The Journal of Supercomputing
https://doi.org/10.1007/s11227-021-04026-6

Regional soft error vulnerability and error propagation analysis for GPGPU applications

Işıl Öz[1] · Ömer Faruk Karadaş[2]

Accepted: 13 August 2021

Abstract
The wide use of GPUs for general-purpose computations as well as graphics programs makes soft errors a critical concern. Evaluating the soft error vulnerability of GPGPU programs and employing efficient fault tolerance techniques for more reliable execution become more important. Protecting only the most error-sensitive program regions maintains an acceptable reliability level by eliminating the large performance overheads due to redundant operations. Therefore, fine-grained regional soft error vulnerability analysis is crucial for the systems targeting both performance and reliability. In this work, we present a regional fault injection framework and perform a detailed error propagation analysis to evaluate the soft error vulnerability of GPGPU applications. We evaluate both intra-kernel and inter-kernel vulnerabilities for a set of programs and quantify the severity of the data corruptions by considering metrics other than SDC rates. Our experimental study demonstrates that the code regions inside GPGPU programs exhibit different characteristics in terms of soft error vulnerability and the soft errors corrupting the variables propagate into the program output in several ways. We present the potential impact of our analysis by discussing the usage scenarios after we compile our observations acquired from our empirical work.

Keywords Soft error reliability · GPGPU programs · Fault injection

✉ Işıl Öz
isiloz@iyte.edu.tr

Ömer Faruk Karadaş
omerkaradas@std.iyte.edu.tr

[1] Computer Engineering Department, Izmir Institute of Technology, Izmir, Turkey

[2] Electrical Electronics Engineering Department, Izmir Institute of Technology, Izmir, Turkey

1 Introduction

Due to the reduction in transistor sizes and the larger system frequencies, the soft error rates in computer systems have been increasing [29, 33, 44]. While the GPUs have been introduced as an accelerator for graphics applications, their high-scale performance improvement with many cores has induced their widespread use in general-purpose computations (GPGPU) [3, 35]. Especially, the applications requiring big data processing utilize a large number of computational resources on GPU systems. However, with many execution cores and complex memory structures, GPUs exhibit high vulnerability to soft errors. Since the GPGPU applications do not tolerate faults as in the graphics applications, which are inherently fault-tolerant, both hardware and software approaches have been proposed to deal with the impact of soft errors in GPU architectures [12, 22, 24, 30, 47].

Redundancy, as a fault tolerance technique to deal with hardware errors, is the replication of hardware and/or software components of a system by targeting to increase reliability [40, 41]. In the software redundancy scheme, the target program code is replicated at the instruction level and the results of the duplicate instructions are compared for error detection.

Since the code redundancy-based fault tolerance techniques require executing the same code multiple times with additional comparison computations, they cause serious performance degradation in the target applications. Therefore, instead of full redundancy, the replication of the most critical and highly vulnerable parts of the target application maintains more performance as well as acceptable reliability level [5, 7, 15, 20, 36]. Hence, evaluating regional soft error vulnerability becomes more critical to quantify fine-grained reliability characteristics.

In this work, we present a fault injection framework for evaluating the regional soft error vulnerability of GPGPU applications and propose a methodology to analyze both intra-kernel and inter-kernel vulnerabilities. Our debugger-based fault injection framework provides source code-level regional analysis of the target applications. As a fault injection tool for GPGPU soft error vulnerability analysis, GPU-Qin [13, 14], which has been implemented in cuda-gdb [2], reports systematically silent data corruption (SDC) and application crash conditions of GPGPU programs. Our fault injection tool, which is based on cuda-gdb [2], provides a general framework to evaluate the regional vulnerabilities of GPGPU programs by injecting faults into specified source code lines. Our fault injector focuses on the specific code portions (the code in the specified line or between the specified lines) and targets the high-level code sections instead of the low-level SASS instructions. To the best of our knowledge, this is the first work that analyzes intra-kernel and inter-kernel vulnerabilities by presenting error propagation through data structures in the program. Furthermore, we perform rigorous fault injection experiments for a set of applications and deep dive into the details of the SDC evaluations by considering the error propagation through the program data. Our main contributions are as follows:

- We present a regional fault injection tool for GPGPU applications, which injects faults into specified code regions in a target program. Our debugger-based tool

provides a general framework to evaluate the target application for the most vulnerable code regions by considering source code lines.
- We perform rigorous fault injection experiments for both intra-kernel and inter-kernel vulnerability evaluation, additionally, we utilize output corruption rate and error metrics to evaluate the data corruption criticality for the silent data corruption (SDC) cases.
- We extend our analysis by tracking program data consumed in the target application and observe the error propagation behavior for high-level data structures.
- Our experimental work reveals that the vulnerability of code regions in GPGPU programs exhibits different characteristics and the soft errors corrupting the program variables propagate into the program output in several ways.
- We present our observations based on the experimental results and potential usage scenarios of our regional soft error vulnerability analysis. Our regional analysis can facilitate proposing selective redundant execution and data protection techniques for GPGPU applications. Additionally, fault prediction frameworks can utilize a larger amount of data by considering the code regions as distinct data points in their machine learning models, and they can achieve potentially more accurate predictions for the target programs' soft error vulnerability.

The remainder of this paper is organized as follows: Sect. 2 presents some background on soft error vulnerability evaluation. We explain our regional soft error vulnerability analysis framework in Sect. 3. Then, the experimental results are outlined in Sect. 4. Section 5 presents a detailed discussion on our observations and the potential usage scenarios of our regional vulnerability analysis. Section 6 presents the related work on GPU vulnerability evaluation methods. Finally, in Sect. 7, we summarize the work with some conclusive remarks and future research directions (Table 1).±

2 Background

2.1 Soft errors in GPGPUs

In this work, we consider soft errors which are induced from transient hardware faults. Soft errors are failures caused by particle strikes including high-energy

Table 1 Table of notations and definitions

Notation	Definition
GPGPU	General-purpose computing on graphics processing units
CUDA	Compute unified device architecture
SDC	Silent data corruption
MAE	Mean absolute error

neutrons, produced by the interaction of cosmic rays within the terrestrial atmosphere, and alpha particles that are emitted by the decay of radioactive impurities used in chip packaging [33]. They might corrupt the program data or crash the program execution. GPGPUs with many cores in a single chip are vulnerable to soft errors and the increasing soft error rate becomes an obstacle to future GPGPU generations by preventing them from decreasing in size or causing erroneous executions [8, 34].

Soft errors do not result from a failure in the circuitry, but due to an external factor causing the data in the memory locations to be modified. In our analysis, we evaluate single-bit errors in the GPU register file by assuming that other GPU memory locations are protected. We also do not consider the errors in any storage structure of the host CPU. We assume that the data is copied into GPU global memory safely before the kernel execution.

2.2 Evaluating soft error vulnerability

The most prominent way to quantify soft error resilience of programs is to perform fault injection experiments [9, 31]. A fault injection experiment starts with specifying fault location and injection time, then introduces the fault at this point during the execution of the program. Then it examines the program output by comparing the expected value with the produced value to determine the effect of the injected fault. In the evaluation of soft error vulnerability of the programs, the outcome of fault injection experiments can be one of the following cases:

- **Correct Execution (Masked)** The application ends successfully and produces the expected output.
- **Silent Data Corruption (SDC)** The application completes the execution, but the output differs from the expected result.
- **Hang** The application continues its execution longer than a predetermined maximum time and does not produce any output during this time.
- **Crash** The application ends with an error code.

Our fault injection tool tracks the execution of the target application after flipping the bit in the specified register and reports the fault behavior by comparing the produced result with the golden output. We focus on the SDC cases due to their importance in vulnerability analysis. Moreover, we utilize some basic metrics to evaluate the criticality of data corruptions as given in Sect. 3.4.

3 Regional soft error vulnerability analysis

3.1 Motivation

All instructions inside a GPU kernel affect the output either directly or indirectly. For instance, the example code below presents a simple kernel function in an image processing application. While the row and column numbers are decided in the first two lines of the kernel, the statement in the fourth line updates the target pixel value. In this simple code snippet, the miscalculation of the row or column value results in missing the calculation of the target pixel or crash due to the array index out of bounds error. However, if the computation in the fourth line fails, it causes a more critical effect in the final result by corrupting the value.

```
__global__ void PictureKernel (float* d_Pin, float* d_Pout, int
    height, int width){
1:  int Row = blockIdx.y * blockDim.y + threadIdx.y;
2:  int Col = blockIdx.x * blockDim.x + threadIdx.x;
3:  if((Row < height) && (Col < Width)){
4:      d_Pout [Row * width + Col] = 2.0 * d_Pin[Row * width + Col
    ];
5:  }
}
```

As an example to demonstrate the effect of the faults on different code regions, this simple kernel code provides hints about more complicated kernel functions or programs with many kernels. Especially, in the GPU programs working with many threads, the errors in different threads, being responsible for either similar or different computations, may affect the overall computation in different ways.

The software redundancy schemes for soft error fault tolerance cause serious performance degradation in the application execution. Therefore, partial redundancy techniques, which are based on the replication of the most critical code region, decrease the performance cost significantly by eliminating the redundant execution of the non-critical codes in terms of soft error vulnerability [5, 36]. Specifically, for GPU programs, instead of replicating all kernel functions in the program or all instructions in a kernel function, the redundant execution of the most vulnerable code region (s) becomes more effective for the program performance. Moreover, the partial redundancy-based fault tolerance becomes more efficient by considering resource constraints in GPU devices such as the number of registers per thread or the shared memory space per thread block.

3.2 Regional fault injection tool

Since the existing fault injection tools targeting GPU programs lack regional fault analysis [13, 14, 16, 19, 45], we design and implement a regional fault injection tool for our purpose. While the general-purpose fault injectors target random fault injection points, our tool introduces faults in a specific code portion (source code lines) and examines the execution of the program after injection time. In this way, the vulnerability of the specified code can be obtained. We build our fault injector based on CUDA GDB debugging tool, cuda-gdb [2], which tracks and controls the CUDA applications externally. We implement our methodology via Python scripts by using gdb module.

Figure 1 presents the flow of our regional fault injection tool which consists of the following phases:

- **Phase 0: Configuration Setup**
  We present an interface to the fault injection process by defining a set of parameters that can be configured before the execution since the correct configuration is the key to the accurate operation of the injection process. Through this text-based interface, the users can specify the parameter values for their purposes. The main configuration options are as follows:
  - Application-specific information (executable name, arguments, output file to check after injection)
  - Fault injection information (whether during a specific line execution or between specific lines or by satisfying a condition in the code; specific register, specific bit)
  - The phases (among 3-phases) to be performed

  We have a text file in a predefined format (parameter-value pairs) for taking all the possible parameters which are filled with default/example values. The users of our tool should modify the given values based on their preferences.
- **Phase 1: Profiling**
  During the profiling phase, the target CUDA program is executed through the debugger (cuda-gdb) to collect information about the application. The profiling finds the number of blocks and threads that the application is actively

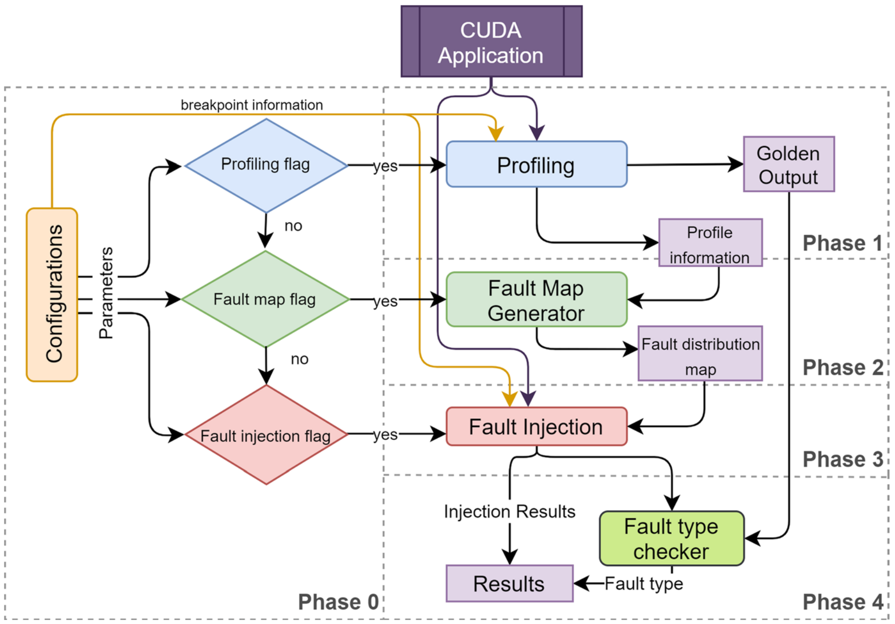


Fig. 1 Flow diagram of our regional fault injection tool

using. To achieve this, the execution stops at the breakpoint specified as the fault injection line in the configuration phase. Furthermore, the corresponding *ptx* instructions being executed for the specified line or between lines are tracked in the program profile phase and passed into the fault generation phase to decide the specific instruction. Another information collected by the profiling is the golden output of the application that is stored to be used in SDC comparison.

