Avoiding Soft Error-induced Illegal Memory Accesses in GPU with Inter-thread Communication

Riku Iwamoto[1] Masanori Hashimoto[2]
[1]Dept. Information Systems Engineering, Osaka University
[2]Dept. Informatics, Kyoto University

***Abstract*—A soft error caused by terrestrial neutrons poses a threat to the reliability of safety-critical systems, such as self-driving applications. These applications, often comprised of neural networks, rely on graphic processing units (GPUs) due to their requirement for massive parallel computation. While neural networks inherently include redundant computation and possess a certain level of error tolerance, detectable unrecoverable errors (DUEs) can be more detrimental than silent data corruption (SDC), as they can result in temporary service unavailability. This study specifically focuses on addressing illegal memory access, a primary cause of DUEs, and proposes a programming method that can detect illegal addresses. In the single instruction, multiple threads (SIMT) scheme, the data address is regularly calculated based on the thread ID, and this regularity is exploited to identify illegal addresses through inter-thread communication. To evaluate the effectiveness of the proposed method, fault injection campaigns were conducted for matrix multiplication, vector addition, and transposition. The experimental results indicate that the proposed method resulted in a reduction of the DUE rate by 17.3%, 86.8%, and 87.1% for these respective operations.**

I. Introduction

Graphics processing units (GPUs) are used to compute neural networks and high-performance computing (HPC) applications thanks to their highly-parallel computing capability. Weather forecasting and automated driving systems require such high computing power, and the use of GPUs is being advanced [1] [2]. Neural networks have achieved remarkable results in several areas, such as object recognition, and are used in automatic driving systems to detect pedestrians and other objects. On the other hand, HPC requires long computation times, which increases the likelihood of faults occurring during execution. In both cases, the final results can be erroneous if the values affected by a fault are used. Reliability is, therefore, a vital metric in HPC systems and safety-critical systems such as automated systems.

Soft errors are one of the factors that pose a threat to system reliability. In a terrestrial environment, these errors are primarily caused by neutrons originating from cosmic rays, which can invert the stored values in memory elements. While soft errors are not permanent failures and can be corrected by rewriting the affected values, they can propagate when incorrect values altered by soft errors are read and utilized. This propagation can result in erroneous calculation results or even program termination.

The effects of soft errors can be categorized into three types:

- Mask: These errors do not have any impact on the program's output.
- Silent data corruption (SDC): While the program terminates normally, the resulting output is different from the expected one.
- Detectable unrecoverable error (DUE): These errors cause the program to stop or become unresponsive.

The severity of SDC or DUE depends on the nature of the specific application. Neural networks, for instance, incorporate redundant computations and exhibit robustness against computational errors [3]. On the other hand, real-time performance is crucial in automated driving systems, where it is difficult to tolerate temporal service unavailability. To illustrate the impact of DUEs in such scenarios, consider an automated driving system that comes to a halt for one second due to a DUE. If the vehicle is traveling at 60 km/h, it will cover approximately 16.6 meters during this time. Considering that the standard distance and time between vehicles traveling at 60 km/h are approximately 45 meters and 2 to 3 seconds, respectively, the consequences of the system halting can be significant. In these applications, the avoidance of DUEs is of utmost importance to ensure safety.

The vulnerability of certain GPU microarchitectures to soft errors has been investigated in the literature (e.g., [4]). To mitigate the impact of SDC, error-correcting code (ECC) is employed as a memory error correction mechanism capable of rectifying single-bit errors [5]. By utilizing such memory protection functions, the effects of SDC can be alleviated. However, it is important to note that ECC primarily aims to safeguard the contents written to memory and may not directly address DUEs resulting from out-of-range memory accesses with incorrect addresses. Besides, ECC can contribute to reducing incorrect address computation. Neutron irradiation experiments conducted on GPUs have revealed that DUEs caused by memory accesses to incorrect addresses account for 40% to 50% of all DUE occurrences [6]. Hence, implementing countermeasures to prevent illegal memory addresses is crucial in order to mitigate DUEs in GPU applications.

Compute Unified Device Architecture (CUDA), developed by NVIDIA, is a programming language used for developing GPU applications. In CUDA, programs are executed using a single instruction multiple threads (SIMT) approach, where each thread executes the same program in parallel. A common example is vector addition, where multiple threads are launched, and each thread computes a single element of the vector. As the address to be loaded depends on each thread, CUDA programs typically involve frequent calculations of thread-dependent addresses.

