[Proceeding] A software-based self test of CUDA Fermi GPUs

A software-based self test of CUDA Fermi GPUs

Authors: Di Carlo, S.; Gambardella, G. ; Indaco, M. ; Martella, I. ; Prinetto, P. ; Rolfo, D. ; Trotta, P..

Published in the Proceedings of the IEEE 18th European Test Symposium

DOI: 10.1109/ETS.2013.6569353

A Software-Based Self Test of CUDA Fermi GPUs

Stefano Di Carlo*, Giulio Gambardella*, Marco Indaco*, Ippazio Martella†, Paolo Prinetto*,
Daniele Rolfo*, Pascal Trotta*

Politecnico di Torino
Dipartimento di Automatica e Informatica
Corso Duca degli Abruzzi 24, I-10129, Torino, Italy
*Email: {*FirstName.FamilyName*}@polito.it
† Email: {*FirstName.FamilyName*}@gmail.com

***Abstract*—Nowadays, Graphical Processing Units (GPUs) have become increasingly popular due to their high computational power and low prices. This makes them particularly suitable for high-performance computing applications, like data elaboration and financial computation. In these fields, high efficient test methodologies are mandatory. One of the most effective ways to detect and localize hardware faults in GPUs is a Software-Based-Self-Test methodology (SBST). In this paper a fully comprehensive SBST and fault localization methodology for GPUs is presented. This novel approach exploits different custom test strategies for each component inside the GPU architecture. Such strategies guarantee both permanent fault detection and accurate fault localization.**

I. Introduction

In the last years, the continuous demand for computational power resulted in an increasing number of cores integrated in a single chip. In this scenario, Graphic Processing Units (GPUs) have replaced multi-core processors in several High Performance Computing (HPC) applications. Modern GPUs support hundreds of cores and they are inherently designed to perform parallel operations. Moreover, dedicated programming instruments for developing GPU ready applications enable programmers to exploit this huge computational power also for the solution of general-purpose computing problems [1][2][3][4][5]. Reliability is still a big issue for GPU cores. While in their original application domain (i.e., video processing) wrong pixels caused by either soft or hard errors have a negligible effect on the user experience, when GPUs are exploited in HPC applications such as financial or scientific computations, correctness and high dependability become a primary requirements. In this context, Software-Based-Self-Test (SBST) represents a promising test solution already exploited in several single-core processor architectures [6][7][8][9]. SBST techniques exploit the microprocessor Instruction Set Architecture (ISA) to generate instruction sequences able to test a wide range of hardware modules, without introducing any hardware modification, and thus stressing the system in its actual operational condition. One of the main drawbacks of SBST, when applied to black box modules such as GPUs whose internal architecture is usually hidden and not available to the systems designer, is a precise evaluation of the obtained fault coverage. This requires a careful design and selection of the type of functional test applied to the core. Techniques such as *Duplication-based* [10][11][12], *Check pointing-based* [13], *Algorithm based* [14], *Static compiler analysis-based* [15] and *Dynamic profiling-based* [16] have been somehow applied to detect errors in specific GPU applications. However, they usually introduce high performance overhead or require custom modifications to be adapted to the highly parallelized GPU architecture. To the best of our knowledge, no comprehensive and effective SBST methodology, targeting GPUs, has been proposed so far. This paper tries to cover this gap, introducing an SBST and fault localization methodology suitable for the last generation of CUDA *Fermi* GPUs. It guarantees fine-grained fault detection exploiting different custom test strategies and an accurate localization of the faulty streaming multiprocessor unit. The paper is organized as follow: in Section II the CUDA architecture overview is provided, in Sections III, IV and V the proposed methodology and test procedures are shown, while in VI the experimental results are depicted. Finally, some conclusions are drawn.

