León, Germán, et al. Evaluating the soft error sensitivity of a GPU-based SoC for matrix multiplication.
In: *Microelectronics reliability, vol. 114,* 113856 *(Special issue: Proceedings of ESREF 2020: 31st European Symposium on Reliability of Electron Devices, Failure Physics and Analysis, October 4th to 8th, Athens, Greece, virtual conference). Elsevier, November 2020, 6 p.*

DOI: https://doi.org/10.1016/j.microrel.2020.113856

Evaluating the Soft Error Sensitivity of a GPU-based SoC for Matrix Multiplication

Germán León[b], José M. Badía[b], Jose A. Belloch[a,*], Almudena Lindoso[a], Luis Entrena[a]

[a]*Depto. de Tecnología Electrónica, Universidad Carlos III de Madrid, Spain.*
[b]*Depto. de Ingeniería y Ciencia de Computadores, Universitat Jaume I de Castellon, Spain.*

Abstract

System-on-Chip (SoC) devices can be composed of low-power multicore processors combined with a small graphics accelerator (or GPU) which offers a trade-off between computational capacity and low-power consumption. In this work we use the LLFI-GPU fault injection tool on one of these devices to compare the sensitivity to soft errors of two different CUDA versions of matrix multiplication benchmark. Specifically, we perform fault injection campaigns on a Jetson TK1 development kit, a board equipped with a SoC including an NVIDIA "Kepler" Graphics Processing Unit (GPU). We evaluate the effect of modifying the size of the problem and also the thread-block size on the behaviour of the algorithms. Our results show that the block version of the matrix multiplication benchmark that leverages the shared memory of the GPU is not only faster than the element-wise version, but it is also much more resilient to soft errors. We also use the cuda-gdb debugger to analyze the main causes of the crashes in the code due to soft errors. Our experiments show that most of the errors are due to accesses to invalid positions of the different memories of the GPU, which causes that the block version suffers a higher percentage of this kind of errors.

Keywords: GPU, Soft Errors, Sensitivity, Fault injection

1. Introduction

Modern accelerators (Graphics Processing Unit) are becoming increasingly relevant because of their massive parallel resources and high performance per watt. The reliability of these devices is especially important in two environments: supercomputer sites and embedded systems used in safety-critical domains. On the former, most of the TOP500 supercomputer sites[1] include high-end GPUs using state-of the-art technology and consisting of a very large number of cores, complex schedulers, large registers and caches. Due to the huge number of GPUs included in these sites, their probability of failure grows rapidly. On the latter, GPUs are one of the main components of embedded systems used in critical domains, such as advanced driver assistance systems [1], avionics or space applications [2, 3]. In these domains other parameters such as the size or the energy consumption have to be taken into account. Therefore, it is very important to evaluate the reliability of modern GPUs and to design fault tolerant techniques and algorithms for this kind of devices.

First approaches in terms of reliability of GPUs were carried out by Paolo Rech in [4, 5], where radiation effects are measured in different GPUs and the fault injection tool SASSIFI [6] is used in high-end GPUs. To our knowledge, there are only few fault injection tools for GPU devices. On the one hand, we can highlight the tool SASSIFI, [6] which allows injecting a variable number of errors in different components of the architecture. However, this tool is not currently supported by Tegra SoCs. On the other hand, GPU-Qin [7] and CAROL-FI [8] are based on cuda-gdb [9], but their use produces significant performance degradation. Finally, the LLFI-GPU [10] tool is the only one that can be installed on the Tegra K1 and does not depend on the CUDA debugger.

The LLFI-GPU tool injects bit-flips in the results of the instructions run by one of the threads that executes the CUDA kernels. In this work, we use this fault injection tool to compare the sensitivity to soft errors of two different versions of matrix multiplication benchmark. To the best of our knowledge this is the first work where the resiliency of different approaches to this fundamental computational kernel are compared. In our experiments we analyzed the effect of modifying the size of the problem and also the thread-block size on the resiliency to soft errors of matrix multiplication. We have also used the cuda-gdb debugger to assess the main causes of the crashes produced on the code when injecting this kind of errors.

