UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE ENGENHARIA DE COMPUTAÇÃO

GERÔNIMO VEIT ACOSTA

Using precision reduction to efficiently improve mixed-precision GPUs reliability

Porto Alegre
2021

GERÔNIMO VEIT ACOSTA

Using precision reduction to efficiently improve mixed-precision GPUs reliability

Work presented in partial fulfillment of the requirements for the degree of Bachelor in Computer Engineering

Advisor: Prof. Dr. Paolo Rech
Coadvisor: Prof. Fernando Fernandes

Porto Alegre
2021

CIP — CATALOGING-IN-PUBLICATION

Acosta, Gerônimo Veit

Using precision reduction to efficiently improve mixed-precision GPUs reliability / Gerônimo Veit Acosta. – Porto Alegre: 2021.

48 f.

Advisor: Paolo Rech

Trabalho de conclusão de curso

– Universidade Federal do Rio Grande do Sul, Escola de Engenharia. Curso de Engenharia de Computação, Porto Alegre, BR–RS, 2021; Coadvisor: Fernando Fernandes.

1. Fault tolerance. 2. Reliability. 3. Radiation. 4. Duplication. 5. DWC. 6. RP-DWC. 7. GPU. 8. HPC. I. Rech, Paolo, orient. II. Fernandes, Fernando. III. Título.

Dedico este trabalho à minha irmã Martina, meu pai Kenni e minha mãe Liliana,
por todo carinho e apoio.

ACKNOWLEDGMENT

First and foremost, I would like to acknowledge everyone that supported me throughout these past 6 years since I was accepted to the Computer Engineering major at Universidade Federal do Rio Grande do Sul. I would especially like to thank my colleagues for creating a competitive and healthy educational environment that motivated me to pursue my main goal. I express my sincere gratitude to the ones that I can now call friends for all the good memories and meaningful experiences. These moments will forever remain in my mind.

To the Institute and to the professors, my deepest gratitude for promoting a setting where the search for knowledge and challenges was always encouraged. Your role is fundamental to train qualified computer engineering professionals and to fully educate citizens that are eager to improve the society while creating prosperous opportunities. To Paolo Rech, my advisor, and to Fernando Fernandes, my co-advisor, I am extremely grateful for your patience, care, thoughtful explanations, and especially for accepting my invitation to be part of this work.

Finally, I thank my family for being truthful to their core values. Certainly I would not have reached this far without their constant support. They are an essential and vital part of me and I wish to go a lot further and to pursue even higher goals alongside them.

AGRADECIMENTOS

Agradeço a todos que me apoiaram ao longo destes últimos 6 anos, desde quando fui aceito para o curso de Engenharia de Computação. Especialmente, gostaria de deixar meus sinceros agradecimentos para os meus colegas de curso que propiciaram um ambiente de competição saudável e de muito conhecimento; para aqueles que hoje posso chamar de amigos, agradeço por todos bons momentos vividos, memórias e aprendizados. Tudo isso sempre terá um logo muito especial em minhas memórias.

Ao Instituto de Informática e seus professores, obrigado por colaborarem de uma maneira tão nobre com o ensino e incentivarem a disseminação de conhecimento. Seus papeis não são apenas fundamentais para formar competentes profissionais da engenharia de computação, mas também para educar e formar cidadãos que vão contribuir em prol de uma sociedade mais igual. Ao professor e orientador Paolo Rech e ao Fernando Fernandes, meu co-orientador, meus sinceros agradecimentos pela paciência, atenção, ensinamentos, conselhos e especialmente por terem aceitado fazer parte deste trabalho ao meu lado.

Por último, não posso de deixar de agradecer à minha família, a quem dedico este trabalho. Certamente eu não teria conseguido chegar a este ponto nem da gradução nem dos meus 24 anos de vida sem o seu apoio. Muito obrigado por serem uma parte tão essencial da minha criação. Espero alncaçar muitos outros objetivos ao lado de vocês.

“If I have seen farther than others,

it is because I stood on the shoulders of giants.”

— Sir Isaac Newton

ABSTRACT

Duplication With Comparison (DWC) is a traditional and accepted method for improving systems' reliability. DWC consists of duplicating critical regions in Software or in Hardware level by creating redundant operations in order to decrease the probability of an unwanted event. However, this technique introduces an expensive overhead in power consumption, processing time and in resources allocation. This obstacle is due to the fact that the critical operations are computed at least two times in this process.

Reduced Precision Duplication With Comparison (RP-DWC) is an effective software level solution to improve the performance of the conventional DWC. RP-DWC aims to mitigate these overheads by enabling parallel processing in underused Floating Point Units (FPUs) in mixed precision Graphic Processing Units (GPUs). By making use of precision reduction to efficiently improve the reliability in mixed precision GPUs, RP-DWC extends the DWC technique, introducing proper ways to handle redundancy with different precision operations.

Improving GPUs reliability is an extremely valuable challenge in the fault tolerance field since GPUs are adopted in both High-Performance Computing (HPC) and in automotive real-time applications. When GPUs are exposed to a natural environment, such as the surface of the Earth at sea level, they are also exposed to the Earth's surface radiation. Furthermore, this exposure can be critical, given that these radiation particles may hit the GPU's internal circuit, corrupt sensitive data and consequently generate undesired outputs.

Introducing duplication with reduced precision in a trustworthy manner to maintain reliability in safety-critical systems is an arduous task that we propose to further investigate in this work.

Keywords: Fault tolerance. reliability. radiation. duplication. DWC. RP-DWC. GPU. HPC.

LIST OF ABBREVIATIONS AND ACRONYMS

DWC	Duplication With Comparison
RP-DWC	Reduced Precision Duplication With Comparison
FPU	Floating Point Unit
GPU	Graphic Processing Unit
HPC	High-Performance Computing
SEE	Single Event Effect
DUE	Detected Unrecoverable Error
SDC	Silent Data Corruption
SEU	Single Event Upset
MBU	Multiple Bit Upset
SER	Soft Error Rate
FIT	Failure in Time
TMR	Triple Modular Redundancy
GPP	General-Purpose Processor
SM	Streaming Multiprocessor
GP-GPU	General-Purpose Graphic Processing Unit
CUDA	Compute Unified Device Architecture
MIMD	Multiple Instruction Multiple Data
SIMT	Single Instruction Multiple Thread
SM	Streaming Multiprocessor
FP64	64-bit Floating Point Instruction
FP32	632-bit Floating Point Instruction
EPL	Expected Precision Lost

LIST OF FIGURES

Figure 2.1 Warp scheme 22
Figure 2.2 Pascal Streaming Multiprocessor (SM) 23
Figure 2.3 Volta Streaming Multiprocessor (SM) 24
Figure 2.4 Blocks distribution 26
Figure 2.5 A simplified vision of the DWC detection strategy 28

Figure 3.1 A simplified pipeline view to illustrate the thread execution gain explored by RP-DWC hardening strategy 31
Figure 3.2 A macro visualization of the RP-DWC strategy 32

Figure 4.1 RP-DWC undetectable region 40
Figure 4.2 Hotspot Fault Detection 41
Figure 4.3 NMF Fault Detection 42
Figure 4.4 CGS Fault Detection 43
Figure 4.5 Hotspot Performance Results 43
Figure 4.6 NMF Performance Results 43
Figure 4.7 CGS Performance Results 44

LIST OF TABLES

Table 1.1 The main attributes of dependability 15
Table 1.2 The 4 phases of Anderson and Lee 16

Table 2.1 A comparison between NVIDIA Tesla GPUs 25
Table 2.2 Compute Capability in modern architectural NVIDIA GPUs 25

Table 3.1 A summary of the benchmarks attributes 38

Table 4.1 Comparison between FP32 and FP64 instructions in Volta 41

CONTENTS

1 INTRODUCTION..12
1.1 Faults, Errors and Failures 12
1.1.1 Radiation 13
1.2 Fault Tolerance..14
1.2.1 The phases of fault tolerance techniques applications..15
1.2.2 Redundancy...16
1.2.3 Reliability...18
2 FAULT TOLERANCE IN GPUS 21
2.1 Modern GPUs Architectures...21
2.2 Vulnerability to transient faults...25
2.3 Duplication With Comparison...27
3 PROPOSAL..30
3.1 Definition...30
3.2 Implementation 32
3.2.1 Expected Precision Lost...33
3.2.2 Fault Injection 34
3.2.3 Benchmarks...35
4 RESULTS...39
5 CONCLUSION 45
REFERENCES..46

1 INTRODUCTION

Beforehand, we will introduce primordial concepts and definitions of the fault tolerance area along with the current challenges of this field of study, which inspired this monograph. In this chapter, we will discuss the importance of reliability and how the generation of unwanted behaviors must be treated in modern computing applications.

1.1 Faults, Errors and Failures

In order to comprehend the motivations that lead to the first research projects in fault tolerance area and the reasons why computing systems are not perfect running engines, it is crucial to introduce the concepts of *fault*, *error* and *failure*. Neither hardware nor software structures are built to predict every erroneous behavior once these systems' correct operation depends on a great number of variable factors. As an example, a circuit break is a natural event that has the potential to originate wrong information in electrical circuits and hence it can take down a running application.

Just like a circuit break is a natural phenomena, there are other natural and non natural factors that can originate the first important concept, a *fault*. Any undesired event capable of giving rise to a system error may be considered a fault, which can be classified according to their origin (LAPRIE, 1995):

- **Physical**: adverse physical phenomena, either internal (threshold changes, short-circuits, open-circuits) or external (environmental perturbations: electro-magnetic, temperature, vibration);
- **Human-made**: imperfections which may be:

  Design faults: triggered either in the initial system's design or subsequent modifications;

  Interaction faults: triggered by violations in operating or maintenance procedures.

Both previously defined faults can potentially prompt another important concept: an *error*. Aforementioned, an error occurs when a fault causes an undesired circumstance to an internal system. Lastly, the final important initial concept is a consequence that comes from the error generation and that can be felt by the end user: a *failure*. It is an undesired event that affects the service while generating wrong outputs. Failures may or may not be critical, depending on whether the incorrect output altered sensitive data or not. Summarizing, a fault is an event that can introduce bad internal data, characterized by an error. In turn, if an error propagates to the system's output, a failure occurs. For the purpose of this work we will focus mainly on the *physical faults*.

1.1.1 Radiation

Physical faults that are originated by radiation events naturally present in the surface of the Earth are especially damaging. The consumer demand for more realistic, functional and reliable applications requires very high standards of electronic systems. This request comes upon the crescent technology advance which is boosting the transistor's layout by reducing its dimensions and operating voltages. In this scenario, the radiation effects in the most fundamental electronic component, the transistors, are responsible for generating major complications, from data disruptions to permanent damage and device failure (BAUMANN, 2005).

