GPU-Trident: Efficient Modeling of Error Propagation in GPU Programs

Abdul Rehman Anwer
University of British Columbia
abanwer@ece.ubc.ca

Guanpeng Li
University of Iowa
guanpeng-li@uiowa.edu

Karthik Pattabiraman
University of British Columbia
karthikp@ece.ubc.ca

Michael Sullivan
NVIDIA
misullivan@nvidia.com

Timothy Tsai
NVIDIA
timothyt@nvidia.com

Siva Kumar Sastry Hari
NVIDIA
shari@nvidia.com

***Abstract*—Fault injection (FI) techniques are typically used to determine the reliability profiles of programs under soft errors. However, these techniques are highly resource- and time-intensive. Prior research developed a model, TRIDENT to analytically predict Silent Data Corruption (SDC, i.e., incorrect output without any indication) probabilities of single-threaded CPU applications without requiring FIs. Unfortunately, TRIDENT is incompatible with GPU programs, due to their high degree of parallelism and different memory architectures than CPU programs. The main challenge is that modeling error propagation across thousands of threads in a GPU kernel requires enormous amounts of data to be profiled and analyzed, posing a major scalability bottleneck for HPC applications.**

In this paper, we propose GPU-TRIDENT, an accurate and scalable technique for modeling error propagation in GPU programs. We find that GPU-TRIDENT is 2 orders of magnitude faster than FI-based approaches, and nearly as accurate in determining the SDC rate of GPU programs.

***Index Terms*—GPU, Error Propagation, Soft Error, Silent Data Corruption, Error Resilience, Program Analysis**

I. INTRODUCTION

Transient hardware faults (i.e., soft errors) are predicted to increase in future processors due to growing system scales, progressive technology scaling, and lowering operating voltages [32]. In the past, such faults were handled in high-reliability systems through hardware-only solutions such as dual modular redundancy (DMR) and circuit hardening [25]. However, these techniques are challenging to deploy in commodity systems as they incur significant performance and energy overheads. This problem is exacerbated in the case of Graphics Processing Units (GPUs), which are commonly deployed in HPC systems, as GPUs typically have a much larger number of execution units than CPUs and hence can have greater exposure to soft errors [35].

One consequence of hardware errors is incorrect program output or silent data corruption (SDC), which is difficult to detect and can have severe consequences [32]. Therefore, it is important to estimate the SDC probability of a program to decide whether protection is needed, and if so, to reduce the cost of protection by selectively protecting the most SDC-prone program state (e.g., selective instruction duplication).

Fault injection (FI) is commonly employed to estimate the SDC probability of a program. FI perturbs program state during execution, and checks the program output to detect failures (if any). Thousands of program executions are typically required to obtain statistically significant results, and hence FI is often very slow and challenging to deploy in practice [12], [13], [21]. Further, selective protection techniques need the SDC probability on a per-instruction basis. This requires hundreds of FIs per instruction, which is very resource intensive. As a result, researchers have attempted to model error propagation to evaluate SDC probabilities quickly without any FIs [8], [15], [33]. Unfortunately, these techniques (1) have significant inaccuracies [8], [33], (2) require large and difficult-to-obtain training corpuses of FI data that are representative of typical faults [15], or (3) do not provide reliability profiles for individual instructions which is critical for the selective protection of HPC programs [15], [22].

In recent work, Li et. al. [21] proposed an automated technique called TRIDENT that analytically models error propagation in programs to predict their SDC probabilities without performing any FIs. While TRIDENT is shown to be accurate and fast in evaluating the vulnerability of *single-threaded CPU programs* to transient errors, we find it is neither accurate nor scalable for kernels of GPU programs, which are highly parallel and have a very different programming model. In particular, inter-thread data dependencies among the threads in a typical GPU program complicate error propagation [20], and not modeling them leads to large inaccuracies in predicting SDC probabilities, e.g., the mean absolute error in SDC prediction without modeling inter-thread dependencies is 17.2% with respect to FI (Section III). Furthermore, when modeling GPU programs, TRIDENT takes a significant amount of time due to the large number of threads in GPU programs, often running into days or even months (Section III). Finally, to accurately estimate per-instruction reliability, TRIDENT needs to consider data characteristics specific to GPU applications, which is not supported (as it was developed for CPU applications).

In this work, we propose an accurate and scalable technique to analytically model error propagation in GPU programs. Our key insight is that error propagation in GPU programs

SC20, November 9-19, 2020, Is Everywhere We Are
978-1-7281-9998-6/20/31.00 ©2020 IEEE

can be abstracted using program execution patterns based on memory accesses, loop iterations, and control-flow. Moreover, there is similar error propagation across different threads in GPU programs, and hence we can achieve significant speedup by carefully selecting a few threads for analysis. Our work builds on top of the open-source TRIDENT framework, and significantly enhances it to solve the above challenges of GPU programs - therefore, we call our technique GPU-TRIDENT.

There are 3 factors that make GPU-TRIDENT practical for assessing the reliability for real-world GPU programs, and protecting them. First, GPU-TRIDENT is nearly as accurate as FI but works without requiring *any* FIs, and is hence significantly faster than FI. Second, it does not require any training corpuses of FI data. Finally, GPU-TRIDENT is able to efficiently guide the selective protection for a GPU program given a reliability target and a performance overhead budget.

To the best of our knowledge, we are the first to efficiently model error propagation for GPU programs, using neither FI nor training data based on FI, to estimate its SDC probability. Our main contributions in this paper are as follows:

- Identify challenges in efficiently and accurately modeling error propagation for GPU applications (Section III).
- Propose heuristics for pruning the state space for modeling error propagation in GPU programs. The heuristics are based on similarity among the control flow and memory access patterns of the program's threads (Section IV).
- Build GPU-TRIDENT, an efficient and accurate model of error propagation for GPU programs that implements the above heuristics. GPU-TRIDENT is implemented using the LLVM compiler [19], and is completely automated.
- Compare the accuracy and scalability of GPU-TRIDENT to FI when predicting the SDC probabilities of individual instructions and that of the entire GPU kernels (Section V) for 17 GPU kernels belonging to 12 applications.
- Demonstrate the use of GPU-TRIDENT to guide selective instruction duplication, under a given overhead budget.

Our main results are as follows:

- SDC probability predictions from GPU-TRIDENT have a high agreement (Pearson correlation coefficient of 0.88) with the FI results, for most of the evaluated kernels. Individual instructions of most kernels also have a high degree of agreement with the FI results (average Pearson correlation coefficient of 0.83). On average, the overall SDC probability predicted by GPU-TRIDENT is 35.67%, while FI measures SDC probability as 33.73% across the kernels (error bars range from $\pm 0.53\%$ to $\pm 1.82\%$ depending on the benchmark).
- GPU-TRIDENT incurs a fixed initial overhead and a small incremental overhead for each sampled instruction, while FI incurs an overhead proportional to the number of sampled instructions (i.e., injected faults). This makes GPU-TRIDENT much more efficient for large programs than FI. For example, for 5000 faults, GPU-TRIDENT is 55.6 times faster than FI. FI takes on average around 4 CPU hours for 5000 sampled instructions, while GPU-TRIDENT takes approximately 6 minutes across benchmarks. This difference is even higher for more complex applications. For example, FI takes 22.7 hrs for the *Circuit* benchmark while GPU-TRIDENT takes just 8 minutes. *On average,* GPU-TRIDENT *is about two orders of magnitude faster than FI across the benchmarks.*
- Using GPU-TRIDENT to guide selective instruction duplication reduces the SDC probability of kernels by approximately 58% and 85% (at 1/3rd and 2/3rd the performance overhead of full duplication respectively). This is comparable to the results obtained using FI.

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. Finally, we provide a brief overview of the TRIDENT technique [21], as it forms the basis of GPU-TRIDENT.

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, and affect the program's execution. We assume these faults manifest as a single bit flip. Many studies [4], [5], [11], [31] have shown that there is little difference between the SDC probability of single and multiple bit flips. Moreover, previous work in this area [15], [20], [26], [36] also uses the single-bit flip model.

We do not consider faults in the GPU's control logic, nor do we consider faults in the instructions' encoding. We also do not consider faults in the memory or caches, as we assume that these are protected with error correction codes (ECC) - this is the case for most modern GPUs used in HPC applications. However, an error can propagate to memory, if an erroneous value is stored by a store instruction into memory, resulting in subsequent loads being faulty (these faults are considered).

Finally, similar to most other work in the area [7]–[10], [12], [18], 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 [28]. However, the program may take a faulty legal branch (the execution path is legal but the branch direction is wrong due to faults propagating to the branch condition).

