PERFORMANCE-RELIABILITY TRADEOFF ANALYSIS FOR SAFETY-CRITICAL SYSTEMS WITH GPUS

A Thesis Submitted to
the Graduate School of Engineering and Sciences of
İzmir Institute of Technology
in Partial Fulfillment of the Requirements for the Degree of

MASTER OF SCIENCE

in Computer Engineering

by
Yağızcan SEZGİN

December 2023
İZMİR

We approve the thesis of **Yağızcan SEZGİN**

Examining Committee Members:

Professor Dr. Cüneyt Fehmi BAZLAMAÇCI
Department of Computer Engineering, İzmir Institute of Technology

Assistant Professor Dr. Deniz ÖZSOYELLER
Department of Computer Engineering, Yaşar University

Assistant Professor Dr. Işıl ÖZ
Department of Computer Engineering, İzmir Institute of Technology

8 December 2023

Assistant Professor Dr. Işıl ÖZ
Supervisor, Department of Computer Engineering
İzmir Institute of Technology

Professor Dr. Cüneyt Fehmi BAZLAMAÇCI
Head of the Department of
Computer Engineering

Professor Dr. Mehtap EANES
Dean of the Graduate School of
Engineering and Sciences

ACKNOWLEDGMENTS

I would like to express my sincere appreciation to my thesis advisor, Assistant Professor Işıl Öz, for her endless patience and her efforts to motivate me. I extend my heartfelt thanks to TTTech Auto Turkey for the provisioning of the NVIDIA Jetson Xavier NX Developer Kit, a crucial element that significantly enhanced the outcome of my thesis. I want to convey my deepest appreciation to my mother who remained a constant source of support and guidance during moments of hopelessness and consideration of giving up.

ABSTRACT

PERFORMANCE-RELIABILITY TRADEOFF ANALYSIS FOR SAFETY-CRITICAL SYSTEMS WITH GPUS

GPUs were mostly used for image processing purposes when they were first introduced. These applications can be considered non-critical, and they were not given sufficient importance for reliability. Due to the evolving nature of GPUs, they offer highly parallelized architecture and provide extremely powerful computation, they become one of the most crucial parts of the systems that have complex applications in safety-critical domains such as automotive and space to fulfill the high computational demand. In this thesis, we evaluate the performance and reliability tradeoff in the safety-critical domain.

We propose software-based redundancy schemes with different spheres of replications on the GPU4S benchmark in the safety-critical domain. Our proposal includes profiling the baseline application without any redundancy, applying fault injection using NVBitFI and changing implementation manually according to proposed redundancy schemes, measuring performance metrics such as execution time, memory copy operations, and power consumption on the real hardware that is widely used on target domain instead of using well-known GPU simulators to see actual performance.

We reveal that our proposed redundancy schemes are managed to eliminate all the soft errors in the cases if we apply full redundancy for single-kernel benchmarks, for the reliability evaluation with the cost of performance degradation, depending on the application. We show that most soft errors can be eliminated using partial redundancy for complex applications, with a small performance impact.

ÖZET

GÜVENLİK KRİTİK SİSTEMLERDE GPU KULLANIMININ PERFORMANS VE GÜVENİRLİK AÇISINDAN DEĞERLENDİRİLMESİ

GPU'lar ilk kez tanıtıldığında çoğunlukla görüntü işleme amaçlı kullanılmıştır. Bu uygulamalar genellikle kritik olmayan olarak kabul edilebilir ve güvenilirlik için yeterli önem verilmemiştir. GPU'ların evrilen doğası nedeniyle yüksek seviyede paralel mimarinin kullanılabilmesi ve son derecede güçlü hesaplama imkanı sağlaması nedeniyle otomotiv ve uzay gibi güvenliğin kritik olduğu alanlarda karmaşık uygulamalara sahip sistemlerin vazgeçilmez bir parçası haline gelmiştir ve alanlarda yüksek hesaplama gücü isteğini karşılayabilmek için kullanılmaktadır. Bu tezde, güvenlik kritik alanlardaki performans ve güvenirlik arasındaki dengeyi ve birbiri arasındaki ilişkiyi değerlendiriyoruz.

Güvenlik kritik alanda kullanılan GPU4S performans göstergesini kullanarak farklı replikasyonlar ile yazılım tabanlı yedekleme yaklaşımları öneriyoruz. Önerimiz, uygulamaların orijinal ve yedekleme olmayan hallerinin profil edilmesini, NVBitFI aracını kullanarak hata enjekte edilmesini ve önerilen yedekleme uygulamalarının yazılım implementasyonuna elle uygulamasını, ardından da hafıza ve yürütülme zamanları, güç tüketimi gibi performans ölçütlerinin hedef alanda yaygın bir şekilde kullanılan bir donanım üzerinde ölçümlenmesini öneriyoruz.

Önerdiğimiz yöntemlerin tek üniteye sahip uygulamalarda eğer tamamen yedekleme uygularsak geçici hataların hepsinin performans düşüşüyle beraber giderilebildiğini ortaya koyduk. Karmaşık ünitelere sahip uygulamalarda ise kısmi yedekleme uygulamanın birçok hatanın giderilmesinde ufak bir performans etkisiyle mümkün olabileceğini ortaya koyduk.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF LISTINGS

CHAPTER 1

INTRODUCTION

As the demand for high computational power continues to increase with each passing day, the use of hardware accelerators with high computational power has become necessary to meet these demands. GPUs are most often preferred among other hardware accelerators because they provide cost-effectiveness, versatility for usage on different applications, wide community support, and continuous improvement by vendors. Initially, GPUs were mainly used with image processing applications but after a while, they became one of the crucial components of other domains such as automotive and aerospace (Fickenscher et al. 2017; Kastensmidt and Rech 2016).

Safety-critical systems can be defined as systems where any malfunction or failure may cause severe damage, injury, or even death. These systems are defined as the results that could lead to unacceptable consequences (Knight 2002). Due to the seriousness of the consequences, hardware and software development, and design procedures also need to be evaluated mindfully to avoid any undesired result (Douglass 1998). Each industry such as automotive, space, and medical devices defines its standard for safety. Although there are sectoral differences, all of them are concerned with parameters such as real-time performance, reliability, power consumption, and determinism. Safety cases for automotive domain patterns and models have been published previously (Wagner et al. 2010; Palin and Habli 2010). Reliability can be provided by offering various levels of redundancy such as hardware and software-based solutions, error detection and correction mechanisms, temperature and power consumption monitoring, and on-board diagnostic mechanisms to detect failures on time (Portet et al. 2020).

Redundancy is one of the key concerns in safety-critical systems when designing applications in mission-critical domains such as autonomous driving or space applications to ensure reliability. GPUs are vulnerable to soft errors because of reduced supply voltages to optimize power consumption, intensive parallelism, where running thousands of threads may increase the likelihood of at least one thread being error-prone, and the high density of memory cells and transistors, which may affect the vulnerability of electrical noise and radiation.

Each redundancy technique costs increased execution time, and power consumption which needs to be considered on safety-critical systems. In this thesis, we present several redundancy schemes with different spheres of replication and evaluate performance and reliability tradeoffs under different cases.

1.1. Contribution of Thesis

We propose software-based redundancy techniques for CUDA kernels to eliminate silent data corruption in safety-critical domain applications. We apply different redundancy schemes with spheres of replication like input and output multiplication and evaluate the best redundancy techniques for single-kernel benchmarks. We consider different CUDA features like stream-based execution to benefit from highly parallel architecture for decreasing the redundant execution effort. After evaluating of best redundancy technique in single-kernel benchmarks for considering elapsed time for memory copy operations and kernel execution, as well as power consumption, we implement the best technique over complex benchmark applications that include more than one kernel. We apply these techniques under partial redundancy to aim to reduce silent data corruption as much as possible with minimal overhead by finding the most error-vulnerable kernel after evaluating the fault injection results for complex benchmarks.

1.2. Organization of Thesis

Chapter 2 provides background information about GPU architecture and CUDA programming model, a soft-error vulnerability in GPUs, and fault injection models. Chapter 3 gives literature research for related works in software-based redundancy, fault injection, and power measurement. Chapter 4 shows our proposed software-based redundancy techniques and evaluation of performance and reliability. Chapter 5 presents experimental results. Chapter 6 discusses the conclusion and future work.

CHAPTER 2

BACKGROUND AND MOTIVATION

2.1. GPU Architecture and Programming Model

GPUs have been used in graphical processing applications previously, but their use is becoming increasingly common in areas requiring high computing power due to their Single Instruction Multiple Data (SIMD) execution model structure which makes use of data parallelism (Aamodt et al. 2018). GPU terminology may vary among semiconductor companies and programming models such as Compute Unified Device Architecture (CUDA) and Open Computing Language (OpenCL). In this thesis, since the development and hardware environment are selected from NVIDIA GPU, further explanations will use CUDA terminology given in Table 2.1 only.

Table 2.1. CUDA Terminology

Terminology	**Definition**
thread	Execution unit
block	Group of threads
grid	Group of blocks
streaming multiprocessor	Computational Grouping
warp	Set of 32 threads

The GPU consists of several Streaming Multiprocessors (SM), and how many vary depending on the architecture and hardware. Each SM may include more than one Streaming Processor (SP) and comprises a warp scheduler, registers, and shared memory (Perez-Cerrolaza et al. 2022). The function which is executed on SMs is called the kernel. Register visibility is restricted to threads, so it is not visible from other threads. Shared memory is accessible to all threads within the same block, and it is the fastest memory element among all GPU architecture, but due to its size limitation, it is not always possible to fully utilize and benefit from implementation. L1 cache is shared among the same SM, so all the blocks within the SM can access the L1 cache, but shared memory is not accessible among blocks. SM is responsible for scheduling by groups of threads instead of individual scheduling using a warp scheduler. A thread is the smallest execution unit and multiples of threads create blocks. The grid consists of an array of thread blocks. Blocks and grids may have one, two, or three dimensions according to requirements on demand. For example, 1D convolution, image processing, and 3D simulations like volumetric data processing may fit one, two, and three dimensions, respectively. Grid configuration, number of blocks, and threads per block parameters are initialized by the user before kernel execution.

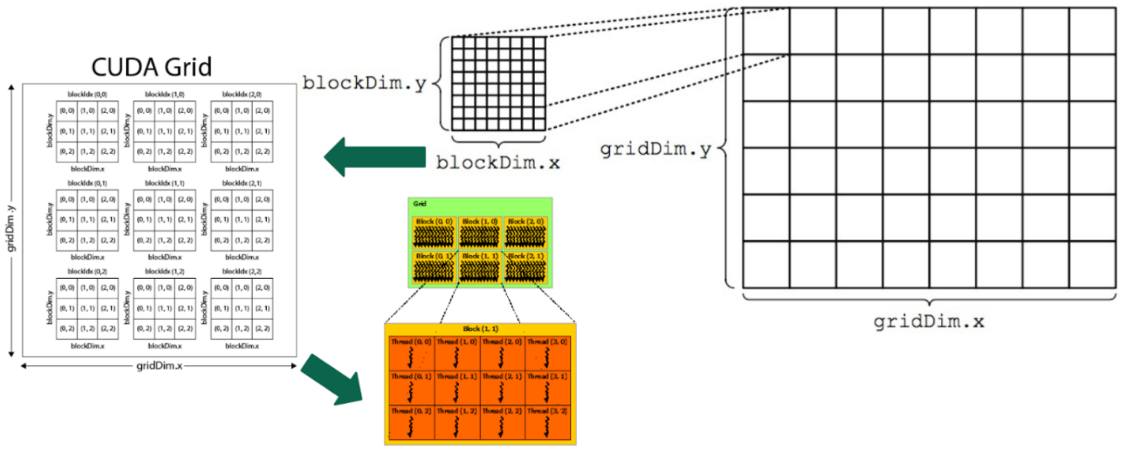


Figure 2.1. CUDA Thread, Block and Grid Architecture

Each warp includes 32 threads, and the same instruction is executed on threads that are in the same warp using the Single Instruction Multiple Thread (SIMT) concept. Warp divergence is widely discussed in performance-related improvement in CUDA development since it may cause threads to follow different execution paths and result in inefficient execution of threads due to conditional execution blocks in code (Kirk and Hwu 2016).

CPU (host) and GPU (device) have different memory regions, necessary memory allocations should be performed before the execution of the kernel on the device. Later, starting from the NVIDIA Pascal GPU architecture, Unified Memory is introduced which is a common memory address space and accessible among CPU and GPU. Data sharing between threads within the same block can be done by using shared memory which is extremely fast on-chip memory. Threads in different blocks can communicate with each other using global memory, atomic operations, or synchronization techniques like memory barriers.

In Figure 2.2, typical CUDA application execution workflow starts with preparing the input data on the host device, and allocation of necessary memory on device ①, since the host and device have separate memory spaces, data should be transferred from host to device ②. After transferring it, the kernel is ready to be launched on the device ③. Processed output is copied back from the device to the host after the calculation finishes on kernel ④. Finally, the buffer is deallocated to device memory ⑤. (Alcaide et al. 2021).

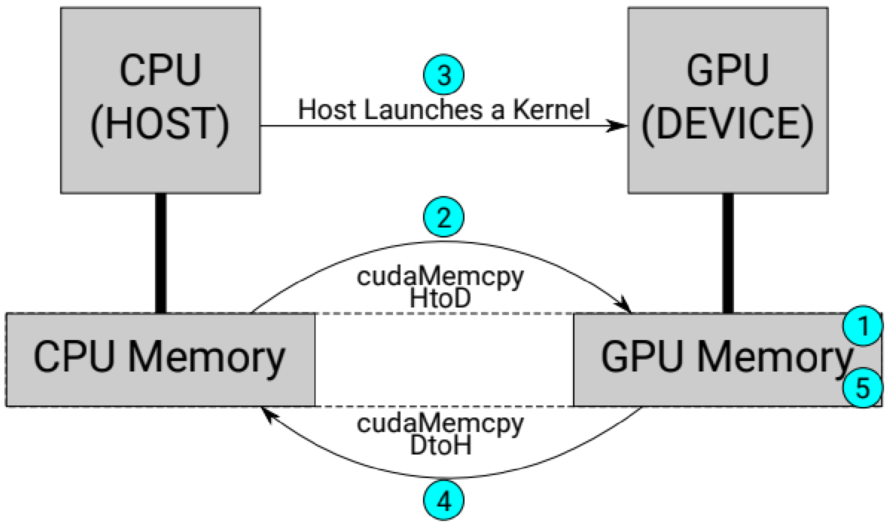


Figure 2.2. CUDA Execution Workflow (Alcaide et al. 2021)

2.2. Error Vulnerability in GPUs

Error vulnerability has become an important issue in reliability analysis in GPUs because reducing transistor sizes, increasing the number of transistors, and reducing supply voltages to optimize power consumption makes GPUs more error-prone. Errors can be classified as transient, permanent, and intermittent in general. Permanent faults are irreversible hardware errors that can be caused by increasing environmental temperatures higher than the thermal protection threshold (Defour and Petit 2013). Intermittent faults can happen sporadically in unpredictable intervals. Transient faults, affect the execution temporarily which may induce a single bit-flip in computer hardware that can be triggered by thermal neutrons, cosmic rays, or electrical noise (Mukherjee 2011). To see the undesired effect of faulty bit-flip, it should be read somewhere during execution and there should be no error detection and correction mechanism such as an error correction code (ECC).

