Practical Resilience Analysis of GPGPU Applications in the Presence of Single- and Multi-bit Faults

Lishan Yang, Bin Nie, Adwait Jog, and Evgenia Smirni

Abstract—Graphics Processing Units (GPUs) have rapidly evolved to enable energy-efficient data-parallel computing for a broad range of scientific areas. While GPUs achieve exascale performance at a stringent power budget, they are also susceptible to soft errors, often caused by high-energy particle strikes, that can significantly affect the application output quality. Understanding the resilience of general purpose GPU (GPGPU) applications is especially challenging because unlike CPU applications, which are mostly single-threaded, GPGPU applications can contain hundreds to thousands of threads, resulting in a tremendously large fault site space in the order of billions, even for some simple applications and even when considering the occurrence of just a single-bit fault. We present a systematic way to progressively prune the fault site space aiming to dramatically reduce the number of fault injections such that assessment for GPGPU application error resilience becomes practical. The key insight behind our proposed methodology stems from the fact that while GPGPU applications spawn a lot of threads, many of them execute the same set of instructions. Therefore, several fault sites are redundant and can be pruned by careful analysis. We identify important features across a set of 10 applications (16 kernels) from Rodinia and Polybench suites and conclude that threads can be primarily classified based on the number of the dynamic instructions they execute. We therefore achieve significant fault site reduction by analyzing only a small subset of threads that are representative of the dynamic instruction behavior (and therefore error resilience behavior) of the GPGPU applications. Further pruning is achieved by identifying the dynamic instruction commonalities (and differences) across code blocks within this representative set of threads, a subset of loop iterations within the representative threads, and a subset of destination register bit positions. The above steps result in a tremendous reduction of fault sites by up to seven orders of magnitude. Yet, this reduced fault site space accurately captures the error resilience profile of GPGPU applications. We show the effectiveness of the proposed progressive pruning technique for a single-bit model and illustrate its application to even more challenging cases with three distinct multi-bit fault models.

Index Terms—GPGPU Applications, Reliability Analysis, Fault Site Pruning

1 Introduction

Parallel Hardware Accelerators such as Graphics Processing Units (GPUs) are becoming an inevitable part of every computing system because of their ability to provide fast and energy-efficient execution for many general-purpose applications. GPUs work on the philosophy of Single Instruction, Multiple Threads (SIMT) programming paradigm [1] and schedule multiple threads on a large number of processing elements (PEs). Thanks to very large available parallelism, GPUs are used in accelerating innovations in various fields such as high-performance computing (HPC), artificial intelligence, deep learning, and virtual/augmented reality. Given the wide-spread adoption of GPUs in many Top500/Green500 supercomputers and cloud data centers, it is becoming increasingly important to develop tools and techniques to evaluate the reliability of such systems, especially since GPUs are susceptible to transient faults from high-energy particle strikes [2], [3], [4].

The typical approach to evaluate general purpose GPU (GPGPU) application resilience is by artificially but systematically injecting faults into various registers and then by examining their effects on the application output. These faults can result in: a) no change in application output (i.e., faults are masked), b) change in output due to data corruption while execution terminates successfully (i.e., faults are silent), and c) application crashes and hangs. With the exception of approximate computing where some executions that result in silent faults may be acceptable to the user [5], silent faults and crashes are considered undesirable. Consequently, high-overhead protection mechanisms such as check-pointing [6] and error correction codes (ECC) [7] are employed to strive for reliable application execution.

One of the major challenges in evaluating error resilience of applications even in the presence of a *single-bit fault* during the application execution is to obtain a very high fault coverage, i.e., inject a fault in all possible fault sites and record its effect. This procedure is very time consuming and tedious, especially in light of the fact that the total space of fault sites can be in the *order of billions*. Assuming a single-bit flip model, Table 1 quantifies the total number of fault injection sites for a large number of diverse GPGPU application kernels. The tremendous size of single-bit fault sites is due to the fact that each GPGPU kernel can spawn thousands of application threads and each thread is assigned to a dedicated amount of on-chip resources. For the calculation of fault sites reported in Table 1, we only consider soft errors that can occur in functional units (e.g., arithmetic logic unit and load-store unit) that are not protected by ECC [8]. Yet, the number of fault sites even for the single-bit case is tremendous, making an exhaustive approach absolutely not practical.

