Neutron Sensitivity and Software Hardening Strategies for Matrix Multiplication and FFT on Graphics Processing Units

Paolo Rech
Instituto de Informática
UFRGS
Av. Bento Gonçalves
Porto Alegre, RS, Brazil
prech@inf.ufrgs.br

Laercio L. Pilla
Instituto de Informática
UFRGS
Av. Bento Gonçalves
Porto Alegre, RS, Brazil
llpilla@inf.ufrgs.br

Francesco Silvestri
Dip. Ingegneria dell'Informazione
Università di Padova
via Gradenigo 6B
35131 Padova, Italy
silvest1@dei.unipd.it

Philippe O. A. Navaux
Instituto de Informática
UFRGS
Av. Bento Gonçalves
Porto Alegre, RS, Brazil
navaux@inf.ufrgs.br

Luigi Carro
Instituto de Informática
UFRGS
Av. Bento Gonçalves
Porto Alegre, RS, Brazil
carro@inf.ufrgs.br

ABSTRACT
In this paper, we compare the radiation response of GPUs executing matrix multiplication and FFT algorithms. The provided experimental results demonstrate that for both algorithms, in the majority of cases, the output is affected by multiple errors. The architectural and code analysis highlight that multiple errors are caused by shared resources corruption or thread dependencies. The experimental data and analytical studies can be fruitfully employed to evaluate the expected error rate of GPUs in realistic applications and to design specific and optimized software-based hardening procedures.

Categories and Subject Descriptors
B.8.1 [**Performance and Reliability**]: Reliability, Testing, and Fault-Tolerance.

Keywords
GPU, radiation effects, parallel architectures sensitivity, software-based hardening.

1. INTRODUCTION
Graphic Processing Units (GPUs) are electronic devices designed to perform high-performance stream processing typically used in desktop computers, laptops or portable devices to accelerate graphics rendering. In order to achieve the proposed objective, GPUs manipulate a large number of memory locations, and are typically able to execute several elementary tasks in parallel at high speeds [1][2].

Due to their highly parallel structure, GPUs are more effective than general-purpose CPUs when large blocks of data need to be processed in parallel. GPUs have recently become popular for high performance computing applications in which parallel algorithms are employed, such as oil exploration, air traffic flow analysis, medical image processing, linear algebra, statistics, 3D reconstruction, and stock options pricing determination [3][4]. Moreover, thanks to their high computing power, GPUs are used in modern supercomputers like TITAN, which is composed of 18,000 GPUs [5].

Modern GPUs are cutting edge processors built with novel technologies and, thus, may be very prone to experience radiation-induced failures. We have already demonstrated in [6][7] that radiation-induced errors, including from the terrestrial neutron radiation environment, are one of the major issues for the newest GPU cores reliability. While CPU radiation responses, test procedure, and hardening techniques are well documented, and standardized [8][9], only few papers describe possible radiation test methods for extreme parallel systems and fewer analyze parallel algorithms behaviors in radiation environments.

Unfortunately, the experimental data presented here demonstrate that, when both matrix multiplication and FFT are concerned, most of the corrupted executions are affected by multiple errors, hence refuting the traditional hypothesis that just single output errors may occur in extreme parallel computing systems. Having multiple output errors is an extremely tricky situation to deal with. On one side, as they are unexpected, multiple output errors may remain undetected, seriously compromising the system dependability. On the other side, most of the available hardening techniques for parallel algorithms are based on the assumption that just one single error affects the output, and may become ineffective or inefficient when multiple errors occur [10]. The purpose of this work is to take advantage of experimental results, algorithm analyses, and architectural studies, to identify the causes of multiple output errors and propose dedicated and efficient hardening strategies to correct them.

FTXS'13, June 18, 2013, New York, NY, USA.

In this paper we analyze the radiation experiments results of extensive test campaigns performed at ISIS, Rutherford Appleton Laboratories, Didcot, UK, and at LANSCE, Los Alamos National Laboratories, Los Alamos, NM, USA. The performed experiments allow the realistic evaluation of the output error rate of a representative set of classical applications in high performance computing executed on GPUs exposed to the natural neutron flux. We will compare the experimental results of matrix multiplication with the FFT ones, giving particular attention to the different causes of multiple errors. As detailed in the following section, matrix multiplication is composed of independent threads but heavily employs cache memory, which is shared among several threads. A radiation-induced error in the cache is then likely to affect the execution of various threads. On the contrary, FFT computation requires sequential iterations and, thus, a thread output may depend on previously executed threads. If a thread fails, the error is then likely to spread over the following threads. Finally, we compare the algorithmic structures and analyze the GPU architecture to propose dedicated and efficient software-based approaches and programming guidelines to lower the impact of multiple errors on massively parallel system and avoid the radiation-induced failure to affect the output.

The hardening philosophy we follow is based on the Algorithm Based Fault Tolerance (ABFT) technique that exploits the properties of the computational problem and of the adopted algorithmic approach to efficiently detect and, whenever possible, correct silent faults. We extend the ABFT solution designing dedicated and optimized procedures for the detection and correction of multiple output errors. Moreover, the correction capability of the technique can be tuned with the experimentally evaluated error rate so to prevent the introduction of useless overheads and avoid excessive performance degradation, which is essential in high performance computing applications.

