IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

IEEE RELIABILITY SOCIETY SECTION

Received 22 May 2025, accepted 26 June 2025, date of publication 1 July 2025, date of current version 9 July 2025.

Digital Object Identifier 10.1109/ACCESS.2025.3584759

RESEARCH ARTICLE

A Hardware-Aware Failure-Detection Method for GPU Control-Logic

HIROAKI ITSUJI[1], (Member, IEEE), TAKUMI UEZONO[1], (Member, IEEE), TADANOBU TOBA[1], KOJIRO ITO[2], AND MASANORI HASHIMOTO[3], (Senior Member, IEEE)

[1]Sustainability Innovation Research and Development, Hitachi, Ltd., Yokohama 244-0817, Japan
[2]Department of Information System Engineering, Osaka University, Suita 565-0871, Japan
[3]Graduate School of Informatics, Kyoto University, Kyoto 606-8501, Japan

Corresponding author: Hiroaki Itsuji (hiroaki.itsuji.pc@hitachi.com)

ABSTRACT Graphics processing units (GPUs) are used for diverse applications and play a major role even in safety-critical applications. Although performance is usually the primary focus of GPUs, their reliability has become a major concern. One of the undesirable failures in GPUs is silent data corruption (SDC), which causes unexpected outputs without any warning. Various failure detection methods have been proposed for SDCs caused by faults in data units such as registers. However, effective methods for detecting SDCs resulting from faults in control logic, such as scheduling units, have not yet been established. This paper assumes three types of control-logic failures for a general GPU architecture and proposes efficient failure detection methods for each type. For instance, the proposed method efficiently detects GPU-specific control-logic failures caused by program counter faults with a detection rate of 99.5% and can be implemented with a runtime overhead of 5.3% and a memory-resource overhead of 4.2% for a matrix multiplication application. These methods are applicable to a wide range of applications and are expected to enhance system resiliency.

INDEX TERMS GPU, silent data corruption, reliability, fault injection, failure detection, diagnosis.

I. INTRODUCTION

Graphics processing units (GPUs) originally developed for image processing are currently used for diverse and general-purpose computations. They are currently used in fields that require safety and reliability such as the automated driving field [1], [2]. Ensuring the reliability of GPUs is becoming increasingly important even though what usually draws attention is the performance of GPUs in boosting application throughputs. GPUs embedded in safety-critical systems may malfunction due to temporal failures possibly due to environmental noise (*e.g.*, neutron-induced data upsets [3], [4]) and permanent failures possibly due to aging (*e.g.*, stuck-at faults [5], burnout). As for temporal failures, GPUs are 1–2 orders of magnitude more vulnerable than central processing units (CPUs) in terms of tasks per failure [6]. Detecting such temporal failures while systems are running is important for preventing critical accidents and huge losses.

The associate editor coordinating the review of this manuscript and approving it for publication was Liudong Xing.

Temporal failures lead to one of four outcomes: (1) no effect on program outputs, (2) detectable corrected errors (*e.g.*, error correction codes), (3) detectable uncorrected errors (DUEs), or (4) undetectable incorrect results, *i.e.*, silent data corruption (SDC). SDCs are undesirable since they may lead to critical accidents and huge losses without any signs such as application hangs and crashes. SDCs originate from faults in data units (such as instruction caches and processing elements) or in control logic that governs the flow of parallel computations. Most research has addressed the detection of failures in data units [7], [8], [9], assuming that the size of control logic is significantly smaller than that of data units and that failures in the control logic are negligible [10].

In this regard, recent analyses of GPU code have suggested that control logic is relatively more vulnerable than data units in terms of the probability of SDCs [11], [12]. Furthermore, it was suggested that a control-logic failure could propagate and affect dozens of output data values [13], [14], [15]. The inability to detect control-logic failures even once may lead to critical accidents and huge losses. Hence, there is a need

to develop suitable countermeasures against these failures. However, as far as we have investigated, little research has addressed countermeasures for GPU control-logic failures.

With respect to countermeasures against temporal failures on the hardware side, modular redundancy techniques [16] detect control-logic failures by comparing program outputs from multiple modules. These techniques require a few times larger amount of original hardware resources and increase the total failure probability and number of system halts. On the software side, control-logic failures can be detected with software duplication or insertion of failure detection codes into original program codes. These techniques significantly degrade application throughputs. For example, software duplication incurs a runtime overhead of 69% [17]. The widely known signature-based techniques, *i.e.*, CFCSS [18] and YACCA [19], incur runtime overheads of 60% and 179%, respectively, for matrix multiplication applications [20]. Such degradation is unacceptable for applications that require high throughput and allow no modular redundancy. Therefore, it is necessary to develop a method for detecting control-logic failures that is efficient in terms of throughput and hardware usage, while maintaining a high failure detection rate.

In light of the above background, our preliminary work proposed a method for efficiently detecting control-logic failures caused by faults in the program counter (PC) [21]. This preliminary method relied on partially-redundant computations assigned to selected cores and achieved a runtime overhead of 28% and a memory-resource overhead of 16% for a matrix multiplication application. However, it required extensive modifications to the original application code because the application computations and partially-redundant computations for detecting control-logic failures were intermixed during execution. This intermixing not only made the code structure significantly more complex but also resulted in inefficient hardware resource utilization, leaving considerable room for improvement. Furthermore, this method alone was insufficient to comprehensively detect various types of control-logic failures.

To address these limitations, the key contributions of this paper are as follows:

1) Building on the prior work [21], we propose a novel method to efficiently detect GPU control-logic failures by enabling parallel and separated execution of application computations, partially-redundant computations, and diagnostic computations. By appropriately tuning GPU-specific thread and block indices, our method significantly reduces the runtime and memory overhead compared to the previous intermixed execution approach.
2) We conduct a detailed evaluation of the impact of injected faults in GPU control logic and confirm that they cause SDCs without triggering any alarms.
3) We demonstrate that the proposed method can detect GPU-specific control-logic failures (*e.g.*, due to unexpected changes in the PC) with 5.3% runtime overhead and 4.2% memory overhead, achieving a failure detection rate of 99.5%.
4) Building on existing techniques, we also explore methods for comprehensively detecting GPU control-logic failures and identify the associated detection challenges.
5) As a result, we provide insights into enhancing GPU reliability through comprehensive detection of control-logic failures, suggesting that the proposed methods can be broadly applied to improve the resilience of various GPU applications.

The rest of this paper is organized as follows. Section II presents related work. Section III introduces a general GPU architecture and code, and defines control-logic failures. Section IV describes the fault-injection results based on the defined control-logic failures. Section V explains the proposed methods for detecting control-logic failures. Section VI presents the evaluation results in terms of runtime overhead, memory-resource overhead, and failure detection rate. Section VII concludes the paper.

II. RELATED WORK

Various fault-injection frameworks have been developed for reliability assessments, and the abstraction level of fault injection was different depending on the framework. For example, faults have been injected into high-level programming codes (*e.g.*, LLVM IR, CUDA) [10], SASS codes [14], [15], [22], [23], [24], or PTX codes [11], [25], [26]. Although fault injections in high-level programming code may be useful for analyzing vulnerable code and its execution duration in applications, they cannot be used for simulating warp-level failures because the instructions are not specified in the code, and faults in PCs leading to warp-level failures cannot be reproduced. SASS code can most accurately reproduce warp-level failures at the cycle level. However, SASS code depends on the GPU architecture, and it is time-consuming to derive generalized results by injecting warp-level faults into the SASS code of various GPU architectures. In addition, commonly available tools cannot inject faults into PTX code to reproduce warp-level failures. In our previous work, we developed a warp-level fault-injection framework utilizing PTX code, which does not depend on the GPU architecture [13], [21]. In this paper, we reused the same framework to simulate control-logic failures.

Various methods for the efficient detection of GPU faults have been developed on the basis of selective hardening. Selective hardening increases the redundancy of selected data (*e.g.*, registers [7], [9], kernels [27], [28]). For example, HAUBERK, which was developed for efficient SDC detection [7], relies on partially-redundant registers and varies the level of redundancy inside and outside loops. Signature-based methods [18], [19], which rely on the partial redundancy of instructions, insert instructions into original-application instructions for checking the control flow by updating the signature value and comparing the signature with the expected value at each checkpoint during application

execution. A method utilizing Self-Test Libraries (STLs) was also developed to efficiently perform in-field testing of GPUs by exercising the control logic [29]. Departing from the selective hardening approach, a thread-index-based method was introduced to detect control-logic failures by verifying the correctness of thread indices, which allows for the identification of control-logic failures such as missing or duplicated threads [30].

