Analyzing DUE Errors on GPUs With Neutron Irradiation Test and Fault Injection to Control Flow

Kojiro Ito, Yangchao Zhang, Hiroaki Itsuji, *Member, IEEE*, Takumi Uezono, *Member, IEEE*, Tadanobu Toba, and Masanori Hashimoto, *Senior Member, IEEE*

***Abstract*— As GPU applications expand, the reliability of GPU is drawing more attention since even reliability-demanding applications are executed on GPUs. Silent data corruption (SDC) is widely studied both in irradiation experiments and fault injection experiments. On the other hand, detectable uncorrected error (DUE) is not well studied. This work focuses on DUEs reported by the GPU driver and analyzes those observed in fault injection and neutron irradiation experiments, where faults are injected in the control flow to change the program counter value unexpectedly. The DUE errors of GPU engine exception, GPU memory page fault, and GPU processing stop are observed in both the experiments. On the other hand, the DUE error categorized as internal microcontroller halt by the GPU driver, which is not found in the fault injection experiment, is observed frequently, suggesting the necessity of investigating the failures originating from the faults in the components invisible to programmers.**

***Index Terms*— GPU, neutron, silent data corruption (SDC), soft error.**

I. Introduction

In A terrestrial environment, neutrons in the secondary cosmic ray are a severe concern for reliability-demanding applications. One such application is autonomous driving, which requires a massive amount of computation on GPUs. Therefore, soft errors occurring in GPUs are drawing a lot of attention, and several irradiation experiments are performed and reported to evaluate the soft error immunity [1]–[5]. de Oliveira *et al.* [1] completed a pioneering work that irradiates modern GPU cards and evaluates the error rates of parallel applications. dos Santos *et al.* [2] evaluated convolutional neural network applications running on various GPU architectures. Lotfi *et al.* [3] reported the resiliency of object detection applications running on GPUs, where the bounding box mismatch is categorized into small shift, inclusion, and nonoverlapping, and their proportions are demonstrated. The impacts of FinFET and ECC are evaluated by Lunardi *et al.* [4], and the effects of core architecture and mixed precision are studied by Basso *et al.* [5].

On the other hand, the commercial GPU has difficulty in radiation immunity characterization and estimation since the circuit structure is only disclosed partially. For example, the numbers of cores and registers and the sizes of shared and cache memories are available since they are necessary for programmers to develop applications. For such visible memory components, the error rate characterization is performed with neutron irradiation tests [1], and fault injection experiments with an architecture-level fault injection tool called SASSIFI [6] are widely performed (e.g., [2], [7]–[11]). On the other hand, pipeline registers, FFs, and registers in datapaths, schedulers, and dispatchers must exist in GPUs, but their counts are unknown. Our previous work pointed out that the circuit components invisible to programmers contributed to silent data corruption (SDC) more than the visible memory components [12].

On the other hand, detectable uncorrected error (DUE) is less studied though the DUE occurs frequently, and its occurrence rate is comparable to the SDC rate [1]. The DUE categorization and consistency analysis with fault injection are crucial to understand the error occurrence mechanism and improve the application reliability. However, the previous works of radiation experiments [1]–[5], which were mentioned in a previous paragraph, and fault injection experiments [2], [7]–[11] focused on the categorization between SDC and DUE and SDC pattern analysis, and detailed categorization of DUE was not investigated. Lunardi *et al.* [7] analyzed transient fault propagation to the application output and reported that not all the output errors observed under radiation could be replicated in fault injection. Previlon *et al.* [8] categorized the DUE errors observed in fault injection into DUE and potential DUE, but the correlation with irradiation experiments is not investigated. Davidson and Bridges [9] evaluated the error resilience of image processing applications for space and reported that general-purpose registers had a higher contribution to SDC and DUE than predicate register (PR), conditional code (CC), and memory store. Ibrahim *et al.* [10] studied the error resilience of deep neural networks with fault injection,

Manuscript received April 30, 2021; revised June 14, 2021; accepted July 15, 2021. Date of publication July 21, 2021; date of current version August 16, 2021. This work was supported in part by the Japan Science and Technology Agency, Program on Open Innovation Platform with Enterprises, Research Institute and Academia (JST-OPERA) Program under Grant JPMJOP1721 and in part by Grant-in-Aid for Scientific Research (S) from Japan Society for the Promotion of Science (JSPS) under Grant JP19H05664.

