HAL open science

Can GPU Performance Increase Faster Than the Code Error Rate?

Fernando Fernandes dos Santos, Paolo Rech

To cite this version:

Fernando Fernandes dos Santos, Paolo Rech. Can GPU Performance Increase Faster Than the Code Error Rate?. Journal of Supercomputing, In press, pp.1-32. 10.1007/s11227-024-06119-4 . hal-04528798

HAL Id: hal-04528798

https://hal.science/hal-04528798v1

Submitted on 2 Apr 2024

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution 4.0 International License

Can GPU Performance Increase Faster Than the Code Error Rate?

Fernando Fernandes dos Santos[1*] and Paolo Rech[2,3*]

[1]Univ Rennes, CNRS, Inria, IRISA - UMR 6074, F-35000 Rennes, France.
[2]Department of Industrial Engineering, University of Trento Trento, Italy.
[3]Institute of Informatics, Federal University of Rio Grande do Sul Porto Alegre, Brazil.

*Corresponding author(s). E-mail(s): fernando.fernandes-dos-santos@inria.fr; paolo.rech@unitn.it;

Abstract

Graphics Processing Units (GPUs) are the reference architecture to accelerate High Performance Computing (HPC) applications and the training/interference of Convolutional Neural Networks (CNNs). For both these domains, performance and reliability are two of the main constraints. It is believed that the only way to increase reliability is to sacrifice performance, e.g., using redundancies. We show in this paper that this is not always the case. As a very promising result, we found that most GPUs performance improvements also bring the benefit of increasing the number of executions correctly completed before experiencing a Silent Data Corruption (SDC). We consider four different common GPUs' performance optimizations: architectural solutions, software implementations, compiler optimizations, and threads degree of parallelism. We compare different implementations of a variety of parallel codes and, through beam experiments and applications profiling, we show that the performance improvement typically (but not necessarily) increases the GPU SDC rate. Nevertheless, for the vast majority of the configurations the performance gain is much higher than the SDC rate increase, allowing to process a higher amount of correct data. As we show, the programmer choices can increase up to **25** $\times$ the number of correctly completed executions without redesigning the algorithm nor including specific hardening solutions.

Keywords: Error rate, Graphics Processing Unit, Reliability

1 Introduction

Graphics Processing Units (GPUs) have evolved from being devices dedicated to gaming, graphics, and video rendering to flexible accelerators for a variety of High Performance Computing (HPC) and safety-critical applications, including autonomous vehicles. In particular, GPUs are the reference architecture for the training and the interference of Convolutional Neural Networks (CNNs), which are required to detect and classify objects in a frame. This market shift led to a burst in GPUs' computing capabilities and efficiency, significant improvements in the programming frameworks and performance evaluation tools, and a sudden concern about their hardware reliability.

A common belief is that reliability and performance are conflicting proprieties with opposite requirements. Reliability and performance should then be traded-off, trying to achieve a sufficient reliability level without losing too much on performance or ensuring that the recovery from faults does not significantly impact the performance. In this paper we show that, on GPUs, this is not the case. Our goal is to show how and why better performance leads to a higher amount of correctly processed data. This trend holds both when the optimization uses less GPU area (thus reducing the Silent Data Corruption rate) but also when a larger area is used since, as we show, the gain brought by better performances is much higher than the drawback of having a higher error rate. For those applications in which the SDC rate must be minimized regardless of performance (as in some safety-critical applications), we will even identify the optimizations that improve both performance and error rate, providing an execution that is simultaneously faster and more reliable.

We consider four common and effective ways to improve GPUs performance, following the common philosophy of GPU efficient programming, which is to use the highest possible amount of parallel resources without incurring in memory or computing cores saturation and avoiding latencies or dependencies. The optimizations we consider are available to the programmer in current GPU hardware or software design frameworks and do not require extra resources, specific hardening solutions, nor extra effort to be implemented. The GPU performance improvement techniques we consider are: (1) software optimizations, i.e., algorithm-efficient implementations, (2) degree of parallelism improvement, i.e., increase the number of parallel threads reducing the operations in each thread, (4) architectural solutions, such as tensor core and mixed-precision, and (3) compiler optimizations. We measure how the performance improvements impact the execution time and the error rate of realistic applications from various domains and of dedicated microbenchmarks, crafted to stimulate specific computing resources of GPUs. Overall, we consider more than 50 different configurations.

Intuitively, the best performance is achieved on a GPU when its parallel capabilities are fully exploited. Nevertheless, when more operations are executed in parallel, the device error rate might increase. Thus, most of the time, a higher occupation of the GPU hardware delivers higher performance at the cost of a higher error rate. We claim in this paper that, in the vast majority of the cases, for GPUs, the performance gain grows faster than the error rate. In other words, the benefit of better using additional resources (in terms of shorter execution time) is, most of the time, higher than the possible drawback (higher Failure In Time (FIT) rate, i.e., failures in $10^9 h$

of operation). Interestingly, some GPU performance improvement techniques, such as mixed-precision and some compiler optimizations, have the effect of reducing both the execution time and the GPU exposed area, exacerbating the reliability benefit.

While software fault injection has been exploited to evaluate the reliability of codes [1–5], the different hardware utilization that derives from a more efficient software mapping in the GPU architecture can be measured only with beam experiments. In fact, most of the considered solutions to improve performance *do not* modify the code, but rather improve how the code is executed in the hardware. Injecting faults in software is likely to mask most of the underlying hardware utilization effects we intend to highlight. Therefore, to measure the realistic impact on the error rate of the different codes implementations, we take advantage of the results of controlled accelerated neutron beams. We collect and combine the data from experiments performed in the last few years on Fermi, Kepler, and Volta GPUs and highlight the common performance-reliability trade-offs. Then, with dedicated software profiling and architectural analysis we explain the observed trend and, whenever possible, generalize it for other algorithms, configurations, and architectures.

For each code and performance improvement strategy, we calculate the Failure In Time (FIT - errors every 10^9 hours of operation), the execution time, and the Mean Executions/Work Between Failures (MEBF/MWBF - the number of operations completed or data produced correctly between two errors) [6–8]. Whenever possible, we correlate the reliability and performance trend to the implementation characteristics. While the trade-off between performance and reliability has already been studied in CPUs [6–8], we go a step beyond by showing how, for GPUs, in most configurations, performance increases faster than the SDC rate. Higher performance, then, allows the GPU to correctly complete a higher number of executions before the error occurrence.

The main contributions of this work are the following:

1. **We demystify the performance vs. reliability myth** by demonstrating, through beam experiments, that, for GPUs, higher performance does not necessarily imply a higher FIT rate and, even when it does, we can still complete more correct executions between failures.
2. **We assess the impact on GPUs reliability of common code optimizations** by measuring the error rate of various codes executed on GPUs applying 4 widely used techniques to improve their performance.
3. **We give highlights on how to guarantee a more reliable computation** by discussing the correlation between programmer choices, performance, and reliability, with the goal of helping software developers and device architects deciding the optimizations to apply to the GPU applications to use the available hardware in the most reliable way.

The remainder of this paper is as follows. The next Section presents the background and related work that precedes this research. We explain the proposed evaluation in Section 3. Section 4 describes the experiments, codes, and devices used in this work. The evaluation results of each optimization solution are presented in Section 5. Implications and projections of our results are discussed in Section 6. Finally, Section 7 concludes the paper.

2 Background and Related works

In this section, we present the background, the motivation and limitations of our work, and related work on GPUs reliability. We give particular attention to the reliability and performance metrics we use to demonstrate our claim that a more efficient use of GPU hardware is preferable as it increases the number of correct executions.

2.1 GPUs Reliability

Galactic cosmic rays interact with the terrestrial atmosphere generating a flux of particles (mainly neutrons) that reach the ground. About 13 $neutrons/((cm^2) \times h)$ reach the earth's surface [9]. A neutron strike may perturb a transistor's state, generating bit-flips in storage elements or producing current spikes in logic circuits that, if latched, lead to an error [10]. A transient error may not affect the program output (i.e., the fault is masked or benign) or may be propagated through the stack of system layers and produce a failure. A Failure can manifest as a **Silent Data Corruption** (SDC - undetected wrong output), or **Detected Unrecoverable Errors** (DUEs), such as a program crash (application hang) or a device turning not responsive (system crash).

In this paper, we focus on SDCs, only. We decide not to include DUEs in our analysis since it has already been shown that DUEs depend mainly on some hardware components and are not directly correlated with the executed operations [11]. Additionally, DUEs, being detected by definition, are considered less critical than SDCs. While we do not show details about DUEs, we can say that the variation between the DUEs FIT rate we observed among all codes and configurations is, at most, 51% (we measure SDC-performance variations of up to 25 $\times$). This observation also demonstrates that the different SDC trends we discuss are not caused by a different propagation of the same fault (an SDC becoming a DUE or vice-versa), but rather on different reliability behaviors induced by the optimizations.

The reliability of GPUs for safety-critical and HPC applications has already been extensively evaluated through beam experiments [12–16] and fault injection at Register-Transfer Level (RTL) [17, 18], microarchitecture level [2, 3] or software level [5, 19–22]. Additionally, based on the error rate analysis, the fault model, and the fault propagation study, some effective hardening solutions to increase GPUs reliability have been proposed [13, 22–29].

While the reliability of GPUs is well studied and some efficient and effective hardening solutions have been proposed, it is still largely unclear how the different implementations of an algorithm modify the utilization of the GPU hardware and, then, the FIT rate. As one of our contributions, we intend to understand how an increased and more optimized utilization of the GPU computing resources affect the device error rate.

2.2 Reliability Characteristics and Evaluation Metrics

The primary metric to measure the reliability of a device is the Failure In Time (FIT) rate. The FIT rate depends on the amount of resources used for computation, i.e., the sensitive area or *cross section* [10], and their corruption probability of affecting the output, i.e., the Architectural or Program Vulnerability Factors (AVF or PVF) [30, 31].

The AVF and PVF are measured with fault injection, which tracks faults propagation to the output. Fault injection, then, assumes that a fault occurred and identifies whether the fault affects the output. We recall that, as discussed in Section 2.1, we focus on the Silent Data Corruption (SDC) FIT rate of GPUs, i.e., the application output error rate.

