POLITECNICO DI TORINO

Master Degree Course in Mechatronic Engineering

Master Degree Thesis

Development of a Fault Injection Environment for the Evaluation of Hardening Techniques on GPGPU via the NVBitFI Framework

Politecnico di Torino

Supervisors

Prof. Luca Sterpone

Ph.D. Sarah Azimi

Candidate

Stefano Pisciotta

Academic Year 2021/2022

Summary

GPGPUs are becoming more essential than ever before, primarily thanks to their flexibility of use and the exponential growth of parallel algorithms. These GPGPUs are widely used in object recognition across a variety of applications. One significant application of image recognition is in fully self-driving cars, which are currently being developed by a wide range of companies, from automakers such as Tesla to well-known GPU manufacturers such as NVIDIA.

These embedded systems must be safe and meet the ISO26262 standard, which is an international model used for electronic systems installed in cars and other road vehicles for safety purposes. As a result, during the last decade, fault tolerance study has become a critical element of the development process; nevertheless, while reliability analysis of CPUs has been extensively studied, tools for testing GPGPUs has been developed only in the last few years.

To accomplish this, an investigation of GPGPU's general architecture and microarchitecture with the aim of determine the GPU's most critical parts and components is provided, along with an examination of a variety of hardware and software methods for handling radiation effects such as bit-flip. For the purposes of this thesis, the state-of-the-art NVBitFI, one of the most powerful software injection tools developed by NVIDIA, was used.

The NVBitFI framework was utilized in this thesis in conjunction with an NVIDIA Jetson nano board, which is a System-on-Chip which mount a Maxwell's microarchitecture GPU. The board was used to conduct fault injection campaign, running a variety of well-known applications. Furthermore, the tool's output was

thoroughly evaluated. The fault injection framework classifieds the results as *masked*, when no effect on expected behaviour have been detected, Device Under Error (DUE), which indicates that the application was unable to complete the task and crashed, or Silent Data Corruption (SDC), which indicates that the application achieved results that differed from the reference one.

Through a change to the original framework, it was possible to carry out a fault injection campaign to determine the relationship between the instruction code, commonly known as OPCODE and the type of error that this entails, whether it is a *DUE*, *SDC* or *masked*.

Finally, an application was developed with the purpose to improve reliability by utilizing a hardening software technique known as Triple Modular Redundancy (TMR), which improves the device's reliability at the expense of a slight performance degradation. Then, the outcomes of the hardened application have been compared to those of the non-hardened application to determine whether there are any improvements in terms of reliability.

Acknowledgements

Ringraziamenti

Questa tesi, come del resto tutti i risultati raggiunti durante la mia permanenza a Torino, la devo principalmente alla mia famiglia, che mi ha supportato in ogni momento con tutti i mezzi possibili.

A Ivana, fonte di speranza, saggezza e ansie di vita. Inseparabili dalla nascita, abbiamo sempre saputo di poter contare l'uno sull'altro, soprattutto nei momenti difficili (che di questi tempi è la norma). In poche parole, un'amica, una sorella, una certezza.

A Daniele, compagno di battaglia durante il periodo universitario, il mio personale "Let Me Google That For You", ma soprattutto una spalla. Se un amico si vede nel momento del bisogno, quell'amico è Daniele.

A Lucia, grazie alla sua lucidità e ai suoi modi gentili e per nulla irruenti ha saputo riportarmi sulla retta via, quando mi pareva di averla persa di vista. È stata e continua ad essere fonte di ispirazione e di crescita personale.

Casa è dove sei nato, ma anche dove stai o sei stato bene. Quindi grazie anche a Gaspare e Francesco, prima ancora che coinquilini, sono stati per me casa.

Contents

List of Figures

Abbreviations

BFM Bit-flip model.

DUE Device Under Error.

eHPC Embedded High Performance Computing.

FPGA Field Programmable Gate Array.

GPC Graphics Processing Clusters.

GPGPU General Pourpose Graphic Processing Unit.

GPU Graphic Processing Unit.

HPC High Performance Computing.

LFI Laser Fault Injection.

MCU Multiple Cell Upset.

MTTF Mean Time to Failure.

MTTR Mean Time to Repair.

SDC Silent Data Corruption.

SEE Single Event Effects.

SEFI Single Event Functional Interrupt.

SEGR Single Event Gate Rupture.

SEL Single Event Latch-Up.

SEU Single Event Upset.

SM Streaming Multiprocessor.

SoC System-On-Chip.

SP Stream Processor.

TMR Triple Modular Redundancy.

Chapter 1

Introduction

The Graphics Processing Unit (GPU) is a specialized microprocessor that is capable of optimizing and significantly reducing the processing time for any programs that use graphics computing. Today, GPUs are found in a wide variety of mainstream devices, ranging from laptops to mobile phones, and are used in a wide variety of applications and fields.

1.1 GPGPUs

The exponential growth of the GPU in recent years at the expense of the CPU is owing to the GPU's higher parallel computational capacity and memory bandwidth for parallel algorithms, particularly in all disciplines where image processing is critical. These benefits have established the so-called General-Purpose GPUs (GPGPUs) as a viable alternative to the most widely used microprocessor for applications requiring high-performance computing (HPC), particularly embedded systems (eHPC) [5]. Because embedded high-performance computing applications such as image processing for object recognition require a high level of data processing and memory capacity, the GPGPU is increasingly employed and praised.

Nowadays, image processing is an integral part of our lives, from facial recognition, which is important for identification on our smartphone, to self-driving cars,

which are becoming a reality year after year.

1.2 Reliability of GPGPU

Frequently, the GPGPU's concentration is on a particular industry, particularly embedded computing, where *safety* and *dependability* are critical. As a result, we may use a variety of approaches to explore the *dependability* characteristics of GPU applications. However, before delving into the details, we must understand why *safety* is critical in those fields and how errors and faults can arise in a general circuit and what causes them.

The notion of *reliability* is inextricably linked to the concept of *availability*, except that *availability* is defined as the chance that a system or algorithm will be available and functioning correctly at a certain point in time.

Rather than that, *dependability* quantifies the chance of a system functioning successfully at time t. The concepts of *reliability* and *availability* can be somewhat confusing; the primary distinction is that a system that is available at time t may be unreliable, since a system may operate without failure but may produce a result that differs from the expected one due to some faults. To summarize, we may state that a reliable-system is required to be an available-system, but that the inverse is not necessarily true in general [20].

The final critical idea is *safety*; it refers to a system's ability to operate properly or to cease operating entirely without causing substantial harm to anyone or anything. This parameter is increasingly critical in all fields where the technology has the potential to cause harm to people, such as autonomous driving systems.

1.2.1 From fault to failure

Generally, the *fault-error-failure* chain is the main threat to a system's *reliability*. A fault may be caused by a particle colliding with the chip's surface or by an electric current flowing between the transistor contacts. The first case illustrates a temporary defect, while the second illustrates a perpetual fault in the system.

Permanent faults persist in the system until a corrective measure is taken to replace the faulty component. By contrast, transient faults are those that influence the system temporarily and may result in an error or be absorbed by the system. A transient fault cannot directly disturb a system's regular operation, but if it is converted to an error, the error generated may lead to system failure under certain circumstances. When a particle collides with the chip surface, a transient voltage pulse known as a *Single Event Transient (SET)* [2] is created at the struck gate's output. *SET* may be propagated via the logic gate until it reaches and is stored in one of the circuit's latching components. In this case, the error generated is referred to as a *soft error* [16].

Figure 1.1. Fault-Error-Failure chain [35].

A soft error corrupts data without causing a circuit fault,it is created by a physical particle passing across a device's critical volume,that has the potential to split the silicon nucleus into nucleons of varying atomic weights. These nucleons travel in opposite directions, conserving mass, energy, and momentum, and the charged nucleons in motion create a wake of charge separation. This separated charge may recombine or may be caught by an active node in the circuitry [27].

With a better understanding of what is a soft error, we have put together a list of the most typical soft errors that can occur in a System on Chip (SoC), along with a brief description.

- **Single Event Upset (SEU)**: Also known as single event error (SEE), this is an error that occurs when a free ionizing particle strikes a microchip, causing the internal circuit to change state [6]. For example, it can charge a capacitor in a memory DRAM, changing its state from 0 to 1, which is referred to as

bit-flip error. When a particle's impact alters the state of more than one cell, this is referred to as a Multiple Cell Upset (MCU) [28].

- **Single Event Transient (SET)**: It is a brief spike in voltage that lasts only a few nanoseconds, causing a drop in the internal voltage. This is a more severe form of error than SEU since it can result in a permanent error [28].

- **Single Event Functional Interrupt (SEFI)**: This error causes a system interrupt due to a single particle strike, momentarily causing the microchip to misbehave, but does not result in permanent damage [28].

Rather than that, when the faulty device experiences a critical error, we are dealing with a hard error. To be more specific, a hard error is an issue that is caused by malfunctioning hardware, especially data transfer and memories. Hard error are unrecoverable, and the device that has been damaged will need to be replaced [27]. Common Hard error are:

- **Single Event Gate Rupture (SEGR)**. The formation of a conducting path caused by a single ionizing particle colliding with the gate oxide of a MOSFET; this might result in system corruption or complete failure [28].

- **Single Event Latch-up (SEL)**. When a single electrically charged particle passes across sensitive areas of the device structure, the system enters an abnormally high current state, causing the device to lose its features and operations [28].

All of these errors, whether they are hard or soft, can have a fatal impact on the system, and in order to increase the system's fault tolerance, a variety of hardening techniques have been developed. The following section will cover a specific form of hardening technique that allows the system's dependability to be enhanced, called fault injection.

1.3 Testing Techniques: Fault Injection

When designing electronic systems that are installed in safety-critical industries such as avionics, aircraft, defence, and transports, reliability and availability are primary concerns [15] These systems are increasingly sensitive to external or internal failures due to the progressive downsizing of microelectronic components. Consequently, several studies are focusing on enhancing system reliability by developing effective procedures and techniques. Fault tolerance is one of the approaches for improving a system's reliability. The aim of fault tolerant computation is to design systems that execute successfully in the presence of faults while maintaining their functionalities. Fault injection is a technique to estimate the dependability and the safety of a system under faults. It is classified as a fault tolerance system dependability certification approach which focuses on the conduction of repeatable experiments to investigate the behaviour of computer systems in the presence of faults. This procedure can accelerate the emergence and spread of faults in a device so that their consequences on the system's performance can be detected. It's essential to establish the fault injection policy, including fault location, injection time, fault duration, and the system's input data, while setting up the fault injection environment. Simulation-based, Emulation-based, and hardware-based approaches are the three categories of fault-injection techniques [26].

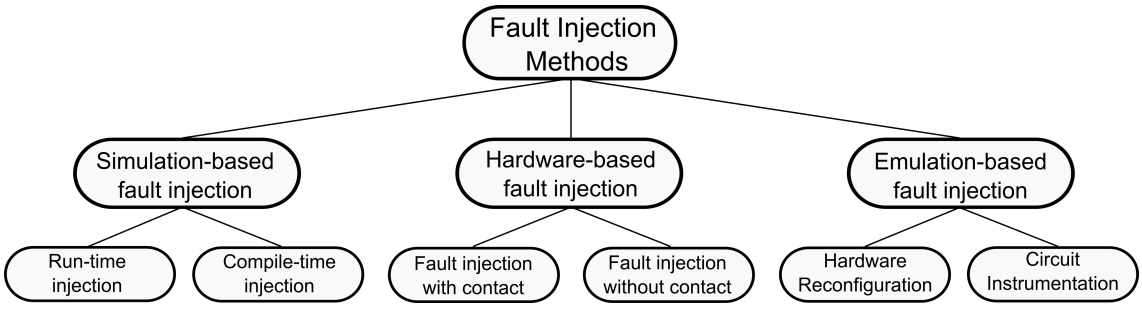


Figure 1.2. Fault Injection overview.

1.3.1 Simulation-based fault Injection

In simulation-based Fault Injection, commonly known as a fault simulator, is a software application that models and simulates the target system as well as any potential hardware faults [37]. The fault simulation is carried out by modifying the hardware model or software state of the target system, making the system to behave as if it were experiencing a hardware failure. There are two types of fault injection: runtime and compile-time [26].

- ***Run-time injection***

  In order to activate fault injection during runtime, a mechanism must be in setup. The following are examples of commonly used triggering mechanisms:

  - *Time-out.* In this method, the injection is triggered when a previously established timer expires, in particular, the timer generates a system interrupt. The result is that there is no way to forecast what will happen during the injection, as the injection is time-dependent rather than provoked by a specific event [23].
  - *Exception/trap.* Unlike time-out, the fault is injected in this situation when an event or specified condition happens. In a software trap, for example, the fault injection can occur prior to the execution of a predefined instruction. To accomplish this, an interrupt is triggered prior to the execution of that instruction, triggering the injection [23].
  - *Code insertion.* This method is quite similar to the Code modification techniques, except that the original code is not modified, but rather additional instructions useful for the injection, are inserted to it [23].

- ***Compile-time injection***

  This strategy distinguished itself from the preceding technique on the target. Compile-time injection simulates the effect of transitory and even permanent faults by modifying the original assembly code or source code. It is necessary

to have additional software that can analyse the result of the injection in this solution, but no further software is required during the application's execution.

The benefit of simulation-based fault injection techniques is that there is no hazard of causing damage to the operational system. Additionally, they are more time and effort efficient than hardware approaches. Furthermore, they improve the system's controllability and observability in the event of faults. However, simulation-based fault injection techniques may suffer from inaccuracies in the fault and system models [26].

1.3.2 Hardware-based fault injection

Hardware Fault Injection is one of the most precise approaches for determining the dependability of a system. It makes use of external physical sources to introduce faults into the system's hardware via several methods classified into two categories: tools that inject faults via contact and systems that do not.

- ***Fault Injection with contact***

  The injector is in direct physical contact with the target device, causing voltage or current changes to the target chip from the outside. Pin-level injection, or hardware fault injection involving direct contact with circuit pins, was the most frequent technique of hardware fault injection. Altering electrical currents and voltages at the pins level can be accomplished in two ways:

  – *Active probes.* This approach alters the electrical currents of the pins by adding current through probes linked to them. Although bridging faults may be recognized by inserting a probe across two or more pins, the probe method is typically limited to stuck-at faults. This approach is extremely delicate, as the additional current flowing through the device has the potential to damage it [23].

  – *Socket insertion.* A socket is inserted between the target hardware and the

circuit board. The inserted socket injects stuck-at, open, or more complicated logic errors into the target hardware by pushing the analogue-signals, that indicate desired logic values, into the pins of the target hardware [23].

- ***Fault Injection without contact***

  There is no direct physical contact between the injector and the target device. Instead, an external source causes spurious currents inside the target semiconductor by introducing physical phenomena such as heavy-ion radiation and electromagnetic interference. We may classify these technologies into two major categories: radio-based fault injection and laser-based fault injection.

  – *Radiation-based.* To begin, we must fully understand what radiation is. Radiation is a set of energetic particles that interact and exchange energy with other particles when they collide and interact [13]. The particle composition of radiation can vary based on its nature, and there are three primary types: cosmic rays, mesons, and alpha particles. Mesons are created when cosmic rays collide with the terrestrial atmosphere; as a result, they have a lesser energy but can still cause negative repercussions on irradiated semiconductor circuits. To address this problem, engineers developed precise vacuum chambers in which they tested electronic devices with heavy-ion particle beams to determine how they would be affected and how they might be hardened [4]. This technique is useful for stimulating the emergence of SEU like bit-flip or even changing the states of adjacent bits, causing MEU [25].

    This approaches is popular among engineers because it closely resemble natural physical processes. However, because you cannot accurately control the exact moment of heavy-ion emission or electromagnetic field production, it is difficult to precisely trigger the time and position of a fault injection using this mechanism.

  – *Laser-based.* The laser injection approach yields carriers in the silicon

substrate, which then cluster in the diffusion area of the targeted circuit. Laser energy absorption produces electron-hole pairs in a similar manner to SEU (Single Event Upset) effects characterized by high energy particles. For example, a transitory change in the input or output of one of the transistors in a flip-flop can cause a change in state.

The Laser Fault Injection (LFI) technique not only enables the validation of fault-tolerant designs by providing a precisely controlled, nonintrusive, and non-destructive method of injecting faults in semiconductor circuits, but it also has the potential to automate the testing of board-level and system-level fault tolerance designs, including fault-tolerant operating systems and application software. In addition, a vacuum test chamber, a radioactive source, or further on-chip fault injection devices are not required [36].

Hardware fault injection techniques have the benefit of being able to access regions that are difficult to reach with other methods. They're also good for hardware triggering and monitoring that require a lot of time resolution. The downsides of such strategies include that the injection strategy necessitates specific hardware and requires access to the target system's hardware, which might be difficult or expensive to obtain. Furthermore, these procedures have a considerable potential of causing serious damage under examination. The outcomes of hardware fault injection techniques are difficult to observe and record, reducing the method's usefulness.