- **Phase 2: Fault map generation**

  After the profiling phase, to determine the fault locations and the fault timing, a fault distribution map is generated according to the information given as part of the configuration or obtained at the profiling phase. As mentioned in Sect. 2, the fault type we consider in this study is the corruption of data stored in the register file, i.e., the corruption of the value stored in the target register as a bit flip (the inversion of the target bit either from 0 to 1, or 1 to 0). Specifically, one bit and one register among the ones specified in the configuration are selected as the fault location, and one instruction is selected among the instructions obtained during the profiling phase as the fault injection point. The uniform distribution is used to ensure an even distribution of faults.
- **Phase 3: Fault injection**

  In the fault injection phase, faults are injected during the execution of the specified instruction generated in the fault generation phase, which is one of the instructions belonging to the line specified in the configurations. The application is run through the debugger by setting a breakpoint at the specified line. If a condition is set, the application is continued step by step until that condition is reached, as shown in Fig. 2b. If not determined, the execution proceeds as in Fig. 2a. When stopped at the breakpoint, the execution continues until the target instruction is reached, and the execution is stopped at the specified instruction execution. Then the value stored at the target register at that time is accessed, the target bit of that value is flipped, and stored back into the target register. After the fault injection, the application is continued in its natural flow and the output of the application is collected.
- **Phase 4: Collection of results**

  In the last phase, our tool gives to the user the fault injection results and the resulting output of the SDC cases to determine the amount of data corruption if SDC is observed. When checking the error type, first, the execution time of the injected application is compared with the execution time of the golden run to check the hang of the application. If the application exceeds the maximum time multiplier specified in the configurations, the application is killed, and the fault type is specified as Hang. If the application is finished in the predetermined time, error codes are checked. If the application ends with an error code, it is marked as a Crash. Otherwise, the injected output is compared with the golden output, and if they do not match, it is marked as SDC. If none of these conditions is observed, it is marked as Masked.

  Additionally, as part of our framework, we can collect information to observe error propagation among program variables during the program execution.

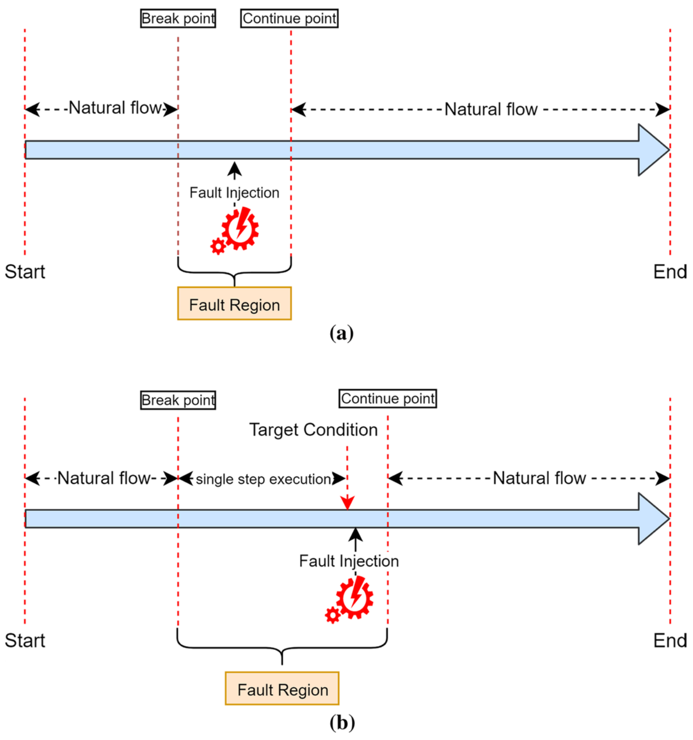


Fig. 2 Flow diagram of our fault injector. **a** Injection at a specific breakpoint. **b** Injection under a condition

Although this feature can be enabled in our fault injection configurations, we do not explain it here and prefer giving the details in Sect. 3.3.

While the given phases can be executed one after the other as one complete flow, the profiling, the fault map generation, and the fault injection phases can operate separately from each other. In this way, the fault injection can start with predefined information instead of a fresh profiling or fault map generation phase.

3.3 Error propagation tracking

Our fault injector tool can be configured to collect values of the data structures and evaluate the error propagation through GPU memory structures. By enabling this feature, it takes memory dumps of the variables at the lines specified by the user. We implement this error propagation tracker in the debugger by stopping the application in the specified line and saving the values of the desired variables without interfering with the source code. In this way, we obtain the propagation of the error corrupting a variable in a certain line of the application toward its output. By examining the error propagation, one can observe the effect of an error on application reliability on the basis of lines and variables.

```
__global__ void mm3_kernel1(float *A, float *B, float *E){
    int j = blockIdx.x * blockDim.x + threadIdx.x;
    int i = blockIdx.y * blockDim.y + threadIdx.y;
    if ((i < size) && (j < size)){
        int k;
        for(k=0; k < size; k++){
            E[i*size+j] += A[i*size+k] * B[k*size+j];
        }
    }
}
__global__ void mm3_kernel2(float *C, float *D, float *F){
    int j = blockIdx.x * blockDim.x + threadIdx.x;
    int i = blockIdx.y * blockDim.y + threadIdx.y;
    if ((i < size) && (j < size)){
        int k;
        for(k=0; k < size; k++){
            F[i*size+j] += C[i*size+k] * D[k*size+j];
        }
    }
}
__global__ void mm3_kernel3(float *E, float *F, float *G){
    int j = blockIdx.x * blockDim.x + threadIdx.x;
    int i = blockIdx.y * blockDim.y + threadIdx.y;
    if ((i < size) && (j < size)){
        int k;
        for(k=0; k < size; k++){
            G[i*size+j] += E[i*size+k] * F[k*size+j];
        }
    }
}
```

We explain our error propagation analysis in a simple example program, 3MM, which is a linear algebra application performing three matrix multiplications in three kernel functions given above, where each kernel computes one multiplication, specifically each thread is responsible for multiply and add operations for one row and column of the input matrices. The program performs the matrix multiplications in the order shown in Fig. 3. Since the values computed in the first two kernels are utilized in the final kernel, we can expect that the number of incorrect elements in the output matrix will be less when the error occurs toward the final stages of the application. In other words, an error that occurred during the calculation of E array (Injection 1 phase) or F array (Injection 2 phase) affects one or two rows completely in the output, while an error during the computation of G array (Injection 3 phase) affects only one or two elements in the output. As part of our error propagation framework, we generate visualizations in order to understand the impact of the errors among the data variables. Figure 4 presents the sample images generated for one SDC case for the fault injection performed during the execution of $mm3\_kernel1$. While the images on the left represent the difference between the expected (golden) and the observed (corrupted by SDC)

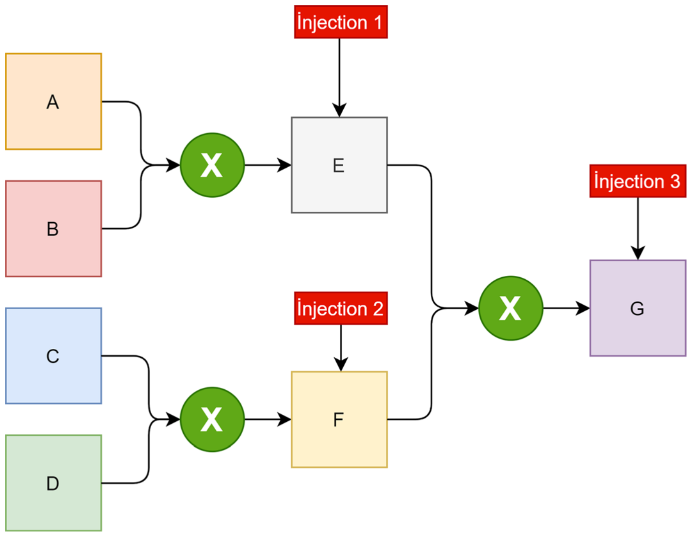


Fig. 3 Flow diagram of 3MM application. Boxes represent matrices and injection points are the locations of the faults injected during the test phase on the application

value for the specified E array element, the images on the right are the same for the G element. We color each array element depending on the magnitude of the corruption, where black represents no difference and white represents the largest difference. The figure specifically demonstrates the propagation of an error corrupted one or two elements of the array E (as shown as small white points in the figure) into the output array G. In the case that one element is miscalculated in the array E, this single erroneous value is propagated into the G array by corrupting one of its rows entirely. Similarly, if the elements from two rows are miscalculated in the array E (due to the corruption of one index variable), two entire rows of the array G are corrupted.

While it is straightforward to see the propagation of the errors by looking at the source code or performing static analysis for simple applications like matrix multiplication, we need to perform empirical fault injection experiments for complex programs. Not only does our fault injector tool work for the GPU programs with simple kernel executions, but it also maintains error propagation analysis for the programs with dynamic behavior that is difficult or not possible to analyze. For example, data-intensive applications processing complex data structures like graphs are increasingly irregular and have input-dependent and unpredictable control flow behavior [48].

Our tool generates both numeric and visual data providing data corruptions for the specified high-level variables inside kernel functions. By this feature, it not only presents the criticality of the errors for each SDC case but also enables the program developers or the application users to track the error propagation among the variables during the program execution.

3.4 Data corruption criticality evaluation

While the silent data corruption rate has been utilized as a soft error vulnerability metric, some corruption might be acceptable or much more critical depending on the application. Therefore, in our analysis, we evaluate the criticality of the data corruption for SDC cases. As mentioned before, our fault injector saves the produced output if the fault injection causes an SDC. By using those outputs, we further utilize several metrics to evaluate the criticality of the corruption cases.

- **Output Corruption Rate**
  For the programs including matrix operations and producing matrices as the output, the number of incorrect elements for SDC conditions can be used as a vulnerability assessment metric instead of raw SDC rates. Since some applications, like image processing programs, can tolerate a number of incorrect matrix elements in their resulting data, we consider output corruption rate as the criticality of the error, which is defined as follows:

$$Output\ corruption\ rate = \frac{number\ of\ incorrect\ matrix\ elements}{matrix\ size}$$

- **Absolute error** For the programs producing one value as the output, the difference between the expected output and the produced output can be used as a vulnerability criticality metric:

$$Absolute\ error = |\ expected\ value - observed\ value\ |$$

- **Mean absolute error**
  For the programs producing several values as the output, the average difference between the expected output and the produced output can also be used as a vulnerability criticality metric. We utilize mean absolute error as the criticality of the error in such applications:

$$Mean\ absolute\ error\ (MAE) = \frac{1}{N}\sum_{1}^{N} |\ expected\ value - observed\ value\ |$$

where N is the number of computed values.
Similarly, some soft errors affecting the output of a classification algorithm could still be acceptable. In fact, output errors could be tolerated as long as the misprediction is not critical for the purpose of the application. As an example, in an object detection framework [42], the error criticality can be evaluated by Precision and Recall metrics by considering Recall more strictly since missing actual pedestrians (identified with lower Recall) may lead to accidents.

Similar metrics to identify the criticality of the data corruption can be defined and evaluated in soft error vulnerability analysis of the programs. Since our target applications produce either matrix data or a single value, we utilize three metrics including output corruption rate, absolute error, and mean squared error in our analysis.