• *Authors are with the Department of Computer Science, William & Mary, Williamsburg, VA 23185 USA.*

TABLE 1: Various metrics (including the total number of possible fault sites) in the presence of a single-bit fault.

Suite	Application	Kernel Name	ID	# Threads	# Total Fault Sites
Rodinia	HotSpot	calculate_temp	K1	9216	3.44E+07
	K-Means	invert_mapping	K1	2304	1.47E+07
		kmeansPoint	K2	2304	9.67E+07
	Gaussian Elimination	Fan1	K1	512	1.63E+05
		Fan2	K2	4096	4.92E+06
		Fan1	K125	512	1.09E+05
		Fan2	K126	4096	8.79E+05
	PathFinder	dynproc_kernel	K1	1280	2.77E+07
	LU Decomposition (LUD)	lud_perimeter	K44	32	1.75E+06
		lud_internal	K45	256	6.84E+05
		lud_diagonal	K46	16	5.26E+05
Polybench	2DCONV	Convolution2D_kernel	K1	8192	6.32E+06
	MVT	mvt_kernel1	K1	512	6.83E+07
	2MM	mm2_kernel1	K1	16384	5.55E+08
	GEMM	gemm_kernel	K1	16384	6.23E+08
	SYRK	syrk_kernel	K1	16384	6.23E+08

In order to develop a robust and practical reliability evaluation for GPUs, prior works have considered a variety of fault injection methodologies such as LLFI-GPU [9] and SASSIFI [8] that sample a random subset of 1,000 fault sites to capture a partial view of the overall error resilience characteristics with 95% confidence intervals and error margins within a 6% range [10]. Here, we take an orthogonal approach – our goal is to prune the tremendously large fault site space using properties of GPGPU applications. Our pruning mechanism dramatically reduces the total number of required fault injections, in some cases to a few hundreds only while still maintaining superior accuracy.

To this end, we focus on the following fundamental observations relevant to GPGPU applications: a) GPGPU applications follow the SIMT execution style that allow many threads to execute the same set of instructions with slightly different input values, b) There is ample commonality in code across different threads, c) Each GPU thread can have several loop iterations that do not necessarily change the register states significantly, and d) changes in the precision/accuracy of register values do not necessarily change the final output of an application. By leveraging these properties, we propose a *progressive* pruning that preserves the application error resilience characteristics and consists of the following steps:

• *Thread-wise Pruning:* The first step focuses on reducing the number of threads for fault injection. We find that a lot of threads in a kernel have similar error resilience characteristics because they execute the same number and type of dynamic instructions. Based on the grouping of threads based on their dynamic instruction count, we select a small set of representative threads per kernel and prune the redundant fault sites belonging to other threads.

• *Instruction-wise Pruning:* Many of these selected representative threads still execute subsets of dynamic instructions that are identical across threads. This implies that the replicated subsets across threads can be considered only once. Therefore, the replicated fault sites are pruned while preserving the application error resilience characteristics.

• *Loop-wise and Bit-wise Pruning:* We observe that there is a significant redundancy in fault sites across loop iterations and register bit positions. Therefore, such redundant fault sites can be further pruned while accurately capturing the application error resilience characteristics.

We illustrate the effectiveness of the proposed methodology by showing that is able to reduce the fault site space by up to seven orders of magnitude while maintaining accuracy that is close to that of ground truth. In addition, we further investigate three multi-bit fault models: (1) multi-bit faults in the same word, (2) multiple single-bit faults occurring in the same thread, and (3) multiple single-bit faults on different threads. We illustrate that the proposed progressive fault site pruning technique can be readily extended to capture the error resilience characteristics of GPGPU applications for multi-bit fault models.

2 BACKGROUND AND METHODOLOGY

Baseline GPU Architecture. A typical GPU consists of multiple cores, also called streaming-multiprocessors (SMs) in NVIDIA terminology. Each core is associated with private L1 data, texture and constant caches, software-managed scratchpad memory, and a large register file. The cores are connected to memory channels (partitions) via an interconnection network. Each memory partition is associated with a shared L2 cache, and its associated memory requests are handled by a GDDR5 memory controller. Recent commercial GPUs typically employ single-error-correction double-error-detection (SEC-DED) error correction codes (ECCs) to protect register files, L1/L2 caches, shared memory and DRAM against soft errors, and use parity to protect the read-only data cache. Other structures like arithmetic logic units (ALUs), thread schedulers, instruction dispatch unit, and interconnect network are not protected [7].