The effects of radiation in computing systems can be summarized as energy disruptions that overflow the semiconductors voltage/current in sensitive regions generating potential deformations. The called single-event effects (SEEs) are caused by a single radiation event that can generate device failures. SEEs can be classified as *hard* or *soft*. The former occurs when a radiation event generates permanent damage being predominant in space and military environment thus not being explored in this scope. On the other hand, soft faults, or *transient faults*, do not permanently damage the device. They are caused by enough charge disruption that flips the state of a bit cell (*i.e* memory, register, latch) and they can potentially originate misleading information or system behaviour. Additionally, faults caused by charge disruption are the most commonly found physical faults.

Transient faults can induce different outcomes in program outputs. A fault is called *masked* when no effect is observed or, in the same way, no failure is created. Conversely, when a fault crashes the system and generates forbidden memory access or corrupts the internal data, it is described as *Detected Unrecoverable Error (DUE)*. Lastly, when the fault's outcome is a data corruption but instead of crashing the system it generates a wrong output, the fault is labeled as *Silent Data Corruption (SDC)*.

Certain systems have shown particular vulnerability to transient faults depending mainly on two factors: the number of sensitive instructions that they execute and the amount of time that they remain exposed to radiation. The prolonged exposure to radiation increases the probability of a system getting hit by a particle in a sensitive internal region and, therefore, of generating a failure. When only one bit value is flipped due to a transient fault it is described as a *Single Event Upset (SEU)*. Nonetheless, the *Multiple Bit Upset (MBU)*, occurs when the radiation energy is enough to flip more than one bit value. Radiation particles refer in majority to *neutron elements*, in which atmospheric flux at sea level in the natural environment is $13n/(cm^2 \times h)$ (JEDEC, 2006).

The description of some of the failures caused by radiation events gives a better comprehension of its impact in device circuits. There are multiple radiation sources that are potential fault generators (BAUMANN, 2005). In the terrestrial surface environment, we are exposed from natural radioactivity of material to industrial, medical, nuclear and high energy physics equipment that generate considerable higher radiation doses. The radiation events greatly increase in higher altitudes as the particles' flux of neutrons and electrons, for instance, can reach up to 1,000 times more than its sea level incidence. Particularly, the space environment is much more exposed to radiation. Galactic cosmic rays generated by supernova explosions or celestial bodies collision, solar wind and flares and Van Allen belts are some of the radioactive events that create a great flux of radiation particles such as ions, protons and electrons. Reliable space technology is, in this perspective, much more challenging to be designed.

The occurrence of radiation events in computing systems can induce mechanism failures in addition to compromising the systems' reliability. In this hypothesis, computations can be adopted to evaluate the error rate, also called the *soft error rate (SER)*. This calculation uses the *failure in time (FIT)* as a measurable value to classify the magnitude of failures. The SER can easily exceed 50,000 FIT/chip if any protection mechanism is adopted, which makes a huge contrast to the typical and acceptable FIT/chip rate of reliable structures that are in the range of 1-50 FIT/chip (BAUMANN, 2005).

1.2 Fault Tolerance

The concept of *fault tolerance* is historically known since the invention of the first computers (NEWMANN, 1956) but it was only formally defined by Aviziens in 1967 (AVIZIENS, 1998). Systems that need to maintain their correctness even in the presence of faults must be built using fault tolerance techniques. Usually such systems are those that need high *reliability*, like aircrafts flight systems, on which human lives relies on its correct operation; or *availability* such as high availability data clusters that need to uninterruptedly operate and provide on-demand information. Either way, fault tolerance and the concept of *redundancy* are extremely interconnected. In fact, fault tolerant systems can be commonly called redundant systems due to the great relation between these terms.

A system may be referred to as fault tolerant as it is also *dependable*. As a matter of fact, the main goal of fault tolerance is to reach *dependability*, which refers to the quality of information that the system provides and it is measured by a number of numeric attributes. Table 1.1 summarizes the main ones (WEBER, 2002). In this chapter, we will focus on some of the most relevant concepts and techniques directly related to this work. The definitions of *reliability* and *redundancy* are detailed in later sections as both of them are crucial to a better comprehension.

Table 1.1 – The main attributes of dependability

Attribute	*Description*
Reliability	Capacity of a system to deliver well defined specifications and to be operational during a well known period of time.
Availability	Probability of a system to be operational in a period of time.
Safety	Probability of a system to be operational and to deliver either information correctly or discontinue it in a way to avoid damage to other systems or people who depend on it.
Security	Capacity of a system to be protected against malicious faults, providing data privacy, authenticity and integrity.

Source: (WEBER, 2002)

1.2.1 The phases of fault tolerance techniques applications

From the moment that a fault is detected, proper ways to handle and repair the erroneous behavior that the fault generated are put into action . The most common classification technique to handle a fault is the four application phases defined by Anderson and Lee (ANDERSON; LEE, 1981). They are: *detection*, *confinement*, *recovery* and *treatment* as shown in Table 1.2.

It is also significant to mention that another phase present in the literature is the *fault masking*, which is the non-manifestation of a fault as an error. This signifies that the system will remain in a correct state and the fault will never be noticed by any error detection mechanism. If so, a permanent damage may exist even if the system never notices its presence, and it may be located and repaired albeit it is not necessary to detect, confine and recover.

Table 1.2 – The 4 phases of Anderson and Lee

Phase	*Description*
Detection	When some of the detection mechanisms notice the system error generated by the fault. One of these detection mechanisms is *duplication with comparison*, which is the technique that motivated the investigation of this work
Confinement	Between the error manifestation and detection there is a time gap where the erroneous data can spread. The confinement proposal is to avoid the damage propagation by implementing specific techniques.
Recovery	When a switch context from an invalid state to a valid one is established. While the system is in an invalid state it may not be recovered. The *forward error recovery* leads the execution to a new and previously unvisited state while *backward error recovery* leads to a previous error-free state.
Treatment	Consists of locating the root cause of the error, investigating and repairing the fault that originated it and at last recovering the remaining of the system.

1.2.2 Redundancy

Every fault tolerance technique involves some level of *redundancy*. This basic concept is related to the provision of functional capabilities that would be unnecessary in a fault-free environment (LAPRIE, 1995). The usage of additional backup components in hardware or in software structures that would prevent the system from operating incorrectly if one component fails illustrates a practical exemplification of this term.

Redundancy intrinsically comes with additional costs in an application design as it demands extra resources or algorithms to deliver the reliability needed. For this reason there is an extra precaution in choosing which redundancy technique will be employed in a system structure. There are relevant trade-offs that may be considered in order to properly implement any redundancy approach. They can be classified by the following categories (WEBER, 2002):

- **Information redundancy**: Comes with a small extra hardware cost since the usage of extra bits are required. These bits are stored and broadcasted alongside the original data to check whether the information is correct or not. Parity bits are common examples of this redundancy method;

- **Time redundancy**: Comes with no extra hardware cost but instead brings with it additional execution time. This technique re-executes the computation periodically and is commonly found in applications that do not require critical execution time;
- **Hardware redundancy**: Consists of replicating physical components in order to maintain the operational state of a system even if one other component fails. This method can have a high additional cost depending on which hardware components must or not be redundant. Hardware redundancy can be divided into:
    - *Static*: This method uses a voting system to decide which result for the same task produced by each redundant component is the correct one. This solution does not propose to investigate and recover the system ergo it is called a *static* method. Instead, the fault is masked by the voting system that decides to reject the component's result where the fault was detected and proceed the execution with the correct output generated by an error-free component. For instance, the *Triple Modular Redundancy (TMR)* is a static method that uses three redundant components and typically a majority voting system to decide which output should be used as the correct one. Since there exists three redundant components, it is enough that two of them produces the same result to win the voting;
    - *Dynamic*: The dynamic flow proposes to detect the fault, locate its occurrence and recover the system. This method utilizes a fault detection mechanism to detect the error, a diagnostic tool to locate it and another functionality to keep the operation in an error-free state;
    - *Hybrid*: The last method proposes to combine both static and dynamic procedures to apply hardware redundancy.
- **Software redundancy**: Duplicating software components is useless to detect and handle errors. Software will probably produce exactly equal outputs for the same identical input as many times as it is executed. There are another methods to make software redundancy effective, classified as follows:
    - *N-version programming*: This method is based on N-different implementations that propose to solve the same problem (CHEN; AVIZIENIS, 1995).

Such as the TMR method, the N-version programming also makes use of a voting system that decides which of the N-different outputs will be labeled as the error-free one. By doing this, the method guarantees that the software redundancy will not always produce the exact same outputs for the same input values, since there are non-identical implementations;

- *Recovery block*: Very similarly to the N-version programming, this method also uses N-different implementations for the same problem. The main difference is that does not exists a voting system, but instead an *acceptance test*. In this case, every different output is submitted to the acceptance test starting by the first version, called the primary version (WEBER, 2002). If the primary version does not pass in the acceptance test, the testing routine proceeds with the N-1 secondary versions until one of them passes.

It is important to clarify that this work is based on a *fault detection* method commonly used in dynamic hardware redundancy, the *Duplication With Comparison (DWC)*. This technique aims to duplicate components, compare their outputs and decide whether to handle or not a potential error.

1.2.3 Reliability

Reliability can be defined as the probability of failure-free operation of a computer program for a specified time in a specific environment (EUSGELD *et al.*, 2005). Reliable programs are often evaluated by software and architectural metrics that measure the component's robustness and how sensitive they are in the presence of faults. Traditionally, the large amount of failures is caused by hardware physical wearout or deterioration that introduce faults into the components and hence lead to an incorrect output (EUSGELD *et al.*, 2005). Some systems implement robust fault treatment in order to maintain their internal correctness in the presence of faults since critical data is computed. In particular, safety-critical or life-critical systems are those that require great reliability to operate safely without causing (KNIGHT, 2002):

- Death or serious injury to people;
- Loss or severe damage to equipment/property;
- Environmental harm.

Fault tolerance has been one of the most challenging subjects in the reliability area of study in recent years for safety-critical applications (LUCAS *et al.*, 2014). One of the main reasons for the considerable relevance given is the increased usage of Graphic Processing Units (GPUs), modern hardware architectures that explores great level of parallelism, in High-Performance Computing (HPCs). This includes programs such as neural networks or real-time object detection in self-driven vehicles, for which reliability needs to be paramount. GPUs are widely adopted since they have physical resources that are able to deliver the high computational power needed by these algorithms. In this scenario, vulnerability has a particular unbearable impact that must be reduced to avoid unwanted effects.

GPUs have shown to be particularly sensitive to transient faults (SANTOS *et al.*, 2020). A considerable number of techniques to detect and decrease erroneous behaviors of computer systems are known for different applications and purposes. Safety-critical ones are specially crucial in this scenario as any undesirable and incorrect data caused by any fault, especially transient ones (GONÇALVES DE OLIVEIRA *et al.*, 2016), has the potential to generate a catastrophic impact. Executing high performance applications for safety-critical purposes demands great computational capacity thus parallel processing units like the GPUs are normally used. In order to detect and correct with assertiveness bad sensitive data in runtime, software or hardware modifications need to be implemented. To achieve reliability in these architectures without relative loss of computational power some trade-offs must be considered.