The remainder of the paper is organized as follows. Section 2 introduces the GPU structure, and the possible radiation-effects on its internal resources. Section 3 describes the neutrons spectra and fluxes, the experimental setup, and the tested algorithms. Section 4 and section 5 describe the matrix multiplication and FFT algorithms, respectively, discussing the obtained experimental results, highlighting the presence and causes of multiple output errors, and propose dedicated hardening procedures, while section 6 concludes the paper.

2. GPU INTERNAL STRUCTURE

GPUs are divided into various computing units, named *Streaming Multiprocessors* (SM), each of which has the ability to execute several threads in parallel (see Figure 1). Each basic computing unit (named *CUDA core* in NVIDIA devices) in the SM executes one thread with dedicated memory locations, avoiding complex resource sharing or the need of long pipelines [2].

It is the programmer's task to divide the instantiated threads into a *grid of blocks*, and each SM in the GPU will treat a block of threads at a time (see Figure 2). Thus, some blocks will be queued for later computation if the grid size exceeds the number of SMs available in the GPU. Before assigning a queued block to the first SM that becomes available, the GPU scheduler needs to check if some SM completed the current block execution and, if so, it transfers the results to the on-board DDR memories. The queued block is then assigned to the SM, input data are read from the DDR, and, finally, the queued block execution is triggered and synchronized [11].

Generally, on modern GPUs each SM can execute a *warp* of up to

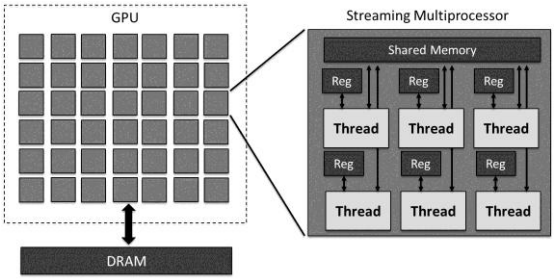

Figure 1: Simplified internal structures of a GPU.

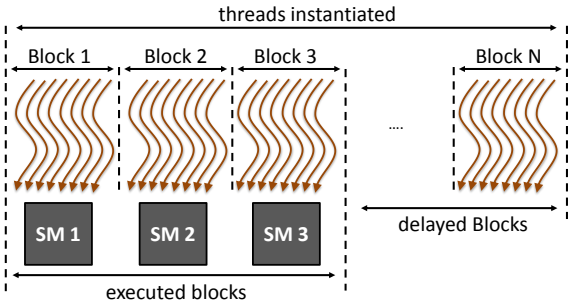

Figure 2: The instantiated threads are grouped into a grid of blocks. Each of the available Streaming Multiprocessor (SM1, SM2, and SM3) treats just one block at a time.

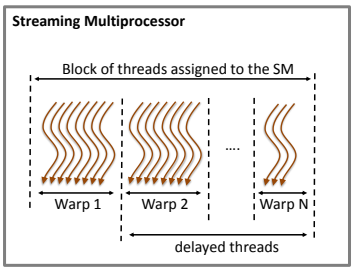

Figure 3: The block of threads assigned to a SM is divided into warps. Just one warp of threads is executed at a time in the SM.

32 parallel threads in a computing cycle. If the block size exceeds 32, some threads execution will be delayed until the computation of the preceding warps of the block has been completed (see Figure 3). It is worth noting that the next block to be treated will be assigned to the SM when all the threads in the current block have been processed. So, if the number of threads in a block is not a multiple of 32, in the last cycle the SM will execute less than 32 threads (warp N in Figure 2), wasting parallel capabilities.

A neutron striking a CUDA core may generate a functional interruption, preventing the thread assigned to it from completing its computation, corrupt an internal register or disturb the logic gates that are calculating an operation, leading the thread to produce a silent fault. CUDA cores are isolated such that a radiation-induced event in one CUDA core will only corrupt the thread assigned to it. Threads that follow the corrupted one or threads assigned to CUDA cores near the struck one will not be affected.

Even thought CUDA cores are isolated, the corruption of resources shared among various threads may lead the GPU to

experience multiple output errors. This occurs, for instance, when various threads in a SM use as input the same data stored in the SM cache or in the GPU shared memory, which is the case of matrix multiplication, as described in section 4. If radiation corrupts shared data, all threads using it as input will produce a faulty output. In some algorithms, like the Fast Fourier Transform (FFT), several iterations are necessary to conclude the calculation. At each iteration, threads use the output of previously executed threads as an input. A neutron-induced error on a thread in one iteration will then propagate over the following ones, as detailed in section 5.

GPU schedulers are needed to trigger and synchronize threads execution, to check if computation is complete, and, if so, to pipeline the exceeding threads allocated [11]. A scheduler is a critical resource, as its radiation-induced failure has severe repercussion on the system functionalities [10]. Potentially, the threads handled by a corrupted scheduler may produce a wrong output. However, if some of the handled threads already finished computation when the scheduler is struck, just the remaining threads will present an incomplete, and thus faulty, result.

On a reliability point of view, it is feasible to avoid cache corruption from affecting computation adding an Error Correction Code (ECC), while adding redundancy on the scheduler will require costly modifications to the GPU physical structure. As we will demonstrate in the following sections, cache corruption normally generates detectable output patterns that could be efficiently corrected with software-based hardening strategy, while a scheduler failure is likely to lead to random output errors, which are not always correctable and, even when possible, output correction requires a great computation effort [10].

