The Journal of Supercomputing
https://doi.org/10.1007/s11227-024-05925-0

Comparative analysis of soft-error sensitivity in LU decomposition algorithms on diverse GPUs

German Leon[1] · Jose M. Badia[1] · Jose A. Belloch[2] · Almudena Lindoso[2] · Luis Entrena[2]

Accepted: 21 January 2024
© The Author(s) 2024

Abstract

Graphics processing units (GPUs) have become integral to embedded systems and supercomputing centres due to their large memory, cutting-edge technology and high performance per watt. However, their susceptibility to transient errors requires a comprehensive analysis of error sensitivity, as well as the development of error mitigation techniques and fault-tolerant algorithms. This study focuses on evaluating the soft-error sensitivity of two distinct versions of LU decomposition algorithms implemented on two very different GPUs—a low-power SoC embedded GPU and a high-performance massively parallel GPU. Through extensive fault injection campaigns on both GPUs, we examine the vulnerability of the algorithms, identify error causes, and determine critical code components requiring enhanced protection. The experiments reveal that most single bit flip fault injections in the instruction results lead to erroneous outcomes or unrecoverable errors. Notably, efficient GPU resource utilisation can increase the number of masked errors, thereby enhancing error resilience. Additionally, while different parts of the code exhibit similar error occurrence types and rates, the propagation of errors to elements within the result matrix differs significantly.

✉ Jose M. Badia
badia@uji.es

German Leon
leon@uji.es

Jose A. Belloch
jbelloc@ing.uc3m.es

Almudena Lindoso
lindoso@ing.uc3m.es

Luis Entrena
entrena@ing.uc3m.es

[1] Depto. de Ingeniería y Ciencia de Computadores, Universitat Jaume I de Castelló, Avda. Sos Baynat, s/n, 12071 Castellón, Spain

[2] Depto. de Tecnología Electrónica, Universidad Carlos III de Madrid, Avda Universidad, 30, 28911 Leganés, Madrid, Spain

Published online: 22 February 2024

Keywords GPU · Soft errors · Sensitivity · Fault injection · LU decomposition

1 Introduction

Due to their massive parallelism and high performance per watt, GPUs have become a staple for accelerating computation [1]. These devices can be found in a wide range of platforms, from low-power system-on-chip (SoC) to the nodes of the largest supercomputers (http://top500.org). Typically, when developing algorithms for such devices, the goal is to optimise computational performance and power consumption. However, in certain environments it is also essential to reduce the number of soft errors and increase reliability [2, 3].

GPUs entered the embedded domain to meet the growing demand for multimedia-based handheld and consumer devices such as smartphones [4]. More recently, mobile GPUs are increasingly being used to accelerate heavy workloads, for applications ranging from signal processing [5], to advanced driving assistance systems (ADAS) in cars [6], or to accelerate the computational requirements of deep neural networks [7]. SoCs with low-power GPUs can meet the requirements of many of these applications due to their high performance per watt and the programming flexibility of their various parallel components. For example, commercial off-the-shelf (COTS) platforms incorporating such devices are candidates to replace traditional radiation-hardened systems in small satellites [8]. However, fault tolerance is a key consideration in all these applications and the reliability of embedded GPUs needs to be assessed and improved. For example, stringent risk-based safety standards such as ISO 26262 must be considered in the design and construction of electrical and electronic systems in production vehicles, including driver assistance, propulsion and vehicle dynamics control systems [9].

On the other hand, the high computational power of GPUs is driving the scientific discovery process on a large scale. Applications in fields as diverse as climate simulation [10], electromagnetic fields [11], and molecular dynamics [12] have efficiently exploited GPU parallelism. This type of device uses last-generation CMOS or FinFET technology and contains thousands of cores with different computing elements and very large and fast memories. Most recent GPUs also include special units, such as tensor cores, to accelerate key steps in machine learning applications [13]. High-performance computing (HPC) workloads are typically long-running simulations that use various techniques, such as checkpointing and restarting, to complete in the event of failures. Most supercomputer sites contain many thousands of high-performance GPUs, which increases the probability of failure due to both hardware and soft errors [14]. Failures in HPC nodes can be caused by temperature, voltage variations, environmental disturbances, firmware errors or manufacturing processes, among other things. However, one of the most common causes is radiation, which consists of particles such as neutrons, electrons, protons and heavy ions, as well as electromagnetic radiation. Experiments in supercomputer centres with more than ten thousand GPUs have detected failures every few tens of hours, and the number of nodes is constantly increasing [3, 15]. Therefore, understanding the

characteristics of GPU-related faults in large-scale systems is likely to benefit system operators, designers and end users.

The use of GPU resources by applications determines their reliability. Obviously, if an application uses more resources or uses them more intensively, it is more likely to be affected by an incoming particle. However, it is also true that more efficient use of resources reduces the execution time of applications and thus the likelihood that they will be affected by radiation. Furthermore, not all components of a GPU are equally susceptible to the effects of radiation, nor do all defects in the architecture lead to errors. Also, not all faults are equally important or affect the same number of tasks running in parallel. For example, a particle may cause a bit flip on the result of an instruction executed by only one thread, but it may also modify a memory cell read by all threads executing the application. As can be seen, modelling the fault tolerance of applications on GPUs is complex and must take into account a large number of factors, some of which have opposing effects [16, 17]. Over the last decade, numerous works have been carried out to analyse all of the above factors. Most of them use fault injection or device radiation as the analysis mechanism.

To achieve high performance on GPUs, we need to increase the utilisation of their many cores, but also use their fastest memories, including shared memory and register file. However, increasing the workload and use of GPU resources can also increase the soft-error sensitivity of the algorithms [2]. As a first approach, in [18] we examined the soft-error sensitivity of two different implementations of matrix multiplication on a GPU-accelerated system-on-chip. Both implementations started only one kernel and made different use of the resources of the GPU, in particular the different memories of the device. One of the algorithms is memory-bound, while the other uses a block strategy that exploits the fast shared memory of the GPU to achieve better performance. The results showed that soft-error sensitivity depends on the implementation of the algorithm and how it uses the resources of the GPU. In particular, the block algorithm is not only much more efficient, but also masks many more errors than the memory-bound implementation. However, the use of shared memory in the block algorithm increases the number of unrecoverable crashes.

