Concurrent Detection of Failures in GPU Control Logic for Reliable Parallel Computing

Hiroaki Itsuji
Center for Technology Innovation
Hitachi, Ltd. R & D Group
Yokohama, Japan
hiroaki.itsuji.pc@hitachi.com

Takumi Uezono
Center for Technology Innovation
Hitachi, Ltd. R & D Group
Yokohama, Japan
takumi.uezono.bo@hitachi.com

Tadanobu Toba
Center for Technology Innovation
Hitachi, Ltd. R & D Group
Yokohama, Japan
tadanobu.toba.ee@hitachi.com

Kojiro Ito
Dept. of Information Systems Engineering
Osaka University
Suita, Japan
i-kojiro@ist.osaka-u.ac.jp

Masanori Hashimoto
Dept. of Information Systems Engineering
Osaka University
Suita, Japan
hasimoto@ist.osaka-u.ac.jp

Abstract—The reliability of GPUs is becoming a major concern due to the increased probability of failures and the high vulnerability of GPUs compared to conventional CPUs in terms of tasks per failure. While there are extensive countermeasures against failures in GPU data units, there are fewer countermeasures for failures in GPU control logics. Currently, software-based techniques, such as inserting signature codes for detecting GPU control-logic failures by comparing the expected signature value with the current signature value, are being utilized. However, in the conventional software-based techniques, application calculations, signature calculations, and signature comparison calculations are executed in sequence, which degrades the application throughputs. We have developed a software-based technique that concurrently detects GPU control-logic failures in a running application while largely maintaining its throughput. Experimental results show that when our technique concurrently executed application calculations, signature calculations, and signature comparison calculations for a matrix multiplication application, the application throughput remains 78% of the original one, whereas 62% is reported in literature. We also developed fault injection simulators specialized for injecting GPU-specific control-logic faults into GPU intermediate codes and found that 100% of GPU-specific failures could be detected both during and after application execution. The proposed approach can be utilized for a wide variety of safety- and reliability-critical applications.

Keywords—GPU, reliability, soft error, error detection, fault injection

I. Introduction

Parallel computing devices represented by graphic processing units (GPUs) are widely used for general-purpose and computing-intensive applications. While their high throughput is drawing attention, the reliability of GPUs is becoming a major concern. Generally, GPUs are 1–2 orders of magnitude more vulnerable than central processing units (CPUs) in terms of tasks per failure [1]. There are two types of failure in GPUs: temporal failures such as soft errors (e.g., bit flips in memory) due to environmental neutrons [2, 3] and permanent failures such as stuck-at-faults and burnout. These failures typically cause undesirable results leading to critical accidents or huge losses [4, 5]. Temporal and permanent failures originate from faults in the data units (such as instruction caches and processing elements) or in the control logics that control the flows of parallel computations. Recent analyses of GPU instruction codes have suggested that control logics are relatively more vulnerable than data units in terms of the probability of faults in the hardware components resulting in incorrect program outputs, i.e., silent data corruption (SDC) [6, 7]. Hence, it is important to develop countermeasures against failures not only in data units but also in control logics.

In terms of countermeasures on the hardware side, redundancy-based techniques detect the control logic failures by comparing program outputs from multiple modules. However, these techniques need a few times the original hardware resource and increase the total failure probability and the number of system halts. On the software side, the control logic failures can be detected by inserting signature codes into program codes. Although the signature-based technique, which was originally developed for CPUs, can be effective for checking control flows by comparing the updated signature value with the expected value at each checkpoint [8–10], it significantly degrades the application throughput because application calculations, signature calculations, and signature comparison calculations are executed in sequence. For example, the widely known signature-based techniques CFCSS [9] and YACCA [10] degrade application throughputs to 62% and 36%, respectively, for matrix multiplication applications [8]. Such throughput degradation is unacceptable for applications that require both high throughput and high reliability without any penalty to the hardware resource.

978-1-7281-9113-3/20/$31.00 ©2020 IEEE

In light of the above background, we have developed a software-based technique that concurrently detects GPU control-logic failures in a running application while maintaining its throughput. The key concept of our proposed technique is that it concurrently executes application calculations, signature calculations, and signature comparison calculations. In section II, we describe the general GPU architecture, the classification of GPU control-logic failures, and the proposed technique for detecting GPU control logics. In section III, we present a fault-injection test framework for investigating the capability of detectable GPU control-logic failures with the proposed technique. Evaluation results are reported in section IV. We conclude in section V with a brief summary.

