Fault Injection in GPGPU Cores to Validate and Debug Robust Parallel Applications

M. De Carvalho, D. Sabena,
M. Sonza Reorda, L. Sterpone
Politecnico di Torino
Torino (TO), Italy

P. Rech, L. Carro
UFRGS
Porto Alegre, RS, Brazil

***Abstract* — Nowadays, General Purpose Graphic Processing Units (GPGPUs) have become popular for processing parallel data more efficiently than CPUs in several domains, from desktop computing to embedded and High Performance Computing (HPC) applications. Unfortunately, GPGPUs have shown to be rather sensible to radiation. Hence, several software mitigation techniques, as well as robust algorithms, are being developed to overcome reliability problems.**

In this paper we propose a fault injection mechanism to evaluate the resiliency of an application running on a GPGPU and to validate software hardening techniques it possibly embeds. The proposed approach is based on the usage of existing software debug facilities and is not invasive to the application. Moreover, the developed fault injector has a low application cost, injection time is rather fast, and it uses the actual GPGPU hardware instead of using models. We report some experimental results gathered on selected cases of study to show the proposed approach advantages and limitations.

Keywords— fault injection, reliable GPGPU applications, robust algorithms, transient faults

I. INTRODUCTION

General Purpose Graphic Processing Units (GPGPUs) have become popular in the past years to deliver high end performance in many different areas, ranging from gaming to HPC [1][2]. Additionally, GPGPUs are increasingly employed also in some safety-critical embedded domains, such as automotive, avionics, space, and biomedical. As an example, the *Advanced Driver Assistance Systems* (ADASs), which are increasingly common in cars, make an extensive usage of the images (or radar signals) coming from external cameras and sensors to detect possible obstacles requiring the automatic intervention of the breaking system [3].

One of the key issues about GPGPUs is their reliability. Depending on the environment and also on the GPGPU complexity, faults can occur due, for example, to the incidence of radiation particles, leading to silent faults or functional interruption. GPGPUs reliability issue can be unacceptable when dealing with safety-critical applications (e.g., in aerospace and automotive) or when a large number of GPGPUs is used for distributed computation (e.g., in HPC centers).

To increase the reliability of their products, GPGPU producers have provided novel GPGPUs with an ECC mechanism able to correct single errors and detect double errors in the available memory components. Nevertheless, ECC is not a definite solution for GPGPUs reliability. Radiation particles can still affect the operation of a logic gate leading to wrong results. Moreover, the corruption of critical resources like the scheduler, which is not protected by the ECC, may lead to multiple output errors [4][5][6]. As demonstrated in [4], even several errors still appear at the output even when the ECC protection is turned on. Moreover, the activation of the ECC mechanisms may results in significant performance degradation, which may not be acceptable in some cases.

The novel necessity to design reliable applications for GPGPUs has raised interest in the research community that has been putting efforts to produce mitigation techniques and more robust algorithms [7][8][9][10]. Fault Injection [11] is a common solution to validate the final GPGPU application code and check its detection or correction capabilities,. Fault Injection can be performed either exposing the GPGPU to accelerated particle beams, or by resorting to software methods. In this work, we propose an efficient fault injector for GPGPUS that does not require any information about GPGPU internal model, since this information is only viable for GPGPU producers. Fault injection solutions with low cost and invasiveness to GPGPU applications are clearly difficult to devise, even because the GPGPU companies do not make available the test access infrastructures (e.g., the IEEE 1149 one). Despite the confidentially of the GPGPU architecture, some reduced intrusiveness techniques that allow injecting faults in GPGPUs have been developed [12][13]. In [12], the authors modify the source code to inject errors; clearly, the effects due to these faults are not the same of those generated by a real fault occurred in the GPGPU modules during the normal computation. In [13] the authors propose a fault injector based on software debugger. Substantially, they inject bit flips in random variables at random instants to evaluate the impact on non-fault tolerant algorithms.