We evaluate in this work a different application that allows us to extend and deepen the evaluation of the influence of the parallelisation strategy on the soft error sensitivity of algorithms on GPUs. Specifically, we compare the soft error sensitivity of two different strategies to perform LU decomposition. First of all, it should be noted that we do not intend to use the most efficient version of the LU on GPUs, but to compare two versions with very different performance and use of GPU resources. Thus, error injection could cause a clearly differentiated behaviour of both versions and allow us to relate the errors detected to the use of resources such as global memory or shared memory. Therefore, we first chose the version of the decomposition most widely used in the literature related to fault tolerance of GPUs. This is the `lud` version included in the Rodinia test suite [19]. This is a relatively efficient block version that uses three different kernels with different degrees of parallelism and takes advantage of the shared memory of the GPU. Second, we modified one of the kernels of the block algorithm to implement a memory-bound version of the decomposition, which is much less efficient, uses only one kernel and with a much lower degree of parallelism. This version relies only on global memory, which has

consequences on its performance, the type of errors that can arise from the injection and the propagation of errors. On each iteration of the block version the algorithm launches successively three kernels. This allows us to analyse the soft error sensitivity of different steps of the algorithm involving different matrix panels. To this end, we inject faults on the different kernels and analyse their error rates and also the error propagation pattern on each of them. Matrix decompositions, such as LU, are fundamental in many applications. Studying their sensitivity to errors on GPUs can help us to design better fault-tolerant algorithms on this kind of accelerator.

We use two fault injection tools to inject single bit flips into the results of a randomly chosen instruction of the algorithms on two very different GPUs: a low-power system-on-chip embedded GPU that can be used in safety-critical environments, and a much faster and power-hungry GPU that can be included in the nodes of high-performance supercomputers. Experimental results show that most fault injections cause Silent Data Corruptions (SDC) or Detected Unrecoverable Errors (DUE). The use of shared memory increases the performance of the block algorithm on both GPUs, but also the number of DUEs caused by illegal memory accesses. As we show in Sect. 5.2, the block algorithm has a larger number of DUE and most of them are due to illegal accesses to the shared memory of the GPU. Finally, the experiments also show that different sections of the code have different fault injection sensitivities and error propagation rates and patterns.

The main contributions of this paper are as follows:

- Compare the soft error sensitivity of two different algorithms for performing LU decomposition on GPUs.
- Analyse the causes of DUEs as a function of algorithm and GPU device.
- Study the error propagation as a function of the code section affected by the SDCs.
- Study the soft error sensitivity of algorithms on two GPU devices that are very different in terms of architecture, computing power and application scope.

The rest of the paper is structured as follows. Section 2 summarises the related work. Section 3 briefly describes the two LU decomposition algorithms analysed on this paper. Section 4 describes the experimental environment and Sect. 5 analyses the experimental results. Finally, Sect. 6 summarises the main conclusions of the paper.

2 Related work

The study of the reliability of heterogeneous embedded systems is gaining momentum because they combine high parallelism, low power consumption, the flexibility offered by their diverse components, and the possibility of using low-cost and lightweight COTS devices [20, 21]. Today, most of these systems can be found in mobile devices, where various reliability assessments are currently being carried out [22].

GPUs are a common computing resource found on such systems. Their reliability has been widely analysed in the literature over the last decade [3, 17, 23]. Many different benchmarks have been used to analyse and improve the reliability of such

devices. These include basic computational kernels [24], including matrix multiplication [2], FFT [25], or microbenchmarks [17] to evaluate their various components, or benchmark suites such as Rodinia [19]. More recently, several papers have analysed the reliability of GPUs using convolutional neural networks (CNNs). For example, in [13], the authors evaluate and propose strategies to improve the object detection reliability of three CNN algorithms using an embedded GPU and two high-performance GPUs. In [26], the authors analyse the impact of faults affecting the hidden modules of GPUs on an entire CNN execution (LeNET) without undermining the correctness of the reliability evaluation by combining simulation and software fault injection. The impact of tensor cores and mixed precision on the reliability of matrix multiplication in a high-performance GPU is studied in [27]. Effective microarchitectural selective hardening of GPU modules to mitigate errors that affect the correct execution of instructions is proposed in [28].

Matrix factorisations such as Cholesky, LU, and QR decompositions have also been used to analyse the reliability of GPUs as they are basic operations in many applications [29–32]. The soft error sensitivity of this type of factorisation has been analysed in several papers, and various algorithm-based fault tolerance (ABFT) techniques have been proposed to improve their reliability [33–35]. Much effort has been devoted to designing ABFT versions of one-sided matrix decompositions, including LU. This design philosophy was introduced in [36], where the author applied it to matrix multiplication and LU decomposition. It is based on modifying the algorithm by adding some elements and computations to detect and in some cases correct the errors. This first approach tolerated only single errors and applied an off-line correction scheme, i.e. it detected and corrected errors at the end of the computation, after the errors had propagated and accumulated. In contrast, online schemes detect, locate, and correct errors in the middle of the computation, avoiding their propagation and reducing the overhead. For example, [37] introduced an online method for LU decomposition, which was also applied to Cholesky and QR decompositions on distributed memory multicomputers in [38], with an overhead fluctuating around 5%. Most previous methods only maintain checksums in one dimension, protecting only a part of the matrix. In [35], the authors propose a full checksum method. They also analyse the error propagation patterns of the three kernels used to perform the decomposition and propose a checksum scheme adapted to the different error sensitivities of each kernel.

On the other hand, various fault injection tools have been used to study the behaviour of the LU decomposition implementation included in the Rodinia suite. In [39], the authors present a fault injection tool that can inject at two levels of abstraction, namely the intermediate representation level (ptx) and the assembler level (sass). The tool can inject into different hardware components of the GPU, such as the register file, shared memory, Single Instruction Multiple Threads (SIMT) stack, or instruction buffer. Experiments on a Fermi GPU show that the error rate depends on the hardware component targeted. For example, register file injections cause errors in less than 5% of tests, while shared memory injections cause errors in more than 75% of cases. The authors also show that the three kernels used to perform LU decomposition have different architectural vulnerability factors (AVFs). Another injection tool (SASSIFI) is introduced in [40] and applied to LU decomposition, among many other Rodinia

benchmarks. The results show that injecting single bit flip errors into the target registers of Kepler GPU instructions causes SDCs in about 40% of the tests, while the injection is masked in the rest of the tests. Finally, in [41] the authors propose a fault injection method that reduces the number of injections based on the identification of frequently executed instructions. They use SASSIFI to perform fault injections in all Rodinia benchmarks, including LU decomposition, causing SDCs or DUE in more than 60% of the tests.

Contrary to previous work, [39–41], where only the version of the LU included in the Rodinia benchmark suite [19] was injected, one of our main objectives was to analyse the influence of the use of different decomposition strategies on its soft-error sensitivity. In this sense, we compare two different versions of the LU in terms of their use of GPU resources. We have also included, as one of the two devices under test, a low-power GPU embedded in a SoC platform. The evaluation of the reliability of this type of device is of particular interest, as they are candidates for inclusion in safety-critical environments such as space and advanced driver assistance systems [42, 43]. In [44], we evaluated the reliability of the two LU decomposition algorithms described in the next section under proton irradiation on one of the Devices Under Test used in this paper, a Jetson TK1 SoC. Results show that a more intensive use of the resources of the GPU increases the cross section and that most of the radiation-induced errors could only be solved by rebooting the platform.