1.3.3 Emulation-based fault-injection

As an alternative to the methods discussed in previous sections, fault-emulation approaches attempt to combine the speed of hardware-based techniques with the precision of software-based approaches. As a result, their implementation phases typically include both software and hardware components. Emulation-based alternatives to fault injection do not require the specialized equipment required for

physical fault injection, making them more cost effective. Another advantage of the emulation-based fault injection technique is that the position of the faults is not constrained [16]. The time it takes to perform software simulations of large-scale circuits can take years, making the simulation process unfeasible. In addition, the difficulties and high prices of physical techniques make them unaffordable for most researchers. To address these issues, one of the most often used solutions is the emulation-based fault injection strategy, which involves simulating circuit behaviour using FPGAs in the presence of faults. As a result, the FPGA replicates the intended behaviour of the circuit under certain situations, allowing the designer to work at a breakneck speed throughout the fault injection process [12]. Based on how they are implemented, emulation-based fault injection techniques are categorized into the following categories:

- ***Hardware reconfiguration-based approaches***

  In these approaches, the design is specified in the first step using an FPGA-specific Hardware Description Languages (HDL). A bit-file is generated from the design, which is subsequently downloaded onto the FPGA. The second step is to choose a benchmark to run on the design to evaluate its performance. Using the chosen benchmark, a set of input bit-strings is applied to the circuit's inputs. These bit-strings contain information about how the design is set up on the FPGA, and a set of initial inputs for the circuit. After the circuit has processed all of the input bit-strings, the values will be sent back to the computer system for proper inspection.

  The tangible advantage of employing hardware reconfiguration for fault simulation is that the faulty bit-string is applied to only a portion of the circuit, rather than the entire circuit. Among the emulation-based options, the hardware reconfiguration approach is the quickest. However, it should be emphasized that the connection structure between the host computer and the FPGA has a significant impact on the performance of fault simulation and is sometimes referred to as the bottleneck in these methods [16].

- ***Instrumentation based approaches***

  In this technique, an additional component, sometimes referred to as a *saboteur*, is added to the circuit for every fault position. This component is capable of altering the value of the specified signal in accordance with the fault model selected by the designer. The sabotage component may be positioned in either the combinational or sequential portion of the circuit. Thus, because the bit-file is only transferred once on the FPGA, this technique is significantly quicker than hardware reconfiguration technique. However, because the area overhead increases significantly with this method, it cannot be used on exceedingly large circuits. Each *saboteur* component has an enable signal that is activated during the prescribed fault injection periods. The enabling signal modulates the value of the signal that flows through the sabotage component. This change can perform test like bit-flip or even Stuck-at faults [16].

Determining the correct approach for fault injection in electronic systems may make a significant difference in terms of lowering project costs, shortening the time to market for a product, and improve the reliability and dependability of a digital device.

Chapter 2

State of the art

Now that we have a clear understanding of what Dependability is and why it is so important for the GPU, as well as what causes an error and how to stimulate it via fault injection, it's important to have a general understanding of the GPU's architecture and microarchitecture, and then consider the specific structure of the GPU used for the purpose of this thesis.

2.1 GPU microarchitecture

As described in the last chapter, a GPU in general can perform a large number of processes in parallel. A GPU's computing capabilities, rather like a CPU, is measured in terms of the number of floating-point operations per second (FLOPS) that it can execute. GPUs can handle operations on the order of GFLOPS, which is orders of magnitude faster than CPUs. For the purposes of this thesis, we shall solely evaluate GPUs manufactured by the business NVIDIA. Nvidia has named his microarchitecture after notable scientists throughout the years, beginning with Fahrenheit in 1998 and ending with Ampere in 2020. For the sake of this thesis, we will focus on the Maxwell microarchitecture, which is embedded in the NVIDIA Jetson nano GPGPU employed in this study.

A GPU's general structure is made up of several small microprocessors known as

CUDA cores, which stand for Compute Unified Device Architecture. These cores are limited to specialized activities and cannot calculate the same instructions as a regular CPU. The CUDA cores are distributed in a block, as shown in the figure 2.1 below, called Streaming Multiprocessor (SM). In addition to CUDA cores, each SM contains an L1 cache that is accessible from every CUDA core inside the same SM. A GPU typically contains more than one SM, and the related L1 cache is not visible to the CUDA cores of the other SMs in the same GPU, and an additional L2 cache is included to permit sharing between distinct SMs. Prior to delving into the mechanics of memory management, it is crucial to advance a general understanding of the GPU's architecture. We will discuss details in greater depth later.

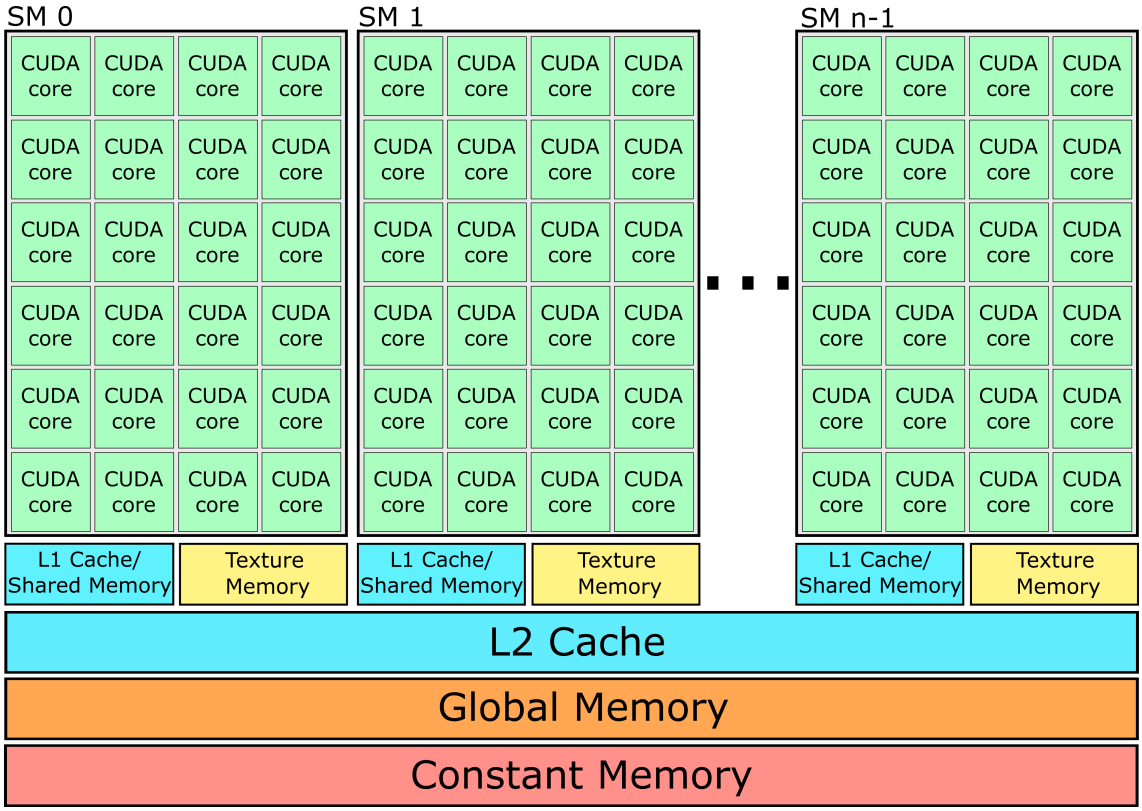


Figure 2.1. General illustration of a GPU's microarchitecture [39].

2.2 GPU architecture

Now that we've figured out how the GPU's microarchitecture operates, we need to figure out how parallel computation is managed and which logic structures are used:

- ***Kernel.*** The Kernel is a dedicated function that is executed in parallel on the GPU. Each kernel instance is referred to as a *Thread* [21].
- ***Thread.*** A thread is a programming abstraction that represents the kernel's execution. Threads are clustered into Thread Blocks to manage data mapping. Each thread has a unique identifier, which could be determined using special specified variables, and it is fundamental to index the thread within a *thread block* [21].
- ***Block.*** A thread block is a form of abstraction in programming that represents a cluster of threads. Historically, the architecture restricted the amount of threads in a thread block to a maximum of 1024. Threads within the same thread block use the same stream processor (SP). Stream Processors (SPs) are a subgroup of Stream Multiprocessors (SMs). While an SM may handle multiple thread Blocks concurrently, an SM can process only multiple thread simultaneously which are in the same block. Threads contained within the same block can communicate via shared memory. As for a thread, each block has a unique identifier that can be retrieved by a particular predefined variable, and it is fundamental to index within a *Grid* [21].
- ***Grid.*** A grid is obtained by combining different thread-blocks. While the number of threads contained within a block is restricted to 1024, grids are not constrained by the number of thread-blocks they can include. It is critical to note, however, that all blocks within the same grid must have the same dimension [21].

The GPU has a variety of memory types, each with a distinct purpose and level of accessibility. We have the following memories in order of accessibility:

- *Registers.* These are thread-private, which means that thread-specific registers are not visible to other threads.

- *L1/Shared memory.* Each SM incorporates a fast memory. It is suitable of being utilized as an L1 cache or a shared memory. All threads contained within the same block can share shared memory, and all blocks included within the same SM can use the cache L1, but cannot share shared memory with other thread-blocks.

- *Read-only memory.* Each SM is equipped with an instruction cache, constant memory, and texture memory. Constant memory is used to store variables that cannot be compiled into the program. Before invoking the kernel function, the host must set these constants. We will overlook texture memory because it is irrelevant to the aim of this thesis [8].

- *L2 cache.* The L2 cache is shared across all SMs, which means that any thread in any CUDA block can access it.

- *Global memory.* Global memory accounts for the vast majority of the GPU's memory. On Fermi architecture, for example, global memory might have a 150x slower latency of 600 ns than registers or shared memory. This reduces GPU performance overall and makes global memory access a significant bottleneck for applications [8].

It's important to note that GPU design is tightly connected to microarchitecture. All the threads are operated on a single CUDA core, the thread-blocks are processed by a single SM, and all the grids are executing on the entire GPU.

To help visualize this hierarchy, figure 2.2 illustrates how abstraction programming is closely related to the GPU hardware and, additionally, the type of memory accessibility employed.

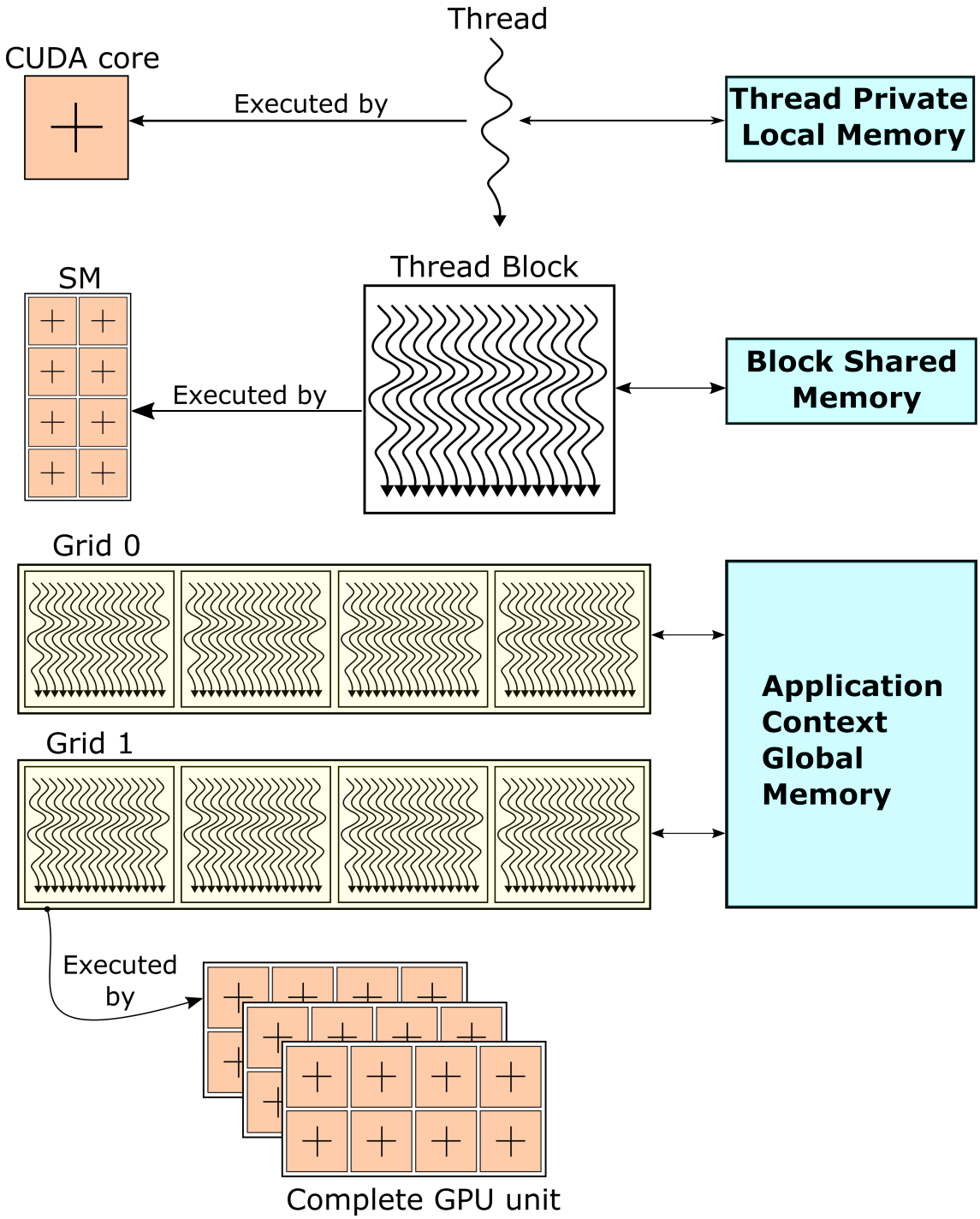


Figure 2.2. Thread Block Grid hardware perspective.

2.3 NVIDIA Jetson nano

We employed the NVIDIA Jetson nano for this thesis, which was originally introduced by NVIDIA in 2019 for development purposes, most specifically AI development. NVIDIA released different Jetson nano models over the years. We utilized the NVIDIA Jetson nano B01. NVIDIA's updates to the various models over the years have been concentrated on the peripherals.

Following that, Figure 2.3 depicts the Jetson nano B01 board. As seen in this image, the board was powered via the *[J25] power jack* and linked to the host PC via the micro USB interface during the experiments. Later, we shall discuss the major steps taken to conduct the studies.

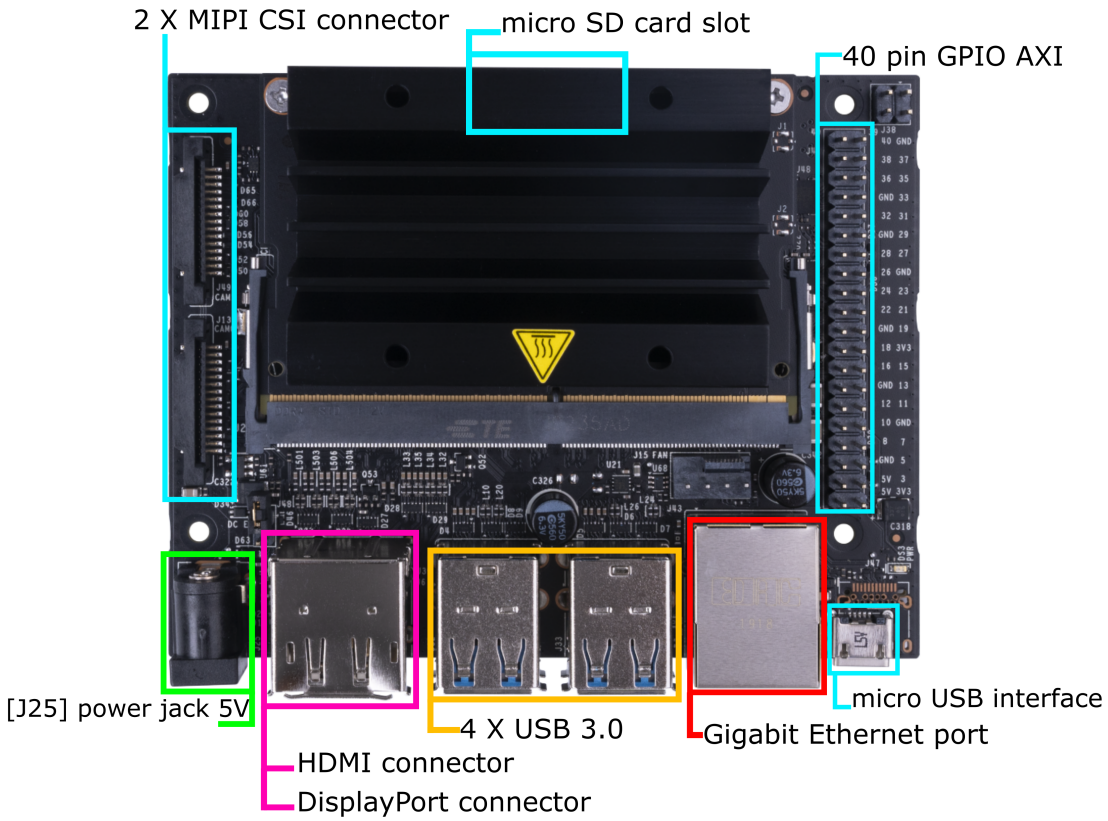


Figure 2.3. The specific structure of NVIDIA Jetson nano B01 [34].

