Towards Studying the Effect of Compiler Optimizations and Software Randomization on GPU Reliability

Pau López Castillón
Universitat Politècnica de Barcelona (UPC), Spain
Barcelona Supercomputing Center (BSC), Spain

Xavier Caricchio Hernández
Universitat Politècnica de Barcelona (UPC), Spain
Barcelona Supercomputing Center (BSC), Spain

Leonidas Kosmidis
Barcelona Supercomputing Center (BSC), Spain
Universitat Politècnica de Barcelona (UPC), Spain

Abstract

The evolution of Graphics Processing Unit (GPU) compilers has facilitated the support for general-purpose programming languages across various architectures. The NVIDIA CUDA Compiler (NVCC) employs multiple compilation levels prior to generating machine code, implementing intricate optimizations to enhance performance. These optimizations influence the manner in which software is mapped to the underlying hardware, which can also impact GPU reliability.

TASA is a source-to-source code randomization tool designed to alter the mapping of software onto the underlying hardware. It achieves this by generating random permutations of variable and function declarations, thereby introducing random padding between declarations of different types and modifying the program memory layout. Since this modifies their location in the memory, it also modifies their cache placement, affecting both their execution time (due to the different conflicts between them, which result in a different amount of cache misses in every execution), as well as their lifetime in the cache.

In this work, which is part of the HiPEAC Student Challenge 2025, we first examine the reproducibility of a subset of data presented in the ACM TACO paper "Assessing the Impact of Compiler Optimizations on GPU Reliability" [10], and second we extend it by combining it with our proposal of software randomization. The paper indicates that the -O3 optimization flag facilitates an increased workload before failures occur within the application. By employing TASA, we investigate the impact of GPU randomization on reliability and performance metrics.

By reproducing the results of the paper on a different GPU platform, we observe the same trend as reported in the original publication. Moreover, our preliminary results with the application of software randomization show in several cases an improved Mean Waiting Before Failure (MWBF) compared to the original source code.

2012 ACM Subject Classification General and reference → Reliability; Software and its engineering → Compilers; Computing methodologies → Graphics processors

Keywords and phrases Graphics processing units, reliability, software randomization, error rate

Digital Object Identifier 10.4230/OASIcs.PARMA-DITAM.2025.4

Funding This work was supported by the ESA funded project "Open Source Software Randomisation Framework for Probabilistic WCET Prediction and Security on (multicore) CPUs, GPUs and Accelerators" as well as European Commission's METASAT Horizon Europe project (grant agreement 101082622). Moreover, it was also partially supported by the Spanish Ministry of Economy and Competitiveness (MINECO) under the grant IJC2020-045931-I.

16th Workshop on Parallel Programming and Run-Time Management Techniques for Many-Core Architectures and 14th Workshop on Design Tools and Architectures for Multicore Embedded Computing Platforms (PARMA-DITAM 2025).
Editors: Daniele Cattaneo, Maria Fazio, Leonidas Kosmidis, and Gabriele Morabito; Article No. 4; pp. 4:1–4:10
OpenAccess Series in Informatics
OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Introduction

With the latest advancements in GPU architectures and their compilers, there are many optimization opportunities, which allow the mapping of complex general purpose GPU (GPGPU) code on these architectures in a significantly different manner. This results not only in a performance difference, but also in different reliability properties [10].

Fernando Fernandes Dos Santo et al. in their ACM TACO article "Assessing the Impact of Compiler Optimizations on GPUs Reliability" [10] demonstrated how different compiler optimization flags impact the final reliability of GPU applications and used GPU fault injection to estimate the MWBF (Mean Work Between Failures) value.

Despite knowing that applications compiled with -O3 may have an elevated risk of encountering critical errors due to the higher optimization levels that eliminate redundant or non-executed code, the substantial reduction in execution time – resulting in enhanced performance – allows for a greater volume of work to be accomplished prior to such errors occurring.

