POLITECNICO DI TORINO
Repository ISTITUZIONALE

An improved fault mitigation strategy for CUDA Fermi GPUs

Original
An improved fault mitigation strategy for CUDA Fermi GPUs / DI CARLO, Stefano; Gambardella, G.; Martella, I.; Prinetto, Paolo Ernesto; Rolfo, D.; Trotta, P.. - ELETTRONICO. - (2014), pp. 1-6. (Intervento presentato al convegno Dependable GPU Computing workshop 2014 tenutosi a Dresden, DE nel 28 March 2014).

Availability:
This version is available at: 11583/2571949 since: 2016-10-07T16:21:32Z

Publisher:

Published
DOI:

Terms of use:

This article is made available under terms and conditions as specified in the corresponding bibliographic description in the repository

Publisher copyright

(Article begins on next page)

12 October 2024

An improved fault mitigation strategy for CUDA Fermi GPUs

Stefano Di Carlo, Giulio Gambardella, 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.LastName*}@polito.it

***Abstract*—High computation is a predominant requirement in many applications. In this field, Graphic Processing Units (GPUs) are more and more adopted. Low prices and high parallelism let GPUs be attractive, even in safety critical applications. Nonetheless, new methodologies must be studied and developed to increase the dependability of GPUs. This paper presents an improved fault mitigation strategy against permanent faults for CUDA Fermi GPUs. The proposed approach exploits the reverse engineering of the block scheduling policy in CUDA Fermi GPUs in order to minimize the fault mitigation timing overhead. The graceful performance degradation achieved by the proposed technique outperforms multithreaded CPU implementations and other fault mitigation strategies for CUDA GPU, even in presence of multiple permanent faults.**

I. Introduction

In the last years, the increased demand of computational power in safety critical applications, like automotive [1], space [2] and medical [3], results in the assignment of the extensive data processing to highly parallel systems.
CUDA-based GPUs [4] are attractive for these applications, since they are able to simultaneously execute the same operations on different data portions, exploiting the Single Instruction Multiple Data (SIMD) architecture.
However, their usage in safety critical systems still remains an open issue since these devices are not natively protected against faults.
The robustness of a safety critical system is commonly ensured by fault-tolerance or fault mitigation techniques. Thus, the development of fault mitigation techniques for CUDA-based GPUs is a primary requirement.
Several fault tolerance and fault mitigation techniques targeting SIMD processors have been published [5] [6] [7] [8] [9]. Unfortunately, these techniques are not applicable to modern GPU architectures, since they rely on a deep knowledge of the hardware internal architecture.
Other techniques completely independent from the processor internal architecture have been proposed as well. However they introduce high performance overhead [10], or they are able to mitigate a limited number of faults [11], [12], [13].
This paper aims at defining a novel strategy to mitigate multiple permanent faults in CUDA-based GPUs. A graceful performance degradation is achieved by the proposed fault mitigation technique, based on program instrumentation, providing correct results even in presence of faults. Such modifications let the GPU programmer to easily change the original CUDA software, to increase the dependability of the target system.
The proposed technique improves the approach we previously propose in [14]. [14] presents two fault mitigation methodologies in order to increase the robustness of a CUDA Fermi GPU-based system against permanent faults. Basically, by periodically running the Software Based Self Test (SBST) methodology presented in [15], it is possible to identify faulty Streaming Multiprocessors (SMs) in the GPU under test. This information is exploited in order to avoid the execution on faulty SMs, and provide in output correct results even in presence of multiple permanent faults. The execution on faulty SMs is prevented through two proposed fault mitigation strategies that do not introduce any performance penalty during the GPU fault-free execution, and they guarantee error-free results also in presence of a high number of faults.
The strategy proposed in this paper exploits the reverse engineering of the block scheduling policy in CUDA Fermi GPUs in order to minimize the timing overhead due to the reduced number of SMs actually performing the computation (i.e., faulty SMs cannot perform computation). The paper is organized as follows: Section II briefly introduces the CUDA Fermi architecture. Section III describes the basic approach to apply the proposed strategy. The improved fault mitigation technique is defined and explained in Section IV. Experimental results are given in Section V. Finally, Section VI concludes the paper.

