Badía, José M. … et al. (2022). Comparison of Parallel Implementation Strategies in GPU-Accelerated System-on-Chip Under Proton Irradiation. *IEEE Transactions on Nuclear Science*, 69(3), pp.: 444-452.

DOI: https://doi.org/10.1109/TNS.2021.3128722

Comparison of parallel implementation strategies in GPU-accelerated System-on-Chip under proton irradiation

Jose M. Badia, German Leon, Jose A. Belloch, *Member, IEEE,* Mario Garcia-Valderas, *Member, IEEE,* Almudena Lindoso, *Member, IEEE,* and Luis Entrena *Member, IEEE,*

***Abstract*—Commercial Off-The-Shelf (COTS) System-on-Chip (SoC) are becoming widespread in embedded systems. Many of them include a multicore CPU and a high-end GPU. They combine high computational performance with low power consumption and flexible multilevel parallelism. This kind of device is also being considered for radiation environments where large amounts of data must be processed or compute intensive applications must be executed. In this paper we compare three different strategies to perform matrix multiplication in the GPU of a Tegra TK1 SoC. Our aim is to analyze how the different use of the resources of the GPU influences, not only the computational performance of the algorithm, but also its radiation sensitivity. Radiation experiments with protons were performed to compare the behaviour of the three strategies. Experimental results show that most of the errors force a reboot of the platform. The number of errors is directly related with how the algorithms use the internal memories of the GPU, and increases with the matrix size. It is also related with the number of transactions with the global memory, which in our experiments is not affected by the radiation. Results show that the smallest cross-section is obtained with the fastest algorithm, even if it uses the cores of the GPU more intensively.**

***Index Terms*—GPU, Embedded Systems, proton irradiation, parallelization.**

I. Introduction

System-on-Chip (SoC) devices composed of low-power multicore processors and small Graphics Processing Units (GPUs) are very attractive because they offer a trade-off between computational capacity and low-power consumption. Graphics Processing Units (GPUs) are highly parallel programmable co-processors that can accelerate many applications if they leverage their many cores and large and fast memories. GPUs offer multiple parallelism levels; however, properly managing their computational resources becomes a very challenging task. Embedded systems with SoC including GPUs are used in critical domains, such as advanced driver assistance systems [1], avionics or space applications [2], [3].

New space missions combine the need to process very large volumes of data and execute compute intensive applications. Current space qualified processors do not fulfil those requirements. Therefore, Commercial Off-The-Shelf SoCs including Central Processing Units (CPUs), GPUs and, in some cases, Field Programmable Gate Arrays (FPGAs), are being considered in space missions as an alternative to ad-hoc rad-hard microprocessors because they can deal with the new computing requirements without increasing the power consumption [4], [5]. However, radiation effects on this kind of device must be analyzed and fault tolerance or mitigation techniques must be implemented in order to use them in harsh environments.

In this paper we analyze the radiation sensitivity of a GPU-accelerated SoC device when running different parallel implementations of a representative application benchmark, namely a matrix multiplication kernel. Those implementations make a very different use of the resources of the GPU, such as the cores, memories and warp schedulers, among others. For example, while one of the algorithms performs a large number of transactions from global memory and L2 cache, the other two leverage the shared memory. Therefore, they have a very different probability of being affected by a particle impinging those components of the GPU. As a consequence, the three algorithms get very different performances, but also different behaviour in terms of radiation-induced errors. Most papers use different benchmarks, such as linear algebra routines, Fast Fourier Transform (FFT) or even neural networks, to evaluate the radiation sensitivity of GPUs. However, very few evaluate the effect of using different strategies to solve the same problem.

Our main goal is to assess how the use of the components of the device affects the radiation sensitivity of the algorithms, and also to detect trade-offs that allow us to obtain maximum computational performance and minimum error rate. Two parameters are commonly used when comparing the performance of different strategies to implement linear algebra routines in COTS GPUs, namely, computational performance and, more recently, energy consumption [6]. In this paper we add radiation sensitivity as a third parameter to take into account. To this end we performed different radiation campaigns executing 1000 matrix multiplications with each algorithm with different matrix sizes. We compare the cross-section of the algorithms and also the percentages of the different types of soft errors occurred during each radiation campaign. We carried out our experiments on a Tegra TK1 SoC using a proton beam with an energy of $15.4MeV$ and a fluence of $3.4 \times 10^{12} p/cm^2$. This device was designed to

This work has been supported by the Valencian Regional Government through PROMETEO/2019/109, the University Jaume I project UJIB2019-36, and the Spanish Ministry of Science and Innovation under projects PID2019-106455GB-C21 and PID2020-113656RB-C21.

J. A. Belloch, M. Garcia-Valderas, A. Lindoso and L. Entrena are with the Depto. de Tecnología Electrónica, Universidad Carlos III de Madrid, Spain.

J. M. Badía and G. León are with the Depto. de Ingeniería y Ciencia de Computadores, Universitat Jaume I, 12071, Castellón, Spain.

provide high performance with low power consumption on mobile devices. It is a COTS device, and so it was not designed to be used in radiation harsh environments.

Results show that the memory-bound strategy with lower performances is the most error prone and that the cross-section increases with the matrix size. Finally, preliminary results show that the spatial distribution of the errors in the matrix depends on the implementation strategy.

The main contributions of our work are the following:

- Three different algorithms to perform matrix multiplication have been analyzed under proton irradiation. We have quantified how the algorithms use some of the main resources of the GPU that can affect their error sensitivity.
- Experiments have been performed on a COTS GPU-accelerated SoC including the CPU and GPU, but without focusing the radiation beam on the DDR memory.
- The cross section of the three algorithms and also the percentages of the different kinds of errors has been analyzed.
- Some comments about the spatial distribution of the errors of the three algorithms have been advanced.

The rest of the paper is structured as follows. Section II summarizes the related work. Section III introduces the GPU architecture and its programming model. Section IV describes the experimental setup and benchmark. Section V presents the experimental results. Finally, section VI summarizes the main conclusions.