II. PROPOSED APPROACH

First, we discuss the general GPU computing framework. Next, we discuss the classification of GPU control-logic failures. Finally, we describe the proposed software-based technique that concurrently detects one critical type of GPU control-logic failure.

A. GPU architecture

Fig. 1 shows a conceptual drawing of a general CPU-GPU framework based on NVIDIA GPUs utilizing the single-instruction multiple thread (SIMT) architecture. The CPU launches parallel-computing kernels while storing data in the global memory as input data for parallel computations. The GPU operates in accordance with instructions from the kernel invoked from the CPU. Then, the instructions are distributed from the instruction cache to the warp scheduler for warp execution. The warp is a group of 32 threads that are simultaneously executed in 32 cores. Each thread in one warp is allocated to one core, and each core in the warp executes the same instructions for the different data in a round-robin fashion. Execution results are stored in the register file, after which they are transferred to the L2 cache, global memory, and CPU. Optionally, L1 cache/shared memory can be used for more efficient data transfer. Even though this hardware information is publically available, the details of the control logics are not disclosed. Therefore, we refer to the architecture of FlexGrip [11] based on the NVIDIA G80 GPU for classifying the temporal failures in the control logics.

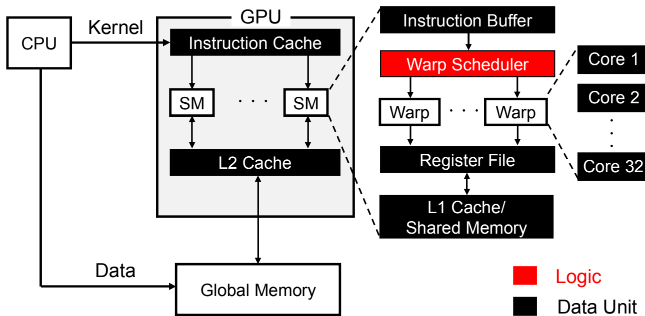

Fig. 1 General CPU-GPU framework for parallel computing.

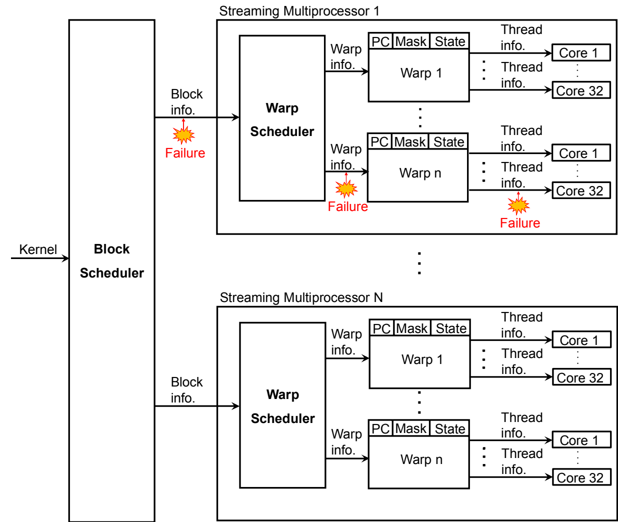

Fig. 2 Three types of GPU control-logic failure.

B. GPU control-logic failure

Faults in GPU control logics lead to one of the following results: (1) no effect on program outputs, (2) detected unrecoverable errors (DUEs) such as application crashes and system hangs, or (3) SDCs. (1) can be neglected, and (2) can be detected by GPU drivers [12] or software watchdogs. We focus on (3) since GPU control-logic failures leading to SDCs may not be detected by GPU drivers or software watchdogs.

We classify GPU-control failures possibly leading to SDCs based on FlexGrip. In FlexGrip, as shown in Fig. 2, the block scheduler distributes block information (e.g., block index) to the warp scheduler, and the warp scheduler then distributes warp information (e.g., program counter (PC), active mask, and warp state) to each warp. The PC is used for directing instructions stored in the instruction buffer. The active mask is used for executing only active threads in each warp. The warp state is used for representing the states: ready, active, waiting, or finished. On the basis of this hardware, we classify GPU-control failures possibly leading to SDCs into three types:

- **Block-level failure**: One thread block is not executed or is executed abnormally. This failure may stem from a fault in the data path or in the registers in the schedulers.
- **Warp-level failure**: One warp is not executed or it does not execute instructions in the expected order. This failure stems from a fault in the data path, PC, or state.
- **Thread-level failure**: One thread in a warp is not executed or is executed abnormally. This failure stems from a fault in the data path or active mask.

