NATIONAL AND KAPODISTRIAN UNIVERSITY OF ATHENS

SCHOOL OF SCIENCE

DEPARTMENT OF INFORMATICS AND TELECOMMUNICATIONS

GRADUATE PROGRAM

"COMPUTER SYSTEMS: SOFTWARE AND HARDWARE"

MASTER'S THESIS

GPGPU injector 4.0: A Framework for Architectural Vulnerability Factor (AVF) Assessments Across Nvidia GPUs Generations using GPGPU-Sim 4.0 simulator

Dimitris A. Sartzetakis

Advisor: Dimitris Gizopoulos, Professor

ATHENS

SEPTEMBER 2021

ΕΘΝΙΚΟ ΚΑΙ ΚΑΠΟΔΙΣΤΡΙΑΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ

ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

ΠΡΟΓΡΑΜΜΑ ΜΕΤΑΠΤΥΧΙΑΚΩΝ ΣΠΟΥΔΩΝ

"ΥΠΟΛΟΓΙΣΤΙΚΑ ΣΥΣΤΗΜΑΤΑ: ΛΟΓΙΣΜΙΚΟ ΚΑΙ ΥΛΙΚΟ"

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

GPGPU injector 4.0: Αυτοματοποιημένο Περιβάλλον για τον υπολογισμό του Architectural Vulnerability Factor (AVF) σε διαφορετικές γενιές καρτών επεξεργασίας γραφικών της Nvidia με χρήση του προσομοιωτή GPGPU-Sim 4.0

Δημήτρης Α. Σαρτζετάκης

Επιβλέπων: Δημήτρης Γκιζόπουλος, Καθηγητής

ΑΘΗΝΑ

ΣΕΠΤΕΜΒΡΙΟΣ 2021

MASTER'S THESIS

GPGPU injector 4.0: A Framework for Architectural Vulnerability Factor (AVF) Assessments Across Nvidia GPUs Generations using GPGPU-Sim 4.0 simulator

Dimitris A. Sartzetakis

R.N.: M1561

Advisor: Dimitris Gizopoulos, Professor

September 2021

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

GPGPU injector 4.0: Αυτοματοποιημένο Περιβάλλον για τον υπολογισμό του Architectural Vulnerability Factor (AVF) σε διαφορετικές γενιές καρτών επεξεργασίας γραφικών της Nvidia με χρήση του προσομοιωτή GPGPU-Sim 4.0

Δημήτρης Α. Σαρτζετάκης

Α.Μ.: M1561

Επιβλέπων: Δημήτρης Γκιζόπουλος, Καθηγητής

Σεπτέμβριος 2021

ABSTRACT

A (Graphics Processing Unit) GPU is a programmable processor on which thousands of processing cores run simultaneously in massive parallelism, where each core is focused on making efficient calculations, facilitating real-time processing and analysis of enormous datasets. Due to the development of general purpose parallel programming environments and languages, all modern GPUs are general purpose GPUs (GPGPUs) as they can be programmed for non-graphics applications and they can direct their processing power towards massively parallel problems. Therefore, as in all general-purpose computing platforms, accurate reliability on GPU hardware structures is a very important factor that architects need to estimate early in the design cycle to weigh the benefits of error protection techniques against their costs.

In this thesis, we introduce GPGPU injector 4.0 which is a fault injection framework for Architectural Vulnerability Factor (AVF) assessment of hardware structures and entire GPU chips that runs over the state-of-the-art performance simulator for Nvidia GPUs architectures: GPGPU-sim. We use GPGPU injector 4.0 for fault injection of transient faults (soft errors) on CUDA enabled GPU architecture. The target hardware structures include the register file, the shared memory, the L1 data/texture cache and the L2 cache which altogether account for several tens of MBs on on-chip GPU storage. More specifically, we compute the AVF of two widely used recent graphic cards which are the RTX 2060 and Quadro GV100 by experimenting with ten different CUDA benchmarks that are simulated on the actual instruction set (SASS).

SUBJECT AREA: computer architecture, reliability assessment, fault injection, graphic processing units (GPUs), accelerators, microarchitecture simulator

KEYWORDS: transient faults, AVF estimation, Failures In Time (FIT), register file, shared memory, cache memories, GPGPU-Sim

ΠΕΡΙΛΗΨΗ

Η κάρτα γραφικών (GPU) είναι ένας προγραμματιζόμενος επεξεργαστής στον οποίο χιλιάδες πυρήνες επεξεργασίας λειτουργούν ταυτόχρονα σε μαζικό παραλληλισμό, όπου κάθε πυρήνας επικεντρώνεται στην πραγματοποίηση υπολογισμών, διευκολύνοντας την επεξεργασία και την ανάλυση σε πραγματικό χρόνο τεράστιων όγκων δεδομένων. Όλες οι σύγχρονες κάρτες γραφικών είναι επίσης γνωστές και ως κάρτες γραφικών γενικής χρήσης (GPGPU) καθώς μπορούν να προγραμματιστούν ώστε να κατευθύνουν αυτήν την επεξεργαστική ισχύ και προς την αντιμετώπιση επιστημονικών υπολογιστικών αναγκών. Επομένως, η αξιοπιστία του υλικού μιας GPU είναι ένας πολύ σημαντικός παράγοντας που πρέπει να εκτιμήσουν οι αρχιτέκτονες νωρίς στον κύκλο του σχεδιασμού ώστε να σταθμίσουν τα οφέλη των τεχνικών προστασίας από σφάλματα έναντι του κόστους.

Σε αυτή τη διπλωματική, παρουσιάζουμε το GPGPU injector 4.0, το οποίο είναι ένα εργαλείο (framework) “εισαγωγής” σφαλμάτων (fault injection) για την εκτίμηση της αξιοπιστίας μιας κάρτας γραφικών σε μικροαρχιτεκτονικό επίπεδο (AVF) και τρέχει πάνω σε ένα γνωστό εργαλείο που προσομοιώνει κάρτες γραφικών: GPGPU-sim. Χρησιμοποιούμε το εργαλείο GPGPU injector 4.0 για την εισαγωγή σφαλμάτων υλικού (transient faults) σε κάρτες γραφικών με δυνατότητα CUDA, πάνω σε δομές υλικού όπως το αρχείο καταχωρητή, την κοινή μνήμη, την L1 προσωρινή μνήμη απλών δεδομένων αλλά και δεδομένων texture και την προσωρινή μνήμη L2. Πιο συγκεκριμένα, υπολογίζουμε το AVF δύο ευρέως χρησιμοποιούμενων πρόσφατων καρτών γραφικών που είναι η RTX 2060 και η Quadro GV100 χρησιμοποιώντας δέκα διαφορετικά CUDA προγράμματα τα οποία εκτελούνται πάνω στον προσομοιωτή σε γλώσσα μηχανής του υλικού (SASS).

ΘΕΜΑΤΙΚΗ ΠΕΡΙΟΧΗ: αρχιτεκτονική υπολογιστών, εκτίμηση αξιοπιστίας, εισαγωγή σφαλμάτων, κάρτα γραφικών, παροδικά σφάλματα, προσομοιωτής μικροαρχιτεκτονικής

ΛΕΞΕΙΣ ΚΛΕΙΔΙΑ: σφάλματα υλικού, υπολογισμός AVF, υπολογισμός FIT rate, αρχείο καταχωρητών, κοινόχρηστη μνήμη, κρυφές μνήμες, GPGPU-Sim

I would like to express my deepest gratitude to my advisor Professor, Dimitris Gizopoulos who has been an ideal teacher all of these years during my undergraduate and graduate studies. I am also thankful to Dimitris for his invaluable assistance and insights leading to the writing of this thesis. I would also like to thank Postdoctoral Researcher of Prof.Gizopoulos' Computer Architecture Lab, George Papadimitriou for supporting me in many ways and was always willing to help me.

TABLE OF CONTENTS

FIGURES' INDEX

TABLES' INDEX

1. INTRODUCTION

Almost everyone nowadays benefits from exploiting the computational power of modern GPU over a CPU for their applications when they can exploit data level parallelism (DLP). Architecturally, the CPU is made up of a few cores and a lot of cache memory, thus it can only manage a few, but control-complicated, software threads at a time. A GPU, on the other hand, is composed of hundreds of cores that can handle thousands of simpler threads simultaneously. The ability of a GPU with 100+ cores to process thousands of threads can accelerate certain types of data parallel software by 100x over a CPU alone. What's more, the GPU achieves this acceleration while being more power- and cost-efficient than a CPU for the particular workload types.

The new advancements in technology establish a new generation of electronic devices with a wealth of transistors to improve their performance and other capabilities but on the other hand, such techniques significantly affect their reliability (i.e. their vulnerability to hardware faults that can be due to multiple sources - external or internal). Like all systems, the reliable operation of graphic cards can be affected by transient faults (soft errors), intermittent faults, and permanent (hard) faults [7] [8] [9]. Such hardware faults can be caused by multiple factors like radiation, process differentials and variability, in-progress wear-out, etc. Several metrics have been proposed for the assessment of reliability in processing units such as the Architecture Vulnerability Factor (AVF), which is the probability that a transient fault in a hardware structure will result in an observable error in an application's output [11]. Similarly, vulnerability factors for intermittent faults (IVF) [14], permanent faults (H-AVF) [13], program vulnerability (PVF) [12], and hardware vulnerability (HVF) [26] have been defined.

Vulnerability assessment of GPU hardware components is very important during the early design stages and can help to weigh the benefits of different error protection techniques against design cost and time. As a result, much effort must be devoted to effectively measuring 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 workloads. As a result, reliability assessments with the use of microarchitecture simulators are preferred. Different approaches, like Architectural Correct Execution (ACE) analysis, and probabilistic approaches can be employed on top of microarchitecture-level simulators for measuring the vulnerability of microprocessor components to soft errors. However, unlike fault injection, probabilistic and ACE approaches overestimate the vulnerability of microprocessor structures [16].

In this thesis, we propose and develop a comprehensive fault injection framework, that is built on top of the state-of-the-art cycle-level GPU simulator, for measuring the AVF of individual hardware structures of GPUs and entire GPU chips. Our framework is capable of injecting transient faults on most of the important hardware components of an Nvidia GPU which are: the register file, the shared memory, the local memory, the L1 data/texture cache, and the L2 cache.

2. BACKGROUND

The Graphic Processing Unit (GPU) was designed for real-time graphics. A modern GPU, on the other hand, is not only a strong graphics processor, but also a general-purpose computer processor that focuses on parallel processing and high data bandwidth. Since the clear slowing of CPU speed over the last two decades, GPUs have gained popularity for general-purpose computing when data parallelism is ample, and it has become a popular topic since 2011. GPU could become a very promising contender in high performance computing with rapid increases in both calculation power and programmability.

2.1 GPU History [1]

Graphic processors have a long history, almost as long as the PC. IBM invented the first graphic processor, the CGA (Color Graphics Adapter), in 1981. Graphic processors became more powerful after one and a half decades of research and were able to handle 3D acceleration on PC desktops. The term Graphic Processing Unit was introduced in 1999 when Nvidia released GeForce256, the "world's first GPU." Since then, research in physics, medical imaging, and other sectors has begun to use GPUs to accelerate their applications. That is the beginning of GPGPU computing. Nvidia changed its GPU to make it more easily programmable after noticing the large market for GPU computing. Nvidia also introduced a new GPU programming model and language called CUDA in 2007. Following that, developers may easily program the GPU using CUDA, which is simply a C extension, and take more advantage of the underlying compute power that modern GPUs provide.

2.2 Nvidia GPU architecture

2.2.1 Overview

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. Figure 1 shows a typical NVIDIA GPU architecture based on the NVIDIA’s Fermi architecture. Streaming multiprocessors (SM), each containing several Stream Processors (SPs) (see Figure 1(a)). The GPU has a global scheduler (Giga Thread) for distributing the work to the SMs and a host interface. Different memory spaces are also available within a GPU, having different latencies, storage capacity and access methods. These memory spaces, ordered from low to high latency are: the register file (32768 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) [2]. The number of the SMs, SPs and the sizes of the memory spaces can be different from one generation to another and usually get larger as time goes by. A quick reference of all the Nvidia microarchitectures and GPUs over the years (RTX 2060 & Quadro GV100 are included), can be seen in Table 1.

