A Two-Level Fault-Tolerance Technique for High Performance Computing Applications

Aishah M. Aseeri[1], Mai A. Fadel[2]
Faculty of Computing and Information Technology
King Abdulaziz University, KSA

***Abstract*—Reliability is the biggest concern facing future extreme-scale, high performance computing (HPC) systems. Within the current generation of HPC systems, projections suggest that errors will occur with very high rates in future systems. Thus, it is fundamental that we detect errors that can cause the failure of important applications, such as scientific ones. In this paper, we have presented a two-level fault-tolerance approach for the detection and classification of errors for Compute United Device Architecture (CUDA)-based Graphics Processing Units (GPUs). In the first level, it detects the existence of errors by using software redundancy that applies design diversity. In the second level, it investigates the problematic software version and re-executes it on a different hardware component to classify whether the error is a permanent hardware error or a software error. We implemented our approach to run on GPUs and conducted proof of concept experiments by running three versions of matrix multiplications with different error scenarios and results show the feasibility of the proposed approach.**

***Keywords*—High performance computing; fault tolerance; graphics processing units (GPUS); error detection; n-version programming (NVP); multi-GPU; reliability**

I. Introduction

High performance computing (HPC) is a term used in reference to integrated computing environments that rely on parallel processing in the running of applications. This boosts efficiency, speed, and reliability, while ensuring that complex scientific problems can be solver faster than if they were performed serially.

HPC systems are used to resolve complex scientific problems that, because of memory or computer performance limitations, either cannot be solved or are impractical to solve using traditional computing systems.

These systems promise to push the boundaries for scientists by augmenting their research across a range of disciplines, including: chemistry, nuclear physics, high energy, astrophysics, nanotechnology, biology, medicine, and material sciences [1]. However, to realize the full potential and reach the breakthroughs of this technology, software development tools are of great importance, such as compilers and debuggers; to be more specific test frameworks are among tools that should be part of the HPC infrastructure [2]. Test frameworks are becoming increasingly important as resilience is one of the major challenges to the growth of the complex systems mentioned above. System resilience is substantially reduced due to the increase in the number of components, regardless of the reliability and efficiency of individual components. Besides the addition of more components, many other factors increase the rate of failure for future HPC applications, including: number of components both memory and processors, smaller circuit sizes, heterogeneous systems, the number of operations, and increasing system and algorithm complexity [3]. This leads to the fact that hardware faults are becoming inevitable [4, 5] and the way is to be aware of and handle its effects [6]. From another point of view, as HPC power is targeting applications beyond the graphics domain, such as scientific applications and stock markets, it faces the challenge of addressing the need to generate accurate results that should be free of errors, as these applications cannot tolerate the existence of errors as graphical applications [7]. Hard errors are not the only concern of the HPC community, soft errors are a concern as well [8]. In [9] a study done on the data of two large-scale sites of a set of systems showed that hardware and software errors covering a considerable large proportion of root causes of failures. Hence, it is imperative to provide effective fault-tolerance capabilities, both at hardware and software levels as part of the test framework. HPC community has developed various solutions to generally tolerate faults, and more specifically to mitigate faults caused by hardware defects [10] and to detect and recover from errors [5, 11]. We will elaborate more on some of the relevant approaches in Section 2. Some of the used approaches depend on using checkpoints/reset [12], redundancy and Algorithm-Based Fault Tolerance ABFT [13, 14]. In our research, we have applied redundancy-based fault tolerance, as checkpointing has high communication overhead and ABFT is customized to fit the algorithm under analysis, thus, it is very difficult to generalize the solution to other applications without addressing the specifics of the new algorithm. In particular, we use software-based redundancy with design diversity; that is, we provide several versions of the same application that differ in their design to check for errors during execution time. Design diversity lessens the likelihood of having all versions fail exactly the same way in the same time. We use this technique in a broader view, as we aim to support the need to detect not only software errors but through these errors we can detect if the actual cause is a hardware error. This two-level approach starts by applying software-based redundancy with design diversity to identify the existence of a problematic copy of the software, then re-execute this copy on a different hardware to determine if the original hardware was the cause of the error or the software itself has an error.

In the following section, we present some of the research related to our work. In Section 3, we briefly describe some of the basics of CUDA-based GPU Architecture and Open MP programming, as they are the main tools of the infrastructure that we used to implement our system. In Section 4, we present our proposed methodology. In Section 5, we present the experimental results, and finally we conclude our paper in Section 6 and highlight some of future research directions.

II. Related Work

This section presents existing error detection techniques based on redundancy that are considered one of the protective techniques that provide resilient computing in the HPC domain. Many approaches based on hardware redundancy have been used successfully in mission-critical systems such as triple-modular redundancy (TMR) and dual-modular redundancy (DMR) [15]. The latter approach is achieved by supplying two similar physical components that can execute the same task. When an error occurs, the extra component transparently recovers from the peer one [16]. A TMR approach is based on three fully redundant components which perform the same process. The result is processed by a voting system to ensure the results are the same. If one component fails, the other two can correct and mask the fault. This approach causes performance overhead because of the need to synchronize original hardware and its replica and also doubles the hardware cost. In addition, running the same copy of the software on all components will not reveal an actual error, as all copies will generate the same incorrect result.

Design diversity among the software replicas is implemented as a solution for this problem. Thus, lessens the likelihood of having all copies failing on the same set of input data. In [17], this approach is categorized as software redundancy. This method has been widely exploited in targeting software errors, i.e., design faults or software bugs [18].

There are several approaches to software redundancy techniques, such as N-version programming [19], recovery blocks [20] and N self-checking [21]. Faults can be detected in these approaches by consistency checking/self-checking or time redundancy. Time redundancy is defined as running the same program several times and compare the results. All the above approaches target sequential applications, as for redundancy-based fault detection approaches that target concurrent applications running on GPUs can be found in [11, 22, 23, 24]. These approaches detect software errors whether they use software or hardware redundancy. In [25], the proposed system detects hardware errors using different types of redundancy.

From a different perspective, part or our method is to execute the problematic version of the software on different hardware to classify whether the error is caused by hardware or software. This idea has been applied in the SWAT tool [26] which will be discussed further in Section VI.

It is noticed that benefiting from software redundancy with design diversity is applied in several researches, to either detect software or hardware errors. However, its power has not been integrated with the step of classifying the error. Up to our knowledge no one applied design diversity in HPC for detecting errors and also no one used software redundancy for detecting hardware errors on GPUs.

III. Basic Concepts of Dependability

We now briefly present basic ideas and terminology used in the field of fault tolerance. A detailed background and taxonomy of the related terms can be found in [27, 28].

