Fault mitigation strategies for CUDA GPUs

Fault mitigation strategies for CUDA 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 effective fault mitigation strategies for CUDA-based GPUs against permanent faults. The methodology to apply these strategies, on the software to be executed, is fully described and verified. The graceful performance degradation achieved by the proposed technique outperforms multithreaded CPU implementation, even in presence of multiple permanent faults.

1 Introduction

Nowadays, automotive [1], space [2] and medical [3] applications are characterized by an increasing reliance on intelligent control and safety. The robustness of a system, i.e., its ability to continue to function despite the existence of faults, even if system performance may be diminished or altered, is often enhanced by fault-tolerance or fault mitigation techniques. Furthermore, dependable systems must guarantee real-time responses, thus requiring high computation capability. Extensive data processing is usually assigned to highly parallel systems, like multi-cores processors or Graphical Processing Units (GPUs). In this field, CUDA-based GPUs [4] are attractive devices, blending low cost and high computation capability.

GPU usage is mostly limited to computer graphics entertainment and, more recently to algorithm acceleration for high-performance computing. Processing is performed in parallel, exploiting a Single Instruction Multiple Data (SIMD) architecture, that executes the same operations on different data portions, simultaneously. Nonetheless, using Commercial Off-The-Shelf (COTS) devices in safety critical systems comes at the cost of dependability. Developing fault mitigation techniques is needed to tolerate hardware defects and the related faults.

Several works targeting SIMD processors have been published. Overall, they propose fault-tolerance and fault mitigation techniques [5] [6] [7] [8] [9]. Unfortunately, these techniques are not applicable to modern GPU architectures, since they rely on a deep knowledge of the processor internal architecture.

Different techniques completely independent from the processor internal architecture, as, for instance, *Check pointing-based* and *Algorithm-based fault tolerance*, are available as well.

Check pointing-based methodologies allow to detect errors in a device by inserting checkpoints into the executed program. At each checkpoint the obtained results are compared with the expected ones to detect errors. Detected errors can then be corrected by re-executing the associated piece of code. [10] proposes a check pointing technique that can be applied to GPUs. It requires a hybrid GPU-CPU system, where the GPU performs the computational task and the CPU checks the correctness of the results at each checkpoint. This approach guarantees a high error-detection capability, but the communication of the results between the CPU and the GPU causes high performance overhead also with a fault-free GPU.

Algorithm-based fault tolerance methodologies detect and correct errors in the system under test by defining a software approach customized and optimized for the target algorithm. In this way, the test guarantees a high reliability of the system and a low impact on performances.

Many of these techniques target matrix multiplication algorithms [11], [12], [13]. The common basic idea is to add a checksum to rows and columns of the result matrix, in order to allow error-detection and, potentially, error-correction. These approaches guarantee small performance overhead and high error-detection capability. Their main drawback is the completely custom approach used to provide the fault tolerance. The designer is forced to completely modify the test procedure when changes are made in the original algorithm. Moreover, these techniques are able to detect and correct a fixed number of errors, only.

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 techniques, 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 methodology introduces three main advantages w.r.t. the already presented methodologies:

1. the required instrumentation is completely independent of the executed code;
2. it does not introduce any execution time penalties during the GPU fault-free execution, thus the executed code can reach the maximum performances provided by the GPU;
3. it guarantees fault-free results also in presence of a high number of permanent faults.

The paper is organized as follows: Section 2 briefly introduces the CUDA-based GPU architecture. Section 3 describes the proposed approach. The fault mitigation techniques are defined and explained in Section 4. Experimental results are given in Section 5. Finally, Section 6 concludes the paper.

2 CUDA Overview

Several GPUs produced by nVidia use CUDA®[14]. 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 *thread blocks*. The complete set of thread blocks composes a *grid*.

A CUDA-based system is composed of a CPU and a CUDA GPU. The CPU executes a program (CUDA program) in order to both create inputs for the kernel and to start the kernel execution. Starting the kernel execution means providing a kernel grid to the GPU. The CUDA GPU 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 threads 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.