Besides, we have performed our experiments in a low-power embedded platform. Specifically, we have use a Jetson TK1, which includes a Tegra K1 SoC with a "Kepler" K20A GPU. This kind of devices is very well suited to safety-critical domains, where it is especially important to use fault tolerant algorithms.

Our experimental results show that the block version of matrix multiplication benchmark, which leverages the shared memory of the GPU, is not only faster, but also much more resilient to soft errors than the element-wise version. Besides, our analysis shows the size of the problem and the thread-block size

*Corresponding author: Phone Number +34-916245987
Email addresses: leon@uji.es (Germán León), badia@uji.es (José M. Badía), jbelloc@ing.uc3m.es (Jose A. Belloch), alindoso@ing.uc3m.es (Almudena Lindoso), entrena@ing.uc3m.es (Luis Entrena)

[1] https://www.top500.org

have very slight influence on the behaviour of both matrix multiplication versions when using the model of fault injection of the LLFI-GPU tool. Finally, we have seen that the main causes of the crashes of the algorithms produced by soft errors are the accesses to invalid positions of the different memories of the GPU.

The rest of the paper is structured as follows. Section 2 reviews related works that deal with radiation and fault injection on GPUs. Section 3 describes the experimental environment. Experimental results and evaluation is shown in Section 4. Finally, Section 5 provides the concluding remarks.

2. Related work

Two main techniques can be used to test the fault tolerance of computer components: radiation and fault injection [11, 12]. A limited number of fault injection tools have been developed for GPUs due to the scarce information about the internal architecture of these accelerators, especially in the case of low-power GPUs included in embedded system-on-chip (SoC) devices [13, 8]. Many experiments have leveraged benchmarks to test the behaviour of GPUs with different capabilities. A few use synthetic codes to test some specific components of the devices, such as caches [14], register files or schedulers [15]. However, most of the experiments employ simple kernels or mini-applications such as those included in the Rodinia and Parboil benchmark suites [15, 10]. In recent years, some experiments have employed neural networks as testbeds [16, 17]. Another interesting aspect that has recently received attention is the impact of data and arithmetic precision on the reliability of novel GPUs, such as NVIDIA's Volta architecture with tensor cores [18, 19].

One of the most employed benchmarks to evaluate the reliability of GPUs is matrix multiplication. In [15] the authors test the overall GPU radiation sensitivity dependence to the Degree of Parallelism of this kernel. They vary both the size of the matrices and the thread-block size to modify the scheduling strain and the use of other resources. Another radiation-induced evaluation is performed in [12], where the authors use matrix multiplication and other benchmarks to evaluate the behaviour of a NVIDIA K40 GPU. In a very recent paper [20] the authors also perform radiation experiments to evaluate the reliability of matrix multiplication implemented using tensor cores and mixed precision on Volta GPUs. In [21] the `cublasSegmm` version of the multiplication was used to check the performance variations and silent data corruption on a large cluster including multiple GPUs

Matrix multiplication benchmark has also been used to develop new frameworks or methodologies to increase the fault tolerance of GPUs. For example, two variants of the Duplication-with-Comparison (DWC) technique are evaluated in [22] using this kernel on a NVIDIA Fermi architecture. In [23] the authors used the `sgemm` benchmark and others included in the Rodinia test suite to develop a framework to predict failures in GPU programs. They modified the SASSIFI tool to identify the scalar and vector instructions of the codes. Triple-Modular Redundancy (TMR), persistent threading and CUDA streams are combined in [24] to mask and mitigate single-event upsets (SEUs) on a Tegra X1 device. The authors implement a simple ad-hoc fault injection technique to test this technique on a image processing method. To the best of our knowledge [24], this is the only paper where Tegra SoCs have been used before to evaluate the fault tolerance of the GPUs included in this kind of devices.