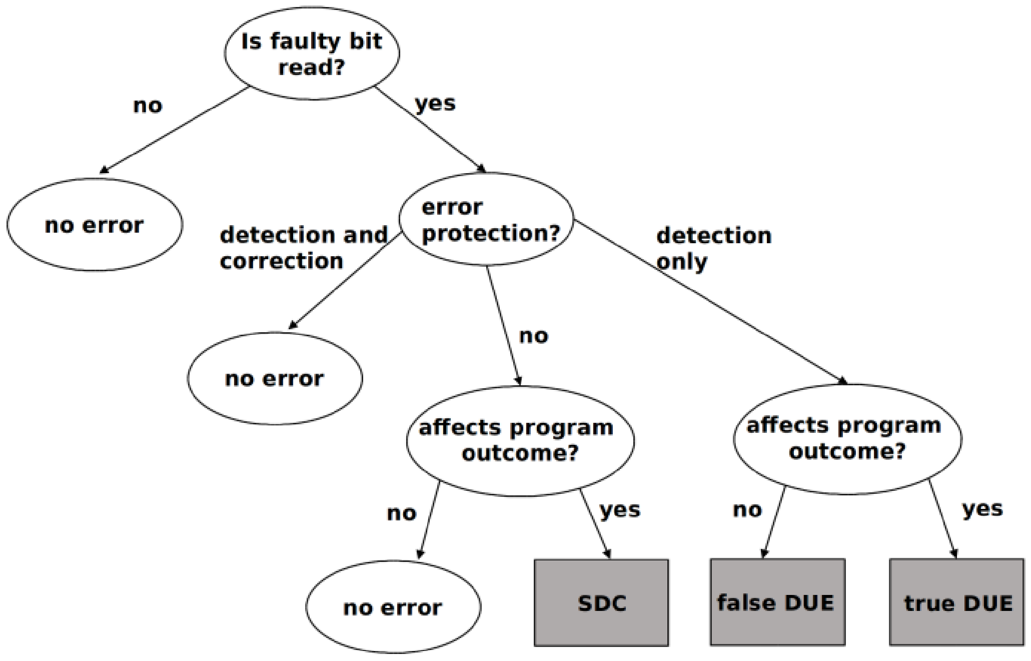


Figure 2.3. Error Classification (Mukherjee 2011)

Soft errors may cause silent data corruption (SDC) or detected unrecoverable errors (DUE) if they occur in a memory location or register. SDC is the most critical concern point since program execution finishes successfully without any error, but output might be different than expected which is undetectable if the system doesn't have any mechanism such as ECC, triple redundant execution, or redundant multithreading with majority voting. In this thesis, we evaluate the performance, reliability, and error resilience of our proposed redundancy techniques under simulation-based fault injection to evaluate soft-error vulnerability.

2.3. Fault Injection Methods

Reliability evaluations on GPUs are mostly performed with fault injection methods such as execution and hardware-based, simulation and software-based, execution and software-based methods (Perez-Cerrolaza et al. 2022). For execution and hardware-based fault injection experiments, the neutron-beam technique is widely used. A neutron source is used to generate a beam of neutrons which are uncharged subatomic particles that can penetrate through hardware components and are mostly used in aerospace and nuclear areas to evaluate reliability. It can be helpful to use combined approaches with architectural fault injection and neutron beam experiments to have a better understanding of the vulnerability of GPUs (Previlon et al. 2017). Harsing the system limits such as voltage, temperature, and operating frequency can accelerate the aging of devices which results in more sensitivity to intermittent errors. The effects of running hardware at high temperatures have been evaluated previously (Defour and Petit 2013).

Fault injection techniques such as neutron beam and pushing the environmental limits such as operating frequency, temperature, and voltage might permanently damage the device which is costly and prevents researchers from trying with several cases until gathering new hardware, it may significantly increase the experimental time and effort. Due to this fact, using fault injection tools is advantageous since most of the tools provide configuration of fault injection models such as single bit-flip, double bit-flip, random value, and zero value injection. There are available fault injection tools in literature such as Hauberk (Yim et al. 2011), GPU-Qin (Fang et al. 2014), LLFI-GPU (Li et al. 2016), SASSIFI (Hari et al. 2017) and NVBitFI (Tsai et al. 2021). In this thesis, we select NVBitFI as a fault injection tool since it supports our target hardware, which is based on Volta architecture, it does not require any source code modification and faster operation compared to previously published tools.

CHAPTER 3

RELATED WORKS

In this chapter, we show the related works that have been studied by other researchers lately because performance and reliability evaluation is becoming one of the hot topics in this field parallel to the wide usage of GPUs. We divide this chapter into three sections: software-based redundancy, fault injection, and power modeling with reliability evaluation.

3.1. Software-Based Redundancy

Redundancy techniques can be applied in several ways such as hardware and architecture, compiler, or software-based techniques. In this thesis, we are only interested the software-based redundancy techniques like serial, stream, or redundant-multithreading and we show the related works in this field.

Alcaide et al. (2019) offer a software-only diverse redundancy scheme by executing two independent kernel calls with duplicated input data and assigning each kernel execution to a different CUDA stream. Execution output is compared on a safety-compliant microcontroller which provides Dual Core LockStep (DCLS) cores. Kernels are classified considering their behavior for concurrency as short, heavy, and friendly kernels. Classification of kernels is used to understand the diverse redundancy nature of staggered execution. They offered solutions for any kernel according to classification to guarantee achieving diverse redundancy.

Mazzocchetti et al. (2022) provide a SafeSoftDR library to ensure diverse redundancy to avoid common cause failure which is required by safety-critical tasks to ensure reliability. It aims to take responsibility for redundant execution of processing, duplication of input and output, also comparison of calculated output.

Wadden et al. (2014) present redundant multithreading (RMT) on OpenCL kernels using automatic compiler transformations that convert GPGPU kernels to have RMT versions. The paper shows performance and power evaluations of several RMT schemes such as Intra-Group RMT, Intra-Group RMT + Local Data Share, and Inter-

Group RMT. They show that compiler-based RMT has significantly variable costs and not only individual components but also several workload properties are responsible for RMT performance.

Mahmoud et al. (2018) implement software-managed instruction duplication (SInRG) for GPU kernel, which is an already explored technique for CPUs, but the authors state that it has never been investigated for GPU before until the paper is written. Since most kernels underutilize the GPU workloads, it prompts authors to work in this field. SInRG is implemented on NVIDIA's production compiler by creating original and shadow register spaces for original and duplicated instructions and shows the duplication overhead is 69%.

Dimitrov, Mantor, and Zhou (2009) propose three different software-based approaches for redundancy. The first approach, R-Naïve executes the GPU kernel twice to ensure temporal redundancy, on the other side spatial redundancy is achieved by copying input and output streams for each execution. Rearranging the input data for cases such as matrix multiplication and using a different input stream compared to the original stream may improve the reliability which is complex and not applicable for all applications. R-Thread approach utilized idle thread blocks to provide redundancy by using the number of thread blocks twice compared to the original execution and executing the same operations on redundant thread blocks. The r-scatter approach is based on instruction-level parallelism. All the proposed solutions are evaluated on six different applications, and it is shown that the result of each approach is application and hardware-dependent.

Oliveira et al. (2014) perform fault injection using radiation experiments with three different duplications with comparison (DWC) techniques: Spatial, E-O Spatial (Even-Odd Spatial), and Time. DWC can be achieved with either block, thread, or execution duplication but block duplication is selected because of implementation simplicity. Authors show that DWC may improve reliability since data is processed at least twice, but still room for SDC improvement because of shared resources. It is shown that the reliability of the DWC technique can be improved together with input data duplication in the Spatial case. According to the measurement result, DWC creates overhead ranging from 90% for Time to 151% for Spatial.

3.2. Fault Injection

Fault injection is a well-known and widely used technique for reliability evaluation and researchers interested in evaluating error resilience of the benchmarks and work on both physical injection like neutron-beam based experiments, or using fault injection tools which offer a variety of options to inject the fault such as single bit flip or random value injection in one register.

Previlon et al. (2020) propose Spoti-FI to accelerate fault injection campaigns via resilience groups. They create resilience groups by using the clustering method just using the profiling data which is captured during the single execution of the program. During the study, only the single-bit flip fault model is evaluated. The accuracy of the Spoti-FI method for reliability is measured using 10K injections as a baseline, via leveraging resilience groups. By using the proposed method, 1317 injections are performed for each application instance on average, and comparison between 10K injections shows that Spoti-FI average error is 1.42% on masked outcomes, 0.88% on DUE outcomes, and 3.92% on SDC outcomes. As a result, they managed to reduce the time to complete the fault injection experiment from 42 days to 5.5 days.

Topçu and Öz (2023) propose regression and classification-based prediction framework soft error vulnerability experiments because fault injection takes so much time to evaluate most of the time. Performance evaluation is done on 23 different applications which are obtained from Rodinia and Polymark benchmarks. Fault-injection experiments are done using cuda-gdb based tool and collect only masked, SDC, and crash results. For simulation, GPGPU-Sim is used, and metric collection is done using the NVIDIA Nsight tool. The proposed solution has 95.91% for masked faults, 88.46% for SDC, and 85.71% for crash rate prediction accuracy.

Defour and Petit (2013) evaluate intermittent errors vulnerability of IC aging because of high temperature on NVIDIA Tesla architecture. In frequency scaling experiments, temperature is increased up to the Thermal Shutdown Protection (TSP) point and observed frequency scaling shows that it is reversible and sets it back to its previous value after the temperature is set lower than TSP. On each newer generation of GPU, the TSP point gets lower indicating that the aging problem is taken seriously by chip manufacturers. For temperature experiments, they set the temperature to 160◦C for NVIDIA Tesla C870 and observed permanent failure of one of the chips after the eleventh day. The same experiment is done with setting the temperature to 170◦C and they observe vectorial and scalar errors on the MAD kernel and did not observe any error on the register bank.

Tselonis and Gizopoulos (2016) present the GUFI (GPGPU-sim Fault Injector) framework that works over the well-known open-source GPU simulator GPGPU-sim. GUFI injects an error into microarchitecture units such as GPU register files, instruction buffer, shared memory, and SIMT stack to measure the Architectural Vulnerability Factor (AVF). AVF is the ratio of the number of fault injections leading to failure and the total number of injections. They discuss error rate differences for fault injection experiments between PTX and SASS instruction sets and show that the proposed framework can be used in the early stages of development to evaluate the performance and reliability of hardware by architects and programmers.

Hari et al. (2017) introduce SASSIFI which is a compiler-based fault injection framework based on an assembly-level SASSI injection mechanism that puts injection at the final phase of SASS code generation by the compiler. SASSIFI is capable of injecting errors in GPU condition code registers, general-purpose registers, GPU memory addresses, and register indices and can be used by several types of studies such as bit-flips into register files and error injections into the output of instructions. SASSIFI workflow starts with the profiling of applications, selection of error injection sites and the last step is to inject the errors and evaluation of injection results. Tool can be used with several architectures, not only limited to a single microarchitecture.

Tsai et al. (2021) introduce the NVBitFI dynamic fault injection tool which is based on NVBit and does not require any source code of the target program to be used on injection. It was stated that a fault injection tool based on cuda-gdb also does not require any source code but cuda-gdb is not intended to be used for fault experiments, it is a debugger and initially not designed for fault injection applications. NVBitFI is built on top of the NVIDIA Binary Instrumentation Tool (NVBit) and supports a wide range of GPU architectures including the Turing and Volta. It works with pre-compiled binaries and works faster than SASSIFI (Hari et al. 2017) since it uses single chosen dynamic kernel whereas SASSIFI uses all dynamic kernels.

Vallero, Gizopoulos, and Di Carlo (2017) present SIFI (Southern Islands Fault Injector) to evaluate soft errors for AMD GPU by experimenting proposed framework over 14 different GPGPU applications and using Architectural Correct Execution (ACE) analysis and fault injection. It is built on top of the Multi2Sim simulator (Ubal et al. 2007).

They evaluate local memory, vector register file, and scalar register file vulnerability and show that fault injection experiment is more accurate than ACE analysis with the cost of simulation time increase, but ACE offers fast evaluation with low accuracy.

Öz and Karadaş (2022) present a regional fault injection framework based on a cuda-gdb debugging tool for the evaluation of soft error vulnerability. The proposed framework includes configuration setup, profiling, fault map generation, fault injection, and collection of results phases. In the configuration setup, the user defines parameters for executable name, arguments, and fault injection information. In the profiling phase, necessary information is collected by using the cuda-gdb debugger tool. In fault map generation, it determines the locations and timing of fault which is specified for corruption of data in the register file. In the fault injection phase, it inserts a breakpoint on the interested line of code and flips the bit, then stores it back in a register file. In the last phase, the collection of results is done considering SDC, Masked, and Crash output of injection.

3.3. Power Modeling and Reliability Evaluation

In this section, we show the related works for power modeling, measurement, and reliability evaluation methods since our performance evaluation metrics include power measurement and reliability evaluation under fault injection.

Nie et al. (2017) evaluate the GPU single-bit error (SBE) for selecting features such as temperature, power consumption, memory utilization, node location, and application execution time on large-scale computer systems and propose a tool to predict several SBE occurrences using neural network techniques. According to measurement results, they achieve higher than 0.69 precision and recall scores for three different neural networks.

Burtscher, Zecena, and Zong (2014) show that using a k20 build-in sensor for power consumption measurement may lead to multiple anomalies such as power consumption may be more than twice when doubling the kernel execution, and power sampling frequency varies when the time goes. They present a proper methodology for measuring instantaneous power and energy consumption.

Aslan and Yilmazer-Metin (2022) propose a true power and energy measurement tool based on built-in sensors using NVIDIA Jetson TX2 GPU and CUDA environment.

They first apply and validate the previous work which is done by power measurement on a k20 built-in sensor (Burtscher, Zecena, and Zong 2014). However, they observe sharp spikes on power measurement graphics and proposed a way to observe a square-shaped power consumption graph by collecting power values every 14 milliseconds periodically, applying a 9-point moving average filter to get a non-sharp power profile with 0.04% difference between measured and corrected values.