In this paper, we propose a novel fault injection platform based on the NVIDIA CUDA-gdb, which is able to automatically generate a fault list from a GPGPU application, to inject transient faults in several accessible memory components, to mimic faults affecting ALUs, FPUs, and L1 cache errors, and to classify their effects. The advantage of the proposed solution is the ability to inject faults in a real GPGPU hardware without being invasive to the application

(i.e., without modifying the source code). With respect to[13], the tool is able to automatically generate the fault list, inject faults, and classify their effects.

To demonstrate the effectiveness and discuss the limitations of our approach we report some experimental results obtained using the proposed fault injection method on a commercial GPGPU device.

The rest of the paper is organized as follows: in section II we give some background concepts about the GPGPU architecture, the effects of injected faults as observed through radiation tests, and the fault injection techniques for GPGPUs which have been previously proposed in the literature. In section III, we describe the proposed approach in details. In section IV we evaluate the proposed fault injection method by reporting experimental results gathered on some benchmark applications. Finally, section V draws some conclusions.

II. BACKGROUND

GPGPU architectures raised a lot of interest in the recent years due to their high performance in parallel computation for delivering fast results in many different areas ranging from gaming to biomedical. Some of the GPGPU application areas (e.g., the automotive and aerospace ones) have safety constraints; moreover, the usage of GPGPUs in HPC also raises concerns about their reliability [14]. In order to assess the reliability of resilient applications and to validate the resiliency mechanisms they embed it is necessary first to analyze the GPGPU architecture.

In this section, we will introduce first the typical NVIDIA GPGPU architecture, and then we will outline some of the available solutions to increase the resiliency of GPGPU applications; moreover, we will overview the types of errors that may or may not produce misbehaviors, and the main fault injection methods proposed so far.

a) NVIDIA GPGPU architecture

The typical GPGPU architecture differs from the CPU one because it includes many serial multiprocessors (SMXs) to increase parallelism. In figure 1, a simplified GPGPU architecture, based on NVIDIA Fermi architecture, is presented. The Fermi GPGPU has an interface for communicating with the host processor (i.e., the CPU), global memory that store the data to be elaborated, a gigathread module for assigning instructions to the SMXs, several SMXs where the computation takes place, and an L2 cache.

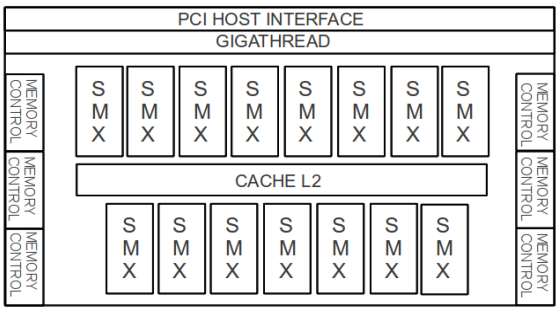

Figure 1. A Simplified GPGPU Architecture

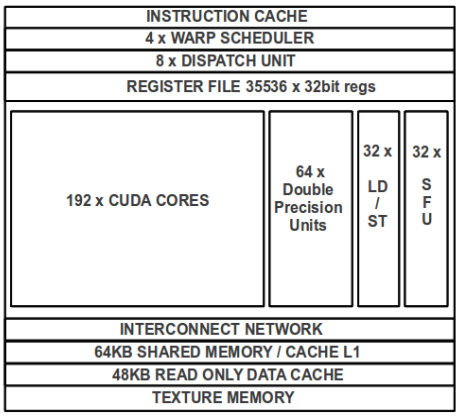

Figure 2. A Simplified SMX Architecture

In figure 2, a simplified SMX architecture is presented. A SMX is typically able to execute several threads in parallel. The number of parallel processes executable in parallel depends on the SMX capabilities and ranges from 32 to 192. In the specific sample case considered in this figure, it contains an instruction cache, a warp scheduler, a dispatch unit, a register file, 192 cores, 64 double precision units, 32 load store units, 32 special function units, a L1 cache, an interconnect network, and a texture memory.

b) Fault effects in GPGPU applications

There are several works (e.g., [4][5][6]) studying the effect of faults in GPGPUs caused by radiation. Also, process variations, aging, high temperatures, high current peaks, and low voltages may help jeopardize electronic devices, like GPGPUs, and may produce critical misbehaviors depending on the application.

