Evaluating Performance and Reliability of Selective Redundant Multithreading for GPGPU Applications

Ercüment Kaya*1*, Ömer Faruk Karadaş*2* and Işil Öz*1*

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

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

Abstract

With the widespread use of GPU architectures in general-purpose computations, evaluating the soft error vulnerability of GPGPU programs and employing efficient fault tolerance techniques for more reliable execution becomes more prominent. Performing full redundancy, based on the redundant execution of the complete program, results in resource consumption and performance loss as well as energy inefficiency. Therefore, determining the most error-prone regions of the target program code and replicating only those parts maintains both high performance and acceptable error rates. In this study, we propose a partial redundant multithreading mechanism based on the soft error vulnerability of GPGPU applications and perform a trade-off analysis between performance and reliability. Firstly, we perform fault injection experiments to evaluate the SDC rates for each kernel function. Then, based on the outcome of the fault injection experiments, we determine the kernel function to-be-replicated. According to the pragmas denoting the redundancy points in the source code, our custom LLVM pass generates the code that enables the redundant execution for the specified code region. We evaluate both the reliability and performance of the redundant execution scenarios measuring the execution time of the redundant program generated by our compiler-managed redundancy technique. Our results demonstrate that protecting only the most vulnerable kernel functions enables high reliability without hurting the performance significantly.

Keywords

Soft error reliability, Fault injection, Redundant execution, GPGPU programs

1. Introduction

Heterogeneous computing systems offer high performance and less energy consumption by combining a wide range of device structures and configurations. Building heterogeneous systems by bringing together general-purpose multi-core processors (CPUs) and data-parallel graphic processing units (GPUs) enables efficient computation for high performance and energy consumption in large-scale computing platforms. Recently, the high computation power of the GPU architectures has been largely utilized for general-purpose computations as well as graphics applications [1]. Therefore, the soft error vulnerability becomes a more significant design concern for the target general-purpose programs than the inherently error-tolerant graphics computations.

To deal with the effects of the hardware errors on the target programs, fault tolerance techniques can be employed by executing the code redundantly. Compiler-level redundancy, employed for both CPU and GPU applications, is one of the widely used approaches in the reliability-aware computing domain. The traditional approach, SWIFT [2], enables instruction-level redundancy by providing high-level protection. Moreover, nZDC [3] presents near-zero silent data corruption by targeting to remove the limitations of the SWIFT, such as not wholly protected branches. Wadden et al. [4] propose a compiler-level redundant multithreading technique for OpenCL-based GPU applications. The proposed method duplicates the work-groups instead of the functions or the instructions by eliminating the overheads of the instruction-based replication. Our work targets CUDA-based programs and kernel function replications.

While the redundant approaches achieve high fault coverage, performing full redundancy, based on the redundant execution of the complete program, results in resource consumption and performance loss as well as energy inefficiency. Therefore, determining the most error-prone regions of the target program code and replicating only those parts maintain both high performance and acceptable error rates. Bohman et al. [5] present compiler-assisted software fault tolerance (COAST) for microcontrollers, which are widely used in task-based programming and/or in extraordinary environments. While the COAST uses selective replication based on the code annotations, its instruction-based replication strategy increases the code size significantly and causes a memory bottleneck. To eliminate this overhead, we target the replication of the functions instead of the small-grained instructions. The recent work, ArmorALL [6], presents a selective compiler-level solution to protect GPUs against soft errors. ArmorALL provides three compiler-based redundancy schemes, including Address Armor, Value Armor, and Hybrid Armor. While Address Armor protects only the addresses used by memory instructions, Value Armor protects the values by duplicating all instructions that participate in the value computation. Hybrid Armor protects both of them. Even though ArmorALL uses selective redundancy based on its redundancy schemes, it does not allow the programmer to define the specific code regions for the redundant execution. To reduce the overhead and give more control to the programmer, our work uses selective redundancy based on the annotations. Moreover, Yang et al. [7] recently propose a selective replication scheme by remapping threads with the same vulnerability behavior into the same warps. They evaluate individual thread vulnerabilities and combine reliable and unreliable threads by placing similar ones into the same warps. Their technique relies on the fact that having reliable and unreliable threads in the same warp can hurt the performance. The replication of the entire warp, which includes only unreliable threads, is more efficient than replicating the individual threads in a mixed warp. In our work, we evaluate kernel-level soft error vulnerabilities instead of thread-level, which is more cost-efficient. While [7] requires architectural modifications to enable thread remapping, our work offers compiler support and the programmer directly controls the replication by utilizing the annotations.

In this study, we propose a partial redundant multithreading mechanism based on the soft error vulnerability of GPGPU applications and evaluate the performance and reliability of the target executions. Firstly, in our fault injection framework, we evaluate the most vulnerable code regions of the target applications by considering kernel functions. Then, based on the outcome of the fault injection experiments, we determine the kernel function to be replicated. According to the pragmas denoting the redundancy points in the source code, our compiler generates the code that enables the redundant execution for the specified code region.

Our debugger-based regional fault injection tool [8] generates fault injection points for each kernel function based on the information gathered during the profiling phase. It evaluates the silent data corruption (SDC) rates to get the soft error vulnerability of each kernel function in the target GPGPU program. Additionally, our LLVM-based compiler framework generates the target executable including redundant code sections for the specified kernel functions marked by the programmer, who utilizes the feedback from our fault injection analysis. We perform an experimental study to reveal the efficiency of our approach for a set of GPGPU applications. Our fault injection experiments demonstrate that the code regions inside GPGPU programs exhibit different characteristics in terms of soft error vulnerability, pointing to a partial redundancy for both higher performance and reliability. Based on the recommendations of our soft error vulnerability analysis, we perform redundant executions that replicate only the most vulnerable parts of the target programs. We evaluate both the reliability and performance of the redundant execution scenarios by conducting fault injection experiments and measuring the execution time of the redundant program generated by our compiler-managed redundancy technique. Our results demonstrate that protecting only the most vulnerable kernel functions enables high reliability without hurting the performance significantly.