II. CUDA Overview

Several GPUs produced by nVidia use CUDA®[16]. CUDA supports a new software architecture that enables CUDA-based GPUs to execute programs written in C, C++, Fortran, OpenCL, DirectCompute, and other languages [4].
Programs executed by CUDA GPUs are called *kernels*. A kernel is basically a set of parallel threads that ensures a high-parallel computation. When a kernel is compiled, its threads are grouped in *blocks*. The complete set of blocks composes a *grid*.
A CUDA-based system is composed of a CPU (*Host*), that executes a program (CUDA program) in order to both create inputs for the kernel and start the kernel execution by providing in input the grid to the GPU, and a CUDA GPU (*Device*), that performs the execution of the kernel. At the end of a kernel execution, the CPU can flush the content of the GPU memory in order to acquire output data.

The software organization of a kernel is strictly related to the GPU hardware architecture, since the thread hierarchy is directly mapped into GPU internal components.
The basic building blocks of a CUDA GPU (Fig. 1) are: (i) a *Block dispatcher*, that manages the scheduling of the input grid by assigning each thread block to the internal logic; (ii) a *Global Memory*, that stores intermediate and final results of the executed kernel; (iii) a *Shared Cache*, that speeds up read/write operations on the global memory; (iv) several *Streaming Multiprocessors* (SMs). Each SM includes many CUDA cores, that perform the computation for each thread.

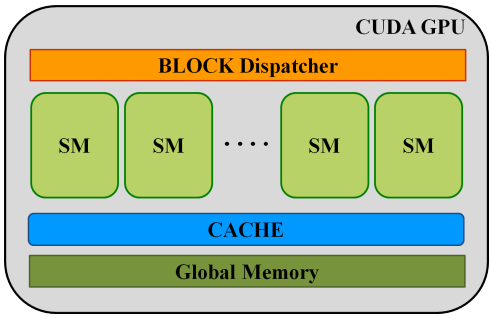


Fig. 1. CUDA GPU internal architecture

When the CPU invokes a kernel grid, each block is numbered (by assigning it a *Block ID*) and dispatched to a SM (i.e., during the execution each SM is numbered through a *SM ID* as well) that guarantees enough available resources. Each thread of a block is executed on a CUDA core. Considering the SIMD architecture of the GPU, the same operation is performed on different input data portions, addressed by the *Block ID*. As blocks terminate, new blocks are dispatched to idle SMs. The number of blocks that can be processed concurrently on the multiprocessor (*Blocks_per_SM*) depends on the number of registers, on the amount of shared memory available in the SM, and on the resources required by the kernel to be executed. The *Blocks_per_SM* value can be defined exploiting a tool released by nVidia in the *CUDA Toolkit*, called *CUDA Occupancy Calculator*. Anyway, the number of blocks that can be assigned to a SM never exceeds 8 in *Fermi* architecture [4].

III. Basic Approach

In order to apply the improved fault mitigation strategy, a similar approach to the one presented in [14] has been exploited (see Fig. 2). This approach defines a methodology to instrument the CUDA program and the kernel, in order to obtain correct results from a kernel running on a faulty GPU. As shown in Fig. 2, the *Instrumented CUDA Program* (ICP), running on the CPU, informs each kernel about the status of each SM (i.e., fault-free or faulty) through the *Faulty Mask* (FM), and runs the Instrumented Kernel (IK) on the GPU. The status of each SM can be defined by periodically running functional test procedures on the GPU, such as the one proposed in [15].
The IK identifies blocks dispatched on faulty SMs (*Faulty Blocks* (FBs)), stops their execution by applying the improved fault mitigation strategy (see section IV), and transmits FBs to the ICP.
Eventually, if no FB still remains, the kernel execution has