In table I, we listed the possible effects of errors provoked by injecting a fault in the GPGPU circuit and some explanation.

Error Classification	Explanation
No effect	The injected fault is masked, for example because the target variable is overwritten by another value before being read
Data Error	The injected fault affects some logic or memory elements and produce a wrong data in the output result
Timeout Error	The injected fault effects affect a control flow variable, thus forcing the program to enter an infinite loop; The fault can also cause the executing thread to finish earlier
Exception Error	The injected fault triggers an exception; the program returns without providing any results
Fault Tolerance Error (Error detection)	The injected fault is detected by some fault tolerance mechanism

Table I. classification of faults affecting a GPGPU

Validating GPGPU applications reliability can be performed exposing the GPGPU to a controlled radiation beam. However, radiation tests, even is fast and effective, are is very expensive. Moreover, radiation test does not allow to

fully understand where faults arise and which are the propagation mechanisms they activate. Therefore, alternative solutions based on fault injection are required.

c) *Fault Injection in GPGPUs*

In the hardware design industry there are mature methodologies and tools to test the architecture behavior by simulating internal components. However, this is only possible when a detailed model of the addressed architecture is available. In this case, faults can be injected at any given time in any precise location of the architecture, giving the possibility to analyze their effects and validate the application.

Techniques for fault injection in GPGPUs have been proposed in several works in different ways. In [12] the authors propose a method for injecting faults by modifying the source code of the application. In [15] the authors propose the usage of a statistical fault injection tool based on a GPGPU simulator to analyze the architectural vulnerability when executing several parallel algorithms. However, in this case the test is not performed on the real hardware and the accuracy of the results depends on how strong the correlation is between the simulator and the real architecture. Lastly, in [13] a fault injection mechanism based on the software debugger (CUDA-gdb) [16] was proposed. This mechanism is able to modify variables of the executable code at runtime. With this tool the authors evaluated the resiliency of some parallel applications by applying random transient errors in random variables.

The works discussed above show that the fault injection tools for GPGPUs have the following advantages with respect to radiation test:

- More accurate and precise results
- Faster fault injection setup
- No risk of damaging the GPGPU
- Reduced cost.

On the other side, fault injection techniques have some drawbacks:

- They may be too slow to gather enough results to statistically assess the final reliability
- They may have difficulties in accessing some elements within the GPGPU
- They may require some modifications in the application code, thus providing results on a system different than the target one.

III. PROPOSED APPROACH

In this work we propose a GPGPU fault injector tool that forces bit flips in variables of GPGPU applications and observes their effects. The method exploits the CUDA-gdb software debugger, which provides the visibility of all runtime GPGPU variables, giving the possibility to inject faults at specific memory elements of the architecture.

In order to fully automate the fault injection campaign, the different phases typically implemented in a traditional fault injection experiments (fault list generation, fault injection, and result analysis) are all demanded to an ad-hoc tool which interacts when necessary with the debugger via a proper command file. This guarantees a minimal effort to perform a GPU fault injection campaign and ensures that the developed tool is code independent.-

More in particular, the steps performed by the proposed fault injection method are the following:

- a CUDA code parser elaborates profiling information about the GPGPU variables and, based on the user inputs, generates a list of faults to be injected.
- The generated list and the profile information are used to create CUDA-gdb commands that will be executed while the program is running.
- The application is executed without any fault injection, thus recording its behavior (*fault-free* or *gold execution*).
- The application is executed with one single fault injected at a time.
- The values of some observable variables (selected by the user) are analyzed to check whether the fault-free and the faulty results match.

The architecture of the platform implementing the proposed approach is shown in figure 3.