A. Fault-Error-Failure

A system is an entity that interacts with other entities. A system can be hardware based, for example a processor, or software based, such as a running application. A system consists of components which can be systems themselves. A system failure is defined as the deviation of the system behavior that is inconsistent with the system's specification. When the observed behavior differs from the specified behavior, we call it a failure. A failure occurs because of an error that is caused by a fault. An error is the part of the system state which results from the activation of a fault and causes the system to be in an illegal state. Errors are liable to lead to a failure. Fault propagation chain from faults to failures in a system is illustrated in Fig 1.

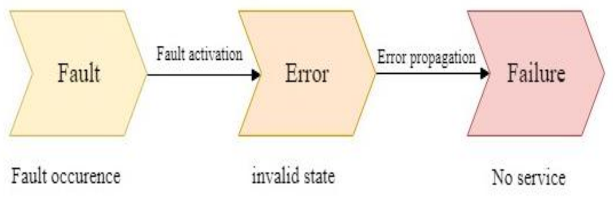


Fig. 1. Relationship between Fault, Error and Failure.

There are numerous sources of a fault that can be either software or hardware [29] as shown in Fig. 2.

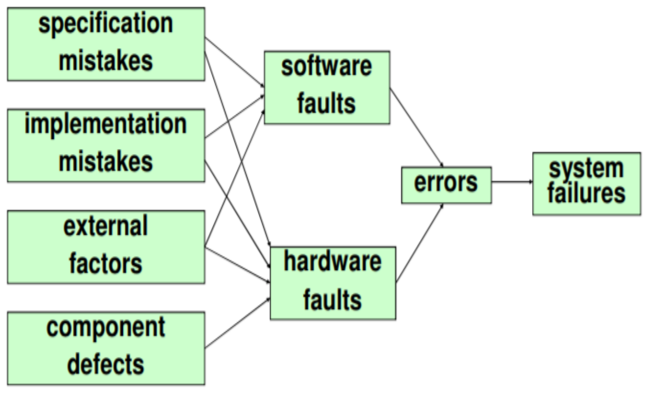


Fig. 2. Classification of the Sources of Faults [29].

Software faults are most often caused by design faults and operational faults [29]. Design faults occur when a designer, either misunderstands a specification or simply makes a mistake. Hardware faults are most often caused by incorrect specification, incorrect implementation, manufacturing imperfections or external factors.

System errors that impact the application's and supercomputer's reliability can be classified as "soft" or "hard": soft errors are usually caused by a transient fault and temporary environmental factors. Soft errors, unlike manufacturing or design faults, do not occur consistently. Some of the factors that can cause this type of faults are radiation-induced upsets in electronic circuits [27, 30], leakage from adjacent circuits, timing violations, and improper signal routing or power design [31]. These events do not cause permanent physical damage to the processor but can alter signal transfers or stored values and thus cause incorrect program execution.

By contrast, hard errors are caused by a permanent fault in the system and are usually caused by design faults or inherent manufacturing defects, thermal stress, wear out, and process variation. Permanent hard errors are easier to detect, because hardware deterioration is often irreversible, and their symptoms tend to be predictable and persistent over time. However, they must be detected because they present a threat to the application stability in a well-maintained environment [32]. Permanent faults usually require that the faulty component be avoided until it is repaired or replaced to avoid errors in system behavior. On the other hand, transient faults do not require repair/replacement of the component, but the impact of the resulting soft error needs to be masked.

B. Fault Tolerance

Fault-tolerance means the ability of a system to continue correct performance of its intended tasks and the ability to avoid failure after the occurrence of hardware and software errors. When a system is said to be fault-tolerant this means that the behavior of the external system is not affected by faults. A fault- tolerant system must be able to detect errors and recover from them.

IV. Overview the CUDA-based GPU Architecture and Openmp Programming

In this section, we give a brief description of the GPU architecture and CUDA, as the target applications that our tool analyzes are implemented using CUDA and Open MP and run on GPUs.

A. GPU Architecture

Fig. 3 illustrates a simplified overview of the GPU architecture. Modern GPU architecture is composed of an array of Streaming Multiprocessors (SM) [33]. The SMs are the main building blocks of a GPU. SMs consist of a set of Stream Processors (SPs) or CUDA cores, in which each core executes several threads in parallel at a specific time. SPs share control logic and an instruction cache, while SMs allow access to the global memory. In modern GPU devices, there are thousands of such SPs; this indicates that each GPU has the potential of executing thousands of threads at any moment. Moreover, each SM has shared memory and the L1 cache that is designed to improve the computational performance by storing the data common to the threads running on the SM.

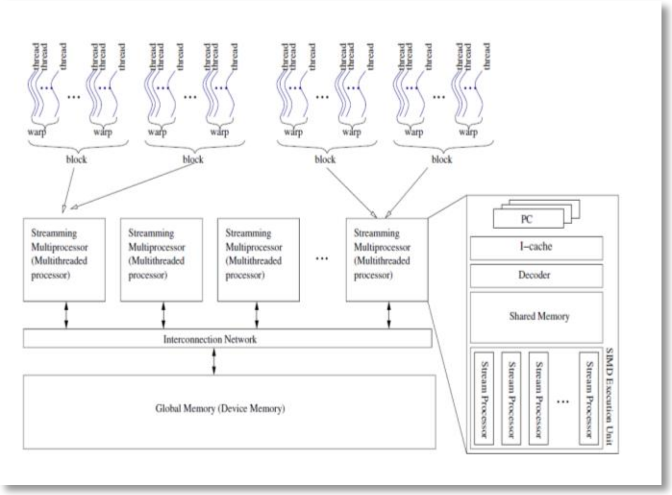


Fig. 3. An Overview of the GPU Architecture [33].

GPUs use this architecture in SIMT (Single Instruction Multiple Threads) [34], in which a group of (currently 32) threads known as a warp performs the same instruction. All the threads in one block are performed on one SM, or they can be implemented as multiple concurrently running blocks. The number of blocks that can be processed concurrently on one SM depends on the resource requirements of each block like shared memory usage and the number of registers.

There are many GPU programming languages that aim to provide an environment in which GPU and CPU programs can exist with each other. The main goal of these programming languages is to offload the GPU friendly portion of the program into the GPU memory. In this work, we use the CUDA programming language that is specifically employed for NVidia GPUs.

B. CUDA

CUDA (Compute United Device Architecture) is a parallel computing architecture developed by NVidia for massively parallel high-performance computing [35]. It can be accessible through CUDA-accelerated libraries, compiler directives, application programming interfaces, and standard programming languages including C, C++, Fortran, and Python. There are several programming models accessible to create program for GPU but CUDA by NVidia is the best option to accomplish parallelism through GPU processing.