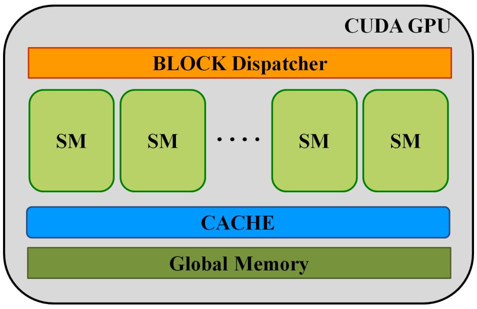


Figure 1: CUDA GPU internal architecture

When the CPU invokes a kernel grid, each thread block is numbered (assigning it a *Thread Block ID*) and dispatched to a SM that guarantees enough available resources. Each thread of a thread 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 *Thread Block ID*. As thread blocks terminate, new blocks are dispatched to idle SMs. The number of thread blocks that can be processed concurrently on the multiprocessor 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. Anyway, the number of thread blocks that can be assigned to a SM never exceeds 8 in *Tesla* [15] and *Fermi* [4] architectures and 16 in the *Kepler* [16] architecture, respectively.

3 Proposed Fault Mitigation Methodology

The methodology proposed in this paper enables software-level fault mitigation in CUDA GPUs. The overall approach requires the knowledge of the full map of faulty SMs. Faulty SMs can be identified by periodically running functional test procedures on the GPU, such as the one proposed in [17]. This activity is out of the scope of this paper that focuses on the mitigation of these faults. We will therefore assume that a *Faulty SM Map* (FM), indicating

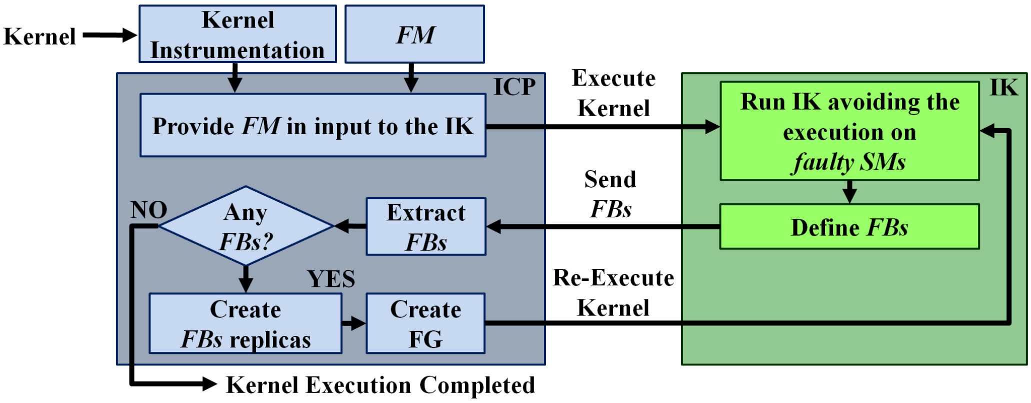


Figure 2: Proposed Approach

the healthiness of each SM, is available and exploitable to perform fault mitigation. The proposed approach defines a methodology to instrument the CUDA program and the kernel, in order to correctly execute a kernel on a faulty GPU. As shown in Fig. 2, the *Instrumented CUDA Program* (ICP) informs each kernel about the status of each SM, providing it the FM, and runs the instrumented kernel (IK) on the GPU.

The instrumented kernel avoids the execution of thread blocks on faulty SMs. Since this task cannot be performed by directly operating on the Block Dispatcher (the CUDA-ISA does not provide this functionality), an ad-hoc procedure must be implemented. Exploiting the FM, the instrumented kernel can identify thread blocks dispatched on faulty SMs (*faulty blocks* (FBs)) and stop their execution (see Sec. 3.2). Moreover, IK is able to transmit to the instrumented CUDA program all FBs.

The instrumented CUDA program exploits this information to identify whether the kernel execution completed correctly, or not. The execution can be considered correctly completed if every thread block is executed on a fault-free SM. If there is at least one FB, the instrumented CUDA program creates several replicas of each FB. These replicas are used to define a new grid (*faulty grid*, *FG*), that is executed on the CUDA GPU. Activating more than one replica ensures that each faulty block is executed on at least one fault-free SM. The process is repeated until no FB is identified.

In the following subsections the proposed approach is deeply analyzed proposing the instrumentations to be applied on the CUDA program and on the kernel.

3.1 CUDA program instrumentation

The ICP (left side of Fig. 2) implements Alg. 1.