In this work we present a technique for Duplication With Comparison (DWC), a software level solution widely used to improve reliability of computing systems, including Graphic Processing Units (GPUs). DWC consists of duplicating sensitive operations of the system and comparing their outputs. This approach is extremely effective as it detects in best cases up to 90% of transient faults (OLIVEIRA *et al.*, 2014), (MAHMOUD *et al.*, 2018). However, this implementation introduces an overhead in power consumption and processing performance that can reach up to more than $2\times$ of the original execution time as it comes intrinsically with a significant increase in instructions and usage of resources.

Reduced Precision Duplication With Comparison (RP-DWC) is proposed in this work as the main software level solution to preserve reliability and improve hardware performance when compared to the traditional method. The intention is to demonstrate how hardware parallelism can be boosted by executing in both underused single precision Floating Point Units (FPUs) and double precision Floating Point Units (FPUs) duplicated instructions without compromising the application correctness and reliability.

2 FAULT TOLERANCE IN GPUS

Considering that GPUs architectures were first invented to introduce considerable performance gain in graphics processing applications, they feature more robust and complex hardware structures when compared to the ones found in general-purpose processors' (GPPs). With the microtechnology advance in recent years, the complexity of modern parallel architectures is enabling more powerful and accelerated Graphic Processing Units to be designed. The architectural components that make up these GPUs enable a great number of different and independent contexts of the same program to be simultaneously executed.

Hence, preserving the reliability and the correctness in such applications is a particularly challenging task. To gain a better comprehension of why fault tolerance is critical in GPUs, a brief introduction of modern architectures is presented.

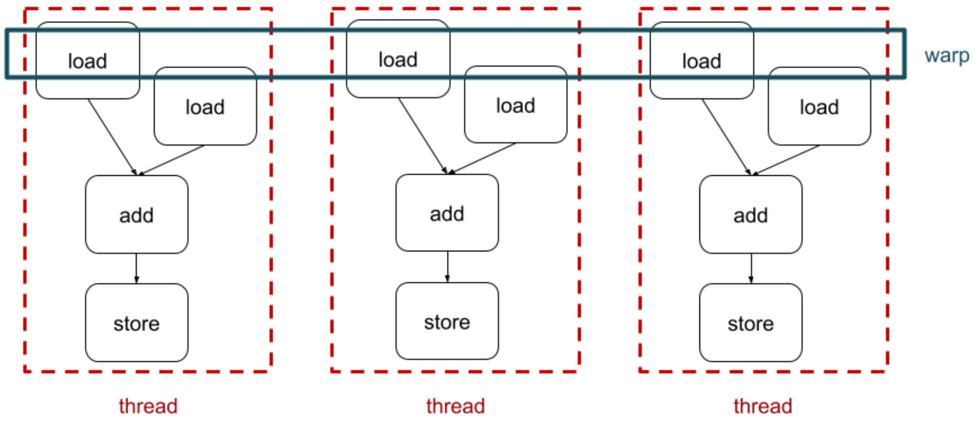
2.1 Modern GPUs Architectures

Since this work is based on implementations made in a NVIDIA Volta architecture GPU, this discussion is limited by some of the GPUs architectures from the NVIDIA's products line. The referred ones are predecessors of the Volta architecture, like Pascal and Kepler.

The NVIDIA line of products is composed by high performance General-Purpose Graphic Processing Units (GP-GPUs) which are known as the GPUs reference in High Process Computing (HPC) market (ACCELERATING..., 2021 (accessed May 12, 2021)). Each of them is based on a parallel computing platform along with an application programming interface model created by NVIDIA named CUDA (Compute Unified Device Architecture) (CUDA..., 2021 (accessed May 12, 2021)).

The probably most basic parallel architecture model is the *multiple instruction multiple data (MIMD)*. Every single thread has its own set of instructions and therefore it also has its own context. At the code level, the programmer can manually configure the number of threads that will be launched in the program context. Traditionally, a MIMD model enables multiple instructions between multiple threads to be executed in parallel. On the other hand, the NIVIDA GPUs handle the parallelism model differently to optimize the performance gain. These GPUs use the *single instruction multiple thread (SIMT)* that unifies a set of equal instructions of each thread into a single large one, called *warp*, the minimal execution unit for GPUs, as shown in Figure 2.1. The usage of a warp is beneficial to the GPU chiefly because the same instruction that accesses the same region of memory is executed by multiple threads differing only in the index of each one. Programs that take advantage of this process normally are those with great number of vector operations. This mechanism reduces the complexity of decoding different instructions by different threads.

Figure 2.1 – Warp scheme



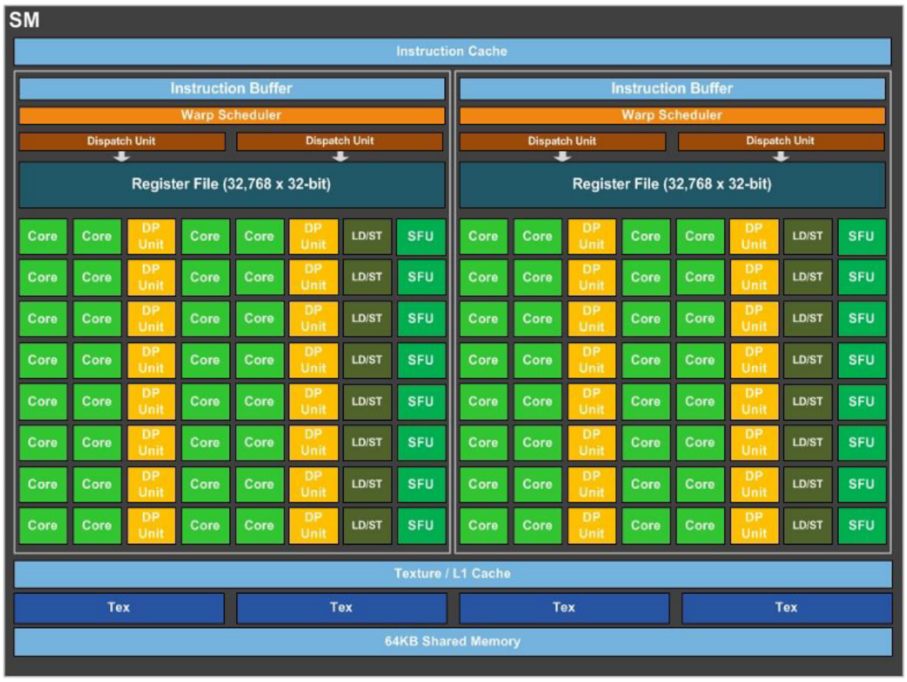
It is fundamental to emphasize that the GPUs optimization models are only advantageous to programs that execute a large set of instructions that can be parallelizable.This applications must present a low dependency level between instructions such as matrix operations mainly found in image processing algorithms.

CUDA GPUs structures comprise a scalable array of multithreaded *streaming multiprocessors (SMs)* (PROGRAMMING. . . , 2010 (accessed May 12, 2021)). Each of them contains a great number of CUDA cores. They are defined to indicate structures inside the SM that are capable to execute one thread with dedicated resources, including the *arithmetic units*. In the Volta SM structure, the concept of the traditional CUDA Core was deprecated. The Floating Point Units (FPUs) found with both 64-bit an 32-bit precision provide, alongside the other components inside the SM, the same parallelism and dedicated execution approach found within the CUDA Cores. The FPUs are responsible for executing the floating point instructions being specially important in this context as they are the enablers mechanisms that motivated this study.

Another important organizational structure present in Volta is the *Tensor Core*.

This specific core will remain apart from the discussion but is noteworthy that it enables even more compute capability to GPUs specially when executing deep learning algorithms. In figures 2.2 and 2.3 is possible to see and compare the organization of two different modern GPU SMs in Pascal and Volta architectures.

Figure 2.2 – Pascal Streaming Multiprocessor (SM)

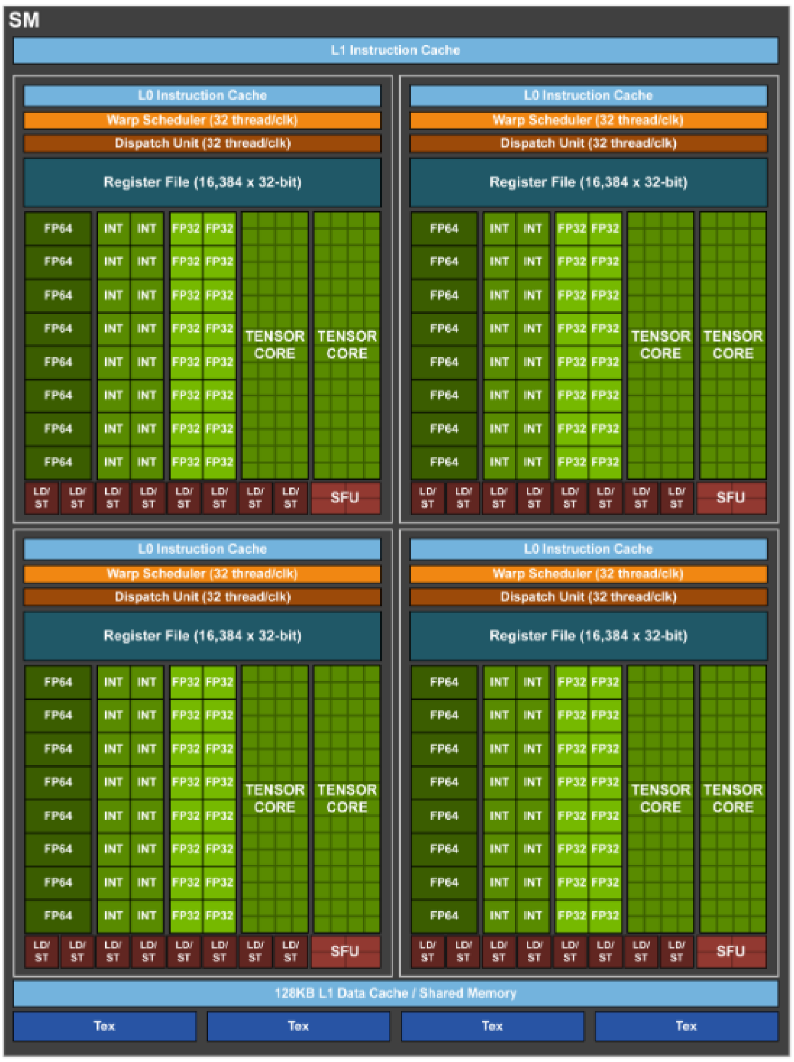


Source: NVIDIA Pascal architecture whitepaper (PASCAL. . . , 2016)

Table 2.1 summarizes this differences and facilitates the visualization of the parallelism level that a GPU powered by a Volta architecture can perform.