The remainder of this paper is organized as follows: Section 2 presents some background on soft error reliability and redundant multithreading. We explain our selective redundant multithreading methodology in Section 3. Then the experimental results are outlined in Section 4. Finally, in Section 5, we summarize the work with some conclusive remarks.

2. Background and Motivation

2.1. Soft Error Reliability in GPGPUs

Soft errors, resulting from single-bit flips in computer system structures, are caused by alpha particles, cosmic rays, thermal neutrons, or other environmental causes [9, 10]. If a soft error hits a register or a memory location, it may cause data corruption or program crash when consumed by the application. While the fault may be masked due to be ignored by the program or corrected by any error correction mechanism, it might affect the program outcome by means of silent data corruption (SDC) or detected unrecoverable error (DUE). Silent data corruption is the most critical one since it produces incorrect program outputs while the program seems to end successfully.

As GPUs are increasingly being utilized for the acceleration of the general-purpose computations, their soft error resilience is becoming more critical than before when they were used only for graphics and considered inherently fault-tolerant. Hence, in recent years, the soft error problem for GPU systems has become a first-class design challenge especially. Consequently, we focus on soft error evaluation of GPU systems in this study.

2.2. Redundant Multithreading

To deal with soft errors in computer systems, redundant multithreading (RMT) approaches have been implemented in various levels [11]. Based on the replication of both the code and data for the target execution, RMT enables the execution of two redundant copies of the

program. The key concepts for redundant multithreading are the components included in the redundant execution, namely sphere of replication (SOR). The SOR identifies the components to be replicated while the components outside SOR must be protected by some other fault tolerance mechanism.

3. Methodology

Our selective redundant multithreading framework consists of three main components, as shown in Figure 1:
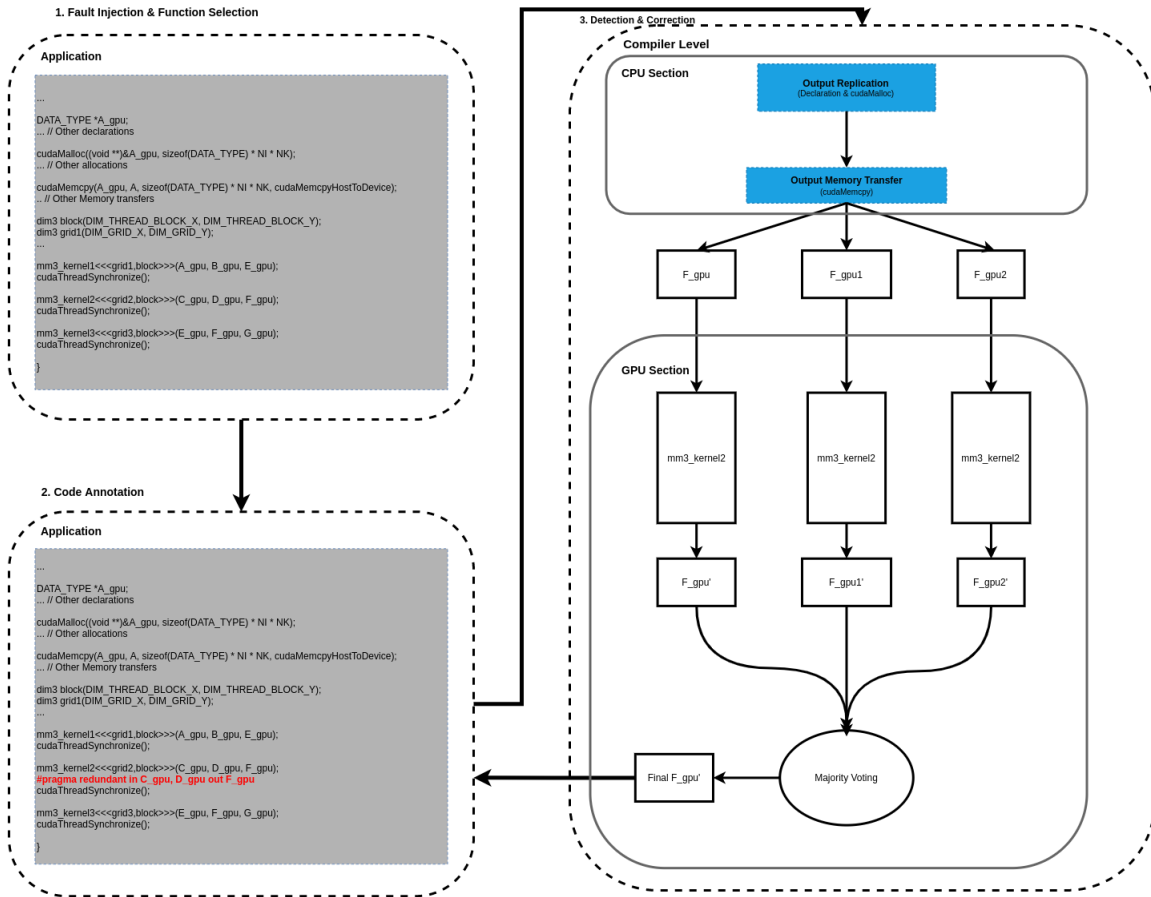

Figure 1: Workflow of our redundant multithreading framework.

- **Fault injection and kernel function selection:** We perform fault injection experiments to obtain the soft error vulnerabilities of the kernel functions in the target GPGPU programs. In this way, we examine the relative soft error rates of the different kernel functions in a given program and decide the most vulnerable one as a candidate function for replication.

- **Code annotation:** After selecting the kernel function to execute redundantly, we insert the *#pragma* directive that notifies the compiler about the redundant execution and provides additional information to the compiler. Our custom compiler supports this directive and enables redundant execution for the given function with the essential parameters.
- **Redundant code generation and execution:** After getting the target kernel function from the *#pragma* directive, our compiler framework generates the code for redundant execution by replicating both kernel instructions and output data to be generated by the redundant copies. Moreover, it adds the majority voting function to obtain the corrected result in case of any error.