II. CUDA Overview

Compute Unified Device Architecture (CUDA) is a parallel computing architecture developed by nVidia for graphic processing, used as computing engine in nVidia GPUs. nVidia supports programmers by releasing the CUDA Toolkit, which embeds the Software Development Kit (SDK), a comprehensive software development environment. Among the tools provided by the CUDA Toolkit, the CUDA Visual Profiler (*cudaprof*) provides the user with feedbacks for code optimization. Furthermore, the CUDA Occupancy Calculator helps to set-up the execution on the GPU in order to achieve high occupancy of internal resources. nVidia also released a software architecture that enables CUDA-based GPUs to execute programs written in C, C++, Fortran, OpenCL, DirectCompute, and other languages [17]. In general, a CUDA program is a set of parallel kernels (i.e., blocks of code executing a given function on the GPU) organized by the compiler into threads, thread blocks, and grids of thread blocks. A *thread block* is a set of concurrently executing threads. Each thread in a thread block executes an instance of the kernel. It is assigned with a thread identifier (thread ID) within its thread block, a program counter, a set of registers, per-thread private memory, inputs, and output results [17]. A *grid* is an array of thread

blocks that execute the same kernel. The CPU provides in input to the GPU the grid in order to start the kernel execution. At the end of the execution, the CPU can flush the GPU global memory to acquire output data. The internal architecture of a CUDA GPU comprises: (1) a *Block dispatcher* that schedules the input grid by assigning each thread block to the internal logic; (2) a *Global Memory* to store data and final results during the kernel execution; (3) a *Shared Cache* to speed up read/write operations on the global memory; and (4) several *Streaming Multiprocessors* (SM), representing the main computational units in charge of executing the scheduled thread blocks. Fig. 1 reports the internal architecture of a SM. The *Thread Dispatcher*, dispatches each thread of a thread block to one of several computational cores. These cores include: *CUDA cores* equipped with a fully-pipelined *Integer Unit* (IU) and a fully pipelined *Floating Point Unit* (FPU), several *Special Function Units* (SFU) able to execute transcendental instructions (i.e., sine, cosine, inverse square root, etc.), a *Shared Memory* shared among threads inside a thread block and several *Load/Store Units* (LD/ST) managing read/write operations on the shared memory.

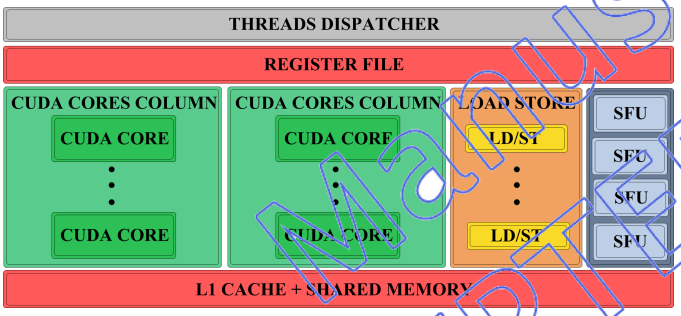

Fig. 1: SM internal structure

The *Compute Capability* (CC) is the main parameter characterizing a CUDA GPU. The *Fermi* architectures, considered in this paper, are characterized by CC equal to 2.0 or 2.1. In *Fermi* architecture with CC 2.0, each SM contains 32 CUDA cores (2 columns of 16 cores). Every CUDA core is able to execute both double and single precision integer and floating point instructions. In CUDA GPUs with CC 2.1 each SM contains 48 CUDA cores (3 columns of 16 cores), with just the third column able to execute double-precision instructions. Moreover, cores on the third column can execute dual-issued single precision instructions. All remaining parts of the SM are the same in both architectures and include 16 Load/Store Units and 4 Special Function Units [17].

III. Our Methodology

The proposed SBST methodology targets each SM inside a *Fermi* GPU, providing both permanent fault detection and localization. Fig. 2 introduces the computational flow exploited to test SMs.

First, a set of test kernels, one for each internal component of the SM, has been defined. Some of these kernels (i.e., the ones for testing CUDA cores) exploit well-known SBST