Recently, this platform has proven successful in parallel computing architecture at programming multi-threaded on many-core GPUs .The GPU acts as a coprocessor that performs data-parallel kernel functions. CUDA has a hierarchy of thread groups. Threads are composed of a three level hierarchy. A grid consists of set of thread blocks that are responsible for executing a kernel function. Each block is composed of hundreds of threads. Threads inside one block have shared memory that allows sharing data. All threads within a block are executed concurrently on a multithreaded model.

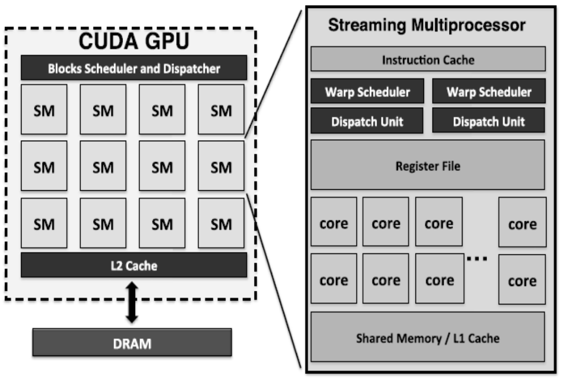


Fig. 4. A Representative CUDA-based GPU Architecture [36].

A CUDA-based system is a type of heterogeneous programming, since a program is usually running on two different platforms: a host and a device. The host system usually consists primarily of the CPU, main memory and its supporting architecture. The device generally includes the video card consisting of a CUDA-enabled GPU and its supporting architecture. The CPU begins to execute a CUDA program in order to provide inputs for the kernel and to start its implantation; this means providing a kernel grid to the GPU. The CUDA GPU begins the implantation of the kernel. Upon completion of a kernel implantation, the CPU can acquire the output data by accessing the contents of the GPU memory. The software organization of a kernel is related to the GPU architecture, since the threads hierarchy assigns immediately into the GPU internal components.

Fig. 4 illustrates CUDA GPU internal architecture. When the CPU starts to invoke a kernel grid, each thread block is assigned a Thread Block ID and dispatched to a SM that ensures enough available resources. Each thread of a thread block is executed on a CUDA core.

The programmer can specify the number of threads per block, and the number of blocks per grid. A thread in the CUDA programming language is characterized as much lighter weight than in traditional operating systems.

C. OpenMP Programming

Open multi-processing (OpenMP) is a programming model that has the ability to handle multithreading by computing in parallel modules. The basic idea of this programming model is that data are processed in parallel. It consists of a number of directives and libraries that are called runtime libraries [37]. The code inserted in these directives executes in parallel on multi-cores in the form of a basic OpenMP unit called “Thread” [38]. It also has the ability to process the looping region in a parallel way by adding compiler directives in the starting region of the OpenMP module that improve the efficiency of the program and overall application performance [39].

V. Proposed Method

In this research, we aim to detect hardware and software errors in CUDA applications that run on GPUs. Our proposed method consists of two levels of detection. The first level detects the presence of error in the results generated by the running software. The second level classifies the source of the error whether it caused by a hardware error or software error. We used multi-version programming at the first level of detection, where several versions having diverse designs of the same application are used. All versions of the software are executing in parallel. The correctness of the results is determined by running a voter in which common answers by the majority are considered the rightful result. In case the voter indicated the presence of an error in one of the versions for example, then the second level of detection is conducted. In this level, we reinvestigate the version that produced the incorrect result, by running it on a different set of cores or a different GPU with the same input data. The cause of the error is classified as hardware error in case the result is correct as stated by the majority and software error otherwise. The steps of the methods are illustrated in Fig. 5.

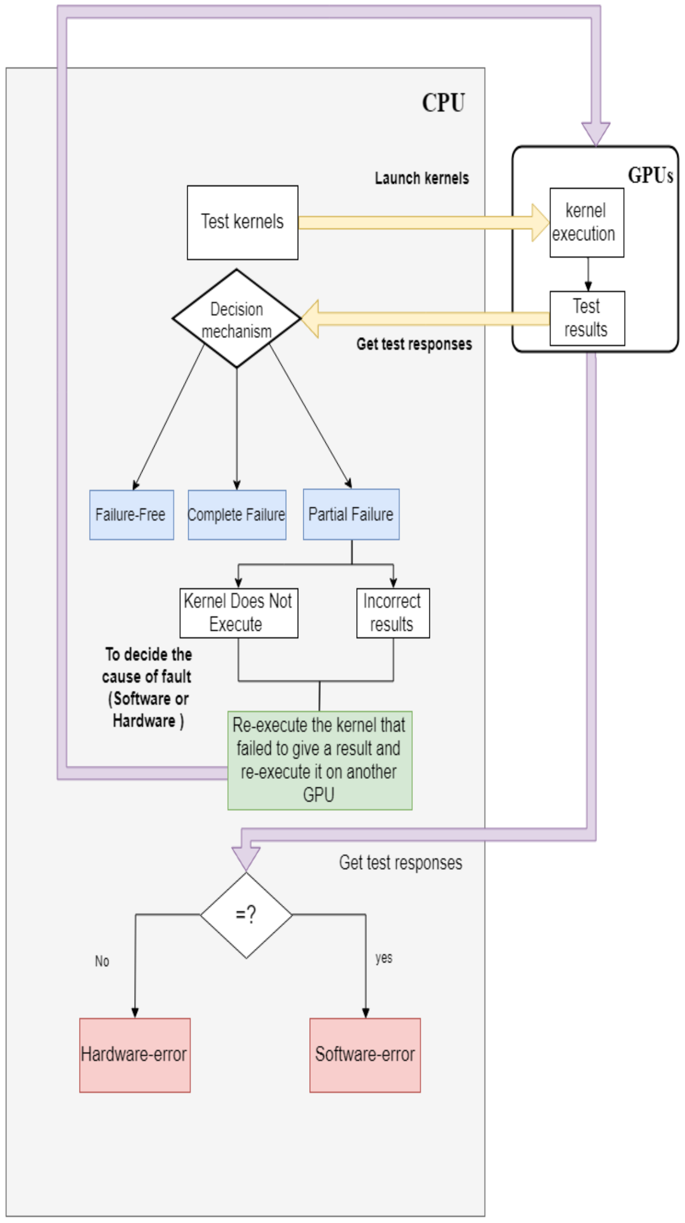


Fig. 5. Proposed Approach.