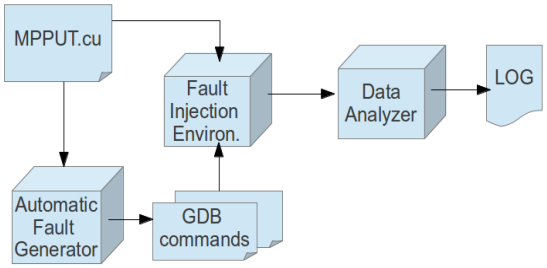

Figure 3. Architecture of the proposed fault injector

The proposed fault injector is composed of three blocks; the *automatic fault generator*, the *fault injection environment*, and the *data analyzer*.

The first module reads the program code and elaborates profiling information about the variables in which faults have to be injected. The automatic fault generator then translates each fault event (i.e., a bit flip in a selected variable) into CUDA-gdb commands. The fault injection environment manages the execution of the executable code and performs the fault injection experiment. The data analyzer compares bitwise the results acquired during the previous execution in terms of values of the output observable variables in the faulty-free and faulty executions.

The proposed methodology may increase slightly the simulation time, because the application needs to be compiled with debugging symbols needed for debugging. Also, execution time increases because the program needs to be halted at a specific point (e.g., when the program counter reaches a specific value) to force a bit flip in the target variable. Finally, the execution time is increased due to file access operations, because CUDA-gdb dumps the output values to a file to allow the comparison of faulty and fault-free results.

In some GPGPU architectures there are ECC mechanisms protecting memory components from transient and permanent faults originated by radiation, high temperature, voltage drops, and different kinds of stresses. On the other hand, other GPGPU components (e.g., FPUs and ALUs) may also be affected by faults, whose result is to have their operations corrupted, thus producing erroneous values that may sneak into the memory component. In this case, the ECC mechanisms will not provide any error correction.

The proposed fault injector has the ability to produce bit flips into runtime memory elements (global, local and shared memories) visible by the programmer to mimic transient and permanent faults in logic elements (ALUs and FPUs), memory elements (L1 caches and shared memory), scheduler (execution flow and thread ID errors), and kernel launch (exception generation). The bit flip operation differs for each data type (e.g., Integer and Floating point representations) as described in the following subsections.

A bit flip, resulting from the corruption of memory elements or reflecting a wrong computation, can be introduced in an integer number by simply using a single XOR operation between the variable and the user-defined bitmask, thus implementing the desired bit flip, as shown in formula 1.

(1)

where *op* is the variable affected by the bit flip.

Bit flip operations in floating point numbers

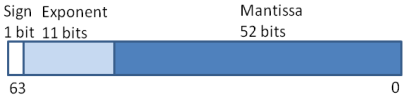

Figure 4: Representation of the 64-bit double floating point number

In FPUs the floating point variables comply with the IEEE754 standard, and contain three parts: sign, exponent and mantissa. Figure 4 shows the representation of a 64bit floating point format. A 64-bit floating point number *a* is considered in scientific notation as:

(2)

where *m* is the mantissa, is the magnitude and *b* is the exponent.

For each of the three parts of the floating point format there is a mathematical model to implement the bit flip operation, as described in [17]. Considering *a* in equation 2, the perturbed floating point number with a bit flip is expressed as:

Equation 3: Bit flip operations on 64 bit floating point numbers

a) Fault Injection Environment

In figure 5, the proposed fault injection environment is presented; it is composed of 4 main steps. Initially (step 1) the Parallel Program Under Test (PPUT) is compiled with debugging capabilities for the target GPGPU: the obtained executable file contains the executable code plus the symbol table useful to debug the code itself. This GPGPU executable code is then manipulated by the cuda-gdb (step 2) according to a set of gdb commands previously defined, which in turn allows visualizing and modifying GPGPU runtime variables (step 3). Finally, in step 4, the produced results are analyzed.

