Analysis of kernel redundancy for soft error mitigation on embedded GPUs

A. Serrano-Cases, S. Alcaide, M.A. Romero, Y. Morilla and S. Cuenca-Asensi

***Abstract*—The use of state-of-the-art commercial processors such as graphical processing units (GPUs) is becoming increasingly common in the New Space industry in order to ensure high performance and power efficiency. However, commercial GPUs are not designed to operate in a harsh environment and therefore different protection techniques need to be applied to mitigate the effects of radiation, including those produced by single events. This paper assesses the effectiveness of redundant kernel execution on tightly constrained embedded GPUs under proton irradiation, with results suggesting a significant improvement in the SDC cross-section without penalizing the stability of the whole system. In addition, the posterior error analysis shows that the CPU is the source of the majority of the events, which are mainly dominated by functional interrupts**

***Index Terms*—GPU, proton irradiation, redundant kernels, soft errors.**

I. Introduction

The New Space industry is demanding a new generation of small satellites able to cope with highly demanding applications [1]. Advanced onboard sensors generate huge amounts of data beyond the bandwidth of downlink technologies, including multispectral and hyperspectral images. To alleviate the problem, commercial off-the-shelf devices (COTS) are proposed to improve both the real-time sensors and the processing capabilities of the spacecraft [2]. Graphical processing units (GPUs) were identified as among the most promising candidates because of their performance and effectiveness when working with complex neural networks and image processing algorithms. However, as is the case with other COTS architectures, they are not specifically designed to work in the space environment, meaning that these systems are prone to errors induced by natural radiation and in particular by soft errors.

Considerable effort has been expended in recent years to characterize and study the radiation tolerance of GPU engines [3], [4], and to improve their reliability by means of software and hardware techniques [5], [6]. Embedded GPUs on multicore System on Chips (SoC), have several advantages related to power efficiency and cost over their desktop counterparts which make them ideal for reduced budget missions. However, their processing capabilities and resources limit the applicability of traditional redundancy techniques. Furthermore, they are tightly coupled to the host system, making it more difficult to analyze and identify the sources of vulnerability in radiation experiments [7], [8]. Therefore, in the case of GPU-accelerated SoCs, further research is needed to improve our understanding of soft error propagation through the hardware and software stack.

This work focuses on the enhancement of the fault tolerance to soft errors of the embedded GPUs. Our goal is twofold. Firstly, to assess the effectiveness of the redundant kernel technique when applied to a GPU with limited resources and secondly, to contribute to the understanding of the different categories of errors produced by radiation in order to apply mitigation strategies conveniently. The main aspects of this work can be summarized as follows. The mitigation technique under study is partitioned between the GPU and the CPU and its effect on the overall reliability of the SoC system was investigated. Previous research has mainly used desktop GPUs to analyze their sensitivity to radiation in isolation [9], [10] while recent studies on GPU-accelerated SoCs independently examined the GPU and CPU components (e.g., using TMR solely on the CPUs to eliminate their contribution to radiation-induced faults) [11]. Our research helps to establish the importance of each part to improve the mitigation technique conveniently. Tightly-constrained embedded GPUs usually lack a specific mechanism (e.g., ECC, RAS, etc.) that can detect and identify the source of the errors. In our approach, we propose the use of common operating system capabilities to analyze the source of the radiation-induced events.

For our experiments, we selected the Nvidia Jetson Nano board [12] for testing due to its low cost and high applicability to space intelligence, surveillance and sensor data processing applications. Jetson Nano board features a constrained version of the original TX1 SoC (TM660M-A2 version) [13].

II. Background and Related Works

A. SoC architecture and programming

GPUs integrated into the same chip as the rest of the system are commonly referred to as embedded GPUs. The main difference with respect to their desktop counterparts is that integrated GPUs do not have their own main memory. Instead, a global memory is shared between the host system (CPU) and the GPU, reducing the amount of memory the GPU uses while avoiding memory transfer between different memories.

We used Compute Unified Device Architecture (CUDA) as the programming model. In CUDA, the GPU tasks are named kernels and are launched asynchronously from the CPU and

This work has been supported by the Spanish Ministry of Science and Innovation as part of the PID2019-106455GB-C22 project.

A. Serrano-Cases and S. Alcaide are with Barcelona Supercomputing Center, 08034 Barcelona, Spain (email: alejandro.serrano@bsc.es, salcaide@bsc.es).

Y. Morilla and M.A. Romero are with the National Accelerator Center (CNA), JA, Universidad de Seville, 41092 Seville, Spain (email: ymorilla@us.es, mrmaestre@us.es).

S. Cuenca-Asensi is with the Computer Technology Department, University of Alicante, Carretera San Vicente del Raspeig s/n, 03690 Alicante, Spain (email:sergio@dtic.ua.es).

defined as "global" functions inside the code. When launched, kernels feature two mandatory arguments, the number of threads in a thread block and the number of blocks (or grid), allowing the number of threads that the kernel will execute to be specified.

Thread blocks are later scheduled to a single Streaming Multiprocessor (SM) inside the GPU by the kernel scheduler, an internal entity inside the GPU that is able to run CUDA threads. Threads within a thread block are then grouped into smaller sets named warps, which typically have a maximum size of 32 threads. SMs execute all threads in a warp concurrently. Internally, SMs contain certain resources that will be shared among the CUDA threads within a warp, such as the register file, the instruction cache, the data cache or internal shared memory. Other (mainly functional) resources will be private for each thread, such as the CUDA cores, load/store units and special function units.

The regular offloading process of tasks to the GPU contains the following steps: (1) GPU memory allocation, (2) Input Data transfer, (3) kernel launching, (4) Result transfer back to the CPU and (5) GPU memory deallocation. All these steps are performed by calling functions of the CUDA driver (e.g., CudaMemCopy) that are executed by a single process, the CUDA runtime process, thus serializing them. Steps (2) and (4) are optional, for example: some kernels may do not need input data or they may be avoided if global memory is shared between the host system and the GPU.