As can be seen, the different versions of the application, referred to as kernels, are executed on the GPU, whereas the control of the method steps are mainly done in the CPU, which are: launching the kernel, running the decision mechanism – the voter – and starting level two of the detection by re-executing the problematic kernel in case the voter's output indicates there is partial failure. Other possible output of the voter is that the software is error-free; i.e. all versions generated the same results, and complete failure; i.e. there is no agreement on the results among any of the versions. The figure also shows that partial failure can have the form of problematic interrupt of execution; i.e. a version of the application hangs, or the form of generating incorrect results by one of the versions.

VI. Experimental Results

In this section, we describe the experiments conducted to test the applicability of our method. First, we describe the system specifications on which the experiments are conducted and then we describe the application chosen to conduct the experiments. After that, we present the techniques used to inject faults in systems. In the following section, we describe the design of the experiments and show the results. Finally, we discuss the findings derived from our experiments.

A. System Specification

First, the hardware specifications of the system on which the experiments are conducted are listed. The machine contains a single Intel (R) Core (TM) i7-7700K CPU @4.20 GHz , equipped with three Nvidia GeForce GPUs: two of them are of the model GTX 1070 and the third GPU is of the model GTX1060. More details of the different GPUs are shown in Table I.

TABLE I. An Overview of the GPUs used. SM Denotes Streaming Multi-Processor

GPU Name	**GTX 1060**	**2x GTX 1070**
Architecture	Pascal	Pascal
# SMs	10	15
# cores/SM	1280	1920

Next, the software specifications are described. The machine runs Windows 10 as an operating system, and the development environment used is Microsoft Visual Studio community 2015 which as it is compatible with CUDA Toolkit 8.0 that we used to develop the different versions of the application – that is described in the next section. Simple visual studio C++, and OpenMP are also needed to develop the application and our tool.

B. The Application used for Testing

In this section, we describe the application we used to conduct our experiments on. We chose Matrix Multiplication as it is a computational mathematical operation that is widely used in the computational sciences in general, and scientific modeling in high performance computing domain as well [14, 22].

Several algorithms and mathematical formulas have been proposed to solve matrix multiplication, one of the proposed approaches exploits the massive parallelism of GPUs to speed up computations. Our method detects errors in parallel applications, thus, we have chosen three different parallel algorithms for solving matrix multiplication to conduct our experiments. The chosen algorithms show the design diversity required by the multi-version programming system. Next, we present the mathematical formula of matrix multiplication and then describe the three different algorithms used to solve this mathematical problem.

The mathematical formula for matrix multiplication is given in equation 1.

$$C_{i,j} = \sum_{k=1}^{n} A_{i,k} * B_{k,j} \quad (1)$$

Equation 1: general formula of Matrix Multiplication

Where A and B are matrixes of the sizes n×m and m×w respectively. C is a matrix of the size n×w that stores the product of matrix A and matrix B. For simplicity, we only created square matrices during our experiments.

The different algorithms chosen for matrix multiplications differ in the kind of memory being used, thus causing adequate changes in the design of each algorithm that are enough to introduce the design diversity required by our method – using different set of steps in each algorithm. The first algorithm used one thread to compute the result of an element of the matrix C. It depends on using global memory causing the performance to become relatively slow. The second algorithm uses shared memory to avoid unnecessarily accessing global memory multiple of times. The third algorithm is different from the second algorithm in that it transposes the second matrix, which is referred to as matrix B in (1).

C. Fault Injection

In our experiments, we need a procedure to inject faults and monitor the effect of these faults on system's behavior [40]. Fault injection is a widely used method for improving the reliability of applications. Reviews of fault injection techniques and methodologies in electronic and computer systems can be found in [18, 41]. Research has also been done to provide a framework to allow fault injection in HPC applications with the focus of facilitating designing complex experiments by defining workloads [42]. This framework, called FINJI, allows the integration of existing fault injection tools for heterogenous types of errors. Testing the detection of hardware and software errors requires fault injection of both types. Hardware errors will result in Silent Data Corruption (SDC) which is a kind of soft error that can simply be described as the flip of a bit or two in both kind of storage volatile and non-volatile [43]. Some of the approaches used to inject hardware errors are FPGA-based fault injection [44] and simulations to conduct microarchitecture-level fault injection [26]. The latter has been applied in a multicore environment called mSWAT for detecting hardware errors [45]. The idea is to detect hardware errors via software anomalies such as fatal-traps and system hangs, these detected errors are then diagnosed to identify which part pf the micro-architectural of the system is the source of the error as described in [46].

FPGA-based fault injections are performed on gate-level models and accelerated by FPGA-based hardware emulation. This approach injects errors at gates based on a user-provided model of hardware design. Another tool that injects errors at gates is Argus [47]. In our tool, there is no need to follow any of the above-mentioned hardware fault injection approaches, as our tool satisfies the objective of the research by reporting back the faulty core without specifying the kind of hardware error. Whereas the other approaches aim to identify the source of the error, for example, Argus, based on running certain instruction, it identifies the source of the error in a simple core. The research applying FPGA-based fault injection is measuring the accuracy of detecting the SDC that results from the specific types of hardware errors.

As for software fault injection, we have performed fault injection at the source code level, by conducting mutation of the source code of the application being analyzed and observing the outcomes. This kind of injection has been applied in other research such as [40, 48, 49].