3. GPU-based experimental environment

3.1. Jetson Tegra K1 SoC

We have used a Tegra K1 (TK1) System-on-Chip (SoC), embedded in the Jetson development kit. This particular system comprises a quad-core ARM Cortex A15 processor (or CPU), an ARM Cortex A15 battery-saving shadow core, and an NVIDIA "Kepler" K20A GPU with 1 Streaming Multiprocessor (SM) containing 192 CUDA cores. Therefore, the TK1 combines the luring low-power consumption of embedded systems with the ample hardware parallelism of graphics accelerators. The code to be executed in parallel on the GPU by multiple elementary processes, called threads, is written as a CUDA kernel function, [25]. The threads are logically grouped into thread blocks which are assigned to an SM of the GPU device and share memory. Thread blocks are then organized in a grid. Software developers select the thread-block size aiming to maximize the occupancy of the GPU and the performance of the application, but this can also affect its reliability.

3.2. LLFI-GPU fault injection tool

LLFI-GPU is an extension of the open-source LLFI fault injection tool [26]. This tool uses the LLVM compiler framework [27] to instrument the code and inject faults. First, the fault injection tool profiles the program to obtain the total number of kernel calls, the total number of threads that execute each kernel, and the number of instructions executed by each thread. Then the tool instruments the LLVM IR (Intermediate Representation) and passes it to the nvcc CUDA compiler. The tool only modifies the CUDA portion of the code.

Afterward, at runtime, the fault injection tool chooses a random thread of a random CUDA kernel call and then modifies the result of one of its instructions. The instruction is also chosen randomly so that all the instructions executed have the same probability of being modified. Specifically, it flips one of the bits of the result of the instruction and resumes the application execution. Therefore, the LLFI-GPU fault injection tool is only modifying the results of the instruction that writes to the general-purpose registers and does not inject errors in other components of the architecture, such as the GPU memory, the condition codes or the store addresses and values. Besides, it only injects one bit-flip in one of the CUDA kernels in each execution of the code.

3.3. Benchmark code: Matrix multiplication

We have tested the effect of this kind of fault injection on matrix multiplication benchmark, $A = B \times C$, which is a widely-used numerical routine. Matrix multiplication benchmark is an

embarrassingly parallel problem, because the computation of the different elements of the result matrix is fully independent.

We have used two well-know implementations of matrix multiplication [25]:

- `mmElem` is a straightforward implementation of the multiplication where every thread computed on 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. All the elements of the matrices are stored in the global memory of the GPU.

- `mmBlock` is a block version of the multiplication. Every thread-block is in charge of computing one block of matrix C in the shared memory of the GPU and then storing it in the global memory. This block is computed as the product of a row of blocks of matrix A and a column of blocks of matrix B. All the threads on each thread-block collaborate synchronously to load different blocks of matrices A and B from global to shared memory. This scheme leverages the fast shared memory and reduces the global memory bandwidth. As in the `mmElem` code, each thread computes one element of C using one row of A and one column of B.

3.4. Experimental methodology

In our experiments, we have used the following error categories:

- `Masked`: we obtain a correct result of matrix multiplication. Results are compared with a golden version previously computed.

- `Silent Data Corruption (SDC)`: one or more elements of the result do not match the golden output.

- `Crash`: an error occurs because the program attempted to perform some invalid action (e.g. read outside its memory segment). This error can be captured using a debugger, the process can be killed and the Operating System can launch the next test (e.g. matrix multiplication).

- `Hang`: the system reaches an abnormal state and cannot continue the execution of the benchmark. In our case, the process performing the matrix multiplication cannot be interrupted and so we cannot use the GPU. Therefore, the only solution is to reboot the Operating System.

We have used a slight modification of the script included in the LLFI-GPU to perform our experiments. We use the Python module `Pexpect` to spawn and control the subprocess executing the multiplication. This module reduces the number of Hangs of the tests. Besides, our experimental setup includes two timeouts. The first timeout, associated to the spawned subprocess, is adjusted to time slightly larger than the maximum foreseeable duration of one iteration of the selected benchmark. We use this timeout or the cuda-gdb debugger to detect the crashes of our test. Three consecutive timeouts account for a hang, in which case we reboot the operating system. Additionally, we have used the watchdog Linux API to implement a hardware watchdog that reboots the system if it is hung during more than 60 seconds.