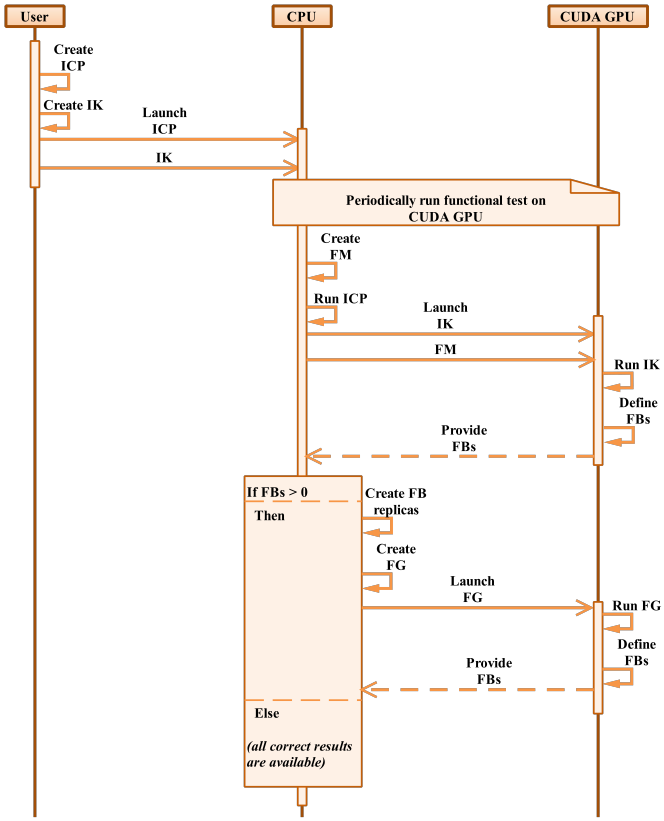


Fig. 2. Sequence Diagram of the basic approach

been completed correctly, otherwise the instrumented CUDA program creates several replicas of each FB, defines a new grid (*faulty grid*, *FG*) containing these replicas, and run the FG on the GPU. This process is repeated until no FB is detected.
For more detailed information about the methodology to create the ICP, the IK, and the FG the reader may refer to [14].

