Towards analytically evaluating the error resilience of GPU Programs

Abdul Rehman Anwer, Guanpeng Li, Karthik Pattabiraman
University of British Columbia
{abanwer, gpli, karthikp}@ece.ubc.ca
Siva Kumar Sastry Hari, Michael Sullivan, Timothy Tsai
NVIDIA
{shari, misullivan, timothyt}@nvidia.com

Abstract—General purpose Graphics Processing Units (GPUs) have become popular for many reliability-conscious uses including their use for high-performance computation, machine learning algorithms, and business analytics workloads. Fault injection techniques are generally used to determine the reliability profiles of programs in the presence of soft errors, but these techniques are highly resource and time intensive. Trident, an analytical model, was developed for predicting SDC probabilities of CPU programs based on its 3-level modeling technique. However, it is not clear how accurate such analytical modeling is in predicting the SDC probabilities of GPU programs, which are highly parallel and have a very different programming model from CPU programs. In this paper, we adopt the original Trident methodology for modeling error propagation in CUDA-based GPU applications, and examine the accuracy of SDC predictions versus fault injection experiments. We find that there is a discrepancy between the prediction results and the fault injection results due to differing memory behavior in GPU programs. We also observe that a large number of threads in the GPU applications increases the information to be profiled, which complicates profiling in Trident, resulting in significant slowdown. We analyze the results, investigate the bottlenecks of Trident in GPU applications, and propose potential solutions to mitigate the problems.

Keywords—Error Propagation, Soft Error, Silent Data Corruption, Error Resilience, Program Analysis

I. INTRODUCTION

Graphics Processing Units (GPUs) have been widely adopted as accelerators for applications such as scientific applications, business-analytic workloads, and self-driving cars due to their high performance and power efficiency. On the other hand, transient hardware faults (i.e., soft errors) are predicted to increase in future processors including GPUs due to growing system scale, progressive technology scaling, and lowering operating voltages [1]. In the past, such faults were masked in high-reliability systems through hardware-only solutions such as dual modular redundancy (DMR) and circuit hardening. However, these techniques are becoming increasingly challenging to deploy as they incur significant overheads in performance and energy consumption. This problem is exacerbated in the case of GPUs as they typically have a larger number of execution units and register file than CPUs [2]. Therefore, researchers have postulated that future GPUs will expose hardware faults to the software and expect the software to tolerate them [3].

One consequence of such hardware errors is incorrect program output, or silent data corruption (SDC), which are difficult to detect and can have severe consequences [1]. Fault injection (FI) has been commonly employed to estimate the SDC probability of programs. FI perturbs program state during execution and checks the program output to detect failures (if any). Thousands of program executions are typically required to gain statistical significance of the results—therefore, FI can be tremendously slow and challenging to deploy in practice [4], [5], [6]. As a result, researchers have attempted to analytically model error propagation to evaluate SDC probabilities quickly and accurately [7], [8], [9].

Li et. al. [6] proposed Trident, an automated technique that analytically models error propagation in programs to predict their SDC probabilities without any FIs. While Trident is shown to be accurate and fast in single-threaded CPU programs, it is not clear whether the Trident methodology works in GPU programs, which are highly parallel and have a very different programming model. Further, GPUs have many differences with CPUs in terms of their memory hierarchy, which can affect the accuracy of the modeling.

This paper examines the feasibility of extending Trident to model error propagation in GPU programs, designs experiments to explore its accuracy and performance, and discusses possible solutions to improve Trident for GPU programs. Our main contributions in this paper are as follows:

- We quantify the effect of the differences between GPU and single threaded CPU applications in terms of modeling error propagation in Trident. We modify Trident to predict the SDC probabilities of GPU programs, without modeling the shared memory. We then design a set of experiments to compare its accuracy with respect to FI.
- We observe that there are discrepancies between the predictions and the actual FI results, and they vary based on the granularity of memory sharing among threads. In our experiments, the difference in predicted SDC from FI-based results ranges from 3.9% to 5.6% for kernels that do not use shared memory, and from 11.4% to 33.5% for kernels that use shared memory.
- We also find that while Trident is faster than FI for most GPU applications, it incurs significant performance overheads when applied to multi-threaded GPU...
environment and hence achieves much less speedup compared to implementation of TRIDENT for CPU applications.