On the other hand, TASA [5] is a source-to-source code randomization compiler designed to alter the mapping of software to the underlying hardware without altering its functionality. However, unlike compiler optimizations, TASA preserves the same number of instructions. The primary goal of TASA's development was to streamline the Worst Case Execution Time (WCET) computation for Critical Real Time Embedded Systems. This analysis is often resource-intensive, particularly when performed in a static manner through abstract interpretation and cache analysis, in which all possible execution paths are considered in order to determine whether each memory access will hit, miss or may miss in the cache. Such analysis is very sensitive to the memory layout and in fact requires knowing the exact memory layout of the program.

Instead, TASA randomizes the memory layout for each execution, allowing the use of Measurement Based Probabilistic Timing analysis for the probabilistic WCET (pWCET) estimation. In particular, TASA facilitates the computation of the pWCET by collecting a series of execution times, which are then processed with a statistical method known as Extreme Value Theory (EVT), which can estimate the maximum of a probability distribution.

The randomization of the memory layout creates different mappings in the cache with different conflicts among the contents of the cache lines, which subsequently randomizes the program's execution time.

This is the main reason why analyzing the conclusions presented in the original paper [10], we wondered how applying TASA to the original source code would impact the reliability metrics and how they would be related with the ones presented in [10].

This work has been performed in the context of the HiPEAC Student Challenge 2025. The purpose of this competition is to reproduce the experiments of a recent paper published in the ACM Transactions on Architecture and Code Optimization (TACO) journal on a different platform, and optionally improve or optimize the proposed solution of the paper.

In summary, the contributions of this work-in-progress paper is the following: a) we reproduce a subset of the GPU fault injection experiments of [10] in a different GPU platform, namely an NVIDIA 1080Ti GPU based on the Pascal architecture which was not included in the original publication. Our results follow the same trend reported in the original publication, which is that the -O3 optimization level is beneficial for GPU reliability. b) we extend the original work by repeating the same experiments under software randomization. Our preliminary results show an improved average MWBF (Mean Work Between Failures) compared to the original source code.

The mean work between failures, as its name indicates, is the total amount of work done by the executable before an error occurs. It is defined with the following equation:

$$MWBF = total\ time/errors \tag{1}$$

2 Background and Related Works

This section presents the previous works in the literature on which our proposal is based, as well as similar works.

2.1 Radiation benchmark

In Fernando Fernandez et al. [10], the authors evaluate the likelihood of encountering neutron beam radiation produced errors during the execution of a kernel on a GPU. More concretely, the study aimed on analyzing how this radiation beam would affect the final output of the application running when errors are appearing and which is the impact on the final output of the application, generating one of the following 3 outcomes: a) correct output (masked error), b) wrong output (silent data corruption) and c) critical error that makes the machine halting (Detected Unrecoverable Error).

The authors investigate this issue using two distinct methodologies: the first one involves employing a fault injection framework to simulate radiation generated errors during kernel execution, while the other one entails deploying a GPU server that is exposed to a radiation machine during kernel execution in order to obtain real life values for this radiation impact on GPU's reliability.

In order to do this, they studied the error propagation caused by radiation in multiple well-known benchmarks using different optimization flags. To achieve this, two different methodologies were used.

The fault injection experiments involved using Nvidia's NVBitFI tool [12] to inject bit flip errors into the GPU application, in order to observe how these errors affected the application's execution depending randomly on which instruction they were injected. Specifically, this examined how many functional units (FUs) were used by the instruction, how a random error affected the output, and how different optimization flags generated varying levels of error criticality. These variations made it more likely for errors to manifest as Silent Data Corruption (SDC) or Detected Unrecoverable Errors (DUE) rather than being masked, depending on the optimization flag used or the number of FUs utilized by the GPU.

Once the impact of random bit flip on instructions was analyzed, they also evaluated it using a neutron ray beam machine that generates actual radiation-induced bit flips. This allowed to obtain the bit flip error rate in real-world scenario, by merging these results with the ones obtained previously. The paper concluded that less optimized code fail less than optimized code, but if we take into account the time spending calculating the results, optimized code tends to fail less per time unit.