II. Related work

GPUs have become ubiquitous accelerators both in HPC centers [7], [8] and safety-critical environments [1], [5]. They provide massive data parallelism, with high performance per watt and programming flexibility. However it is important to guarantee that this kind of device meet the reliability requirements of many applications, more so when they are used in radiation environments. Radiation including protons, neutrons, heavy-ions or electromagnetic radiation can produce Single Event Effects (SEE) on this kind of device. Impinging particles may produce bit-flips in memory elements or generate transient voltage pulses in combinational logic [9], [10]. Transient SEEs, also called soft errors, can be very common for example in space environments and they should be evaluated and avoided or, at least, mitigated.

Therefore, hardware and software hardening strategies should be designed, implemented and evaluated to increase the reliability of this kind of device. A considerable number of papers have been published during the last decade evaluating the reliability under radiation of GPUs. Most of the experiments have been performed on high-performance NVIDIA architectures, including Fermi, Kepler or Pascal using neutron beams [11]–[13]. Several papers have evaluated the efficiency, efficacy and overhead of the hardware hardening ECC mechanism provided by most modern GPUs [14], [15]. Most of them conclude that this protection mechanism significantly reduces the SDCs, but increases the Single Event Functional Interrupts (SEFI) of the algorithms, for example when multiple-bit faults arise. Other authors have evaluated different software hardening strategies, such as Algorithm Based Fault Tolerance (ABFT), Duplication With Comparison (DWC) or Triple Modular Redundancy (TMR) [11], [16].

Matrix multiplication is one of the most used benchmarks to evaluate the radiation sensitivity of GPUs, because it is a basic component in many applications, such as signal or image processing, audio editing or machine learning. Some papers analyze not only the SDCs and SEFI rates of matrix multiplications, but also the spatial error distribution on the result matrices [7], [13]. The effect of the degree of parallelism in the soft-error sensitivity of this operation is analyzed in [17]. The authors evaluate the effect of modifying the scheduling strain and the usage of resources by using different grid and thread block sizes. A memory-bound and a compute-bound version of matrix multiplication have been compared in [11]. The authors use high-energy neutron radiation to evaluate their behaviour on two high-performance NVIDIA GPUs with different architectures, namely Fermi and Kepler. Their experimental results show that increasing the performance of the multiplication can also increase the radiation-induced errors. In this paper we add a third matrix multiplication algorithm (`Cublas`) which optimizes the use of the GPU obtaining much higher performance. We conduct the experiments on a low-power GPU included on a SoC and using proton irradiation, which also affects the CPU of the device but not its global memory.

Radiation tests have also been performed with different GPU-accelerated SoC [18]. For example, Jetson TX1 and TX2 platforms were tested in [19], [20] using proton irradiation to provide a baseline assessment of their radiation susceptibility. Results showed upsets in all tests with different types and crash conditions, but power cycle mitigated all non-destructive events. Jetson TX1 and Snapdragon 820 were irradiated in [21] using protons, heavy ions and also laser testing.

Radiation evaluations show that different types of Jetson SoCs are acceptable for many low earth orbit short duration missions using the proper mitigation techniques. For example, high-energy protons are used in [22] to evaluate the cross-section and the total dose performance of the Tegra K1 SoC. In [9] gamma-ray photons were employed to evaluate the tolerance to radiation effects of a Jetson Nano board, which includes a 128-core NVIDIA Maxwell GPU. Preliminary results suggest operation beyond 20 krad(Si). Similar conclusions were reached in [23] for a Jetson AGX Xavier board, including a 512-core NVIDIA GPU, using proton irradiation.

Results of testing several Snapdragon devices with different scaling technology and including different Adreno GPUs are described in [24]–[26]. Specifically, SEE tests results for the Snapdragon 820 using proton, neutron and heavy ion radiations are reported in [24]. Authors state that the interpretation of the results is complicated by mixing of errors between the components of the platform. Results with other three Snapdragon devices are reported in [25], where experiments evaluate the SEE and cross-sections of the devices for different events and energy levels.

All in all, radiation experiments using different kinds of particles on COTS GPU-accelerated SoC show that SEFI errors are a common problem in this type of device. However,

in almost all cases the error is solved by rebooting the device and Single Event Latchups (SEL) are very rare.

We have only found two papers where the authors perform the same kind of experiments that we carry out with this kind of device. In [27] the authors evaluate under neutron radiation three NVIDIA GPUs, including a Jetson TX1 SoC. They evaluate three neural networks and also a matrix multiplication algorithm. Their results show that the matrix multiplication crash rates are lower than the Silent Corruption Data (SDC) rates. The authors also state that the Tegra TX1 has a higher crash rate than the other two GPUs because the CPU of the board is also irradiated. Finally, they conclude that the transistor technology can have a significant impact on the reliability of the GPUs running neural network algorithms. They show that the error rate of FinFET devices is an order of magnitude lower than the error rate of planar CMOS devices. Similar matrix multiplication results with the same device are also included in [28], where the authors analyze the dependence of the radiation-induced errors on the code and the architecture of six different devices, including an Intel co-processor, three NVIDIA GPUs, an AMD APU, and an embedded ARM.

III. GPU ARCHITECTURE AND PROGRAMMING

The Compute Unified Device Architecture (CUDA) [29] was defined by NVIDIA and can only be used with GPUs of this company. However, OpenCL, which is an open source language with a very similar programming model, can be used with many other GPUs and also with some multicore CPUs [30]. CUDA is based on an array of Streaming Multiprocessors (SM). Every SM contains multiple CUDA cores, which can execute in parallel multiple elementary processes, called threads. The threads are logically grouped into thread blocks which are dispatched to an SM and can leverage its shared memory. Thread blocks are then divided in *warps* of size 32, which are scheduled to be executed on the cores of the SM. Thread blocks are organized in a grid. CUDA programs combine a host code run on the CPU with one or several kernel functions to be executed in the CUDA cores using a Single Instruction Multiple Threads (SIMT) model. That is, the same instruction is dispatched to the cores, so that one thread per core executes it. Usually, each core executes the instruction using different data, which can be stored in registers assigned to the thread, in the memory shared with other threads of the same block or in global memory of the GPU. SIMT model combines threads with the Single Instruction, Multiple Data (SIMD) model, which was one of the computer architectures included in Flynn's taxonomy [31].