3. EXPERIMENTAL METHODOLOGY AND DEVICES

3.1 Tested Devices

The Devices Under Test (DUT) were commercial-off-the-shelf GeForce GTX480 GPUs designed by NVIDIA in a 40nm technology node [12] and can run with a maximum frequency of 1.215GHz. The DUT is composed of 15 Streaming Multiprocessors, and disposes of 480 CUDA cores (32 for each SM). For the GTX480 GPU, 15 blocks of threads can be executed in parallel with a maximum of 32 threads in each block for a total of 480 threads. If more threads or blocks are instantiated, their execution will be delayed until they can be scheduled.

3.2 Neutron fluxes and spectra

Experiments were performed at Los Alamos National Laboratory's (LANL) Los Alamos Neutron Science Center (LANSCE) Irradiation of Chips and Electronics House II, called ICE House II, in September 2012 and at VESUVIO, in ISIS, Rutherford Appleton Laboratories, Didcot, UK, in December 2012. Both of these facilities provide a white neutron source that emulates the energy spectrum of the atmospheric neutron flux (see Figure 4).

The available neutron flux was approximately $1x10^6$ n/(cm^2·s) at LANSCE and $4x10^4$ n/(cm^2·s) at ISIS for energies above 10MeV. The flux denotes the number of particles hitting the device per unit area and time. The higher the flux is, the higher the probability of observing output errors in the GPU. The flux used for tests allows one to obtain a statistically significant amount of errors in a short time. As we will detail in the following section, the experimental setup was tuned in order to prevent more than

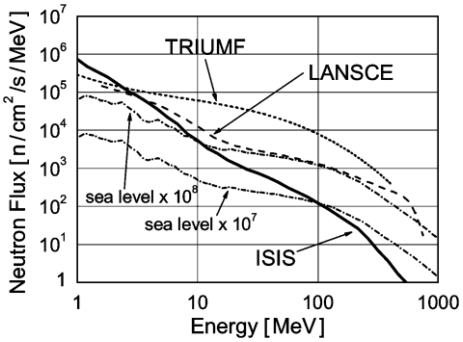

Figure 4: ISIS and LANSCE spectrum compared to those of the TRIUMF facilities and to the terrestrial one at sea level multiplied by 10^7 and 10^8 [14].

one neutron from corrupting the device during the same code execution. This is necessary to avoid the occurrence of multiple errors caused by the high flux used during experiments and, so, the experimentally measured error rate can be scaled to the realistic error rate caused by the much lower atmospheric neutron flux.

The beam was focused on a spot with a diameter of 2cm plus 1cm of penumbra, which provided uniform irradiation of the GPU chip without affecting nearby DDR memories. Even if the LANSCE and ISIS beams are well focused, some thermal neutrons (i.e. low energy neutrons) may still be produced by scattering and may collide with devices in the proximity of the beam azimuth, possibly causing failures if the struck device includes boron-10 [15]. The DDR content was periodically checked during the experiments, and no radiation-induced error was ever found. It is worth noting that input and output data were stored in the DDR, so the observed errors were only caused by the corruption of logic resources or internal memories or registers. Irradiation was performed at room temperature with normal incidence and nominal voltages.

3.3 Test Procedure

The Device Under Test (DUT) was controlled by a desktop PC through a 2.5GHz PCI-Express bus. The PC was kept out of the beam using a 20cm GPI-Express bus extension and shielded by boron plastic panels to avoid errors induced by scattering of thermal neutrons from the beam to affect its functionalities and to better align the DUT with the neutron beam. Moreover, the PCI-Express extension was provided with fuses on power lines, so to prevent eventual radiation-induced latchups on the GPU to propagate to the PC motherboard. Operating voltage was provided externally with a current-controlled power supply to the DUT in order to promptly cut power in the event of latchup. It is worth mentioning that we never observed any destructive latchup on the irradiated GPU even after several week of irradiation. The tested GPUs can then be considered immune to neutron-induced latchups.

The only role of the CPU during the test is to initialize the GPU and gather experimental results. The procedure designed for experiments is divided in three parts:

1. *Initialization*: the CPU loads data and instructions in the GPU.
2. *Test*: the GPU runs the instruction; the test is actually performed while the CPU is in idle state.
3. *Readback*: test results are transferred from the GPU to the CPU and checked.

Thanks to the extreme high frequencies of operation of both the PCI-express bus and the CPU, steps 1 and 3 are performed so quickly (order of ms in the worst case) that one can consider the probability of a neutron to generate an error during their execution negligible (observed error rates were lower than 0.1 errors/s). Steps 1 to 3 were then performed repetitively under the neutron beam to gain a statistically significant amount of errors.

4. MATRIX MULTIPLICATION (*MxM*)

4.1 *MxM* Algorithm Description

We executed on the irradiated GPU an algorithm that multiplies two matrices (A and B) composed of 2048x2048 random double precision floating-point elements taking full advantage of the GPU parallelism capabilities [11]. The algorithm, named *MxM*, instantiates 2048x2048 parallel threads, each of which is in charge of calculating a single element of the resulting matrix M following Equation 1.

$$M[i,j] = \sum_{k=1}^{n} A[i,k] \cdot B[k,j] \qquad 0 < i,j,k < n \tag{1}$$

Where n is the number of rows and columns of the matrices A, B, and M.