The block information is transferred from the block scheduler to each warp in several cycles when a kernel is launched. On the other hand, the warp information remains in the warp scheduler and is susceptible to failures during the application execution time (> several cycles). Also, the probability of faults in the warp scheduler resulting in SDCs can be higher than that in the data unit such as register file and shared

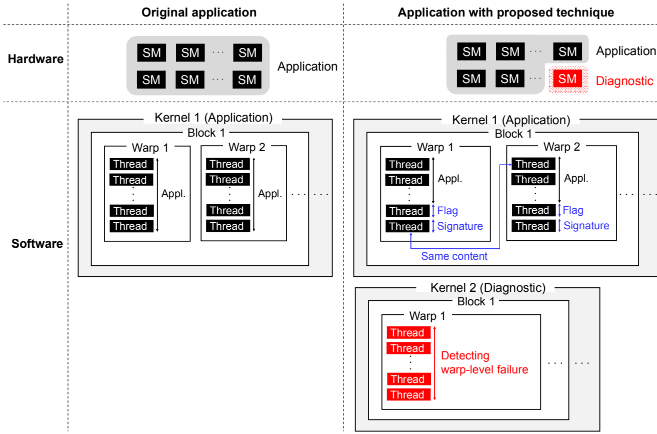

Fig. 3 Concept of proposed technique in terms of resource assignments.

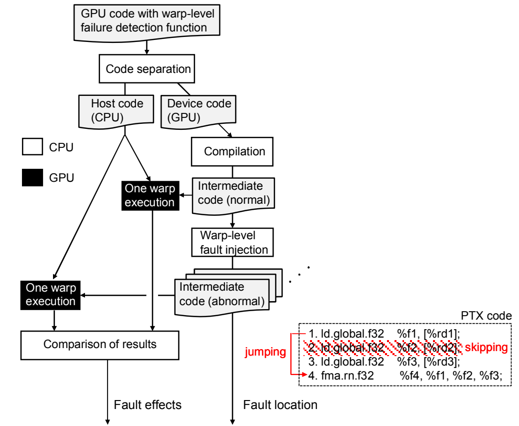

Fig. 4 Test flow for investigating warp-level failure detection capability.

memory [6]. Therefore, the frequency of block-level failure is expected to be lower than the frequency of warp-level failure. Moreover, compared to warp-level failure, thread-level failure does not affect the program output significantly, since a warp executes 32 threads (> 1 thread) simultaneously. Hence, we focus on warp-level failure. Meanwhile, obtaining insights on the frequency of each failure would be helpful for robust system design and should be studied, but it is beyond the scope of this work.

C. *Proposed failure-detection technique*

The concept of our proposed technique is shown in Fig. 3. In original codes, before implementing our technique, all available SMs and all threads are usually allocated to a specific application. After the implementation, one SM is assigned for diagnosing GPU control logics at the warp level, while the remaining ones are assigned to the application. Here, two threads in one warp (32 threads) are assigned for signature and flag, respectively. In one warp, application, signature, and flag threads execute the same code simultaneously with one common PC. The flag value is used to indicate that the warp is ready to be diagnosed and updated after the calculated results are written to the global memory. In one code, a conditional instruction is implemented so that only the flag thread meets a specific condition on the thread index and write the non-zero flag value in the global memory. The signature thread executes an identical application thread in the adjacent warp and write calculated results to the global memory.

For example, when the flag value in warps 1 and 2 in block 1 is the same, the outputs from the signature thread in warp 1 and the application thread in warp 2 are compared with the diagnostic kernel. The diagnostic kernel, which is launched concurrently with the application kernel, compares the outputs calculated with the application and signature threads repeatedly during the application execution for detecting warp-level failures. The calculated value is stored in the global memory so that it can be compared with the diagnostic kernel. In addition to cases where the comparison results are not the same, the diagnostic kernel detects warp-level failure when the flag value is not an expected value after the application execution. By selecting programs to be executed based on user inputs, users can enable or disable the diagnosis function in executing applications.