This paper focuses on addressing out-of-range memory accesses caused by soft errors and proposes a programming

979-8-3503-4135-5/23/$31.00 © 2023 IEEE

2023 IEEE 29th International Symposium on On-Line Testing and Robust System Design (IOLTS) | 979-8-3503-4135-5/23/$31.00 ©2023 IEEE | DOI: 10.1109/IOLTS59296.2023.10224889

Authorized licensed use limited to: Kyoto University. Downloaded on September 12,2023 at 19:12:36 UTC from IEEE Xplore. Restrictions apply.

Listing 1: Example of thread ID usage.

```
1 int treadID = blockIdx.x * blockDim.x + threadIdx
      .x;
2 C[threadID] = A[threadID] + B[threadID];
```

method to mitigate DUEs by detecting such accesses. The main contribution of this work lies in providing a software-level solution that reduces DUE occurrences by verifying the correctness of memory addresses prior to accessing them. The proposed method does not rely on redundant computation or storage; instead, it leverages inter-thread communication for address checking.

The remaining sections of this paper are organized as follows. Section II provides an overview of related works. Section III discusses the necessary preliminaries concerning GPU computation and memory. Section IV presents the proposed method for preventing wrong memory accesses. Section V presents experimental results demonstrating the effectiveness of the proposed method in reducing DUEs. Finally, Section VI concludes the paper.

II. Related Work

M. M. Goncalves et al. propose a method to reduce overhead by selectively applying fault tolerance techniques to registers based on their occupation and impact on performance [7]. D. A. G. G. de Oliveira et al. apply an algorithm-based fault tolerance technique utilizing checksum vectors for matrix multiplication on GPUs [8]. F. F. dos Santos et al. propose a reduced-precision duplication-with-comparison method that executes redundant copies in reduced precision to minimize the overhead of duplication with comparison [9]. The following two methods for neural networks are not specifically designed for GPUs but are applicable to them. G. Li et al. propose using the value ranges of activations as a symptom to detect SDC-causing faults. L.-H. Hoang et al. propose utilizing a clipped ReLU function [10]. The clipped ReLU function returns 0 if the input x is too large, mitigating the impact of bit flips. These papers primarily focus on mitigating SDC originating from register values.

Itsuji et al. propose a software-based technique that concurrently detects control-logic failures in GPUs by utilizing signature computation and comparison within a running application, while primarily maintaining its throughput [11]. While this work specifically focuses on control-logic failures in GPUs, its objective is the mitigation of SDC. Existing works mainly concentrate on SDC mitigation, and to the best of the authors' knowledge, explicit research on reducing DUEs in GPUs has not been conducted. Although some techniques developed for CPUs (e.g., [12]) may be applicable to GPUs, their effectiveness should be evaluated specifically for GPU implementations.

III. Preliminaries

A. CUDA programming

CUDA programs operate through collaboration between the host (CPU) and the device (GPU). The host side assumes control over the entire program, including tasks such as launching functions to be executed on the device side and transferring data. On the other hand, the device side carries out operations that leverage the highly parallel computing capabilities of the GPU. These operations on the GPU are executed in a SIMT mode, where a single instruction is executed across multiple threads simultaneously.

The function invoked from the host side running on the GPU is referred to as a kernel function. When invoking a kernel function, it is necessary to specify the block and grid sizes. A block represents a group of threads, while a grid represents a collection of blocks. Both the block and grid sizes can be specified using three-dimensional values (x, y, z). Each thread executes the same program, but built-in variables such as threadIdx, blockIdx, and blockDim are utilized to calculate the data address for each thread. This enables operations on distinct data within different threads. Listing 1 presents an example code snippet. Since the value of threadID varies for each thread, each thread can access different addresses and perform operations using distinct data.

A GPU is composed of multiple modules known as streaming multi-processors (SMs). Each SM encompasses various components, including the instruction cache, warp scheduler, warp dispatcher, register file, and CUDA cores. At the software level, an SM corresponds to a block, and one or more blocks are assigned to an SM. A warp refers to a group of threads allocated to a block, with a maximum of 32 threads per warp. Within a block, each thread is assigned a lane number ranging from 0 to 31. Instructions are shared among the threads within a warp during the execution phase. In other words, threads belonging to the same warp in an SM share a common program counter and execute the same instructions.