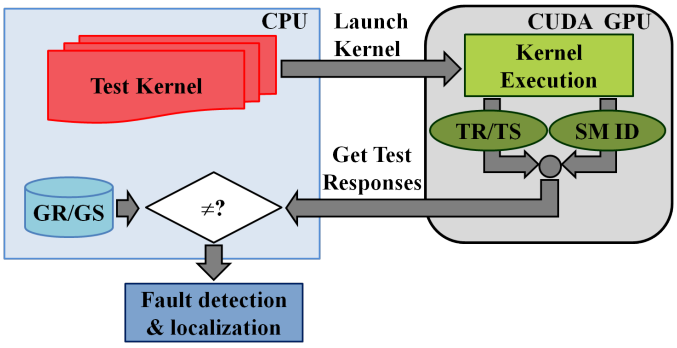

Fig. 2: Basic test approach

procedures adapted to be executed on a GPU. The remaining kernels targeting the *Thread Dispatcher* and the SFUs, instead, implement ad-hoc test procedures. When the test kernel is created, the CPU starts the kernel on the GPU. The GPU executes several parallel instances of the test kernel (one for each internal module of each SM), and, for each instance, it computes the related test results (TR) or test signatures (TS). Each thread appends to the computed TS/TR the identifier of the SM (SM ID) where it has been executed, and this information is sent back to the CPU. The SM ID is computed reading the content of the special register *%smid* [18]. The CPU compares the TR/TS contained in each test response with the precomputed Golden TR (GR)/Golden TS (GS). If a mismatch occurs, a fault is detected and the SM ID is used to localize the faulty SM. In order to execute test procedures on the GPU, every test kernel must be written exploiting the inline-assembly provided by the CUDA-ISA [18]. The inline-assembly makes it possible to precisely define the instructions that the CUDA core must execute, preventing changes introduced by the compiler. Moreover, all *for loops* required by the test procedure must be unrolled exploiting the compiler directive *#pragma unroll* [19]. This directive placed before each *for loop*, prevents the insertion of extra operations at compile time that could alter the test coverage.

Finally, to ensure the complete execution of the test procedure on each internal module of a SM, the test kernel must be executed with the right configuration, taking into account the size of the grid and the size of the thread blocks (see Section II). The way this can be achieved will be deeply analyzed in Section IV.

IV. Test Kernel Configuration

Fig. 3 shows the steps required to compute the optimal test kernel configuration. This task exploits the Visual Profiler and the CUDA Occupancy Calculator (see Section II). The CUDA Occupancy Calculator is able to define the size of the grid (GrS) required to achieve the complete occupation of all resources available in a SM. This computation is based on three parameters, namely: (1) the *Register per Thread* (RpT), (2) the *Shared Memory per Block* (SMpTB) and (3) the *Threads Block Size* (TBS). RpT and SMpTB define respectively the number of registers used by each thread and the amount of shared

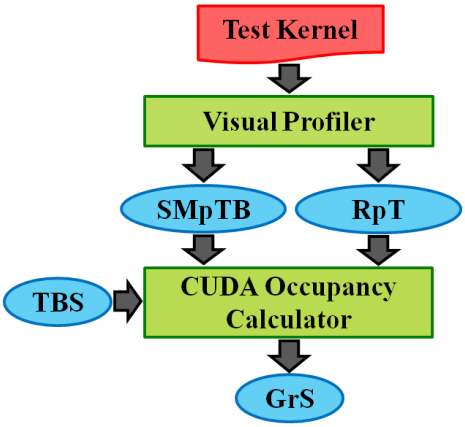

Fig. 3: Distribution methodology flow diagram

memory used by each thread block. These two parameters can be defined by running the kernel inside the Visual Profiler (see Fig. 3).
Providing in input to the CUDA Occupancy Calculator the value of *Register per Thread* and *Shared Memory per Block*, the kernel configuration is computed by testing different values of *Threads Block Size* until the full SM resources are properly allocated.

V. Test Procedures

This Section defines the procedures for testing the modules inside a SM. A particular remark must be done for the *Shared Memory* (see Section II). The test of this component is not taken into account in the proposed work, because in every *Fermi* CUDA GPU this memory is natively protected by a Single-Error Correction Double-Error Detection Code (SECDED ECC) [17]. Any time an uncorrectable error is detected by the ECC, the kernel execution is stopped and an error message is sent to the CPU. Programmers can catch this error and identify the fault on the memory.

