gpuFI-4: A Microarchitecture-Level Framework for Assessing the Cross-Layer Resilience of Nvidia GPUs

Dimitris Sartzetakis George Papadimitriou Dimitris Gizopoulos
Department of Informatics and Telecommunications
University of Athens, Greece
{sartzet | georgepap | dgizop}@di.uoa.gr

***Abstract*—Pre-silicon reliability evaluation of processors is usually performed at the microarchitecture or at the software level. Recent studies on CPUs have, however, shown that software level approaches can mislead the soft error vulnerability assessment process and drive designers towards wrong error protection decisions. To avoid such pitfalls in the GPUs domain, the availability of microarchitecture level reliability assessment tools is of paramount importance. Although there are several publicly available frameworks for the reliability assessment of GPUs, they only operate at the software level, and do not consider the microarchitecture. This paper aims at accurate microarchitecture level GPU soft error vulnerability assessment. We introduce *gpuFI-4*: a detailed microarchitecture-level fault injection framework to assess the cross-layer vulnerability of hardware structures and entire GPU chips for single and multiple bit faults, built on top of the state-of-the-art simulator GPGPU-Sim 4.0. We employ gpuFI-4 for fault injection of soft errors on CUDA-enabled Nvidia GPU architectures. The target hardware structures that our framework analyzes are the register file, the shared memory, the L1 data and texture caches and the L2 cache, altogether accounting for tens of MBs of on-chip GPU storage. We showcase the features of the tool reporting the vulnerability of three Nvidia GPU chip models: two different modern GPU architectures – RTX 2060 (Turing) and Quadro GV100 (Volta) – and an older generation – GTX Titan (Kepler), for both single-bit and triple-bit fault injections and for twelve different CUDA benchmarks that are simulated on the actual physical instruction set (SASS). Our experiments report the Architectural Vulnerability Factor (AVF) of the GPU chips (which can be only measured at the microarchitecture level) as well as their predicted Failures in Time (FIT) rate when technology information is incorporated in the assessment.**

***Index Terms*—reliability; GPU; transient faults; microarchitecture-level fault injection; silent data corruptions; crashes**

I. Introduction

Silicon manufacturing advancements have established new generations of digital devices with a wealth of transistors that improve the performance and functionality. On the downside, however, the ever-increasing rate of hardware faults in newer technologies can jeopardize the aggressive evolution of CPUs, GPUs, and other processing units. CPUs and GPUs have major architectural differences but their reliable operation can be similarly and significantly affected by transient faults (soft errors), intermittent faults, and permanent faults (hard errors) [1] [2] [3]. Such hardware faults can be caused by several different phenomena related to radiation, process variation, wear-out, aging, etc. [4]. Even though the sources of failure are well-known, it is crucial to precisely evaluate their impact on modern CPU and GPU architectures across the full stack of hardware and software.

The most comprehensive way to measure the vulnerability of the entire system stack (cross-layer vulnerability) including the microarchitecture, the architecture, and the software layers (both user and kernel space) is to determine the Architectural Vulnerability Factor (AVF) [5] of each individual microarchitectural structure during end-to-end program execution [6]. AVF is the probability that a fault (bit flip) in a hardware structure will result in an observable error at the application output. AVF can be measured either using analytical methods such as the Architecturally Correct Execution (ACE) analysis [5] or using fault injection [7]. AVF measurements through statistically significant fault injection provide useful insights for the vulnerability across the entire system stack, but this comes with the expense of long simulation runs. While AVF has been initially proposed for the assessment of reliability in CPUs, it has also been adapted to GPUs [8] [9] [10]. Although several tools and methodologies have been reported in the literature for the evaluation of the vulnerability of CPUs at different layers of abstraction, from the microarchitecture all the way to the software, vulnerability evaluation tools for GPUs are either limited to the software level only [11] [12] [13] [14] or do not support recent GPU architectures [15]. Among these GUFI [15] is the only microarchitecture level framework for GPUs reliability assessment, however, (a) it is incomplete (many essential GPU hardware components are not considered), (b) it is limited to older GPU generations, (c) it supports only single-bit faults, and (d) it is not publicly available.

Microarchitecture-level vulnerability assessment of GPUs is very important during the early design stages and can help to measure the benefits of different error protection techniques against the overheads they incur on an initially unprotected design. As a result, much effort must be devoted to effectively measure a system's vulnerability as early as possible and making appropriate design decisions for error protection. Early decisions on protective mechanisms, on the other hand, are difficult to make because critical factors are unknown at the early phases of a system's design such as the final size of hardware components and the diversity of workloads. As a result, early reliability assessment using microarchitectural simulators and the flexibility it offers for design space exploration is of great importance before the late stages of chip design.

In this paper, we describe a comprehensive GPUs fault injection framework built on top of the latest version of the state-of-the-art cycle-level GPGPU-Sim 4.0 simulator [16], for measuring the AVF of individual hardware structures of GPUs as well as entire GPU chips. Unlike previous efforts [11] [12] [13] [15], our proposed framework can evaluate modern GPU architectures at the hardware level considering a large number of important microarchitectural components, it can inject both single and multi-bit faults and reports the complete cross-layer AVF. Our framework is capable of injecting transient faults on most of the important hardware components of modern Nvidia GPUs: the register file, the shared memory, the local memory, the L1 data/texture cache, and the L2 cache.

We make the following contributions:

1. We propose a new microarchitecture-level fault injection framework (gpuFI-4) for assessing the cross-layer resilience of modern GPUs, which is built on top of the well-known state-of-the-art GPGPU-Sim 4.0 microarchitectural simulator. gpuFI-4 consists of three modules: a fault masks generator, an injection campaign controller and a parser of the logged information. It allows extensive studies with single or multiple faults injected in: (i) different bits of the same entry of a hardware structure, (ii) different entries of a structure, (iii) different hardware structures simultaneously, (iv) all combinations of the above.
2. We report results for single-bit and triple-bit fault (to report significant differences) on important microarchitectural structures to demonstrate how our tool can comprehensively assist design decisions for error protection of individual components unlike other related frameworks in the literature that consider faults only at the software layer (and thus cannot pinpoint the weakest hardware structures). Therefore, our injection framework can serve many different reliability studies.
3. We showcase the effects of soft errors on three different GPU generations (the most recent RTX 2060 of Turing microarchitecture and Quadro GV100 of Volta microarchitecture, and an older generation GTX Titan of Kepler microarchitecture), for the important microarchitectural components and for both single-bit and triple-bit faults.
4. We finally present the predicted Failures in Time (FIT) rate of the entire GPUs (failures per billion hours of operation). By using FIT rates (which incorporate technology failure rate information), we can further extend our analysis on the vulnerability of different GPU generations. The FIT rate can be used to provide essential insights about the vulnerability trends among older and newer GPU architectures for the same workloads.

II. Background & Related Work

A. Background

The Graphic Processing Unit (GPU) was initially designed for real-time graphics. A modern GPU, however, is not only a strong graphics processor, but also a general purpose accelerator that focuses on parallel processing and high data bandwidth. Given that the performance improvement of CPUs has significantly slowed down over the last two decades, GPUs have gained the interest of the industry for general-purpose computing purposes when data parallelism is ample. GPUs could become a very promising contender in high performance computing with rapid increases in both the computational power and the programmability.