During the code execution, threads can be considered independent from each other, as the thread output is evaluated using just the input data, not the result of other threads calculations. To limit execution latency, the SM stores in its cache the data required by the threads it hosts.

As stated in section 3, the GTX480 can execute up to 480 threads in parallel. Thus, not all the threads instantiated in *MxM* will be physically executed in parallel. *MxM* computation will then require several different threads and blocks allocations to be completed, exacerbating scheduler employment. The presented results were obtained grouping threads in blocks of size 1024. Results with different block sizes can be found in [16]. The matrices dimension were chosen big enough to allow the gathering of a statistically significant amount of data in a short time, but yet small enough so that one can be reasonably sure that at most one neutron per *MxM* execution generates errors (observed error rates were lower than $3x10^{-2}$ errors/execution). The observed multiple output errors were then not caused by the high particle flux used during experiments, but rather by the corruption of GPU critical resources and so are likely to occur also when the GPU is irradiated with the natural neutron flux.

4.2 *MxM* Experimental Results Analysis

The first column of Table I encloses the experimentally measured cross section for *MxM* executed on the GTX480 with double floating point data. The experimental cross section for *MxM* executed with integer data is $2.22 \cdot 10^{-6}$, as reported in [10]. The *MxM* cross section is obtained dividing the observed output errors per unit time by the flux (i.e., the number of neutrons hitting the device per unit time and unit area). The cross section is then the probability for a neutron that hits the GPU executing *MxM* to produce an output error. The output error rate of *MxM* on a realistic application can be evaluated multiplying its cross section by the expected amount of neutrons that hit the device in the environment of interest. Table I also reports the Failure In Time (FIT), i.e. the number of errors in 10^9 hours of continuous operation, expected at New York City when *MxM* is executed with double precision floating point data. The FIT is calculated multiplying the *MxM* cross section experimentally measured and the natural neutron flux in New York (i.e., 14 n/(cm^2·h) [17]). The

TABLE I
MXM CROSS SECTION AND FIT AT NYC

	Cross section	FIT
MxM GTX480	$2.01 \cdot 10^{-6}$cm^2	$2.81 \cdot 10^4$

TABLE II
MXM SINGLE AND MULTIPLE ERRORS PERCENTAGES AND FIT

	Single	Multiple	Row/Col	Random
Percentage	48.48%	51.51%	47.47%	4.04%
FIT [10^3]	13.60	14.50	13.36	1.14

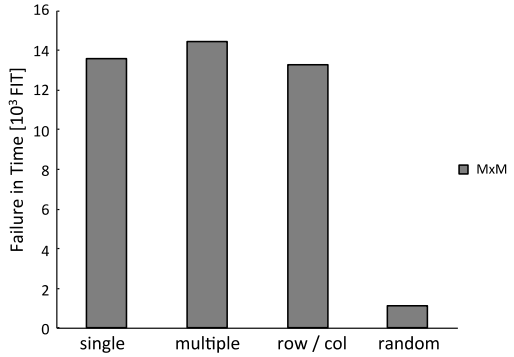

Figure 5: *MxM* output errors patterns FIT at NYC when executed with double data.

evaluated FIT is realistic and precise as the spectrum of neutron energies at ISIS and LANSCE resembles the atmospheric one (see Figure 4). The experimentally evaluated FIT of *MxM* is $2.81 \cdot 10^4$, which equals of about one error every 3.5 years. The *MxM* neutron-induced error rate may seem reasonable for entertainment applications or video/audio editing. However, the GPU FIT is extremely high for safety critical applications, in which a high reliability is required. For instance, the car pedestrian detection system, designed for promptly activating the vehicle brakes in order to prevent the collision with pedestrians, is actually being implemented with a GPU. In the event of radiation-induced failure, the GPU may suddenly trigger the brakes, possibly causing accidents. Moreover, the GPU FIT may be unacceptable for supercomputers, in which thousands of GPUs run in parallel, as the probability of having at least one GPU corrupted is very high. It is then mandatory to design and evaluate hardening strategies for GPUs.

The first row of Table II report the percentage of corrupted matrices affected by single and multiple errors. As one can see, in the majority of cases multiple errors occur when *MxM* is executed on a GPU. We can further study the observed phenomena analyzing the different multiple error patterns detected in the output matrix. As reported in the last two column of the first row of Table II, in most of the cases multiple errors are distributed on a single row or column of the resulting matrix. Errors on the same row/column are due to cache corruption. As demonstrated in [11], for evaluating a row *MxM* uses the same row of input matrix A (see Eq. 1), which is stored in the cache to avoid multiple accesses to the DDR. If that row is corrupted, all threads using its values will produce a wrong output. The same considerations apply to column. It is very unlikely to have more than 10 errors in a row or column and more than 32 errors were never observed in our experiments [10]. This is because not all the threads in the SM whose cache has been corrupted are calculating elements on the same row or column.