B. Terms and Definitions

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

Crash: The raising of a hardware trap or exception due to an error (e.g., read outside its memory segments). The program is terminated as a result by the operating system.

Silent Data Corruption (SDC): A mismatch between the output of a faulty program run and that of an error-free execution of the program, without any exception being thrown.

SDC Probability: The probability that the program had an SDC given that the fault was activated—other work uses a similar definition [8], [13], [29].

C. GPU Architecture and Programming Model

We focus on GPU applications that are implemented on the NVIDIA Compute Unified Device Architecture (CUDA). A GPU application consists of a control program running on the CPU and a computation program called the kernel that runs in parallel on the GPU(s), in form of multiple threads.

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 loaded from the large and comparatively-slow device memory. The shared memory space is software managed. It is much smaller and built on chip, and is hence much faster to access. Thread-local memory is typically stored in the fast register file, though compiler-inserted spill and fill operations occasionally place thread-local state in slower areas of the storage hierarchy.

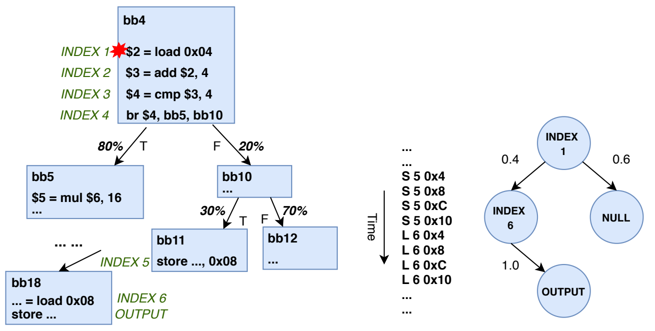
The CUDA programming model allows (1) sharing data among a subset of threads in a thread block (e.g., warp-shuffle), (2) sharing data across all the threads in a thread block using the shared-memory scratchpad, (3) sharing data across the threads in a compute kernel using device global memory, and (4) sharing across devices using unified virtual memory (UVM) [14]. This gives rise to two types of memory dependencies between instructions. (1) *Intra-thread memory dependency:* Static loads and stores of same thread are dependent on each other due to the same memory being accessed by them, and (2) *Inter-thread memory dependency:* Static loads and stores in different threads are dependent on each other due to the same memory being accessed in different threads.

D. TRIDENT *Framework*

TRIDENT is an open-source framework that analytically models error propagation at the instruction-, control-flow-, and memory-levels to derive an estimate of the overall SDC rate and SDC rate per instruction for a given CPU program [21]. However, TRIDENT does not handle the modeling of error propagation in multi-threaded programs, as well as other GPU-specific structures. As we show in Section III, modeling these is critical for GPU programs. Our technique is built on top of the TRIDENT framework, but extends it significantly.

Example: Figure 1a shows how TRIDENT works using an example from the CPU version of the *pathfinder* benchmark. TRIDENT works at the LLVM Intermediate Representation (IR) level, and hence these correspond to LLVM instructions [19]. TRIDENT consists of three sub-models that work synergistically together to calculate error propagation probability from a given location of error occurrence to the program output instruction, as follows:

1) Static-instruction sub-model (f_s): If an error occurs at INDEX 1 (say), f_s calculates the overall propagation probability for the error from Index 1 to Index 4. Each instruction is assigned an individual error propagation probability from that instruction to the next data-dependent instruction in *the same basic block*. This individual probability is derived by analyzing



(a) Code example for TRIDENT, with propagation probabilities.

(b) A memory access trace and corresponding memory dependency graph.

Fig. 1: Working of TRIDENT [21].

the instruction and profiling the values of its operands. f_s computes the overall propagation probability by aggregating individual probabilities of the instructions in the basic block.

2) Control-flow sub-model (f_c): It computes the probabilities of store instructions getting corrupted (divergence from a fault free run) due to corruption of branch instructions, which results in control-flow divergence. TRIDENT identifies all stores that are dominated by the corrupted branch (Index 4) and computes the probabilities of them getting corrupted.

3) Memory sub-model (f_m): Memory sub-model models how errors propagate via memory dependencies. Continuing with our running example, if Index 5 is corrupted, the erroneous value stored by the instruction may be loaded later by the load instruction (Index 6) in bb18. Hence, the error propagates via a memory dependency.

To figure out the memory dependencies between all loads and stores, TRIDENT profiles a *memory execution trace* from the program. The trace (Figure 1b) contains all the executed store and load instructions at runtime (ordered by execution time). Each record in trace contains the type of operation (load/store), index of the static instruction and the address that is accessed by the instruction. TRIDENT leverages the *memory execution trace* to track error propagation in f_m by constructing a *memory dependency graph* for f_m. The graph contains memory dependencies between static store and load instructions, and it is weighted based on the dynamic instruction counts of these instructions. An example of the graph is shown in Figure 1b. A node means an instruction (load or store), and the edges indicate possible dependencies. The weights (dependent on dynamic instruction count) are marked besides each edge. This graph is used during execution to track error propagation due to memory dependencies.

III. CHALLENGES

In this section, we discuss the challenges in modeling error propagation in GPU programs. As described in section II-C, GPU kernels are massively parralel, and many kernels share data across threads. Therefore, errors may propagate not only within each thread, but also between the threads.

```
1. __global__ void cudaSolve(double *data, double *odata)
2. {
3.
4.   __shared__ float sdata[12][12];
5.   ...
6.   sdata[e_li][e_lj] = data[index];
7.   ...
8.   __syncthreads();
9.
10.   odata[index] = invD*(V - sdata[e_li-1][e_lj]
11.                  - sdata[e_li+1][e_lj] - sdata[e_li][e_lj-1]
12.                  - sdata[e_li][e_lj+1]);
13.
14. }
```

Fig. 2: A slightly modified code segment from the *Circuit* benchmark [3].

Because GPU programs have a large number of threads (e.g., the *Circuit* application [3] has 4.25 million threads), tracking error propagation via memory dependencies incurs high overheads due to the large amount of memory dependency information required. Moreover, even small inaccuracies in modeling error propagation of individual threads are exacerbated when aggregating the kernel's overall SDC probability due to the large number of threads.

Figure 2 shows a code example of the *cudaSolve* kernel from the *Circuit* benchmark. This program solves a 2D circuit grid in parallel using the Jacobian method. For ease of explanation, we have slightly modified the code and removed the irrelevant parts. In the example, a subset of shared memory (line 4) is initialized in each thread (line 6). After all threads have completed the initialization (line 8), the data in the shared memory is read by adjacent threads (line 10) for computation. This way, data can be efficiently transferred between threads via fast on-chip shared memory. However, this can also result in error propagation from one thread to another, and eventually to the entire thread block and the output of the kernel. Li et al. [20] have previously shown that a single fault can lead to high contamination (up to ~60%) of memory states in GPU applications, due to error propagation across threads and data flow between global and shared memory.

We identify three challenges in modeling error propagation for GPU programs, and illustrate them with examples.

(1) **Large amount of profiling data:** To track error propagation via memory dependencies, one needs to profile all memory accesses in the kernel and identify the data dependencies between each load and store. This can be done using dynamic analysis, matching the runtime memory addresses of each load and store. However, a typical GPU kernel consists of hundreds or thousands of threads, each of which may interleave dependencies between threads in their respective thread blocks. Consequently, a huge amount of memory access information needs to be collected during the profiling phase. For example, *Lulesh*, which is a typical HPC application [17], generates a *memory execution trace* larger than 1 TB even for medium sized inputs. Processing the trace i.e., loading it into memory and traversing it (Section II-D3), incurs significant overheads, which makes it impractical to use TRIDENT for modeling of GPU HPC benchmarks. It is not possible to bypass this memory bottleneck by applying TRIDENT to individual threads, as the time for modeling thousands of threads runs into days or even months - see Table III[1].

(2) **Large number of threads:** Tracing error propagation between threads in a GPU kernel requires a fine-grained error propagation model for every thread in the GPU kernel, which potentially incurs a large modeling overhead. For example, if an error occurs in a thread, it first propagates within the thread. Later, the error may propagate to another thread in its thread block via a shared memory dependency. Hence, the error continues to propagate in both threads. Finally, the error may propagate to some other threads or back to the original thread at any time, depending on program execution. To track the error, one should model all the threads in the program, and trace the propagation in a lock-step fashion.