III. FAULT-INJECTION TEST FRAMEWORK

In this section, we introduce our test framework for investigating whether the proposed technique can detect warp-level failures.

Various fault injection frameworks have been developed, and the abstraction level of fault injection is different depending on the framework. Source and assembly (SASS) codes can be used for reproducing warp-level failures most accurately in cycle levels. However, SASS codes depend on the GPU architectures, and it is time-consuming to derive generalized results by injecting warp-level faults into the SASS codes of various GPU architectures. We focus on parallel thread execution (PTX) codes, which do not depend on the GPU architecture. PTX codes are widely utilized for fault injections [13,14]. For example, GUFI [13] can inject faults into the active mask, PC, and warp state. However, the number of GPU products to which GUFI is applicable is limited. Therefore, we built an original test framework specialized for simulating warp-level failures in GPU control logics.

Fig. 4 shows the test flow of simulating warp-level failures. First, GPU codes with warp-level failure detection functionality are divided into host code (e.g., C++ code) and device code (e.g., CUDA code). For the host code, the kernel launch setting is changed so that only one specific warp is executed. Then, the device code is compiled to generate normal PTX code. Then, based on the normal PTX code, abnormal PTX codes that include a small piece of fault injection code are generated. In this work, faults leading to warp-level failures are divided into unexpected skipping of one instruction and jumping of instructions. The skipping of one instruction during the application execution due to faults in the warp state or PCs is reproduced by adding instructions to the PTX codes that disable one specific instruction once per application run. The jumping from one arbitrary point to another arbitrary point (excluding skipping) once per application run due to faults in the PC is simulated by adding instructions to the PTX codes. In this way, we generate abnormal PTX codes assuming all possible points in skipping and start/end points in jumping. Then, the normal and abnormal PTX codes are loaded from the host code and executed one by one. We compared the execution results from normal and abnormal PTX codes based on the proposed technique to investigate the warp-level failure detection capability.

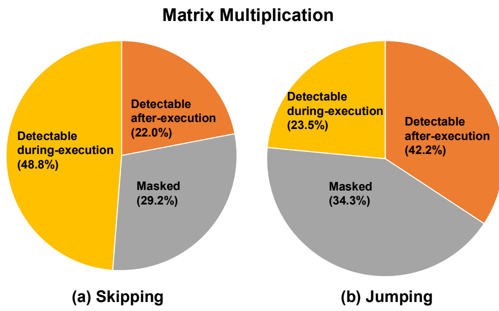

Fig. 5 Test results of matrix multiplication application for (a) skipping and (b) jumping failures. Fault injection results were classified with detectable during execution and detectable after execution with the proposed technique, and masked (correct output).

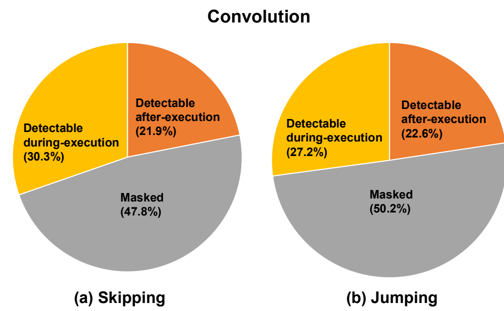

Fig. 6 Test results of convolution application for (a) skipping and (b) jumping failures. Fault injection results were classified with detectable during execution and detectable after execution with the proposed technique, and masked (correct output).

IV. EVALUATION

In this section, we evaluate the proposed technique in terms of its warp-level failure detection capability and application throughputs. As a test vehicle, we used the NVIDIA Quadro P2000 with eight SMs, each of which had 1024 cores.

A. *Capability of detecting warp-level failure*

We injected warp-level faults into two basic applications with the functionality of detecting warp-level failures. For simplicity, the applications did not use shared memory.