The FIT rate, being by definition a *rate*, does not depend on the execution time. The error rate is usually measured with accelerated beam experiments, dividing the number of observed application output errors by the neutron *fluence* (n/cm^2). The fluence is given per cm^2 to ease the error rate calculation, as measuring the exact area of the transistors/device would be unfeasible. Given these definitions, if the same amount of memory is exposed for a time interval t or $2t$, its FIT rate will not change. In fact, under a constant flux ($neutrons/cm^2/sec$), in $2t$, we expect twice the errors and a double neutrons fluence (n/cm^2) to hit the device, giving the same error rate. This is under the assumptions -proved true for terrestrial environment [9]- that the flux of particles is constant and sufficiently low not to have 2 corruptions from 2 different particles in $2t$. On the contrary, if we double the amount of memory exposed for the same time interval t, we expect twice the number of errors but the same neutron fluence (n/cm^2) to hit the device: we are doubling the error rate. Similarly, executing x or $2x$ *sequential* (independent, for simplicity) instructions does not change the code FIT rate. On the contrary, If the additional x instructions 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 performance improvement can impact the GPU FIT rate and reliability.

The interesting aspect, only apparently playing against GPUs, is that a slower execution *does not* increase the FIT rate while using more parallel resources or bigger hardware cores, with a potential benefit on GPUs performance, increases the FIT rate. We will demonstrate in Section 5 that on GPUs most of the common optimizations increase the performance faster than the error rate.

To combine error rate and performance, the Mean Instructions, Executions, or Work Between Failure metrics (MIBF, MEBF, MWBF) were introduced [6–8]. The idea is to consider how many instructions, executions, or workloads can be correctly completed before the output error occurrence. Considering a constant FIT rate, then the faster configuration will have a higher MIBF, MEBF or MWBF.

In this work, we use the SDC FIT metric to quantify the impact of performance improvements strategies on the GPUs output error rate. Then, we use the MEBF and MWBF to identify the configuration that delivers a better performance-reliability trade-off.

2.3 Motivation, Contributions, and Limitations

GPUs architecture is voted to performance, and several previous works, listed in Section 2.1, have highlighted some intrinsic reliability vulnerabilities. Our intuition is that GPUs are so speed-oriented that performance increase faster than the error rate. In other words, faster configurations can be more *proficuous*, i.e., they produce more correct data before the error occurrence. We underline that all the configurations we test are easily selected when designing, programming, or compiling a code

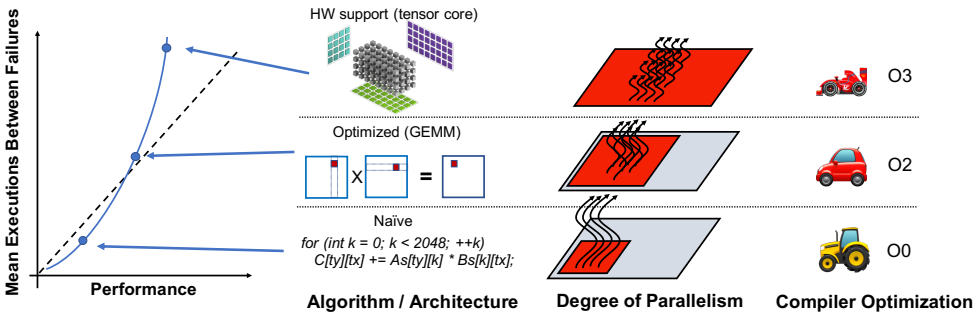


Fig. 1: Overview of the proposed study. Improving the performance of codes executed on GPUs by (1) optimizing the code, (2) taking advantage of dedicated architectural solutions, (3) increasing the Degree of Parallelism, or (4) using aggressive optimization flags will also impact the error rate (not necessarily for worse, as we will show). We aim to identify the configurations that provide a performance gain higher than the FIT rate increase, thus improving the MEBF.

for GPUs. There is no additional effort, hardware, or hardening strategy required to improve reliability in the optimizations we consider. Some previous works have already shown that a given optimization modifies the error rate of GPUs. This is the case of parallelism distribution [8] and the use of mixed-precision [32]. Nonetheless, these works are focused on specific optimization and specific implementation. With this paper we want to highlight the big picture, combining existing data, dedicated experiments, and application profiling to highlight a common trend in different performance improvements in GPUs, so to understand and generalize the observed trends.

The main pragmatic contribution we intend to provide, besides the experimental evaluation of the impact of optimizations on GPUs error rate, is that, as a general rule, on GPUs, if what matters is the amount of data correctly produced, then it is advisable to improve as much as possible the execution performance. The proposed analysis also highlights which optimizations are more likely to bring benefits in terms of the amount of data correctly produced. We also identify some optimizations that improve both performance and reliability and should then be definitely used even in safety-critical applications.

The error rate we present is obtained with accelerated particles beam experiments. This choice is dictated by our interest in presenting a realistic reliability behavior and by the fact that some of the optimizations we evaluate act at a lower level of abstraction than software fault injection. In other words, in some configurations the code does not change, but the compiler or hardware optimization improves performance. For these configurations, the AVF and PVF should experience negligible changes (the source code is the same), while the FIT rate is significantly modified since different hardware resources, with potentially different error rates, are used for computation. For instance, let us consider whether we change the Degree of Parallelism or use tensor cores instead of adders and multipliers. In such cases, we would not need to significantly modify the GPU kernel, provided that the tensor core instructions can

seamlessly replace the sequence of adders and multipliers. However, the error rate is expected to change significantly. Then, a software fault simulation might mask some of the effects we intend to highlight, regardless of the amount of code modifications we make. This is because the fault simulation will not consider the changes in the used hardware resources.

We acknowledge that beam experiments have the limitation of hiding the fault propagation, as faults are observed only when they manifest as output errors. With beam experiments data, we cannot distinguish which level of the hardware or the software contributes to the device error rate. Nonetheless, *we always test configurations that differ of one and only one characteristic*. For instance, when we apply a different compiler optimization, we do not change the code, the inputs, the device, etc. We can then correlate the observed impact on the error rate to the characteristic we have changed. Additionally, as mentioned in Section 2.1, while the SDC rate changes significantly between configurations (up to 25×), the DUE rate changes at most of 50%. This is a strong indication that the different SDC trends observed are not caused by a different fault propagation but by different hardware error rates. Finally, as we discuss four different strategies applied to various codes, it is impossible to give all the details about the implementations. We will discuss the main ones, referencing the related documents where all implementation details can be found.

3 Performance-Reliability in GPUs

In this Section, we give an overview of the idea behind the paper, discussing the reasons why, in GPUs, better performance can provide a higher number of corrected executions before the error occurrence even if they increase the error rate.

When an algorithm is written or the code is implemented or compiled to be executed on a GPU, several possible optimizations can be applied to improve the performance. As shown in Figure 1, these optimizations, by changing the way the algorithm is mapped in the underlying hardware, modify the amount or kind of resources used for computation and thus impact performance but also the error rate. We recall that, since the FIT rate is independent on the execution time (see Section 2.2), the error rate is modified because of the different source code implementation, not because of its shorter execution time.

Figure 1 illustrates the main idea of our paper. It is not meant to be exhaustive but to give a quick overview of our evaluation. FIT and performance are the two main variables we consider, separately or combined in the MEBF metric. The GPU architectural advances, programming strategies, and compiler optimizations are voted to improve the performance. We aim at evaluating the impact of these strategies on the FIT rate and at identifying when the performance gain is higher than the corresponding (possible) SDC FIT rate increase. As schematized in the graph at the left of Figure 1, better performance most of the time implies better MEBF. When the performance-MEBF trend is over the diagonal it means that the benefit of optimizing the code clearly exceeds the drawback (possible FIT increase). We consider four possible ways to improve the GPU performance. We anticipate that most, but not all, performance improvements we evaluate increase the FIT rate.

Algorithm Implementation: there are several ways to write the code that solves an algorithm. On GPUs, the programmer has an additional degree of freedom, which is the parallelism implementation. Most of the techniques to improve the performance of an algorithm are focused on maximizing the GPU occupancy and the locality of data. General Matrix Multiplication (GEMM) [33] is the cornerstone example. By dividing the matrices to multiply in a custom approach it is possible to avoid caches saturation and ensure that each thread works on data already available in the Streaming Multiprocessor (SM). An Optimized GEMM implementation can reduce of 1 order of magnitude the execution time compared to the Naive implementation of GEMM [33]. While an optimized algorithm implementation typically provides significant performance gain, it is hard to predict, a priori, the optimization effect on the error rate. The optimization normally involves code modification and, thus, the executed operations differ, impacting both the raw FIT rate (different functional units have different FIT rates), the AVF (the probability for a fault to propagate to an architectural visible state), and the PVF (the probability for a fault to reach the program output). Nonetheless, the optimizations normally increase the computation *density*, i.e., the number of operations executed in parallel and, then, increase also the FIT rate. This is confirmed by the experimental data we provide and justify in Section 5.1.

Degree of Parallelism (DOP): when a workload needs to be executed on a GPU, the higher the number of parallel threads in which the workload is divided, the higher the DOP. Given a workload, then, the higher the DOP, the lower the number of operations each thread has to execute. If there are no dependencies and the memory/computing resources are not saturated, a higher DOP is expected to significantly improve the performance, especially on GPUs as shown in [8]. Unfortunately, a higher number of parallel threads means that a larger area of the GPU is used for computation and thread scheduling, and, thus, the FIT rate is also expected to increase. We anticipate that, as we show in Section 5.2, depending on how a higher DOP is implemented, it can also decrease the FIT rate (if the memory per thread is reduced).

Architectural Solutions: lately, the GPU architecture has been significantly improved by adding dedicated functional units for operations that are of strategic interest. This is the case of tensor cores [34] and multiple float precisions [35]. A tensor core is a hardware unit that multiplies a matrix tile of NxM in a single instruction. For Volta microarchitecture, this tile has the dimensions of 4×4 [34]. Rather than implementing matrix multiplication in software, as a sequence of ADD-MUL or FMA, the programmer can boost the performance using dedicated cores. Obviously, a tensor core is much larger than a multiplier or an adder [34] and, thus, it also has a higher error rate than a single ADD or MUL. The use of lower float precision (32bit, 16bit, or lower precision) cores is a peculiar optimization as, besides being faster, lower precision cores are also smaller and, thus, are expected to reduce not just the execution time but also the FIT rate of the code, as shown in [32, 36]. The prize to pay is the reduced precision of the output. Whenever this reduction is acceptable, as we show in Section 5.3, using lower float precision than FP64 significantly improves the reliability of GPUs.

Compiler Optimizations: the process of translating the C++ CUDA code into GPU executable is very complex, and it includes several optimizations the programmer

Table 1: Characteristics and features of the evaluated GPUs, Fermi (GTX480), Kepler (K20), and Volta (V100).