A natural question that arises is *can we ignore error propagation between threads in GPU programs?* To answer this question, we modify GPU-TRIDENT to stop tracing the error propagation when any shared memory access is encountered. We show the results in Figure 3 for the benchmarks that use shared memory in our evaluation (the experimental setup and benchmarks are described in Section V-A). As can be seen, the SDC probabilities predicted by the modified GPU-TRIDENT are significantly lower than the FI ground truth (mean absolute difference of 17.2%). The only exception is LUD K3, which only uses shared memory sparingly (only two stores are to shared memory by each thread). This result indicates that we cannot ignore inter-thread error propagation in GPU programs.

(3) **Accumulated inaccuracy from individual threads:** The overall SDC probability of a GPU kernel is the aggregation of the SDC probabilities of the individual threads. Because GPU programs consist of hundreds or thousands of threads, if the model is inaccurate in each thread (even slightly), the overall SDC probability of the kernel will be very inaccurate.

IV. APPROACH

In GPU-TRIDENT, we first propose two heuristics to select a subset of threads in a GPU kernel to find the intra-thread memory dependencies within it (Section IV-A). We then propose a third heuristic to a select a new subset of threads to construct inter-thread memory dependencies (Section IV-B). Finally, we identify 2 sources of inaccuracies in TRIDENT, and propose heuristics to mitigate them (Section IV-C).

A. Profiling for Intra-Thread Memory Dependency (H-Intra)

The goal of this step is to construct the *memory dependency graph* for a given GPU kernel without profiling all the memory accesses in all the threads. Recall that the *memory dependency graph* in TRIDENT is required to establish data dependencies between static store and load instructions (Section II-D3). It is possible to profile all memory addresses used by load and store instructions, and then match the addresses to build the graph. However, this can be extremely time-consuming in massively parallel GPU programs. Instead, we profile only a small subset of threads based on control-flow similarity,

[1]We obtained these numbers by multiplying the time taken by TRIDENT for each thread of the program with the number of threads in the kernel.

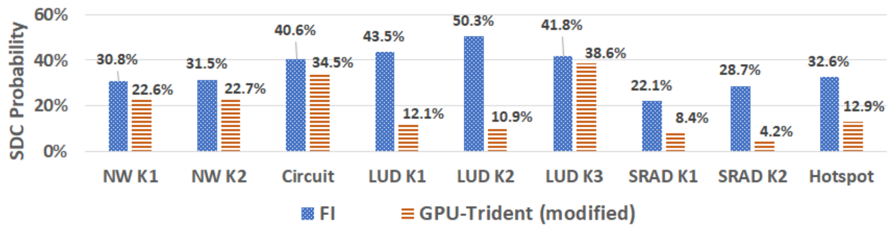


Fig. 3: Prediction of SDC Probability by a modified GPU-TRIDENT that does not model error propagation across threads, for benchmarks that use shared memory; NW K1 means the first kernel of NW benchmark.

and match loads and stores in those carefully-selected threads to build the *memory dependency graph*. We call these *intra-dep threads*. These are the threads whose memory reads and writes can be profiled to get all the intra-thread memory dependencies between static load and store instructions of the kernel. Algorithm 1 describes how to choose *intra-dep threads*.

Note that in this section, we use the entire control-flow profile of a thread to perform the grouping of threads, rather than the number of instructions executed per thread used in prior work [26] as we found it to be more accurate.

Algorithm 1: Selecting *intra-dep threads*.

```
Input  : $CF_T$: Control flow of all threads
Output: $T_P$: Set of threads to profile
1  $T_P = \{\}$
2  for all threads $T$ do
3      if conditional branches in loop then
4          // Apply H-Intra-Loop heuristics to remove
             iterations with nonunique memory accesses
5          $CF_{sim} = Remove\_redundant(CF_T)$
6      else
7          $CF_{sim} = CF_T$
8      // Profile thread if its simplified execution path is
         new
9      if $CF_{sim}$ not in $T_P$ then
10         $T_P.insert(T)$
11 end
```

1) Selection based on thread execution path (H-Intra-Path): We first profile the execution path of each thread and group the threads that have identical execution paths. We then choose a single thread from each group to be part of *intra-dep threads*. The intuition is that threads that have identical execution paths have the same static dependencies between loads and stores. This is because only control-flow divergence can cause divergence in the memory dependencies.

Table I shows the number of threads invoked in each kernel of the benchmark applications (Section V-A has more details), and the number of *intra-dep threads* as the result of the grouping. As can be seen, most of the kernels have only a small number of *intra-dep threads*. The two exceptions are *pathfinder* and *NW*, as they have conditional branches inside loops, and hence loops execute different number of times for different threads, which results in different control-flow paths.

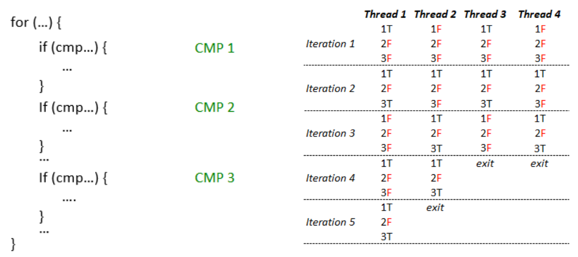
Kernel	Total threads	After *H-Intra-Path*	After *H-Intra-Loop*	After *H-Inter*
NW K1	48	46	25	48
NW K2	16	16	13	16
Pathfinder	592,640	563,606	15	3840
BFS K1	32,768	781	8	-
BFS K2	32,768	2	2	-
Gaussian	3,840	4	4	-
HotSpot	473,344	250	234	35,328
Particlefilter	9,216	38	7	-
LUD K1	64	16	16	64
LUD K2	192	2	2	64
LUD K3	3,584	1	1	256
CudaBenchMarking	8,192,000	1	1	-
SRAD K1	32,768	144	144	16,384
SRAD K2	32,768	16	16	4,096
Circuit	4,156,416	18	18	2,592
Lulesh	66,816	2	2	-
Blur	236,544	7	7	-

TABLE I: Thread group details for the studied kernels

2) Selection based on loop patterns (H-Intra-Loop): To further reduce the number of *intra-dep threads*, we pick threads that have unique loop patterns. We use a code example from the Pathfinder benchmark to explain this heuristic. Figure 4a shows a kernel that has three conditional branches inside a loop. Pathfinder has a high number of *intra-dep threads*, as different threads take different branches in each iteration, and hence have different number of iterations resulting in diverging execution paths.

We profile the execution paths inside each loop iteration and list a subset of them in Figure 4b. Numbers in the figure represent the comparison instruction, while *T* and *F* represent the branch direction taken. We find that many iterations contain repeated branch patterns, which results in the same memory dependency from the profiling. As seen from the figure, the first three iterations in *Thread 1* and *Thread 3* are identical. Therefore, we only need to profile one of them to obtain the inter-thread dependencies. Similarly, the order of the repeated patterns within a thread does not matter, as it would yield similar dependencies. For example, *Thread 3* and *Thread 4* follow the same branch patterns but in different orders of iterations, yet yield the same memory dependency. Therefore, we only need to profile one of the two threads.

Table I also shows the number of *intra-dep threads* after applying the above loop selection heuristic. As seen, Pathfinder, and BFS K1 see a significant reduction in the number of

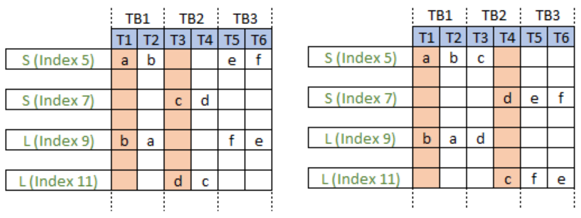


```
for (…) {
      if (cmp…) {                CMP 1
            …
      }
      If (cmp…) {                CMP 2
            …
      }
      …
      If (cmp…) {                CMP 3
            ….
      }
      …
}
```

	Thread 1	Thread 2	Thread 3	Thread 4
	1T	1F	1T	1F
Iteration 1	2F	2F	2F	2F
	3F	3F	3F	3F
	1T	1T	1T	1T
Iteration 2	2F	2F	2F	2F
	3T	3F	3T	3F
	1F	1T	1F	1T
Iteration 3	2F	2F	2F	2F
	3F	3T	3F	3T
	1T	1T	exit	exit
Iteration 4	2F	2F		
	3F	3T		
	1T	exit		
Iteration 5	2F			
	3T			

(a) Code snippet of pathfinder (b) Control flow traces of threads.

Fig. 4: Conditional branches inside a loop in the *Pathfinder*.