Compared to existing techniques, the proposed method for detecting control-logic failures—including those caused by program counter faults—achieves lower runtime and memory overheads while maintaining comparable or superior failure detection rates, as demonstrated by the experimental evaluations presented in this study. Furthermore, we extend the thread-index-based method proposed in the literature [30] by incorporating block-index verification, and quantitatively evaluate its associated overheads and failure detection rate.

The main differences from our preliminary works [13], [21] are as follows:

- This work proposes a novel method for efficiently detecting GPU control-logic failures by enabling the parallel and independent execution of application computations, partially-redundant computations, and diagnostic computations. Unlike the previous method [21], in which the application computations and partially-redundant computations were intermixed, the proposed approach clearly separates these computations. This separation significantly reduces runtime and memory overhead while maintaining a high failure detection rate.
- Using the PTX-based fault-injection environment established in prior works [13], [21], we assess the impact of diverse control-logic failures—including not only warp-level failures but also block-level and thread-level failures.

III. GPU CONTROL-LOGIC FAILURE

This section overviews a general GPU architecture and code. Next, modes of GPU control-logic failures for the general GPU architecture and code are described.

A. GPU ARCHITECTURE AND CODE

Fig. 1(a) shows a conceptual drawing of the general CPU-GPU framework for NVIDIA GPUs utilizing single-instruction, multiple-thread (SIMT) [31], [32]. The CPU launches parallel-computing kernels while storing data in global memory as input data for parallel computations. The GPU operates in accordance with instructions from the kernel invoked by the CPU. Subsequently, the instructions are distributed from the instruction cache and block scheduler to the streaming multiprocessors (SMs) that have the warp schedulers for warp execution. A block, also referred to as a cooperative-thread array, is a group of warps. A warp is a group of 32 threads that are simultaneously executed. Each thread in one warp is allocated to one core, and each core in the warp executes the same instructions for different data. Execution results are stored in a register file, after which they are transferred to the L2 cache and then to global memory. Optionally, L1 cache and shared memory can be used for more efficient data transfer.

Further details of the data flow in the control logic [21], [31] are illustrated in Fig. 1(b). A block scheduler distributes block information (*e.g.*, block index) to a warp scheduler, and the warp scheduler then distributes warp information to each warp. Each warp has a SIMT stack containing information on the PC, reconvergence program counter (RPC), mask, and warp state. The SIMT stack is used to handle the execution of branch divergence. The PC is used for directing instructions stored in the instruction buffer while the RPC is used for directing reconvergence points after divergence. The mask is used for executing only active threads in each warp, and each mask is represented in 32 bits, corresponding to the number of threads in a warp. The warp state is used for representing one of four states: ready, active, waiting, or finished. The SIMT stack of each warp is updated after each instruction issue of the warp.

On the software side, code for executing kernels is composed of CPU code (host-side code) and GPU code (device-side code). The CPU code specifies information such as the kernel launch information (*e.g.*, number of blocks and threads assigned for each kernel) and memory allocation information. The GPU code specifies kernel contents. Fig. 2 shows an example of GPU code for NVIDIA CUDA. Even though all the warps execute the same GPU code, all of them perform calculations for different input data and output the calculated data to different memory addresses, utilizing the difference in the block index (*e.g., blockIdx.x*) and thread index (*threadIdx.x*). The index value is automatically determined by the number of threads and blocks assigned for each kernel. For example, when the total number of threads in one block is 64 and the threads are one-dimensional, the thread index (*e.g., threadIdx.x*) ranges from 0 to 63. In this case, the first warp corresponds to the thread index where *threadIdx.x* ranges from 0 to 31, and the second warp corresponds to the thread index where *threadIdx.x* ranges from 32 to 63. The maximum dimension of the block index and thread index is three. As such, parallel computations are performed on the basis of the GPU architecture and code.

B. MODES OF GPU CONTROL-LOGIC FAILURES

Environmental noise, such as that caused by neutrons, can induce multiple-bit flips—known as soft errors—in memory elements [3], [4], which are assumed to be the fault mode in this study. These soft errors can lead to unexpected values in data stored within the GPU control logic, potentially resulting in GPU control-logic failures. On the basis of the GPU hierarchy, control-logic failures leading to SDCs are classified into three types as shown in Table 1.

Block-level failures possibly originate from faults in the block index, the information of which is transferred from the block scheduler and stored in per-thread special registers.

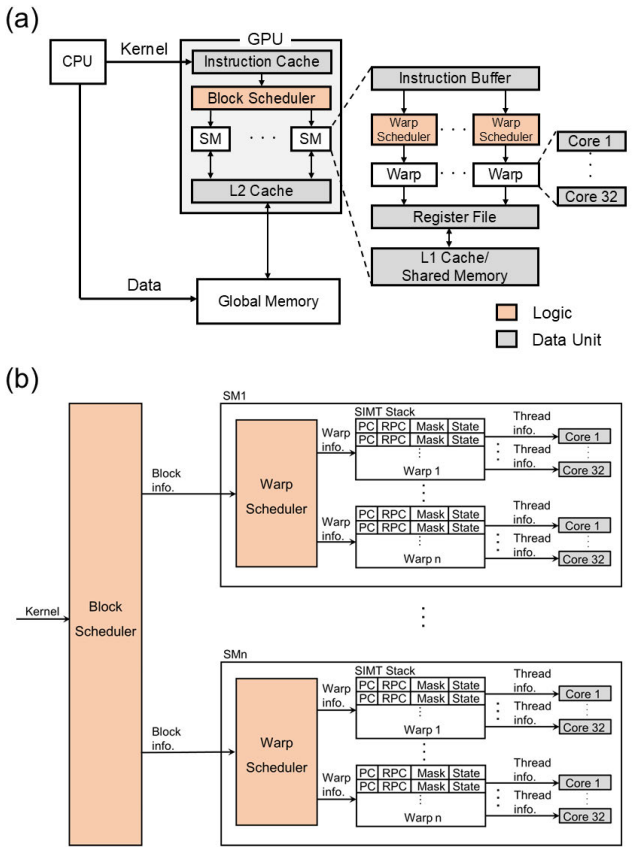

FIGURE 1. (a) General CPU-GPU framework for parallel computing, (b) data flow in GPU control logic.

```
__global__ void vectorAdd (float *A, float *B, float *C)
{
  int i = 64 * blockIdx.x + threadIdx.x ;
  C[i] = A[i] + B[i] ;
}
```

FIGURE 2. Example of GPU code for adding vectors.

Since a block normally includes more than 32 threads, a block-level failure directly affects the calculation of $>$ 32 threads, and the block is not executed or is executed abnormally. Warp-level failures possibly originate from faults in the PC, RPC, or warp state. Since a warp is a group of 32 threads, a warp-level failure directly affects the calculation of 32 threads, and the warp is not executed or is executed abnormally. Thread-level failures possibly originate from faults in the thread block or mask. These failures directly affect the calculation of $\leq$ 32 threads, and some threads in a warp are not executed or are executed abnormally.

Even though faults in hidden parts in the control logic may also cause SDCs [33], the resultant SDCs should be mapped to either a block-level failure ($>$ 32 threads), a warp-level failure (32 threads) or a thread-level failure ($\leq$ 32 threads) when the GPU hierarchy is considered. The insertion of a small piece of code to simulate a warp-level fault reproduced some system errors detected by a GPU driver (XID errors [34], [35]) during neutron irradiation tests, proving the occurrence of warp-level failures in a real environment [13]. On the other hand, thread-level and block-level failures have not been observed, but are supposed to occur due to the GPU structure in principle.

IV. FAULT-INJECTION EXPERIMENT TO SIMULATE GPU CONTROL-LOGIC FAILURE

This section demonstrates that the control-logic failures trigger SDCs for various applications, which remain undetected by the GPU driver. With fault injection, we used parallel thread execution (PTX) code, a pseudo-assembly language in the NVIDIA CUDA environment that is independent of the GPU architecture.

A. TARGET DEVICE AND APPLICATION

As a test vehicle for the GPU, an NVIDIA Quadro P2000 was used. Note that the fault-injection results did not depend on a specific GPU architecture since PTX codes were used. The device specifications are shown in Table 2. The number of cores per SM was 128, and the maximum number of threads in one block was 1024. Each SM had 96 kB of shared memory, which was also the maximum amount allocatable to a single block. The host CPU used for controlling the GPU was a Xeon CPU E3-1225 v5 @ 3.30GHz.