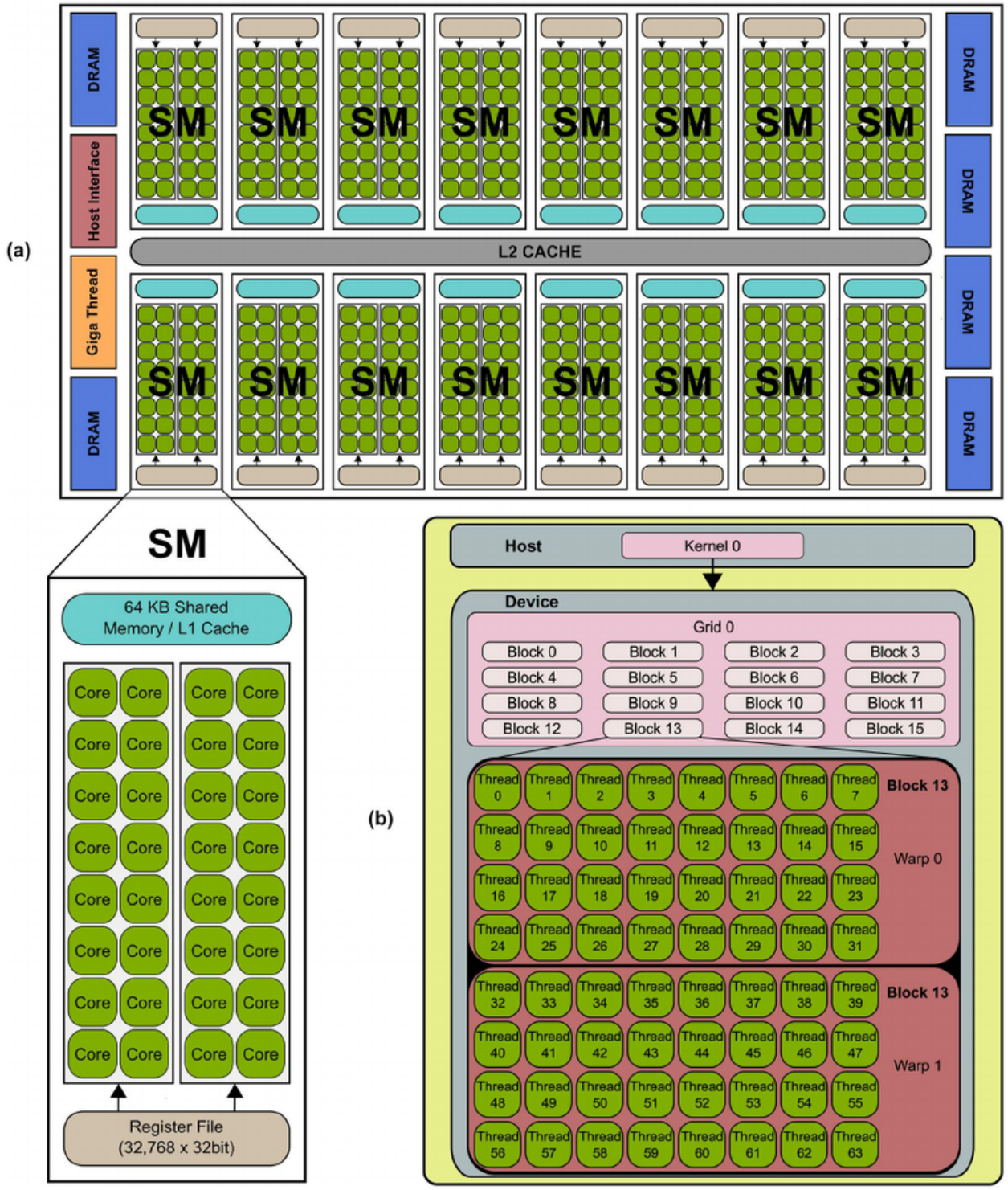

Figure 1: Typical NVIDIA GPU architecture based on the NVIDIA's Fermi architecture [2]

Table 1: Nvidia microarchitectures and GPUs [29]

Microarchitecture	Year	Compute capability (SASS version)	GPUs
Tesla	2006	1.0, 1.1, 1.2, 1.3	G80, G92, G94, G96, G98, G84, G86, GT218, GT216, GT215, GT200, GT200b
Fermi	2010	2.0, 2.1	GF100 (e.g. GTX 480), GF110, GF104, GF106 GF108, GF114, GF116, GF117, GF119
Kepler	2012	3.0, 3.2, 3.5, 3.7	GK104, GK106, GK107, GK20A, GK110 (e.g. GTX Titan), GK208, GK210
Maxwell	2014	5.0, 5.2, 5.3	GM107, GM108, GM200, GM204, GM206, GM20B
Pascal	2016	6.0, 6.1, 6.2	GP100, GP102, GP104, GP106, GP107, GP108, GP10B
Volta	2017	7.0, 7.2	GV100 (e.g. **Quadro GV100**), GV10B
Turing	2018	7.5	TU102, TU104, TU106 (e.g. **RTX 2060**), TU116, TU117
Ampere	2020	8.0, 8.6, 8.7	GA100. GA102, GA104, GA106, GA107

2.2.2 Multiprocessor Structure

A Graphic Processing Unit (GPU) is made up of numerous streaming multiprocessors, each with multiple streaming processors. Figure 2 shows the structure of a multiprocessor. The streaming processors with one multiprocessor share the constant cache, texture cache and instruction unit. Each streaming processor has its own register file for storing data that is frequently used. The register file 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 likewise designed for communication across streaming processors and is implemented on-chip with very low access latency.

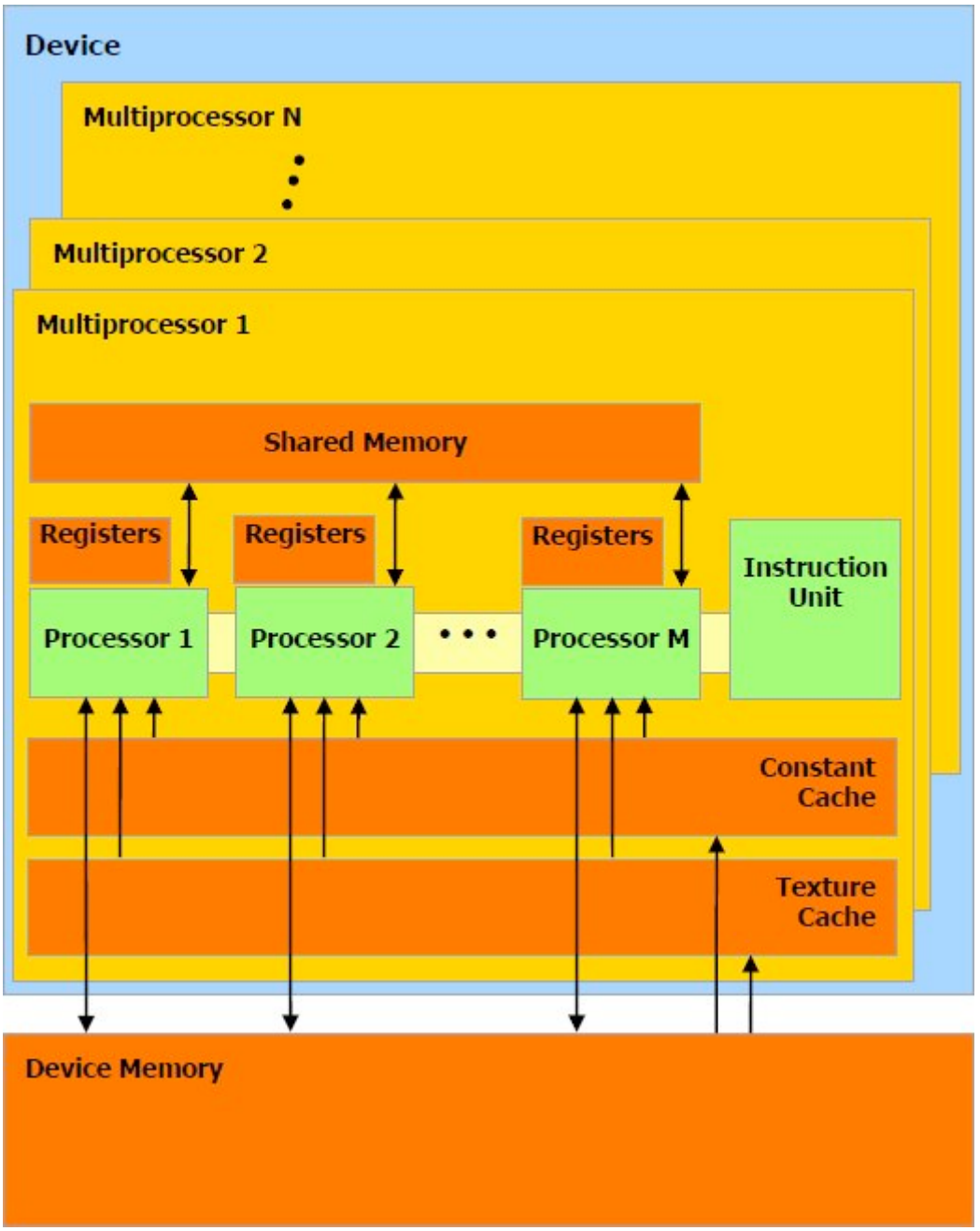

Figure 2: Multiprocessor Structure [3]

2.2.3 Memory Hierarchy

A GPU has local memory, global memory, shared memory, data cache, constant cache, texture cache, and registers, as shown in Figure 2. The sizes of these memory components vary between different architectures from a few KB (e.g. caches) up to GB (e.g. global memory). The total size of the supported memories that we used in our experiments for RTX 2060 and Quadro GV100 can be seen at Table 2 (tag bits for caches are included which we will explain in 9.3.2). Each CUDA thread may access data from them during their execution as illustrated by Figure 3. Each thread has private local memory. The local memory space resides in device memory, so local memory accesses

have the same high latency and low bandwidth as 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 it will not be changed during the execution of the program. Shared memory is used to allow streaming processors to communicate with one another. All threads have access to the same global memory and is used for communication between host CPU and GPU since GPU can not access the CPU main memory. Data that will be handled by the GPU must first be copied to global memory and results from the GPU must be copied to the CPU memory back with the appropriate API (i.e. cudaMemcpy).

Table 2: RTX 2060 and Quadro GV100 memory components total sizes

	RTX 2060 (#SMs: 30)	Quadro GV100 (#SMs: 80)
Register File	30 x 256KB = 7.5 MB	80 x 256KB = 20 MB
Shared Memory	30 x 64KB = 1.875 MB	80 x 96KB = 7.5 MB
L1 data cache	30 x 67.56KB = 1.98 MB	80 x 33.78KB = 2.64 MB
L1 texture cache	30 x 135.13KB = 3.96 MB	80 x 135.13KB = 10.56 MB
L1 instruction cache	30 x 135.13KB = 3.96 MB	80 x 135.13KB = 10.56 MB
L1 constant cache	30 x 71.13KB = 2.08 MB	80 x 71.13KB = 5.56 MB
L2 cache	3.17 MB	6.33 MB

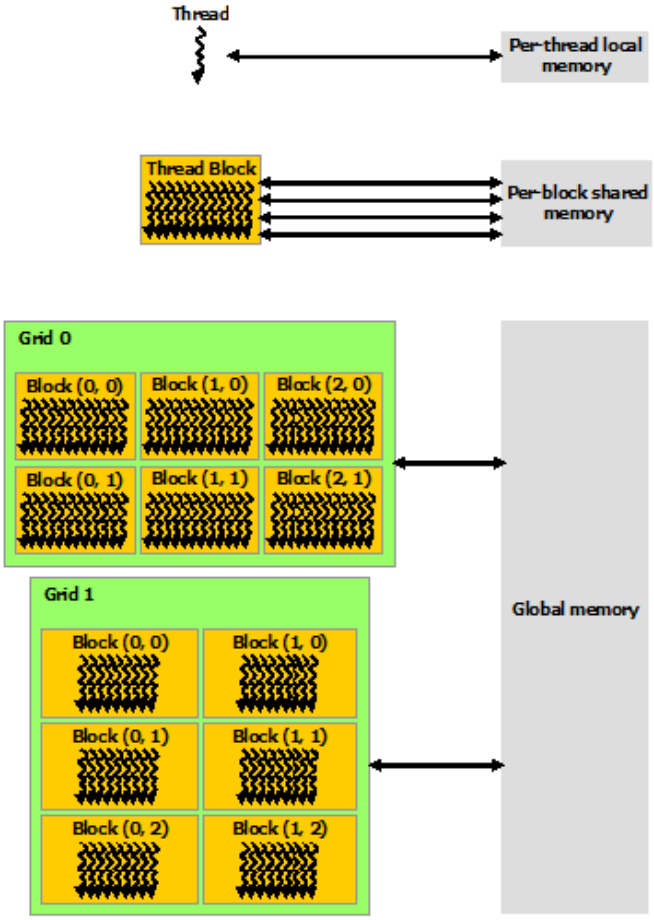

Figure 3: Memory Hierarchy [4]

2.2.4 CUDA programming model

The CUDA programming model utilizes this hardware architecture and is based on a hierarchy of abstraction layers, see Figure 1 (b). The *thread* is the basic software execution unit that is mapped to a single SP.

A thread-block, or simply block, or a Common Thread Array (CTA), is a group of threads that are all assigned to the same SM and hence share all of the multiprocessor's resources, such as the register file and the shared memory. The shared memory allows threads within a block to communicate. Finally, a grid is composed of several blocks which are equally distributed and scheduled across all SMs in a nondeterministic manner.

Threads of the same block are divided into groups of 32 threads called warps [5]. The warp is the scheduled unit, so the threads of the same block are executed in a given multiprocessor warp-by-warp. Because threads (and not data) are mapped to the multiprocessor and executed in a Single-Instruction, Multiple-Data (SIMD)-like fashion, the style of execution is called Single-Instruction, Multiple-Thread (SIMT). SIMT is very similar to SIMD. In SIMD, multiple data can be processed by a single instruction. In SIMT, multiple threads are processed by a single instruction in lock-step. Each thread executes the same instruction, but possibly on different data. The programmer arranges parallelism by declaring the number of blocks and the number of threads per block to use in a specific kernel. To avoid wasting SP resources, the number of threads per block should be a multiple of 32 (i.e. a warp).

Kernels can be written using the CUDA instruction set architecture, called Parallel Thread Execution assembly (PTX). It is however usually more effective to use a high-level programming language such as C++. In both cases, kernels must be compiled into binary code, named SASS, which is the low-level assembly language that can be executed natively on NVIDIA GPU hardware. The different SASS versions per generation can be seen in Table 1. In general, the newer SASS versions offer additional instructions in order to take advantage of the hardware-level features that are introduced per generation.

2.3 GPGPU-Sim overview [6]

GPGPU-Sim is a cycle-level simulator modeling contemporary Nvidia graphics processing units (GPUs) running GPU computing workloads written in CUDA or OpenCL. The simulator is capable of running 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 and not the actual binary code that runs on the hardware there was an accuracy limit. For that reason, the developers of GPGPU-Sim decided to extend PTX with the required features in order to provide a one-to-one mapping to SASS. PTX along with the extentions is called PTXPlus.

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 an Streaming Multiprocessor (SM) or what AMD calls a Compute Unit (CU). The organization of an SIMT core is illustrated in Figure 4. 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 as shown in Figure 5. An SIMT core, as shown in Figure 6, models a highly multithreaded pipelined SIMD processor roughly equivalent to what NVIDIA calls an Streaming Multiprocessor (SM) or what AMD calls a Compute Unit (CU). A Stream Processor (SP) or a CUDA Core would correspond to a lane within an ALU pipeline in the SIMT core.