3 LU decomposition algorithms

In this paper, we evaluate the error sensitivity of two algorithms that implement LU decomposition. We have used the Compute Unified Device Architecture (CUDA) programming model to implement these algorithms [45]. The code executed in the cores of the GPU is written as functions called kernels. Each kernel is executed in parallel by multiple elementary processes called threads. The threads are logically grouped into thread blocks, which are assigned to a Streaming Multiprocessor (SM) of the GPU device and share memory. Thread blocks are then organised in a grid. Efficient algorithms try to take advantage of all the architecture's components, including its multiple cores and fastest memories.

The first LU decomposition algorithm, called `block`, is the block-parallel version implemented with CUDA and included in the Rodinia benchmark suite.

Figure 1 shows the main panels of the first iteration of the LU decomposition included in Rodinia. The algorithm starts from the upper left corner and performs the following three steps:

$$A_{11} \leftarrow L_{11} U_{11} \tag{1}$$

$$L_{21} \leftarrow A_{21} U_{11}^{-1} \qquad U_{12} \leftarrow L_{11}^{-1} A_{12} \tag{2}$$

$$A'_{22} \leftarrow A_{22} - L_{21} U_{12} \tag{3}$$

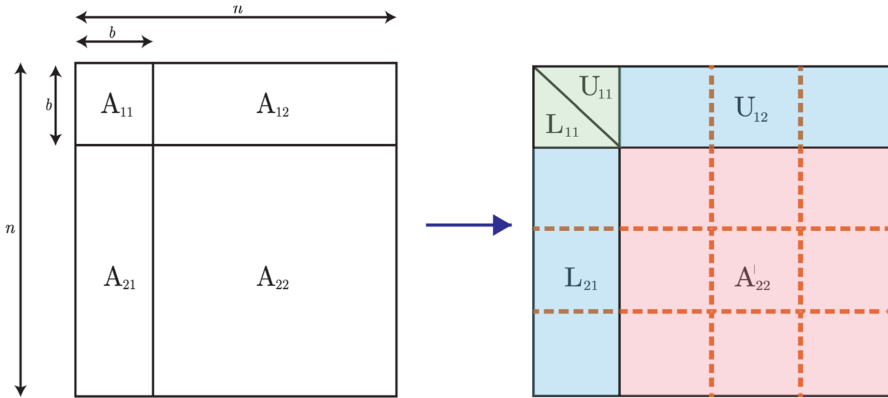

Fig. 1 Main panels and updates during the first iteration of the block LU decomposition

The decomposition is then continued by recursively applying the same steps to the A'_{22} panel. The three steps are implemented as three CUDA kernel functions [45] called `diagonal`, `perimeter` and `internal`, respectively. Figure 1 shows the panels affected by each kernel during the first iterations on a different colour. The last two kernels divide the panels A_{12}, A_{21} and A_{22} into square blocks of size $b \times b$ (red lines in Fig. 1) and copy and update them in parallel, using the shared memory of the GPU to increase the performance of the algorithm.

The kernel `diagonal` uses a simple method where a thread block of `b` threads performs the LU decomposition of the upper left panel A_{11}. Each thread is responsible for computing one row of U_{11} and one column of L_{11} in shared memory. The second parallel algorithm we tested, called `rc`, uses the same method but applied to the whole matrix. In addition, all computations are performed in the global memory of the GPU, and each thread can compute more than one row and column, depending on the size of the matrix (see Fig. 2). The `rc` algorithm, which is obviously slower than the block version, allows us to analyse the effects of error injection using a method that is much less efficient in terms of GPU resources.

Both algorithms make very different use of not only memory but also other components of the GPU. The `block` algorithm iterates the execution of three kernels with different grid and thread block sizes. For example, suppose Fig. 1 represents a matrix of size 64×64, where each small square block is of size 16×16, then during the first iteration of the algorithm, kernel `diagonal` will start a block of threads of size 16; kernel `perimeter` will start 3 blocks of threads of size 32; and kernel `internal` will start 3×3 blocks of threads of size 16×16. The `rc` algorithm uses a grid of size 1 and starts only once a block of threads of size 64. Therefore, the number of kernels launched when running the two algorithms is quite different, as is the occupancy of the cores and the use of the GPU's block and warp schedulers.

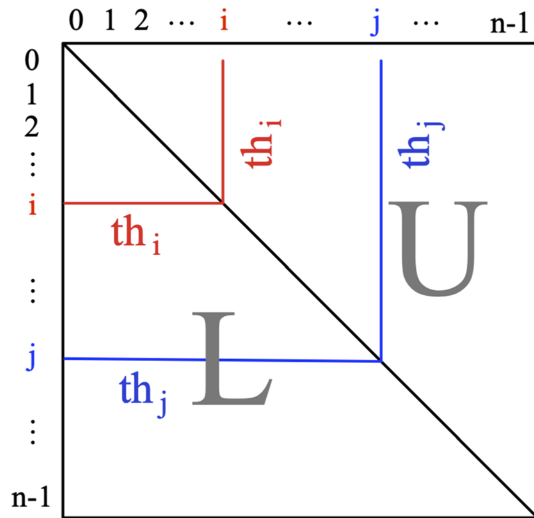

Fig. 2 In the `rc` version of the LU decomposition each thread (e.g. th_i and th_j) computes one or more rows of matrix L and the corresponding columns of matrix U

4 GPU-based experimental environment

4.1 GPU devices

We used two very different GPU devices to perform our experiments, the main features of which can be seen in Table 1. The first experimental platform is a low-power Tegra K1 (TK1) system-on-chip (SoC) embedded in the Jetson developer kit [46]. This particular system consists of a quad-core ARM Cortex A15 processor (or CPU), a battery-saving ARM Cortex A15 shadow core, and an NVIDIA "Kepler" K20A GPU with 1 SM containing 192 CUDA cores. The SM has four warp schedulers and eight instruction dispatchers. It can then select four warps of 32 threads and issue two independent instructions per warp per cycle. The TK1 SoC is designed to increase performance per watt. It typically consumes between 0.6W and 3W of power during normal use, and a maximum of 15W when the CPU, GPU, and codec hardware are pushed to their limits.