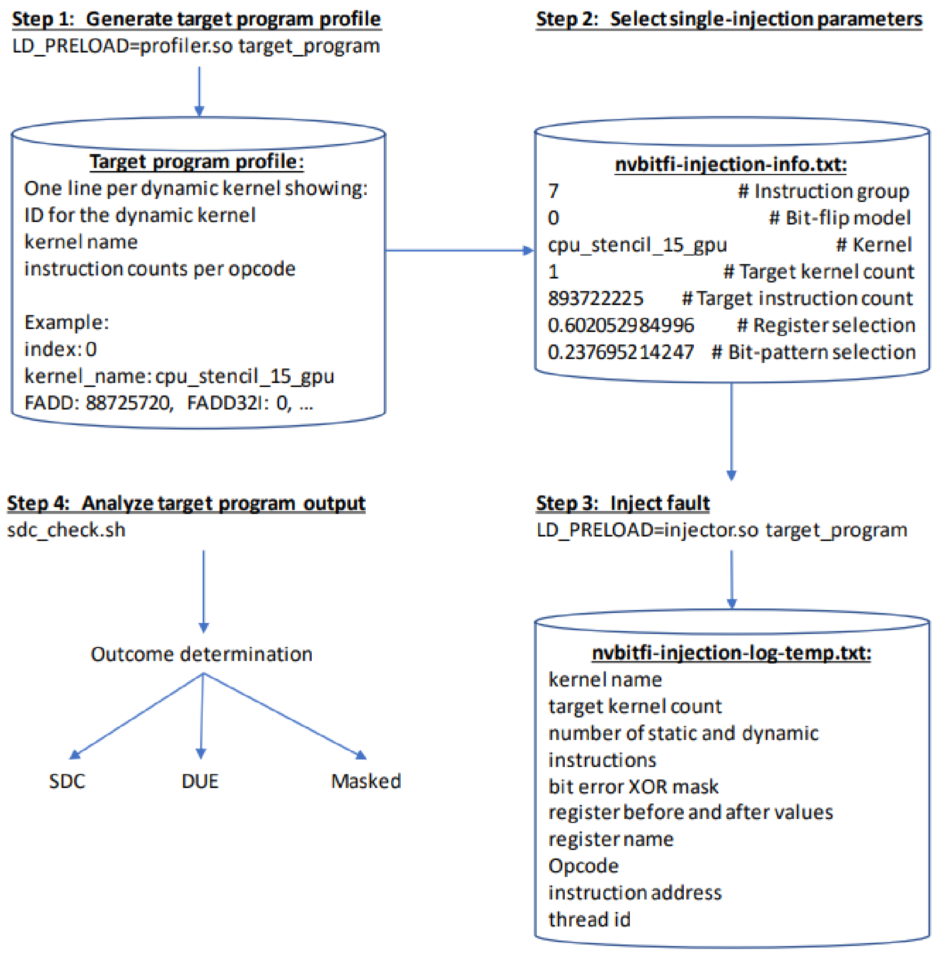
CHAPTER 4

PROPOSED SOLUTION

In this thesis, we explore the reliability and performance analysis of GPU kernels under different spheres of replication for input and output data under various redundancy schemes. We first start with collecting the performance metrics on the original version of the benchmark with selected applications to understand the behavior of each application whether memory or compute bound by collecting various metrics such as achieved occupancy, multiprocessor activity, warp execution efficiency, shared memory efficiency, global memory load efficiency and L2 cache utilization. After gathering the information about benchmark characteristics, we evaluate the soft-error vulnerability of these benchmarks under a single-bit flip fault injection model using the NVBitFI tool and collect masked SDC and DUE rates for each of them. We apply different redundancy schemes perform fault injection experiments collect metrics again and evaluate soft-error vulnerability and performance metrics such as kernel execution time, power consumption, memory copy operations from host to device, and vice versa.

4.1. Fault Injection and Reliability Evaluation

In this thesis, reliability evaluation is performed using a simulation based NVBitFI (NVIDIA Binary Instruction Tool Fault Injector) (Tsai et al. 2021) fault injection tool to evaluate masked, SDC, program crash rates, and resilience of benchmark applications under different redundancy schemes with different spheres of replications. NVBitFI is a state-of-the-art dynamic fault injection tool which is released by NVIDIA and based on NVBit (NVIDIA Binary Instrumentation Tool). It supports a wide range of NVIDIA GPU architecture including recent ones like Volta and Turing, since the target NVIDIA Jetson Xavier NX board is powered GPU with Volta architecture, we selected this fault injection tool because of suitability. The tool does not require any source code access, so makes it easier to use in complex benchmark applications.



Step 1: Generate target program profile
LD_PRELOAD=profiler.so target_program
Target program profile:
One line per dynamic kernel showing:
ID for the dynamic kernel
kernel name
instruction counts per opcode
Example:
index: 0
kernel_name: cpu_stencil_15_gpu
FADD: 88725720, FADD32I: 0, ...
Step 2: Select single-injection parameters
nvbitfi-injection-info.txt:
7 # Instruction group
0 # Bit-flip model
cpu_stencil_15_gpu # Kernel
1 # Target kernel count
893722225 # Target instruction count
0.602052984996 # Register selection
0.237695214247 # Bit-pattern selection
Step 3: Inject fault
LD_PRELOAD=injector.so target_program
nvbitfi-injection-log-temp.txt:
kernel name
target kernel count
number of static and dynamic instructions
bit error XOR mask
register before and after values
register name
Opcode
instruction address
thread id
Step 4: Analyze target program output
sdc_check.sh
Outcome determination
SDC
DUE
Masked

Figure 4.1. NVBitFI Profiling and Transient Fault Injection Procedure (Tsai et al. 2021)

NVBitFI mainly supports the transient fault model that happens in the compute pipeline of the memory read subsystem (Tsai et al. 2021). In the transient fault configuration file, each parameter can be specified such as type of instructions (FP64 and FP32 arithmetic, read from memory, write to registers only, no destination registers, write to general purpose and predicate registers) and bit-flip model (single bit, two adjacent bits, a random value, write value 0), kernel name, kernel count, instruction count, destination register and bit pattern value. It also supports a permanent fault model that affects all the threads in the same SM and hardware lane and only requires SM ID, hardware lane ID, XOR bit mask, and opcode ID as injection parameters. In this thesis, we select a transient fault model with instructions that write to general-purpose registers and single-bit flip as a bit-error pattern.

Execution flow for the transient fault injection procedure given in Figure 4.1, starts with target program profiling by dynamically attaching profiler.so file into execution to set eligible injection points. Profiling is the phase that includes instrumentation of each dynamic instruction which may take a while to finish, so to overcome this, exact and approximate profiling is introduced. The approximate approach only profiles the first invocation of every static kernel and counts dynamic instructions according to that and assumes the rest of the instances will have the same instruction count. In the thesis, exact profiling is used since time overhead is not considered. In the second step, it selects single-injection parameters and then injects the error by changing the target program’s binary and finally compares the execution of the altered fault-injected program with the original target application, named golden output in terminology. Output classification is performed according to possible outcomes given in Table 4.1.

Table 4.1. Outcome Classification (Tsai et al. 2021)

SDC	Standard output is different
	The output file is different
DUE	Timeout, indicating a hang (Monitor detection)
	Process crash (OS detection)
	Non-zero exit status (Application detection)
	Application-specific check failed
Masked	No difference detected
Potential DUE	(SDC or Masked) with CUDA error
	(SDC or Masked) with dmesg error

4.2. Metrics Profiling

Benchmark application profiling is performed using the nvprof build-in tool (NVIDIA Profiler User’s Guide. 2023) which is provided by NVIDIA to profile CUDA kernels to collect information such as execution time for kernel and memory copy operations, global and shared memory usage, cache and streaming multiprocessor usage.

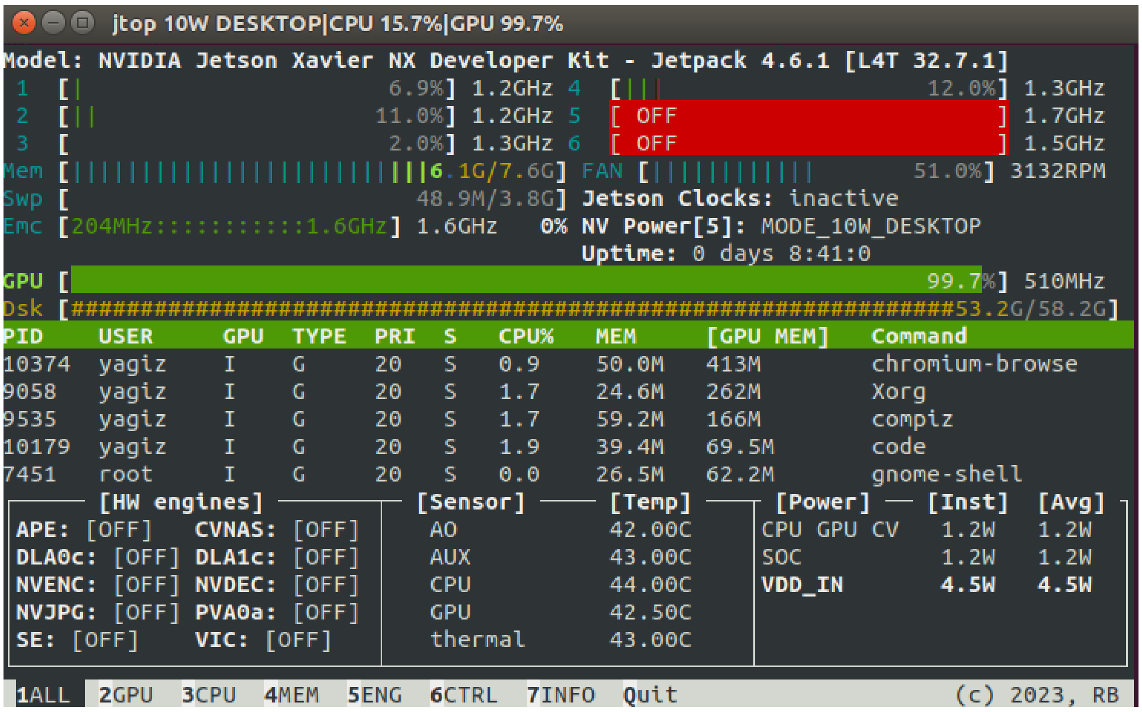
We collect 6 different metrics given in Table 4.2. to understand CUDA kernel behavior and identify whether compute or memory bound. We use sm_efficiency and achieved_occupancy metrics to decide if the kernel is compute-bound. In general, if sm_efficiency is greater than 90%, the kernel can be evaluated as compute bound. For the memory-bound evaluation, gld_efficiency is used, and a 90% threshold can be also applied for this metric. We evaluate sm_efficiency to understand if we utilize GPU's computational resources well. A lower sm_efficiency value usually means that utilization is not enough to take advantage of overall resources which leaves room for further performance improvement in such cases.

Table 4.2. Profiling Metrics and Descriptions (NVIDIA Profiler User's Guide 2023)

Metric	Description
achieved_occupancy	The ratio of the average active warps per active cycle to the maximum number of warps supported on a multiprocessor
sm_efficiency	The percentage of time at least one warp is active on a specific multiprocessor
warp_execution_efficiency	The ratio of the average active threads per warp to the maximum number of threads per warp supported on a multiprocessor
shared_efficiency	The ratio of requested shared memory throughput to required shared memory throughput expressed as a percentage
gld_efficiency	The ratio of the requested global memory load throughput to the required global memory load throughput is expressed as a percentage.
l2_utilization	The utilization level of the L2 cache relative to the peak utilization on a scale of 0 to 10

4.3. Power Measurement

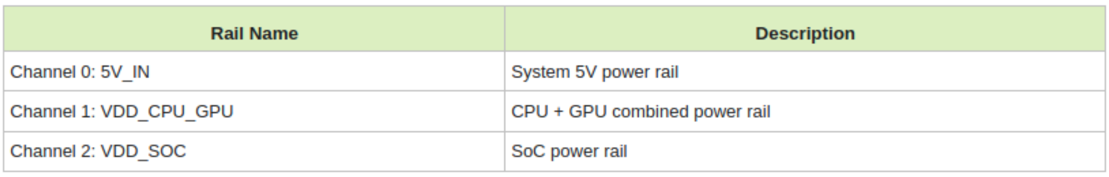
Power consumption is a significant performance parameter because safety-critical systems often work with limited power resources; each reliability improvement has a tradeoff over power, so it needs to be considered carefully. In this thesis, GPU memory usage and power consumption metrics are collected by reading the sysfs node on each 20 ms periodically. There are available packages such as jetson-stats and jtop (NVIDIA Developer's Guide, Jetson Stats. 2023) for monitoring and controlling NVIDIA Jetson products. Jtop uses a sysfs node to collect information from the power monitor chip. jetson-stats and jtop utility uses the same sources to collect information, jtop gives it a visual way.



```
jtop 10W DESKTOP|CPU 15.7%|GPU 99.7%
Model: NVIDIA Jetson Xavier NX Developer Kit - Jetpack 4.6.1 [L4T 32.7.1]
 1  [|                     6.9%] 1.2GHz 4  [||                 12.0%] 1.3GHz
 2  [||                   11.0%] 1.2GHz 5  [ OFF                   ] 1.7GHz
 3  [                      2.0%] 1.3GHz 6  [ OFF                   ] 1.5GHz
Mem [||||||||||||||||||||||||6.1G/7.6G] FAN [||||||||||||      51.0%] 3132RPM
Swp [                       48.9M/3.8G] Jetson Clocks: inactive
Emc [204MHz::::::::::1.6GHz] 1.6GHz    0% NV Power[5]: MODE_10W_DESKTOP
                                         Uptime: 0 days 8:41:0
GPU [                                                              99.7%] 510MHz
Dsk [################################################################53.2G/58.2G]
PID    USER     GPU   TYPE   PRI   S    CPU%   MEM      [GPU MEM]    Command
10374  yagiz    I     G      20    S    0.9    50.0M    413M         chromium-browse
9058   yagiz    I     G      20    S    1.7    24.6M    262M         Xorg
9535   yagiz    I     G      20    S    1.7    59.2M    166M         compiz
10179  yagiz    I     G      20    S    1.9    39.4M    69.5M        code
7451   root     I     G      20    S    0.0    26.5M    62.2M        gnome-shell
      [HW engines]              [Sensor]      [Temp]    [Power]   [Inst]   [Avg]
 APE: [OFF]    CVNAS: [OFF]      AO            42.00C    CPU GPU CV   1.2W    1.2W
 DLA0c: [OFF]  DLA1c: [OFF]      AUX           43.00C    SOC          1.2W    1.2W
 NVENC: [OFF]  NVDEC: [OFF]      CPU           44.00C    VDD_IN       4.5W    4.5W
 NVJPG: [OFF]  PVA0a: [OFF]      GPU           42.50C
 SE: [OFF]     VIC: [OFF]        thermal       43.00C
1ALL  2GPU  3CPU  4MEM  5ENG  6CTRL  7INFO  Quit                    (c) 2023, RB
```

Figure 4.2. jtop utility

Power consumption is measured by using the maximum average power consumption value that is collected during execution and subtracted from the value just collected right before the execution. This measurement was made 100 times, and the average was taken for each case. It is not possible to collect GPU or kernel-specific power consumption, but the total CPU + GPU combined power rail. Since our benchmarks have both CPU and GPU parts to be executed, it is meaningful to collect both CPU and GPU power usage. The Jetson Xavier NX module has one INA3221 power monitor IC at I2C address 0x40. The sysfs nodes to read for rail names, voltage, current, power, and instantaneous and average current limits are given in Figure 4.3.