B. FAULT-INJECTION SETUP

1) WARP-LEVEL FAULT INJECTION SETUP

Fig. 3 shows the flow for warp-level fault injection with a developed fault-injection framework [13], [21]. First, original GPU code was divided into host-side code (C++ code) and device-side code (CUDA code). The host-side code was changed so that only a single block that included a warp with a faulty jump was executed. The fault was injected into only one warp because all the warps of a kernel shared the same PTX code, differing only in thread and block indices. The device-side code was compiled to generate normal PTX code. Subsequently, a set of abnormal PTX code, each including a different small piece of code, was generated on the basis of normal PTX code. The small piece of code was inserted to simulate warp-level faults at the virtual PC, *i.e.*, by forcing a jump from one arbitrary point (start PC) to another arbitrary point (end PC) once per application run. We generated abnormal PTX codes assuming all possible start/end PCs. Subsequently, the normal and abnormal PTX code, which was then compiled with a no-optimization option (-O0) into binary code, was loaded from the host-side code and executed one by one. The no-optimization option was necessary for the added PTX code to be executed as the programmer specified without disturbance due to compiler optimization [9]. For each execution of the abnormal PTX code, XID errors were checked to identify the warp-level failures leading to DUEs.

TABLE 1. Definition of GPU control-logic failure.

Failure type	Failure cause	# of directly affected threads
Block-level failure	Fault in block information (*e.g.,* block index)	$>$ 32 threads
Warp-level failure	Fault in warp information (*e.g.,* PC, RPC, warp state)	32 threads
Thread-level failure	Fault in thread information (*e.g.,* thread index, mask)	$<$ 32 threads

We compared the execution results from the normal and abnormal PTX code to identify the warp-level failures leading to SDCs.

Algorithm 1 is an example of a small piece of code inserted for warp-level fault injections (*e.g.*, unexpected PC alternation due to neutron-induced multiple-bit upsets) in a matrix multiplication application. In this matrix multiplication application, the number of threads in a block was 1024 (32 warps). There were 32 threads in both the x and y dimensions. Thus, *threadIdx.x* and *threadIdx.y* ranged from 0 to 31. Here, only the warp (*threadIdx.x* $= 0$ to 31 and *threadIdx.y* $= 0$) satisfied the jump condition. The code inserted at the jump start PC enabled the warp that satisfied *threadIdx.y* $= 0$ to jump to L1. Subsequently, at the jump end PC, where the characters "L1:" were added to the original instruction, only the warp satisfying *threadIdx.y* $= 0$ executed instructions from the add instruction (add.u32 %r10, %r10, 1) after the jump. The faulty jump was enabled only once per application run. Note that when the start PC was inside the loop, the jump was enabled only for the first time in the loop since the number of loops executed before the jump should not affect the classification of outcomes into masked, SDCs, or DUEs. The inserted code varied depending on the application, due to the difference in the three-dimensional configuration of threads and blocks.

2) BLOCK-LEVEL AND THREAD-LEVEL FAULT INJECTION SETUP

The purpose of fault injection is to demonstrate that SDCs can occur due to block-level and thread-level faults, which may be caused by neutrons. For block-level failures, faults that corrupted retained data were injected into the block index, while for thread-level failures, the same type of faults were injected into the thread index. Since direct modification of the index value was not possible, the register value containing the index was modified immediately after the value was transferred from the special register.

Algorithm 2 is an example of a small piece of code inserted for block-level fault injection in a matrix multiplication application. Faults were injected in the block where *blockIdx.x* $= 0$ and *blockIdx.y* $= 0$. Here, *blockIdx.x* was replaced with each possible value from 1 to (1 + the maximum value of *blockIdx.x*). In the same way, a similar small piece of code was inserted for the block-level fault injection in other applications, such as the separable convolution application and histogram application.

Algorithm 3 is an example of a small piece of code inserted for thread-level fault injection in a matrix multiplication application. Faults were injected in the thread with *blockIdx.x* $= 0$, *blockIdx.y* $= 0$, *threadIdx.x* $= 0$, and *threadIdx.y* $x= 0$. Here, *threadIdx.x* was replaced with each possible value from 1 to (1 + the maximum value of *threadIdx.x*). A similar small piece of code was also inserted for other applications.

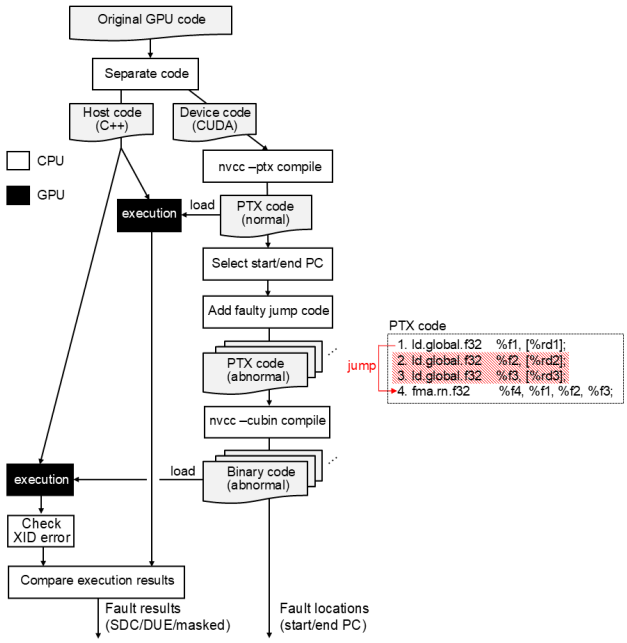

FIGURE 3. Flow for warp-level fault injection using developed fault-injection framework.

Algorithm 1 Code Inserted for Warp-Level Fault Injections in Matrix Multiplication Application

Code inserted at jump start point

```
// register %r0 stores jump flag (initial value: 0)
// register %r1 stores threadIdx.y
setp.ne.s32 %p0, %r0, 0;
// check whether %r0 == 0 to identify the first jump
add.u32 %r0, %r0, 1;
// modify %r0 so that an abnormal jump occurs only the first time
@%p0 bra L0;
// jump to L0 when %r0 != 0 (return to normal operation)
setp.eq.s32 %p1, %r1, 0;
// check whether a warp jumps abnormally
// (assuming only one warp satisfies threadIdx.y == 0)
@%p1 bra L1;
// jump abnormally to L1 when threadIdx.y == 0
L0:
```

Code inserted at jump end point

```
L1: add.u32 %r10, %r10, 1;
// label L1 is located at the beginning of instruction
```

Algorithm 2 Code Inserted for Block-Level Fault Injections

```
// register %r0 stores blockIdx.x
// register %r1 stores blockIdx.y
setp.ne.s32 %p0, %r0, 0;
// check whether blockIdx.x != 0
setp.ne.s32 %p1, %r1, 0;
// check whether blockIdx.y != 0
@%p0 bra L0;
// jump to L0 when %r0 != 0 (return to normal operation)
@%p1 bra L1;
// jump to L1 when %r1 != 0 (return to normal operation)
mov.u32 %r0, 1;
// change blockIdx.x from 0 to 1 (block-level fault)
L1:
L0:
```

Algorithm 3 Code Inserted for Thread-Level Fault Injections

```
// register %r0 stores blockIdx.x
// register %r1 stores blockIdx.y
// register %r2 stores threadIdx.x
// register %r3 stores threadIdx.y
setp.ne.s32 %p0, %r0, 0;
// check whether blockIdx.x != 0
setp.ne.s32 %p1, %r1, 0;
// check whether blockIdx.y != 0
setp.ne.s32 %p2, %r2, 0;
// check whether threadIdx.x != 0
setp.ne.s32 %p3, %r3, 0;
// check whether threadIdx.y != 0
@%p0 bra L0;
// jump to L0 when %r0 != 0 (return to normal operation)
@%p1 bra L1;
// jump to L1 when %r1 != 0 (return to normal operation)
@%p2 bra L2;
// jump to L2 when %r2 != 0 (return to normal operation)
@%p3 bra L3;
// jump to L3 when %r3 != 0 (return to normal operation)
mov.u32 %r2, 1;
// change threadIdx.x from 0 to 1 (thread-level fault)
L3:
L2:
L1:
L0:
```

TABLE 2. Device information on NVIDIA Quadro P2000.

Architecture	Pascal
SM	8
Cores per SM	128
Total Register File [kB]	2,048
Total Shared Memory [kB]	768
Total L1 Cache [kB]	960
Total L2 Cache [kB]	1,280
GDDR5 Memory [GB]	5

C. FAULT-INJECTION RESULTS

1) WARP-LEVEL FAULT INJECTION RESULTS

Table 3 summarizes the fault-injection results for the matrix multiplication, separable convolution, and histogram applications. This table demonstrates that warp-level faults in the PC caused SDCs across various applications, and the SDC ratio depended on both the application and the start PC (the PC value before the occurrence of warp-level faults). One loop was included in the PTX instructions for the matrix multiplication and histogram applications, whereas the separable convolution applications did not include any. Since a warp consists of 32 threads, when an SDC was observed, all 32 threads exhibited SDCs. The SDC ratio was relatively high when the start PC was inside the loop. For example, the SDC ratio was 65.1% for the matrix multiplication application and 51.8% for the histogram application.