Kojiro Ito, Yangchao Zhang, and Masanori Hashimoto are with the Department of Information Systems Engineering, Osaka University, Osaka 565-0871, Japan (e-mail: sankou.pinoko0601@gmail.com).

Hiroaki Itsuji, Takumi Uezono, and Tadanobu Toba are with the Center for Technology Innovation—Production Engineering, Research and Development Group, Hitachi Ltd., Yokohama 244-0817, Japan.

Color versions of one or more figures in this article are available at https://doi.org/10.1109/TNS.2021.3098845.

Digital Object Identifier 10.1109/TNS.2021.3098845

TABLE I
GPU DEVICE INFORMATION OF NVIDIA QUADRO P2000 AND GEFORCE GTX960

	Quadro P2000	GeForce GTX960
Architecture	Pascal	Maxwell
SMs	8	8
CUDA Cores	1,024	1,024
Process [nm]	16	28
Total Register Files [KB]	2,048	2,048
Total L1 Caches [KB]	960	960
Total Shared Memory [KB]	768	768
Shared L2 Cache [KB]	1,280	1024
GDDR5 Memory [GB]	5	2

TABLE II
GPU DEVICE AND PERFORMED EXPERIMENTS

Experiment	Fault Injection	Irradiation	
Program	Matrix multiplication		Object detection
P2000	✓	✓	✓
GTX960			✓

but their focus is only SDC. Previlon *et al.* [11] revealed that the error resilience was time-varying, and this characteristic was explained by code block analysis and exploited for accelerating fault injection.

This work evaluates and categorizes the DUE by analyzing the syslog that records GPU errors in a neutron irradiation test. This work also performs a fault injection experiment that disturbs the control flow by inserting a parallel thread execution (PTX) [13] code of jump. This fault injection experiment reproduces graphics engine exception, GPU memory page fault, and GPU processing stop, while the errors categorized as internal microcontroller halt are not reproduced. On the other hand, in the irradiation experiment, the errors of internal microcontroller halt occur frequently. This result suggests the hardware that is not disclosed to the users contributes to DUE errors substantially.

The rest of this article is organized as follows. Section II describes GPU cards under evaluation and explains GPU errors reported by the GPU driver. Section III discusses the fault injection method focusing on the control flow disturbance and presents the fault injection results. Section IV shows the results of neutron irradiation experiments, and Section V gives concluding remarks.

II. GPU ERRORS AND PROGRAMS UNDER EVALUATION

A. GPUs and Programs Under Evaluation

The target GPU devices in this work are NVIDIA Quadro P2000 and GeForce GTX960. Table I shows their main specifications. P2000 and GTX960 are based on Pascal and Maxwell architectures, respectively. Both GPUs do not have ECC capability for on-chip SRAMs, register files, or off-chip DRAMs. The operating system in the host PCs is Ubuntu 18.04.

Table II lists the experiments performed in this work. On P2000 and GTX960, we run Yolov3-tiny [14], a neural network-based object detection framework consisting of 13 convolutional layers for the irradiation experiment. During the experiment, 200 images selected from COCO dataset [15] are used for inference, that is, object detection. On P2000, we also run a single-precision matrix multiplication program implemented with C++ and CUDA in the irradiation experiment. The multiplicand and multiplier matrixes are both 240×240. This matrix multiplication program is used for the fault injection experiment explained in Section III as well.

B. GPU Errors

In this work, we evaluate Xid message [16], which is generated by NVIDIA driver and recorded in syslog. Xid is often used for large server maintenance and reliability analysis [17]. Xid message covers driver issues, hardware issues, NVIDIA software issues, and user application issues. We explain the Xid errors listed in Table III, where Xid 13, 31, 43, and 62 are observed in this work. Here, the errors found only once are omitted. Table III also indicates the error causes for each Xid, where they include HW error, driver error, user app error, system memory corruption, bus error, thermal issue, and frame buffer (FB) corruption. We first introduce Xid errors of 13, 31, 43, and 62. Note that the detailed explanations of those errors are found in [16].

- 13: Typically, this is an out-of-bounds error where the user has walked past the end of an array, but it could also be an illegal instruction, illegal register, or other cases. All the possible causes except HW bring this error.
- 31: This event is logged when a fault is reported by the memory management unit (MMU), such as when a functional unit makes illegal address access on the chip. Typically, these are application-level bugs but can also be driver bugs.
- 43: This event is logged when a user application hits a software-induced fault and must terminate. The GPU remains in a healthy state. In most cases, this is not indicative of a driver bug but rather a user application error.
- 62: This event is named internal microcontroller halt, but its detailed explanation is not provided in [16]. Also, the role of the internal microcontroller is not disclosed.