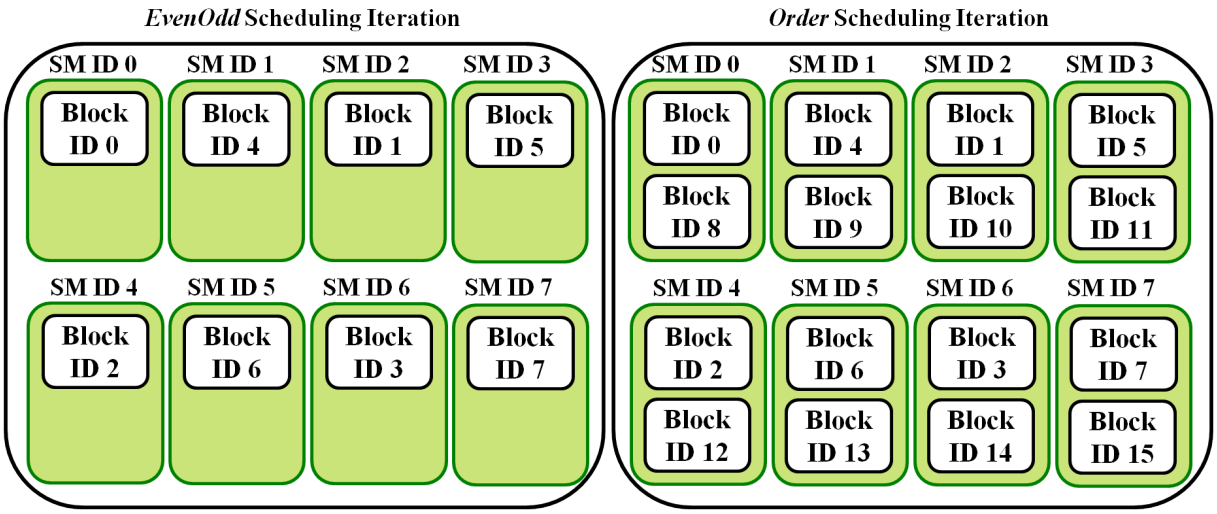
IV. Improved Fault Mitigation Strategy

The improved fault mitigation strategy, called *Smart Wait*, exploits the reverse engineering of the block scheduling policy in CUDA Fermi GPUs. Comparing with the fault mitigation strategies presented in [14], *Smart Wait* leads to the following benefits: (i) no grid re-execution is required to obtain correct results, and (ii) the computational effort is reduced, since only the strictly required FB replicas are executed (i.e., in the *Sleep* and *Wait* strategy of [14] more FB replicas must be generated to ensure correct results in output). The following subsections deeply describe the reverse engineering of the block scheduling, and the *Smart Wait* strategy.

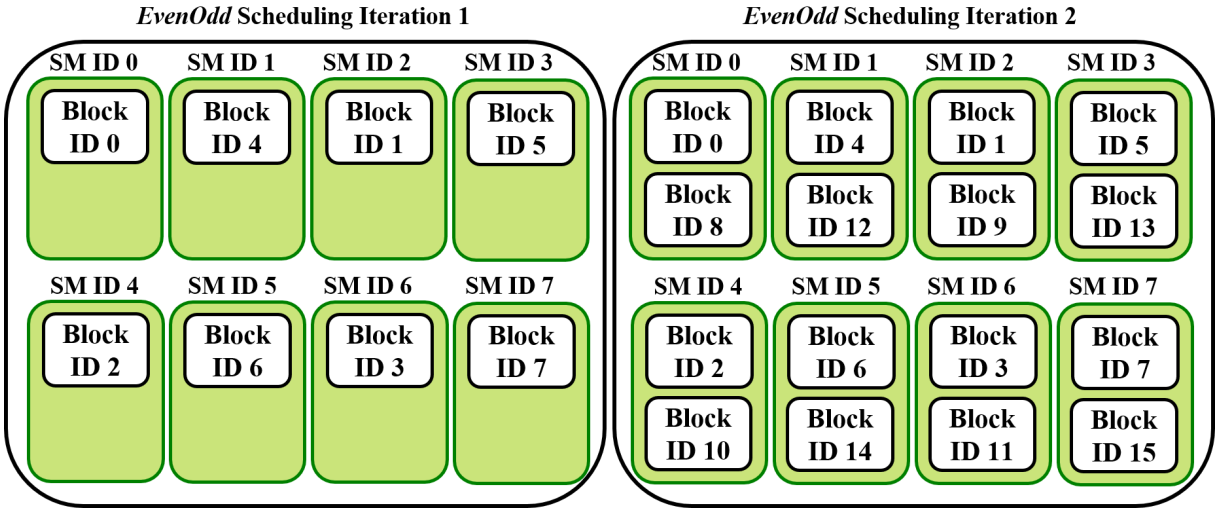
A. Block Scheduling in CUDA Fermi GPUs

The block scheduling in CUDA Fermi GPUs is partially static and partially dynamic. At the beginning of the kernel execution, a number of blocks equals to *Blocks_per_SM* (see Section II) are statically dispatched on every SM (Eq. 1 defines the number of the statically scheduled blocks).

$$Static_Blocks_Allocation = \#SM * Blocks_per_SM \quad (1)$$



(a) Static scheduling example with a grid composed by 16 blocks



(b) Static scheduling example with a grid composed by more than 16 blocks

Fig. 3. Static scheduling examples in a CUDA Fermi GPU equipped with 8 SMs and characterized by the *Blocks_per_SM* parameter equal to 2

where $\#SM$ is the number of SMs equipped on the GPU.