In [29] there is a description of the main parameters that determine the performance of the algorithms running on GPUs. Firstly, the GPU occupancy, which is defined as the ratio of active warps on an SM to the maximum number of active warps supported by the SM, which depends on the GPU architecture. An SM usually contains more than one warp scheduler and, in order to hide latencies between dependent instructions, schedulers must have enough warps to dispatch an instruction every clock cycle. Therefore, maintaining as many active warps as possible throughout the execution of the kernels helps to avoid clock cycles where no instruction is executed in the cores. Another method to increase the performance is to leverage registers that are local to each thread and the fast memory shared by the threads of each block to reduce the negative impact of accessing the slow global memory. Besides, data should be distributed in memory so that most of the accesses can be coallesced, thus reducing the number of load operations. That is, threads with consecutive identifiers in its block should access adjacent elements in memory whenever possible.

There are several limiting factors to the performance associated with the resources of the GPU architecture. Table I describes the physical limiting factors in the case of the K20A GPU used in our experiments. Those factors are the maximum number of blocks and warps that can be active at once on each SM and the maximum number of registers and shared memory that can be assigned to a thread block. Programmers should implement their algorithms taking into account those limits. For example, one of the limits of the K20A GPU is that there cannot be more than 16 thread blocks scheduled in its only SM. If we select a very small block size including only 32 threads (1 warp), we can only have 16 active warps on the GPU. This means that, in this case, we can only reach 25% of the maximum number of active warps supported by the GPU, and so, we can only reach 25% of its theoretical occupancy. Nevertheless, as we will see when comparing the three CUDA matrix multiplication algorithms, getting the maximum occupancy does not always guarantee the best performance.

A. Device under test

We have used as DUT a Tegra K1 (TK1) System-on-Chip (SoC), embedded in the Jetson development kit [32]. TK1 was launched by NVIDIA in 2014 as a powerful and flexible device for mobile devices. As stated by NVIDIA in [33] "Jetson TK1 will enable a new generation of applications for computer vision, robotics, medical imaging, automotive, and many other areas".

The main components of the System-on-Chip are displayed in Figure 1. The TK1 SoC is fabricated on the 28 nm High Performance Mobile (HPM) process, which reduces its size and power demand. It was the first mobile processor to use 4-PLUS-1 ARM Cortex A15 CPU architecture and variable Symmetric Multiprocessing (vSMP) technology. It combines four high-performance A15 CPU complex cores for performance intensive tasks, and switches to the power optimized "battery saver" A15 CPU core to handle low performance tasks.

This SoC includes an NVIDIA "Kepler" K20A GPU with compute capability 3.2 [34]. It includes a number of optimizations for mobile system usage to conserve power and deliver industry-leading mobile GPU performance. In contrast with the highest-end Kepler GPUs with thousands of cores that consume a few hundred watts of power, the K20A consumes only a few watts [32].

This GPU includes one Streaming Multiprocessor (SM) containing 192 CUDA cores, each including fully pipelined

floating-point and integer arithmetic logic units. The SM features four warp schedulers and eight instruction dispatchers. It can then select four warps of 32 threads and issue two independent instructions per warp each cycle. The SM has 64 KB of on-chip memory that can be configured as 48 KB of shared memory and 16 KB of L1 cache or vice versa. The GPU also includes 128 KB of L2 cache and 65536 registers of 32 bits that can be distributed among the thread blocks. Finally, both the CPU and GPU share four DDR3 modules with a total of 2 GB of memory. It is worth mentioning that, contrary to other similar Kepler GPUs, the internal memory of the K20A GPU and the DDR3 memory do not support Error Correction Codes (ECC).

TABLE I
PHYSICAL FACTOR LIMITS OF THE K20A GPU.

Physical limit	**value**
Threads per Warp	32
Max Warps per SM	64
Max Thread Blocks per SM	16
Max 32-bit registers per Thread	255
Max 32-bit registers per Thread Block	65536
Max Shared Memory per Thread Block (bytes)	49152

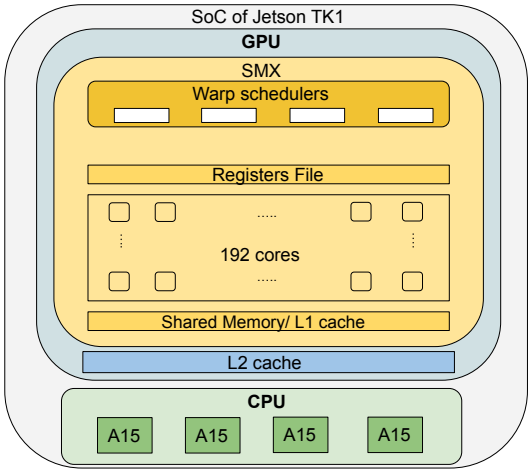

Fig. 1. Main components of the Tegra K1 SoC.

IV. DESCRIPTION OF THE EXPERIMENTS

A. Setup and procedure

Experiments were accomplished at CNA - Centro Nacional de Aceleradores (Sevilla, Spain) in January 2021. A low energy proton campaign was performed in the external beam line installed in the 18/9 compact cyclotron (see Figure 2). The average proton flux was approximately $1.3\times10^{8}p/(cm^{2}.s)$ for energies of $15.4MeV$, with a homogeneous spot of 1.5 cm of diameter. The total fluence after all the radiation runs was of $3.4\times10^{12}p/cm^{2}$. The irradiation affected all the components of the SoC, but the four DDR memory modules shared by the GPU and CPU were not exposed, neither was the SD card that hosts the operating system. Therefore, the matrix elements transferred to the GPU were not affected by the beam radiation while stored in the global memory.

The Jetson TK1 board sent the logs of the tests to a host controller through the serial communication. The host controller is around 1.5 meters apart from the DUT and not under direct beam exposure. The controller was also connected to the GPIO pins of the DUT so that it can be used to remotely reset the Jetson TK1 board when needed. The whole test was managed remotely from a laptop connected to the host controller through ethernet. The operating system Ubuntu 14.04 with the CUDA 6.5 driver was run from the SD card of the Jetson TK1 board, so that we avoided the radiation effect on the system files.