A. CUDA Core Test Procedure

As described in Section II, a Floating-Point Unit (FPU) and an Integer Unit (IU) are the building blocks of a CUDA core. SBST of these kind of components has been deeply analyzed in the literature. Among the available solutions, in this paper we exploit those proposed by Paschalis et al. [20] and Xenoulis et al. [21], which can be applied to black box modules and, at the same time, guarantee high coverage. Regardless the implementation details that are available in the cited papers, the overall idea of these two procedures is to execute a sequence of arithmetic/logic operations, involving the unit under test, repeated using several deterministic (in the case of [20]) or pseudo-random (in the case of [21]) input test patterns. Finally, the selected test procedures require to compact the outcomes of all into an output signature.
The way this test procedure can be implemented in a GPU changes between *Fermi* architectures with CC 2.0 and *Fermi* architectures with CC 2.1. In CUDA GPUs with CC 2.0 both the IU and the FPU can perform single and double precision instructions. No modifications are required to the original test procedure to cope with this architecture and the test can follow the flow introduced in Fig. 2. A different situation arises when CUDA GPUs with CC 2.1 are considered. In this architecture only the third column of CUDA cores is able to execute double-precision instructions. Moreover, during the execution of single-precision instructions, this column of cores executes only dual-issued instructions. Therefore, to ensure the complete test execution on every CUDA core, the test procedure for single-precision IU and FPU must involve dual-issue instructions. Dual issue instructions are used whenever the thread dispatcher (see Section II) finds two consecutive independent instructions. Duplicating each instruction of the test procedure is enough to guarantee this condition. Let us consider the following example of instruction duplication where the left column reports the original test sequence and the right column reports the duplicated one:

$$\begin{array}{l} mul(r1,r2,r3) \\ add(r4,r4,r1) \end{array} \Rightarrow \begin{array}{l} mul(r1,r2,r3) \\ mul(r5,r6,r7) \\ add(r4,r4,r1) \\ add(r8,r8,r5) \end{array}$$

To guarantee that two duplicated instructions are independent and therefore dual-issued, it is enough to use disjoint sets of registers. The correct activation of this mechanism can be monitored by comparing the instruction throughput of a test procedure with duplicated instructions, with the corresponding version without duplicated instructions. A throughput increment of about 30% is a good indicator that dual-issued instructions are used and therefore the test procedure is executed also on the third column of CUDA cores. Finally, since double-precision operations are inherently executed only on the third CUDA core column, instructions to test double precision IU and FPU operations do not need to be duplicated.

B. Special Function Unit Test Procedure

SFUs in a CUDA GPU execute a fast approximation of transcendental instructions on 32-bit input floating-point numbers. The supported transcendental operations (*SFU operations*) are: sine, cosine, base-2 logarithm, base-2 exponential, reciprocal, square root and inverse square root.
The way to test these functions still need to be properly investigated. The test procedure proposed in this paper computes each SFU operation for a set of pseudo-random test patterns. Then, the obtained test results (GPU results) are compared with golden results precomputed on the CPU.
The fault-detection cannot be performed with a simple equivalence check, because golden and GPU results are affected by different tolerances. In fact, CPU results are affected by the machine epsilon [22], that depends on the used processor, while GPU results are influenced by the SFU tolerance. Moreover, each SFU operation is affected by a different tolerance and, in the CUDA-ISA User Guide [18], these tolerances are not well documented. To overcome these issues, tolerances of SFU operations must be characterized. A test campaign

performed on different *Fermi* GPUs has highlighted that SFU operations do not provide reasonable results for the entire 32-bit floating-point range. Outside a given input range (*valid input range*), the obtained results are saturated to a fixed value. Valid input ranges for each SFU have been computed exploiting a binary search approach and reported in Table I.

TABLE I: Valid input ranges for SFU operations

SFU operation	Valid input range			
	Negative range		Positive range	
	UB	LB	LB	UB
sine	-	-	1.87E-07	1.57
cosine	-	-	1.31E-06	1.57
base-2 logarithm	-	-	1.17E-38	3.36E+38
base-2 exponential	-1.27E+02	-1.19E-07	1.19E-07	1.28E+02
reciprocal	-8.50E+37	-1.18E-38	1.17E-38	8.50E+37
square root	-	-	1.17E-38	3.36E+38
inverse square root	-	-	1.17E-38	3.36E+38

SFU characterization consists of computing the relative error between results obtained executing the SFU operation on a golden CPU and on a golden GPU, according to (1).

$$\varepsilon_{gpu} = \frac{y_{cpu} - y_{gpu}}{y_{cpu}} \quad (1)$$

where ε_{gpu} is the relative error, y_{gpu} is the result obtained from a golden GPU, and y_{cpu} is the result obtained from the CPU.