3.1. Regional Soft Error Reliability Evaluation

We utilize the fault injection tool that enables regional vulnerability analysis for GPGPU programs [8]. The debugger-based fault injector targets specified kernel function execution as each fault injection point and enables us to evaluate the fault rates for the target kernel functions. By utilizing its *Configuration* interface, we specify the kernel functions that we target for the fault injection. Then *Profiling* phase enables us to collect information about the target code and *Fault Generation* phase determines the specific instructions in the given function and the target register bits. Finally, *Fault Injection* phase flips in the specified register bit during the execution of the specified instruction generated in the fault generation phase. We perform fault injection experiments for each kernel function in the given GPGPU program and obtain silent data corruption (SDC) rates to quantify the soft error vulnerabilities. In this way, we can get the vulnerability values for each kernel function and compare the fault characteristics to obtain the kernel function with the largest vulnerability for the redundant execution.

3.2. Compiler-Level Redundant Execution

We build our compiler-based redundant execution framework on top of LLVM framework [12]. As shown in Figure 2, LLVM consists of three major components: Front-end, Optimizer, and Back-end.
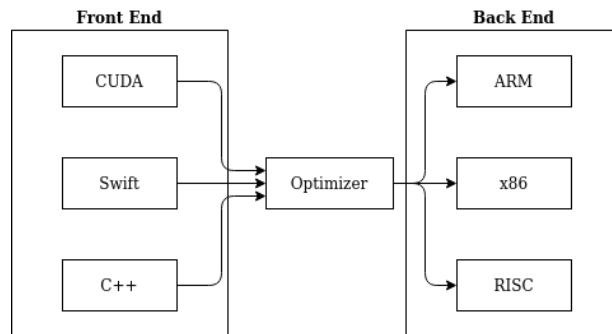

Figure 2: Major LLVM components [13].

The Front-end component is language dependent, it takes the source code as an input and generates the LLVM IR code. The Back-end component is architecture dependent, it takes the LLVM IR code as an input and generates the machine code. The Optimizer component takes LLVM IR code and generates the optimized LLVM IR code. Our general compilation flow, based on LLVM framework and shown in Figure 3, consists of three parts: 1) Generating the LLVM IR code, 2) Generating the new LLVM IR code for both the host and the device using the custom LLVM pass, 3) Generating the executable by using the generated LLVM IR codes.
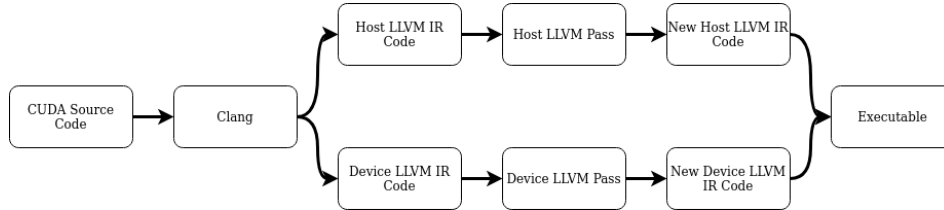

Figure 3: Compiling CUDA with Clang.

