Analysis and Mitigation of Soft-Errors on High Performance Embedded GPUs

Analysis and Mitigation of Soft-Errors on High Performance Embedded GPUs

L. Sterpone, S. Azimi, C. De Sio
Dipartimento di Automatica e Informatica
Politecnico di Torino
Torino, Italy
luca.sterpone@polito.it

F. Parisi
Electronic, SW and Testing Division
Punch Soft Tronix
Torino, Italy
filippo.parisi@punchtorino.com

***Abstract*—Multiprocessor system-on-chip such as embedded GPUs are becoming very popular in safety-critical applications, such as autonomous and semi-autonomous vehicles. However, these devices can suffer from the effects of soft-errors, such as those produced by radiation effects. These effects are able to generate unpredictable misbehaviors. Fault tolerance oriented to multi-threaded software introduces severe performance degradations due to the redundancy, voting and correction threads operations. In this paper, we propose a new fault injection environment for NVIDIA GPGPU devices and a fault tolerance approach based on error detection and correction threads executed during data transfer operations on embedded GPUs. The fault injection environment is capable of automatically injecting faults into the instructions at SASS level by instrumenting the CUDA binary executable file. The mitigation approach is based on concurrent error detection threads running simultaneously with the memory stream device to host data transfer operations. With several benchmark applications, we evaluate the impact of soft-errors classifying Silent Data Corruption, Detection, Unrecoverable Error and Hang. Finally, the proposed mitigation approach has been validated by soft-error fault injection campaigns on an NVIDIA Pascal Architecture GPU controlled by Quad-Core A57 ARM processor (JETSON TX2) demonstrating an advantage of more than 37% with respect to state of the art solution.**

Keywords—Embedded GPUs, Soft-Errors, High performances, Error Detection and Correction

I. INTRODUCTION

The GPU devices were first introduced as accelerators for graphic processing, later driven by the market demand for the gaming industry, modern GPUs were developed for higher performance in multimedia processing such as texturing, shading, and in recent years ray tracing [1][2]. People soon realized those GPU devices could be used for other applications demanding higher computational power such as scientific simulation in the bio-medical area and data mining in the big data area. Thus, General-Purpose GPU (GPGPU) devices started to gain popularity in other areas with the help of corresponding software programming environments for researchers and developers to better utilize the many-core architectural parallelism in those devices. With the GPGPU application expanding within the domain of High-Performance Computing (HPC) to safety and mission-critical application, for example, self-driving vehicles [3][4][5], reliability becomes one of the major constraints to be addressed [6] [7].

As each technology of GPGPU devices employs the latest semiconductor processing technology pushing toward higher clock frequency and higher device density, they become more susceptible to soft errors due to outside distributors, for example, radiation particle strikes even at sea level [8].

Thus, when safety and mission-critical applications are targeted, the reliability of the implemented software and target hardware device should be investigated. One of the techniques for reliability analysis is through fault injection. Depending on the availability of certain information, the injection of faults could be carried out at different levels of abstraction. Unluckily, the internal implementation of GPGPU hardware is unavailable, so fault injection on GPGPU devices is usually carried out using models, simulators, accelerated radiation beams on real hardware, or instrumented software code [9] [10].

This paper proposes a reliability analysis framework in terms of a fault injection environment, named CUBINJ, for NVIDIA GPGPU devices based on instrumenting the Streaming ASSembler (SASS) instructions and software mitigation method able to provide dependable computing strategies without introducing a relevant overhead in terms of computational time. The developed fault injection environment can mimic the faults affecting the control logic and other parts within the pipeline of GPGPU cores by instrumenting the actual instructions executed on the device. As the fault injection is executed while the target application is running at speed on real GPGPU devices, it is much faster than using a model or simulator. Please notice that in contrast to the existing tool, CUBINJ is able to inject transient faults into all threads as well as into specified thread(s). However, despite state-of-the-art tools such as NVBitFI, CUBIJN requires access and instrumentation of the CUDA source code.

The rest of this paper is organized as follows: Section II presents state-of-the-art techniques for reliability analysis targeting GPGPU devices. Section III introduces the proposed fault injection environment. Section IV presents the developed mitigation approach, while Section V describes the fault injection experiments on selected benchmark applications with

an analysis of the results. Finally, Section VI draws the conclusions and discusses future works.

II. Background

Depending on which information is available, different techniques are developed to perform reliability analysis of GPUs. Several radiation experiments have been performed using accelerated radiation beams, focusing on various aspects of analysis and mitigation of soft errors in GPGPU devices. These experiments present realistic data as actual devices are tested under radiation particle strike. However, radiation beam hits the device indistinctively and the information regarding the internal implementation is not available. Therefore, many simulation environments have been developed and exploited to perform more detailed analyses.

Several simulators have been developed for micro-architectural level analysis, such as the functional GPGPU simulator Barra [14], and heterogenous CPU-GPU simulator gem5-gpu [16][17] and GPGPU-sim [18]. These tools mimic the parallelism implemented in commercial GPGPU devices and could be used at early stages to investigate the reliability of GPUs [15].