B. Redundancy on GPUs

GPU software redundancy can be achieved at different granularities. **Intra-thread duplication** performs each instruction twice and checks the result of the original instruction and the shadow, a technique that has been successfully studied in GPU-based programs in [14], [15]. The most interesting aspect of this strategy is that it is totally transparent to the programmer, as it does not require any change in the programming model or the code, with the compiler directly applying all modifications. The fault detection latency is also the shortest of the different strategies. The downside is that intra-thread duplication requires explicit checking instructions, increasing register file usage and doubling the number of arithmetic operations, which are serialized as the checking instruction requires the results of both source instructions. Since there is no control over the functional units employed by each redundant instruction, this strategy is susceptible to permanent faults affecting the functional units since the redundant computations will use the same functional units.

Inter-thread duplication employs half a warp (16 threads) as shadow threads. This approach would seem to be of interest in the case of GPUs, as they may use inter-thread GPU synchronization primitives such as shuffle instructions and shared memory to communicate between threads [16], [17], avoiding the much slower general GPU memory used in the intra-thread approach. However, this technique doubles the number of threads employed and may not work on programs that use more than half the warp threads. Moreover, it is not programmer-transparent, as programs that use intra-warp communications instructions or the shared memory may be incompatible with this technique or require non-trivial modifications. In contrast to the intra-thread approach, different functional units may be employed by the redundant threads, which improves the reliability against permanent faults appearing in the functional units. Still, this technique is susceptible to permanent faults affecting shared resources (e.g., L1 cache, register file). This approach can also be susceptible to Common Cause Failure (CCF), failures due to a specific type of faults, the Common Cause Faults (e.g., voltage droop) that can affect similarly multiple units.

Under some conditions, **kernel duplication** may allow a diverse redundant execution (redundant execution with staggering) [18]. The necessary conditions are that (1) the original kernel needs to use no more than half the GPU's resources to enable concurrent execution, which helps to reduce execution time (otherwise, kernels are serialized) and ensures that kernels use different Streaming Multiprocessor to perform their executions and (2) the GPU must have a minimum of two SMs. If the conditions are met, redundant kernels start with an initial staggering (due to small serialization created by the CPU driver) but without the means to control it or monitor it. Staggering is beneficial as it may protect the redundant executions against CCF (e.g., voltage droop), similar to the Dual-Core Lockstep (DCLS) execution. However, this approach requires duplicating the GPU resources from the original kernel. Since the result is performed at the end of the GPU execution, the fault-detection latency is also the highest of all the techniques discussed. In the present work, both conditions were relaxed to accommodate the resources of a low-cost embedded GPU.

Nevertheless, the modifications to the original code are simple. Generally, it involves duplicating the calls to the original functions with the replicated data and a CUDA Stream created for each redundant kernel in order to enable (if possible) concurrent execution. Kernels that do not use an explicit CUDA stream use the default CUDA stream which serializes their execution, as discovered by Amert et. al. [19] in certain Nvidia GPUs.

C. Related Works

Numerous works have been published that analyze the effect of common software-based techniques applied to advanced GPU architectures, mainly Fermi, Kepler or Pascal. For example, Duplication With Comparison (DWC) [26], intra-thread replication [27], backward-recovery mechanisms [28], [29] and control-flow checkers [28]. However, the majority of these used fault injection methodology to conduct the experiments and only a few have been tested under radiation. In this regard, authors in [30] have shown that hardening techniques have to deal with multiple radiation-induced errors to efficiently protect the applications running on modern GPUs. They proposed an optimized algorithm-based technique (ABFT) to reduce the overheads on matrix multiplication. The technique was tested under a neutron beam on a high-end desktop GPU [31] showing a better performance and effectiveness than Error Correction Codes (ECC). In addition, the impact of the parallelism on the applications reliability

TABLE I: Comparison Table of Related Works

Device, arch., node	Work	Platform	Particle/Energy	Technique	Benchmarks	Reported results
Tegra TK1 Kepler (1SM) 28nm CMOS	Wang2017 [7]	Jetson TK1	$p^+/120MeV$	–	particle, FFT	TID, SEU, SEL
	Badía2022a [8]	Jetson TK1	$p^+/15MeV$	Imp. strategies	MxM/block/CUBLAS	SDC, Crash/Hang
	Badía2022b [20]	Jetson TK1	$p^+/15MeV$	L/U Decomp.	MxM	SDC, Crash/Hang
Tegra TX1 Maxwell (2SM) 20nm CMOS	Wyrwas2016 [21]	Jetson TX1	$p^+/200MeV$	–	particle, fluidsGL	SEU, SEFI, SEL
	Santos2019 [10]	Jetson TX1	$n/atmos.$	–	YOLO, CNN, GEMM	SDC, Crash/Hang
	Fratin2018 [22]	Jetson TX1	$n/atmos.$	–	LavaMD, Hotspot, GEMM	SDC, Crash/Hang
Tegra TX1 Maxwell (1SM) 20nm CMOS	Slater2020 [13]	Jetson nano	$gamma/Cobalt\,60$	–	MxM	TID, SEFI
	This work	Jetson nano	$p^+/15MeV$	DWC	MxM	SDC, Crash/Hang
Tegra TX2 Pascal (2SM) 16nm FinFET	Wyrwas2019 [23]	Jetson TX2	$p^+/200MeV$	–	particle, fluidsGL	SEU, SEFI, SEL
Tegra Xavier Volta (6/8SM) 12nm FinFET	Hiemstra2020 [11]	Jetson AGX	$p^+/105MeV$	TMR (cpu)	FFT	TID, SEFI, SEL
	Rodriguez2022 [24]	Jetson NX/Ind.	$p^+/200MeV$	RAS system	MXM	SEU, SEFI, SEL
Snapdragon 835/845 14/10nm FinFET	Guertin2019 [25]	Intrinsyc Open-Q 835	$Ar/40MeV$	–	Video, graphics	SEU, SEFI
		Intrinsyc Open-Q 845	$Ar/40MeV$	–	FFT	SEU, SEFI

was extensively studied under radiation in [9]. The work showed that the distribution of threads on the blocks and their complexity (degree of parallelism) directly affect the effort of warp schedulers and the memory access latency, ultimately determining radiation sensitivity.