Rail Name	Description
Channel 0: 5V_IN	System 5V power rail
Channel 1: VDD_CPU_GPU	CPU + GPU combined power rail
Channel 2: VDD_SOC	SoC power rail

Figure 4.3. Rail Names and Descriptions

4.4. Redundancy Implementation

We implement kernel-based triple redundancy with a majority voting function for each case under different spheres of replication for input and output and evaluate serial, stream-based and redundant multithreading techniques' effects on performance and reliability. Typical CUDA program source code is given in Listing 4.1, so we consider this as base implementation without any redundancy, and all the proposed techniques will be added to this baseline. In literature, redundancy implementation mostly starts with doubling the execution units, but in this case, we only manage to detect failure, but we demand to create a fail-operational mechanism. In fail-operational systems, even if the system execution is malformed because of an error, the operation will continue by using redundant systems, which will be crucial for safety-critical systems like autonomous driving (Kohn et al. 2015). In fail-safe systems, in case of an error system will detect the error in fault-tolerant time which may depend on the architecture and requirements, the system will immediately switch to a safe state by resetting the hardware, changing the system mode, closing communication buses, or similar.

```
// Input and output data on the host
float *h_Input, *h_Output;

// Input and output data allocation declaration
float *d_Input, *d_Output;

// Input and output data allocation on the device
cudaMalloc(d_Input, data_size * sizeof(float));
cudaMalloc(d_Output, data_size * sizeof(float));

// Transfer data from host to device
cudaMemcpy(d_Input, h_Input, data_size * sizeof(float), cudaMemcpyHostToDevice);

// Grid and block size configuration
dim3 dimBlock(BLOCK_SIZE_X, BLOCK_SIZE_Y);
dim3 dimGrid(GRID_SIZE_X, GRID_SIZE_Y);

// Kernel execution
kernel<<dimGrid,dimBlock>>(d_Input, d_Output, data_size);

// Transfer data from device to host
cudaMemcpy(h_Output, d_Output, data_size * sizeof(float), cudaMemcpyDeviceToHost);
```

Listing 4.1. Baseline Implementation without Redundancy

In our assumption, since we are injecting one single-bit flip error using a fault injection tool during each execution, only one of the redundant copies will be affected by the error, resulting in SDC. The majority voting function is responsible for finding two identical copies of the input array out of three, returning one of them to be used for further execution assuming only one error may happen during fault injection. If there are no identical arrays found, the *findIdenticalArray()* function returns a null pointer and the caller prints “None of the arrays are identical!” log on a standard error which may be evaluated as a difference between the golden output, so results with SDC. We intentionally put majority voting calculations on the CPU side, since most of the CPUs that are used in safety-critical domains include dual-core lockstep execution in CPUs. After calculation on the GPU device, redundant outputs are transferred from the device to the host and perform majority voting over it.

```
int areArraysIdentical(float *arr1, float *arr2, unsigned int N)
{
   for (unsigned int i = 0; i < N; i++)
   {
      if (fabs(arr1[i] - arr2[i]) > 1e-4)
      {
         return 0; // Arrays are not identical
      }
   }
   return 1; // Arrays are identical
}

float* findIdenticalArray(float *arr1, float *arr2, float *arr3, int N)
{
   if (areArraysIdentical(arr1, arr2, N) || areArraysIdentical(arr2, arr3, N))
   {
      return arr2; // Return the identical array (arr2 in this case)
   }
   else if (areArraysIdentical(arr1, arr3, N))
   {
      return arr1;
   }
   return NULL; // No identical arrays found
}
```

```
float *majorityVotingResult = findIdenticalArray(h_Output, h_Output_redundant_1,
h_Output_redundant_2, size);
if (NULL == majorityVotingResult)
{
   fprintf(stderr, "None of arrays are identical!\n");
}
else
{
   h_Result = majorityVotingResult;
}

// h_Result will be used for the data validation step on the CPU.
```

Listing 4.2. Majority Voting

4.4.1. Full Redundancy for Single-Kernel Benchmarks

In this case, we implement redundancy on benchmarks that have a single CUDA kernel only, since we aim to implement full redundancy, instead of dealing with complex benchmarks that might include several kernels that are hard to adapt redundantly, our first step is to deal benchmarks that have a single kernel to understand the behavior of proposed redundant schemes, after evaluating the most suitable redundancy operation considering performance and reliability, we applied this into complex benchmarks partially to mitigate SDC rate.

4.4.1.1. Serial Triple Redundancy

In this case, we implement a redundancy case by serially executing each kernel three times in a row, including declaration replication, memory allocation on the device, transferring redundant outputs from device to host, performing majority voting, and finding one of the identical copies to be used for output validation. The sphere of replication defines the granularity level of redundancy to decide which part of the system multiplied, either kernel, input, or output. Each sphere of replication brings performance overhead and re-design costs (Portet et al. 2020). In the first case, we redundantly increase output copies and let kernel executions use the same input only, producing different outputs for each execution. In the second case, we include both input and output multiplication for a sphere of replication.

4.4.1.1.a. Serial Triple Redundancy for Output Data Multiplied

In this case, we present serial triple execution for redundancy by just multiplying the output, all the kernels are using the same input source but produce different outputs that will be used by majority voting to detect error-free output.

```
// Input and output data on the host
float *h_Input, *h_Output, *h_Output_redundant_1, *h_Output_redundant_2;

// Input and output data allocation declaration
float *d_Input, *d_Output, *d_Output_redundant_1, *d_Output_redundant_2;

// Input and output data allocation on the device
cudaMalloc(d_Input, data_size * sizeof(float));
cudaMalloc(d_Output, data_size * sizeof(float));
cudaMalloc(d_Output_redundant_1, data_size * sizeof(float));
cudaMalloc(d_Output_redundant_2, data_size * sizeof(float));



// Transfer data from host to device
cudaMemcpy(d_Input, h_Input, data_size * sizeof(float), cudaMemcpyHostToDevice);

// Grid and block size configuration
dim3 dimBlock(BLOCK_SIZE_X, BLOCK_SIZE_Y);
dim3 dimGrid(GRID_SIZE_X, GRID_SIZE_Y);

// Kernel execution
kernel<<dimGrid,dimBlock>>(d_Input, d_Output, data_size);
kernel<<dimGrid,dimBlock>>(d_Input, d_Output_redundant_1, data_size);
kernel<<dimGrid,dimBlock>>(d_Input, d_Output_redundant_2, data_size);

// Transfer data from device to host
cudaMemcpy(h_Output, d_Output, data_size * sizeof(float), cudaMemcpyDeviceToHost);
cudaMemcpy(h_Output_redundant_1, d_Output_redundant_1, data_size * sizeof(float),
cudaMemcpyDeviceToHost);
cudaMemcpy(h_Output_redundant_2, d_Output_redundant_2, data_size * sizeof(float),
cudaMemcpyDeviceToHost);

// Majority voting
```

Listing 4.3. Serial Triple Redundancy for Output Data Multiplied

4.4.1.1.b. Serial Triple Redundancy for Input/Output Data Multiplied

In this case, input multiplication is added in addition to output to observe the behavior of our redundancy schemes if kernels will use a common input source. Using common input resources might affect the cache utilization in a good manner according to our assumptions, so it is expected to see a reduction in execution time.

```
// Input and output data on the host
float *h_Input, *h_Output, *h_Output_redundant_1, *h_Output_redundant_2;

// Input and output data allocation declaration
float *d_Input, *d_Input_redundant_1, *d_Input_redundant_2, *d_Output, *d_Output_redundant_1,
*d_Output_redundant_2;

// Input and output data allocation on the device
cudaMalloc(d_Input, data_size * sizeof(float));
cudaMalloc(d_Input_redundant_1, data_size * sizeof(float));
cudaMalloc(d_Input_redundant_2, data_size * sizeof(float));
cudaMalloc(d_Output, data_size * sizeof(float));
cudaMalloc(d_Output_redundant_1, data_size * sizeof(float));
cudaMalloc(d_Output_redundant_2, data_size * sizeof(float));

// Transfer data from host to device
cudaMemcpy(d_Input, h_Input, data_size * sizeof(float), cudaMemcpyHostToDevice);
cudaMemcpy(d_Input_redundant_1, h_Input, data_size * sizeof(float), cudaMemcpyHostToDevice);
cudaMemcpy(d_Input_redundant_2, h_Input, data_size * sizeof(float), cudaMemcpyHostToDevice);

// Grid and block size configuration
dim3 dimBlock(BLOCK_SIZE_X, BLOCK_SIZE_Y);
dim3 dimGrid(GRID_SIZE_X, GRID_SIZE_Y);
```

```
// Kernel execution
kernel<<dimGrid,dimBlock>>(d_Input, d_Output, data_size);
kernel<<dimGrid,dimBlock>>(d_Input_redundant_1, d_Output_redundant_1, data_size);
kernel<<dimGrid,dimBlock>>(d_Input_redundant_2, d_Output_redundant_2, data_size);

// Transfer data from device to host
cudaMemcpy(h_Output, d_Output, data_size * sizeof(float), cudaMemcpyDeviceToHost);
cudaMemcpy(h_Output_redundant_1, d_Output_redundant_1, data_size * sizeof(float),
cudaMemcpyDeviceToHost);
cudaMemcpy(h_Output_redundant_2, d_Output_redundant_2, data_size * sizeof(float),
cudaMemcpyDeviceToHost);

// Majority voting
```

Listing 4.4. Serial Triple Redundancy for Input/Output Data Multiplied

4.4.1.2. Stream-Based Triple Redundancy

CUDA streams are introduced to handle independent kernel executions in parallel by assigning each kernel to a different stream. In the serial redundant execution case, since we serially execute the kernels, we don’t get the benefit of the highly parallel architecture of GPU, so it is expected to see a reduction in the execution time of kernels by overlapping the kernel executions. In this case, in addition to the baseline of implementation, we create three different CUDA streams before the kernel execution assign each redundant execution to different streams, and evaluate both input and input/output multiplied cases.

4.4.1.2.a. Stream-Based Triple Redundancy for Output Data Multiplied

In this case, we present a triple redundancy scheme by assigning each kernel execution to different CUDA streams to get the benefit of parallelization. Each redundant kernel uses the same input source but produces different outputs.

```
// Input and output data on the host
float *h_Input, *h_Output, *h_Output_redundant_1, *h_Output_redundant_2;
```

```
// Input and output data allocation declaration
float *d_Input, *d_Output, *d_Output_redundant_1, *d_Output_redundant_2;

// Input and output data allocation on the device
cudaMalloc(d_Input, data_size * sizeof(float));
cudaMalloc(d_Output, data_size * sizeof(float));
cudaMalloc(d_Output_redundant_1, data_size * sizeof(float));
cudaMalloc(d_Output_redundant_2, data_size * sizeof(float));

// Transfer data from host to device
cudaMemcpy(d_Input, h_Input, data_size * sizeof(float), cudaMemcpyHostToDevice);

// Grid and block size configuration
dim3 dimBlock(BLOCK_SIZE_X, BLOCK_SIZE_Y);
dim3 dimGrid(GRID_SIZE_X, GRID_SIZE_Y);

// Stream create
cudaStream_t stream[3];
cudaStreamCreate(&stream[0]);
cudaStreamCreate(&stream[1]);
cudaStreamCreate(&stream[2]);

// Kernel execution
kernel<<dimGrid,dimBlock, 0, stream[0]>>(d_Input, d_Output, data_size);
kernel<<dimGrid,dimBlock, 0, stream[1]>>(d_Input, d_Output_redundant_1, data_size);
kernel<<dimGrid,dimBlock, 0, stream[2]>>(d_Input, d_Output_redundant_2, data_size);

// Transfer data from device to host
cudaMemcpy(h_Output, d_Output, data_size * sizeof(float), cudaMemcpyDeviceToHost);
cudaMemcpy(h_Output_redundant_1, d_Output_redundant_1, data_size * sizeof(float),
cudaMemcpyDeviceToHost);
cudaMemcpy(h_Output_redundant_2, d_Output_redundant_2, data_size * sizeof(float),
cudaMemcpyDeviceToHost);

// Majority voting
```
Listing 4.5. Stream-Based Triple Redundancy for Output Data Multiplied

4.4.1.2.b. Stream-Based Triple Redundancy for Input/Output Data Multiplied

In addition to the implementation of stream-based triple redundancy for output data multiplied, we add input multiplication to observe the effects on the usage of different input resources.

```
// Input and output data on the host
float *h_Input, *h_Output, *h_Output_redundant_1, *h_Output_redundant_2;

// Input and output data allocation declaration
float *d_Input, *d_Input_redundant_1, *d_Input_redundant_2, *d_Output, *d_Output_redundant_1,
*d_Output_redundant_2;

// Input and output data allocation on the device
cudaMalloc(d_Input, data_size * sizeof(float));
cudaMalloc(d_Input_redundant_1, data_size * sizeof(float));
cudaMalloc(d_Input_redundant_2, data_size * sizeof(float));
cudaMalloc(d_Output, data_size * sizeof(float));
cudaMalloc(d_Output_redundant_1, data_size * sizeof(float));
cudaMalloc(d_Output_redundant_2, data_size * sizeof(float));

// Transfer data from host to device
cudaMemcpy(d_Input, h_Input, data_size * sizeof(float), cudaMemcpyHostToDevice);
cudaMemcpy(d_Input_redundant_1, h_Input, data_size * sizeof(float), cudaMemcpyHostToDevice);
cudaMemcpy(d_Input_redundant_2, h_Input, data_size * sizeof(float), cudaMemcpyHostToDevice);

// Grid and block size configuration
dim3 dimBlock(BLOCK_SIZE_X, BLOCK_SIZE_Y);
dim3 dimGrid(GRID_SIZE_X, GRID_SIZE_Y);

// Stream create
cudaStream_t stream[3];
cudaStreamCreate(&stream[0]);
cudaStreamCreate(&stream[1]);
cudaStreamCreate(&stream[2]);

// Kernel execution
```

```
kernel<<dimGrid,dimBlock, 0, stream[0]>>(d_Input, d_Output, data_size);
kernel<<dimGrid,dimBlock, 0, stream[1]>>(d_Input_redundant_1, d_Output_redundant_1, data_size);
kernel<<dimGrid,dimBlock, 0, stream[2]>>(d_Input_redundant_2, d_Output_redundant_2, data_size);

// Transfer data from device to host
cudaMemcpy(h_Output, d_Output, data_size * sizeof(float), cudaMemcpyDeviceToHost);
cudaMemcpy(h_Output_redundant_1, d_Output_redundant_1, data_size * sizeof(float),
cudaMemcpyDeviceToHost);
cudaMemcpy(h_Output_redundant_2, d_Output_redundant_2, data_size * sizeof(float),
cudaMemcpyDeviceToHost);

// Majority voting
```

Listing 4.6. Stream-Based Triple Redundancy for Input/Output Data Multiplied

4.4.1.3. Redundant-Multithread-Based Redundancy

Kernel multiplication-based redundancy techniques improve the reliability of the system using serial, stream-based with different spheres of replication without requiring any huge effort on code implementation. However, it is shown that it may increase the cost of launch overhead. (Zhan et al. 2019). Multiple kernel execution-based redundancy techniques may not benefit from the highly parallel nature of GPU hardware, which might leave an open door for further improvement even if CUDA streams are used, since stream creation may cause additional overhead. The redundant multithread-based approach aims at the idea of multiplying the number of threads according to the level of redundancy, so each redundant thread will be responsible for the same calculation redundantly to increase the reliability of the overall system. This approach can be done in several ways, either multiplying the number of threads in blocks while keeping the threads per block the same or multiplying the number of threads inside blocks while keeping the number of blocks the same. Since grid or block configurations are multi-dimensional, it can be done either using the X or Y axis. We evaluate both X and Y-axis threads and block multiplication-based redundancy with different spheres of replication like input only or both input and output multiplied. Without any redundant version, thread ID calculation in the X and Y axis is performed as given in Listing 4.7.

```
unsigned int threadIdX = blockIdx.x * blockDim.x + threadIdx.x;
unsigned int threadIdY = blockIdx.y * blockDim.y + threadIdx.y;
```

Listing 4.7. Original ThreadID – X and Y Calculation

For the thread-based multiplication, since we have 3x threads compared to the original version, thread ID calculation is replaced with Listing 4.8 and 4.9 for Y and X-axis-based schemes, respectively.

```
unsigned int threadIdY = blockIdx.y * BLOCK_SIZE_Y + (threadIdx.y % BLOCK_SIZE_Y);
```

Listing 4.8. Modified ThreadID – Y Calculation for Thread-Based Redundant Multithreading

```
unsigned int threadIdX = blockIdx.x * BLOCK_SIZE_X + (threadIdx.x % BLOCK_SIZE_X);
```

Listing 4.9. Modified ThreadID – X Calculation for Thread-Based Redundant Multithreading

```
unsigned int threadIdY = (blockIdx.y % GRID_SIZE_Y) * blockDim.y + threadIdx.y;
```

Listing 4.10. Modified ThreadID – Y Calculation for Block-Based Redundant Multithreading

```
unsigned int threadIdX = (blockIdx.x % GRID_SIZE_X) * blockDim.x + threadIdx.x;
```

Listing 4.11. Modified ThreadID – X Calculation for Block-Based Redundant Multithreading