B. GPU memory structure

Since a GPU executes a large number of threads concurrently, multiple memories are provided to accommodate the accessible range of each thread. Table I provides an overview of the available memories and their respective access restrictions.

- Registers: Registers are memory elements that are exclusively accessible by each thread. The maximum number of registers can be utilized per thread depends on the GPU architecture.
- Shared memory: Shared memory is provided for each streaming multi-processor (SM) and is thus shared among threads within the same block. It can be shared with the L1 cache and Texture caches in new GPU architectures.
- Global memory: Global memory is a memory component accessible to all threads and is commonly used for data transfer from the host due to its larger size.
- Constant memory: Constant memory is employed to store constants and can be read via device functions but not written to. During the execution of kernel functions, the size of the block or grid is typically stored here. Constant memory is accessible by all threads.
- Special registers: Special registers store specific variables such as lane ID and block ID. They are read-only from threads and serve special purposes.

Authorized licensed use limited to: Kyoto University. Downloaded on September 12,2023 at 19:12:36 UTC from IEEE Xplore. Restrictions apply.

Listing 2: Code example of vector addition.

```
1  #define NB  5
2  #define LEN 32
3  __global__ void vectorAdd(int *A,int *B,int *C);
4  int main(){
5    // pre-processing
6    vectorAdd<<<NB,LEN>>>(A,B,C);
7    // post-processing
8  }
9  __global__ void vectorAdd(int *A,int *B,int *C){
10   int tid = blockIdx.x * blockDim.x + threadIdx.x
       ;
11   C[tid] = A[tidy] + B[tid];
12 }
```

TABLE I: Memory access restrictions.

Memory	User write	Accessible range
(general-purpose) register	OK	within a thread
shared memory	OK	within a block
global memory	OK	within a GPU
constant memory	not allowed	within a GPU
special register	not allowed	unknown

Let us consider Listing 2, which presents the kernel function vectorAdd, as a code example. In Line 10, registers, special registers, and constant memory are utilized. Specifically, one register is used to store the result of the `tid` calculation. It is important to note that intermediate calculation results are also temporarily stored in registers, even though they are not explicitly shown in the source code. Access to special registers is required to retrieve the values of `blockIdx.x` and `threadIdx.x`. Copying data from a special register to a general-purpose register necessitates the use of a dedicated instruction known as S2R, which is an instruction in the CUDA assembly language (SASS). Constant memory is accessed to retrieve the value of `blockDim.x`. This particular value of `blockDim.x` is shared by all threads created by the kernel function invoked in Line 6.

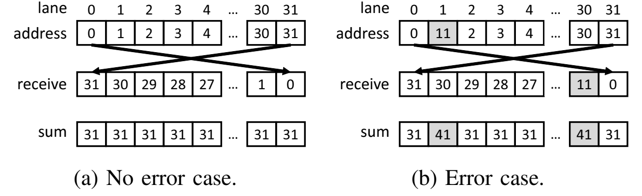
In Line 11, both global memory and registers are employed. Similar to Line 10, registers are utilized to reference `tid` and temporarily store intermediate calculation results. Global memory is accessed to retrieve the values of `A[tid]` and `B[tid]`, and also to store the result in `C[tid]`.

IV. Proposed method

A. Idea

In CUDA programs, thread IDs are computed using built-in variables, which are subsequently used to perform similar calculations on different data in each thread. As a result, threads within the same warp frequently access addresses that are equally spaced or the same, starting from the thread in lane 0. Therefore, instead of redundant multiple calculations such as DMR (Dual Modular Redundancy) and TMR (Triple Modular Redundancy), where address calculations are repeated in the same thread, address values can be shared between threads through inter-thread communication to verify if they have been altered to invalid values.

Let us consider the example provided in Listing 1. When accessing the address represented by `A[threadID]`, the threads within the warp have consecutive thread IDs, resulting in consecutive indices. For instance, let us examine the



(a) No error case. (b) Error case.

Fig. 1: Illustration of the proposed address checking, where $address(0) = 0$ and $d = 1$.

addresses accessed when threads with thread IDs 0, 1, and 2 access `A[threadID]`. Assuming that the array `A` is of type int (which is a 32-bit value with a size of 4), and considering the starting address of the array to be 30, the addresses accessed by each thread would be 30, 34, and 38, respectively. In this manner, if the thread IDs are consecutive, the accessed addresses are aligned at equal intervals. This characteristic is leveraged to detect errors.