With respect to the original paper, we reproduced the experiments on a Nvidia's GTX 1080 Ti card, which uses the Pascal microarchitecture as opposed to the the Kepler and Volta architecture used in [10]. Moreover, our card does not have ECC recovery errors, so we omitted the analysis of that part.

For obvious budgetary reasons, we were unable to replicate the beam experiments either, limiting our analysis exclusively to NVBitFI fault injection analysis.

2.2 Radiation Evaluation of Automotive GPUs

In addition to high performance desktop and server GPUs, a series of works from Iván Fernandez et al. has studied the radiation performance of embedded GPUs, particularly targeting the automotive domain, such as NVIDIA Xavier and NVIDIA Orin.

Given the safety critical nature of the automotive sector, automotive products require compliance with high quality manufacturing standards such as AEC-Q100 as well as functional safety standards such as ISO 26262.

Both NVIDIA Xavier and NVIDIA Orin have been certified for ISO 26262 safety standards from TÜV SÜD for the highest automotive assurance level (ASIL D).

In order to achieve these certification, these architectures include several reliability features such as ECC protection not only in the DRAM, as it is the case of GPUs targeting the supercomputer market, but also in almost any hardware structure, e.g. in the caches, communication interfaces etc.

In [9] the authors have evaluated the NVIDIA Xavier under a proton beam, using the matrix multiplication benchmark from the GPU4S Bench [6] / OBPMark Kernels [11, 3] Benchmarking suite. Consistently with the safety oriented design of the NVIDIA Xavier, the authors were not able to identify any wrong output during the irradiation experiments. Therefore, the errors either were corrected by the hardware or the resulted in DUE which caused a system restart.

By exploiting the RAS (reliability and serviceability) feature of the System-on-Chip (SoC) they were able to identify for the first time in the literature the source of DUEs in the radiation testing of such complex architectures, which in this case was the tags of the cache, which were not protected with ECC.

The authors made similar observations in [8], in which the same experiments were performed in the NVIDIA Orin. Given the introduction of ECC in the tag array of the Orin, its reliability was higher.

The main difference between [10] and [9, 8] is that the first one focused on the evaluation on high performance, less reliable GPUs, using a software solution. On the other hand, [9, 8] focused on highly reliable automotive products, in which the hardware provides protection.

2.3 TASA

Kosmidis et al. [5] introduced TASA, which stands for "Toolchain-Agnostic Static Software Randomization". TASA is a source-code level static software randomization tool. This means that it works at the application source code level. It adds a random padding among memory objects and reorders them from run to run resulting in a randomized memory layout and subsequently random execution time. The original purpose of TASA was to enable a measurement based method for assessing the Worst Case Execution Time (WCET)[13] of programs for Critical Real Time Embedded Systems called Measurement Based Probabilistic Timing Analysis (MBPTA) [2].

TASA was first prototyped in a custom compiler parser as a proof of concept [5]. Later, it was re-implemented in the CIL framework[7] adding support for CUDA, but only for its C subset. Currently, TASA is being re-implemented from scratch in the Clang compiler framework within the European Space Agency (ESA) funded project "Open source software randomization framework for probabilistic WCET prediction and security on (multicore) CPUs, GPUs and Accelerators" [4].

In order to randomomize the placement of the program memory objects, instead of doing it in a low level randomization within the compiler banckend, TASA performs the randomization at source code level, by randomizing the placement of their declarations.

This is because, as we know, executables are stored in ELF (executable and linkable) format files. These files distribute code and data declarations across different program sections, which are mapped to the process memory when it is executed. This means that, by grouping these high-level declarations in the source code and distributing them based on their section in the ELF file, randomizing their order within the same section would introduce different memory placements depending on the permutations generated for each section. The reason for this is that as observed by [5], compilers place by default the program elements in the order of their declaration.

In this work in progress, we extended TASA to work with CUDA code instead of C which is supported by [5] and we support at least the constructs found in the benchmarks we use. Then we applied software randomization to the original source files in order to randomize their memory layout and be able to assess the impact of randomization on the program reliability.