To understand the differences in the SDC ratios across applications, the number of PTX instructions for each application was analyzed, as summarized in Table 4. The PTX instructions were categorized into four groups: data transfer (*e.g.*, load and store operations), data conversion (*e.g.*, data shifting and type conversion), basic computations (*e.g.*, AND, OR, addition, and fused multiply-add), and others (*e.g.*, synchronization). This table shows the number of executed instructions for each of the four groups. When the ratio of data conversion was low, the SDC ratio was relatively high. When the ratio of data conversion was high, the DUE ratio was relatively high. For example, the ratio of data conversion was 0.0% (lowest) when the start PC was inside the loop for the matrix multiplication application where the SDC ratio was 65.1% (highest). The ratio of data conversion was 28.6% (highest) when the start PC was outside the loop for the matrix multiplication application, where the DUE ratio was 69.4% (highest). With respect to data conversion, when data conversion fails, or the data size or type to be processed is incorrect, GPUs can generally detect such DUE errors as XID errors as part of their error-handling mechanisms. These observations indicate that the ratio of data conversion in PTX instructions is one of the important factors determining the SDC ratio.

2) BLOCK-LEVEL AND THREAD-LEVEL FAULT INJECTION RESULTS

The fault injections caused either SDCs or DUE, *i.e.*, XID errors. In the case of SDCs, it was found that the block-level or thread-level failures caused SDCs in dozens of output values. The failures led to unexpected global memory accesses because the index value was used for determining the global memory addresses where computation results were stored. For the block-level failure, the number of SDCs was equal to the number of outputs handled by one block. The maximum number of incorrect output values due to the block-index failure was 1024 for the matrix multiplication application, 512 for the separable convolution application (row), 1024 for the separable convolution application (column), and 64 for the histogram application.

For the thread-level failure, the number of SDCs was equal to the number of outputs handled by one thread, and it depended on whether a single thread used shared

TABLE 3. Warp-level fault-injection results.

Application	# of loop	start PC	SDC	DUE	Masked	Total
Matrix multiplication	1	Inside loop	9,888 (65.1%)	454 (3.0%)	4,852 (31.9%)	15,194
		Outside loop	437 (8.5%)	3,546 (69.4%)	1,129 (22.1%)	5,112
Separable convolution (row)	0	Outside loop	73,288 (33.4%)	36,386 (16.6%)	109,818 (50.0%)	219,492
Separable convolution (column)	0	Outside loop	43,202 (23.0%)	50,669 (27.0%)	94,051 (50.0%)	187,922
Histogram	1	Inside loop	23,950 (51.8%)	21,116 (45.7%)	1,186 (2.5%)	46,252
		Outside loop	43,783 (47.0%)	12,059 (12.9%)	37,408 (40.1%)	93,250

TABLE 4. Execution count of PTX instructions by category.

Application	Location	Data transfer	Data conversion	Basic calculation	Other	Total
Matrix multiplication	Inside loop	68 (63.6%)	0 (0.0%)	35 (32.7%)	4 (3.7%)	107
	Outside loop	11 (31.4%)	10 (28.6%)	13 (37.2%)	1 (2.8%)	35
Separable convolution (row)	Outside loop	306 (65.4%)	8 (1.7%)	147 (31.4%)	7 (1.5%)	468
Separable convolution (column)	Outside loop	252 (58.2%)	12 (2.8%)	163 (37.6%)	6 (1.4%)	433
Histogram	Inside loop	36 (29.0%)	17 (13.7%)	69 (55.7%)	2 (1.6%)	124
	Outside loop	94 (37.8%)	14 (5.6%)	134 (53.8%)	7 (2.8%)	249

memory. The maximum number of incorrect output values due to the thread-level failure was 63 for the matrix multiplication application, 120 for the separable convolution application (row), 64 for the separable convolution application (column), and 64 for the histogram application. In the case of XID errors, the block-level or thread-level failures caused illegal accesses to unallocated global memory addresses.

V. PROPOSED DETECTION METHOD FOR GPU CONTROL-LOGIC

This section describes the method for detecting warp-level failures that cause SDCs for all 32 threads in the warp, and describes the methods for detecting block-level and thread-level failures that cause SDCs in dozens of output values. All these methods utilize a large number of diagnostic threads; however, due to their lightweight nature, the diagnostic overhead is expected to be low.

A. WARP-LEVEL FAILURE DETECTION

When a warp-level failure occurs, 32 threads in a warp fail. Therefore, it is not necessary to duplicate every thread in each warp for failure detection. Unlike conventional duplication methods, the proposed method maintains a low redundancy ratio of 6.3% (2/32).

As illustrated in Fig. 4(a), from a hardware perspective, partial redundancy is applied to two out of 32 cores, and the diagnosis is performed using a single SM. Depending on the GPU's scheduling policy, the partial redundancy and diagnosis kernels may be executed on different SMs. From a software perspective, as shown in Fig. 4(b), partial redundancy is applied to two out of 32 threads. The corresponding diagnosis is performed by a single block, ensuring lightweight and efficient execution. The diagnosis is performed by comparing the results of the application computations and the partially-redundant computations. It should be noted that, for the detection of a warp-level failure, two threads in each warp are duplicated to determine whether a detected failure stems from a warp-level failure or from data-unit failures, under the assumption that the simultaneous failure of two threads within a warp due to data-unit faults is highly unlikely.

The diagnostic kernel is composed of a single block with the maximum number of threads (*e.g.*, 1024 for NVIDIA Quadro P2000), and therefore only one SM is occupied by the kernel during concurrent execution. Each diagnostic thread simultaneously compares the computation results of two threads in a warp of the application kernel with those of the corresponding two threads in the partial redundancy kernel. To indicate that each warp in the application kernel is ready to be diagnosed, a conditional statement for the flag value is inserted at the end of the application code so that only one thread in each warp writes a non-zero flag value in the global memory (*e.g.*, "if $threadIdx.x == 31$, flag value = 1" when the equation $threadIdx.x = 31$ is satisfied by only one thread in each warp).

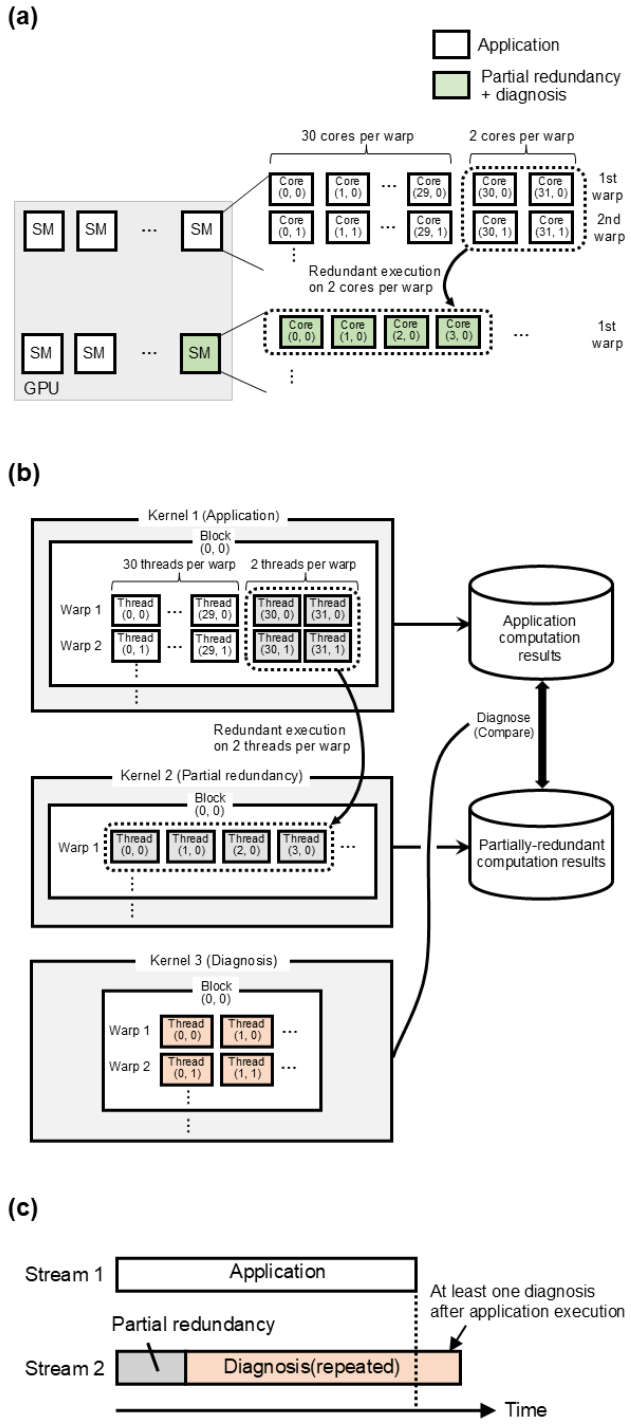

FIGURE 4. Proposed concurrent failure-detection methods for warp-level failures, illustrated from three perspectives:(a) hardware-level implementation, where diagnosis is performed using one SM by executing the same computations with 2 out of 32 cores per warp and comparing the results;(b) software-level implementation, where the partial redundancy kernel executes the same computations for 2 out of 32 application threads per warp, and the diagnostic kernel compares their outputs;(c) execution timing flow, where the partially-redundant and diagnostic computations are performed in parallel with the main application.