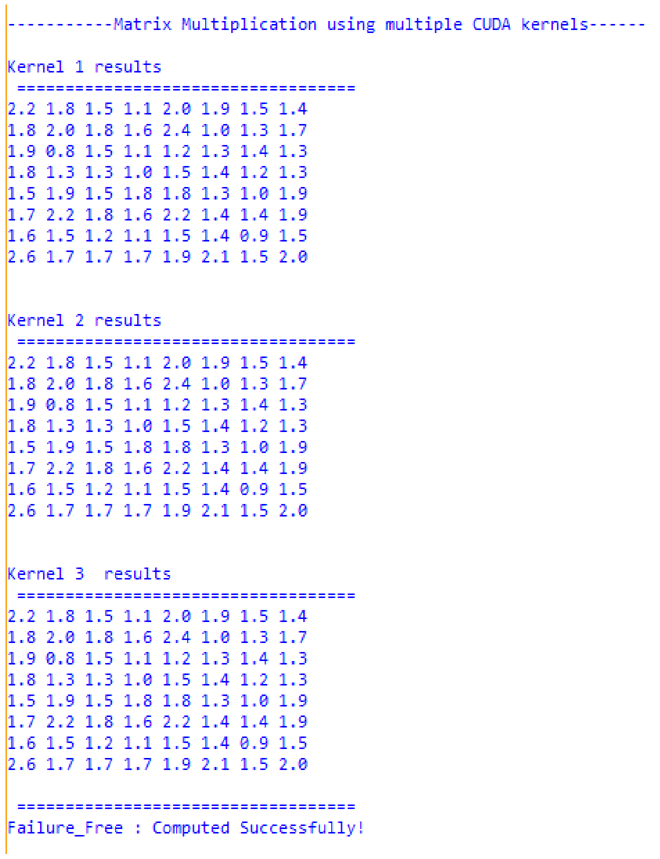
D. Experiments Design and Results

In order to test the applicability of our method, we need to ensure that the method is able to report back error-free, partial failure and complete failure cases. In this section, we focus on the error-free and partial failure cases as the complete failure case can be tested in the same way we test partial failure. In addition, to summarize the results, we report back the partial failure case in which the problematic version of the application generates in correct results. In addition, for the partial failure case, we conduct an experiment to detect hardware errors and another experiment to detect software errors. Soft errors are injected by changing the one or more of the code instructions to generate incorrect result. Hard errors are assumed they exist in one of the GPUs and we designed a method that returns an incorrect result in both levels of our detection method. In the following, we present the result of each experiment, and then present a table showing relevant measurements:

Fig. 6 shows the result of executing the three algorithms in which all are error-free, consequently depicting the error-free case:

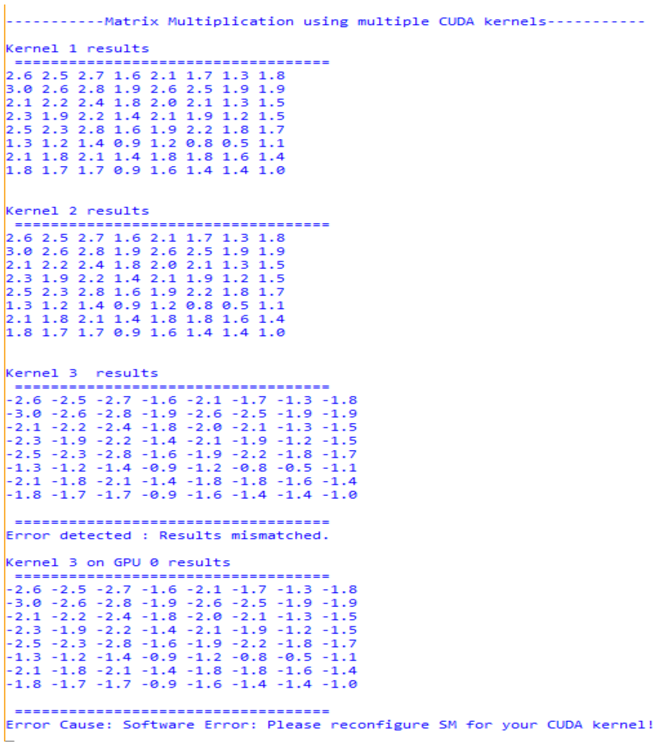
Fig.7 shows the result of executing the three algorithms, where one of the algorithms has an injected soft error, thus generating incorrect results. This depicts the case of partial failure caused by a soft error. In this case, the second level of the detection method is used to determine that it type of error is a soft error, since re-executing the algorithm on a different GPU generated an incorrect result as well.

Fig.8 shows the result of executing the three algorithms, where one of the algorithms generates in correct results (as returned by our designed method that mimics hard errors as described in the fault injection section). This depicts the case of partial failure caused by a hard error. In this case, the second level of the detection method is used to determine that the type of error is a hard error, since the algorithm generated a correct result when run on a different GPU.



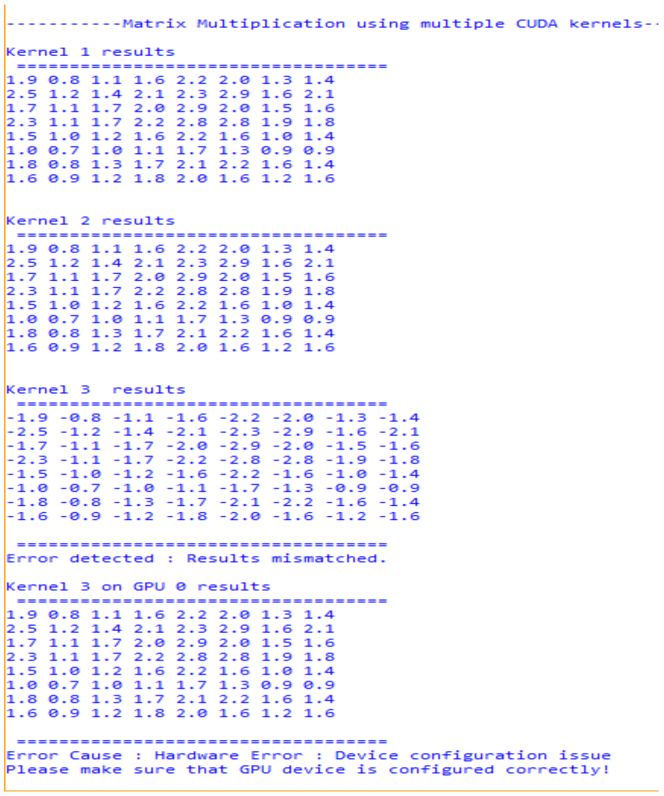
```
-----------Matrix Multiplication using multiple CUDA kernels------

Kernel 1 results
 ===================================
2.2 1.8 1.5 1.1 2.0 1.9 1.5 1.4
1.8 2.0 1.8 1.6 2.4 1.0 1.3 1.7
1.9 0.8 1.5 1.1 1.2 1.3 1.4 1.3
1.8 1.3 1.3 1.0 1.5 1.4 1.2 1.3
1.5 1.9 1.5 1.8 1.8 1.3 1.0 1.9
1.7 2.2 1.8 1.6 2.2 1.4 1.4 1.9
1.6 1.5 1.2 1.1 1.5 1.4 0.9 1.5
2.6 1.7 1.7 1.7 1.9 2.1 1.5 2.0


Kernel 2 results
 ===================================
2.2 1.8 1.5 1.1 2.0 1.9 1.5 1.4
1.8 2.0 1.8 1.6 2.4 1.0 1.3 1.7
1.9 0.8 1.5 1.1 1.2 1.3 1.4 1.3
1.8 1.3 1.3 1.0 1.5 1.4 1.2 1.3
1.5 1.9 1.5 1.8 1.8 1.3 1.0 1.9
1.7 2.2 1.8 1.6 2.2 1.4 1.4 1.9
1.6 1.5 1.2 1.1 1.5 1.4 0.9 1.5
2.6 1.7 1.7 1.7 1.9 2.1 1.5 2.0


Kernel 3  results
 ===================================
2.2 1.8 1.5 1.1 2.0 1.9 1.5 1.4
1.8 2.0 1.8 1.6 2.4 1.0 1.3 1.7
1.9 0.8 1.5 1.1 1.2 1.3 1.4 1.3
1.8 1.3 1.3 1.0 1.5 1.4 1.2 1.3
1.5 1.9 1.5 1.8 1.8 1.3 1.0 1.9
1.7 2.2 1.8 1.6 2.2 1.4 1.4 1.9
1.6 1.5 1.2 1.1 1.5 1.4 0.9 1.5
2.6 1.7 1.7 1.7 1.9 2.1 1.5 2.0

 ===================================
Failure_Free : Computed Successfully!
```