The address accessed by a thread with lane number k can be calculated using the address $address(0)$ accessed by the thread with lane number 0. The relationship is given by the following equation:

$$address(k) = address(0) + k \times d, \tag{1}$$

where d represents the size of the value type being loaded. In the context of inter-thread communication, when a thread with lane number k receives the address value, denoted as $receive(k)$, from another thread within the same warp, the expression for $receive(k)$ is as follows when the number of threads in a warp is 2^n and $receive(k)$ is received from the thread with lane number $2^n - 1 - k$ in the same warp.

$$\begin{aligned} receive(k) &= address(2^n - k - 1), \\ &= address(0) + (2^n - k - 1) \times d. \end{aligned} \tag{2}$$

Then, let us consider the sum of Eqs. (1) and (2), $sum(k)$, as follows.

$$\begin{aligned} sum(k) &= address(k) + receive(k) \\ &= address(k) + address(2^n - k - 1), \\ &= (address(0) + k \times d), \\ &\quad + (address(0) + (2^n - k - 1) \times d), \\ &= 2 \times address(0) + (2^n - 1) \times d. \end{aligned} \tag{3}$$

Now, we can observe that $sum(k)$ is a constant value that is independent of the lane number k. By comparing this constant with the values obtained from other threads in the same warp, we can determine whether the address accessed by each thread is correct.

Fig. 1 illustrates the proposed address checking method. In this example, we assume that $address(0) = 0$ and $d = 1$. If the address accessed by lane #1 is incorrect and is 11, the sum of the values from lane #1 and lane #30 would be 41, which differs from the sum of 31 obtained by the other threads.

B. Overhead reduction

The basic idea presented in this section is to perform one comparison and exchange operation for each memory access.

Authorized licensed use limited to: Kyoto University. Downloaded on September 12,2023 at 19:12:36 UTC from IEEE Xplore. Restrictions apply.

However, this approach can introduce significant overhead in programs that have frequent memory accesses. To address this issue, the following methods are proposed to reduce the overhead by leveraging multiple addresses and loop unrolling.

1) Multiple arrays: First, let us discuss the handling of multiple array addresses in a thread. Taking Listing 1 as an example, as explained in Section IV-A, the address of `A[threadID]` can be exchanged between the corresponding threads in the warp and added together. As a result, the sum in Eq. (3) becomes a constant value that is independent of the lane number k. The same approach can be applied to `B[threadID]` and `C[threadID]`, but it would involve repetitive calculations and operations. Therefore, we need a method to handle multiple addresses simultaneously. In the following discussion, we will focus on `A[threadID]` and `B[threadID]` for simplicity.

Suppose that the thread with lane number k is accessing the address of `A[threadID]`. Let $addrA(k)$ and $addrB(k)$ represent the addresses of `A[threadID]` and `B[threadID]`, respectively. By utilizing the addresses $addrA(0)$ and $addrB(0)$ accessed by the thread with lane number 0 and the size of the data type d, we can express the sum $addrA(k)$ and $addrB(k)$, denoted as $addr(k)$, as follows.

$$\begin{aligned} addr(k) &= addrA(k) + addrB(k), \\ &= addrA(0) + k \times d + addrB(0) + k \times d, \\ &= (addrA(0) + addrB(0)) + k \times 2d. \end{aligned} \tag{4}$$

When this value is exchanged with the thread in lane number $2^n - 1 - k$, $receive(k)$ becomes as follows.

$$\begin{aligned} receive(k) &= addr(2^n - 1 - k), \\ &= (addrA(0) + addrB(0)) \\ &\quad + (2^n - 1 - k) \times 2d. \end{aligned} \tag{5}$$

Therefore, the sum of $addr(k)$ and $receive(k)$ is expressed by

$$\begin{aligned} sum(k) &= addr(k) + receive(k), \\ &= 2 \times (addrA(0) + addrB(0)) + (2^n - 1) \times 2d, \end{aligned} \tag{6}$$

where $sum(k)$ is a constant independent of k. Thus, even in the case that a thread uses multiple addresses calculated using the thread ID, the sum of the addresses in Eq. (6) can be exchanged between the threads in the warp, and the sum of the received values can be compared for equality within the warp.

2) Loop unrolling: Next, consider operations on identical arrays to which loop unrolling is often applied. Loop unrolling is a method for speeding up the computation in loops. By (partially) expanding the loop process, the number of branches and conditional decisions can be reduced, and the computation in the loop is speeded up. The following assumes Listing 3, a code example of partially loop-unrolled matrix multiplication. First, let us investigate whether the basic idea can be applied to arrays `A` and `B`.