Nvidia is one of the biggest GPU providers in the world and its CUDA-capable GPUs are of great performance in the field of GPU computing. GPUs contain streaming multiprocessors (SM), each containing several Stream Processors (SPs). The GPU has a global scheduler (Giga Thread) for distributing the work to the SMs and a host interface to the CPU. Different large memory spaces are available within a GPU chip, having different latencies, storage capacity and access methods. These memory spaces, from the faster to the slower are: the register file (32,768 32-bit registers per SM in Nvidia compute capability devices 2.X), the shared memory/L1 cache (64 KB per SM), the L2 cache (768 KB) and the global memory Graphic Double Data Rate (GDDR) DRAM (1 – 6 GB) [17]. The number of the SMs, SPs and the sizes of the memory spaces can be different from across generations and usually get larger in newer generations.

The streaming processors within one streaming multiprocessor share the constant cache, the texture cache and the instruction unit. Each streaming processor has its own register file for storing data that are frequently used. The register file of each processor is a small on-chip memory that has an extremely short access time. There is also a block memory referred to as shared memory. This is designed for communication across streaming processors and is implemented on-chip with very low access latency.

A GPU has a local memory, a global memory, a shared memory, a data cache, a constant cache, a texture cache, and the registers. The sizes of these memory components vary between different microarchitectures from a few kilobytes (e.g., caches) to some gigabytes (e.g., global memory). The total size of the supported memories that we used in our experiments for RTX 2060, Quadro GV100 and GTX Titan are shown in Table I. Each CUDA thread may access data from them during its execution. Each thread has a private local memory. The local memory space resides in the device memory, so local memory accesses have the same high latency and low bandwidth as the global memory accesses. Registers are private to a streaming processor to store the most frequently used data. Constant cache is designed to cache in the constant memory. Data can be declared as constant if they will not be changed during the execution of the program. Shared

TABLE I. MEMORY STRUCTURES SIZES ACROSS GENERATIONS.

	RTX 2060 (#SMs: 30)	Quadro GV100 (#SMs: 80)	GTX Titan (#SMs: 14)
Register File	7.5 MB	20 MB	3.5 MB
Shared Memory	1.875 MB	7.5 MB	672 KB
L1 data cache	1.98 MB	2.64 MB	N/A
L1 texture cache	3.96 MB	10.56 MB	709.38 KB
L1 instruction cache	3.96 MB	10.56 MB	59.08 KB
L1 constant cache	2.08 MB	5.56 MB	248.92 KB
L2 cache	3.17 MB	6.33 MB	1.58 MB

memory is used to allow streaming processors to communicate. All threads have access to the same global memory and is used for communication between host CPU and GPU, since GPU cannot access the main memory of the CPU. Data that will be handled by the GPU must first be copied to global memory and the results obtained from the GPU must be copied back to the CPU memory with the appropriate API (i.e., cudaMemcpy).

B. GPGPU-sim Overview

GPGPU-Sim is a cycle-level simulator modeling contemporary Nvidia GPUs running GPU computing workloads written in CUDA or OpenCL. The simulator is capable of running either Parallel Thread Execution assembly (PTX) or SASS assembly. The earlier versions of the simulator supported only PTX executions but since PTX is only a virtual ISA (Instruction Set Architecture) and not the actual binary code that runs on the hardware this was a major limitation of the previous versions. In order to lift the limitation, the developers of GPGPU-Sim decided to extend PTX with the required features to provide a one-to-one mapping to SASS. PTX along with the extensions is called PTXPlus in GPGPU-Sim terminology.

The GPU architecture that is modeled by GPGPU-Sim is composed of Single Instruction Multiple Thread (SIMT) cores connected via an on-chip interconnection network to memory partitions that interface to graphics GDDR DRAM. An SIMT core models a highly multithreaded pipelined SIMD processor roughly equivalent to what Nvidia calls a Streaming Multiprocessor (SM) or what AMD calls a Compute Unit (CU). The SIMT Cores are grouped into SIMT Core Clusters. The SIMT Cores in a SIMT Core Cluster share a common port to the interconnection network. A Stream Processor (SP) or a CUDA Core would correspond to a lane within an ALU pipeline in the SIMT core.

GPGPU-Sim 4.0 supports the various memory spaces as visible in PTX. Each SIMT core has 4 different on-chip level 1 memories: shared memory, data cache, constant cache, and texture cache. Table II shows which on chip memories service which type of memory access. Regarding the memory system in GPGPU-Sim 4.0, it is modelled by a set of memory partitions. Each memory partition contains an L2 cache bank, a DRAM access scheduler and the off-chip DRAM channel. The L2 cache (when enabled) services the incoming texture and (when configured to do so) non-texture memory requests. For our analysis L2 cache is configured to service all memory requests. The reader is referred to [16] for a comprehensive overview of the GPGPU-Sim microarchitecture.

C. Related Work

The reliability of GPU architectures has been studied recently, with the starting point of the faults being either at the hardware or at the software. GUFI [15] is an older microarchitecture-level fault injection framework which was built on top of GPGPU-Sim [16]. Unlike GUFI which uses the obsolete GPGPU-Sim 3.0, our gpuFI-4 tool is built on top of the latest simulator version 4.0. Another main difference is that our framework studies transient faults on more crucial hardware components, and thus, our experiments report measurements on a significantly larger area of a GPU silicon than GUFI (18.5MB and 47MB in total for RTX 2060 and Quadro GV100, respectively). Moreover, by using the latest version of the simulator we are capable of testing the most modern GPU architectures as well (GPGPU-Sim 3.0 (partially) supports up to Kepler architecture, while GPGPU-Sim 4.0 supports up to Ampere architecture). Other reliability evaluation approaches that employ microarchitectural simulators like GPGPU-Sim and Multi2sim [18] are also available, although they both measure the Architectural Vulnerability Factor (AVF) of hardware structures using the Architectural Correct Execution (ACE) analysis [8] [10] [19] and not through statistically significant fault injections. As a result, such approaches come with the inherent overestimation of the AVF of the microprocessor structures [6] [20] [21] [22] and cannot provide detailed fault effect classifications (output corruptions vs. crashes for example), which is what our injector does. Software-level fault injections in real Nvidia GPUs have also been studied with tools like NVBitFI [11], SASSIFI [23], GPU-Qin [14], and LLFI-GPU [24]; these tools only focus on the program resilience and do not deliver AVF measurements as in our analysis which allows injection of faults in the actual hardware structures. Table III summarizes the fault injection frameworks in both software and microarchitecture level, showing the major limitations of the previously published tools.

III. GPUFI-4: A MICROARCHITECTURE-LEVEL FRAMEWORK FOR ASSESSING THE CROSS-LAYER RESILIENCE OF GPUS

A. Overview

gpuFI-4 is a complete and extensible framework for reliability evaluation of Nvidia GPU architectures that runs on top of a well-known simulator of GPUs architectures: GPGPU-Sim 4.0. gpuFI-4 considers transient fault injection campaigns on

TABLE II. CUDA SUPPORTED MEMORY SPACES IN GPGPU-SIM.