In ThreadID calculation, BLOCK_SIZE_X and BLOCK_SIZE_Y are the initial block size, and GRID_SIZE_X and GRID_SIZE_Y are the initial grid size in a version without redundancy depending on the X and Y axis, respectively. Thread ID manipulation is performed differently, so if X-axis-based redundancy is used, only threadIdX is altered, the same rule will also apply to threadIdY. Redundant threads are dealing with the same computation, but the result will be stored on different output. The signature of the kernel in a version without redundancy is given in Listing 4.12 which needs to be replaced with Listing 4.13 in the RMT scheme.

```
__global__ kernel(d_Input, d_Output, data_size)
```

Listing 4.12. Original Kernel Signature

The original kernel signature is changed to store redundant calculations in different outputs with the following:

```
__global__ kernel(d_Input, d_Output, d_Output_redundant_1, d_Output_redundant_2, data_size)
```

Listing 4.13. Modified Kernel Signature for Redundant Multithreading

Since each redundant thread will produce a different output, the result calculation in the kernel is changed to handle more than one output according to the thread or block index. If a thread-based RMT scheme is selected, output redirection is performed according to Listing 4.14.

```
int thread_index = threadIdx.y; // Use threadIdx.y for Y-axis thread-based redundancy
int thread_index = threadIdx.x; // Use threadIdx.x for X-axis thread-based redundancy

// Replace BLOCK_SIZE with BLOCK_SIZE_X or BLOCK_SIZE_Y depending on axis selection
// Use  X-axis => BLOCK_SIZE_X
// Use  Y-axis => BLOCK_SIZE_Y

if (thread_index < BLOCK_SIZE)
{
   d_Output[idx] = result;
}
else if (thread_index < (BLOCK_SIZE * 2))
{
   d_Output_redundant_1[idx] = result;
}
else
{
   d_Output_redundant_2[idx] = result;
}
```

Listing 4.14. Modified Output Direction for Thread-Based Redundant Multithreading

For block-based RMT output direction, changes in Listing 4.15 are performed.

```
int block_index = blockIdx.y; // Use blockIdx.y for Y-axis block-based redundancy
int block_index = blockIdx.x; // Use blockIdx.x for X-axis block-based redundancy

// Replace GRID_SIZE with GRID_SIZE_X or GRID_SIZE_Y depending on axis selection
// Use  X-axis => GRID_SIZE_X
// Use  Y-axis => GRID_SIZE_Y

if (block_index < GRID_SIZE)
{
   d_Output[idx] = result;
}
else if (block_index < (GRID_SIZE * 2))
{
   d_Output_redundant_1[idx] = result;
}
else
{
   d_Output_redundant_2[idx] = result;
}
```

Listing 4.15. Modified Output Direction for Block-Based Redundant Multithreading

4.4.2. Partial Redundancy for Multiple-Kernel Benchmarks

Most of the safety-critical applications may include more than one kernel since it includes several operations together. We first evaluate our proposed redundancy schemes with single-kernel benchmarks due to ease of applicability. For the complex applications, instead of applying redundancy for each kernel, we propose partial redundancy for one or more kernels inside complex applications to reduce the SDC rate as much as possible by selecting most SDC-vulnerable kernels after evaluating the soft error vulnerability of overall complex benchmark. We propose to apply redundant multithreading with thread and block multiplication-based redundancy methods for both the X and Y axes based, then evaluate the performance and reliability of the proposed techniques.

CHAPTER 5

EXPERIMENTAL RESULTS

In this chapter, we provide experimental setup details including selected hardware and benchmark, compilation parameters, and discuss the experimental results including execution time, memory copy operations, soft-error vulnerability, and performance metrics.

5.1. Experimental Setup

Reliability and performance tradeoff analysis is crucial in resource-limited and safety-critical systems, due to this fact we select the NVIDIA Jetson Xavier NX Developer Kit board given in Figure 5.1 which provides up to 21 TOPS of compute performance powered with NVIDIA Volta architecture with 384 NVIDIA CUDA cores given in Table 5.1, 48 Tensor core and 2 NVDLA (NVIDIA Deep Learning Accelerator), combined with 8GB of LPDDR4x RAM. For the target developer kit, the operating system version is selected as Jetpack 4.6.1 [L4T 32.7.1] which is provided by NVIDIA and built on top of the Ubuntu distribution. We compile our applications with CUDA 10.2 and use the nvcc compiler which is provided inside the CUDA toolchain.

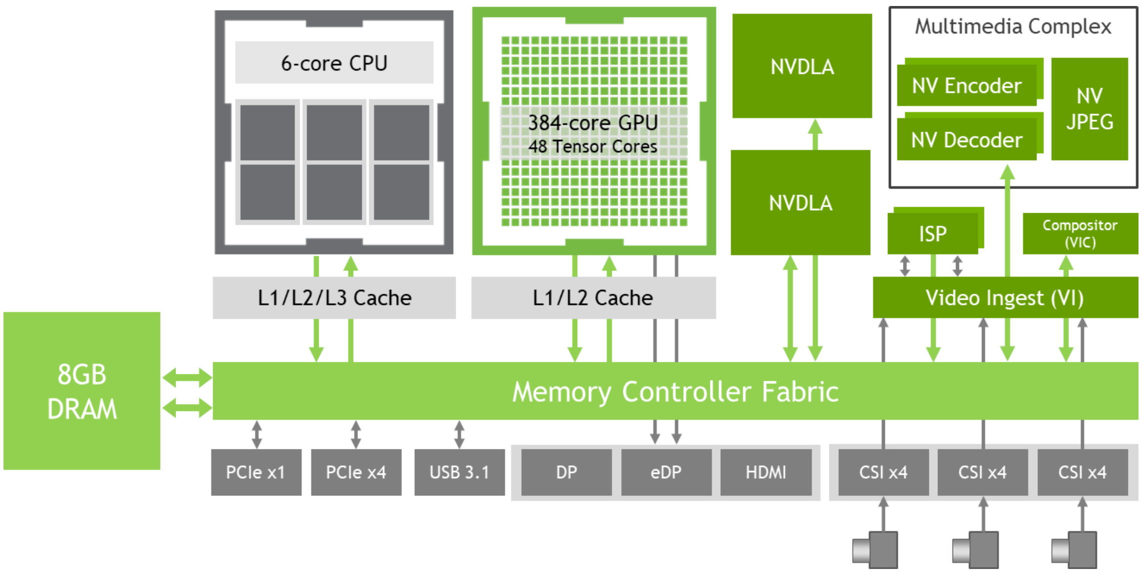


Figure 5.1. NVIDIA Board Block Diagram (NVIDIA Technical Blog. 2019)

Table 5.1. NVIDIA Jetson Xavier NX Properties

	NVIDIA Jetson Xavier NX
CUDA Capability Major/Minor version number	7.2
Total amount of global memory	7765 MBytes (8142626816 bytes)
Multiprocessors	6
CUDA Cores/Multiprocessors	64
GPU Max Clock rate	1109 MHz (1.11 GHz)
Memory Clock rate	1109 MHz
Memory Bus Width	256-bit
L2 Cache Size	524288 bytes
Total number of registers available per block	65536
Maximum number of threads per multiprocessor	2048
Maximum number of threads per block	1024
The max dimension size of a thread block (x,y,z)	(1024, 1024, 64)
Max dimension size of a grid size (x,y,z)	(2147483647, 65535, 65535)

We use GPU4S (GPU for Space) benchmark suite (Kosmidis et al. 2019) for performance and reliability evaluations on the proposed framework, selecting 5 different applications (*convolution_2D_bench*, *matrix_multiplication_bench*, *relu_bench*, *wavelet_transform*, *max_pooling*) which have one CUDA kernel only, and one complex application (*cifar_10*) that include several kernels to evaluate serial, stream-based and redundant multithreading schemes. GPU4S is intended to be used by space applications but it would be a good candidate for safety-critical domains since it is like computation algorithms in applications and can be used for reliability evaluations (Perez-Cerrolaza et al. 2022).

Table 5.2. Benchmark Applications and Domains

Domain	**Application**	**Definition**
Convolution Kernel	convolution_2D_bench	Matrix Convolution
Matrix Computation	matrix_multiplication_bench	Matrix Multiplication
Neural Network	relu_bench	Rectified Linear Unit
Discrete Wavelet Transform	wavelet_transform	Wavelet Transform
Neural Network	max_pooling	Pooling Operation
Neural Network	cifar_10	Complex Application

For the benchmark compilation, according to GPU4s documentation, the user should provide the block size using a square of the size that is provided, so the recommended values are 4, 8, 16, and 32. We selected block size as 16, and data type as float and used the command given in Listing 5.1 to build each application.

Listing 5.1. Benchmark Build Options

```
make CUDA DATATYPE=float BLOCKSIZE=16
```

For execution and memory operations time evaluation, we use CUDA events to measure by placing them at the beginning and end of the unit (Listing 5.2).

Listing 5.2. CUDA Event Time Measurement

```
cudaEvent_t start_memory_copy_device;
cudaEvent_t stop_memory_copy_device;
cudaEvent_t start_memory_copy_host;
cudaEvent_t stop_memory_copy_host;
cudaEvent_t start_kernel_execution;
cudaEvent_t stop_kernel_execution;

cudaEventRecord(start_memory_copy_device);
// Memory copy from host to device
cudaEventRecord(stop_memory_copy_device);

cudaEventRecord(start_kernel_execution);
// Kernel execution
cudaEventRecord(stop_kernel_execution);

cudaEventRecord(start_memory_copy_host);
// Memory copy from device to host
```

```
cudaEventRecord(stop_memory_copy_host);
```

5.2. Experimental Results

For the evaluation of performance and reliability, before applying any of the redundancy techniques with different spheres of replication, we first start with baseline implementation of selected benchmark applications to analyze characteristics of execution and resilience of soft-error vulnerabilities by performing profiling using nvprof and fault injection with NVBitFI tool. For the fault injection model, we use a single bit-flip as one bit-flip in one register in one thread model and select several injections as 10000.

For the full redundancy techniques, we only use single-kernel benchmarks for evaluation since complex benchmarks might include several kernels that need to be redundantly executed, causing significant performance overhead. Due to this, instead of applying for full redundancy, according to fault injection results, we select the two most error-vulnerable and one least kernel and apply block and thread multiplication-based redundant multithreading. We intentionally do not add original execution results for normalized comparisons since the baseline unit is always one.

5.2.1. Single-Kernel Benchmark Results

For the comparison convenience, we use different cases that match with offered redundancy techniques and spheres of replication.

Table 5.3. Single-Kernel Benchmark Experiment Cases

Case 1	Original execution, without any redundancy
Case 2	Serial triple redundancy, input, and output multiplied
Case 3	Serial triple redundancy, output multiplied
Case 4	Stream-based triple redundancy, input and output multiplied
Case 5	Stream-based triple redundancy, output multiplied
Case 6	Redundant multithreading, thread Y-axis-based multiplication
Case 7	Redundant multithreading, thread X-axis-based multiplication
Case 8	Redundant multithreading, block Y-axis-based multiplication
Case 9	Redundant multithreading, block X-axis-based multiplication

According to nvprof measurement results, for all the kernels warp_execution_efficiency and shared_efficiency metrics are measured as the same value, so we exclude these metrics from the table. For all the kernels, warp_execution_efficiency is measured as 100.00%, if this is less than 100% then the kernel has either thread divergence or the kernel was not launched with a multiple of 32 threads per block. For the shared_efficiency metric, measured as 0.00% which means that none of the kernels are using shared memory.

Table 5.4. Metrics Profiling for Single-Kernel Benchmarks

	achieved_occupancy	sm_efficiency	gld_efficiency	l2_utilization
convolution_2D_bench	0.887312	94.57%	21.01%	Low (1)
max_pooling_bench	0.915331	97.74%	25.00%	Low (2)
matrix_multiplication_bench	0.93599	98.96%	13.24%	Low (1)
wavelet_transform	0.124184	15.22%	42.43%	Low (1)
relu_bench	0.84969	98.88%	25.00%	Low (3)