In a CUDA model programming language, the programmer can directly specify the GPU kernel size which describes the parallelism level for the specific program. A multithreaded CUDA program can be divided into sets of threads named *blocks* that specifies the kernel size as shown in figure 2.4. Depending on the quantity of available SMs, the blocks distribution may vary. The parallelism level depends on the availability of resources, such as the arithmetic units, and the blocks allocation within the SMs. Indeed, each block of threads can be scheduled on any of the available streaming multiprocessors within a GPU, in any order, concurrently or sequentially, so that a compiled CUDA program can execute on any number of multiprocessors (PROGRAMMING. . . , 2010 (accessed May 12, 2021)). It is important to mention that GPU Blocks are organized into a one-dimensional, two-dimensional, or three-dimensional grid of thread blocks. The dimensional aspect gives a unique accessible index within the kernel to identify separately

Figure 2.3 – Volta Streaming Multiprocessor (SM)


SM
L1 Instruction Cache
L0 Instruction Cache
Warp Scheduler (32 thread/clk)
Dispatch Unit (32 thread/clk)
Register File (16,384 x 32-bit)
FP64
INT
FP32
TENSOR CORE
LD/ ST
SFU
128KB L1 Data Cache / Shared Memory
Tex

Source: NVIDIA Volta architecture whitepaper (VOLTA. . . , 2017)

each block inside the grid. Finally, the number of thread blocks in a grid is usually dictated by the size of the data being processed (PROGRAMMING. . . , 2010 (accessed May 12, 2021)).

When comparing the compute capabilities between streaming multiprocessor in modern GPUs architectures, table 2.2 exposes that there are no considerably relevant differences between these GPUs architectures. This information demonstrates that parallelism capability basically remains the same in newer versions yet the number of cores almost doubled between Kepler and Volta architectures.

Table 2.1 – A comparison between NVIDIA Tesla GPUs

Architecture	**Kepler**	**Maxwell**	**Pascall**	**Volta**
GPU	GK180	GM200	GP100	GV100
SMs	15	24	56	80
FP32 Cores/SM	192	128	64	64
FP32 Cores/GPU	2880	3072	3584	5120
FP64 Cores/SM	64	4	32	32
FP64 Cores/GPU	960	96	1792	2560
FP64 Cores/GPU + FP32 Cores/GPU	3840	3168	5376	7680
Transistors	7.1 billion	8 billion	15.3 billion	21.1 billion
Manufacturing Process	28nm	28nm	16nm	12nm

Source: NVIDIA Volta architecture whitepaper (VOLTA. . . , 2017)

Table 2.2 – Compute Capability in modern architectural NVIDIA GPUs

Architecture	**Kepler**	**Maxwell**	**Pascall**	**Volta**
GPU	GK180	GM200	GP100	GV100
Threads/Warp	32	32	32	32
Max Warps/SM	64	64	64	64
Max Threads/SM	2048	2048	2048	2048
Max Thread Blocks/SM	16	32	32	32
Max Thread Block Size	1024	1024	1024	1024

Source: NVIDIA Volta architecture whitepaper (VOLTA. . . , 2017)

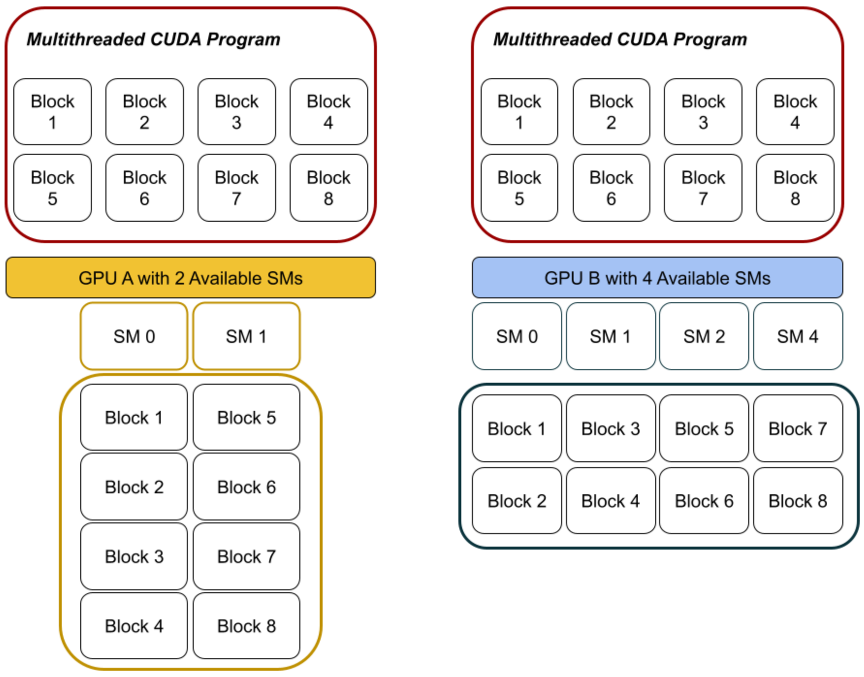
2.2 Vulnerability to transient faults

The preceding section has shown how powerful modern GPUs computational capabilities can be. The increasing number of Streaming Multiprocessors and processing units within these structures facilitates the understanding of why GP-GPUs are such potent engines. The GPU employed in this work can execute up to 32 parallel threads per warp distributed in 32 thread blocks within an SM providing up to 1024 threads per computing cycle. Even when a kernel is composed of a number of threads smaller than the warp size, the GPUs will dispatch the hole warp, since it is the minimal possible execution unit. If the number of threads is exceeded in a thread block, some of them will have their execution postponed.

In addition, Volta architecture improves both latency and bandwidth when compared to Pascal (VOLTA. . . , 2017) as the L1 cache size is more than 7x larger.

From a radiation point of view, the higher the number of processing units, the

Figure 2.4 – Blocks distribution


Multithreaded CUDA Program
Block 1
Block 2
Block 3
Block 4
Block 5
Block 6
Block 7
Block 8
GPU A with 2 Available SMs
SM 0
SM 1
Multithreaded CUDA Program
GPU B with 4 Available SMs
SM 0
SM 1
SM 2
SM 4

The same set of blocks in a multithreaded program distributed in two different GPUs in two different executions. The leftmost one has two available SMs, while the other one, four.

higher the potential of one of them being corrupted. Nonetheless such structures are isolated from each other which means that one single radiation-induced event will only corrupt the thread assigned to it (OLIVEIRA *et al.*, 2014). In a worst case scenario, the hole set of warp threads will be corrupted if the warp scheduler is affected. The GPU radiation response, however, demonstrates to be strictly related to resources distribution such as the chosen grid and block sizes (RECH; PILLA *et al.*, 2014) which have direct relation with the programmer decision.

Parallel programs are usually written to execute with greater number of blocks and threads than the maximum supported by GPUs capabilities. To provide optimal operation, the GPUs have internal complex warp schedulers that are responsible for implementing the threads parallelism. The scheduler assigns a number of blocks to each streaming multiprocessor depending on the resources needed by the block. If the grid size exceeds the maximum number of thread blocks that can be dispatched, some of the blocks are queued to execute whenever some SM becomes available. It is important to highlight that the queued block will only be dispatched if the GPU scheduler asserts that one or more SMs have already concluded their execution correctly. If a corruption occurs in some of these structures and the execution is not concluded the scheduler may present potential erroneous operation.

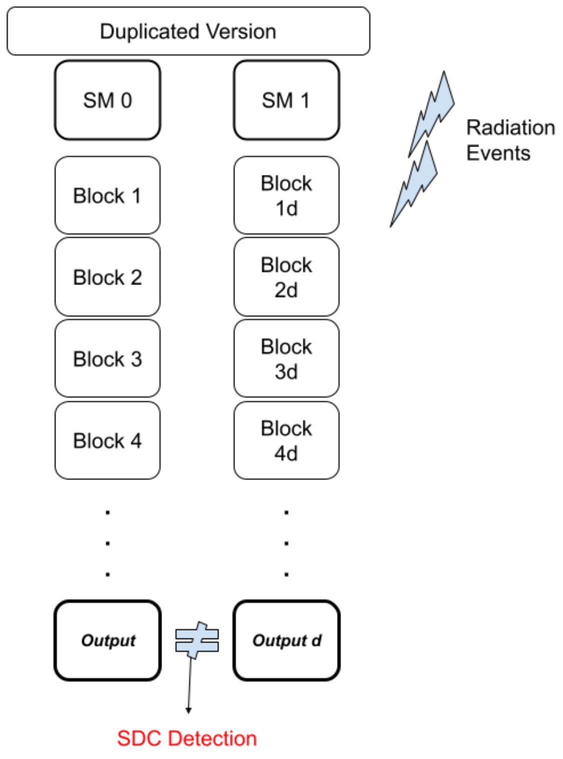
2.3 Duplication With Comparison

As shown in (OLIVEIRA *et al.*, 2014), both memory and logic GPU resources are sensitive to radiation events. In order to alleviate such effects, efficient solutions to reduce vulnerability must be implemented within GPUs. Hardening strategies are commonly adopted to deal with reliability issues. They consist of implementing both error detection and handling mechanisms which introduce intrinsic redundancy overheads. A complete software level hardening strategy can be designed as show in (RECH; AGUIAR *et al.*, 2013) and (PILLA *et al.*, 2014). If an error occurs the system is efficient on correcting it or triggering recomputation. Even so, for most applications such hardening solutions are not available requiring great effort to be implemented (OLIVEIRA *et al.*, 2014). In later sections, error detection techniques will be presented for which an complete hardening strategy were not implemented.

When discussing error detection, the Duplication With Comparison (DWC) technique is a very efficacious solution as it can detect more than 90% of transient faults in GPUs. It consists of a very simple software level implementation based on duplicating critical program operations or regions and comparing their outputs as an attempt of guaranteeing that any potential system corruption is detected. Usually, DWC implementation requires an exact copy of the critical code regions, re-executing them and saving each output resultant from each execution for further comparison. To decide whether a fault is detected or not, DWC detection technique checks if there were differences between both outputs and flags them if true. The methodology assures that the duplication points to the existence of a potential failure. This hardening strategy, even being software-implemented, refers to a hardware level fault tolerance technique. A simplified duplication and detection mechanism is illustrated in figure 2.5

In the GPUs scenario, DWC can be introduced as an instruction, block or thread duplication. The solution's simplicity and efficiency is overshadowed by the substantial overheads in power consumption, processing time and resources allocation specially when referring to its GPU implementation. These overheads have the potential to reach up to 2.5$\times$ higher in execution time as every critical instruction is executed at least 2 times. Even if advanced software strategies for instruction replication are employed these

Figure 2.5 – A simplified vision of the DWC detection strategy



In the *Duplicated Version* redundant blocks (represented with *d*) are executed and compared to the original version. If a difference is noticed, an SDC detection is raised

overheads still can be near of 1.7×. In the case of deep learning applications that require high resources usage, the overheads could be even greater than 2.5×. The more the GPUs are stressed, the more expensive the overheads can be. Thereafter, DWC shows to be ineffective for high processing computation and real-time safety critical applications (SANTOS *et al.*, 2020).