Table 1 Main features of the GPUs under test

Feature	K20A (Kepler)	GV100
Architecture	Kepler	Volta
Compute capability	3.2	7.0
Number of SMs	1	80
Total number of cores	192	5120
Global memory	2 GB (DDR3)	32 GB (HBM2)
Shared memory	48 KB	7680 KB
L2 cache size	128 KB	6144 KB
Register file size	256 KB	20480 KB

We also used a high-performance Tesla V100 with the Volta architecture consisting of 80 SM [47]. Each SM is divided into four processing blocks, each with 16 FP32 cores, 8 FP64 cores, 16 INT32 cores, and two new mixed-precision tensor cores for deep learning matrix arithmetic. Each block also includes a warp scheduler and two dispatch units. This GPU is connected to a server with 2 Intel Xeon Gold 6126 2.6 GHz processors, each with 12 cores and a total of 64 GB of memory.

The two GPUs represent very different types of devices with different targets. While the K20A GPU is a low-power device that can be used in embedded systems to reduce power consumption, the Tesla V100 is a high-performance GPU that provides massive data parallelism with thousands of cores, suitable for use in HPC nodes. Both GPUs are built on very different technologies. While the K20A uses standard 28nm CMOS transistors, the Tesla V100 is built on new 12nm FFN technology, NVIDIA's proprietary FinFET manufacturing process. Based on physical simulations and test-chip experiments, FinFET transistors have been reported to have a much lower neutron-induced error rate compared to CMOS [48]. Matrix multiplication and neural networks used to compare both technologies under neutron irradiation in [13] show an order of magnitude lower error rate of 16 nm FinFET compared to 28 nm CMOS.

Another important difference between the two GPUs is the automatic error protection of the memory. Unlike other Kepler GPUs, the K20A does not include error correction code (ECC) in any of its memories. However, the Tesla V100 global memory subsystem supports Single-Error Correction Double-Error Detecting (SECDED). In addition, other key structures such as the register file and L1 and L2 caches are protected by the same mechanism. As with the technology, this mechanism could detect radiation-induced errors. However, faults injected into the results of instructions are not detected by any ECC mechanism.

4.2 Fault injectors

We did not use the same fault injection tool on both experimental platforms because they do not support compatible versions of the software they require (i.e. CUDA toolkit, CUDA GDB, LLVM compilation framework, etc.). Instead, we used two injection tools that allow us to perform the same type of fault injection and result analysis.

In the Jetson board, we used LLFI-GPU, which is an extension of the open source LLFI fault injection tool [49]. It first profiles the programme to get the number of kernels, threads, and instructions. Next, it uses the LLVM compiler framework to instrument the IR (intermediate representation) of the CUDA code. It can then inject single bit flips into the registers that store the results of a random instruction in a random thread of one of the kernels. It uses the profiling information to inject the error with a uniform probability in all instructions executed. You can annotate the code to choose specific kernels to inject the fault. Its main limitation is that it does not inject errors into other components of the architecture, such as the GPU memories, the condition codes, or the memory addresses and their values.

In the Volta GPU, we used CAROL-FI, a fault injector based on the GNU GDB debugger, which was introduced in [50] to analyse the behaviour of Xeon Phi

processors. The original version of the tool injects faults at any source code variable allocated to a memory position. It works in two main phases. It first uses the debugger to launch the code and interrupt its execution after a random time. Then, it selects one of the available threads and active subroutines and flips one or more bits of one of its variables. It allows several fault models, including single and double bit flips and overwriting every bit with zeros or random values. We have modified the version based on CUDA GDB [51] to perform a fault injection similar to the one performed by the LLFI-GPU injector. Specifically, we have modified one of the injection modes to perform a single bit flip in the target register values of the instructions. We have also modified the injector to be able to select the instruction to inject, and thus the kernel affected by the injection. One of the main limitations of CAROL-FI is that it is based on the CUDA debugger and so the injection process is slower than with tools like LLFI-GPU.

4.3 Experimental methodology

We have performed injection campaigns using both injectors, always consisting of 1000 tests for each algorithm and problem size. We always injected one error in each LU decomposition. The following error categories were used in all experiments:

- `Masked`: we get a correct result of the LU decomposition. The results are compared with a previously computed golden version.
- `Silent Data Corruption (SDC)`: one or more elements of the result do not match the golden output.
- `Detected Unrecoverable Error (DUE)`: an error occurs because the programme has tried to perform an invalid action (e.g. read outside its memory segment). This error can be captured by a debugger, the process can be killed, and the operating system can start the next test (e.g. LU decomposition).

Our experimental setup includes two timeouts. The first timeout, associated with the process performing the decomposition, is set to a time slightly greater than the maximum predictable duration of the operation. We use this timeout on the CUDA debugger to detect the DUE errors of our test [52]. In addition, we have used the watchdog Linux API to implement a hardware watchdog that reboots the platform if it hangs for more than 30 s. This is not considered a DUE error as it is not application related but operating system related and is a very rare type of error.

5 Experimental results and evaluation

5.1 Performance of the algorithms

To compare the performance of the two LU decomposition algorithms, we measured their execution times on the Jetson TK1 board with different problem sizes. Figure 3 shows the results for matrices ranging from 128 to 2048. We can see that

the `block` algorithm is always much faster than the `rc` algorithm. Specifically, it is 8 times faster for matrices of size 1024 and 10 times faster for matrices of size 2048. This behaviour is mainly due to two factors. First, the `block` algorithm uses the fast shared memory of the GPU and reduces accesses to the slower global memory, increasing the ratio of floating point operations per memory access. Second, the `rc` algorithm uses the GPU cores less efficiently.

Using the CUDA profiler [53], we have seen that this algorithm only achieves a 50% occupancy of the GPU. On the contrary, the `internal` kernel, which is the most used on the `block` algorithm, achieves a 91.96% occupancy. In addition, the `rc` algorithm executes only 0.12 instructions per cycle (IPC), which is far from the optimal IPC that can be executed using the eight instruction dispatchers of the K20A GPU. In contrast, the `internal` kernel executes 1.23 instructions per cycle, which is more than ten times better than the `rc` algorithm. There are more efficient implementations of LU decomposition on GPUs, such as the one included in the MAGMA library [54]. However, our goal was not to test the fastest implementation available, but to compare the soft-error sensitivity of two implementations with very different uses of GPU resources.

5.2 Soft error sensitivity of the algorithms

We ran similar fault injection campaigns on both platforms. The results for the error categories shown in this section do not depend on the size of the matrices or the block size used in the block algorithm.