	Micro arch.	Node size	SMs	Cores per SM	Main memory	GPU clock (MHz)	CUDA Version
GTX480	Fermi	CMOS 40nm	15	32 FP32/ INT32	1.5GB GDDR5	701	v. 6.0 d. 331.38
K20	Kepler	CMOS 28nm	13	192 FP32/ INT32 64 FP64	5GB GDDR5	706	v. 10.2 d. 440.33
V100	Volta	FFN 12nm	80	64 FP16/FP32 32 FP64 8 Tensor Cores	16GB HBM2	1530	v. 11.3 d. 465.19

can decide to apply. Even if the source code does not change, the compiler optimization modifies the machine code to be executed. As a result, the compiler can impact the instructions distribution and memory utilization, which changes the way the software is mapped in the underlying hardware. As we show in Section 5.4, both performance and FIT rate are impacted by compiler optimization and, generally, the higher the optimization, the more reliable the execution.

4 Evaluation Methodology

To evaluate the FIT/performance trade-off we consider several representative applications from different computing domains executed on three different GPU architectures. In this Section, we describe the tested codes, the considered GPUs, and the neutron beam experiment procedure.

4.1 Codes and Devices

The codes we test are: General Matrix Multiplication (Naive GEMM and Optimized GEMM), LavaMD, Fast Fourier Transform (FFT), Sorting (Quicksort and Mergesort), Needleman-Wunsch (NW), an object detection convolutional neural network (YOLO), and two dedicated microbenchmarks (a sequence of independent ADDs and MULs). We have selected the most suitable codes for each optimization and use Naive GEMM as a common benchmark for all optimizations. Testing all benchmarks for all optimization would be unfeasible due to beam time limitation (we need about 24h of beam time to characterize one code). We have choose an input size to set the execution time of the codes (in their optimized or naive implementation) from 5 ms to 2.5 sec., tuned to guarantee a sufficiently high error rate to collect enough SDCs without allowing more than one neutron to generate a fault in one execution.

We consider three GPUs architectures: Fermi, Kepler, and Volta. Table 1 shows the details of the devices that we consider, Fermi (GTX480), Kepler (Tesla K20), and Volta (Tesla V100) NVIDIA GPUs. Fermi, belonging to an older GPU generation, has computing cores for FP32 and can run efficiently FP64 operations [37]. Kepler supports float and double (FP64) precisions and Volta GPUs support three IEEE754 float point precisions (FP64, FP32, and FP16) plus eight tensor cores.

In some configurations, we also compare the results when the available Single Error Correction Double Error Detection (SECDED) Error Correcting Code (ECC) is turned ON or OFF (when enabled ECC protects the register file, shared memory, caches, and DDR). As already mentioned, we only compare different configurations on one device. Even if our intention is not to compare different devices but how optimizations impact the error rate and the performance on a GPU, in Section 6 we give an overview of how the reliability of GPUs evolved through the years.

GPUs have internal hardware that adjusts frequency based on temperature to avoid damaging the device. In our experiments, the room temperature was controlled and kept constant. Moreover, before starting the experiment, we ran several executions of the code for the warm-up, ensuring a stable device configuration. We have measured the execution time of each execution (each experimental run is composed of thousands of executions) and never observed significant variations (less than 5%).

For all evaluated codes, we used NVIDIA Profiler tools to measure the kernel execution time on the GPU, focusing only on the device code. We repeated the same kernel (or set of kernels) 30× and calculated the average of the measured time.

4.2 Neutrons Beam Experiments

To measure the SDC FIT rate, we take advantage of controlled neutron beam experiments performed in the past few years. In each experiment, that lasts for some days, we test one particular performance improvement, i.e., we always compare data obtained in the same experiment and facility. To cross-validate the results and ensure reproducibility, we tested matrix multiplication on the same GPU at both LANSCE and ChipIR. The difference between the measured FIT rates was well inside the error margins.

Beam experiments were performed at ChipIR facility of the Rutherford Appleton Laboratory (RAL) in Didcot, UK, and at the LANSCE facility of the Los Alamos National Laboratory (LANL) in Los Alamos, US. ChipIR and LANSCE deliver a neutron beam with a spectrum of energy that is suitable to mimic the atmospheric neutron effects in electronic devices [38], allowing to measure the realistic FIT rate of the device executing a code. With the experiments, we measure the probability for a neutron to cause an error in the GPU. Since the spectrum of energy is similar to the terrestrial one and since neutron effects are not cumulative (the fault probability is independent of the number of neutrons that hit the device [10], the error rate calculated with our experiments is the expected error rate caused by terrestrial neutrons on a GPU.

To detect SDCs we execute continuously the code in the irradiated GPU with a known input and check the application output correctness. When the experimental output differs from the expected (fault-free) output we count an SDC. We perform hundreds of thousands executions in the irradiated GPU, collecting at least 100 SDCs per configuration, ensuring a 95% confidence intervals to be lower than 15%. Thanks to watchdogs we are also able to measure the occurrences of DUEs (application crashes, device reboot, device hang). As mentioned in Section 2, we do not report DUEs in this paper since the DUE rate is not significantly affected by the performance optimizations we tested (DUE rate changes of at most 50%).

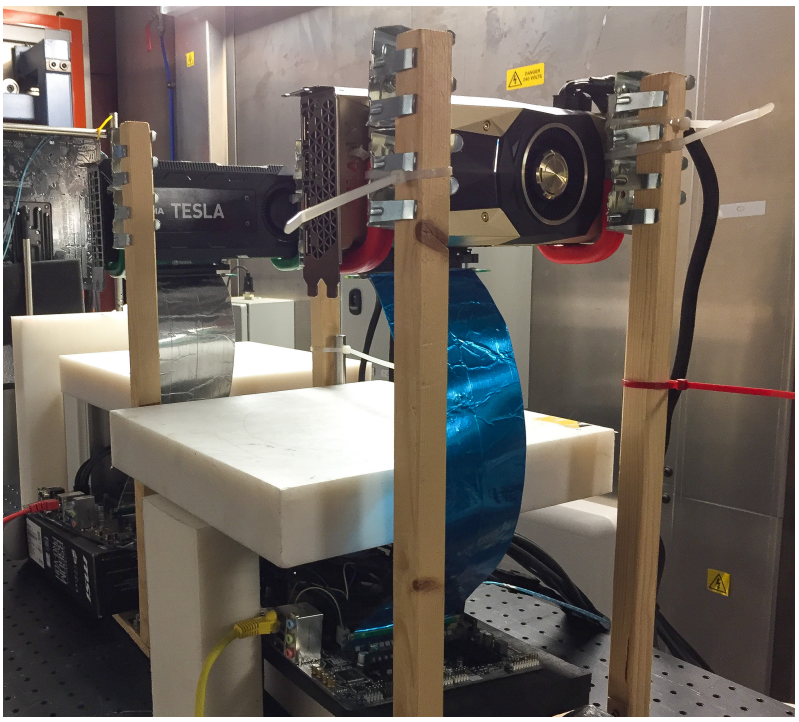


Fig. 2: Portion of the neutron beam setup during one of the test campaigns at ChipIR. Each GPU is aligned with the beam and connected, via PCIe extensors, to a motherboard that is covered with boron plastic to avoid scattering neutrons corruption.

The available neutron flux at ChipIR and LANSCE are about $10^6 n/(cm^2/s)$ and $10^5 n/(cm^2/s)$, respectively, i.e. about 7–8 orders of magnitude higher than the terrestrial flux (3.61×10^{-3} $n/(cm^2 \times s)$ at sea level [9]) Since the terrestrial neutron flux is low, in a realistic application, it is highly unlikely to observe more than a single corruption during the program execution. We have carefully designed the experiments to maintain this property (observed error rates were lower than 1 error per 1,000 executions). Experimental data, then, can be scaled to the natural radioactive environment without introducing artifacts.

While the probability for a neutron to cause a permanent fault in SRAM memory or logic components is negligible [10], DDR can experience permanent or intermittent faults caused by neutrons [39]. Our experimental setup can detect any signs of a permanently damaged GPU during our experiments. During neutron irradiation, we execute additional runs at every output error occurrence to ensure that the observed error is not repeated in the following executions. Moreover, we regularly perform hardware integrity checks (memory checks) when the neutron beam is turned off, and we also log any operating system messages that indicate driver errors related to permanent faults, such as ECC uncorrectable errors. If there is any indication that

the GPU is permanently damaged, we remove the data obtained between the cheeks. Consequently, the data is not used for the paper's conclusion.

Figure 2 shows a portion of the setup installed at ChipIR during one of the multiple test campaigns we performed. We align multiple GPUs with the neutron beam and control them using motherboards. The supporting motherboards and equipment are covered with boron plastic to protect them from scattering thermal neutrons. Only the GPU core is irradiated, that is, DDR and power control circuitry of the board is outside of the beam spot. We are, in fact, interested in the radiation effects on the computing core of the GPU not on the DDR, which has already been extensively studied. For Volta GPUs, in which the HBM2 memories are very close to the GPU core, we either turn the ECC ON or triplicate the data in the main memory to guarantee that all observed errors are generated in the GPU core. Additionally, ECC on GPUs uses Single Error Correction and Double Error Detection (SECDED). Thus, ECC can correct a single error and detect if two bits are flipped in a memory word. During the experiments, all executions where double-bit flips are detected are considered DUEs as the NVIDIA driver raises an exception and is thus removed from the SDC rate analysis. Since with beam experiments, faults are observed only when they manifest at the output, in our evaluation, we focus on the error rate, i.e., the probability for the output to be corrupted. We compare configurations in which one and only one characteristic is changed. Even if we cannot determine the cause of the error, we can still conclude that the only changed characteristic is resulting in a change in the failure rate. It is also interesting to note that the impact of the fault in the output correctness depends on the resource that has been corrupted. Recent works have shown, with low-level fault-injections, that faults in memories have a naive fault model (single or double bit flips) while a fault in the Arithmetic Logic Unit (ALU), scheduler, or special functional units induce a not trivial syndrome, eventually corrupting multiple parallel processes [40–43].

While ChipIR and LANSCE have been shown to provide similar FIT rates, we only compare configurations tested at the same facility to reduce data uncertainty. We always compare configurations that differ of only *one* characteristic. Thus, we can derive that the observed trend is strictly related to the characteristic that we have modified. Moreover, thanks to the application profile and analysis, we can provide a justification for the obtained results.