3 Reliability Metrics And Evaluation Methodology

In this Section, we outline the evaluation methodology we employed, as well as the tools utilized throughout the process. As already mentioned, our primary objective was to replicate a subset of the data collected from the compiler's optimization reliability on radiation study [10] and compare it with the new data obtained from the execution of the randomization framework. The motivation for reproducing prior data stems from the use of a different GPU, which is discussed in the following section.

3.1 Devices

To conduct these experiments, we used an Nvidia GTX 1080Ti GPU, which is a desktop/server GPU. As a result, this device does not have support for ECC. Moreover, this GPU has a Pascal GPU microarchitecture and compute capability 6.1.

Moreover, as explained in Section 2.2, automotive grade GPUs like NVIDIA Xavier and Orin could not be used, since they don't exhibit the same behavior thanks to their hardware protection.

3.2 Benchmarks, compilers and flags

The GPU applications utilized for testing are a reduced subset from the original paper, due to the massive number of executions required. This decision is primarily influenced by time constraints affecting our ability to execute all benchmarks. We have chosen to evaluate the following compiler optimization flags: -O0, -O1, -O3 and -use_fast_math.

The compiler version we used is NVCC 11.3, which is the same with [10].

The selected benchmarks for execution are BFS from the Rodinia Benchmarking suite [1] and GEMM from the CUDA Toolkit. This choice is based on their fundamentally different behaviors, which allows for a more insightful comparison. One kernel is primarily memory access-bound, while the other is compute-bound, making their contrasting characteristics particularly interesting for analysis.

3.3 Fault Injection Framework

For the fault injection framework, we utilize the same one employed in [10]: NVBitFI. This framework allowed us to gather data regarding execution performance, including instances of successful execution, situations where errors occurred but the execution completed without

terminating, and cases where the kernel encountered an exception and was subsequently terminated by the system. Additionally, this framework enables us to ascertain the failure rate associated with different types of instructions.

3.4 Randomization Framework

As previously discussed, we utilized a software randomization tool to randomize the memory layout. Specifically, we employed TASA [5], which has been previously demonstrated mainly for CPU code and with a GPU application [7]. We used the Clang TASA implementation which is under development in an ESA-funded project [4]. This tool supports C and C++ code, and part of our current work involved adding support for CUDA code. It is worth noting that our current TASA Clang support for CUDA is still work in progress. As such, it is not a complete implementation, but rather an update to enable the execution of the aforementioned programs associated with the experiments. Our objective is to determine whether generating these software randomized binaries yields different results compared to those compiled using only nvcc.

3.5 Methodology Challenges

During the experimentation process, we realized that due to the timing constraints related to the deadline of the HIPEAC challenge's student competition, we could not conduct a more exhaustive experiment of TASA in the context of reliability. This limitation was primarily because the use of NVBitFI introduces a significant execution time overhead for obtaining results, which varied between 2-5 seconds per iteration. While this overhead might be considered acceptable for a 1000-iteration analysis of the original benchmarks from [10], it becomes unacceptable for TASA. Given that we applied 100 different randomizations (which is not a high number of randomizations) and 100 iterations per randomization, analyzing the 7 flags of the original study with 7 benchmarks would result in a computation time much longer than the 3 months of the Student Challenge duration. Nonetheless, we have approached this study as a demonstration of the potential for a more comprehensive analysis that could cover the entirety of the original study. Still, the results obtained meet our expectations and allow us to remain optimistic about a larger-scale study.

3.6 Methodology

In the study, two distinct experiments have been carried out. First, we decided to check whether the error rate with the error injection of NVBitFI from the original paper was correctly replicated in our architecture. To do this, we performed an error injection identical to [10], with 1000 errors per iteration and 1000 iterations. The final results show the average of these 1000 iterations.

As for TASA, with the objective described earlier of presenting the results on time, we performed 100 randomizations on the original source code, and an error injection identical to the one in the original paper, but limited to 100 iterations per randomized source code. Note that performing the original 1000 iterations would increase the total time by a factor of 10, taking 2-3 days per benchmark and flag. Once these results were obtained, we observed the average error rate of each of the TASA randomizations compared to the original source code, as well as the average MWBF compared to the original code.