The kernel for computing y_{gpu} is written exploiting the inline assembly associated with each SFU operation. In such a way, it is possible to ensure that obtained results are really computed by a SFU. To achieve a high characterization precision, the ε_{gpu} has been computed for each value contained in the valid input range (i.e., exhaustive characterization), leading to an overall execution time of 20 minutes. Since the characterization task must be performed during the design phase only, the required time is totally acceptable.

The exhaustive characterization highlights that ε_{gpu} changes depending on the input provided to the SFU operations. The ε_{gpu} ranges from a minimum ($\varepsilon_{gpu,min}$) to a maximum ($\varepsilon_{gpu,max}$). For each SFU, $\varepsilon_{gpu,min}$ is always zero, while $\varepsilon_{gpu,max}$ have significant variation depending on the input values. Since $\varepsilon_{gpu,max}$ fluctuations are of some order of magnitude, for obtaining a better characterization the following steps are performed: (i) the valid input range of each SFU is split, (ii) each sub-range is characterized with a $\varepsilon_{gpu,max}$, (iii) sub-ranges with similar $\varepsilon_{gpu,max}$ are grouped, and (iv) grouped sub-ranges are characterized with the maximum $\varepsilon_{gpu,max}$ inside the group. This approach ensures a fine-grain characterization resulting in a better precision.

After the characterization task, input test patterns are generated maintaining their values inside input valid ranges reported in Table I. They are defined by generating a set of equally distributed 32-bit floating point numbers inside each valid input range.

Then, CPU golden results (y_{cpu}) and GPU results are computed for each SFU operation and for each input test pattern. For each couple of CPU and GPU results, ε_{gpu} is computed. Fault detection is performed exploiting a well known approach used in Oscillator-Based Test (OBT) [23] methodologies (see Fig.4).

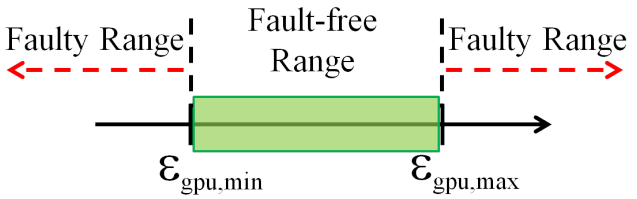

Fig. 4: Fault detection strategy for SFUs

Basically, the approach assumes as fault-free all results contained in the ε_{gpu} range, defined by the characterization task, and faulty all others. Obviously, the proposed approach is not able to detect faults that alter a result without bringing it outside the ε_{gpu} range. However, from a functional point of view, these faults introduce an error on the results totally comparable with the one natively introduced by the SFU operations. Their escape can be therefore accepted (i.e., these faults can be maintained undetected).

C. Thread Dispatcher Test Procedure

The *Thread Dispatcher* dispatches threads of a thread block among the SM internal logic cores. It is a completely custom module developed by nVidia, so an ad-hoc test methodology must be developed.

A fault on a thread dispatcher can lead to different kinds of errors. For example, two threads could be allocated, at the same time, on the same hardware component or a thread could never be allocated. In both cases one thread is never executed on the GPU. Another kind of fault could cause a change of the thread identifier (*thread ID*).

The proposed test procedure is composed of a part executed on the CPU and one executed on the GPU. Algorithm 1 shows the basic operations performed by the proposed test methodology on the CPU side, where *max_thread_per_block* is the maximum number of threads that can be contained into a thread block (i.e., in *Fermi* architecture this value is equal to 1024).

Algorithm 1 CPU-side test procedure

```
1: flag_vector[max_thread_per_block] = {−1, ..., −1}
2: Run_kernel(flag_vector)
3: fault = FALSE
4: for i = 0 → size_of(flag_vector) do
5:     if flag_vector[i] != i then
6:         fault = TRUE
7:     end if
8: end for
```

The kernel is executed on the GPU (*Run_kernel* in Alg. 1)

as a grid. The kernel simply assigns to the *flag_vector* cell, pointed by thread ID, the thread ID value. At the end of the GPU execution, the CPU receives the vector and it verifies that each cell contains a value equal to the cell index (statements from 4 to 8 of Alg. 1), any mismatch represents a fault.
The only fault that is not covered is the case in which two threads swap their identifier. Although, in this case the fault cannot lead to a wrong execution because the complete execution is anyway computed in the right way.
Since in each SM there is only one *thread dispatcher*, to ensure the complete test of this component, the kernel must be executed as a grid composed of thread blocks of *max_thread_per_block* threads.

