Assessing the Impact of Compiler Optimizations on GPUs Reliability

Fernando Fernandes dos Santos, Luigi Carro, Flavio Vella, Paolo Rech

Assessing the Impact of Compiler Optimizations on GPUs Reliability

FERNANDO FERNANDES DOS SANTOS*, Univ Rennes, INRIA, Rennes, France
LUIGI CARRO, Institute of Informatics, Federal University of Rio Grande do Sul, Brazil
FLAVIO VELLA, University of Trento, Italy
PAOLO RECH, University of Trento, Italy

Graphics Processing Units (GPUs) compilers have evolved in order to support general-purpose programming languages for multiple architectures. NVIDIA CUDA Compiler (NVCC) has many compilation levels before generating the machine code and applies complex optimizations to improve performance. These optimizations modify how the software is mapped in the underlying hardware; thus, as we show in this paper, they can also affect GPU reliability. We evaluate the effects on the GPU error rate of the optimization flags applied at the NVCC Parallel Thread Execution (PTX) compiling phase by analyzing two NVIDIA GPU architectures (Kepler and Volta) and two compiler versions (NVCC 10.2 and 11.3). We compare and combine fault propagation analysis based on software fault injection, hardware utilization distribution obtained with application-level profiling, and machine instructions radiation-induced error rate measured with beam experiments. We consider eight different workloads and 144 combinations of compilation flags, and we show that optimizations can impact the GPUs' error rate of up to an order of magnitude. Additionally, through accelerated neutron beam experiments on a NVIDIA Kepler GPU, we show that the error rate of the unoptimized GEMM (-O0 flag) is lower than the optimized GEMM's (-O3 flag) error rate. When the performance is evaluated together with the error rate, we show that the most optimized versions (-O1 and -O3) always produce a higher amount of correct data than the unoptimized code (-O0).

CCS Concepts: • **Computer systems organization** → **Reliability**; • **Hardware** → *Fault models and test metrics.*

Additional Key Words and Phrases: Graphics Processing Units, reliability, neutron-induced errors, error rate, reliability

ACM Reference Format:
Fernando Fernandes dos Santos, Luigi Carro, Flavio Vella, and Paolo Rech. 2024. Assessing the Impact of Compiler Optimizations on GPUs Reliability. *ACM Trans. Arch. Code Optim.* XX, XX (January 2024), 22 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Thanks to the high performance, flop/watt efficiency, and support for general-purpose languages, GPUs are today adopted in several domains, including High-Performance Computing (HPC) [38], and to accelerate emerging workloads such as Deep Learning (DL) [28] and High-Performance Data

Authors' addresses: Fernando Fernandes dos Santos, fernando.fernandes-dos-santos@inria.fr, Univ Rennes, INRIA, Rennes, 263 Av. Général Leclerc, Rennes, Brittany, France, 35000; Luigi Carro, carro@inf.ufrgs.br, Institute of Informatics, Federal University of Rio Grande do Sul, Bento Gonçalves avenue 9500, Porto Alegre, Brazil; Flavio Vella, flavio.vella@unitn.it, University of Trento, Trento, Italy; Paolo Rech, paolo.rech@unitn.it, University of Trento, Via Calepina, 14, Trento, Trentino, Italy.

1544-3566/2024/01-ART $15.00
https://doi.org/XXXXXXX.XXXXXXX

Analytics [5]. The most recent GPU architecture advances, such as tensor core, Multi-Instance GPU (MIG), which allows multiple users to access the same GPU, and mixed-precision functional units, move toward improving HPC performance and software flexibility in Deep Learning applications. In this context, the evolution and the enhancements of the GPU compiler allow tailoring the code to the new features, transparently improving the performance.

The compilation process maps the complex source code in the underlying and even more complex hardware, and it is then likely to impact not only the performance but also the code's reliability. This latter aspect is crucial for many applications in which GPUs operate, such as autonomous driving and HPC. The main scope of this paper is to quantify and understand the impact of compiler optimizations on the error rate of codes executed on GPUs. Unfortunately, modern GPUs have been demonstrated to be susceptible to transient faults [3, 24, 46, 49, 51].

In this paper, we specifically address the effect of compiler version and compiler optimizations on GPUs reliability. Developers that aim at using compiler optimizations for safety-critical systems must ensure that the compiled machine code is still sufficiently reliable. In fact, the compiler changes the way software is mapped in the available hardware resources with a potentially significant impact on the device error rate. Since the fault is generated at the hardware level, the error manifests at the software level, and the compiler maps the latter in the former, an accurate reliability analysis must consider the whole abstraction stack. Hardware reliability is normally evaluated with beam experiments [20, 48] while software error propagation is studied with fault injection [39, 52]. Unfortunately, due to high cost, evaluating all optimization combinations for a code with beam experiments would be unfeasible, and fault injection is commonly performed at an abstraction level that is too high for considering hardware utilization, thus risking to generate an unrealistic result.

To accurately understand the impact of the NVIDIA compilers on GPUs reliability, considering both the different hardware units sensitivities and the software propagation, we adopt a cross-layers analysis combining (1) beam experiments on specific functional units that provide the probability for a fault to be generated, (2) application profiling, that identifies the distribution of resources used for computation, and (3) software fault injection, that measures the probability for the error to propagate to the output. We present an ablation study to identify why and how different compiler configurations modify the code error rate. We start with random software fault injection, which considers neither the machine instructions distribution nor the functional unit hardware sensitivity. Our results show that software fault injection measures a difference between compiler optimizations of, on average, 15%, indicating that the code modifications induced by the compiler have a negligible impact on the code reliability. Then, to consider the compiler modifications to the machine code, we normalize the injection based on the machine instructions distribution. When considering hardware sensitivity, the predicted error rates between compiler optimizations can differ by more than 1,000%. Most of the impact of the compiler on the GPU reliability can be attributed to the different sensitivity of the hardware functional units selected for computation by the flags more than to the different distribution of machine instructions. This behavior can only be observed by considering the hardware error rate during analysis.

To have a broad evaluation, we evaluate the reliability of several compiler optimizations on eight representative codes from different benchmark suites and application domains. We evaluate the codes' reliability when compiled with two versions of the NVIDIA CUDA Compiler (NVCC), i.e., 10.2 and 11.3. For each compiler version, we consider nine unique NVCC flag configurations, totaling 144 configurations per architecture. The compiler maps the code in the underlying hardware depending on the available computing resources. Since each new GPU generation has novel features and disposes of additional resources, we consider two GPU architectures (Kepler and Volta) to understand if there is a compiler optimization reliability dependence on the hardware. As we show, the GPU architecture plays a significant role in performance and reliability. Volta architecture

significantly improves computing power and efficiency over Kepler (5.5 GFLOPS/watt for Kepler and 24 GFLOPS/watt for Volta), with tensor cores for deep learning, support for half-precision floats, and higher double-precision computing.

We found that, in general, higher optimizations have the drawback of increasing the code error rate. Nonetheless, as a very promising result, we found that the performance gain brought by the compiler optimizations is much higher than the increase they impose on the error rate. In other words, the amount of work that the optimized code can correctly produce before experiencing a failure is more than double the unoptimized code. In all applications in which the amount of correctly produced data is the main concearn, then, an optimized code is to be prefered.

All data extracted for our evaluation, including the normalized instruction error rate and performance-related metrics, are available in an online repository for reproducibility[1].

The remainder of the paper is structured as follows. In the next section, we present the past works related to our research. The experiments and the tools that we used are described in Section 3. The performance impact of each configuration is presented in Section 4. Section 5 presents the fault injection results, as well the evaluation of the optimizations using the SDC probability and SDC rate prediction. Section 6 shows the beam experiments results, and Section 7 concludes de paper.

2 BACKGROUND AND RELATED WORKS

This section presents the background and related works on radiation effects on GPUs. We highlight the strengths and limitations of the available reliability evaluation strategies for GPUs.

2.1 Radiation Effect on GPUs

The natural flux of high-energy neutrons at sea level is about 13 $neutrons/((cm^2) \times h)$ [26]. A terrestrial neutron strike may perturb a transistor's state, generating bit-flips in memory or current spikes in logic circuits that, if latched, lead to an error [33]. A transient fault that propagates from the hardware to the software level can lead to the following outcomes: (1) **Masked**: no effect on the program output. The corrupted data is not used, or the circuit functionality is not affected. (2) **Silent Data Corruption (SDC)**: undetected output corruption, that is, the application finishes, but the output is not correct. (3) **Detected Unrecoverable Error (DUE)**: program or system crash. Since GPUs execute several processes in parallel, corruption in the scheduler or a single error in shared resources affects various output elements [3, 20, 22, 29, 46, 51]. The error rate of a code executed on a device depends on various factors: (i) the sensitivity of the hardware, i.e., the probability for the impinging particle to modify the transistor's state, thus inducing a *fault*. This probability is normally measured experimentally with beam experiments. (ii) the probability for the transistor modification to be latched, modifying a software visible state, thus becoming an *error*. (iii) the probability for the error to propagate in the executed software and reach the output, inducing a *failure* (SDC or DUE). As we show in this paper, different compiler configurations modify how the functional units are used for computation (thus the hardware sensitivity) and the instruction distribution (thus the error propagation probability). An accurate reliability evaluation should then consider both aspects.

Lately, GPU vendors have been working on many architecture modifications to improve the GPUs' reliability while maintaining high performance [21, 22, 45, 55]. The researchers have proposed hardening at different levels of the GPU architecture, from the memory cell [45], to Error Correction Code (ECC) in the memories [21], Redundant Multithreading execution [55], and Algorithm-Based Fault Tolerance for deep learning applications [19, 22, 31, 34, 49].

[1]https://github.com/UFRGS-CAROL/taco_2023_data

2.2 Reliability Evaluation Methodologies

There are various reliability evaluation methodologies available, from beam experiments to fault injection, each operating at a different level of abstraction. We highlight the benefits and drawbacks of each methodology in the assessment of the compiler optimizations' effect on GPU reliability.

The GPU reliability has already been evaluated through **beam experiments** in previous works [3, 20, 24, 29, 44, 49, 51]. The accelerated particle beam induces transient faults in the device hardware, which can manifest as output errors. When a device is exposed to a neutron beam, the whole chip is irradiated: faults are not limited to a subset of resources. Consequently, beam experiments provide the realistic error rate of the device running a code. Since errors are observed only when they reach the output, with beam experiments, we cannot distinguish which level of the hardware or the software contributes to the device error rate, making it challenging to identify the most vulnerable parts of the system. Moreover, since there are many possible compiling options to consider, the cost of a broad evaluation of their effect on GPU reliability using beam experiments is prohibitive.