5 Performance and Error Rate Comparison

In this section, we consider four different solutions to improve GPU performance: (a) algorithm solutions, i.e., code optimization to better fit GPUs computing capabilities, (b) degree of parallelism, to fully use the GPU parallel architecture, (c) architectural solutions, such as tensor core and mixed-precision, and (d) compiler optimization to produce a more efficient machine code. For each evaluated solution, we consider the benefit in terms of performance and the impact on the error rate.

To compare the performance and the error rate of the different codes implementations, we plot the *relative* FIT rate and execution time of the tested configurations. That is, we divide the FIT rate (execution time) of each tested configuration by the

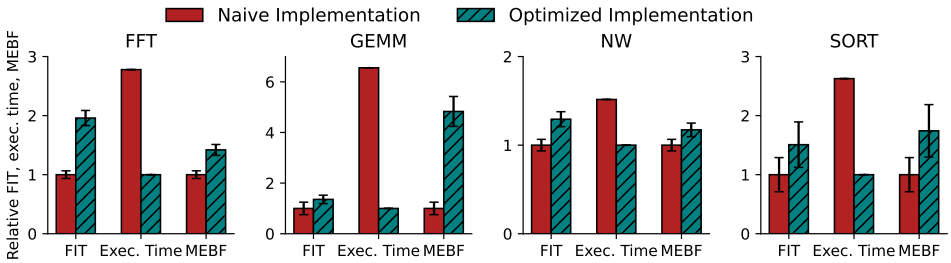


Fig. 3: Relative effects of changing the algorithm implementation on the Failure In Time rate, Execution time, and Mean Execution Between Failures for FFT, GEMM, NW, and Sort executed on the Kepler K20 with ECC OFF. For each benchmark, we evaluate the Naive vs. Optimized implementation. The input and algorithm is the same for each configuration. For all codes, the lower FIT and MEBF (1.0 reference point) is achieved for the naive implementation, while the lower execution time with the optimized implementation (i.e., 860 ms, 7 ms, 1280 ms, and 344 ms for FFT, Optimized GEMM, NW, and SORT, respectively.). Then, increasing the code performance increases the FIT rate, but the performance gain is higher than the error rate increasing. Consequently, more executions are correctly completed before a failure.

lower FIT rate (execution time) we measured. Thus, we show how the different optimizations impact the FIT rate and the execution time. Additionally, we show the relative MEBF (and MWBF for mixed-precision, as the amount of produced data changes). In other words, in each of the following graphs the lower measured value for FIT rate, execution time, and MEBF is plotted with value 1.0. We decided to present relative FIT rate values to ease the comparison between configurations and not to reveal business-sensitive data. In literature, it is possible to find experimental data showing the absolute FIT rate of GPUs [44, 45]. Depending on the device and code executed, the GPU error rate ranges between 10s to 1000s of FITs.

We show how and detail why most of (but not all) the solutions that improve the performance have the drawback of increasing the error rate. However, in the vast majority of the cases, the performance gain is much higher than the error rate increase, leading to a more reliable execution.

All experimentally measured relative FIT rates and MEBF are shown with 95% confidence intervals with a Poisson distribution. The methodology from Quinn [46] was used for this purpose. The error bars for MEBF were then calculated based on the FIT rate error bars alone.

5.1 Algorithmic Solutions

The same algorithm can be implemented in various ways and, on GPUs, it is also necessary to find the parallel implementation that delivers the best performance. The most common code optimizations reduce the memory latencies, increase data

locality, and adapt the computation to the parallel architecture of GPUs, e.g., limiting the interactions between active threads, removing conditional statements and synchronizations.

We have selected four common algorithms for GPUs: 512x512 FFT, 2Kx2K GEMM, 16K NW, and 32M Sorting. For each, we have considered a naive parallel implementation, i.e., not crafted to fully exploit GPU characteristics, and the optimized implementation. Both implementations are taken from available benchmarks suites [47–49]. We underline that our goal is not to discuss how to optimize the coding for GPUs but rather to understand the existing optimizations that impact on the performance and error rate. In the following, we will provide an overview of the differences between the two implementations. Further details about the optimizations can be found in the benchmarks repositories [47–49].

In the Naive GEMM implementation, each thread is responsible for computing one output element, performing 2048 sums and additions, thus saturating the number of registers and caches available in the SM. The Optimized GEMM implementation is taken from the NVIDIA CuBLAS library and, by dividing the matrices in custom tiles sizes and applying specific algorithm policies, ensures that each thread works only on local data and increases the parallel occupancy of the SM [33]. For FFT, in the naive implementation, each thread computes an FFT on 512 points independently from the others, forcing most data to be stored in the main memory, increasing the memory latency. The efficient FFT implementation, developed by Volkov and Kazian [50], divides the 512 FFT computations among 64 threads (2 warps), using only shared data and avoiding synchronizations. For NW (sequencing of DNA), in the naive implementation, each algorithm step, performing a diagonal search, is done in a dedicated kernel while the optimized implementation organizes the input in matrices of 32x32 cells, and each group is assigned to a warp, reducing the number of kernels and improving the efficiency. For sorting, the naive implementation is Quicksort, and the optimized one is Mergesort. Both MergeSort and QuickSort use Bitonic sort to sort batches of array elements. However, the way they are organized can impact their performance. MergeSort's sub-kernel calls are optimized for sorting midsized (key, value) array pairs. In our case, the CPU manages these sub-kernel calls. On the other hand, QuickSort uses CUDA Device Parallelism to manage the sub-kernel calls, which adds some synchronization inside the GPU kernel. This synchronization reduces the performance of the input array sorting.

Figure 3 shows the relative FIT rate, execution time, and MEBF of the naive and optimized implementation of the four algorithms executed on Kepler K20 GPUs, with ECC OFF. As shown, the naive implementation is from 0.5 × (for NW) to 6 × (for Naive GEMM) slower than the optimized one. The FIT rate has the opposite trend for all the codes, with the optimized implementation having an up to 2 × higher error rate than the naive one (for FFT). The higher FIT rate is caused by the denser computation imposed by the optimizations. In fact, by removing the parallel thread stalls (due to synchronizations or memory latencies) or by avoiding the saturation of local memory (caches and registers), we can increase the number of threads that are active in parallel. This, as shown in Figure 3, has the benefit of improving the performance but increases the number of parallel operations, thus increasing the FIT rate.

A promising result is that, for all the considered optimizations, the performance benefit is higher than the FIT rate increase. The MEBF (number of executions completed between failures) is always higher for the optimized version, being up to 5 × higher for Optimized GEMM. A optimized algorithm is then to be preferred, if the goal is to increase the amount of data correctly produced by the GPU. On average, the optimized version of the algorithm allows to correctly complete about 2.2 × more executions than the naive one.

5.2 Degree of Parallelism

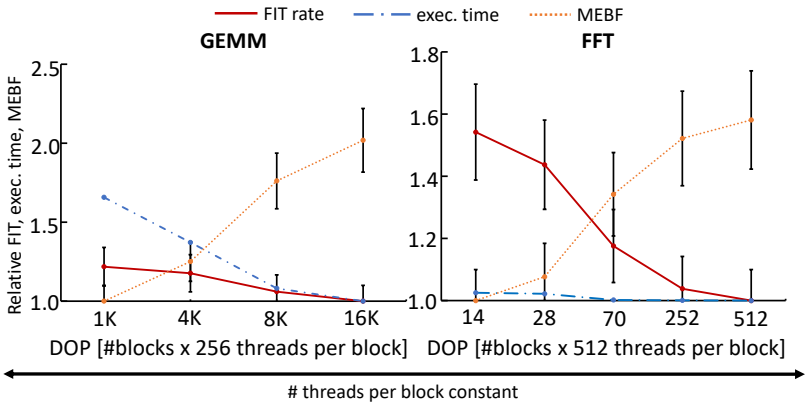
To maximize the performance of a code on a GPU, it is fundamental to increase as much as possible the Degree of Parallelism (DOP) at the thread and instruction level. We consider two applications, 2048x2048 double precision floating point GEMM (Naive version) and a 512x512 FFT, and two microbenchmarks (ADD and MUL), implemented with increasing DOP. The idea is to keep the workload constant, gradually increasing the number of threads in which the workload is divided (reducing the number of operations each thread has to perform). We have used a Fermi GPU with the ECC turned OFF for this experiment.

Figure 4 shows how increasing the DOP influences the FIT, execution time, and MEBF of Naive GEMM and FFT. On the left side of Figure 4 we keep constant the number of threads per block (256 for Naive GEMM and 512 for FFT), increasing just the number of blocks. On the right side of Figure 4 we keep the number of blocks constant (16K for Naive GEMM and 512 for FFT), increasing just the number of threads per block. The number of blocks and threads per block has been chosen to fit the algorithm better and ease its coding.

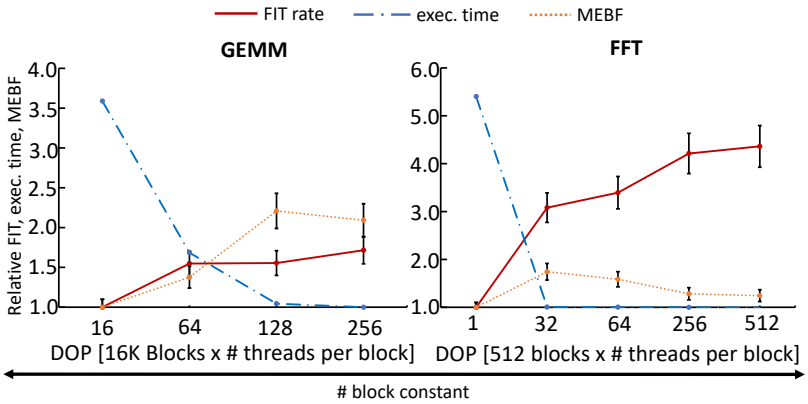
5.2.1 Constant number of threads per block

A very interesting trend can be observed on the left part of Figure 4, where the DOP is increased, keeping the number of threads per block constant to 256 for Naive GEMM and 512 for FFT. In both algorithms, the execution time and the FIT rate *decrease* when we increase the number of blocks. This should not surprise, given that even the configuration with the lowest number of blocks fully exploits the GPU hardware (except for FFT with 14 blocks, which leaves idle 1 of the 15 SMs, to ease the coding of the algorithm), and each block has 256 or 512 threads, ensuring that all 32 CUDA cores in each SM of the Fermi GPU are being used. Thus, the DOP is increased without increasing the GPU exposed computing area but actually reducing the SM memory footprint (which influences the error rate as ECC is OFF). Since the number of operations to perform is constant and we increase the number of blocks (but keeping the number of threads per block constant), each thread has fewer operations to complete and, thus, less memory to allocate. The number of blocks that can be scheduled in parallel is fixed, and thus, the memory footprint at a given time in the GPU is reduced. In other words, the overall memory utilization is the same, but the memory allocated at a given moment is reduced.