- We find that shared memory between threads complicates error propagation in memory, resulting in very large dependency graphs to process, which is not scalable for analyzing HPC applications.

- Finally, we propose several mitigation plans to improve both the accuracy and performance of TRIDENT when applied to GPU programs.

II. BACKGROUND

In this section, we first present our fault model, then define the terms we use, followed by a brief primer on GPU architecture and the compiler infrastructure we use for FI experiments and analysis. Finally, we provide a brief overview of the Trident technique from our prior work [6], as it forms the basis of this paper.

A. Fault Model

In this paper, we consider transient hardware faults that occur in the computational elements of the GPU, including architectural registers and functional units. We do not consider faults in the memory or caches, as we assume that these are protected with error correction codes (ECC). We do not consider faults in the GPU’s control logic in our framework. Neither do we consider faults in the instructions’ encodings. Finally, we assume that the program does not jump to arbitrary illegal addresses due to faults during the execution, as this can be detected by control-flow checking techniques [10]. However, the program may take a faulty legal branch (the execution path is legal but the branch direction can be wrong due to faults propagating to it). Our fault model is in line with other work in the area [11], [7], [4], [8].

B. Terms and Definitions

Fault Occurrence: The event corresponding to the occurrence of a hardware fault in the processor. The fault may or may not result in an error.

Fault Activation: The event corresponding to the manifestation of the fault to the software, i.e., the fault becomes an error and corrupts some portion of the software state (e.g., register, memory location). The error may or may not result in a failure (i.e., SDC, crash or hang).

Crash: The raising of a hardware trap or exception due to the error, because the program attempted to perform an action it should not have (e.g., read outside its memory segments). The OS terminates the program as a result.

Silent Data Corruption (SDC): A mismatch between the output of a faulty program run and that of an error-free execution of the program.

Benign Faults: Program output matches that of the error-free execution even though a fault occurred during its execution. This means either the fault was masked or overwritten by the program.

Error propagation: Error propagation means that the fault was activated, and has affected some other portion of the program state, say ‘X’. In this case, we say the fault has propagated to state X. We focus on the faults that affect the program state and therefore consider error propagation at the application level.

SDC Probability: We define the SDC probability as the probability of an SDC given that the fault was activated – other work uses a similar definition [7], [5], [12], [13].

C. GPU Architecture and Programming Model

In this study, we focus on GPGPU applications that are implemented on the NVIDIA Compute Unified Device Architecture (CUDA), a widely adopted programming model and toolset for GPUs. In the CUDA programming model, a GPGPU application consists of a control program running on the CPU, and a computation program called the kernel running on the GPU(s), in parallel with the CPU. The kernel is implemented as a collection of functions in a C-like language, but with CUDA-specific annotations. CUDA kernels adopt the single instruction multiple thread (SIMT) execution model to exploit the massive parallelism of GPU devices. From a software perspective, the CUDA programming model abstracts the SIMT model into a hierarchy of kernels, blocks and threads. The CUDA kernels consist of blocks, which consist of threads. This hierarchy allows various levels of parallelism such as fine-grained data parallelism, coarse-grained data parallelism and task parallelism.

From a hardware perspective, blocks of threads run on streaming multiprocessors (SMs) which have on-chip shared memory for threads inside the same block. Within a block, threads are launched in fixed groups of 32 threads, which are called warps. Threads in a warp execute the same sequence of instructions but with different data values. Each GPU has its own memory space that is distinct from the host CPU’s memory. In the CUDA programming model, there are various kinds of memory: (1) global, (2) constant, (3) texture, (4) shared, and (5) thread-local memory allocations and accesses. Global, constant, and texture memory accesses that miss in the on-chip caches are taken from the large and comparatively slow device memory. The shared memory space is software managed, and it is much smaller and built on chip, hence it is much faster to access. Thread-local memory is typically stored in the fast register file, though compiler-inserted spill and fill operations can occasionally place thread-local state into slower areas of the storage hierarchy. CUDA applications need to be aware of the memory hierarchy to efficiently access the GPU memory.