Fault injection measures the probability for a fault to propagate to the output generating a failure (Architectural, Program, or Software Vulnerability Factor, AVF, PVF, SVF [35, 41, 50]). Faults can be injected by the user at different levels of abstractions: from Register-Transfer Level (RTL) [12, 13, 25] to microarchitecture [9, 14] and software [1, 11, 15, 18, 39, 52, 56]. Fault injection assumes that the fault has occurred and tracks its propagation without giving any information about the probability of the fault originating. Fault injection has two main limitations: (1) the fault model and fault injection probabilities are defined/modeled by the user and/or the simulator, thus, the obtained results risk being unrealistic. (2) faults can be injected only in that subset of available and accessible resources.

The abstraction level at which the reliability evaluation is performed is particularly important when evaluating the impact of the compiler in the GPU reliability, as we do in this paper. In fact, the compiler can modify both the machine instructions distribution and the functional units used for computation. Injecting faults in the source code might then not be sufficient to fully understand the effect of the compiler in the code reliability. In an effort to improve the error rate estimation using fault injection, the metric *SDC probability* was introduced [1, 16, 23, 30, 39, 57]. The SDC probability is the PVF normalized by the instruction probability to be sampled from the code's instructions. In other words, the SDC probability combines the PVF with the instruction profiling of the application and can, in principle, evaluate the impact of the machine instruction distribution modification imposed by the compiler. However, since a fault is considered to be equally probable in each instruction, the SDC probability still lacks information on the hardware sensitivity. The probability for an instruction to be corrupted should actually be correlated with the hardware functional units it uses. As we show, this limitation makes the SDC probability insufficient to evaluate the effect of compiler (or other source code modifications) on the device's reliability.

Other works have proposed to combine the fault propagation probability (i.e., PVF) with the hardware fault probability (i.e., functional units error rate) to accurately estimate the error rate of an application running in a device [8, 9, 43, 44, 48]. This multi-level evaluation considers both the hardware fault probability (with beam experiments) and the software error propagation (with fault injection). While the combination of beam experiments and fault injection has been shown to provide insights into device reliability and fault propagation, this methodology has not yet been adopted to understand the impact of compiler optimizations on the code's reliability.

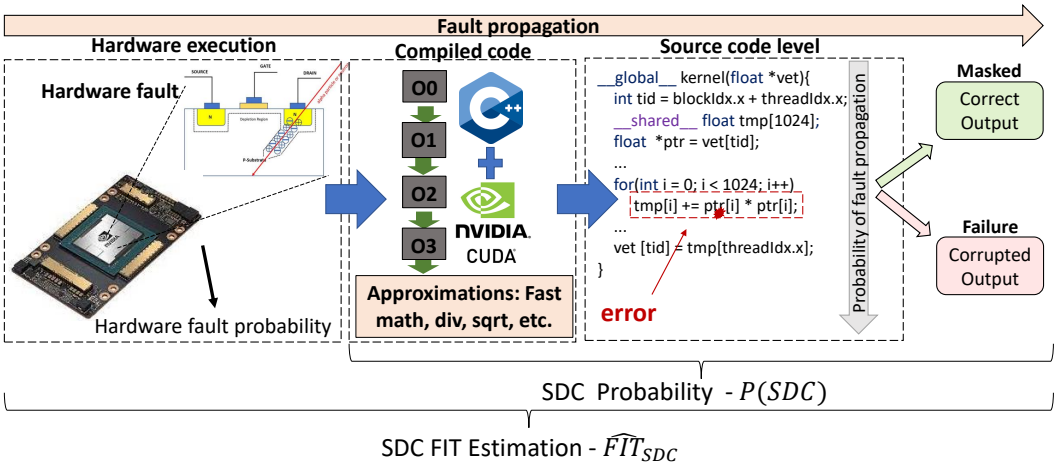


Fig. 1. The CUDA application compiling is performed by NVIDIA CUDA Compiler (NVCC). NVCC translates the high-level C++ code to Shader Assembly (SASS) code to run in a GPU device. In the compilation phases, NVCC also applies a range of optimizations that can be disabled or enabled by the user. Two primary error analyses are considered to evaluate the error propagation: SDC Probability and SDC FIT Estimation. While SDC Probability considers only the software level SASS fault injection and kernel profiling, the SDC FIT Estimation also considers the probability of the fault to be generated in the hardware (instruction FIT rate).

2.3 Main Idea, Contributions, and Limitations

Our idea is to investigate the impact of compiler versions and optimizations on the reliability of GPUs. To do so, we perform the reliability evaluation at various levels of abstraction (purely software, SDC probability, hardware-software combination, and beam experiments).

Figure 1 shows the fault propagation, from the generation at the hardware level to the effect at the application level. The fault originates at **hardware level** with a probability expressed as cross-section or Failure In Time. The hardware fault probability depends on the layout and technology of the transistors, circuit size, operation frequency, and memory cell design [4]. The fault then has a probability of propagating at **source code level** that is usually measured through fault injection.

The software source code must be compiled to be deployed and executed in the underlying hardware. The compilation process often consists of multiple optimization steps that can change both the hardware resources used to perform computation (i.e., the hardware fault probability) and the fault propagation in the software (Program Vulnerability Factor (PVF)). A source code compiled with different optimization flags, then, is likely to have both different hardware fault probability and different fault propagation models [2].

Recent studies have assessed the impact of compiler optimizations on the reliability of machine codes [2, 40, 42]. These studies indicate that achieving optimal performance is desirable, as it leads to more executions before failure. However, they often focus either on fault injection or beam experiments. We argue that incorporating both hardware and software fault probabilities is necessary for realistic evaluation. For instance, consider an algorithm performing a float MUL and ADD operations sequence in a loop. The fault propagation may not change, or slightly change, at the source code level. Nevertheless, optimizations such as loop unrolling and fusing MUL and ADD into a single MAC (multiply and accumulate) can improve performance and change the generated machine code. As a result, the hardware fault probability will change, ultimately impacting the final code's FIT rate.

To understand the compiler impact on the reliability of GPUs, we separate the contribution of the fault propagation models and the hardware sensitivity. First, we inject faults randomly in the machine instructions. Then, we consider the SDC Probability that normalizes the fault injection result with the probability for the machine instruction to be executed [1, 30]. Finally, we also include the raw sensitivity of the hardware functional units, gathered from beam experiments, introducing the **SDC FIT estimation**. As we will show, most of the compiler optimization impact relies on the hardware sensitivity, while the instruction distribution has a minor effect.

The accuracy of the SDC FIT prediction we introduce is limited by the complexity of measuring the error rate of each GPU microinstruction. NVIDIA Instruction Set Architecture (ISA) contains up to 100 different microinstructions. Measuring the error rate of each microinstruction would allow detailed and accurate analysis but is, unfortunately, unfeasible due to beam time limitations. What we propose is to group microinstructions and measure, with beam experiments, the error rate of only the most used and critical sets. To the best of our knowledge, this is the first paper that (1) proposes a multi-level analysis for the estimation of the SDC rate for GPUs, combining application profile, fault injection, and beam experiments to deeply understand the reliability of GPUs and (2) uses the SDC probability and the error rate estimation to evaluate the impact of the compiler optimizations on the final machine code for GPUs.

It is worth mentioning that the NVCC compiler, although proprietary, restricts a thorough reliability analysis of the machine binary executed on the hardware. However, even without full access to the compiler, we can still perform effective evaluations by using instruction profiling tools and the assembly (SASS) code generated through NVIDIA cuobjdump. This method helps us calculate the probability of faults in each microinstruction by combining instruction fault injection with the instruction error rate. Then, we can establish correlations with the code error rate, providing valuable insights and enabling comprehensive assessments.

3 RELIABILITY METRICS AND EVALUATION METHODOLOGY

This section describes the devices and codes we characterize, the compilers and optimization flags we use, the reliability evaluation metrics we adopt, and how they are measured on GPUs.

3.1 Devices

We consider two NVIDIA GPU micro-architectures, Kepler (Tesla K40c) and Volta (Titan V). These architectures cover almost six years of architecture upgrades, with significant advances in terms of performance, efficiency, and compiler optimizations. The additional and improved resources available in Volta will likely modify how the optimization flags impact the code performance and reliability. As part of our contribution, we aim to understand if the improved architecture impacts the reliability of compilers.

Tesla K40 is built with the Kepler architecture and fabricated in a 28*nm* TSMC standard CMOS technology. Error Correcting Code (ECC) protects the register file, shared memory, and caches, while read-only data cache is parity protected. We test K40 both with the ECC enabled and disabled for the radiation experiments to evaluate the impact of the ECC on the register file/caches in the error rate. It is worth noting that in our evaluation, we consider errors occurring in the GPU core, not in the main memory. That is, we do not inject faults in the main memory.

Titan V is designed with the Volta micro-architecture and built with TSMC FinFET 12nm. Volta GPUs feature hardware acceleration for three IEEE754 float point precisions: double, float, and half. The Titan V has 80 SMs, where each SM has 64 FP32 cores, 64 INT32 cores, and 32 FP64 cores. Titan V also has tensor cores that support FP16 operations for applications that require a small range of float values (e.g., deep neural networks).

Table 1. Codes used for reliability evaluation and their main characteristics.

	Short name	Data type	Bound	Domain	Suite
Breadth-First Search	BFS	INT32	Memory	Graph	Rodinia[10]
Lava MD	LVA	FP32	Compute	Molecular Dynamics	
Hotspot	HST	FP32	Compute	Structured Grid	
Gaussian elimination	GSS	FP32	Compute	Linear Algebra	
LU Decomposition	LUD	FP32	Compute	Linear Algebra	
CFD Solver	CFD	FP32	Compute	Fluid Dynamics	
General Matrix Multiplication	GEMM	FP32	Compute	Linear Algebra	CUDA Toolkit
Merge Sort	MST	INT32	Memory	Sorting	

3.2 Codes, compilers, and optimization flags

Table 1 lists the codes used in our reliability evaluation. Each benchmark is tested in the two architectures, Kepler and Volta. We have selected ten benchmarks from different domains to increase the statistical significance of this work. We have built each code from Table 1 with two versions of NVIDIA CUDA Compiler (NVCC), NVCC 10.2 and 11.3. The select versions of NVCC are the latest ones that simultaneously support Kepler and Volta architectures. Each major update on NVCC introduces support for newer architectures and compiler optimizations, which can be beneficial or not to reliability.