The causes for randomly distributed errors could be various. For instance, pairwise errors in the cache flags or multiple cells upset in the cache array may lead various threads to work with corrupted data and, thus, to produce an output error. As stated in section 2, a scheduler failure is likely to affect multiple thread executions. The scheduler is in charge of designating the block of threads that has to be executed in a Streaming Multiprocessor, and of detecting if all threads have completed the assigned computation. If so, results are presented at the output and another block of threads is triggered for execution. In the event of scheduler corruption, some results may be transferred to the output even if some threads have not yet completed their computation, leading to a faulty output matrix. The same considerations apply to the scheduling necessary when the number of threads inside a block exceeds 32 (i.e., the highest amount of threads a Streaming Multiprocessor can execute in parallel). Threads affected by scheduler failure may be calculating different locations of the output matrix, not necessarily on the same column of row, leading to randomly-distributed multiple errors. Unfortunately, no detailed information is currently available on the scheduler structure. Nevertheless, radiation tests of GPUs varying scheduler strains show that the occurrence of randomly distributed errors is lower when an efficient scheduling is exploited [16]. This empiric result supports the statement that the scheduler is a critical resource for the GPU and its corruption may indeed lead to randomly distributed errors. In the future, further tests are going to be performed to investigate the probability of occurrence and the effects of scheduler failures in a parallel algorithm output.

The occurrence of different error distributions on a real application is evaluated by calculating the FIT at New York City for all the observed error patterns (see second row of Table II). As shown in Figure 5, a consistent portion of the observed multiple errors is due to cache corruption. Cache bits can be protected adding ECC. To mitigate the radiation effects on their devices, NVIDIA actually introduced in the latest GPUs an ECC able to correct single errors and detect double errors on the same word. Nevertheless, ECC will not detect errors affecting the scheduler or the logic resources nor functional interruption on a thread or on a SM. Furthermore, the ECC introduced overhead is far from being negligible. In fact, when the NVIDIA ECC is enabled, the amount of user accessible memory (DDR and internal cache) is reduced by 12.5% [13]. Moreover, depending on the algorithm and device, typically the ECC reduces the GPU performances in the range of 20-30% [18]. If the reliability ensured by ECC or the overhead introduced by ECC are found to be unacceptable for an application, programmers can exploit software-based hardening techniques to correct radiation-induced errors. Some hardening techniques for matrix multiplication are presented and compared in [11]. In particular, as cache corruption produces multiple errors on the same row or column of the output matrix, errors can be corrected in constant time by adding checksums in the input matrices extending the Algorithm Based Fault Tolerance (ABFT) technique proposed in [19], based on Freivalds' results [20].

As can be seen in Figure 5, random multiple errors are less likely to occur then single errors or errors on a row or column. Nevertheless their occurrence is definitely not negligible ($1.14 \cdot 10^3$ FIT, see last column of Table II) and, mostly, they are potentially very dangerous. On one side it is not possible to physically harden the scheduler, whose corruption is the main cause for multiple random errors and, on the other, a high amount of random errors are extremely difficult to be corrected with software-based techniques. As demonstrated in [11], in fact, random errors are surely correctable only if no more than 3 locations are found as corrupted in the output matrix. In all the other situations it is necessary to check if correction is possible and, even when possible, output correction requires a great computation effort. Experimentally we never observed more than 4 randomly distributed errors on the output matrix.

TABLE III
FFT CROSS SECTION AND FIT AT NYC

	Cross section	FIT
FFT	$3.69 \cdot 10^{-6} cm^2$	$5.17 \cdot 10^5$

TABLE IV
REAL AND IMAGINARY PART PERCENTAGE, CROSS SECTION, AND FIT

	Percentage	Cross section	FIT
FFT Real	94.96%	$3.50 \cdot 10^{-6} cm^2$	$4.90 \cdot 10^5$
FFT Imaginary	96.17%	$3.55 \cdot 10^{-6} cm^2$	$4.97 \cdot 10^5$

TABLE V
FFT SINGLE AND MULTIPLE ERRORS

	FFT Real		FFT Imaginary	
	Single	Multiple	Single	Multiple
Percentage	1.61%	98.39%	4.00%	96.00%
FIT	$7.89 \cdot 10^3$	$4.82 \cdot 10^5$	$19.80 \cdot 10^3$	$4.67 \cdot 10^5$

5. FAST FOURIER TRANSFORM (*FFT*)

5.1 *FFT* Algorithm Description

The second tested algorithm on the irradiated GPU, named *FFT*, is a 64-points 1D Fast Fourier Transform (FFT) calculated on 512x512 vectors. The *FFT* input is then composed of a 64x512x512 double precision floating-point matrix for the real part and a 64x512x512 matrix for the imaginary part. The implemented algorithm is based on the FT kernel of the NAS Parallel Benchmarks [21] implemented in C and ported to the GPU using CUDA. The executed algorithm performs a 64-points complex-to-complex 1D FFT on each of the 512x512 vectors.

The 64-points 1D FFT kernel is composed of 6 iterations ($\log_2 64=6$) of a variant of the Stockham FFT algorithm [22]. All iterations are executed sequentially in the GPU using two temporary matrices for scratch. At each iteration, the GPU instantiates 512x512 parallel threads, grouped in blocks of 512 threads each. A thread is in charge of evaluating the intermediate FFT values on the assigned complex vector of size 64, updating in each iteration two-by-two the values of all the floating-point elements in the complex matrix.

The *FFT* algorithm is then divided into threads that are not completely independent. In fact, a thread in one iteration takes the output of threads executed on the previous iteration as input. If a thread output is corrupted, in the next iteration a thread will take a faulty value as input and evaluate two intermediate values of the *FFT* with it. A single output error in one thread will then spread in the following iterations, generating multiple corrupted locations in the *FFT* output. The worst case occurs when errors appear in the first iteration. If one of the 64 locations updated by a thread in the first iteration is corrupted one can expect to have $2^5 = 32$ errors in the output (the number of errors doubles at each iteration). If two or more locations are wrongly updated, all the 64 elements of the 1-D FFT will be faulty.