In fact, at most 15 blocks can be executed concurrently on the Fermi and, as we increase the DOP, each thread has less operations to perform and, consequently,



(a) DOP Blocks



(b) DOP Threads

Fig. 4: Relative effects of varying the Degree of Parallelism on the FIT rate, execution time, and MEBF of Naive GEMM and FFT executed on the Fermi GPU with ECC OFF. We first maintain the number of threads per block constant (256 for Naive GEMM and 512 for FFT), increasing the number of blocks, and then keep the number of blocks constant (16K for Naive GEMM and 512 for FFT), increasing the number of threads per block. The workload is constant for all configurations. Increasing the number of blocks (left figures) or increasing the number of parallel threads per block (right figures) increases the DOP, reducing the number of operations each thread has to execute. The execution time for the fastest configurations (highest DOP, i.e., 256 threads and 16K Blocks) is 655ms for Naive GEMM and 104ms for FFT.

smaller memory requirements. While the overall memory utilization is the same, at a given time, the amount of exposed memory is then reduced and the fact that there are more blocks to schedule does not increase the exposed memory area, as at most 15 of them can be executed in parallel. As a result, a higher DOP with constant block size implies fewer operations executed and less memory stored in each SM, slightly

reducing the FIT rate. Concurrently, the execution time is also reduced as a higher DOP better fits the GPU architecture. This leads to a significant MEBF improvement, with the configuration delivering a higher performance having a MEBF that is 2 × and 1.6 × higher than the slower one for Naive GEMM and FFT, respectively.

5.2.2 Constant number of blocks

A different trend is observed when the DOP is increased keeping the number of blocks constant (16K for Naive GEMM and 512 for FFT) and increasing the number of threads per block (right figures). In Figure 4b, it is worth noting that we have used thread blocks smaller than 32. This can lead to inactive threads within a warp since the smallest warp size is 32 threads. Although this scenario should be avoided if aiming for the highest performance, we have used it as a theoretical starting point for our analysis. This approach helps highlight the increasing trend of FIT, making it easier to understand our claim.

For both Naive GEMM and FFT, when the number of threads per block is increased, the FIT rate increases, and the execution time decreases. This is because increasing the number of threads per block imposes higher memory and scheduler requirements in each SM (which increases the FIT) and reduces the number of sequential operations of each thread (which reduces the execution time). The number of threads per block should then be carefully engineered. We also consider a configuration that does not fully exploit the GPU parallelism (16 threads per block for Naive GEMM and 1 for FFT). As shown in Figure 4, using fewer computing cores than the 32 available per SM reduces the FIT rate but jeopardizes the performance. Passing from 16 to 64 threads per block for Naive GEMM and from 1 to 32 for FFT, in fact, increases the FIT rate of 1.5 × and 3.2 × , respectively, but reduces the execution time to 1/2 and 1/5, respectively. Intuitively, as 32 CUDA cores are available per SM, instantiating less than 32 threads per block underutilizes the GPU hardware reducing the FIT rate but compromising the GPU parallel efficiency. As shown in Figure 4, underutilizing the GPU is *not* a good reliability solution as the FIT reduction is not sufficient to compensate for the performance degradation. When the instantiated threads are fewer than the available CUDA cores, in fact, both Naive GEMM and FFT have the lowest MEBF.

When the number of threads per block saturates the available cores, as shown in the right side of Figure 4, the FIT rate is basically constant for Naive GEMM (number of threads per block from 64 to 256) and, for FFT, the FIT increase is of about 0.5 × (threads per block are from 32 to 512). The (slight) FIT increase is caused by the higher strain in the scheduler (more threads to manage) and the different memory distribution. For FFT, the FIT increases faster as threads interact with each other (butterfly modules), while in Naive GEMM, as there is no interaction between threads, it is easier for the scheduler to dispatch parallel threads. As a result, when the DOP saturates the number of available computing cores, the FIT rate slightly increases and the performance gain slows down. For Naive GEMM, the best MEBF is achieved when 128 threads per block are instantiated, and it is just 0.2% lower than the configuration that provides the best performance (256 threads). For the FFT, the data exchange between threads reduces the benefit of a higher DOP, increasing the FIT rate faster

than the performance. As a result, the best MEBF is achieved as soon as there are sufficient threads to occupy all available CUDA cores (32).

What we can derive from the proposed DOP analysis is that underutilizing the GPU is never a good idea as, even if the FIT rate is reduced, the performance are jeopardized. When the GPU resources are saturated, the FIT rate normally grows slower than the execution time reduction, increasing the MEBF. Particular attention needs to be given to the block size. Having too many threads in a block can lead to memory latencies or scheduler overcharges that reduce the benefit brought by the faster execution.

Note that the DOP must be combined with an intelligent algorithm design to enhance thread- and instruction-level parallelism. Simply increasing the number of threads or blocks without a proper strategy may result in poor performance. For instance, the Naive GEMM schedules more threads per block than the Optimized GEMM (1024 vs. 256), but it is significantly slower (6x) due to the lack of hierarchical decomposition strategies used in the Optimized GEMM in cuBLAS. In contrast, the Optimized GEMM splits matrix tiles to extract maximum instruction- and thread-level parallelism while minimizing unnecessary memory movements. This approach follows the concepts discussed in Volkov and Kazian [50] and in Volkov and Demmel [33].

While increasing the number of blocks seems a good reliability solution, increasing the number of threads per block modifies the SM occupancy and deserves better attention. To further investigate the GPU reliability dependence on the SM occupancy, we consider two microbenchmarks, ADD and MUL, in which each of the 15 blocks of

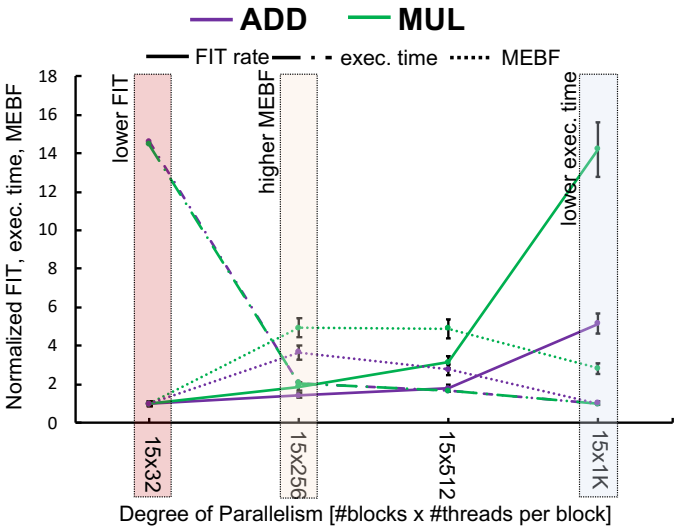


Fig. 5: Effect of increasing the DOP, keeping the workload constant, for ADD and MUL microbenchmarks executed on the Fermi GPU with ECC OFF. The number of blocks is fixed to 15 (one per SM), the number of threads per block increases from 32 to 1024.

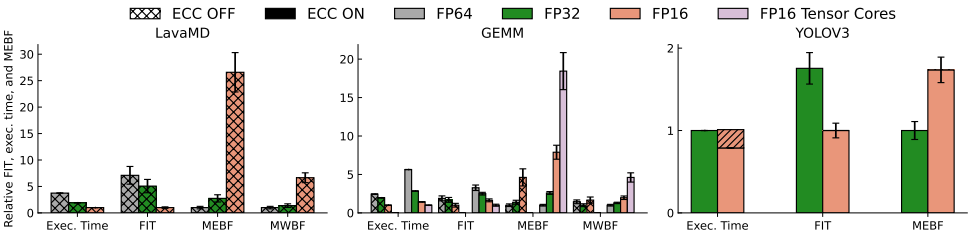


Fig. 6: Relative impact of using mixed-precision or tensor core in the Failure In Time, Execution Time, and Mean Executions or Work Between Failures of the tested Volta V100 GPU. For the FP16 YOLOv3 we have highlighted the time wasted in the casting of data. The use of dedicated reduced precision cores improves the performance and reduces the FIT rate, both when ECC is OFF and ON (ECC ON reduces of 1 order of magnitude the SDC rate). Tensor core is particularly efficient in reducing the execution time, thus exacerbating the benefit in terms of MEBF. The execution time of the most optimized version is 254 ms for GEMM with FP16 Tensor Cores, 1,029 ms for GEMM optimized with FP16, 315 ms for LavaMD with FP16, and 29 ms for YOLOV3 with FP16.

threads need to execute 10,240,000 operations (ADDs or MULs). Unlike Naive GEMM and FFT, for the microbenchmarks, there is no caches utilization and no interaction nor dependencies between threads (each thread executes the operations on dedicated registers). We test 15 blocks to avoid stimulating the block scheduler. We start by instantiating one warp of threads (32 threads) per block, executing the 10,240,000 operations (320,000 operations for each thread). Then, we increase the number of threads by a multiple of 32, gradually reducing the operations each thread has to execute.

Figure 5 shows the relative FIT rate, execution time, and MEBF of both ADD and MUL microbenchmarks as we increase the block size from 32 to 1024 threads, keeping the overall workload constant. It is worth noting that ADD and MUL have different FIT rates since they use different data paths and since GPUs possess dedicated hardware for both sums and multiplications. In fact, even if MUL is much more complex than ADD, the execution time of the two operations is very similar. Moreover, ADD and MUL are very different operations, which means that the probability of the hardware fault propagating to the output is different, further justifying the observed FIT rates.

As observed for Naive GEMM and FFT on the right side of Figure 4, increasing the number of threads (15x32, 15x256, 15x512, 15x1K) gradually increases the FIT rate. The FIT increases because of the higher strain on the warp scheduler and of the data distribution in the SM register file. In particular, as the threads in ADD and MUL benchmarks do not interact with each other, there are no dependencies, and the whole block of threads can be put in the active waiting queue of the GPU and can then be corrupted. As a result, the FIT increases for ADD and MUL at a rate much faster than for Naive GEMM and FFT. That is, while Figure 4 shows that the maximum increase in the FIT rate is up to 6×, the FIT rate increase for the microbenchmarks

ADD and MUL is up to 16×. As shown in Figure 5, increasing the code parallelism still improves the performance, with the configuration with the highest number of parallel threads (15x1K) being the one with the lowest execution time (and highest FIT rate). The configuration with the lowest FIT rate does not provide the higher MEBF, on the contrary, it is the configuration with the worst MEBF because of the very high execution time.

