. 6, pp. 3100, 2002, 3106

[14] H.T. Nguyen, Y. Yagil, N. Seifert, and M. Reitsma, "Chip-level Soft Error Estimation Method", IEEE Trans. Device and Materials Reliability, vol. 5, no. 3, 2005, pp. 356, 381

[15] D. B. Kirk, W.W. Hwo, "Programming Massively Parallel Processors", MK Publishers, 2010.

[16] P. Rech, C. Aguiar, C. Frost, and L. Carro, "An Efficient and Experimentally Tuned Software-Based Hardening Strategy for Matrix Multiplication on GPUs", IEEE Trans. Nucl. Sci, 2013

[17] P. Rech, L. Pilla, F. Silvestri, C. Frost, M. Sonza Reorda, P. Navaux, and L. Carro, "Neutron Sensitivity and Hardening Strategies for Fast Fourier Transform on GPUs", in proceeding IEEE RADECS 2013, Oxford, UK

[18] M. Violante, L. Sterpone, A. Manuzzato, S. Gerardin, P. Rech, M. Bagatin, A. Paccagnella, C. Andreani, G. Gorini, A. Pietropaolo, G. Cargarilli, S. Pontarelli, and C. Frost, "A New Hardware/Software Platform and a New 1/E Neutron Source for Soft Error Studies: Testing FPGAs at the ISIS Facility", IEEE Trans. on Nucl. Sci., vol. 54, no. 4, August 2007, pp. 1184-1189

[19] JEDEC Standard JESD89A, 2006

[20] D. Bailey, et al., "The NAS Parallel Benchmarks", RNR Technical Report RNR-94-007, March 1994

[21] T. G. Stockham, "High-Speed Convolution and Correlation", Proceedings of the Spring Joint Computer Conference, 1966, pp. 229-233

[22] NVIDIA Tesla K20 GPU Datasheet

[23] M.D. Lerner, "Algorithm Based Fault Tolerance in Massively Parallel Systems", Columbia University, 1988

[24] K.H. Huang and J.A. Abraham, "Algorithm-Based Fault Tolerance for Matrix Operations", IEEE Trans. on Computers, vol. c-33, no. 6, June 1984, pp. 518-528

[25] J.-Y. Jou, J.A. Abraham, "Fault-Tolerant FFT Networks", IEEE Trans. on Computers, Vol. 37, No. 5, May 1988, pp. 548-561

[26] 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

[27] M.-C. Hsueh, T. Tsai, and R. Iyer, "Fault injection techniques and

tools," Computer, vol. 30, no. 4, pp. 75–82, 1997

[28] Thomas Y. Yeh, Glenn Reinman, Sanjay J. Patel, and Petros Faloutsos. 2009. Fool me twice: Exploring and exploiting error tolerance in physics-based animation. ACM Trans. Graph. 29, 1, Article 5 (December 2009), 11 pages

[29] 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

[30] 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

[31] 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

[32] 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 Computer Performance and Dependability Symposium, 2000. IPDS 2000. Proceedings. IEEE International, 2000, pp. 91 –100

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

[34] 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

[35] 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

[36] 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

[37] S. Mukherjee, C. Weaver, J. Emer, S. Reinhardt, and T. Austin, "Measuring architectural vulnerability factors," IEEE Micro, vol. 23, no. 6, 2003, pp. 70–75

[38] 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

[39] 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

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

[41] A. Bakhoda, G. Yuan, W. Fung, H. Wong, and T. Aamodt, "Analyzing cuda workloads using a detailed gpu simulator," in IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS) 2009. pp. 163–174

[42] B. Fang, K. Pattabiraman, M. Ripeanu and S. Gurumurthi, "GPU-Qin: A Methodology for Evaluating the Error Resilience of GPGPU Applications", to appear in the IEEE International Symposium on. Performance Analysis of Systems and Software (ISPASS), March 2014

[43] 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 scientic and commercial throughput computing", in *IMPACT Technical Report*, 2012.

[44] 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. Washington, DC, USA: IEEE Computer Society, 2009, pp. 44–54. [Online]. Available: http://dx.doi.org/10.1109/IISWC.2009.5306797

[45] NVIDIA CUDA SDK. https://developer.nvidia.com/gpu-computing-sdk