GPGPU Applications and Execution Model. GPGPU applications leverage the single-instruction-multiple-thread (SIMT) philosophy and concurrently execute thousands of threads over large amounts of data to achieve high throughput. A typical GPGPU application execution starts with the launch of kernels on the GPU. Each kernel is divided into groups of threads, called *thread blocks*, which are also known as *Cooperative Thread Arrays* (CTAs) in CUDA terminology. A CTA encapsulates all synchronization and barrier primitives among a group of threads [11]. Having such an abstraction allows the underlying hardware to relax the execution order of the CTAs to maximize parallelism.

We selected applications from commonly used suites (i.e., Rodinia [12] and Polybench [13]) that cover a variety of workloads. As kernels of GPGPU applications implement independent modules/functions, we perform resilience analysis separately for each kernel. We focus on every static application kernel. For static kernels with more than one dynamic invocations, we randomly select one for fault injection experiments. Table 1 shows the evaluated 10 applications (16 kernels). In the rest of this paper, if the kernel index is not specified, it implies that the application contains only one kernel.

2.1 Baseline Fault Injection Methodology

We employed a robust fault injection methodology based on GPGPU-Sim [14], a widely-used cycle-level GPU architectural simulator. The usability of GPGPU-Sim with PTXPlus mode (which provides a one-to-one instruction mapping to actual ISA for GPUs [14], [15]) for reliability assessment is validated by GUFI [15], a GPGPU-Sim based framework. We inject faults using GPGPU-Sim with the PTXPlus mode.

For each experiment, we examine the application output to understand the effect of the injected fault. We classify the outcome of a fault injection into: (1) *masked output*, where the injected fault leads to no change in the application output, (2) *silent data corruption (SDC) output*, where the injected fault allows the application to complete successfully albeit with an incorrect output, and (3) *other output*, where the injected fault results in application hangs or crashes. The distribution (or percentage) of fault injection outcomes in these three different categories form the error resilience profile of a GPGPU application.

2.2 Baseline Fault Model

We focus on injecting faults in the destination registers to mimic the effect of soft errors occurred in the functional units (e.g., arithmetic and logic units (ALUs) and the load-store units (LSUs)) [8], [16]. The destination registers and associated storage are identified by thread id, instruction id, and bit position. Table 1 shows the number of threads spawned by each kernel and the total number of fault sites (also called fault coverage). The fault coverage for each application kernel (consisting of N threads) is calculated using Equation (1). Suppose that a target thread t ($t \in [1, N]$) consists of $M(t)$ dynamic instructions and that the number of bits in the destination register of instruction i ($i \in [1, M(t)]$) is $bit(t, i)$. The number of exhaustive fault sites is the summation of every bit in every instruction from every thread in the kernel:

$$FaultCoverage = \sum_{t=1}^{N} \sum_{i=1}^{M(t)} bit(t, i). \tag{1}$$

This number for the GPGPU kernels that we consider here is reported in the rightmost column of Table 1. These numbers are obtained under the context of a single-bit fault model. We start with the single-bit fault model to build a fault site pruning technique. Then, we extend the proposed technique to our multi-bit fault model.

2.3 Statistical Considerations

Looking at the number of exhaustive fault sites shown in Table 1, it is clear that it is not practical to perform fault injection runs for all fault sites. This is especially true when application execution time is very long. Taking GEMM from Polybench as an example and assuming that it takes (nominal) one minute to execute one fault injection experiment, then 7.73E+08 minutes (or about 1331 years) are needed to complete experiments for the entire fault site space (see the first row in Table 2). Therefore, it is desirable to reduce the number of fault injection experiments but also guarantee a statistically sound resilience profile (i.e., percentages of masked, SDC, and other outputs – see Section 2.1) of the considered kernel. Prior work [10] has shown that given an initial population size N (in our case, N is the number of exhaustive fault sites), a desired error margin e, and a confidence interval (expressed by the t-statistic), the number of required experiments n (in our case, fault sites) is:

$$n = \frac{N}{1 + e^2 \times \frac{N-1}{t^2 \times p \times (1-p)}}. \tag{2}$$

Note that p in the above equation is the program vulnerability factor, i.e., the percentage of fault injection outcomes that are in the masked output category. If $n \ll N$, (e.g., if the percentage of samples is less than 5% of the entire population), then N can be approximated by ∞, resulting in the following equation [17]:

$$\lim_{N \to \infty} n = \lim_{N \to \infty} \frac{N}{1 + e^2 \times \frac{N-1}{t^2 \times p \times (1-p)}} = \frac{t^2}{e^2} \times p \times (1-p). \tag{3}$$

Since p is the result of fault injection experiments, p is still unknown. To ensure that the number of fault injection experiments n is sufficient to capture the true p [10], then

$$n = max\{\frac{t^2}{e^2} \times p \times (1-p)\} = \frac{t^2}{4 \times e^2}, \tag{4}$$

where n is the minimum sample size (i.e., number of fault injection experiments) required to calculate the fraction of fault injection outcomes in the masked output category, with a certain confidence interval and a user-given error margin e. To maximize the term $p \times (1-p)$, p is set to 0.5.

Table 2 presents the required number of fault injection experiments (i.e., fault sites) in GEMM given a confidence interval and an error margin. We consider the reliability profile results of 60K experiments (with 99.8% confidence interval and an error margin of $e = 0.63\%$) as the *ground truth* [18]. Clearly, there is a significant discrepancy between the percentage of *masked* outputs for 60K versus 1K fault injections (see last column). Similarly to techniques in the CPU domain that aim at high accuracy [18], the goal of fault site pruning is to achieve the accuracy of the 60K results but with a much reduced number of experiments. Indeed, as we show in later sections for the 13 out of the 16 kernels that we analyzed here, we achieve the high accuracy of ground truth but with 1K experiments or less, for some kernels with as few as 318 experiments only (see Figure 10).

TABLE 2: Fault sites and other statistics for GEMM.

Confidence Interval	Error Margin	# Fault Sites	Estimated Time	Masked Output (%)
100%	0.0%	7.73E+08	1331 years	?
99.8%	±0.63%	60,181	40 days	24.2%
95%	±3.0%	1,062	16 hours	21.6%

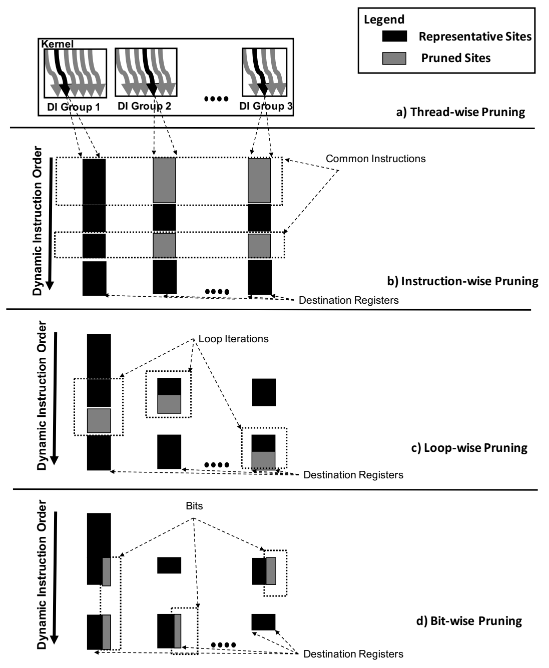


Fig. 1: Overview our Fault Site Pruning Mechanism.

3 Progressive Fault Site Pruning

Figure 1 provides an overview of our fault site pruning four-stage mechanism. This mechanism is progressive, i.e., every successive stage further reduces the number of fault sites of the previous one. There are four primary stages: *a) Thread-wise Pruning*, *b) Instruction-wise Pruning*, *c) Loop-wise Pruning*, and *d) Bit-wise Pruning*. In each stage as depicted in Figure 1, black parts represent the selected fault sites while the gray parts represent the pruned ones.