4. Experimental results and evaluation

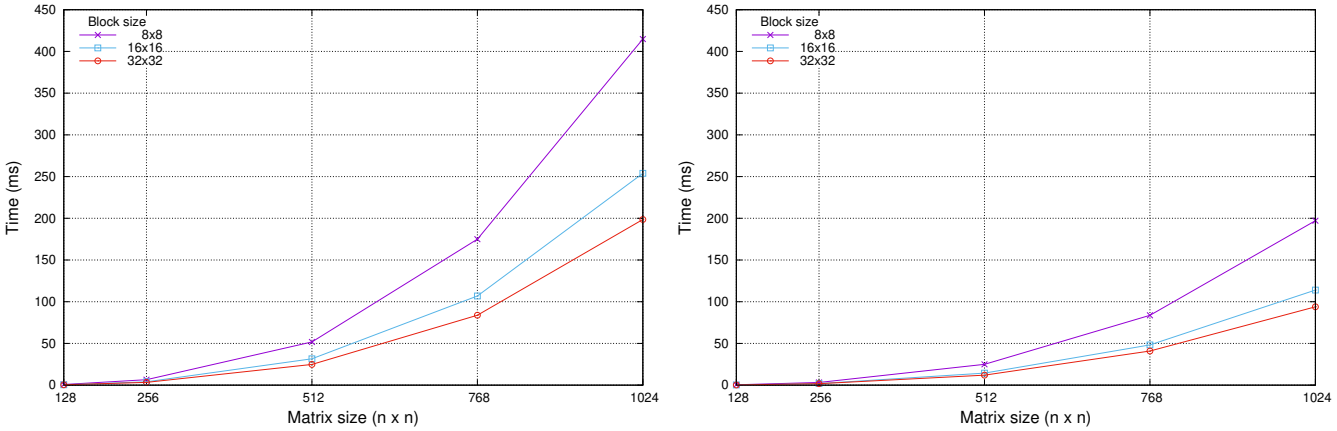
We have evaluated the effects of the matrix size and the thread-block size on the sensitivity of matrix multiplication benchmark. In the system under test, the size of the CUDA grid depends on the size of the matrix and the thread-block size.

Figure 1 illustrates the matrix multiplication benchmarks behaviour by showing the relationship between performance and matrix size, ranging from 128 to 1024. It includes the results with three thread-block sizes, 8x8, 16x16 and 32x32. Both versions of the benchmark have a very similar behaviour but, as expected by the use of the shared memory, the `mmBlock` version is twice as fast as the `mmElem` version. Figure 1 also shows that the use of larger blocks of threads improves parallel performance of both versions of the code, as we are increasing the number of threads that can leverage the 192 cores of the Jetson TK1 GPU.

We performed a fault injection campaign for both versions of matrix multiplication benchmark with varying matrix size. We performed 1,000 matrix multiplication per version and size and injected one fault per matrix multiplication.