The compiler version has been the same as the original paper NVCC 11.3 and we generated code for the SM 6.1 compute capability of our GPU.

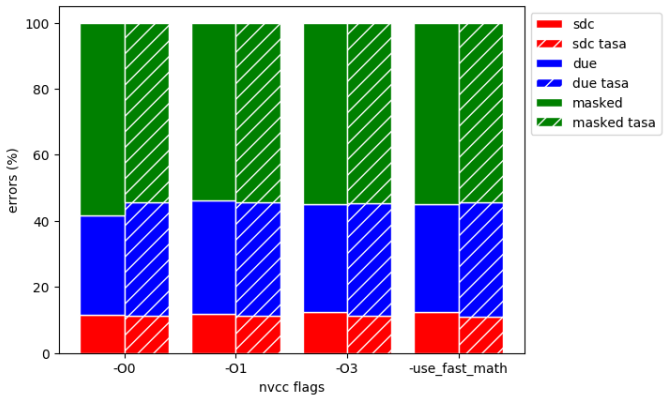

Figure 1 Plot comparing masked, SDC and DUE errors between regularly compiled code and TASA code for BFS using different compiler flags.

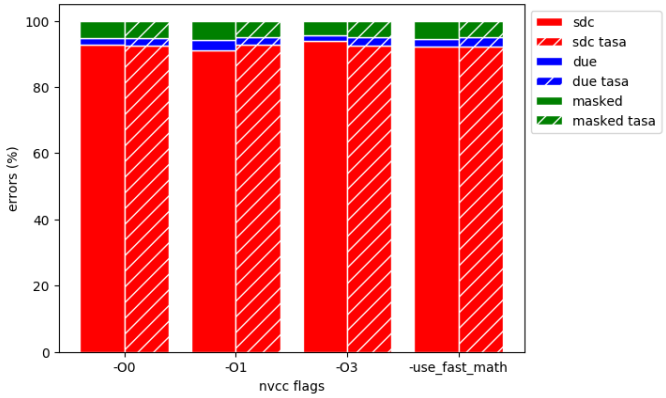

Figure 2 Plot comparing masked, SDC and DUE errors between regularly compiled code and TASA code for GEMM using different compiler flags.

4 Reliability results

Our reproduced results, shown in the left part of Figures **1** and **2** do not differ to much with the presented data from the original paper [10]. The results share a similar pattern and the slight difference could be driven by the use of the different GPU we used.

As illustrated in Figures **1** and **2**, the error patterns for BFS and GEMM differ significantly. GEMM, which is more computationally intensive, tends to experience a higher incidence of Silent Data Corruption (SDC) errors, while the occurrence of Detected Unrecoverable Errors (DUE) is comparatively lower. Conversely, BFS is characterized by a greater number of DUE failures, a trend that may be attributed to its memory-bound nature.

This difference can also be observed in Figures **3** and **4**, where the MWBF metric is presented (higher is better). In the memory bound benchmark (BFS), in which most of the time is spent in memory transfers, the -O0 flag provides better results, because of the redundant memory accesses of the non optimized code. On the other hand, in the computationally intensive benchmark (GEMM), the highest optimization level (-O3), achieves the best performance, therefore increasing the amount of work performed without an error per time unit.

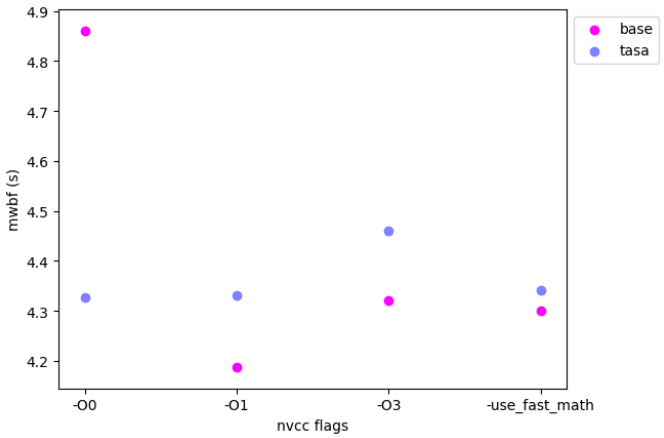

Figure 3 Difference in MWBF between regularly compiled code and TASA code for BFS using different compiler flags.