The Jetson nano, internally, is composed of a variety of components. The core is the Tegra X1 series SoC, which is produced by NVIDIA and composed of a Maxwell GPU with a 128-core and a CPU Cortex A57. For internal storage, the board supports 4GB of LPDDR4 memory and a 16GB eMMC 5.1.

In the following, we will concentrate on the Maxwell microarchitecture used by the GPU and neglect the Cortex A57 microarchitecture used by the CPU, as it is superfluous for the experiments considered during these studies.

2.3.1 Maxwell GPU microarchitecture

This architecture, released by NVIDIA in 2014, offered a completely new design for the SM, reconfigured all unit and crossbar structures, streamlined data flows, and dramatically improved power management. The SM scheduler design and algorithms were modified to be more intelligent and minimize needless delays, thus decreasing the energy required for scheduling. It divides the CUDA core into processing-blocks. Each block contains 32 CUDA cores. Furthermore, the blocks are organized into Streaming Multiprocessors (SMMs), for this specific microarchitecture. Finally, SMMs are grouped as Graphics Processing Clusters (GPC) [33]. Figure 2.4 shows via blocks the architecture of a single SMM of Maxwell's microarchitecture with 128 cores, like the Jetson nano's one. As can be seen, the SMM has different warp schedulers. It is useful to divide the threads into warps. A warp is a programming abstraction that, per definition, is a group of 32 threads that work in parallel. The warp scheduler determines which warp should be processed next on the CUDA cores in a specific SM. Every block has their own registers. For every block, there are 16.384 registers of 32 bits each, for a total of 65.536. The GPU shares the same physical RAM as the CPU. The Jetson nano has 128 cores but has only one SM.

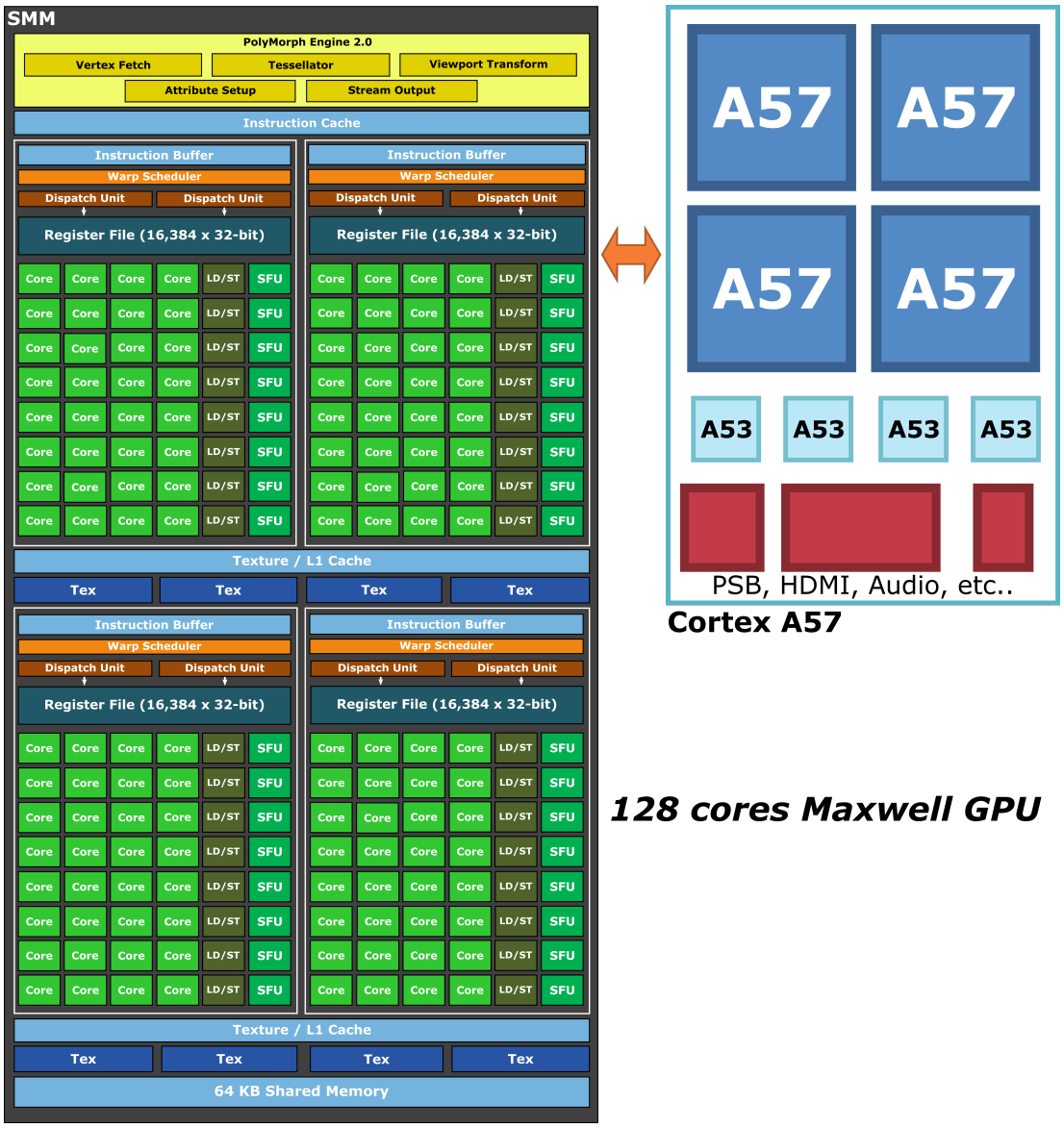


Figure 2.4. Illustration of Tegra X1 highlighting GPU's microarchitecture [10] [9].

Having established the architecture and microarchitecture of the NVIDIA Jetson nano used for the fault injection experiments, we must inquire what types of fault injection approach are commonly used for GPU testing. We will concentrate on software-based fault injection in particular, but will also discuss various techniques such as radiation testing, laser-based fault injection, and others discussed in the first chapter.

2.4 Fault injection techniques for GPGPUs

As mentioned in the previous chapter, fault injection is critical for GPGPU because to the intensive use of modern applications. This, as we all know, results in attempts to increase the reliability of the graphics processor. Additionally, GPU have a higher probability of being faulty due to the massive parallel computations performed by the GPU per second. The parallelism functionality makes it much more difficult to inject faults and understand what the error chain is, what causes a failure, and why it occurs. As a result, researchers develop various techniques to introduce faults and strive to improve GPU reliability. As with any other sort of fault injection, various types of fault injection-based tests are possible. These include hardware-based methods such as radiation, simulation-based methods such as code-instrumentation, and finally, emulation-based methods. We'll examine how each approach works and the benefits and drawbacks of each fault injection technique.

2.4.1 Emulation-based fault injection: FlexGripPlus

As is well known, an emulation-based fault-based injection tool is one that makes use of a model of the target hardware. This model is typically implemented on a reconfigurable board, such as an FPGA.

The FlexGripPlus is a model based on the FlexGrip tool that was previously developed. Instrumentation of this model is performed in order to induce faults into the desired module. Then, using this module, the effect of a fault occurring during the execution of a specific program may be determined. The outputs can be seen and examined to determine the dependability of a GPGPU or to detect structural or application faults. Additionally, these evaluations are used to determine the most appropriate mitigation technique for a given application at various levels [11].

The FlexGrip is an open-source, VHDL-based soft-GPGPU system that relies on the NVIDIA G80 microarchitecture. The University of Massachusetts developed this GPGPU model, which it was developed to be totally consistent with the CUDA development platform. There are 27 assembly (SASS) instructions that can be

used with the FlexGrip model. It is made up of the characteristic of a Streaming Multiprocessor (SM) module [1].

The main difference between this model and the newer one is that it lacks the cache memory used in commercial devices. Additionally, it misses floating-point modules and specialized accelerators. These FlexGrip structural restrictions may have an effect on the implementation of more modern applications.

FlexGripPlus made several enhancements to the model, including architectural and functional adjustments that increased flexibility while keeping the structure of the initial model. Additionally, to be fully compatible with a larger number of SASS instructions, some model changes were required, such as the addition of some registers and connections.

The model must be imported into a simulation environment, such as ModelSim, before the fault injection campaign can proceed. The fault injection platform was written in Python and consists of three major modules:

- Fault Controller.
- Fault Injector.
- Fault Checker and Classifier.

After the fault injection campaign was completed, the tool categorised the faults as follows:

- *Silent Data Corruption (SDC).* It occurs when an injected defect impacts only the final outcome in memory.
- *Detected Unrecoverable Error (DUE).* It occurs when the application for some reason end-up with an unexpected crash or hang.
- *Timeout.* This occurs when the SEU results in a performance loss during the simulation time.
- *Masked.* Basically, the injected errors don't affect the outcome of the application.

After a lot of testing in benchmark applications developed with the aim of being used with these tools, we can summarize that with FlexGripPlus it is possible to do fault-injection campaigns targeting different modules and components in the GPGPU model. It also has the benefits of technological independence, which enables the model to be used at a lower level without regard to the intended gate library or simulation tool.

Additionally, the model's compatibility with commercial development tools enables the model to be developed using the same tools as real-world applications, with some limits. Thus, FlexGripPlus is capable of performing reliability tests on components that are comparable across generations of GPGPU systems.

Although FlexGripPlus, like other emulation tools, does not fully emulate the architecture of the most modern GPGPUs, this is the primary reason why it is critical to have the option of analysing dependability directly on the real hardware.

2.4.2 Hardware-based fault injection: Neutron Irradiation

As we previously established, hardware fault injection, like radiation tests, is popular among engineers for its reliable results. Although CPU irradiation tests have been thoroughly investigated in the past, GPUs have recently been considered for testing because of their significant application in a multitude of sectors. The primary challenge with executing fault injection with a detailed outcome is that commercial GPGPUs frequently lack detailed microarchitecture characterization, unless it is absolutely important for developers, such as the number of registers and cores and also memory size [24]. As a result, we will examine a 2016 study that was able to investigate the effect of neutron irradiation on the CUDA cores by utilizing two different NVIDIA GPU architectures. As previously stated, neutron strikes can cause GPUs to malfunction in a multitude of ways. A neutron can cause memory element bit flips as well as transitory over-voltage in logic processing resources or control circuits. From the perspective of radiation testing, the CUDA cores are compartmentalized such that a single radiation-induced occurrence in one of them will corrupt just the thread assigned to it. Threads that are allocated to CUDA

cores adjacent to the faulty one or those that follow it will be unaffected. Errors in the L1 cache or shared memory, on the other hand, are likely to affect multiple threads in the SM. Similarly, errors in the L2 cache, which is shared by all SMs, are likely to damage several thread blocks. The Fermi-based C2050 and Kepler-based K20 were used in this research, both of which contain a unified L2 cache and a GDDR5 main memory module. The C2050 has 14 SMs, each of which contains 32 CUDA cores, meanwhile the K20 has 13 SMs, each of which contains 192 CUDA cores. The experiments were done in two distinct facilities, each of which provided a white neutron source with an energy spectrum between 10 and 750 MeV. The neutron flux with energies greater than 10 MeV was around $1x10^6(\frac{n}{cm^2s})$ in the first facility and $4x10^4(\frac{n}{cm^2s})$ in the other. Numerous trials were conducted with various Benchmarks software, demonstrating that the measured errors were less than $10^{-2}(\frac{errors}{execution})$. Due to the fact that a GPU receives a substantially lower neutron flux in a realistic environment, it is very implausible that more than one corruption occurs during a single execution. The studies on the two GPUs were conducted with the beam focused on a 2-inch-diameter area, which provided uniform irradiation of the CUDA cores without disrupting neighbouring board power control circuitry or DRAM chips. This ensures that data in the RAM is not altered, allowing for study of the GPU core alone. The GPUs were linked to the motherboard with a PCIe extension, which is a separate power line that allows you to ward off the GPU from the motherboard itself. The host computer's function is to start the test by passing input data pre-determined to the GPU and collecting the calculation results. When the results are ready, the host makes a comparison between the golden output and the results computed in that experiment. When a mismatch is identified, the execution is indicated as having been disrupted by an SDC. All the tests were performed using the CUDA programming language and executed on the DUT directly. The tests were carried out by measuring the cross section of the register file cells along with the L1 and L2 cache cells of both the GPUs, which is a widely used metric to determine radiation sensitivity [7] for the register file cells and the L1 and L2 cache cells of both K20 and C2050 boards. The cross section is calculated by dividing

the number of observed errors by the received neutron fluence ($\frac{neutron}{cm^2}$).The average sensitive area of a single cell is calculated by dividing the cross section of the memory structure by the number of cells. This is the portion of the cell that will fail if struck by a neutron. The higher the cross section, the higher the probability that a neutron will corrupt a bit [19]. As demonstrated in this study, hardware fault injection using a neutron beam source was utilized to determine the reliability of the GPU core without affecting other GPU components via irradiation, which might invalidate the results, and also provide the failure's probability for register files, L1, and L2 cache memories. Additionally, the experiments conducted in this study used a matrix multiplication application with different input size matrices, as did the experiments conducted in this thesis, although we shall discuss it in greater detail in the next chapters [24].

2.4.3 Simulation based fault injection Techniques

Researchers have devoted significant effort to simulation-based fault injection techniques for their flexibility and relative cost savings over hardware-based techniques, without sacrificing the precision of the results that may be achieved. As previously said, we shall concentrate on software fault injection techniques because they are inextricably linked to the works of this thesis. Researchers develop several tools that enable the injection of faults into applications that are executed on specific hardware. Using a variety of approaches, we will examine the primary ones, which was developed directly by NVIDIA, as well as tools that did not. We shall analyse the following tools in chronological order:

- **GPU-Qin**

  GPU-Qin is a fault injection tool capable of injecting faults in real hardware. It is one of the first instruction-level fault injection tools able do perform fault injection in a real GPU microarchitecture, such as Fermi. The tool injects faults at the assembly level. The tool focuses on transient faults affecting a variety of components, including the ALU and the load-store unit (LSU). The

researchers consider only the bit-lip model in their tests, although the tool supports multiple bit-flip model too. As with previous fault injection technologies, GPU-Qin must meet three distinct criteria. Representativeness, the tool must efficiently reflect the actual hardware faults that can occur, it must run in a fair period of time, and it must cause as little run-time interference as possible with the GPU's normal operation. Their methodology is built on a tool called *cuda-gdb*, which enables them to control GPU-accelerated applications. The control flow of the tool can be separated into four distinct phases. The first one organizes threads according to their similarity, then randomly selects one and profiles it, and last, randomly selects one instruction to instrument from the previously selected thread.

At the conclusion of the process, the generated results are collected and then analysed. This tool was used to conduct a range of test benchmarks, allowing the researcher to explore the end-to-end error resilience of GPGPU [17].

- **SASSIFI**

  One of the most well-known fault injection tools is called SASSIFI, and it was made by NVIDIA. SASSIFI works at low level, using an assembly instrumentation tool called SASSI to profile the applications and then injecting errors with the help of the SASSI Fault Injector tool. The tool profiles the application first, then selects the error injection sites, and finally injects the errors into the application while monitoring the results. SASSIFI identifies the information needed to determine when and what error to inject for each injection. It injects one error each application run and keeps track of any hangs, crashes, or output corruption. The host PC is used for the last phase. SASSIFI may then introduce bit-flips into the register file and the instruction output to see how soft faults displayed at the architectural level propagate to the output. The user must indicate where to inject the fault (for example, destination register) and what error model to employ for each injection (bit flip model, as an example). The tool isn't restricted to a single microarchitecture and may be

applied to a wide range of GPUs [22].

The key difference between SASSIFI and GPU-Qin is that the former cannot capture the exact effect of lower-level faults, whilst the latter is substantially faster.

In this chapter, we discussed the microarchitecture and architecture of a GPU, with a particular emphasis on the microarchitecture of the NVIDIA Jetson nano, named Maxwell. Following that, we discussed the history of fault injection techniques for GPGPUs, focusing on simulation-based fault injection. The following chapter will continue our examination of simulation-based fault injection tools by examining the most recent injection tool developed by NVIDIA in 2021. We'll look at how the tool was implemented in the Jetson nano and how NVIDIA's fault injection tool, NVBitFI, was used to operate on the Jetson nano.