Additionally, we explain Xid 79 and 80 since they are the failures that originate from hardware issues in the GPU cards used in this work. Other Xid errors caused by hardware issues occur in ECC or video output, which is not utilized in our experiments.

- 79: This event is logged when GPU has fallen off the bus. Not only hardware issues but also thermal issues can cause this error.
- 80: This event is observed when corrupted data is sent to GPU. The error also occurs when there is a problem with the GPU bus.

III. FAULT INJECTION EXPERIMENT

A. Fault Injection Strategy

This work supposes that the control flow disturbance is one of the main causes for DUE and then performs fault injection that disturbs the control flow. SASSIFI [6], which is a popular fault injector for NVIDIA GPUs, can disturb

TABLE III
XID MEANING

Xid	Failure	Example	HW	Driver	UserApp	System Memory	Bus	Thermal	FB*
13	Graphics Engine Exception	out-of-bounds of an array		✓	✓	✓	✓	✓	✓
31	GPU memory page fault	an illegal address access		✓	✓				
43	GPU stopped processing	user application error		✓	✓				
62	Internal micro-controller halt		✓	✓				✓	
79	GPU has fallen off the bus		✓	✓		✓	✓	✓	
80	Corrupted data sent to GPU		✓	✓		✓	✓		✓

FB*: frame buffer

the control flow by manipulating the instructions that write to CC and a PR. Instead, this work manipulates the program counter (PC) to disturb the control flow directly. The direct PC manipulation is expected to accelerate the fault injection experiment since single instruction multiple thread (SIMT) stack, which includes PC, has higher susceptibility, that is, higher architectural vulnerability factor (AVF) [19]. Then, the injected faults disturb the control flow more frequently.

Let us review conventional fault injection methods regarding whether they can inject a fault into the PC. Fault injection at a high-level language [20] cannot reproduce the PC fault. SASSIFI cannot directly manipulate the PC as mentioned above. GPU-Qin [21] injects a fault into ALU and load-store unit (LSU), but it cannot inject a fault into the PC. GPU-SODA [19] can inject a fault into the PC, but the code is not available in public, and the reproduction is difficult. GUFI [22] can also inject a fault into the PC, but the applicable GPU architectures are limited. When writing a dedicated code to inject a fault into the PC, there is a significant difference in the necessary effort between SASS and PTX levels. Especially, SASS code is not portable to various GPU chips and architectures, which discouraged us from adopting SASS-level code insertion. Therefore, we took the approach that directly manipulates the PC with PTX-level code insertion. It should be noted that bit-flips did not directly cause our control flow disturbance in memory components and flip-flops. Therefore, the injected disturbance can be different from that triggered by bit-flips. However, it can analyze what Xid error could be triggered by the control flow disturbance.

B. Fault Injection Setup

The actual fault injection is performed as follows. First, we compile the CUDA code of the matrix multiplication and obtain the PTX code. Then, we edit the PTX code such that an unexpected jump happens, and the value of the PC varies in one of the active warps, where the threads in a warp share the same PC. The original program consists of 32 blocks, but only a single block that includes the warp with the faulty jump is executed in the fault injection experiment since all the warps share the same PTX code. The difference is only the thread and warp numbers. The control flow disturbance in each warp is expected to have the same impact, though, rigidly speaking, there might be a boundary or similar effect depending on the warp locations.

List 1 exemplifies a PTX code for fault injection. Instructions at lines 4 and 6 limit the number of unexpected jumps to 1. To specify the warp that executes

```
1  // reg. %r0 stores jump flag.
2  // reg. %r1 stores tid.y value
3
4  setp.eq.s32 %p0, %r0, 0;
5  // check if first jump (%r0==0) or not
6  add.u32 %r0, %r0, 1;
7  // to prevent jump after first jump
8  @!%p0 bra L0;
9  // jump to L0 when %r0!=0 at line 4
10 setp.eq.s32 %p0, %r1, 0;
11 // check if warp to jump or not
12 @%p0 bra L1;
13 // jump to L1 when %r1==0 at line
14 L0: // Label for no jump
15
16 // label L1 locates anywhere
```

Listing 1. Sample PTX code for fault injection.

unexpected jump as a fault, we use compare and branch instructions at lines 10 and 12. In PTX code, `%tid` register, which is denoted as `%tid.x`, `%tid.y`, and `%tid.z` in three-dimensional thread number specification, is a special register that stores the thread number. List 1 assumes that `%r1` copies from `%tid.y` beforehand. In this example, when `%r1` $= 0$, unexpected jump to `L1` occurs. Note that this warp specification depends on the user program and execution specification. Any place can be labeled as `L1` within the PTX code of the user program.