The remaining blocks will be dispatched dynamically after the complete execution of the statically scheduled blocks.
To understand how the blocks are actually scheduled, two different kernels have been defined:

- *Static Kernel*: defines the static allocation policy. This kernel has been compiled in a grid composed of a number of blocks equal to *Static_Blocks_Allocation*. In order to properly define on which SM each block is scheduled, the kernel processes an input vector with *Total_Blocks* cells by storing the *SM ID* in the cell pointed by the *Block ID* (see Section II).
- *Dynamic Kernel*: defines the dynamic allocation policy. This kernel exploits the same approach used in the *Static Kernel*, but in addition it delays the execution on one or more SMs in order to defined the dynamically scheduling policy when the *Static_Blocks_Allocation* execution is not synchronized (i.e., the execution of statically scheduled blocks ends in different times), also. Obviously, this kernel must be compiled in a grid with a number of blocks greater than *Static_Blocks_Allocation*.

A long sequence of tests on these kernels, modifying the number of blocks composing the grid and the number of delayed SMs, has been performed. Analysing the obtained results, three possible scheduling sequences have been identified, two associated with the static scheduling, namely *Order* and *EvenOdd*, and the *Random* one associated to the dynamic scheduling. In every scheduling sequence, blocks are iteratively scheduled on SMs, starting from the first block (i.e., *Block ID* = 0) to the last one. These iterations continue until a number of blocks equal to *Blocks_per_SM* have been scheduled to each SM, or when all the blocks composing the grid have been scheduled. The unique difference among the three defined sequences concerns the selection of the SM on which each block must be scheduled.

TABLE I. EXECUTION TIME AND PERFORMANCE GAIN ASSOCIATED WITH THE FAULT MITIGATION STRATEGY

# Faulty	Matrix Transpose			CUDA Separable Convolution			Fast Walsh Transform		
	CPU MT [ms]	Smart Wait [ms]	PG [ms]	CPU MT [ms]	Smart Wait [ms]	PG [ms]	CPU MT [ms]	Smart Wait [ms]	PG [ms]
0	62	2	31.00	743	6	123.83	977	14	69.01
1	62	2	31.00	743	8	92.88	977	16	59.05
2	62	2	31.00	743	8	92.88	977	19	50.45
3	62	3	20.67	743	10	74.3	977	16	57.71
4	62	3	20.67	743	11	67.55	977	15	64.88
5	62	4	15.5	743	15	49.53	977	18	53.05
6	62	7	8.86	743	22	33.77	977	24	40.60
7	62	15	4.13	743	42	17.69	977	29	33.12

The *Order* sequence defines the SMs in an ordered way (i.e., from the first to the last), instead the *EvenOdd* first schedules blocks to the SMs with an even *SM ID* and then to the one with an odd *SM ID*. Eventually, in the *Random* sequence SMs are randomly selected.

In the static scheduling, the selection between *Order* or *EvenOdd* sequence is performed depending on the *Static_Blocks_Allocation* (SBA) value and the grid dimension (*K*). According to Eq. (2), if the dimension of the grid is less than or equal to the *Static_Blocks_Allocation* parameter, $\left(\left\lceil \frac{K}{\#SM} \right\rceil - 1\right)$ scheduling iterations are performed following the *EvenOdd* sequence, while the remaining according to the *Order* one. Instead, if if the dimension of the grid is greater than the *Static_Blocks_Allocation*, the complete static scheduling is performed following the *EvenOdd* sequence.

$$\begin{cases} \left(\left\lceil \frac{K}{\#SM} \right\rceil - 1\right) * EvenOdd + Order, & K \leq SBA \\ EvenOdd * Blocks_per_SM, & K > SBA \end{cases} \quad (2)$$

Instead, the dynamic scheduling is always performed following the *Random* sequence.