Fig. 5 shows the results of (a) skipping failures and (b) jumping failures for the matrix multiplication application. For skipping failures, we found that 48.8% of warp-level faults that led to an incorrect program output value could be detected during the application execution with the diagnostic kernel, since the flag value was correct. However, 22.0% of these faults could not be detected because the flag value was incorrect and then a comparison with the diagnostic kernel could not be triggered during application execution. Interestingly, the 22.0% of warp-level faults all caused program outputs of "0" including the flag value. Therefore, the 22.0% of warp-level faults could be detected after application execution by checking the incorrect flag value with the diagnostic kernel. The observed "0" program output was probably because almost all of the GPU instructions dealt with register or global memory addresses where the temporal or final calculation results are stored. This observed "0" program output is a critical issue when it comes to reliable GPU operations since it may cause failures in, for example, detecting objects in image processing applications. The remaining 29.2% of the faults were masked and did not cause SDCs. Under all fault-injection conditions, no termination of streams of processes—applications, diagnostics, and data transfer (read/write)—was observed. Hence, we could detect 100% of the warp-level failures with the diagnostic kernel. As for the jumping failures, we could detect 23.5% of warp-level faults during the execution and 42.2% of warp-level faults (all "0" program outputs were 98.7%) after the execution, and 34.3% of warp-level faults were masked. Moreover, as with the skipping failures, no termination of stream was observed. These results demonstrate that the proposed technique can detect 100% of the warp-level failures.

Fig. 6 shows the results of (a) skipping failures and (b) jumping failures for the convolution application. In this application, row-convolution and column-convolution kernels were separately and sequentially launched. We injected warp-level faults into the latter column convolution kernel. As with the matrix multiplication application, all "0" program outputs were observed, and 21.9% of the skipping failures (all "0" program outputs were 80.3%) and 22.6% of the jumping failures (all "0" program outputs were 81.6%) could be detected after application execution. Similarly, no termination of stream was observed. These results demonstrate that the proposed technique can again detect 100% of the warp-level failures. We can see from Figs. 5 and 6 that the ratios of detectable during-execution, detectable after-execution, and masked are highly dependent on the applications.

B. *Application throughput and memory overhead*

TABLE I shows the application throughputs. Here, "with shared memory" means that shared memory was used as cache for the global memory, while "without shared memory" means the shared memory was not used. Experimental application throughputs were measured using the NVIDIA Nsight profiler. The time needed for diagnosing the flag value and thus detecting warp-level failures leading to all "0" program outputs after the application execution was included in the application execution time. Note that the experimental application throughputs were slightly increased with the use of shared memory because the number of memory accesses to the global memory while executing the application was reduced, and thus the number of memory contentions was decreased. For a matrix multiplication application, 78% of application throughputs were 16% higher than those of a conventional signature-based approach (62%, CFCSS [9])

TABLE II shows the global memory overheads. This table highlights that the memory overheads involved with the proposed technique were smaller than those of the conventional double-modular redundancy technique which necessitates the twofold memory resource. The global memory overheads were determined by the number of global memory locations additionally assigned for the kernels.

The results above demonstrate the effectiveness of the proposed technique in terms of warp-level failure detection capability, application throughput, and memory overhead. Our proposed technique can be applied to various applications merely by changing the original codes to assign the signature and flag threads in each warp and launch the diagnostic kernel. Our future work will apply the proposed technique to more complex applications such as neural network-based inference to investigate its scalability.

TABLE I. APPLICATION THROUGHPUT FOR EACH APPLICATION.

Matrix multiplication (256 × 256)		**Matrix multiplication (512 × 512)**		**Convolution (384 × 384)**	
Without shared memory	***With shared memory***	***Without shared Memory***	***With shared memory***	***Without shared memory***	***With shared memory***
71%	74%	76%	78%	68%	71%

TABLE II. GLOBAL MEMORY OVERHEAD FOR EACH APPLICATION.

	Matrix multiplication (256 × 256)	**Matrix multiplication (512 × 512)**	**Convolution (384 × 384)**
Global memory overhead	16%	16%	19%

V. CONCLUSION

We have developed a software-based technique for safety- and reliability-critical GPU applications that concurrently detects GPU control-logic failures while the application is running. We categorized GPU control-logic failures as block-level, warp-level, and thread-level failures and focused on the warp-level failure affecting 32 threads. Experimental results showed that the proposed technique can detect warp-level failures leading to SDCs while mostly maintaining the application throughputs. We ran an application using the proposed technique and found that concurrently executed application calculations, signature calculations, and signature comparison calculations achieved throughputs of 78% for matrix multiplication. Application throughputs were 16% higher than those of a conventional signature-based approach for a matrix multiplication application. We also developed a test framework and showed through PTX-level fault-injection testing that 100% of warp-level failures can be detected with the proposed technique both during and after the running of matrix multiplication and convolution applications. These results demonstrate that the proposed technique can detect warp-level failures leading to critical SDCs where program outputs all become "0". These critical SDCs are important because they may lead to failure in object detection for image processing applications. Our proposed warp-level failure detection technique with only 16% memory overhead for matrix multiplication applications can apply to various GPU applications simply by modifying the original GPU codes.