Optimization flags: We compile all the GPU kernels with multiple NVCC compilation flags to measure the impact of each optimization on the final Source and Assembly[2] (SASS) code. We select the flags that generate a SASS code that differs from the default NVCC optimization (O3). It is expected that two identical codes will have the exact fault propagation probabilities and the same FIT rate. We evaluated the following optimization flags:

(1) **O0, O1, and O3:** The optimization levels available on NVCC for the tested benchmarks. Although O2 is supported for NVCC, the O2 option generates the same code as O3 for the tested applications. All flags, but O0 and O1, are related to float approximations or register file usage. The approximation flags are tested on top of the default NVCC configuration (03).
(2) **FTZ-ON:** The *–ftz=true* flag on NVCC flushes *subnormal* numbers (i.e., values smaller than the smallest possible value) to zero. Faults that generate subnormal numbers will probably be masked with FTZ-ON.
(3) **FMAD-OFF:** The default NVCC action is to contract the multiply and accumulate instructions into HFMA, FFMA, or DFMA. To disable the contraction of the multiply and accumulate instructions, we have to pass *–fmad=false* as a parameter. The contracted instruction may generate a different result due to destructive cancellation, so we expect the impact of a fault will be different on contracted instructions.
(4) **MinRF:** Each thread on a CUDA kernel can use up to 255 registers, respecting the limits of the GPU occupation. With the flag *–maxrregcount=N*, we can set the maximum number of registers per thread, where N is the limit of registers per thread. We chose to set the number of registers per thread to the minimum for each architecture, 16 for Kepler, and 24 for Volta. Limiting the number of registers per thread can be used to increase the GPU

[2]Often denoted as Shader Assembly.

occupation. However, it also increases the register spill to the memory, which can decrease the performance.

(5) **PrecSqrt-OFF and PrecDiv-OFF:** The flag *–prec-sqrt=false* and *–prec-div=false* allow the NVCC to use a fast approximation for the square root and float divisions, respectively. Float approximations improve performance but may change the impact of the fault in the final result.

(6) **FAST-MATH:** Passing the flag *–use_fast_math* to NVCC enables all the fast approximations for arithmetic float operations. That is, all the flags *–ftz=true*, *–prec-div=false*, *–prec-sqrt=false*, *–fmad=true* are set in the compilation process.

Recent works have demonstrated that the effectiveness of a particular flag in enhancing code reliability depends on how it modifies the resources used for computation (more or larger resources imply a higher error rate) and how performance is improved (faster executions imply a higher amount of data produced before the fault happens) [40, 42]. To fully assess these two tradeoffs, in this paper, we correlate the impact on the reliability of different compiler compilation flags by using software fault injection and the hardware fault probability. In Section 4 we evaluate the impact of each optimization flag on the code size and execution time. Then, in Section 5, we evaluate how each optimization flag modifies how the code is mapped in the hardware (resource utilization) and the implications on the error rate. Finally, we also combine the impacts on error rate and performance in Section 5.3. Note that we perform characterization at the instruction level based on profiling. However, optimization flags used by the compiler can have effects beyond the instruction profile, requiring specific analysis to understand their impact on reliability.

3.3 Fault Injection Framework

We use the NVBitFI [52] fault injector to simulate faults in the software. NVBitFI injects transient errors in the GPU's Instruction Set Architecture (ISA) visible states, general-purpose registers, predicate registers, condition instructions, and arithmetic instructions. NVBitFI perfectly suits the proposed evaluation since it can instrument the kernels at the SASS level. Other fault injectors such as GPUQin, CAROL-FI, Kayotee, GPGPU-SIM, SIFI, GUFI [15, 27, 36, 53, 54], neither allow to inject faults at the SASS level nor offer support for Kepler and Volta architectures at the same time. We inject up to 1,000 (750 for applications that only have integer operations) single-bit flips faults per code on NVBitFI, for a total of more than 200,000 faults per ISA, ensuring 95% confidence intervals to be lower than 5% [52]. With NVBitFI, we measure the probability of a fault propagating in software leading to a failure (SDC or DUE), i.e., the Program Vulnerability Factor (PVF) [50]. The PVF identifies which instruction or resource, once corrupted, is more likely to affect the GPU computation.

3.4 Beam experiments setup for validation

Beam experiments are the most accurate method to evaluate and validate a device's reliability. The Failure In Time (FIT) can be measured on beam experiments by dividing the numbers of observed errors by the received particles ($neutrons/cm^2$) and multiplying by the terrestrial neutron flux. When multiplied by the terrestrial flux (13 $neutrons/(cm^2 \cdot h)$ at sea level), we estimate the realistic error rate, expressed in FIT, i.e., errors per 10^9 hours of operation.

Our experiments were performed at the ChipIR facility of the Rutherford Appleton Laboratory, UK. Figure 2 shows the setup mounted at ChipIR. The facility delivers a beam of neutrons with a spectrum of energies that resembles the atmospheric neutrons [7]. The available neutron flux was about $3.5\times10^6 n/(cm^2/s)$, ~8 orders of magnitude higher than the terrestrial flux (13 $neutrons/(cm^2 \cdot h)$ [26]).

Since the terrestrial neutron flux is low, in a realistic application, it is unlikely to see more than one error during a single program execution. Hence the experiments have been designed to maintain this property (observed error rates were lower than four errors per 1,000 executions). Experimental data can be scaled to the natural terrestrial environment without introducing artifacts.

The FIT rate is linearly dependent on the amount of resources used for computation, the probability of the fault to occur (the technology sensitivity), and the probability of a fault propagating to the computation output (PVF). The FIT rate does not depend on the execution time, though. For instance, if the same amount of memory is exposed for a time interval t or $2t$, its FIT rate will not change. In fact, in $2t$, we expect twice the errors and twice the neutrons (i.e., twice the fluence). Similarly, under the assumption that at most one fault can affect the GPU during code execution (because the natural flux is very low), executing x or $2x$ *sequential* MUL instructions does not change the probability of having one MUL corrupted by neutrons. However, what can change is the probability that an error in one of the MULs propagates to the output of the sequence of the operations (i.e., the PVF). If the additional x MULs are executed in *parallel* with the original sequence, the FIT rate is expected to double (same execution time, same fluence, but doubled the error rate). We use these premises to comment on how the performance can impact the error prediction on Section 5.

To evaluate also the performance impact of each optimization flag and its relation to the error rate, we use the Mean Work Between Failures (MWBF) [47]. The MWBF allows measuring the amount of useful work produced before the system experiences a failure. We calculate the MWBF for all configurations evaluate in the radiation experiments (See Section 6). As we show, the performance gain brought by the compiler optimization is higher than the increase in the error rate, resulting in a more reliable execution.

We select GEMM to evaluate the impact of the optimizations on the code error rate under a realistic environment. We could not test all the configurations from figures 3 and 4 due to beam time limitations. We evaluated the GEMM compiled with NVCC 10.2 and 11.3 with different levels of optimizations (O0, O1, O3, and MinRF, see Section 3.2). In fact, even for a simple code like GEMM, compiler optimizations can considerably impact the execution of the dynamic instructions. In the next section, we demonstrate this behavior by presenting the instruction profiling.

4 DYNAMIC INSTRUCTIONS PROFILING

This section presents the *dynamic instruction profiling* on the GPU codes, which is necessary to understand the compiler impact on the code's reliability. Dynamic instruction profiling extracts the code characteristics that can contribute to the final error rate. It has been shown that fault masking is directly related to the basic block organizations and instruction dependencies of the code [1]. Consequently, a code compiled with the O0 flag can mask more faults than a code compiled with O3, since the unoptimized code (O0) has more dead code and redundant instructions. A fault in a redundant instruction output will be masked [2]. However, the unoptimized code will take much longer to execute than the optimized one, being able to deliver a lower number of correct executions before experiencing a failure. Thus, it is necessary to consider both the code size and the execution time in the reliability analysis of different compiler optimizations.

For all the benchmarks listed in table 1 built with all the flags and compilers described in Section 3.2, we profile the GPU kernels using Nvprof (Kepler) and Nsight-Compute (Volta). Figures 3 and 4 show the code size and the execution time relative to the default NVCC compilation (O3). The code size is measured by counting the kernels' SASS lines compared with the default NVCC configuration. The O3 configuration is marked as a red line in the figure to represent the relative size for each configuration.

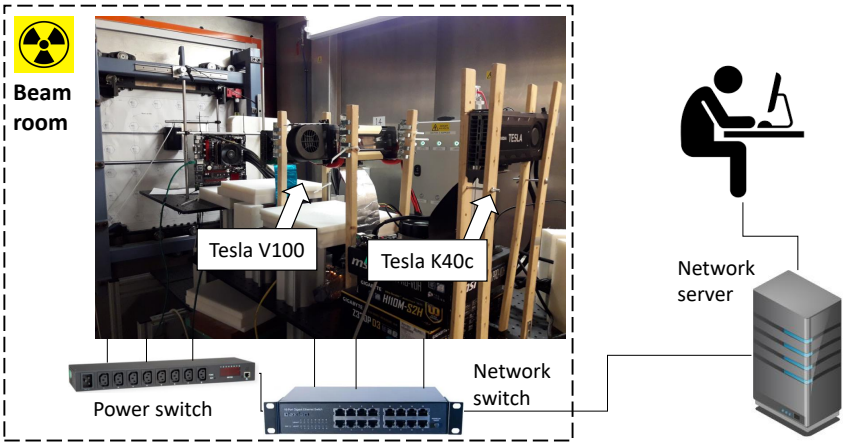


Fig. 2. Neutron beam experiments setup mounted at ChipIR. All devices are connected to a socket server by a network switch through an Ethernet connection. The socket server controls the experiments running on the beam. If the devices stop responding, the server will use the network-controlled power switch to power cycle the power supplies.

From Figure 3, we can derive some interesting trends. Reducing the hardware core iterations to execute a floating-point operation, e.g., using the *fast math* option, is a common type of approximation. While fast math generally reduces the accuracy of the result, it can reduce the code size and the execution time, as shown in Figures 3 and 4. From a reliability perspective, approximation improves performance and can thus reduce the error rate (smaller area and/or fewer operations executed). However, the impact of the fault in the approximate result can be higher as fewer bits are used to represent the output [17]. The probability for the fault to propagate (PVF) can then be change. Our results in Section 5.3 confirm this analysis.