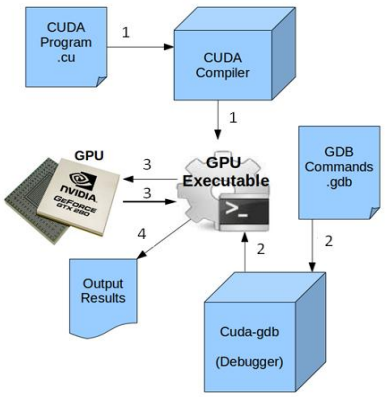

Figure 5. Fault Injection Environment

The automatic fault list generator is a module that parses the GPGPU code and generates a look-up table to generate suitable CUDA-gdb commands triggering the fault simulation.

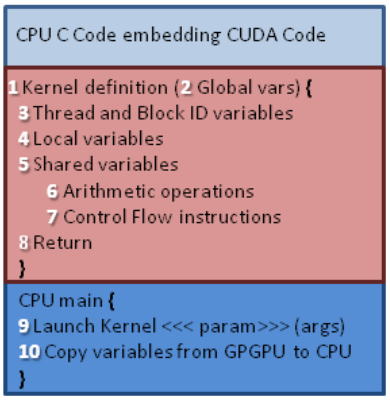

Figure 6. C code embedding a CUDA code and the possible fault injection points in the GPGPU application.

A simple CUDA code (GPGPU in red) embedded in a C code scheme (CPU in blue) is presented in figure 6 to demonstrate how the parser selects the variables where to inject faults. The parser is able to identify the fault injection points (numbers 2 to 8) which are the only the GPGPU variables accessible at runtime by CUDA-gdb as well as the number of blocks and threads involved in the GPGPU computation. By identifying these variables, the user is able to inject a single bit flip in one variable at a time per thread and study their effects on the final result.

These variables are inserted in the fault list table which is organized to mimic a fault injection in the application. In this table there are some useful information needed to create the gdb commands to implement fault injection.

The table elaborated by the tool contains the data shown in table II, their corresponding numbers in the code depicted in figure 6, and their given name abbreviation that will be later used to show how they are translated in cuda-gdb code.

	Corresponding no. in figure 6	Abbreviation
Kernel definition	1	KD
Target Variables where to inject a fault	2-5,8	VF
Where the target variable can be modified	Line number of 6-8	LTF
Bitmask used for the bit flip operation (defined by the user)	6-8	BM
How many executed lines containing the target variable before a fault is injected (user defined)	# of executed lines containing 6-8	#LH
Kernel Launch line number in the CPU code	9	KL
Output variables to detect errors	10	OBSV
Instant to check the output variable	Line number + 1 of 10	OBSL

Table II. List of the parsed information needed to inject a fault

The fault injection technique is only possible due to the CUDA-gdb debugger provided by NVIDIA. Although it is very powerful for debugging parallel software it is somewhat limited for our purposes. One of the limitations found is the inability to modify a variable at a random time even if located in the current context. In some cases, the bit flip operation cannot be performed because the thread that is being executed has already finished. Although the variable still exists in the GPUGPU it is not accessible by CUDA-gdb which is not allowed to modify it. . The second major problem is that watch points are not supported. However, by inserting strategic breakpoints for modifying the target variable and observing output variables, we were able to obtain the desired functioning.

In order to inject faults using the debugger, the fault injector translates the information parsed shown in table II into the CUDA-gdb code shown in code 1. Note that in line 11, where the bit flip operation takes place, the CUDA-gdb commands depend on the variable type, as explained in subsection 3b. Also, line 12 is optional and allows mimicking the effect of a transient fault. The corrupted variable (VF) is used in a CUDA instruction in the GPGPU application code and the contents with the fault is propagated to another destination variable. Then, at the next instruction of the application code, the fault is removed from the target variable which originated the fault.