(a) (b)

Fig. 5: Profiling inter-thread memory dependencies. (Values in squares represent memory addresses.)

threads, but not the other kernels as the threads do not have conditional branches with divergent control-flow within loops.

B. Profiling for Inter-Thread Memory Dependencies (H-Inter)

Intra-thread memory dependencies can be established based on the thread selection method in the previous section. However, inter-thread dependencies may still exist between threads with differing control-flows and we need to model these. As before, we propose a heuristic to reduce the number of loads and stores that must be profiled to capture the inter-thread dependencies. Our heuristic is to first select the thread blocks that each of *intra-dep threads* belongs to, and then profile the shared memory accesses in each thread block. This is based on the observation there is often a high ratio of threads to thread blocks, as well-designed GPU kernels have little control-flow divergence within each thread block to avoid stalls. Hence, we find that selecting thread blocks based on representative thread groups exhibiting the same execution paths (i.e., thread blocks containing *intra-dep threads*) captures most of the unique shared memory access patterns (Table II).

Figure 5 shows the executions of two kernels as examples. Each row represents a possible execution of an instruction, while each column indicates a thread. For simplicity, we only show instructions accessing shared memory. The value inside each square is the memory address that the instruction uses. Having the same address means that the load and the store have an inter-thread dependency, which we need to identify.

In Figure 5a, there are 3 thread blocks (TBs) as shown. Each TB contains two threads. T1, T2, T5 and T6 have the same execution path, whereas T3 and T4 follow a different execution path. Note that the control-flow paths of threads are identical within each thread block in this example. T1 and T3 are selected as *intra-dep threads* based on their execution paths (Section IV-A). According to our heuristic, we profile all the shared memory accesses in the thread blocks that T1 and T3 belong to. In other words, we profile the shared memory in T1, T2, T3, and T4. This way, we establish 2 inter-thread memory dependencies, namely (Index 5 and Index 9), and (Index 7 and Index 9), based on the addresses. Since the ratio of control-flow similarity across thread blocks is high (100%), our heuristic identifies all the inter-thread dependencies.

The example in Figure 5b also has 6 threads in 3 thread blocks. Threads in TB1 follow the same execution path, as do the threads in TB3. However, threads in TB2 (i.e., T3, T4) follow different paths. In this case, the ratio of the similarity of control-flow of threads inside thread blocks is only 66.67% ((100%+0%+100%)/3). Since threads T1 and T4 are selected as *intra-dep threads*, all the threads in TB1 and TB2 are considered for profiling, and the inter-thread memory dependencies (Index 5 and Index 9), and (Index 5 and Index 11) are identified. However, since the control-flow of T4 is different from T3 in TB2, the selection of T4 does not include the profiling of the threads in TB3, thereby missing another inter-thread dependency (Index 7 and Index 11) in T5 and T6.

Therefore, our heuristic achieves high coverage only when the control-flow similarity ratio is high in a thread block. So we ask the question *what are the ratios of control-flow similarity in thread blocks in GPU programs?* Table II shows the results, for the kernels that have inter-thread memory dependencies. On average, about 75% of the threads in a thread block exhibit control-flow similarity, with the highest being 100% in *LUD K3*, and the lowest being 0% in *LUD K1*. Therefore, our proposed heuristic (*H-inter*) is able to identify most of the inter-thread memory dependencies in practice. The threads that we profile to get inter-thread memory dependencies in a kernel are called *inter-dep threads*.

Kernel	CF Similarity	Kernel	CF Similarity
NW K1	54.16%	NW K2	25%
HotSpot	98.31%	LUD K1	0%
LUD K2	70.32%	LUD K3	100%
Circuit	99.35%	Pathfinder	96.61%
SRAD K1	98.28%	SRAD K2	99.6%

TABLE II: Average Control-flow(CF) similarity percentage of threads in a thread block

Table I shows the number of threads chosen for profiling inter-thread memory dependency. On average, we only select ~40% of the total threads as *inter-dep threads*. It can be seen that kernels that have lower number of total threads do not show much reduction (0% for *NW K1*, *NW K2*), while kernels that have a large number of threads show a much higher reduction in the number of threads chosen (~99.3% for *Circuit*). Note that only load and store instructions to shared memory are profiled for these threads. The algorithm for choosing *inter-dep threads* is in Algorithm 2.

We construct the *memory dependency graph* for a given GPU kernel applying the above heuristic. We then follow the

Algorithm 2: Selecting *inter-dep threads.*

```
Input  : $T_P$: Threads from H-Intra
Output: $T_{PS}$: Threads to profile
1 $T_{PS} = \{\}$
2 $TB_P = \{\}$
3 for all threads $T$ in $T_P$ do
4     $tb = find\_threadblock(T)$
5     if $tb$ not in $TB_P$ then
6         $TB_P.insert(tb)$
7         // Profile all threads of this thread block
8         $T_{PS}.insert(all\_threads(tb))$
9 end
```

method of TRIDENT to weight the edges of the graph based on the dynamic instruction count of each static load and store, and then trace error propagation using the graph. Note that f_m does not need the memory addresses (i.e. *memory execution trace*) any more for extracting the dependency; instead, it reads it directly from the *memory dependency graph* that we created for this purpose, and aggregates the propagation probabilities based on the edge weights (Section II-D3).

C. Value-Based Masking (H-Value):

As mentioned, it is very important to have a high accuracy in modeling error propagation in GPU threads due to the potential of aggregating small errors into large ones. We identify 2 sources of inaccuracies in TRIDENT and propose the *H-Value* heuristic to mitigate them - *this leads to an average accuracy improvement of approximately 1.9% across our benchmarks.*

The sources of inaccuracy are as follows.

1) Multiplication by zero: If a target operand of an instruction is multiplied by zero, the result will be always a zero regardless of errors present in the target operand. TRIDENT does not consider this effect and hence will overestimate the error propagation. We find that there is a non-negligible amount of multiplication-by-zero in GPU kernels (i.e., as high as 56.6% in Lulesh, and on average 11.2%). To account for this, we profile the operands of multiplication instructions, and calculate the frequency of zero operands of *mul* instructions. We then weight the propagation and masking probabilities in the relevant instruction tuple (Section II-D1) accordingly.

2) Lucky stores: Recall that if a branch is modified by an error, the store instructions dominated by the corrupted branch will be not be executed correctly (Section II-D2). In this case, TRIDENT assumes that the error always propagates to the store instructions. However, if a store instruction was supposed to overwrite the value in memory, errors will not propagate to it even though the instruction is skipped due to a corrupted branch. We call such store instructions *lucky stores*, similar to *lucky loads* found in prior studies on CPUs [7], [21].

Identifying all the lucky stores requires recording every operand of store instructions in the *memory execution trace*, which can be extremely time-consuming. We observe that "zero" values are dominant in the operands of output stores of GPU kernels as most initializations of kernel memory use zeros. Therefore, we record only the frequencies of output stores that have a zero operand at runtime (average of 7.6% (maximum of 40%) for the kernels used in this paper), and weight the propagation probabilities in f_c accordingly.

D. Analysis workflow

Based on the above heuristics, the analysis workflow of GPU-TRIDENT consists of following steps:

1) *H-Value* is applied to get the f_s and f_c sub-models based on the dynamic data profiled from the threads.
2) *H-Intra* is used to select a sub-set of threads to get the intra-thread memory dependencies, whose memory access instructions are then profiled.
3) *H-Inter* is used to select a sub-set of threads to get the inter-thread memory dependencies, whose memory access instructions are then profiled.
4) f_m model is created from the memory dependency graph, based on the profiled memory access instructions.
5) The sub-models (f_c,f_s and f_m) are used to get the SDC probability of individual instructions and the kernel.

V. EVALUATION

In this section, we first describe the experimental setup in evaluating GPU-TRIDENT, and then the experimental results.

A. Experimental Setup

1) Workflow and implementation of GPU-TRIDENT*:* GPU-TRIDENT is implemented as a set of LLVM compiler passes that are integrated into NVIDIA's NVCC compiler [27]. We have made GPU-TRIDENT publicly available[2]. Although NVCC is based on LLVM, it does not expose its IR representation to external tools. We, therefore, attach a dynamic library [24] to insert the LLVM passes of GPU-TRIDENT in the toolchain (as done in prior work [20]).

Note that the heuristics and algorithms proposed in this paper are not limited to the CUDA platform, but can apply to any generic GPU architecture as they have similar structures. Moreover, any source language that can be compiled to the LLVM IR can be used with the GPU-TRIDENT infrastructure.