Both algorithms have a different sensitivity to soft errors, as can be seen in Fig. 4. Most of the error injections result in SDCs or DUEs of the algorithms, but the percentages are highly dependent on the algorithm and the platform. For example, the `block` algorithm masks more errors than the `rc` algorithm on the

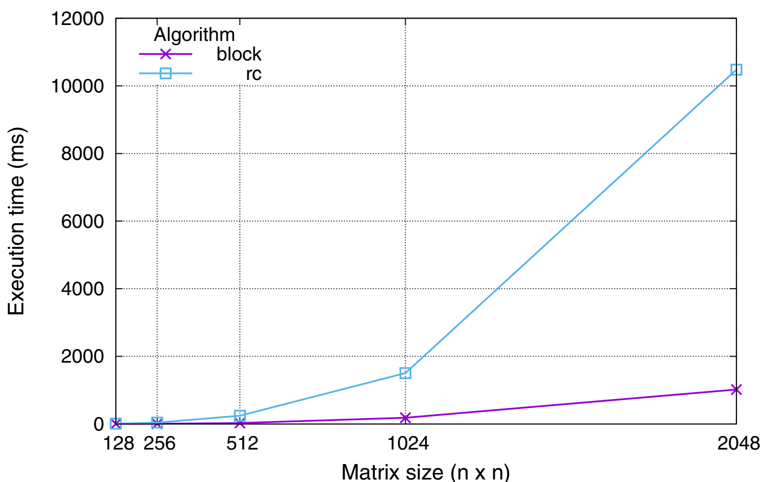

Fig. 3 Execution times of both algorithms on a Jetson TK1

Kepler GPU, while the opposite is true on the Volta GPU. The SDC percentages are higher for `rc` than for `block` on both platforms. The different behaviour of the two algorithms on the two GPUs could be due to two main reasons. First, it may be due to their different architectures and how the faults injected into the registers with the results of the instructions affect the behaviour of the code and the results it produces in the LU decomposition. Secondly, the different behaviour may be due to the fact that the two injectors used may not select the kernels, threads and instructions into which the faults are injected in the same way. The probabilities with which different result registers and instructions are affected on the two GPUs may be different due to the details of how the two injection tools work. This behaviour of the tools and their effect on the errors detected in the code is independent of the architecture of the GPUs. In addition, the thread block and shared memory size is defined by the size of the matrix in the `rc` algorithm and by the size of the blocks of elements in the `block` algorithm. We have performed the experiments using matrices and blocks of elements with the same size in both architectures. Therefore, the different behaviour is not due to differences in those sizes.

Nevertheless, we tried to get more information about the causes of the DUEs by using the cuda-gdb debugger to run the fault injection tests. This allows us to catch the various exceptions that trigger most of the DUEs. Figure 5 shows the percentage of DUEs caused by different types of errors. In particular, only 3 of the 15 types of exceptions defined in [52] occurred on each platform (4 types in total), all of them related to memory access problems. Specifically, cuda-gdb caught the following exceptions:

- `Ex5`: Occurs when any thread within a warp accesses an address outside its valid range of local or shared memory regions.
- `Ex6`: Occurs when any thread within a warp accesses an address in the local or shared memory regions that is not correctly aligned.
- `Ex10`: Occurs when a thread accesses an illegal (out of bounds) global address. Only on the Jetson platform.

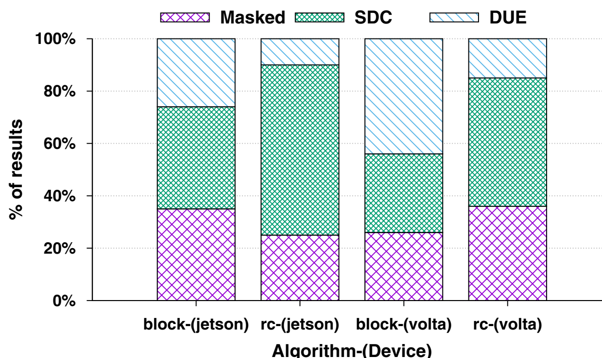

Fig. 4 Fault injection results of the LU decomposition algorithms in both GPUs

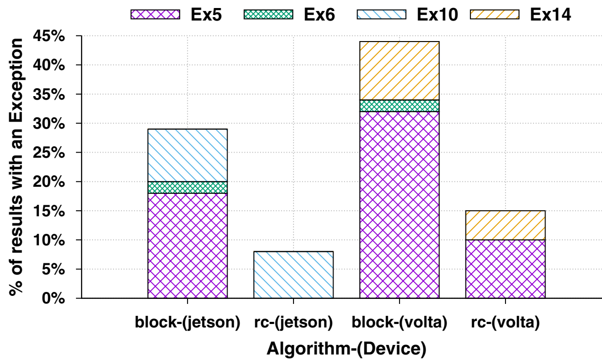

Fig. 5 Causes of the DUEs in the injected LU algorithms in both GPUs. Percentage over the all the injected tests

- `Ex14`: Occurs when a thread accesses an illegal (out of bounds) global/local/shared address. Only on the Volta platform.

Figure 5 shows, on the one hand, that most errors of the `block` algorithm are caused by illegal or misaligned accesses to shared memory (`Ex5` and `Ex6`). This type of memory improves the performance of the algorithm, but greatly increases the total number of DUEs. Shared memory is much smaller than global memory, and is also distributed across the different blocks of threads. Therefore, any small change in a shared memory address can easily produce an address outside the valid address range for a thread, resulting in a `Ex5` and eventually a DUE. In contrast, much larger global memory can be accessed by all threads, and it is more difficult for address changes to result in illegal global addresses. On the other hand, the `rc` algorithm only accesses global memory, which explains why the only exceptions caught on the Jetson platform are related to illegal accesses to this type of memory (`Ex10`). This type of illegal access produces a `Ex14` exception in the Volta platform. The `Ex5` in the case of the `rc` algorithm detected in the Volta platform can only be due to illegal accesses to local memory, as this algorithm does not use shared memory. This algorithm does not produce this kind of exception in the Jetson platform.

A very interesting aspect to analyse is the behaviour of the different kernels used by the `block` algorithm. This allows us, for example, to tailor the mitigation techniques used at each step of the code. Figure 6 shows the relative distribution of errors in each of the three kernels, and that they have very similar sensitivities to error injection. Injection of bit flips in the instructions of the three basic parts of the code has almost the same effect on the result of LU decomposition. About 36% of the fault injections are masked, 37% cause SDCs and 27% cause a DUE. Moreover, this behaviour does not depend on the size of the matrix.

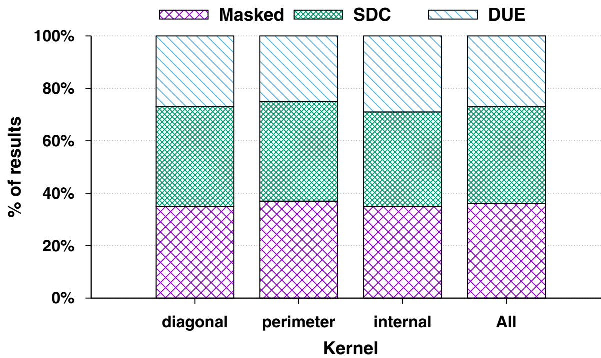

Fig. 6 Fault injection results of the CUDA kernels of the block algorithm in the Jetson board

5.3 Error propagation

One of the main problems of the SDC soft errors introduced in any matrix element during the computation of the LU decomposition is that they can propagate to a large number of elements of the matrix [33, 35]. The propagation rate depends on the algorithm and, more specifically, on the kernel affected by the error injection. If the bit flip changes the value of one element of the result matrix, this error can propagate to other elements that are computed with the wrong result. The number of elements affected depends on the position of the element initially modified by the fault injection and how it is used to compute the remaining elements of the LU decomposition.