The PTX code of the matrix multiplication consists of the following three parts.

1) An inner loop that executes multiply-accumulate (MAC) computation.
2) An external loop that changes the address of the data for the MAC computation and stores the values in the shared memory.
3) Other codes outside the loops load arguments, configure threads and blocks, and store the data into memory.

For each code-line execution in one of the three parts, we inject a single fault during the program execution, which means an unexpected jump happens only in a particular loop in the first two cases. We tested all jump target addresses. Namely, all the combinations of the timing of fault injection and the jump address are tested. Note that due to the PTX specification, the jump address is limited within the PTX code of the matrix multiplication.

When performing the above fault injection, we must pay attention to how the inserted PTX code is executed on GPU since SASS optimization is applied. Then, we considered

TABLE IV
ERROR COUNTS IN MATRIX MULTIPLICATION UNDER FAULT INJECTION (E1 OPTION)

Xid	Errors			
	total	inner loop	external loop	outside loop
31	1,550	991	159	400
13, 31	55	0	0	55
13, 43	19,392	4,201	11,449	3,742
# of injected faults	180,943	152,967	21,720	6,256

TABLE V
ERROR COUNTS IN MATRIX MULTIPLICATION UNDER FAULT INJECTION (E2 OPTION)

Xid	Errors			
	total	inner loop	external loop	outside loop
31	1,557	999	159	399
13, 31	55	0	0	55
13, 43	19,341	4,189	11,422	3,730
# of injected faults	180,943	152,967	21,720	6,256

the two execution options below. In both options, the PTX code of the matrix multiplication is obtained by nvcc with optimization.

E1: The PTX code of the matrix multiplication and fault injection is called from cuModuleLoad function.

E2: The PTX code of the matrix multiplication and fault injection is compiled without optimization, that is, −O0 option. This binary code is called by cuModuleLoad function.

Each option has its advantage. In E1, the program behavior of the matrix multiplication is more consistent with that in the irradiation experiment since the PTX code of the matrix multiplication is called and executed similarly in the irradiation experiment. On the other hand, in E2, the fault injection is more consistent with our expectation since the perturbation in the fault injection execution due to SASS optimization and simplification is supposed to be minimized.

We use the same input matrices for multiplication in the experiment. For SDC detection, we prepare the golden output, that is, the product of the two matrices is calculated beforehand. During the experiment, the calculated result and the golden output are compared for every matrix multiplication to detect SDC.

C. DUE Results

Tables IV and V list the number of DUEs recorded in syslog when we inject faults into the matrix multiplication program with E1 and E2 options, respectively. The rows of 13, 31 means the errors of Xid 13 and 31 are recorded simultaneously for single fault injection. The row of 13, 43 is similar. The bottom row represents the total number of injected faults. For each executed PTX instruction, same number of faults are injected in the experiment. We can see only small differences between Tables IV and V, and the error count differences are lesser than 1%. We, therefore, think both execution options are applicable.

TABLE VI
DUE ERROR OCCURRENCE PER INJECTED FAULT FOR EACH XID IN MATRIX MULTIPLICATION (E1 OPTION)

Xid	DUE error occurrence per fault (%)			
	total	inner loop	external loop	outside loop
13	10.7	2.8	52.7	56.4
31	0.9	0.7	0.7	7.3
43	10.7	2.8	52.7	59.8

TABLE VII
SDC COUNTS IN MATRIX MULTIPLICATION UNDER FAULT INJECTION (E1 OPTION)

	Errors			
	total	inner loop	external loop	outside loop
SDC	178,707	152,967	20,282	5,458
mask	2,228	0	1,430	798
CUDA error	8	0	8	0
within-warp SDC (all0)	70,648 (418)	67,261 (375)	2,778 (18)	609 (25)
inter-warp SDC all0 (DUE)	33,368 (20,997)	16,198 (5,192)	12,454 (11,608)	4,716 (4,197)
inter-warp SDC non-all0	74,961	69,508	5,050	133
# of injected faults	180,943	152,967	21,720	6,256