We choose to test 64-points FFT to limit the number of iterations and ease the study of errors propagation in the code, while having 512x512 1D FFTs eases gathering a statistically significant

TABLE VI
FFT REAL AND IMAGINARY SINGLE AND MULTIPLE ERRORS PERCENTAGES

	1	2	4	6	8	12	14	16	18	19	22	24	26	27	28	30
Real	1.63	6.91	5.28	5.28	4.47	4.47	2.44	2.85	0.81	0.40	0.81	5.29	0.40	0	6.50	6.91
Imaginary	4.00	7.20	6.40	4.00	4.00	4.40	3.60	3.60	0.40	0.40	0.80	4.00	0.40	0.40	6.40	6.80

	32	34-39	40	42	44	48	52	56	60	62	64	66-126	128	>130
Real	8.94	0	1.63	0.81	0.80	1.63	0.80	2.40	1.63	4.88	8.54	5.60	0.41	3.66
Imaginary	9.20	1.20	0.40	0.82	0.80	2.00	0.41	2.01	1.21	4.44	8.80	3.20	1.60	4.00

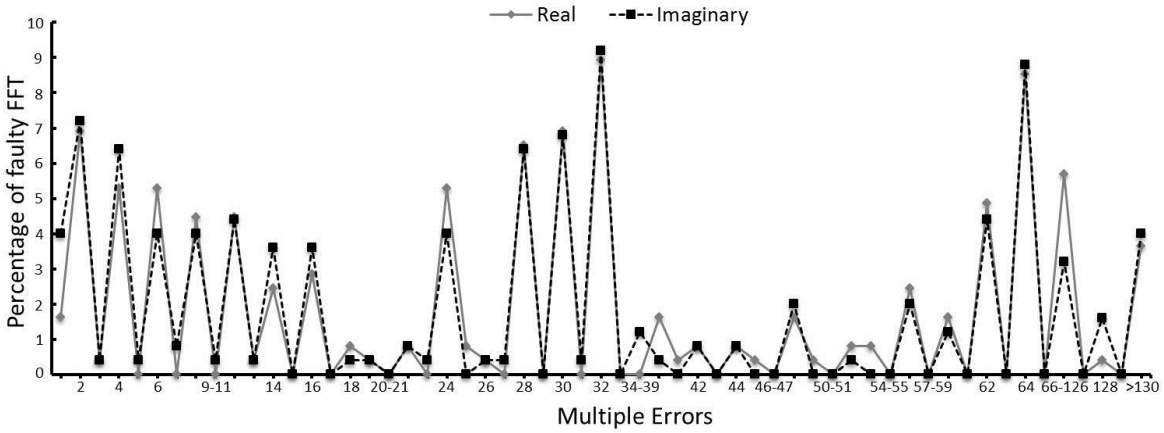

Figure 7: *FFT* real and imaginary multiple output errors percentages. Some distributions that were never experimentally observed are grouped in the picture (it is the case of 9 to 11 errors, 20 and 21 etc.). In most of the cases, the output is affected by less then 16 errors, about 32 or 64 errors. It is very unlikely to have an odd number of errors or more than 128 errors.

amount of errors. Each thread works over its own sequential vector in memory to complete the 1D 64-point FFT, thus no shared cache memory is used. Vector dimensions were chosen small enough to be reasonably sure that at most one impinging neutron per *FFT* execution generates errors (observed error rates were lower than $2x10^{-2}$ errors/execution). The observed multiple output errors were then not caused by the high particle flux used during experiments.

5.2 *FFT* Experimental Results Analysis

Table III reports the experimentally measured cross sections and the FIT for *FFT*. *FFT* gives complex double data as an output, divided in real and imaginary parts. An execution is considered as faulty if at least one error is detected in the real, imaginary or both part of the output. The cross section is obtained dividing the number of faulty executions per unit time by the flux.

The experimentally evaluated FIT is $5.17 \cdot 10^5$, which is higher than the *MxM* one. Even if executed on the same GPU, *FFT* and *MxM* have different throughput and execution time, which may indeed turn into different radiation-induced error rates. In particular, the *FFT* elaborates 8 times more data than *MxM* (64x512x512 complex elements in the *FFT* versus 2048x2048 real elements in the *MxM*), but instantiate less threads (512x512 for each of the 6 iterations in the *FFT* versus 2048x2048 in the *MxM*). The computational effort required by each thread to complete the assigned task is higher in the *MxM* algorithm, as each thread executes 2047 sums and 2048 multiplications, while a thread in one iteration of the *FFT* requires 3 sums, 3 subtractions, and 4 multiplications, for each couple of values to update, for a total of 96 sums, 96 subtractions and 128 multiplications.

The implemented algorithm is symmetric for the real and imaginary parts. One thread, in fact, uses the same operations and the same amount of internal registers to calculate both the real and imaginary part of an element, updating two output values using two complex numbers as an input. Nevertheless, some executions experienced errors just in the imaginary part or just in the real part. Table IV shows the percentage of faulty execution of the *FFT* algorithm in which the real or imaginary part was detected as corrupted as well as the cross section and FIT of just the real and imaginary part. As stated in the second column of Table IV, in less than 5% of the executions considered as faulty no error was detected in the real part. This means that in those executions the *FFT* experienced an error just in the imaginary part. The same considerations apply to errors in just the real part, which are less than the 4% of the overall faulty executions. Errors in just the real or imaginary part are due to the corruption of internal registers used by the thread for storing the intermediate values of the complex number.