Chapter 3

Background

3.1 NVBitFI: Fault Injection framework for GPUs

In this chapter, we will concentrate on the state-of-the-art NVBitFI fault injector. We'll see how it works, and then we'll analyse how it's implemented in the NVIDIA Jetson nano. In comparison to GPU-Qin and SASSIFI, NVBitFI (short for NVIDIA Binary Instruction Tool Fault Injector) can execute dynamic code instrumentation without access to source code. This is possible since the tool instruments the SASS code directly (binary instrumentation of assembly code).

The NVBit framework is capable of intercepting and instrumenting any dynamic CUDA kernel call via just-in-time compilation. If the CUDA call must be instrumented, a specific function is called to modify the instruction code in a specific thread, and then the function is executed. Thus, instrumentation incurs minimal time overhead and enables the production of outputs in real time, consistent with the regular execution of the instrumented application. Since the tool is fully compatible with all types of architecture, it may be used with a broad variety of various microarchitectures, such as the Maxwell architecture used on the Jetson nano. The fault injector tool is based on the dynamic binary instrumentation framework NVBit, which was developed for NVIDIA GPUs and provides APIs for inspecting and injecting arbitrary functions into any application prior to kernel start.

The tool allows for both transient and permanent fault injection to be executed without impacting time performance. The tool is divided into two components: a profiler and an injector. Both components are implemented as dynamic libraries which are attached to the target program. Because each dynamic instruction must be instrumented, the profiling procedure can take a long time. To overcome this issue, the profiler can work in two ways: by profiling every dynamic instruction of every kernel call of the application (exact profiling); or by profiling the instruction only in the first instance of every kernel, assuming that every thread of that specific kernel has the same instruction count (approximate profiling). Depending on the application, the difference in terms of time overhead can be significant. As a result, the researcher recommends that precise profiling be used only when strictly necessary.

To inject a transient fault, the tool produces a program profile in order to provide an eligible list of injection sites. Second, choose one or more injection points. Third, inject the errors during the application's runtime by changing the program's binary, and last, compare the output of the instrumented application to the output of the non-instrumented application after the program's execution concludes (also known as golden output). During the injection phase of a transient fault, the tool injects the fault into a single dynamic instruction. Instead, the instruction is corrupted for all threads in the target kernel for permanent fault injection (only one instruction per thread). There are numerous models for transient faults. The tool may inject one-bit or double-bit bit flip faults into a single destination register in a single kernel, or it can execute a zeroing effect.

As mentioned previously, the program compares the output of the corrupted application to the expected output during normal operation. NVBitFI assigns a range of outcomes to the fault injection experiment based on the difference between the two outputs. We may discuss this in further detail later. Prior to that, we'll discuss how the tool was implemented on the Jetson nano and the standard flow procedures that were required for each experiment. We will pay special attention to the changes made to the original version on GitHub of NVBitFI in order to

simplify the flow of the many experiments and make them more suitable with the experiments themselves [38].

3.2 Environment Configuration

3.2.1 Configuration of the Jetson nano Development Kit

To install the NVBitFI framework, the Jetson nano ecosystem must first be configured according to NVIDIA's installation guide [18]. According to the documentation, there are two ways to power the board: directly via the micro-USB (which is used for both power and communication with the host computer), or via the J25 connector powered by a 5V2A power supply.

After configuration was complete, the board was linked via the micro-USB connector to the host PC. The host PC communicated with the board via an SSH connection. To make the process easier and more accessible, we used the software *MobaXterm*, which enables us to connect to the board through SSH using a Linux terminal, displaying all the directories and files necessary for the transfer on both sides. The CUDA toolkit 10.2 was already installed during the initial installation. This allowed us to use all of the CUDA libraries we needed to use the framework for fault injection, and it also gave us a wide range of typical applications for the experiments that we performed. Finally, it is necessary to activate the root user since we will utilize the already installed `nvprof` software to obtain extensive information on the time performance of all sorts of CUDA applications. We will now exclusively use the root user.

3.2.2 Installation of NVBit and NVBitFI

With the environment prepared, we can now install the NVBit framework by following the instructions on GitHub [30]. Although we can install the software anywhere, I prefer to do so in the root directory.

Following the completion of the installation, we can install the NVBitFI utility by following the installation tutorial available on GitHub. It is essential that this application is installed in the same directory as the NVbit Framework. The framework is compatible with both Linux x86-based PCs and ARM-based devices. Because the Jetson nano is an ARM-based board, we shall follow the previous instruction [32].

3.2.3 Standard flow operations of NVBitFI

As indicated in the preceding command, it performs a fault injection test using the framework's included test application to ensure that everything works properly.

Now we'll look at the bare minimum of usual flow operations required to use the tool. Following that, we'll examine the components that can be tweaked to increase the framework's flexibility when used with Python scripts.

To use the tool, we must follow the following procedure:

1. After the installation is complete, assuming it was performed in the root directory, we have the path `/root/nvbit_release/tools/nvbitfi`. From now on, when we say the NVBitFI folder, we will refer to the folder that can be found on this path. If we want to perform a fault injection in a specific application, we must copy the project folder with all the necessary files in the `nvbitfi/test-apps` directory. For example, if you have the folder `your-project`, you must copy it in the above directory in such a way that the project is in this directory: `nvbitfi/test-apps/your-project`.

2. We must alter the file `test.sh` after transferring the project into the `test-apps` folder. This bash program is designed to automate certain Linux tasks necessary for doing fault injection in the test application. As a result, all major lines of code can remain unchanged. We need to alter only line 73 and, if necessary, line 49 of this bash file as follows:

   - 73) `cd test-apps/simple_add/` → `cd test-apps/your-project/`

- 49) `TOOL_VERBOSE=1` If set to 1, this will allow debugging mode and print all the information during the fault injection.

3. The application needs the golden output of the application to make the comparison with the output of the corrupted application. To collect the data, it is needed to add the following lines of code to the `Makefile` of *your-project*:

   - `golden:`

     `./your_project >golden_stdout.txt 2>golden_stderr.txt`

   This enables the collection of the golden output in two distinct files, one holding the application's normal output and the other containing the application's standard error (often empty).

4. The framework requires two more `*.sh` files: one called `run.sh`, which is required to attach either the profiler or the injector library, and another called `sdc_check.sh`, which allows for the creation of a report at the end of the fault injection by comparing the golden output with the corrupted output. To accomplish this, navigate to the folder `nvbitfi/test-apps/simple_add/` and copy the files `run.sh` and `sdc_check.sh` before pasting them into the folder `your_project`.

5. Now that we have the files mentioned above in our project directory, we must alter the file `run.sh` by replacing `simple_add` with `your-project` as seen below:

   - `eval ${PRELOAD_FLAG} ${BIN_DIR}/simple_add>stdout.txt 2>stderr.txt`
   - `eval ${PRELOAD_FLAG} ${BIN_DIR}/your_project>stdout.txt 2>stderr.txt`

6. We can locate a folder called `scripts` within the `nvbitfi` folder. This folder contains a collection of Python programs for fault injection. We must adjust the script `params.py` to change the application's name and the path to the workload; specifically, we must modify the code below, replacing `simple_add` with the name `your-project`. The following code is located on line 203:

```
apps = {
  'simple_add': [
  NVBITFI_HOME + '/test-apps/simple_add', #workload directory
  'simple_add', #binary name
  NVBITFI_HOME + '/test-apps/simple_add/', #path to the binary
   file
  1, #expected runtime
  "" #additional parameters to the run.sh
  ],
}
```

7. After analysing the application, the framework generates a preset number of injection points. This value can be specified in the script `params.py`. The application then generates an injection point list by selecting a given number of fault injection sites from the previously generated list. This value can be modified by altering the `THRESHOLD JOBS` variable in the file `params.py` line 73. We conducted 10,000 fault injections for each application in order to obtain accurate statistics data. Bear in mind that each fault injection instruments only one instruction code in a single thread.

8. As previously indicated, we discussed approximate and precise profiling. This parameter can be modified by altering the corresponding flag in the Makefile located in the profiler folder `nvbitfi\profiler\Makefile`. Uncomment the following flag (line 24) to enable quick approximation profiling:

   - `FAST_APPROXIMATE_PROFILE = SKIP_PROFILED_KERNELS`

9. After modifying all required files, we can execute the set fault injection campaign by running the bash file `test.sh` located in the `nvbitfi` directory. Once the fault injection campaign is complete (which may take considerable time depending on the application), the log files are located in the directory `nvbitfi/logs/results/`. The framework generates multiple `*.tsv` files containing various outcomes. Later in this chapter, we will focus on the framework's outputs.

After presenting a detailed approach for configuring the framework to perform fault injection in several applications, we will now discuss the changes made to make the process easier.

3.2.4 Modified flow operations of NVBitFI

As seen in the preceding chapter, there are several challenging sequences that must be performed in order to successfully fault inject a program. Several of them can be automated with the use of a simple script. To accomplish this, we updated only the NVBitFI folder, leaving the NVBit framework folder untouched. To begin, we developed the script `fault injector.sh`. It enables fault injection of any application in any path without requiring knowledge of the preceding passages. To utilize the script, we must first consider the following three parameters:

- PATH of the application.
- The executable's name within the application's PATH.
- The desired number of fault injections.

A common example of a standard use of the script is having an application in the path `/root/app/your-project` with the executable name `your-project-name` and intending to perform a 10000 fault injection. To accomplish this, use the following command.

- `./fault_injector.sh /root/app/your-project your-project-name 10000`

The script was developed using the previously stated `test.sh` script as a starting point. To facilitate the process of automating the entire normal workflow, we created a new folder named pattern and included the following four files:

1. `Makefile`. This file is used to build the executable and generate the golden output during the workflow activity. If there is no makefile, the *fault-injector.sh* script inserts this file in the application folder. If a makefile already exists, it just adds the lines of code necessary to generate the golden result (see passage 3 of the previous part).

2. `params.py`. This file, which is located within the script folder, must be updated in accordance with the steps in passages 6 and 7 of the preceding part. The script edits those sections and then replaces the altered file with one already present in the script folder.

3. `run.sh` and `sdc_check.sh`. These two files must be copied into the program folder and updated as described in the previous section's passages three and four.

Essentially, the `fault_injector.sh` script extends the existing test.sh script by adding all the necessary code and files to automate all substantial passages from the previous section, allowing us to do fault injection in a more flexible manner.

Apart from those additions, the injector folder's file `injectfunc.cu` was altered. This file provides all information regarding the currently injected fault. It is launched prior to instrumenting the application. It is used to configure the fault injection parameters.

Because those details are not always saved in the file *nvbitfi-injection-log-temp.txt*, they are first temporarily saved in the `std_out.txt` file, and then some lines of code are added to the script `run_one_injection.py` to cut and print those details in the personal file that will be discussed later. Finally, a `personal_parser.py` script was added to manage the newly created data. We'll discuss it further later.

The following section will examine the framework's output and provide a full explanation.

3.3 Classification of Outcome of NVBitFI

NVBitFI's framework splits the results of the fault injection campaign in the same way that the other tools we've looked at previously (GPU-Qin and SASSIFI).

However, before delving into the details of all the reports and files that comprise the framework, it is necessary to grasp the group ID and the bit flip model that contain the framework. In essence, the framework splits all SASS instructions into distinct groups, each with a similar set of properties. The following is a quick rundown of the various groupings that the framework employs:

- `G_GPPR`. All those instructions are written in general purpose registers and predicate registers.
- `G_NODEST`. All instructions that do not have a register as a target register.
- `G_PR`. Unlike the group ID `G_GPPR`, the instruction of this group targets only predicate registers.
- `G_LD`. Collect all of the instructions that are stored in memory in this group.
- `G_FP64`. This category includes all instructions that perform floating point operations on a variable with a 64-bit dimension.
- `G_FP32`. The same as above, but with a 32-bit dimension.
- `G_GP`. This group ID is the one that is most frequently used in this thesis and the experiments conducted. Unlike the `G_GPPR`, this group ID contains all of the instructions that operate in the general purpose register. All of the instruction opcodes that target the predicate registers are missing in this group ID.

Bit-flip models (BFMs) supported by the framework include the following:

- *Single bit-flip.* During fault injection, the framework flips only one bit in a register.

- *Double bit-flip.* During the fault injection, the framework flips two adjacent bits in a register.
- *Random value.* During the fault injection, the framework flips a random value.
- *Zero value.* During fault injection, the framework sets the bits of a register to zero.

The group ID described above is referenced in the framework with a number between 1 and 7. The bit-flip model is instead referenced in the framework with a number between 0 and 3. The framework parses all of the results received throughout the fault injection campaign and puts them in the folder `/nvbitfi/logs` at the end of the campaign. The NVBitFI creates a folder with the application's name and divides the results into two folders:

- *Results.* It includes three distinct `.tsv` files. These files contain the fault injection campaign's reports, the file's name, and modifications for each fault injection campaign based on the amount of fault injections conducted and the type of fault injection, but they all terminate the same form, as follows:
  - `_stats.tsv`. This file contains summary data regarding the fault injection campaign that was conducted. The group ID and BFM of the fault injection, as well as the total number of fault injections completed. Additionally, it contains the time required in seconds to complete the fault injection campaign.
  - `_NVBitFI_details.tsv`. This is the most relevant file; it contains all of the fault injection's results, separated into SDC, DUE, and Masked. The framework subdivided the three categories and added a new one labelled *Uncategorised*, resulting in a total of 19 categories. There are three subcategories that result in a masked outcome, twelve that result in a DUE, and three that result in an SDC. During the studies conducted for this thesis, we discovered that the framework makes use of only a few of the 19 categories. We observed that the framework results are always "masked

for other reasons" for the masked outcomes among the three alternatives. Although the most common DUE error is the number seven, referred to as *Pot DUE: SDC or Kernel Error*, in a very rare scenario and only for matrix multiplication, some *DUE: Timeout* happens, which is comparable to the *Hang* of the other tool used for GPU fault injection. Rather than that, the SDC, like the masked results, always produces the outcome number sixteen, named *SDC: Standard output is different*. To summarize, the most frequently occurring outcomes of the experiments conducted as part of this thesis's framework are:

1. *SDC: Standard output is different.*
2. *Pot DUE: SDC but Kernel Error.*
3. *Masked: other reasons.*

– `_instruction-fractions.tsv`. This file contains all of the information about the application that was gathered during the profiling process. It comprises the entire amount of SASS instructions as well as a list of all possible OPCODE instructions. A percentage of instruction is provided in the program for each instruction opcode in relation to the application's total instruction. Additionally, the file contained the number of SASS instructions assigned to each group ID.

- *your-project-name.* This folder will include an equal number of folders as the number of fault injections carried out throughout the fault injection campaign. The name of each folder is made up of the application's name plus the group ID utilized, which is a number between one and seven, and the bit-flip model used for the fault injection campaign, which is a number between zero and three. Within such folders, several text files containing various pieces of information can be found. We will describe the most significant ones in the following list:

  – `nvbitfi-injection-info.txt`. This file provides all information regarding the currently injected fault. The group ID, the bit-flip fault model used, the name of the target kernel, the number of target kernels, the

register selection (a floating point number range between zero and one that the framework uses to determine the register selection), and the bit-pattern selection (like register selection, a number range from zero to one that is used for creating the mask used for choosing what bit in the register value must flip in the specific fault injection)

- `nvbitfi-injection-log-temp.txt`. This file contains comprehensive information regarding all types of fault injection. Among the information that is critical to get is the following: The CTAs represent the current application's block amount. The mask, as computed from the data in the file above. The register's value both before and after the mask is applied. The register's identification number. The instruction's OPCODE. The pcOffset ,that specifies the instruction number in the kernel's SASS file and the thread ID where the fault injection occurred.
- `stdout.txt`. The output of the corrupted application can be found in this file.
- `stdout_diff.log`. This file contains the difference between the corrupted and golden outputs.
- `stderr.txt` and `stderr_diff.log`. As with the previous two files, but this time focusing just on standard errors.
- `personal_info.txt`. This file is not included in the official release, however it is a modification to the framework that was developed to include a standard file that is required for the experiments. The changes was necessary because when a DUE occurs in the application, the framework overwrites all relevant information in the file *nvbitfi-injection-log-temp.txt* with an error message from the kernel execution. This is because the fatal error prevented the fault injection from being performed at all, most likely due to a memory violation. In certain instances, prior to the kernel being instrumented, all information regarding the present fault injection is recorded in this personal file, which is critical for the experiments. The

OPCODE is kept in this file as a reference number, and the ERROR type is stored as a number (they will be translated into a comprehensible name thanks to a dictionary during the parsing phase).

Apart from the folders mentioned previously, this folder entitled *your-project-name* has two additional folders named *sdcs* and *injection-list*:

– *sdcs.* This folder contains the same number of compressed files as the number of fault injections completed. They contain all of the information provided in the preceding folders.

– *injection-list.* It consists of a single text file containing all of the information necessary to conduct the fault injection campaign. It consists essentially of a series of fault injection rows, each of which comprises the kernel target name, kernel ID, target instruction count, register selection, and bit pattern selection.