When we limit the compiler optimizations (MinRF, O0, and FMAD-OFF), the generated code is larger and execution time increases (Figure 4). When we compile the code with limited/no optimizations, the generated SASS is not reorganized for performance or optimized memory accesses. Similarly, when we limit the register file usage, the compiler inserts many registers spills instructions in the code to compensate for the reduced registers per thread. The MinRF code size has on average 23% more instructions than the only O3 code.

Figure 4 shows the execution time for the considered configurations, relative to the default NVCC compilation (O3). The execution time follows the same trend as code size. The optimizations that approximate the float instructions have a lower execution time than O3 compiled code. In fact, the FAST-MATH flag can reduce 40% of the execution time for some codes.

Contrarily, O0 and MinRF have the longest execution time. The unoptimized code (O0) significantly reduces the instruction-level parallelism, and reduces the memory accesses performance, increasing the latency of the instructions. Equivalently, for the MinRF version, the register spills necessary for reduced register file usage have a high cost in terms of execution time. The instructions must wait for the operands which are not in the register file.

It is worth noting that an incorrectly applied optimization can be 5× slower than the optimized one (O3), as shown in Figure 4. In sections 5 and 6, we show how these optimizations can impact not only the fault propagation and error rate but also the amount of work that the application process before experiencing a failure (i.e., the Mean Work Between Failures [47]). For instance, an unoptimized code may have a lower error rate than an optimized one, however, the much longer

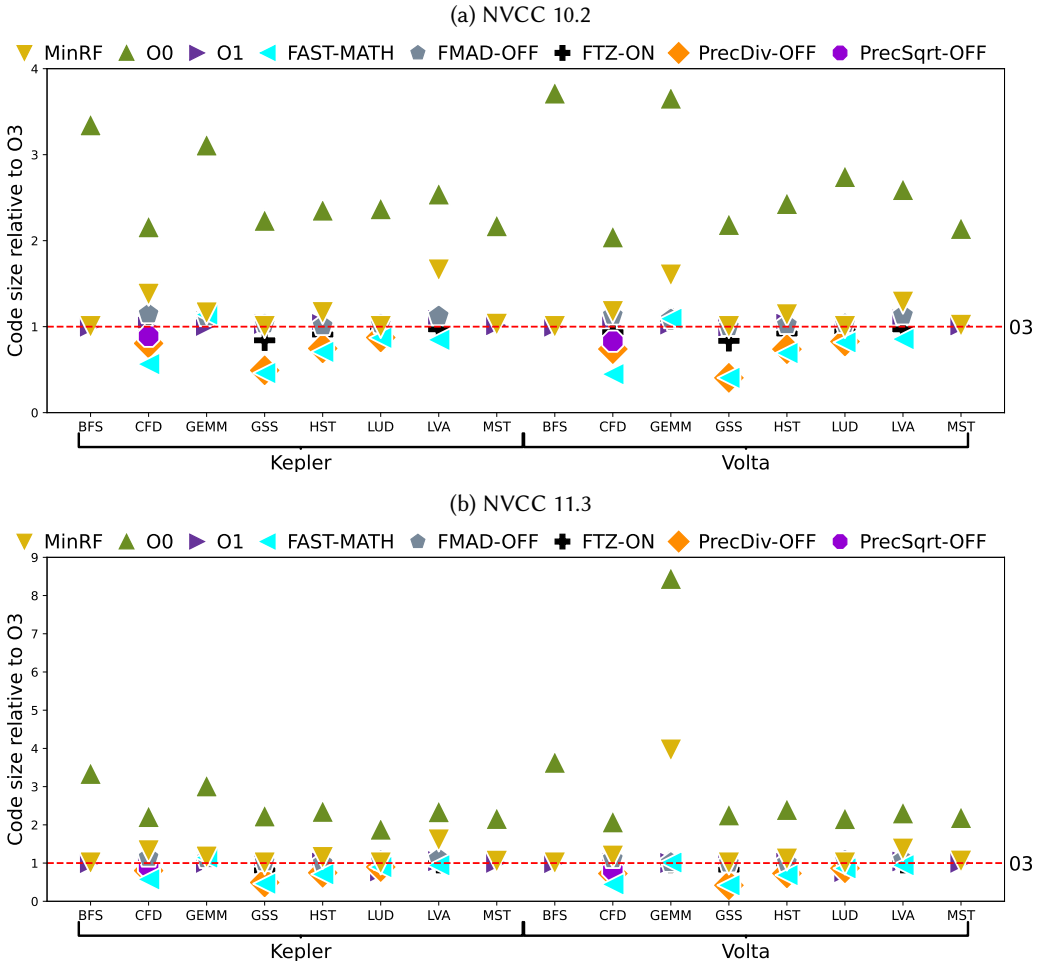


Fig. 3. Code size relative to default NVCC compilation for Nvcc versions 10.2 and 11.3

time needed to complete the execution of the unoptimized version will produce less useful work than the optimized one (details in Section 6).

5 COMPILER OPTIMIZATIONS IMPACT ON RELIABILITY

This section presents an ablation study on the impact of compiler optimization on the device error rate. First, we present the fault propagation analysis Program Vulnerability Factor (PVF) to understand the possible impact of compiler versions and optimizations on the fault propagation probability. We also consider the machine instruction distribution and the impact of using different hardware units. Compiler optimization, as it alters the machine code, is likely to modify the probability of a fault propagating to the output. Furthermore, certain compiler optimizations may change the choice of hardware functional units for calculations (e.g., MUL and ADD instead of FMA). Additionally, we investigate how compiler optimizations can alter the probability of a fault occurring.

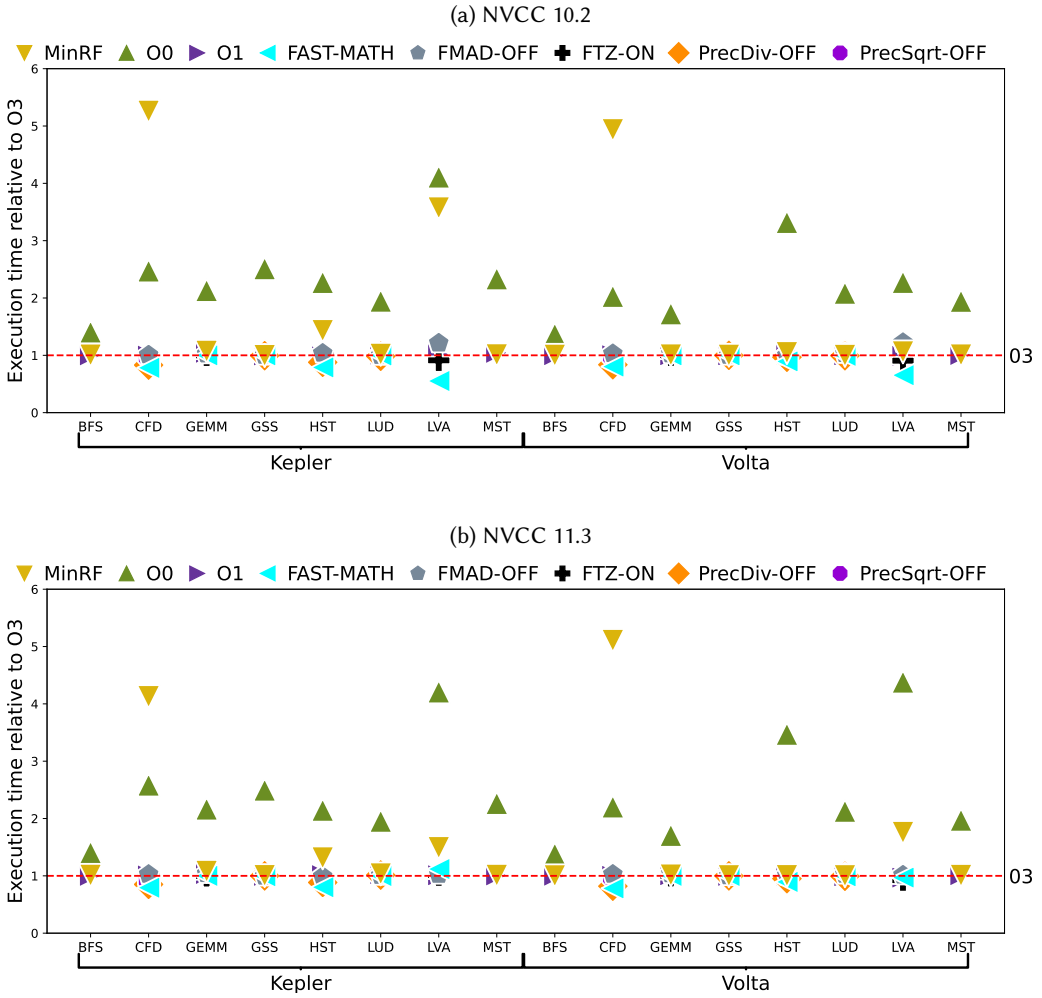


Fig. 4. Execution time relative to default NVCC compilation for versions 10.2 and 11.3

We discuss the baseline PVF for the default configuration of NVCC 10.2 and 11.3, specifically with the O3 optimization flag, aiming to understand if and how compiler versions impact the probability of a fault corrupting the output. Subsequently, we explore various NVCC optimizations for both compiler versions and compare their effects. We utilize both the SDC probability (solely software-based) and the SDC error rate estimation (which incorporates the hardware fault rate) to evaluate the impact of each optimization.

5.1 Random Software Fault Injection

We start our analysis by evaluating the PVF for two NVCC versions, namely 10.2 and 11.3, utilizing the default configuration with O3 optimization. With each major update of NVCC (e.g., from version 10 to 11), the compiler undergoes modifications to accommodate a new Instruction Set Architecture (ISA) for a new GPU family, including support to new mixed-precision instructions and data types.

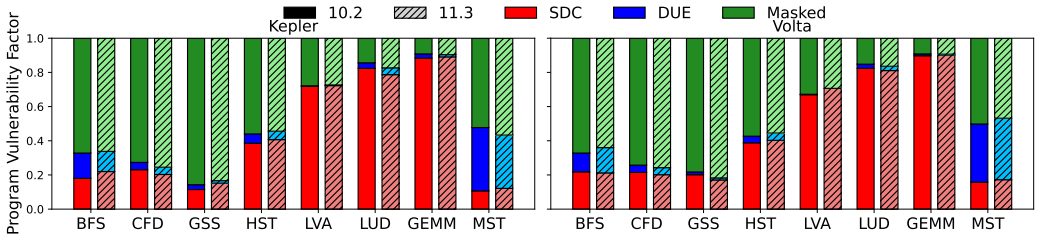


Fig. 5. Program Vulnerability Factor (PVF) comparison for the default NVCC 10.2 and 11.3 configurations with *O*3 flag optimization enabled.