Core Memory	PTX Accesses
Shared memory (R/W)	shared memory accesses only
Constant cache (Read Only)	Constant and parameter memory
Texture cache (Read Only)	Texture accesses only
Data cache (R/W - evict-on-write for global memory, writeback for local memory)	Global and Local memory accesses

TABLE III. RELATED RELIABILITY FRAMEWORKS AND COMPARISON.

	Layer	GPGPU-sim Version	Multi-bit Support	# of Target Components	GPU Generations
SASSIFI [23]	SW	-	✓	-	2010-2014
NVBitFI [11]	SW	-	✓	-	2012-2020
GPU-Qin [14]	SW	-	✗	-	N/A
G-SEPM [13]	SW	-	✗	-	N/A
LLFI-GPU [24]	SW	-	✗	-	2012-2015
GUFI [15]	uArch	3.0	✗	2	2006-2011
This Work	**uArch**	**4.0**	✓	**6**	**2006-2020**

PTX or SASS mode, using single or multiple bit-flips during the execution of an application as explained in Table IV for each hardware component. The fault injection campaign in a hardware component can be set either for a user-defined kernel invocation or the whole application. We focused our study on CUDA applications running on SASS mode and using single and multiple bit-flips per kernel injection campaigns.

gpuFI-4 consists of two parts: a back-end and a front-end. The back-end is the actual implementation of the fault injection. It has been developed on top of GPGPU-Sim 4.0 and several input parameters have been created for this purpose which are passed through the *gpgpusim.config* configuration file to the simulator. The front-end part is a bash script, which initializes the newly created parameters, executes the campaigns, and collects the results. The following subsections explain the frontend part and what steps should be followed until the execution of the injection campaigns. The backend implementation is discussed in the Section IV.

B. *CUDA application preparation*

gpuFI-4 relies its evaluation process on the printed message of a CUDA application which states whether it succeeded or failed. As a result, the applications should be slightly modified to compare the results of the GPU part execution with either a predefined result file (based on a fault-free execution) or the results that come from the CPU "golden" reference execution and print a custom message in the standard output accordingly. We employ the predefined result file in our implementation for faster execution of our experiments.

C. *Profiling and campaign preparation*

The bash script of the front-end requires several parameters to be configured before the injection campaigns are performed. We can classify these parameters into four abstract groups: (1) the first group contains one-time parameters, and it is called *one time*, (2) the second group contains parameters that need to be initialized once per GPGPU card and are necessary to define values that describe some of the hardware structures; it is called *per GPGPU card* and (3) in the third group, there are parameters that need to be initialized every time we analyze the vulnerability of a new CUDA application or single kernel; it is called *per kernel/application*. (4) Parameters that belong to the fourth group are responsible for executing different injection campaigns; it is called *per injection campaign parameters*. For the last group, per injection campaign, the values of the

TABLE IV. GPUFI-4 TARGET HARDWARE STRUCTURES.

HW Component	Support
Register File	• Single or multiple bit-flips in one or more registers of a thread. • Single or multiple bit-flips in one or more registers of a warp. Meaning that every thread of the warp will be affected with the same injections.
Shared Memory	• Single or multiple bit-flips in a shared memory of one or more blocks. Shared memory in an Nvidia GPU is private per block (CTA) and in that case, a user can perform the same shared memory injections on multiple blocks.
L1 Data Cache	• Single or multiple bit-flips in the L1 data cache of one or more SIMT cores. L1 cache in an Nvidia GPU is private, per-SIMT core and in that case, a user can inject the same errors on multiple L1 data caches.
L1 Texture Cache	• Same as L1 data cache.
L2 Cache	• Single or multiple bit flips.
Local Memory (off-chip)	• Single or multiple bit-flips in a local memory of a thread or a warp. Local memory in an Nvidia GPU is private memory per thread.

parameters corresponding to a component that we are not injecting faults will be ignored.

IV. IMPLEMENTATION

In this section we discuss the backend of the proposed framework, and how gpuFI-4 is implemented on top of GPGPU-Sim 4.0. First, we present the main technical challenges of the simulator that we had to overcome in order to model the transient faults as they were injected on a real GPGPU and then we will discuss how the actual fault injections are implemented on each supported hardware structure.

A. *Technical challenges of GPGPU-Sim 4.0*

One of the main challenges imposed by the simulator is that it consists of three major modules which had to be synchronized: the functional simulator, the performance simulator, and the interconnection network simulator. Our framework is developed in the first two modules. The functional simulator is responsible for executing the PTX or SASS kernels, while the performance simulator is the one that simulates the timing behavior of a GPU. As a result, the task of injecting faults at a hardware structure was a complex one as it had to communicate information between these two modules. We use the performance module to define the timing constraints of the injected faults, while the functional module is used to define the spatial constraints.

Another challenge of GPGPU-Sim 4.0 is that, due to the nature of a simulator, it does not have the actual hardware structures in place or fully allocated at the beginning of kernel execution. In that case, the implementation first had to identify the necessary running elements (e.g., threads, CTAs – Compute Thread Arrays, SIMT cores) to get access to the hardware components on which we want to inject the transient faults.

The third and last major challenge is that the caches in GPGPU-Sim 4.0 are holding only the tag value along with some other information and not the actual data. The data are kept on different memory structures and the connection between the cache line and the data is known later on during cache access. This made the fault injections harder to implement and we had to come up with several hooks during cache access and recognize accordingly if the fault should be injected or not on the actual arrays that hold the caches data.

B. *Fault injection implementation*

In this section, we discuss the procedure of a fault injection on each supported hardware structure, which are the register file, the local memory, the shared memory, the L1 data and texture cache, and the L2 cache. The fault injection takes place at a specific cycle of the application requested by the user.

1) Register File

Each thread on an Nvidia GPU uses a subset of the register file and the simulator does not reserve the registers of an active thread from a hardware structure nor does it make all the registers available from the beginning but it rather allocates them dynamically during its execution. An active thread is a thread that is created and is accessible from the simulator during the application execution until its workload is completed. The tool at a given cycle chooses a random active thread and injects the transient fault at a random register of that thread among the registers allocated to the thread. The ability to target a register, which is not yet allocated from that thread, comes from the fact that the register allocation policy per thread is deterministic and such injections have no effect on the execution. The same technique is used to inject faults on an entire warp but instead of choosing a random thread, the implementation chooses a random warp and applies the same transient faults on all the threads of the warp. This way, the tool achieves randomized fault injection with statistical significance (which of course depends on the number of injections).

2) Local Memory

The same approach as the register file injections applies also in local memory, but for the local memory the granularity is by thread and not by registers.

3) Shared Memory

Each block of threads (CTA – Compute Thread Array) on an Nvidia GPU uses its own instance of the shared memory and the shared memories that are visible from the simulator are the ones that their block is active. An active block is a block that is created and accessible from the simulator during the application execution until its workload is completed. The framework at a given cycle chooses one, or multiple, if requested, active blocks and it proceeds with the fault injections on their assigned shared memory. If multiple blocks are requested then the same fault injections will occur on each shared memory.