Other popular mitigation approaches, such as DWC and TMR, were compared under neutron irradiation in [9] and [31]. The DWC technique was thoroughly examined in [32], using the HotSpot benchmark. Three different implementation strategies were analyzed: (i) spatial duplication of the blocks (similar to kernel duplication), (ii) spatial duplication with blocks interleaving and (iii) time (intra-thread) duplication. The comparison resulted in two noteworthy findings. Firstly, DWC approximately doubled the number of errors. Secondly, the kernel duplication strategy exhibited the best performance in terms of error rate (nearly zero), although it did lead to an increase in functional interrupt events compared to the plain version. Our study complements these findings by evaluating the effectiveness of kernel duplication on resource-constrained devices that require serialization of the kernels. Furthermore, unlike the conventional approach, our experiments encompass the entire system in which the GPU is integrated.

Radiation tests were conducted on different embedded GPU devices as well, with the Nvidia SoCs being most prevalent compared to other devices (e.g., Snapdragon family). Table I provides a representative number of those works and summarizes the relevant features of the experimental setups employed: SoCs, platform, beam and benchmark. The techniques tested and the categories used to report the results are outlined in the last two columns. Most of these studies focused on evaluating the inherent susceptibility of the SoCs to radiation in terms of TID and SEE rates [7], [21], [23], [13], [25], [11]. No further analysis about the consequences of the SEE, i.e. how many of them turn into silent data corruption or system crashes, were performed during these tests. Matrix multiplication, Fast Fourier Transform and graphic simulations were the common benchmarks used as the baseline.

The results showed that CMOS-based mature node processes were approximately ten times more susceptible to SEE than modern FinFET nodes [23]. However, no latchup events (SEL) were reported in any of the studies conducted, regardless of the technology used. Functional interrupt events (SEFI) are shown to be a common problem in such devices and in most studies they dominate over the bit upsets (SEU) [21], [23]. Nevertheless, there is no clear relation between them since other studies, such as those in [22] and [10], reported a different behavior on Tegra X1 SoC. They used complex benchmarks such as optimized matrix multiplication (GEMM), LavaMD and HotSpot, which exhibited SDC rates that doubles the number of SEFI under atmospheric neutrons.

Only a few of the works focused on the assessment of specific mitigation techniques. Authors in [8] studied the impact of different implementation strategies on the reliability of the matrix multiplication benchmark running on TK1 SoC. Based on GPU resource utilization, they explain the differences on reliability obtained and detect the different trade-offs between performance and error rate. A similar approach is followed in [20] to assess the L/U decomposition strategy. Our work offers insights on baseline MxM implementation by incorporating a software-based protection technique (DWC).

The advanced Volta architecture was tested in [11], although the TMR technique was only applied to the CPU side of the SoC. Finally, it is worth mentioning the study in [24] which assessed the new ARM Reliability Availability and Serviceability system (RAS) which is only included in the high-end Xavier family. RAS comprises built-in hardware and software diagnostic components that are specifically designed for error detection. This system enables the direct identification of the specific hardware resource where the radiation event occurred (caches, GPU, CPU). Our approach provides a solution for low-cost SoCs without a RAS system by utilizing the standard capabilities of operating systems to study the origin of SEFI events.

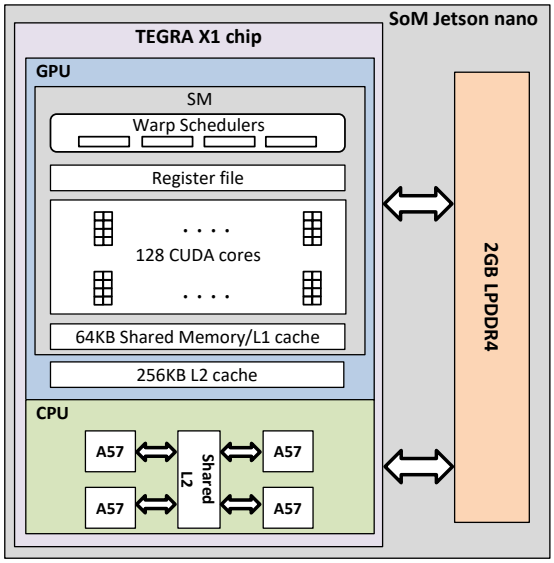

Fig. 1: Nvidia Jetson Nano architecture overview.

III. Radiation Experiments

The system under test (SUT) was the Nvidia Tegra X1 (TX1) System on Chip [33] which is manufactured in 20nm planar technology (TNMSC process) and was launched by Nvidia in 2019. This limited version of the original TX1 (2015) includes a Quad-Core ARM Cortex A57 64-bit processor (see Fig. 1). Each A57 core has a 48 KB L1 Instruction cache and a 32 KB L1 Data Cache, while the 2 MB L2 cache is unified and shared across all the cores. It also contains an Nvidia Maxwell GPU with 1 SM containing 128 CUDA cores and a shared L2 of 256 KB. The SM contains four warp schedulers, although all SM core functional units are assigned to a particular scheduler with no shared units. The CPU and GPU share the same physical external memory, a 2 GB LPDDR4.

TX1 is mounted on Nvidia Jetson Nano SoM (embedded system on module) together with the LPDDR4 volatile memory and a flash-based micro-SD card for non-volatile storage (see Fig.1).

A. Case study

The generic Matrix Multiplication application was selected for the case study due to its simplicity and extended use (see Table I). The GPU baseline version of the Matrix Multiplication uses one thread to compute a single cell of the resulting matrix. Therefore, each thread needs to gather an entire row of the first input matrix and an entire column of the second input matrix to compute its result value. To evaluate the resilience of the software Kernel duplication approach we modified the original code according to the changes shown in the previous section. In our implementation the technique was partitioned between the GPU, which executed the redundant kernels, and the CPU, responsible for result comparison and error detection.