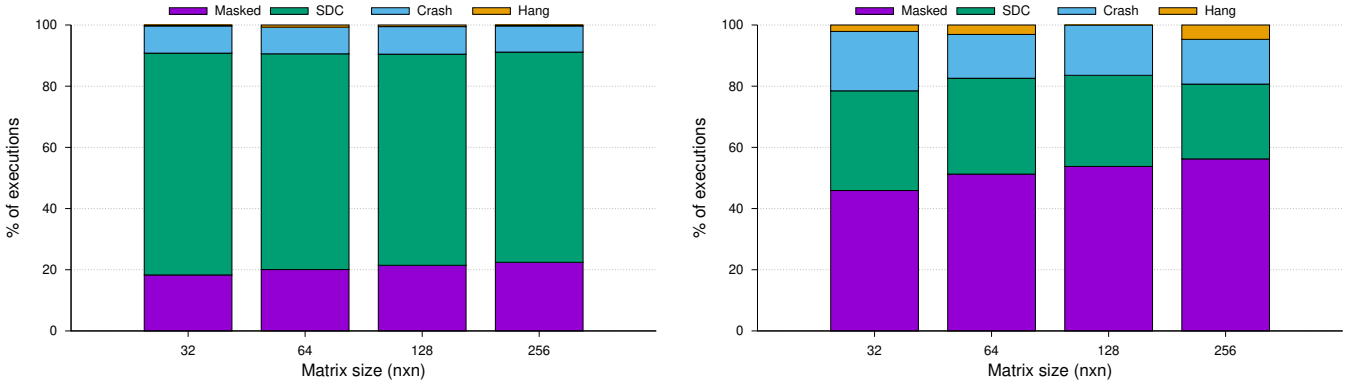
In each matrix multiplication the fault injection tool performs a bit-flip in the result of one instruction on one thread. Hang and crash errors are detected with watchdogs and SDC with golden result comparison accomplished at the end of the benchmark execution. Figure 2 shows that the effect of the matrix size on the behaviour of both versions of the code is very similar. The number of Masked errors and SDCs is almost constant in the `mmElem` version. In the case of the `mmBlock` version, the Masked errors increase slightly while the SDCs decrease accordingly when we increase the size of the matrices. On the contrary, the percentages of errors on both versions is very different. The `mmElem` version is much more sensitive, with around 80% of the executions resulting in SDC or crash. In the `mmBlock` version, around 50% of the injected faults are masked. However, the number of crashes and even hangs of this version is much larger: between 20% and 25%. We think that the matrix size has such a small effect on the behaviour of the multiplication due to the fault injection model employed by LLFI-GPU. Recall that this tool always injects only one bit-flip per execution, independently of the size of the problem or the time spent by the algorithm to solve it. The amount of global memory employed by the algorithm does not affect the fault injection results.

We have observed that all the SDCs produced with both versions of the code only affect one element of the result matrix C. This is due to the kind of parallelism exploited in the implementations and the kind of faults injected by the LLFI-GPU tool. In both versions the results of all the instructions executed by each thread modify only values that are used by the same thread. Therefore, any bit-flip in the results of those instructions will not affect the results obtained by other threads. It is



(a) `mmElem` (b) `mmBlock`

Figure 1: Execution time of the two matrix multiplication versions, (a) `mmElem` and (b) `mmBlock`, versus matrix size.



(a) `mmElem` (b) `mmBlock`

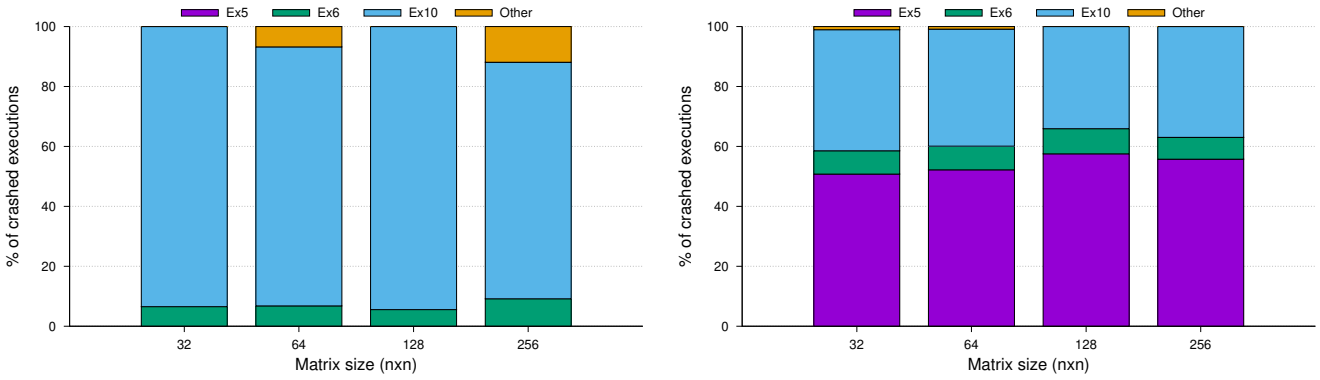
Figure 2: Fault injection results of matrix multiplication versus matrix size. In (a) and (b), we show the implementations denoted as `mmElem` and `mmBlock`, respectively. We use in both cases a thread-block size = 32×32.

true that the threads share the elements of matrices A and B, which are stored in the global or shared memories of the GPU. However, the threads only read those shared elements and their values are not affected by the fault injection.