In the program of Listing 3, the thread with $(tidx, tidy) = (n, m)$ computes the elements (n, m) of the matrix `C`. Here,

Listing 3: Example code of partially loop-unrolled matrix multiplication.

```
1  int tidx = blockIdx.x * blockDim.x + threadIdx.x;
2  int tidy = blockIdx.y * blockDim.y + threadIdx.y;
3  float tmp=0.0f;
4  for (int i=0; i<LENGTH; i+=4){
5      tmp += A[LENGTH*tidy+i] + B[LENGTH*i+tidx];
6      tmp += A[LENGTH*tidy+(i+1)] + B[LENGTH*(i+1)+
           tidx];
7      tmp += A[LENGTH*tidy+(i+2)] + B[LENGTH*(i+2)+
           tidx];
8      tmp += A[LENGTH*tidy+(i+3)] + B[LENGTH*(i+3)+
           tidx];
9  }
10 C[tidx+LENGTH*tidy] = tmp;
```

`LENGTH` is a macro constant representing the length of one side of the square matrix. The value of $tidy$ is common within the warp.

Denote the array element in the loop as `A[LENGTH*tidy+(i+t)]`, where t is an index $(0 \leq t \leq 3)$ specifying the unrolled statements. Noting the address of `A[LENGTH*tidy+(i+t)]`, the value of $tidy$ is shared within the warp, and the index is the same within the warp as it does not contain any $tidx$. Therefore, the address of `A[LENGTH*tidy+(i+t)]` can be compared.

Next, let us consider the address of `B[LENGTH*(i+t)+tidx]`. Here, as t increments by one, the array index value increases by LENGTH. If the address at $t = 0$ is $addr_{t=0}$, the sum is $4 \times addr_{t=0} + (1 + 2 + 3) \times LENGTH \times d$. When the lane number is k, the sum of addresses $sum(k)$ can be written as follows.

$$\begin{aligned} sum(k) &= 4 \times (addrB(0) \times k \times d) \\ &\quad + 6 \times d \times LENGTH, \\ &= 4 \times addrB(0) + 6 \times d \times LENGTH \\ &\quad + k \times (4 \times d), \end{aligned} \tag{7}$$

where $addrB(0)$ is the address of `B[LENGTH*i+tidx]` in the thread with lane number 0. As before, when exchanging the values with the thread in lane number $2^n - 1 - k$ and adding the received value, the sum becomes a constant independent of k.

The method introduced in Section IV-B1 and the method in this section are orthogonal and compatible, allowing them to be used simultaneously. By summing up all the addresses of arrays `A` and `B` in Listing 3 and comparing their sum for equality within a warp, the number of comparisons within the warp is reduced by 1/8, thereby contributing to overhead reduction.

C. Implementation with CUDA

Thread-to-thread communication uses the `__shfl_xor_sync()` and `__match_all_sync()` functions available in compute capability 7.0 and above. The exchange and comparison of values using these methods are limited to threads in the same warp but have the advantage that no address calculation is required. Listing 4 shows an example implementation of address exchange and comparison using these functions.

Authorized licensed use limited to: Kyoto University. Downloaded on September 12,2023 at 19:12:36 UTC from IEEE Xplore. Restrictions apply.

Listing 4: Implementation example.

```
1  unsigned long long addr = (unsigned long long) &A
       [threadID];
2  unsigned long long receive  =  __shfl_xor_sync(
       __activemask(), addr, WARPSIZE-1, WARPSIZE);
3  unsigned long long sum = addr + receive;
4  int isMatch;
5  __match_all_sync(__activemask(), sum, &isMatch);
```

In Line 1, the address of `A[threadID]` is stored in `addr`. Line 2 exchanges the value of `addr` with other threads in the warp and stores the received value in the variable `receive`. In Line 3, the sum of `addr` and `receive` is stored in the variable `sum`. Line 4 declares the variable that stores the boolean value of the comparison result. Line 5 compares the value of the variable `sum` within the warp.

Next, the functions and constants used in Listing 4 are explained. The `__activemask()` function returns the lanes owned by the warp to which the thread belongs, and when the Nth bit is 1, it indicates that the Nth lane exists. `WARPSIZE` is a macro constant representing the maximum number of threads per warp, currently set to 32 for all architectures.