Both versions, original and redundant, were profiled using the Nvidia profiler. The results for input matrices of 512×512

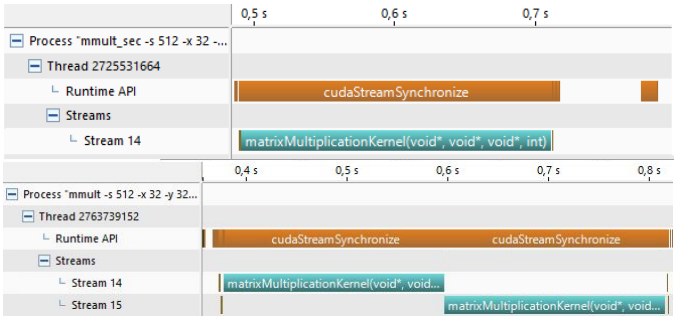

Fig. 2: Profiling of the Matrix Multiplication benchmark: original version (above), redundant version (below).

elements (32-bit floating point) and blocks size of 32×32 threads (i.e.: a total of 256 blocks of 1024 threads executed) are shown in Figure 2. As can be seen in the profiler images, as the Nvidia Jetson Nano only has one SM, the kernels cannot execute in parallel. Having executed each version 500 times, we saw that the kernel execution time for the original version, not considering data transfer and supplementary tasks, was 222.55 ms and 442.64 ms for the two kernels in the redundant version. As expected, the kernel execution time is almost doubled due to serialization.

In Table II we can see the characterization of the time spent in different sections of the execution for different configurations, again an average of 500 executions. It can be observed that most of the time is spent in the kernel, both in the original version and in the redundant one. Due to the serialization of the kernels, the percentage of kernel time increases on the hardened versions. It can be seen that time spent in data transfer is almost negligible in both the numbers and profiler images (tiny lines before and after the kernel).

TABLE II: Execution time percentages (Matrix Size 512x512)

	Original			Redundant		
	32b	8b	2b	32b	8b	2b
Kernel (%)	90.57	93.41	98.78	91.41	94.64	99.01
Data Transf.(%)	2.69	1.90	0.36	1.79	1.28	0.23
Others (%)	6.73	4.69	0.86	6.80	4.10	0.76

B. Experimental setup

Several versions of the Matrix multiplication algorithm were tested in the external beamline of the 18/9 Ion Beam Applications compact cyclotron located at National Accelerator Center (*Centro Nacional de Aceleradores* or CNA) in Seville, Spain (see Fig. 3). The SUT was irradiated without thinning in the open air and removing the metallic heatsink to get the maximum proton energy available in the active area. However, to prevent any thermal events resulting from overheating, the GPU's operating frequency was restricted to the minimum allowable value (768 MHz).

The beam energy at the SUT surface was 15.4 MeV, with an estimated spread in the order of 400 keV. The flux had to be adjusted after several tests to avoid repeatedly hanging the

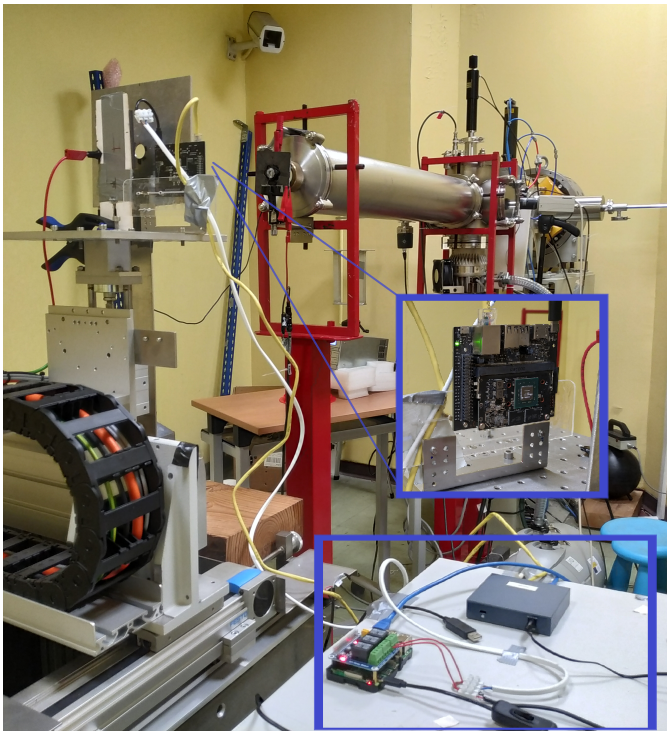

Fig. 3: Experimental setup within the radiation chamber at CNA. Front view of the Jetson board without the metallic heatsink and detailed view of the Control Computer.

CPU during the reboot process. The average proton flux was maintained in the order of 3×10^7 proton/cm^2/s, and the overall fluence following all radiation runs was 7.4×10^{11} proton/cm^2. The homogeneous beam spot of 16 mm in diameter covered the interest area of the system, including all the computing hardware (i.e.: CPU and GPU cores and their respective memory caches). During the test, the DDR memory chips and the micro-SD card hosting the operating system were kept out of the beam. Moreover, an aluminum mask with a 22 mm diameter hole was positioned in front of the board. The environmental radiation levels were insignificant under such conditions.

The experimental setup comprised an external computer (CC) not under direct beam exposure, which remotely controlled SUT operation by means of a Local Area Network (LAN). Using an SSH connection the CC computer sent the commands to control the execution of the benchmarks as well as receiving their outputs. This way, only minimal instrumentation was applied to the benchmark code. Furthermore, since no SUT local scripts were needed the operation of the system remained as close to the real configuration as possible.

Benchmarks were coded in C/C++ and compiled with Nvcuda V10.2.300. The code includes several calls to the CUDA runtime API [34] which provides real-time information about the errors related to the GPU operation. Each radiation experiment involves only one version of the benchmark which is continuously executed in batches of N runs. One run comprised the execution of the kernels in the GPU and the check of the results in the CPU. At the end of each run, a message was printed to provide information on the number of errors obtained. Once all the runs of a batch had finished, a message was printed with the total number of errors and the CC launches another batch of N runs. Should the API CUDA capture an error during the GPU work, a message is immediately printed and the current run finishes without completing the batch. A global timeout is defined in the control script of the CC. If the current run does not complete before the timeout, the CC remotely produces a power cycle in the Jetson to reboot the operating system. In addition, the control computer is continually monitoring and recording the log messages from the Linux kernel via a serial communication interface. This valuable information is subsequently processed to analyze the origin and effect of the events.