4 Experimental study

4.1 Experimental setup

For inter-kernel and intra-kernel regional vulnerability analysis, we select six CUDA applications from Polybench [17] benchmark suite including ATAX, BICG, COVAR, CORR, FDTD-2D, GRAMSCHM, and an open-source graph coloring application implemented in CUDA [1]. While the details are given in Sect. 4.2, ATAX, BICG, COVAR, CORR, FDTD-2D, GRAMSCHM programs include matrix operations in a number of kernel functions. The graph coloring program is the CUDA implementation of the CJP algorithm given in [39], which is composed of one kernel function that is responsible for checking the colors of vertex neighbors and assigning a color for each node in the given graph. We choose the first set of applications with different simple kernel functions to evaluate inter-kernel analysis, and the other program with one complex kernel function to examine intra-kernel behavior.

We compile our target programs with CUDA 9.0 and run fault injection experiments in an Intel-based workstation with an NVIDIA Quadro P4000 GPU. We use 1000 fault injections per each regional fault analysis by using a statistical approach [27] with the confidence level of 95% and the error margin 3%.

4.2 Experimental results

In this section, we present the details of the benchmark applications to examine the regional soft error vulnerability and our experimental results.

As we discuss in Sect. 3.4, depending on the application output, the evaluation of the errors' impact on the execution may vary. The program developers or the users, who want to perform vulnerability analysis, can utilize our output evaluation feature to understand the criticality of the data corruption. As part of this study, we examine the output corruption rate and the mean absolute error for the programs including matrix operations, while we focus on the absolute error values for the graph coloring program, which calculates one value as its output. Hence, we analyze which code region has affected the output by how much. Since the programs contain different computations and demonstrate different data access patterns, we investigate them one by one by examining the basic functionalities.

ATAX This application computes the multiplication of the matrix transpose and the matrix-vector multiply (A^T times Ax). While the first kernel performs matrix-vector multiplication, the second kernel multiplies the matrix transpose with the result of the first kernel:

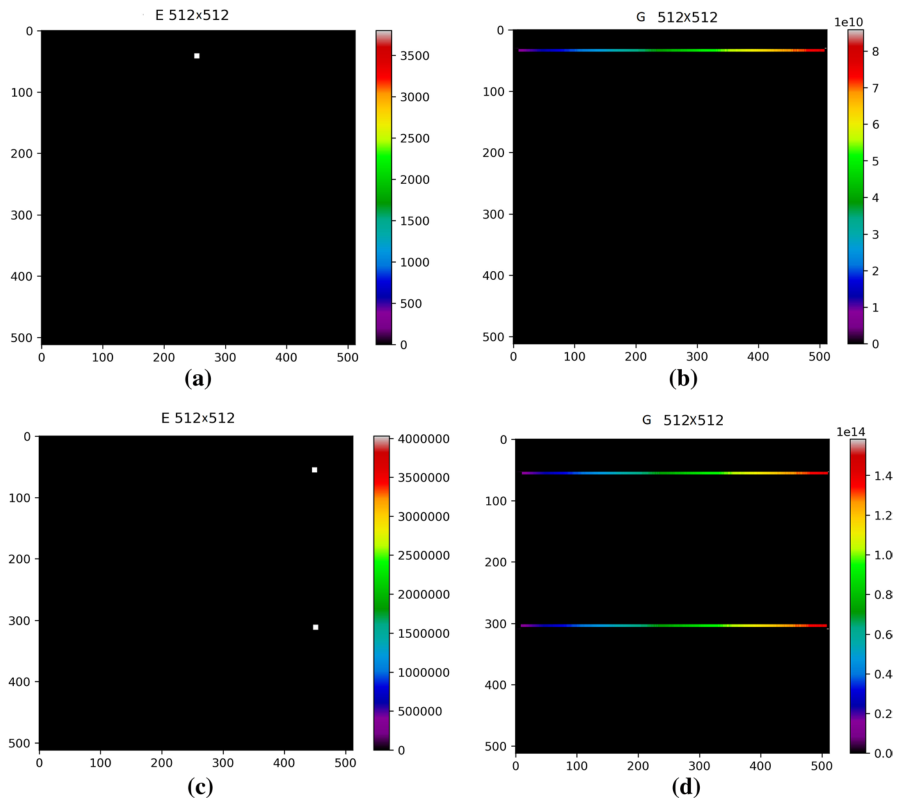


Fig. 4 Visualization of output difference between golden and SDC output

```
__global__ void atax_kernel1(float *A, float *x, float *tmp){
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < size){
        int j;
        for(j=0; j < size; j++){
            tmp[i] += A[i * size + j] * x[j];
        }
    }
}
__global__ void atax_kernel2(float *A, float *y, float *tmp){
    int j = blockIdx.x * blockDim.x + threadIdx.x;
    if (j < size){
        int i;
        for(i=0; i < size; i++){
            y[j] += A[i * size + j] * tmp[i];
        }
    }
}
```

We perform fault injection on two kernel functions (K1 for *atax_kernel*1 and K2 for *atax_kernel*2) and analyze the results. Figure 5a presents the boxplot for the number of the incorrect elements in the silent data corruption cases for each code region. For K2 region, all of the fault injection scenarios ending with SDC result in exactly one element's miscalculation. On the other hand, for K1 region, most of the SDC cases result in the corruption of all elements (i.e., 4096), while the fault affects the fewer elements for a few cases. We can see that a fault that occurred during the execution of the first kernel impacts the final output more seriously, while a fault in the second kernel computation corrupts only one output element in the result array. When one element of the *tmp* array is miscalculated during *atax_kernel*1 execution, all the elements of the *y* array are affected by this data corruption. Depending on the magnitude of the miscalculation, the effect on the output elements may get visible. Specifically, when we examine the case that results in the corruption of the fewer output elements (the circle in the boxplot) by using our error propagation tracker, we see that the difference between the original and the corrupted value (of the *tmp* array element) is the smallest among all SDC cases. Therefore, the difference may be hidden by the larger values contributing to the computation of the target *y* element.

BICG This application is the implementation of the BiCGSTAB (BiConjugate Gradient STABilized method), which is an iterative method for the numerical solution of linear systems. The two kernel functions given as part of the benchmark consist of two independent matrix-vector multiplication operations and produce two different matrix elements as the output. We evaluate fault injection experiments for those kernel functions, specifically we define two code regions per kernel function, the first at array element initialization part, the other at multiplication operation. As a result, we have four different code regions (K1.1, K1.2, K2.1, K2.2) for the application. Figure 5b presents the incorrect number of output elements obtained from the fault injection experiments for each code region. Since all the computations are independent and there is no data reuse in the kernel functions, all SDC cases result in exactly one output element miscalculation.

COVAR This application computes covariance value, which shows statistically how linearly related two variables are. The covariance is defined as the mean of the product of the deviations for *x* and *y* variables:

$$\sigma_{x,y} = \frac{\sum_{i=1}^{N}(x_i - \bar{x})(y_i - \bar{y})}{N - 1}$$

As seen in the formula shown above, covariance computation includes three main operations: (1) the average of the data elements, (2) the deviation of the data elements from the mean, and (3) the summation of the multiplication of those deviations. Polybench implements those three operations in three kernel functions, *mean_kernel*, *reduce_kernel*, *covar_kernel*, where the data elements are processed in parallel CUDA threads.

```
__global__ void mean_kernel(float *mean, float *data){
1:   int j = blockIdx.x * blockDim.x + threadIdx.x + 1;
     ...
2:   for(i = 1; i < (size+1); i++){
3:       mean[j] += data[i * (size+1) + j];
4:   }
     ...
}

__global__ void reduce_kernel(float *mean, float *data){
1:   int j = blockIdx.x * blockDim.x + threadIdx.x + 1;
2:   int i = blockIdx.y * blockDim.y + threadIdx.y + 1;
     ...
3:   data[i * (size+1) + j] -= mean[j];
}

__global__ void covar_kernel(float *symmat, float *data){
1:   int j1 = blockIdx.x * blockDim.x + threadIdx.x + 1;
     ...
2:   for (j2 = j1; j2 < (size+1); j2++){
3:       symmat[j1*(size+1) + j2] = 0.0;
4:       for(i = 1; i < (size+1); i++){
5:           symmat[j1 * (size+1) + j2] += data[i *(size+1) + j1] *
      data[i *(size+1) + j2];
6:       }
7:       symmat[j2 * (size+1) + j1] = symmat[j1 * (size+1) + j2];
     }
}
```

We specify four different regions for fault injections, namely one for the *mean_kernel* (KM), one for the *reduce_kernel* (KR), one for the inner loop of the *covar_kernel* (KC.1), and one for the outer loop of the *covar_kernel* (KC.2). Figure 5c presents the boxplots for the number of incorrect elements. We can see that the variance for the values is smaller in the KM region, while the values differ more substantially for the other regions in different SDC cases. Essentially, data computed in the mean kernel (KM) is utilized in the reduce kernel (KR), which directly results in the corruption of the output data. The small oscillations in the affected number of elements result from the magnitude of the data corruption in the mean kernel (similar to the K1 at ATAX). To further investigate the behavior, we look at the details of the SDC cases we obtain for the fault injection on the mean kernel (KM). When we inject fault during the execution of the statement at line 3, the computed value of the specific *mean* array element is corrupted by some magnitude (difference), and we get unexpected values in some *symmat* output array elements (the number of incorrect elements). Figure 6 presents the number of incorrect *symmat* elements and the difference between the original value and the corrupted value in the *mean* array. We scale the difference value (by dividing all the values by 16) to be able to demonstrate the relationship in the same plot, namely, the *y*-axis in the figure represents the incorrect elements (e.g., 2047) while the difference value is 16 times that value (e.g., approximately 32752). Since we apply the same scaling factor for the regions of the same program, we believe that the comparison among different regions for one program is feasible. In general, we obtain more incorrect elements in the output array if we have larger differences in the faulty mean array. We can see this correlation in the figure with some exceptions, where we have larger incorrect elements (for the SDC instance 22) or smaller difference values (for the instances 39, 40, and 48). We further investigate the reason for those instances. For the first case, where we have a larger number of incorrect elements while the corrupted value does not differ, we observe that two elements of the *mean* array are corrupted instead of one in a single fault injection experiment. The reason is that the index value (*j*) in the target core region is corrupted (e.g., it gets the value 1 instead of 0), and two elements of the *mean* array indexed by the original and modified values (e.g., 1 and 0) are being miscalculated due to the swap of the values of those two elements. Therefore, the impact on the final output gets visible for a larger number of elements. For the latter case, where we have smaller differences resulting in the same number of elements, we can see that the differences in the final output values are also very small while there is the same number of corrupted elements. For the *reduce_kernel*, we encounter more double-element corruptions than the *mean_kernel* due to the index value corruptions, where more variables contribute to the index value calculation including *i*, *j*, and *size* (Line 3 in the *reduce_kernel*). Instead of having modification on the array element, most of the cases result in the index value corruption, and swapping two elements' values does not cause large modifications in the element values. Therefore, the number of incorrect elements is less than those in the *mean_kernel*. On the other hand, depending on both the number of corrupted data array elements and the magnitude of the differences, there is a larger range for the possible number of incorrect elements in the output array. For the regions inside the *covar_kernel*, the median values for the number of incorrect elements are similar for both regions. Since the value inside the inner loop is accumulated by several iterations, the impact gets less visible in the output elements.