In order to perform a more detailed analysis, several models at a lower level are exploited. Among them, FlexGrip [19] was presented as a VHDL-based model supporting CUDA binary of streaming multiprocessor computation based on NVIDIA G80 architecture.

In addition to radiation tests and simulators, fault injection can evaluate the reliability of a system. In [12], the SASSIFI tool was proposed to inject faults in various locations including the register files, shared memory, and instruction operands. The approach exploits a low-level assembly-language instrumentation tool called SASSI to profile and inject errors.

Similar to SASSIFI, NVBitFI platform has been proposed by NVIDIA [13], with the main difference between performing dynamic code instrumentation that intercepts dynamic kernel calls and inserts error injection without the need to the source code and without affecting any instruction scheduling or register allocation of the target program.

The proposed fault injection environment in this paper is close to the NVBitFI platform developed by NVIDIA which is meant to mimic the hardware faults using instrumented software code. However, contrary to VNBitFI, our developed platform is capable of isolating single specified threads and evaluating the reliability of each individual thread.

III. Embedded GPUs Computation Scenario

In order to evaluate the reliability of GPUs, we have developed a new fault injection environment, named CUBINJ, which has the ability to produce bitflip in Streaming ASSembly (SASS) instructions affecting either all the threads or specific thread(s). It requires instrumenting the CUDA source code.

A. Fault Model

The NVIDIA GPGPU devices employ a Single Instruction Multiple Threads (SIMT) model, meaning that the same instructions are executed by the many cores inside the device. However, each core maintains its own execution flow as a thread. The threads could diverge to perform different tasks and converge via synchronization or barrier instructions. When the CUDA application is launched, users could organize threads in Blocks and Grids. As tasks are not always perfectly distributed in threads, the same fault in terms of corrupted bits and instructions in different threads could yield different results. Thus, the current implementation of CUBINJ allows two types of fault injection:

1. Bitflip in instruction affecting all the threads
2. Bitflip in instruction affecting only certain threads

Please note that to the best of our knowledge, there is no research work dedicated to the evaluation of bitflip affecting specified threads.

B. Compilation of Source Code

When developing applications running on CUDA-enabled platforms, the NVIDIA CUDA runtime environment (runtime APIs) is often used as it provides high-level features such as context management, kernel invoke syntax extension and PTX for Just-In-Time (JIT) compilation support. This is beneficial if the application is to be deployed onto different GPGPU devices (different generations) as PTX code will be generated and embedded into the executable file along with the host code. When the application is being executed, the JIT compiler will transform PTX code to the device-specific SASS code to be executed without user interaction.

However, the low level of controls that the runtime compiler provides leads to difficulties for fault injection mechanisms that target actual SASS instruction such as the one proposed in this paper. Therefore, we have exploited the NVIDIA Driver API (DAPI), as represented in Figure 1.

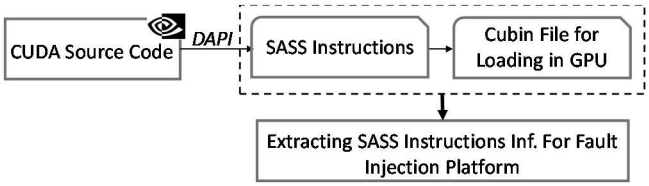

Fig. 1. The Compilation Flow of CUDA SourceCode.

In comparison, the DAPI offers more fine-grained control, especially over contexts and module loading. Kernel launches are much more complex to implement, as the execution configuration and kernel parameters must be specified with explicit function calls. However, DAPI is only dealing with CUDA binary files.

CUDA binary file, also referred to as Cubin file, is an ELF-formatted file that consists of CUDA executable code sections as well as other sections containing symbols, relocators, debug info, etc. By default, the CUDA compiler driver *nvcc* embeds cubin files into the host executable file. But they can also be generated separately and loaded at run time by the CUDA DAPI.

DAPI allows CUDA code to be compiled into device-specific SASS binary code and a Cubin file and then be loaded into GPGPU device to be executed. In this way, it is much easier to extract information regarding the SASS instructions and manipulate them for fault injection.

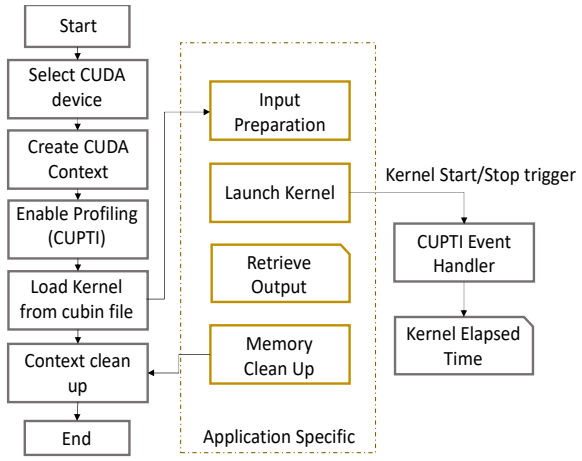

Fig.2. The Host Application Execution Flow

C. *The Development of the Host Application*

Exploiting DAPI requires extra coding to prepare the host application code for loading the Cubin file with some differences in invoking the function (kernel) to be launched on GPU devices. Please note that the host application is also in charge of preparing input data, recording/checking output data, and setting up for performance measurement using CUDA Profiling Tools Interface (CUPTI). Figure 2 represents the general flow of the host application used in the CUBINJ fault injection environment.