For further analysis and comparison, a brief demonstration of the applicable results found in (OLIVEIRA *et al.*, 2014) is made relevant for this proposal. Using the final outcomes in the detailed fault injection campaigns, the error detection rates demonstrated to be very effective. When exposed to a radiation flux similar to the one found in New York City (NYC), detailed in (OLIVEIRA *et al.*, 2014), the demonstrated benchmark executed in a NVIDIA Kepler GPU presented near 200 *Failures In Time* (FITs) which 100% of them resulted in *Silent Data Corruptions (SDCs)*. When enabling *ECC*, an NVIDIA error correction tool embedded in the GPU, the benchmark was able to recover from the majority of the errors originated. Still 22% of the SDCs occurred in the Plain version (program version without any hardening strategy) were still undetected while the FIT rate reduced considerably. With the introduction of the DWC hardening strategy, the benchmark showed great resilience. In short, more than 90% of the SDCs were detected and corrected. This first analysis shows the potential corruption impacts that such vulnerability can introduce in high processing applications, for instance, the referred benchmark when executed in a potent GPU.

3 PROPOSAL

As an alternative to the traditional DWC hardening solution for mixed precision GPUs, the Reduced Precision DWC (RP-DWC) (SANTOS *et al.*, 2020) proposes to optimize the overheads introduced by the former one. It consists of parallel executing the redundant operations using both *full precision (FP)* and *single precision (SP)* floating point units available in the referred GPUs architectures. While the original operation is executed with full precision operands, the replica is rounded to single precision and executed in underused less precise floating point units (FPUs). This model is the proposal solution adopted for this work and it is detailed in further sections. RP-DWC hardening strategy is illustrated in figure 3.2 and detailed in the *Implementation* section.

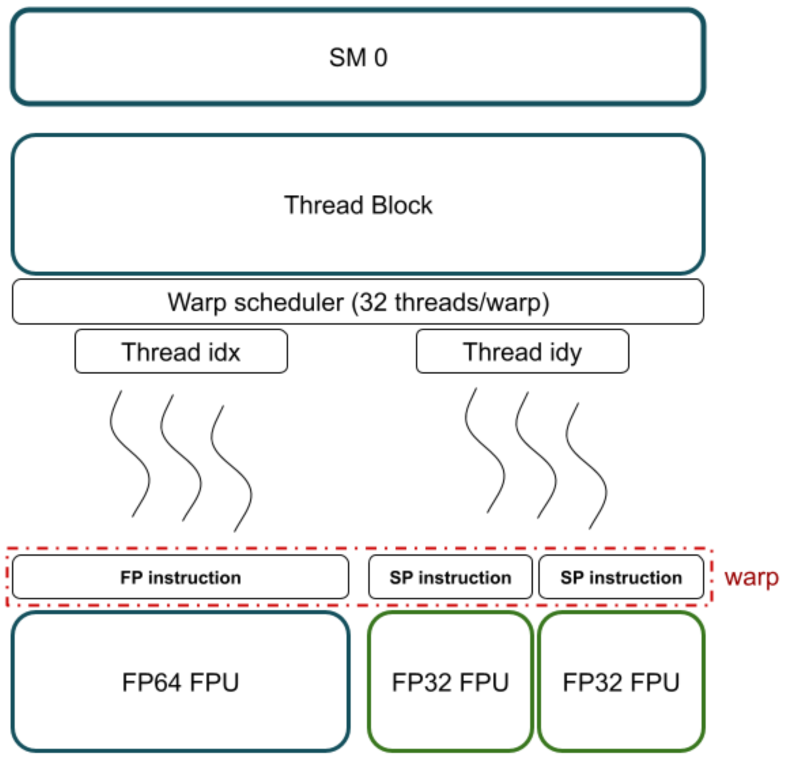
3.1 Definition

When discussing the purpose of GPUs utilization in computing applications one major attribute that makes these processing units stand out is the high parallelism level that they provide. In order to get the most out of GPUs powerful data processing is necessary that specific applications with a set of instructions that enables parallel computation is executed. Such applications, as mentioned earlier, are in their majority those ones that uses a great number of vector or matrix operations such as image processing ones. Matrix operations usually do not own relevant dependency levels between internal instructions which favors parallelism.

Aiming to enable a even greater level of parallelism improving the execution time in high processing applications without considerably reliability loss, RP-DWC provides a simple software level hardening solution for mixed precision GPUs. Image processing applications are an ideal example that could benefit from RP-DWC reliability capability as they tolerate a low level of data corruptions in their outputs. In this scenario, a bit differing from the original one does not have the same impact in an application where every single output bit has a major relevance. Due to human beings' various insensitivities to images with different frequencies, an acceptable error difference between two outputs can be established in order to preserve the correctness in human image interpretation (HSIEH; PENG; KU, 2013).

When mentioned in this work, *full precision (FP)* and *single precision (SP)* floating point operands refer to 64-bit and 32-bit instruction width respectively since these precisions are those adopted in referred FPUs. Considering figures 2.2 and 2.3, the GPU processing operation when using SP or FP floating point instructions remains the same. Both of them can be executed independently since the streaming multiprocessor has independent floating point units specific for each one and supposing that there are enough available resources. The difference that enables both execution and power consumption reduction resides in the number of parallel instructions that can be executed in the GPU pipeline. While executing one 64-bit floating point (FP64) instruction the pipeline can execute two 32-bit floating point (FP32) instructions. This procedure introduces intrinsic reduction both in total time execution and power consumption when compared to DWC hardening method. With precision reduction, since the number of instructions per warp increases, the total execution time is lessened. Consequently, the GPU spends less time processing operations which leads to a reduction in power consumption. Figure 3.1 summarizes the thread execution gain explored by RP-DWC method in a basic vision.

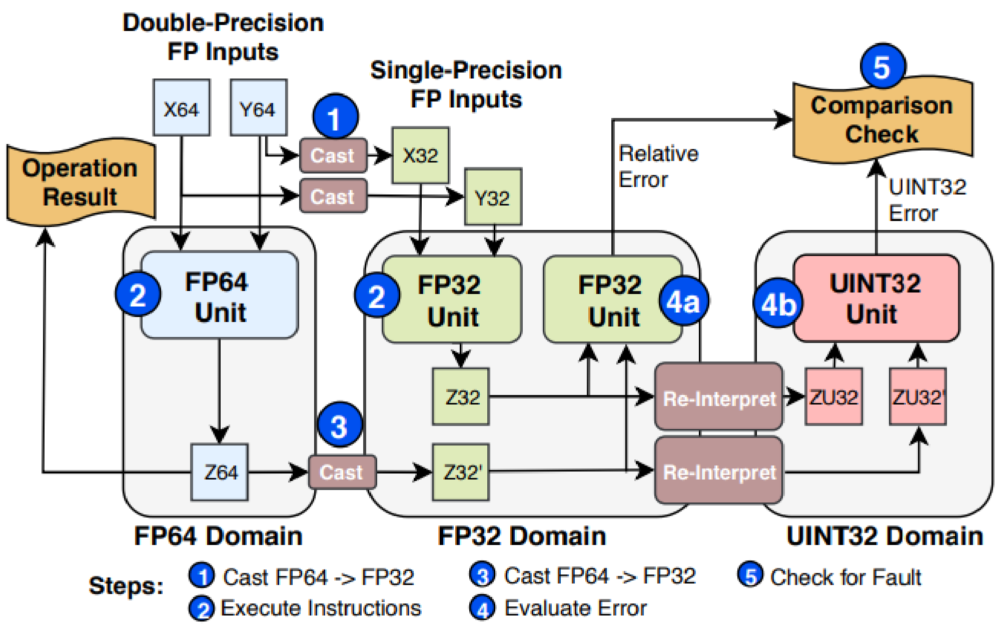
Figure 3.1 – A simplified pipeline view to illustrate the thread execution gain explored by RP-DWC hardening strategy



3.2 Implementation

Differing from the original DWC implementation where an instruction is executed twice, RP-DWC adds extra steps that must be followed to operate correctly. In this proposal it is assumed that all the original instructions are 64-bit large and in order to reduce them to 32-bit, casting steps are needed. Figure 3.2 illustrates in a high-level vision the RP-DWC procedure.

Figure 3.2 – A macro visualization of the RP-DWC strategy



Source: Reduced-Precision DWC for Mixed-Precision GPUs (SANTOS *et al.*, 2020)

Initially, the original and the redundant FP64 inputs are ready to be used. In DWC execution, both inputs would be executed in the FP64 FPUs, referred as *FP64 Domain* in the figure. Instead, in **step 1**, both inputs, $X64$ and $Y64$, must be casted to 32-bit precision ($X32$ and $Y32$). This first step can potentially introduce rounding errors as the difference between two nearby quantities with different precisions can be very large as shown in (GOLDBERG, 1991). Here, the NVIDIA rounding standard *round-to-nearest* method is employed since it was designed to reduce as much as possible the difference between the original data from the casted one.

Following the flowchart, in **step 2**, both FPUs execute the instructions with the respective precision considering the availability of hardware resources. As previously shown, while one 64-bit instruction is executed, another two 32-bit instructions can be carried out as well. When the instructions complete their executions, the outputs are created, represented by $Z64$ and $Z32$. In this stage of the process, the original output is set, since the original program 64-bit instructions have been accomplished. In this manner, the *Operation Result* is the $Z64$ output itself. **Step 3** consists of casting the original 64-bit output to 32-bit precision represented by $Z32'$ and dispatched to a 32-bit FPU.

The final steps of the process consists of comparing both FP32 outputs in **step 4a**. An additional evaluation of the output difference, an *integer (INT) unit*, is needed. This computation does not require any rounding operation, but instead an *re-interpretation* of the 32-bit output which considers *unsigned integer (UINT)* interpretation of the operand. **Step 4b** compares the re-interpreted values to find bit-level differences without considering the floating point structure. This comparison operation can be made in parallel since it makes use of a dedicated integer core and executes a simple 32-bit subtraction operation, improving even more the comparison operation and giving a better evaluation of the magnitude of the difference.

In general, in **step 4** occurs the evaluation of the output differences. When discussing the *Relative Error*, the difference between both FP32 outputs is given by $\sigma = \frac{Z32}{Z'32}$. Concurrently, the *Absolute Error (UINT32 Error)* is the absolute difference between them, represented by $\alpha = \mid Z32 - Z'32 \mid$. This evaluation is relevant because depending on the comparison result, the execution will raise or not a flag indicating that a potential failure occurred while the program was executing. In this work the *absolute difference* was adopted to handle the error detection since it presents faster execution and greater precision of the divergences. To decide whether each disparity must be treated as a failure, RP-DWC uses the concept of *Expected Precision Lost (EPL).*

3.2.1 Expected Precision Lost

An EPL can be defined as the intrinsic difference between two operands when precision reduction occurs. In traditional methodology the EPL is a number used to estimate an acceptable range where the difference may vary in order to distinguish an error due to a fault from the intrinsic divergence resulting from reduced precision computation. When the comparison is made the result may not be within the EPL's range and in this circumstance an exception is raised. This deviation can stop the program's execution or roll-back the computation to the operation that caused the execution to be redone.