Fig. 6. A Screenshot of the Result in the Case of Failure-Free.



```
-----------Matrix Multiplication using multiple CUDA kernels-----------

Kernel 1 results
 ===================================
2.6 2.5 2.7 1.6 2.1 1.7 1.3 1.8
3.0 2.6 2.8 1.9 2.6 2.5 1.9 1.9
2.1 2.2 2.4 1.8 2.0 2.1 1.3 1.5
2.3 1.9 2.2 1.4 2.1 1.9 1.2 1.5
2.5 2.3 2.8 1.6 1.9 2.2 1.8 1.7
1.3 1.2 1.4 0.9 1.2 0.8 0.5 1.1
2.1 1.8 2.1 1.4 1.8 1.8 1.6 1.4
1.8 1.7 1.7 0.9 1.6 1.4 1.4 1.0


Kernel 2 results
 ===================================
2.6 2.5 2.7 1.6 2.1 1.7 1.3 1.8
3.0 2.6 2.8 1.9 2.6 2.5 1.9 1.9
2.1 2.2 2.4 1.8 2.0 2.1 1.3 1.5
2.3 1.9 2.2 1.4 2.1 1.9 1.2 1.5
2.5 2.3 2.8 1.6 1.9 2.2 1.8 1.7
1.3 1.2 1.4 0.9 1.2 0.8 0.5 1.1
2.1 1.8 2.1 1.4 1.8 1.8 1.6 1.4
1.8 1.7 1.7 0.9 1.6 1.4 1.4 1.0


Kernel 3  results
 ===================================
-2.6 -2.5 -2.7 -1.6 -2.1 -1.7 -1.3 -1.8
-3.0 -2.6 -2.8 -1.9 -2.6 -2.5 -1.9 -1.9
-2.1 -2.2 -2.4 -1.8 -2.0 -2.1 -1.3 -1.5
-2.3 -1.9 -2.2 -1.4 -2.1 -1.9 -1.2 -1.5
-2.5 -2.3 -2.8 -1.6 -1.9 -2.2 -1.8 -1.7
-1.3 -1.2 -1.4 -0.9 -1.2 -0.8 -0.5 -1.1
-2.1 -1.8 -2.1 -1.4 -1.8 -1.8 -1.6 -1.4
-1.8 -1.7 -1.7 -0.9 -1.6 -1.4 -1.4 -1.0

 ===================================
Error detected : Results mismatched.

Kernel 3 on GPU 0 results
 ===================================
-2.6 -2.5 -2.7 -1.6 -2.1 -1.7 -1.3 -1.8
-3.0 -2.6 -2.8 -1.9 -2.6 -2.5 -1.9 -1.9
-2.1 -2.2 -2.4 -1.8 -2.0 -2.1 -1.3 -1.5
-2.3 -1.9 -2.2 -1.4 -2.1 -1.9 -1.2 -1.5
-2.5 -2.3 -2.8 -1.6 -1.9 -2.2 -1.8 -1.7
-1.3 -1.2 -1.4 -0.9 -1.2 -0.8 -0.5 -1.1
-2.1 -1.8 -2.1 -1.4 -1.8 -1.8 -1.6 -1.4
-1.8 -1.7 -1.7 -0.9 -1.6 -1.4 -1.4 -1.0

 ===================================
Error Cause: Software Error: Please reconfigure SM for your CUDA kernel!
```

Fig. 7. A Screenshot of the Result in the Case of Partial Failure (Software Error).



```
-----------Matrix Multiplication using multiple CUDA kernels-·

Kernel 1 results
 ====================================
1.9 0.8 1.1 1.6 2.2 2.0 1.3 1.4
2.5 1.2 1.4 2.1 2.3 2.9 1.6 2.1
1.7 1.1 1.7 2.0 2.9 2.0 1.5 1.6
2.3 1.1 1.7 2.2 2.8 2.8 1.9 1.8
1.5 1.0 1.2 1.6 2.2 1.6 1.0 1.4
1.0 0.7 1.0 1.1 1.7 1.3 0.9 0.9
1.8 0.8 1.3 1.7 2.1 2.2 1.6 1.4
1.6 0.9 1.2 1.8 2.0 1.6 1.2 1.6


Kernel 2 results
 ====================================
1.9 0.8 1.1 1.6 2.2 2.0 1.3 1.4
2.5 1.2 1.4 2.1 2.3 2.9 1.6 2.1
1.7 1.1 1.7 2.0 2.9 2.0 1.5 1.6
2.3 1.1 1.7 2.2 2.8 2.8 1.9 1.8
1.5 1.0 1.2 1.6 2.2 1.6 1.0 1.4
1.0 0.7 1.0 1.1 1.7 1.3 0.9 0.9
1.8 0.8 1.3 1.7 2.1 2.2 1.6 1.4
1.6 0.9 1.2 1.8 2.0 1.6 1.2 1.6


Kernel 3  results
 ====================================
-1.9 -0.8 -1.1 -1.6 -2.2 -2.0 -1.3 -1.4
-2.5 -1.2 -1.4 -2.1 -2.3 -2.9 -1.6 -2.1
-1.7 -1.1 -1.7 -2.0 -2.9 -2.0 -1.5 -1.6
-2.3 -1.1 -1.7 -2.2 -2.8 -2.8 -1.9 -1.8
-1.5 -1.0 -1.2 -1.6 -2.2 -1.6 -1.0 -1.4
-1.0 -0.7 -1.0 -1.1 -1.7 -1.3 -0.9 -0.9
-1.8 -0.8 -1.3 -1.7 -2.1 -2.2 -1.6 -1.4
-1.6 -0.9 -1.2 -1.8 -2.0 -1.6 -1.2 -1.6

 ====================================
Error detected : Results mismatched.

Kernel 3 on GPU 0 results
 ====================================
1.9 0.8 1.1 1.6 2.2 2.0 1.3 1.4
2.5 1.2 1.4 2.1 2.3 2.9 1.6 2.1
1.7 1.1 1.7 2.0 2.9 2.0 1.5 1.6
2.3 1.1 1.7 2.2 2.8 2.8 1.9 1.8
1.5 1.0 1.2 1.6 2.2 1.6 1.0 1.4
1.0 0.7 1.0 1.1 1.7 1.3 0.9 0.9
1.8 0.8 1.3 1.7 2.1 2.2 1.6 1.4
1.6 0.9 1.2 1.8 2.0 1.6 1.2 1.6

 ====================================
Error Cause : Hardware Error : Device configuration issue
Please make sure that GPU device is configured correctly!
```

Fig. 8. A Screenshot of the Result in the Case of Partial Failure (Hardware Error).