D. *The Fault Injection Workflow*

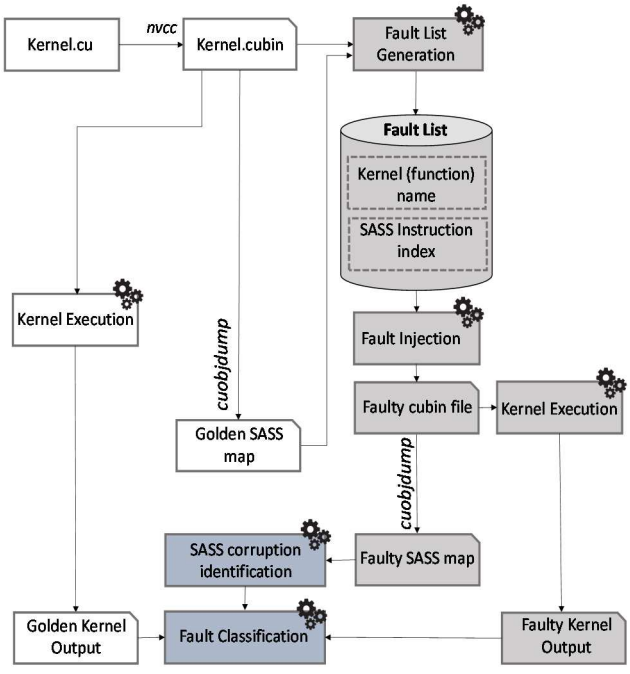

Fig. 3. The Developed Fault Injection Environment

The developed fault injection environment is represented in Figure 3. As a first step, the *nvcc* is used to compile CUDA code into *Cubinfile* while host code is compiled separately into an executable file. Then, CUBINJ extracts the binary code related to each kernel in the cubin file (*kernel.cubin*). Since the cubinfile has the same binary layout as the commonly used Executable and Link Format (elf) file, the extraction of binary code related to each kernel in the cubin file is a straightforward process. Moreover, exploiting the *cuobjdump* tool included in the CUDA environment, CUBINJ maps the extracted binary code to the associated SASS instructions (*Golden SASS Mapping*).

As the next step, CUBINJ starts building the fault list exploiting the data represented by cubin file as well as the binary code of each kernel. The generated fault list is represented as a table whose entry contains kernel (function) name, instruction index, and corresponding SASS instruction. The fault list is provided to the fault injection campaigns.

For each run of the fault injection campaign, a fault is selected targeting one of the SASS instruction's corresponding binary codes. Then, a faulty *cubin file* with the targeted flipped (corrupted) bit is generated. The *cuobjdump* tool is used again on the faulty Cubin file to build the instruction map as the previous step in order to gather information on how the SASS instruction is corrupted by the target bit.

Finally, the CUBINJ automatically launches the host application to load the faulty cubin file and gather results from the execution. The output of the execution of the faulty kernel, as well as the faulty SASS map, is compared with the golden kernel output and golden SASS map in order to classify the injected bitflip as well as eventually observed faults in the output.

E. *Threads Selection*

At this point, the developed fault injection platform supports the bitflip in instructions affecting all the threads. This section is dedicated to the further instrumentation of developed fault injection platform to target the second type of faults, bitflip in instruction affecting only certain threads. To do so, the CUBINJ is modified to isolate the targeted thread(s). Algorithm 1 represents the modification introduced in the CUDA source code.

```
If (thread_filter_condition)
{
    tag_instruction begin;
    original_cuda_code_block; // target fault sites
    tag_instruction end;
}
else{
    original_cuda_code_block;
}
```

Algorithm. 1. The Pseudocode of instrumentation of thread(s) isolation

For each CUDA thread, an ID structure is assigned depending on the Block and Grid configuration when the kernel is invoked. With the ID structure, it is possible to create a divergence path to isolate certain threads. Firstly, the original CUDA code block is duplicated and split into the two clauses of *if-else* statement, with the *thread_filter_condition* determining which thread will be affected by the faults injected. For example, if the condition is *threadID == 0* where *threadId* is the linear ID of all the threads (considering the grids and blocks), then only thread 0 will be affected by the injected faults. Secondly, two extra instructions are inserted surrounding the duplicated CUDA

code to make target fault sites much easier to locate while not affecting the computational logic.