Table VI lists the probabilities of DUE error occurrence per injected fault for each Xid with E1 option. We can see that the parts of the external loop and outside loop are sensitive to fault injection to the PC, and in this case, DUE is observed with more than 50% probability once the PC value changes unexpectedly. On the other hand, the inner loop is not so sensitive, even though the number of injected faults is significant since the execution count is high. Most of the inner loop faults do not cause DUE, and the DUE probability is a few percent at most. This sensitivity difference could be associated with the time-varying resilience reported in [11].

In total, the DUEs of Xid 13 and 43 are observed in 10.7% cases, and Xid 31 is found in less than 1% cases. Overall, even though faults are injected into the PC, the DUE occurrence proportion is less than 25% in our experimental setup. This low proportion might originate from the constraint of our fault injection method that the PC value is varied within the range of the matrix multiplication program. Further study with other fault injection methods, such as NVBit [23], is necessary.

D. SDC Results

We also evaluated SDC errors caused by the fault injection in addition to the DUE errors. Tables VII and VIII list the number of SDC errors in the matrix multiplication with E1 and E2 execution options, respectively. We can see the result in Table VII and that in Table VIII are highly correlated, where the error categorization is explained in the following. CUDA error occurrence is different between the tables, but the error count is small.

Fig. 1 illustrates the error categorization. All fault injections are split into three categories: SDC, mask, and CUDA error. SDC includes any inconsistency with the golden output, and mask means that the output is identical with the golden output.

TABLE VIII
SDC COUNTS IN MATRIX MULTIPLICATION UNDER FAULT INJECTION (E2 OPTION)

	Errors			
	total	inner loop	external loop	outside loop
SDC	178,950	152,967	20,454	5,529
mask	1,993	0	1,266	727
CUDA error	0	0	0	0
within-warp SDC (all0)	70,703 (411)	67,207 (368)	2,882 (18)	614 (25)
inter-warp SDC all0 (DUE)	33,389 (20,953)	16,203 (5,188)	12,401 (11,581)	4,785 (4,184)
inter-warp SDC non-all0	74,858	69,557	5,171	130
# of injected faults	180,943	152,967	21,720	6,256

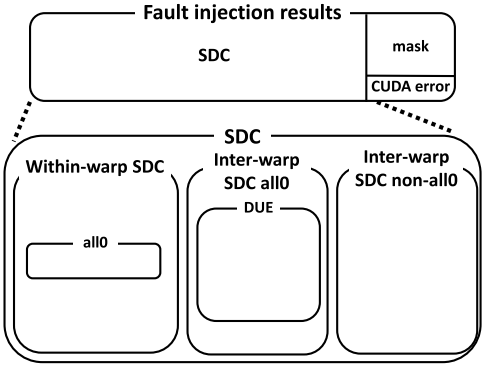

Fig. 1. SDC error categorization.

CUDA error means the program is terminated with a message of CUDA_ERROR_LAUNCH_OUT_OF_RESOURCES. The sum of these three categories is equal to the number of injected faults.

The SDC errors are further split into three categories: within-warp SDC, inter-warp SDC all0, and inter-warp SDC non-all0. Within-warp SDC means that the SDC is found only in the warp to which an error is injected. Some of within-warp SDC errors are SDC with all0, which are listed in Tables VII and VIII. Here, let us explain SDC all0. When a DUE or abnormal kernel exit occurs due to the PTX error injection, the GPU kernel stops and returns all0. Meanwhile, the host code continues to run, checks the computed result with the golden output, and reports SDC occurrence. We call such SDC with all0 as SDC all0. The category of "inter-warp SDC all0" means that all0 errors are found in multiple warps, whereas "within-warp SDC all0" means that SDC with all0 is found within a warp. Some of "inter-warp SDC all0" errors are accompanied by the DUE errors and are categorized as "inter-warp SDC all0 (DUE)." The other remaining errors are called "inter-warp SDC non-all0."