4) L1 data / texture cache

The L1 data cache per SIMT core is private in an Nvidia GPU. The tool at a given cycle first chooses a random SIMT core among the SIMT cores that a user has defined as an input parameter. Then the cache line of that core's L1 data cache can be retrieved based on the bit that we want to flip. That bit can be either in the tag or in the data part of the cache line. In the first case, we can easily inject the error (flip the bit) into the tag. In the second case and only if the cache line is valid, then we create a fault injection hook. This is because the connection between the cache line and where the data lives is known upon cache access. That hook is activated every time we have access to the aforementioned cache line. When there is read access then if there is a hit and the bit that we want to flip is between the data bits, we perform the fault injection in the retrieved data and if it's a miss then we completely deactivate that hook since the cache line is going to be replaced. When there is write access, then the hook gets deactivated if it's a hit. On a write miss, we are not doing anything since the L1 data cache has write no-allocate write miss policy [16]. For multiple bit-flip injections the procedure is the same for each bit.

5) L2 cache

The same approach is applied as in the case of the L1 data/texture cache with the difference that the L2 cache is public to all of the applications. Internally the simulator splits the L2 cache into banks where each bank is assigned in a memory partition [16]. For that reason, the simulator creates an abstraction and treats the L2 cache as a single entity where the first N lines of the cache belong to the first bank with zero identification and so on. With that said, the range of the bits that we can flip is within the total size of the L2 cache. An important thing to note is that the injection hooks of that cache are working only on local, global, and texture data and not for instruction and constant data. This is due to some problems that appeared with the instruction and constant data caching as explained below.

C. *Miscellaneous*

1) L1 constant/instruction cache

These caches were not implemented for fault injection campaigns and will be added in future version of our tool after resolving some technical limitations. For the L1 constant cache, during the development, we found out that the connection between a cache line and the corresponding data was impossible to locate, hence the hooks could not work properly in this case. For the L1 constant cache, there is no connection between a cache line and the corresponding data, and thus, the hooks could not work properly in this case. Luckily, the issue is propagated only to the performance part (constant cache hits/miss statistics) and does not affect the execution of the application. For the L1 instruction cache similar limitations apply.

2) Cache line and tag

A cache line in general consists of the data bits and some extra bits like tag/dirty/valid, bits for the replacement policy and more. Since the simulator does not have a real hardware structure for caches, this framework is capable of modeling an abstract view of the cache row as if there were tag bits before the data bits. This gives us the ability to have more accurate results in our experiments. We do not take into account other bits because we decided to make the implementation simpler, and we believe that the impact on the results would be negligible because of their size. The fraction of those extra bits is very small compared to the whole cache and so the probability of injecting a transient fault is very low. Note that the tag length that we were able to include consists of 57 bits.

V. Methodology & Benchmarks

A. Methodology

gpuFI-4 evaluates the Architectural Vulnerability Factor (AVF) of each kernel and each hardware component. To measure the Architectural Vulnerable Factor (AVF_{GPU}) of an Nvidia GPU chip during the execution of a CUDA application, we first measure the AVF for each application's kernel (AVF_{kernel}) independently, and then we compute their *weighted* arithmetic mean using the kernel execution cycles as weights. In the measurement of AVF we also take into consideration the sizes of every hardware structure as we explain below.

The AVF_{kernel} measurement exploits the features of gpuFI-4, which supports fault injection in the GPU register file, the local memory, the shared memory, the L1 data/texture cache, and the L2 cache. It is calculated by dividing the sum of products, where each product is between the structure failure ratio ($FR_{structure}$) and its corresponding hardware structure size, by the size of all the previous hardware structures combined. The aforementioned structure failure ratio is calculated simply by dividing the number of fault injection experiments on a hardware component that results in any failure by the total number of injected faults.

$$FR_{structure} = \frac{\#Fault\ Injections\ leading\ to\ Failure}{\#Total\ Fault\ Injections} \tag{1}$$

$$AVF_{kernel} = \frac{\sum_{i \in \{structure\}} FR_i \times Size_i}{\#Total\ Size} \tag{2}$$

$$wAVF = \frac{\sum_{i \in \{kernel\}} AVF_i \times Cycles_i}{\#Total\ cycles\ of\ all\ applications} \tag{3}$$

One of the main drawbacks of modeling with GPGPU-sim (also mentioned in [15]), is that each thread of a kernel constructs and accesses its own register file and doesn't reserve a set of registers from a real physical register file that would be constructed once for each SM (this would have been a more convenient model for reliability assessment). Moreover, in GPGPU-sim each CTA that is assigned to an SM uses its own instance of shared memory and doesn't occupy a subset of a unified shared memory within an SM (this would have been also a better model for injections). To overcome these two modeling issues of GPGPU-sim, in our analysis for the register file and the shared memory, we define a *derating factor* for each of these two structures: df_reg and df_smem. To estimate the final AVF of the register file and the shared memory, we must multiply each factor with the relative percentage of failures [15]. We take into consideration the dynamic allocation/deallocation of each thread of a kernel and as a result the dynamic allocation/deallocation of CTAs. That means that the number of running threads and CTAs in an SM are not fixed or stay the same throughout the execution of a kernel. That being said, for the running number of threads and CTAs in an SM, we get their mean values instead.

The df_reg is an intuitive quantification of the fraction of a GPU physical register file that we can target in a given cycle during the execution of a given kernel (therefore the remaining area of the register file in not vulnerable). It depends on:

- **#REGS_PER_THREAD**: the number of registers that a thread uses during the execution of a kernel,
- **#THREADS_MEAN**: the mean number of running threads in an SM during the execution of a given kernel,
- **#REGFILE_SIZE_SM**: the number of registers in the register file of an SM.

$$\text{df\_reg} = \frac{\#REGS\_PER\_THREAD\ x\ \#THREADS\_MEAN}{\#REGFILE\_SIZE\_SM}$$

The df_smem is an intuitive quantification of the fraction of shared memory that we can target in a given cycle during the execution of a given kernel (therefore the remaining area of the shared memory in not vulnerable). It depends on:

- **#CTA_SMEM_SIZE**: the size of shared memory that is used by a CTA of a kernel,
- **#CTAS_MEAN**: the mean number of running CTAs in an SM during the execution of a given kernel,
- **#SMEM_SIZE**: shared memory size in an SM in bits.

$$\text{df\_smem} = \frac{\#CTA\_SMEM\_SIZE\ x\ \#CTAS\_MEAN}{\#SMEM\_SIZE}$$

B. Fault Effects & Benchmarks

The fault injection campaign can be easily executed by simply running the bash script. The script eventually will go on a loop (until it reaches #RUNS cycles), where each cycle will modify the framework's new parameters at *gpgpusim.config* file before executing the application. Since our tool is implemented on top of GPGPU-Sim 4.0, the steps of setting up the backend are the same as setting up the GPGPU-Sim 4.0 and can be found in [16].

After completion of every batch of fault injections, a parser post-processes the output of the experiments one by one and aggregates the results. The final results are printed when all the batches have finished and the script quits. The parser classifies the fault effects of each experiment as Masked, Silent Data Corruption (SDC), Crash, Timeout, or Performance. Such fault effects are used in several injection-based studies.

Masked: The application runs until the end and the result is identical to that of a fault-free execution.

Silent Data Corruption (SDC): The behavior of an application with these types of faults is the same as with masked faults but the application's result is incorrect. Such faults are the more severe as they occur without any indication that a fault has been recorded (an abnormal event such as an exception, etc.).

Crash: In this case, an error is recorded and the application reaches an abnormal state without the ability to recover.