Our compiler-level RMT scheme replicates the kernel functions specified by our directive and the output data, implements the majority function, and enables the redundant execution of the code marked by the programmer. There are four major components in our implementation: 1) Compiler directive, 2) Output replication, 3) Kernel function replication, 4) Majority voting implementation. We perform both host and device code modifications in the compilation phase given in Figure 3. While the implementation of the *Output Replication* and the *Kernel Function Call Replication* components requires modifications in the host code, we update both the host code and the device code for the *Majority Voting*. Listing 3 and Listing 4 present an example code with our annotation and the target code to be generated by our compiler, respectively. In the following part, we will explain our implementation details by presenting them in the example code snippet.

```
void mm3Cuda(int ni,
            ... // Other parameters
            )
{
    DATA_TYPE *A_gpu;
    ... // Other declarations

    cudaMalloc((void **)&A_gpu, sizeof(DATA_TYPE) * NI * NK);
    ... // Other allocations

    cudaMemcpy(A_gpu, A, sizeof(DATA_TYPE) * NI * NK,
                        cudaMemcpyHostToDevice);
    .. // Other Memory transfers

    dim3 block(DIM_THREAD_BLOCK_X, DIM_THREAD_BLOCK_Y);
    dim3 grid1(DIM_GRID_X, DIM_GRID_Y);

    mm3_kernel1<<<grid1, block>>>(A_gpu, B_gpu, E_gpu);
    cudaThreadSynchronize();

    mm3_kernel2<<<grid2, block>>>(C_gpu, D_gpu, F_gpu);
    #pragma redundant in C_gpu, D_gpu out F_gpu

    cudaThreadSynchronize();

    mm3_kernel3<<<grid3, block>>>(E_gpu, F_gpu, G_gpu);
    cudaThreadSynchronize();
}
```
Listing 1: Annotated CUDA code

```
%call42 = call i32 @cudaConfigureCall(i64 %120, i32 %122,
                                      i64 %126, i32 %128,
                                      i64 0,
                              %struct.CUstream_st* null)
%tobool43 = icmp ne i32 %call42, 0
br i1 %tobool43, label %kcall.end45,
                label %kcall.configok44

kcall.configok44:
%139 = load i32, i32* %ni.addr, align 4
%140 = load i32, i32* %nj.addr, align 4
%141 = load i32, i32* %nk.addr, align 4
%142 = load i32, i32* %nl.addr, align 4
%143 = load i32, i32* %nm.addr, align 4
%144 = load float*, float** %C_gpu, align 8
%145 = load float*, float** %D_gpu, align 8
%146 = load float*, float** %F_gpu, align 8
call void @_Z11mm3_kernel2iiiiiPfS_S_(i32 %139, i32 %140,
                                      i32 %141, i32 %142,
                                      i32 %143, float* %144,
                                      float* %145, float* %146),
                                      !Redundancy !3

br label %kcall.end45
...

!3 = !{!"Inputs &C_gpu&D_gpu Outputs &F_gpu"}
```
Listing 2: Compiled code

```
1   __global__ void majorityVoting2(float *d_A1,
2                                    float *d_A2,
3                                    float *d_A3,
4                                    float *Output,
5                                    int size){
6       unsigned int i = blockDim.x * blockIdx.x
7                          + threadIdx.x;
8       if(i < size){
9          if(d_A1[i] == d_A2[i]){
10            Output[i] = d_A1[i]
11         } else {
12            Output[i] = d_A3[i]
13         }
14      }
15  }
16
17  ...
18  DATA_TYPE *A_gpu;
19  ... // Other declarations
20
21  cudaMalloc((void **)&A_gpu, sizeof(DATA_TYPE) * NI * NK);
22  ... // Other allocations
23
24  cudaMemcpy(A_gpu, A, sizeof(DATA_TYPE) * NI * NK,
25                       cudaMemcpyHostToDevice);
26  .. // Other Memory transfers
27
28  dim3 block(DIM_THREAD_BLOCK_X, DIM_THREAD_BLOCK_Y);
29  dim3 grid1(DIM_GRID_X, DIM_GRID_Y);
30
31  mm3_kernel1<<<grid1, block >>>(A_gpu, B_gpu, E_gpu);
32  cudaThreadSynchronize();
33
34
35  mm3_kernel2<<<grid2, block >>>(C_gpu, D_gpu, F_gpu);
36  DATA_TYPE *F_gpu1;
37  cudaMalloc((void **)&F_gpu1,
38                 sizeof(DATA_TYPE) * NI * NK);
39  cudaMemcpy(F_gpu1, F, sizeof(DATA_TYPE) * NI * NK,
40                       cudaMemcpyDeviceToDevice)
41
42  mm3_kernel2<<<grid2, block >>>(C_gpu, D_gpu,
43                                 F_gpu1);
44
45
46
47  DATA_TYPE *F_gpu2;
48  cudaMalloc((void **)&F_gpu2,
49                 sizeof(DATA_TYPE) * NI * NK);
50  cudaMemcpy(F_gpu2 F, sizeof(DATA_TYPE) * NI * NK,
51                       cudaMemcpyDeviceToDevice)
52  mm3_kernel2<<<grid2, block >>>(C_gpu, D_gpu,
53  F_gpu2);
54  cudaThreadSynchronize();
55
56
57  majorityVoting <<<grid , block >>>(F_gpu, F_gpu1,
58      F_gpu2, F_gpu,
    NI * NK);
59
60
61  mm3_kernel3<<<grid3, block >>>(E_gpu, F_gpu, G_gpu);
62  cudaThreadSynchronize();
63
64  }
```

Listing 4: Target code

```
1   void main(){
2   ...
3   DATA_TYPE *A_gpu;
4   ... // Other declarations
5
6   cudaMalloc((void **)&A_gpu, sizeof(DATA_TYPE) * NI * NK);
7   ... // Other allocations
8
9   cudaMemcpy(A_gpu, A, sizeof(DATA_TYPE) * NI * NK,
10                       cudaMemcpyHostToDevice);
11  .. // Other Memory transfers
12
13  dim3 block(DIM_THREAD_BLOCK_X, DIM_THREAD_BLOCK_Y);
14  dim3 grid1(DIM_GRID_X, DIM_GRID_Y);
15
16  mm3_kernel1<<<grid1, block >>>(A_gpu, B_gpu, E_gpu);
17  cudaThreadSynchronize();
18
19  mm3_kernel2<<<grid2, block >>>(C_gpu, D_gpu, F_gpu);
20  #pragma redundant in C_gpu, D_gpu out F_gpu
21  cudaThreadSynchronize();
22
23  mm3_kernel3<<<grid3, block >>>(E_gpu, F_gpu, G_gpu);
24  cudaThreadSynchronize();
25
26  }
```

Listing 3: Initial code

3.2.1. Compiler Directive

In order to get the kernel function to be replicated from the programmer, our compiler scheme defines a compiler directive. By utilizing the directive, the programmer can annotate the function as well as its input and output variables. The syntax of the directive is as follows:

$$\#pragma\ redundant\ in\ \text{<input>}\ out\ \text{<output>}$$

Listing 2 presents how customized Clang generates the annotated function call for the target code given in Listing 1. The annotated function call has metadata, *Redundancy*, which indicates both the input and the output of the function call. The attached metadata can be used in the optimizer phase.

3.2.2. Output Replication

In order to store the result values for all the function executions, that is, one original and two redundant executions, we replicate the output variable of the annotated GPU kernel (given as ① in Listing 4). This phase consists of three steps:

1. Variable declaration: We define two more variables in the type of the output variable of the kernel function to be replicated in the CPU code (Line 36 and Line 47 in Listing 4)
2. Memory allocation: We allocate space in the GPU by utilizing *cudaMalloc* function calls (Line 37 and Line 48 in Listing 4).
3. Initialization: Since we need to initialize the output variables due to the utilization of the initial values in the target function executions, we copy the data values from the original output variable into the redundant copies (Line 39 and Line 50 in Listing 4). Since we have the initialized values in the GPU memory, we utilize the device to device memory copy operation (via *cudaMemcpyDevicetoDevice* parameter) to avoid the overhead due to copy from the host CPU to the GPU device.

3.2.3. Kernel Function Call Replication

Since our aim is to execute the target kernel function redundantly, we include two more function calls in addition to the original one (given as ② in Listing 4). We provide the redundant copies of the output variables, which are created before, as the output parameters of the redundant function calls. The other parameters including the grid size, the block size, and the input variables remain the same as the original function call.