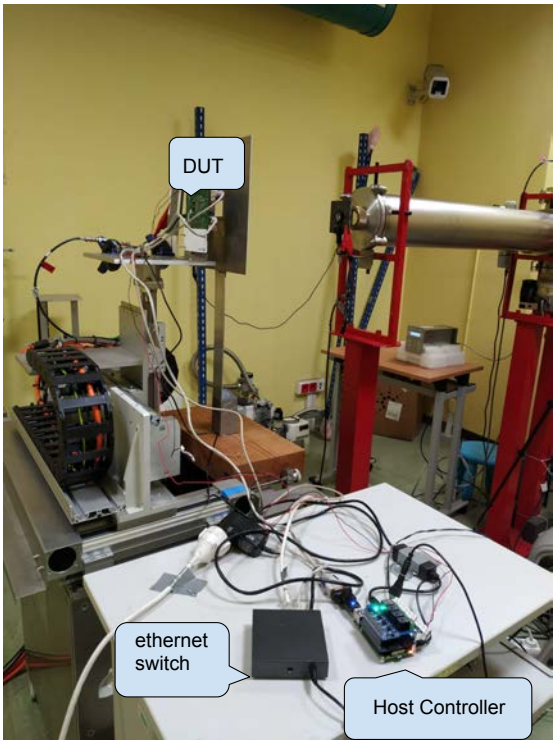

Fig. 2. Radiation test setup at CNA.

The matrix multiplication algorithms were programmed using C and we used a Python 2.7 script to run the different benchmarks. We used the Python module `Pexpect` to spawn and control a subprocess in charge of running each matrix multiplication. Our experiments included three watchdogs to detect and recover from different types of hangs, including those related to the operating system. The first timeout, associated with the spawned process, was set to a time larger than the maximum expected duration of one multiplication. The second watchdog was executed in the Jetson board and leveraged the Watchdog Timer (WDT) included in the board. It was configured to reboot the operating system if it was not kept alive by a process during more than 20 seconds. We assumed that the operating system hung when this process did not perform this task. Finally, a third watchdog was executed on the host controller, so it could reset the device if the Jetson operating system hung. Namely, the results of each matrix multiplication was being continuously sent to the controller through the serial port. Besides, the process in charge of keeping alive the WDT also sent messages to the controller

every 10 seconds. If none of those messages arrived to the controller during more than 20 seconds, we assumed that the Jetson operating system had hung and remotely performed a hardware reset of the board using its GPIO pinout.

B. Matrix multiplication benchmark

We have analyzed the radiation effects on the whole SoC, affecting the CPU cores and all the components of the GPU, including its internal memories. However, global memory, which is out of the SoC, has not been exposed. To test the radiation reliability of the SoC, we have used as benchmark three well-known parallel versions of matrix multiplication implemented with CUDA:

- `Elem` is a straightforward implementation of matrix multiplication, $C = A \cdot B$, where every thread computes one element $C[i, j]$ of the result matrix as the dot product of the i-th row of matrix A and the j-th column of matrix B. Every thread loads all the elements of both matrices to perform the dot product from *global memory*, and also stores the result in the same memory.
- `Block` is a block version of matrix multiplication. Every thread-block is in charge of computing one square block of matrix C, and each thread of the block is in charge of computing one element $C[i, j]$. This block of C is computed as the product of a row of blocks of matrix A and a column of blocks of matrix B. To compute each product of a block of A and a block of B, every thread starts by loading from global to shared memory one element of each matrix, in parallel with the rest of threads of its thread-block. Then every thread updates in a register its element of C by performing a small dot product between a row of its block of A and a column of its block of B. Finally, once finished all its products of blocks, every thread stores in global memory its element $C[i, j]$
- `Cublas` uses the `cublasSgemm` optimized routine included in the CUBLAS library [35], based on the classical BLAS library [36]. It combines blocking and loop unrolling strategies adapted to the characteristics of the GPU to optimize the performance of the algorithm. It is a closed code, so we cannot know the details of its matrix multiplication implementation.

The algorithms `Elem` and `Block` are included in the "CUDA_C Programming Guide" [29] as a representative example of the effect of leveraging the *shared memory* of the GPUs to minimize *global memory* accesses. The first algorithm is a straightforward implementation of matrix multiplication that does not take advantage of *shared memory*. On the contrary, the algorithm `Block` uses a typical blocking strategy employed in linear algebra libraries to improve the performance of the routines by increasing the ratio of floating point operation to memory accesses. In the case of the GPUs, the blocking strategy saves a lot of *global memory* transfer bandwidth by leveraging the fast *shared memory*. Regarding the `Cublas` algorithm, we have used `nvprof` profiler [37] to analyze how it uses the resources of the GPU (see Table II). It uses an amount of shared memory very similar to the algorithm `Block`, but it uses a much larger number of registers per thread than the other two algorithms. Registers are used to store variables locally to each thread and are the fastest memory available in GPUs. `Elem` and `Block` use 17 and 26 registers per thread respectively, while `Cublas` leverages 127 registers per thread to reduce the number of accesses even to the fast *shared memory*, and so greatly increases the performance of the algorithm.

C. GPU resources usage and computational performance

We have used the CUDA profiler nvprof [37] in order to assess and quantify the GPU resources usage of each of the three algorithms described in the previous section. We have chosen a few of the metrics provided by the profiler that allow us to measure how each algorithm uses the main components of the GPU. Table II shows the values of those metrics for the three matrix multiplication algorithms.

TABLE II
PROFILING METRICS THE GPU RESOURCE USAGE WITH MATRICES OF SIZE 1024×1024.

Metric	Elem	Block	Cublas
Achieved Occupancy (%)	99,46	99,87	24,97
Eligible Warps Per Cycle	7,26	6,34	3,41
Executed IPC	1,20	1,16	4,60
Number of Registers	17	26	127
Shared mem (B/th-blk)	0	8192	8340
DDR Load Trans. (M)	67,11	2,1	2,06
L2 Read Trans (M).	167,81	8,40	2,09
Shared Load Trans. (M)	0	50,33	2,11
FLOP Efficiency (Peak Single) (%)	3,31	7,12	68,88

Profiling results show that the algorithms `Elem` and `Block` do not exceed any of the limiting factors of the GPU, and so both algorithms reach an occupancy very close to the theoretical value of 100% obtained using the CUDA Occupancy calculator (https://docs.nvidia.com/cuda/cuda-occupancy-calculator). On the contrary, the algorithm `Cublas` can only keep 16 of the 64 warps per SM allowed by the architecture active because it needs $127 \times 128 = 32K$ registers per thread-block of size 16×16. Therefore, this algorithm can only reach an occupancy of 25%, as it is shown in Table II. However, 16 active warps are enough to efficiently leverage the 192 cores and reach much better performances than the other two algorithms. Specifically, the algorithm `Cublas` is 10x faster than `Block` and 20x faster than `Elem`, for all matrix sizes, as we can see in Figure 3.