First, the instrumented CUDA program provides the FM to the kernel (row 3 in Alg. 1), through the *Execution Configuration* structure. This structure contains the FM and additional information used to describe the status of each thread block of the grid. In detail, this structure includes:

Algorithm 1 Instrumented CUDA Program

```
1:  for each $Kernel$ do
2:      $Execution\_Complete = FALSE$
3:      $Provide\_FM\_to\_IK()$
4:      $Run\_IK()$
5:      while $Execution\_Complete = FALSE$ do
6:          $Extract\_FBs()$
7:          if $\#FB = 0$ then
8:              $Execution\_Complete = TRUE$
9:          else
10:             $Compute\_Replicas()$
11:             $Update\_BlockList()$
12:             $Create\_Grid()$
13:             $Run\_IK()$
14:         end if
15:     end while
16: end for
```

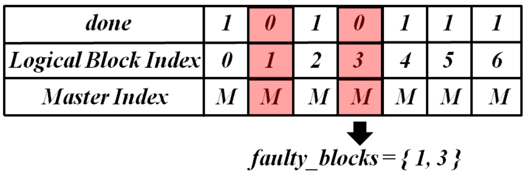
- *faulty_SM_map*: it contains the FM. It is a vector composed of a cell for each SM in the GPU, asserted when the corresponding cell is faulty.

- *block_list*: it is a vector of structures describing the status of each thread block. The size of the *block_list* vector is equal to the overall number of thread blocks composing the grid. Each structure contains three fields: *done*, *Logical block index* (LBI), and *Master Index*. The *done* field, indicates the execution status of a thread block (i.e., if the thread block has been already executed on a fault-free SM or not). LBI identifies an input data portion on which operations are perfomed.

The *Master Index* can assume three different values: *Master* (M), *Replicated Master* (RM) or the index of the master associated with the considered thread block. The meaning of this field is described later in this section.
During the initialization phase, each *Master Index* is set to M for every thread block.

- *grid_size*: it specifies the dimension of the grid to be executed on the GPU, in terms of thread blocks. During the initialization phase, the grid has a dimension equal to the number of thread blocks composing the kernel. After the first execution of the kernel, the grid dimension becomes equal to the number of identified faulty blocks.

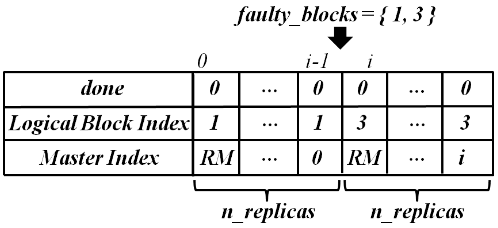
With FM correctly delivered to the kernel, the IPC runs the IK (see Sec. 3.2 for kernel instrumentation) on the GPU (row 4 in Alg. 1), and it updates the *Execution Configuration* structure (see Sec. 3.2).
At the end of the execution, the ICP analyzes the *Execution Configuration* structure to identify the thread blocks dispatched to a faulty SM (row 6 in Alg. 1). This task is performed by analyzing the *done* field in the *block_list* vector (Fig. 3).



done	*1*	*0*	*1*	*0*	*1*	*1*	*1*
Logical Block Index	*0*	*1*	*2*	*3*	*4*	*5*	*6*
Master Index	*M*	*M*	*M*	*M*	*M*	*M*	*M*

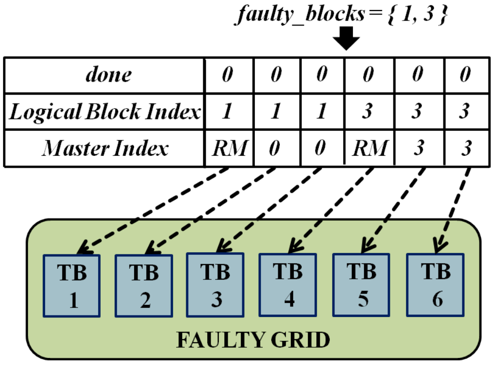
Figure 3: Example of FBs extraction

If there are no FBs (row 7 in Alg. 1), the execution of the kernel is completed. Otherwise, each FB must be re-executed. This task is performed by creating a new grid composed of replicas of each FB. The number of replicas (*n_replica*) is computed (row 10 in Alg. 1) in a different way depending on the adopted fault mitigation strategy (see Sec. 4). Then, the *block_list* vector is updated (row 11 in Alg. 1), in order to insert all information items used to create the *faulty grid* (FG). Fig. 4 shows an example of how the *block_list* is updated.
The *Master Index* field is set to *RM* only for the first replica of each faulty block. In all other replicas, this field is equal to the index of the *block list* cell associated with the first replica.
The FG is then created (row 12 in Alg. 1) by inserting a number of thread blocks (TB in Fig. 5) equal to the number of elements in the *block_list* vector. Finally, the kernel is executed on the GPU, providing the updated *Execution Configuration* and the FG (row 13 in Alg. 1).