Error injection is performed only in one warp, but SDC often spreads over the entire block (inter-warp SDC). It is due mainly to DUE in the case of the external loop and outside loop. On the other hand, in the inner loop, all the error injection results are observed as SDC, which indicates that the inner loop has extremely high SDC sensitivity. The sensitive parts of the program to DUE and SDC are different,

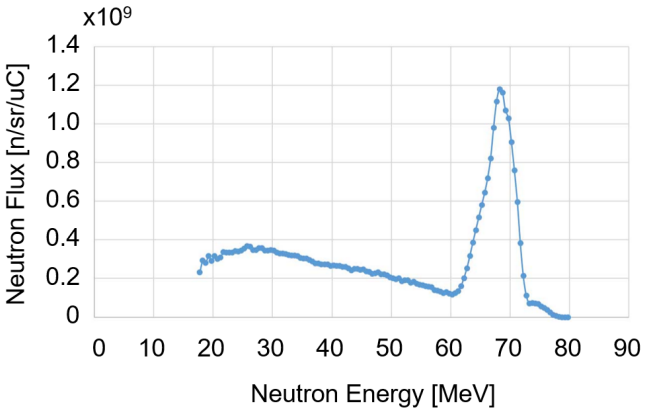

Fig. 2. Neutron energy spectrum at CYRIC.

Fig. 3. GPU card alignment on the beam track.

which could be an interesting observation. Besides, inter-warp SDC is caused by the values in the shared memory, which are shared by the warps. In this case, the entire block may use a faulty value affected by fault injection and written into the shared memory and propagates the contamination.

IV. NEUTRON IRRADIATION EXPERIMENT

A. Setup

We performed a quasi-monoenergetic neutron irradiation experiment at the Cyclotron and Radioisotope Center (CYRIC) at Tohoku University [18]. Fig. 2 shows the energy spectrum of the neutron beam. A 70-MeV proton source produces the neutron beam, and the neutron beam has a flux peak at the energy near 70 MeV.

Fig. 3 depicts the setup of the irradiation experiment. Five P2000 cards and two GTX960 cards are placed on the beam track. The GPU cards are connected to their corresponding host PCs through PCI-express extension cables. The host PCs, on which Linux is running, in the irradiation room are remotely controlled through Ethernet cables. Also, we put

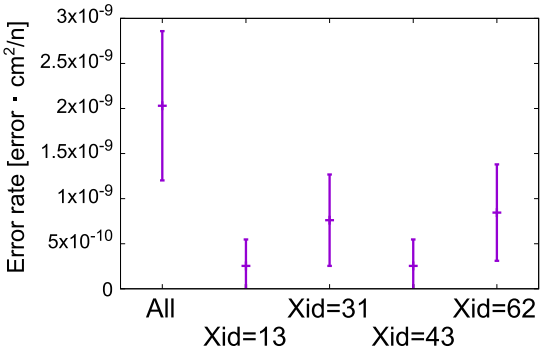

Fig. 4. Measured DUE rates in matrix multiplication program on Q2000. Error bars correspond to 95% confidence interval.

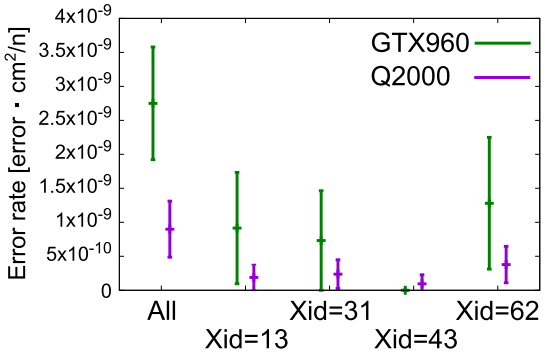

Fig. 5. Measured DUE rates in object detection program on Q2000 and GTX960. Error bars correspond to 95% confidence interval.

remote-control rebooters in the irradiation room to forcibly and selectively host the PCs through Ethernet cables. With this setup, the programs of object detection and matrix multiplications are executed on P2000 cards. On GTX960, only the object detection program is executed, as explained with Table II.

The average flux over the location and time was 3.25×10^5 [n/s/cm^2]. The concrete fluence values for each experiment are 1.18×10^{10} [n/cm^2] for matrix multiplication, 5.46×10^9 [n/cm^2] for object detection (GTX960), and 2.11×10^{10} [n/cm^2] for object detection (P2000), where the irradiation times are 23.81 h for matrix multiplication, 4.67 h for object detection (GTX960), and 18.02 h for object detection (P2000). We check the syslog message to see Xid error information. During the irradiation experiment, when the response of nvidia-sim command, which returns the GPU usage status, is prolonged, the PC is rebooted.