Timeout: The simulation did not finish within a certain amount of time, equal to two times the fault-free execution time.

Additionally, we use the term "**Performance**" as a fault effect which is nothing but a Masked fault effect, but the total cycles of the application are different from the fault-free execution. We do not consider the Performance fault effect in our AVF results, since they do not affect functionality.

In the context of our reliability evaluation, we use a set of 12 different applications from Rodinia benchmark suite [25] and from Nvidia CUDA SDK [26]. These benchmarks are: *Hot Spot (HS)*, *K-Means (KM)*, *Speckle Reducing Anisotropic Diffusion v1 and v2 (SRAD1 and SRAD2)*, *Lower Upper Decomposition (LUD)*, *Breadth-First Search (BFS)*, *Pathfinder (PATHF)*, *Needleman-Wunsch (NW)*, *Gaussian Elimination (GE)*, *Backpropagation (BP)*, *Vector Addition (VA)*, *Scalar Product (SP)***.**

VI. Experimental Evaluation

In this section, we discuss how gpuFI-4 is used and we present the results of an extensive reliability and performance evaluation for all applications listed above.

A. Experimental Evaluation Methodology

To inject faults on a kernel, we consider all its invocations together (i.e., all dynamic instances of a static kernel); otherwise, it would be extremely time consuming to examine every invocation individually. This was possible by creating the input cycle file to match the cycles of all the invocations of the kernel. We also had to provide as an input, the SIMT cores that all the invocations use so we know which L1 caches we need to target. In general, for every static kernel of an application we performed an injection campaign on every supported hardware structure. Every injection campaign (either for single-bit or triple-bit faults) is performed using 3,000 application executions, in which either a single-bit or triple-bit are flipped on each execution. This number comes from the formula of [7] and results in a statistically significant number of fault injection with confidence level 99% and error margin less than 2%. Table V shows the microarchitectural parameters of each Nvidia card employed in the study.

Table V. Microarchitectural Parameters for RTX 2060, Quadro GV100, and GTX Titan.

	RTX 2060	**Quadro GV100**	**GTX Titan**
SMs	30	80	14
Warp size	32	32	32
Maximum Threads per SM	1024	2048	2048
Maximum CTAs per SM	32	32	16
Registers per SM (size per register: 4 bytes)	65536	65536	65536
Shared Memory per SM	64 KB	96 KB	48 KB
L1 data cache size per SM	64 KB	32 KB	N/A
	67.56 KB*	33.78 KB*	N/A
L1 texture cache size per SM	128 KB	128 KB	48 KB
	135.13 KB*	135.13 KB*	50.67 KB*
L1 instruction cache per SM	128 KB	128 KB	4 KB
	135.13 KB*	135.13 KB*	4.22 KB*
L1 constant cache per SM	64 KB	64 KB	12 KB
	71.13 KB*	71.13 KB*	17.78 KB*
L2 cache size	3 MB	6 MB	1.5 MB
	3.17 MB*	6.33 MB*	1.58 MB*

** With 57 tag bits per cache line*

There are some important aspects relevant to our experiments worth mentioning at this point. Firstly, even though commercial Nvidia GPU chips incorporate ECC protection the GPGPU-Sim does not model it and thus our tools allows a complete investigation of the reliability of a completely unprotected GPU chip. Secondly, we had to use SM compute capability < 2.0 since the simulator did not support the PTXPlus mode otherwise. In future work, we will investigate the case of employing the tracing capabilities of AccelSim [27] in our injector so that newer SASS versions can be used.

B. Fault Effects Breakdown of the Register File

In GPUs, the register file is the largest storage component, and thus, the most critical one regarding the vulnerability. Fig. 1 presents the detailed AVF results for the register file of all three cards we use in this study and for twelve different benchmarks. In Fig. 1, we can see not only the total vulnera-

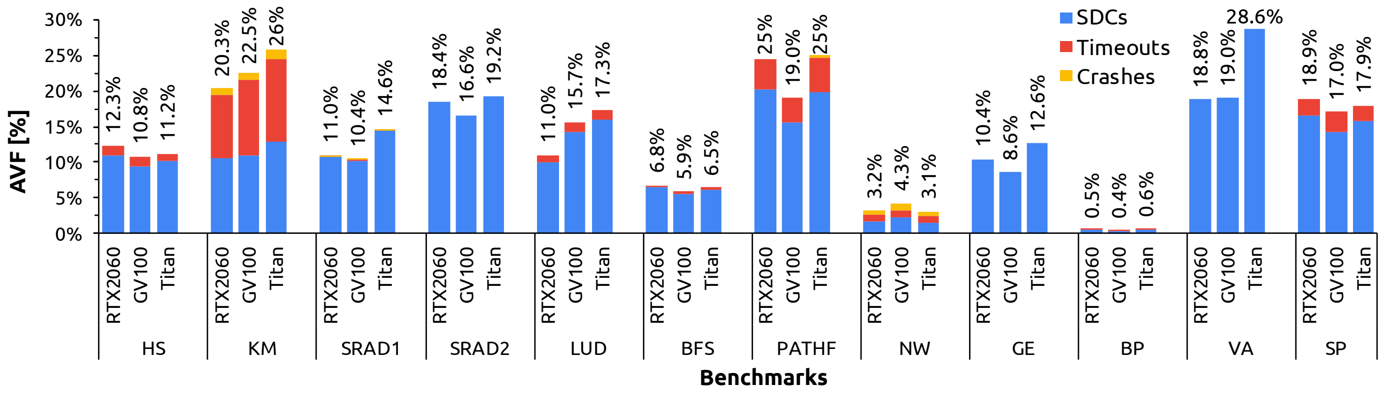


Fig. 1. Fault effects breakdown of register file for all three cards (RTX 2060, Quadro GV100, and GTX Titan), for all twelve benchmarks.

bility of each benchmark for each card, but also breakdown in the four different fault effect classes, which is very important since hardware architects can take the appropriate decisions about the protection schemes depending on the fault effects. Specifically, in this graph we can see that the dominant fault effect class for all the benchmarks and GPU generations is the SDC. There are also some benchmarks (HS, KM, LUD, PATHF, NW, and SP) which show a great number of Timeouts, however, Crashes are practically zero in most cases.

Another observation is that the AVF differences among different GPU generations are very small in most cases. This means that most applications are very sensitive to transient faults, and the sensitivity range to these faults primarily depends on the behavior of the application. For example, as we can see in Fig. 1, the BP benchmark shows nearly zero vulnerability of the register file for all GPUs, while on the other hand, KM benchmark is consistently the benchmark with the highest vulnerability in all chips. We also present the individual structures vulnerability breakdown on the total program's vulnerability, for two benchmarks. Fig. 2 shows the contribution of the different hardware structures on the total AVF for SRAD2 and HS benchmarks. Each piece of the pies is the AVF portion of the overall benchmark AVF that is attributed to each structure.

C. *AVF Results for each GPU generation*

Fig. 3 presents the weighted AVF (wAVF) and the occupancy (red dots) of each workload for the three GPU generations used in this study. The wAVF is calculated based on equation (3) discussed in the previous section. The average warp occupancy shows the ratio of active warps to the maximum number of warps supported on a multiprocessor of the GPU. We collect the warp occupancy from the GPGPU-Sim output of every static kernel, and when there are multiple invocations of a static kernel with different number of threads, we calculate it as a mean value. Afterwards, to compute the average warp occupancy of an application we weight the warp occupancy with the ratio of the static kernel's cycles over the application's cycles and then add the individual weighted warp occupancies of all static kernels.