As shown in Fig. 4(c), the proposed warp-level failure detection method relies on concurrent execution with the application kernel. Streams are sequences of parallel operations (e.g., kernel executions and memory transactions), and multiple streams can be created via commands in the host code. The partial redundancy kernel, launched concurrently with the application kernel, performs the same computations as the application kernel for two designated threads in each warp. The diagnostic kernel in stream 2 repeatedly compares the outputs of corresponding threads between the application and partial redundancy kernels to diagnose each warp. When a mismatch is found between the two threads in a warp and their redundant counterparts, a warp-level failure is detected. When a flag (*i.e.*, a readiness signal) is set to a non-zero value by the application kernel, the diagnostic kernel performs diagnosis. In contrast, after the application kernel completes, it performs a one-time check for all target addresses, regardless of the flag value. Note that the application kernel is executed separately from the partial redundancy kernel after implementation. This separation contributes to the reduction of both runtime and memory overhead, compared to the prior intermixed execution approach [21].

Fig. 5 illustrates the index transformation required for the partial redundancy kernel. In this kernel, the block and thread indices are automatically determined by the three-dimensional configuration of blocks and threads specified in the host code. To align the computation targets with those of the original application threads, these indices must be transformed into desired values. Transformations of each index are represented by mathematical expressions for the original and target indices. The code for the index transformation is inserted at the beginning of the partially-redundant computation code, which is based on the original application code. Since the block index and thread index are stored in special registers that cannot be directly modified, the values in the special registers are copied to general-purpose registers and then modified. After the transformation, the modified register values are used for the rest of the calculations.

As a limitation, the proposed method cannot be implemented if computation results of an original thread depend on other threads. For example, in histogram applications, all original threads need to be duplicated for duplicated threads to obtain the same computation results by original threads. Also note that for original applications that use a shared memory, all the data required for partially-redundant computations may not be stored as a result of index transformations. For example, for row-convolution and column-convolution kernels, all the data required for the partially-redundant computations was not stored as a result of the index transformations. For that purpose, we increased the size of per-block shared memory as shown in Table 5 and added a few lines of code to ensure that all data required for partial redundancy is stored in the shared memory.

The proposed method can detect a warp-level failure unless it occurs simultaneously in two or more of the following kernels: the application, partial redundancy, and diagnostic kernels. Given that the control logic is significantly smaller

TABLE 5. Shared memory used for proposed methods.

Application	Failure type	Application (original)	Application (modified)	Partial redundancy	Diagnosis
Matrix multiplication	Block	8,192 kB	8,192 kB	0 kB	0 kB
	Warp	8,192 kB	8,192 kB	8,192 kB	0 kB
	Thread	8,192 kB	8,192 kB	0 kB	0 kB
Separable convolution (row)	Block	2,560 kB	2,560 kB	0 kB	0 kB
	Warp	2,560 kB	2,560 kB	23,040 kB	0 kB
	Thread	2,560 kB	2,560 kB	0 kB	0 kB
Separable convolution (column)	Block	5,184 kB	5,184 kB	0 kB	0 kB
	Warp	5,184 kB	5,184 kB	15,552 kB	0 kB
	Thread	5,184 kB	5,184 kB	0 kB	0 kB

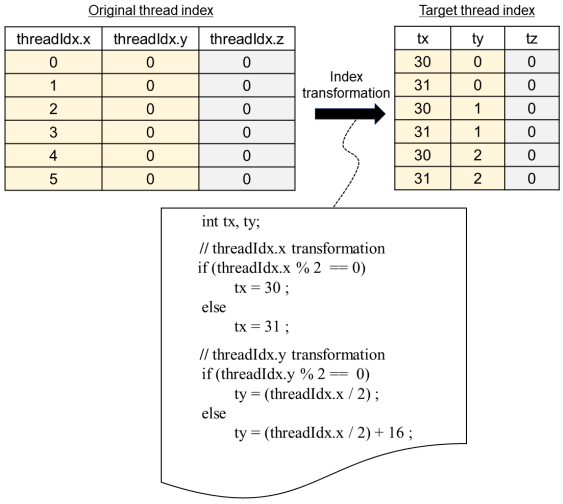

FIGURE 5. Example of transforming *threadIdx.x* and *threadIdx.y* for partially-redundant computations.

than the data units, its failure probability is inherently lower. Accordingly, the likelihood of concurrent failures at multiple points within such small control logic is exceedingly rare. Therefore, such simultaneous failures are not considered in the proposed method.

B. BLOCK-LEVEL AND THREAD-LEVEL FAILURE DETECTION

The proposed methods for detecting block-level and thread-level failures, similar to the warp-level failure detection, rely on the concurrent execution of a diagnostic kernel alongside the application kernel. An overview of these methods is provided in Fig. 6. Specifically, Fig. 6(a) depicts the hardware-level implementation, where the diagnostic kernel operates within a single SM to ensure low overhead; Fig. 6(b) presents the software-level implementation, where each diagnostic thread in the diagnostic kernel compares its actual and expected block and thread indices. In such implementations, failures are detected by comparing the actual index values obtained during application execution with the expected values, using the diagnostic kernel. Fig. 6(c) shows the corresponding execution timing, where diagnostic operations are repeatedly performed in parallel with the main application.

In the proposed methods, at the start of each thread's execution, the register values for block or thread indices (*blockIdx.x*, *blockIdx.y*, *blockIdx.z*, *threadIdx.x*, *threadIdx.y*, *threadIdx.z*) are read from special registers and immediately stored in global memory. Based on these index values, a unique value is computed for each thread or block, which is then used to compare the actual values obtained during execution with the expected values, enabling the detection of control-logic failures. or example, a unique value representing the block index is calculated using a composite expression—for instance, when *blockIdx.x* and *blockIdx.y* range from 0 to 15 and *blockIdx.z* is 0, the value can be computed as $256 \times blockIdx.z + 16 \times blockIdx.y + blockIdx.x$. The number of memory addresses needed to store this information equals the total number of threads.

As in the warp-level detection method, a conditional flag is written by each warp at the end of the application kernel. When this flag is set to a non-zero value, the diagnostic kernel in stream 2 [shown in Fig. 6(c)]—composed of a single block—compares the expected and actual index values repeatedly in sequence. The diagnostic kernel performs the same checks once, after the application kernel completes, regardless of the flag.

As a limitation, the proposed methods can detect block-level and thread-level failures unless they occur