CORR This application computes the Pearson's correlation coefficients, which is normalized covariance. The correlation is computed as follows:

$$r_{x,y} = \frac{\sum_{i=1}^{N}(x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_{i=1}^{N}(x_i - \bar{x})^2}\sqrt{\sum_{i=1}^{N}(y_i - \bar{y})^2}}$$

The correlation computation includes four main operations: (1) the average of the data elements, (2) the standard deviation of the data elements, (3) the division of deviation of the data elements from the mean by the standard deviation, and (4) the summation of the multiplication of those deviations. Similar to COVAR, Polybench implements those operations in four separate kernel functions, *mean_kernel*, *std_kernel*, *reduce_kernel*, and *corr_kernel*, where the data elements are processed in parallel CUDA threads. For this application, we perform a set of fault injection experiments similar to COVAR, one for the *mean_kernel* (KM), one for the *std_kernel* (KS), one for the *reduce_kernel* (KR), one for the inner loop of the *corr_kernel* (KC.1), and one for the outer loop of the *corr_kernel* (KC.2). Figure 5d presents the boxplots for the number of the incorrect elements. While the faults in the *mean_kernel* and the *std_kernel* cause similar impacts on the output elements for all SDC cases, the *reduce_kernel* and the *corr_kernel* demonstrate more diverse behavior. While the median values are similar to the regions for COVAR, we can

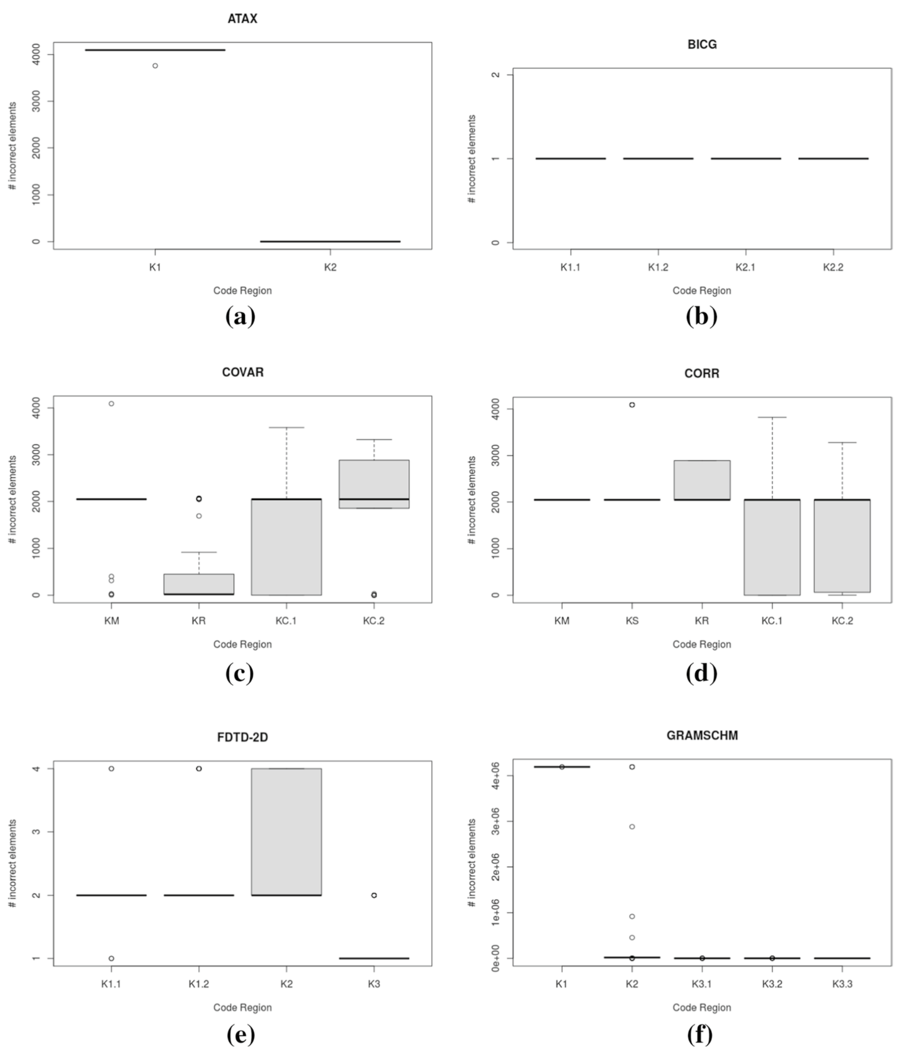


Fig. 5 Boxplot for incorrect elements of inter-kernel functions

observe a few differences. The more incorrect elements in the KR of CORR (than the KR of COVAR) result from the fact that the corrupted value in the kernel is modified by consecutive operations, which increases the impact of the corruption.

FDTD-2D This application is the implementation of the simplified finite-difference time-domain method for 2D data. The implementation consists of the following three kernel functions:

```
__global__ void fdtd_step1_kernel(float *_fict_, float *ex, float
    *ey, float *hz, int t){
1:  int j = blockIdx.x * blockDim.x + threadIdx.x;
2:  int i = blockIdx.y * blockDim.y + threadIdx.y;
3:  if ((i < size) && (j < size)){
4:      if (i == 0){
5:          ey[i*size+j] = _fict_[t];
6:      }
7:      else{
8:          ey[i*size+j] = ey[i*size+j] - 0.5f*(hz[i*size+j] - hz
    [(i-1)*size+j]);
9:      }
10: }
}
__global__ void fdtd_step2_kernel(float *ex, float *ey, float *hz,
     int t){
1:  int j = blockIdx.x * blockDim.x + threadIdx.x;
2:  int i = blockIdx.y * blockDim.y + threadIdx.y;
3:  if ((i < size) && (j < size) && (j > 0)){
4:      ex[i*(size+1)+j] = ex[i*(size+1)+j] - 0.5f*(hz[i*size+j] -
     hz[i*size+(j-1)]);
5:  }
}

__global__ void fdtd_step3_kernel(float *ex, float *ey, float *hz,
     int t){
1:  int j = blockIdx.x * blockDim.x + threadIdx.x;
2:  int i = blockIdx.y * blockDim.y + threadIdx.y;
3:  if ((i < size) && (j < size)){
4:      hz[i*size+j] = hz[i*size+j] - 0.7f*(ex[i*(size+1)+(j+1)] -
     ex[i*(size+1)+j] + ey[(i+1)*size+j] - ey[i*size+j]);
5:  }
}
```

Similar to the previous programs, we select regions inside the kernel functions. While we select two different regions for the first step, Line 5 (K1.1) and Line 8 (K1.2), specifically, we perform fault injection while running Line 4 in the other kernel functions (namely *fdtd_step*2*_kernel* (K2) and *fdtd_step*3*_kernel* (K3)). While the first kernel updates *ey* variable, the second kernel independently modifies the values of the *ex* variable, and finally the last kernel utilizes both variables to obtain the resulting data variable, *hz*. Figure 5e presents the incorrect elements observed in the output array (*hz*), where the faults in three regions ($K1.1$, $K1.2$, $K2$) result in two-element corruption in the final output, since both *ey* and *ex* array elements are utilized by the computation of two different output (*hz*) elements. Depending on the number of modified elements on the target kernels (on either *ey* or *ex*) by the introduced fault or the magnitude of the modification, we can get one or four incorrect elements as well. On the other hand, any fault in the last kernel (K3) directly causes the corruption of the final output array without any error propagation among array structures. Therefore, only one output element is miscalculated as seen from the figure. We have the outlier value 2, which results from the double-element corruption case discussed before.

GRAMSCHM This application represents QR decomposition with modified Gram Schmidt. The implementation consists of the following three kernel functions:

```
__global__ void gramschmidt_kernel1(float *a, float *r, float *q,
    int k){
1:  int tid = blockIdx.x * blockDim.x + threadIdx.x;
2:  if(tid==0){
3:      float nrm = 0.0;
4:      int i;
5:      for (i = 0; i < size; i++){
6:          nrm += a[i * size + k] * a[i * size + k];
7:      }
8:      r[k * size + k] = sqrt(nrm);
9:  }
}

__global__ void gramschmidt_kernel2(float *a, float *r, float *q,
    int k){
1:  int i = blockIdx.x * blockDim.x + threadIdx.x;
2:  if (i < size){
3:      q[i * size + k] = a[i * size + k] / r[k * size + k];
4:  }
}

__global__ void gramschmidt_kernel3(float *a, float *r, float *q,
    int k){
1:  int j = blockIdx.x * blockDim.x + threadIdx.x;
2:  if ((j > k) && (j < size)){
3:      r[k*size + j] = 0.0;
4:      int i;
5:      for (i = 0; i < size; i++){
6:          r[k*size + j] += q[i*size + k] * a[i*size + j];
7:      }
8:      for (i = 0; i < size; i++){
9:          a[i*size + j] -= q[i*size + k] * r[k*size + j];
10:     }
11: }
}
```

We perform fault injections on five different regions, specifically, Line 6 in the *gramschmidt_kernel*1 (K1), Line 3 in the *gramschmidt_kernel*2 (K2), Line 3, Line 6, and Line 9 in the *gramschmidt_kernel*3 (K3.1, K3.2, K3.3). Figure 5f presents the number of incorrect elements in the output array. Since the first kernel is executed by only one thread (with $tid = 0$) and computes the *nrm* value, which is utilized by the computation of all output elements, a fault during the *gramschmidt_kernel*1 (K1) execution corrupts all output data elements (4192256 in total). On the other hand, a fault in the computation of the *q* array inside the *gramschmidt_kernel*2 (K2) results in a different number of output array elements. Finally, due to lack of error propagation effect (namely, direct impact on the output array), the faults for the regions inside the *gramschmidt_kernel*3 (given as K3.1, K3.2, and K3.3 in Fig. 7, as a closer visualization to observe the details)

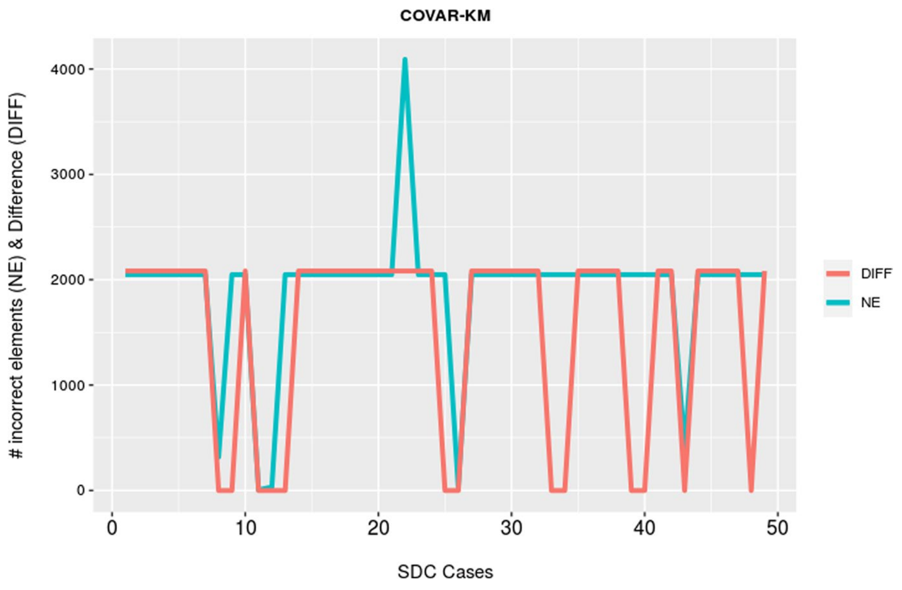


Fig. 6 SDC cases of fault injection on COVAR mean kernel

cause the fewer incorrect elements. Specifically, the K3.3 region tends to have fewer incorrect elements due to its final touch in the output array (but still multiple elements due to loop execution).

For the benchmark applications producing multiple values (as the output array elements), we further analyze the fault outcomes by measuring mean absolute error values. Since our earlier results present the number of incorrect elements for our target programs, we still focus on the array elements computed differently than the expected value. Therefore, we take the average values over those incorrectly computed elements, namely, *N* in the formula given in Sect. 3.4 is taken as the number of incorrect elements for the specific SDC cases. We also exclude some outlier values to represent the data more accurately.