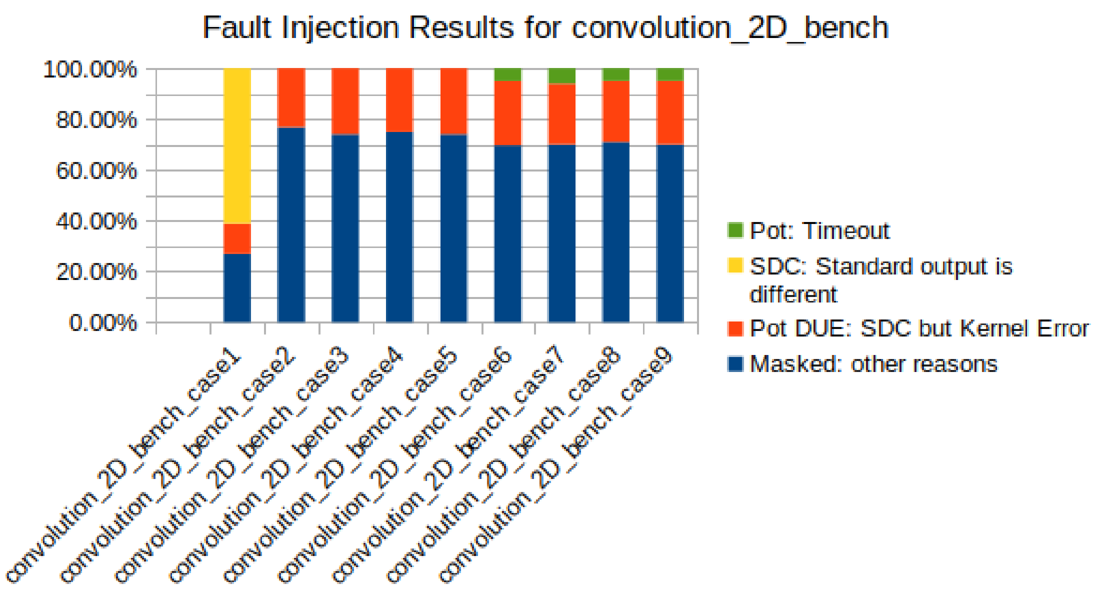


Figure 5.2. Fault Injection Results for *convolution_2D_bench*

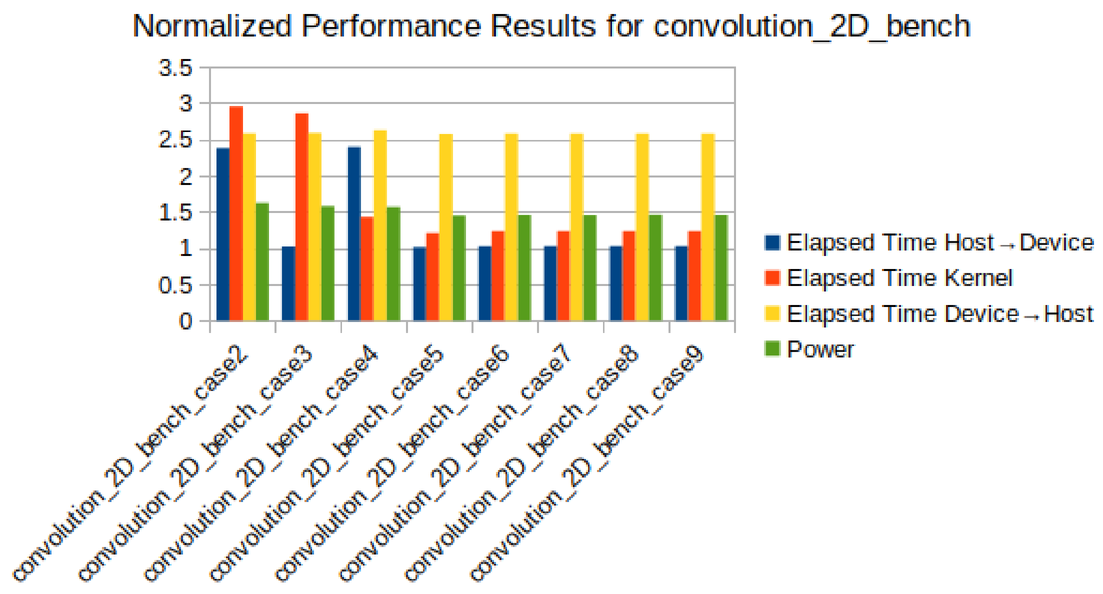


Figure 5.3. Normalized Performance Results for *convolution_2D_bench*

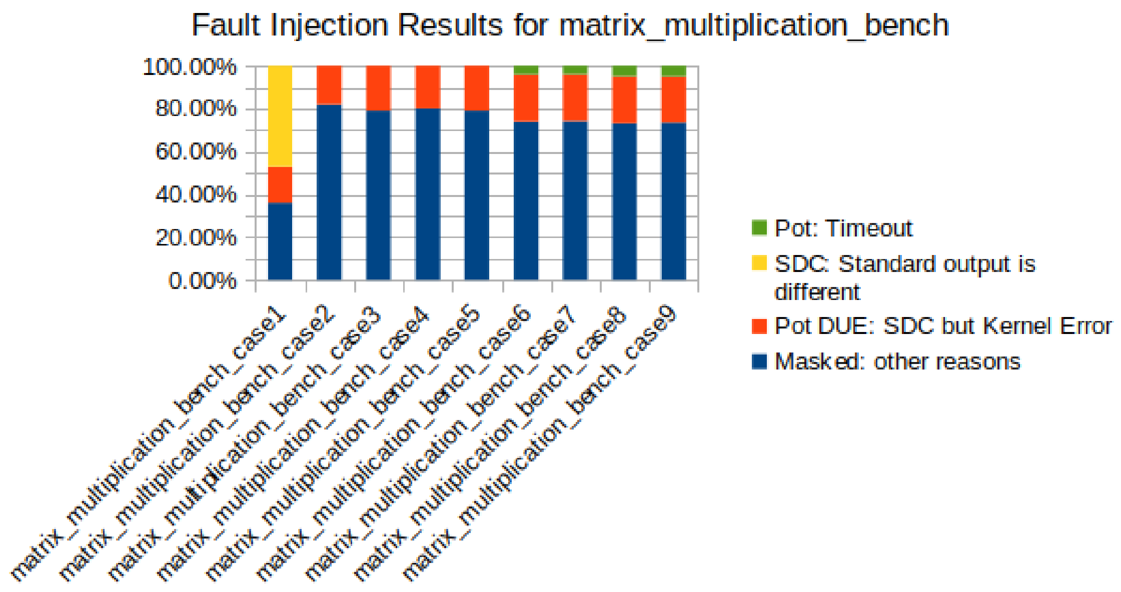


Figure 5.4. Fault Injection Results for *matrix_multiplication_bench*

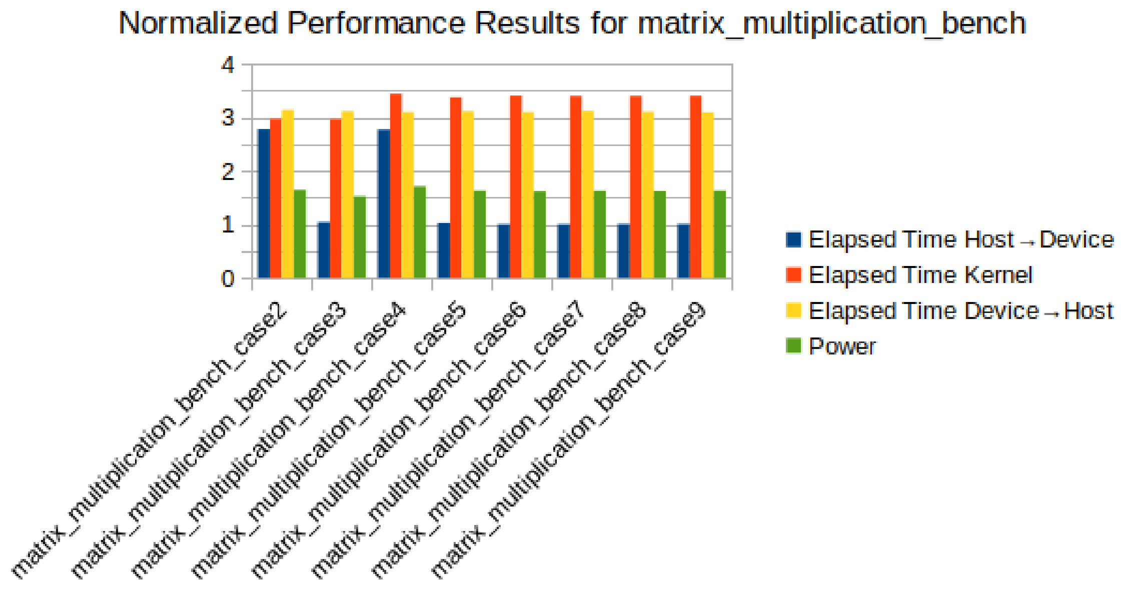


Figure 5.5. Normalized Performance Results for *matrix_multiplication_bench*

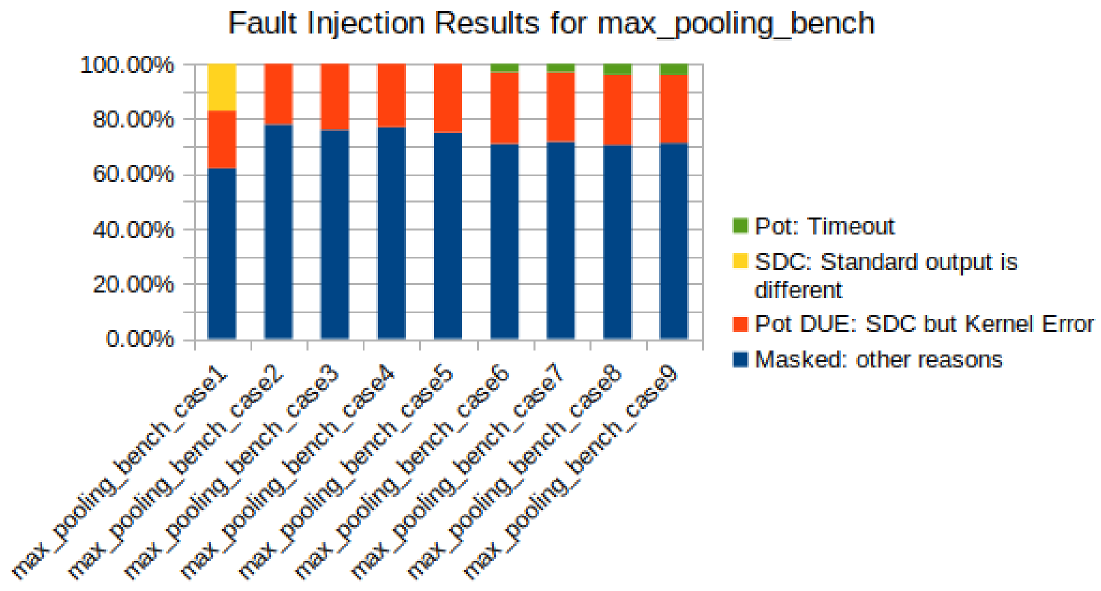


Figure 5.6. Fault Injection Results for *max_pooling_bench*

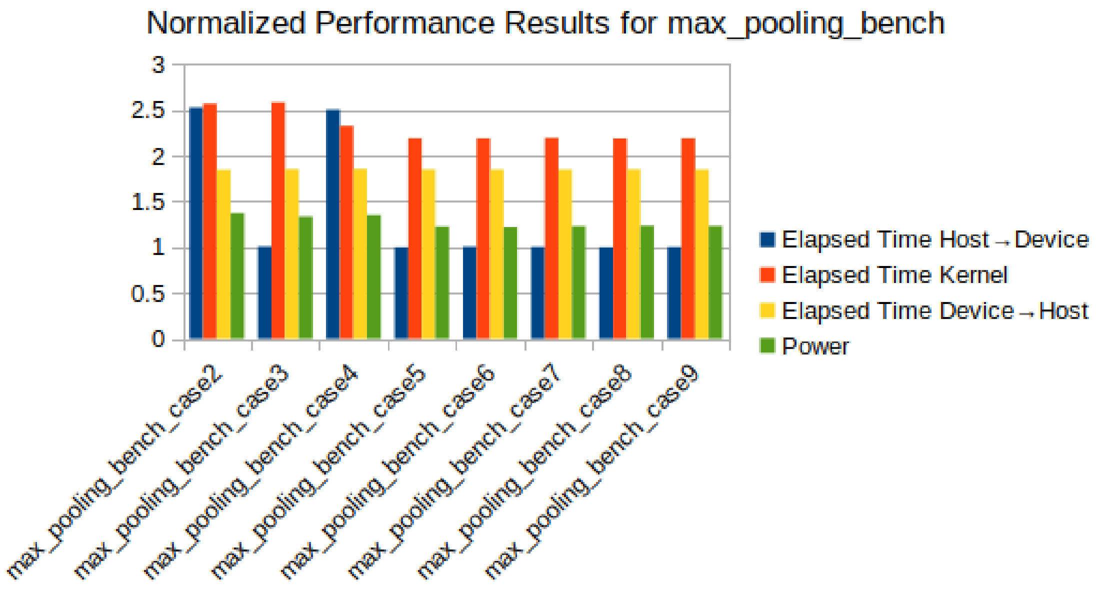


Figure 5.7. Normalized Performance Results for *max_pooling_bench*

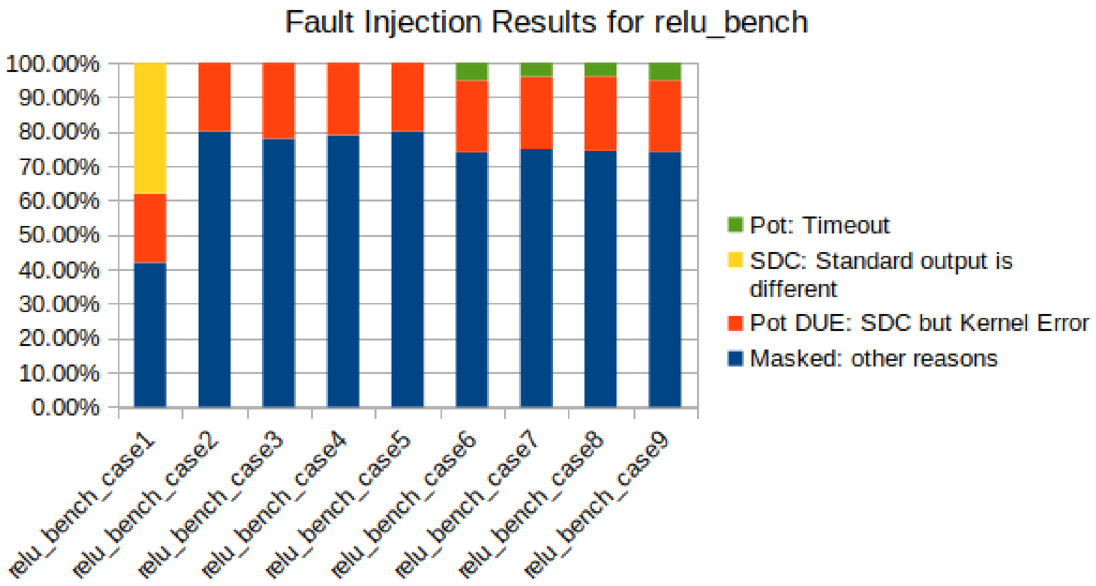


Figure 5.8. Fault Injection Results for *relu_bench*

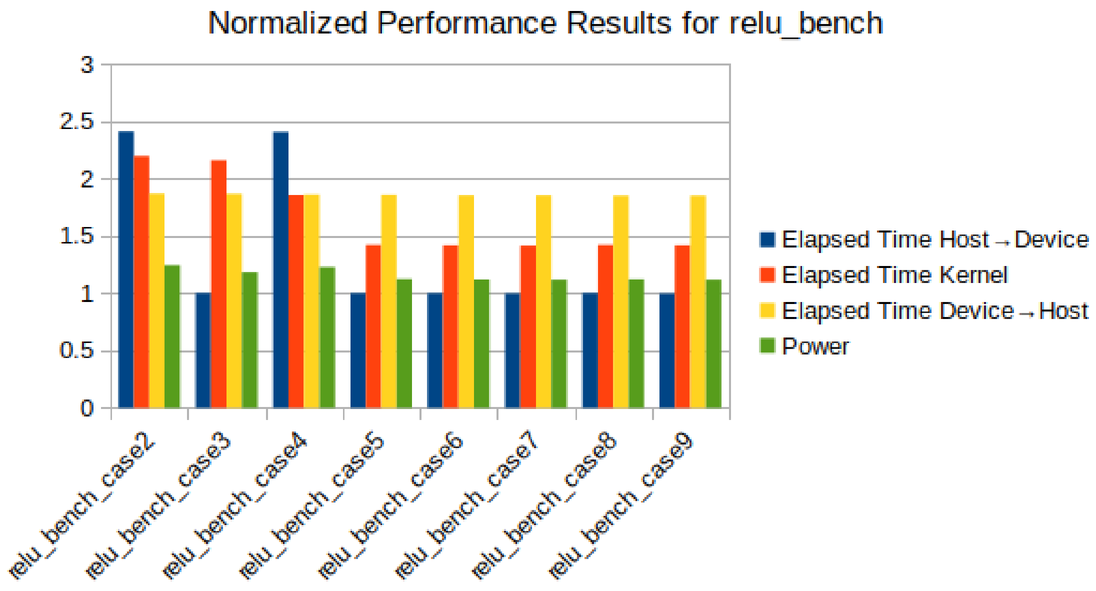


Figure 5.9. Normalized Performance Results for *relu_bench*

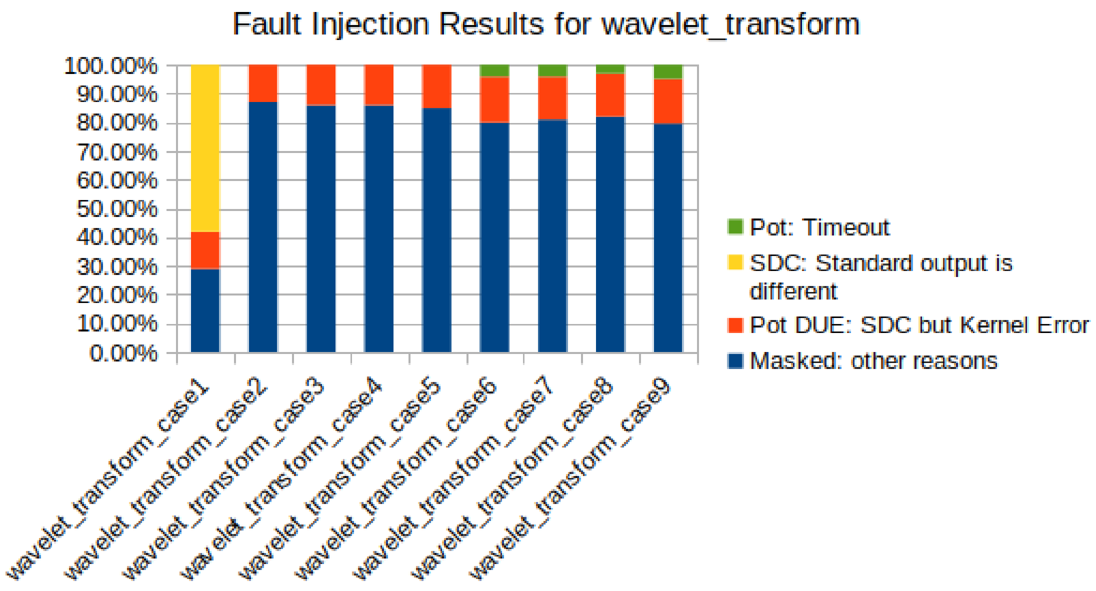


Figure 5.10. Fault Injection Results for *wavelet_transform*

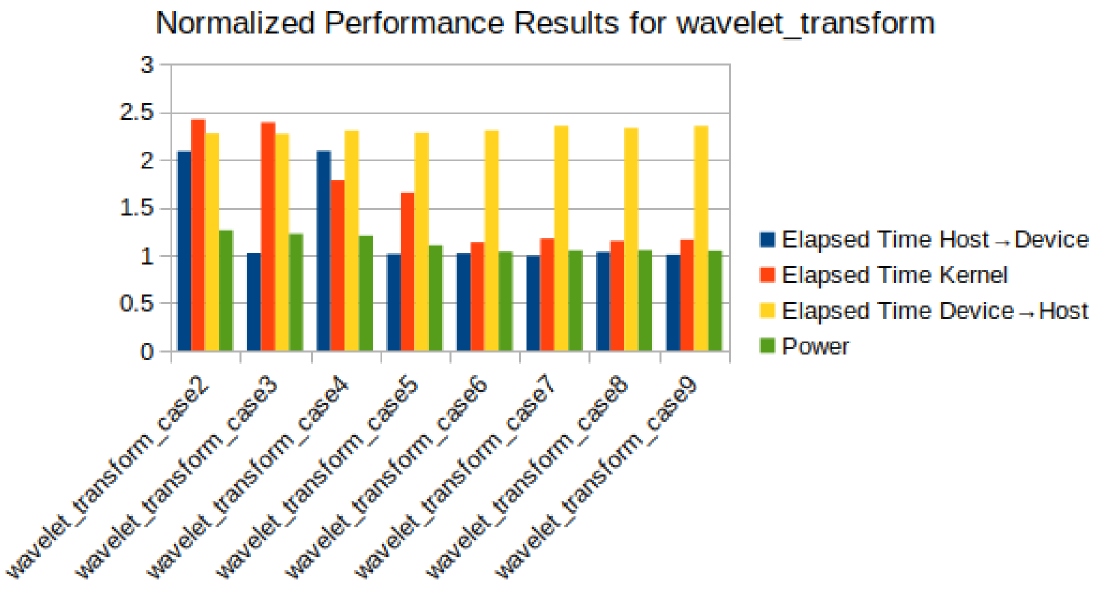


Figure 5.11. Normalized Performance Results for *wavelet_transform*

For all the cases in single-benchmark applications, we eliminate the SDCs by offering triple redundancy with proposed techniques and spheres of replication. Since triple redundancy and majority voting aim to eliminate SDCs, according to measurement results this target is achieved.

For memory copy between host to device, it takes less than 3 times probably because of buffering effect compared to baseline for cases 2 and 4 where input is multiplied, almost 2.5 times for both *convolution_2D_bench* (Figure 5.3), *matrix_multiplication_bench* (Figure 5.5), *max_pooling_bench* (Figure 5.7), *relu_bench* (Figure 5.9), 2 times for *wavelet_transform* (Figure 5.11). Even if the triple replication of input, it is observed that the copy operation does not take 3 times, which can be interpreted as using triple replication would be advantageous.

For serial triple execution in cases 2 and 3, kernel execution takes almost 3 times for *convolution_2D_bench* (Figure 5.3), *matrix_multiplication_bench* (Figure 5.5), 2.5 times for *max_pooling_bench* (Figure 5.7), slightly higher than 2 times for *relu_bench* (Figure 5.9) and *wavelet_transform* (Figure 5.11). In serial execution, it is not possible to get the benefit of the highly parallel architecture of GPUs, so utilization of all resources is possible using redundancy techniques like stream and RMT-based multithreading. For *convolution_2D_bench* (Figure 5.3), due to CUDA stream creation overhead, kernel execution times are slightly higher than RMT cases, but compared to serial execution both stream and RMT-based cases offer dramatic improvement. For *matrix_multiplication_bench* (Figure 5.5), kernel execution time slightly increases more than 3 times for stream and RMT-based cases. In all cases except *matrix_multiplication_bench* kernel, RMT offers the best performance among all redundancy techniques, after this stream-based offers better performance compared to serial execution but due to stream creation overhead, RMT offers better performance than stream-based.

For memory copy between the device and to host, since all the techniques include output multiplication, measured time is consistent among cases for the same kernel. Except for *matrix_multiplication_bench* (Figure 5.5), overhead is less than 3 times and even better for *relu_bench* (Figure 5.9), which is almost 1.5 times compared to the baseline.