D. LLVM Compiler

In this paper, we use the LLVM compiler [14] to perform our program analysis and FI experiments and to implement our model. LLVM uses a typed intermediate representation (IR) that can easily represent source-level constructs. In particular, it preserves the names of variables and functions, which makes source mapping feasible. This allows us to perform a fine-grained analysis of which program locations cause certain failures and map them to the source code. We have shown that LLVM IR is accurate for performing FI studies, for SDC measurements [13], which is our focus in this paper.
E. Trident

Figure 1 gives an overview about the workflow of Trident. Trident requires three inputs (1) LLVM IR of the program code, (2) Instructions considered output i.e., instructions determining if an SDC occurred, (3) program input to execute program for profiling [6]. Trident estimates the SDC probabilities of the individual instructions as well as the whole program.

![Workflow of Trident][1]

Running Trident can be divided into two phases: (1) Profiling: In the profiling phase, Trident performs static and dynamic analysis of the program such as instruction counts, data dependency, branch probabilities etc. (2) Inferencing: In the inferencing phase, Trident uses the information profiled to track error propagation that leads to SDCs, and hence determine the SDC probabilities of individual instruction as well as the whole program.

Trident models error propagation in three levels, they are (1) static instruction level, (2) control flow level, and (3) memory level. Each level is modeled by a sub-model which in turn abstracts the error propagation into probabilistic events. The three sub-models synergistically work on top of each other to track error propagation from the error occurrence to the final program output as follows.

1. Static-instruction sub-model ($f_s$): static data-dependent instruction sequence is a sequence of statically data-dependent instructions that are present in the same basic block. A fault activated in a static data-dependent instruction sequence propagates till its end [15]. This sub-model of Trident computes propagation probability of fault activated in any instruction of static data-dependent instruction sequence traveling to its end. For example in Figure 2a fault activated at load instruction has 3% probability of propagating to end of static data-dependent instruction sequence. Propagation probabilities of each instruction are individually calculated using information collected from profiling and predefined rules associated with each instruction type.

![Examples of Trident sub-models][2]

2. Control-flow sub-model ($f_c$): If a static data-dependent instruction sequence ends with a branch instruction like in the case of Figure 2a the fault activated in static data-dependent instruction sequence has the potential to diverge the control flow of the program as compared to fault-free execution. This control flow divergence may lead to memory corruption as a store that was going to be executed in fault-free execution may be executed and vice versa which results in contents of memory being different from fault-free execution. Control flow sub-model of Trident identifies such store instruction and their probabilities to be affected due to a fault activation [15].

For example in Figure 2b if a fault is activated in bb0, then memory can be corrupted in two ways (1) bb4 was to be executed but due to the error T branch was taken instead of F (2) T branch was taken but due to error F branch was taken and subsequently bb4 was executed. Control-flow sub-model finds the probability of both cases and aggregates them to find the total probability of memory corruption due to store in bb4. Using this logic, equations can be derived to find the possibility of store being corrupted ($P_c$), Eq. 1. shows the equation for non-loop terminating instructions.

\[
P_c = \frac{P_e}{P_d}
\]  

$P_d$ is the probability of execution of the branch dominating the store and $P_e$ is the probability of execution of store instruction in fault-free scenario. Similarly Eq. 2. shows the equation for loop terminating instructions.

\[
P_c = P_b \times P_e
\]  

$P_b$ is the probability of execution of back-edge of the branch and $P_e$ is probability of execution of store instruction which is dominated by the back-edge.

3. Memory sub-model ($f_m$): If a store is corrupted due to a fault it may lead to error propagating to other parts of the program that read from that memory. Memory sub-model of Trident uses knowledge of corrupted stores and memory dependencies of the stores to track error propagation. All the dynamic memory loads and stores are profiled during profiling phase of Trident to form a memory dependency graph, which is then used to calculate the probability of fault propagation from corrupted stores to program output [15].

The main algorithm of Trident is shown in Algorithm 1, the three sub-models are recursively called in the inferencing phase of Trident, for each instruction in which an error can occur. After computing the SDC probabilities of each individual instructions, an aggregated SDC probability is calculated as the overall SDC probability of the program. Overall, Trident is shown to be able to predict SDC probabilities almost as close as FI, but only takes a fraction of the time FI requires [6].