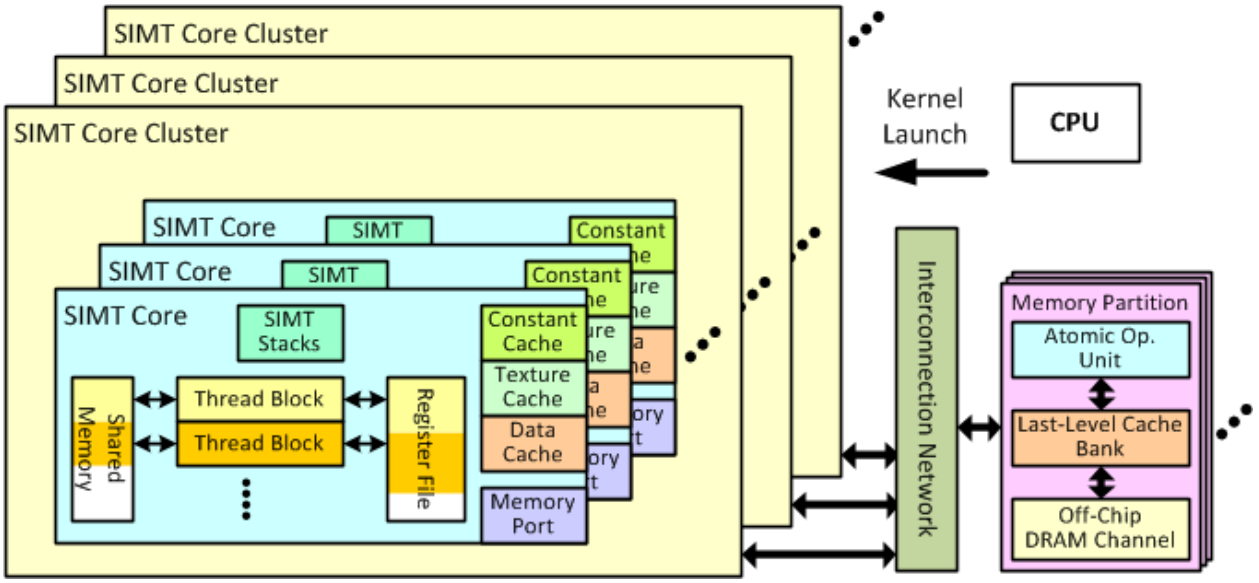

Figure 4: Overall GPU Architecture Modeled by GPGPU-Sim

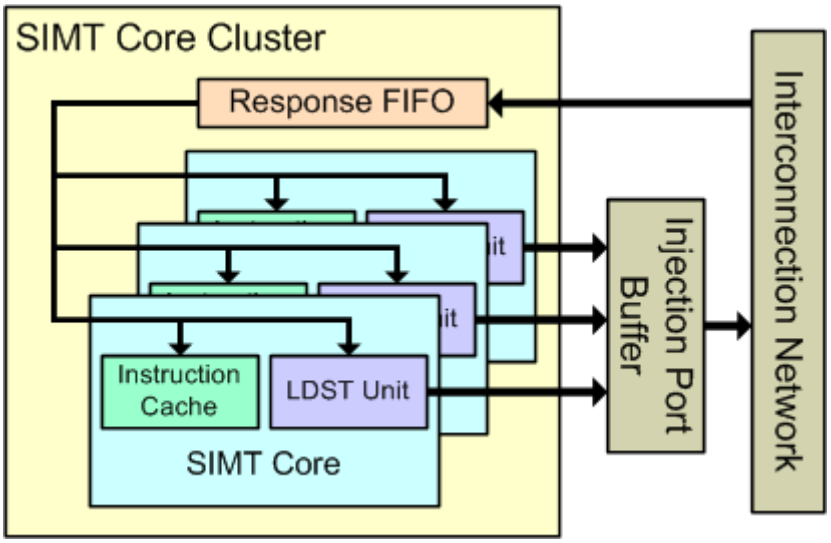

Figure 5: SIMT Core Clusters

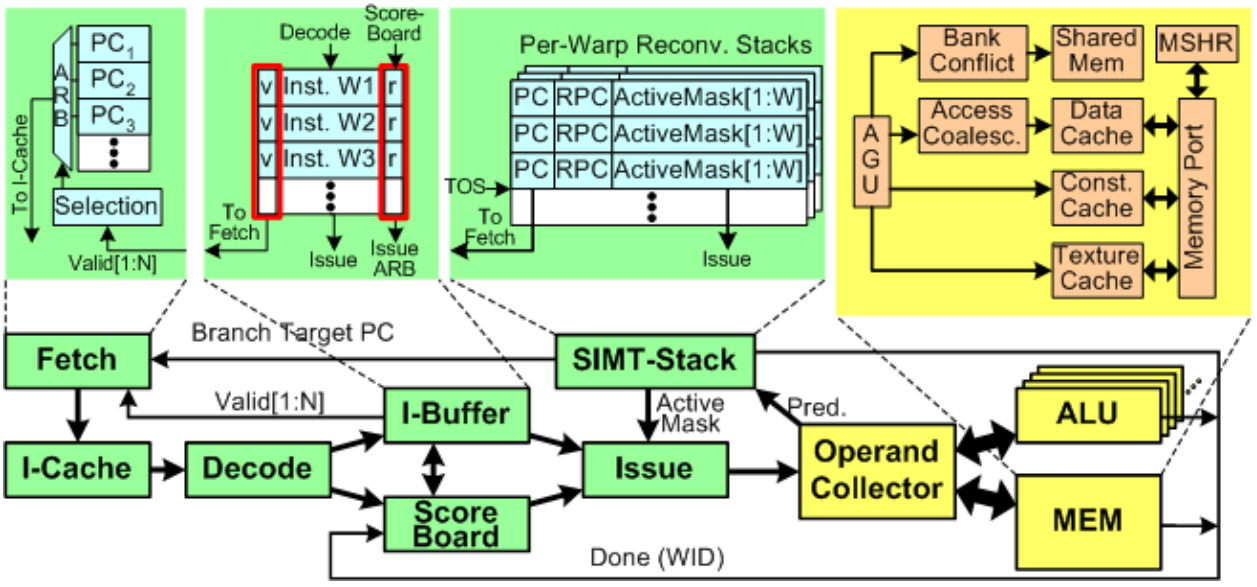

Figure 6: Detailed Microarchitecture Model of SIMT Core in GPGPU-Sim

GPGPU-Sim 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. The following Table 3 shows which on chip memories service which type of memory access

Table 3: CUDA supported memory spaces in GPGPU-Sim

Core Memory	PTX Accesses
Shared memory (R/W)	shared memory accesses only
Constant cache (Read Only)	Constant memory 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

Regarding the memory system in GPGPU-Sim, it is modelled by a set of memory partitions. As shown in Figure 7 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 [6] for a comprehensive overview of the GPGPU-Sim microarchitecture.

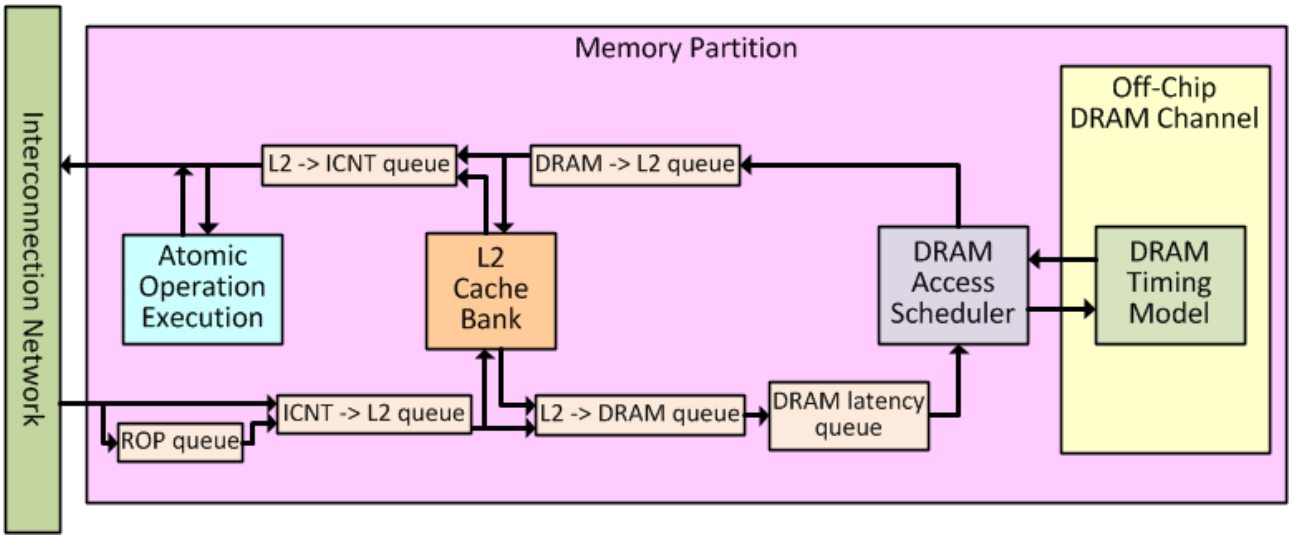

Figure 7: Memory partition in GPGPU-Sim

3. HARDWARE FAULTS AND SOFT ERRORS

Hardware faults or soft errors can compromise the reliability of present and future computing systems. An erroneous bit in a microprocessor, for example, may have no effect, may modify the expected output of a running program, or may trigger an error capable of terminating the computer system's function. There are three types of faults that a computer system can be affected by, which are transient faults (soft errors), intermittent faults, and permanent (hard) faults [7] [8] [9].

The soft error type of fault, also known as a single-event upset (SEU), is the one that will be studied in our framework and is mainly caused by ionizing radiation from cosmic rays and alpha particles from the chip packages. Transient faults have also been caused by thermal neutrons and even by random noise or signal integrity problems, such as inductive or capacitive crosstalk. However, in general, these sources represent a small contribution to the overall soft error rate when compared to radiation effects. Soft errors can cause a storage element’s bit value to be flipped (inverted) which typically are resolved passively when the affected bits are overwritten or by a system reboot. If a flipped bit is, however, read before it is overwritten it can potentially affect the correct execution of a program.

Intermittent hardware faults are bursts of errors that occur at the same location (micro-architectural component) and last from a few cycles to a few seconds, depending on their causes [10]. Intermittent faults can be caused by oxide relegation, process differentials, industrialization residuals, and in-progress wear-out. Unlike a soft error, they can cause a storage element’s bit value to be stuck at logical “1” or “0” for a relatively small number of clock cycles.

Permanent faults have the same behavior as intermittent faults but they persist infinitely (or at least until repair). These faults can be caused by many different physical processes from normal operation, for example, thermal stress, electromigration, hot carrier injection, gate oxide wear-out, and negative bias temperature instability. While permanent faults may cause undesirable behavior, they are quite easy to identify and eliminate because their incorrect output is consistent and they are isolated to a particular piece of hardware.

4. VULNERABILITY FACTORS

Vulnerability factors (also known as derating factors or soft error sensitivity factors) indicate the probability that an internal fault (hardware fault or soft error) during a system's operation will cause a visible external error. Such factors are very important in the sense that the designers need to estimate their effects early in the design cycle to weigh the benefits of error protection techniques against their costs.

There are several definitions of vulnerability factors that have been proposed over the years. The Architecture Vulnerability Factor (AVF), expresses the probability that a visible system error will occur given a transient fault in a storage cell [11]. The Program Vulnerability Factor (PVF) measures software vulnerability and is responsible for characterizing the inherent soft error masking rate in a program [12]. The Hard-Fault Architectural Vulnerability Factor (H-AVF) is a metric for permanent faults and allows designers to more effectively compare alternate hard-fault tolerance schemes [13]. The Intermittent Vulnerability Factor (IVF) is a metric to compute the probability that an intermittent fault in a structure will manifest itself in an observable program output [14]. Lastly, the Hardware Vulnerability Factor (HVF) quantifies the hardware portion of AVF, independent of program-level masking effects [26].

In our analysis, we will determine the AVF by using our microarchitecture-level fault injection tool which is built on top of GPGPU-Sim. There are several other methods for computing the AVF of a hardware structure that utilize a performance simulator, such as ACE-based (Architectural Correct Execution) analysis and probabilistic methods. Other approaches also use performance measurements for online AVF estimation, while others try to separate the masking effects that hardware and software can have on hardware faults. Nevertheless, in comparison to the fault injection method, all of these approaches have one major disadvantage, they severely overestimate the vulnerability of microprocessor structures [16] and for this reason they can lead to pessimistic, thus costly, system design.

Regarding the overall vulnerability of a chip, it is measured in FIT (Failure in TIme) and MTBF (Mean Time Between Failures) which are the two most commonly used units for error rates. MTBF measures the average time that equipment is operating between breakdowns or stoppages. Measured in hours, MTBF helps businesses understand the availability of their equipment (and if they have a problem with reliability). FIT reports the number of expected failures per one billion hours of operation for a device. Although FIT is another way of reporting MTBF, the majority of the designers work with it because it is additive which is not the case for MTBF which, however, is more intuitive. The overall FIT rate of a chip is calculated by summing the FIT rates of all chip's hardware structures, where the FIT rate for a structure is the product of structure's vulnerability factor and the raw circuit FIT rate.

5. RELATED WORK

Previous works on the reliability of GPU architectures have been studied recently, both on hardware and software.

GUFI [15] is a fault injection framework using a simulator which is, in a way, similar to ours. More specifically, It is a microarchitecture-level fault injection framework which is built on top of GPGPU-Sim [6]. Unlike GUFI which uses GPGPU-Sim version 3.0, our framework is built on top of the latest simulator version which is 4.0. Another main difference is that our framework studies transient faults on more crucial hardware components which are the L1 and L2 caches and thus, our experiments can take place on a larger area of a GPU hardware (18.5MB and 47MB in total for RTX 2060 and Quadro GV100 respectively). Lastly, by using the latest version of the simulator we are capable of testing newer GPUs as well.