The data analyzer is a module that compares the results obtained by the fault-free and the faulty executions. It makes a bitwise comparison showing the disparities observed in the output variables where we observe the error effects.

```
1.   start
2.   break KL
3.   command 2
4.   break LF
5.   break OBSL
6.   ignore 3 #LH
7.   delete 2
8.   continue
9.   end
10.  command 3
11.  Bitflip operation -- VF=VF^BM
(12). Step; Reverse bitflip VF=VF^BM (Optional)
13.  delete 3
14.  continue
15.  end
16.  command 4
17.  set logging on KD.txt
18.  print/t OBSV
19.  set logging off
20.  delete 4
21.  continue
22.  end
23.  continue
24.  quit
```

Code 1: CUDA-gdb command implementing the fault injection

IV. CASES STUDIES AND RESULTS

In order to evaluate the effectiveness of the approach we use 2 case studies with (*robust*) and without (*plain*) software hardening:

a) Parallel Matrix Multiplication
b) Parallel Fast Fourier Transform.

To demonstrate the features of the proposed fault injector, we inject a single bit flip in each kernel variable in each accessible code line for only one thread. Although the tool is able to consider all threads, we choose to corrupt only one thread to make the experiments simple. Then, we analyze the impact of each injected fault on the final result by evaluating the disparity between the faulty and the fault-free executions in terms of CUDA-gdb execution time and number of errors classified according to Table I. We carried out the experiments in an NVIDIA GPU GeForce 9600M GS containing one serial multiprocessor including 32 cuda cores, 512MB of memory, and 100MHz clock frequency.

a) Parallel Matrix Multiplication

The first case study is a set of parallel matrix multiplication algorithms that use the duplication with comparison technique. The algorithms, proposed in [10], use thread and time redundancy. Moreover, each of these algorithms is implemented with data input duplication (*full*) or without data input duplication (*partial*). Table III shows the results of this case study in its different versions. It presents the number of errors classified according to Table I of the fault-free and faulty executions. The fault-free version (plain) requires 12 ms to complete its computations, whereas the average faulty execution lasts for about 2 seconds.

In the plain version, the most "dynamic" injected fault produced 140 data errors in the output variables. The robust versions were able to detect the errors and reproduce the execution, thus providing the correct results.

	Injected Faults	No Effect	Data Error	Exception	Timeout
Plain	16	4	7	3	2
Full Thread	26	16	0	6	4
Partial Thread	21	15	0	4	2
Full Time	16	11	0	3	2
Partial Time	16	11	0	3	2

Table III. Matrix Multiplication results

b) Redundant Parallel FFT algorithm

The second case study is an FFT algorithm that uses the GPGPU. It includes a robust control flow technique that allows identifying if the program execution flow is correct or not [4]. If the technique detects the error in the execution flow, it runs the FFT algorithm again until there are no control flow errors. Moreover, the technique embeds error masking by executing several times the same kernel code in each thread. In table IV the results of both the plain and robust implementation of the FFT algorithm are presented. The table shows the total number of injected faults and the number of injected faults producing errors (according to Table I). In the plain version, the most "dynamic" fault propagated the error to 16,383 variables. The number of injected faults producing data errors is 49 for the plain version and 0 for the robust version, because they were masked or corrected by the fault tolerant techniques employed in the algorithm. The robust version, however, triggered more exception errors, because the control flow variables are vectors, whose indexes have been corrupted by the fault injection, generating exceptions. The average execution time increased from 45 ms (fault-free) to approximately 4s (faulty), due to the large amount of data to analyze.

	Injected Faults	No Effect	Data Error	Exception	Timeout	Fault Tolerant
Plain	74	16	49	6	3	-
Robust	137	82	0	24	3	28

Table IV. FFT results

V. CONCLUSIONS

In this work we presented a fault injection tool based on CUDA-gdb which is able to inject transient faults in GPGPU devices and can be used to validate parallel applications. Moreover, the proposed tool was able to mimic transient faults by applying bit flip operations to corrupt the operation of ALUs, FPUs, and memory elements, thus allowing the evaluation of performance and robustness of parallel applications. The tool is non-invasive to the application and is fully automated to find variables and instructions in the parallel code where a fault can be injected. Faults were injected in two case studies resorting to plain and hardened software. The tool is being improved to mimic L1 cache and scheduling errors, and to support the possibility to inject faults on a probabilistic basis.