Figures 8, 9, and 10 present the histogram of the mean absolute error values for SDC cases belonging to each fault injection region in our target programs. While the *x*-axis represents the mean absolute value intervals, the *y*-axis represents the number of SDC cases resulting in errors between those values. For instance, in Fig. 8a, for ATAX, we have six fault injection experiments ended with SDC case, where the mean absolute error value obtained from each case is between 0 and 0.02xE+14. We can observe different values among the programs having distinct computations in their kernel functions. For ATAX, where the K2 performs a computation affecting the final result directly and the K1 computes the intermediate values utilized by the K2, while an error in the K1 propagates to the final result by corrupting multiple elements (see Figure 5a), it is more probable that an error in the K2 corrupts the final array elements by a larger magnitude (see Figure 8b). On the other hand, any error during the execution of the independent kernels in BICG results in similar error rates, as we see the similar behavior for the incorrect number of elements. For

COVAR, since the KR is the kernel just negating the mean values computed by the KM, its effect is the lowest among the other COVAR kernels (see Figure 9a). The effect of the other kernels is larger and does not differ significantly among themselves. We can see a similar behavior for the error distributions of the CORR kernels. The trend in the magnitude of the error, which demonstrates an increase in the last kernel of the programs, is not valid for the FDTD-2D (see Figures 10a, 10b, 10c, and 10b). Since the computation performed in the last kernel, K3, aggregates many different values (loaded from the memory locations to the separate registers), the impact of any error hitting one of the registers would not be so high in terms of the corruption value. Moreover, the error magnitudes (difference between the expected and the observed value) are not so large due to the individual computations other than cumulative (summation) operations similar to CORR or COVAR programs. Therefore, we do not see significant difference among the mean absolute errors for different kernels. We also see smaller error values for GRAMSCHM, where we have multiple operations like negation or division as well as addition, which makes the final values not so large.

CJP This application is the CUDA implementation of the Counting-based Jones-Plassmann (CJP) graph coloring algorithm [39]. The kernel function performs coloring of the neighbors of each vertex, where each thread works on a separate neighbor set of the target vertex. The computation inside the kernel includes bit-level operations and each thread keeps a 256-bit variable of which each bit may be utilized during the computation. To evaluate intra-kernel vulnerability, we choose this program since it represents a heavy-kernel CUDA program with several computations inside its kernel. In our fault injection experiments, we pick a graph instance from the University of Florida sparse matrix collection [11].

We select 19 different code regions for our fault injection analysis and conduct fault injection experiments to get the regional soft error vulnerabilities. While the first six code regions are responsible for the initialization and the distribution of the work, the last nine code regions perform global update operations.

Figure 11a presents the number of incorrect computations observed among the fault injection instances. As the CJP program produces one value as the output (i.e., the minimum number of colors to color the graph), we simply compare the resulting output with the expected one and mark the individual fault injection case as an incorrect computation/silent data corruption (SDC) if they do not match. For instance, out of 1000 fault injections, we get 25 incorrect values (the minimum number of colors to color the graph) in region 19. When we look at the results in detail, we can see that the number of incorrect results is higher in the first and the last regions. Since the initialization and work distribution is performed by one thread in the first part, namely the regions between 1 and 6, it is more probable that the incorrect work assignment affects the whole computation. Moreover, since the regions between 11 and 19 are responsible for the update of the global result after the completion of thread-local operations, the fault that occurred during this time can affect the result directly. In the other regions, namely the regions between 7 and 10, since the individual threads are working on their local data and the corrupted values computed by bit-level operations can become ineffective by the other threads or the thread itself, we consider that fault injections on those code regions do not affect the final result. For instance, assume that one thread stores the colors of its neighbors in an 8-bit variable (four 64-bit registers in the application). According to the CJP algorithm, the bit locations corresponding to its neighbors' colors need to be 1 while the others are set to 0. As an example, if there are 2 neighbors to be processed by the thread and their colors are 1 and 3, the 8-bit variable needs to contain 00000101 bit sequence. The thread should select color 2, as the minimum color other than the neighbors' colors. With an error flipping the third bit of the 8-bit variable (convert it to 0 from 1), we may have 00000001 value instead of 00000101. However, this does not change the decision of the thread, because the minimum unassigned color is still 2, and the erroneous local variable does not impact the result at all. We can see that CJP, which is working with bit-level operations, is inherently fault-tolerant for some part of its calculation. On the other hand, if an error hits the second bit and converts it to 1 from 0, we will have 00000111, the minimum unassigned color becomes 4, not 2 anymore. Therefore, the decision of the erroneous thread will potentially affect the outcome. The fault tolerance is not valid for this specific error. Although there is a possibility of an SDC in the final output, which depends on the bit-wise operations, it is also possible to tolerate the faults even if the error hits a utilized register.

We perform a sensitivity analysis to understand how the CJP threads utilize the registers during their execution. Since the multiple threads store the colors in multiple bits (possibly in multiple registers), we think that injecting the faults among different registers does not change the SDC values. Due to the heavy utilization of all registers, we would get similar corrupted results by corrupting the valuable data in any of the registers. While Fig. 11a presents the SDC numbers for fault injection experiments that select the injection point among 64 registers, Fig. 11b demonstrates the results for the experiments with 128 registers. Contrary to our expectations, we observe fewer SDC cases in the latter case. After analyzing the results in detail and consider our observations about the results before, we realize that the SDC cases result from the faults in the code regions executed by a single thread, specifically during initialization or modification of the global data, where the register utilization is not high. Therefore, we encounter fewer SDC cases for the setting with more register alternatives (128 registers) due to the lower probability of fault impact in a larger fault space.

We also collect absolute error values, obtained by taking the difference of the expected value and the observed value, for each SDC case in our target regions. Since CJP computes one output value as its result, we utilize the absolute error metric to evaluate the criticality of the data corruptions. We observe four different values in the SDC cases, where we encounter the value 257 for most of the cases. Since the kernel function needs to continue at the next phase if the first phase cannot set a color smaller than 256, an error hitting the kernel computations during the first phase may result in proceeding the next phase erroneously and end with 257 different colors to color the vertices of the graph. Figure 12 demonstrates the distribution of the absolute error values for the SDC cases for each region with fault injection performed at 64 and 128 registers. Similar to Fig. 11, the x-axis represents the different code regions and the y-axis represents the number of SDC cases, where the different colors represent the different absolute error values. For instance, as shown in Fig. 12a, for R1, we have 239 as absolute error value for all 22 SDC cases, while we have 47 for 11 SDC cases and 239 for the remaining 26 SDC cases for R15. Specifically, the value 239 is obtained as the absolute error by subtracting the erroneous 257 value (as explained earlier) from the expected 18 value (our golden output for the CJP execution). We have the value 239 for most of the cases for all regions, while there are other values for a few cases. Especially, for the regions closer to the last part of the computation, we can observe other values as the output of the program since the injected errors corrupt the bit-wise computations in different bit locations.

5 Discussion

In this section, we present our observations and potential usage scenarios of our region-based soft error vulnerability analysis.

5.1 Observations

We make the following observations in our study.

Observation 1: Both inter-kernel and intra-kernel code regions in GPGPU programs exhibit different soft error vulnerability.

By performing regional fault injections either on different kernel functions or different regions inside one kernel function, we evaluate diverse soft error effects for different code regions. Depending on the operation performed or the data being utilized by the target code, the output of the program is affected in various ways. Consequently, we see that black-box, coarse-grained fault injection analysis does not provide details about the vulnerability of the GPGPU programs and we need to perform fine-grained regional analysis including data corruption criticality evaluation to be able to take cautions (as we discuss in Sect. 5.2) about that level of the vulnerabilities. Even for the same code region, the effect of a soft error may appear in different ways (e.g., either the corruption of the variable holding the value and having a single incorrect output element, or the corruption of the index of the array element and having two incorrect output elements). Hence, we consider that finer-grained vulnerability analysis maintains further details for fault-tolerant computing.

Observation 2: The code regions affect the other code regions (in the same kernel or in the other kernel functions) or data structures, as well as the final output in different ways depending on the flow and the data utilization of the GPGPU application.

Through our empirical error propagation analysis, we observe the spread of an error among the application code and data. Depending on the characteristics of the target program and its memory access pattern, a fault might manifest in different parts of the application data. The criticality and the speed of this propagation also depend on the data accessed and the operations performed during the program execution. For instance, a single error hitting any data element that would be utilized for the computation of several output data elements corrupts the several output elements. Furthermore, multi-bit errors, which are not evaluated as part of this paper, but targeted as future work, may result in more serious effects on the program output by corrupting a larger amount of program data. Since the programs, especially GPGPU programs dealing with a large amount of data, exhibit diverse data dependency relations; it gains more importance to evaluate the error propagation for GPGPU program execution [4, 28]. While it is crucial to understand the effects of the soft errors on the final program outcome, tracking the error propagation through program execution provides insights into the vulnerability behavior of the target program and enables us to consider the ways to reduce the risk of spread among the application.

Observation 3: The degree of parallelism in GPGPU programs affects the soft error vulnerability in different ways depending on the work performed or the data utilized by the CUDA threads.

As we analyze as part of our inter-kernel evaluation, the utilization of the local memory structures, like registers, directly affects the soft error vulnerability of the GPGPU programs. From our experimental analysis, we see that it is more probable to have data corruption if we have fewer threads utilizing the available registers. Essentially, the total number of registers utilized at any time depends on the number of threads executing simultaneously. While the number of registers used by one thread limits the parallelism in a CUDA program, the impact also needs to be examined for soft error vulnerability. Analyzing the register usage of one thread in the program may not be adequate without having information about the parallelism in the program, e.g., the number of threads executing in parallel to that thread. When one thread heavily uses the registers (but not all the available registers) to hold its data, but there are not many parallel threads, overall vulnerability may not be high. On the other hand, in an execution where we have many concurrent threads with a few register usage, we may have a larger vulnerability due to higher overall register utilization.

5.2 Potential usage scenarios

Our regional fault analysis tool can be used to evaluate error resilience characteristics of general-purpose GPU applications in a number of contexts. We provide five usage scenarios to show how our tool can be utilized.

5.2.1 Correlating code characteristics and fault behavior

Recently, there have been some works that propose machine-learning techniques to predict soft errors in GPU programs [23, 37]. The proposed prediction frameworks utilize the program characteristics as input features in their ML models, similar to works based on failure prediction for HPC systems [21]. In order to train the model, they collect both program features like memory address instructions, arithmetic instructions, register usage; and fault injection outcomes like SDC, crash rates. Due to the fact that the more data they collect, the more accurate prediction they can obtain; dividing the execution into smaller parts (regions) and collecting data for each part may increase the number of data points for the target ML model. Our regional fault injection framework employs fine-grained fault injection experiments by introducing faults in the target registers for any statement in the target CUDA program. Instead of performing fault injection for the complete program (or individual kernel), the developers can utilize our tool to conduct multiple fault injection experiments for the target program and obtain multiple data points for their ML model input. Potentially, the prediction accuracy of their models increases.

5.2.2 Selective redundant execution

Since the software redundancy techniques [12, 18, 47] introduce performance overheads, the selective redundancy schemes based on the replication of the most error-sensitive parts employ more efficient execution by providing the best coverage in terms of reliability and performance. Our regional fault injection framework can be utilized by the redundant execution techniques to determine the most vulnerable parts of the target program and it can lead to the selective redundancy performed by those methods. By performing fault injection experiments in different parts of the target program, one can decide the criticality of the code regions under observation and prefer to employ redundancy on the most critical parts instead of the full replication.