For power measurement, it is not possible to measure GPU kernel power consumption individually, so both CPU and GPU power consumption are measured which includes both measurements of CPU operations such as scheduling, context switching, GPU kernel offloading, and memory operations. Proposed redundancy techniques are performed on the GPU side except for majority voting for evaluation of error-free output on the CPU side. In both kernels and cases, power consumption never reaches 2 times compared to baseline, we evaluate this result as CPU operations consume more power than GPU operations even if we apply triple redundancy with different techniques and spheres of replications.

5.2.2. Complex Benchmark Results

In the *cifar_10* complex benchmark, we only evaluate redundant multithreading-based redundancy since according to our evaluation results on single-kernel benchmarks, it offers the best performance for all the cases. We first start with the fault injection and analyze the most and least error-vulnerable kernels considering the SDC rate and measure execution times for each kernel to understand if there is a correlation between them. After evaluating most and least error-vulnerable kernels, we apply both block and thread-based RMT with X and Y axes.

Table 5.5. Execution Cases for *cifar_10*

Case 1	Original execution, without any redundancy
Case 2	RMT Y-Axis Thread Based, matrix_multiplication kernel
Case 3	RMT X-Axis Thread Based, matrix_multiplication kernel
Case 4	RMT Y-Axis Block Based, matrix_multiplication kernel
Case 5	RMT X-Axis Block Based, matrix_multiplication kernel
Case 6	RMT Y-Axis Thread Based, matrix_multiplication and covolution_kernel kernels
Case 7	RMT X-Axis Thread Based, matrix_multiplication and covolution_kernel kernels
Case 8	RMT Y-Axis Block Based, matrix_multiplication and covolution_kernel kernels
Case 9	RMT Y-Axis Block Based, matrix_multiplication and covolution_kernel kernels
Case 10	RMT Y-Axis Thread Based, softmax_kernel kernel
Case 11	RMT X-Axis Thread Based, softmax_kernel kernel
Case 12	RMT Y-Axis Block Based, softmax_kernel kernel
Case 13	RMT X-Axis Block Based, softmax_kernel kernel

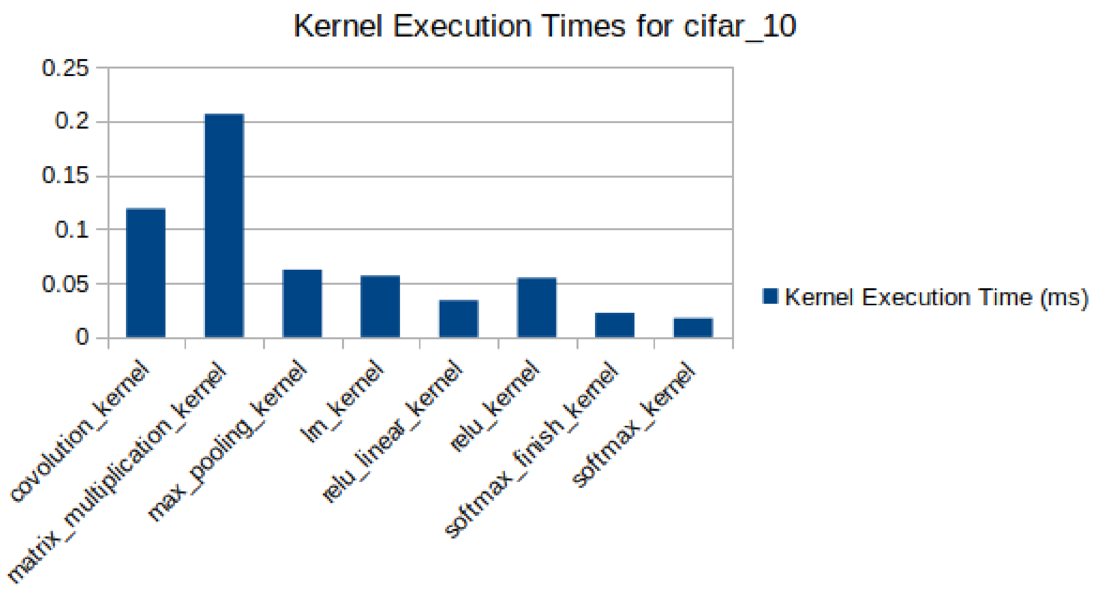


Figure 5.12. Kernel Execution Times for *cifar_10*

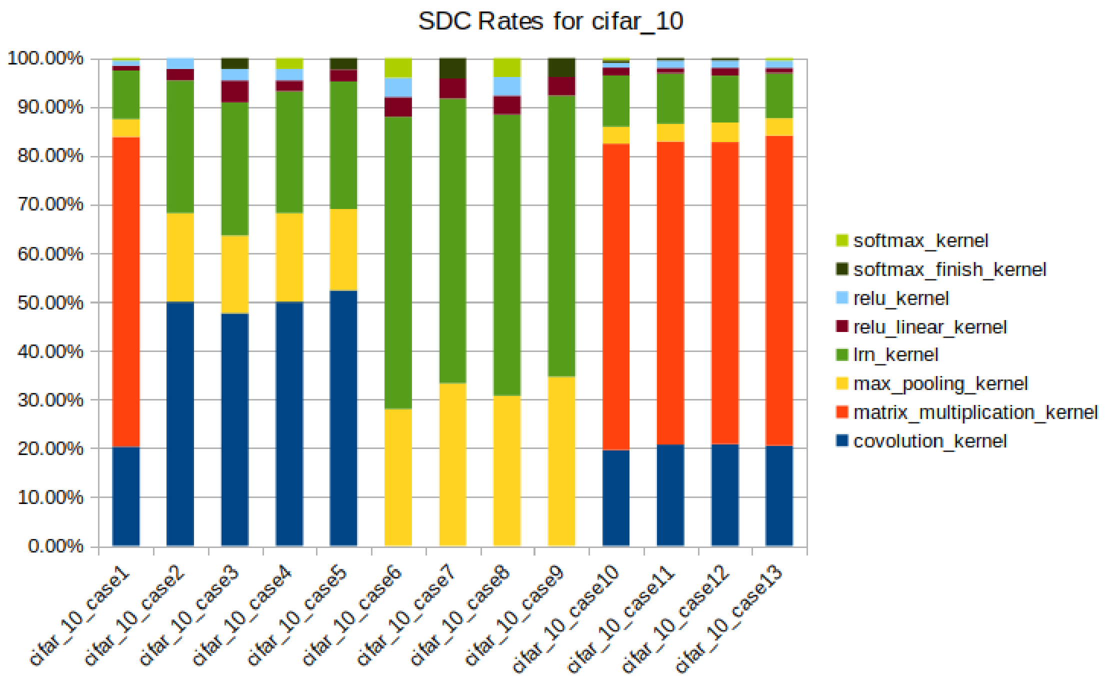


Figure 5.13. SDC Rates for *cifar_10*

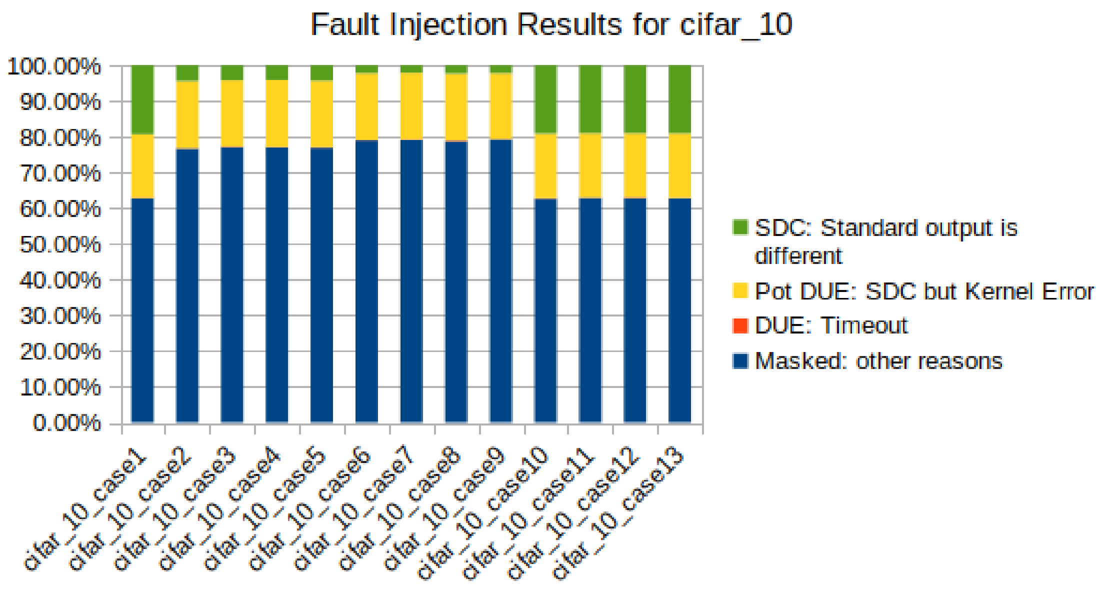


Figure 5.14. Fault Injection Results for *cifar_10*

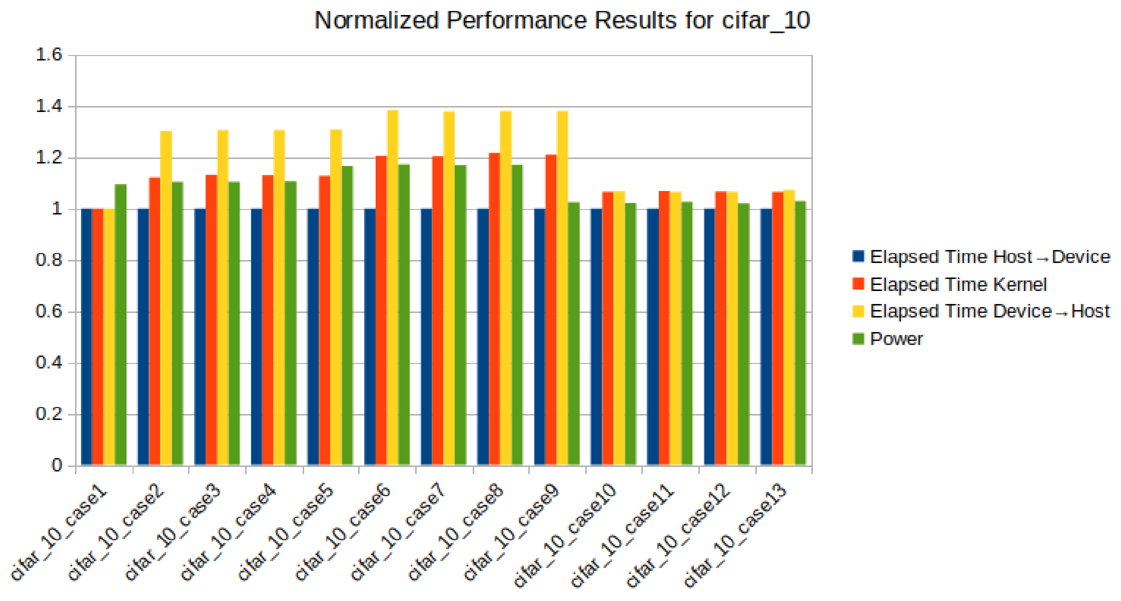


Figure 5.15. Normalized Performance Results for *cifar_10*

We first start with the measurement of execution times for each kernel inside *cifar_10* applications to evaluate if there is a correlation between SDC rates and longer execution time. In Figure 5.13, *matrix_multiplication_kernel* and *covolution_kernel* are the most error-vulnerable kernels in terms of SDC, and according to Figure 5.12, we show that a longer kernel execution time (Figure 5.12) can increase the SDC rate. In Figure 5.14, cases 2,3,4 and 5 only consider the most error-vulnerable kernel, *matrix_multiplication_kernel* to check if we can eliminate the majority of SDC by applying partial redundancy for only one kernel, for the following cases 6,7,8 and 9 evaluate the two most error-vulnerable kernel, *matrix_multiplication_kernel* and *covolution_kernel* and observed that most of the SDCs are masked just applying RMT based redundancy with these kernels, with very small performance overhead, less than 1.4 times compared to baseline for each performance metrics. For cases 10,11,12 and 13, we target the least error-vulnerable *softmax_kernel* kernel, to observe results on applying for redundancy, our initial assumption is to observe minimal SDC reduction with little performance impact. According to measurement results, our initial assumptions are consistent since almost no change in performance metrics, but also did not achieve an SDC improvement, so applying redundancy is a completely pointless effort.