We have implemented a variant of the `mmElem` code to study the effect of the Degree of Parallelism (DOP) in the error sensitivity. In this variant of the code we use a constant number of threads which is independent of the size of the matrices, thus keeping constant the DOP. As we increase the size of the matrices, the workload each thread has to complete also increases. Specifically, if we duplicate the size of the matrices, the number of elements computed by each thread increases four-fold. Irradiation experiments reported in [15] using this variant of the code show that the number of SDCs clearly depends on the DOP, because different DOPs affect the use of the schedulers in the GPU and also the number of registers per thread or the occupancy of the SMs. However, if we only modify the result of one instruction in one thread independently of the size of the problem or the number of threads, then the number of SDCs is not affected by the DOP. Therefore, our experiments show that the results obtained with this variant of the code are almost identical to the ones shown on all the figures for the `mmElem` code.

In order to have more information about the causes of the crashes we have performed other injection campaigns with the same codes and cases, but using cuda-gdb to run the executions. This execution mode allows us a better control of the execution and also the possibility of capturing different types of exceptions using the debugger. However, it has the disadvantage of increasing considerably the execution time which makes testing with large matrices prohibitive. Figure 3 shows the percentage of crashed executions causes by different kinds of errors. Specifically, the cuda-gdb debugger cached only 3 of the 15 types of exceptions defined in [9], all of them related with memory access problems:

- `Ex5`: occurs when any thread within a warp accesses an address that is 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.



(a) `mmElem` (b) `mmBlock`

Figure 3: Causes of the crashes of matrix multiplication versus matrix size. In (a) and (b), we show the implementations denoted as `mmElem` and `mmBlock`, respectively. We use in both cases a thread-block size = 32×32.

Results in Figure 3 show that most of the crashes of the `mmElem` code are due to illegal accesses to global memory and only a few are due to problems when accessing the local memory. Recall that this version of the code does not use shared memory. On the contrary, the use of shared memory in the `mmBlock` version increases the performance of the code, but at the expense of incurring in many more crashes related with the access to that kind of fast memory.

Finally, we evaluate the effect of the thread-block size on the behaviour of the matrix multiplication benchmarks under evaluation. The results of the instructions executed by each thread do not depend on the thread-block size. Therefore, the observed cases do not vary when we modify this parameter without changing the matrices size, as we can see in Figure 4.

5. Conclusions

In this work we compared the sensitivity to soft errors of two CUDA versions of the matrix multiplication benchmark. To this end, we used the LLFI-GPU tool to perform fault injection campaigns on a low power Kepler GPU included in a Tegra K1 SoC.

An important conclusion is that the distribution of soft errors does not depend on the size of the matrices and depends only slightly on the thread-block size in the `mmBlock` version. This results are due to the working mode of the fault injection tool, since it only injects a bit-flip in the result of only one instruction in only one of the threads that are employed in the computation. Besides, matrix multiplication benchmark, in all the evaluated variants, is highly parallel since threads do not share intermediate results in order to compute each element, which precludes cascade errors.

We have been able to appreciate small differences among the different strategies in carrying out matrix multiplication. The block version of matrix multiplication (`mmBlock`) not only is clearly more efficient that the element-wise version (`mmElem`), due to its use of the shared memory, but it is also much less sensitive to soft errors affecting the results of the instructions. However, the `mmBlock` version is more prone to crashes due to soft errors occurred when accessing the shared memory.