The function `__shfl_xor_sync()` is a function that exchanges values with other threads in the warp. The arguments, in order, specify the lanes involved in the exchange, the values to be exchanged, the lane mask, and the lane width for exchanging values. The lane mask is a value used to calculate the lane number of the exchange destination. The destination lane, $destination_lane$, is calculated by applying the bitwise XOR operation between the source lane, $source_lane$, and the lane mask, $lanemask$, as follows.

$$destination_lane = source_lane \oplus lanemask, \quad (8)$$

where the lanes can be exchanged in reverse order by setting the lane mask to 31.

The function `__match_all_sync()` is used to perform equality comparisons among threads in a warp. The function takes three arguments: the lanes involved in the comparison, the values to be compared, and the variables that store the true and false values. The third argument will be true only if the values of the second argument are equal across all participating lanes in the comparison.

V. Experimental Results

In this section, we show the execution results of the programs that avoid illegal memory accesses by error detection and re-execution.

A. Experimental setup

The GPU used in the experiments is NVIDIA RTX A4000. NVBitFI [13] was used for the fault injection experiments. NVBitFI consists of two programs: a profiler and an injector. A profiler is a tool for counting the number of instructions in a dynamic sequence of CUDA programs. The injector uses the counted instructions to perform fault injection. The injector injects a fault into the destination register of the specified instruction. NVBitFI specifies a predefined instruction group and fault model at runtime. The instruction to inject a fault is selected from the instructions belonging to the instruction group. Since the random number threshold is determined based on the percentage of instructions, the percentage of selected instructions becomes equal to the percentage of executed instructions after many fault injections. Four failure modeling methods are provided: single-bit bit inversion, multiple-bit bit inversion, fixed at 1, and fixed at 0. We injected single-bit bit inversion.

Fault injection experiments using NVBitFI were performed on programs with and without the proposed detection method. The programs used for the experiments were as follows. The total number of instructions and the percentage of each instruction for the programs with and without the proposed method are listed in Table II. The programs that do not implement the proposed method start with “n”.

- Matrix multiplication (MxM): This program requires the most frequent memory accesses among the prepared programs. Therefore, it has the most load instructions in Table II. Floating-point instructions are also more frequent than in other programs.
- Vector addition (vectorAdd): Besides addressing computation, minimal numerical computation is included. Unlike other programs, vector addition is one-dimensional data. Then, the index calculation only uses thread IDs and thus requires the fewest number of instructions for address calculation.
- Matrix transposition (Transpose): This program only loads matrix elements (n, m) and stores them in the resulting matrix element (m, n). Then, it does not contain any arithmetic operations except address computation. Table II shows that the program consists of only IMAD instructions for address calculation, load/store instructions (LDG, LEA, ULDC, STG), and move instructions (MOV, S2R).

B. DUE reduction per one fault injection

We performed approximately 200,000 fault injection experiments using NVBitFI for each program. The results are shown in Table III. The errors detected and recalculated are counted as masks.

The results indicate that the proposed method effectively reduces the number of DUEs for matrix multiplication, vector addition, and matrix transposition to 58.6%, 12.5%, and 11.9%, respectively. It is noteworthy that the DUE reduction is more significant in vector addition and matrix transposition compared to matrix multiplication. In the case of matrix multiplication, the instructions that resulted in DUEs due to fault injection were IMAD and VOTEU instructions. The inclusion of VOTEU instructions through the proposed method accounted for 11.9% of the DUEs in the matrix multiplication experiment, which explains the smaller reduction in DUEs for that particular program. On the other hand, for vector addition and matrix transposition, only IMAD instructions contributed to the occurrence of DUEs.

Authorized licensed use limited to: Kyoto University. Downloaded on September 12,2023 at 19:12:36 UTC from IEEE Xplore. Restrictions apply.

TABLE II: Numbers of instructions and proportions of individual instructions [%]. Programs starting with "n" corresponds to normal programs without the proposed method.