F. Error Classification

After execution of the faulty cubin file, the fault injection environment logged various information into a local database including the fault itself, application exit condition, output data, kernel time, and SASS corruption as represented in Figure 3. Moreover, the CUBINJ compares the faulty SASS map with the golden one and identifies the SASS corruption information. This information indicates which and how a SASS instruction is affected by the injected fault. For example, the binary code "ef5c000000080405" corresponds to SASS instruction "@!P0IADD R5, R5, R6 ;", if its 54th bit is flipped, it becomes an invalid instruction; if its 55th but is flipped, it becomes "@!P0ALD.PHYS R5, a[R4], R0"; if its 44th bit is flipped, the instruction remains the same. Please note that when the injected fault results in an invalid instruction (indicated by cuobjdump tool), the GPGPU device will report (through driver) that an invalid instruction is detected, but the kernel may continue to the end.

With the information stored in the local database, fault injection results are classified as follows:

1) Silent Data Corruption (SDC): the kernel finished normally, the host application is able to copy the results from GPGPU device memory to host memory, but there is a mismatch(es) with the faulty-free run.

2) Detected Unrecoverable Error (DUE): the kernel did not finish normally as DAPI function call returns an error or the host application is not able to copy the results from GPGPU device memory to host.

3) Hang: the kernel execution plus the memory copy operations are not able to finish within a detection latency time (the exact duration is determined on the basis of the benchmark application execution time).

4) Masked: the kernel finished normally and no error is observed in the output.

IV. Mitigation Approach

The present mitigation approach investigates the computational characteristics of a typical application running on embedded GPGPUs thus being based on cyclic computational stream consisting of the following phases: initialization, synchronization with the external data sample, data stream from the host memory to the GPU memory (i.e., copy the memory controlled by the host processor to the memory of the embedded GPU), running the kernel threads, data stream from the GPU device memory to the host and then performed cyclically.

The study of the application has been settled giving a different range of constraints including the length and size of the data stream, the timing constraints of the cyclic operations which is limiting the quantity of time dedicated to the various operations (included the data transfer) and the resolution of the data stream, fundamental to achieve the desired level of sigma error. We developed the solution reported in Figure 4, where we modified the memory transfer CUDA functions using an asynchronous memory copy from the device to the host that allows the simultaneous execution of computational threads during the transfer able to perform a data comparison. We labeled these computational threads as Beacon Threads.

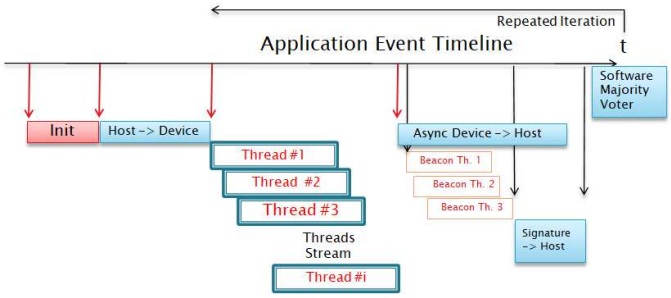

Fig. 4. The event timeline of the application mitigation method. Beacon threads are executed during the memory data transfer from the device to host.

V. Experimental Results

In this section, results and analysis from fault injection campaigns are presented including the error classification and distribution, SDC error comparison, and impact on the performance in terms of kernel time.

A. Experimental Benchmarcks

To demonstrate the capability of the proposed fault injection environment, three benchmark applications were selected:

1) *matSum:* It calculates the sum of two matrices.
2) *matMul:* it implements the tiled multiplication of two 512*512 matrices by taking advantage of shared memory.
3) *histogram:* it calculates the histogram of a collection of bytes (integer from 0 to 255 as commonly used in image applications for values of Red, Green, Blue channels).

All three applications have been instrumented as described in Section III for supporting fault injections in all threads and also for performing injection only in thread 0 (*threadId == 0*).

The three benchmark applications are compiled by *nvcc* using CUDA compute capability 5.0 (SM_50). The characteristics of the applications are reported in Table I in which kernel time is the average of 1000 runs reported by CUPTI profiling API. Please note that the *histogram* has also been tested with the version where thread 1 is affected by the faults.

TABLE I. CHARACTERISTICS OF BENCHMARK APPLICATIONS

Benchmark	Selected Thread	SASS instructions [#]	Kernel time [ns]	Faults [#]
matSum	All Threads	18	4,483	1,536
	Thread 0	24	4,489	832
matMul	All Threads	234	2,361,958	19,908
	Thread 0	456	2,529,229	14,144
histogram	All Threads	354	839,492	30,208
	Thread 0	762	849,401	22,976
	Thread 1	762	848,201	22,976

B. Error Rate Distribution

Exploiting the described fault injection environment, CUBINJ, fault injection campaigns were carried out with the three

selected benchmark applications. Figure 5 represents the distribution of different errors. Please note that the number of faults for different versions of each application due to instrumentation is represented in Table I.