ACKNOWLEDGEMENT

This research was supported by Program on Open Innovation Platform with Enterprises, Research Institute and Academia, Japan Science and Technology Agency (JST, OPERA, JPMJOP1721).

REFERENCES

[1] A. Chatzidimitriou, M. Kaliorakis, S. Tselonis, and D. Gizopoulos, "Performance-Aware Reliability Assessment of Heterogeneous Chips," IEEE VLSI Test Symposium, pp. 1-6, April 2017.

[2] E. Ibe, H. Taniguchi, Y. Yahagi, K. Shimbo, and T. Toba, "Impact of Scaling on Neutron-Induced Soft Error in SRAMs from a 250 nm to a 22 nm Design Rule," IEEE Transactions on Electron Devices, vol. 57, pp. 1527-1538, May 2010.

[3] T. Uezono, T. Toba, K. Shimbo, F. Nagasaki, and K. Kawamura, "Evaluation Technique for Soft-Error Rate in Terrestrial Environment utilizing Low-Energy Neutron Irradiation," IEEE Asian Test Symposium, pp. 293-297, November 2016.

[4] G. Li, S. K. S. Hari, M. Sullivan, T. Tsai, K. Pattabiraman, J. Emer, and S. W. Keckler, "Understanding Error Propagation in Deep Learning Neural Network (DNN) Accelerators and Applications," International Conference for High Performance Computing, Networking, Storage and Analysis, pp. 1-12, November 2017.

[5] A. Lotfi, S. Hukerikar, K. Balasubramanian, P. Racunas, N. Saxena, R. Bramley, and Y. Huang, "Resiliency of Automotive Object Detection Networks on GPU Architectures," IEEE International Test Conference, paper 12.1, pp. 1-9, November 2019.

[6] J. Tan, N. Goswami, T. Li, and X. Fu, "Analyzing Soft-Error Vulnerability on GPGPU Microarchitecture," IEEE International Symposium on Workload Characterization, pp. 226-235, November 2011.

[7] N. Farazmand, R. Ubal, and D. Kaeli, "Statistical Fault Injection-Based AVF Analysis of a GPU Architecture," IEEE workshop on Silicon Errors in Logic – System Effects, January 2012.

[8] J. Vankeirsbilck, V. B. Thati, H. Hallez, and J. Boydens, "Inter-Block Jump Detection Techniques: a Study," IEEE XXV International Scientific Conference Electronics, September 2016.

[9] N. Oh, R. P. Shirvani, and E. J. McCluskey, "Control-Flow Checking by Software Signatures," IEEE Transactions on Reliability, vol. 51, pp. 111-122, March 2002.

[10] O. Goloubeva, M. Rebaudengo, M. S. Reorda, and M. Violante, "Soft-Error Detection using Control Flow Assertions," IEEE Symposium on Defect and Fault Tolerance in VLSI Systems, December 2003.

[11] K. Andryc, M. Merchant, and R. Tessier, "FlexGrip: A Soft GPGPU for FPGAs," International Conference on Field-Programmable Technology, pp. 230-237, 2013.

[12] Xid Errors, NVIDIA (2020) [Online]. https://docs.nvidia.com/deploy/xid-errors/index.html

[13] S. Tselonis and D. Gizopoulos, "GUFI: a Framework for GPUs Reliability Assessment," IEEE International Symposium on Performance Analysis of Systems and Software, pp. 90-100, April 2016.

[14] B. Fang, K. Pattabiraman, M. Ripeanu, and S. Gurumurthi, "GPU-Qin: a Methodology for Evaluating the Error Resilience of GPGPU Applications," IEEE International Symposium on Performance Analysis of Systems and Software, pp. 221-230, March 2014.