The optimal EPL varies for each instance of an operation depending on its inputs. Calculating the ideal EPL in each iteration would require knowing the fault-free version of each different precision operation for every new input at runtime, which is, unfortunately, impractical in high processing applications (SANTOS *et al.*, 2020).

The EPL estimate is a very challenging and fundamental task to RP-DWC since the difference of each operation changes while the program is running depending on the inputs. Furthermore, the EPL determines whether the execution must handle or not an error. An *sub-optimal* EPL alternative, adopted in this work, is defined as an interval over which the outputs difference may diverge and no longer a number for each input and operation. This method can define an interval that is too narrow leading to ***false positives*** errors or too wide leading to ***false negatives*** ones when not properly calculated. In this solution an sub-optimal EPL interval is measured by observing the lowest and highest difference in a fault free code execution generating a very effective fault coverage. In order to corroborate with the *absolute comparison*, the EPL was defined as an unsigned integer 32-bit number.

3.2.2 Fault Injection

NVBitTFI is an architecture-level fault injection tool for GPUs resilience evaluation specially designed on top of a dynamic NVIDIA binary instrumentation library. In order to simulate an approximated realistic radiation environment and evaluate RP-DWC performance, NVBitTFI was adopted.

The usage of NVBitTFI allows the programmer to select instruction groups to evaluate how errors in them can propagate to the program's output, such as:

- Instructions that write to general purpose registers;
- Single precision floating point instructions;
- Double precision floating point instructions;
- Load instructions.

Besides, this tool implements different routines to modify the current bit value in one specific register, listed as below:

- Single bit-flip: one bit-flip in one register in one thread;
- Double bit-flip: bit-flips in two adjacent bits in one register in one thread;
- Random value: random value in one register in one thread;
- Zero value: zero out the value of one register in one thread.

In this project, in order to simplify the final analysis, the chosen instruction groups were *single precision*, *double precision* and *instructions that write to general purpose registers*. Only the *single bit-flip* injection was used.

The *fault injection campaigns* consisted of a total of 1200 fault injections per different benchmark, reaching up to more than 5h hours of execution. The total number of injections were equally divided by every different instruction group, resulting in 400 fault injections per group. Every single fault injection originates a different output log that contains detailed information about when the fault was introduced, which GPU kernel was executing at the time, to which group the affected instruction belongs and what was the final program output originated from the fault injection.

To evaluate the final result of a single fault injection, NVBitTFI classifies the outcome in three different categories: *Masked*, *Detected Unrecoverable Error (DUE)* and *Silent Data Corruption (SDC)*. The same concept presented in section 1.1 must be considered here to differentiate each of these outcomes. The injection evaluation relies on comparing the program output to a fault-free output generated in the beginning of the campaign, named *golden output*. If a difference between these two outputs is accused in the analysis stage of the injection the fault outcome is computed as an SDC. On the other hand, if no difference is noticed, the computation results in *Masked*. In turn, the majority of DUEs occurs when an injection:

- hangs the program execution until the maximum execution time configured is reached, resulting in a DUE timeout;
- tries to access an illegal memory address.

The results analysis in the following section is attached to the concepts of the fault injection campaign previously delineated.

3.2.3 Benchmarks

For the experiments, a Tesla GPU with Volta Architecture, the NVIDIA Titan V was used. The aforementioned GPU features dedicated hardware resources that speed up instruction execution in both 32-bit and 64-bit precision units as show in figure 2.3. The metrics measured from the results provided information to evaluate both reliability and execution time. The results obtained from the fault injection campaign were used to generate the RP-DWC reliability analysis, while a NVIDIA profiler embedded in the CUDA

Toolkit (CUDA..., 2021 (accessed May 12, 2021)) package for CUDA programming was utilized to analyze the computational performance in execution time for RP-DWC.

Ahead of presenting the experiment benchmarks, is important to mention that the full Duplication With Comparison method was also implemented for every chosen program in order to create better results for the study. Moreover, all the benchmarks thread block sizes configurations were set accordingly to the input matrices of each program in order to provide greater execution capacity.

Hotspot is a well known application that requires HPC capabilities. This algorithm consists of solving a great number of differential equations that calculate the estimated processor surface temperature given an architectural floorplan and simulated power measurements. The surface is divided into a number of *spots*; each spot is represented as an matrix index input (*i.e* a matrix of size *1024 X 1024* results in *1,048,576* different spots).

To compute the surface temperature in a specific spot, Hotspot calculates an average value considering the estimated temperature at the neighbor spots. This procedure can potentially mask a least significant bit difference at one spot considering that the averaging calculation reduces the precision impact compared to a single element alone. Therefore, when computing the Hotspot RP-DWC *EPL* the *absolute comparison* of each output presented a very low difference in the worst case. The chosen input size was a matrix of size $1024\ X\ 1024$, the maximum provided by the Rodinia Benchmark Suite (CHE; BOYER *et al.*, 2009)-(CHE; SHEAFFER *et al.*, 2010). The block size adopted was not set as the optimal size to avoid thread queuing since the GPU occupancy evaluation for the RP-DWC version of the benchmark showed not to be critical.

Nonnegative Matrix Factorization (NMF) calculates the factorization of the matrix *V* into two other matrices, *W* and *H*, so that all matrices have no negative elements. Matrix factorization consists basically of decomposing a matrix into the product of two lower dimensionality rectangular matrices. For the given experiments, the matrices sizes were larger than the original benchmark provided by the Vienna Computing Library (RUPP *et al.*, 2016), since greater sizes allow a facilitated evaluation of the execution time comparison between RP-DWC and DWC techniques. The *EPL* in this case resulted from the intrinsic difference between the rounding operations of RP-DWC. Since NMF does not present the same averaging treatment as the Hotspot benchmark the *EPL* was also greater than the former one. Finally, the thread block size appeared to have low impact in the algorithm performance, since the matrices sizes adopted were relatively small and the GPU occupancy for the benchmark was also not critical. The block size was chosen to fit one thread per output index (*i.e* a factorized matrix size of *100 X 60* results in *6000* different threads divided in blocks of *60* threads each).

Conjugate Gradient Solver (CGS) proposes to solve a system of linear equations in the $Ax = b$ form where *A* is a well known *n x n* symmetric matrix (when the transpose equals to the original matrix) and also *positive-definite* (*i.e* $x^T Ax > 0$ for all non-zero vectors *x*). In this benchmark, a GPU accelerated solution is proposed by the NVIDIA CUDA Samples (CUDA. . . , 2018 (accessed May 12, 2021)).

Since the benchmark is an iterative algebric linear system solution, the algorithm consists of a great number of matrix operations that use the previous result to calculate the output for the next iteration. This treatment potentiates the propagation of an intrinsic precision difference between the original value and the less precise redundant one in RP-DWC, specially if the difference occurs in an early stage of the iterations. In order to mitigate this potential inequality the precision used in the full precision method was reduced. This decision softened the precision loss effects in the most significant bits of the output. Nevertheless, it also seemed to have no impact when compared to the original program, which doesn't use a representation where all the output values require the 64-bit precision. Instead, a much smaller quantity of bits is necessary to correctly represent the resultant numeric values.

Similar to the Hotspot and the NMF, the CGS RP-DWC solution doesn't stress the GPU resources when using the default block size configuration of the benchmark which is *512* threads per block. Since the output matrix size is of $1024\ X\ 1024$, increasing the number of threads per block and the number of blocks to *1024* could be possible. This would theoretically elevate the parallelism level. Yet, it was observed that augmenting the block size caused the benchmark to stress the GPU resources to the limit, which could lead to a potential commitment of the multithreaded parallel execution.

Bisect is an abbreviation for the *bisection algorithm*, a simple method for finding the eigenvalues of a tridiagonal matrix. The algorithm routine is to implement a root-finding method to any continuous function defined on a given interval $[a, b]$, where $f(a)$ and $f(b)$ have opposite signs. For a provided input size, the algorithm generates a random tridiagonal matrix along with the respective super-diagonal. By having these values, the algorithm is capable of describing a continuous function and calculating the respective eigenvalues (RUPP *et al.*, 2016).

However, when implementing the method on a computer system such as a GPU, there can be complications with finite precision. Even though $f(x)$ is continuous, finite precision may preclude a function value from ever being zero, for example when using the π value in a function and trying to find an x value that equals π. Unfortunately this equality is not precise in finite computation, which leads to the first problem found in *Bisect*. In order to correctly evaluate the algorithm, additional verification tests are necessary, which harms the high processing application, and thus, they were not implemented in the experiment.

After executing the fault injection campaign in Bisect it was possible to notice that the vast majority of the fault injections led to SDCs. Not enough, a great number of *false positives* was detected. This happens when the result of a fault injection is not computed as an SDC, but the comparison algorithm accuses a difference greater than the *EPL*. Even when increasing the EPL interval, the algorithm continued to show extreme sensitivity to every single fault injection. Due to this behaviour the Bisect reliability evaluation was not satisfactory, but the benchmark was not discarded yet.

When evaluating the execution time and the GPU allocation resources to either the DWC and RP-DWC version, relevant information was noted. Despite the low resource utilization in both methods, the Bisect did not show any improvement in the execution time from the DWC version to the RP-DWC which was unexpected. After investigating the reason for this behaviour, it is supposed that there are a synchronization mechanism within the benchmark which blocks and hangs the parallel kernel execution until the first kernel is not completed, resulting in a sequential execution procedure. Finally, it was decided that the Bisection-algorithm would be detailed as a counterexample in this analysis.

Table 3.1 – A summary of the benchmarks attributes

Benchmark	*UINT32 EPL*	*Matrix Input Size*	*Thread Block Size*
Hotspot	1	1024 X 1024	1024
Conjugate Gradient Solver (CGS)	4	1024 X 1024	512
Nonnegative Matrix Factorization (NMF)	41	100 X 60	60

The main attributes of each experimental benchmark for results analysis

4 RESULTS

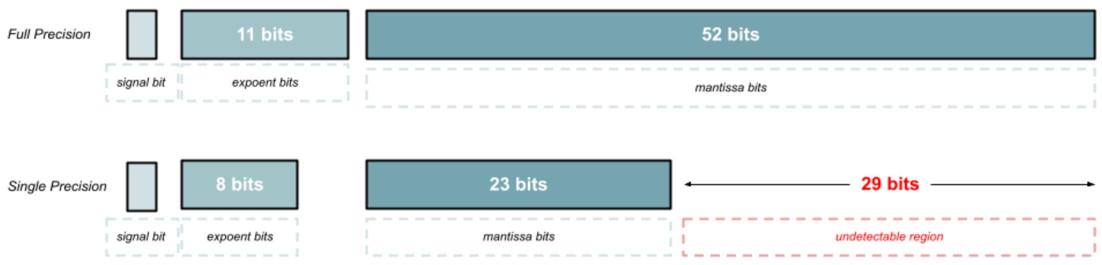
This final chapter presents the experiments results from both original DWC and RP-DWC benchmarks. They are divided into two major groups: the *reliability* metric and the *performance* metric. The first is based on the concepts of detected and undetected Silent Data Corruptions (SDCs), whereas the second is greatly influenced by the execution time. Here we refer to both redundancy techniques as their abbreviations, DWC and RP-DWC. Whenever the *reliability* metric is cited, it refers to the SDC detection capability of each method. On the other hand, the *performance* metric is related to the execution time of the benchmarks when using the full precision redundancy and the single precision one.