Another two metrics included in Table II are directly related with the use of the cores of the GPU and their floating point arithmetic unit. Specifically, the table shows that while the algorithm `Cublas` executes on average 4.6 Instructions per Cycle (IPC), the other two algorithms execute slightly more than 1 IPC. Even more significant regarding the execution time of the three algorithms is their single floating point efficiency with respect to the peak performance of the device. As we can see `Cublas` reaches 68.88% of the peak, which is 20x larger the percentage reached by `Elem` and 10x larger the percentage reached by `Block`. These values exactly reflect the relative execution time of the three algorithms.

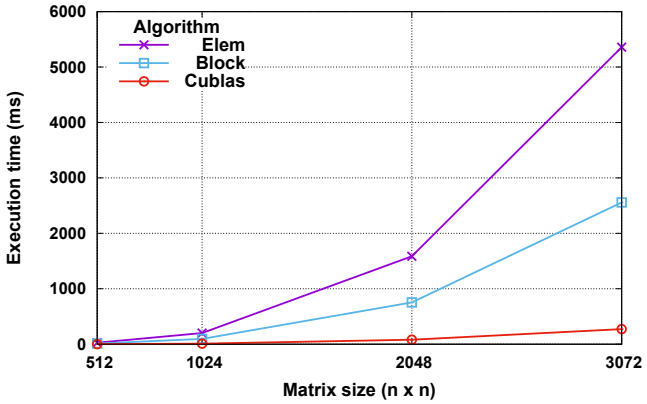

Fig. 3. Execution times vs matrix size of all the algorithms.

The use of the different memories of the platform is also related with the computational efficiency of the algorithm. For example, we have included in Table II the number of DDR Load Transactions to show how the algorithm `Elem` makes a much more intensive use of the main memory of the device. As a consequence, this algorithm also uses much more intensively the L2 cache of the GPU, as it is shown for example by the number of L2 Read Transactions included in the table. Finally, the table also shows that the algorithm `Cublas` makes a much more efficient use of the shared memory than the algorithm `Block` as it greatly reduces for example the number of Shared Load Transactions.

V. Experimental results and discussion

We performed several experiments to evaluate the behaviour of the device under radiation depending on the algorithm used to carry out matrix multiplication and also on the size of the matrices. Single precision elements were used in all the experiments. We run 1000 matrix multiplications for each combination of algorithm and matrix size.

The values of the elements of the matrices are computed as a function of their row and column, so that verifying the result of the product has a very low cost. This way, they always have the same value in all the experiments with matrices of the same size and for the three algorithms. Radiation experiments were always performed using the thread-block size that produces the fastest execution for each algorithm. Thus, algorithms `Elem` and `Block` were always launched using a thread-block size of 32×32. Meanwhile, a thread-block size equal to 16×16, adapted to the Kepler GPU capability, was always automatically chosen as the optimal one by the `cublasSgemm` function used in `Cublas` algorithm.

We launched our first experiments using a proton flux of $4.0 \times 10^8 p/(cm^2 s)$. However, with this flux the device repeatedly hung during the boot process or after launching the first one or two matrix multiplications. We had to restart the device several times without being able to reduce the SEFI occurrences that hang the operating system. Therefore, we decreased the flux to $1.2 \times 10^8 p/(cm^2 s)$ in order to be able to launch at least 10 consecutive matrix multiplications without hanging the test or the operating system. All the experiments were performed with an average flux between $1.0 \times 10^8 p/(cm^2 s)$ and $1.4 \times 10^8 p/(cm^2 s)$.

One of the main problems to know the causes of the errors occurred while radiating a GPU-accelerated SoC is that it is affecting all the components of the device, including its CPU and GPU. The number and type of errors can depend on the processes being executed on both components of the SoC and also on the percentage of time of each test that the matrix multiplication is being executed in the GPU. For example, when using the algorithm `Elem` to multiply matrices of size 1024, 95% of the time of the test is devoted to the kernel executing the matrix multiplication, while the remaining 5% of the time the CPU is initializing the matrices and verifying the result, and the GPU is idle. In the case of the algorithm `Block` those percentages are 87% and 13% respectively. The percentages in the case of the algorithm `Cublas` are very different. Only 16% of the time of the test is devoted to executing the matrix multiplication, while 84% is devoted to initializing the CUBLAS library, transferring the matrices and verifying the result. The initialization time takes most of the time of the CPU and is independent of the size of the matrices, thus the percentage of the time of the matrix multiplication in the GPU grows with this size. Therefore, most of the errors detected in our experiments with the first two algorithms are probably due to the effect of the radiation on the kernel being executed in the GPU. Meanwhile, a larger portion of the errors detected in the case of the algorithm `Cublas` can be caused by the particles impinging the processes being executed in the CPU.

A. Cross section results

Figure 4 shows the cross-section of the three matrix multiplication algorithms running on the GPU, including the 95% confidence intervals. Results correspond to matrices of size 1024×1024. Cross sections have been computed including all the errors detected. Specifically, the Figure shows that the cross-section of the errors depends on the matrix multiplication algorithm running on the GPU. One of the main differences among the three algorithms, that produces quite different performances, is their use of the *global memory* of the GPU. Specifically, `Elem` algorithm totally depends on the *global memory*, while the other two algorithms greatly reduce their cost by leveraging the *shared memory* and the registers of the GPU. In the case of the Tegra TK1 device, the global memory is the same used by the CPU and was not exposed to radiation. However, matrix elements must be loaded from *global memory* by the GPU threads and may temporarily reside in the L2 and L1 caches. Therefore, if we increase the data loads, as in the `Elem` algorithm, we increase the possibility that particles affect elements stored in those internal memories that will be used by the threads. Table II helps us to quantify this behaviour. We can see how the algorithm `Elem` makes many more load transactions from global memory and also many more L2 read transactions than the other two algorithms. It is worth recalling that the internal memories of the K20A GPU are not protected by any ECC mechanism. We can see that the radiation sensitivity of the algorithm decreases when

it makes a better use of the *shared memory* and reduces the data transfers from *global memory*. In the same sense, the algorithm `Block` uses more intensively the *shared memory* than the algorithm `Cublas` and this can increase its cross section.