III. RELATED WORK

Modeling of error propagation has been widely employed to estimate the resilience of CPU programs. The faster execution of modeling techniques as compared to FI makes them ideal to incorporate in the software development cycle. Shoestring [7] was one of the first techniques that instead of using FI used compile-time analysis and symptom based error detection techniques to model the resilience of instructions.

Sridharan et al. [9] introduced a new analytical model, program vulnerability factor (PVF) to capture the masking properties of the program related to architecture-level faults without FI. This technique has the drawback that it does not distinguish between SDCs and crashes, leading to a loss in...
A significant amount of research has gone towards finding the resilience of GPU platforms by using FI. Yim et al. [18] developed one of the first FI tools for GPU applications, which operated at the source code level. Fang et al. [19] developed GPU-Qin, which is a fault injector that operates on CUDA assembly code level using CUDA-gdb. Subsequently, Hari et al. [20] also developed a fault injector (SASSIFI) that works at the SASS assembly code level using the Nvidia compiler (SASSI). Li et al. [21] developed a FI tool (LLFI-GPU) which operates at the LLVM IR level. The main advantage of LLFI-GPU is that it is independent of the specific GPU platform or architecture, as long as the compiler provides support for the platform. LLFI-GPU is used as the FI baseline in this paper.

Running FI experiments on GPUs is very resource intensive as GPU applications invoke thousands of threads during their execution, and performing a FI experiment to cover this state space requires a lot of time. Nie et al. [22] propose a systematic way to prune FI sites based on identifying and only injecting faults in representative threads from threads displaying similar behavior, thus making FI more time efficient. However, this technique still involves performing thousands of FIs.

Tan et al. [2] propose an analytical framework to estimate the vulnerability of GPU microarchitectures due to soft errors. However, their technique does not consider the characteristics of the GPU program as it is microarchitecture dependent. Further, they do not distinguish between different failure types such as crash, SDC etc.

In very recent work, Kalra et al. [23] uses features independent of underlying GPU microarchitecture to train machine learning models to predict the SDCs in GPU programs. At a high-level, their goal is similar to our work. However, there are two differences between our work and theirs. First, they require a training phase for the model to learn the features from representative programs. It is not clear what happens if such a set is not available, or if the training programs are not representative. Second, they only compute the aggregate SDC rate of the program, and not those of individual instructions. Our work on the other hand attempts to build an analytical model of individual GPU instructions’ error resilience, and thus can be used to guide application-level protection techniques (e.g., selective duplication of the SDC-prone instructions).

### IV. Experimental Setup

In this section, we first describe the implementation of TRIDENT on GPU programs, then explain the FI method we use to measure its accuracy, and finally, present the benchmarks we use in this study.

#### A. Implementation

TRIDENT is implemented as a set of LLVM compiler plugins. We insert those passes into the NVCC compiler from Nvidia, and execute them with the GPU kernels in each benchmark, similar to the method in Li et al. [21]. We only make minor implementation changes when necessary in order to execute TRIDENT with NVCC compiler. In applying TRIDENT to GPU kernel we consider each thread as an independent execution, whose profiled values are then used together to calculate SDC probability of the kernel. In the case of kernels in which profiling of memory dependencies can not be done together for all threads as different threads access same memory, threads are randomly sampled to profile memory dependencies.

#### B. FI Method

We use LLFI-GPU [21], which is a publicly available fault injector for GPUs, to perform FIs at the LLVM IR level of GPU applications. We inject faults into the destination registers of the executed instructions to simulate faults in the computational elements of the processor as per our fault model. Further, we inject single bit flips as these are the de-facto model for emulating soft errors at the program level, and have been found to be accurate for SDCs [24]. There is only one fault injected in each run, as soft errors are rare events with respect to the time of execution of a program. Our FI method ensures that all faults are activated, i.e., read by an instruction of the program, as we define SDC probabilities based on the activated instructions (Section II). The FI method is in line with our earlier work in the area [21], [6].