5.2.3 Partial data protection

Similar to the redundant execution, the fault tolerance techniques based on the data redundancy like ECC or parity induce additional cost and performance overhead. Although protecting especially highly utilized memory structures like registers or shared memory is not preferred mostly, some critical systems may need to employ this level of protection. Instead of all memory structures in a GPU system, protecting

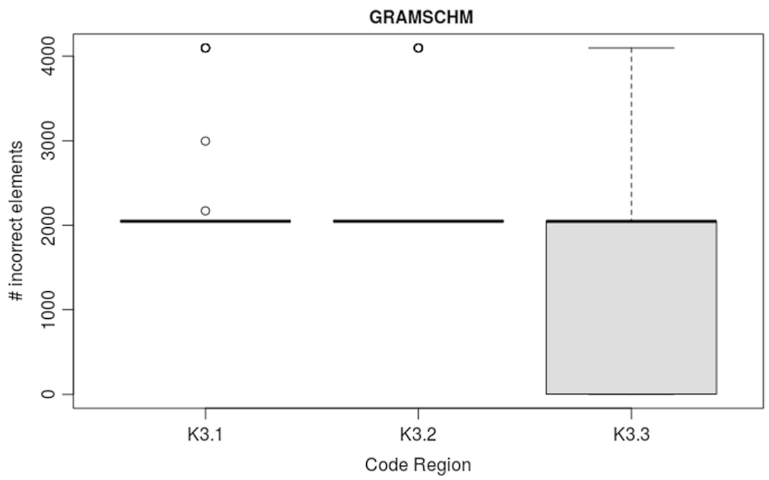


Fig. 7 Boxplot for incorrect elements of *gramschmidt_kernel*3 kernel

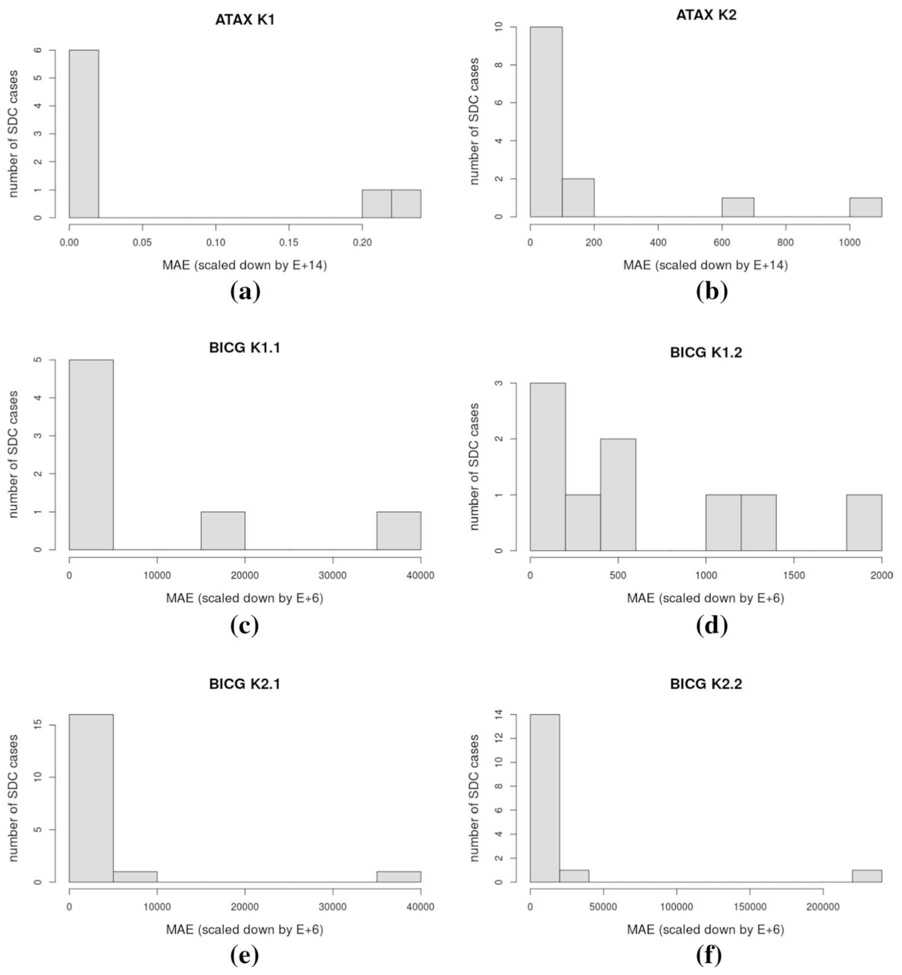


Fig. 8 Histogram of mean absolute error (MAE)

a subset of them and utilizing the protected resources (like registers) for more vulnerable operations decrease the performance overhead of the replication.

5.2.4 Approximation methods

To deal with the large performance overheads for applications not requiring 100% correct output, the approximation techniques have been applied in different computing levels [32]. Since our data corruption criticality evaluation reports the data corruptions over different parts of the output data, we can understand that how much the output is affected by an error (or any miscalculation) in any code region. Consequently, even non-safety-critical programs can utilize our framework to make decisions about the program parts to be approximated. As we demonstrate in our

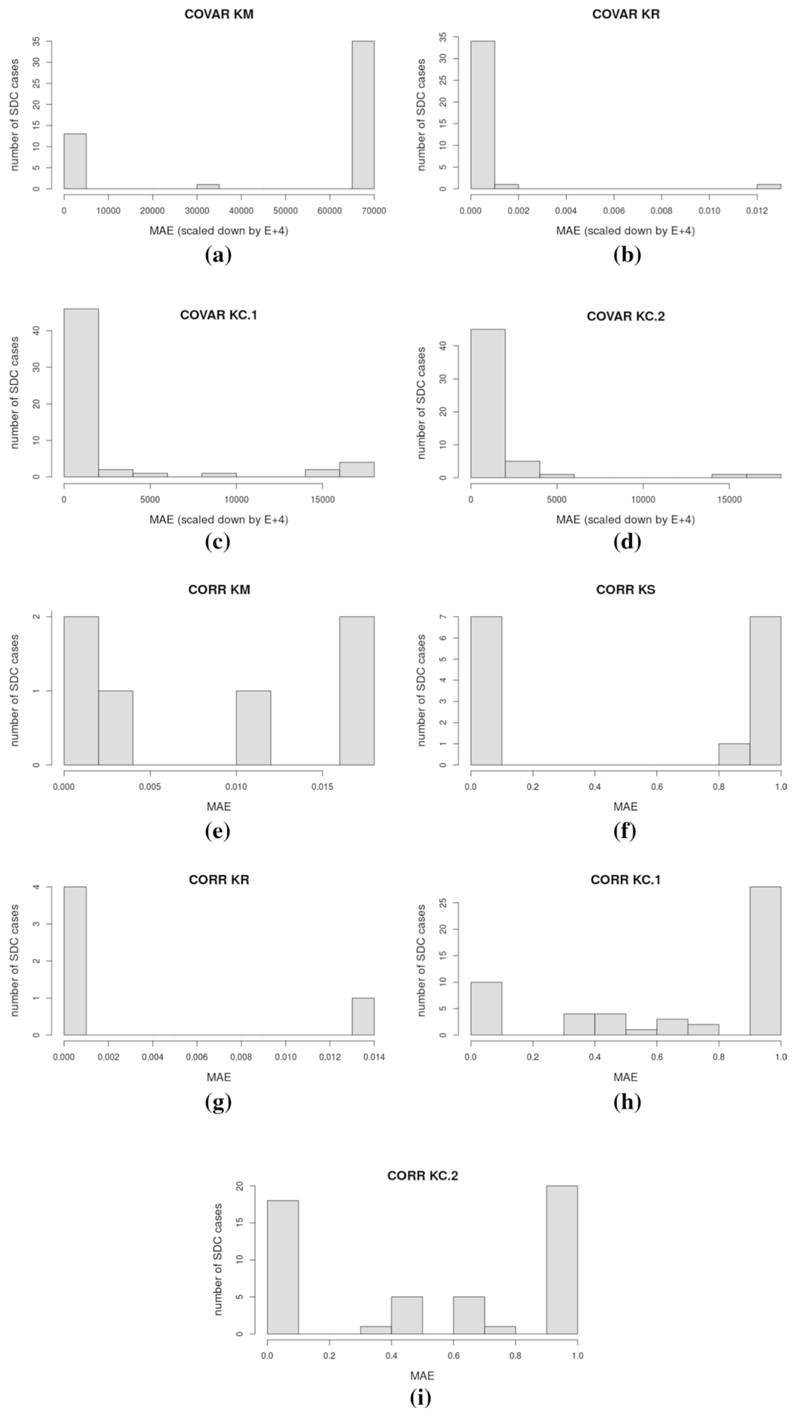


Fig. 9 Histogram of mean absolute error (MAE)

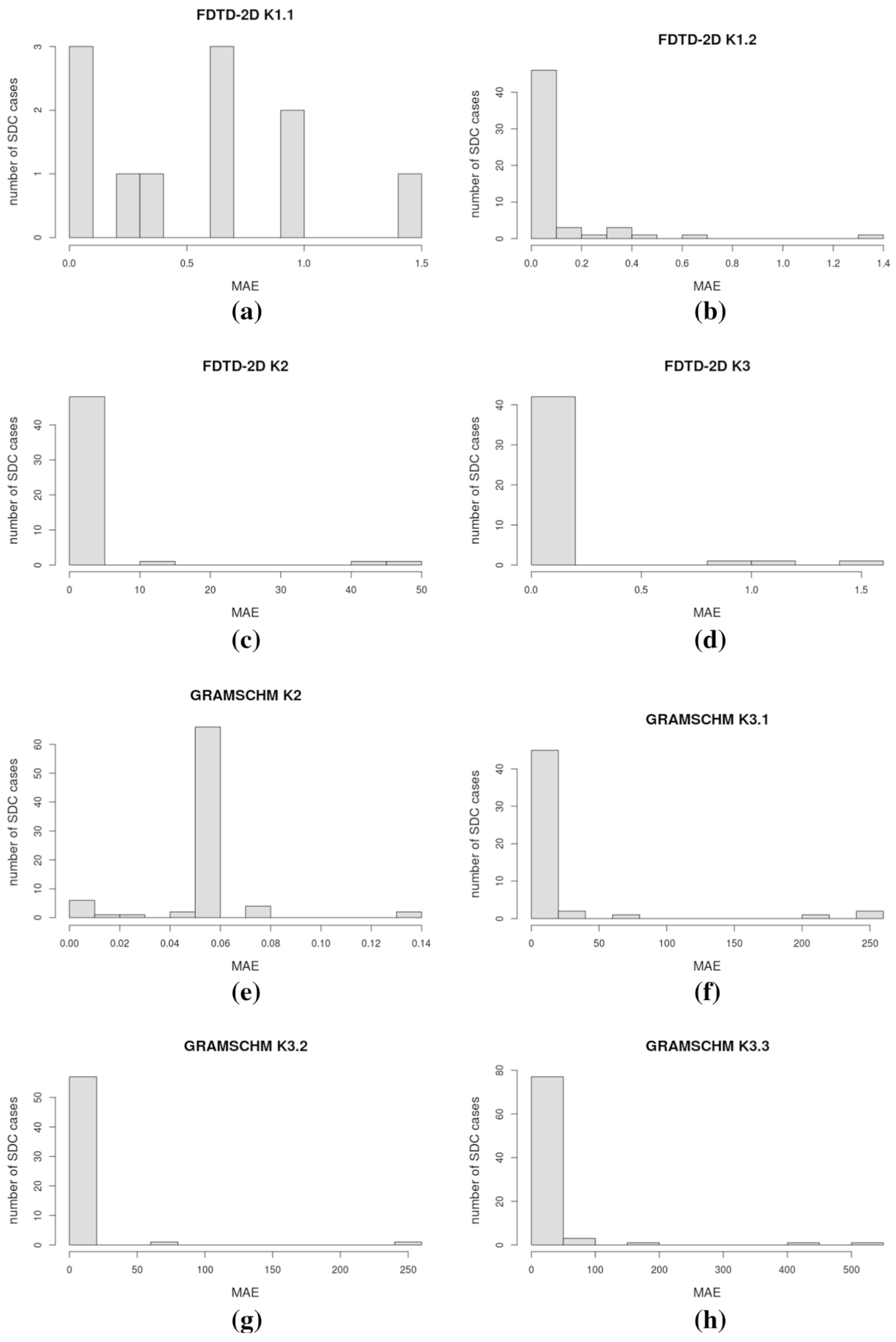


Fig. 10 Histogram of mean absolute error (MAE)

experimental study, our error propagation analysis allows tracking the program data during the program execution. By observing the errors destroying the program data during the execution of the specific code regions, one can decide to perform approximation on less vulnerable program points. Since less vulnerable corresponds to less influence on the program output, the computations in those code regions can be skipped or the data types with less precision can be utilized by considering that the output will not be affected by the incorrect or imprecise values computed during the execution of those code regions.