Program	MxM	nMxM	vectorAdd	nvectorAdd	Transpose	nTranpose
#instructions	451,584	114,688	26,624	14,336	49,152	16,384
FFMA	7.26	28.57	3.84	7.14	0.0	0.0
IADD3	24.72	0.89	15.38	28.75	4.17	0.0
IMAD	21.32	4.46	23.08	0.0	27.08	25
LEA	0.00	0.0	0.0	0.0	0.0	12.5
SHF	0.00	0.89	0.0	0.0	0.0	0.0
MOV	0.00	1.79	0.0	14.29	0.0	12.5
ISETP	4.99	0.0	7.69	0.0	8.33	0.0
LOP3	4.08	0.0	3.85	0.0	6.25	0.0
SEL	2.04	0.0	3.85	0.0	4.17	0.0
SHFL	4.08	0.0	7.69	0.0	8.33	0.0
R2UR	2.04	0.0	0.0	0.0	4.17	0.0
LDG	14.51	57.14	7.69	14.29	2.08	6.25
STG	0.23	0.89	3.85	7.14	2.08	6.25
MATCH	2.04	0.0	3.85	0.0	4.17	0.0
ULDC	0.23	0.89	3.85	7.14	2.08	6.25
BRA	4.76	0.0	0.0	0.0	6.25	0.0
EXIT	0.23	0.89	3.85	7.14	2.08	6.25
BSSY	0.23	0.0	0.0	0.0	2.08	0.0
BSYNC	0.23	0.0	0.0	0.0	0.0	0.0
CALL	0.07	0.0	0.0	0.0	0.0	0.0
S2R	0.91	3.57	7.69	14.29	8.33	25
VOTE	4.08	0.0	3.85	0.0	6.25	0.0
VOTEU	2.04	0.0	0.0	0.0	2.08	0.0

TABLE III: Result of fault injection.

Programs	# of executions	Mask	SDC	DUE	Relative DUE ratio [%]
MxM	210,000	151,489	52,121	6,390	58.6
nMxM	199,999	26,066	163,553	10,380	100
vectorAdd	209,989	177,725	25,380	6,884	12.5
nvectorAdd	199,987	32,435	105,307	62,245	100
Transpose	199,998	175,866	12,770	11,362	11.9
nTranspose	199,975	17,420	86,704	95,851	100

TABLE IV: Execution time of kernel function.

	Runtime[us]			
Illegal access detection	Not applied	Applied		Increase
Overhead reduction		Not applied	Applied	ratio
Matrix multiplication	9.82	37.70	13.86	1.41
Vector addition	5.60	6.37	5.92	1.06
Matrix transpose	8.12	8.73	8.45	1.08

C. Runtime overhead

Next, we evaluate the impact of the proposed method on the execution time of the programs. The execution times of kernel functions with and without the proposed method were measured using the `nv-nsight-cu-cli` profiler. We also examined the execution time when the overhead reduction technique discussed in Section IV-B was not applied. The results, obtained from the `gpu__time_duration.sum` metric, are presented in Table IV.

The proposed method resulted in a 1.41x increase in execution time for matrix multiplication, a 1.06x increase for vector addition, and a 1.08x increase for matrix transposition. The larger overhead in the matrix multiplication program is due to the higher number of comparisons compared to vector addition and matrix transposition. Since more comparisons require frequent exchange and comparison of address values, it leads to a longer runtime.

We next evaluate the overhead reduction introduced in Section IV-B. The runtime of matrix multiplication is reduced by 63.2%, while those of vector addition and matrix transposition are reduced by 7.1% and 3.2%, respectively. Vector addition and matrix transposition require fewer memory accesses per thread, three for vector addition and two for matrix transposition. When the overhead is reduced, both comparisons are performed only once, resulting in a small reduction in the number of comparisons. In matrix multiplication, on the other hand, the number of comparisons per thread is 65 for a 32×32 matrix without overhead reduction. When the overhead is reduced, the number of comparisons is reduced to 9, which is more effective in reducing the execution time compared to the vector addition and matrix transposition cases.

TABLE V: Overall DUE reduction per program execution.

Program	DUE reduction per program execution [%]
Matrix multiplication	17.3
Vector addition	86.8
Matrix transpose	87.1

D. Overall DUE reduction considering runtime overhead

Soft error occurrences are temporally probabilistic, meaning that longer execution times increase the likelihood of encountering soft errors during program execution. To further analyze this, we calculated the DUE rate per program execution and examined the DUE reduction rate.

In the case of matrix multiplication, the execution time is 1.41 times longer, and the DUE ratio per fault injection is 0.586 times smaller, resulting in a DUE ratio per execution that is 0.826 times smaller. This indicates that applying the proposed method reduces DUE by 16.8%. Similarly, for vector addition and matrix transposition, the DUE ratios per execution are 0.133 and 0.128 times, respectively. This corresponds to DUE reductions of 86.8% and 87.1%, respectively.