---

**Algorithm 1:** Algorithm in TRIDENT [6]

```plaintext
1 functions f_s, f_c, and f_m;
Input : I: Instruction where the fault occurs
Output: P_{SDC}: SDC probability
2 p_s = f_s(I);
3 if inst. sequence containing I ends with branch I_b
   then
   \[
   \begin{align*}
   &// Get the list of stores corrupted and their prob. \\
   &<I_s, p_c>, \ldots = f_c(I_b);
   \end{align*}
   \]
   // Maximum propagation prob. is 1
4 For(<I_s, p_c>): P_{SDC} \leftarrow p_s * p_c * f_m(I_s);
5 else if inst. sequence containing I ends with store I_s
   then
6 \[ P_{SDC} = p_s * f_m(I_s); \]
```
C. Benchmarks

Table I shows the benchmarks we used for evaluation chosen from Rodinia benchmark suite. The choice of benchmarks was governed by whether they can be compiled with LLVM based infrastructure on both LLFI-GPU and TRIDENT, and that the FI experiments can be completed in reasonable time. We removed any sources of randomness in the benchmarks to get reproducible results for the FI experiments.

![Table I: Characteristics of Benchmarks](image)

## V. Results

### A. Accuracy

We perform FI experiments with LLFI-GPU, which is used to evaluate the accuracy of SDC probabilities predicted by TRIDENT. For each application, 3,000 FI runs are executed, each with a randomly chosen dynamic instruction. After each FI, we compare the data written to memory by the kernel with the contents of memory in a fault free execution, to find SDC probability of the kernel. We use the memory changes made by the kernel rather than the final output of the application, as we are only interested in finding the SDC probabilities of CUDA kernels. Sampling 3,000 dynamic instructions for FI yields tight error bars at 95% confidence level (±0.93% to ±1.79%). Moreover, it is also observed that for 3,000 FI experiment, the results are stable for all the kernels and increasing the number of FI does not cause a significant change in observed results.

The SDC probabilities predicted by TRIDENT and FI are presented in Figure 3. It can be seen that TRIDENT predicts the SDC probability of GPU kernels with varying degree of accuracy. For example, the difference between SDC probability predicted by TRIDENT and FI results for kernel 1 of Gaussian benchmark is 3.98%, while for kernel 1 of NW it is 33.57%. The benchmarks chosen for experiments can be divided into two groups in term of accuracy. The first group, consisting of both the kernels of Gaussian and BFS kernel 2, exhibits a difference of less than 6%. The second group, consisting of Pathfinder, Hotspot and both kernels of NW, shows a difference of more than 10%. The reason for the difference between two groups is that the kernels in the second group use shared memory, while those in the first group do not. Shared memory can be accessed by all the threads within a thread block, so if a fault is injected in one thread it can propagate to all the threads in the same thread block via shared memory. The effect of this error propagation between threads on TRIDENT’s prediction is further discussed in Section VI. This set of experiments clearly shows that accuracy of SDC prediction suffers, if memory dependencies among threads (in case of shared memory) are ignored while modeling error propagation.

![Figure 3: SDC probability of GPU Kernels predicted by TRIDENT and FI (Margin of Error for FI: ±0.93% to ±1.79% at 95% Confidence)](image)

### B. Overhead

The main motivation for porting TRIDENT to CUDA applications is to reduce the time required for finding SDC probabilities of the GPU kernels. As there is no parallelization in either TRIDENT or LLFI-GPU, we can use wall-clock time for performance comparison. Figure 4 shows the comparison of time required for finding SDC probability of the GPU kernels using TRIDENT and FI in which 3,000 faults are injected.

Similar to the case in section V-A, we see that there is a stark difference in performance overhead between the kernels that use shared memory and kernels that do not. In the former group, TRIDENT is significantly faster than FI (i.e., BFS and the two kernels of Gaussian). In these cases, the difference ranges from 5X to 11X. This is because, in these kernels, profiling for memory dependency and recursively applying TRIDENT can be done in a single pass, which results in a significant performance improvement when compared to FI.

![Figure 4: Performance comparison of TRIDENT and FI](image)
Individually applying the whole process to each thread would cause a substantial performance overhead in TRIDENT, as hundreds of thousands of threads can be invoked in GPU applications. To solve this problem, a subset of threads is chosen, and the TRIDENT model is applied to the reduced set of threads. While this is faster than applying TRIDENT separately for each thread, it still incurs significant overheads compared to running TRIDENT on CPU applications.

It should be noted that Figure 4 shows the time taken for 3,000 experiments in case of FI. If a more exhaustive FI experiment is performed, TRIDENT will be much more time efficient in comparison to FI.