We distinguished the following error categories depending on the behavior of the SUT:

- SDC (Silent Data Corruption): the GPU kernel finishes successfully but some of the results are wrong.
- Crash: the run finishes abruptly after an error is reported by the CUDA API during the execution of a benchmark. A new run can be executed afterward without rebooting the operating system.
- HANG: the run hangs and the device needs to be remotely rebooted by the CC.

The latter two categories are usually included in the SEFI (Single Event Functional Interrupt).

IV. Experimental Results and Discussion

Each version of the benchmark was run repeatedly until a total fluence greater than 4.3×10^{10} proton/cm^2 was reached. The cross-section was computed by accumulating event counts from multiple runs to attain statistical significance, as depicted in Figure 4. Only effective execution time was taken into account, excluding the duration of the system reboot (about 16 seconds), the time to power off and on the board (4 seconds) and the watchdog timeout to identify the HANGs (20 seconds). Upper and lower error margins were calculated using the inverted chi square distribution as described in [35], at a confidence level of 95% considering a fluence uncertainty of $\pm10\%$.

The benchmark used 32-bit floating-point data and tested matrix sizes of either 512×512 or 1024×1024. The name of the different versions include an R prefix to indicate redundant versions. The matrix size is denoted by the first number, while the second number represents the thread block size, with nB indicating a n×n thread block size. For better analysis of the results, they are shown grouped into three categories: the 512 unhardened versions on the left, the 1024 redundant versions in the center, and the 512 redundant versions on the right. The unhardened version displays a decrease in the SDC cross-section with increasing block size, as is clear in the figure. However, no SDC was detected in any of the experiments conducted on the redundant versions. On the other hand, the protection technique identified several errors that followed a consistent pattern in both groups. According to those results, a lower number of threads per block leads to a more reliable execution.

An explanation can be found in the way the threads use memories. Matrices are stored in DDR memory, which remains out of the beam. However, the L1 and L2 caches, where the data is accessed by the threads, are susceptible to radiation. Comparing the center and right groups, it is apparent that the bigger the matrices, the more use of the caches and the higher the probability of error.

It is interesting to note the effect inside each group, where configurations with a larger block size exhibit higher SDCs (or detected) rates compared to other configurations. This can be explained by the fact that the L2 cache is more heavily utilized in configurations with a higher number of threads per block. To back this theory up, we ran additional analysis by checking these configurations without radiation and obtaining all the metrics available using the Nvidia profiler (nvprof). The resulting statistics reveal that in configurations with a greater number of threads per block, the L2 cache was utilized more frequently with fewer misses. This indicates that more data was being stored in the cache with a longer lifetime due to fewer replacements, which definitely makes it more vulnerable to SDCs.

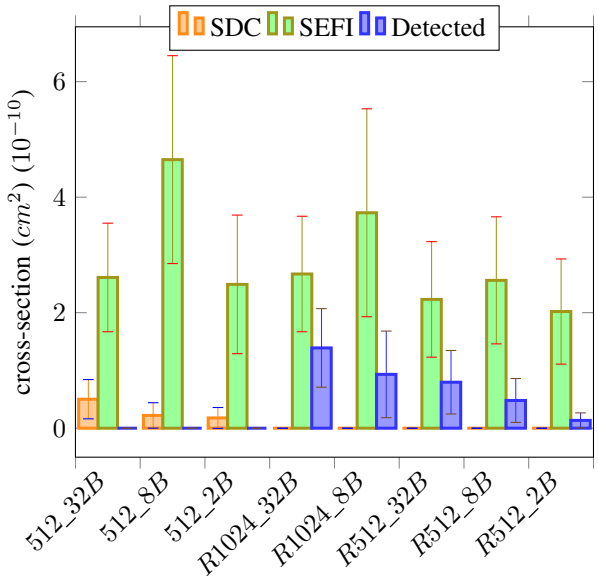

Fig. 4: Cross-section of the originals and the redundant versions

For clarity purposes, HANGs and Crashes were accumulated and represented as a SEFI cross-section. Remarkably, the SEFI cross-section is not worsened by the control overhead imposed by running redundant kernels, with all R versions showing slightly better figures than the equivalent unhardened code. It should be noted that the $8B$ versions present higher SEFI values than both the $32B$ and the $2B$ configurations for all the groups.

Two factors may contribute to the results obtained. The first factor is related to the significance of the CPU as the primary source of HANG events. Benchmark versions where the CPU plays a larger role in the computation will therefore have a higher susceptibility to HANGs, such as unhardened versions. The following section provides a detailed analysis of the sources of HANGs and justifies this aspect. The second factor relates to the extent to which the application utilizes GPU resources. To determine this, an additional analysis was conducted using data from the Nvidia profiler. Three metrics were considered: WarpOccupancy, which was provided by the profiler, and SM occupancy (SMOcc), which was computed relative to both the maximum number of blocks and active threads supported by the Maxwell architecture, namely MaxBlocksPerSM (32) and MaxThreadsPerSM (2048). These metrics indicate the efficiency of the GPU resources used and the efforts of the warp schedulers to support them. The $8B$ versions exhibit metrics close to 100% in all three areas, making them more vulnerable to any type of SEFI events such as Crashes and HANGs. Conversely, the $2B$ versions have a low percentage in WarpOccupancy and SMOccWrtMaxThreads (12.5% and 6.3%, respectively). The $32B$ version fails in the SMOccWrtMaxBlocks metric with a score of 6.3%.

In summary, it can be said that the redundant kernel technique effectively reduces the SDC cross-section without penalizing the system stability (SEFI events). Additionally, the study highlights that resource usage affects the sensitivity to SEFIs.