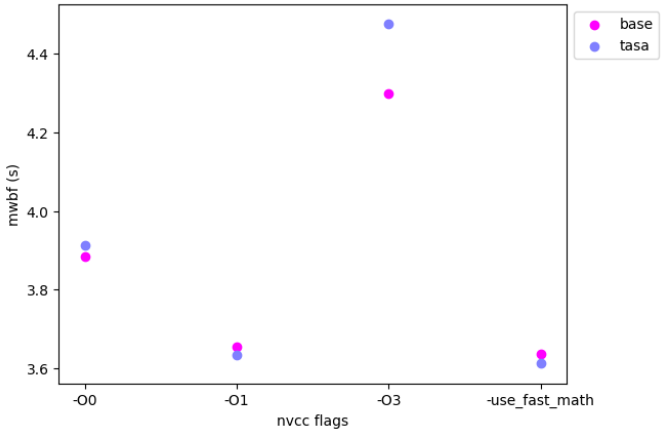

Figure 4 Difference in MWBF between regularly compiled code and TASA code for GEMM using different compiler flags.

5 Evaluation of TASA results

Figures **1** and **2** show the TASA results on the right bar of each compiler flag. Software randomized code experiences slightly less SDCs and a higher percentage of DUEs. A possible reason for this is that software randomization can expose errors that due to the memory layout were masked.

Moreover, as it can be seen in the Figures **3** and **4**, the mean of the TASA results could improve the default result. We observe that the obtained results with different flags are correlated with the ones obtained with TASA. An interesting observation is that TASA executions have better or similar results for MWBF in some configurations. This could be due to the fact that, within the normal distribution of executions produced by TASA, the average execution time is lower than that of the original source code. An increase in the number of randomizations in TASA would be necessary to justify this.

6 Conclusion

In this paper, we partially reproduced the results of [10] in a different GPU architecture, and we observed the same trends, i.e. that the -O3 compiler flag increases the MWBF metric, that is the useful work performed on average before an fault occurs.

Moreover, by introducing software randomization, our preliminary results show that the same trend is observed, and in several cases the MWBF is improved.

7 Future Work

We have to highlight that our conclusions are not decisive, as in this time-limited exercise performed in the HiPEAC Student Challenge 2025 we lacked the resources to conduct an extensive analysis with a larger dataset. As previously mentioned, we have only executed a total of 100 different seeds of the same program. This sample size may not be sufficient to assure the validity of the presented results. Also the original paper uses bigger code samples. The choice of BFS and GEMM is justified for their different behavior. Despite this, the results are pretty optimistic, so as this previous analysis has given us a good tendency, we will increase the total amount of randomizations of the original source code and the iterations per each one, and make an exhaustive analysis including all flags and benchmarks.

References

1 Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A Benchmark Suite for Heterogeneous Computing. In *2009 IEEE International Symposium on Workload Characterization (IISWC)*, pages 44–54, 2009. doi:10.1109/IISWC.2009.5306797.

2 Liliana Cucu-Grosjean, Luca Santinelli, Michael Houston, Code Lo, Tullio Vardanega, Leonidas Kosmidis, Jaume Abella, Enrico Mezzetti, Eduardo Quiñones, and Francisco J. Cazorla. Measurement-Based Probabilistic Timing Analysis for Multi-path Programs. In Robert Davis, editor, *24th Euromicro Conference on Real-Time Systems, ECRTS 2012, Pisa, Italy, July 11-13, 2012*, pages 91–101. IEEE Computer Society, 2012. doi:10.1109/ECRTS.2012.31.