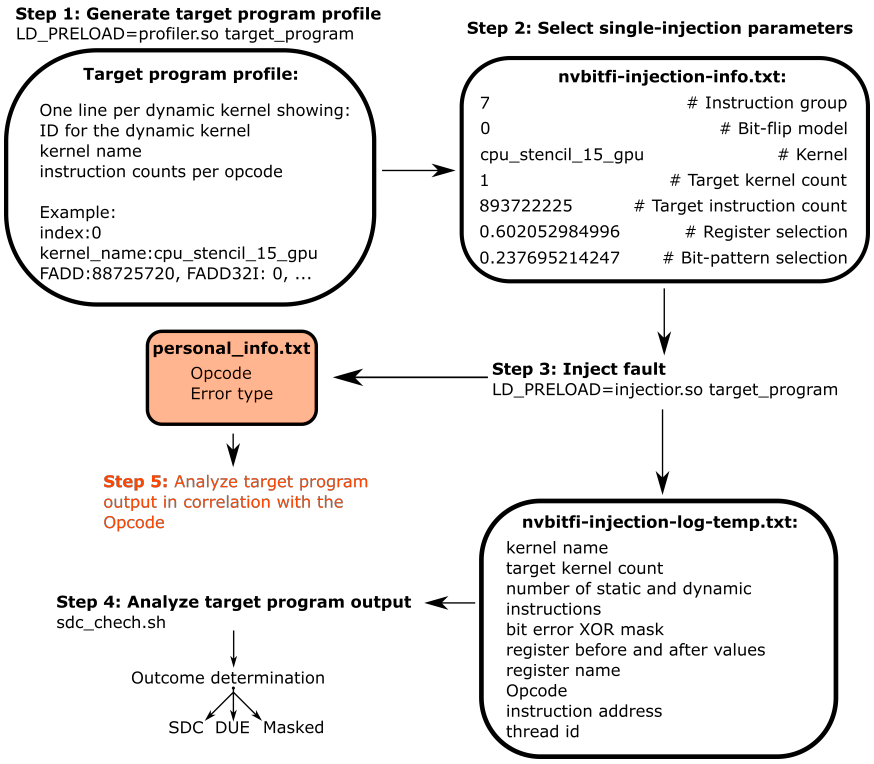


Figure 3.1. NVBitFI workflow [38]. In orange the added file.

In the flowchart above, it is clear where the new file is generated and the main passage has been used to conduct a fault injection campaign. Step 5 is carried out with the involvement of a Python script that takes use of a dictionary and the framework's already-provided output.

All of the modifications detailed in this section, as well as the framework itself, may be found in the branch built for this purpose on GitHub [31].

The following chapter will discuss the experiments conducted on a variety of applications. We'll examine the application's critical elements and the most critical instructions opcodes. After that, we'll use a hardening technique to an application to determine how to mitigate faults and so increase reliability.

Chapter 4

Experiments

Now that we have a solid grasp of the thesis framework and the adjustments necessary to collect all of the desired data, we can show which applications were utilized in the experiments, what they do, and finally the simulation results for those applications.

After evaluating the framework in a test application provided by the framework (`test-apps/simple add`), we evaluated the fault injector in several applications included with the NVDIA toolkit. We are aware that the framework has the ability to inject errors into a subset of those applications. We recognize that the framework may not always be capable of performing the fault injection. Occasionally, the application needs be altered in order to accomplish it. The framework wasn't able to profile the application, which was the most prevalent issue. Occasionally, this error was fixed by enabling the approximation profiling flag in the profiler's makefile. However, we recognize that there is no one-size-fits-all solution to this problem, and that each application should be modified prior to starting the fault injection campaign.

As a result, we developed a benchmark-test python script capable of testing the framework in all of the programs included with the Nvidia CUDA toolkit without modifying them. The CUDA toolkit organizes applications in subfolders that share a common working field. For instance, we have the folders `1-Utilities`,

`2-Graphics`, and `6-Advanced`, among others. After the testing were completed, we discovered that the framework was frequently capable of performing fault injection for simple applications. The test conducted in the 0-simple folder involved 40 applications, and the framework was able to conduct the fault injection campaign on 70% of them. However, for applications that required graphics libraries, such as the one we observed in `2-Graphics`, no fault injection was attempted. Further research should be conducted to determine why the framework was unable to profile the application and initiate the fault injection process in such apps.

Additionally, while the `sdc-check.sh` file is capable of discriminating between SDC and masked, the tool is unable of doing so in the presence of a time-dependent program, as the time performance changes with each execution. As a result, it is critical to consider updating the application's source code or, if that is not possible, modifying the `sdc-check.sh` file to handle this type of situation.

4.1 Benchmark applications

We've chosen to conduct fault injection experiments using the apps contained in the subdirectory `0_Simple`. This folder contains a variety of apps that are not too memory intensive. We conduct a fault injection campaign for a total of 20 applications in this folder. To obtain a more accurate estimation of the result created by an injected error in an application for a certain opcode, we ran a large number of fault injections, precisely 10,000 per application. This was the intention, however during the studies, it occurred on a rare occasion that some faults were not injected at all for various reasons. To circumvent this issue, we ran 11,500 fault injections each application and then collected only the top 10,000. We selected a subset of those twenty apps, eight, under analysis in the following sections. A brief description of each is shown below.

- ***cppIntegration***. This example explains how to incorporate CUDA into an existing C++ application, in which the CUDA entry point on the host side is simply a function called from C++ code, and only the file containing this function is compiled with nvcc. Additionally, it demonstrates how vector types can be utilized from within C++ [29].

- ***cdpSimplePrint***. This sample explains how to use CUDA Dynamic Parallelism to perform a basic printf [29]. The application specifically launches two blocks of two threads each. On the device, each thread prints its ID and then launches two blocks and two threads. The GPU will repeat this process until it reaches the maximum depth of two.

- ***cdpSimpleQuicksort***. This sample explains how to use CUDA Dynamic Parallelism to set up a simple basic quicksort algorithm [29]. Each thread recursively creates new blocks for the following level, one for sorting the left half and one for sorting the right part.

- ***fp16ScalarProduct***. This simplest application computes the scalar product of two vectors of FP16 numbers [29].

- ***matrixMul***. This sample demonstrates how to perform matrix multiplication. It was written for the sake of clarity and to demonstrate various CUDA programming concepts, not to provide the fastest generic kernel for matrix multiplication [29]. The matrix multiplication app operates with two matrices, both of which are 352 x 352, with a block size of 32 x 32.

- ***simpleAtomicIntrinsic***. A straightforward demonstration of atomic instructions in global memory [29]. The kernel is implemented in an external file in this version. After the kernel is launched, it performs a sequence of atomic functions successively as a test. Examples include *atomicDec*, *atomicMin*, *atomicAdd*, and so on.

- ***simpleCallback***. This sample supports multi-threaded heterogeneous computing tasks using the CUDA 5.0-introduced CPU callbacks for CUDA streams and events. Using the CUDA API's thread safety, easily and efficiently create heterogeneous workloads that float between CPU threads and GPUs. The workloads in the sample are organized in the following order: CPU preprocessing -> GPU processing -> CPU post-processing.GPU workloads are distributed all through the system's available GPUs [29]. In the case of the Jetson nano, a single GPU is provided.

- ***vectorAdd***. This sample is a basic implementation of vector addition, element by element [29]. The workload was divided into 196 blocks of 256 threads each by the program. Each vector contains 50,000 integer elements. Additional lines of code are included in this version for error checking.

4.2 NVBitFI - Evaluation of the Benchmark applications

With a firm understanding of what the apps do, and prior to observing the outcome of the framework's fault injection campaign, we can gain some interesting results from the profiling process of those applications.

4.2.1 Profiler Results

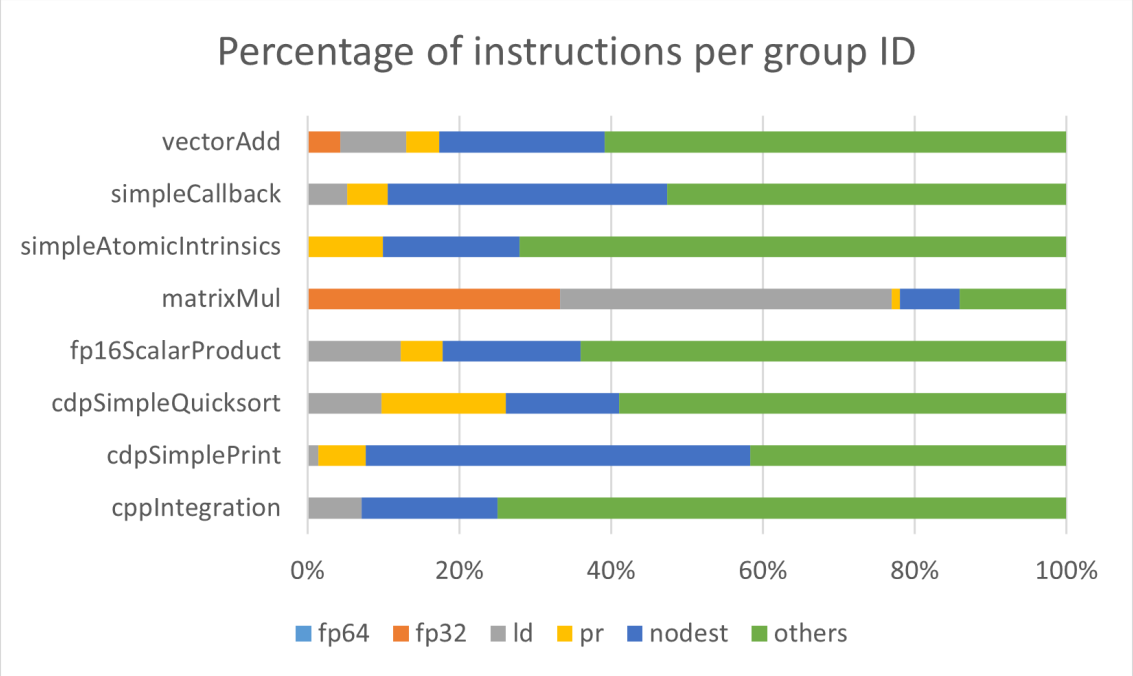


Figure 4.1. Group ID percentage for each application

As illustrated above, the framework labels the majority of the instruction code as **others**, which is the most often used group-ID. vectorAdd and matrixMul are the applications that contain the most SASS instructions in the group-ID **fp32**. This is because of the massive amount of mathematical operations conducted there. Finally, we can see that none of the applications listed above make use of the **fp64**

instruction type group-ID. Finally, we did not show it in the chart, but the group-ID utilized throughout the fault injection campaign was always **GP**, which stands for general purpose register. It is the total of the group-IDs above, except for the predicate registers.

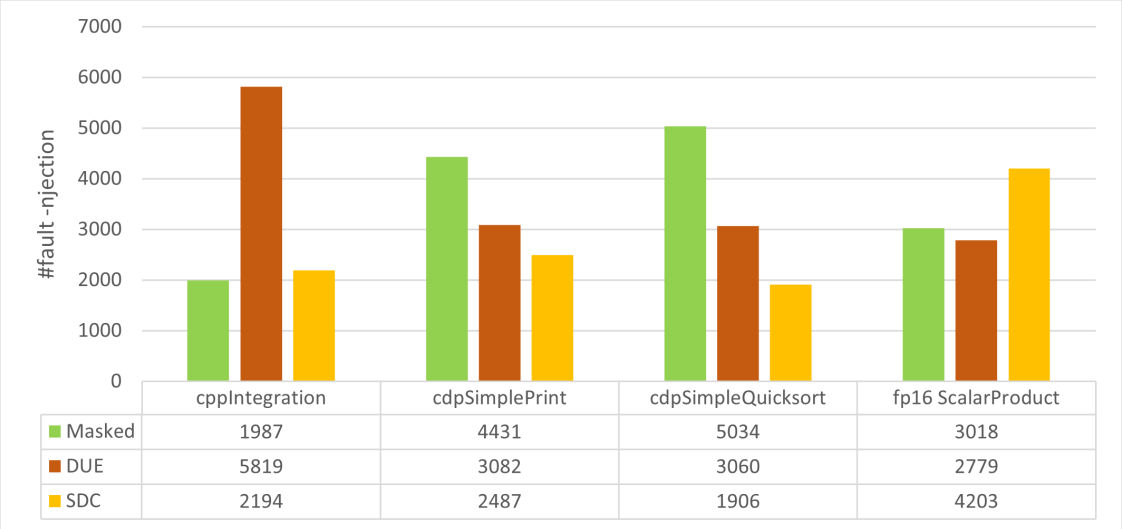
4.2.2 NVBitFI - Standard Outcome

Following the completion of the profiling phase, as is highlighted above, the framework begins injecting the fault into the application. The results are organized in this section into two parts, each comprises four applications, to provide a clear representation of the data.

The first four applications are depicted in the chart below. As can be observed, the fault injection generated a large number of DUE errors in the *cppIntegration*. This means that the application was frequently unable to complete without crashing. Rather than that, the SDC faults are relatively equivalent in magnitude to the instances when the application's output was unaffected (*masked* outcome).

We observe a higher number of masked fault injections in the *SimplePrint*, similar to the *simpleQuicksort*, but the DUE errors are always more than the SDC. It's worth noting that several faults in the *simpleQuicksort* application were labelled as *Uncategorized*. By analysing those types of fault, we discover that they were caused by internal framework issues and that no fault injection was conducted at all. As a result, we excluded those errors and analysed only the first 10,000 fault injections that did not fall under the category of *Uncategorized*.

Finally, we can see that the *fp16ScalarProduct* application is the only one of the apps that has more SDC than DUE.



	cppIntegration	cdpSimplePrint	cdpSimpleQuicksort	fp16 ScalarProduct
Masked	1987	4431	5034	3018
DUE	5819	3082	3060	2779
SDC	2194	2487	1906	4203

Figure 4.2. Fault injection results with standard outcome of NVBitFI

The following figure contains the outcomes of the past four applications, which contain important data to analyze. Starting with the matrixMul application, we can see that SDC fault injections account for a minor percentage of all fault injections. This means that, for the most part, the application is unaffected by the error or crashes (DUE). Additionally, this application is the only one among those utilized for tests that includes a timeout as an outcome. The framework treats the outcome of the timeout as a DUE. As a result, it is not mentioned specifically, but is totalled up with the other DUE faults.

There are only a few SDCs for the *simpleCallback* application. The results are quite similar to those for *matrixMul*, although as with *simpleQuicksort*, a sizeable portion of the errors (967 of the total fault injection) were labelled *Uncategorized*. Again, we explore reasoning, and by inspecting the `std_err` file's contents, we discover that the application does not complete owing to a segmentation fault. As a result, the correct error in this instance may have been *Pot DUE: Different Error Message*. As a result, those errors are grouped along with the already-labelled DUE errors.

All exposed SDCs were labelled as *Uncategorized* for the *vectorAdd* application.

We examine and discover that the application has an internal results checker in the `std_err` file. As previously stated, this internal checker terminates the application when it detects that some elements of the sum are incorrect. As a result, it can be categorized as an SDC, because without the checker, the application will produce different output from the golden ones. However, because this is the least reliable application in the set picked, we will attempt to improve its reliability and compare the results achieved later.

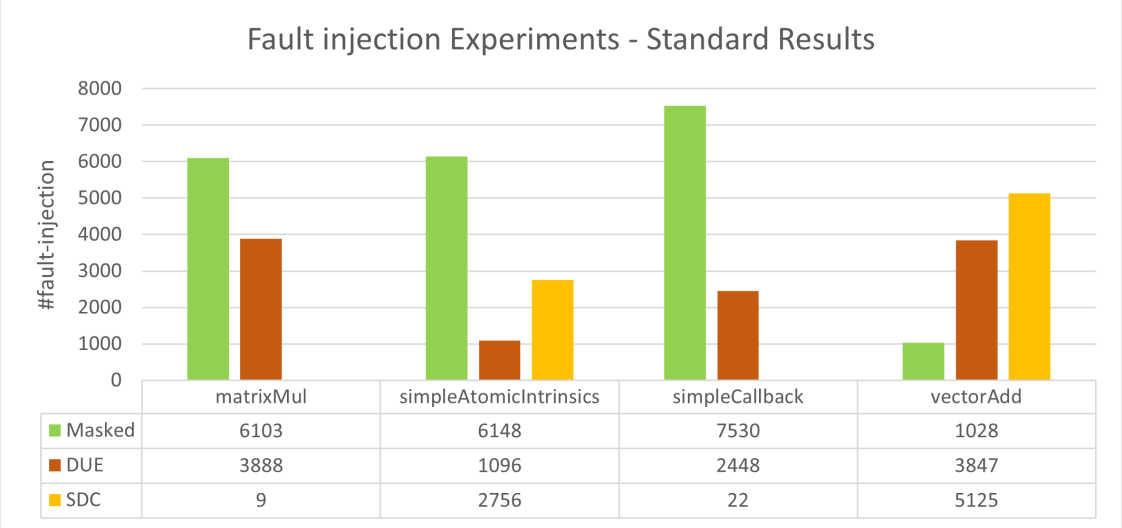


Figure 4.3. Fault injection results with standard outcome of NVBitFI

In this part, we examined the framework's functionality against a set of benchmark apps. We observed the group ID differentiation following the profiling process and, more crucially, we observed the framework's output following the completion of the fault injection campaign. Additionally, we recognize how critical it is to assess the outcome acquired upon the completion of the fault injection, as the framework is sometimes unable to determine the type of error generated by the fault injection without applying modification to the application or checker.