For the sake of completeness, Fig. 3 shows an example for both cases, where #SM = 8, and *Blocks_per_SM* = 2 (i.e., SBA = 16). In particular, Fig. 3(a) shows the static scheduling of a grid composed by 16 blocks. Thus, according to the first row of Eq. (2) , the scheduling is performed in two iterations, the first following the *EvenOdd* sequence, while the second following the *Order* one. In the case reported in Fig. 3(b), since K $>$ 16, the scheduling task is performed in two iterations as well, but in both cases the *EvenOdd* sequence is used (second row of Eq. (2)).

B. Smart Wait

The *Smart Wait* fault mitigation strategy improves the *Wait* strategy presented in [14]. The basic idea is to stop the blocks executed on the faulty SM until all blocks dispatched on the fault-free SMs are completely executed. Moreover, since only statically scheduled blocks can be dispatched on a faulty SM, FBs can be defined a priori exploiting the reverse engineering of the scheduling policy (see Section IV-A). For this reason, the FB replicas can generated and added to the normal blocks composing the IK grid. In this way, it is possible to obtain the correct results avoiding the FG execution. Thus, the *Smart Wait* allows to speed up the computation w.r.t. the methodologies presented in [14], that require one or more FGs to ensure a fault-free execution.

Moreover, the proposed methodology reduces the computational effort required to ensure correct results from the kernel execution on a faulty GPU. In fact, in the *Smart Wait* the number of required replicas for each FB is equal to 1, only. Instead, in the *Wait* strategy, since it is completely independent from the scheduling policy, the number of required replicas is higher and can be computed according to Eq. 2.

$$n_replicas = \#Faulty_SM * Blocks_per_SM + 1 \quad (2)$$

V. EXPERIMENTAL RESULTS

The effectiveness of the proposed mitigation strategy has been proved with an extensive test campaign. The used testbed is composed of a *Gigabyte GeForce GTX 560Ti*, equipped with 1 GB of dedicated RAM and 8 Streaming Multiprocessors (SM), and an Intel Core i5-2500k CPU.

Since in the last year safety-critical embedded applications (e.g., Assistance Drive, Unmanned Avionic Vehicle, and Space) require more and more computational power, the usage of a GPU device in this systems could became of primary importance. For this reason, the proposed fault mitigation strategy have been applied to a set of CUDA SDK applications that can be exploited in this kind of embedded systems. In particular, the selected applications are:

- *Matrix Transpose*: it computes in parallel the transpose of a matrix. Matrix transpose is essential in the Synthetic Aperture Radar (SAR) imaging algorithm, since it is the bottleneck in this kind of signal processing [17]. In the last year, SARs have became more and more important for their application in many safety critical applications like Unmanned Avionic Vehicle (UAV) and satellite.
- *CUDA Separable Convolution*: it performs the gaussian smoothing of an image with a 7x7 kernel. This kind of filtering operation is widely used as a preprocessing stage in computer vision algorithms in order to enhance image structures at different scales, and to reduce image noise [18].
- *Fast Walsh Transform*: it compute the Hadamard-ordered Fast Walsh Transform of an image with a 32x32 kernel. Walsh Hadamard Transform allows to efficiently perform the pattern matching (i.e., it searches and matches the kernel inside the input

image), that is a widely used operation in signal processing, computer vision, image and video processing [19].