Additionally, for every upgrade, code generation and optimizations are improved, and compiler bugs are corrected. The NVCC modifications may impact the fault propagation probabilities and possibly affect the error rate.

We inject 1,000 faults on codes build with the two latest NVCC versions that support Kepler and Volta architectures simultaneously (i.e., 10.2 and 11.3). Figure 5 shows, for the evaluated codes, the PVF, i.e., the probability for an injected fault to propagate to the output.

For the same code build with different NVCC versions, 10.2 and 11.3, the SDC and DUE PVF are similar for the majority of the benchmarks. On average, the PVF variation for the same code and a different compiler is only 5%. On average, the SDC PVF difference between Kepler (which has a much older instruction set) and Volta is 2%. For DUEs, the PVF for Kepler is, on average, 1.36× higher than Volta.

From the results, we observe that GEMM, LVA, and LUD consistently exhibit higher PVFs across all GPUs. In contrast, GSS and CFD display the lowest PVFs. The characteristics of the code directly influence the PVF, as fault propagation varies depending on how the faulty value is used. For instance, GEMM and LAVA are codes that involve repetitive execution of a limited set of instruction types (e.g., FMA and FMUL). Thus, a fault causing a slight deviation in the values will likely propagate throughout the computation. On the other hand, codes like GSS and CFD encompass a series of diverse instruction types, including the utilization of transcendental units, which may generate very small values. This inherent behavior can lead to increased rounding and catastrophic cancellation errors, effectively masking faults that generate small-magnitude errors. In the following sections, we will discuss if the tendencies observed on the baseline PVF (O3) hold for the SDC probability and the SDC rate estimation with different optimization configurations.

5.2 Contribution of Instructions Distribution

As discussed in previous works [1, 16, 30, 39, 57], the PVF provides an overview of the reliability of a code, but it does not consider the distribution of instructions, since faults are randomly injected during the code execution. To consider also the probability of the instruction to be picked on the fault injection campaign and generate an SDC, we also consider the instruction distribution on the fault propagation analysis, measuring the SDC probability [30], as shown in equation 1.

$$P_{SDC} = \sum (PVF_i * \frac{N_i}{N_{total}}) \quad (1)$$

Where PVF_i represents an PVF for a given instruction i, and $\frac{N_i}{N_{total}}$ represents the probability of an instruction i in the total instruction count (the ratio of instruction i).

Figure 6 shows the SDC probability (P_{SDC}) distribution (vertical axis) for all the evaluated codes (horizontal axis). We show the results for two GPUs, Kepler and Volta, with two NVCC versions.

We test the flags presented in Section 3 for each device and compiler. In most benchmarks, the SDC probability exhibits only a slight variation. The average Coefficient of variation, indicating the deviation of SDC probabilities from the mean, is ≈15%. With the exception of BFS (i.e., a graph search code not well suited for GPUs), most codes demonstrate a SDC probability that aligns closely with the default configuration (O3) for various optimization flags. This suggests that despite significant modifications in the executed machine code (see Section 4), the error rate is unlikely to change significantly. However, we find this result counterintuitive as the SDC probability does not account for a significant aspect of reliability, namely the hardware fault probability. The subsequent subsection and the experimental validation proposed in Section 6 provide further analysis and confirm this observation.

BFS has coefficients of variations of 32% and 43%, for NVCC 10.2 and 11.3, respectively. BFS is a code with a low PVF compared with the other codes. BFS is also naive and not tuned for performance implementations. Consequently, the benchmark instruction distribution is very concentrated in load/store and MISC (i.e., sync and NOP) instructions. The reliability evaluation of load, store, and MISC instructions is challenging, and it is hard to determine their sensitivity by software fault injection. Thus, the SDC probability may not reflect a realistic scenario. In the next section, we discuss the impact on the error rate prediction of the performance of the outliers codes.

The compilation flag that produces the lowest values of SDC probability is the unoptimized code (O0). This can be justified considering that without optimization, the compiler leaves basic blocks in the code that do not contribute to the final output or update registers that are overwritten. A fault in any of these optimized resources is simply masked and does not propagate to the final output, reducing the code PVF.

On the other hand, the MinRF, FAST-MATH, and FTZ-ON have the highest SDC probabilities. When the number of registers is limited, the criticality of each register increases, as the compiler will continuously optimize to use all registers available. As the number of instructions is reduced for the approximation flags, the fault impact in an approximated instruction is expected to be higher at the software level. Obviously, SDC probability does not consider the fact that a lower number of registers and approximation reduces the probability for the hardware fault to be generated, as we show next.

Each functional unit has a specific probability of being corrupted. Consequently, if an optimization changes the code instructions distribution, it will change the PVF (evaluated by the SDC probability) but also the instruction's contribution to the final error rate. We need to consider both the functional units' error rate and the performance of the code to estimate a more accurate error rate. We present this hardware/software analysis in the following subsection.

5.3 Contribution of the Hardware Sensitivity

The authors in [48] proposed a methodology to enhance the accuracy of error rate estimation by considering both the fault propagation probability of instructions (PVF) and the probability of faults originating in the hardware (instruction error rate, FIT). Equation 2 provides an estimation of the SDC FIT rate (†FIT).

$$\dagger\ FIT = IPC * AO * \sum (PVF_i * \frac{N_i}{N_{total}} * FIT_i) \tag{2}$$

Where PVF_i represents the PVF for a given instruction i, and $\frac{N_i}{N_{total}}$ represents the probability of an instruction i in the total application instruction count. The FIT_i represents the FIT rate of a given instruction measured through beam experiments. We use the instruction FIT_i data available from [48]. The FIT rate of the instructions is measured with microbenchmarks that execute the same instruction most of the time (99% of the instructions), with the maximum optimization (O3).

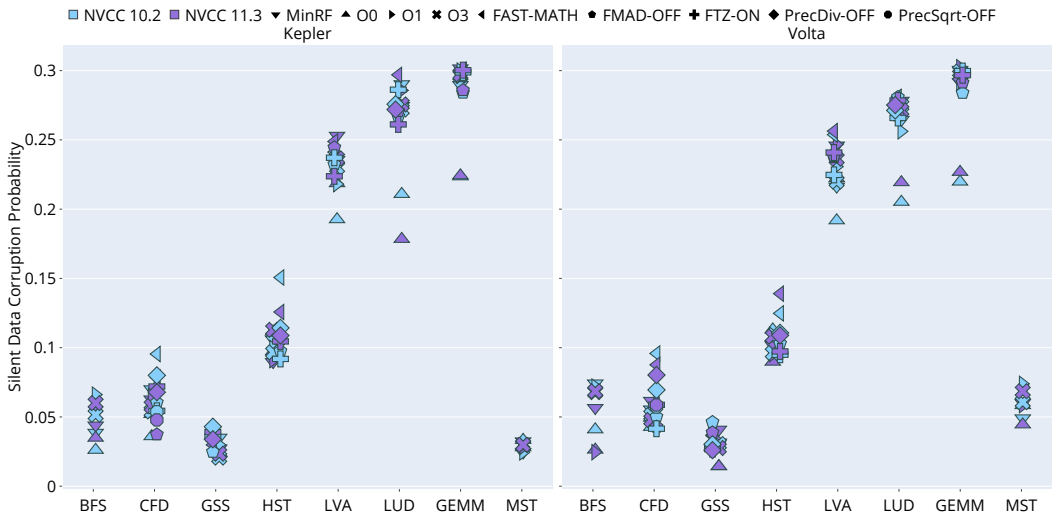


Fig. 6. Silent Data Corruption probability ($P_{(SDC)}$) distribution. Based on [30]. Despite various optimization flags used for both GPUs and NVCC versions, the SDC probability shows relatively low variations.

The goal of the microbenchmarks is only to evaluate the functional units' error rate and not the criticality of the errors introduced by approximations or compilation flags.

Additionally, as the code's FIT rate is directly related to the device parallelism management, the authors proposed a normalization factor. The normalization is obtained by multiplying the application *Instruction Per Cycle* (*IPC*) by the *Achieved Occupancy* (*AO*). It is worth noting that, for this work, the SDC probability and SDC rate estimation consider only faults in the functional units and the output registers. The caches and shared memories error rate are not considered for SDC probability or the SDC rate estimation. This scenario would be comparable to the ECC ON on a real device.

Figure 7 displays the SDC rate estimation for the codes listed in Table 1. The results include all configurations obtained from the SDC probability experiments. To facilitate comparison, the data in Figure 7 is normalized by the highest estimated SDC rate for each board, namely HST FAST-MATH 10.2 for Kepler and GEMM O0 11.3 for Volta.

The average Coefficient of Variation for the SDC rate estimation is 33.6% across all configurations shown in Figure 7. The highest variations are observed in LVA, with 98% for Kepler and 103% for Volta. From the selected codes, LVA is the most computationally intensive code, with 75% of the instructions involving float arithmetic. The error rate of LVA is directly influenced by the error rate of the functional units and how they are utilized. Any changes in the IPC or instruction distribution caused by flag configurations or the compiler will directly impact the SDC rate.