The following section will examine not only the fault injection results as specified by the framework, but also the most critical instruction opcodes that cause errors in those applications.

4.2.3 NVBitFI - Outcome differentiated by OPCODE

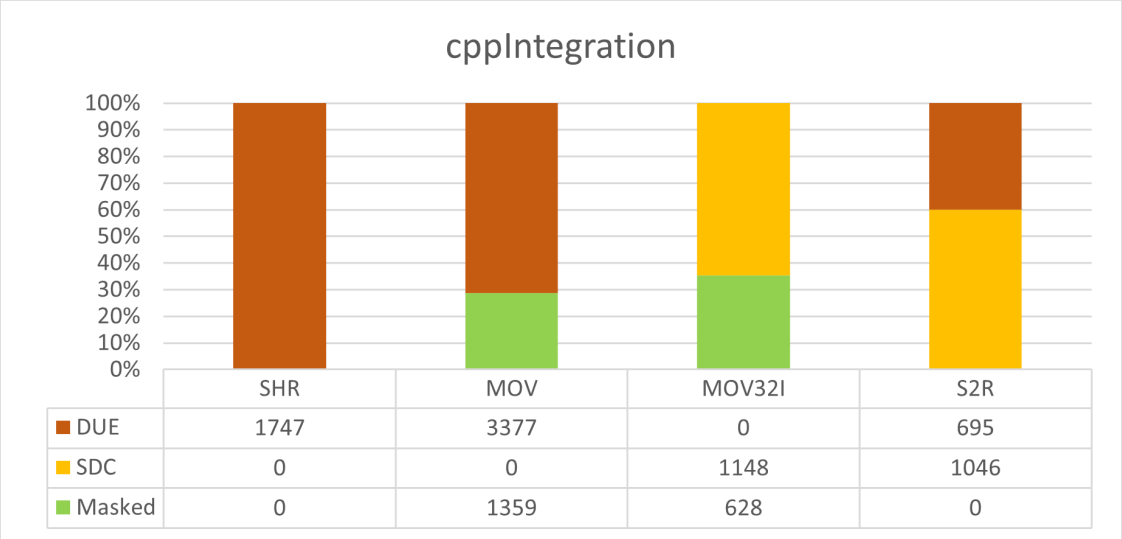
In Chapter 3, we saw how some modifications to the framework were made to enable the creation of `personal_info.txt` file. This file contains all of the information necessary for the fault injection. For the purposes of this thesis, just the OPCODE and error numbers were considered. After the fault injection campaign was completed, the files were analysed and the outcomes were collected in the tables shown below in an excel file using a personal Python script.

To compare the results obtained with those obtained in the previous experiments, we chose the same subset of 8 of those applications. It's critical to note that the framework randomly selects the SASS instruction and thread to execute. This means that some opcodes are statistically more targeted than others. As a result, at the end of the 10,000 fault injections, we filtered the fault injections performed in the instruction opcode where the total number of fault injections was less than 5% of the total (500), as data collected below this threshold has poor statistical meaning.

The charts acquired for the eight applications are then provided, along with some explanation of the obtained results (the charts and details are displayed one by one on each page for readability).

- ***cppIntegration.***

  The chart below refers to the application *cppIntegration*. As can be seen, the MOV (move) instruction opcode is the most targeted, causing 70% of DUE and 30% of *masked*. When the fault injection target is MOV, the application never creates an SDC. Rather than that, with the MOV32I, it was the inverse, with no DUE and 65% SDC. The SHR (Integer Shift Right [14]) opcode always causes the system to crash. This outcome is consistent across all applications. Even with the S2R (Move Special Register to Register) opcode, no injection produces a *masked* result.

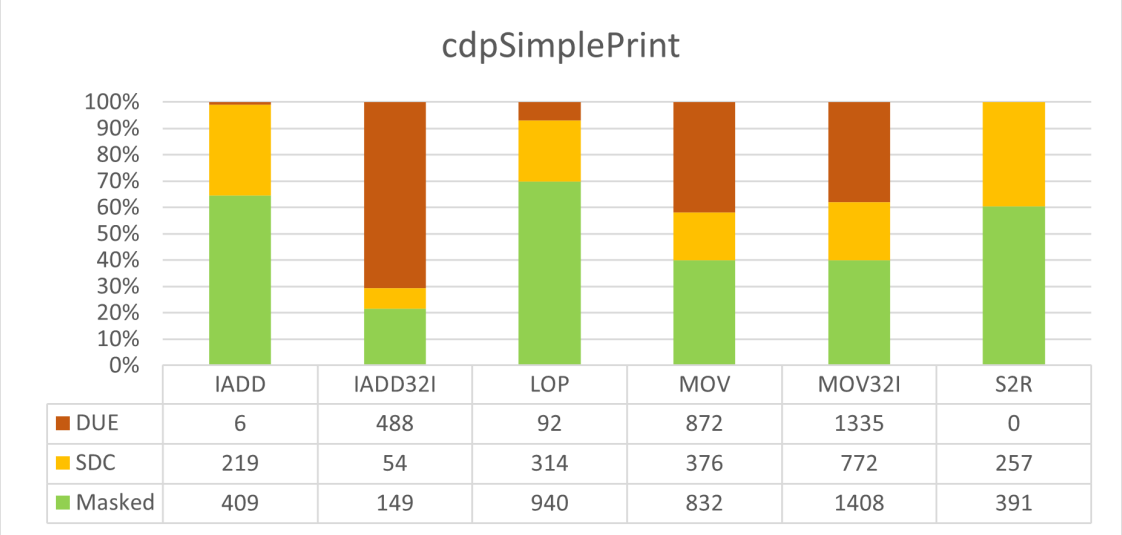


	SHR	MOV	MOV32I	S2R
DUE	1747	3377	0	695
SDC	0	0	1148	1046
Masked	0	1359	628	0

Figure 4.4. Fault injection results by opcode - cppIntegration

- ***cdpSimplePrint***.

  The opcodes targeted by this application are significantly more than those targeted by the cppIntegration application. We will only comment on the most critical opcode instruction. The surprising conclusion is that the IADD (Integer Addition) opcode has a large number of *masked* results, in contrast to the outcome of the following applications. In comparison, the IADD32I leads to a significant amount of DUEs (70%). The LOP instruction has a greater number of *masked* outcome. Similarly to the preceding application, the most targeted instruction was the MOV and MOV32I, with comparable results. It's worth noting that the fault injection executed on target S2R contains a significant amount of *masked* data. Rather than that, the same instruction opcode has a zero outcome labelled as *masked* in the *cppIntegration*.

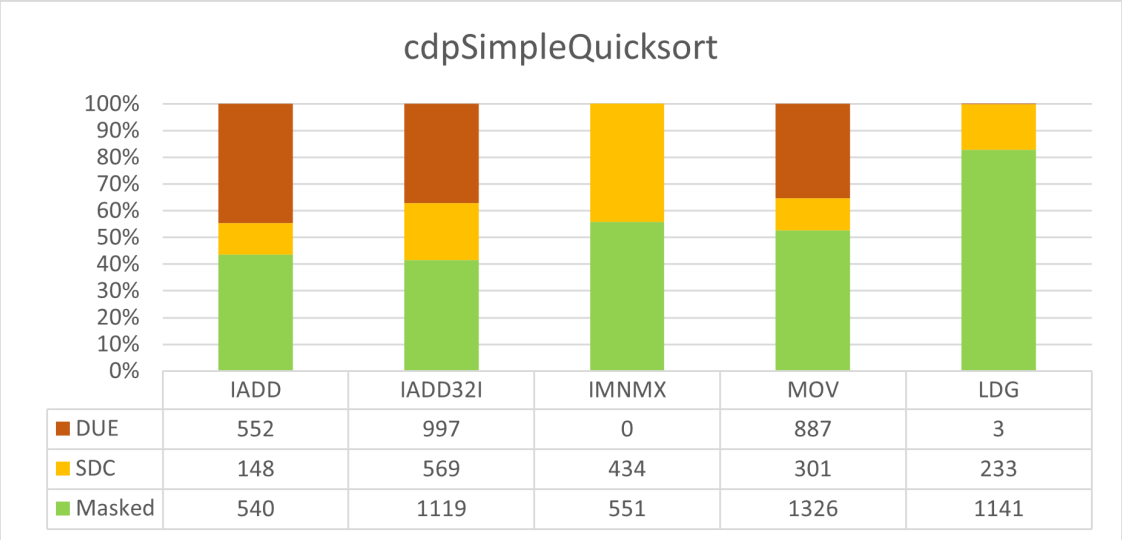


	IADD	IADD32I	LOP	MOV	MOV32I	S2R
DUE	6	488	92	872	1335	0
SDC	219	54	314	376	772	257
Masked	409	149	940	832	1408	391

Figure 4.5. Fault injection results by opcode - cdpSimplePrint

- ***cdpSimpleQuicksort***.

  The results of the *simpleQuicksort* are displayed in the chart below. Notable outcomes include the IADD and IADD32I, which behave similarly to *cdpSimplePrint* but with a higher number of DUE in the case of the IADD. The instruction IMNMX (Integer Minimum/Maximum) produces several SDC or *masked* results and never lead in a DUE error. This behaviour will be replicated in subsequent applications. The MOV instruction is the most targeted, and the LDG (Non-coherent Global Memory Load) instruction has the highest *masked* result.

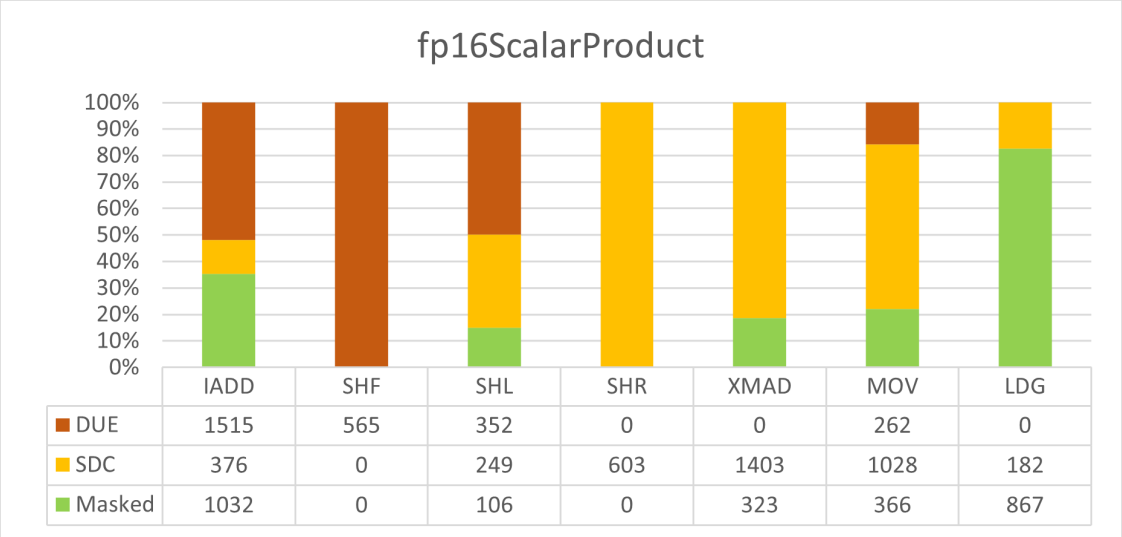


	IADD	IADD32I	IMNMX	MOV	LDG
DUE	552	997	0	887	3
SDC	148	569	434	301	233
Masked	540	1119	551	1326	1141

Figure 4.6. Fault injection results by opcode - cdpSimpleQuicksort

- ***fp16ScalarProduct***.

  It's worth noting that the SHR opcode always results in an SDC in this application, in contrast to all other applications in this subset, which always produce a DUE error when targeted. As shown before, the LDG instruction is the most resilient opcode instruction, with 80% of *masked* results in this case. On the contrary, the SHF (Funnel Shift) opcode is the worst because it always results in a DUE. Even in this scenario, the IADD behaves identically to the application described above.

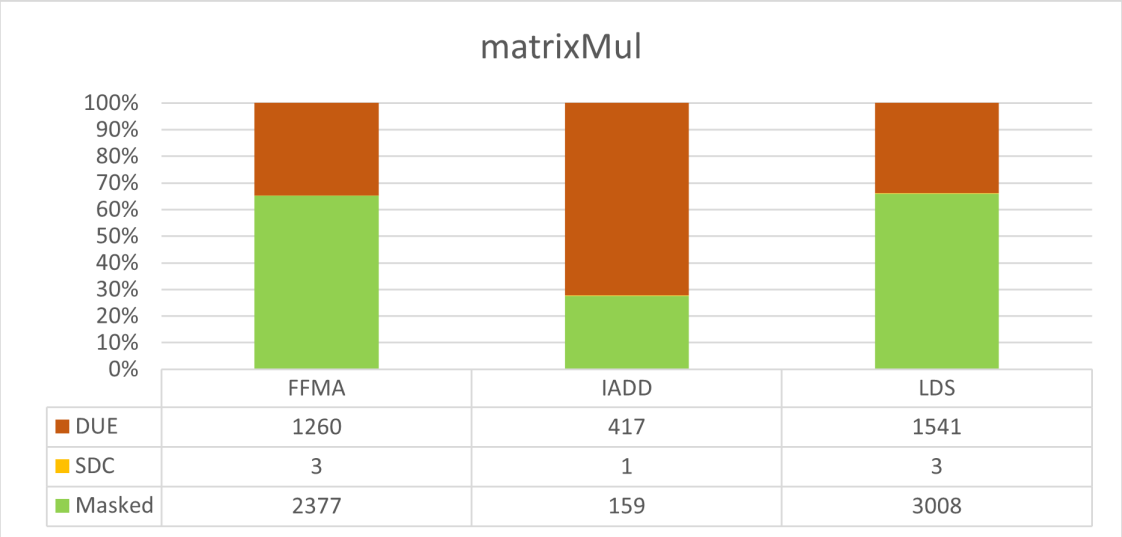


	IADD	SHF	SHL	SHR	XMAD	MOV	LDG
DUE	1515	565	352	0	0	262	0
SDC	376	0	249	603	1403	1028	182
Masked	1032	0	106	0	323	366	867

Figure 4.7. Fault injection results by opcode - fp16ScalarProduct

- ***matrixMul***.

  The *matrixMul* is frequently used as a benchmark. It's worth noting that the framework was unable to do the fault injection in this scenario without altering the source code. Within the sample, we attempted to solve the problem by varying the dimension parameters of both the A and B matrices. We conclude that by altering the grid and block size, the framework was able to accomplish the fault injection. When the block size is set to 32 x 32, the maximum matrix dimension is 352 x 352. The maximum size of a 16 x 16 block is 272 $\times$ 272.



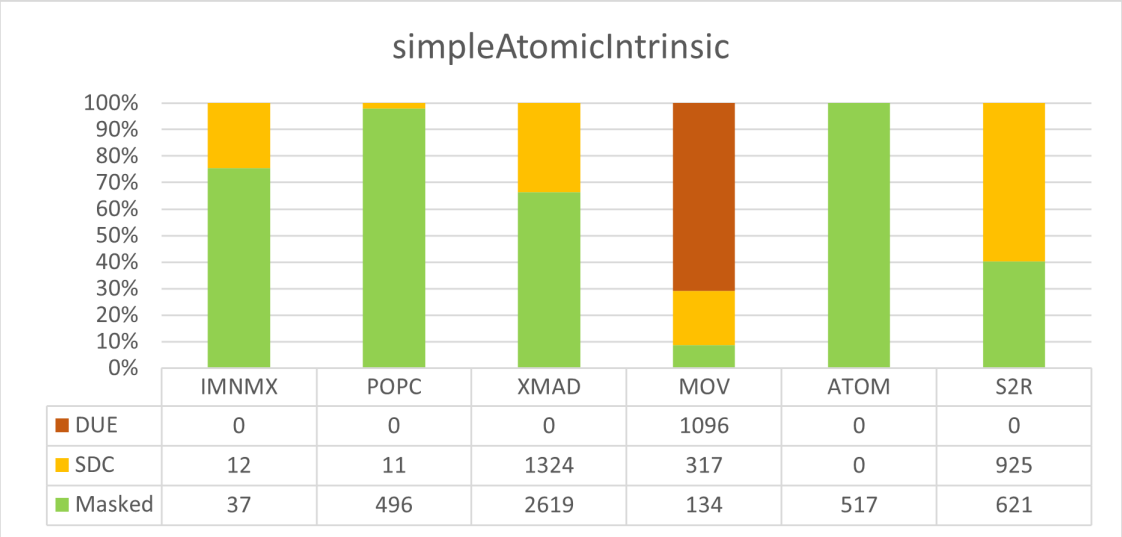
	FFMA	IADD	LDS
DUE	1260	417	1541
SDC	3	1	3
Masked	2377	159	3008

Figure 4.8. Fault injection results by opcode - matrixMul

Furthermore, while the fault injection in this matrixMul targeted 11 distinct instruction opcodes, the most of these were targeted only a few times. Only the three instruction opcodes highlighted above exceeded the 5% threshold. The opcode LDS (Load inside Shared Memory Window) was the primary target, accounting for 45% and 36% of FFMA respectively (FP32 Fused Multiply and Add). Even in this instance, the IADD confirms previously observed behaviour. The LDS opcode and the FFMA opcode are comparable.