Dissimilarly to *MxM*, the *FFT* algorithm is composed of threads that are not independent, as a thread uses the output of previously executed threads to update the real and imaginary part of the complex output. It is then very likely that a radiation-induced event affecting a thread in the early stages of the *FFT* execution will spread, affecting various location of the output. Experimental results support this assertion. In fact, as stated in Table V, just 1.41% of the faulty executions have a single error in the real part and 4% have a single error in the imaginary part. The importance

of the occurrences of multiple errors in realistic applications is highlighted in the last row of Table V, in which the FIT of *FFT* affected by single or multiple errors in the real and imaginary parts are reported.

Table VI lists the percentage of *FFT* faulty executions affected by the different multiple errors distributions for both the real and imaginary part. Distributions observed in none or in less than the 0.4% of the faulty executions are not listed in Table VI.

As shown in Figure 7, it is very unlikely to have an odd number of errors in the real or imaginary part of the output. Actually radiation can lead the thread to produce a single output error corrupting, for instance, the internal register that stores the value of just one of the elements to update or disturbing just one of the operations needed to calculate the *FFT*. If such a situation occurs, the thread can complete its execution, allowing the correct calculation of the remaining complex numbers. However, even if radiation produces a single output error in a thread, in the consequent iteration two complex numbers will be updated using the corrupted value, leading to an even amount of errors in the final output. Single error in the output occurs when radiation leads a thread in the last iteration to produce just one faulty complex value. As stated in Table VI, this occurred in just 1.63% of the faulty executions for the real and in 4% of the faulty executions for the imaginary part.

The multiple errors distributions shown in Figure 7 are explained considering how errors propagate from one iteration to the following ones. As said, the amount of errors doubles at each iteration. Let e_i be the amount of errors in the i-th iteration. The amount of error in the j-th iteration, named e_j is evaluated through Equation 2.

$$e_j = 2^{j-1} \cdot e_i \qquad 0 < i < j \leq 6 \tag{2}$$

The worst case occurs when radiation affects the execution of a thread in the first iteration. As 6 iterations are necessary to complete the *FFT* execution, if a single error occur in the first iteration, the amount of output errors is going to be $e_6 = 2^5 \cdot e_1$ or $e_6 = 32$. As said in Section 2, radiation can either corrupt a thread internal registers or operation execution, leading to a single error, or prevent the thread from completing the execution, generating a thread functional interruption that possibly leads to have all the 64 elements wrongly updated. In the former situation, $e_1 = 1$, and the error will spread through the iterations, generating at most 32 errors in the output vector. In the latter, e_1 depends on how many elements were correctly updated before the occurrence of the functional interruption. As 2 elements are updated at the same time in a thread, e_1 will be at least 2 and, thus, 64 errors will be present in the output vector.

Experimental results support these assertions, as in most of the executions less then 64 errors were observed while it is very unlikely to have more than 64 errors in the output. This rare situation occurs when radiation leads a Streaming Multiprocessor to experience a functional interruption preventing a whole warp of 32 threads or even a whole block of 512 threads from completing execution. Those errors will then spread as described in Equation 2, possibly in more than a single 64-points FFT, and a huge amount of errors are expected at the output.

As errors double at each iteration, even the 6 sequential iterations required to compute the 64-points FFTs are sufficient to have a considerable amount of multiple output errors. In order to reduce the occurrence of multiple output errors and increase the *FFT* reliability it is essential to avoid errors propagation from an iteration to the following one. This can be achieved extending to the parallel FFT the results presented in the recent work [22] that proposes ABFT recursive algorithms for dynamic programming in sequential platforms. The idea is to exploit fingerprints for a prompt detection of errors to avoid their propagation. A fingerprinting algorithm is a procedure that maps an arbitrarily large data item (e.g., a vector) to a short bit string, i.e. its fingerprint, with a negligible probability for two items to have the same fingerprint.

The ABFT version has the same organization of the tested *FFT* algorithm, but exploits fingerprints of the input and output vectors of each iteration for detecting errors. Fingerprints could be computed in linear time using the Karp-Rabin fingerprints procedure [24] that just requires a scanning of the vector. When the *i-th* iteration is called, the input fingerprints (or the fingerprints returned by previous iterations) is used for checking data correctness while processing data and multiplying for the twiddle factors. If an error is detected, the call fails and returns to the previous iteration, *i-1*. The input of the failed subproblem is then recomputed and the *i-th* iteration is executed again. If no faults are detected, the output vector and the associated fingerprint are returned, since this output is used as input in subsequent iterations. We suppose that each iteration uses different vectors (e.g., operations are not in place). In such a way, if an iteration fails and returns to the previous one, the input can be restored by using another copy of the data which has a small probability to be corrupted as well (nevertheless, new corruptions are detected through fingerprints).

The proposed procedure applied to a N-point FFT is an $O(N \cdot log_2 N)$ algorithm and captures radiation induced errors requiting re-calculation. Unfortunately, the characteristics of *FFT* do not allow a prompt error correction like in *MxM*. Nevertheless, since most of the work of the proposed *FFT* algorithm is devoted to solve small subproblems, the occurrence of an error will force the re-computation of just small parts of the execution, limiting the introduced overhead.