VI. Experimental results

Performed experiments target two main characteristics of the proposed methodology: the execution time of each test procedure and the fault detection capability.
The CUDA devices used for the tests are: a GeForce GTX 560Ti and a GeForce GTX 580. The former has 1 GB of dedicated RAM, 8 SMs, each one equipped with 48 CUDA cores and 4 SFU, and 2.1 compute capability. The latter is a card with 1.5 GB of dedicated RAM, 16 SMs and a 2.0 compute capability. Each SM contains 32 CUDA cores and 4 SFUs. An Intel Core i5-2500k CPU is used as CPU.
The execution time of each test procedure is characterized in terms of GPU Execution Time (GET), and the time spent on the CPU to perform the faults detection and localization (CET). GET does not include the time required to execute the kernel, only, but also the time to transfer TR/TS to the CPU. Table II shows the execution times associated with each test procedure.

TABLE II: Test procedure execution times

CUDA GPU	Test procedure	GET [ms]	CET [ms]
GeForce GTX560ti (CC 2.1)	*IU*	3.191	0.096
	FPU	1,201.252	0.181
	SFU (10k patterns)	11.996	18.015
	SFU (100k patterns)	77.230	193.024
	Thread Dispatcher	0.611	0.039
GeForce GTX580 (CC 2.0)	*IU*	1.596	0.109
	FPU	609.991	0.173
	SFU (10k patterns)	26.701	18.125
	SFU (100k patterns)	133.495	194.023
	Thread Dispatcher	0.314	0.039

As shown in Table II, the CET related to the IU and FPU test is shorter than the one associated with the SFU test. This strictly depends on the operations that must be performed by the CPU to detect a fault. The procedure adopted to test the IU and the FPU provides a single test signature, thus the CPU must check the correctness of one data, only. Instead, the SFU test procedure provides in output a test result for each applied input test pattern, leading to a high number of checks to detect a fault.

Moreover, comparing the execution time of the two proposed test cases, the only differences concern the IU and the FPU test procedures. This is due to the different test approaches to test the IU and FPU in GPUs with CC 2.0 and 2.1. In fact, as described in Section V-A, more operations are required to test GPU with a CC 2.1 (e.g., *GeForce GTX560ti*) compared to the operations required by GPU with a CC 2.0 (e.g., *GeForce GTX580*).
In order to verify the fault detection capability of the proposed test procedures a fault injection campaign has been performed. The fault injection aims at injecting single and multiple stuck-at faults on each module inside a SM.
Since nVidia does not provide a description of the internal structure of SMs, the faults are injected only on the input and output interfaces of each SM functional block.
Basically, the CPU creates a set of faults equally distributed among the input/output interfaces. These information are provided in input to the GPU exploiting a bitmask. The GPU, before starting the test procedure, performs the injection of the faults identified by the bitmask. In addition, the GPU takes trace of the SM in which faults have been injected and, at the end of the test, it provides this information to the CPU.
Finally, the CPU counts the number of the detected faults in order to verify the achieved fault coverage. It also compares the localized faulty SMs with the ones traced by the GPU, in order to verify the fault localization.
Table III reports the number of faults injected in each SM functional block, and the related fault coverage.

TABLE III: Fault injection campaign

Module	Injected Faults	Detected Faults [%]
IU	16,648,768	99.9%
FPU	16,648,768	99.8%
SFU (10k patterns)	16,484,549	92.38%
SFU (100k patterns)	16,484,549	93.16%
Thread Dispatcher	16,463,528	100%

The high fault detection rate achieved by the IU and FPU test procedures (see Table III) is due to the adopted test procedures (i.e., [20] for IU test and [21] for FPU test). In fact, these two procedures ensure by them self very high-fault coverage independently by the internal architecture of the module under test.
A particular remark has to be done for the SFU test. The fault detection rate of this test procedure strongly depends on the precision adopted to characterize the ε_{gpu} (see Section V-B). In our tests, the ε_{gpu} associated with each SFU operation is characterized by defining six different ranges. For the sake of completeness, Fig. 5 reports an example of ε_{gpu} characterization. As shown in Table III, increasing the number of test patterns the fault detection rate does not increase too much. This is due to the fact that the fault escapes associated to our methodology do not depend on the number of used test patterns, but they depend more on the precision whereby the ε_{gpu} is characterized.