- ***simpleAtomicIntrinsic***.

  The instruction opcode XMAD (Integer Short Multiply Add) was targeted the most in this program, accounting for 39% of total fault injections, and we can observe that the most critical opcode is the MOV, which generates a large amount of DUE errors. By comparison, the ATOM (Atomic Operation on generic Memory) opcode did not contain any errors in any of the injected faults. This demonstrated that atomic operations are the safest.



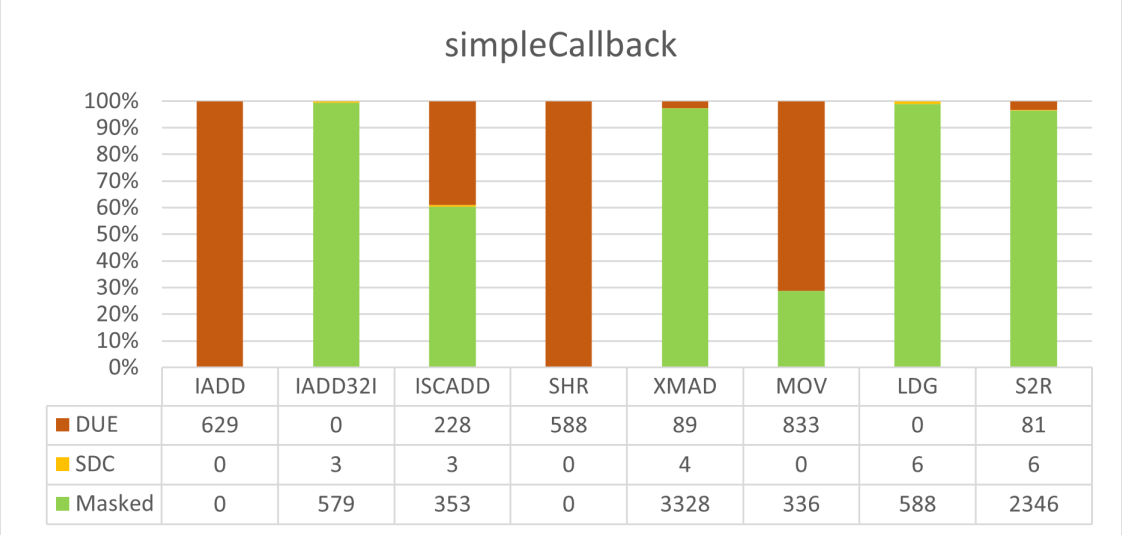
	IMNMX	POPC	XMAD	MOV	ATOM	S2R
DUE	0	0	0	1096	0	0
SDC	12	11	1324	317	0	925
Masked	37	496	2619	134	517	621

Figure 4.9. Fault injection results by opcode - simpleAtomicIntrinsic

- ***simpleCallback***.

  This application's fault injection was the most balanced in terms of targeted instruction code. The XMAD instruction opcode was the most targeted, while the LDG instruction opcode was the least addressed, with 34% targeting the first and 6% targeting the last.

  In this case, the IADD altered a lot of the previously observed behaviour with only DUE errors. On the other hand, the IADD32I produces almost exclusively *masked* output. With only DUE errors, the SHR instruction opcode confirms his behaviour. The LDG, XMAD, and MOV all validate the preceding application's behaviour. Instead, the S2R opcode displays about 95% of *masked* results, in contrast to previous applications that produced more SDC outcomes.

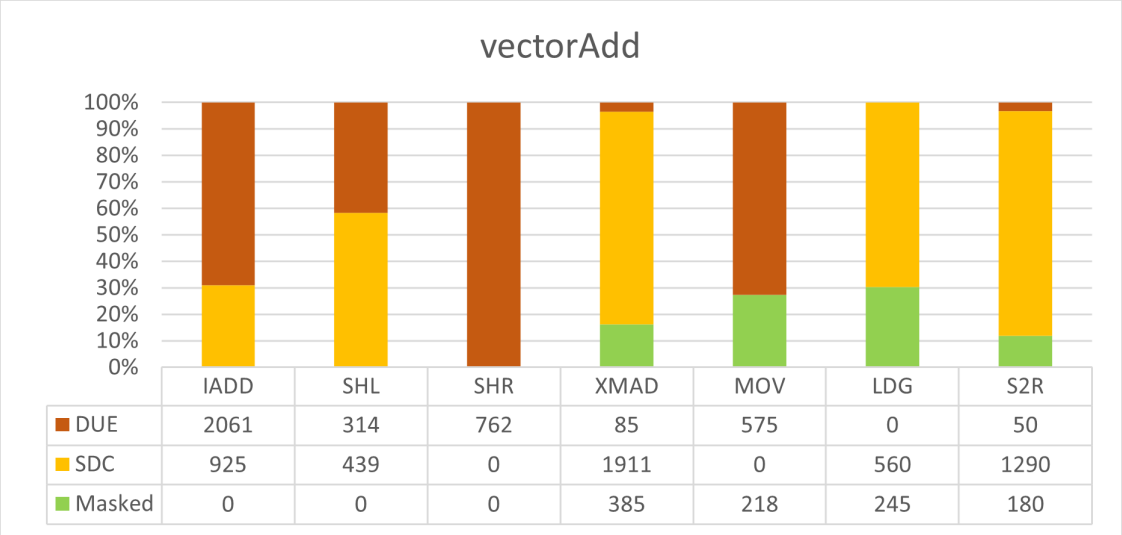


	IADD	IADD32I	ISCADD	SHR	XMAD	MOV	LDG	S2R
DUE	629	0	228	588	89	833	0	81
SDC	0	3	3	0	4	0	6	6
Masked	0	579	353	0	3328	336	588	2346

Figure 4.10. Fault injection results by opcode - simpleCallback

- ***vectorAdd***.

  Finally, we have some confirmation regarding the SHR and MOV opcodes in the vectorAdd application. The primary distinction is in how the XMAD, LDG, and S2R instruction opcodes behave. There are more SDCs in this application than in the previous ones.



	IADD	SHL	SHR	XMAD	MOV	LDG	S2R
DUE	2061	314	762	85	575	0	50
SDC	925	439	0	1911	0	560	1290
Masked	0	0	0	385	218	245	180

Figure 4.11. Fault injection results by opcode - vectorAdd

As can be observed from the results above, some opcodes are more prevalent than others, which typically results in the framework constantly targeting the same instruction opcode. Additionally, while the behaviour of some specific opcodes, such as MOV and SHR, is fairly consistent across applications, the behaviour of others, such as IADD, might vary significantly depending on the application. The next section will attempt to combine the results of all benchmark apps and summarize the outcomes acquired from the experiments conducted.

Summarized results of the fault injection differentiated by opcode

As previously stated, fault injections were performed in a total of 20 applications at a rate of 10,000 per application. We highlighted only eight of them in order to demonstrate some of the variations in how the same opcode behaves across applications and to better understand what occurs when a fault is injected into a particular opcode.

As a consequence, we compiled all the data acquired for each application for a total of 200,000 fault injections into a table and then filtered out the most targeted opcodes in order to get more comparable statistics across applications and to highlight the most relevant opcodes. The selection was limited to opcodes with a minimum of about 10,000 fault injections. This resulted in the identification of a subset of five opcodes. As illustrated in the chart adjacent, the most targeted opcodes are MOV (42.6%), XMAD (11.2%), S2R (8.8%), IADD (8.45%), and FFMA (4.52%), in decreasing order of percentage, while the sum of the remaining opcodes is 24.4%.

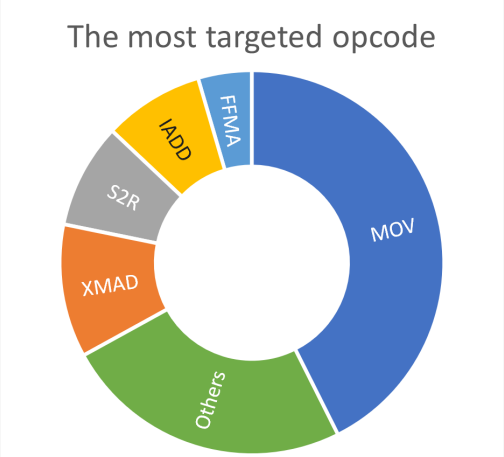
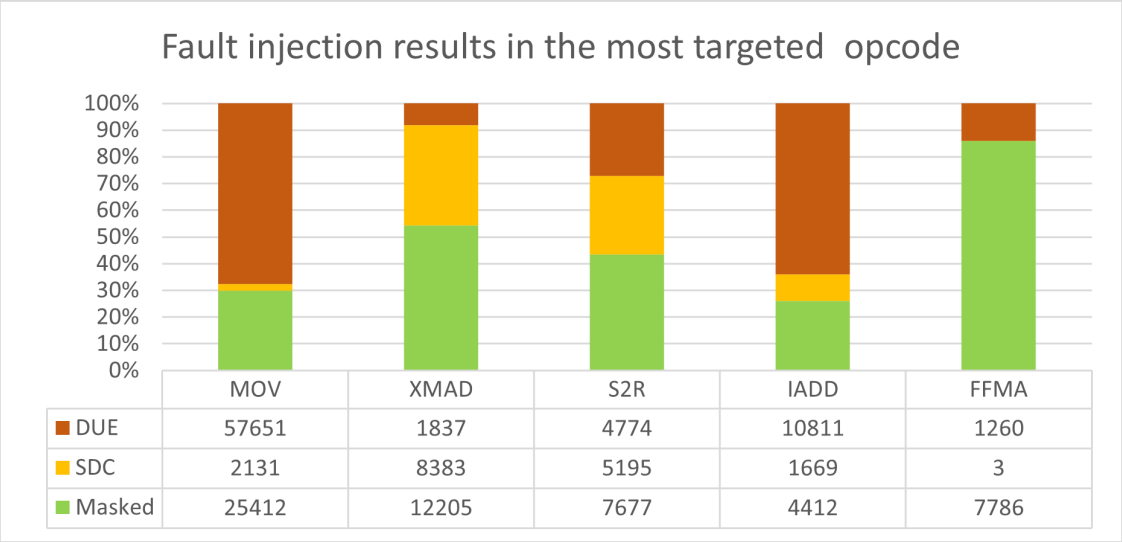


Figure 4.12. The most targeted opcode

The chart below summarizes the results of the fault injection differentiated by opcode for each of the five opcodes discussed previously. As can be observed, the MOV opcode behaves identically to the previously stated benchmark application, generating a large number of DUE errors. The XMAD and S2R opcodes had some weird behaviour in prior applications, with a few apps generating a large number of SDC and others generating a large number of *masked* outcomes, occasionally resulting in DUE errors. This behaviour is clearly apparent in the chart, and it prevents us from establishing a definite pattern for those opcodes. In prior applications, the IADD opcode frequently results in a crash, resulting in a DUE,

and occasionally in some "masked" outcome and very rarely in SDCs. In terms of DUE errors, this is the worst opcode. Finally, the FFMA opcode is the most robust, accounting for 85% of masked outcomes.



	MOV	XMAD	S2R	IADD	FFMA
DUE	57651	1837	4774	10811	1260
SDC	2131	8383	5195	1669	3
Masked	25412	12205	7677	4412	7786

Figure 4.13. Fault injection results in the most targeted opcode

Overall, the experiments indicate that some opcodes are more targeted than others, and that the IADD opcode appears to have the lowest resilience. Application hardening strategies can improve the system's and application's reliability. As a result, after conducting all of the above trials, we chose an application and, after implementing a hardening technique, repeated the above experiments to determine the level of reliability enhancement possible.

4.3 Fault injection Experiments in a fault resilient application

Following the experiments described above, we wanted to improve an application's reliability via the implementation of hardening technique [3]. To accomplish this, we chose to recreate the *vectorAdd* application seen in prior experiments from scratch and then apply a software hardening approach known as TMR (Triple Modular Redundancy). This technique involves redounding the execution of a given algorithm three times in order to obtain the same results. After obtaining all three findings, a second kernel handles the so-called voting process. It compares the three possible outcomes and chooses the correct one. This is accomplished through the use of a majority algorithm. If two of them are equal, one of them (whatever it is) is judged correct, while the third is considered incorrect. If all three are correct, the result is identical. Thus, even if one of the outcomes is incorrect, the algorithm will still function properly.

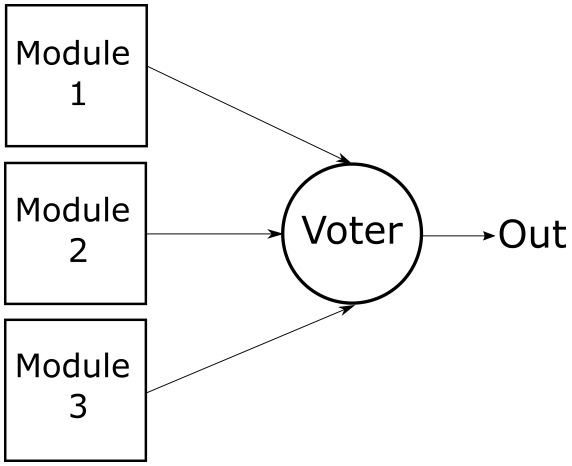


Figure 4.14. TMR workflow

We develop a kernel for *vectorAdd* that is called three times and saves the final vector in three distinct vectors. Following that, a second kernel named *Voter* selects and prints the chosen one. The issue with this technique is that the fault injection

could be directed towards the voter, which is the application's weak point. As a result, we conducted two distinct fault injection campaigns. One employing simply the kernel and doing the sum of the two vectors, and another involving only the voter as a target.

The chart below compares the fault injection performed in the standard *vectorAdd* program to the version that includes the TMR implementation. As can be observed, when the kernel responsible for performing the sum of the two vectors is targeted, the application works successfully in the majority of cases, with only twenty SDC faults but a large number of DUE errors. This will be the subject of the following debate.

Despite that, when the voter is the targeted kernel, the consequences are same to, if not worse than, the conventional application. This issue can be rectified by tripling the number of voters.

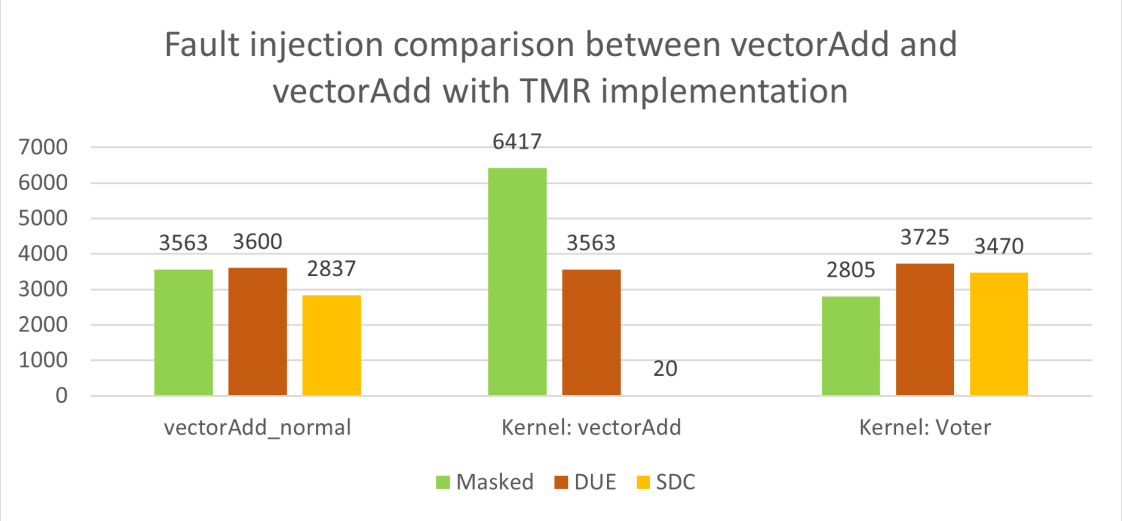


Figure 4.15. Fault injection comparison between vectorAdd and vectorAdd with TMR implementation

The figure above illustrates the expected outcome of NVBitFI and analyze the results. Following that, in order to gain a better understanding of the instruction opcodes that result in DUE and SDC errors, we employ the modified version seen

before to collect data for both the Voter and vectorAdd kernels' fault injection campaigns.