Other reliability evaluation approaches that employ microarchitectural simulators like GPGPU-sim and Multi2sim [19] are available but with the difference that they determine the Architectural Vulnerable Factor (AVF) of hardware structures using Architectural Correct Execution (ACE) analysis [20] [21] [22] and as a result they overestimate the vulnerability of microprocessor structures [16]. Source-level fault injections in real NVIDIA GPUs have also been studied such as NVBitFI [23], SASSIFI [24], and GPU-Qin [25] but they are useful for estimating the Program Vulnerable Factor (PVF) and not a complete AVF measurement like in our analysis that allows injection of faults in the target hardware structures. As it has been recently shown for CPUs [27], we strongly believe that high level vulnerability measurements (PVF) can mislead design protection decisions because they report incorrect relative vulnerabilities among benchmarks compared to the correct, ground truth AVF measurements.

6. METHODOLOGY

To measure the Architectural Vulnerable Factor (AVF_{GPU}) of an NVIDIA GPU from a CUDA application, we first measure the AVF for each application's kernel (AVF_{kernel}) independently, and then we compute the weighted arithmetic mean on them with the kernel's execution cycles as the weights. In the measurement of AVF we take into consideration the sizes of every hardware structure as we explain below.

The AVF_{kernel} measurement is exploiting the features of the developed framework 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 rate ($FR_{structure}$) and its corresponding hardware structure size, by the size of all the previous hardware structures combined. The aforementioned structure failure rate is calculated by dividing the number of fault injection experiments on a hardware component that results in application failure by the total number of injected faults.

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

$$AVF_{kernel} = \frac{\sum_{i \in \{structure\}} FRi\ x\ SIZEi}{\#Total\ Size}$$, where all sizes are in bits

$$AVF_{GPU} = \frac{\sum_{i \in \{kernel\}} AVFi\ x\ CYCLESi}{\#Total\ cycles\ of\ the\ application}$$

One of the main drawbacks of modeling with GPGPU-sim, as the GUFI also mentions, 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 structure **df_reg** and **df_smem**. To estimate the final AVF of the register file and the shared memory, we have to multiply each factor with the relative percentage of failures [15].

We slightly modified these derating factors of GUFI in order to 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. With that 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. 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. 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:** the size of shared memory in an SM in bits.

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

7. GPGPU injector 4.0

GPGPU injector 4.0 is a complete framework for reliability evaluation of NVIDIA GPU architectures that runs over a well-known simulator of GPUs architectures: GPGPU-Sim 4.0. Our framework is capable of running transient fault injection campaigns on PTX or SASS mode using single or multiple bit flips during the execution of an application as explained below for each hardware component:

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.

Local Memory

- 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.

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.

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 bit flips per kernel injection campaigns.

8. USING GPGPU injector 4.0

The GPGPU injector 4.0 framework consists of two parts: a back-end and a front-end. The back-end part is the actual implementation of the fault injection. It is 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 file to the simulator. The front-end part is a bash script (see appendix 13.1) which is responsible for initializing the newly created parameters, executing the campaigns, and collecting the results. The main focus of this chapter is to explain the frontend part and what steps should be followed until the execution. The backend implementation will be discussed in the next chapter.

Frontend steps:

8.1 CUDA application preparation

This framework relies its evaluation process (3rd step) on the evaluation of the application itself. As a result, the applications should be slightly modified to compare the results of the GPU part execution with either a predefined result file (taken from 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. Since there is a higher probability for something to go wrong during the CPU execution, the predefined result file is preferred in our implementation.

8.2 Profiling and campaign preparation

As we mentioned earlier, the campaign.sh script requires several parameters to be configured before the injection campaigns are performed. We can differentiate these parameters into four abstract groups. The first group contains one-time parameters. The second one contains parameters that need to be initialized once per GPGPU card and are necessary to define values that describe some of the hardware structures. 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. Parameters that belong to the fourth and last group are responsible for executing different injection campaigns. Let’s call these groups: one time, per GPGPU card, per kernel/application and per injection campaign parameters respectively. For the last group, per injection campaign, the values of the parameters corresponding to a component that we are not injecting faults will be ignored.

One-time parameters

- **CONFIG_FILE**: This is the GPGPU-Sim configuration file where our new input parameters are defined. The filename cannot be changed as it is the input of GPGPU-Sim [6].
- **RUNS**: This is how many executions our campaign is going to run. For example, if we set our campaign.sh script to inject a bit flip on a register and we have RUNS=3000 then 3000 application executions will be performed by injecting a bit flip on a random register on each run.
- **BATCH**: To make our framework faster we provided it with some kind of parallelism. Specifically, #BATCH number of executions run in parallel until all are finished before starting the next batch. A better approach would be to start the next execution when one of the executions from the batch is done and not to wait for all

of the executions of a batch but this was difficult to implement within the script. The default value of this parameter is the number of processors (or virtual cores if hyper-threading is supported) minus one core so the system will not hang.
- **TMP_FILE**: This is a file that contains GPGPU-Sim execution default output [6] along with the CUDA application output.
- **TMP_DIR**: This is the directory where the CONFIG_FILE and GPGPU-Sim output (TMP_FILE) files are saved for each execution. In fact, roundup(RUNS/BATCH) number of TMP_DIR directories will be created appended with an identifier. For example, if we have RUNS=10, BATCH=5, TMP_DIR=logs and TMP_FILE=tmp then logs1 and logs2 directories will be created where each one contains the files {gpgpusim.config1,gpgpusim.config2,...,gpgpusim.config5} and {tmp1, tmp2,...,tmp5}.
- **CACHE_LOGS_DIR**: This is a directory where logs are saved for all the executions when we run injection campaigns on caches. The information that is saved is the cache line that the fault was injected and the exact bit that was flipped.

Per GPGPU card parameters
- **L1D_SIZE_BITS**: This is the total size in bits of the L1 data cache per SM. Tag bits should be included. The tag information will be covered in chapter 9.
- **L1T_SIZE_BITS**: Same as the L1D_SIZE_BITS but for the texture cache.
- **L2_SIZE_BITS**: This is the total size in bits of the L2 cache. Tag bits should be included here as well.

Per kernel/application parameters
- **CUDA_UUT**: The CUDA application command that a user wants to examine.
- **CYCLES**: The total cycles that the application took on a fault-free execution, meaning without any fault injections. GPGPU-Sim is deterministic and thus each fault free execution of the same program with the same inputs takes the same number of clock cycles.
- **profile=1**: This will run the application once without any fault injections and output the cycles for each kernel's invocation at TMP_FILE during the last cycle of the application, which we can use as input to initialize the CYCLES_FILE parameter. The two previous parameters CUDA_UUT and CYCLES are required for this profiling to work.
- **CYCLES_FILE**: This is a file that contains all the cycles one by one per line that will be used for our injections. A random cycle from this file is chosen before every execution. With this file, our framework is capable of performing injections on specific cycles like on a kernel invocation, on all the invocations of a kernel, or the whole application. A useful command on how to create this file for a chosen kernel with the help of profile=1 can be found here (see appendix 13.2).
- **MAX_REGISTERS_USED**: This is the maximum number of registers that a kernel uses per thread.
- **SHADER_USED**: This is the SIMT core that a kernel uses.
- **SUCCESS_MSG, FAILED_MSG**: This is the success and failure message respectively that an application prints after its own evaluation.
- **TIMEOUT_VAL**: This is the timeout of an execution which is useful in case the execution of an application hangs. The format is the one needed for the timeout command in Linux.
- **LMEM_SIZE_BITS:** This is the size in bits that a kernel uses for the local memory per thread.

- **SMEM_SIZE_BITS**: This is the size in bits that a kernel uses for the shared memory per CTA.

Per injection campaign parameters

- **profile=0**: By setting the profile value to 0, the profiling procedure will be disabled and the actual injection campaigns will be executed.
- **components_to_flip**: This is the hardware structure on which the injections will be applied. The value that describes a specific structure can be found within the campaign script. If a user wishes can also perform injections on multiple components per execution by inserting more than one component value with a colon as a delimiter. For example, with components_to_flip=0:2 injections will be done on both register file and shared memory at the same execution.
- **register_rand_n**: This is the number of the register that the transient faults will be injected. In this framework we are not targeting specific registers by name, so the value can be a number between 1 to MAX_REGISTERS_USED. Again this parameter can be crafted with more registers using a colon as a delimiter in case we want to inject the same fault on multiple registers and the same practice has been applied to all the parameters that end with '_n'. Furthermore, a '_rand' on a parameter's name indicates that on each execution the value will be changed randomly between some boundaries.
- **reg_bitflip_rand_n**: This is the specific bit that will be flipped.
- **per_warp**: If activated with the value of 1 then #register_rand_n registers will have their #reg_bitflip_rand_n bits flipped on every thread of an active warp. Otherwise, one running thread only will be affected.
- **shared_mem_bitflip_rand_n**: Same as reg_bitflip_rand_n but for the shared memory. This will randomly choose, in every execution, value(s) between 1 to SMEM_SIZE_BITS.
- **blocks**: This is on how many running CTAs, hence shared memories, to inject #shared_mem_bitflip_rand_n bit flips.
- **l1d_cache_bitflip_rand_n**: Same as reg_bitflip_rand_n but for the L1 data cache. This will randomly choose, in every execution, value(s) between 1 to L1D_SIZE_BITS.
- **l1d_shader_rand_n**: This is in which running SIMT core, hence L1 data cache, to inject shared_mem_bitflip_rand_n bit flips.
- **l1t_cache_bitflip_rand_n**, **l1t_shader_rand_n**: Same like L1 data cache but they are used for the texture cache.
- **l2_cache_bitflip_rand_n**: Same as reg_bitflip_rand_n but for L2 cache. This will randomly choose, in every execution, value(s) between 1 to L2_SIZE_BITS.

8.3 Injection campaign and Evaluation

The fault injection campaign can be easily executed by simply running the campaign.sh 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 framework is implemented on top of GPGPU-Sim 4.0, then the steps of setting up the backend is the same as setting up the GPGPU-Sim 4.0 and can be found in [6].

After completion of every batch of fault injections, a parser post-processes the output of the experiments one by one and accumulates the results. The final results will be printed

when all the batches have finished and the script has quit. The parser classifies the fault effects of each experiment as Masked, Silent Data Corruption (SDC), or Detected Unrecoverable Error (DUE). Such fault effects are used in several injection-based studies [16].

- **Masked:** Faults in this category let the application run 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. These faults are difficult to identify as they occur without any indication that a fault has been recorded (an abnormal event such as an exception, etc.).
- **Detected Unrecoverable Error (DUE):** In this case, an error is recorded and the application reaches an abnormal state without the ability to recover.

We additionally use the term "**Performance**" as a fault effect which is nothing but a Masked fault effect where the total cycles of the application are different from the fault-free execution. An abstract evaluation process of an output execution can be viewed on the flowchart below.

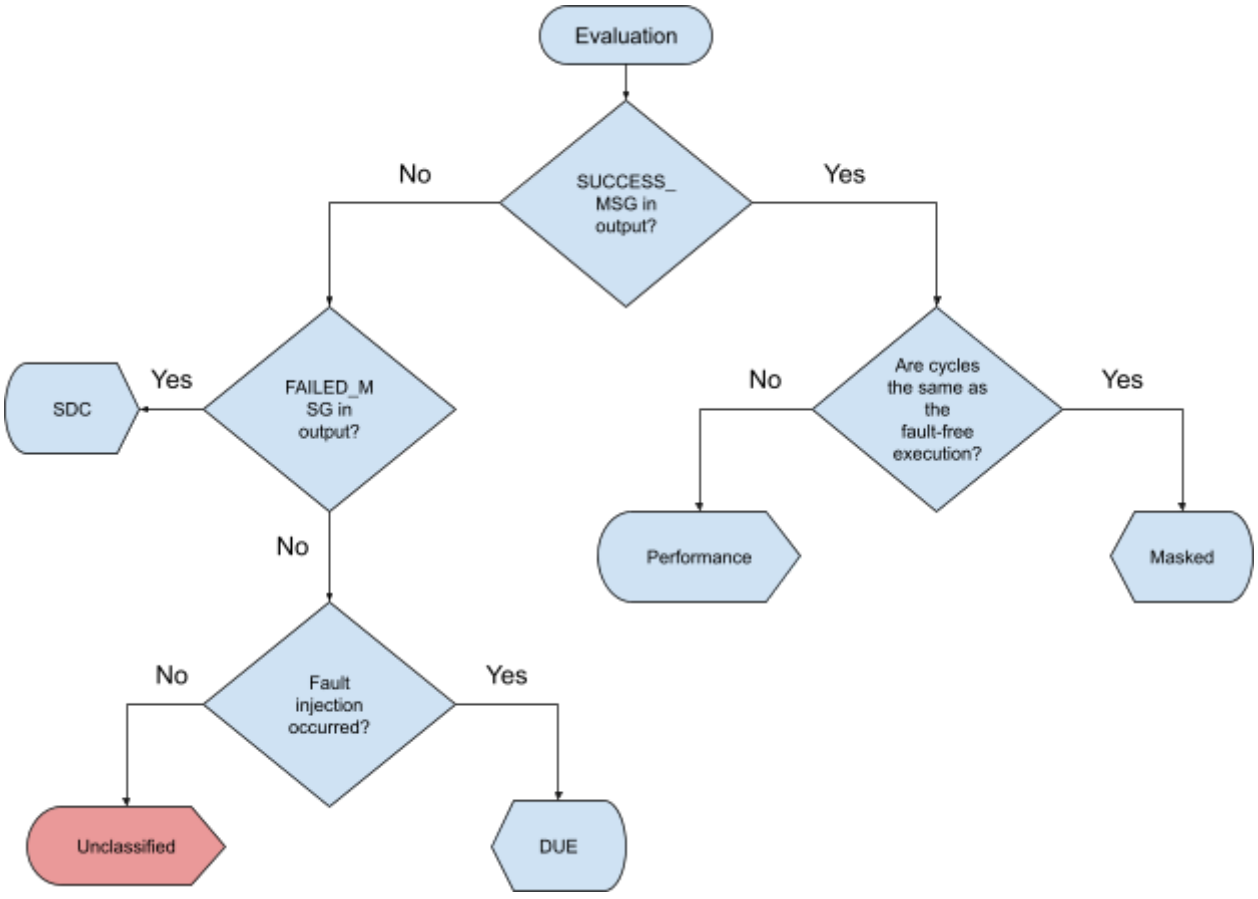

Figure 8: Fault effect parser flowchart

Evaluation process clarifications

- If an experiment output is evaluated as unclassified then the same experiment will be executed again. In a very few cases, GPGPU-Sim could not start the application at all because of some internal errors that are not of any concern.