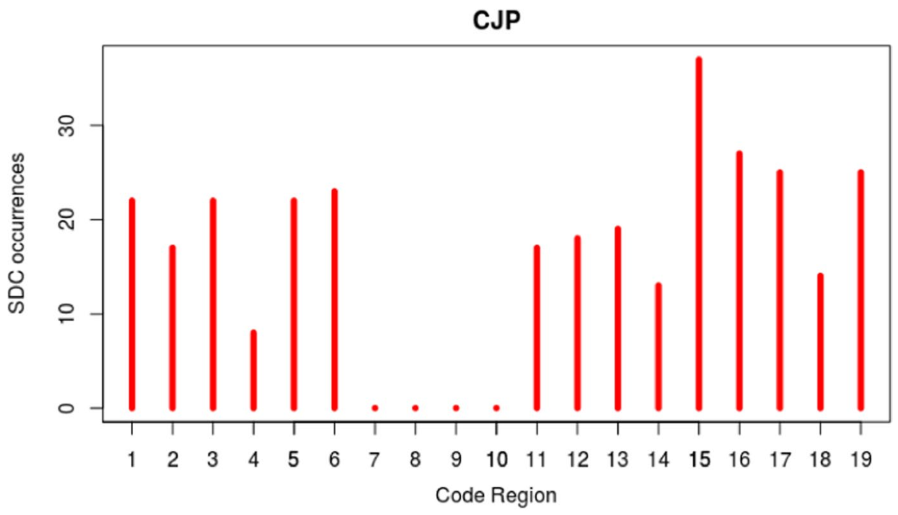
5.2.5 Guidance to the software developer

While the software developers first aim to write correct code, the performance or reliability may be seen as a major feature for applications running in high-performance computing and safety-critical systems, respectively. Since the programmers may not know the architectural details of the underlying system, providing high-level instructions to guide them for writing much faster or more reliable code can be a good contribution to satisfying the target programs' requirements. While more substantial work exists for high-performance and efficient program development guides [10, 25], there is a lack of reliable execution suggestions in the literature. Especially, fine-grained code analysis and offering vulnerability-aware suggestions for more reliable execution are not employed. Our framework, which performs a debugger-level fault injection and provides a connection between the high-level program (e.g., CUDA kernel functions) and the underlying architecture resources (like register file), allows the programmers to learn more vulnerable parts of their programs. According to the guidelines deduced from the results of our empirical study, the programmer can utilize our framework for developing a more reliable program. For instance, the number of threads and the parallelism incurred in the target code can be adjusted by considering the reliability as well as performance. As we discuss in our observations, the higher occupancy might result in higher performance; however, it also makes the program more vulnerable to the soft errors in the register file. By taking into account both performance and reliability considerations, the software developer can conduct a trade-off analysis between a faster or less vulnerable program and make decisions based on the guidelines provided by our framework.

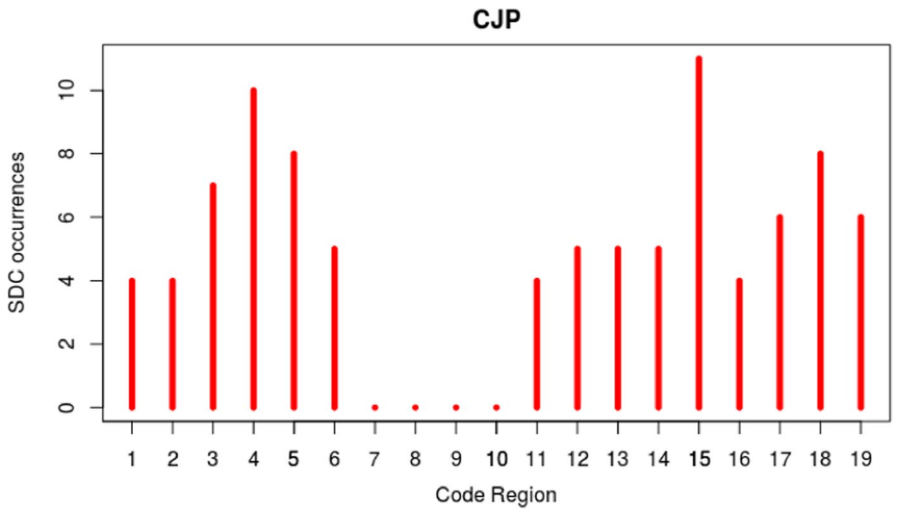
6 Related work

Assembly-level fault injections introduce errors in the assembly instructions of the target application during its execution. The execution trace is performed by profiling or in the debugging phase. While the assembly-level fault injection yields less detailed analysis than the microarchitectural-level fault injection, it provides more information than the higher (source code) level injection methodologies.

Fang et al. [13, 14] present a methodology for the reliability evaluation of GPGPU applications by using a debugger-based fault injection framework. First, the proposed fault injection environment (GPU-Qin) groups the similar CUDA threads to profile only one thread in the same group. Then it profiles the threads



(a)



(b)

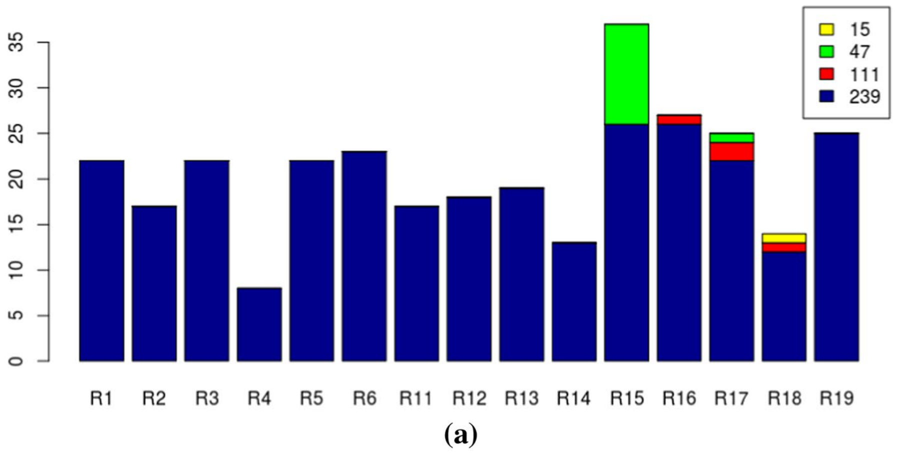
Fig. 11 SDC distribution for CJP application (with FI on 64-128 registers)

by executing the program in GPGPU-Sim simulator [6] and determines the number of instructions. In the final fault injection phase, a set of fault injection runs is conducted to get the fault rates for the program. For each fault injection run, one instruction is selected from the instructions collected in the profiling phase and a breakpoint is added at that instruction. When the execution reaches the target instruction, a fault is injected by flipping a randomly chosen single bit in the register used in the instruction. The authors present a detailed experimental study to demonstrate the use of the proposed methodology to characterize GPGPU applications' soft error vulnerability. Our fault injection tool is similar to GPU-Qin, as it is built on cuda-gdb; on the other hand, it differs in the way that its target fault injection point is the high-level source code sections instead of the low-level instructions.

Hari et al. [19] present a fault injection tool (SASSIFI) for NVIDIA GPUs based on assembly-language instrumentation tool (SASSI), which modifies the registers and memory. SASSIFI works in three steps including: profiling the program, selecting fault injection points, and injecting errors into programs. The authors also demonstrate that how SASSIFI can be utilized for the resilience evaluation of applications. Similar to GPU-Qin, SASSIFI employs at SASS instruction level, while our fault injector targets high-level source code sections and performs the fault injection for the random instructions that are part of the specified code regions. Previlon et al. [38] employ SASSIFI for the evaluation of the correlation between the performance phases of GPU programs and the soft error vulnerability. Based on their observations, the authors propose a fault injection methodology, Spoti-FI, to determine the most representative fault injection points in the program and reduce the total number of faults necessary for fault injection experiments. Recently, Tsai et al. [45] present a fault injection tool (NVBitFI), which offers functionality that is similar to SASSIFI. NVBitFI is based on NVBit (NVidia Binary Instrumentation Tool) [46], which is recommended to use for recent GPU architectures. NVBitFI can run on newer GPUs like Turing and Volta architectures, additionally, it works with pre-compiled libraries and is faster than the SASSIFI. Santos et al. [43] perform a comparison study between the real beam experiments and NVBitFI-based fault injection simulations.

Santos et al. [16] evaluate the soft error vulnerability of mixed-precision architectures and utilize a fault injection tool implemented on cuda-gdb debugger for target GPU devices. In their other work [42], Santos et al. evaluate the soft error vulnerability of the object detection frameworks by utilizing CAROL-FI and investigate Histogram of Oriented Gradients (HOG) and You Only Look Once (YOLO) benchmarks to examine kernel and layer vulnerabilities. The authors focus on the programs from a specific domain and do not present a generic vulnerability analysis.

Yang et al. [50] present an efficient error resilience evaluation methodology (SUGAR) based on the estimations by executing fault injections with small input sizes. Instead of performing long-running experiments with real data, SUGAR proposes a pattern discovery scheme to represent the vulnerability of the GPU programs as a function of the input size. By utilizing the dynamic instruction counts from the experiments with small inputs, it accurately predicts the error resilience for large input files. While SUGAR performs reliability evaluations on GPGPU-Sim simulator [6] by working on instructions, the authors claim that it can also be used with SASSIFI [19] and NVBitFI [45]. In their other recent work, Yang et al. [49] propose a progressive fault site pruning mechanism including thread-wise, instruction-wise, loop-wise, and bit-wise pruning stages. By performing those pruning stages, they reduce the fault space and decrease the fault injection cost significantly. Since the main target of our work is not efficient execution, we do not focus on performance issues. However, we believe that our fault injection tool can be optimized by utilizing the proposed methodologies.



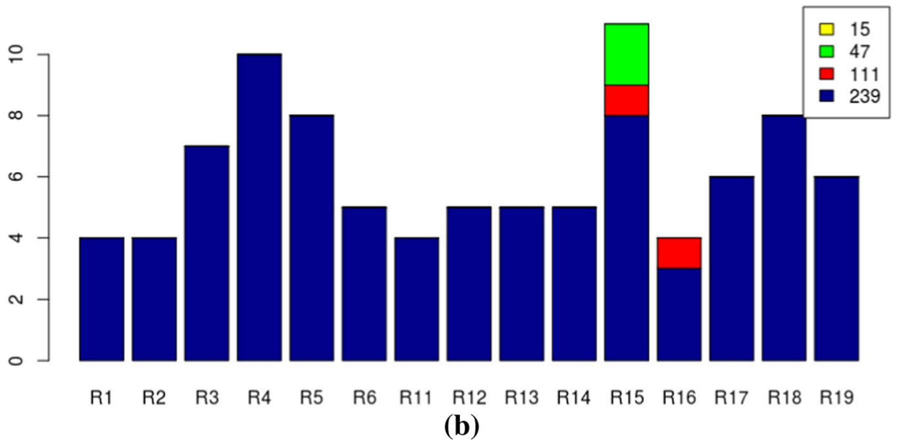


Fig. 12 Absolute error values for CJP application (with FI on 64-128 registers)

Leng et al. [26] examine the GPU activities to understand the source of the voltage droops during the program execution. Similar to our inter-kernel and intra-kernel analysis, they monitor inter-kernel and intra-kernel activities, where the consecutive launch of kernels can cause large voltage droops and intra-kernel micro-architectural events can affect voltage droops, respectively. By identifying the activities affecting the voltage levels significantly, the authors propose the elimination of the specific activity to reduce the droop. Furthermore, Zamani et al. [51, 52] propose a fault tolerance algorithm for matrix multiplication for the case that they perform undervolting in the GPU beyond the minimum operating voltage to save energy, but have possible faults during the execution. Based on the activities introduced by Leng et al. [26], they observe that at a specific voltage, the different inter-kernel and intra-kernel activities can lead to different failure rates. While the authors discuss similar concepts by examining both inter-kernel and intra-kernel activities, our work focuses directly on the soft error vulnerability of the GPU programs by performing a high-level fault injection study without dealing with the effect of the specific activities such as consecutive kernel function calls or microarchitectural events on the voltage levels and fault rates.

In our work, we propose a debugger-based fault injector tool that can inject faults in predetermined regions to analyze the regional soft error vulnerability of GPGPU programs. Our methodology enables us to evaluate both intra-kernel and inter-kernel vulnerability analysis. It also presents error propagation through data structures in the program in case of error occurrence in different program regions.

7 Conclusions and future work

In this work, we present a region-based soft error vulnerability analysis for GPGPU applications. We build a fault injection framework to evaluate both inter-kernel and intra-kernel vulnerabilities of the programs running on GPU architectures. We also design and develop an error propagation tracker to evaluate the spread of an error through data variables during the execution of multiple kernel functions.