In the experiments, every execution was completed even when an error was detected since the goal was to evaluate and compare the error detection rate in both hardening strategies. Moreover, it was considered an error any difference greater than the minimal tolerance between the computed output and the expected one. It was adopted as minimal tolerance for RP-DWC the maximum difference between a fault-free output resultant from: a *full precision* execution of the algorithm, and a *single precision* one. For the DWC method since there is no precision loss and consequently no differences originated from this, there was no minimal tolerance adopted.

In order to demonstrate the intrinsic undetected errors for the RP-DWC technique, figure 4.1 illustrates the bits where the method is incapable to detect differences. By demonstrating the *IEEE 754* floating point formats it becomes clearer to visualize this characteristic. The 29 least significant bits from the full precision operand's *mantissa* compose the undetectable region. Since these bits may not be represented in a single precision format it becomes impractical to detect any bit difference in this region. Therefore, if any corruption occurs in one of those bits it will be intrinsically undetected by the RP-DWC strategy.

Figures 4.2 to 4.4 illustrate the detection rate in every benchmark for both RP-DWC (left column) and DWC (right column). For a total of $1,200$ fault injections per campaign a number of SDCs was detected. 100% of detection means that every single fault that generated an SDC was detected. As expected, DWC detects a relatively larger number of SDCs since no precision reduction is required and consequently the implications caused by the bit-loss are non-existent. For both NMF (Figure 4.3) and Hotspot (Figure 4.2) the DWC detection rates are greater than 90% and for CGS (Figure 4.4) it is approximately 80%. This can be explained from the CGS algorithm sensitivity when

Figure 4.1 – RP-DWC undetectable region

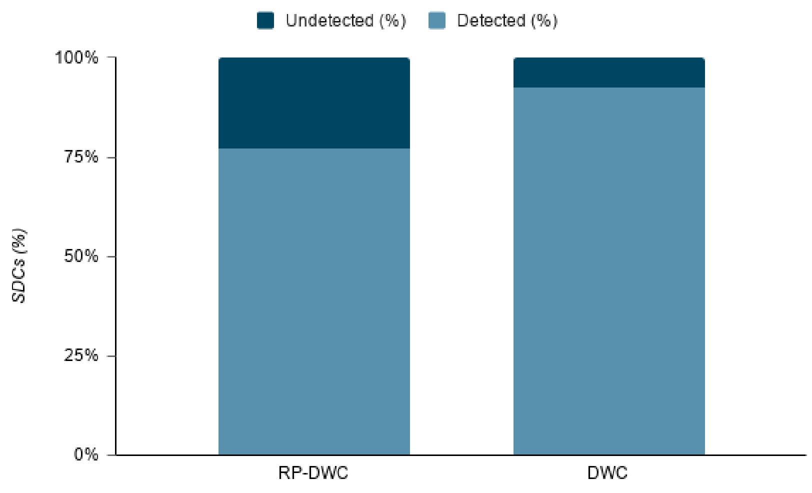


bit corruptions are introduced, as detailed in the benchmark section. When it comes to the RP-DWC it can be observed that the undetected errors increase for all benchmarks. However for the Hotspot and the NMF the detecion rates remains high which means that for both algorithms the precision reduction method adopted did not introduced significant differences.

The CGS results prove that the algorithm is incredibly sensitive to faults, which becomes even more evident when precision reduction is added to it. Even if presenting the worst reliability results between the benchmarks, the detection rates for both DWC and RP-DWC are unsatisfactory, that reduces in this way the negative impact of the RP-DWC. The DWC also detects a low number of errors when compared to the other DWC benchmark results.

The RP-DWC technique introduced the expected reduction in the algorithms execution time as demonstrated in figures 4-5 to 4.7. Considering the: *high availability* for the single precision FPUs since originally the algorithms were implemented to operate with 64 bits; and the *floating point computing capacity* which is considerably higher for the FP32 instructions (VOLTA. . . , 2017) a summarized comparison is demonstrated in table 4.1. The respective table shows that, for the GPUs Volta architecture, exists a greater number of 32-bit floating point units per GPU. This favors the RP-DWC since the technique makes a balanced use of both FP64 and FP32 units while the DWC uses, in the majority, 64-bit FPUs. Ultimately, the *peak TFLOPS* exposes that the FP32 instructions take advantage over the full precision ones. This metric describes the maximum number

Figure 4.2 – Hotspot Fault Detection



of the respective floating point instructions that the GPU can execute in one second.

Table 4.1 – Comparison between FP32 and FP64 instructions in Volta

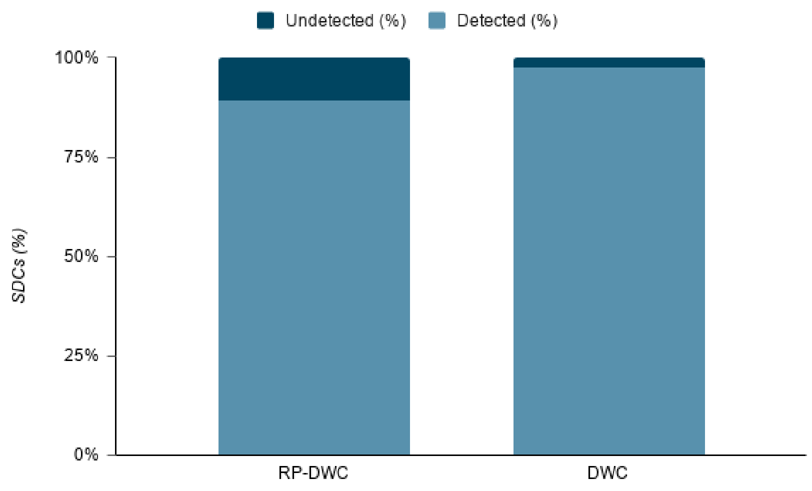
Method	**DWC**	**RP-DWC**
FPUs Cores/GPU	2560 FP64	2560 FP64 + 5120 FP32
Peak TFLOPS	7.8 (FP64)	15.7 (FP32)

Source: NVIDIA Volta architecture whitepaper (VOLTA. . . , 2017)

The data from the tables proves the gain that comes with the precision reduction in execution time. Original kernel time refers to the kernels execution time without any redundant technique applied. DWC and RP-DWC kernel time, in turn, consider each duplication technique respectively. In average, more than 40% of the total time spent is saved when compared to the DWC technique. The metrics are divided into *kernel time* and *comparison time*. The former one is related to the period the algorithm spends to compute the GPUs kernels without considering the memory instructions while the last references the comparison operations to detect errors at the end of both methods.

It is observed different behavior in the comparison time when correlating the three benchmarks. Hotspot has a very expensive kernel particularly when the full precision redundancy is applied. The comparison method for both techniques when compared to the kernel execution time are also considerable. This can be explained by the relevance of the time the method spends comparing the outputs with the kernel execution time itself. In Hotspot specially, the kernel seems to be less complex than the other benchmarks. The

Figure 4.3 – NMF Fault Detection



NMF benchmark presents very expensive kernels and relatively fast comparison operations due to the modest matrices sizes. This characteristic minimizes the comparison time when compared to the original kernel, leading to the removal of this data from the exhibition. Finally, it is possible to conclude from these results that the RP-DWC hardening strategy introduces very interesting performance gain in both NMF and CGS benchmarks. The total time in this cases is less than 200% which suggests that even with the duplicated versions of the kernel and the comparison method, the total time is not even doubled from the original kernel. On the other hand, with the DWC technique, this barrier is surpassed only considering the kernel time. When adding the comparison time to this equation, DWC becomes more expensive. The Hotspot algorithm also seems to benefit from the RP-DWC adoption. Whereas presenting greater overheads in the comparison step, the redundancy itself granted interesting performance gain.