The Mean Work To Failure metric, defined as the mean number of executions to a failure, reveals the trade-off between the fault tolerance improvement and the time overhead introduced by the technique. Table III presents the unhardened and hardened versions working on 512×512 matrices. As expected, due to the performance penalty produced by the serialization of the redundant kernels, no significant improvement is observed in this metric. It should be noted that in the case of protected versions, only HANGs failures are involved in the MWTF, which usually are considered less critical than SDC.

TABLE III: Total execution times and MWTF metric

Version	ExecTime (s)	MWTF
512_32B	0.345	9.3×10^9
512_8B	0.455	4.5×10^9
512_2B	1.888	2.0×10^9
R512_32B	0.533	8.4×10^9
R512_8B	0.746	5.2×10^9
R512_2B	3.617	1.4×10^9

During the experiments, most runs finished without errors and no significant temperature rises were recorded, which could be a possible indication of a single-event latchup (SEL) event. Figure 5 shows the percentage of each type of event per thousand runs. As can be seen, SEFIs (HANGs plus Crahses) are preponderant over the rest of the events (SEU and Detected) and their relative weight increases as the thread block size decreases. Furthermore, the rate of events directly related to the GPU (Crashes) is very low in comparison to those that provoke the HANGs, in line with other results reported for previous Nvidia SoCs [8]. Regarding the redundant versions they present the same behavior in both 512 and 1024 matrix sizes (for clarity purposes the R1024 versions were excluded from the figure since the error percentages grow one order of magnitude).

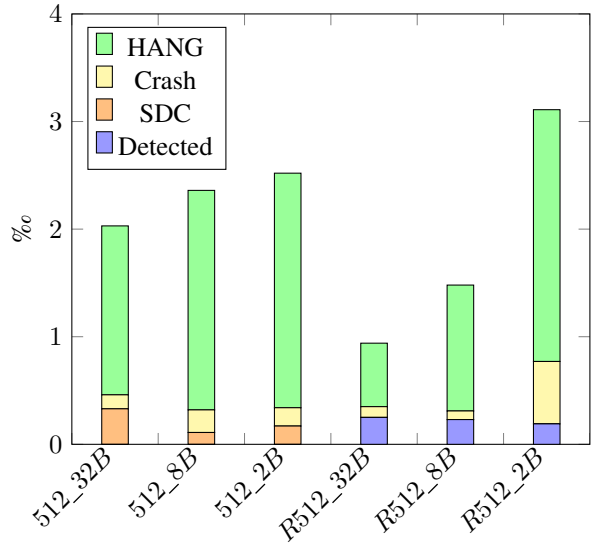

Fig. 5: Error rate per thousand runs of each version of the benchmark

Error analysis

This study proposes utilizing the Linux syslog feature to gain deeper insights into the internal mechanisms related to SEFI events and in particular those that produces the system hang. The syslog is a tool that captures and stores various messages generated by the OS and its applications. The Jetson Nano is equipped with a Serial Debug Console, which is set up to transmit the syslog information during the boot process as well as the kernel events related to the system's normal operation. Both, sys and kernel logs were recorded in the CC computer during the experiments by means of the serial console. Afterward, the logs were analyzed offline and their data was correlated with the timestamp of the observed events to identify their source (GPU or CPU) and, if possible, the cause that triggered them.

Figure 6 shows an example of a log file recorded during the experiments. The initial lines represent messages issued during the boot sequence which terminates with the login request. Subsequently, any kernel, driver, or application error will be reported, along with a timestamp enclosed in square brackets. The error log has a common structure, as depicted in the figure. The first line provides a summary of the error that occurred 128.095 seconds after the boot sequence finished. The kind of exception raised is then obtained from the Exception Syndrome Register (ESR), where the cause of the exception is coded in the first two digits. After some lines for decoding the exception syndrome information, the type of the error is reported with a timestamp of [128.144]. In this particular case, it is a kernel error categorized as *Oops*, indicating a malfunctioning of the kernel. The *Oops* word is followed by the syndrome code denoting the cause of the error. The report continues with information regarding the status of the CPU registers, a dump of the stack memory and, if feasible, the action taken by the kernel to solve the error (not shown in the figure).

```
[0000.125][L4T TegraBoot] (version 00.00.2018.01-l4t)
...
[0000.704] Bootloader downloaded successfully.
...
[0000.866] Starting CPU & Halting co-processor

Starting kernel ...
[0.000000] Booting Linux on physical CPU 0x0
...
Ubuntu 18.04.5 LTS jetson-nano ttyS0
jetson-nano login:
[128.095] Unable to handle kernel NULL pointer
dereference at virtual address 0000005f
[128.103] Mem abort info:
[128.106]   ESR = 0x96000021
...
[128.144]Internal error: Oops: 96000021 [#1] PREEMPT SMP
```

Fig. 6: Example of syslog recorded during the radiation experiments

Table IV displays all error types observed in the experiments. The majority of them were kernel *Oops*, which have a clear source in the CPU operation as the associated exceptions suggest. Among all the possible exception classes, only two of them were reported by the kernel log: Data Abort (0x96) and Instruction Abort (0x86). The possible causes of those exception were: translation fault when reading/writing data or reading instructions, permission faults in the access to an instruction and alignment faults when reading data. Two error types were notified as internal errors. The first indicated that the kernel code was attempting to access the user space memory in an incorrect manner "Accessing user space memory outside uaccess.h". The second was stated as "undefined instruction", which typically occurs when the processor tries to execute a machine code that is corrupted or incomplete. Both types of errors are directly related to CPU operation. The same source was assigned for the error type "BUG: Bad page state in process systemd", which denotes a problem with the operating system's memory management subsystem. The error type described as "not syncing Watchdog detected hard LOCKUP" is usually displayed by the Linux kernel when it detects that the system has become unresponsive or has locked up for an extended period of time. It can be caused by an external device like the GPU, so subsequent messages in the log had to be examined before to assign the source of the error. Finally there was a type of error directly related to the GPU operation. This type explicitly mention the GPU driver as the source of error: "nvgpu:57000000.gpu" and it was usually followed by a description of the problem (e.g., gr_gk20a_handle_sm_exception, pmu halt intr not implemented, Timeout detected, etc.).