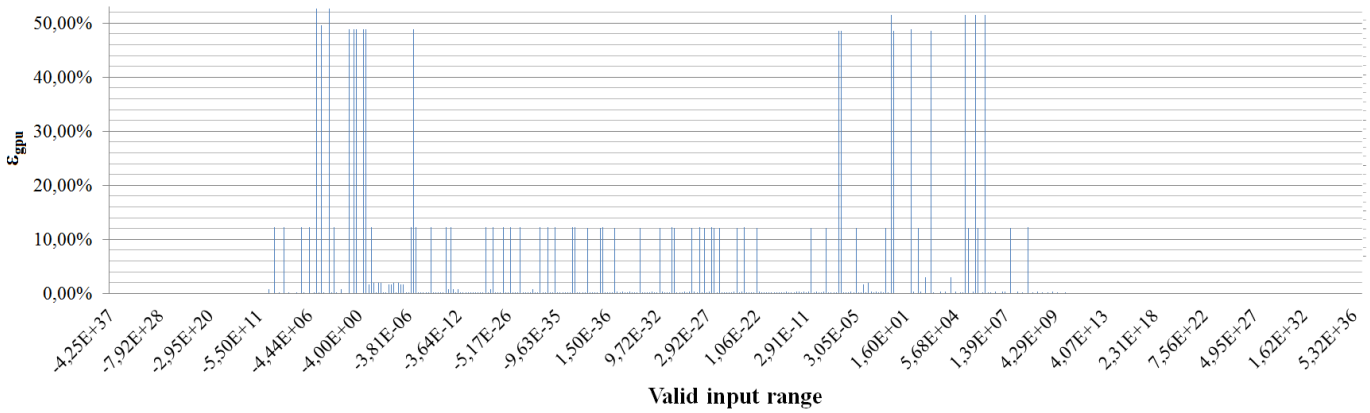

Fig. 5: ε_{gpu} characterization for reciprocal SFU operation

VII. Conclusion

This paper proposes an SBST methodology to test CUDA *Fermi* GPUs. The proposed methodology targets the test of each functional block inside SMs, and it exploits different custom test strategies in order to guarantee a fine-grained permanent fault detection. Moreover, it performs fault-localization at SM-level, enabling the possibility of localizing faulty SMs inside the CUDA *Fermi* GPU under test.

The experimental results show that the proposed test procedures have a very high fault detection capability, and they require a short time to be executed, making them suitable for on-line test.

References

[1] A. Benso, S. Di Carlo, G. Politano, and A. Savino, "GPU acceleration for statistical gene classification," in *Proc. of 2010 IEEE International Conference on Automation Quality and Testing Robotics (AQTR)*, vol. 2, pp. 1–6, 2010.

[2] A. Benso, S. Di Carlo, G. Politano, A. Savino, and A. Scionti, "GPU cards as a low cost solution for efficient and fast classification of high dimensional gene expression datasets," *Control Engineering and Applied Informatics*, vol. 12, no. 3, pp. 34–40, 2010.

[3] Y.-T. Lo, Y.-L. Tsai, H.-W. Wang, Y.-P. Hsu, and T.-W. Pai, "Using solid angles to detect protein docking regions by cuda parallel algorithms," in *Proc. of 8th International Symposium on Parallel and Distributed Processing with Applications (ISPA)*, pp. 536–541, 2010.

[4] H. Patel, "GPU accelerated real time polarimetric image processing through the use of cuda," in *Proc. of 2010 IEEE National Aerospace and Electronics Conference (NAECON)*, pp. 177–180, 2010.