BFS and GSS exhibit the lowest estimated SDC rates. The low SDC estimation for GSS is attributed to its low PVF, which is influenced by the instructions used in the code. Similarly, GSS also has the lowest SDC probability. On the other hand, BFS has a low SDC estimation due to its low IPCs and GPU occupancy. This aspect may pose a limitation in the methodology as the error rate estimation is normalized based on performance metrics. Still, for codes that demonstrate poor performance metrics, it is necessary to assess other instruction types on the GPU, such as branch instructions, synchronization instructions, and load/store operations from various cache levels. In other words,

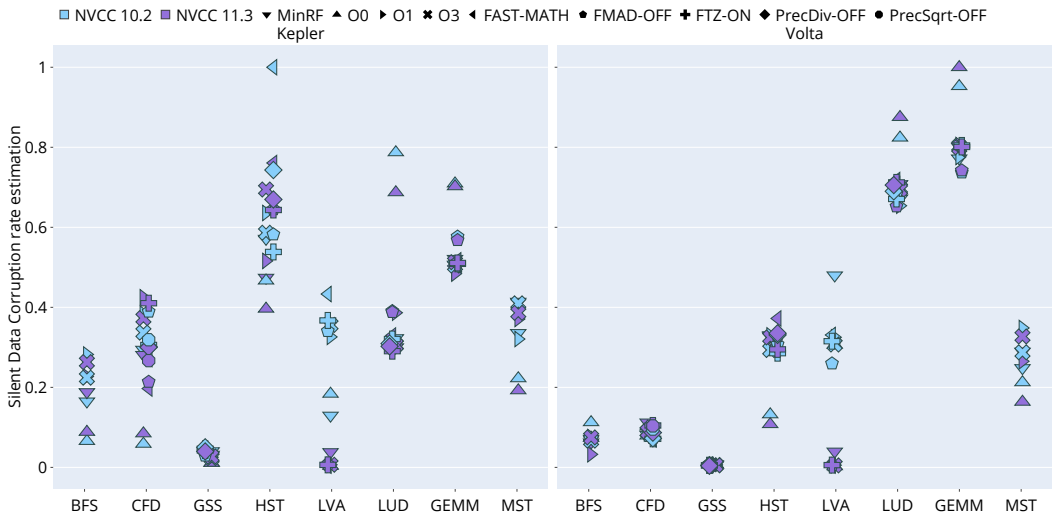


Fig. 7. SDC rate estimation (†*FIT*) distribution. Based on [48]. Contrary to the SDC probability, the SDC FIT estimation shows considerable variation for the same code with different compiler versions and optimizations, highlighting the importance of considering hardware in fault propagation analysis.

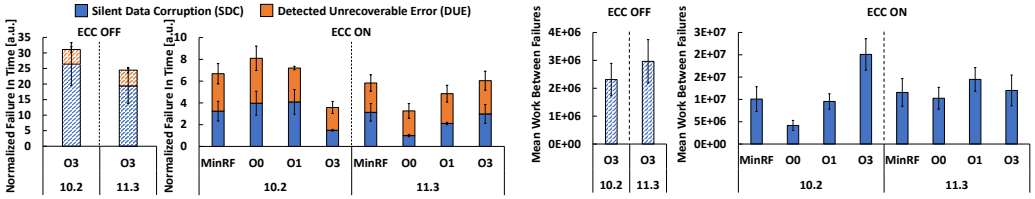
the analysis must also consider instructions that impact the kernel's performance (e.g., instructions causing pipeline stalls) to enhance the accuracy of the estimation.

The differences between the SDC probabilities for the two compilers **NVCC 10.2 and 11.3** follow the trend shown in Figure 5. The SDC probability ratio between NVCC 10.2 and 11.3 is, on average, 1.03×. Contrarily, for the SDC rate estimation, the ratio between the NVCC 10.2 and 11.3 is, on average, 8× higher for version 10.2. In fact, the SDC rate estimation ratio between NVCC 10.2 and 11.3 is not homogeneous as the SDC probability. That is, some benchmarks have much higher differences than the mean. Considering the SDC rate estimation ratio between the NVCC 10.2 and 11.3 in the 75% quartile of values, the average difference between the two compilers is 22% higher for NVCC 10.2. This discrepancy could not be observed using only software fault injection and the instruction distribution as it directly depends on the hardware fault probability and the usage of the resources. A subset of configurations evaluated with beam experiments will be analyzed in the next section to help to demonstrate this observation.

It is worth noting that the ideal compiler optimization depends on the application's priorities. In cases where the primary concern is the amount of data processed, such as in HPC, opting for a more aggressive optimization strategy is advisable. This choice, while potentially leading to higher PVF and SDC rates, allows more executions to be completed before the error (i.e., higher MWBF). Contrarily, the reliability focus shifts to minimizing the error rate in safety-critical real-time applications. For instance, if a camera must deliver 40 frames per second, an optimization flag that allows the GPU to process more frames is unnecessary. A less aggressive optimization strategy is preferable in such cases, as it effectively reduces the SDC rate.

6 NEUTRON BEAM EXPERIMENTS RESULTS

In this section, we validate the analysis proposed in Section 5 by leveraging beam experiments on real GPUs. We begin by examining the error rate of GEMM (i.e., *Failure In Time*, FIT), measured through neutron beam experiments, to evaluate the influence of each optimization on the code's



(a) Normalized Failure In Time (FIT) (b) Mean Work Between Failures (MWBF)

Fig. 8. Normalized SDC and DUE FIT rates and MWBF for CUDA 10.2 and 11.3. The FIT rate for GEMM compiled with O0 ECC ON is used as a normalization factor.

reliability. Additionally, we combine the performance of each configuration with the error rate to present the results of *Mean Work Between Failures* (MWBF).

Figure 8a shows the Silent Data Corruption (SDC) and Detected Unrecoverable Errors (DUEs) FIT rates for GEMM executed on a Kepler GPU exposed to a neutron beam. To ensure a meaningful comparison of error rates across different configurations while safeguarding business-sensitive data, the FIT rate is normalized using the smallest value as the reference (i.e., GEMM compiled with O0 on NVCC 11.3 when ECC is ON). Due to limitations in beam time, our analysis primarily focuses on configurations with ECC ON while displaying the most optimized code when ECC is OFF (O3). Previous studies have established that disabling ECC on GPUs makes memories the primary source of errors [32, 37, 48, 51]. Consequently, the FIT rate is predominantly influenced by memory-related errors. However, this analysis is less intriguing and less representative of real-world scenarios compared to examining the impact of compiler optimizations on code, where the focus is on errors arising from arithmetical and thread/warp synchronization instructions.

When ECC is OFF, the SDC FIT rate for the most optimized configurations (O3) is more than one order of magnitude higher compared to when ECC is ON (17.8× higher for NVCC 10.2), and 6.5× higher for NVCC 11.3. The register file and the caches are unprotected when ECC is OFF, leading to an increased SDC rate. An interesting observation is that the SDC FIT rate is consistently higher than the DUE FIT rate when ECC is OFF, whereas, with ECC enabled, the DUE rate is similar to or higher than the SDC rate. A double-bit flip detected by the ECC triggers an exception that leads to a crash. The increase in the DUE rate when ECC is ON aligns with previous research findings [32].

Figure 8a shows significant differences in the FIT rate for the same code occur when ECC is ON, and the code is compiled with different compilers. Surprisingly, the FIT rate obtained with NVCC 10.2 is the opposite of the FIT rate obtained with NVCC 11.3. This behavior can be attributed to the variations in the code generated by the two compilers. For instance, the NVCC 10.2 compiler generates code with a difference in the number of MISC instructions across all four configurations (on average, NVCC 10.2 has 10^6 more MISC instructions, i.e., ≈3% more MISC instructions). These findings align with the observations discussed in Section 5 and highlight the importance of considering factors beyond software fault injection alone to ensure accurate estimations. The GEMM build compiled with NVCC 10.2 using the O3 optimization leverages GPU resources more effectively, including register usage, resulting in better overall performance than the O3 version compiled with NVCC 11.3. Consequently, the FIT rate is lower for the NVCC 10.2 O3 configuration.

Compiler optimizations significantly impact the code's error rate. Furthermore, these optimizations also improve the performance of the code. When the code executes faster, it has the potential to generate a higher amount of correct data before encountering a failure. We measure the MWBF for each configuration tested to establish a correlation between reliability and performance. MWBF is

Table 2. Comparison of GEMM SDC FIT rates obtained using the approach described in Section 5.3 and those obtained from Neutron Beam experiments. The SDC FIT rates are normalized using the smallest FIT rate, which corresponds to NVCC 11.3 compiled with the O0 flag.

NVCC	Optimization flag	SDC FIT rate		Beam/Predicted
		Predicted	Beam	
10.2	**MinRF**	2.41	3.24	1.35
	O0	3.37	3.97	1.18
	O1	2.32	4.09	1.76
	O3	2.39	1.48	0.62
11.3	**MinRF**	2.47	3.13	1.27
	O0	3.33	1.00	0.30
	O1	2.30	2.11	0.92
	O3	2.43	2.99	1.23

defined as the amount of correct data produced by the system before a failure occurs, encompassing both SDCs and DUEs [47]. The calculation of MWBF is obtained by multiplying the number of executions between failures by the workload of the application. A higher MWBF rate indicates the system can process a larger workload before encountering an error.

Figure 8b shows the MWBF for the float matrix multiplication (GEMM) across all configurations tested in the neutron beam experiments. Despite the increase in error rates in some configurations, optimizations that enhance performance also lead to higher MWBF values for the application. In other words, the performance improvements achieved through optimization outweigh the increase in error rate. As expected, when ECC is OFF, the MWBF values for both compiler versions are lower than when ECC is ON (8.7× lower for NVCC 10.2 and 4.1× lower for NVCC 11.3, for the O3 configuration). Despite increased DUEs with ECC ON, this version remains more reliable and generates more correct data than the ECC OFF version.

When ECC is ON, the O3 configuration for NVCC 11.3 increases the total FIT rate by 69% without improving performance. Consequently, the NVCC 10.2 compiled with O3 when ECC is ON exhibits the highest MWBF among all configurations. It is worth noting that even with the lower improvement in NVCC 11.3 compared to NVCC 10.2, the MWBF values for the O1 and O3 configurations of NVCC 11.3 are consistently higher than the less optimized MinRF and O0 codes. Specifically, the MWBF of O1 is 41.2% higher than O0 and 25.2% higher than MinRF, while the MWBF of O3 is 17.11% higher than O0 and 4% higher than MinRF. This indicates that, even with higher FIT rates, the most optimized versions of the code, O1 and O3, can still produce more correct results before experiencing a failure.

In order to provide a comprehensive overview of the results shown in Figure 8, we compare the SDC FIT obtained from beam experiments with the predicted SDC FIT (Section 5.3). Table 2 presents both SDC FIT rates (Predicted and Beam) and the ratio between both (Beam divided by Predicted). Predicted and Beam SDC FITs are normalized by the lowest value in the table, corresponding to the SDC FIT obtained with NVCC 11.3 O0 from the beam experiments. On average, the predicted SDC FIT for NVCC 10.2 is underestimated, as it is 1.23× lower than the beam SDC FIT. Conversely, the beam SDC FIT for NVCC 11.3 is 0.93× lower than the predicted SDC FIT. It is worth noting that, in Table 2 , we are comparing a measurement based on physical stress (beam) and a prediction based on simulation. An intrinsic difference between the two methodologies is then to be expected. The reported beam/predicted ratio is actually more accurate than a similar comparison made on simpler computing devices, such as the ARM A5 and A9, in [6]. Additionally, without considering

the hardware sensitivity, as we do in the SDC FIT rate prediction, the differences would diverge and be higher than one order of magnitude. This discrepancy directly relates to the instruction types used in the SDC FIT prediction, which are all related to arithmetic instructions (i.e., FMA, ADD, and MUL). Even a slight variation in the code's instructions can significantly impact the final FIT rate, as observed in Figure 8. Nonetheless, using a small set of instruction types still allows for estimation within the same order of magnitude as the measured beam FIT.