VI. DISCUSSION

In this section, we discuss the challenges associated with adapting TRIDENT modeling to GPU programs.

Memory sharing among threads: As described in section II-C, GPU programs are inherently multi-threaded and provide the ability to share memory among threads with different levels of granularity. CUDA programming model allows (1) threads to maintain thread-local memory threads (not shared with any other thread), (2) sharing data among a subset of threads in a thread block (e.g., warp-shuffle), (3) sharing data across all the threads in a thread block using shared memory or scratchpad, (4) sharing data across all the threads in a compute kernel (using device global memory), and (5) sharing across devices using unified virtual memory (UVM). This memory sharing at multiple levels introduces the prospect of faults that are activated in one of the threads, propagating to other threads due to memory dependencies. TRIDENT was developed for single threaded applications and cannot handle this scenario, which gives rise to inaccuracies in the predicted SDC rate for CUDA kernels. Though we have found that the shuffle instruction and UVM are not used in the programs considered in this paper, it is nonetheless important to consider them for accurate modeling of error propagation, especially for highly-tuned programs.

These observations show that the current memory model of TRIDENT needs to incorporate memory sharing among threads at different levels of granularity to accurately predict SDC probabilities of CUDA kernels.

Performance: During the profiling phase, TRIDENT profiles different aspects such as memory dependency, argument values of dynamic instructions etc. During the inference phase, this information is used for finding SDC probabilities. In the case of GPUs, hundreds of thousands of threads are spawned, so profiling information such as memory dependencies and argument values for all the threads in an execution, introduces a huge memory overhead, especially if the number of instructions in the kernel, or the number of kernel invocations is large. It is observed that, for HPC applications the memory dependency record gets so large that it does not fit in the memory of even state of the art NVIDIA GPUs so profiling all information in one execution is not possible in such scenario. For example, for the LULESH benchmark, the size of profiled memory dependencies is greater than 100 GB for input size of 13 edge elements (which is significantly smaller than its default input of 45 edge elements). On the other hand, profiling each thread separately, increases the execution time of profiling phase of TRIDENT to unacceptable levels, as multiple executions of GPU application are required for every thread. Therefore, the profiling method used by TRIDENT needs to be made more memory and time efficient for GPU environment, e.g., intelligently pruning the number of threads to be profiled while keeping the memory dependencies among threads intact.

Implementation of TRIDENT: TRIDENT for GPU applications uses the LLVM compiler for profiling and instrumentation. The NVCC compiler used for compiling CUDA applications compiles source code to the PTX representation, which is then compiled to SASS code. Although NVCC is based on LLVM, it does not expose the IR representation, so the tools developed for TRIDENT cannot be directly used for GPU applications. This problem is handled by attaching a dynamic library [25] to the NVCC compiler, which intercepts calls to LLVM module, adds instrumentation code in it, and then returns that modified code to rest of the compilation chain. However, any changes made by NVCC to the sequence of instruction or how memory is used by threads when code is converted from LLVM IR representation to the PTX representation will not be visible to TRIDENT, and hence may result in inaccuracies in predicting the SDC probabilities.

VII. CONCLUSION

In this paper, we applied TRIDENT, developed for analyzing soft error propagation in CPU programs, to GPU based CUDA kernels. TRIDENT can be used to find the reliability profile of a program without using fault injections. We tested TRIDENT on 7 GPU kernels and found out that the accuracy of the predicted SDC rate by TRIDENT can widely vary depending on the level of memory sharing among the threads of the kernel. As TRIDENT was initially developed for single threaded CPU applications, it does not model shared memory. We also observed a substantial memory and performance overhead while profiling CUDA application in TRIDENT.

In future work, we intend to improve TRIDENT for GPU applications by (1) Improving the mechanism for profiling, by pruning threads from analysis, based on their similarity, while retaining the memory dependencies among threads (2) Updating the memory model of TRIDENT to incorporate memory sharing among threads such that error propagation between threads is properly modeled while predicting SDC probability.

ACKNOWLEDGEMENTS

This work was supported in part by a Strategic Project Grant and DIscovery Grant from the Natural Sciences and Engineering Research Council of Canada (NSERC), as well as a Four Year Fellowship from UBC. We thank the anonymous reviewers of SELSE for their comments.

REFERENCES