E. Discussion

This section aims to give insights into our proposed method and to compare it with other approaches in terms of detecting faults and ability to distinguish them. As explained in the previous section, our proposed method was designed as a two-level technique, in which the first level based on design diversity that applies N-Version Programming technique. Whereas the second level is designed to distinguish the type of error that is detected.

In [30, 50], NVP is used for detecting hardware and software faults, however, they do not address concurrent applications. It is noteworthy to mention that in [30] the system detects transient faults either software or hardware and permanent hardware faults. More specifically, it can detect errors that cause one of the components become disabled or cause the generation of incorrect results.

The most relevant tool to our work is mSWAT [45]. It applies the two-level approach for detecting permanent hardware and software errors, in a similar manner to our work. However, they use TMR approach in the first level, whereas we use NVP. In the second level, mSWAT stores traces of execution for each core, then checks if there is divergence in the execution of one of the cores then it will be considered as a faulty core. In addition, they conduct further analysis to identify the faulty micro-architectural component for repair. In our work, we only report back that there is a permanent hardware error or a software error.

mSWAT also addresses transient errors at the beginning by re-executing the process on all cores in a similar manner of rollback/replay. If the error is not repeated, then it is considered a transient software bug. In our work, we have not included the detection of transient errors.

We detect the existence of errors in the first phase by identifying that one of the software versions are producing in correct results or experiencing application hang. In mSWAT, they have addressed four kinds of software anomalies, including hangs, fatal traps, panic, etc.

It is also worth mentioning that using NVP in our work has the difficulty of designing and implementing three versions of the software, however, it needs no tracing of execution and it only re-executes the problematic version once unlike mSWAT. We also do not need to store the re-execution and do comparisons for divergence checking, we only compare the results in case the application do not hang.

mSWAT, addresses more error types and faulty micro-architectural components identification. However, in our work we investigated the possibility of benefiting from NVP that up-to-our knowledge has not been previously investigated for HPC applications.

VII. Conclusion

Faults are becoming more frequent in large supercomputers, and their impact is higher in the case of long-duration applications. This research seeks to address resilience challenges by presenting an innovative method to detect software and hardware errors that can be become a concern for the performance of scientific applications running on these future systems.

We have investigated an approach to detect and classify faults for CUDA applications using multiple GPUs. Our approach benefits from NVP for detecting errors then carrying another analysis by running the problematic software version on a different GPU to classify the type of error. Our proposed approach is flexible in the sense that it can be applied to different applications not just matrix multiplication. Experimental results indicate the capability of the proposed method to detect errors and classify whether they are permanent hardware errors or software errors. Hence, assisting in improving reliability. We plan to integrate this detection algorithm in a more comprehensive framework that includes error recovery and sophisticated fault injection techniques and test our approach on other types of applications to collect further measurements of the coverage and overhead of our approach.

Acknowledgment

We would like to express our gratitude and appreciation to Prof. Fathy Eassa the head of our research group in the department of Computer Science at KAU. He always provides us with promising research directions and continuous guidance.

References

[1] Ashby, Steve, et al. "The opportunities and challenges of exascale computing–Summary report of the advanced scientific computing advisory committee (ASCAC) subcommittee." US Department of Energy Office of Science ,2010.

[2] Van De Vanter, Michael L., D. E. Post, and Mary E. Zosel. "HPC needs a tool strategy." Proceedings of the second international workshop on Software engineering for high performance computing system applications. ACM, 2005.

[3] G.Rinku, et al. "Introspective fault tolerance for exascale systems." US Department of Energy Advanced Scientific Computing Research, OS and Runtime Technical Council Workshop. 2012 .

[4] Constantinescu, Cristian. "Trends and challenges in VLSI circuit reliability." IEEE micro 4 (2003): 14-19.

[5] Gizopoulos, Dimitris, et al. "Architectures for online error detection and recovery in multicore processors." Design, Automation & Test in Europe Conference & Exhibition (DATE), 2011. IEEE, 2011.

[6] Tselonis, Sotiris, Vasilis Dimitsas, and Dimitris Gizopoulos. "The functional and performance tolerance of gpus to permanent faults in registers." On-Line Testing Symposium (IOLTS), 2013 IEEE 19th International. IEEE, 2013.

[7] Wunderlich, Hans-Joachim, Claus Braun, and Sebastian Halder. "Efficacy and efficiency of algorithm-based fault-tolerance on GPUs." On-Line Testing Symposium (IOLTS), 2013 IEEE 19th International. IEEE, 2013.

[8] Guan, Qiang, et al. "Empirical studies of the soft error susceptibility ofsorting algorithms to statistical fault injection." Proceedings of the 5th Workshop on Fault Tolerance for HPC at eXtreme Scale. ACM, 2015.

[9] Schroeder, Bianca, and Garth Gibson. "A large-scale study of failures in high-performance computing systems." IEEE Transactions on Dependable and Secure Computing 7.4 (2010): 337-350.

[10] Di Carlo, Stefano, et al. "Fault mitigation strategies for CUDA GPUs." Test Conference (ITC), 2013 IEEE International. IEEE, 2013.

[11] Dimitrov, Martin, Mike Mantor, and Huiyang Zhou. "Understanding software approaches for GPGPU reliability." Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units. ACM, 2009.

[12] X. Xu, et. Al. HiAL-Ckpt: A hierarchical application-level checkpointing for cpu-gpu hybrid systems, 2010

[13] K.-H Huang and Abraham, "Algorithm-based fault tolerance for matrix operations, 1984

[14] C. Ding. Et. Al. "Matrix multiplication on GPUs with on-line fault tolerance" 2011

[15] Lyons, Robert E., and Wouter Vanderkulk. "The use of triple-modular redundancy to improve computer reliability." IBM Journal of Research and Development 6.2 (1962): 200-209.

[16] Bartlett, Wendy, and Lisa Spainhower. "Commercial fault tolerance: A tale of two systems." IEEE Transactions on Dependable Secure Computing, 1(1):87–96, 2004.

[17] Pullum, Laura L. Software fault tolerance techniques and implementation. Artech House, 2001.