From the data on Figure 4 and Figure 5 we can derive that, when the number of threads per block is increased, the execution time decreases fast until saturating (at 128 threads per block for Naive GEMM, 32 for FFT, 256 for ADD and MUL). This is also the configuration that seems to provide the best MEBF. Further increasing the block size only slightly reduces the execution time but because each thread executes fewer operations, not because the GPU is better utilized. As a general rule, we can state that the number of threads per block should be engineered to fully exploit the GPU parallelism without saturating the SM register/cache.

5.3 Architectural Solutions

The GPUs architects are making available dedicated hardware resources to improve the efficiency and the performance of key operations. This is the case of *tensor core* and *mixed-precision* functional units. In modern GPUs, NVIDIA introduced dedicated functional units to execute operations in various floating point precision (FP64, FP32, FP16, and 8-bit integer). Lower precision units are smaller, faster, and more efficient. On the other hand, there are normally more low precision than high precision units available. In the Volta architecture, there are twice as many FP32 cores as FP64 cores. Nonetheless, the execution time and energy consumption of the FP64 implementation of the code is higher than the FP32, indicating that a globally larger area is required to execute an FP64 code. The user can select the precision of the executed operations to tune the hardware utilization (and the execution time) with the application's needs. Tensor core is an architectural solution to improve the performance of matrix multiplication execution. A tensor core is a hardware unit that multiplies a matrix tile of $N \times M$ in a single instruction. For Volta microarchitecture, this tile has the dimensions of 4×4 [34]. The tensor core is based on Matrix-Multiply-Add (MMA) operation, accumulating the multiplication of two matrices in the output and it delivers up to 9x higher performance than the software implementation of GEMM (sequence of sums and multiplications) on GPUs and up to 47x than a CPU-based system [35].

The strategic choice to design dedicated hardware for mixed-precision and matrix multiplication is justified by the importance of these operations for the training and interference of Convolutional Neural Networks (CNNs). In fact, executing CNNs in low precision significantly reduce the GPU's power consumption and execution time, with a negligible impact on the object detection accuracy [51]. Additionally, more than 70% of operations in a CNN are matrix-multiplication (convolution) related [52] and can benefit from the use of the tensor core. Despite improving the performance, the use of tensor core and mixed-precision functional units, by modifying the hardware used for computation, is likely to impact (for better or worse) the GPU error rate.

Figure 6 shows, for the Volta V100 GPU, the relative FIT rate, execution time, MEBF, and MWBF for LavaMD, and GEMM implemented in FP64, FP32, and FP16.

GEMM is also executed with ECC ON and, in FP16, using tensor core. We also consider YOLOV3 executed in FP32 and FP16. Both YOLOV3 precisions are executed with ECC ON. To have a fair comparison, YOLO was *not* retrained. We simply cast the FP32 data to FP16 and execute the convolutions. This choice guarantees that the FP32 and FP16 implementations share the same PVF and the same weights, thus, the observed different error rate is solely related to the precision reduction. The cast operations reduce about 23% the benefit, in terms of performance, of executing the CNN in lower precision. In Figure 6, we also show the Mean Work Between Failure (MWBF) to have a fair comparison between the different precisions (an FP64 execution produces 4× more work than an FP16 execution).

From Figure 6, it is evident that both reduced precision and tensor core improve the performance. On average, the FP64 implementation has a 3.94 × higher execution time than the FP16 implementation. As mentioned, for YOLO, the execution time of FP32 and PF16 is similar due to data casting. A purely FP16 execution would have a 23% lower execution time. Interestingly, the FIT rate follows the same trend of the execution time, with the FP64 implementation having, on average, a 4 × higher FIT rate compared to the FP16 implementation. This is not surprising, as the lower the precision, the lower the area of the functional unit. Consequently, reducing the precision of the operations significantly improves the MEBF, and even if we consider the MWBF (4 × more bits are output for FP64 than for FP16), the reduced precision execution outperforms the other configurations. In accordance with previous studies [53], when ECC is enabled, we observed that the SDC rate is reduced of about 1 order of magnitude (here we only show relative comparisons). What is interesting, is that when ECC is ON (only shown for GEMM), the benefit in the FIT rate is maintained, indicating that the lower error rate is not solely attributed to the lower amount of memory (with ECC ON memory faults are masked) but also to the smaller sizes of the lower precision functional units. Whenever possible, the use of lower precision units is definitely recommended on GPUs. It is worth noting that such a positive result for reduced precision is achieved only thanks to the dedicated hardware available in GPUs. The opposite trend has been observed if the reduced precision operations are executed in devices that lack dedicated functional units [32].

Tensor core, as shown for GEMM with ECC ON in Figure 6, has a 63% lower FIT rate compared to the software implementation of FP16 matrix multiplication. The tensor core circuit, despite being bigger than an adder and multiplier, is slightly more reliable than the combination of adders, multiplications, and the loop control variables required to implement GEMM in software. Tensor core is particularly efficient in improving the GPU performance, providing (in the FP16 implementation) a 2 × higher MEBF (and MWBF) than the FP16 software GEMM. Whenever possible, then, the use of a tensor core is highly recommended.

Finally, when the ECC is disabled, all memory levels remain unprotected. This means the SDC rate will be directly correlated to the memory used. However, even if the ECC is turned off, how memory values are utilized can still affect the SDC rate. For instance, if two different optimizations have the same memory footprint, the way each code uses the memory values (i.e., the optimization technique) will have an impact on the SDC rate.

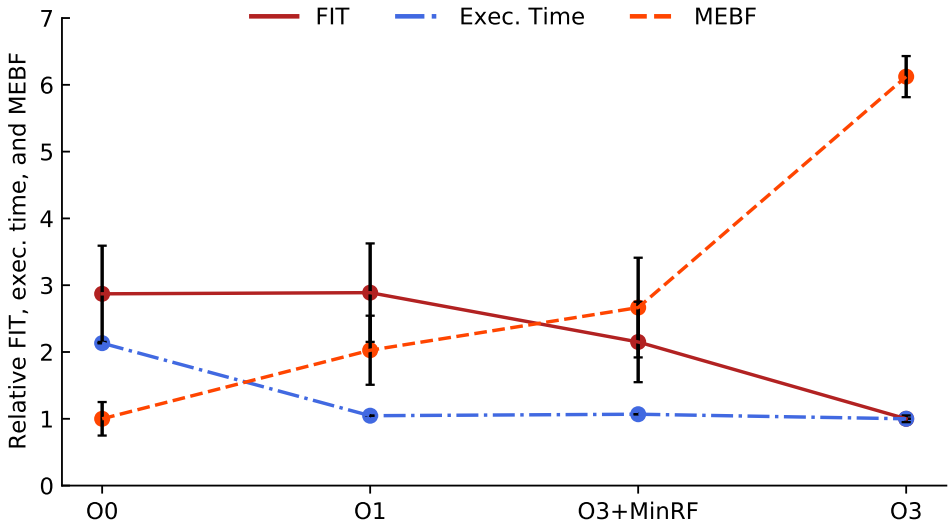


Fig. 7: Relative differences of changing the compiler optimizations of Failure In Time, Execution Time, and Mean Executions Between Failures for GEMM executed on Kepler GPU with ECC ON. The reduction in the number of instructions and the better usage of the GPU resources improves the performance and the reliability, consequently increasing the MEBF. The execution time of the baseline and most optimized version (O3) of GEMM is 233 ms.

5.4 Compiler Optimizations

Figure 7 shows the relative execution time, FIT rate, and the MEBF for the GEMM compiled with four different optimizations, O0, O1, O3+MinRf (O3 restricting the thread register usage to the minimum), and O3. The GEMM kernel uses 25, 16, and 29 registers per thread for optimizations O0/O1, O3+MinRF, and O3, respectively. We show the value for each metric relative to the lower value between the tested configurations. That is, the lower value for each metric in each graph is always 1. Results are obtained on a Kepler K20 GPU with ECC ON. We choose to test the compiler optimization with ECC enabled to focus on the computation errors.

The static compiler optimizations, by changing the number and kind of machine instructions of each thread, can impact the reliability of a code running on a GPU. The O0 flag produces the least optimized machine code, including many instructions that could be simplified or reordered. The O0 version has the highest number of instructions, being 290% higher than the O3 version, the O0 version also includes 21× more memory movements instructions than the O3 and 9× more integer instructions than the O3 version. As shown in Figure 7, these inefficient set of instructions increases both the execution time and the error rate, consequently reducing the MEBF.

The execution time decreases significantly when the first optimization level is applied (O1), but the FIT rate remains similar to O0. As discussed in Section 2.2,

the FIT rate does not depend on the execution time but on how the code uses the resources. O1 significantly reduced the number of machine instructions compared to O0, however, the instruction organization is still inefficient. In fact, the stalls caused by instruction dependence on the code generated with O1 are still higher than the code generated with O3 (19% higher). O1 basically only organizes the instructions differently, but still reaches a similar execution time compared to O3. As the execution time for O1 is almost half the O0 one, and the FIT remains similar, the MEBF for the O1 configuration is $\sim 2x$ the O0 one.

O3 optimization level is the most aggressive optimization that can be applied to the code without approximation, algorithm improvement, or architectural optimizations. We consider the O3 both when the register per thread is limited to 16 (45% fewer registers than only O3 code) and when all registers can be used. The execution time for the two versions are similar (O3+MinRF has a 5% higher execution time), and O3+MinRF has 5.5% more instructions than the O3 version, necessary to perform the register spill to the memory. These additional instructions executed on the O3+MinRF increase the FIT rate. We recall that, as the ECC is enabled, the extra register footprint of O3 compared to O3+MinRF does not increase the FIT rate.

O3 compiled code provides the optimal resource utilization, with the smallest number of machine instructions to reach the code solution (in O3, only 13% of the stalls are caused by instruction dependency). Consequently, O3 has the lowest FIT rate compared to the other configurations, and the MEBF of O3 can be 6.1× higher than O0. Compiler optimizations, then, can significantly impact the reliability of a code, and better resources utilization can reduce the FIT rate while improving the performance. Higher compiler optimizations, then, are definitely to be preferred.