3.2.4. Majority Voting Implementation

After executing the redundant copies of the kernel functions with redundant output variables, we need to compare their results to detect and correct the potential errors. Therefore, we implement a majority voting function, which produces a single output by comparing the three outputs generated by the redundant function executions (given as ③ in Listing 4).

We implement the majority function as a GPU kernel since the outputs are already in the GPU global memory, and also the parallel execution can accelerate the comparison operations. In our compiler implementation, we need to modify both the host code and the device code. While the device code is updated by the addition of the function body, the modified host code includes the majority function call with the argument setup. Since a GPU application may contain multiple data types, we need multiple majority function implementations. We use the built-in type ID of the type of the output in the name of the majority voting function and call the appropriate one depending on the output's data type.

Our *majority voting function* consists of five parameters. The first three of them are outputs from kernel function executions. The remaining are the final output and the size of the marked output. There are two steps of our majority voting function: Thread ID calculation and Comparison. *Thread ID calculation* is very straightforward as it can be observed on line 6 in Listing 4. The next step, *Comparison*, lies on the assumption that at least two values are equal to each other. Therefore only one comparison would be sufficient. The comparison is between the first and second output. If they are equal to each other, we will assign the first value to the output. If they are not, we can say that the third value will be equal to either one based on our assumption. Therefore, we can assign the third value to the output without further comparison.

4. Experimental Study

In this section, we present the experiments for our redundant executions after providing the SDC rates obtained from our fault injection experiments. Firstly, we explain our experimental setup and evaluation methodology, and then we present our results.

4.1. Experimental Setup

We utilize the GPU programs from PolyBench benchmark suite [14] and evaluate the effects of the redundant execution on the kernel functions of the target programs. Specifically, we choose five applications including *Bicg*, *Correlation*, *Covariance*, *Fdtd2d*, and *Gramschmidt*. Firstly, we perform fault injection experiments for 15 kernel functions from those five programs and obtain the SDC rates as the soft error vulnerability metric. Then, we compile the codes that are annotated to specify the functions to be replicated.

We choose the NVIDIA Pascal architecture [15] for our evaluation platform. We build our compilation framework on Clang compiler version 10.0 and generate the target binaries for our redundant execution scenarios by compiling the modified Clang version. To collect the execution times for the kernel functions and the specific operations (e.g., memory copy, kernel function execution) during the program execution, we utilize nvprof [16] and NVIDIA Nsight Compute [17], which are NVIDIA built-in tools for the CUDA programs. On the other hand, due to the performance and the compatibility reasons, we compile our target programs with nvcc compiler [18] to perform the fault injection experiments. We use 1000 fault injections per each kernel function by using a statistical approach [19] with the confidence level of 95% and an error margin of 3%.

4.2. Experimental Results

We perform fault injection experiments to obtain the most vulnerable kernel function(s) in our target programs. While our experiments report Masked, Crash, and SDC rates, we only utilize the SDC rates as the soft error vulnerability metric of the execution. Figure 4 presents the SDC rates for each kernel function revealed from our fault injection experiments. Additionally, Table 1 presents the execution times of the corresponding kernels, which we refer to in our redundant execution discussion.
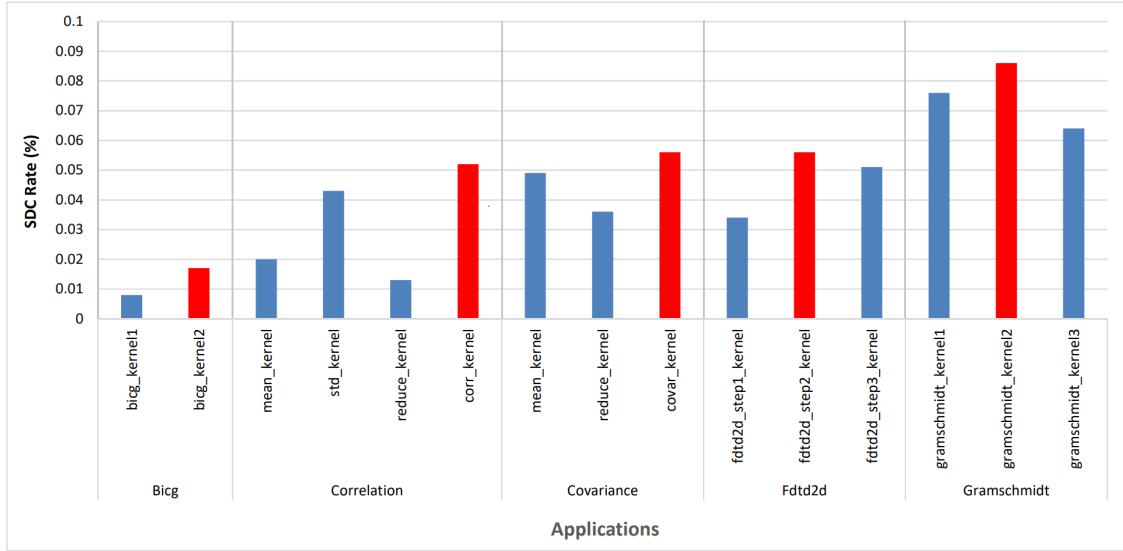
Figure 4: SDC rates for the kernel functions.

Table 1
Execution times for the kernel functions.