Fig. 7 show that errors introduced in both algorithms propagate to different percentages of elements of the result. Errors in the `block` algorithm propagate on average to 17% of the elements, while errors in the `rc` algorithm propagate to more than 28% of the elements of the result.

In fact, the propagation rate of the algorithm depends on the part of the code where the error is injected. As can be seen in Fig. 8, errors introduced in the three kernels of the `block` algorithm propagate on average to quite different numbers of elements of the result.

These results are consistent with those obtained in [35]. Although the authors analysed the error propagation pattern of a slightly different block algorithm, their conclusions can be applied to the LU implementation included in the Rodinia suite. Errors in kernel `internal` can only propagate to one row or column of the matrix panel modified by the kernel, while errors in kernels `diagonal` and `perimeter` can propagate beyond one row or column. Figure 9 shows the results obtained after completing the first iteration of the block LU decomposition when we inject an error into each of the three kernels. These are representative examples of how the errors usually propagate to blocks of rows and columns in the case of the `diagonal` and `perimeter` kernels, while in most cases they only affect one element in the case of the `internal` kernel. For each of the three kernels, the figure only shows the propagation of the errors in the elements of the panels modified by the corresponding

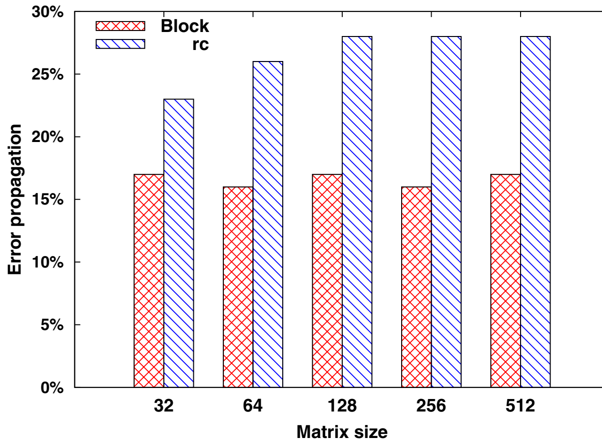

Fig. 7 Average percentage of wrong elements in the result when a SDC occurs on each algorithm. Results in the Jetson platform

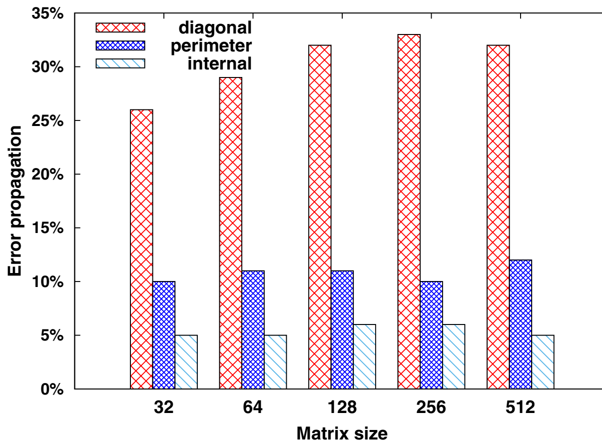

Fig. 8 Average percentage of wrong elements in the result of the algorithm `block` when a SDC occurs depending on the kernel where the fault is injected

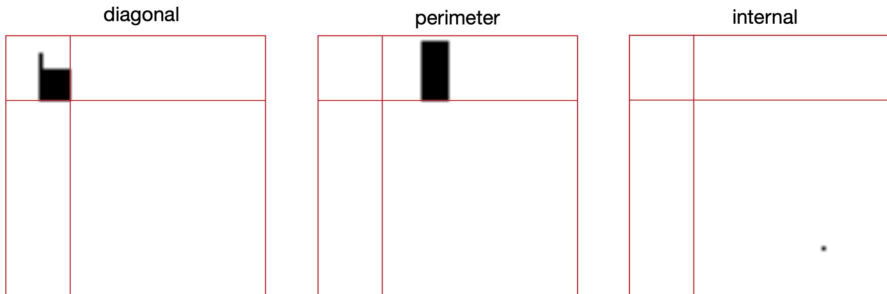

Fig. 9 Example of error propagation in each of the three kernels of the algorithm `block`

kernel. Wrong elements computed by the `diagonal` or `perimeter` kernels will propagate to the elements of the rest of the matrix computed from them. This behaviour justifies that errors produced in the last kernel propagate to a much smaller percentage of elements than errors produced in the `diagonal` kernels (see Fig. 8).

We have seen that the errors produced in the `diagonal` and `perimeter` kernels propagate to a larger number of elements of the result matrix. Therefore, detection and correction of these errors would reduce the total number of elements affected. Therefore, it is critical to apply mitigation techniques to these two kernels if we want to reduce error propagation.

For example, experiments show that the `diagonal` and `perimeter` kernels are executed with occupancy of both GPUs well below 30%. This fact allows us to apply mitigation techniques based on duplicating or even tripling the execution of these kernels without significantly increasing the execution time of these steps of the algorithm.

6 Conclusions

Fault injection campaigns on two different versions of LU decomposition show that the soft-error sensitivity of the algorithm depends on the strategy for performing such a transformation and on how the resources of the GPU are used. For example, using the shared memory of the GPU is a common approach that increases the performance of algorithms in GPUs. However, it can also increase the sensitivity to soft errors by generating more DUEs. Specifically, we have performed fault injection campaigns on a slow memory-bound (`rc`) and a faster compute-bound (`block`) version of LU decomposition. The second version uses the fast shared memory of the GPU to reduce the execution time significantly. However, this results in a significant increase in the number of DUE. This behaviour is the same on a low-power embedded GPU and on a high-performance GPU. On the contrary, the number of SDCs in the second version of the algorithm is much smaller on both GPUs.

Our experiments also show that the SDCs generated when injecting the instructions of both algorithms and their kernels propagate to a quite different number of elements of the result matrix. While in the `rc` algorithm the average percentage of wrong elements as a result of an SDC is 28%, in the `block` algorithm this figure drops to 17%. Moreover, while the percentage of SDCs and DUE is quite similar in the three kernels of the algorithm `block`, the error propagation is quite different. That is, the percentage of elements affected is very different, as is the spatial pattern of error propagation.