Naturally, as Fig. 3 shows, different benchmarks have different AVF values. However, the vulnerability trends of the benchmarks among different GPU generations are virtually the same. For example, the SP benchmark is more vulnerable than VA and BP in all three GPUs in our study. Another important observation, which applies to all GPUs, is that benchmarks

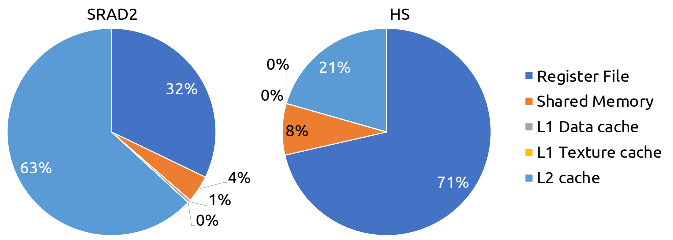
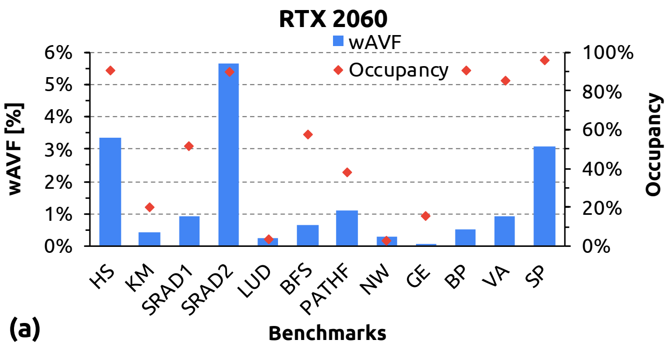
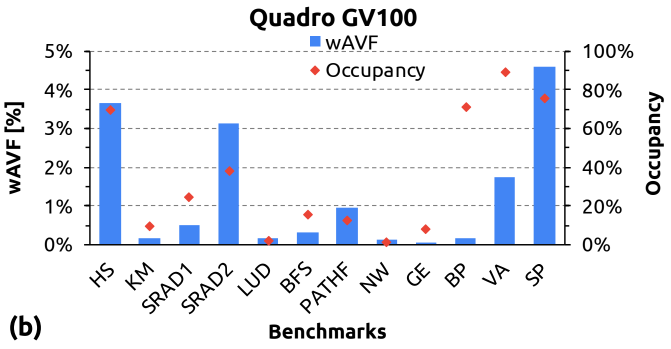


Fig. 2. Breakdown of the hardware structures contribution on the total vulnerability (AVF) for two different benchmarks.





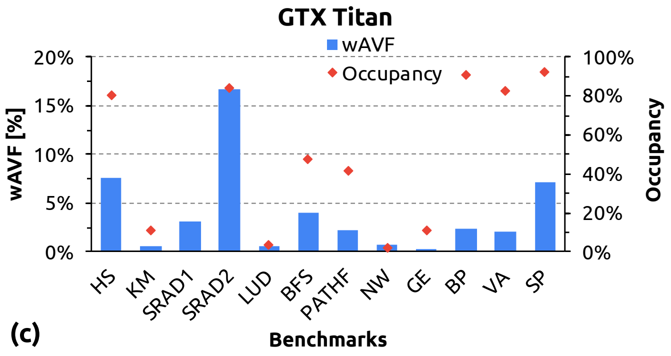


Fig. 3. Total GPU chip AVF results for (a) RTX 2060, (b) Quadro GV100, and (c) GTX Titan, for all benchmarks used in this study, for single-bit faults. The red dots in the graphs present the occupancy for each different GPU generation and for each different workload.

with higher occupancy tend to show higher vulnerabilities. For example, the SRAD2 shows increased occupancy compared to SRAD1, which has increased occupancy compared to KM. The same trend applies for their vulnerabilities. Of course, this trend applies for most of the benchmarks, but not for all benchmark combinations.

D. *Performance Fault Effect*

Faults in a GPU hardware structure can be masked either at the microarchitecture level or at the software level, and thus, the fault will not affect the execution of the application. However, there are also faults, which do not affect the functionality of the program, but can result in performance degradation. The reason in such a case is because the fault can affect the program's execution flow only. gpuFI-4 can reveal such kind of fault effect in GPUs, which only a microarchitecture-level reliability evaluation framework like gpuFI-4 can evaluate with high accuracy. Fig. 4 shows the magnitude of Performance fault effects on each application for RTX 2060. It is clearly shown in Fig. 4, that the Performance fault magnitude can be as high as 8.6% of the total masked faults (i.e., faults that do not affect the functionality), while on average it can be nearly 4% for all applications. The same observation is also valid for the other two GPU cards used in this study; however, we omit the graphs due to space limitations. Specifically, for Quadro GV100, the Performance fault effect can be as high as 16.2%, while for GTX Titan it can be as high as 12.2%.

E. *Single-bit Flip vs. Triple-bit-Flip*

As we discussed earlier, the proposed microarchitecture-level framework for GPUs can accurately evaluate the vulnerability of applications on any modern GPU architecture, not only for single-bit faults, but also for multiple bit faults. Recent studies have shown that although multiple-bit faults (e.g., double-bit, triple-bit, and so on) can show relatively small vulnerability difference compared to the single-bit faults, their contribution is very important for assessing the hardware reliability to soft errors [28] [29]. The reason is that in most modern microarchitectures, in which the technology nodes are gradually reduced, the probability of a multi-bit fault to affect the program's execution is continuously grown. Thus, it is essential to study not only single-bit faults, but also multiple-bit faults. To this end, in Fig. 5 we present the breakdown of fault effects of triple-bit faults in RTX 2060. As shown in Fig. 5, the trends among different fault effects for each benchmark is consistently the same between single-bit faults (see Fig. 1) and triple-bit faults (see Fig. 5) for RTX 2060.

To demonstrate the validity of evaluation results of gpuFI-4, we also present in Fig. 6 a comparison between single-bit and triple-bit fault injections for RTX 2060. As shown in Fig. 6, the AVF of triple-bit faults is around two times the AVF of single-bit faults in most of the benchmarks. The tool can be used for any other cardinality of transient faults; we show the comparison between single-bit and triple-bit so that differences are more visible.

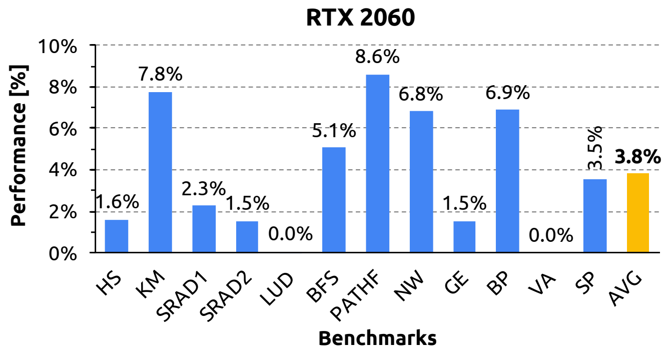


Fig. 4. Single-bit faults that result in performance degradation but the functionality of the application is correct.