Application Name	Kernel Name	Execution Time (s)
Correlation	mean_kernel	0.003
	std_kernel	0.003
	reduce_kernel	0.004
	corr_kernel	5.945
Covariance	mean_kernel	0.003
	reduce_kernel	0.001
	covar_kernel	6.174
Bicg	bicg_kernel1	8.056×10^{-3}
	bicg_kernel2	22.496×10^{-3}
Fdtd2d	fdtd2d_step1_kernel	8.056×10^{-3}
	fdtd2d_step2_kernel	2.703×10^{-3}
	fdtd2d_step3_kernel	3.043×10^{-3}
Gramschmidt	gramschmidt_kernel1	0.922×10^{-3}
	gramschmidt_kernel2	0.008×10^{-3}
	gramschmidt_kernel3	4.344×10^{-3}

After determining the vulnerability of the kernel functions, we select the most vulnerable one with the highest SDC rate for each GPU application, which is marked with red in Figure 4. Then we perform redundant executions with different redundancy schemes by employing our

RMT framework. Specifically, we execute our target programs with the following scenarios: 1) No Redundancy, where we run the program as it is, 2) Full Redundancy, where we run all the kernel functions in the program redundantly, that is, three times, 3) One kernel redundant, where we select only one kernel and run that redundantly.
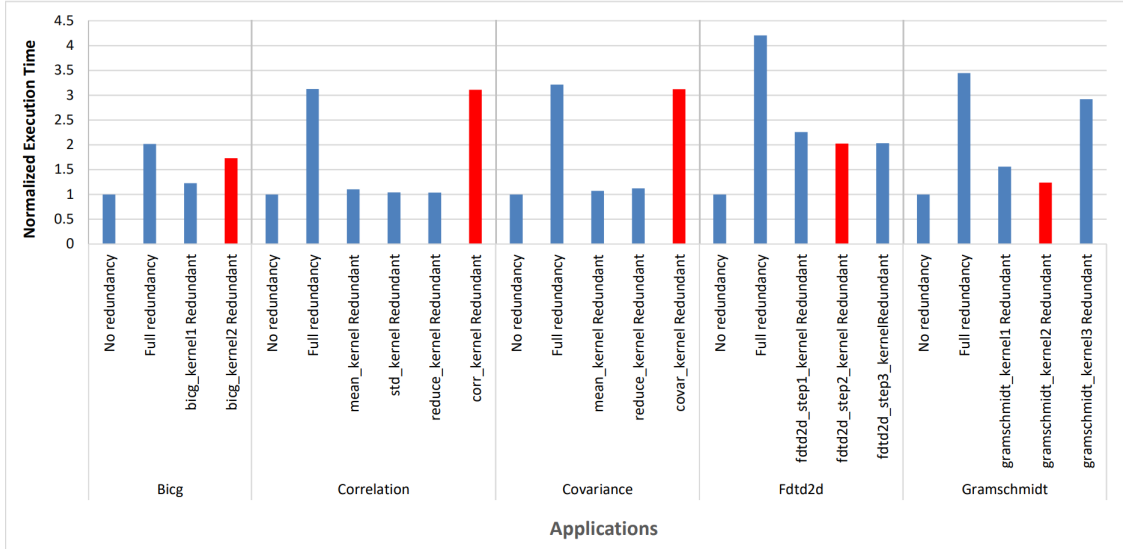

Figure 5: Normalized execution times for the redundant executions.

Figure 5 shows the normalized execution times of our target applications with different redundant schemes. We mark the execution scenario that the kernel function with the highest SDC rate is replicated in the figure for each program, e.g., *bicg_kernel2* in *Bicg*, *corr_kernel* in *Correlation*. If we look at *Bicg*, with the highest SDC rate for the *bicg_kernel2* function, we can see that the redundant execution, where the *bicg_kernel2* is replicated, takes significantly shorter time than the full redundant case. Compared to the full redundancy, our selective scheme yields performance gain. On the other hand, *Correlation* and *Covariance* do not behave in the same way. The *corr_kernel* of the *Correlation* with the highest SDC rate, has also the highest execution time, which dominates the other kernels. Therefore we can say that the difference between the Full redundancy and only *corr_kernel* redundancy is negligible as it can be observed in the Figure 5. Even though, the *corr_kernel* has the highest SDC rate, the SDC rate of the *std_kernel* is not so different (see Figure 4). Since the execution time of the *std_kernel* is so small in comparison to the execution time of the *corr_kernel* (see Table 1), it would be beneficial if we replicate both of them in order to decrease the SDC rate without significant performance degradation. Similarly, in *Covariance*, the *covar_kernel* dominates all the application since the execution times of the other kernels are negligible. On the other hand, the *mean_kernel* has close SDC rate to the *covar_kernel* as it can observed in Figure 4. If we replicate both of those functions, we achieve potentially lower vulnerability with negligible execution time difference. We must keep in mind that the time difference between the Full redundancy and only *covar_kernel* redundancy is tiny. Therefore, we can say that the best redundancy option is the Full redundancy. By observing the trends in *Correlation* and *Covariance*, we can say that for the programs that one kernel function

dominates the others in terms of the execution time, our selective scheme does not help, and applying full redundancy can be a good option.