6 Implications and Projections

Figure 8, recalling the graph of Figure 1, resumes obtained results for the different optimizations we have considered. We have considered, for each optimization, the naive, average, and high optimization level (that depends on the optimization itself). Details about each optimization can be found in the previous Sections. We have observed that GPUs seem particularly suitable for fast execution, even when reliability is concerned. The chosen optimization impacts the area of the GPU used for computation and, thus, the SDC FIT rate. Using compiler optimizations (Section 5.4) or taking advantage of architectural solutions such as reduced-precision or tensor cores (Section 5.3) reduces the SDC FIT rates and improves performance. Optimizing the algorithm (Section 5.1) or exploiting the Degree of Parallelism (Section 5.2), on the contrary, has the side effect of increasing the exposed area and, thus, the FIT rate. Nonetheless, in most configurations, the performance gain is higher than the drawback brought by a higher FIT rate. In particular, when the programmer optimizes the algorithm for the GPU architecture, the FIT increases, but a significant performance improvement ensures a higher MEBF (over 2 × more executions correctly performed for any of the considered optimization strategies). The Degree of Parallelism is highly beneficial for both the execution time and error rate when implemented by increasing the number of blocks and keeping the

number of threads per block constant to the maximum that can be executed in parallel in a SM. When the number of threads per block increases, the higher MEBF is achieved when the computing resources are not saturated (i.e., threads do not wait to be executed). Architectural improvements are the most effective optimizations in terms of MEBF. If the programmer chooses to use a lower precision execution, both the FIT rate and execution time are reduced significantly, achieving, on average, over $10\times$ (up to $\sim 25\times$ for LavaMD) a higher number of executions correctly completed between failures. The tensor core does not impact much the FIT rate but bursts the performance, reaching a $\sim 16\times$ higher MEBF than the software GEMM. Finally, for GEMM, higher compiler optimizations are preferred as they reduce the FIT rate and the execution time, guaranteeing an over $6\times$ higher MEBF.

If the goal is to increase the amount of data that can be correctly produced, whenever possible, it is then always preferable to choose the configuration that delivers the best performance and avoids inter SM resources saturation. If the GPU is used in a project where high reliability is required, such as autonomous vehicles, then to be compliant with existing reliability standards such as the ISO26262, it is necessary to reduce the FIT rate, regardless of the performance. We have seen that DOP at blocks level, mixed-precision, and compiler optimizations are excellent ways to (slightly) reduce the FIT rate with a positive side effect of improving (significantly) the performance. We believe that the most impactful result we have presented is that, without particular effort, the programmer has the capability of easily improving (significantly) the reliability of the execution.

In this paper, not to reveal business-sensitive information, we have not explicitly compared the error rate of the three architectures we have tested (Fermi, Kepler, Volta). Nonetheless, we can mention that the reliability of the tested GPUs have significantly improved through the generations. Overall, executing the same code on a Kepler GPU has an experimentally measured error rate of about one order of magnitude *lower* than the Fermi, and the same code on the Volta would be almost one order of magnitude more reliable than on the Kepler. The improved reliability seems much higher than the one reported for similar technologies and even higher than the reported one for the memory structures of NVIDIA GPUs [54]. Obviously, as the technology and architecture of GPU advance, the performance improves [35]. If we also consider the improvement in the performance and efficiency between the different GPU generations, the number of correctly completed executions are likely to increase by almost two orders of magnitude. The intense research to improve GPUs' reliability has borne fruit and paves a promising path for the future.

7 Conclusions

In this paper, we have evaluated the impact in the final code reliability of the most common optimizations available to GPU developers and architects. We have considered software optimizations, Degree Of Parallelism, compiler optimizations, and architectural solutions to speed up a set of applications executed on GPUs belonging to three different generations. The results we have presented, based on extensive neutron beam experiment campaigns, are highly encouraging, as the faster execution

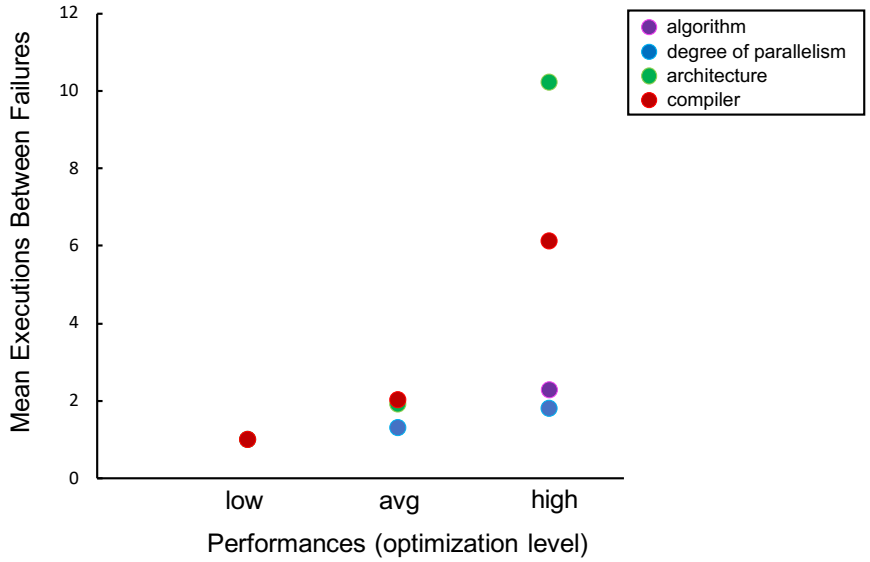


Fig. 8: Overview of the results for the different performance optimizations strategies we have considered. We have averaged the performance/MEBF improvements for a naive implementation (low optimization level), average optimization (not available for algorithm solution), and aggressive optimization (high optimization level).

does not necessarily increase the GPU error rate, and even if it does, the execution time is reduced much more than the FIT rate increase. To improve reliability, then, it is not mandatory to add extra hardware or specific hardening solutions. It is actually not even necessary to sacrifice performance as, in most configurations, we can have it all: a faster configuration that can also produce more correct data.

In particular, the use of dedicated hardware functional units and compiler optimizations are particularly efficient in improving the reliability-performance trade-off. Degree Of Parallelism increase needs a special attention, as saturating the inter-SM resources delivers an MEBF that is (slightly) smaller than the best achievable one. Finally, the technology and GPU architecture evolution play a significant role in improving reliability. This is a promising indication that, once being only voted to applications with low-reliability requirements, GPUs are now becoming devices designed with a particular attention to reliability.

Novel GPU architectures have additional architectural resources to improve the efficiency of computation. While we leave the test of these novel resources and devices as future work, our evaluation suggests that in the vast majority of the cases, a more efficient execution leads to a higher amount of data correctly produced. We think this also holds for the new generations of GPUs. Finally, as a future direction, we can focus on evaluating the impact of specific optimizations that newer architectures like Ampere and Ada bring on reliability. Two examples of such optimizations include new

float and integer precision (e.g., FP8 and INT4) and using approximation flags such as approximations to division or approximation to float math functions.

Acknowledgments. 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. We warmly acknowledge the help of the instrument scientists in mounting the setup and allowing us to perform the experiments remotely during the pandemic year.

8 Declarations

8.1 Ethical Approval

Not applicable.

8.2 Competing interests

Not applicable.

8.3 Authors' contributions

All the authors contributed the same to the article.

8.4 Funding

This project has received funding from the European Union's Horizon 2020 research and innovation program under the Marie Sklodowska-Curie grant agreement No 886202 and 899546, and 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.

8.5 Availability of data and materials

Not applicable.

References

[1] Li, G., Pattabiraman, K., Cher, C.-Y., Bose, P.: Understanding error propagation in gpgpu applications. In: SC '16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, pp. 240–251 (2016). https://doi.org/10.1109/SC.2016.20

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

[3] Yang, L., Nie, B., Jog, A., Smirni, E.: Practical resilience analysis of gpgpu applications in the presence of single-and multi-bit faults. IEEE Transactions on Computers **70**(1), 30–44 (2021) https://doi.org/10.1109/TC.2020.2980541

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

[5] Tsai, T., Hari, S.K.S., Sullivan, M., Villa, O., Keckler, S.W.: Nvbitfi: Dynamic fault injection for gpus. In: 2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), pp. 284–291 (2021). https://doi.org/10.1109/DSN48987.2021.00041

[6] Weaver, C., Emer, J., Mukherjee, S.S., Reinhardt, S.K.: Techniques to reduce the soft error rate of a high-performance microprocessor. In: Proceedings. 31st Annual International Symposium on Computer Architecture, 2004., pp. 264–275 (2004). https://doi.org/10.1109/ISCA.2004.1310780

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

[8] Rech, P., Pilla, L.L., Navaux, P.O.A., Carro, L.: Impact of GPUs parallelism management on safety-critical and HPC applications reliability. In: 2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, pp. 455–466 (2014). https://doi.org/10.1109/DSN.2014.49

[9] JEDEC: Measurement and Reporting of Alpha Particle and Terrestrial Cosmic Ray-Induced Soft Errors in Semiconductor Devices. Technical Report JESD89A, JEDEC Standard (2006)

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

[11] Fratin, V., Oliveira, D., Lunardi, C., Santos, F., Rodrigues, G., Rech, P.: Code-dependent and architecture-dependent reliability behaviors. In: 2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), pp. 13–26 (2018). https://doi.org/10.1109/DSN.2018.00015

[12] Goncalves de Oliveira, D.A.G., Pilla, L.L., Santini, T., Rech, P.: Evaluation and mitigation of radiation-induced soft errors in graphics processing units. IEEE Transactions on Computers **65**(3), 791–804 (2016)

[13] Santos, F.F., Pimenta, P.F., Lunardi, C., Draghetti, L., Carro, L., Kaeli, D., Rech, P.: Analyzing and increasing the reliability of convolutional neural networks on

GPUs. IEEE Transactions on Reliability **68**(2), 663–677 (2019) https://doi.org/10.1109/TR.2018.2878387

[14] León, G., Badía, J.M., Belloch, J.A., Lindoso, A., Entrena, L.: Evaluating the soft error sensitivity of a gpu-based soc for matrix multiplication. Microelectronics Reliability **114**, 113856 (2020) https://doi.org/10.1016/j.microrel.2020.113856 . 31st European Symposium on Reliability of Electron Devices, Failure Physics and Analysis, ESREF 2020