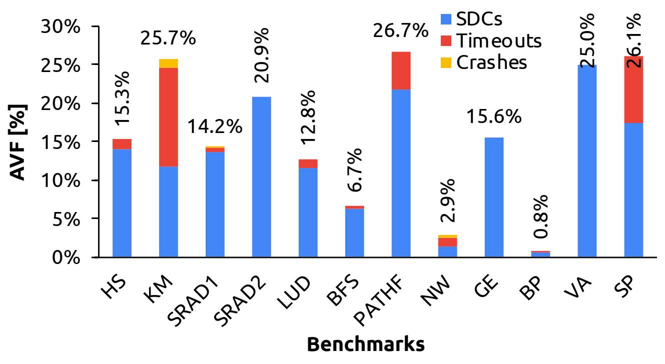


Fig. 5. Fault effect breakdown for triple-bit faults in RTX 2060.

F. *Failure in Time (FIT) Rates*

Failures in Time (FIT) rate of a device estimates the number of failures that can be expected in one billion (10^9) device-hours of operation. For each hardware structure of a microprocessor, a different FIT is computed using the formula in equation below.

$$FIT_{struct} = AVF_{struct} \times \text{raw } FIT_{bit} \times \#Bits_{struct}$$

The FIT of the structure depends on three parameters: (1) the FIT_{bit} (or raw FIT) rate, which is determined by the fabrication technology and the operational conditions and expresses the fault probability of a single bit, (2) the number of bits of the structure, and (3) the AVF of the structure, which is affected by the microarchitecture and the executed workload. The raw FIT rate expresses the rate of transient faults introduced in the component, while the AVF is the derating factor that quantifies how many of these bit upsets will lead to a failure. The product provides the FIT rate of a particular hardware structure. The FIT rate of the entire GPU is calculated by adding the individual FITs of the structures. For the calculation of the FIT, we use the raw FIT rate per bit for each GPU card, as described in [21] [30] [31]. Specifically, the raw FIR rate for RTX 2060 and Quadro GV100 (which are fabricated at 12nm) is 1.8 x 10^{-6} and for GTX Titan (which is fabricated at 28nm) is 1.2 x 10^{-5}. Fig. 7 shows the FIT rates for all GPU generations and benchmarks used in this study. For most of the benchmarks we can see that the older the GPU is, the larger the overall FIT values. This is expected, since older GPU generations (i.e., GTX Titan) were

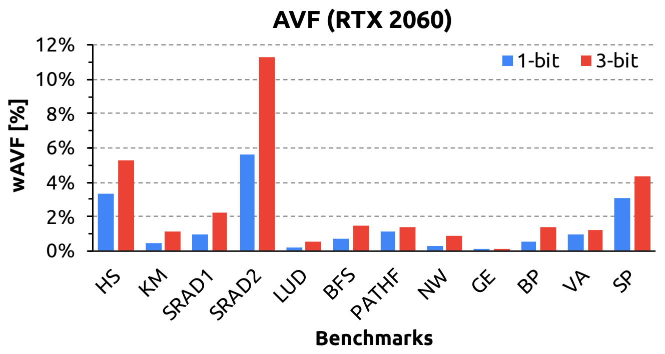


Fig. 6. Weighted AVF for single-bit (blue color) vs. triple-bit (red color) fault injections for GTX 2060 and for twelve benchmarks.

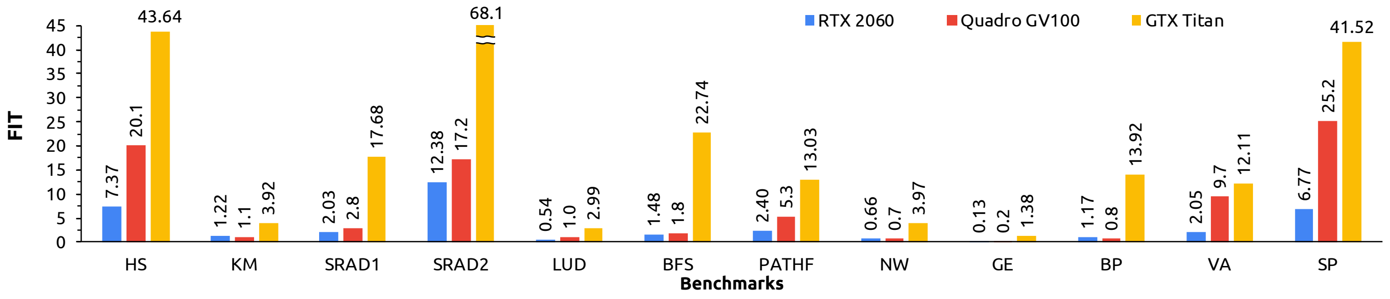


Fig. 7. Total FIT rates for the three GPU cards and twelve benchmarks used in this study.

fabricated in higher technology (e.g., 28nm for GTX Titan, and 12nm for RTX 2060 and Quadro GV100), and thus, the raw FIT rates are significantly higher than the newer ones.

VII. Conclusion

In this paper, we presented gpuFI-4, a detailed fault injection framework built on top of a state-of-the-art microarchitectural simulator of GPGPU architectures, GPGPU-sim 4.0. High-throughput, comprehensive injection campaigns for single and multiple transient faults on one or more of the critical hardware components of a GPU are supported by this fully parameterized framework. The supported hardware components are the register file, the local and shared memory, the L1 data and texture cache, and the L2 cache. Using 12 different CUDA programs, we performed a complete reliability test of the target hardware components, thus estimating the Architectural Vulnerability Factor (AVF) of a GPU. Our study reveals significant diverging behaviors on the results of fault injections on different workloads as well as on different hardware capabilities by comparing the results between GPU cards: RTX 2060, Quadro GV100, and GTX Titan. The framework can be used for differential studies on the reliability of hardware components running any CUDA workload and support early design decisions for fault protection mechanisms. gpuFI-4 is publicly available at https://github.com/caldi-uoa/gpuFI-4.

Acknowledgment

This research has been supported by the European Union Horizon 2020 programme through a H2020 Tetramax project TTX (Grant 761349) and the H2020 UniServer project (Grant 688540), as well as the FP7 Clereco project (Grant 611404).

References

[1] R. Baumann, "Soft errors in advanced computer systems," in IEEE Design & Test of Computers, vol. 22, no. 3, pp. 258-266, May-June 2005, doi: 10.1109/MDT.2005.69.

[2] C. Constantinescu, "Trends and challenges in VLSI circuit reliability," in IEEE Micro, vol. 23, no. 4, pp. 14-19, July-Aug. 2003, doi: 10.1109/MM.2003.1225959.

[3] L. Huang and Q. Xu, "AgeSim: A simulation framework for evaluating the lifetime reliability of processor-based SoCs," 2010 Design, Automation & Test in Europe Conference & Exhibition (DATE 2010), 2010, pp. 51-56, doi: 10.1109/DATE.2010.5457238.

[4] S. R. Nassif, N. Mehta and Y. Cao, "A resilience roadmap," 2010 Design, Automation & Test in Europe Conference & Exhibition (DATE 2010), 2010, pp. 1011-1016, doi: 10.1109/DATE.2010.5456958.