Different from the other applications, the proportion between the execution time and the SDC rate is inverse in *Fdtd2d* and *Gramschmidt*. For example, while the *fdtd2d_step2_kernel* has the highest SDC rate (see Figure 4), its execution time is the shortest (see Table 1) among the other kernel functions in the application. Among all the redundancy options of *fdtd2d* shown in Figure 5, the option with the *fdtd2d_step2_kernel* redundant has the lowest execution time. Since the kernel has the highest SDC rate, we get a decent amount of SDC rate reduction by sacrificing relatively less execution time. On the other hand, the SDC rate of the *fdtd2d_step3_kernel* is close to the *fdtd2d_step2_kernel*, and the difference between their execution times is not large (see Table 1). Therefore, we can say that it would be beneficial if we replicate both of them. *Gramschmidt* has similar characteristics. For instance, the *gramschmidt_kernel2* has the highest SDC rate, yet it has the lowest execution time among the other kernels of the application. Therefore replicating only this function or two functions, namely the *gramschmidt_kernel1* and the *gramschmidt_kernel2*, with similar execution times can be beneficial in terms of both performance and reliability.
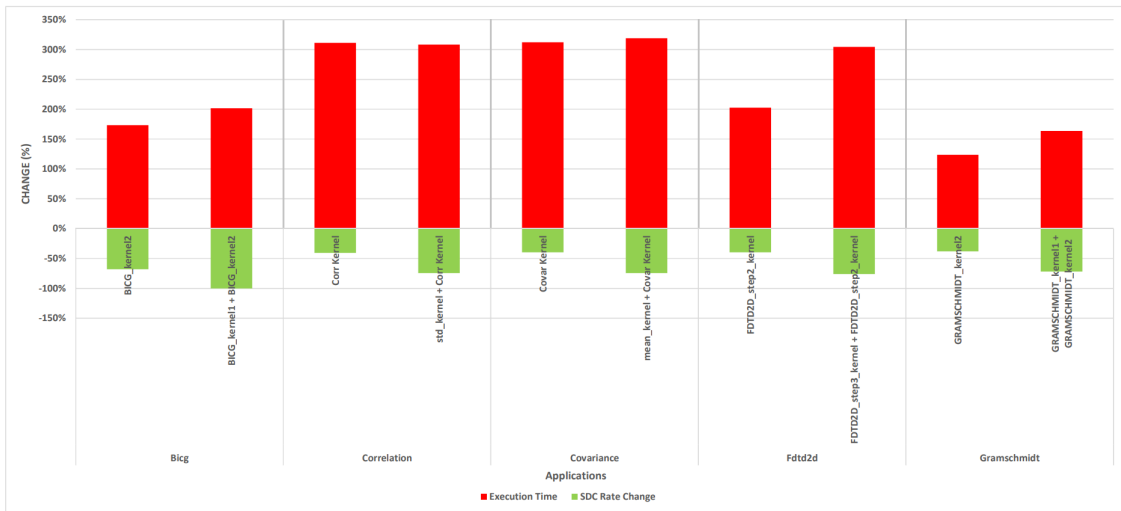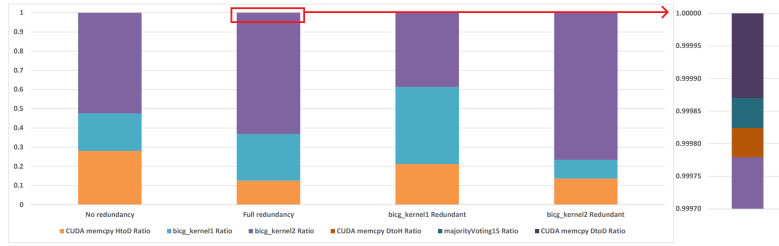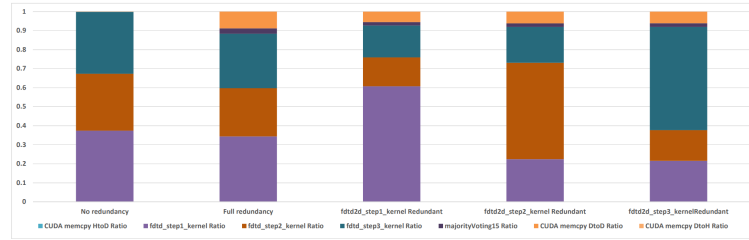

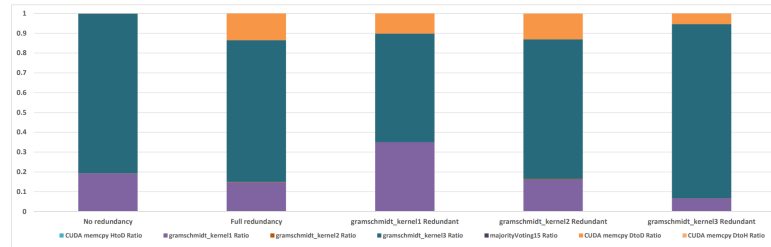**Figure 6:** The change in percentage of SDC rates and execution times for the redundant executions.

While we discuss the performance of the redundant execution schemes previously, we also want to analyze the performance and reliability gains together. To evaluate the performance-reliability tradeoff between the different replication schemes, we demonstrate the change (in percentage) for both the SDC rates as the vulnerability metric and the execution times in Figure 6. We assume that our redundancy method, based on triple replication, provides full protection, and our redundantly executed code will not cause any erroneous cases by masking the faults. We calculate the SDC rates accordingly. We utilize two redundant execution scenarios for each application including the replication of only the most vulnerable kernel function and the replication of the most vulnerable two kernel functions. For *Bicg*, essentially, we present the Full redundancy and the most vulnerable function redundancy schemes since it consists of

(a) Bicg

(b) Fdtd2d

(c) Gramschmidt

Figure 7: Execution time profile of each function.

two kernel functions. If we perform Full redundancy, where we replicate two kernel functions, the SDC rate drops to zero (100% decrease). However, the reliability gain obtained from the replication of the *bicg_kernel2* is limited ($\sim$ 65%). Therefore we can say that if we have no time limitation and/or no tolerance to SDCs, it would more useful to apply full redundancy. Otherwise, using only *bicg_kernel2* redundancy would be beneficial. For both *Correlation* and *Covariance*, as discussed earlier, the execution time does not differ while the vulnerability gain gets larger for the two-function replication case. Even though, in Figure 6, there is a large percentage difference between only *fdtd2d_step2_kernel* redundant option and both of *fdtd2d_step2_kernel* and *fdtd2d_step3_kernel* redundant option, the absolute difference is not significant since the execution times of the kernels are short (see Table 1). However, for the cases, where *Fdtd* is executed with a very large amount of data, and the execution incurs longer times, the performance gain obtained from the replication of only the most vulnerable function becomes significant. For *Gramschmidt*, we can clearly see the tradeoff between the vulnerability and the performance for the alternative redundancy schemes. The replication of only the most vulnerable function, i.e., *gramschmidt_kernel2*, provides almost the same ($\sim$ 30%) performance and vulnerability gains. Therefore, one needs to consider the requirements

of the system including the execution time and the reliability, and make a decision about the redundancy level accordingly.