As future work, it would be interesting to include an ABFT version of the LU to compare its behaviour with the two versions of the decomposition used in this work.

Author contributions All authors contributed equally to this work.

Funding Open Access funding provided thanks to the CRUE-CSIC agreement with Springer Nature. This work has been supported by the Spanish Government PID2020-113656RB-C21,

PID2022-138696OB-C21, PID2022-1370480A-C43 as well as the Regional Government of Madrid throughout the project MIMACUHSPACE-CM-UC3M.

Data availability No additional data or materials are available.

Declarations

Conflict of interest The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit http://creativecommons.org/licenses/by/4.0/.

References

1. Belloch JA, Badía JM, Igual FD, Cobos M (2019) Practical considerations for acoustic source localization in the IoT era: platforms, energy efficiency, and performance. IEEE Internet Things J 6(3):5068–5079. https://doi.org/10.1109/JIOT.2019.2895742
2. Rech P, Pilla LL, Navaux POA, Carro L (2014) 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, pp 455–466
3. Gomez LB, Cappello F, Carro L, DeBardeleben N, Fang B, Gurumurthi S, Pattabiraman K, Rech P, Reorda MS (2014) Gpgpus: how to combine high computational power with high reliability. In: 2014 Design, Automation & Test in Europe Conference & Exhibition (DATE). IEEE, pp 1–9
4. Belloch JA, Ramos G, Badia JM, Cobos M (2020) An efficient implementation of parallel parametric HRTF models for binaural sound synthesis in mobile multimedia. IEEE Access 8:49562–49573
5. Belloch JA, Badía JM, Igual FD, Gonzalez A, Quintana-Ortí ES (2018) Optimized fundamental signal processing operations for energy minimization on heterogeneous mobile devices. IEEE Trans Circuits Syst I Regul Pap 65(5):1614–1627
6. Sakhare KV, Tewari T, Vyas V (2020) Review of vehicle detection systems in advanced driver assistant systems. Arch Comput Methods Eng 27(2):591–610
7. Hatcher WG, Yu W (2018) A survey of deep learning: platforms, applications and emerging research trends. IEEE Access 6:24411–24432
8. Furano G, Menicucci A (2018) Roadmap for on-board processing and data handling systems in space. In: Dependable Multicore Architectures at Nanoscale. Springer, pp 253–281
9. Alcaide S, Kosmidis L, Tabani H, Hernandez C, Abella J, Cazorla FJ (2018) Safety-related challenges and opportunities for GPUs in the automotive domain. IEEE Micro 38(6):46–55. https://doi.org/10.1109/MM.2018.2873870
10. Demeshko I, Maruyama N, Tomita H, Matsuoka S (2013) Multi-GPU implementation of the NICAM atmospheric model. In: Caragiannis I, Alexander M, Badia RM, Cannataro M, Costan A, Danelutto M, Desprez F, Krammer B, Sahuquillo J, Scott SL, Weidendorfer J (eds) Euro-Par 2012: parallel processing workshops. Springer, pp 175–184
11. Badía JM, Amor-Martin A, Belloch JA, García-Castillo LE (2020) GPU acceleration of a non-standard finite element mesh truncation technique for electromagnetics. IEEE Access 8:94719–94730
12. Barreales GN, Novalbos M, Otaduy MA, Sanchez A (2021) Mdscale: Scalable multi-GPU bonded and short-range molecular dynamics. J Parall Distrib Comput 157:243–255. https://doi.org/10.1016/j.jpdc.2021.07.006

13. dos Santos FF, Pimenta PF, Lunardi C, Draghetti L, Carro L, Kaeli D, Rech P (2019) Analyzing and increasing the reliability of convolutional neural networks on GPUs. IEEE Trans Reliab 68(2):663–677
14. Canal R, Hernandez C, Tornero R, Cilardo A, Massari G, Reghenzani F, Fornaciari W, Zapater M, Atienza D, Oleksiak A et al (2020) Predictive reliability and fault management in exascale systems: state of the art and perspectives. ACM Comput Surv (CSUR) 53(5):1–32
15. Tiwari D, Gupta S, Rogers J, Maxwell D, Rech P, Vazhkudai S, Oliveira D, Londo D, DeBardeleben N, Navaux P, et al (2015) Understanding gpu errors on large-scale hpc systems and the implications for system design and operation. In: 2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA). IEEE, pp 331–342
16. Sastry Hari SK, Rech P, Tsai T, Stephenson M, Zulfiqar A, Sullivan M, Shirvani P, Racunas P, Emer J, Keckler SW (2020) Estimating silent data corruption rates using a two-level model. arXiv e-prints, 2005
17. dos Santos FF, Hari SKS, Basso PM, Carro L, Rech P (2021) Demystifying GPU reliability: comparing and combining beam experiments, fault simulation, and profiling. In: 2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS). IEEE, pp 289–298
18. León G, Badía JM, Belloch JA, Lindoso A, Entrena L (2020) Evaluating the soft error sensitivity of a GPU-based SoC for matrix multiplication. Microelectron Reliab 114:113856
19. Che S, Boyer M, Meng J, Tarjan D, Sheaffer JW, Lee S-H, Skadron K (2009) Rodinia: a benchmark suite for heterogeneous computing. In: 2009 IEEE International Symposium on Workload Characterization (IISWC). IEEE, pp 44–54
20. Didehban M, Shrivastava A (2018) A compiler technique for processor-wide protection from soft errors in multithreaded environments. IEEE Trans Reliab 67(1):249–263. https://doi.org/10.1109/TR.2018.2793098
21. Bodmann P, Papadimitriou G, Gizopoulos D, Rech P (2021) The impact of SoC integration and OS deployment on the reliability of ARM processors. In: 2021 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), pp 223–225. https://doi.org/10.1109/ISPASS51385.2021.00040. IEEE
22. Cotroneo D, Iannillo AK, Natella R, Rosiello S (2021) Dependability assessment of the android OS through fault injection. IEEE Trans Reliab 70(1):346–361. https://doi.org/10.1109/TR.2019.2954384
23. de Oliveira DAGG, Pilla LL, Santini T, Rech P (2016) Evaluation and mitigation of radiation-induced soft errors in graphics processing units. IEEE Trans Comput 65(3):791–804
24. Zhu Y, Liu Y, Zhang G (2020) FT-PBLAS: PBLAS-based fault-tolerant linear algebra computation on high-performance computing systems. IEEE Access 8:42674–42688. https://doi.org/10.1109/ACCESS.2020.2975832
25. Pilla LL, Rech P, Silvestri F, Frost C, Navaux POA, Reorda MS, Carro L (2014) Software-based hardening strategies for neutron sensitive FFT algorithms on GPUs. IEEE Trans Nucl Sci 61(4):1874–1880
26. Condia JER, dos Santos FF, Reorda MS, Rech P (2021) Combining architectural simulation and software fault injection for a fast and accurate CNNs reliability evaluation on GPUs. In: 2021 IEEE 39th VLSI Test Symposium (VTS). IEEE, pp 62–68. https://doi.org/10.1109/VTS50974.2021.9441044
27. Basso PM, Santos FFD, Rech P (2020) Impact of tensor cores and mixed precision on the reliability of matrix multiplication in GPUs. IEEE Trans Nucl Sci 67(7):1560–1565. https://doi.org/10.1109/TNS.2020.2977583
28. Condia JER, Rech P, dos Santos FF, Carrot L, Reorda MS (2021) Protecting GPU's Microarchitectural Vulnerabilities via Effective Selective Hardening. In: 2021 IEEE 27th International Symposium on On-Line Testing and Robust System Design (IOLTS), pp. 1–7. https://doi.org/10.1109/IOLTS52814.2021.9486703. IEEE
29. Abdelfattah A, Haidar A, Tomov S, Dongarra J (2018) Optimizing GPU kernels for irregular batch workloads: a case study for cholesky factorization. In: 2018 IEEE High Performance Extreme Computing Conference (HPEC). IEEE, pp 450–456. https://doi.org/10.1109/HPEC.2018.8547576
30. Alventosa FJ, Alonso P, Vidal AM, Piñero G, Quintana-Orti ES (2018) Fast block QR update in digital signal processing. J Supercomput 75(1):1051–1064
31. Bank B, Belloch JA, Välimäki V (2017) Efficient design of a parallel graphic equalizer. J Audio Eng Soc 65(10):817–825. https://doi.org/10.17743/jaes.2017.0029
32. Zhou G, Bo R, Chien L, Zhang X, Shi F, Xu C, Feng Y (2017) GPU-based batch LU-factorization solver for concurrent analysis of massive power flows. IEEE Trans Power Syst 32(6):4975–4977. https://doi.org/10.1109/TPWRS.2017.2662322