[5] S. S. Mukherjee, C. Weaver, J. Emer, S. K. Reinhardt and T. Austin, "A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor," Proceedings. 36th Annual IEEE/ACM International Symposium on Microarchitecture, 2003. MICRO-36., 2003, pp. 29-40, doi: 10.1109/MICRO.2003.1253181.

[6] G. Papadimitriou and D. Gizopoulos, "Demystifying the System Vulnerability Stack: Transient Fault Effects Across the Layers," 2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA), 2021, pp. 902-915, doi: 10.1109/ISCA52012.2021.00075.

[7] R. Leveugle, A. Calvez, P. Maistri and P. Vanhauwaert, "Statistical fault injection: Quantified error and confidence," 2009 Design, Automation & Test in Europe Conference & Exhibition, Nice, 2009, pp. 502-506, doi: 10.1109/DATE.2009.5090716.

[8] H. Jeon, M. Wilkening, V. Sridharan, S. Gurumurthi, G. H. Loh, "Architectural Vulnerability Modeling and Analysis of Integrated Graphics Processors", in Silicon Errors in Logic – System Effects (SELSE), 2013.

[9] N. Farazmand, R. Ubal, D. Kaeli, "Statistical fault injection-based AVF analysis of a GPU architecture", in Silicon Errors in Logic – System Effects (SELSE), 2012.

[10] J. Tan, N. Goswami, T. Li and X. Fu, "Analyzing soft-error vulnerability on GPGPU microarchitecture," 2011 IEEE International Symposium on Workload Characterization (IISWC), 2011, pp. 226-235, doi: 10.1109/IISWC.2011.6114182.

[11] T. Tsai, S. K. S. Hari, M. Sullivan, O. Villa and S. W. Keckler, "NVBitFI: Dynamic Fault Injection for GPUs," 2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), 2021, pp. 284-291, doi: 10.1109/DSN48987.2021.00041.

[12] S. K. S. Hari, T. Tsai, M. Stephenson, S. W. Keckler and J. Emer, "SASSIFI: An architecture-level fault injection tool for GPU application resilience evaluation," 2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), 2017, pp. 249-258, doi: 10.1109/ISPASS.2017.7975296.

[13] H. Yue, X. Wei, G. Li, J. Zhao, N. Jiang, and J. Tan, "G-SEPM: building an accurate and efficient soft error prediction model for GPGPUs," Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '21), 2021, doi: 10.1145/3458817.3476170.

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

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

[16] A. Bakhoda, G. L. Yuan, W. W. L. Fung, H. Wong and T. M. Aamodt, "Analyzing CUDA workloads using a detailed GPU simulator," 2009 IEEE International Symposium on Performance Analysis of Systems and Software, 2009, pp. 163-174, doi: 10.1109/ISPASS.2009.4919648.

[17] M. Hernandez, G. D. Guerrero, J. M. Cecilia, J. M. Garcia, A. Inuggi and S. N. Sotiropoulos, "Accelerating Fibre Orientation Estimation from Diffusion Weighted Magnetic Resonance Imaging Using GPUs," 2012 20th Euromicro International Conference on Parallel, Distributed and Network-based Processing, 2012, pp. 622-626, doi: 10.1109/PDP.2012.46.

[18] R. Ubal, B. Jang, P. Mistry, D. Schaa and D. Kaeli, "Multi2Sim: A simulation framework for CPU-GPU computing," 2012 21st International Conference on Parallel Architectures and Compilation Techniques (PACT), 2012, pp. 335-344, doi: 10.1145/2370816.2370865.

[19] J. Tan, Z. Li, and X. Fu, “Cost-effective soft-error protection for SRAM-based structures in GPGPUs,” Proceedings of the ACM International Conference on Computing Frontiers (CF ’13), 2013, pp. 1-10, doi: 10.1145/2482767.2482804.

[20] G. Papadimitriou and D. Gizopoulos, “Characterizing Soft Error Vulnerability of CPUs Across Compiler Optimizations and Microarchitectures”, IEEE International Symposium on Workload Characterization (IISWC), 2021, pp. 113-124, doi: 10.1109/IISWC53511.2021.00021.

[21] A. Chatzidimitriou, P. Bodmann, G. Papadimitriou, D. Gizopoulos and P. Rech, "Demystifying Soft Error Assessment Strategies on ARM CPUs: Microarchitectural Fault Injection vs. Neutron Beam Experiments," 2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), 2019, pp. 26-38, doi: 10.1109/DSN.2019.00018.

[22] P. Bodmann, G. Papadimitriou, R. L. Rech Junior, D. Gizopoulos and P. Rech, "Soft Error Effects on Arm Microprocessors: Early Estimations vs. Chip Measurements," in IEEE Transactions on Computers, doi: 10.1109/TC.2021.3128501.

[23] S. K. S. Hari, T. Tsai, M. Stephenson, S. W. Keckler and J. Emer, "SASSIFI: An architecture-level fault injection tool for GPU application resilience evaluation," 2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), 2017, pp. 249-258, doi: 10.1109/ISPASS.2017.7975296.

[24] G. Li, K. Pattabiraman, C. Cher and P. Bose, "Understanding Error Propagation in GPGPU Applications," SC '16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, 2016, pp. 240-251, doi: 10.1109/SC.2016.20.

[25] S. Che et al., "Rodinia: A benchmark suite for heterogeneous computing," 2009 IEEE International Symposium on Workload Characterization (IISWC), 2009, pp. 44-54, doi: 10.1109/IISWC.2009.5306797.

[26] NVIDIA CUDA SDK 5.5, [Online]. Available: https://developer.nvidia.com/cuda-toolkit-55-archive

[27] M. Khairy, Z. Shen, T. M. Aamodt and T. G. Rogers, "Accel-Sim: An Extensible Simulation Framework for Validated GPU Modeling," 2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA), 2020, pp. 473-486, doi: 10.1109/ISCA45697.2020.00047.

[28] H. Cho, S. Mirkhani, C. Cher, J. A. Abraham and S. Mitra, "Quantitative evaluation of soft error injection techniques for robust system design," 2013 50th ACM/EDAC/IEEE Design Automation Conference (DAC), 2013, pp. 1-10, doi: 10.1145/2463209.2488859

[29] A. Chatzidimitriou, G. Papadimitriou, C. Gavanas, G. Katsoridas and D. Gizopoulos, "Multi-Bit Upsets Vulnerability Analysis of Modern Microprocessors," 2019 IEEE International Symposium on Workload Characterization (IISWC), 2019, pp. 119-130, doi: 10.1109/IISWC47752.2019.9042036.

[30] G. Li, S. K. Sastry Hari, M. Sullivan, T. Tsai, K. Pattabiraman, J. Emer, and S. W. Keckler, “Understanding error propagation in deep learning neural network (DNN) accelerators and applications,” Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC ’17), 2017, pp. 1-12 doi: 10.1145/3126908.3126964.

[31] A. Neale and M. Sachdev, "Neutron Radiation Induced Soft Error Rates for an Adjacent-ECC Protected SRAM in 28 nm CMOS," in IEEE Transactions on Nuclear Science, vol. 63, no. 3, pp. 1912-1917, June 2016, doi: 10.1109/TNS.2016.2547963.