GPU-TRIDENT needs the application code (with kernel under test annotated), and an input required to execute it. The programmer also needs to annotate the static store instructions that are used by the kernel for transferring data to the host, and designate them as output instructions. Note that FI also requires this annotation, as this is used to identify which memory has to be checked for determining SDCs. In our experience, it took only a few minutes to identify these instructions for all the benchmarks. This annotation can potentially be automated for both techniques, but is outside the scope of this work.

Using these inputs, GPU-TRIDENT profiles the program (using LLVM), and estimates the SDC probability of individual instructions and the complete kernel *without FIs*.

[2]https://github.com/DependableSystemsLab/GPU-Trident

2) FI method: Recall that GPU-Trident aims to predict SDC probabilities, which are usually measured using FI. Therefore, we use FI as the baseline to establish our ground truth when evaluating GPU-Trident. We use an open-source injector, LLFI-GPU [20] to perform FIs. LLFI-GPU aids comparison with GPU-Trident, as both are implemented using LLVM. Therefore, we can map GPU-Trident predictions to the FI result of individual instructions, and examine the accuracy of GPU-Trident on a per-instruction basis.

Because we consider faults in the computational elements of the GPU (Section II-A), we inject faults into the destination registers of executed instructions. Only one fault is injected per program execution. In each FI trial, the application is executed from the beginning, and we uniformly choose an instruction at random from the set of executed instructions in the kernel. We then flip a single bit in its destination register, and execute the program to completion. This method has been shown to be accurate for estimating SDC probabilities [4].

To determine if an SDC occurred, the entire array in device memory that is written by the kernel (to transfer data to the host code) is compared to its contents in a fault free execution (i.e., golden run). This is because GPU-Trident only analyzes error propagation within the GPU kernel, and hence cannot use the application output, which includes masking by the CPU portions of the program. It is possible to combine GPU-Trident with other CPU error propagation models (e.g., Trident) to obtain an end-to-end resilience estimate.

3) Benchmarks: We choose a total of 17 kernels from 12 applications belonging to different domains, 8 of which are from the Rodinia benchmark suite [6], and 4 are open source HPC applications [3], [17], [1], [2]. These are listed in Table III, and range in size from 16 to 511 static LLVM IR instructions and range from 16 to 8,192,000 in terms of total threads launched. The choice of benchmarks was governed by whether they can be compiled with the LLVM-based infrastructure of LLFI-GPU and GPU-Trident, and whether FI can be completed in a reasonable amount of time. We removed all sources of randomness in the benchmarks (e.g., changed random numbers to constant values), to get reproducible results for the experiments.

We use the smaller inputs that come with each benchmark. We have also tested GPU-Trident with bigger inputs for a subset of the same kernels, and find that the results closely match those with FI (though they take much longer). However, to keep the time for the detailed FI experiments manageable, we choose to use the smaller inputs. Prior work [15], [20], [26], [35], [36] has also used similar input sizes.

Table III also shows the times Trident and GPU-Trident take for each of the kernels (as mentioned earlier, Trident times are estimates). We consider the time taken for both systems to obtain the overall SDC probability and per-instruction SDC probability. As can be seen, GPU-Trident *takes less than about 10 minutes for all the kernels*. In contrast, Trident takes days or even months to analyze the kernels.

B. Accuracy

We evaluate the accuracy of GPU-Trident in predicting SDC probabilities of the kernel, and individual instructions.

1) Overall SDC probability: We use GPU-Trident to predict the SDC probability of a given kernel (and input), and then measure its SDC probability (under the same input) using random FI. We perform 5000 FI experiments per kernel - the error bars for the SDC measurements range from ±0.53% to ±1.82% at the 99% confidence level.

Figure 6 shows the result of our experiments. As can be seen, GPU-Trident provides reasonably accurate predictions. *On average, the SDC probability predicted by* GPU-Trident *is 35.67%, in comparison to 33.73% measured by FI.* The average difference (*mean absolute error*) between FI and GPU-Trident is 5.7% (in comparison, Trident had an average difference of 4.75% [21] for CPU programs). This number is inflated by the results of *Lulesh*, *BFS K1* and *Pathfinder*, which have a difference of ~24%, ~15% and ~16% respectively (the difference is 2.97% if we remove these kernels). We explain the reasons in Section VI.

We also calculate the Pearson correlation coefficient between the FI results and GPU-Trident predictions for all the kernels. The correlation coefficient is 0.88, showing high agreement between them. In fact, the correlation coefficient increases to 0.99, if we ignore the 3 outliers described earlier.

Finally, we use a paired t-test to examine if the predictions are statistically different from the FI measurements, in line with Trident's method [21]. We first checked that the differences between the predictions and FI measurements are approximately normally distributed, as required by the t-test. In the t-test, our null-hypothesis is that there is no statistically significant difference between the results from FIs and the predicted SDC probabilities by GPU-Trident for the 17 kernels. We find that the t-test yields a p-value of 0.36, which is much greater than the customary threshold of 0.05 [34], and *hence we fail to reject the null hypothesis.*

2) Instruction SDC probability: We also evaluate the prediction accuracy of per-instruction SDC probabilities for each kernel. We use GPU-Trident to predict the SDC probabilities of all static instructions, and then compare the predicted values with the FI results. In FI, we inject 100 random faults in each static instruction of the kernel (a random dynamic instance of the instruction is chosen for FI in each run).

As before, we calculate the Pearson correlation coefficient between the SDC contribution by each static instruction found using FI and by GPU-Trident for all kernels. The average correlation coefficient was 0.83, excluding the outliers mentioned in Section V-B1 and *Gaussian*. Gaussian has a low coefficient which is reflected in the predicted SDC percentage being more than double the measured SDC percentage, although the percentage difference is ~5%. The correlation analysis shows that the per-instruction SDC probability obtained by GPU-Trident has high agreement with FI results.

We again check if the distribution of the differences between the prediction and the FI measurement are approximately normally distributed. The normality does not hold for *SRAD*

Benchmark	**Suite**	**LLVM Insts.**	**Kernel ID**	**Uses shared memory**	**Total threads**	**Time with TRIDENT (in days)**	**Time with GPU-TRIDENT (in minutes)**
Needleman-Wunsch	Rodinia	248	NW K1	Yes	48	0.48	4.77
		249	NW K2	Yes	16	0.16	4.76
Pathfinder	Rodinia	132	Pathfinder	Yes	592,640	3,908	6.56
BFS	Rodinia	47	BFS K1	No	32,768	80	1.41
		20	BFS K2	No	32,768	36	0.93
Gaussian	Rodinia	59	Gaussian	No	3,840	8.6	1.21
HotSpot	Rodinia	259	HotSpot	Yes	473,344	5,026	5.14
Particlefilter	Rodinia	39	Particlefilter	Yes	9,216	18	1.46
LU Decomposition	Rodinia	142	LUD K1	Yes	64	0.28	2.58
		238	LUD K2	Yes	192	1.2	3.27
		78	LUD K3	Yes	3,584	8.9	1.66
SRAD	Rodinia	511	SRAD K1	Yes	32,768	927	10.5
		193	SRAD K2	Yes	32,768	187	2.84
Lulesh	OS HPC [17]	29	Lulesh	No	66,816	56	2.77
Circuit	OS HPC [3]	167	Circuit	Yes	4,156,416	20,532	4.96
Perf Benchmark	OS HPC [2]	16	Perf_BM	No	8,192,000	6463	7.43
HPCCUDA	OS HPC [1]	397	HPCCUDA	No	236,544	670	4.45

TABLE III: Benchmarks used: 8 are from the Rodinia suite, and the other 4 are open source HPC applications (OS HPC).

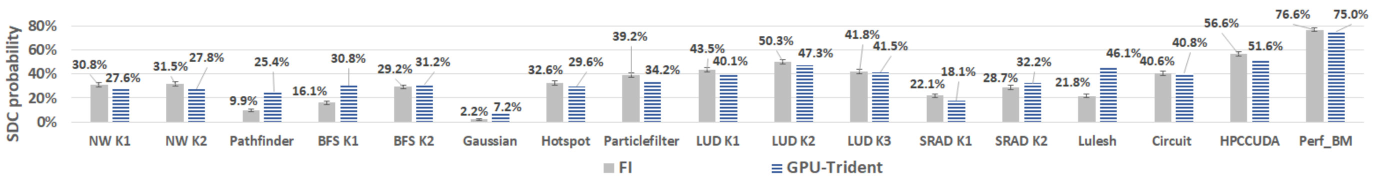


Fig. 6: The SDC probability of GPU kernels predicted by FI and GPU-TRIDENT. Error bars are shown for the FI estimates at the 99% confidence level - they range from $\pm 0.53\%$ to $\pm 1.82\%$ depending on the benchmark.