Most of the SDC errors that happen in the voter's kernel are caused by the SHL (Integer Shift Left), XMAD, or S2R opcodes, which are shown in the two charts below. Rather than that, DUE faults are primarily produced by the SHR, MOV, and IADD opcodes in both kernels.

Those results are effectively comparable with the results obtained in the figure 4.13.

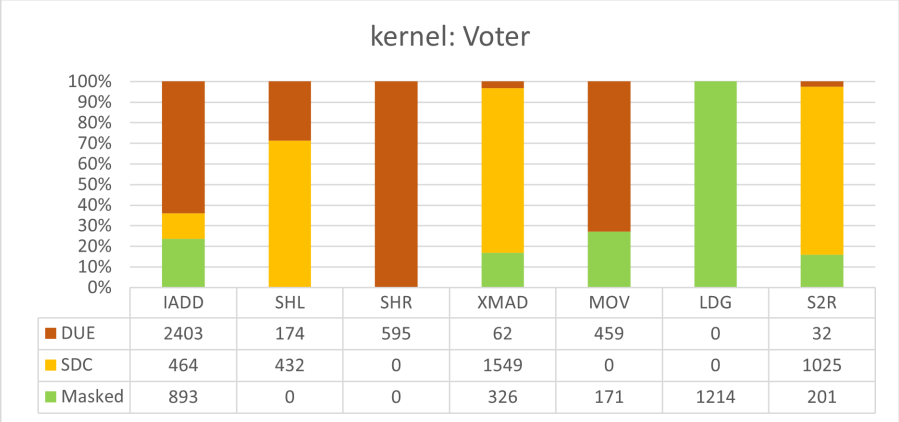


Figure 4.16. Results of fault injection differentiated by opcode - kernel: Voter

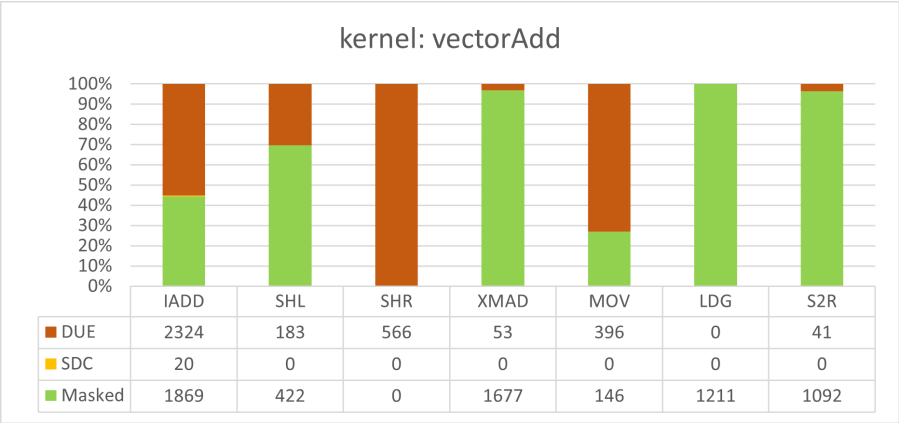


Figure 4.17. Results of fault injection differentiated by opcode - kernel: vectorAdd

We examined all of the experiments conducted for this thesis in this chapter. We observed the framework's output in a variety of apps and subsequently changed the framework to gather additional information on the fault injection. This enabled us to do experiments on the same apps, but this time gathering data on fault injection and classifying it according to opcode. Following that, we construct a fault-tolerant application using a technique called TMR, and with this application, we use the tool NVBitFI to compare the fault injection results, both standard and opcode-driven, between the untouched and hardened one.

Chapter 5

Conclusion

Various types of studies were conducted in this thesis using the tool NVBitFI and the board Jetson nano. We began by reviewing not only what a GPGPU is, but also the major testing techniques that have been developed in the past, ranging from hardware fault injection (including all possible types such as radiation, laser, etc.) to simulation and emulation fault injection, in order to gain a better understanding of all the possibilities considered for testing purposes.

Following that, we concentrated on understanding the GPU's microarchitecture and architecture, and then presented the board used in this thesis, the NVIDIA Jetson nano. After gaining a better understanding of the GPU architecture and the various types of fault injection techniques, we focus our efforts on assessing the currently available tools and methodologies for GPU fault injection. We discover that software fault injection tools for GPUs are relatively new in comparison to those developed for CPUs. Our focus was then on the framework employed in this thesis, NVBitFI, which has undergone extensive analysis.

During an overview of the framework, we addressed in depth each of the steps required to install and configure the platform in order to conduct a fault injection campaign in a generic application. This flow was quite intricate, and as a result, we created a fork on Github with some additional files to enable us to do fault injection campaign on several applications without having to edit the files that

must be modified in accordance with the original release. Finally, we discussed how the tool partitioned the output, what kind of files were created, and which ones were critical for our aim.

With the environment prepared, we discussed the framework's criticality, beginning with the premise that the framework is not always ready-to-use for every application, and that it occasionally fails to function effectively without intervention. Therefore, we selected a group of applications and conducted the fault injection campaign on them in order to see their standard behaviour. After doing the fault injection in a standard environment, we repeat the experiment using the same application but with additional framework customizations. We were able to correlate the type of faults with the opcode targeted during the fault injection as a result of those improvements. Thus, we collect data from the fault injection campaign and establish a correlation between the various applications of certain opcodes' behaviour.

Finally, we develop a fault-tolerant application by hardening the vectoAdd application using the TMR approach. Following that, we use the framework to conduct a fault injection campaign, which results in the TMR functioning properly, allowing the application to drastically reduce SDC errors while maintaining the same level of DUE errors. To have a clear understanding of what caused those failures, we ran a fault injection campaign, separating the result by opcode. Thus, we understand that the opcodes causing the DUE errors are always the same as those recognized in the previous benchmark programs, validating those results.

5.1 Future Work

- While the framework NVBitFI appears to be quite useful for testing applications that use CUDA in a real hardware environment, it is not possible to do fault injection without altering the source code for some apps. Further research must be conducted to determine why the tool is unable to conduct certain fault injection campaigns.

- Another issue with the tool was the introduction of bugs through matrix multiplication. While we identified a solution for this application by executing the fault injection on a smaller matrix, it is vital to understand why this occurs and to conduct fault injection campaigns on a range of different matrix dimensions.

- The studies conducted by opcode were really useful, but because the framework injects faults randomly, there was no way to target certain opcodes. Future work may attempt to expand the framework's capability to target specific opcodes.

- As a result, the most frequently used opcodes are the primary targets. Deeper analysis could focus on developing ways for hardening certain opcodes in order to increase their reliability.

- Following our investigation of the opcode's results, we observe that while the behaviour of some opcodes is quite consistent across applications, the behaviour of others varies significantly. Further research will be conducted to determine why this occurred.

- Finally, we examined the fault injection effects on the vectorAdd application using a TMR implementation. The results demonstrated the voter kernel's vulnerability. As a result, future researcher may implement a triple TMR and attempt to replicate this implementation on various applications.

Bibliography

[1] Kevin Andryc, Murtaza Merchant, and Russell Tessier. «FlexGrip: A soft GPGPU for FPGAs». In: *2013 International Conference on Field-Programmable Technology (FPT)*. 2013, pp. 230–237. DOI: 10.1109/FPT.2013.6718358.

[2] S. Azimi et al. «A new CAD tool for Single Event Transient Analysis and mitigation on Flash-based FPGAs». In: *Integration* 67 (2019), pp. 73–81. ISSN: 0167-9260. DOI: https://doi.org/10.1016/j.vlsi.2019.02.001. URL: https://www.sciencedirect.com/science/article/pii/S0167926018305753.

[3] Sarah Azimi, Corrado De Sio, and Luca Sterpone. «A Radiation-Hardened CMOS Full-Adder Based on Layout Selective Transistor Duplication». In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 29.8 (2021), pp. 1596–1600. DOI: 10.1109/TVLSI.2021.3086897.

[4] Sarah Azimi, Boyang Du, and L. Sterpone. «On the prediction of radiation-induced SETs in flash-based FPGAs». In: *Microelectronics Reliability* 64 (Sept. 2016). DOI: 10.1016/j.microrel.2016.07.106.

[5] Sarah Azimi, Boyang Du, and Luca Sterpone. «Evaluation of transient errors in GPGPUs for safety critical applications: An effective simulation-based fault injection environment». In: *Journal of Systems Architecture* 75 (2017), pp. 95–106. ISSN: 1383-7621. DOI: https://doi.org/10.1016/j.sysarc.2017.01.009. URL: https://www.sciencedirect.com/science/article/pii/S1383762117300528.

[6] Sarah Azimi et al. «Analysis of Single Event Effects on Embedded Processor». In: *Electronics* 10.24 (2021). ISSN: 2079-9292. DOI: 10.3390/electronics10243160. URL: https://www.mdpi.com/2079-9292/10/24/3160.

[7] R. Baumann. «Radiation-induced soft errors in advanced semiconductor technologies». In: *Device and Materials Reliability, IEEE Transactions on* 5 (Oct. 2005), pp. 305 –316. DOI: 10.1109/TDMR.2005.853449.

[8] André R. Brodtkorb, Trond R. Hagen, and Martin L. Sætra. «Graphics processing unit (GPU) programming strategies and trends in GPU computing». In: *Journal of Parallel and Distributed Computing* 73.1 (2013). Metaheuristics on GPUs, pp. 4–13. ISSN: 0743-7315. DOI: https://doi.org/10.1016/j.jpdc.2012.04.003. URL: https://www.sciencedirect.com/science/article/pii/S0743731512000998.

[9] Paolo Burgio et al. «A Software Stack for Next-Generation Automotive Systems on Many-Core Heterogeneous Platforms». In: Aug. 2016, pp. 55–59. DOI: 10.1109/DSD.2016.84.

[10] Esteban Clua and Marcelo Zamith. «Programming in CUDA for Kepler and Maxwell Architecture». In: *Revista de Informática Teórica e Aplicada* 22 (Nov. 2015), p. 233. DOI: 10.22456/2175-2745.56384.

[11] Josie E. Rodriguez Condia et al. «FlexGripPlus: An improved GPGPU model to support reliability analysis». In: *Microelectronics Reliability* 109 (2020), p. 113660. ISSN: 0026-2714. DOI: https://doi.org/10.1016/j.microrel.2020.113660. URL: https://www.sciencedirect.com/science/article/pii/S0026271419307978.

[12] C. De Sio et al. «Radiation-induced Single Event Transient effects during the reconfiguration process of SRAM-based FPGAs». In: *Microelectronics Reliability* 100-101 (2019). 30th European Symposium on Reliability of Electron Devices, Failure Physics and Analysis, p. 113342. ISSN: 0026-2714. DOI: https://doi.org/10.1016/j.microrel.2019.06.034. URL: https://www.sciencedirect.com/science/article/pii/S0026271419305621.

[13] Corrado De Sio et al. «Analyzing Radiation-Induced Transient Errors on SRAM-Based FPGAs by Propagation of Broadening Effect». In: *IEEE Access* 7 (2019), pp. 140182–140189. DOI: 10.1109/ACCESS.2019.2915136.

[14] *Documentation CUDA*. https://docs.nvidia.com/cuda/cuda-binary-utilities/index.html. Accessed: 2022-03-12.

[15] Boyang Du et al. «Ultrahigh Energy Heavy Ion Test Beam on Xilinx Kintex-7 SRAM-Based FPGA». In: *IEEE Transactions on Nuclear Science* 66.7 (2019), pp. 1813–1819. DOI: 10.1109/TNS.2019.2915207.

[16] Mohammad Eslami et al. «A survey on fault injection methods of digital integrated circuits». In: *Integration* 71 (2020), pp. 154–163. ISSN: 0167-9260. DOI: https://doi.org/10.1016/j.vlsi.2019.11.006. URL: https://www.sciencedirect.com/science/article/pii/S016792601930402X.

[17] Bo Fang et al. «GPU-Qin: A methodology for evaluating the error resilience of GPGPU applications». In: *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 2014, pp. 221–230. DOI: 10.1109/ISPASS.2014.6844486.

[18] *Getting Started with Jetson Nano Developer Kit*. https://developer.nvidia.com/embedded/learn/get-started-jetson-nano-devkit. Accessed: 2022-02-20.

[19] Daniel Alfonso Gonçalves Gonçalves de Oliveira et al. «Evaluation and Mitigation of Radiation-Induced Soft Errors in Graphics Processing Units». In: *IEEE Transactions on Computers* 65.3 (2016), pp. 791–804. DOI: 10.1109/TC.2015.2444855.

[20] J. Gray and D.P. Siewiorek. «High-availability computer systems». In: *Computer* 24.9 (1991), pp. 39–48. DOI: 10.1109/2.84898.

[21] Design Guide. «Cuda c programming guide». In: *NVIDIA, July* 29 (2013), p. 31.

[22] Siva Kumar Sastry Hari et al. «SASSIFI: An architecture-level fault injection tool for GPU application resilience evaluation». In: *2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 2017, pp. 249–258. DOI: 10.1109/ISPASS.2017.7975296.

[23] Mei-Chen Hsueh, T.K. Tsai, and R.K. Iyer. «Fault injection techniques and tools». In: *Computer* 30.4 (1997), pp. 75–82. DOI: 10.1109/2.585157.

[24] Kojiro Ito et al. «Analyzing DUE Errors on GPUs With Neutron Irradiation Test and Fault Injection to Control Flow». In: *IEEE Transactions on Nuclear Science* 68.8 (2021), pp. 1668–1674. DOI: 10.1109/TNS.2021.3098845.

[25] J. Karlsson et al. «Using heavy-ion radiation to validate fault-handling mechanisms». In: *IEEE Micro* 14.1 (1994), pp. 8–23. DOI: 10.1109/40.259894.

[26] Maha Kooli and Giorgio Di Natale. «A survey on simulation-based fault injection tools for complex systems». In: *2014 9th IEEE International Conference on Design & Technology of Integrated Systems in Nanoscale Era (DTIS)*. IEEE. 2014, pp. 1–6.

[27] Kevin Lavery. «Discriminating Between Soft Errors and Hard Errors in RAM». In: *SPNA109*. 2008.

[28] Michael Nicolaidis. *Soft errors in modern electronic systems*. Vol. 41. Springer Science & Business Media, 2010.

[29] *NVBit - GitHub Cuda Sample*. https://github.com/tpn/cuda-samples/tree/master/v8.0/0_Simple. Accessed: 2022-03-08.

[30] *NVBit - GitHub Installation Guide*. https://github.com/NVlabs/NVBit. Accessed: 2022-02-20.

[31] *NVBitFI - GitHub customized version*. https://github.com/StefanoPisciotta/nvbitfi. Accessed: 2022-03-05.

[32] *NVBitFI - GitHub Installation Guide*. https://github.com/NVlabs/nvbitfi. Accessed: 2022-02-20.

[33] *NVIDIA Jetson Nano System-on-Module Datasheet*. https://developer.nvidia.com/embedded/dlc/jetson-nano-system-module-datasheet. Accessed: 2022-02-02.

[34] Zheng Qinghe et al. «CLMIP: cross-layer manifold invariance based pruning method of deep convolutional neural network for real-time road type recognition». In: *Multidimensional Systems and Signal Processing* 32 (Jan. 2021). DOI: 10.1007/s11045-020-00736-x.

[35] A. U. Rehman, Rui Aguiar, and João Barraca. «Fault-Tolerance in the Scope of Software-Defined Networking (SDN)». In: *IEEE Access* PP (Sept. 2019). DOI: 10.1109/ACCESS.2019.2939115.

[36] J.R. Samson, W. Moreno, and F. Falquez. «A technique for automated validation of fault tolerant designs using laser fault injection (LFI)». In: *Digest of Papers. Twenty-Eighth Annual International Symposium on Fault-Tolerant Computing (Cat. No.98CB36224)*. 1998, pp. 162–167. DOI: 10.1109/FTCS.1998.689466.

[37] L. Sterpone et al. «A 3-D Simulation-Based Approach to Analyze Heavy Ions-Induced SET on Digital Circuits». In: *IEEE Transactions on Nuclear Science* 67.9 (2020), pp. 2034–2041. DOI: 10.1109/TNS.2020.3006997.

[38] Timothy Tsai et al. «NVBitFI: Dynamic Fault Injection for GPUs». In: *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 2021, pp. 284–291. DOI: 10.1109/DSN48987.2021.00041.

[39] Tianyi Wang and Qian Kemao. *GPU Acceleration for Optical Measurement*. Dec. 2017. ISBN: 9781510617346. DOI: 10.1117/3.2314949.