As it can be observed from Figure 5, the *matSum* application has a very low *hang* rate while *matMul* and *histogram* show around 8% and 15% of faults result in *hang* with a very small difference between the fault affecting all threads and only one thread. This is happening since *matSum* is an application without a synchronization point in the code, which results in a very low probability of *hang*. Further looking into the exact fault causing *hang* in *matSum* application, it turns out to be caused by faults corrupting a memory load instruction, an integer addition instruction, and especially the exit instruction at the end of the kernel. While for *matMul* and *histogram*, the situation is more complicated as some faults affecting the loop controls in the code are also causing *hang* (endless loop leading to timeout due to the 15 seconds limit). Besides, faults in the synchronization instructions such as SSY (Set Sync Relative Address) would also lead to *hang*.

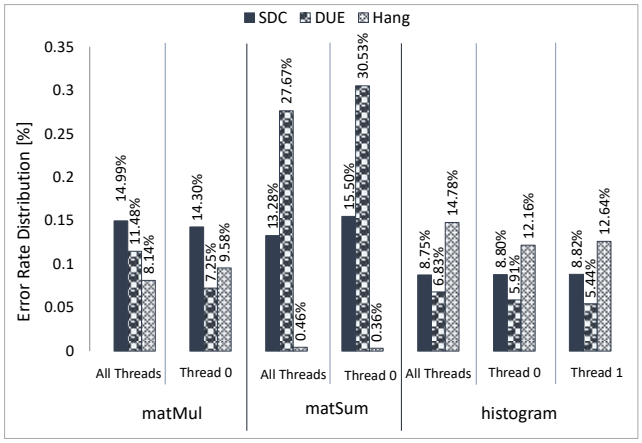

Fig. 5. The Error Rate Distribution Considering Bitflip in Instructions Affecting all Threads and Specified Thread

Comparing the distribution of errors for the two versions of injection in *All Threads* and *Thread0/1* of the same application shows a subtle difference. Furthermore, it could be noticed that the *hang* rate is higher in *Thread0/1* version.

When synchronization barriers are used, all the threads must reach the same synchronization point, otherwise, the application will wait. However, when instrumentation for fault injection is done, thread synchronization function calls are excluded from the fault list as only instructions surrounded by the tag instructions are targeted. Therefore, our initial expectation is that the fault injection campaign affecting only one thread should produce a lower *hang* rate. After investigating the exact fault leading to *hang*, it becomes trivial as these faults not only include those corrupting synchronization instructions such as SYNC and SSY, but also these corrupting loop related instructions (causing huge loop or dead loop) such as simple SEL and BRA instructions.

While the total number of faults is decreased, it is reasonable that the rate of *hang* is comparable or even higher in the situation when only one thread is affected. This shines the light on the necessity of a mitigation solution for parallel algorithm implementations as even when only one thread (one computational core) is affected by a fault, the whole application execution could be corrupted.

When comparing the histogram result in which *Thread 0* and *Thread 1* are affected, different error distributions could be observed. Please notice that the faults in the two cases are exactly the same, but due to the fact that the tasks are unevenly distributed among threads, different threads tolerate different criticality against possible fault. Thus, when designing mitigation techniques, selective strategies at the thread level could yield better results in terms of performance, power, and reliability trade-off.

C. SDC Comparison

Regarding the most undesired SDC, even if for the All Threads version and the *Thread0/1* version the SDC error rate is similar, the numbers of errors in the output are quite different. Table II reports the distribution of numbers of errors in the case of SDC.

For *matSum* application, if the affected thread is corrupting the calculation, most likely (96.90%), only one element in the output will be different as reported in Table II. Further investigations show that the faults are affecting the instructions which store the results into global memory at the last step.

TABLE II. NUMBER OF ERRORS IN OUTPUT IN CASE OF SCD

Benchmark	Selected Thread	One Error [%]	All Wrong [%]	Average Errors [#]
matSum	All Threads	0.00	61.76	126.53
	Thread 0	96.90	0.00	1.03
matMul	All Threads	0.00	73.35	254,414.31
	Thread 0	87.24	0.00	6.51
histogram	All Threads	0.53	53.33	181.99
	Thread 0	74.02	0.89	39.70
	Thread 1	72.77	0.89	39.50

For *matMul* application, it is more complicated as one element in the output does not depend on just one thread's calculation. Nonetheless, comparing the output data errors shows that for faults affecting all the threads resulting in SDC, most of the faults corrupt most of the output elements (73.35% of faults corrupt all 262.144 integers in the result of 512x512 matrix multiplication). On the other hand, for faults affecting only thread 0, most of the faults corrupt only one output element. However, some faults could corrupt up to 1024 elements as the algorithm which implemented the multiplication is divided into 32x32 sub-matrix operations. The special fault causing 1,024 wrong output elements is a fault injected into an SSY (Set Sync Relative Address) instruction causing an erroneous relative address corruption in the calculation of the while sub-matrix.

For the histogram application, the situation is even more complicated as 33,554,432 integers in the range between 0 and 255 are counted into 256 bins and two kernel functions are implemented instead of just one as in *matSum* and *matMul*. Nonetheless, a comparison of the percentage of fault causing just one error for thread 0 and thread 1 versions shows a vast difference. This difference is happening due to the unbalanced

distribution of workload among threads as in the implementation of *Thread 0* in the block (*threadIdx.x == 0*) is in charge of merging partially counted results into final results.