B. Results

Fig. 4 shows the DUE rates of the matrix multiplication program measured in the experiment, where the error bar corresponds to the 95% confidence interval. The total number of DUE errors is 24. We can see that DUEs of Xid 13, 31, and 43, which are observed in the fault injection experiment, are also found in the irradiation experiment. On the other hand, the error rates of Xid 13 and 43 are lower than that of Xid 31, which is different from the fault injection result. The higher rate of Xid 31 might come from the errors in register files since some errors in the register files affect the jump address. However, the error bars are large, and hence a solid conclusion is difficult to draw from the result.

Meanwhile, the DUE of Xid 62 is observed frequently while it was not observed in the fault injection experiment. This observation is consistent with [7], which reports that not all the errors can be replicated by fault injection. Meanwhile, this work suggests the category of such nonreproducible errors. Our result demonstrates that the microcontroller, which is not visible to programmers, contributes to the DUE rate considerably. A further study on the contribution and error mode of such invisible components is necessary.

Fig. 5 shows the DUE rates of the object detection program. The DUE errors observed are 15 for object detection (GTX960) and 19 for object detection (P2000). The DUE rates of GTX960 tend to be higher while some error bars are overlapped. A reason is that Q2000 is manufactured in FinFET technology, and then the error rate is low. Focusing on Q2000, the relative error occurrences of each Xid look similar to those of the matrix multiplication case. The object detection program performs many MAC operations and hence the error pattern could be similar to the matrix multiplication case.

C. Discussion

Errors of Xid 62 were observed only in the radiation experiments. The cause of Xid 62 is either hardware issue, driver issue, or thermal issue, as explained with Table III. On the other hand, we experimentally confirmed that the thermal issue-related error did not occur even when we intentionally stopped the fan. Therefore, it is implausible that a thermal problem will occur. Other Xid errors that originate from hardware issues in our experimental setup are 79 and 80 in Table III. Both are related to the miscommunication between GPU and others. Such Xid errors are not observed in the radiation experiment, and hence the communication between the GPU and others is not disturbed. We, therefore, suspect that most of the Xid 62 errors of internal microcontroller halt are caused by the hardware invisible to programmers inside the GPU chip.

V. Conclusion

This work analyzed the DUEs reported by the GPU driver under fault injection and neutron radiation. The fault injection experiment that reproduced PC error with PTX code manipulation shows that the sensitivity to DUE is different depending on the fault location. The codes for loading arguments, configuring threads and blocks, and writing back to the main memory are highly sensitive. The neutron irradiation test shows that the DUEs found in the fault injection experiment are also observed. The comparison between Q2000 and GTX960 shows that Q2000 has a lower DUE rate, probably thanks to FinFET technology. An important observation is that the DUE categorized as internal microcontroller halt by the GPU driver occurred frequently, which suggests that components invisible to programmers considerably contribute to DUEs. Our future work includes fault injection experiments to object detection using NVBit [23], which can inject faults even into proprietary accelerated libraries.

ACKNOWLEDGMENT

The authors thank Prof. Masatoshi Itoh of Tohoku University for their support in the neutron irradiation experiments at the Cyclotron and Radioisotope Center (CYRIC).

REFERENCES

[1] D. A. G. de Oliveira, L. L. Pilla, T. Santini, and P. Rech, "Evaluation and mitigation of radiation-induced soft errors in graphics processing units," *IEEE Trans. Comput.*, vol. 65, no. 3, pp. 791–804, Apr. 2016.

[2] F. F. dos Santos *et al.*, "Analyzing and increasing the reliability of convolutional neural networks on GPUs," *IEEE Trans. Rel.*, vol. 68, no. 2, pp. 663–677, Jun. 2019.

[3] A. Lotfi *et al.*, "Resiliency of automotive object detection networks on GPU architectures," in *Proc. IEEE Int. Test Conf. (ITC)*, Nov. 2019, pp. 191–199.

[4] C. Lunardi, F. Previlon, D. Kaeli, and P. Rech, "On the efficacy of ECC and the benefits of FinFET transistor layout for GPU reliability," *IEEE Trans. Nucl. Sci.*, vol. 65, no. 8, pp. 1843–1850, Aug. 2018.