- To differentiate the DUE class, a user should parse the results and decide about the specific errors. Such errors can be segmentation faults (crashes), timeouts, not aligned memory, etc. An automated parser would be ideal for this job but during the development, this was impossible to create since there was not much error data available.

9. HIGH LEVEL IMPLEMENTATION

In this chapter, we will talk about the backend part of the framework, and how it is implemented on top of GPGPU-Sim 4.0. We will first go through 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.

9.1 Technical challenges of GPGPU-Sim 4.0

One of the main challenges of the simulator is that it consists of three major modules which are 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 and 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 bit complicated as it had to communicate between these two modules. We had to use the performance module to know when we will inject the faults and use the functional module to know where we will inject them.

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, SIMT cores) to get access to the hardware components on which we want to inject the transient faults.

The third and last major challenge was 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 some kind of hooks during cache access and recognize accordingly if the fault should be injected or not.

9.2 Fault injection implementation

In this section, we will 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.

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 start but it 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 framework at a given cycle chooses a random active thread and injects the transient fault at a random register of that thread between the maximum register usage per 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 a whole warp but instead of choosing a random thread, the implementation chooses a random warp and applies the same transient faults on all of the threads belonging to that warp.

Local Memory
The same logic as the register file injections applies here but for the local memory of a thread and not the registers.

Shared Memory
Each block (CTA) 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.

L1 data cache
The L1 data cache per SIMT core is private in an NVIDIA GPU. The framework at a given cycle first chooses a random SIMT core between 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 a member of either the tag or 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 [6]. For multiple bit flips injections the procedure is the same for each bit.

L1 texture cache
The same logic applies here for fault injections as the L1 data cache.

L2 cache
The same logic also applies here like 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 [6]. 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 between 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.

9.3 Miscellaneous

9.3.1 L1 constant/instruction cache

Because of some technical difficulties and issues, these caches were not implemented for fault injection campaigns as we were wanted to. 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 be located, hence the hooks could not work properly. We assume that it happens because this information is wrongly calculated by the simulator. Luckily, the issue is propagated only to the performance part (constant cache hits/miss statistics) and does not affect the execution of the application. On the other hand, the reason that we skipped the L1 instruction cache is the implementation complexity. The simulator creates its own pseudo-assembly by mapping the SASS to PTXPlus instructions. Afterward, it parses the PTXPlus instructions one by one to create the necessary objects to execute the application. This procedure along with the fact that PTXPlus instructions do not have any binary representation made the implementation of injecting a bit flip on a SASS instruction difficult and time-consuming. Thereby we decided to omit these caches in our analysis.

9.3.2 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 maybe 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 didn't take into consideration other bits because we wanted to make the implementation simpler and we believe that the impact on the results would be negligible. The reason for the latter is that the fraction of those extra bits is minimal compared to the whole cache and so the probability of injecting a transient fault is very low.

The tag length that we were able to include consists of 57 bits. This number came up from the combination of the two following things.

- The simulator states at a comment within the code [17] that:
  // For generality, the tag includes both index and tag. This allows for more
  // complex set index calculations that can result in different indexes
  // mapping to the same set, thus the full tag + index is required to check
  // for hit/miss. Tag is now identical to the block address.
- The maximum size of the instruction used by PTXPlus is 64-bit and the offset is 7-bits since the cache line is 2^7 bytes. Based on the first bullet, we were unable to extract the real tag part from the block address in the implementation and as a result, we used 57-bits tag length.

10. APPLICATIONS

In the context of our reliability evaluation, we use 10 different applications from Rodinia benchmark suite [18]. In Table 4 we report the simulation time of the applications, the kernels of each application, the number of invocations of each kernel, the number of CTAs (gridDim), and the number of threads per CTA (blockDim) in each kernel.

Hot Spot (HS): HS estimates processor temperature based on an architectural floor plan and simulated power measurements. We try HS with input for temperature and power values that are organized on two individual 256x256 matrices.

K-Means (KM): KM is a data-mining algorithm that features a high degree of data parallelism. We run KM with 800 objects and each object consists of 34 features.

Speckle Reducing Anisotropic Diffusion (SRAD): SRAD is a diffusion method for ultrasonic and radar imaging applications based on partial differential equations. We examined both versions of this algorithm:

- **SRAD version 1 (SRAD1):** We use SRAD1 with 2 iterations, 0.5 saturation coefficient, 128 rows and columns in the input image.
- **SRAD version 2 (SRAD2):** We use SRAD2 with 256 rows and columns in the domain, (y1=0,y2=127,x1=0,x2=127) positions of the speckle, 0.5 lambda value, 2 iterations.

Lower Upper Decomposition (LUD): LUD is an algorithm that calculates the solutions of a set of linear equations. We run LUD with an internally generated 128x128 matrix.

Breadth-First Search (BFS): BFS is a breadth-first search algorithm that traverses all the connected components in a graph. We use BFS with an input of 32K nodes.

Pathfinder (PATHF): PATHF finds a path on a grid from the bottom to the top with the smallest accumulated weights and each step of the path moves straight ahead or diagonally ahead. We run PATHF with 10000 rows, 100 columns and 20 height.

Needleman-Wunsch (NW): NW is a nonlinear global optimization method for DNA sequence alignments. We run NW with 288 length of both sequences and 10 penalty value.

Gaussian Elimination (GE): GE is an algorithm for solving systems of linear equations. We employ GE to solve a system of 80 linear equations.

Backpropagation (BP): BP is an algorithm for supervised learning of artificial neural networks using gradient descent. We use BP with 8192 number of input elements.

Table 4: Applications

Application	Simulation time (s)	kernel	Invocations	gridDim	blockDim
HS	180	_Z14calculate_tempiPfS_S_iiiiffffff	1	22x22	16x16
KM	160	_Z14invert_mappingPfS_ii	1	4	256
		_Z11kmeansPointPfiiiPiS_S_S0_	25	2x2	256
SRAD1	119	_Z4sradfiilPiS_S_S_PfS0_S0_S0_fS0_S0_	2	32	512
		_Z5srad2fiilPiS_S_S_PfS0_S0_S0_S0_S0_	2	32	512
		Z6reduceliiPfS	4	32,1,32,1	512
		_Z7extractlPf	1	32	512
		_Z7preparelPfS_S_	1	32	512
		_Z8compresslPf	1	32	512
SRAD2	178	_Z11srad_cuda_1PfS_S_S_S_S_iif	2	16x16	16x16
		_Z11srad_cuda_2PfS_S_S_S_S_iiff	2	16x16	16x16
LUD	122	_Z12lud_diagonalPfii	8	1	16
		_Z12lud_internalPfii	7	7x7 to 1x1	16x16
		_Z13lud_perimeterPfii	7	7 to 1	32
BFS	120	_Z6KernelP4NodePiPbS2_S2_S1_i	10	64	512
		_Z7Kernel2PbS_S_S_i	10	64	512
PATHF	150	_Z14dynproc_kerneliPiS_S_iiii	5	47	256
NW	118	_Z20needle_cuda_shared_1PiS_iiii	18	1 to 18	16
		_Z20needle_cuda_shared_2PiS_iiii	17	17 to 1	16
GE	151	_Z4Fan1PfS_ii	79	1	512
		_Z4Fan2PfS_S_iii	79	20x20	4x4
BP	120	_Z22bpnn_layerforward_CUDAPfS_S_S_ii	1	512	16x16
		_Z24bpnn_adjust_weights_cudaPfiS_iS_S_	1	512	16x16

11. EXPERIMENTAL RESULTS

In this chapter, we will discuss the way that we used GPGPU injector 4.0 for our analysis and then we will present the results of our reliability and performance evaluation for all applications of the experimental analysis. Apart from reporting the overall application vulnerability, a breakdown into kernels' AVF per application for all the hardware structures of our study can be found in appendix 13.4.

We used GPGPU injector 4.0 by injecting a single bit flip on each supported hardware structure (register file, shared memory, L1 data/texture cache, L2 cache) for every kernel of an application using the PTXPlus mode of the simulator. Furthermore, in order to inject a transient fault on a kernel we took into consideration all of its invocations (meaning all of the dynamic kernel of a static kernel) otherwise it would be time consuming to examine every invocation one by one. 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 was done with 3000 application executions where a single bit was flipped on each execution. This number comes from the formula of [28] and results in a statistical safe number of fault injection with confidence level 99% and error margin less than 2%. We did not include the local memory in our analysis since all of the applications that we examined did not use this memory at all.

For our results, we used the RTX2060 and QV100 NVIDIA cards and details about their microarchitecture can be found in Table 5 and more details about the L1 and L2 cache architectures, (e.g. number of sets, write policy) can be found in appendix 13.3.

Table 5: RTX 2060 and Quadro GV100 microarchitecture

	RTX 2060	Quadro GV100
SMs	30	80
Warp size	32	32
Maximum Threads per SM	1024	2048
Maximum CTAs per SM	32	32
Registers per SM (size per register: 4 bytes)	65536	65536
Shared Memory per SM	64 KB	96 KB
L1 data cache size per SM	64 KB	32 KB
	67.56 KB (with 57 tag bits per cache line)	33.78 KB (with 57 tag bits per cache line)
L1 texture cache size per SM	128 KB	128 KB
	135.13 KB (with 57 tag bits per cache line)	135.13 KB (with 57 tag bits per cache line)

L1 instruction cache per SM	128 KB	128 KB
	135.13 KB (with 57 tag bits per cache line)	135.13 KB (with 57 tag bits per cache line)
L1 constant cache per SM	64 KB	64 KB
	71.13 KB (with 57 tag bits per cache line)	71.13 KB (with 57 tag bits per cache line)
L2 cache size	3 MB	6 MB
	3.17 MB (with 57 tag bits per cache line)	6.33 MB (with 57 tag bits per cache line)

There are some important things 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. Secondly, we had to use SM compute capability < 20 since the simulator did not support the PTXPlus mode otherwise. As a result, despite that we use relatively new GPU cards we are forced to execute on them, through the PTXPlus mode, a SASS version which is much lower than what they support. Lastly, for our analysis we changed some configuration parameters of the simulator from their default values which are:

- **gpgpu_kernel_launch_latency:** This is the kernel launch latency in cycles and could make the experiments slower that's why we set it to 0.
- **gpgpu_perfect_inst_const_cache**: This is a perfect instruction and constant cache mode when activated where all instructions and constant cache accesses never miss. We decided to disable this flag as we wanted a more realistic execution as possible.
- **gpgpu_flush_l1_cache**: This is a parameter that flushes L1 data cache at the end of each kernel call when activated. We disabled this flag in order to benefit from the locality of the cache. However, it will be also correct to run our experiments with this parameter enabled since NVIDIA is using such mode to many of its GPU cards. In general, an individual can make experiments in both ways without changing the main purpose of this analysis.
- **gpgpu_adaptive_cache_config** (applies to Quadro GV100 only): This is a parameter that features adaptive L1 data cache size, based on the shared memory, when enabled. For example, the default configuration for Quadro GV100 is 32KB L1 data and 96KB shared memory but when the shared memory is zero then the L1 data cache size becomes 128KB. We disabled this parameter not only to make our experiments simpler, but GPGPU-Sim also suggests disabling this mode in case of multi kernels/apps execution.

At Figure 9, we present the overall architectural vulnerability factor (AVF) per application for the two GPU cards under test. The AVF is calculated based on the procedure that we talked about in Chapter 6. Furthermore, the IPC as well as the average warp occupancy per application is shown in Table 6. 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 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.

The first thing that we can notice from the results is that LUD, NW and GE have a higher tolerance than the others. This is a trend that can be also verified from GUFI [15] and it is caused by the fact that the hardware components are under small pressure as a result of their input dataset Thus, the majority of fault injection experiments hit idle resources. For a comparison in Figures 10 & 11 we present the AVF of the register file for the common benchmarks from our experiments and GUFI's results respectively. Low IPC along with low average warp occupancy is an indication that an application does not pressure the GPU enough. As we can see in Table 6 for RTX 2060 (same applies to Quadro GV100) the small hardware pressure of LUD, NW and GE is due to the low IPC (14.68, 14.07 and 76.63 respectively) along with the low average warp occupancy (3.87%, 3.13% and 15.57% respectively). On the other hand, high IPC and high average warp occupancy does not mean that the applications will be more vulnerable to transient faults. For example, HS, SRAD2 and BP applications put a lot of pressure on the GPU, but BP looks to be much more resilient than the first two.

Another interesting observation comes from the comparison of the AVF results from the two NVIDIA GPUs. For all applications, except the HS, the RTX 2060 seems to be more vulnerable than Quadro GV100. It is something we were expecting as the hardware of Quadro GV100 can execute more demanding workloads which implies that on RTX 2060 the GPU hardware is under more pressure for the same workload. This is due to the fact that the RTX 2060 has a much smaller number of SMs and smaller hardware component sizes such as L2 cache and shared memory (see Table 5). Regarding the oulier HS application, we believe that it is caused mainly due to the GPGPU-Sim modeling issue of the register file that we had to overcome by using the derating factor (df_reg) as we explained in Chapter 6. The total number of the register file's fault effects (SDCs, DUEs) of HS on both cards are almost the same but on Quadro GV100 the number of register usage per SM is much higher. Therefore, the impact of the register file's fault effects during AVF calculation is bigger on Quadro GV100, hence the higher total AVF.