D. Impact of fault on kernel execution time

For the faults not causing any anomaly execution, i.e. the faults categorized into SDC and Masked, it could still affect performance in terms of kernel time. From the average kernel time as reported in Table I, we established a baseline for each application and set the error margin to 10% of kernel time around the average one. The percentage of faults either increases or decreases the kernel time by more than 10%, as it is represented in Figure 6.

As can be seen in Figure 6, the performance impact of *All Threads* version is much higher than the single thread version which is happening since the kernel finished when all threads are completed. Even though with performance monitoring, a portion of SDC could be detected, the actual percentage is still low if only one of the many cores is affected (corruption in single-thread execution). Further techniques such as Duplication With Comparison (DWC) and Triple Modular Redundance (TMR) should be employed for SDC detection and/or mitigation.

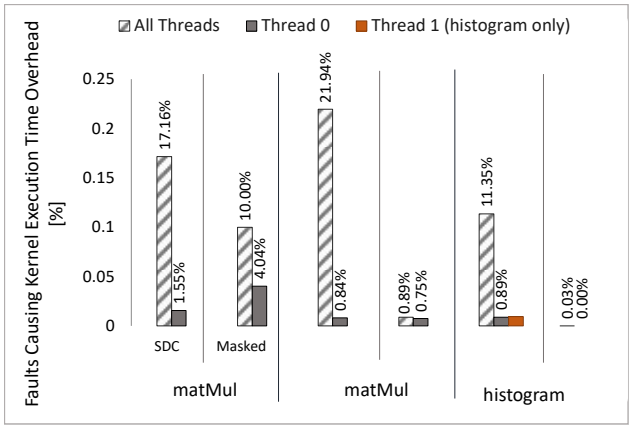

Fig. 6. Distribution of Faults Affecting Kernel Execution Time Considering Error Margin of 10% Average Kernel Time.

For matSum application, it shows a higher percentage of Masked faults, causing abnormal kernel time, particularly 1.79% of the faults are causing the kernel time to decrease. Some of those faults are related to an IADD instruction close to the end of the code which got corrupted, and others are related to the final EXIT instruction.

For matMul application, similarly, 0.84% of faults that cause SDC also lead to decreased kernel time when only one thread is affected. The faults mainly corrupted ISETP (Integer Set Predicate), SEL (Conditional Select/Move) and BRA (Branch to Relative Address) which are related to loop structures implemented in the application. Those faults cause some iterations to skip, thus, resulting in the reduced execution time. Besides, 0.79% of Masked faults also lead to decreased kernel time. While some faults are similar to the ones provoking SDCs, some faults are related to corrupted SSY and SYNC instructions for threads synchronization operation. It appears that the corrupted synchronization shortened execution time and luckily no race condition is triggered causing data error. However, for both cases, the exact chain of events leading the overall kernel time since only one thread is directly affected by the fault requires further investigation.

E. More information

As CUBINJ tool records other information including how the SASS instruction is corrupted, further analysis could be performed. For example, among the Masked faults, there are faults that are not actually corrupting the SASS instruction due to "don't care" bits in the instruction encoding. Besides, in some cases, the faulty cubin file could not be decoded by the cuobjdump tool due to invalid instruction coding. It could still be loaded into GPGPU device and executed where the information reported by the driver (e.g. dmesg in Linux system) could be utilized to identify the instruction and corresponding effects. This is not yet included in the current implementation, but it is under consideration.

F. Comparing with State-of-the-art Fault Injection Tools

In order to perform a comparison between the proposed fault injection tool and the state-of-the-art NVBitFI tool, as the latest fault injection tool developed by NVIDIA, we have used the same benchmark applications to perform fault injection analysis. Table III reports the characteristics of the selected benchmarks compiled and executed by NVBitFI tool.

Please notice that as it is expected, the number of SASS instructions is the same as compiling the applications using our proposed fault injection platforms, while the kernel time has been reported exploiting *nvprof*, the NVIDIA profiling tools for reporting and optimizing the performance of CUDA applications.

TABLE II. CHARACTERISTICS OF BENCHMARK APPLICATIONS EXECUTING ON NVBITFI TOOL

Benchmark	SASS instructions [#]	Kernel time [ns]
matSum	18	56,801
matMul	234	1,671,000
histogram	354	1,425,500

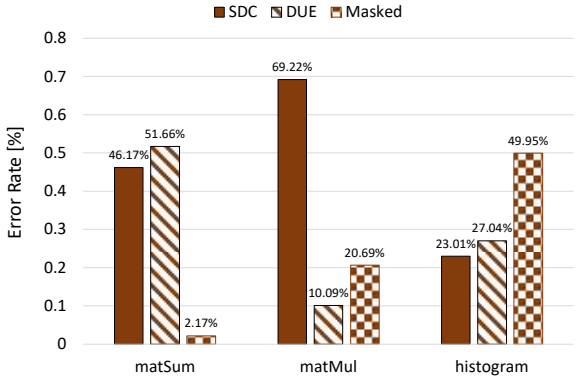

Fig. 7. Error Rate Report for 10,000 BitFlip Fault Injection Using NVBitFI Tool.