	0		*i-1*	*i*		
done	*0*	...	*0*	*0*	...	*0*
Logical Block Index	*1*	...	*1*	*3*	...	*3*
Master Index	*RM*	...	*0*	*RM*	...	*i*

Figure 4: Example of *block_list* vector update



done	*0*	*0*	*0*	*0*	*0*	*0*
Logical Block Index	*1*	*1*	*1*	*3*	*3*	*3*
Master Index	*RM*	*0*	*0*	*RM*	*3*	*3*

Figure 5: Example of faulty grid creation

It is worth to note that, before the dispatch, TBs are numbered (i.e., the dispatcher of the CUDA GPU assigns them a TB ID), and, in general, the TB ID is different from the user-defined LBI.

3.2 Kernel instrumentation

The kernel, invoked by the ICP, must be instrumented, as well. Alg. 2 summarizes the operations that the Instrumented Kernel has to perform.
First, it identifies the SM on which each thread block is running (row 1 in Alg. 2). This task is performed by reading the *%smid* special register value using the CUDA PTX inline assembly [18]. This register stores the identifier of the SM (SM ID) on which the thread block is running.
By reading the location of the *faulty_SM_map* pointed by the SM ID, it is possible to check if a thread block is running on a faulty SM (row 2 in Alg. 2). In this case, the execution of the thread block must be stopped (row 3 in Alg. 2). The approach to stop the execution changes depending on the adopted fault mitigation strategy (analyzed in Sec. 4).
A faulty SM must be able to execute the first three statements of Alg. 2, at least. Thus, this capability of faulty SMs must be tested before starting the execution of the CUDA program. This task can be performed with a simple starting small approach.
To reduce the performance overhead introduced by the

Algorithm 2 Instrumented Kernel

```
 1: $Get\_SM\_ID()$
 2: if $Exec\_on\_faulty\_SM()$ then
 3:     $Avoid\_Execution()$
 4: else
 5:     if $is\_M\_or\_RM()$ then
 6:         $TB\_status \leftarrow read\_done(TB\_ID)$
 7:     else
 8:         $TB\_status \leftarrow read\_done(MI)$
 9:     end if
10:     if $TB\_status = NOT\_EXECUTED$ then
11:                 ...
12:         ...Original Kernel Code...
13:                 ...
14:         $Update\_done()$
15:     end if
16: end if
```

adopted fault-mitigation strategy, each faulty block must be executed only once on a fault-free SM. Since the FG contains replicas of each faulty block (see Sec. 3.1), more than one replica of the same faulty block might be dispatched on different fault-free SMs.

For this reason, when the SM on which the thread block is running is fault-free (row 4 in Alg. 2), the IK checks whether this thread block has been already executed on a fault-free SM, or not. Such a check is performed by analyzing the *Master Index* field in the *block_list* vector (row 5 in Alg. 2).

If the value of the *Master Index* field is equal to *M*, it means that the current thread block has never been replicated inside the grid. If it is *RM*, the current thread block is a replica, but it is the reference copy for its other replicas. In both cases, the execution status of the current thread block can be directly extracted from the *done* field, contained in the *block_list* vector, pointed by the current TB ID (row 6 in Alg. 2).

Otherwise, if the *Master Index* field contains a number, the current thread is not a reference replica. In this case, the execution status is extracted by reading the *done* field and by pointing the *block_list* vector with the *Master Index* value (row 8 in Alg. 2). It this way all replicas of the same thread block point to the same execution status flag.

If the current thread block has never been executed on a fault-free SM (*TB_status=NOT_EXECUTED* in Alg. 2), instructions of the original kernel are executed (row 12 in Alg. 2).

Obviously, these operations must be performed on the correct portion of input data. As mentioned in Sec. 3.1 and shown in Fig. 5, after the creation of the faulty grid, the TB ID, assigned by the dispatcher to each thread block inside the grid, may be different from the user-defined LBI. The LBI indicates the data portion on which the operations must be executed. Thus, before starting the original kernel execution, the LBI associated with the current thread block is extracted. The extraction is done selecting the *block_list* vector pointed by the TB ID, and reading the LBI field of the pointed structure. Then, the input data set is indexed using the extracted LBI. This ensures the execution of the original kernel on the correct data portion.