As we increase the size of the matrices, so increases the use of the different memories of the GPU. Figure 5 shows that even in the case of the `Block` algorithm, that reduces the accesses to *global memory* by leveraging the *shared memory*, the cross-section increases linearly with the size of the matrices.

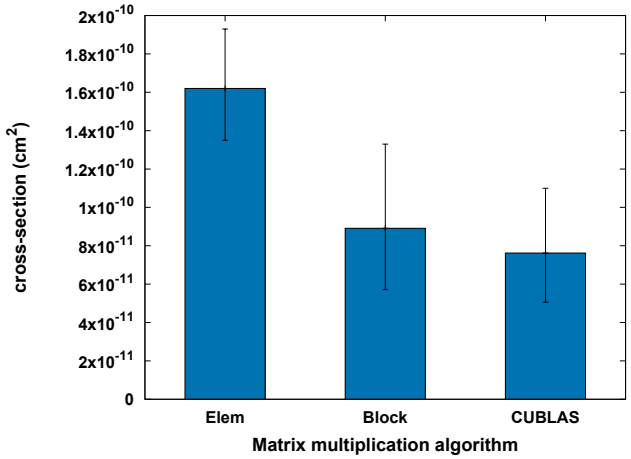

Fig. 4. Cross-section of the matrix multiplication algorithms.

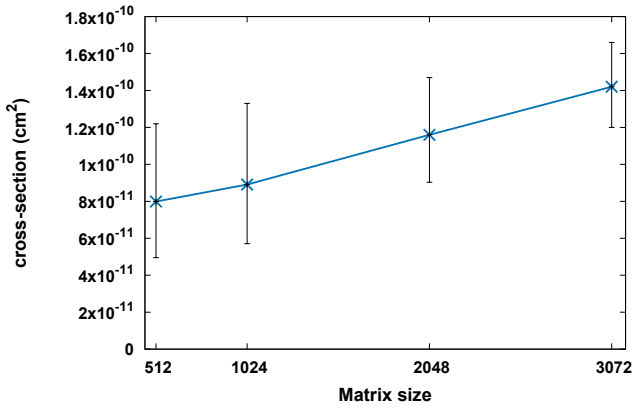

Fig. 5. Effect of the matrix size on the cross-section of the `Block` algorithm with matrices of different sizes.

All in all, it seems that the use of the internal memories of the GPU is more critical to the cross section of the algorithms than the use of other resources. Table II shows that the algorithm `Cublas` makes a much intensive usage of the cores and floating point units, but it has the lowest cross section because it greatly reduces the transactions from the DDR and L2 memories with respect to the algorithm `Elem`, and it reduces the transactions from the *shared memory* with respect to the `Block` algorithm.

B. Types of errors

We have also analyzed the types of errors produced on the different experiments using the following classification:

- `Fail`: the GPU kernel performing the matrix multiplication crashes or finishes with an SDC, that is, the matrix multiplication produces a wrong result that can affect from one to thousands of elements of the result.
- `Timeout`: the GPU kernel performing the matrix multiplication hangs and the software watchdog interrupts the CPU process that launched it. Afterwards, a new matrix multiplication can be run successfully without restarting the device.
- `SEFI_test`: The system process launching the test hangs and is unable to launch the next matrix multiplication. The Jetson TK1 is then rebooted after 20 seconds by the hardware watchdog.
- `SEFI_restart`: The restart process of the Jetson hangs and the device is remotely rebooted by a watchdog running in the external host that controls the experiment. All these errors are due to particles affecting the processes being executed in the CPU during the reboot process.

Most of the matrix multiplication executions on the GPU were not affected by the radiation and produced correct results. Figures 6 and 7 show that the number of events that affect the matrix multiplication process or the Operating system processes depends on the algorithm and also on the size of the matrix. Specifically, Figure 6 shows that the algorithm `Elem` is the one most affected by all types of errors, which impacts on more than a 10% of its executions. On the contrary, more than 97% of the executions of the other two algorithms finish with the correct result. The percentage of executions affected by an error increases with the size of the matrix, as can be seen in Figure 7 in the case of `Block` algorithm.

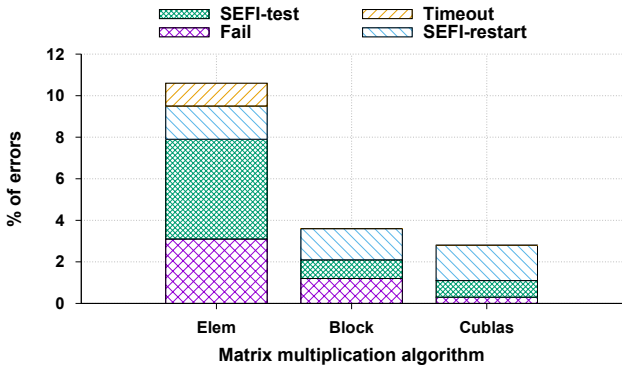

Fig. 6. Percentage of the different errors over 100% of the tests with each algorithm using matrices of size 1024.

Figure 6 also shows that most of the radiation-induced errors hang the operating system and force a reboot of the platform. Only a small percentage of those errors crashes the GPU-based multiplication or produces an SDC. We can also see that the number and percentage of SEFI_restart does not depend on the algorithm, as this kind of error is due to the effect of the radiation on the CPU during the reboot of the platform. Meanwhile, the number of SEFI_test depend on the algorithm being executed in the GPU. It is close to 1% for the algorithms `Elem` and `Block` and close to 5% for the algorithm `Cublas`

Obviously, a large number of unrecoverable errors that force the platform to reboot is unacceptable on safety critical

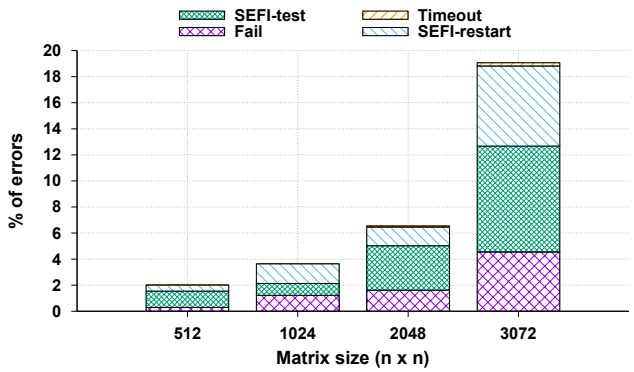

Fig. 7. Effect of the matrix size on the percentage of errors in the `Block` algorithm with matrices of different sizes. Results over 100% of the tests with each size.

applications. Therefore, some fault tolerance or mitigation techniques, such as some level of redundancy of the algorithms or the devices, should be applied to reduce this kind of error.