We have performed 10,000 fault injections, configuring the NVBitFI tool targeting the single bitflip in the general-purpose registers. As it has been mentioned before, contrary to our developed fault injection platform, using NVBitFI, it is not

possible to target a specific thread for injection and the target thread for fault injection is chosen randomly by the tool.

Figure 7 represents the results of 10,000 fault injections on the three chosen benchmarks. The results are classified the same way as our proposed tool. Please notice that during the fault injection, we have set the timeout threshold to 15 seconds, the same value as fault injection for our developed platform.

G. Performance Comparison on NVIDIA TX2 embedded GPU

A Multiply and Accumulate (MAC) algorithm has been tested on the Jetson TX2 NVIDIA development kit implemented on the basis of the techniques illustrated in [21]. The algorithm has been implemented following five different approaches:

1. *Original Application*: The original application is related to the execution flow using the plain implementation without any mitigation feature.
2. *Kernel Threads Duplication using Single Memory*: The Kernel Threads duplication using single memory is an alternative solution where the threads computation is duplicated, and the computational memory area is equivalent.
3. *Beacon Over Data Transfer using Single Memory*: The Kernel Threads and the memory are equivalent to the original applications, but there is a further computation, executed during the data transfer from the device to the host.
4. *Threads Duplication using Duplicated Memory*: The Kernel Threads are duplicated, and the memory is duplicated. This is a typical configuration that it is used for duplication with comparison approach where the need is oriented to the checking of the operation between the two executions. For the purpose of this work, we only include the threads duplication on a double memory block.
5. *Beacon Over Data Transfer using Duplicated Memory*: The Kernel Threads and the memory are equivalent to the original applications, but there is a further computation, executed during the data transfer from the device to the host and the memory is duplicated.

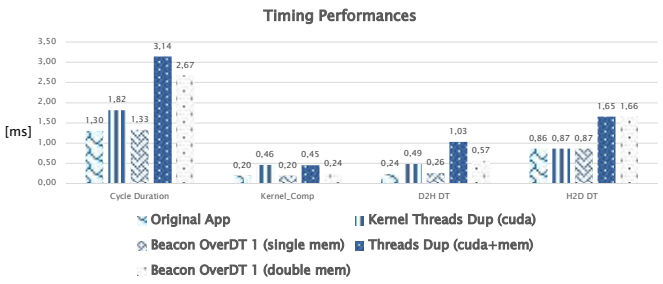

Fig. 8. Timing performance comparison between the original application and the beacon thread algorithm with two memory configurations.

We performed the experimental analysis by comparing the timing performance on a data package of 12,000 KB considering a cyclic and continuous computational stream and injecting a total of 120K single bit flip faults.

The fault injection results are illustrated in Table IV. The beacon threads over data transfer outperforms the soft error resiliency with respect to traditional duplication with comparison techniques within CUDA kernels. Besides, it is interesting to notice that the usage of the beacon threads are reducing the error detection latency. This is mainly due to the anticipated detection performed by the beacon thread with respect to the execution comparison done at the end of each computing cycle.

TABLE IV. FAULT INJECTION RESULT

Benchmark	SDC [%]	Detected SDC [%]	Detection Latency [ms]
Original	96.7	-	-
Kernel Dup Single Mem	68.2	84.7	2.14
Beacon OverDT Single Mem	12.5	98.0	1.20
Duplicated Kernel and Mem	23.8	87.4	3.08
Beacon OverDT Duplicated Mem	4.7	98.8	1.08

The performance results are depicted in figure 8, where the timing performances are illustrated in terms of execution time for: a unique algorithm cycle, the kernel threads and the data transfer time for device to host (D2H) and host to device (H2D) operations.

We observed that for the developed beacon threads over data transfer have an advantage of 37% with respect to state-of-the-art solution with the CUDA kernel thread's duplication. Furthermore, we compared the algorithms considering the duplicated memory area and we observed that the beacon approach provides a relevant advantage even if the overall computing cycle duration is affected by the duplicated time needed for the data transfer threads.

VI. CONCLUSION AND FUTURE WORK

A new reliability analysis and mitigation approach for GPGPU is presented in this paper. The analysis is based on a fault injection solution targeting the SASS instructions executed on real devices. Based on the implementation of the CUDA code, CUBINJ is able to target all threads, or some specific thread(s) determined by a condition statement in the code. Three benchmark applications were selected to assess the effectiveness of the approach, furthermore a MAC application has been analyzed and mitigated on a Jetson TX2 development board manufactured by NVIDIA. The mitigation solution presented and based on detection thread executed in parallel on the data transfer between host and GPGPU devices shows an improved resiliency of more than 37% with a limited performance degradation.