The logs collected during the radiation tests were processed following the described guidelines, with the result of the analysis shown in Table V.

The upper half of the table displays the HANG percentages assigned to the CPU and GPU as the most probable error sources. Please, note that Crashes were not included in the analysis. Although some of them were registered in the log messages, they were easily traced by the CUDA API and notified to the CC computer.

Consistent with prior studies, the majority of the HANGs

TABLE IV: Error types and associated exceptions

Error Msg	Source
Unable to handle kernel paging request (Oops)	CPU
Unable to handle kernel write to read-only memory (Oops)	CPU
Unable to handle kernel read from unreadable memory (Oops)	CPU
Kernel BUG (Oops)	CPU
Internal error: undefined instruction	CPU
Internal error: Accessing user space memory outside uaccess.h	CPU
not syncing Watchdog detected hard LOCKUP	CPU/GPU
BUG: Bad page state in process systemd	CPU
Unhandled Exception in EL3	CPU/GPU
nvgpu: 57000000.gpu	GPU

TABLE V: Summary of log analysis (512×512 versions)

Source	$32B$	$8B$	$2B$	R_32B	R_8B	R_2B	Total
CPU	0.91	0.68	0.91	0.67	0.93	0.58	0.78
GPU	0.09	0.05	0.09	0.17	0.07	0.33	0.12
Unknown	0.00	0.26	0.00	0.17	0.00	0.08	0.9
HANG	0.73	0.70	0.38	0.33	0.67	0.53	0.59
Autoreboot	0.27	0.25	0.46	0.50	0.33	0.27	0.33
Non-critical	0.00	0.05	0.15	0.17	0.00	0.20	0.09

can be attributed to the CPU, with a smaller percentage to the GPU. This trend can be seen across all versions, accounting for a total of 78% for the CPU and 12% for the GPU. Notably, some events could not be assigned to any source due to the absence of anomalous behavior recorded by the Linux kernel, comprising 9% of the total events registered in the log files.

The criticality of the errors reported by the syslog is depicted in the lower half of the table. A percentage of the problematic behaviors indicated by the log did not have negative consequences and the run finished successfully. They were classified as non-critical. The remaining percentage caused a HANG, prompting the CC computer to execute a power cycle. In a considerable number of cases, the Linux kernel responded by programming an auto-reboot that could have restored the system without any external intervention. In the light of the results, it is desirable to complement the protection technique with additional mechanisms capable of dealing with the SEFI events in an efficient way.

V. Conclusions

This paper presents the assessment of a traditional redundancy technique applied to embedded GPU with limited resources. Results showed a clear improvement of the SDC cross section without negatively affecting the SEFI error rate. However, the forced serialization of redundant kernels due to resource scarcity produced a significant time overhead. As a result, the mean workload that could be executed by the redundant versions before the occurrence of a failure was slightly lower than the original codes. In addition, the Linux syslog tool was proposed as a low-cost solution to analyse the impact of the Functional interrupt errors. Using the information reported by the Linux kernel, it was possible to identify the source of the majority of the HANG events and to analyze their severity level.

References

[1] G. Furano, G. Meoni, A. Dunne, D. Moloney, V. Ferlet-Cavrois, A. Tavoularis, J. Byrne, L. Buckley, M. Psarakis, K.-O. Voss, and L. Fanucci, "Towards the use of artificial intelligence on the edge in space systems: Challenges and opportunities," *IEEE Aerospace and Electronic Systems Magazine*, vol. 35, no. 12, pp. 44–56, Dec. 2020.

[2] R. L. Davidson and C. P. Bridges, "Error resilient gpu accelerated image processing for space applications," *IEEE Transactions on Parallel and Distributed Systems*, vol. 29, no. 9, pp. 1990–2003, Sept. 2018.

[3] E. Wyrwas, K. A. LaBel, M. Campola, and M. O'Bryan, "Guidance on standardizing gpu test approaches," in *2018 IEEE Radiation Effects Data Workshop (REDW)*, Jul. 2018, pp. 116–119.

[4] F. F. d. Santos, S. K. S. Hari, P. M. Basso, L. Carro, and P. Rech, "Demystifying gpu reliability: Comparing and combining beam experiments, fault simulation, and profiling," in *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, May 2021, pp. 289–298.

[5] F. F. d. Santos, P. F. Pimenta, C. Lunardi, L. Draghetti, L. Carro, D. Kaeli, and P. Rech, "Analyzing and increasing the reliability of convolutional neural networks on gpus," *IEEE Transactions on Reliability*, vol. 68, no. 2, pp. 663–677, June 2019.

[6] M. Goncalves, F. Fernandes, I. Lamb, P. Rech, and J. R. Azambuja, "Selective fault tolerance for register files of graphics processing units," *IEEE Transactions on Nuclear Science*, vol. 66, no. 7, pp. 1449–1456, March 2019.

[7] H. Wang, Q. Chen, L. Chen, D. M. Hiemstra, and V. Kirischian, "Single event upset characterization of the tegra k1 mobile processor using proton irradiation," in *2017 IEEE Radiation Effects Data Workshop (REDW)*, Dec. 2017, pp. 127–130.

[8] J. M. Badia, G. Leon, J. A. Belloch, M. Garcia-Valderas, A. Lindoso, and L. Entrena, "Comparison of parallel implementation strategies in gpu-accelerated system-on-chip under proton irradiation," *IEEE Transactions on Nuclear Science*, vol. 69, no. 3, pp. 444–452, March 2022.

[9] P. Rech, L. Pilla, P. Navaux, and L. Carro, "Impact of gpus parallelism management on safety-critical and hpc applications reliability," in *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, June 2014, pp. 455–466.

[10] F. F. d. Santos, P. F. Pimenta, C. Lunardi, L. Draghetti, L. Carro, D. Kaeli, and P. Rech, "Analyzing and increasing the reliability of convolutional neural networks on gpus," *IEEE Transactions on Reliability*, vol. 68, no. 2, pp. 663–677, June 2019.