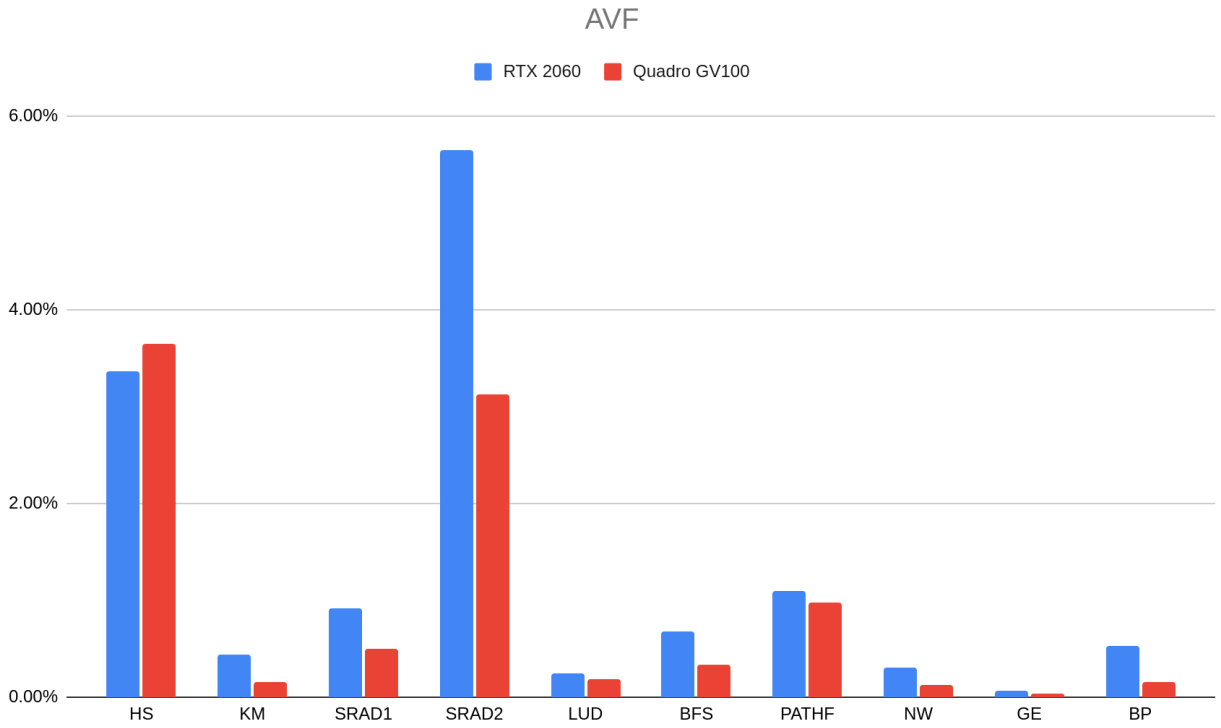

Figure 9: AVF results for RTX 2060 and Quadro GV100.

Table 6: IPC and Average Warp Occupancy of the simulated applications.

Application	RTX206		Quadro GV100	
	IPC	Average Warp Occupancy	IPC	Average Warp Occupancy
HS	1634.62	91%	2911.78	69.56%
KM	125.72	19.88%	111.03	9.95%
SRAD1	542.9	51.45%	544.96	24.94%
SRAD2	817.64	90.4%	1536.76	38.06%
LUD	14.68	3.87%	14.3	1.95%
BFS	99	58%	127.29	15.81%
PATHF	737.98	38.36%	932.77	12.48%
NW	14.07	3.13%	13.1075	1.56%
GE	76.63	15.57%	67.28	8.3%
BP	1415.19	90.62%	3121.23	71.54%

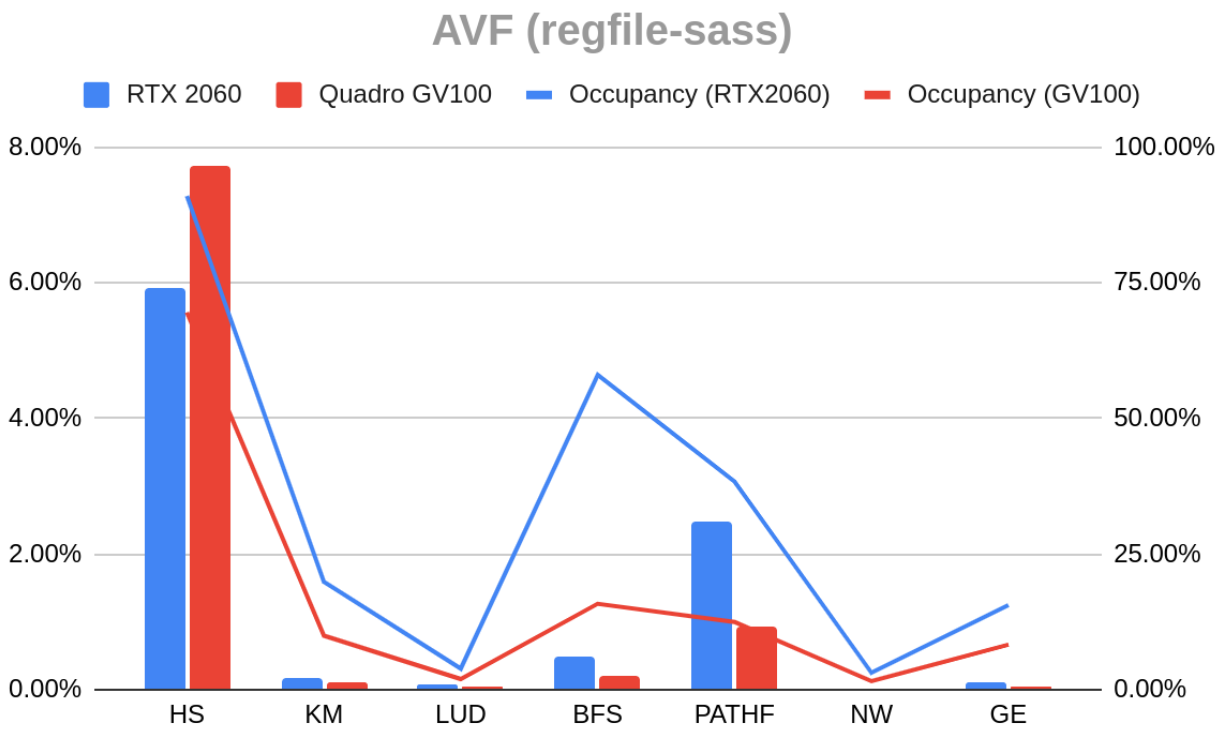

Figure 10: AVF results for RTX 2060 and Quadro GV100 for the register file.

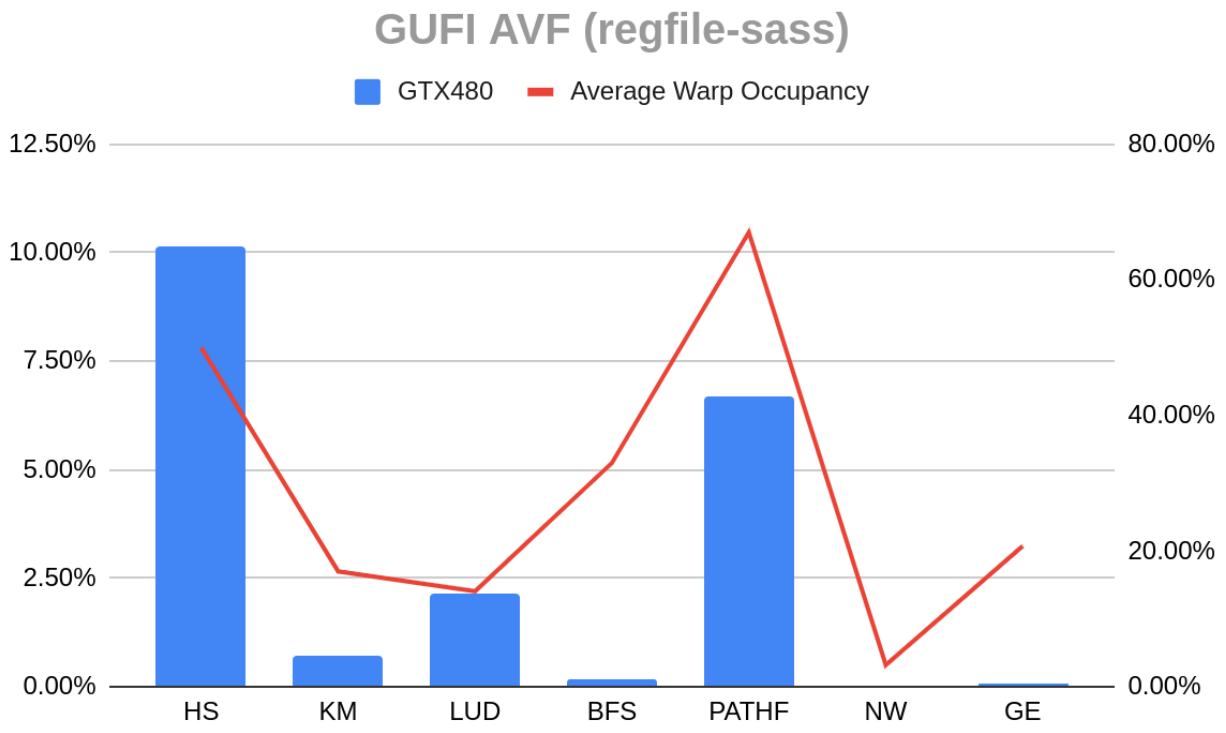

Figure 11: GUFI's AVF results for the register file on GTX480.

In figure 12 we show the RTX 2060 percentage of executions resulted in a fault effect (SDC, DUE) over the total executions per static kernel on 3 selected applications HS, KM and BFS. Register file and shared memory derating factors have been applied. As we can

see for HS the fault effect ratio on L1 data cache is zero. In general this can happen for two reasons. The first one is when the cache access ratio is very low, thus the probability of injecting a transient fault is very low as well. The second reason is when the application does not take advantage of the cache locality and as a result the cache lines get replaced on every access, hence overwriting the injected transient faults (if any). The second reason applies for the HS execution which has a significant amount of L1 data cache accesses but the miss ratio is 99%.

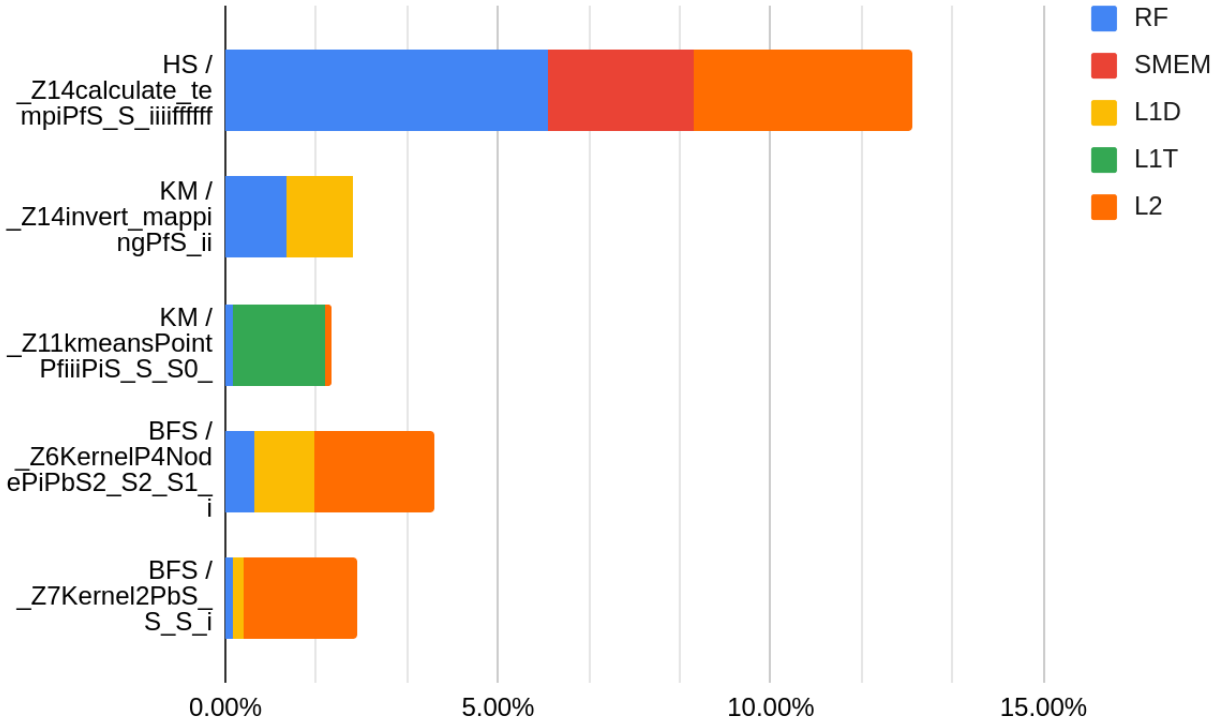

Figure 12: Fault effect ratio per static kernel on selected applications (RTX 2060)

12. CONCLUSIONS

We have presented GPGPU injector 4.0, a detailed fault injection framework built on top of a state of the art microarchitectural simulator of GPGPU architectures, GPGPU-sim. 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 10 different CUDA programs from Rodinia benchmark suite 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 and Quadro GV100. 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.