All applications have been executed on the CPU (i.e., exploiting Multi-Thread (MT) implementations) and on the CUDA GPU. The performance of the two platforms have been compared and the correctness of the results evaluated.
CUDA programs and kernels have been instrumented according to the methodology presented in Section III.
Each application has been tested in 8 different conditions, with different number of faulty SMs, in order to characterize the execution times related to the CPU and GPU implementation. Table I lists the execution times for the exploited test algorithms. The comparison is made between the execution times of the CPU (CPU MT) and GPU (Smart Wait) algorithms by reporting the Performance Gain (PG). From the presented data it is clear that the usage of a GPU, also in presence of faults, with both proposed fault mitigation strategies ensures better performance w.r.t. a mutlithread execution on CPU. Clearly, the execution time on the GPU increases with the number of faulty SMs.
Moreover, the exploited test algorithms have been run applying the *Wait* strategy [14], also. The performance of the proposed fault mitigation strategy and of the *Wait* one have been evaluated comparing the associated execution time. Fig. 4 reports the Absolute Performance Gain (APB) provided by the *Smart Wait*.

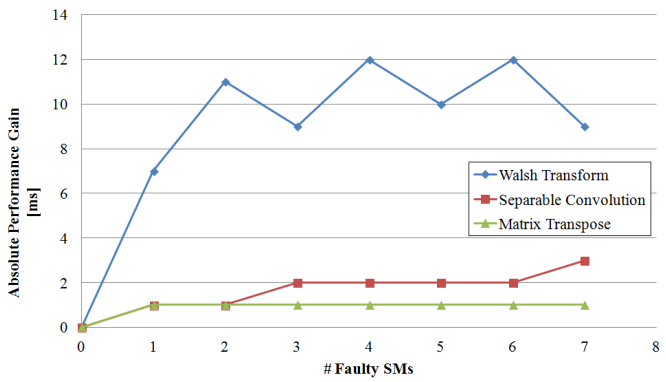


Fig. 4. Absolute performance gain of the *Smart Wait* strategy w.r.t. the *Wait* one

Clearly, when no Faulty SM are present the performance gain is equal to 0, since both strategies do not require any computation overhead (i.e., FBs to be re-executed) to ensure correct results. Instead, as shown in Fig. 4, increasing the number of the Faulty SMs the benefit provided by the proposed strategy increases. The provided performance gain is especially due to two main aspects. The former concerns the communication overhead between CPU and GPU due to the launch of the FG execution (see Section IV-B) that is not introduced by proposed strategy. However, this is not valuable in the proposed test algorithms since the dimension of the grid is limited, and the associated transfer time is negligible. This contribute could be more valuable if the executed program is more complex, leading to a big grid dimension.
Instead, the latter concerns the reduction of the CPU computation required by the *Smart Wait* in order to create the replicas of each FB. In fact, in the proposed approach the extra CPU computation concerns the creation of one replica of each FB before the launch of the IK, only. Instead, the *Wait* strategy requires a higher number of replicas for each FB (see Eq. 2). As shown in Fig. 4, this benefit is lower in the *Separable Convolution* and in the *Matrix Transpose* than in the *Walsh Transform*, since the first two applications require one kernel execution only to perform the computation (i.e., FB replicas must be created once, only). Instead, Since the *Walsh Transform* to provide final results requires three kernels to be re-executed four times each, the FB replicas must be created twelve times to achieve correct results (i.e., one for each kernel execution), leading to a higher performance gain.

VI. Conclusion

The paper presents a methodology to allow the use of a CUDA-based GPU, even in presence of faulty streaming multiprocessors. While the other already presented methodologies are algorithm dependent or introduce performance, the presented fault mitigation technique is completely algorithm independent and, exploiting the reverse engineering of the block scheduling policy in CUDA Fermi GPUs, allows to reach the maximum performance during CUDA fault-free execution. Moreover, a validation campaign has been performed to highlight the benefits of the proposed technique w.r.t. to the current state-of-the-art fault mitigation strategies.

Acknowledgment

This research has been partly supported by the 7th Framework Program of the European Union through the the CLERECO Project, under Grant Agreement 611404.

References

[1] K. Haklin and H. Ho-sang, "Integrated Fault Tolerant System for Automotive Bus Networks," in *2nd International Computer Engineering and Applications Conference*, 2010, pp. 486–490.