Figure 4.4 – CGS Fault Detection

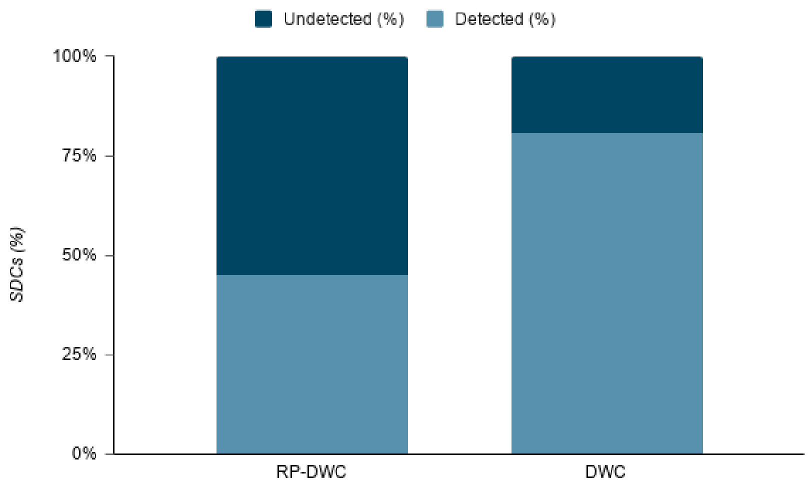


Figure 4.5 – Hotspot Performance Results

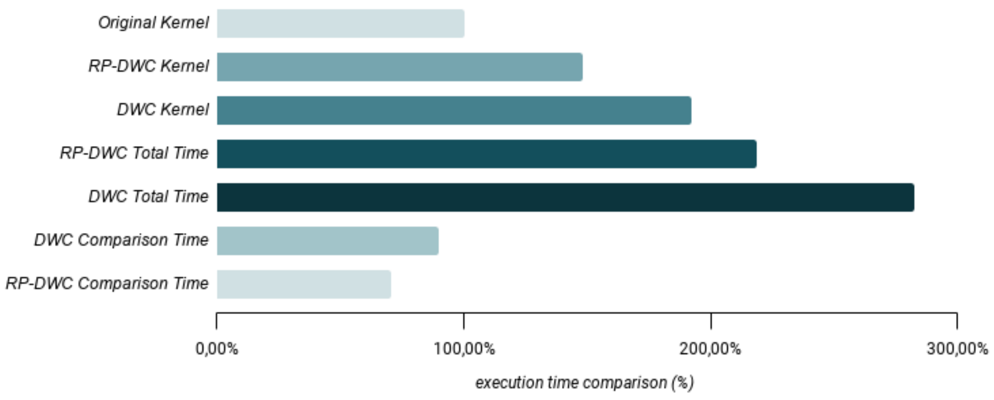


Figure 4.6 – NMF Performance Results

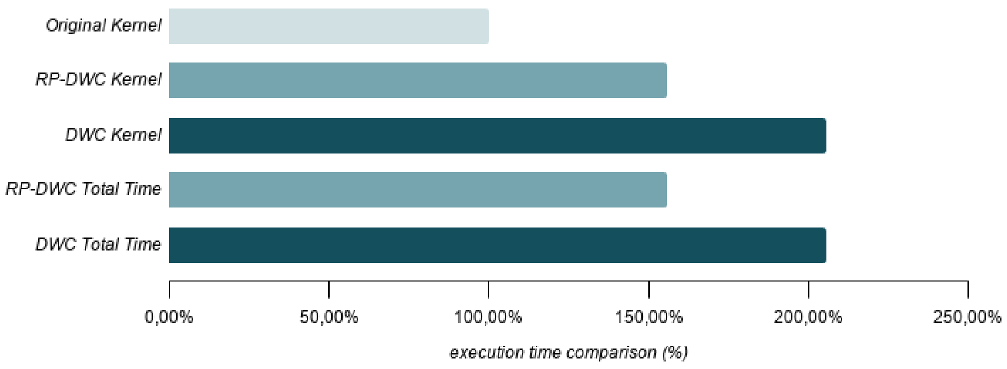
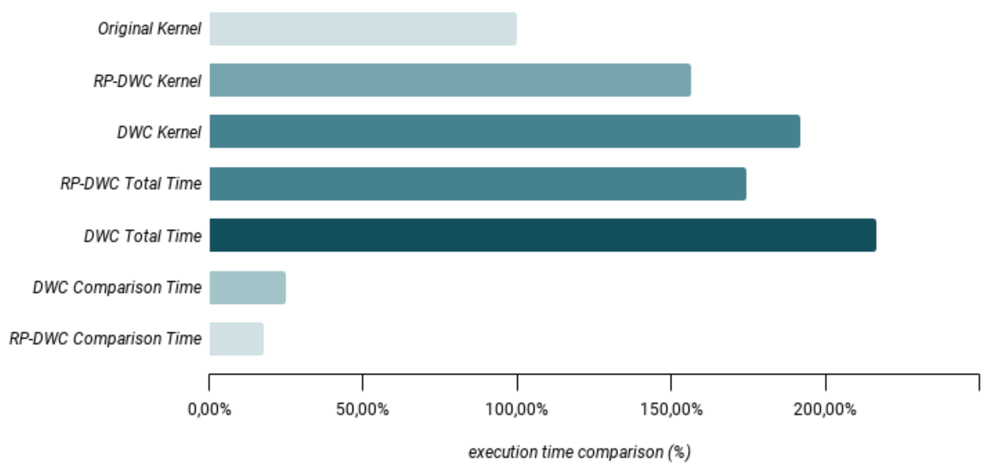


Figure 4.7 – CGS Performance Results



5 CONCLUSION

When referring to high-computing applications in mixed precision GPUs, the reduced precision technique to maintain reliability even in the presence of radiation events can introduce interesting benefits. The RP-DWC technique requires no complex code modification, being necessary only to introduce the redundancy and the comparison methods. Besides, the performance gain when compared to the original DWC technique provides more satisfactory execution time for these applications, a considerable metric when powerful GPUs architectures like Volta are adopted. Moreover, the detection rate with precision reduced seems to have a small impact in the final outputs.

As cited previously, one of the main challenges found in RP-DWC is to maintain acceptable error detection rates even with the intrinsic limitation that comes with the precision loss. Finally, to improve fault coverage and reduce the sensitivity when a bit corruption occurs, better floating point handling approaches might be studied and employed.

REFERENCES

ACCELERATING the Rate of Scientific Discovery. [*S. l.*], 2021 (accessed May 12, 2021). Disponível em: `https://developer.nvidia.com/hpc`.

ANDERSON, T.; LEE, P. A. Fault tolerance-principles and practice. **Englewood Cliffs, Prentice-Hall.**, 1981. DOI: `10.1007/978-3-7091-8990-0`.

AVIZIENS, A. Infraestructure-based design of fault-tolerant systems. **Proceedings of the IFIP International Workshop on Dependable Computingand its Applications.**, p. 51–69, 1998.

BAUMANN, R.C. Radiation-Induced Soft Errors in Advanced Semiconductor Technologies. **IEEE TRANSACTIONS ON DEVICE AND MATERIALS RELIABILITY**, v. 5, n. 3, 2005. DOI: `10.1109/TDMR.2005.853449`.

CHE, Shuai; BOYER, Michael *et al.* Rodinia: A benchmark suite for heterogeneous computing. *In*. 2009 IEEE International Symposium on Workload Characterization (IISWC). [*S. l.: s. n.*], 2009. p. 44–54. DOI: `10.1109/IISWC.2009.5306797`.

CHE, Shuai; SHEAFFER, Jeremy W. *et al.* A characterization of the Rodinia benchmark suite with comparison to contemporary CMP workloads. *In*. IEEE International Symposium on Workload Characterization (IISWC'10). [*S. l.: s. n.*], 2010. p. 1–11. DOI: `10.1109/IISWC.2010.5650274`.

CHEN, Liming; AVIZIENIS, A. N-VERSION PROGRAMMINC: A FAULT-TOLERANCE APPROACH TO REL1AB1L1TY OF SOFTWARE OPERAT1ON, p. 113–, 1995. DOI: `10.1109/FTCSH.1995.532621`.

CUDA. [*S. l.*], 2021 (accessed May 12, 2021). Disponível em: `https://developer.nvidia.com/cuda-zone`.

CUDA Samples Benchmark. [*S. l.*], 2018 (accessed May 12, 2021). Disponível em: `https://github.com/NVIDIA/cuda-samples`.

EUSGELD, Irene *et al.* Software Reliability, p. 104–125, jan. 2005. DOI: `10.1007/978-3-540-68947-8_10`.

GOLDBERG, David. What Every Computer Scientist Should Know About Floating-Point Arithmetic. **ACM Computing Surveys**, v. 23, n. 1, 1991.

GONÇALVES DE OLIVEIRA, Daniel Alfonso Gonçalves *et al.* Evaluation and Mitigation of Radiation-Induced Soft Errors in Graphics Processing Units. **IEEE**

Transactions on Computers, v. 65, n. 3, p. 791–804, 2016. DOI: `10.1109/TC.2015.2444855`.

HSIEH, Tong-Yu; PENG, Yi-Han; KU, Chia-Chi. An Efficient Test Methodology for Image Processing Applications Based on Error-Tolerance. *In*. 2013 22nd Asian Test Symposium. [*S. l.: s. n.*], 2013. p. 289–294. DOI: `10.1109/ATS.2013.60`.

JEDEC. Measurement and Reporting of Alpha Particle and Terrestrial Cosmic Ray-Induced Soft Errors in Semiconductor Devices. **JEDEC Standard, Tech. Rep. JESD89A**, 2006.

KNIGHT, J.C. Safety critical systems: challenges and directions. *In*. PROCEEDINGS of the 24th International Conference on Software Engineering. ICSE 2002. [*S. l.: s. n.*], 2002. p. 547–550.

LAPRIE, J-C. DEPENDABLE COMPUTING AND FAULT TOLERANCE : CONCEPTS AND TERMINOLOGY. *In*. TWENTY-FIFTH International Symposium on Fault-Tolerant Computing, 1995, ' Highlights from Twenty-Five Years'. [*S. l.: s. n.*], 1995. p. 2–. DOI: `10.1109/FTCSH.1995.532603`.

LUCAS, Robert *et al.* DOE Advanced Scientific Computing Advisory Subcommittee (ASCAC) Report: Top Ten Exascale Research Challenges, fev. 2014. DOI: `10.2172/1222713`. Disponível em: `https://www.osti.gov/biblio/1222713`.

MAHMOUD, Abdulrahman *et al.* Optimizing Software-Directed Instruction Replication for GPU Error Detection. *In*. SC18: International Conference for High Performance Computing, Networking, Storage and Analysis. [*S. l.: s. n.*], 2018. p. 842–854. DOI: `10.1109/SC.2018.00070`.

NEWMANN, J. Von. Probabilistic logics and the synthesis of reliableorganisms from unreliable components. **Automata Studies, Shannon & McCarthy eds. Princeton Univ. Press, 1956. p. 43-98.**, 1956.

OLIVEIRA, Daniel A. G. *et al.* Modern gpus radiation sensitivity evaluation and mitigation through duplication with comparison. **IEEE Transactions on Nuclear Science**, v. 61, n. 6, p. 3115–3122, 2014. DOI: `10.1109/TNS.2014.2362014`.

PASCAL architecture whitepaper. [*S. l.*], 2016. Disponível em: `https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf`.

PILLA, L. L. *et al.* Software-Based Hardening Strategies for Neutron Sensitive FFT Algorithms on GPUs. **IEEE Transactions on Nuclear Science**, v. 61, n. 4, p. 1874–1880, 2014. DOI: `10.1109/TNS.2014.2301768`.

PROGRAMMING Guide. [*S. l.*], 2010 (accessed May 12, 2021). Disponível em: `https://docs.nvidia.com/cuda/cuda-c-programming-guide/`.

RECH, P.; AGUIAR, C. *et al.* An Efficient and Experimentally Tuned Software-Based Hardening Strategy for Matrix Multiplication on GPUs. **IEEE Transactions on Nuclear Science**, v. 60, n. 4, p. 2797–2804, 2013. DOI: `10.1109/TNS.2013.2252625`.

RECH, P.; PILLA, L.L. *et al.* Impact of GPUs Parallelism Management on Safety-Critical and HPC Applications Reliability, p. 455–466, 2014. DOI: `10.1109/DSN.2014.49`.

RUPP, Karl *et al.* ViennaCL - Linear Algebra Library for Multi- and Many-Core Architectures. **SIAM Journal on Scientific Computing**, v. 38, s412–s439, jan. 2016. DOI: `10.1137/15M1026419`.

SANTOS, Fernando Fernandes dos *et al.* Reduced-Precision DWC for Mixed-Precision GPUs. *In*. 2020 IEEE 26th International Symposium on On-Line Testing and Robust System Design (IOLTS). [*S. l.: s. n.*], 2020. p. 1–6. DOI: `10.1109/IOLTS50870.2020.9159748`.

VOLTA architecture whitepaper. [*S. l.*], 2017. Disponível em: `https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf`.

WEBER, Taisy Silva. Um roteiro para exploração dos conceitos básicos de tolerância a falhas. **Instituto de Informática – UFRGS, Curso de Especialização em Redes e Sistemas Distribuídos.**, 2002.