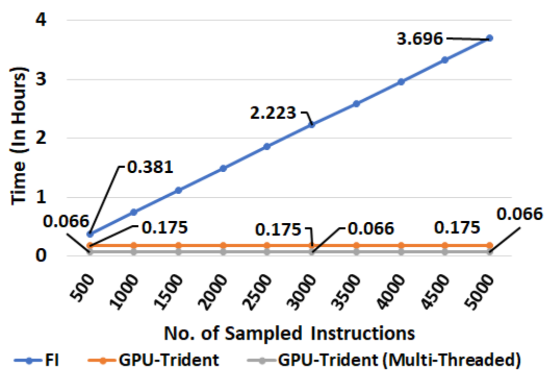
K1, *SRAD K2*, *Circuit* and *Particlefilter*, and so we exclude them from this experiment (however, the predicted SDC probabilities for these kernels are close to the FI results (Figure 6)). We perform paired t-test experiments in the remaining 13 kernels. The number of paired data in the t-test for each kernel is the number of static instructions in it. As before, our null hypothesis is that there is no difference between the FI measurement and the predicted SDC probability of each (static) instruction by GPU-TRIDENT. The p-values are higher than 0.05 in 11 out of the 13 kernels (all except *Lulesh* and *NW K1*), and hence we *cannot reject the null hypothesis*.

C. Scalability

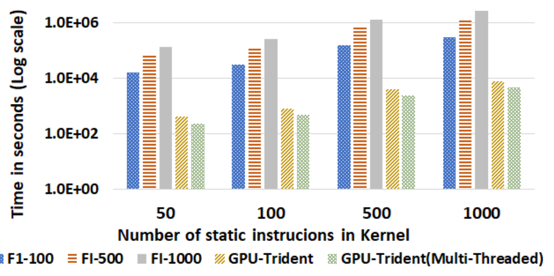
In this section, we assess the scalability of GPU-TRIDENT with respect to FI. We evaluate the scalability both in terms of the overall SDC probabilities of kernels as well as those of individual instructions. As mentioned earlier, our accuracy experiment considered $5,000$ trials and obtained error bars at the 99% confidence interval. However, if one is prepared to accept lower confidence intervals or looser error bars, it is possible to decrease the number of FI experiments (i.e., samples). To evaluate the scalability of GPU-TRIDENT with respect to FI, we compare the time taken by GPU-TRIDENT and FI when using 500 to $5,000$ samples. This is in line with the number of samples used in other studies [8], [15], [20].

As mentioned earlier (Section II-D), executing GPU-TRIDENT can be divided into two phases. (1) Profiling phase, which requires multiple executions of the program to collect data, and the (2) inference phase, which uses the information obtained from the profiling phase to calculate the SDC probabilities for static instructions, and requires no program executions. We implement the inference phase of GPU-TRIDENT in (1) single threaded mode, and (2) multi-threaded mode. However, we do not parallelize the profiling phase, to keep it consistent with FI, which is not parallelized either. Note that parallelizing the profiling phase is non-trivial as it requires either multiple GPUs or the ability to run multiple applications simultaneously on a single GPU.

1) Kernel SDC probability: Figure 7a shows the average time taken by FI and both implementations of GPU-TRIDENT to predict the SDC probability of CUDA kernels. Due to space constraints, we show the average time across all the kernels. It can be seen that the differences between the times taken by GPU-TRIDENT and FI increase sharply as the number of samples is increased. *For example, for* 500 *samples,* GPU-TRIDENT *is 2.2 and 5.7 times faster than FI (for single and multi-threaded implementations respectively). This increases to 12.7 and 33.4 times for* $3,000$ *samples, and 21.1 and 55.6 times for* $5,000$ *samples*. This is because FI has negligible fixed cost at startup, but every FI trial requires a complete



(a) Overall SDC Probability



(b) Instruction SDC Probability

Fig. 7: Computation Effort to Predict SDC Probability

program execution, and so the time required for FI experiments increases linearly with the number of FI samples. In contrast, GPU-TRIDENT has an initial fixed cost for building the model. Once the model is built, the cost of calculating the SDC probability of individual instructions is negligible, as it only involves a table lookup. This results in predominantly flat lines for GPU-TRIDENT in Figure 7a. It can be seen that multi-threaded GPU-TRIDENT is on average around 2.5X faster (on an 8-core CPU) than its single threaded counterpart.

2) Instruction SDC probability: Figure 7b compares the average time taken by GPU-TRIDENT and FI, for getting instruction wise SDC probability, for different numbers of static instructions in the kernel, and with different numbers of faults injected (100, 500 and 1,000) per instruction. The time shown in the figure is projected based on the per instruction times recorded for kernels that we use in our evaluation. Number of samples per instructions are appended as suffixes in the Figure 7b. For example, FI-100 means that 100 faults are injected into each static instruction. Because GPU-TRIDENT does not need to sample individual instructions, the time taken by it remains constant; therefore it has only one curve for each implementation. It can be seen that as the number of instructions in a kernel increases, the difference between the time taken by FI and GPU-TRIDENT also increases. For example, at 50 instructions, FI-100 takes around 4 hours more than GPU-TRIDENT (averaged for both implementations). This difference increases to around 84 hours for 1,000 instructions (more than 20X increase).

Figure 8 shows the wall-clock time taken by both implementations of GPU-TRIDENT and FI (100 faults injected per instruction), for obtaining instruction wise SDC probability, for each kernel. There is a wide variation in times taken by GPU-TRIDENT and FI for different kernels. For example, *BFS K2* takes 0.16 hours, while *Circuit* takes 63.75 hours for FI, so we use a logarithmic scale in figure on the y-axis. From the figure, we can see that on average, single-threaded GPU-TRIDENT is faster than FI by more than one order of magnitude (~38X), *while multi-threaded* GPU-TRIDENT *is faster than FI by about two orders of magnitude on average*. Further, higher the complexity of the application, greater the improvement, e.g., *Circuit* and *Lulesh* in Figure 8.

VI. REASONS FOR INACCURACY OF GPU-TRIDENT

In this section, we discuss the sources for inaccuracies in GPU-TRIDENT. There are 4 sources of inaccuracies.

1. Validity of faulty store addresses: GPU-TRIDENT assumes that if a wrong memory location is accessed, it will either result in a crash if the memory location is invalid, or an SDC if the memory location is valid. However, in some cases, the accessed wrong memory location is valid, but is out of the scope of the kernel. This results in a benign outcome.

On the other hand, because FI records the kernel's output in the host code, it has the knowledge about the memory that should be written by the kernel in order to lead to an SDC - therefore, it can obtain the exact SDC probability. This is one of the major reasons for SDC overestimation by GPU-TRIDENT for both *Luelsh* and *Pathfinder*. This inaccuracy can be mitigated by incorporating the information about the ratio of the memory used by the kernel that causes an SDC into GPU-TRIDENT. For example, in the case of *Lulesh*, if we integrate the information about the ratio of total memory used by FI to detect SDC (0.5 - we obtained this via analysis of the host code) into GPU-TRIDENT, the absolute error in prediction is reduced from 24% to just 0.31%.

2. Conservativeness in determining memory corruption: In Section IV-C2, we discuss the phenomenon of lucky stores and introduce a heuristic to identify lucky output store instructions. However, lucky stores can also occur in the intermediate stores of the kernel, e.g., when a store to local or shared memory dominated by a faulty branch is missed or incorrectly executed, but the value written happens to be the same as the correct value. GPU-TRIDENT assumes that any fault propagating to a store instruction results in an erroneous memory value, and hence over-estimates the SDC probability. This is another reason for the inaccuracy in *Pathfinder* (in addition to the first).