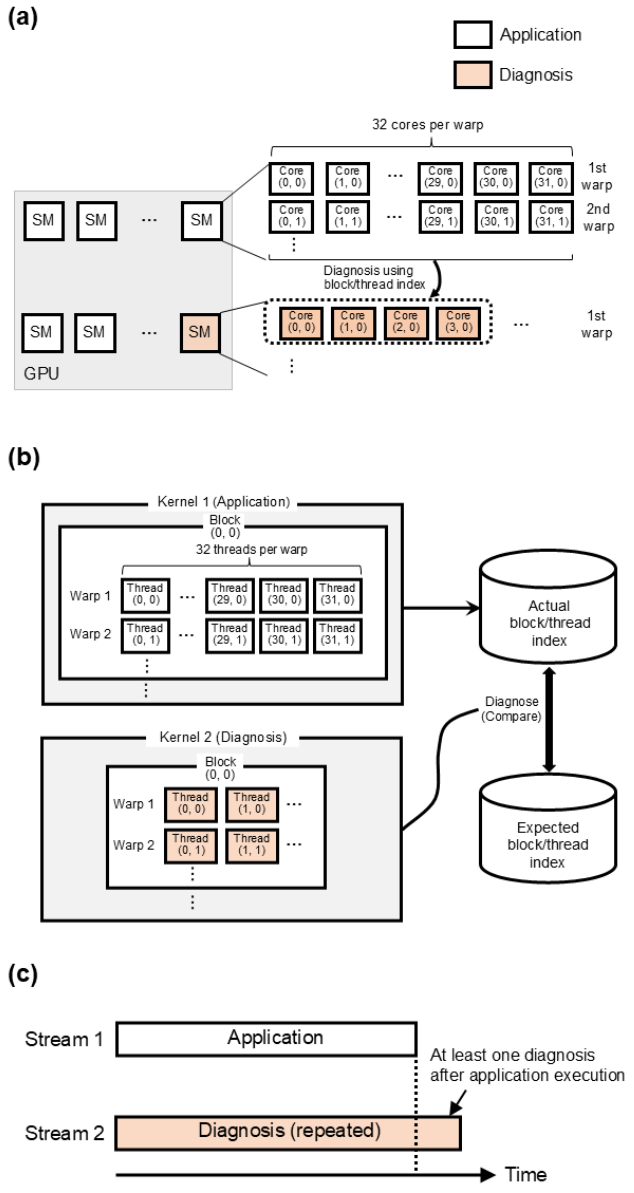

FIGURE 6. Proposed concurrent failure-detection methods for block- and thread-level control-logic failures, illustrated from three perspectives: (a) hardware-level implementation, where diagnosis is performed using one SM by comparing the actual and expected block/thread indices; (b) software-level implementation, where each diagnostic thread compares its actual and expected block/thread index; (c) execution timing flow, where diagnostic operations are repeatedly performed in parallel with the main application.

simultaneously in the application and diagnostic kernels. Given that the control logic is significantly smaller than the data units, its failure probability is inherently lower. Accordingly, the likelihood of concurrent failures at multiple points within such small control logic is exceedingly rare. Therefore, such simultaneous failures are not considered in the proposed method.

The proposed methods also cannot detect failures that occur in the mask in the SIMT stack. Protecting the mask requires additional measures—such as an error detection and correction mechanism—during the circuit design phase. Since the size of the mask in the SIMT stack is much smaller than the data units, the probability of a failure occurring in the mask is inherently lower than that in the data units. Furthermore, even if a failure occurs in the mask, its impact is limited to a maximum of 32 threads within a warp. Additionally, because the values in the SIMT stack are frequently overwritten with correct values during program execution, the overall influence of such failures is expected to be limited. As a result, failures occurring in the SIMT mask are not considered in the proposed method.

VI. EVALUATION OF PROPOSED METHOD

This section evaluates the proposed failure detection methods in terms of runtime overheads, memory-resource overheads, and failure detection rates for matrix multiplication and separable convolution applications. As described in Section V, simultaneous faults occurring at multiple locations in the control logic are considered extremely rare and are therefore excluded from the calculation of failure detection rates. In addition, faults in the mask of the SIMT stack are also excluded from the calculation, as their impact is expected to be limited due to the small size of the mask compared to data units and the fact that the mask is frequently updated during execution.

A. RUNTIME OVERHEAD

Table 6 summarizes the evaluation results of the runtime overheads for a matrix multiplication kernel and two separable convolution kernels. As in the fault-injection experiment, an NVIDIA Quadro P2000 was used. For the matrix multiplication application, only *threadIdx.x* was transformed while for the separable convolution application, *threadIdx.x*, *threadIdx.y*, *blockIdx.x* and *blockIdx.y* were transformed. This table demonstrates that the runtime overheads for the detection of warp-level failures in the three kernels were within 13.2%. Here, the application (original) refers to the original application, while the application (modified) refers to the slightly modified application for applying the proposed methods. The diagnosis refers to the comparison calculations for all the target addresses, while partial redundancy refers to the partially-redundant computations.

The time shown in the table was the average time for 10 trials, which was measured with NVIDIA Nsight. It was also used for visually confirming the concurrent execution of kernels since the kernels were executed asynchronously. The variation in the measured times was within 6.5%. The times for partial redundancy and application (modified) were measured when the concurrent execution of kernels was performed, while the time was not for application (original) and diagnosis. The time for diagnosis was measured without concurrent execution, as the diagnosis was performed independently at least once after the completion of the application execution. The runtime overhead was calculated as the ratio of [application (modified) + diagnosis] to

application (original) where the numerator corresponds to the total execution time of the kernels with the proposed methods.

For example, the runtime overhead for the matrix multiplication application was 5.3%. This small runtime overhead was attributed to the fact that the redundancy ratio was only 6.3% (2/32), and the concurrent execution of the multiple kernels was performed for efficient calculations. The time for diagnosis was shorter than the time for application (modified) since the diagnostic kernel performed only simple comparison calculations. The ratio of the time for application (modified) to that for partial redundancy was 12.4 for the matrix multiplication application. This value of 12.4 was reasonable since the total number of threads for the application (modified) kernel was 16 times that for the partial redundancy kernel. In contrast, the ratios for the separable convolution application were 1.5 for row-convolution kernel and 5.8 for column-convolution kernel, which were significantly less than 16. This was because, as described in the previous section, a few lines of code were added for the shared memory to load the required data for the partially-redundant computations. This modification increased the execution time for the partially-redundant computations for row-convolution and column-convolution kernels but did not increase the runtime overheads significantly.

Since the diagnostic kernel only occupied one SM during the kernel execution and the time for application (modified) was likely dependent on the number of remaining SMs, the use of GPUs with an increased number of SMs (eight for Quadro P2000) may further reduce the runtime overheads.

B. MEMORY OVERHEAD

Table 7 summarizes the evaluation results for the global memory overheads for each kernel in the matrix multiplication and separable convolution applications. These overheads in global memory were investigated since all the input and output data of each kernel were stored in the global memory. These overheads depend on the application and failure type. This table demonstrates that the global memory overhead for detecting warp-level failures in the three kernels was within 4.2%. The overheads for the detection of block-level and thread-level failures ranged from 18.9% to 101%.

For the detection of block-level and thread-level failures, global memory was required for storing the actual and expected index values, index comparison results, and flag value. In total, the required global memory size strongly depended on the number of threads. This was because the number of global memory addresses required for the actual index value and expected index value was equal to the number of threads. For the matrix multiplication, row convolution, and column convolution kernels, the total number of threads was 262,144, 18,432, and 18,432, respectively. The number of global memory addresses required for the flag value was equal to the number of warps, which was equal to the total number of threads divided by 32.

For the detection of warp-level failures, global memory was required for storing partially-redundant computation results, comparison results, and flag values. In total, the required global memory size was less than that required for block-level and thread-level failures. The size of the global memory required for storing the partially-redundant computation results was 2/32 of the size required to store the original application's computation results, since the partially-redundant computations were performed only for the two threads per warp in the original application. The size of the global memory for storing the flag values was equal to the number of warps.

C. FAILURE DETECTION RATE

A preliminary analysis was conducted to estimate the detection rate of control-logic failures using the proposed method. Although a precise estimation of the actual failure detection rate would ideally involve simultaneous fault injection into multiple kernels, this approach presents significant technical challenges, including the difficulty of managing concurrent fault states across kernels and the complexity of isolating interactions between faults. Furthermore, control-logic failures, which are the primary detection target of this study, have a low probability of occurrence in real-world environments, making a direct evaluation of the detection rate challenging. Therefore, such an approach lies beyond the scope of this paper.

To provide a preliminary analysis, the detection rate of the proposed method was calculated based on two key assumptions. First, it was assumed that the failure rates of three types of control-logic failures—within the application kernel, the partial redundancy kernel, and the diagnostic kernel—were proportional to both the number of threads and the execution time of each kernel. Second, it was assumed that the proposed method could detect failures that occurred within the application kernel, but not within the partial redundancy kernel or the diagnostic kernel, regardless of the timing or location of the failures. In other words, the detection rate of control-logic failures was defined as the probability of a failure occurring in the application kernel divided by the probability of a failure occurring across all kernels. This definition reflects a conservative estimation approach, and it assumes that no failures within the partial redundancy kernel or the diagnostic kernel can be detected.

Based on the aforementioned assumptions, the detection rate can be expressed as follows. Let N_a and t_a denote the number of threads and the runtime for the application kernel, respectively. Similarly, let N_p and t_p represent the number of threads and the runtime for the partial redundancy kernel, and N_d and t_d denote the number of threads and the runtime for the diagnostic kernel. The detection rate of control-logic failures can then be expressed as:

$$\frac{N_a \times t_a}{N_a \times t_a + N_p \times t_p + N_d \times t_d}$$

TABLE 6. Runtime overhead by proposed method.