Finally, the *done* field, in the *block_list* vector, is updated (row 14 in Alg. 2). Since the *block_list* vector is shared between all thread blocks, during writes of its values *race conditions* must be avoided. Thus, the update operation is done exploiting an atomic CUDA instruction called *CAS* [18], which performs an atomic test-and-set operation.

4 Fault Mitigation Strategies

The proposed fault mitigation strategies allow to use a CUDA GPU which includes faulty modules, in a safety way. In the follow we present two fault mitigation strategies, namely *Sleep* and *Wait*, highlighting their pro's and con's.

4.1 Sleep

The *Sleep* fault mitigation strategy avoids the incorrect execution of a thread block by performing a procedure that sleeps the faulty SM. Whenever a thread block is dispatched to a faulty SM, the *Avoid_Execution()* (see Algorithm 2) function is executed. The *Sleep* strategy is based on the execution of a *For loop* to sleep the faulty SM.

The sleep time (i.e., the period of the *For loop*) depends on the number of iterations executed in the loop, defined by the programmer. Furthermore, it is possible to unroll the *For loop* by using the *#pragma_unroll* directive, generating a sequence of *NOP*. This operation decreases the level of required functionality on the faulty SM.

Clearly, if the sleep time is short, many thread blocks will be dispatched to the faulty SM. Otherwise, less thread blocks are allocated to the faulty SM.

With a short sleep time, the performance overhead introduced by the *Sleep* procedure is low, since the SM is quickly released from the loop procedure. Thus, the cost induced to complete the execution of the current grid is low, even if the number of blocks executed on the faulty SM increases, leading to a bigger faulty grid. Thus, several blocks will have to be re-executed.

With a long sleep time, the procedure introduces a performance overhead to complete the execution of the grid, since the SM is not released quickly (i.e., the faulty block sleeps for more time). Nevertheless, the faulty grid is smaller, because few thread blocks are dispatched to the faulty SM and have to be re-executed.

Thus, the optimal sleep time strictly depends on the dimension of the thread block. The best case is achieved whenever the sleep time equals the overall time required to execute all thread blocks executed in the fault-free SMs. In this case, both timing overhead and faulty grid size (i.e., thread blocks executed on faulty SMs) are minimized. Another key parameter, that must be carefully quantified, is *n_replica*. It strongly influences the number of faulty grids (i.e., the number of kernel execution) required to perform the error-free execution of the complete kernel. Mathematically, to ensure the complete error-free execution of a kernel on a faulty CUDA GPU, (1) must be respected:

$$Grid\_size - (Faulty\_SM * BpSM) > Faulty\_Block \tag{1}$$

where BpSM is the maximum number of thread blocks that can be allocated on a SM (i.e., this number is fixed for each CUDA GPU [19]), and *Grid_Size* is equal to:

$$Grid\_Size = Faulty\_Block * n\_replica \tag{2}$$

From (1) and (2), one deduces that:

$$n\_replica > \frac{Faulty\_SM}{Faulty\_Block} * BpSM + 1 \tag{3}$$

Thus, in the worst case, *n_replica* is:

$$n\_replica = Faulty\_SM * BpSM + 1 \tag{4}$$

Since the *Sleep* strategy cannot guarantee a fixed number of faulty blocks, experimentally it is possible to demonstrate that (4) is valid only if the number of faulty blocks is lower than the number of faulty SMs. Otherwise, *n_replica* must be fixed to 2.

4.2 Wait

The *Wait* fault mitigation strategy adopts a different approach to implement the *Avoid_Execution()* function (see Algorithm 2). The basic idea is to stop the thread block executed on the faulty SM, until all thread blocks dispatched on the fault-free SMs are completely executed. In this way, it is possible to ensure that the total number of faulty blocks is fixed to:

$$Faulty\_Blocks = Faulty\_SM * BpSM \tag{5}$$

This ensures that *n_replica* is fixed to 2, also.

To enable this kind of execution avoidance, the *Execution Configuration* structure must be modified by adding two fields: *target_done* and *num_done*.

The former identifies the number of thread blocks that must be executed. When the instrumented CUDA program (see Subsection 3.1) starts the instrumented kernel for the first time, this parameter is set to the number of thread blocks in the grid.