Similarly, GPU-TRIDENT assumes that a fault in the address operand of a load instruction will propagate to the instruction's output. However, if the incorrect memory location has the same contents as the fault-free memory (i.e., lucky load [7]), it will result in a benign outcome. This is the major reason for inaccuracy for *BFS K1* (especially in the earlier kernel invocations), where most of the device memory allocated initially contains zeroes. Therefore, when a load instruction reads from an incorrect memory address, it is likely to load a zero, which is the same value it would obtain in a

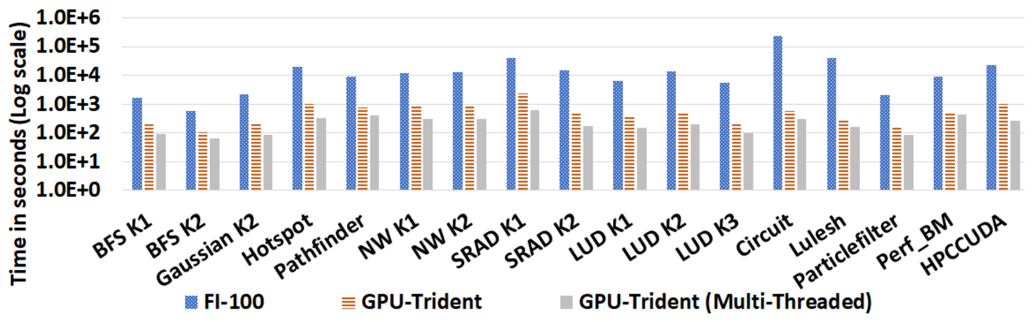


Fig. 8: Performance comparison of FI and GPU-Trident

fault-free run. Using FI, we find that around 15% of load instructions in *BFS K1* are affected by this issue, resulting in over-estimation of SDC probability by GPU-Trident.

3. Heuristics about error propagation: The original Trident considers only three instruction types for logical masking, namely comparisons, logical and cast operations. GPU-Trident also includes multiply instructions based on dynamic profiling (Section IV-C1). However, other instructions can also mask errors. For example, the *fdiv* instruction can mask errors while averaging the corrupted bits in the mantissa. This will also result in GPU-Trident over-estimating SDCs.

4. H-Inter Heuristic: A kernel which has control-flow divergence between threads in a thread block (Figure 5b) can result in GPU-Trident missing some inter-thread memory dependencies in the *memory dependency graph*, which can result in inaccuracies (both under- and over-estimates).

VII. Use Case: Selective Instruction Duplication

In this section, we use GPU-Trident to guide selective instruction duplication, which is a standard protection technique to detect SDCs [8], [16], [18], [30]. Selective instruction duplication duplicates selected instructions and compares their outputs at runtime to detect any differences as errors [8], [18], [21]. Typically, it is assumed that the maximum performance overhead for the protection is fixed to a budget value.

Then the question becomes: *Which static instructions should be duplicated to reduce the overall program SDC, given a performance overhead budget?*.

The predominant way to answer this question for GPU applications today is through FI, which is very time consuming, as it requires per instruction SDC probabilities. In contrast, we study the usefulness of GPU-Trident in replacing FI to answer this question, by accurately predicting the SDC probability of individual instructions.

To study this, we model the problem as a classical 0-1 knapsack problem [23], where the performance overhead of duplicating instructions is the sack capacity, and the SDC reduction by duplicating the instructions is the profit. We use a standard dynamic programming algorithm for the 0-1 knapsack problem, similar to previous work [21]. For simplicity, we consider instruction SDC probabilities to be independent of each other, which is a conservative assumption that has also been made by previous work [21]. We use GPU-Trident to estimate the SDC probability of each instruction, and use the dynamic instruction count as a proxy of the instruction's performance overhead (measuring the exact overhead is tedious).

We consider two protection overhead levels, which correspond to one-third and two-thirds performance overhead of duplicating all the instructions in a given GPU kernel. We use FI to measure the SDC probability of the protected kernel.

Figure 9 shows the SDC probabilities of kernels after applying selective duplication. Without protection, the average SDC probability across benchmarks is 33.73%. *With one-third protection overhead, the average SDC probability is reduced to 14.2%, which is a reduction of about 58%. With a two-thirds protection overhead, the average SDC probability is further reduced to 5.27%, which is a reduction of about 85%.*

Figure 10 shows the average SDC coverage, for all benchmarks, provided by instruction duplication, when it is guided by FI and GPU-Trident at different overhead levels. *We can see that the protection curve of* GPU-Trident *closely follows that of FI throughout the range*. The maximum difference between these two curves is about 13.34% which translates to an absolute difference of only 4.4% in terms of SDC percentages. Thus, GPU-Trident is an accurate and efficient replacement for FI in guiding selective instruction duplication.

VIII. Related Work

Modeling error propagation: Error propagation models are widely employed to estimate the resilience of CPU programs. Shoestring [8] uses compile-time analysis and symptom-based error detection techniques to model error propagation and identify vulnerable instructions. Sridharan et al. [33] introduce an analytical model, program vulnerability factor (PVF), to capture the masking properties of the program. However, it does not mode control-flow divergence or memory dependencies, and their accuracy suffers as a result. Further, it is not targeted towards GPUs. Guo et al. [10] identify naturally resilient code patterns in HPC applications, but do not quantify resilience of the over-all program.

GPU error resilience: Yim et al. [37] developed one of the first FI tools for GPU applications to explore efficient error detectors in GPU programs. Li et al. [20] design LLFI-GPU, which operates at the LLVM IR level, and use it to investigate error propagation in GPU kernels. Unfortunately, FI requires significant time and resources in evaluating GPU applications.

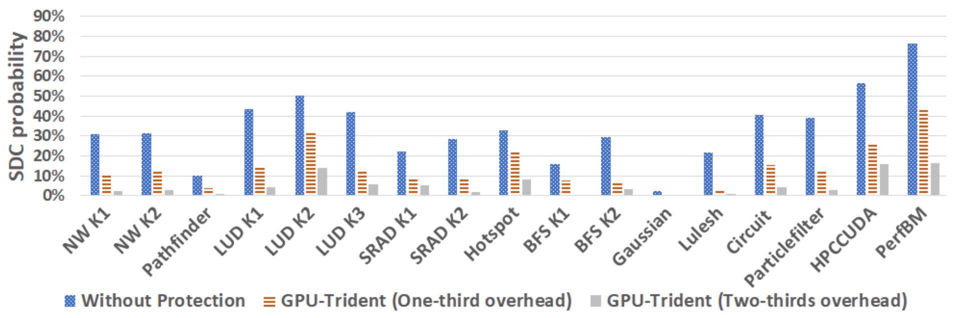


Fig. 9: SDC Probabilities of selective instruction duplication.

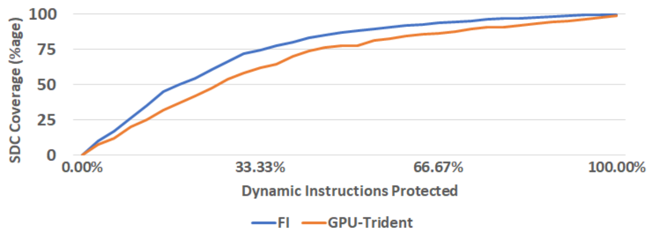


Fig. 10: Protection curves obtained by FI and GPU-TRIDENT.

Tan et al. [35] propose an analytical framework to estimate the vulnerability of GPU microarchitectures. However, their technique does not consider the characteristics of the GPU program as it is microarchitecture dependent. Further, they do not distinguish between crashes and SDCs in their analysis. Wei et al. [36] study the approximation properties of soft errors and use them to guide approximate instruction duplication. However, this requires thousands of fault injections and intimate knowledge of the application's domain and functionality.

Pruning of FI space: Pruning FI space based on similar execution patterns has been proposed to speed up FI. Hari et al. [12], [13] propose techniques to prune the FI space in CPU applications based on similar error propagation patterns in programs, reducing time taken in FIs.

There are two techniques for pruning the FI space of GPU programs that share the same high-level goal as GPU-TRIDENT, i.e. to estimate the resilience of GPU programs with either limited or no FI. Nie et al. [26] propose a method to prune the FI space of GPU programs. While useful, their method still requires hundreds of FI trials (and a few thousands of trials in a handful of cases) per kernel invocation, to obtain SDC estimates for programs. Running FI experiments on GPUs is resource intensive as a GPU program has a huge FI space among millions of threads. Further, unlike in CPUs, the process of FI in GPU is difficult to parallelize as only one GPU process is allowed per GPU card. Finally, these heuristics are specific for pruning of FI sites, and so they cannot be used in modeling error propagation, which is our goal. Kalra et al. [15] propose a machine learning technique to estimate the resilience of GPU programs. However, their technique relies on a large amount of representative FI corpus in the training phase, which are used to learn the characteristics of SDCs. These are difficult to obtain in practice. Moreover, both techniques only predict the vulnerability of the overall program, and not that of individual instructions - this is needed for selective protection approaches.

IX. CONCLUSION

This paper introduces GPU-TRIDENT to model soft error propagation in GPU kernels without performing expensive Fault Injections (FI). Because of the typically large number of threads in a GPU kernel, it is challenging to model error propagation accurately and scalably. To address this challenge, we propose three heuristics to prune the error propagation space, and improve the accuracy of GPU-TRIDENT based on the similarities among the threads in a typical GPU kernel, and on GPU-specific behaviors.