Figure 7 presents the execution time profile of each function for the redundancy scenarios. For each redundant execution case, we measure the percentage of the operations performed during the execution. Specifically, we profile the kernel function executions, the majority function, and the memory copy operations including the copy of the input from CPU to GPU (*CUDA memcpy HtoD*), the copy of the output from GPU to CPU (*CUDA memcpy DtoH*), and the redundant output copy operations form GPU to GPU (*CUDA memcpy DtoD*). We can see that the most of the time is spent during the kernel function executions. While the redundant output copy operations also take significant time in *Fdtd2d* (Figure 7b) and *Gramschmidt* (Figure 7c), the percentage of those operations is small for the other programs. We provide the small percentages in a more detailed view for Bicg (see Figure 7a), however, we omit the details for *Correlation* and *Covariance*, which both spend almost all the time in the dominant kernel function executions, *corr_kernel* and *covar_kernel*, respectively.

Although we include additional memory operations and majority function as well as redundant kernel functions in our redundant scenarios, the main reason of the increase in the execution time is the replicated function executions. The majority voting function and the memory operations do not take significant time in comparison to the kernel executions. Therefore, we need to focus on reducing the time spent for the redundant kernel executions. It is possible to utilize the parallel execution units of the GPU by executing the redundant copies in parallel either using the streams or replicating the number of threads working on the redundant copies.

5. Conclusion

In this study, we propose a partial redundant multithreading mechanism based on the soft error vulnerability of GPGPU applications and perform a trade-off analysis between performance and reliability. Firstly, we run fault injection experiments and collect SDC rates for the programs. Based on the SDC rates, we determine the kernel function to be replicated. Using our custom *pragma* annotation, our custom LLVM pass generates additional kernel function calls, required memory operations, and *majority voting* function. Our experiments indicate that the majority voting function and the memory operations such as *cudaMemCpy* do not consume significant time. The main reason for the performance overhead is the execution time of the kernel functions. Therefore, for future work, we will focus on increasing the execution overlaps of the redundant kernel function executions.

Acknowledgments

References

[1] T. M. Aamodt, W. W. L. Fung, T. G. Rogers, M. Martonosi, General-Purpose Graphics Processor Architecture, 2018.

[2] G. Reis, J. Chang, N. Vachharajani, R. Rangan, D. August, Swift: software implemented fault tolerance, in: International Symposium on Code Generation and Optimization, 2005, pp. 243–254. doi:10.1109/CGO.2005.34.

[3] M. Didehban, A. Shrivastava, Nzdc: A compiler technique for near zero silent data corruption, in: Proceedings of the 53rd Annual Design Automation Conference, DAC '16, Association for Computing Machinery, New York, NY, USA, 2016. URL: https://doi.org/10.1145/2897937.2898054. doi:10.1145/2897937.2898054.

[4] J. Wadden, A. Lyashevsky, S. Gurumurthi, V. Sridharan, K. Skadron, Real-world design and evaluation of compiler-managed gpu redundant multithreading, in: 2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA), 2014, pp. 73–84. doi:10.1109/ISCA.2014.6853227.

[5] M. Bohman, B. James, M. J. Wirthlin, H. Quinn, J. Goeders, Microcontroller compiler-assisted software fault tolerance, IEEE Transactions on Nuclear Science 66 (2019) 223–232. doi:10.1109/TNS.2018.2886094.

[6] C. Kalra, F. Previlon, N. Rubin, D. Kaeli, Armorall: Compiler-based resilience targeting gpu applications, ACM Trans. Archit. Code Optim. 17 (2020). URL: https://doi.org/10.1145/3382132. doi:10.1145/3382132.

[7] L. Yang, B. Nie, A. Jog, E. Smirni, Enabling software resilience in gpgpu applications via partial thread protection, in: 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE), 2021, pp. 1248–1259. doi:10.1109/ICSE43902.2021.00114.

[8] I. Oz, O. F. Karadas, Regional soft error vulnerability and error propagation analysis for gpgpu applications, Journal of Supercomputing (2021) 1–1. doi:10.1007/s11227-021-04026-6.

[9] P. Shivakumar, M. Kistler, S. Keckler, D. Burger, L. Alvisi, Modeling the effect of technology trends on the soft error rate of combinational logic, in: Proceedings of International Conference on Dependable Systems and Networks (DSN), 2002.

[10] C. Weaver, J. Emer, S. S. Mukherjee, S. K. Reinhardt, Techniques to reduce the soft error rate of a high-performance microprocessor, in: 31st Annual International Symposium on Computer Architecture (ISCA), 2004.

[11] I. Oz, S. Arslan, A survey on multithreading alternatives for soft error fault tolerance, ACM Comput. Surv. 52 (2019). URL: https://doi.org/10.1145/3302255.

[12] C. Lattner, V. Adve, Llvm: a compilation framework for lifelong program analysis amp; transformation, in: International Symposium on Code Generation and Optimization, 2004. CGO 2004., 2004, pp. 75–86. doi:10.1109/CGO.2004.1281665.

[13] C. Lattner, Llvm, 2011. URL: https://www.aosabook.org/en/llvm.html, last accessed 27 August 2021.

[14] L.-N. Pouchet, Polybench/c, 2016. URL: https://web.cse.ohio-state.edu/%7epouchet.2/software/polybench/, last accessed 27 August 2021.

[15] Nvidia, pascal architecture whitepaper, 2021. URL: https://www.nvidia.com/en-us/data-center/resources/pascal-architecture-whitepaper.

[16] Nvidia nvprof profiling tool, 2021. URL: https://docs.nvidia.com/cuda/profiler-users-guide/index.html#nvprof-overview.

[17] Nvidia nsight compute, 2021. URL: https://developer.nvidia.com/nsight-compute.

[18] Nvidia, cuda llvm compiler, 2021. URL: https://developer.nvidia.com/cuda-llvm-compiler.

[19] R. Leveugle, A. Calvez, P. Maistri, P. Vanhauwaert, Statistical fault injection: Quantified error and confidence, Proceedings of the Conference on Design, Automation and Test in Europe (DATE), 2009.