Otherwise, when a faulty grid is executed, *target_done* is set to the number of faulty blocks. The latter enumerates the thread blocks executed on the fault-free SMs.

These two fields are exploited to ensure stopping of the faulty SMs for the proper amount of time, to avoid performance overhead. Basically, when a thread block has finished its execution on a fault-free SM, it increments the *num_done* field. This operation is performed using an atomic add operation (i.e., the CUDA API called *atomicInc*) to avoid race conditions on the *num_done* field, shared among all thread blocks.

Whenever a thread block is dispatched on a faulty SM, it waits until all thread blocks dispatched on fault-free SMs are completed. This operation is performed using a *while* loop, that iterates until *num_done* reaches *target_done*. Before starting the *while* loop, each thread block dispatched on a faulty SM must decrement the *target_done* field. Without this operation, the value of *num_done* could never reach the value of *target_done*. Since the reduction must be performed just once (i.e., for each thread block executed on faulty SM), the *target_done* is decremented only by the first thread contained in the thread block.

4.3 Comparison

An accurate comparison between the two proposed fault mitigation techniques is required to understand whether the one should be preferred to the other.

On the one hand, the *Sleep* fault mitigation strategy has two main pros: (i) it only requires that each faulty SM is able to perform a for loop (or a sequence of *NOP*), and (ii) it does not require any information on the thread block scheduling policy to be applied.

The biggest con concerns the number of faulty grids that must be executed on the CUDA GPU to ensure the complete correct execution of the kernel. This number can increase depending on the number of faulty SMs. Thus, the execution of a kernel on a faulty CUDA GPU could require more than one faulty grid.

On the other hand, the *wait* fault mitigation strategy guarantees a fixed number of faulty blocks (see (5)). If *n_replica* is set to the value obtained from (4), it is possible to guarantee a complete error-free execution of the kernel executing one faulty grid, only.

Thus, the *Wait* fault mitigation strategy, unlike the *Sleep* strategy, requires just two iterations of the kernel (i.e., the first kernel execution and the execution of the faulty grid). However, it requires the execution of more complex operations on faulty SMs. In fact, each faulty SM must be able to execute the *while* loop, to perform the atomic subtraction,

Table 1: Execution Time and Performance Gain associated with the fault mitigation strategies

# Faulty	Matrix Multiplication					Histogram Computation				
	CPU	Sleep		Wait		CPU	Sleep		Wait	
SMs	[ms]	GPU [ms]	PG	GPU [ms]	PG	[ms]	GPU [ms]	PG	GPU [ms]	PG
0	9,850	507	19,4	471	20,9	627	48	13.1	49	12.8
1	9,850	582	16.9	540	18.2	627	65	9.7	64	10.3
2	9,850	678	14.5	631	15.6	627	88	7.1	72	9.2
3	9,850	817	12.1	756	13.0	627	99	6.3	82	7.8
4	9,850	1,024	9.6	942	10.4	627	114	5.5	97	6.5
5	9,850	1,364	7.2	1,258	7.8	627	167	3.8	132	4.8
6	9,850	2,044	4.8	1,885	5.2	627	230	2.7	194	3.2
7	9,850	4,089	2.4	3,768	2.6	627	450	1.4	380	1.6

and to identify the first thread of a block.

Thus, the programmer should first evaluate the capability of the faulty SMs. If they are able to perform the operations needed to execute the *Wait* function, this mitigation strategy should be preferred since it guarantees the lowest performance overhead. Otherwise, the *Sleep* strategy should be used, thanks to the lowest required capability of faulty SMs.

5 Experimental Results

A set of experiments has been performed to prove the effectiveness of the proposed mitigation strategies. The CUDA GPU used during the tests is a *Gigabyte GeForce GTX 560Ti*, equipped with 1 GB of dedicated RAM and 8 Streaming Multiprocessors (SM). An Intel Core i5-2500k CPU is used as CPU.

The fault mitigation strategies have been applied to two applications available in the CUDA SDK: *Matrix Multiplication* and *CUDA Histogram*. The former performs the multiplication between two 2048x2048 input matrix. The latter sorts 335,544,320 char data into 256 bins. Input data, for both applications, are generated randomly with a uniform distribution, exploiting the standard *rand()* C/C++ function. Both applications execute the same computation on the CPU (i.e., exploiting Multi-Thread (MT) implementations) and on the CUDA GPU, so the performance of the two platforms can be compared and the correctness of the results can be checked.