Our results show that modifications in the compilation process, such as a slight increase in register file usage, can impact the application's reliability. Code generation can influence the probability of fault propagation, hardware usage (e.g., IPC), and the stress placed on specific functional units, thereby altering the code's error rate. While software-level analysis provides a reasonable estimate of code fault propagation, it does not account for hardware factors such as instruction latency, GPU occupancy, and IPC, which have been shown to influence the final error rate. Therefore, it is essential to consider all these factors in the overall analysis. Overall, the results indicate that increasing performance, even if it leads to a linear increase in the FIT rate, outweighs the increase in the FIT rate. The MWBF for the most optimized configurations increases more than those less optimized configurations. This suggests that for GPUs, it is preferable to push the device to its maximum capacity to obtain more correct results before encountering a failure.

7 CONCLUSIONS

In this paper, we have evaluated the impact of compiler versions and compiler optimizations on the reliability of codes executed on GPUs. We have considered eight representative codes and two NVIDIA GPU architectures. A pure software reliability evaluation is not sufficient for an accurate reliability evaluation. Even if the SASS code generated with different optimization flags differs significantly, the resulting SDC probability changes only slightly, which is highly unrealistic. As we have shown, also considering the hardware fault probability is necessary to estimate the SDC error rate better.

To validate our observations and have a realistic evaluation, we have experimentally measured the impact of optimization flags and compiler version on the Kepler GPU FIT rate using an accelerated neutron beam. The compiler optimizations significantly impact the code error rate, thus demonstrating the need to consider the hardware fault probability. Moreover, we have shown that while a more optimized code has a higher PVF and even a higher FIT rate, in general, optimization increases the MWBF. In other words, the performance gain increases more than the error rate. Optimizing the code, then, has the benefit of increasing the device's reliability.

ACKNOWLEDGMENTS

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 899546 (Marie Skłodowska-Curie) with the support of the Brittany Region. This research also has been partially funded by The Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brazil (CAPES) - Finance Code 001. This research also has been partially funded by Indam GNCS Project 2022 under Grant CUP_E55F22000270001, and in part by the European Union under NextGenerationEU (Italy). Neutron beam time was provided by ChipIR (DOI:10.5286/ISIS.E.RB2200004-1, 10.5286/ISIS.E.RB2000137-1, 10.5286/ISIS.E.101136531) thanks to Chris Frost, Carlo Cazzaniga, and Maria Kastriotou and by LANSCE thanks to Steve Wender and Gus Sinnis.

REFERENCES

[1] Abdul Rehman Anwer, Guanpeng Li, Karthik Pattabiraman, Michael Sullivan, Timothy Tsai, and Siva Kumar Sastry Hari. 2020. GPU-Trident: Efficient Modeling of Error Propagation in GPU Programs. In *Proceedings of the International*

Conference for High Performance Computing, Networking, Storage and Analysis (Atlanta, Georgia) *(SC '20)*. IEEE Press, Article 88, 15 pages.

[2] R. A. Ashraf, R. Gioiosa, G. Kestor, and R. F. DeMara. 2017. Exploring the Effect of Compiler Optimizations on the Reliability of HPC Applications. In *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 1274–1283.

[3] Jose M. Badia, German Leon, Jose A. Belloch, Mario Garcia-Valderas, Almudena Lindoso, and Luis Entrena. 2021. Comparison of parallel implementation strategies in GPU-accelerated System-on-Chip under proton irradiation. *IEEE Transactions on Nuclear Science* (2021), 1–1. https://doi.org/10.1109/TNS.2021.3128722

[4] R. Baumann. 2005. Soft errors in advanced computer systems. *IEEE Design Test of Computers* 22, 3 (May 2005), 258–266. https://doi.org/10.1109/MDT.2005.69

[5] Massimo Bernaschi, Mauro Bisson, Enrico Mastrostefano, and Flavio Vella. 2018. Multilevel Parallelism for the Exploration of Large-Scale Graphs. *IEEE Transactions on Multi-Scale Computing Systems* 4, 3 (2018), 204–216. https://doi.org/10.1109/TMSCS.2018.2797195

[6] Pablo R. Bodmann, George Papadimitriou, Rubens L. Rech Junior, Dimitris Gizopoulos, and Paolo Rech. 2022. Soft Error Effects on Arm Microprocessors: Early Estimations versus Chip Measurements. *IEEE Trans. Comput.* 71, 10 (2022), 2358–2369. https://doi.org/10.1109/TC.2021.3128501

[7] Carlo Cazzaniga and Christopher D. Frost. 2018. Progress of the Scientific Commissioning of a fast neutron beamline for Chip Irradiation. *Journal of Physics: Conference Series* 1021 (may 2018), 012037. https://doi.org/10.1088/1742-6596/1021/1/012037

[8] A. Chatzidimitriou, P. Bodmann, G. Papadimitriou, D. Gizopoulos, and P. Rech. 2019. Demystifying Soft Error Assessment Strategies on ARM CPUs: Microarchitectural Fault Injection vs. Neutron Beam Experiments. In *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 26–38.

[9] Athanasios Chatzidimitriou, Manolis Kaliorakis, Dimitris Gizopoulos, Maurizio Iacaruso, Mauro Pipponzi, Riccardo Mariani, and Stefano Di Carlo. 2017. RT Level vs. Microarchitecture-Level Reliability Assessment: Case Study on ARM(R) Cortex(R)-A9 CPU. In *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*. IEEE. https://doi.org/10.1109/dsn-w.2017.16

[10] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S. Lee, and K. Skadron. 2009. Rodinia: A benchmark suite for heterogeneous computing. In *2009 IEEE International Symposium on Workload Characterization (IISWC)*. 44–54.

[11] Zitao Chen, Guanpeng Li, Karthik Pattabiraman, and Nathan DeBardeleben. 2019. BinFI: An Efficient Fault Injector for Safety-Critical Machine Learning Systems. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Denver, Colorado) *(SC 2019)*. Association for Computing Machinery, New York, NY, USA, Article 69, 23 pages. https://doi.org/10.1145/3295500.3356177

[12] Hyungmin Cho, Shahrzad Mirkhani, Chen-Yong Cher, Jacob A. Abraham, and Subhasish Mitra. 2013. Quantitative evaluation of soft error injection techniques for robust system design. In *Proceedings of the 50th Annual Design Automation Conference on - DAC '13*. ACM Press. https://doi.org/10.1145/2463209.2488859

[13] Josie E. Rodriguez Condia, Boyang Du, Matteo Sonza Reorda, and Luca Sterpone. 2020. FlexGripPlus: An improved GPGPU model to support reliability analysis. *Microelectronics Reliability* 109 (2020), 113660. https://doi.org/10.1016/j.microrel.2020.113660

[14] Cristian Constantinescu, Mike Butler, and Chris Weller. 2012. Error Injection-Based Study of Soft Error Propagation in AMD Bulldozer Microprocessor Module. In *Proceedings of the 2012 42nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN) (DSN 2012)*. IEEE Computer Society, USA, 1–6.

[15] B. Fang, K. Pattabiraman, M. Ripeanu, and S. Gurumurthi. 2014. GPU-Qin: A methodology for evaluating the error resilience of GPGPU applications. In *Performance Analysis of Systems and Software (ISPASS), 2014 IEEE International Symposium on*. 221–230. https://doi.org/10.1109/ISPASS.2014.6844486

[16] Shuguang Feng, Shantanu Gupta, Amin Ansari, and Scott Mahlke. 2010. Shoestring: Probabilistic Soft Error Reliability on the Cheap. *SIGPLAN Not.* 45, 3 (March 2010), 385–396. https://doi.org/10.1145/1735971.1736063

[17] Fernando Fernandes dos Santos, Caio Lunardi, Daniel Oliveira, Fabiano Libano, and Paolo Rech. 2019. Reliability Evaluation of Mixed-Precision Architectures. (2019), 238–249. https://doi.org/10.1109/HPCA.2019.00041

[18] Davide Ferraretto and Graziano Pravadelli. 2016. Simulation-based Fault Injection with QEMU for Speeding-up Dependability Analysis of Embedded Software. *Journal of Electronic Testing* 32, 1 (Jan 2016), 43–57. https://doi.org/10.1007/s10836-015-5555-z

[19] Dionysios Filippas, Nikolaos Margomenos, Nikolaos Mitianoudis, Chrysostomos Nicopoulos, and Giorgos Dimitrakopoulos. 2022. Low-Cost Online Convolution Checksum Checker. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 30, 2 (2022), 201–212.

[20] D. A. G. Goncalves de Oliveira, L. L. Pilla, T. Santini, and P. Rech. 2016. Evaluation and Mitigation of Radiation-Induced Soft Errors in Graphics Processing Units. *IEEE Trans. Comput.* 65, 3 (2016), 791–804.

[21] Siva Kumar Sastry Hari, Sarita V. Adve, and Helia Naeimi. 2012. Low-cost program-level detectors for reducing silent data corruptions. In *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2012)*. 1–12. https://doi.org/10.1109/DSN.2012.6263960

[22] Siva Kumar Sastry Hari, Michael B. Sullivan, Timothy Tsai, and Stephen W. Keckler. 2022. Making Convolutions Resilient Via Algorithm-Based Error Detection Techniques. *IEEE Transactions on Dependable and Secure Computing* 19, 4 (2022), 2546–2558. https://doi.org/10.1109/TDSC.2021.3063083

[23] Siva Kumar Sastry Hari, Radha Venkatagiri, Sarita V. Adve, and Helia Naeimi. 2014. GangES: Gang error simulation for hardware resiliency evaluation. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*. 61–72. https://doi.org/10.1109/ISCA.2014.6853212

[24] Kojiro Ito, Yangchao Zhang, Hiroaki Itsuji, Takumi Uezono, Tadanobu Toba, and Masanori Hashimoto. 2021. Analyzing DUE Errors on GPUs With Neutron Irradiation Test and Fault Injection to Control Flow. *IEEE Transactions on Nuclear Science* 68, 8 (2021), 1668–1674. https://doi.org/10.1109/TNS.2021.3098845

[25] Xabier Iturbe, Balaji Venu, and Emre Ozer. 2016. Soft error vulnerability assessment of the real-time safety-related ARM Cortex-R5 CPU. In *2016 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*. IEEE. https://doi.org/10.1109/dft.2016.7684076

[26] JEDEC. 2006. *Measurement and Reporting of Alpha Particle and Terrestrial Cosmic Ray-Induced Soft Errors in Semiconductor Devices*. Technical Report JESD89A. JEDEC Standard.