[2] Q. Hu, B. Xiao, and M. Friswell, "Robust fault-tolerant control for spacecraft attitude stabilisation subject to input saturation," *Control Theory Applications*, vol. 5, no. 2, pp. 271–282, 2011.

[3] N. Z., "Investigation of Fault-Tolerant Adaptive Filtering for Noisy ECG Signals," in *IEEE Symposium on Computational Intelligence in Image and Signal Processing*, 2007, pp. 177–182.

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

[5] A. Sengupta and C. Raghavendra, "All-to-all broadcast and matrix multiplication in faulty SIMD hypercubes," *IEEE Transactions on Parallel and Distributed Systems*, vol. 9, no. 6, pp. 550–560, 1998.

[6] J.-H. Kim, F. Lombardi, and N. Vaidya, "An improved approach to fault tolerant rank order filtering on a simd mesh processor," in *Proceedings IEEE Int. Workshop on Defect and Fault Tolerance in VLSI Systems*, 1995, pp. 137 –145.

[7] A. Strano, D. Bertozzi, A. Grasset, and S. Yehia, "Exploiting structural redundancy of SIMD accelerators for their built-in self-testing/diagnosis and reconfiguration," in *IEEE International Conference on Application-Specific Systems, Architectures and Processors*, 2011, pp. 141–148.

[8] J.-H. Kim, S. Kim, and F. Lombardi, "Fault-tolerant rank order filtering for image enhancement," *IEEE Transactions on Consumer Electronics*, vol. 45, no. 2, pp. 436–442, 1999.

[9] C. Raghavendra and M. Sridhar, "Global commutative and associative reduction operations in faulty simd hypercubes," *IEEE Transactions on Computers*, vol. 45, no. 4, pp. 495 –498, 1996.

[10] X. Xu, Y. Lin, T. Tang, and Y. Lin, "HiAL-Ckpt: A hierarchical application-level checkpointing for CPU-GPU hybrid systems," in *Proceedings IEEE Int. Conf. on Computer Science and Education (ICCSE)*, 2010, pp. 1895–1899.

[11] K.-H. Huang and J. A. Abraham, "Algorithm-based fault tolerance for matrix operations," *IEEE Transaction on Computers*, vol. 33, no. 6, pp. 518–528, 1984. [Online]. Available: http://dx.doi.org/10.1109/TC.1984.1676475

[12] C. Braun and H.-J. Wunderlich, "Algorithmen-basierte fehlertoleranz fr many-core-architekturen (algorithm-based fault-tolerance on many-core architectures)." *it - Information Technology*, vol. 52, no. 4, pp. 209–215, 2010. [Online]. Available: http://dblp.uni-trier.de/db/journals/it/it52.html#BraunW10

[13] C. Ding, C. Karlsson, H. Liu, T. Davies, and Z. Chen, "Matrix multiplication on GPUs with on-line fault tolerance," in *9th Int'l Symposium on Parallel and Distributed Processing with Applications (ISPA)*, 2011, pp. 311–317.

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

[15] S. Di Carlo, G. Gambardella, M. Indaco, I. Martella, D. Rolfo, P. Prinetto, and P. Trotta, "A Software-Based Self Test of CUDA Fermi GPUs," in *18th European Test Symposium (ETS)*, 2013.

[16] nVidia, *NVIDIA CUDA Architecture - Introduction & Overview*, Internet, 2009.

[17] M. Bian, F. Bi, and F. Liu, "Matrix transpose methods for sar imaging system," in *Signal Processing (ICSP), 2010 IEEE 10th International Conference on*, 2010, pp. 2176–2179.

[18] R. González and R. Woods, *Digital Image Processing*. Pearson/Prentice Hall, 2008.

[19] W. Ouyang and W.-K. Cham, "Fast algorithm for walsh hadamard transform on sliding windows," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 32, no. 1, pp. 165–171, 2010.