Finally, a meaningful conclusion can be extracted by using the cuda-gdb debugger in order to identify the origin of the results of the error type Crash. Our results show that three kinds of exceptions are catched when a crash failure occurs. Depending on the strategy for computing matrix multiplication, the quantity and type of exceptions varies. The element-wise version produces mostly exceptions related to the access to the global memory, while the block version of matrix multiplication, despite of obtaining higher performance, produces a higher number of crashes due to the access to the shared memory.

Acknowledgements

This work has been supported by the Spanish Government through TIN2017-82972-R and ESP2015-68245-C4-1-P, and by the Valencian Regional Government through PROMETEO/2019/109.

References

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

[2] L. Kosmidis, J. Lachaize, J. Abella, O. Notebaert, F. J. Cazorla, D. Steenari, GPU4S: Embedded GPUs in Space, in: 2019 22nd Euromicro Conference on Digital System Design (DSD), 2019, pp. 399–405.

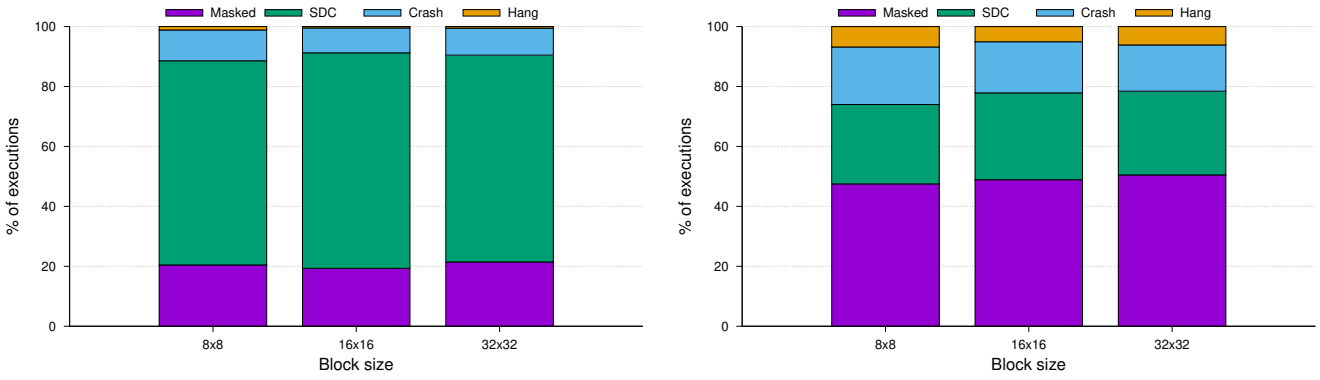
[3] F. Kastensmidt, P. Rech (Eds.), FPGAs and parallel architectures for aerospace applications, Springer, 2016.

[4] P. Rech, L. L. Pilla, P. O. A. Navaux, 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, 2014, pp. 455–466.

[5] F. F. d. Santos, P. F. Pimenta, C. Lunardi, L. Draghetti, L. Carro, D. Kaeli, P. Rech, Analyzing and increasing the reliability of convolutional neural networks on gpus, IEEE Transactions on Reliability 68 (2019) 663–677.

[6] S. K. S. Hari, T. Tsai, M. Stephenson, S. W. Keckler, 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), 2017, pp. 249–258.

[7] B. Fang, K. Pattabiraman, M. Ripeanu, S. Gurumurthi, GPU-Qin: A methodology for evaluating the error resilience of GPGPU applications, in: 2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), IEEE, 2014, pp. 221–230.



(a) `mmElem` (b) `mmBlock`

Figure 4: Fault injection results of matrix multiplication versus thread-block size. Matrix size = 128×128.

[8] D. Oliveira, V. Frattin, P. Navaux, I. Koren, P. Rech, CAROL-FI: an efficient fault-injection tool for vulnerability evaluation of modern hpc parallel accelerators, in: Proceedings of the Computing Frontiers Conference, 2017, pp. 295–298.

[9] NVIDIA, CUDA-GDB. CUDA Debugger. DU-05227-042_v10.2. User Manual, 2019.