3 David Steenari et al. On-Board Processing Benchmarks, 2021. http://obpmark.github.io/.

4 Leonidas Kosmidis, Matina Maria Trompouki, Pau Lopez Castillon, Eric Rufart Blasco, Javier Fernandez Salgado, and Andreas Jung. Open Source Software Randomisation Framework for Probabilistic WCET Prediction on Multicore CPUs, GPUs and Accelerators. In *Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*, Lecture Notes in Computer Science. Springer, 2024.

5 Leonidas Kosmidis, Roberto Vargas, David Morales, Eduardo Quiñones, Jaume Abella, and Francisco J. Cazorla. TASA: Toolchain-Agnostic Static Software Randomisation for Critical Real-time Systems. In Frank Liu, editor, *Proceedings of the 35th International Conference on Computer-Aided Design, ICCAD 2016, Austin, TX, USA, November 7-10, 2016*, page 59. ACM, 2016. doi:10.1145/2966986.2967078.

6 Ivan Rodriguez, Leonidas Kosmidis, Jerome Lachaize, Olivier Notebaert, and David Steenari. GPU4S Bench: Design and Implementation of an Open GPU Benchmarking Suite for Space On-board Processing. Technical Report UPC-DAC-RR-CAP-2019-1, Universitat Politècnica de Catalunya, 2019. URL: https://www.ac.upc.edu/app/research-reports/public/html/research_center_index-CAP-2019,en.html.

7 Ivan Rodriguez Ferrandez, Alvaro Jover Alvarez, Matina Maria Trompouki, Leonidas Kosmidis, and Francisco J. Cazorla. Worst Case Execution Time and Power Estimation of Multicore and GPU Software: A Pedestrian Detection Use Case. *Ada Lett.*, 43(1):111–117, October 2023. doi:10.1145/3631483.3631502.

8 Ivan Rodriguez-Ferrandez, Leonidas Kosmidis, Maris Tali, David Steenari, Alex Hands, and Camille Bélanger-Champagne. Proton Evaluation of Single Event Effects in the NVIDIA GPU Orin SoM: Understanding Radiation Vulnerabilities Beyond the SoC. In *30th IEEE International Symposium on On-Line Testing and Robust System Design, IOLTS 2024, Rennes, France, July 3-5, 2024*, pages 1–7. IEEE, 2024. doi:10.1109/IOLTS60994.2024.10616076.

9 Ivan Rodriguez-Ferrandez, Maris Tali, Leonidas Kosmidis, Marta Rovituso, and David Steenari. Sources of Single Event Effects in the NVIDIA Xavier SoC Family under Proton Irradiation. In Alessandro Savino, Paolo Rech, Stefano Di Carlo, and Dimitris Gizopoulos, editors, *28th IEEE International Symposium on On-Line Testing and Robust System Design, IOLTS 2022, Torino, Italy, September 12-14, 2022*, pages 1–7. IEEE, 2022. doi:10.1109/IOLTS56730.2022.9897236.

10 Fernando Fernandes Dos Santos, Luigi Carro, Flavio Vella, and Paolo Rech. Assessing the Impact of Compiler Optimizations on GPUs Reliability. *ACM Trans. Archit. Code Optim.*, 21(2), February 2024. doi:10.1145/3638249.

11 David Steenari, Leonidas Kosmidis, Ivan Rodríguez-Ferrández, Álvaro Jover-Álvarez, and Kyra Förster. OBPMark (On-Board Processing Benchmarks) - Open Source Computational Performance Benchmarks for Space Applications. In *2nd European Workshop on On-Board Data Processing (OBDP)*, 2021. doi:10.5281/zenodo.5638577.

12 Timothy Tsai, Siva Kumar Sastry Hari, Michael Sullivan, Oreste Villa, and Stephen W. Keckler. NVBitFI: Dynamic Fault Injection for GPUs. In *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 284–291, 2021. doi:10.1109/DSN48987.2021.00041.

13 Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David B. Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter P. Puschner, Jan Staschulat, and Per Stenström. The Worst-case Execution-time Problem - Overview of Methods and Survey of Tools. *ACM Trans. Embed. Comput. Syst.*, 7(3):36:1–36:53, 2008. doi:10.1145/1347375.1347389.