13. APPENDIX

13.1 Campaign run script

```
#!/bin/bash

# ------------------------------------------------ START ONE-TIME PARAMETERS
------------------------------------------------
# needed by gpgpu-sim for real register usage on PTXPlus mode
export PTXAS_CUDA_INSTALL_PATH=/usr/local/cuda-11.2

CONFIG_FILE=./gpgpusim.config
TMP_DIR=./logs
CACHE_LOGS_DIR=./cache_logs
TMP_FILE=tmp.out
RUNS=7
BATCH=$(( $(grep -c ^processor /proc/cpuinfo) - 1 )) # -1 core for computer not to hang
DELETE_LOGS=0 # if 1 then all logs will be deleted at the end of the script
# ------------------------------------------------ END ONE-TIME PARAMETERS
------------------------------------------------

# ------------------------------------------------ START PER GPGPU CARD PARAMETERS
------------------------------------------------
# L1 cache size per SIMT core (30 SIMT cores on RTX 2060, 30 clusters with 1 core each) - 80 for Volta
QV100
L1D_SIZE_BITS=276736  # nsets=1, line_size=128 bytes + 57 bits, assoc=256
L1C_SIZE_BITS=582656 # nsets=128, line_size=64 bytes + 57 bits, assoc=8
L1T_SIZE_BITS=1106944 # nsets=4, line_size=128 bytes + 57 bits, assoc=256
# L2 cache total size from all sub partitions
L2_SIZE_BITS=53133312 # (nsets=32, line_size=128 bytes + 57 bits, assoc=24) x 24 sub partitions (64 sub
partitions in Volta QV100)
# ------------------------------------------------ END PER GPGPU CARD PARAMETERS
------------------------------------------------

# ------------------------------------------------ START PER KERNEL/APPLICATION PARAMETERS (+profile=1)
------------------------------------------------
CUDA_UUT="./srad 2 0.5 128 128"
# total cycles for all kernels
CYCLES=49799
# Get the exact cycles, max registers and SIMT cores used for each kernel with profile=1
# fix cycles.txt with kernel execution cycles
# (e.g. seq 1 10 >> cycles.txt, or multiple seq commands if a kernel has multiple executions)
# use the following command from profiling execution for easier creation of cycles.txt file
# e.g. grep "_Z12lud_diagonalPfii" cycles.in | awk  '{ system("seq " $12 " " $18 ">> cycles.txt")}'
CYCLES_FILE=./cycles.txt
MAX_REGISTERS_USED=24
SHADER_USED="0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34
35 36 37 38 39 40 41 42 43 44 45 46 47 48 66 67 68 69 70 71 72 73 74 75 76 77 78 79"
SUCCESS_MSG='Test PASSED'
FAILED_MSG='Test FAILED'
TIMEOUT_VAL=400s
DATATYPE_SIZE=32
# lmem and smem values are taken from gpgpu-sim ptx output per kernel
# e.g. GPGPU-Sim PTX: Kernel '_Z9vectorAddPKdS0_Pdi' : regs=8, lmem=0, smem=0, cmem=380
# if 0 put a random value > 0
LMEM_SIZE_BITS=10
SMEM_SIZE_BITS=1024
# ------------------------------------------------ END PER KERNEL/APPLICATION PARAMETERS (+profile=1)
------------------------------------------------

FAULT_INJECTION_OCCURRED="Fault injection"
CYCLES_MSG="gpu_tot_sim_cycle ="
```

```
masked=0
performance=0
SDC=0
crashes=0

# ------------------------------------------- START PER INJECTION CAMPAIGN PARAMETERS (profile=0)
-------------------------------------------
# 0: perform injection campaign, 1: get cycles of each kernel, 2: get mean value of active threads, during
all cycles in CYCLES_FILE, per SM
profile=0
# 0:RF, 1:local_mem, 2:shared_mem, 3:L1D_cache, 4:L1C_cache, 5:L1T_cache, 6:L2_cache (e.g.
components_to_flip=0:1 for both RF and local_mem)
components_to_flip=0
# 1: per warp bit flip, 0: per thread bit flip
per_warp=0
# in which kernels to inject the fault. e.g. 0: for all running kernels, 1: for kernel 1, 1:2 for kernel 1
& 2
kernel_n=0
# in how many blocks (smems) to inject the bit flip
blocks=1

initialize_config() {
    # random number for choosing a random thread after thread_rand % #threads operation in gpgpu-sim
    thread_rand=$(shuf -i 0-6000 -n 1)
    # random number for choosing a random warp after warp_rand % #warp operation in gpgpu-sim
    warp_rand=$(shuf -i 0-6000 -n 1)
    # random cycle for fault injection
    total_cycle_rand="$(shuf ${CYCLES_FILE} -n 1)"
    # in which registers to inject the bit flip
    register_rand_n="$(shuf -i 1-${MAX_REGISTERS_USED} -n 1)";
register_rand_n="${register_rand_n//$'\n'/:}"
    # example: if -i 1-32 -n 2 then the two commands below will create a value with 2 random numbers,
between [1,32] like 3:21. Meaning it will flip 3 and 21 bits.
    reg_bitflip_rand_n="$(shuf -i 1-${DATATYPE_SIZE} -n 1)";
reg_bitflip_rand_n="${reg_bitflip_rand_n//$'\n'/:}"
    # same format like reg_bitflip_rand_n but for local memory bit flips
    local_mem_bitflip_rand_n="$(shuf -i 1-${LMEM_SIZE_BITS} -n 3)";
local_mem_bitflip_rand_n="${local_mem_bitflip_rand_n//$'\n'/:}"
    # random number for choosing a random block after block_rand % #smems operation in gpgpu-sim
    block_rand=$(shuf -i 0-6000 -n 1)
    # same format like reg_bitflip_rand_n but for shared memory bit flips
    shared_mem_bitflip_rand_n="$(shuf -i 1-${SMEM_SIZE_BITS} -n 1)";
shared_mem_bitflip_rand_n="${shared_mem_bitflip_rand_n//$'\n'/:}"
    # randomly select one or more shaders for L1 data cache fault injections
    l1d_shader_rand_n="$(shuf -e ${SHADER_USED} -n 1)"; l1d_shader_rand_n="${l1d_shader_rand_n//$'\n'/:}"
    # same format like reg_bitflip_rand_n but for L1 data cache bit flips
    l1d_cache_bitflip_rand_n="$(shuf -i 1-${L1D_SIZE_BITS} -n 1)";
l1d_cache_bitflip_rand_n="${l1d_cache_bitflip_rand_n//$'\n'/:}"
    # randomly select one or more shaders for L1 constant cache fault injections
    l1c_shader_rand_n="$(shuf -e ${SHADER_USED} -n 1)"; l1c_shader_rand_n="${l1c_shader_rand_n//$'\n'/:}"
    # same format like reg_bitflip_rand_n but for L1 constant cache bit flips
    l1c_cache_bitflip_rand_n="$(shuf -i 1-${L1C_SIZE_BITS} -n 1)";
l1c_cache_bitflip_rand_n="${l1c_cache_bitflip_rand_n//$'\n'/:}"
    # randomly select one or more shaders for L1 texture cache fault injections
    l1t_shader_rand_n="$(shuf -e ${SHADER_USED} -n 1)"; l1t_shader_rand_n="${l1t_shader_rand_n//$'\n'/:}"
    # same format like reg_bitflip_rand_n but for L1 texture cache bit flips
    l1t_cache_bitflip_rand_n="$(shuf -i 1-${L1T_SIZE_BITS} -n 1)";
l1t_cache_bitflip_rand_n="${l1t_cache_bitflip_rand_n//$'\n'/:}"
    # same format like reg_bitflip_rand_n but for L2 cache bit flips
    l2_cache_bitflip_rand_n="$(shuf -i 1-${L2_SIZE_BITS} -n 1)";
l2_cache_bitflip_rand_n="${l2_cache_bitflip_rand_n//$'\n'/:}"
# ------------------------------------------- END PER INJECTION CAMPAIGN PARAMETERS (profile=0)
```

```
-----------------------------------------------

    sed -i -e "s/^-components_to_flip.*$/-components_to_flip ${components_to_flip}/" ${CONFIG_FILE}
    sed -i -e "s/^-profile.*$/-profile ${profile}/" ${CONFIG_FILE}
    sed -i -e "s/^-last_cycle.*$/-last_cycle ${CYCLES}/" ${CONFIG_FILE}
    sed -i -e "s/^-thread_rand.*$/-thread_rand ${thread_rand}/" ${CONFIG_FILE}
    sed -i -e "s/^-warp_rand.*$/-warp_rand ${warp_rand}/" ${CONFIG_FILE}
    sed -i -e "s/^-total_cycle_rand.*$/-total_cycle_rand ${total_cycle_rand}/" ${CONFIG_FILE}
    sed -i -e "s/^-register_rand_n.*$/-register_rand_n ${register_rand_n}/" ${CONFIG_FILE}
    sed -i -e "s/^-reg_bitflip_rand_n.*$/-reg_bitflip_rand_n ${reg_bitflip_rand_n}/" ${CONFIG_FILE}
    sed -i -e "s/^-per_warp.*$/-per_warp ${per_warp}/" ${CONFIG_FILE}
    sed -i -e "s/^-kernel_n.*$/-kernel_n ${kernel_n}/" ${CONFIG_FILE}
    sed -i -e "s/^-local_mem_bitflip_rand_n.*$/-local_mem_bitflip_rand_n ${local_mem_bitflip_rand_n}/"
${CONFIG_FILE}
    sed -i -e "s/^-block_rand.*$/-block_rand ${block_rand}/" ${CONFIG_FILE}
    sed -i -e "s/^-block_n.*$/-block_n ${blocks}/" ${CONFIG_FILE}
    sed -i -e "s/^-shared_mem_bitflip_rand_n.*$/-shared_mem_bitflip_rand_n ${shared_mem_bitflip_rand_n}/"
${CONFIG_FILE}
    sed -i -e "s/^-shader_rand_n.*$/-shader_rand_n ${shader_rand_n}/" ${CONFIG_FILE}
    sed -i -e "s/^-l1d_shader_rand_n.*$/-l1d_shader_rand_n ${l1d_shader_rand_n}/" ${CONFIG_FILE}
    sed -i -e "s/^-l1d_cache_bitflip_rand_n.*$/-l1d_cache_bitflip_rand_n ${l1d_cache_bitflip_rand_n}/"
${CONFIG_FILE}
    sed -i -e "s/^-l1c_shader_rand_n.*$/-l1c_shader_rand_n ${l1c_shader_rand_n}/" ${CONFIG_FILE}
    sed -i -e "s/^-l1c_cache_bitflip_rand_n.*$/-l1c_cache_bitflip_rand_n ${l1c_cache_bitflip_rand_n}/"
${CONFIG_FILE}
    sed -i -e "s/^-l1t_shader_rand_n.*$/-l1t_shader_rand_n ${l1t_shader_rand_n}/" ${CONFIG_FILE}
    sed -i -e "s/^-l1t_cache_bitflip_rand_n.*$/-l1t_cache_bitflip_rand_n ${l1t_cache_bitflip_rand_n}/"
${CONFIG_FILE}
    sed -i -e "s/^-l2_cache_bitflip_rand_n.*$/-l2_cache_bitflip_rand_n ${l2_cache_bitflip_rand_n}/"
${CONFIG_FILE}
}

gather_results() {
    for file in ${TMP_DIR}${1}/${TMP_FILE}*; do
        grep -iq "${SUCCESS_MSG}" $file; success_msg_grep=$(echo $?)
        grep -i "${CYCLES_MSG}" $file | tail -1 | grep -q "${CYCLES}"; cycles_grep=$(echo $?)
        grep -iq "${FAILED_MSG}" $file; failed_msg_grep=$(echo $?)
        result=${success_msg_grep}${cycles_grep}${failed_msg_grep}
        case $result in
        "001")
            let RUNS--
            let masked++ ;;
        "011")
            let RUNS--
            let masked++
            let performance++ ;;
        "100" | "110")
            let RUNS--
            let SDC++ ;;
        *)
            grep -iq "${FAULT_INJECTION_OCCURRED}" $file
            if [ $? -eq 0 ]; then
                let RUNS--
                let crashes++
                echo "Crash appeared in loop ${1}" # DEBUG
            else
                echo "Unclassified in loop ${1} ${result}" # DEBUG
            fi ;;
        esac
    done
}

parallel_execution() {
```

```
    batch=$1
    mkdir ${TMP_DIR}${2} > /dev/null 2>&1
    for i in $( seq 1 $batch ); do
        initialize_config
        # unique id for each run (e.g. r1b2: 1st run, 2nd execution on batch)
        sed -i -e "s/^-run_uid.*$/-run_uid r${2}b${i}/" ${CONFIG_FILE}
        cp ${CONFIG_FILE} ${TMP_DIR}${2}/${CONFIG_FILE}${i} # save state
        timeout ${TIMEOUT_VAL} $CUDA_UUT > ${TMP_DIR}${2}/${TMP_FILE}${i} 2>&1 &
    done
    wait
    gather_results $2
    if [[ "$DELETE_LOGS" -eq 1 ]]; then
        rm _ptx* _cuobjdump_* _app_cuda* *.ptx f_tempfile_ptx gpgpu_inst_stats.txt > /dev/null 2>&1
        rm -r ${TMP_DIR}${2} > /dev/null 2>&1 # comment out to debug output
    fi
    if [[ "$profile" -ne 1 ]]; then
        # clean intermediate logs anyway if profile != 1
        rm _ptx* _cuobjdump_* _app_cuda* *.ptx f_tempfile_ptx gpgpu_inst_stats.txt > /dev/null 2>&1
    fi
}

main() {
    if [[ "$profile" -eq 1 ]] || [[ "$profile" -eq 2 ]]; then
        RUNS=1
    fi
    # MAX_RETRIES to avoid flooding the system storage with logs infinitely if the user
    # has wrong configuration and only Unclassified errors are returned
    MAX_RETRIES=3
    LOOP=1
    mkdir ${CACHE_LOGS_DIR} > /dev/null 2>&1
    while [[ $RUNS -gt 0 ]] && [[ $MAX_RETRIES -gt 0 ]]
    do
        echo "runs left ${RUNS}" # DEBUG
        let MAX_RETRIES--
        LOOP_START=${LOOP}
        unset LAST_BATCH
        if [ "$BATCH" -gt "$RUNS" ]; then
            BATCH=${RUNS}
            LOOP_END=$(($LOOP_START))
        else
            BATCH_RUNS=$(($RUNS/$BATCH))
            if (( $RUNS % $BATCH )); then
                LAST_BATCH=$(($RUNS-$BATCH_RUNS*$BATCH))
            fi
            LOOP_END=$(($LOOP_START+$BATCH_RUNS-1))
        fi

        for i in $( seq $LOOP_START $LOOP_END ); do
            parallel_execution $BATCH $i
            let LOOP++
        done

        if [[ ! -z ${LAST_BATCH+x} ]]; then
            parallel_execution $LAST_BATCH $LOOP
            let LOOP++
        fi
    done

    if [[ $MAX_RETRIES -eq 0 ]]; then
        echo "Probably \"${CUDA_UUT}\" was not able to run! Please make sure the execution with GPGPU-Sim
works!"
    else
        echo "Masked: ${masked} (performance = ${performance})"
```

```
        echo "SDCs: ${SDC}"
        echo "DUEs: ${crashes}"
    fi
    if [[ "$DELETE_LOGS" -eq 1 ]]; then
        rm -r ${CACHE_LOGS_DIR} > /dev/null 2>&1 # comment out to debug cache logs
    fi
}

main "$@"
exit 0
```