33. Wu P, DeBardeleben N, Guan Q, Blanchard S, Chen J, Tao D, Liang X, Ouyang K, Chen Z (2017) Silent data corruption resilient two-sided matrix factorizations. In: Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pp 415–427
34. Wu P, Guan Q, DeBardeleben N, Blanchard S, Tao D, Liang X, Chen J, Chen Z (2016) Towards practical algorithm based fault tolerance in dense linear algebra. In: Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing, pp 31–42
35. Chen J, Li H, Li S, Liang X, Wu P, Tao D, Ouyang K, Liu Y, Zhao K, Guan Q, et al (2018) Fault tolerant one-sided matrix decompositions on heterogeneous systems with GPUs. In: SC18: International Conference for High Performance Computing, Networking, Storage and Analysis. IEEE, pp 854–865
36. Huang K-H, Abraham JA (1984) Algorithm-based fault tolerance for matrix operations. IEEE Trans Comput 100(6):518–528
37. Davies T, Chen Z (2013) Correcting soft errors online in LU factorization. In: Proceedings of the 22nd International Symposium on High-Performance Parallel and Distributed Computing. ACM, pp 167–178
38. Wu P, Chen Z (2014) FT-ScaLAPACK: correcting soft errors on-line for ScaLAPACK Cholesky, QR, and LU factorization routines. In: Proceedings of the 23rd International Symposium on High-Performance Parallel and Distributed Computing. ACM, pp 49–60
39. Tselonis S, Gizopoulos D (2016) GUFI: A framework for GPUs reliability assessment. In: 2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS). IEEE, pp 90–100
40. Hari SKS, Tsai T, Stephenson M, Keckler SW, Emer J (2017) 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, pp 249–258
41. Previlon FG, Kalra C, Tiwari D, Kaeli DR (2019) PCFI: Program counter guided fault injection for accelerating gpu reliability assessment. In: 2019 Design, Automation & Test in Europe Conference & Exhibition (DATE). IEEE, pp 308–311
42. Fickenscher J, Reinhart S, Hannig F, Teich J, Bouzouraa ME (2017) Convoy tracking for ADAS on embedded GPUs. In: 2017 IEEE Intelligent Vehicles Symposium (IV). IEEE, pp 959–965
43. Kosmidis L, Rodriguez I, Jover Á, Alcaide S, Lachaize J, Abella J, Notebaert O, Cazorla FJ, Steenari D (2020) GPU4S: embedded GPUs in space-latest project updates. Microprocess Microsyst 77:103143
44. Badia JM, Leon G, Belloch JA, Lindoso A, Garcia-Valderas M, Morilla Y, Entrena L (2022) Reliability evaluation of LU decomposition on GPU-accelerated system-on-chip under proton irradiation. IEEE Trans Nucl Sci 69(7):1467–1474
45. NVIDIA (2021) CUDA C++ programming guide. PG-02829-001_v11.2. Design Guide
46. NVIDIA (2014) NVIDIA Tegra K1. A new era in mobile computing. NVIDIA Whitepaper
47. NVIDIA (2017) NVIDIA Tesla V100 GPU Architecture. The World's Most Advanced Data Center GPU. WP-08608-001_v1.1. NVIDIA,. NVIDIA
48. Noh J, Correas V, Lee S, Jeon J, Nofal I, Cerba J, Belhaddad H, Alexandrescu D, Lee Y, Kwon S (2015) Study of neutron soft error rate (SER) sensitivity: investigation of upset mechanisms by comparative simulation of FinFET and planar MOSFET SRAMs. IEEE Trans Nucl Sci 62(4):1642–1649
49. Li G, Pattabiraman K, Cher C-Y, Bose P (2016) Understanding error propagation in GPGPU applications. In: SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. IEEE, pp 240–251
50. Oliveira D, Pilla L, DeBardeleben N, Blanchard S, Quinn H, Koren I, Navaux P, Rech P (2017) Experimental and analytical study of xeon phi reliability. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, pp 1–12
51. Oliveira D, Frattin V, Navaux P, Koren I, Rech P (2017) Carol-fi: an efficient fault-injection tool for vulnerability evaluation of modern hpc parallel accelerators. In: Proceedings of the Computing Frontiers Conference, pp 295–298
52. NVIDIA (2021) CUDA-GDB. CUDA Debugger. User Manual. DU-05227-042_v11.2
53. NVIDIA (2023) Profiler User's Guide. https://docs.nvidia.com/cuda/profiler-users-guide, v12.1
54. Tomov S, Dongarra J, Baboulin M (2010) Towards dense linear algebra for hybrid GPU accelerated manycore systems. Parallel Comput 36(5–6):232–240. https://doi.org/10.1016/j.parco.2009.12.005

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.