VI. Conclusion

We proposed a method to avoid DUE caused by incorrect addresses using inter-thread communication for error detection and kernel reruns. We applied the proposed method to three programs and conducted software-level fault injection experiments and evaluation of execution time. The fault injection experiments demonstrated that the proposed method successfully reduced the occurrence of DUEs in all programs. Additionally, we observed that the proposed method increased the program execution time by 6% to 41%. Furthermore, we evaluated the reduction in DUE rate per program execution and found that the proposed method can achieve up to 87.1% reduction in DUEs.

Acknowledgement

This work is supported by the Grant-in-Aid for Scientific Research (S) from Japan Society for the Promotion of Science (JSPS) under Grant JP19H05664 and by JST CREST Grant Number JPMJCR19K5, Japan.

Authorized licensed use limited to: Kyoto University. Downloaded on September 12,2023 at 19:12:36 UTC from IEEE Xplore. Restrictions apply.

REFERENCES

[1] Renesas, “Surround view,” 2023, https://www.renesas.com/us/en/application/automotive/adasautonomous/surround-view.

[2] J. Michalakes and M. Vachharajani, “Gpu acceleration of numerical weather prediction,” *Parallel Processing Letters*, vol. 18, no. 04, pp. 531–548, 2008.

[3] M. A. Neggaz, I. Alouani, P. R. Lorenzo, and S. Niar, “A reliability study on cnns for critical embedded systems,” in *2018 IEEE 36th International Conference on Computer Design (ICCD)*. IEEE, 2018, pp. 476–479.

[4] J. Tan, N. Goswami, T. Li, and X. Fu, “Analyzing soft-error vulnerability on gpgpu microarchitecture,” in *2011 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 2011, pp. 226–235.

[5] A. Lotfi, S. Hukerikar, K. Balasubramanian, P. Racunas, N. Saxena, R. Bramley, and Y. Huang, “Resiliency of automotive object detection networks on gpu architectures,” in *2019 IEEE International Test Conference (ITC)*. IEEE, 2019, pp. 1–9.

[6] F. F. dos Santos, S. Malde, C. Cazzaniga, C. Frost, L. Carro, and P. Rech, “Experimental findings on the sources of detected unrecoverable errors in gpus,” *IEEE Transactions on Nuclear Science*, vol. 69, no. 3, pp. 436–443, 2022.

[7] M. M. Goncalves, I. P. Lamb, P. Rech, R. M. Brum, and J. R. Azambuja, “Improving selective fault tolerance in gpu register files by relaxing application accuracy,” *IEEE Transactions on Nuclear Science*, vol. 67, no. 7, pp. 1573–1580, 2020.

[8] K.-H. Huang and J. A. Abraham, “Algorithm-based fault tolerance for matrix operations,” *IEEE transactions on computers*, vol. 100, no. 6, pp. 518–528, 1984.

[9] F. F. dos Santos, M. Brandalero, M. B. Sullivan, P. M. Basso, M. Hübner, L. Carro, and P. Rech, “Reduced precision dwc: An efficient hardening strategy for mixed-precision architectures,” *IEEE Transactions on Computers*, vol. 71, no. 3, pp. 573–586, 2021.

[10] L.-H. Hoang, M. A. Hanif, and M. Shafique, “Ft-clipact: Resilience analysis of deep neural networks and improving their fault tolerance using clipped activation,” in *2020 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2020, pp. 1241–1246.

[11] H. Itsuji, T. Uezono, T. Toba, K. Ito, and M. Hashimoto, “Concurrent detection of failures in GPU control logic for reliable parallel computing,” in *2020 IEEE International Test Conference (ITC)*, 2020, pp. 1–5.

[12] M. Rebaudengo, M. S. Reorda, M. Torchiano, and M. Violante, “Soft-error detection through software fault-tolerance techniques,” in *Proceedings 1999 IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems (EFT’99)*. IEEE, 1999, pp. 210–218.

[13] T. Tsai, S. K. S. Hari, M. Sullivan, O. Villa, and S. W. Keckler, “Nvbitfi: Dynamic fault injection for gpus,” in *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2021, pp. 284–291.

Authorized licensed use limited to: Kyoto University. Downloaded on September 12,2023 at 19:12:36 UTC from IEEE Xplore. Restrictions apply.