13.2 Useful commands for campaign script preparation

13.2.1 CYCLES_FILE creation per kernel

If a user wants to create the input cycle file for the framework that corresponds to all the cycles of a static kernel then the process might be very inefficient when there are hundreds or thousands of cycles. For that reason the following bash command can be used which can take an input file “cycles.in” and the cycle file of the static kernel with name “_Z12lud_diagonalPfii” will be created.

grep "_Z12lud_diagonalPfii" cycles.in | awk '{ system("seq " $12 " " $18 ">> cycles.txt")}'

The input file “cycles.in” can have the format as shown in Figure 13 below. This information can be also retrieved when a user executes the framework with profile=1 mode and will be printed on the GPGPU-Sim output during the last cycle of the application.

```
Kernel = 1 with name = _Z12lud_diagonalPfii, started on cycle = 1 and finished on cycle = 18729
Kernel = 2 with name = _Z13lud_perimeterPfii, started on cycle = 18922 and finished on cycle = 51731
Kernel = 3 with name = _Z12lud_internalPfii, started on cycle = 51924 and finished on cycle = 54603
Kernel = 4 with name = _Z12lud_diagonalPfii, started on cycle = 54702 and finished on cycle = 71326
Kernel = 5 with name = _Z13lud_perimeterPfii, started on cycle = 71519 and finished on cycle = 99626
Kernel = 6 with name = _Z12lud_internalPfii, started on cycle = 99819 and finished on cycle = 101076
Kernel = 7 with name = _Z12lud_diagonalPfii, started on cycle = 101272 and finished on cycle = 117896
Kernel = 8 with name = _Z13lud_perimeterPfii, started on cycle = 118089 and finished on cycle = 146200
Kernel = 9 with name = _Z12lud_internalPfii, started on cycle = 146393 and finished on cycle = 147560
Kernel = 10 with name = _Z12lud_diagonalPfii, started on cycle = 147764 and finished on cycle = 164388
Kernel = 11 with name = _Z13lud_perimeterPfii, started on cycle = 164581 and finished on cycle = 187097
```

Figure 13: Input file for the command to create CYCLES_FILE

13.2.2 Stats accumulation per kernel

When there are multiple static kernels and multiple dynamic kernels on an application the statistics like cycles and instructions are scattered throughout the standard output of GPGPU-Sim. The three following bash commands can help a user accumulate these statistics per static kernel. The “_Z4Fan2PfS_S_iii” is the example’s kernel name and “tmp.out” is a text file that contains the simulator’s output. The result of GPU occupancy of a static kernel in the third command should later be divided by the number of dynamic kernels (invocations) of that static kernel.

- **Cycles:** *grep -r -A 2 "kernel_name = _Z4Fan2PfS_S_iii" tmp.out | grep "gpu_sim_cycle =" | awk '{print $NF}' | awk '{s+=$1} END {print s}'*

- **Instructions:** *grep -r -A 3 "kernel_name = _Z4Fan2PfS_S_iii" tmp.out | grep "gpu_sim_insn =" | awk '{print $NF}' | awk '{s+=$1} END {print s}'*
- **GPU occupancy:** *grep -r -A 9 "kernel_name = _Z4Fan2PfS_S_iii" tmp.out | grep "gpu_occupancy =" | awk '{print $NF}' | awk '{s+=$1} END {print s}'*

13.3 L1 & L2 cache architectures

Table 7: L1 data cache write policy

L1 data cache write policy		
	Local Memory	Global Memory
Write Hit	Write-back	Write-evict
Write Miss	Write no-allocate	Write no-allocate

Table 8: L2 cache write policy

L2 cache write policy		
	Local Memory	Global Memory
Write Hit	Write-back for L1 write-backs	Write-evict
Write Miss	Write no-allocate	Write no-allocate

Table 9: Caches architecture on RTX 2060

RTX 2060				
Cache	Number of sets	Cache line size (bytes)	Associativity	Evict policy
L1 data per SM	1	128	512	LRU
L1 texture per SM	4	128	256	LRU
L1 instruction per SM	64	128	16	LRU
L1 constant per SM	128	64	8	LRU
L2 per memory sub partition (24 sub partitions in total)	64	128	16	LRU

Table 10: Caches architecture on Quadro GV100

Quadro GV100				
Cache	Number of sets	Cache line size (bytes)	Associativity	Evict policy
L1 data per SM	1	128	256	LRU

L1 texture per SM	4	128	256	LRU
L1 instruction per SM	64	128	16	LRU
L1 constant per SM	128	64	8	LRU
L2 per memory sub partition (24 sub partitions in total)	32	128	24	LRU

13.4 AVF per kernel breakdown

Table 11: Breakdown of kernel's AVF for RTX 2060

RTX 2060				
Application	**kernel**	**Cycles**	**AVF_{kernel}**	**AVF**
HS	_Z14calculate_tempiPfS_S_iiiiffffff	13978	3.36%	3.36%
KM	_Z14invert_mappingPfS_ii	10753	0.47%	0.44%
	_Z11kmeansPointPfiiiPiS_S_S0_	249946	0.43%	
SRAD1	_Z4sradfiilPiS_S_S_PfS0_S0_S0_fS0_S0_	10103	1.59%	0.92%
	_Z5srad2fiilPiS_S_S_PfS0_S0_S0_S0_S0_	7528	1.63%	
	Z6reduceliiPfS	26792	0.52%	
	_Z7extractlPf	1834	0.70%	
	_Z7preparelPfS_S_	2184	0.63%	
	_Z8compresslPf	1547	0.76%	
SRAD2	_Z11srad_cuda_1PfS_S_S_S_S_iif	24552	5.41%	5.65%
	_Z11srad_cuda_2PfS_S_S_S_S_iiff	16034	6.01%	
LUD	_Z12lud_diagonalPfii	136639	0.18%	0.25%
	_Z12lud_internalPfii	10571	0.75%	
	_Z13lud_perimeterPfii	191601	0.26%	
BFS	_Z6KernelP4NodePiPbS2_S2_S1_i	71143	0.71%	0.67%
	_Z7Kernel2PbS_S_S_i	11844	0.44%	
PATHF	_Z14dynproc_kerneliPiS_S_iiii	63118	1.09%	1.09%
NW	_Z20needle_cuda_shared_1PiS_iiii	153960	0.48%	0.30%
	_Z20needle_cuda_shared_2PiS_iiii	123605	0.08%	
GE	_Z4Fan1PfS_ii	66857	0.02%	0.06%
	_Z4Fan2PfS_S_iii	70519	0.10%	

BP	_Z22bpnn_layerforward_CUDAPfS_S_S_ii	21947	0.45%	0.53%
	_Z24bpnn_adjust_weights_cudaPfiS_iS_S_	7199	0.79%	

Table 12: Breakdown of kernel's AVF for Quadro GV100

Quadro GV100				
Application	**kernel**	**Cycles**	**AVF_{kernel}**	**AVF**
HS	_Z14calculate_tempiPfS_S_iiiiffffff	7847	3.65%	3.65%
KM	_Z14invert_mappingPfS_ii	10969	0.50%	0.16%
	_Z11kmeansPointPfiiiPiS_S_S0_	284208	0.14%	
SRAD1	_Z4sradfiilPiS_S_S_PfS0_S0_S0_fS0_S0_	10947	0.91%	0.51%
	_Z5srad2fiilPiS_S_S_PfS0_S0_S0_S0_S0_	8223	0.73%	
	Z6reduceliiPfS	25107	0.26%	
	_Z7extractlPf	1839	0.63%	
	_Z7preparelPfS_S_	2199	0.23%	
	_Z8compresslPf	1484	0.73%	
SRAD2	_Z11srad_cuda_1PfS_S_S_S_S_iif	13706	2.76%	3.12%
	_Z11srad_cuda_2PfS_S_S_S_S_iiff	7888	3.75%	
LUD	_Z12lud_diagonalPfii	137036	0.11%	0.18%
	_Z12lud_internalPfii	12409	0.29%	
	_Z13lud_perimeterPfii	198149	0.23%	
BFS	_Z6KernelP4NodePiPbS2_S2_S1_i	54500	0.34%	0.33%
	_Z7Kernel2PbS_S_S_i	10046	0.28%	
PATHF	_Z14dynproc_kerneliPiS_S_iiii	49937	0.97%	0.97%
NW	_Z20needle_cuda_shared_1PiS_iiii	165197	0.19%	0.13%
	_Z20needle_cuda_shared_2PiS_iiii	132790	0.05%	
GE	_Z4Fan1PfS_ii	82643	0.01%	0.03%
	_Z4Fan2PfS_S_iii	73827	0.05%	
BP	_Z22bpnn_layerforward_CUDAPfS_S_S_ii	9411	0.10%	0.15%
	_Z24bpnn_adjust_weights_cudaPfiS_iS_S_	3804	0.28%	

ABBREVIATIONS

AVF	Architectural Vulnerability Factor
IVF	Intermittent Vulnerability Factor
H-AVF	Hard-Fault Architectural Vulnerability Factor
PVF	Program Vulnerability Factor
HVF	Hardware Vulnerability Factor
ACE	Architectural Correct Execution
SM	Streaming multiprocessor
SP	Stream Processor
CTA	Common Thread Array
SIMD	single instruction, multiple data
PTX	Parallel Thread Execution assembly
SIMT	Single Instruction Multiple Thread
FIT	Failure in TIme
MTBF	Mean Time Between Failures
SDC	Silent Data Corruption
DUE	Detected Unrecoverable Error

BIBLIOGRAPHY – REFERENCES

[1] R. Borgo, K. Brodlie, State of the Art Report on GPU Visualization.

[2] Herna´ndez M, Guerrero GD, Cecilia JM, Garcı´a JM, Inuggi A, et al. (2013) Accelerating Fibre Orientation Estimation from Diffusion Weighted MagneticResonance Imaging Using GPUs. PLoS ONE 8(4): e61892. doi:10.1371/journal.pone.0061892.

[3] A. Lippert, NVIDIA GPU Architecture for General Purpose Computing

[4] https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#memory-hierarchy

[5] http://download.nvidia.com/developer/cuda/seminar/TDCI_Arch.pdf

[6] Ali Bakhoda, George Yuan, Wilson W. L. Fung, Henry Wong, Tor M. Aamodt, Analyzing CUDA Workloads Using a Detailed GPU Simulator, in IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), Boston, MA, April 19-21, 2009.

[7] R.C.Baumann, “Soft errors in advanced computer systems”, IEEE Design & Test of Comp., vol. 22, no. 3, pp. 258-266, May/June 2005.

[8] C.Constantinescu, “Trends and challenges in VLSI circuit reliability”, IEEE Micro, vol. 23, pp. 14-19, July 2003.

[9] L.Huang, Q.Xu, “AgeSim: A simulation framework for evaluating the lifetime reliability of processor-based SoCs”, DATE 2010.

[10] Constantinescu, C., Impact of Intermittent Faults on Nanocomputing Devices, in Workshop on Dependable and Secure Nanocomputing. 2007.

[11] Shubhendu S. Mukherjee, Christopher Weaver, Joel Emer,Steven K. Reinhardt, Todd Austin, “A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor”, Microarchitecture, 2003. MICRO-36. Proceedings. 36th Annual IEEE/ACM International Symposium.

[12] V. Sridharan and D. R. Kaeli, “Using pvf traces to accelerate avf modeling,” in Proceedings of the IEEE Workshop on Silicon Errors in Logic-System Effects, 2010.

[13] F.A.Bower, D.Hower, M.Yilmaz, D.Sorin, S.Osev, “Applying architectural vulnerability analysis to hard faults in the microprocessor”, SIGMETRICS 2006.

[14] S.Pan, Y.Hu, X.Li “IVF: Characterizing the vulnerability of microprocessor structures to intermittent faults”, IEEE Transactions on VLSI Systems, vol. 20, no. 5, pp. 777-790, May 2012.

[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] M.Kaliorakis, S.Tselonis, A.Chatzidimitriou, N.Foutris, D.Gizopoulos, “Differential Fault Injection on Microarchitectural Simulators”, IEEE International Symposium on Workload Characterization (IISWC), 2015.

[17] https://github.com/gpgpu-sim/gpgpu-sim_distribution/blob/v4.0.1/src/gpgpu-sim/gpu-cache.h#L696

[18] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, “Rodinia: A benchmark suite for heterogeneous computing,” IISWC ‘09,.

[19] R. Ubal, B. Jang, P. Mistry, D. Schaa, D. Kaeli,"Multi2Sim: A Simulation Framework for CPU-GPU Computing", PACT '12

[20] H. Jeon, M. Wilkening, V. Sridharan, S. Gurumurthi ,G. Loh, "Architectural vulnerability modeling and analysis of integrated Graphics Processors", SELSE ‘13.

[21] J.Tan, N.Goswami, T.Li, X.Fu, "Analyzing soft-error vulnerability of GPGPU microarchitecture", IISWC ‘11.

[22] J. Tan, Z. Li, X. Fu, "Cost-effective soft-error protection for SRAMbased structures in GPGPUs", CF ‘13.

[23] 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.

[24] S. K. S. Hari, T. Tsai, M. Stephenson, S. W. Keckler, and J. Emer, "SASSIFI: Evaluating resilience of GPU applications", SELSE ‘15.

[25] B. Fang, K. Pattabiraman, M. Ripeanu, S. Gurumurthi,“GPU-Qin: A methodology for evaluating the error resilience of GPGPU applications”, ISPASS ‘14.

[26] V. Sridharan and D. R. Kaeli, "Using Hardware Vulnerability Factors to Enhance AVF Analysis," In Proceedings of the 37th annual international symposium on Computer architecture (ISCA '10), SaintMalo, France, pp. 461–472, 2010, doi: 10.1145/1815961.1816023.

[27] 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.

[28] 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.

[29] https://en.wikipedia.org/wiki/CUDA#GPUs_supported