CUDA programs and kernels have been instrumented according to the methodologies presented in Section 4. In addition, in order to find the best sleep time (see Subsection 4.1) several test executions have been carried out. Eventually, the sleep time has been set to: 10,000 cycles for the matrix multiplication and to 65,000 cycles for the histogram computation. Each application has been tested in 8 different conditions, with different number of faulty SMs (i.e., changing the *faulty_sm_map* vector), in order to characterize the execution times related to the CPU and GPU implementation.

Table 1 lists the execution times for both test algorithms. The comparison is made between the execution times of the CPU and GPU algorithm 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. to a CPU.

Clearly, the execution time on the GPU increases with the number of faulty SMs (see Fig. 6 and Fig. 7).

From the reported graph it can be noticed that the *Wait* strategy offers slightly better performance than the *Sleep* strategy, especially when the number of faulty SMs increases.

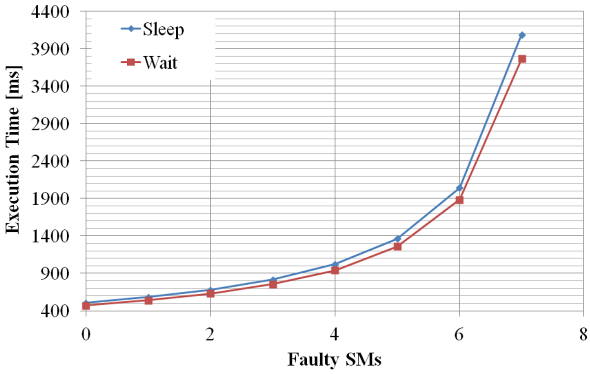


Figure 6: Fault mitigation Strategies comparison - Matrix Multiplication

6 Conclusion

The paper presents an innovative methodology to allow the use of a CUDA-based GPU, even in presence of faulty streaming multiprocessors. While the other already pre-

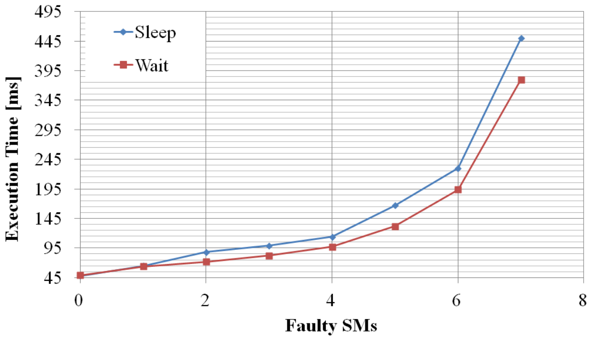


Figure 7: Fault mitigation Strategies comparison - Histogram Computation

sented methodologies are algorithm dependent or introduce performance penalty without faults, also, the two presented fault mitigation techniques are completely algorithm independent and they allow to reach the maximum performance during CUDA fault-free execution.
Moreover, a validation campaign has been performed to assess the graceful performances degradation granted by the proposed techniques in presence of faults.

References

[1] H. Kimm and H. sang Ham, "Integrated fault tolerant system for automotive bus networks," in *Proceedings IEEE Int. Computer Engineering and Applications Conf.*, pp. 486–490, 2010.

[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. Zhang, "Investigation of fault-tolerant adaptive filtering for noisy ecg signals," in *Proceedings IEEE Symp. on Computational Intelligence in Image and Signal Processing*, pp. 177 –182, april 2007.

[4] nVidia, *nVidia's Next Generation CUDA Computer Architecture: Fermi*, 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*, pp. 137 –145, 1995.

[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 *Proceedings IEEE Int. Conf. on Application-Specific Systems, Architectures and Processors*, pp. 141 –148, 2011.

[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)*, pp. 1895–1899, 2010.

[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.

[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.

[13] C. Ding, C. Karlsson, H. Liu, T. Davies, and Z. Chen, "Matrix multiplication on gpus with on-line fault tolerance," in *Proceedings IEEE Int. Symp. on Parallel and Distributed Processing with Applications*, pp. 311–317, 2011.

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

[15] nVidia, *nVidia Tesla - GPU Computing Technical Brief*, 2007.

[16] nVidia, *nVidia Kepler GK110 Next-Generation CUDA Compute Architecture*, 2012.

[17] 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 *Proceedings IEEE European Test Symp.*, pp. 33–38, 2013.

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

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