[15] Ito, K., Zhang, Y., Itsuji, H., Uezono, T., Toba, T., Hashimoto, M.: Analyzing due errors on gpus with neutron irradiation test and fault injection to control flow. IEEE Transactions on Nuclear Science **68**(8), 1668–1674 (2021) https://doi.org/10.1109/TNS.2021.3098845

[16] Sullivan, M.B., Saxena, N., O'Connor, M., Lee, D., Racunas, P., Hukerikar, S., Tsai, T., Hari, S.K.S., Keckler, S.W.: Characterizing And Mitigating Soft Errors in GPU DRAM, pp. 641–653. Association for Computing Machinery, New York, NY, USA (2021). https://doi.org/10.1145/3466752.3480111

[17] Condia, J.E.R., Du, B., Sonza Reorda, M., Sterpone, L.: FlexGripPlus: An improved GPGPU model to support reliability analysis. Microelectronics Reliability **109** (2020)

[18] Santos, F.F.d., Condia, J.E.R., Carro, L., Reorda, M.S., Rech, P.: Revealing gpus vulnerabilities by combining register-transfer and software-level fault injection. In: 2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), pp. 292–304 (2021). https://doi.org/10.1109/DSN48987.2021.00042

[19] Ibrahim, Y., Wang, H., Bai, M., Liu, Z., Wang, J., Yang, Z., Chen, Z.: Soft error resilience of deep residual networks for object recognition. IEEE Access **8**, 19490–19503 (2020) https://doi.org/10.1109/ACCESS.2020.2968129

[20] Fang, B., Pattabiraman, K., Ripeanu, M., Gurumurthi, S.: 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, pp. 221–230 (2014). https://doi.org/10.1109/ISPASS.2014.6844486

[21] Anwer, A.R., Li, G., Pattabiraman, K., Sullivan, M., Tsai, T., Hari, S.K.S.: Gpu-trident: Efficient modeling of error propagation in gpu programs. In: SC20: International Conference for High Performance Computing, Networking, Storage and Analysis, pp. 1–15 (2020). https://doi.org/10.1109/SC41405.2020.00092

[22] Yang, L., Nie, B., Jog, A., Smirni, E.: Enabling software resilience in gpgpu applications via partial thread protection. In: 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE), pp. 1248–1259. IEEE Computer Society, Los Alamitos, CA, USA (2021). https://doi.org/

10.1109/ICSE43902.2021.00114 . https://doi.ieeecomputersociety.org/10.1109/ICSE43902.2021.00114

[23] Chen, J., Li, S., Chen, Z.: Gpu-abft: Optimizing algorithm-based fault tolerance for heterogeneous systems with gpus. In: 2016 IEEE International Conference on Networking, Architecture and Storage (NAS), pp. 1–2 (2016). https://doi.org/10.1109/NAS.2016.7549404

[24] Milluzzi, A., George, A., George, A.: Exploration of tmr fault masking with persistent threads on tegra gpu socs. In: 2017 IEEE Aerospace Conference, pp. 1–7 (2017). https://doi.org/10.1109/AERO.2017.7943882

[25] Chen, J., Li, H., Li, S., Liang, X., Wu, P., Tao, D., Ouyang, K., Liu, Y., Zhao, K., Guan, Q., Chen, Z.: Fault tolerant one-sided matrix decompositions on heterogeneous systems with gpus. In: SC18: International Conference for High Performance Computing, Networking, Storage and Analysis, pp. 854–865 (2018). https://doi.org/10.1109/SC.2018.00071

[26] Camargo, E.T., Duarte, E.P.: An algorithm-based fault tolerance strategy for the bitonic sort parallel algorithm. In: 2021 10th Latin-American Symposium on Dependable Computing (LADC), pp. 1–10 (2021). https://doi.org/10.1109/LADC53747.2021.9672590

[27] Condia, J.E.R., Rech, P., Santos, F.F., Carrot, L., Reorda, M.S.: Protecting gpu's microarchitectural vulnerabilities via effective selective hardening. In: 2021 IEEE 27th International Symposium on On-Line Testing and Robust System Design (IOLTS), pp. 1–7 (2021). https://doi.org/10.1109/IOLTS52814.2021.9486703

[28] Hari, S.K.S., Sullivan, M., Tsai, T., Keckler, S.W.: Making convolutions resilient via algorithm-based error detection techniques. IEEE Transactions on Dependable and Secure Computing, 1–1 (2021) https://doi.org/10.1109/TDSC.2021.3063083

[29] Santos, F.F., Brandalero, M., Sullivan, M.B., Basso, P.M., Hübner, M., Carro, L., Rech, P.: Reduced precision dwc: An efficient hardening strategy for mixed-precision architectures. IEEE Transactions on Computers **71**(3), 573–586 (2022) https://doi.org/10.1109/TC.2021.3058872

[30] Mukherjee, S.S., Weaver, C., Emer, J., Reinhardt, S.K., Austin, T.: 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, p. 29. IEEE Computer Society, Washington, DC, USA (2003)

[31] Sridharan, V., Kaeli, D.R.: Eliminating microarchitectural dependency from architectural vulnerability. In: 2009 IEEE 15th International Symposium on High Performance Computer Architecture, pp. 117–128 (2009). https://doi.org/10.

1109/HPCA.2009.4798243

[32] Santos, F., Lunardi, C., Oliveira, D., Libano, F., Rech, P.: Reliability evaluation of mixed-precision architectures. In: 2019 IEEE International Symposium on High Performance Computer Architecture (HPCA), pp. 238–249 (2019). https://doi.org/10.1109/HPCA.2019.00041

[33] Volkov, V., Demmel, J.W.: Benchmarking gpus to tune dense linear algebra. In: SC '08: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing, pp. 1–11 (2008). https://doi.org/10.1109/SC.2008.5214359

[34] Jia, Z., Maggioni, M., Staiger, B., Scarpazza, D.P.: Dissecting the NVIDIA Volta GPU Architecture via Microbenchmarking (2018)

[35] NVIDIA: Nvidia tesla v100 gpu architecture - whitepaper. Technical Report 1.1, NVIDIA (aug 2017)

[36] Luza, L.M., Soderstrom, D., Tsiligiannis, G., Puchner, H., Cazzaniga, C., Sanchez, E., Bosio, A., Dilillo, L.: Investigating the impact of radiation-induced soft errors on the reliability of approximate computing systems. In: 2020 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT), pp. 1–6 (2020). https://doi.org/10.1109/DFT50435.2020.9250865

[37] NVIDIA: Nvidia's next generation cuda compute architecture: Fermi. Technical Report 1.1, NVIDIA (aug 2010)

[38] Cazzaniga, C., Frost, C.D.: Progress of the scientific commissioning of a fast neutron beamline for chip irradiation. Journal of Physics: Conference Series **1021**, 012037 (2018) https://doi.org/10.1088/1742-6596/1021/1/012037

[39] Srour, J.R., Marshall, C.J., Marshall, P.W.: Review of displacement damage effects in silicon devices. IEEE Transactions on Nuclear Science **50**(3), 653–670 (2003) https://doi.org/10.1109/TNS.2003.813197

[40] Dos Santos, F.F., Carro, L., Rech, P.: Understanding and improving gpus' reliability combining beam experiments with fault simulation. In: 2023 IEEE International Test Conference (ITC), pp. 176–185 (2023). https://doi.org/10.1109/ITC51656.2023.00034

[41] Rodriguez Condia, J.E., Rech, P., Santos, F.F.d., Carro, L., Reorda, M.S.: An effective method to identify microarchitectural vulnerabilities in gpus. IEEE Transactions on Device and Materials Reliability **22**(2), 129–141 (2022) https://doi.org/10.1109/TDMR.2022.3166260

[42] Santos, F.F.d., Condia, J.E.R., Carro, L., Reorda, M.S., Rech, P.: Revealing gpus vulnerabilities by combining register-transfer and software-level fault injection. In: 2021 51st Annual IEEE/IFIP International Conference on Dependable Systems

and Networks (DSN), pp. 292–304 (2021). https://doi.org/10.1109/DSN48987.2021.00042

[43] Papadimitriou, G., Gizopoulos, D.: Demystifying the system vulnerability stack: Transient fault effects across the layers. In: 2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA), pp. 902–915 (2021). https://doi.org/10.1109/ISCA52012.2021.00075

[44] Tiwari, D., Gupta, S., Rogers, J., Maxwell, D., Rech, P., Vazhkudai, S., Oliveira, D., Londo, D., DeBardeleben, N., Navaux, P., Carro, L., Bland, A.: Understanding GPU errors on large-scale HPC systems and the implications for system design and operation. In: 2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA), pp. 331–342 (2015). https://doi.org/10.1109/HPCA.2015.7056044

[45] Ito, K., Zhang, Y., Itsuji, H., Uezono, T., Toba, T., Hashimoto, M.: Analyzing due errors on gpus with neutron irradiation test and fault injection to control flow. IEEE Transactions on Nuclear Science **68**(8), 1668–1674 (2021) https://doi.org/10.1109/TNS.2021.3098845

[46] Quinn, H.: Challenges in testing complex systems. IEEE Transactions on Nuclear Science **61**(2), 766–786 (2014) https://doi.org/10.1109/TNS.2014.2302432

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

[48] Nvidia, C.: Cublas library. NVIDIA Corporation, Santa Clara, California **15**(27), 31 (2008)

[49] CUDA Code Samples (2018). https://developer.nvidia.com/cuda-code-samples

[50] Volkov, V., Kazian, B.: Fitting fft onto the g80 architecture. Methodology (2011)

[51] Venkatesh, G., Nurvitadhi, E., Marr, D.: Accelerating deep convolutional networks using low-precision and sparsity. In: 2017 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), pp. 2861–2865 (2017). https://doi.org/10.1109/ICASSP.2017.7952679

[52] Redmon, J., Farhadi, A.: Yolov3: An incremental improvement. arXiv (2018)

[53] Lunardi, C., Previlon, F., Kaeli, D., Rech, P.: On the efficacy of ecc and the benefits of finfet transistor layout for gpu reliability. IEEE Transactions on Nuclear Science **65**(8), 1843–1850 (2018) https://doi.org/10.1109/TNS.2018.2823786

[54] Rech, P., Carro, L., Wang, N., Tsai, T., Hari, S.K.S., Keckler, S.W.: Measuring

the Radiation Reliability of SRAM Structures in GPUS Designed for HPC. In: IEEE 10th Workshop on Silicon Errors in Logic - System Effects (SELSE) (2014)