Application	Failure type	A. Application (original)	B. Application (modified)	C. Diagnose	D. Partial redundancy	Runtime overhead (B+C)/A-1
Matrix multiplication	Block	946.4 μs	1,037.1 μs	305.9 μs	–	41.9%
	Warp	946.4 μs	978.9 μs	17.6 μs	78.7 μs	5.3%
	Thread	946.4 μs	1,022.6 μs	305.9 μs	–	40.4%
Separable convolution (row)	Block	85.5 μs	89.2 μs	24.4 μs	–	32.9%
	Warp	85.5 μs	89.1 μs	6.3 μs	58.5 μs	11.6%
	Thread	85.5 μs	88.9 μs	24.4 μs	–	32.5%
Separable convolution (column)	Block	85.0 μs	87.3 μs	24.6 μs	–	31.6%
	Warp	85.0 μs	89.2 μs	7.0 μs	15.3 μs	13.2%
	Thread	85.0 μs	87.6 μs	24.5 μs	–	31.9%

TABLE 7. Global memory overhead by proposed method.

Application	Failure type	A. Application (original)	B. Application (modified)	C. Diagnose	D. Partial redundancy	Memory overhead (B+C+D)/A-1
Matrix multiplication	Block	3,072 kB	4,128 kB	2,048 kB	0 kB	101.0%
	Warp	3,072 kB	3,104 kB	32 kB	64 kB	4.2%
	Thread	3,072 kB	4,128 kB	2,048 kB	0 kB	101.0%
Separable convolution (row)	Block	1,152 kB	1,226 kB	144 kB	0 kB	18.9%
	Warp	1,152 kB	1,154 kB	2 kB	36 kB	3.5%
	Thread	1,152 kB	1,226 kB	144 kB	0 kB	18.9%
Separable convolution (column)	Block	1,152 kB	1,226 kB	144 kB	0 kB	18.9%
	Warp	1,152 kB	1,154 kB	2 kB	36 kB	3.5%
	Thread	1,152 kB	1,226 kB	144 kB	0 kB	18.9%

D. COMPARISON WITH EXISTING METHODS

Table 8 summarizes the comparison results for the failure detection rates for the matrix multiplication application. The values for the conventional methods (#4–7) are drawn from the literature [7], [16], [18], [29]. Specifically, the CFCSS result corresponds to the matrix multiplication application [18], while the results for HAUBERK and STL represent averages across multiple applications [7], [29]. In addition, #7 represents a general duplication-based method, which is commonly used in safety-critical systems [16].

To ensure a consistent basis for comparison, we assume that all conventional methods (#4–6) detect faults by comparing actual and expected values stored in global memory. For the purpose of estimating memory overhead, we further assume that each thread stores one actual value and one expected value, each represented as a 32-bit unsigned integer (*i.e.*, 4 bytes). The general duplication-based method (#7) detects failures by executing the same application on two independent GPUs with identical input data and comparing their output results. In this method, we assume that control-logic failures occurring on one GPU do not propagate to the other GPU.

The table demonstrates that the proposed methods incur limited runtime overhead while maintaining high failure

TABLE 8. Comparison of methods in terms of runtime overhead, memory overhead, and failure detection rate.

#	Method	Detectable Failure	Category	Runtime overhead	Memory overhead	Failure detection rate
1	Proposed	Warp-level failure	Control logic	5.3%	4.2%	99.5%
2		Block-level failure	Control logic	41.9%	101.0%	99.9%
3		Thread-level failure	Control logic	40.4%	101.0%	99.9%
4	CFCSS [18]	Warp-level failure	Control logic	60.0%	67.0%	90.2%
5	HAUBERK [7]	Data failure (*e.g.*, register)	Data unit	15.3%	67.0%	86.8%
6	STL [29]	Warp-level failure	Control logic	<5%	67.0%	58.5%
7	Duplication [16]	All failures	All components	4.4%	100.0%	100.0%

detection rates. Notably, for warp-level failure detection, our method incurs only a 5.3% runtime overhead. We attribute this efficiency to the parallel and separated execution of application computations, partially-redundant computations, and diagnostic computations. In addition, the use of dedicated lightweight diagnostic threads, which operate independently of the application kernel, further contributes to the effectiveness of the proposed methods. Regarding memory overhead, Method #1 results in a comparatively low overhead of 4.2%, due to its redundancy ratio of 2/32. In contrast, Methods #2 and #3 require storing both actual and expected indices for all threads, along with additional flag values, resulting in higher memory overhead compared to Methods #4–#7.

When the three proposed methods are combined, over 99% of all types of control-logic failures can be detected. Furthermore, this combination enables identification of the specific type of control-logic failure, albeit with a total runtime overhead close to 90%. As the detection target expands from warp-level failures to block- and thread-level failures, both runtime and memory overheads increase (up to 101.0% for memory overheads), introducing a trade-off between detection coverage and computational efficiency. Selecting the optimal combination of these methods and existing methods based on application requirements, particularly to detect critical control-logic failures, remains an area for future research.

As highlighted in this section, the proposed methods can achieve high failure detection rates, and are efficient in terms of runtime and memory-resource overheads, particularly for detecting warp-level failures. Furthermore, they can be implemented in various applications by changing only a small portion of original application code. For example, for the matrix multiplication application, the total number of changed or added statements for detecting block-level, warp-level, or thread-level failures, was less than 6. In the future, the methods will be usable for detecting control-logic failures and measuring the rate of each failure for various GPU architectures in diverse environments. They will also be useful for distinguishing between failures originating from data units and those originating from control logic, which will contribute to further improvements in system resiliency. If such failures can be accurately distinguished, appropriate countermeasures can be applied to each failure type, leading to more effective mitigation and enhanced system resiliency.

VII. CONCLUSION

In this paper, we defined and analyzed three types of GPU control-logic failures: block-level, warp-level, and thread-level failures. By using our developed PTX-based fault-injection framework, we demonstrated that these failures can cause SDCs without triggering any alarms, highlighting the critical importance of addressing them for the reliability of GPU-embedded systems.

For each failure type, we proposed detection methods and evaluated their detection effectiveness, runtime overhead, and memory-resource usage. Notably, for the matrix multiplication application, our proposed method for detecting warp-level failures—based on the parallel and separated execution of kernels—achieved a detection rate of 99.5%, with a runtime overhead of 13.2% and a memory overhead of 4.2%. These methods can be implemented with small modifications to the original application code and can be applied across a wide range of GPU applications.

Overall, our work contributes to enhancing the resilience of GPU-based systems by enabling efficient detection of control-logic failures. The proposed techniques can be used for analyzing the frequency and impact of such failures in real-world applications. This opens the door to future studies on system-level resilience in safety-critical domains.

As future work, we plan to apply the proposed detection methods to applications other than matrix multiplication, and compare their effectiveness and overhead with other conventional fault-detection techniques. This will help evaluate the further validity of the proposed approach for a broader GPU applications. In addition, we aim to investigate hybrid

approaches that combine the proposed methods with existing techniques for further detection efficiency.

REFERENCES

[1] A. Lotfi, S. Hukerikar, K. Balasubramanian, P. Racunas, N. Saxena, R. Bramley, and Y. Huang, "Resiliency of automotive object detection networks on GPU architectures," in *Proc. IEEE Int. Test Conf. (ITC)*, Nov. 2019, pp. 1–9.

[2] Z. Guo, S. Perminov, M. Konenkov, and D. Tsetserukou, "HawkDrive: A transformer-driven visual perception system for autonomous driving in night scene," in *Proc. IEEE Intell. Vehicles Symp. (IV)*, Jun. 2024, pp. 2598–2603.

[3] E. Ibe, H. Taniguchi, Y. Yahagi, K.-I. 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 Trans. Electron Devices*, vol. 57, no. 7, pp. 1527–1538, Jul. 2010.

[4] 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," in *Proc. IEEE 25th Asian Test Symp. (ATS)*, Nov. 2016, pp. 293–297.

[5] R. A. Nawrocki and R. M. Voyles, "Artificial neural network performance degradation under network damage: Stuck-at faults," in *Proc. Int. Joint Conf. Neural Netw.*, Jul. 2011, pp. 442–449.

[6] A. Chatzidimitriou, M. Kaliorakis, S. Tselonis, and D. Gizopoulos, "Performance-aware reliability assessment of heterogeneous chips," in *Proc. IEEE 35th VLSI Test Symp. (VTS)*, Apr. 2017, pp. 1–6.

[7] K. S. Yim, C. Pham, M. Saleheen, Z. Kalbarczyk, and R. Iyer, "Hauberk: Lightweight silent data corruption error detector for GPGPU," in *Proc. IEEE Int. Parallel Distrib. Process. Symp.*, May 2011, pp. 287–300.