TABLE III
NUMBER OF THE DIFFERENT KINDS OF ERRORS FOR THE THREE ALGORITHMS. 1000 MATRIX MULTIPLICATIONS PER ROW.

Algorithm	Size	FAIL	SEFI_test	SEFI_restart	Timeout
Elem	1024	31	48	16	11
Block	512	3	12	5	0
	1024	12	9	15	0
	2048	16	34	14	1
	3072	46	81	62	2
Cublas	1024	3	8	17	0

Table III shows the number of errors of each type for the different algorithms used to draw the previous figures. One additional and interesting fact to point out is that the number of Fails can help us to highlight the effect of the radiation on the GPU. This kind of error is due to the effect of the particles in the algorithm being executed on this device. A remarkable fact shown in the Table III and also in Figure 6 is the very small number of Fails of the algorithm `Cublas` with matrices of size 1024 when compared with the other two algorithms. This behaviour could be partially due to the small percentage of the time of the test that the GPU is executing the matrix multiplication (16%). Thus, most of the particles impinging the GPU during each test will not cause any detectable error.

One of the main drawbacks of using radiation to evaluate the soft error sensitivity of GPUs is that it is very difficult to establish which component of the device caused the error and how it affected the threads and the results of the benchmarks being executed. This problem worsens in the case of SoCs, because the error can be also due to particles impinging the processes run in the CPU or affecting the data transferred between CPU and GPU. An alternative method to analyze the vulnerability of different components of the GPU is to inject faults at the architecture-level or into compiler-level intermediate representations [38], [39]. In [40] we employed the LLFI injection tool to evaluate the soft error sensitivity of the algorithms `Elem` and `Block` in the same Jeston TK1 used in this paper. We injected single bit-flips in the results of the instructions of randomly chosen threads. The results show that, as in the case of the radiation results analyzed in this paper, the most efficient algorithm was also the least sensitive to this kind of fault injection. By using the CUDA debugger we found out that almost all the crashes of the algorithm `Elem` were due to illegal accesses to global memory, while most of the crashes of the algorithm `Block` were due to illegal accesses to shared memory. These results can give us indications about some of the causes of the crashes and SEFI_test errors detected during the radiation campaigns. However, many of the SEFI_test errors may be caused by the radiation effect over components of the SoC that are not affected by the kind of injection performed using the LLFI tool.

C. Spatial distribution of the errors

We have also analyzed the spatial distribution of the errors whenever a SDC happened during one of the experiments. However, as the number of SDCs per benchmark is so small it is not possible to perform any kind of statistical analysis of the results, nor compare the behaviour of the different algorithms. We can advance a few comments on the results, but they should be taken with caution. Most of the SDCs of the algorithm `Elem` modified only one element of the result matrix, and only in one case 448 elements of the same column were affected. In the case of the algorithm `Block` most of the SDCs modified between a few tens to some hundreds of elements, but in every case all the affected elements were in the same row. Finally, the algorithm `CUBLAS` was only affected by one SDC that modified one element of the result matrix.

We know that in the algorithm `Elem` every thread computes one element of the result matrix without sharing any intermediate result with other threads. It reads all the elements from the global memory, which is not directly exposed to the beam radiation. Only if some of the elements were read from the internal caches of the GPU, the radiation could have affected data used by several threads of the algorithm. This almost complete isolation of the resources used by each thread could justify that most of the SDCs affected only one element. On the contrary, in the algorithm `Block` every thread also computes one element of the result, but all the threads of each block read and write the same elements stored in the shared memory. Therefore, any bit-flip on one shared element can affect several elements of the result matrix, which usually will be in the same row or column as they are computed by threads in the same block of threads.

VI. CONCLUSIONS

Radiation experiments with a GPU-accelerated Tegra TK1 SoC show that if we choose the appropriate parallelization strategy to implement basic linear algebra routines such as matrix multiplication, we can greatly reduce radiation induced errors. Most of those errors are unrecoverable and thus, additional fault tolerance or mitigation techniques should be applied if this kind of device is used in radioactive environments, such as space.

Our experimental results using high-energy protons show that cross section depends on how the algorithm uses the resources of the GPU. Specifically, the slower memory-bound

algorithm is more error prone, while the most efficient algorithm gets the smallest cross section. The cross section increases with the size of the matrices, as we make a more intensive usage of the internal memories of the GPU. This behaviour is not due to the increasing in the global memory area affected by the radiation because this memory is not affected by the radiation. Finally, results show that the spatial distribution of the errors in the result matrix seems to depend on the algorithm.

REFERENCES

[1] J. Fickenscher, S. Reinhart, F. Hannig, J. Teich, and M. E. Bouzouraa, "Convoy tracking for ADAS on embedded GPUs," in *2017 IEEE Intelligent Vehicles Symposium (IV)*. IEEE, 2017, pp. 959–965.

[2] F. Bruhn, K. Brunberg, J. Hines, L. Asplund, and M. Norgren, "Introducing radiation tolerant heterogeneous computers for small satellites," in *2015 IEEE Aerospace Conference*, 2015, pp. 2586–2596.

[3] M. Benito, M. M. Trompouki, L. Kosmidis, J. D. Garcia, S. Carretero, and K. Wenger, "Comparison of GPU Computing Methodologies for Safety-Critical Systems: An Avionics Case Study," in *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2021, pp. 717–718.

[4] G. Lentaris, K. Maragos, I. Stratakos, L. Papadopoulos, O. Papanikolaou, D. Soudris, M. Lourakis, X. Zabulis, D. Gonzalez-Arjona, and G. Furano, "High-performance embedded computing in space: Evaluation of platforms for vision-based navigation," *Journal of Aerospace Information Systems*, vol. 15, no. 4, pp. 178–192, 2018.