[5] S. Potluri, A. Fasih, L. Vutukuru, F. Al Machot, and K. Kyamakya, "CNN based high performance computing for real time image processing on GPU," in *in Proc. Joint 3rd International Workshop on Nonlinear Dynamics and Synchronization (INDS) & 16th International Symposium on Theoretical Electrical Engineering (ISTET)*, pp. 1–7, 2011.

[6] A. Paschalis and D. Gizopoulos, "Effective software-based self-test strategies for on-line periodic testing of embedded processors," *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, vol. 24, no. 1, pp. 88–99, 2005.

[7] S. Di Carlo, P. Prinetto, and A. Savino, "Software-based self-test of set-associative cache memories," *IEEE Trans. on Computers*, vol. 60, no. 7, pp. 1030–1044, 2011.

[8] S. Alpe, S. Di Carlo, P. Prinetto, and A. Savino, "Applying march tests to k-way set-associative cache memories," in *Proc. of 13th IEEE European Test Symposium (ETS)*, pp. 77–83, 2008.

[9] A. Benso, S. Di Carlo, and A. Savino, "Software-based self-test for reliable applications in railway systems," in *Railway Safety, Reliability and Security: Technologies and Systems Engineering*, pp. 198–220, IGI Global, 2012.

[10] M. Dimitrov, M. Mantor, and H. Zhou, "Understanding software approaches for GPGPU reliability," in *Proc. of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, pp. 94–104, ACM, 2009.

[11] J. W. Sheaffer, D. P. Luebke, and K. Skadron, "A hardware redundancy and recovery mechanism for reliable scientific computation on graphics processors," in *Proc. of the 22nd ACM Symposium on Graphics hardware (SIGGRAPH/EUROGRAPHICS)*, pp. 55–64, Eurographics Association, 2007.

[12] K. S. Yim, C. Pham, M. Saleheen, Z. Kalbarczyk, and R. Iyer, "Hauberk: Lightweight silent data corruption error detector for GPGPU," in *Proc. of 2005 Parallel Distributed Processing Symposium (IPDPS)*, pp. 287–300, 2011.

[13] F. Sinclair, "G-cp: Providing fault tolerance on the GPU through software checkpointing," 2010.

[14] C. Ding, C. Karlsson, H. Liu, T. Davies, and Z. Chen, "Matrix multiplication on GPUs with on-line fault tolerance," in *Proc. of 9th IEEE International Symposium on Parallel and Distributed Processing with Applications (ISPA)*, pp. 311–317, IEEE Computer Society, 2011.

[15] K. Pattabiraman, Z. Kalbarczyk, and R. Iyer, "Automated derivation of application-aware error detectors using static analysis," in *13th IEEE International On-Line Testing Symposium (IOLTS)*, pp. 211–216, 2007.

[16] M. Ernst, J. Cockrell, W. Griswold, and D. Notkin, "Dynamically discovering likely program invariants to support program evolution," *IEEE Trans. on Software Engineering*, vol. 27, no. 2, pp. 99–123, 2001.

[17] nVidia, *NVIDIA's Next Generation CUDA Computer Architecture: Fermi*, 2006.

[18] nVidia, *Parallel Thread Execution ISA v.3.0*, 2012.

[19] nVidia, *nVidia CUDA C Programming Guide v.4.2*, 2012.

[20] A. Paschalis, D. Gizopoulos, N. Kranitis, M. Psarakis, and Y. Zorian, "Deterministic software-based self-testing of embedded processor cores," in *Proc. of 2001 Conference on Design, Automation and Test in Europe (DATE)*, pp. 92–96, 2001.

[21] G. Xenoulis, D. Gizopoulos, M. Psarakis, and A. Paschalis, "Instruction-based online periodic self-testing of microprocessors with floating-point units," *IEEE Trans. on Dependable and Secure Computing*, vol. 6, no. 2, pp. 124–134, 2009.

[22] M. Overton, *Numerical computing with IEEE floating point arithmetic including one theorem, one rule of thumb, and one hundred and one exercises*. Society for Industrial and Applied Mathematics (SIAM), 2001.

[23] K. Arabi and B. Kaminska, "Oscillation-test methodology for low-cost testing of active analog filters," *IEEE Trans. on Instrumentation and Measurement*, vol. 48, no. 4, pp. 798–806, 1999.