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

[11] D. A. Oliveira, C. B. Lunardi, L. L. Pilla, P. Rech, P. O. Navaux, L. Carro, Radiation sensitivity of high performance computing applications on kepler-based GPGPUs, in: 2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, IEEE, 2014, pp. 732–737.

[12] D. A. G. De Oliveira, L. L. Pilla, M. Hanzich, V. Fratin, F. Fernandes, C. Lunardi, J. M. Cela, P. O. A. Navaux, L. Carro, P. Rech, Radiation-induced error criticality in modern HPC parallel accelerators, in: 2017 IEEE International Symposium on High Performance Computer Architecture (HPCA), IEEE, 2017, pp. 577–588.

[13] S. K. S. Hari, T. Tsai, M. Stephenson, S. W. Keckler, 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.

[14] D. A. Oliveira, P. Rech, H. M. Quinn, T. D. Fairbanks, L. Monroe, S. E. Michalak, C. Anderson-Cook, P. O. Navaux, L. Carro, modern gpus radiation sensitivity evaluation and mitigation through duplication with comparison, IEEE Transactions on Nuclear Science 61 (2014) 3115–3122.

[15] P. Rech, L. L. Pilla, P. O. A. Navaux, 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.

[16] F. F. dos Santos, L. Draghetti, L. Weigel, L. Carro, P. Navaux, P. Rech, Evaluation and mitigation of soft-errors in neural network-based object detection in three gpu architectures, in: 2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W), IEEE, 2017, pp. 169–176.

[17] F. F. dos Santos, P. F. Pimenta, C. Lunardi, L. Draghetti, L. Carro, D. Kaeli, P. Rech, Analyzing and increasing the reliability of convolutional neural networks on GPUs, IEEE Transactions on Reliability 68 (2018) 663–677.

[18] F. F. dos Santos, C. Lunardi, D. Oliveira, F. Libano, P. Rech, Reliability evaluation of mixed-precision architectures, in: 2019 IEEE International Symposium on High Performance Computer Architecture (HPCA), IEEE, 2019, pp. 238–249.

[19] F. F. dos Santos, P. Navaux, L. Carro, P. Rech, Impact of Reduced Precision in the Reliability of Deep Neural Networks for Object Detection, in: 2019 IEEE European Test Symposium (ETS), IEEE, 2019, pp. 1–6.

[20] P. Martins Basso, F. F. dos Santos, P. Rech, Impact of Tensor Cores and Mixed-Precision on the Reliabiity of Matrix Multiplication in GPUs, IEEE Transactions on Nuclear Science in press (2020) 1–6.

[21] L. B. Gomez, F. Cappello, L. Carro, N. DeBardeleben, B. Fang, S. Gurumurthi, K. Pattabiraman, P. Rech, 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. 1–9.

[22] D. Sabena, M. S. Reorda, L. Sterpone, P. Rech, L. Carro, On the evaluation of soft-errors detection techniques for GPGPUs, in: 2013 8th IEEE Design and Test Symposium, IEEE, 2013, pp. 1–6.

[23] C. Kalra, F. Previlon, X. Li, N. Rubin, D. Kaeli, PRISM: Predicting resilience of gpu applications using statistical methods, in: SC18: International Conference for High Performance Computing, Networking, Storage and Analysis, IEEE, 2018, pp. 866–879.

[24] A. Milluzzi, A. George, Exploration of TMR fault masking with persistent threads on Tegra GPU SoCs, in: 2017 IEEE Aerospace Conference, IEEE, 2017, pp. 1–7.

[25] NVIDIA, CUDA C++ Programming Guide. PG-02829-001_v10.2. Design Guide, 2019.

[26] C. Lattner, V. Adve, Llvm: A compilation framework for lifelong program analysis & transformation, in: International Symposium on Code Generation and Optimization, 2004. CGO 2004., IEEE, 2004, pp. 75–86.

[27] J. Wei, A. Thomas, G. Li, K. Pattabiraman, Quantifying the accuracy of high-level fault injection techniques for hardware faults, in: 2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, IEEE, 2014, pp. 375–382.