[5] P. M. Basso, F. F. D. Santos, and P. Rech, "Impact of tensor cores and mixed precision on the reliability of matrix multiplication in GPUs," *IEEE Trans. Nucl. Sci.*, vol. 67, no. 7, pp. 1560–1565, Jul. 2020.

[6] 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 *Proc. Int. Symp. Perform. Anal. Syst. Softw. (ISPASS)*, 2017, pp. 249–258.

[7] C. Lunardi, H. Quinn, L. Monroe, D. Oliveira, P. Navaux, and P. Rech, "Experimental and analytical analysis of sorting algorithms error criticality for HPC and large servers applications," *IEEE Trans. Nucl. Sci.*, vol. 64, no. 8, pp. 2169–2178, Aug. 2017.

[8] F. G. Previlon, C. Kalra, D. R. Kaeli, and P. Rech, "Evaluating the impact of execution parameters on program vulnerability in GPU applications," in *Proc. Design, Autom. Test Eur. Conf. Exhib. (DATE)*, Mar. 2018, pp. 809–814.

[9] R. L. Davidson and C. P. Bridges, "Error resilient GPU accelerated image processing for space applications," *IEEE Trans. Parallel Distrib. Syst.*, vol. 29, no. 9, pp. 1990–2003, Sep. 2018.

[10] Y. Ibrahim *et al.*, "Soft error resilience of deep residual networks for object recognition," *IEEE Access*, vol. 8, pp. 19490–19503, 2020.

[11] F. G. Previlon, C. Kalra, D. Tiwari, and D. R. Kaeli, "Characterizing and exploiting soft error vulnerability phase behavior in GPU applications," *IEEE Trans. Dependable Secure Comput.*, early access, Apr. 28, 2020, doi: 10.1109/TDSC.2020.2991136.

[12] K. Ito *et al.*, "Characterizing neutron-induced SDC rate of matrix multiplication in Tesla P4 GPU," in *Proc. Eur. Conf. Radiat. Effects Compon. Syst. (RADECS)*, 2019.

[13] NVIDIA. (2019). *Parallel Thread Execution ISA Version 6.5*. [Online]. Available: https://docs.nvidia.com/cuda/parallel-thread-execution/index.html

[14] J. Redmon and A. Farhadi, "YOLOv3: An incremental improvement," 2018, *arXiv:1804.02767*. [Online]. Available: http://arxiv.org/abs/1804.02767

[15] T.-Y. Lin *et al.*, "Microsoft COCO: Common objects in context," 2014, *arXiv:1405.0312*. [Online]. Available: http://arxiv.org/abs/1405.0312

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

[17] D. Tiwari *et al.*, "Understanding GPU errors on large-scale HPC systems and the implications for system design and operation," in *Proc. Int. Symp. High Perform. Comput. Archit. (HPCA)*, 2015, pp. 331–342.

[18] Y. Sakemi, M. Itoh, and T. Wakui, "High intensity fast neutron beam facility at CYRIC," *Int. Atomic Energy Agency*, vol. 46, no. 9, pp. 229–233, 2014.

[19] J. Tan, N. Goswami, T. Li, and X. Fu, "Analyzing soft-error vulnerability on GPGPU microarchitecture," in *Proc. Int. Symp. Workload Characterization*, 2011, pp. 226–235.

[20] G. Li, K. Pattabiraman, C.-Y. Cher, and P. Bose, "Understanding error propagation in GPGPU applications," in *Proc. Int. Conf. for High Perform. Comput., Netw., Storage Anal. (SC)*, Nov. 2016, pp. 240–251.

[21] B. Fang, K. Pattabiraman, M. Ripeanu, and S. Gurumurthi, "GPU-Qin: A methodology for evaluating the error resilience of GPGPU applications," in *Proc. IEEE Int. Symp. Perform. Anal. Syst. Softw. (ISPASS)*, Mar. 2014, pp. 221–230.

[22] S. Tselonis and D. Gizopoulos, "GUFI: A framework for GPUs reliability assessment," in *Proc. IEEE Int. Symp. Perform. Anal. Syst. Softw. (ISPASS)*, Apr. 2016, pp. 90–100.

[23] O. Villa, M. Stephenson, D. Nellans, and S. W. Keckler, "NVBit: A dynamic binary instrumentation framework for NVIDIA GPUs," in *Proc. Int. Symp. Microarchitecture*, 2019, pp. 372–383.