6. CONCLUSIONS

Radiation-induced errors from the terrestrial neutron radiation environment are one of the major issues for the newest GPU cores reliability. As experimentally demonstrated, neutrons generate mostly multiple output errors even if threads are executed on independent processors.

Multiple output errors are caused by the corruption of resources shared among various threads, like the Streaming Multiprocessor cache, or of critical resources, like the GPU scheduler. In the case of the FFT, the iterations needed to complete the convolution have the countermeasure of letting the errors spread.

Specific and efficient software-based hardening strategies can be designed analyzing the code structure and the GPU architecture. Matrix Multiplication can be hardened adding a checksum on the input matrices, as every thread is independent and does not interact with the others. On the contrary, FFT requires the introduction of checkpoints at every iteration to avoid error propagation. In order to optimize the hardening procedure and avoid useless overhead or excessively affect the code performance, it is necessary to tune its correction capabilities with experimental results.

7. ACKNOLEDGMENTS

This work is partially supported by CAPES foundation of the Ministry of Education, CNPq research council of the Ministry of Science and Technology, FAPERGS research agency of the State

of Rio Grande do Sul, Brazil, Microsoft Corporation, and by University of Padova, Italy, under projects STPD08JA32 and CPDA121378.

Experiments were performed in the ISIS facility, Rutherford Appleton Laboratories, Didcot, UK thanks to Christopher Frost and were founded by the Science and Technology Faculty Council (STFC), UK. Experiments in the LANSCE facility, Los Alamos National Laboratories, NM, USA were performed thanks to Heather Quinn and Thomas Fairbanks.

8. REFERENCES

[1] J.D. Owens, M. Houston, D. Luebke, S. Green, J.E. Stone, and J.C. Phillips, "GPU Computing" Proceedings of the IEEE, vol.96, no.5, pp.879-899, May 2008.

[2] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, "NVIDIA Tesla: A Unified Graphics and Computing Architecture" *IEEE MICRO*, vol. 28, n. 2, March/April 2008, pp. 39-55.

[3] J. Kruger and R. Westermann, "Linear Algebra operators for GPU implementation of numerical algorithms", *ACM Trans. Graph.* n. 22, vol. 3, 2003, pp. 908-916.

[4] J. Liepe, C. Barnes, E. Cule, K. Erguler, P. Kirk, T. Toni, and M. P. H. Stumpf, "ABC-SysBio-approximate Bayesian computation in Python with GPU support" – *Bioinformatics*, vol. 26, n. 14, July 2012, pp. 1797-1799.

[5] Introducing Titan, *www.olcf.ornl.gov/titan*.

[6] P. Rech, C. Aguiar, R. Ferreira, M. Silvestri, A. Griffoni, C. Frost, and L. Carro, "Neutron-Induced Soft Error in Graphic Processing Units", in proc. *IEEE REDW* 2012, Miami, FL, USA.

[7] P. Rech, C. Aguiar, C. Frost, and L. Carro, "Neutron Radiation Test of Graphic Processing Units", *in proc. IEEE IOLTS* 2012, Sitges, Spain.

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

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

[10] P. Rech, C. Aguiar, C. Frost, and L. Carro, "Experimental Evaluation of Software Hardening Techniques for GPUs", *in proc. IEEE RADECS 2012*, Bordeaux, France.

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

[12] NVIDIA GeForce GTX 480/470/465 GPU Datasheet

[13] NVIDIA Tesla C2050/C2075 GPU Datasheet

[14] M. Violante, et al., "A New Hardware/Software Platform and a New 1/E Neutron Source for Soft Error Studies: Testing FPGAs at the ISIS Facility", *IEEE Trans. Nucl. Sci.*, vol. 54, no. 4, pp. 1184-1189

[15] R.C. Baumann, "Neutron-induced boron fission as a major source of soft errors in deep submicron SRAM devices", in proc. *IEEE IRPS 2000*, pp. 152-157

[16] P. Rech, C. Aguiar, C. Frost, and L. Carro, "Experimental Evaluation of Thread Distribution Effects on Multiple Output Errors in GPUs", in proc. *IEEE ETS 2013*, Avignon, France

[17] E. Normand, "Single Event Effects in Avionics", *IEEE Trans. Nucl. Sci.*, Vol. 43, No. 2, Apr. 1996, pp. 461-474.

[18] NVIDIA BENCH: Tesla C2050 Performance Benchmarks

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

[20] R. Freivalds, Fast Probabilistic Algorithms, In Matematical Formulations of CS, *Lecture notes in Computer Science*, vol. 74, 1979, pp. 57-69

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

[22] T. G. Stockham, "High-Speed Convolution and Correlation", *in proc. Spring Joint Computer Conference*, 1966, pp. 229-233.

[23] S. Caminiti, I. Finocchi, E. G. Fusco, and F. Silvestri, "Dynamic programming in faulty memory hierarchies (cache-obliviously)", *in proc. of 31st FSTTCS*, LIPIcs 13, pp. 433-444.

[24] R. M. Karp and M. O. Rabin, "Efficient randomized pattern-matching algorithms", IBM J. Res. Dev., 1987, vol. 31, no. 2, pp. 249–260.