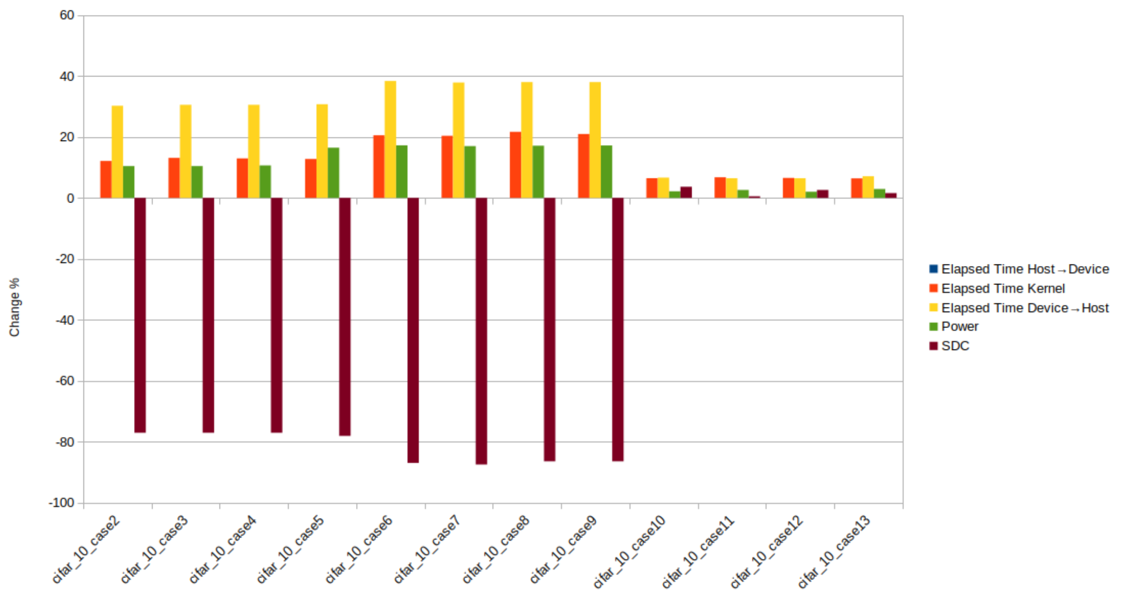


Figure 5.16. Tradeoff Analysis for *cifar_10*

In Figure 5.16, when partial redundancy is applied to the two most error-vulnerable kernels, we achieved eliminate 85% of SDCs with a cost of +20% of kernel execution time, +15% of power consumption, and +35% of memory copy time. According to these results, we show that a significant amount of SDCs can be eliminated with a small cost of overhead on runtime for complex benchmark applications by applying the software-based redundancy techniques that we offered.

CHAPTER 6

CONCLUSION AND FUTURE WORKS

In this thesis, we present several redundancy schemes with different spheres of replication units to evaluate the performance and reliability of our selected safety-critical domain benchmark. We first evaluate the performance and reliability of our redundancy techniques on single-kernel benchmarks and decide the best technique considering performance. We reveal that redundant multithreading is the best redundancy technique among all others since serial execution does not fully utilize the parallel architecture of GPUs, and stream creation may cause additional overhead. After that, we apply redundant multithreading on complex benchmarks partially for some kernels considering most and least error-vulnerable ones and show that most of the silent data corruptions can be eliminated with a small overhead.

6.1. Future Works

In this thesis, we propose different redundancy schemes but apply all the changes in source code and perform metrics profiling and fault injection manually on benchmarks. Instead of dealing the things manually, an automatized tool that profiles applications first to understand kernel behavior, performs fault injection to show the most-error vulnerable kernels, applies redundancy schemes with automatic implementation change, and collects performance metrics is the complete way of offering redundancy in safety-critical domain. In our selected benchmark, most of the applications are compute-bound and we did not evaluate memory-bound applications performance and reliability analysis in detail. We only used the same input arguments for execution of each benchmark and did not change it, so did not observe the effects of parameters such as matrix size. Several different benchmarks can be selected instead of dealing with only one would be better for understanding application-specific behavior. For the different spheres of replication, we aim to observe cache utilization in case of common input usage, but only the l2_utilization metric is collected and elapsed memory copy operation between host to device is used for evaluation, which can be extended with different parameters.

REFERENCES

Aamodt, Tor M., Wilson Wai Lun Fung, Timothy G. Rogers, and Margaret Martonosi. *General-Purpose Graphics Processor Architectures.* Morgan & Claypool Publishers, 2018.

Alcaide, Sergi, Leonidas Kosmidis, Carles Hernandez, and Jaume Abella. "Software-only diverse redundancy on GPUs for autonomous driving platforms." In *2019 IEEE 25th International Symposium on On-Line Testing and Robust System Design (IOLTS)*, 90-96. IEEE, 2019.

Alcaide, Sergi, Leonidas Kosmidis, Carles Hernandez, and Jaume Abella. "High-integrity GPU designs for critical real-time automotive systems." In *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 824-829. IEEE, March 2019.

Alcaide, Sergi, Leonidas Kosmidis, Carles Hernandez, and Jaume Abella. "Achieving Diverse Redundancy for GPU Kernels." *IEEE Transactions on Emerging Topics in Computing* 10, no. 2 (2021): 618-634.

Aslan, Büşra, and Ayse Yilmazer-Metin. "A Study on Power and Energy Measurement of NVIDIA Jetson Embedded GPUs Using Built-in Sensor." In *2022 7th International Conference on Computer Science and Engineering (UBMK)*, pp. 1-6. IEEE, 2022.

Burtscher, Martin, Ivan Zecena, and Ziliang Zong. "Measuring GPU power with the K20 built-in sensor." In *Proceedings of Workshop on General Purpose Processing Using GPUs*, pp. 28-36. 2014.

Defour, David, and Eric Petit. "GPUburn: A system to test and mitigate GPU hardware failures." In *2013 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*, pp. 263-270. IEEE, 2013.

Dimitrov, Martin, Mike Mantor, and Huiyang Zhou. "Understanding software approaches for GPGPU reliability." In *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, pp. 94-104. 2009.

Douglass, Powel. "Safety-critical systems design." *Electronic engineering* 70, no. 862 (1998): 45-6.

Mazzocchetti, Fabio, Sergi Alcaide, Francisco Bas, Pedro Benedicte, Guillem Cabo, Feng Chang, Francisco Fuentes, and Jaume Abella. "SafeSoftDR: a library to enable software-based diverse redundancy for safety-critical tasks." *arXiv preprint arXiv:2210.00833* (2022).

Fang, Bo, Karthik Pattabiraman, Matei Ripeanu, and Sudhanva Gurumurthi. "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)*, pp. 221-230. IEEE, 2014.

Fickenscher, Jörg, Sebastian Reinhart, Frank Hannig, Jürgen Teich, and Mohamed Essayed Bouzouraa. "Convoy tracking for ADAS on embedded GPUs." In *2017 IEEE Intelligent Vehicles Symposium (IV)*, pp. 959-965. IEEE, 2017.

Hari, Siva Kumar Sastry, Timothy Tsai, Mark Stephenson, Stephen W. Keckler, and Joel Emer. "Sassifi: An architecture-level fault injection tool for gpu application resilience evaluation." In *2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pp. 249-258. IEEE, 2017.

Kastensmidt, Fernanda, and Paolo Rech. "FPGAs and parallel architectures for aerospace applications." *Soft Errors and Fault-Tolerant Design* (2016).

Kirk, David B., and W. Hwu Wen-Mei. *Programming massively parallel processors: a hands-on approach*. Morgan kaufmann, 2016.

Knight, John C. "Safety critical systems: challenges and directions." In *Proceedings of the 24th international conference on software engineering*, pp. 547-550. 2002.

Kohn, Andre, Michael Käßmeyer, Rolf Schneider, Andre Roger, Claus Stellwag, and Andreas Herkersdorf. "Fail-operational in safety-related automotive multi-core systems." In *10th IEEE International Symposium on Industrial Embedded Systems (SIES)*, pp. 1-4. IEEE, 2015.

Kosmidis, Leonidas, Jérôme Lachaize, Jaume Abella, Olivier Notebaert, Francisco J. Cazorla, and David Steenari. "GPU4S: Embedded GPUs in space." In *2019 22nd Euromicro Conference on Digital System Design (DSD)*, pp. 399-405. IEEE, 2019.

Li, Guanpeng, Karthik Pattabiraman, Chen-Yang Cher, and Pradip Bose. "Understanding error propagation in GPGPU applications." In *SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 240-251. IEEE, 2016.

Mahmoud, Abdulrahman, Siva Kumar Sastry Hari, Michael B. Sullivan, Timothy Tsai, and Stephen W. Keckler. "Optimizing software-directed instruction replication for gpu error detection." In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 842-854. IEEE, 2018.

Mukherjee, Shubu. *Architecture design for soft errors*. Morgan Kaufmann, 2011.

Nie, Bin, Ji Xue, Saurabh Gupta, Christian Engelmann, Evgenia Smirni, and Devesh Tiwari. "Characterizing temperature, power, and soft-error behaviors in data center systems: Insights, challenges, and opportunities." In *2017 IEEE 25th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pp. 22-31. IEEE, 2017.

NVIDIA Developer's Guide Jetson Stats. Accessed November 8, 2023. https://developer.nvidia.com/embedded/community/jetson-projects/jetson_stats

NVIDIA Profiler User's Guide. Accessed November 8, 2023. https://docs.nvidia.com/cuda/profiler-users-guide/index.html

NVIDIA Technical Blog, Introducing Jetson Xavier NX, the World's Smallest AI Supercomputer. Accessed November 8, 2023. https://developer.nvidia.com/blog/jetson-xavier-nx-the-worlds-smallest-ai-supercomputer/

Oliveira, Daniel AG, Paolo Rech, Heather M. Quinn, Thomas D. Fairbanks, Laura Monroe, Sarah E. Michalak, Christine Anderson-Cook, Philippe OA Navaux, and

Luigi Carro. "Modern GPUs radiation sensitivity evaluation and mitigation through duplication with comparison." *IEEE Transactions on Nuclear Science* 61, no. 6 (2014): 3115-3122.

Öz, Işıl, and Ömer Faruk Karadaş. "Regional soft error vulnerability and error propagation analysis for GPGPU applications." *The Journal of Supercomputing* 78, no. 3 (2022): 4095-4130.

Palin, Robert, and Ibrahim Habli. "Assurance of automotive safety–a safety case approach." In *Computer Safety, Reliability, and Security: 29th International Conference, SAFECOMP 2010, Vienna, Austria, September 14-17, 2010. Proceedings 29*, pp. 82-96. Springer Berlin Heidelberg, 2010.

Perez-Cerrolaza, Jon, Jaume Abella, Leonidas Kosmidis, Alejandro J. Calderon, Francisco Cazorla, and Jose Luis Flores. "GPU devices for safety-critical systems: A survey." *ACM Computing Surveys* 55, no. 7 (2022): 1-37.

Portet, Sergi Alcaide, Leonidas Kosmidis, Carles Hernandez, and Jaume Abella. "Software-only triple diverse redundancy on GPUs for autonomous driving platforms." In *2020 50th Annual IEEE-IFIP International Conference on Dependable Systems and Networks-Supplemental Volume (DSN-S)*, pp. 82-88. IEEE, 2020.

Previlon, Fritz G., Babatunde Egbantan, Devesh Tiwari, Paolo Rech, and David R. Kaeli. "Combining architectural fault-injection and neutron beam testing approaches toward better understanding of GPU soft-error resilience." In *2017 IEEE 60th International Midwest Symposium on Circuits and Systems (MWSCAS)*, pp. 898-901. IEEE, 2017.

Previlon, Fritz, Charu Kalra, Devesh Tiwari, and David Kaeli. "Characterizing and exploiting soft error vulnerability phase behavior in gpu applications." *IEEE Transactions on Dependable and Secure Computing* 19, no. 1 (2020): 288-300.

Topçu, Burak, and Işıl Öz. "Soft error vulnerability prediction of GPGPU applications." *The Journal of Supercomputing* 79, no. 6 (2023): 6965-6990.

Tsai, Timothy, Siva Kumar Sastry Hari, Michael Sullivan, Oreste Villa, and Stephen W. Keckler. "Nvbitfi: Dynamic fault injection for gpus." In *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pp. 284-291. IEEE, 2021.

Tselonis, Sotiris, and Dimitris Gizopoulos. "GUFI: A framework for GPUs reliability assessment." In *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pp. 90-100. IEEE, 2016.

Ubal, Rafael, Julio Sahuquillo, Salvador Petit, and Pedro Lopez. "Multi2sim: A simulation framework to evaluate multicore-multithreaded processors." In *19th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD'07)*, pp. 62-68. IEEE, 2007.

Vallero, Alessandro, Dimitris Gizopoulos, and Stefano Di Carlo. "SIFI: AMD southern islands GPU microarchitectural level fault injector." In *2017 IEEE 23rd International Symposium on On-Line Testing and Robust System Design (IOLTS)*, pp. 138-144. IEEE, 2017.

Wadden, Jack, Alexander Lyashevsky, Sudhanva Gurumurthi, Vilas Sridharan, and Kevin Skadron. "Real-world design and evaluation of compiler-managed GPU redundant multithreading." *ACM SIGARCH Computer Architecture News* 42, no. 3 (2014): 73-84.

Wagner, Stefan, Bernhard Schätz, Stefan Puchner, and Peter Kock. "A case study on safety cases in the automotive domain: Modules, patterns, and models." In *2010 IEEE 21st International Symposium on Software Reliability Engineering*, pp. 269-278. IEEE, 2010.

Yim, Keun Soo, Cuong Pham, Mushfiq Saleheen, Zbigniew Kalbarczyk, and Ravishankar Iyer. "Hauberk: Lightweight silent data corruption error detector for gpgpu." In *2011 IEEE International Parallel & Distributed Processing Symposium*, pp. 287-300. IEEE, 2011.