[5] L. Kosmidis, I. Rodriguez, Á. Jover, S. Alcaide, J. Lachaize, J. Abella, O. Notebaert, F. J. Cazorla, and D. Steenari, "GPU4S: Embedded GPUs in space-Latest project updates," *Microprocessors and Microsystems*, vol. 77, p. 103143, 2020.

[6] J. A. Belloch, J. M. Badía, F. D. Igual, A. Gonzalez, and E. S. Quintana-Ortí, "Optimized fundamental signal processing operations for energy minimization on heterogeneous mobile devices," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 65, no. 5, pp. 1614–1627, 2017.

[7] L. B. Gomez, F. Cappello, L. Carro, N. DeBardeleben, B. Fang, S. Gurumurthi, K. Pattabiraman, P. Rech, and M. S. Reorda, "GPGPUs: How to combine high computational power with high reliability," in *2014 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2014, pp. 1733–1741.

[8] B. Nie, D. Tiwari, S. Gupta, E. Smirni, and J. H. Rogers, "A large-scale study of soft-errors on gpus in the field," in *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2016, pp. 519–530.

[9] 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)*. IEEE, 2020, pp. 639–641.

[10] F. Kastensmidt and P. Rech, "Radiation effects and fault tolerance techniques for FPGAs and GPUs," in *FPGAs and parallel architectures for aerospace applications*. Springer, 2016, pp. 3–17.

[11] D. A. G. G. 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, 2015.

[12] A. Lotfi, S. Hukerikar, K. Balasubramanian, P. Racunas, N. Saxena, R. Bramley, and Y. Huang, "Resiliency of automotive object detection networks on gpu architectures," in *2019 IEEE International Test Conference (ITC)*, 2019, pp. 191–199.

[13] K. Ito, Y. Zhang, H. Itsuji, T. Uezono, T. Toba, and M. Hashimoto, "Analyzing due errors on gpus with neutron irradiation test and fault injection to control flow," *IEEE Transactions on Nuclear Science*, vol. 68, no. 8, pp. 1668–1674, 2021.

[14] D. A. Oliveira, P. Rech, L. L. Pilla, P. O. Navaux, and L. Carro, "GPGPUs ECC efficiency and efficacy," in *2014 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*. IEEE, 2014, pp. 209–215.

[15] D. A. Oliveira, P. Rech, H. M. Quinn, T. D. Fairbanks, L. Monroe, S. E. Michalak, C. Anderson-Cook, P. O. 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, 2014.

[16] M. Niazi-Razavi, A. Savadi, and H. Noori, "Toward real-time object detection on heterogeneous embedded systems," in *2019 9th International Conference on Computer and Knowledge Engineering (ICCKE)*, 2019, pp. 450–454.

[17] P. Rech, L. L. Pilla, P. O. A. 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*. IEEE, 2014, pp. 455–466.

[18] M. V. O'Bryan, K. A. LaBel, E. P. Wilcox, D. Chen, M. J. Campola, M. C. Casey, J.-M. Lauenstein, E. J. Wyrwas, S. M. Guertin, J. A. Pellish *et al.*, "Compendium of current single event effects results from nasa goddard space flight center and nasa electronic parts and packaging program," in *2017 IEEE Radiation Effects Data Workshop (REDW)*. IEEE, 2017, pp. 37–47.

[19] E. J. Wyrwas, "Proton testing of nvidia jetson tx1," NASA Goddard Space Flight Center, Tech. Rep., october 2016.

[20] ——, "Proton testing of nvidia jetson tx2," NASA Goddard Space Flight Center, Tech. Rep., July 2019.

[21] E. Wyrwas, K. A. LaBel, M. Campola, and M. O'Bryan, "Guidance on standardizing gpu test approaches," in *Nuclear and Space Radiation Effects Conference (NSREC)*, 2018, pp. 116–119.

[22] 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)*. IEEE, 2017, pp. 127–130.

[23] D. 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)*. IEEE, 2020, pp. 31–34.

[24] S. M. Guertin and M. Cui, "See test results for the snapdragon 820," in *2017 IEEE Radiation Effects Data Workshop (REDW)*. IEEE, 2017, pp. 155–160.

[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*. IEEE, 2019, pp. 157–161.

[26] S. Guertin, "Radiation Effects on ARM Devices," June 2014.

[27] F. F. dos 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, 2018.

[28] 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)*, 2018, pp. 13–26.

[29] NVIDIA, *CUDA C++ Programming Guide. PG-02829-001_v11.3. Design Guide*, April 2021.

[30] M. Scarpino, *OpenCL in action: how to accelerate graphics and computations*. Simon and Schuster, 2011.

[31] M. J. Flynn, "Some computer organizations and their effectiveness," *IEEE Transactions on Computers*, vol. C-21, no. 9, pp. 948–960, 1972.

[32] NVIDIA, "NVIDIA Tegra K1. A new era in mobile computing. NVIDIA Whitepaper," January 2014.

[33] ——, "NVIDIA Jetson TK1 Development Kit. Bringing GPU-accelerated computing to Embedded Systems. Technical Brief," January April 2014.

[34] ——, *NVIDIA's Next Generation CUDA Compute Architecture: Kepler G110/2010. Whitepaper.*, 2014.

[35] ——, *cuBLAS Library. User Guide*, December 2020.

[36] L. S. Blackford, A. Petitet, R. Pozo, K. Remington, R. C. Whaley, J. Demmel, J. Dongarra, I. Duff, S. Hammarling, G. Henry *et al.*, "An updated set of basic linear algebra subprograms (BLAS)," *ACM Transactions on Mathematical Software*, vol. 28, no. 2, pp. 135–151, 2002.

[37] NVIDIA, *Profiler. User's Guide. DU-05982-001_v11.3*, April 2021.

[38] G. Li, K. Pattabiraman, C.-Y. Cher, and P. Bose, "Understanding error propagation in GPGPU applications," in *SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2016, pp. 240–251.

[39] S. K. S. Hari, T. Tsai, M. Stephenson, S. W. Keckler, and J. Emer, "SASSIFI: An architecture-level fault injection tool for GPU application resilience evaluation," in *2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 2017, pp. 249–258.

[40] G. León, J. M. Badia, J. A. Belloch, A. Lindoso, and L. Entrena, "Evaluating the soft error sensitivity of a GPU-based SoC for matrix multiplication," *Microelectronics Reliability*, vol. 114, p. 113856, 2020.