[8] N. Maruyama, A. Nukada, and S. Matsuoka, "Software-based ECC for GPUs," in *Proc. Symp. Appl. Accel. High Perform. Comput.*, Jan. 2009, pp. 1–3.

[9] M. Goncalves, F. Fernandes, I. Lamb, P. Rech, and J. R. Azambuja, "Selective fault tolerance for register files of graphics processing units," *IEEE Trans. Nucl. Sci.*, vol. 66, no. 7, pp. 1449–1456, Jul. 2019.

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

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

[12] N. Farazmand, R. Ubal, and D. Kaeli, "Statistical fault injection-based AVF analysis of a GPU architecture," in *Proc. IEEE Workshop Silicon Errors Log.-Syst. Effects*, Jan. 2012, pp. 1–6.

[13] K. Ito, Y. Zhang, H. Itsuji, T. Uezono, T. Toba, and M. Hashimoto, "Analyzing DUE errors on GPUs with neutron irradiation test and fault injection to control flow," *IEEE Trans. Nucl. Sci.*, vol. 68, no. 8, pp. 1668–1674, Aug. 2021.

[14] J. E. R. Condia, P. Rech, F. F. D. Santos, L. Carro, and M. S. Reorda, "An effective method to identify microarchitectural vulnerabilities in GPUs," *IEEE Trans. Device Mater. Rel.*, vol. 22, no. 2, pp. 129–141, Jun. 2022.

[15] J.-D. Guerrero-Balaguera, J. E. R. Condia, F. F. dos Santos, M. S. Reorda, and P. Rech, "Understanding the effects of permanent faults in GPU's parallelism management and control units," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, Nov. 2023, pp. 1–15.

[16] A. Golander, S. Weiss, and R. Ronen, "DDMR: Dynamic and scalable dual modular redundancy with short validation intervals," *IEEE Comput. Archit. Lett.*, vol. 7, no. 2, pp. 65–68, Dec. 2008.

[17] A. Mahmoud, S. K. S. Hari, M. B. Sullivan, T. Tsai, and S. W. Keckler, "Optimizing software-directed instruction replication for GPU error detection," in *Proc. SC18: Int. Conf. High Perform. Comput., Netw., Storage Anal.*, Nov. 2018, pp. 842–854.

[18] N. Oh, P. P. Shirvani, and E. J. McCluskey, "Control-flow checking by software signatures," *IEEE Trans. Rel.*, vol. 51, no. 1, pp. 111–122, Mar. 2002.

[19] O. Goloubeva, M. Rebaudengo, M. Sonza Reorda, and M. Violante, "Soft-error detection using control flow assertions," in *Proc. 18th IEEE Symp. Defect Fault Tolerance VLSI Syst.*, Nov. 2003, pp. 581–588.

[20] J. Vankeirsbilck, V. B. Thati, H. Hallez, and J. Boydens, "Inter-block jump detection techniques: A study," in *Proc. 25th Int. Sci. Conf. Electron. (ET)*, Sep. 2016, pp. 1–4.

[21] H. Itsuji, T. Uezono, T. Toba, K. Ito, and M. Hashimoto, "Concurrent detection of failures in GPU control logic for reliable parallel computing," in *Proc. IEEE Int. Test Conf. (ITC)*, Nov. 2020, pp. 1–5.

[22] 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. IEEE Int. Symp. Perform. Anal. Syst. Softw. (ISPASS)*, Apr. 2017, pp. 249–258.

[23] A. Vallero, D. Gizopoulos, and S. Di Carlo, "SIFI: AMD southern islands GPU microarchitectural level fault injector," in *Proc. IEEE 23rd Int. Symp. On-Line Test. Robust Syst. Design (IOLTS)*, Jul. 2017, pp. 138–144.

[24] NVBITFI. *NVIDIA*. Accessed: Jan. 2025. [Online]. Available: https://github.com/NVlabs/nvbitfi

[25] 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.

[26] 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.

[27] Y. Ibrahim, H. Wang, M. Bai, Z. Liu, J. Wang, Z. Yang, and Z. Chen, "Soft error resilience of deep residual networks for object recognition," *IEEE Access*, vol. 8, pp. 19490–19503, 2020.

[28] F. F. D. Santos, P. F. Pimenta, C. Lunardi, L. Draghetti, L. Carro, D. Kaeli, and P. Rech, "Analyzing and increasing the reliability of convolutional neural networks on GPUs," *IEEE Trans. Rel.*, vol. 68, no. 2, pp. 663–677, Jun. 2019.

[29] J. E. R. Condia, F. A. da Silva, A. Ç. Bagbaga, J.-D. Guerrero-Balaguera, S. Hamdioui, C. Sauer, and M. S. Reorda, "Using STLs for effective in-field test of GPUs," *IEEE Des. Test. IEEE Des. Test. Comput.*, vol. 40, no. 2, pp. 109–117, Apr. 2023.

[30] S. Di Carlo, G. Gambardella, M. Indaco, I. Martella, P. Prinetto, D. Rolfo, and P. Trotta, "A software-based self test of CUDA Fermi GPUs," in *Proc. 18th IEEE Eur. Test Symp. (ETS)*, Avignon, France, May 2013, pp. 1–6.

[31] K. Andryc, M. Merchant, and R. Tessier, "FlexGrip: A soft GPGPU for FPGAs," in *Proc. Int. Conf. Field-Programmable Technol. (FPT)*, Dec. 2013, pp. 230–237.

[32] *GPGPU-Sim Developed By the University of British Columbia*. Accessed: Jan. 2025. [Online]. Available: http://www.gpgpu-sim.org/

[33] 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.

[34] XID Errors. *NVIDIA*. Accessed: Jan. 2025. [Online]. Available: https://docs.nvidia.com/deploy/XID-errors/index.html

[35] D. Tiwari, S. Gupta, J. Rogers, D. Maxwell, P. Rech, S. Vazhkudai, D. Oliveira, D. Londo, N. DeBardeleben, P. Navaux, L. Carro, and A. Bland, "Understanding GPU errors on large-scale HPC systems and the implications for system design and operation," in *Proc. IEEE 21st Int. Symp. High Perform. Comput. Archit. (HPCA)*, Feb. 2015, pp. 331–342.

HIROAKI ITSUJI (Member, IEEE) received the M.S. and Ph.D. degrees in electrical engineering and information systems from The University of Tokyo, Tokyo, Japan. He is currently with Hitachi, Ltd., Yokohama, Japan. His current research interests include design and diagnostic technologies to enhance the reliability of electronic systems across various sectors.

TAKUMI UEZONO (Member, IEEE) received the B.E., M.E., and Ph.D. degrees in computer science and electrical and electronic engineering from Tokyo Institute of Technology, Tokyo, Japan, in 2005, 2007, and 2010, respectively. He was a Research Fellow with Japan Society for the Promotion of Science, from 2009 to 2011. He was a Postdoctoral Researcher with Kyoto University, Kyoto, Japan, from 2010 to 2011. He is currently with Hitachi, Ltd., Yokohama, Japan. His current research interests include circuit and system design for reliable and safe systems.

TADANOBU TOBA received the M.S. and Ph.D. degrees in system safety and information science from Nagaoka University of Technology, Niigata, Japan. He is currently with Hitachi, Ltd., Yokohama, Japan. His current research activity focuses on dependable technology and system design.

KOJIRO ITO received the bachelor's degree from the School of Engineering Science, Osaka University, in 2019, and the master's degree from the Graduate School of Information Science and Technology, Osaka University, in 2021. He is currently with Asahi Kasei Corporation. His research interests include radiation effects on computer systems, factory automation, and DX promotion.

MASANORI HASHIMOTO (Senior Member, IEEE) received the B.E., M.E., and Ph.D. degrees in communications and computer engineering from Kyoto University, Kyoto, Japan, in 1997, 1999, and 2001, respectively.

He is currently a Professor with the Graduate School of Informatics, Kyoto University. His current research interests include design for reliability, timing and power integrity analysis, reconfigurable computing, soft error characterization, and low-power circuit design. He was on the technical program committees of international conferences, including DAC, ICCAD, ITC, IRPS, Symposium on VLSI Circuits, ASP-DAC, and DATE. He serves/served as the TPC Chair for ASP-DAC 2022, the Editor-in-Chief for *Microelectronics Reliability* (Elsevier), and an Associate Editor for IEEE TRANSACTIONS ON VERY LARGE SCALE INTEGRATION (VLSI) SYSTEMS, IEEE TRANSACTIONS ON CIRCUITS AND SYSTEMS—I: REGULAR PAPERS, and ACM TODAES.