[18] Ziade, Haissam, Rafic A. Ayoubi, and Raoul Velazco. "A survey on fault injection techniques." Int. Arab J. Inf. Technol. 1.2 (2004): 171-186.

[19] Chen, L. (1978). V-version programming: A fault-tolerance approach to reliability of software operation. FTCS-8, 1978, 6.

[20] B. Randell, "System Structure for Software Fault Tolerance," IEEE Trans, on Software Engineering, Vol. 1, No. 2, June 1975, pp.220-232

[21] Laprie, J. C., Arlat, J., Beounes, C., & Kanoun, K."Definition and analysis of hardware-and software-fault-tolerant architectures". *Computer*, 1990, 23.7: 39-51.

[22] Sabena, Davide, et al. "On the evaluation of soft-errors detection techniques for GPGPUs." Design and Test Symposium (IDT), 2013 8th International. IEEE, 2013.

[23] Sheaffer, Jeremy W., David P. Luebke, and Kevin Skadron. "A hardware redundancy and recovery mechanism for reliable scientific computation on graphics processors." Graphics Hardware. Vol. 2007.

[24] K.S. Yim, C. Pham, M. Saleheen, Z. Kalbarczyk, R. Iyer "Hauberk: Lightweight silent data corruption error detector for gpgpu " ,Proceedings of the 2011 IEEE International Parallel & Distributed Processing Symposium, IPDPS '11, IEEE Computer Society, Washington, DC, USA (2011), pp. 287-300.

[25] Lei Zhang, Yinhe Han, Qiang Xu, and Xiaowei Li. Defect tolerance in homogeneous manycore processors using core-level redundancy with unified topology. In DATE '08: Proceedings of the conference on Design, automation and test in Europe, pages 891–896. ACM, 2008.

[26] M.-L.Li, et al., "Understanding the Propagation of Hard Errors to Software and Implications for Resilient System Design", ASPLOS 2008.

[27] J.-C. Laprie, Dependability: Basic Concepts and Terminology, 1992.

[28] Pradhan, Dhiraj K. *Fault-tolerant computer system design*. Vol. 132. Englewood Cliffs: Prentice-Hall, 1996.

[29] Dubrova, Elena. *Fault-tolerant design*. New York: Springer, 2013.

[30] DUGAN, J. Bechta; LYU, Michael R. System reliability analysis of an N-version programming application. *IEEE Transactions on Reliability*, 1994, 43.4: 513-519.

[31] B.Schroeder and Garth A Gibson. Understanding failures in petascale computers. Journal of Physics: Conference Series, 78, 2007

[32] Navarro, Cristobal A., Nancy Hitschfeld-Kahler, and Luis Mateu. "A survey on parallel computing and its applications in data-parallel problems using GPU architectures." *Communications in Computational Physics* 15.02 (2014): 285-329.

[33] D. B. Kirk and W. H. Wen-Mei. Programming massively parallel processors: a hands-on approach, 3rd edition. Morgan Kaufmann, 2016.

[34] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym. NVIDIA Tesla: A unified graphics and computing architecture. IEEE micro, 28(2), 2008.

[35] D. Göddeke, R. Strzodka, J. Mohd-Yusof, P. McCormick, S. Buijssen, M. Grajewski, S. Tureka, Exploring weak scalability for FEM calculations on a GPU-enhanced cluster, Parallel Comput. 33 (Nov. 2007) 685–699.

[36] NVIDIA, "NVIDIA kepler K20 GPU datasheet," 2012.

[37] J. M. Yusof et al, "Exploring weak scalability for FEM calculations on a GPU-Enhanced cluster", 33.685–699. Nov, 2007.

[38] Yang, Chao-Tung, Chih-Lin Huang, and Cheng-Fang Lin. "Hybrid CUDA, OpenMP, and MPI parallel programming on multicore GPU clusters." Computer Physics Communications 182.1 (2011): 266-269.

[39] C.T. Yang, C.L. Huang and C.F. Lin, "Hybrid CUDA, OpenMP, and MPI parallel programming on multicore GPU". Computer Physics Communications. Pp. 266-269. 2011.

[40] M.-C. Hsueh, T. Tsai, and R. Iyer, "Fault injection techniques and tools," Computer, vol. 30, no. 4, pp. 75–82, 1997.

[41] Song, Ningfang, et al. "Fault injection methodology and tools." Electronics and Optoelectronics (ICEOE), 2011 International Conference on. Vol. 1. IEEE, 2011.

[42] Netti, Alessio, et al. "FINJ: A Fault Injection Tool for HPC Systems." arXiv preprint arXiv:1807.10056 (2018).

[43] Fiala, David, et al. "Detection and correction of silent data corruption for large-scale high-performance computing." Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis. IEEE Computer Society Press, 2012.

[44] Pellegrini, Andrea, et al. "CrashTest: A fast high-fidelity FPGA-based resiliency analysis framework." Computer Design, 2008. ICCD 2008. IEEE International Conference on. IEEE, 2008.

[45] Hari, Siva Kumar Sastry, et al. "mSWAT: low-cost hardware fault detection and diagnosis for multicore systems." Microarchitecture, 2009. MICRO-42. 42nd Annual IEEE/ACM International Symposium on. IEEE, 2009.

[46] Li, Man-Lap, et al. "Trace-based microarchitecture-level diagnosis of permanent hardware faults." Dependable Systems and Networks With FTCS and DCC, 2008. DSN 2008. IEEE International Conference on. IEEE, 2008.

[47] Meixner, Albert, Michael E. Bauer, and Daniel Sorin. "Argus: Low-cost, comprehensive error detection in simple cores." Microarchitecture,

2007. MICRO 2007. 40th Annual IEEE/ACM International Symposium on. IEEE, 2007.

[48] K.S. Yim, C. Pham, M. Saleheen, Z. Kalbarczyk, R. Iyer Hauberk: Lightweight silent data corruption error detector for gpgpu Proceedings of the 2011 IEEE International Parallel & Distributed Processing Symposium, IPDPS ’11, IEEE Computer Society, Washington, DC, USA (2011), pp. 287-300.

[49] M. Hiller, A. Jhumka, and N. Suri, “Propane: an environment for examining the propagation of errors in software,” ACM SIGSOFT Software Engineering Notes, vol. 27, no. 4, pp. 81–85, 2002.

[50] FUHRMAN, Christopher P.; CHUTANI, Sailesh; NUSSBAUMER, Henri J. Hardware/software fault tolerance with multiple task modular redundancy. In: *Computers and Communications, 1995. Proceedings., IEEE Symposium on*. IEEE, 1995. p. 171-177.