[11] D. M. Hiemstra, C. Jin, Z. Li, R. Chen, S. Shi, and L. Chen, "Single event effect evaluation of the jetson agx xavier module using proton irradiation," in *2020 IEEE Radiation Effects Data Workshop (in conjunction with 2020 NSREC)*, Dec. 2020, pp. 1–4.

[12] [Online]. Available: https://developer.nvidia.com/embedded/jetson-nano

[13] W. S. Slater, N. P. Tiwari, T. M. Lovelly, and J. K. Mee, "Total ionizing dose radiation testing of nvidia jetson nano gpus," in *2020 IEEE High Performance Extreme Computing Conference (HPEC)*, Sept. 2020, pp. 639–641.

[14] M. Dimitrov, M. Mantor, and H. Zhou, "Understanding software approaches for gpgpu reliability," in *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*. New York, NY, USA: Association for Computing Machinery, March 2009, p. 94–104.

[15] A. Mahmoud, S. K. S. Hari, M. B. Sullivan, T. Tsai, and S. W. Keckler, "Optimizing software-directed instruction replication for gpu error detection," in *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, Nov. 2018, pp. 842–854.

[16] J. Wadden, A. Lyashevsky, S. Gurumurthi, V. Sridharan, and K. Skadron, "Real-world design and evaluation of compiler-managed gpu redundant multithreading," in *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, June 2014, pp. 73–84.

[17] M. Gupta, D. Lowell, J. Kalamatianos, S. Raasch, V. Sridharan, D. Tullsen, and R. Gupta, "Compiler techniques to reduce the synchronization overhead of gpu redundant multithreading," in *2017 54th ACM/EDAC/IEEE Design Automation Conference (DAC)*, June 2017, pp. 1–6.

[18] S. Alcaide, L. Kosmidis, C. Hernandez, and J. Abella, "Software-only based diverse redundancy for asil-d automotive applications on embedded hpc platforms," in *2020 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*, Oct. 2020, pp. 165–168.

[19] T. Amert, N. Otterness, M. Yang, J. H. Anderson, and F. D. Smith, "Gpu scheduling on the nvidia tx2: Hidden details revealed," in *2017 IEEE Real-Time Systems Symposium (RTSS)*, Dec. 2017, pp. 104–115.

[20] J. M. Badia, G. Leon, J. A. Belloch, A. Lindoso, M. Garcia-Valderas, Y. Morilla, and L. Entrena, "Reliability evaluation of lu decomposition on gpu-accelerated system-on-chip under proton irradiation," *IEEE Transactions on Nuclear Science*, vol. 69, no. 7, pp. 1467–1474, March 2022.

[21] E. Wyrwas, "Proton testing of nvidia jetson tx1," in *NASA Goddard Space Flight Center,Space Flight Center, Greenbelt, MD, USA, Tech. Rep. 20170009004*, Oct. 2016.
[22] V. Fratin, D. Oliveira, C. Lunardi, F. Santos, G. Rodrigues, and P. Rech, "Code-dependent and architecture-dependent reliability behaviors," in *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, June 2018, pp. 13–26.
[23] E. Wyrwas, "Proton testing of nvidia jetson tx2," in *NASA Goddard Space Flight Center,Space Flight Center, Greenbelt, MD, USA, Tech. Rep. 20190031856*, Jul. 2019.
[24] I. Rodriguez-Ferrandez, M. Tali, L. Kosmidis, M. Rovituso, and D. Steenari, "Sources of single event effects in the nvidia xavier soc family under proton irradiation," pp. 112–118, Sept. 2022.
[25] S. M. Guertin, W. P. Parker, A. C. Daniel, and P. Adell, "Recent see results for snapdragon processors," in *2019 IEEE Radiation Effects Data Workshop*, July 2019, pp. 1–5.
[26] H. Jeon and M. Annavaram, "Warped-dmr: Light-weight error detection for gpgpu," in *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, Dec. 2012, pp. 37–47.
[27] A. Mahmoud, S. K. S. Hari, M. B. Sullivan, T. Tsai, and S. W. Keckler, "Optimizing software-directed instruction replication for gpu error detection," in *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, Nov. 2018, pp. 842–854.
[28] A. Nukada, H. Takizawa, and S. Matsuoka, "Nvcr: A transparent checkpoint-restart library for nvidia cuda," in *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum*, May 2011, pp. 104–113.
[29] R. Garg, A. Mohan, M. Sullivan, and G. Cooperman, "Crum: Checkpoint-restart support for cuda's unified memory," in *2018 IEEE International Conference on Cluster Computing (CLUSTER)*, Sept. 2018, pp. 302–313.
[30] P. Rech, C. Aguiar, C. Frost, and L. Carro, "An efficient and experimentally tuned software-based hardening strategy for matrix multiplication on gpus," *IEEE Transactions on Nuclear Science*, vol. 60, no. 4, pp. 2797–2804, April 2013.
[31] D. A. G. Gonçalves de Oliveira, L. L. Pilla, T. Santini, and P. Rech, "Evaluation and mitigation of radiation-induced soft errors in graphics processing units," *IEEE Transactions on Computers*, vol. 65, no. 3, pp. 791–804, March 2016.
[32] D. A. G. Oliveira, P. Rech, H. M. Quinn, T. D. Fairbanks, L. Monroe, S. E. Michalak, C. Anderson-Cook, P. O. A. Navaux, and L. Carro, "Modern gpus radiation sensitivity evaluation and mitigation through duplication with comparison," *IEEE Transactions on Nuclear Science*, vol. 61, no. 6, pp. 3115–3122, Dec. 2014.
[33] Nvidia, *NVIDIA Tegra X1 white paper*, January, 2015.
[34] NVIDIA, *CUDA toolkit documentation: CUDA runtime API*, 2022.
[35] ESA/ESCC, "Single event effects test method and guidelines. escc basic specification no. 25100," Oct 2014.