As future work, we are building SASS encoding maps to further exploit the possibility for instrumentation to support the fault injection environment and to extend the fault injection analysis and mitigation on GPGPU clusters.

REFERENCES

[1] H. Nvdia, "NVIDIA Unveils Quadro RTX, World's First Ray-Tracing GPU," 2018. [Online]. Available: https://NVIDIAnews.NVIDIA.com/news/NVIDIA-unveils-quadrortx worlds-first-ray-tracing-gpu.

[2] D. C. Anderson and J. Cychosz, "An introduction to ray tracing," Image Vis. Comput., 1990.

[3] W. Shi, M. B. Alawieh, X. Li, and H. Yu, "Algorithm and hardware implementation for visual perception system in autonomous vehicle: A survey," Integration, the VLSI Journal. 2017.

[4] V. Campmany, S. Silva, A. Espinosa, J. C. Moure, D. Vázquez, and A. M. López, “GPU-based pedestrian detection for autonomous driving,” in Procedia Computer Science, 2016.

[5] S. Kato et al., “Autoware on Board: Enabling Autonomous Vehicles with Embedded Systems,” in Proceedings - 9th ACM/IEEE International Conference on Cyber-Physical Systems, ICCPS 2018, 2018.

[6] S. Azimi, B. Du, L. Sterpone, “Evaluation of transient errors in GPGPUs for safety critical applications: An effective simulation-based fault injection environment”, Journal of System Architecture, vol. 75, pp 95-106, 2017.

[7] L. Sterpone, S. Azimi and B. Du, "A selective mapper for the mitigation of SETs on rad-hard RTG4 flash-based FPGAs," 2016 16th European Conference on Radiation and Its Effects on Components and Systems (RADECS), 2016, pp. 1-4.

[8] B. Nie, D. Tiwari, S. Gupta, E. Smirni, and J. H. Rogers, “A largescale study of soft-errors on GPUs in the field,” in Proceedings - International Symposium on High-Performance Computer Architecture, 2016.

[9] B. Fang, K. Pattabiraman, M. Ripeanu and S. Gurumurthi, "A Systematic Methodology for Evaluating the Error Resilience of GPGPU Applications," in IEEE Transactions on Parallel and Distributed Systems, vol. 27, no. 12, pp. 3397-3411, 1 Dec. 2016.

[10] L. Sterpone, B. Du, S. Azimi, Radiation-induced single event transients modeling and testing on nanometric flash-based technologies,Microelectronics Reliability, Volume 55, Issues 9–10, 2015, Pages 2087-2091.

[11] S. Tselonis and D. Gizopoulos, “GUFI: A framework for GPUs reliability assessment,” in ISPASS 2016 - International Symposium on Performance Analysis of Systems and Software, 2016, pp. 90–100.

[12] S. K. S. Hari, T. Tsai, M. Stephenson, S. W. Keckler, and J. Emer, “SASSIFI: An architecture-level fault injection tool for GPU application resilience evaluation,” in ISPASS 2017 – IEEE International Symposium on Performance Analysis of Systems and Software, 2017.

[13] T. Tsai, et.al., "NVBitFI: Dynamic Fault Injection for GPUs," 2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), 2021, pp. 284-291.

[14] S. Collange, M. Daumas, D. Defour, and D. Parello, “Barra: A parallel functional simulator for GPGPU,” in Proceedings - 18th Annual IEEE/ACM International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems, MASCOTS 2010, 2010.

[15] N. Maruyama, A. Nukada and S. Matsuoka, "A high-performance fault-tolerant software framework for memory on commodity GPUs," 2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS), Atlanta, GA, 2010, pp. 1-12.

[16] J. Power, J. Hestness, M. S. Orr, M. D. Hill, and D. A. Wood, “Gem5-gpu: A heterogeneous CPU-GPU simulator,” IEEE Comput. Archit. Lett., 2015.

[17] R. Ubal, L. Pedro, Z. Chen, and D. R. Kaeli, “The Multi2Sim Simulation Framework,” Architecture, 2010.

[18] A. Bakhoda, G. L. Yuan, W. W. L. Fung, H. Wong, and T. M. Aamodt, “Analyzing CUDA workloads using a detailed GPU simulator,” in ISPASS 2009 - International Symposium on Performance Analysis of Systems and Software, 2009.

[19] K. Andryc, M. Merchant, and R. Tessier, “FlexGrip: A soft GPGPU for FPGAs,” in FPT 2013 - Proceedings of the 2013 International Conference on Field Programmable Technology, 2013.

[20] B. Du, J. E. R. Condia, and M. S. Reorda, “An extended model to support detailed GPGPU reliability analysis,” in Proceedings – 2019 14th IEEE International Conference on Design and Technology of Integrated Systems In Nanoscale Era, DTIS 2019

[21] D. Sabena, M. S. Reorda, L. Sterpone, P. Rech and L. Carro, "On the evaluation of soft-errors detection techniques for GPGPUs," 2013 8th IEEE Design and Test Symposium, Marrakesh, 2013, pp. 1-6.