[27] Saurabh Jha, Timothy Tsai, Siva Hari, Michael Sullivan, Zbigniew Kalbarczyk, Stephen W. Keckler, and Ravishankar K. Iyer. 2019. Kayotee: A Fault Injection-based System to Assess the Safety and Reliability of Autonomous Vehicles to Faults and Errors. arXiv:1907.01024 [cs.SE]

[28] Paolo Sylos Labini, Marco Cianfriglia, Damiano Perri, Osvaldo Gervasi, Grigori Fursin, Anton Lokhmotov, Cedric Nugteren, Bruno Carpentieri, Fabiana Zollo, and Flavio Vella. 2021. On the Anatomy of Predictive Models for Accelerating GPU Convolution Kernels and Beyond. *ACM Trans. Archit. Code Optim.* 18, 1, Article 16 (jan 2021), 24 pages. https://doi.org/10.1145/3434402

[29] Germán León, José M. Badía, Jose A. Belloch, Almudena Lindoso, and Luis Entrena. 2020. Evaluating the soft error sensitivity of a GPU-based SoC for matrix multiplication. *Microelectronics Reliability* 114 (2020), 113856. https://doi.org/10.1016/j.microrel.2020.113856 31st European Symposium on Reliability of Electron Devices, Failure Physics and Analysis, ESREF 2020.

[30] Guanpeng Li and Karthik Pattabiraman. 2018. Modeling Input-Dependent Error Propagation in Programs. In *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 279–290. https://doi.org/10.1109/DSN.2018.00038

[31] F. Libano, B. Wilson, J. Anderson, M. J. Wirthlin, C. Cazzaniga, C. Frost, and P. Rech. 2019. Selective Hardening for Neural Networks in FPGAs. *IEEE Transactions on Nuclear Science* 66, 1 (2019), 216–222.

[32] C. Lunardi, F. Previlon, D. Kaeli, and P. Rech. 2018. On the Efficacy of ECC and the Benefits of FinFET Transistor Layout for GPU Reliability. *IEEE Transactions on Nuclear Science* 65, 8 (Aug 2018), 1843–1850. https://doi.org/10.1109/TNS.2018.2823786

[33] N.N. Mahatme, T.D. Jagannathan, L.W. Massengill, B.L. Bhuva, S.-J. Wen, and R. Wong. 2011. Comparison of Combinational and Sequential Error Rates for a Deep Submicron Process. *Nuclear Science, IEEE Transactions on* 58, 6 (2011), 2719–2725.

[34] Thibaut Marty, Tomofumi Yuki, and Steven Derrien. 2018. Enabling Overclocking Through Algorithm-Level Error Detection. In *FPT*.

[35] Shubhendu S. Mukherjee, Christopher Weaver, Joel Emer, Steven K. Reinhardt, and Todd Austin. 2003. A Systematic Methodology to Compute the Architectural Vulnerability Factors for a High-Performance Microprocessor. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, Washington, DC, USA, 29–.

[36] Daniel Oliveira, Laércio Pilla, Nathan DeBardeleben, Sean Blanchard, Heather Quinn, Israel Koren, Philippe Navaux, and Paolo Rech. 2017. Experimental and Analytical Study of Xeon Phi Reliability. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Denver, Colorado) *(SC '17)*. ACM, New York, NY, USA, Article 28, 12 pages. https://doi.org/10.1145/3126908.3126960

[37] Daniel A. G. Oliveira, Paolo Rech, Laércio L. Pilla, Philippe O. A. Navaux, and Luigi Carro. 2014. GPGPUs ECC efficiency and efficacy. In *2014 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*. 209–215. https://doi.org/10.1109/DFT.2014.6962085

[38] John D. Owens, Mike Houston, David Luebke, Simon Green, John E. Stone, and James C. Phillips. 2008. GPU Computing. *Proc. IEEE* 96, 5 (2008), 879–899. https://doi.org/10.1109/JPROC.2008.917757

[39] Lucas Palazzi, Guanpeng Li, Bo Fang, and Karthik Pattabiraman. 2019. A Tale of Two Injectors: End-to-End Comparison of IR-Level and Assembly-Level Fault Injection. In *2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE)*. 151–162. https://doi.org/10.1109/ISSRE.2019.00024

[40] George Papadimitriou and Dimitris Gizopoulos. 2021. Characterizing Soft Error Vulnerability of CPUs Across Compiler Optimizations and Microarchitectures. In *2021 IEEE International Symposium on Workload Characterization (IISWC)*. 113–124. https://doi.org/10.1109/IISWC53511.2021.00021

[41] George Papadimitriou and Dimitris Gizopoulos. 2021. Demystifying the System Vulnerability Stack: Transient Fault Effects Across the Layers. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. 902–915. https://doi.org/10.1109/ISCA52012.2021.00075

[42] Rafael B. Parizi, Ronaldo R. Ferreira, Luigi Carro, and Álvaro F. Moreira. 2013. Compiler Optimizations Do Impact the Reliability of Control-Flow Radiation Hardened Embedded Software. In *Embedded Systems: Design, Analysis and Verification*, Gunar Schirner, Marcelo Götz, Achim Rettberg, Mauro C. Zanella, and Franz J. Rammig (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 49–60.

[43] P. Peronnard, R. Ecoffet, M. Pignol, D. Bellin, and R. Velazco. 2008. Predicting the SEU error rate through fault injection for a complex microprocessor. In *2008 IEEE International Symposium on Industrial Electronics*. 2288–2292. https://doi.org/10.1109/ISIE.2008.4677290

[44] Fritz G. Previlon, Babatunde Egbantan, Devesh Tiwari, Paolo Rech, and David. R. Kaeli. 2017. Combining architectural fault-injection and neutron beam testing approaches toward better understanding of GPU soft-error resilience. In *2017 IEEE 60th International Midwest Symposium on Circuits and Systems (MWSCAS)*. 898–901. https://doi.org/10.1109/MWSCAS.2017.8053069

[45] Paolo Rech, Luigi Carro, Nicholas Wang, Timothy Tsai, Siva Kumar Sastry Hari, and Stephen W. Keckler. 2014. Measuring the Radiation Reliability of SRAM Structures in GPUS Designed for HPC. In *IEEE 10th Workshop on Silicon Errors in Logic - System Effects (SELSE)*.

[46] P. Rech, T.D. Fairbanks, H.M. Quinn, and L. Carro. 2013. Threads Distribution Effects on Graphics Processing Units Neutron Sensitivity. *Nuclear Science, IEEE Transactions on* 60, 6 (Dec 2013), 4220–4225. https://doi.org/10.1109/TNS.2013.2286970

[47] G.A. Reis, J. Chang, N. Vachharajani, S.S. Mukherjee, R. Rangan, and D.I. August. 2005. Design and evaluation of hybrid fault-detection systems. In *32nd International Symposium on Computer Architecture (ISCA'05)*. 148–159. https://doi.org/10.1109/ISCA.2005.21

[48] Fernando Fernandes dos Santos, Siva Kumar Sastry Hari, Pedro Martins Basso, Luigi Carro, and Paolo Rech. 2021. Demystifying GPU Reliability: Comparing and Combining Beam Experiments, Fault Simulation, and Profiling. In *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 289–298. https://doi.org/10.1109/IPDPS49936.2021.00037

[49] Fernando Fernandes dos Santos, Pedro Foletto Pimenta, Caio Lunardi, Lucas Draghetti, Luigi Carro, David Kaeli, and Paolo Rech. 2019. Analyzing and Increasing the Reliability of Convolutional Neural Networks on GPUs. *IEEE Transactions on Reliability* 68, 2 (2019), 663–677. https://doi.org/10.1109/TR.2018.2878387

[50] Vilas Sridharan and David R. Kaeli. 2009. Eliminating microarchitectural dependency from Architectural Vulnerability. In *2009 IEEE 15th International Symposium on High Performance Computer Architecture*. 117–128. https://doi.org/10.1109/HPCA.2009.4798243

[51] Michael B. Sullivan, Nirmal Saxena, Mike O'Connor, Donghyuk Lee, Paul Racunas, Saurabh Hukerikar, Timothy Tsai, Siva Kumar Sastry Hari, and Stephen W. Keckler. 2021. *Characterizing And Mitigating Soft Errors in GPU DRAM*. Association for Computing Machinery, New York, NY, USA, 641–653. https://doi.org/10.1145/3466752.3480111

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

[53] Sotiris Tselonis and Dimitris Gizopoulos. 2016. GUFI: A framework for GPUs reliability assessment. In *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 90–100. https://doi.org/10.1109/ISPASS.2016.7482077

[54] Alessandro Vallero, Dimitris Gizopoulos, and Stefano Di Carlo. 2017. SIFI: AMD southern islands GPU microarchitectural level fault injector. 138–144. https://doi.org/10.1109/IOLTS.2017.8046209

[55] Jack Wadden, Alexander Lyashevsky, Sudhanva Gurumurthi, Vilas Sridharan, and Kevin Skadron. 2014. Real-World Design and Evaluation of Compiler-Managed GPU Redundant Multithreading. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture* (Minneapolis, Minnesota, USA) *(ISCA '14)*. IEEE Press, 73–84.

[56] L. Yang, B. Nie, A. Jog, and E. Smirni. 2021. Enabling Software Resilience in GPGPU Applications via Partial Thread Protection. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE Computer Society, Los Alamitos, CA, USA, 1248–1259. https://doi.org/10.1109/ICSE43902.2021.00114

[57] Keun Soo Yim, Cuong Pham, Mushfiq Saleheen, Zbigniew Kalbarczyk, and Ravishankar Iyer. 2011. Hauberk: Lightweight Silent Data Corruption Error Detector for GPGPU. In *2011 IEEE International Parallel Distributed Processing Symposium*. 287–300. https://doi.org/10.1109/IPDPS.2011.36