We perform detailed fault injection experiments for a set of GPGPU programs and observe that GPGPU programs demonstrate different soft error vulnerability and propagation for their different code regions. Based on our empirical data, we make prominent observations about the fault behavior of GPGPU programs that can be utilized in multiple ways. Furthermore, we present the usage scenarios of our regional fault analysis framework as a guide to the researchers, system users, or software developers seeking ways to employ high fault tolerance in their systems or programs.

We believe that fine-grained vulnerability analysis can help both professional and academic systems targeting reliability. By considering domain-specific metrics for the vulnerability and performing fault injections at various levels (like inter-kernel and intra-kernel), more efficient fault tolerance techniques can be employed in safety-critical systems requiring both performance and reliability in their executions. The performance and reliability trade-off analysis can be performed by selectively replicating only the most vulnerable parts of the target programs or protecting only the most vulnerable data. Moreover, approximate computing techniques can be utilized by performing approximation for the less vulnerable/influential code regions.

Acknowledgements This work was supported by the Scientific and Technological Research Council of Turkey (TÜBİTAK), Grant No: 119E011.

References

1. Implementing-graphcoloring-on-gpu (2020). https://github.com/cemsakizci/Implementing-graphColoring-on-GPU
2. Nvidia, cuda-gdb (2020). https://developer.nvidia.com/cuda-gdb
3. Aamodt TM, Fung WWL, Rogers TG, Martonosi M (2018) General-purpose graphics processor architecture

4. Anwer AR, Li G, Pattabiraman K, Sullivan M, Tsai T, Hari SKS (2020) Gpu-trident: efficient modeling of error propagation in gpu programs. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC '20
5. Arslan S, Unsal O (2021) Efficient selective replication of critical code regions for sdc mitigation leveraging redundant multithreading. Journal of Supercomputing pp. 1. https://doi.org/10.1007/s11227-021-03804-6
6. Bakhoda A, Yuan GL, Fung WWL, Wong H, Aamodt TM (2009) Analyzing cuda workloads using a detailed gpu simulator. In: International Symposium on Performance Analysis of Systems and Software
7. Borodin D, Juurlink BH (2010) Protective redundancy overhead reduction using instruction vulnerability factor. Proceedings of the 7th ACM International Conference on Computing Frontiers (CF)
8. Cini N, Yalcin G (2020) A methodology for comparing the reliability of gpu-based and cpu-based hpcs. ACM Comput Surv 53(1). https://doi.org/10.1145/3372790
9. Clark JA, Pradhan DK (1995) Fault injection: a method for validating computer-system dependability. Computer 28(6):47–56
10. Cook S (2013) Chapter 9 - optimizing your application. In: S. Cook (ed.) CUDA Programming, Applications of GPU Computing Series, pp. 305 – 440. Morgan Kaufmann, Boston. https://doi.org/10.1016/B978-0-12-415933-4.00009-0. http://www.sciencedirect.com/science/article/pii/B9780124159334000090
11. Davis TA, Hu Y (2011) The university of florida sparse matrix collection. ACM Trans Math Softw 38(1)
12. Dimitrov M, Mantor M, Zhou H (2009) Understanding software approaches for gpgpu reliability. In: Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units, GPGPU-2, p. 94–104
13. Fang B, Pattabiraman K, Ripeanu M, Gurumurthi S (2014) Gpu-qin: A methodology for evaluating the error resilience of gpgpu applications. IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)
14. Fang B, Pattabiraman K, Ripeanu M, Gurumurthi S (2016) A systematic methodology for evaluating the error resilience of gpgpu applications. IEEE Transac Parallel Distrib Syst 27(12):3397–3411
15. Feng S, Gupta S, Ansari A, Mahlke S (2010) Shoestring: probabilistic soft error reliability on the cheap. Proceedings of the Fifteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), p. 385–396
16. Fernandes dos Santos F, Lunardi C, Oliveira D, Libano F, Rech P (2019) Reliability evaluation of mixed-precision architectures. IEEE International Symposium on High Performance Computer Architecture (HPCA)
17. Grauer-Gray S, Xu L, Searles R, Ayalasomayajula S, Cavazos J (2012) Auto-tuning a high-level language targeted to gpu codes. 2012 Innovative Parallel Computing (InPar)
18. Gupta M, Lowell D, Kalamatianos J, Raasch S, Sridharan V, Tullsen D, Gupta R (2017) Compiler techniques to reduce the synchronization overhead of gpu redundant multithreading. In: 2017 54th ACM/EDAC/IEEE Design Automation Conference (DAC), pp. 1–6 (2017). https://doi.org/10.1145/3061639.3062212
19. Hari SKS, Tsai T, Stephenson M, Keckler SW, Emer J (2017) Sassifi: an architecture-level fault injection tool for gpu application resilience evaluation. In: International Symposium on Performance Analysis of Systems and Software (ISPASS), International Symposium on Performance Analysis of Systems and Software (ISPASS)
20. Hukerikar S, Teranishi K, Diniz PC, Lucas RF (2018) Redthreads: an interface for application-level fault detection/correction through adaptive redundant multithreading. International Journal of Parallel Programming 46
21. Jauk D, Yang D, Schulz M (2019) Predicting faults in high performance computing systems: an in-depth survey of the state-of-the-practice. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC '19
22. Jeon H, Annavaram M (2012) Warped-dmr: light-weight error detection for gpgpu. In: International Symposium on Microarchitecture (MICRO), International Symposium on Microarchitecture (MICRO)
23. Kalra C, Previlon F, Li X, Rubin N, Kaeli D (2018) Prism: predicting resilience of gpu applications using statistical methods. In: SC18: International Conference for High Performance Computing, Networking, Storage and Analysis, pp. 866–879 . https://doi.org/10.1109/SC.2018.00072

24. Kalra C, Previlon F, Rubin N, Kaeli D (2020) Armorall: compiler-based resilience targeting gpu applications. ACM Trans. Archit. Code Optim. 17(2). https://doi.org/10.1145/3382132
25. Kirk DB, mei W Hwu W (2017) Chapter 5 - performance considerations. In: D.B. Kirk, W. mei W. Hwu (eds.) Programming Massively Parallel Processors (Third Edition), third edition edn., pp. 103 – 130. Morgan Kaufmann . https://doi.org/10.1016/B978-0-12-811986-0.00005-4. http://www.sciencedirect.com/science/article/pii/B9780128119860000054
26. Leng J, Buyuktosunoglu A, Bertran R, Bose P, Reddi VJ (2015) Safe limits on voltage reduction efficiency in gpus: a direct measurement approach. In: Proceedings of the 48th International Symposium on Microarchitecture, MICRO-48, p. 294–307
27. Leveugle R, Calvez A, Maistri P, Vanhauwaert P (2009) Statistical fault injection: quantified error and confidence. Proceedings of the Conference on Design, Automation and Test in Europe (DATE)
28. Li G, Pattabiraman K, Cher C, Bose P (2016) Understanding error propagation in gpgpu applications. In: SC '16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, pp. 240–251. https://doi.org/10.1109/SC.2016.20
29. Li T, Ambrose JA, Ragel R, Parameswaran S (2016) Processor design for soft errors: challenges and state of the art. ACM Comput. Surv. 49(3)
30. Mahmoud A, Hari SKS, Sullivan MB, Tsai T, Keckler SW (2018) Optimizing software-directed instruction replication for gpu error detection. International Conference for High Performance Computing, Networking, Storage and Analysis (SC), pp. 842–854
31. Hsueh Mei-Chen, Tsai TK, Iyer RK (1997) Fault injection techniques and tools. Computer 30(4):75–82
32. Mittal S (2016) A survey of techniques for approximate computing. ACM Comput Surv 48(4)
33. Mukherjee S (2008) Architecture Design for Soft Errors. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA
34. Nie B, Tiwari D, Gupta S, Smirni E, Rogers JH (2016) A large-scale study of soft-errors on gpus in the field. In: 2016 IEEE International Symposium on High Performance Computer Architecture (HPCA), pp. 519–530. https://doi.org/10.1109/HPCA.2016.7446091
35. Owens JD, Luebke D, Govindaraju N, Harris M, Krüger J, Lefohn AE, Purcell TJ (2007) A survey of general-purpose computation on graphics hardware. Comput Graphics Forum 26(1):80–113
36. Oz I, Topcuoglu HR, Tosun O (2019) A user-assisted thread-level vulnerability assessment tool. Concurrency and Computation: Practice and Experience 31(13)
37. Palazzi L, Li G, Fang B, Pattabiraman K (2020) Improving the accuracy of ir-level fault injection. IEEE Transactions on Dependable and Secure Computing pp. 1–1. https://doi.org/10.1109/TDSC.2020.2980273
38. Previlon FG, Kalra C, d. tiwari, Kaeli DR (2020) Characterizing and exploiting soft error vulnerability phase behavior in gpu applications. IEEE Transactions on Dependable and Secure Computing pp. 1
39. Quang Anh Pham N, Fan R (2018) Efficient algorithms for graph coloring on gpu. 2018 IEEE 24th International Conference on Parallel and Distributed Systems (ICPADS)
40. Reis GA, Chang J, Vachharajani N, Rangan R, August DI (2005) Swift: software implemented fault tolerance. International Symposium on Code Generation and Optimization
41. Reis GA, Chang J, Vachharajani N, Rangan R, August DI, Mukherjee SS (2005) Software-controlled fault tolerance. ACM Trans Archit Code Optim 2(4):366–396. https://doi.org/10.1145/1113841.1113843
42. dos Santos FF, Carro L, Rech P (2019) Kernel and layer vulnerability factor to evaluate object detection reliability in gpus. IET Computers and Digital Techniques 13
43. Santos FFd, Hari SKS, Basso PM, Carro L, Rech P (2021) Demystifying gpu reliability: comparing and combining beam experiments, fault simulation, and profiling. In: 2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS), pp. 289–298. https://doi.org/10.1109/IPDPS49936.2021.00037
44. Shivakumar P, Kistler M, Keckler SW, Burger D, Alvisi L (2002) Modeling the effect of technology trends on the soft error rate of combinational logic. In: International Conference on Dependable Systems and Networks (DSN), International Conference on Dependable Systems and Networks (DSN)
45. Tsai T, Hari SKS, Sullivan MB, Villa O, Keckler SW (2021) Nvbitfi: dynamic fault injection for gpus. In: International Conference on Dependable Systems and Networks, (DSN)
46. Villa O, Stephenson M, Nellans D, Keckler SW (2019) Nvbit: a dynamic binary instrumentation framework for nvidia gpus. In: Proceedings of the 52nd Annual IEEE/ACM International

Symposium on Microarchitecture, MICRO ’52, p. 372–383. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3352460.3358307

47. Wadden J, Lyashevsky A, Gurumurthi S, Sridharan V, Skadron K (2014) Real-world design and evaluation of compiler-managed gpu redundant multithreading. In: International Symposium on Computer Architecture (ISCA), International Symposium on Computer Architecture (ISCA)
48. Wang J (2017) Acceleration and optimization of dynamic parallelism for irregular applications on gpus. Ph.D. thesis, Georgia Institute of Technology, Atlanta, GA, USA. http://hdl.handle.net/1853/56294
49. Yang L, Nie B, Jog A, Smirni E (2021) Practical resilience analysis of gpgpu applications in the presence of single- and multi-bit faults. IEEE Transac Comput 70(1):30–44. https://doi.org/10.1109/TC.2020.2980541
50. Yang L, Nie B, Jog A, Smirni E (2021) Sugar: speeding up gpgpu application resilience estimation with input sizing. Proc. ACM Meas. Anal. Comput. Syst. 5(1). https://doi.org/10.1145/3447375
51. Zamani H, Liu Y, Tripathy D, Bhuyan L, Chen Z (2019) Greenmm: energy efficient gpu matrix multiplication through undervolting. In: Proceedings of the ACM International Conference on Supercomputing, ICS ’19, p. 308–318
52. Zamani H, Tripathy D, Bhuyan L, Chen Z (2020) Saou: safe adaptive overclocking and undervolting for energy-efficient gpu computing. In: Proceedings of the ACM/IEEE International Symposium on Low Power Electronics and Design, ISLPED ’20, p. 205–210