We implemented GPU-TRIDENT as LLVM compiler passes, and evaluated it on 17 GPU Kernels. We found that the accuracy of GPU-TRIDENT is comparable to FI both for the kernel as a whole, and for individual instructions, for most applications. Further, GPU-TRIDENT scales much better than FI with the number of samples, and is nearly 2 orders of magnitude faster. Finally, GPU-TRIDENT can guide selective instruction duplication techniques with comparable accuracy as FI, thus demonstrating its usefulness.

As future work, we will improve the accuracy of GPU-TRIDENT based on the inaccuracies identified in Section VI. We will also use GPU-TRIDENT to guide other selective protection techniques than instruction duplication. Finally, we plan to extend GPU-TRIDENT to other GPU families and programming models than CUDA.

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 (FYF) from UBC. We also thank the reviewers of SC 2020 for their insightful comments.

REFERENCES

[1] A program of blurring picture by multi-threads in CUDA. https://github.com/caofengnian/HPCCUDA. [Online; accessed March 2020].
[2] Benchmarks for CPU, GPU, Memory, Disk Performance. https://github.com/sswarnak77/Performance-Benchmarking. [Online; accessed March 2020].

[3] Parallel circuit solver. https://github.com/glaswep/hpc. [Online; accessed Apr. 2016].

[4] C. Chang, S. Lym, N. Kelly, M. B. Sullivan, and M. Erez. Evaluating and accelerating high-fidelity error injection for hpc. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 577–589, 2018.

[5] A. Chatzidimitriou, P. Bodmann, G. Papadimitriou, D. Gizopoulos, and P. Rech. Demystifying soft error assessment strategies on arm cpus: Microarchitectural fault injection vs. neutron beam experiments. In *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 26–38, 2019.

[6] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *International Symposium on Workload Characterization (IISWC 2009)*, pages 44–54. IEEE, 2009.

[7] Jeffrey J Cook and Craig Zilles. A characterization of instruction-level error derating and its implications for error detection. In *International Conference on Dependable Systems and Networks(DSN)*, pages 482–491. IEEE, 2008.

[8] Shuguang Feng, Shantanu Gupta, Amin Ansari, and Scott Mahlke. Shoestring: Probabilistic soft error reliability on the cheap. In *Architectural Support for Programming Languages and Operating Systems*, pages 385–396, 2010.

[9] L. Guo and D. Li. Moard: Modeling application resilience to transient faults on data objects. In *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 878–889, 2019.

[10] L. Guo, D. Li, I. Laguna, and M. Schulz. Fliptracker: Understanding natural error resilience in hpc applications. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 94–107, 2018.

[11] S. K. S. Hari, T. Tsai, M. Stephenson, S. W. Keckler, and J. Emer. Sassifi: An architecture-level fault injection tool for gpu application resilience evaluation. In *2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 249–258, 2017.

[12] Siva Kumar Sastry Hari, Sarita V Adve, Helia Naeimi, and Pradeep Ramachandran. Relyzer: Exploiting application-level fault equivalence to analyze application resiliency to transient faults. In *Architectural Support for Programming Languages and Operating Systems*, pages 123–134, 2012.

[13] Siva Kumar Sastry Hari, Radha Venkatagiri, Sarita V Adve, and Helia Naeimi. Ganges: Gang error simulation for hardware resiliency evaluation. In *ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, pages 61–72. IEEE, 2014.

[14] Mark Harris. Unified memory for CUDA beginners. NVIDIA Blog, 2016. [Online; accessed 18-Jan-2018].

[15] Charu Kalra, Fritz Previlon, Xiangyu Li, Norman Rubin, and David Kaeli. Prism: Predicting resilience of gpu applications using statistical methods. In *International Conference for High Performance Computing, Networking, Storage, and Analysis (SC), 2018*. ACM, 2018.

[16] Charu Kalra, Fritz Previlon, Norm Rubin, and David Kaeli. Armorall: Compiler-based resilience targeting gpu applications. *ACM Trans. Archit. Code Optim.*, 17(2), May 2020.

[17] I Karlin. Lulesh programming model and performance ports overview. https://computing.llnl.gov/projects/co-design/lulesh_ports1.pdf. [Accessed Apr. 2016].

[18] I. Laguna, M. Schulz, D. F. Richards, J. Calhoun, and L. Olson. Ipas: Intelligent protection against silent output corruption in scientific applications. In *2016 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 227–238, 2016.

[19] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization*, page 75. IEEE, 2004.

[20] Guanpeng Li, Karthik Pattabiraman, Chen-Yang Cher, and Pradip Bose. Understanding error propagation in GPGPU applications. In *International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 240–251. IEEE, 2016.

[21] Guanpeng Li, Karthik Pattabiraman, Siva Kumar Sastry Hari, Michael Sullivan, and Timothy Tsai. Modeling soft-error propagation in programs. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, 2018.

[22] Abdulrahman Mahmoud, Siva Kumar Sastry Hari, Michael B. Sullivan, Timothy Tsai, and Stephen W. Keckler. Optimizing software-directed instruction replication for gpu error detection. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, SC 18. IEEE Press, 2018.

[23] George B Mathews. On the partition of numbers. *Proceedings of the London Mathematical Society*, 1(1):486–490, 1896.

[24] Dmitry Mikushin. Enabling on-the-fly manipulations with llvm ir code of cuda sources. https://github.com/apc-llc/nvcc-llvm-ir. [Accessed Nov. 2018].

[25] Ming Zhang and N. R. Shanbhag. A CMOS design style for logic circuit hardening. In *2005 IEEE International Reliability Physics Symposium, 2005. Proceedings. 43rd Annual.*, pages 223–229, April 2005.

[26] Bin Nie, Lishan Yang, Adwait Jog, and Evgenia Smirni. Fault site pruning for practical reliability analysis of gpgpu applications. In *International Symposium on Microarchitecture (MICRO), 2018*, pages 749–761. IEEE, 2018.

[27] NVIDIA. Nvcc. https://developer.nvidia.com/cuda-llvm-compiler.

[28] Nahmsuk Oh, Philip P Shirvani, and Edward J McCluskey. Control-flow checking by software signatures. *Transactions on Reliability*, 51(1):111–122, 2002.

[29] Lucas Palazzi, Guanpeng Li, Bo Fang, and Karthik Pattabiraman. A tale of two injectors: End-to-end comparison of ir-level and assembly-level fault injection. pages 151–162, 10 2019.

[30] Vijay Janapa Reddi, Meeta S Gupta, Michael D Smith, Gu-yeon Wei, David Brooks, and Simone Campanoni. Software-assisted hardware reliability: abstracting circuit-level challenges to the software stack. In *Design Automation Conference*, pages 788–793. IEEE, 2009.

[31] B. Sangchoolie, K. Pattabiraman, and J. Karlsson. One bit is (not) enough: An empirical study of the impact of single and multiple bit-flip errors. In *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 97–108, 2017.

[32] Marc Snir, Robert W Wisniewski, Jacob A Abraham, Sarita V Adve, Saurabh Bagchi, Pavan Balaji, Jim Belak, Pradip Bose, Franck Cappello, Bill Carlson, et al. Addressing failures in exascale computing. *Institute for Computing in Science (ICiS). More infor*, 4:11, 2012.

[33] Vilas Sridharan and David R Kaeli. Eliminating microarchitectural dependency from architectural vulnerability. In *15th International Symposium on High Performance Computer Architecture*.

[34] Student. The probable error of a mean. *Biometrika*, pages 1–25, 1908.

[35] Jingweijia Tan, Nilanjan Goswami, Tao Li, and Xin Fu. Analyzing soft-error vulnerability on gpgpu micro architecture. In *IEEE International Symposium on Workload Characterization (IISWC), 2011 IEEE International Symposium on*, pages 226–235. IEEE, 2011.

[36] Shang Gao Lina Li Ruyu Zhang Jingweijia Tan Xiaohui Wei, Hengshan Yue. G-seap: Analyzing and characterizing soft-error aware approximation in gpgpus. In *Future Generation Computer Systems (2020)*, 2020.

[37] Keun Soo Yim, Cuong Pham, Mushfiq Saleheen, Zbigniew Kalbarczyk, and Ravishankar Iyer. Hauberk:lightweight silent data corruption error detector for gpgpu. In *International Parallel & Distributed Processing Symposium (IPDPS), 2011*, page 287. IEEE, 2011.