VI. REFERENCES

[1] Zhe Fan et al., "GPU Cluster for High Performance Computing", in Proceedings of SC '04. IEEE Computer Society, Washington, DC, USA. 2004.

[2] D. Luebke, "CUDA: Scalable parallel programming for high performance scientific computing", 5th IEEE International Symposium on Biomedical Imaging: From Nano to Macro, 2008.

[3] B. Ranft, T. Schoenwald, B. Kitt, "Parallel Matching-based Estimation – a Case Study on Three Different Hardware Architectures", 2011 IEEE Intelligent Vehicles Symposium (IV), pp. 1060-1067.

[4] L. Pilla, P. Rech, F. Silvestri, C. Frost, P. O. A. Navaus, M. Sonza Reorda, and L. Carro, "Software-Based Hardening Strategies for Neutron Sensitive FFT Algorithms on GPUs", IEEE Trans. Nucl. Sci, 2014, Vol. PP, pp. 1-7.

[5] P. Rech, G. L. Nazar, C. Frost, L. Carro, "GPUs Reliability Dependence on Degree of Parallelism", 2013 IEEE Radiation and its Effects on Components and Systems (RADECS), 2013

[6] P. Rech, T. D. Fairbanks, H. M. Quinn, L. Carro, "Threads Distribution Effects on Graphics Processing Units Neutron Sensitivity", 2014 IEEE Nuclear and Space Radiation Effects Conference (NSREC), 2013.

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

[8] S. Di Carlo, G. Gambardella, M. Indaco, I. Martella, P. Prinetto, D. Rolfo, P. Trotta, "A Software-Based Self Test of CUDA Fermi GPUs"; IEEE European Test Symposium, pp. 33 – 38, May 2013.

[9] S. Tselonis, V. Dimitsas, D. Gizopoulos, "The Functional and Performance Tolerance of GPUs to Permanent Faults in Registers," 2013 IEEE 19th International On-Line Testing Symposium (IOLTS), pp. 236 – 239, July 2013.

[10] D. Sabena, L. Sterpone, M. Sonza Reorda, P. Rech, L. Carro, "On the Evaluation of Soft-Error techniques for GPGPUs", 8th IEEE International Design and Test Symposium, December 2013.

[11] Fault Injection Techniques and Tools for Embedded Systems Reliability Evaluation, Edited by A. Benso and P. Prinetto, Kluwer, 2003.

[12] J. Tan, N. Goswami, T. Li, X. Fu, "Analyzing Soft-Error Vulnerability on GPGPU Microarchitecture", 2011 IEEE International Symposium on Workload Characterization (IISWC), 2011.

[13] B. Fang, J. Wei, K. Pattabiraman, M. Ripeanu, "Towards Building Error Resilient GPGPU Applications", 2012 SC Companion: High Performance Computing, Networking Storage and Analysis

[14] L. Bautista Gomez, F. Cappello, L. Carro, N. DeBardeleben, B. Fang, S. Gurumurthi, K. Pattabiraman, P. Rech, M. Sonza Reorda, "GPGPUs: How to Combine High Computational Power with High Reliability", Design, Automation & Test in Europe Conference & Exhibition (DATE), 2014

[15] N. Farazmand, R. Ubal, D. Kaeli, "Statistical Fault Injection-Based AVF Analysis of a GPU Architecure", IEEE Workshop on Silicon Errors in Logic - System Effects (SELSE), 2012.

[16] http://docs.nvidia.com/cuda/cuda-gdb

[17] J. Elliott, F. Mueller, M. Stoyanov, C. Webster, "Quantifying the Impact of Single Bit Flips on Floating Point Arithmetic", http://moss.csc.ncsu.edu/~mueller/ftp/pub/mueller/papers/TR-2013-2.pdf

[18] Keun Soo Yim et al. Hauberk, "Lightweight Silent Data Corruption Error Detector for GPGPU", 2011 IEEE International Parallel & Distributed Processing Symposium (IPDPS)