In the first stage, we perform *a) thread-wise pruning* where kernel threads are classified into different groups. This classification is based on the distribution of fault injection outcomes: threads in the same group share a similar application error resilience profile. From each group, we are able to randomly select *one thread* as the group representative. In Section 3.1, we show that the *dynamic instruction (DI) count* per thread can be used as proxy for effective thread classification. We classify threads based on their dynamic instruction count into several groups, then select one representative (i.e., one black thread) per group.

Next, we perform *b) instruction-wise pruning*, which leverages common blocks of code that are shared among the selected representative threads of the previous pruning step. Because of the SIMT nature of the GPU execution model, many threads execute the same subsets of instructions. These common instruction blocks are likely to have similar resilience (discussed further in Section 3.2), thus become candidates for pruning (see gray segments in Figure 1, stage b) Instruction-wise Pruning). Black segments are selected for fault injection and move to the next pruning stage.

In the subsequent stage, *loop-wise pruning*, we identify loops in the threads that are selected from the previous stage and we randomly sample several loop iterations to represent the entire loop block (we elaborate on how we do this sampling in Section 3.3). Within each loop, we are able to use a part of representative iterations (marked as black) and discard the rest (marked as gray), see Figure 1 stage c.

As a last step, with *bit-wise pruning*, we consider several pre-selected bit positions for fault injection. These bit positions are selected to cover a range of positions in registers to further reduce the fault site space (Section 3.4 gives the rationale behind the bit position selection). Similarly, to the rest of Figure 1, black bit positions are the selected fault sites while gray ones are pruned. Overall, Figure 1 gives a road-map of the progressive pruning steps.

3.1 Thread-Wise Pruning

GPGPU applications typically spawn thousands of threads. Therefore, injecting faults to all thread registers is not practical. To this end, we classify threads into groups that share similar resilience behavior. The challenge here is to choose an effective metric that can be easily extracted from the application to guide this classification.

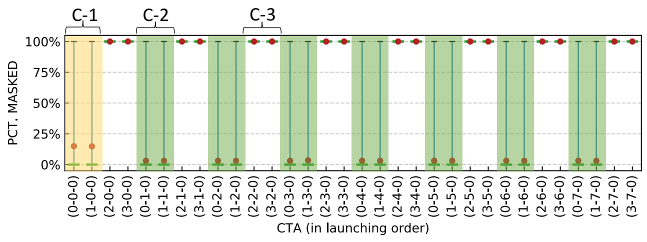
In order to develop a classification process, we study the error resilience characteristics of CTAs and threads of a kernel through a fault injection of over 2 million fault injection runs. We investigate the fault resilience features *hierarchically*, at the CTA-, thread-, and instruction-level. Our analysis illustrates that: 1) a few representative CTAs and threads can capture the error resilience characteristics of the entire kernel and 2) the number of dynamic instructions per thread (*iCnt*) can be used as an effective classifier to identify representative threads and guide the first pruning step.

3.1.1 CTA-wise Pruning

We first focus on understanding the error resilience characteristics at the CTA level. Although it is not feasible to perform an exhaustive fault injection campaign at this level, it is relatively manageable to run exhaustive experiments for target instructions. We select a diverse set of dynamic instructions including memory access (e.g., *ld*), arithmetic (e.g., *add* and *mad*), logic (e.g., *and* and *shl*), and special functional instructions (e.g., *rcp*), and from different code locations (e.g., beginning, middle, and end). Although the fault sites are already reduced by targeting certain instructions and narrowing down to few locations, the number of (reduced) fault sites per kernel is still large, e.g., $1,217K$ for HotSpot, $774K$ for 2DCONV, $412K$ for K-Means.

We resort to Equation 4 to obtain n=$60K$ random samples for every target instruction in a kernel. We use 2DCONV and HotSpot, which are diverse in terms of number of threads and similarity across threads. For each application kernel, we manually select 5 instructions that cover the aforementioned diversity, resulting in $300K$ fault injection runs per application kernel. Figure 2(a)-(b) shows the grouping results given by one target instruction for 2DCONV and HotSpot, respectively. The results for the remaining four target instructions are not shown for brevity.

Figure 2(a) shows the distribution of fault injection outcomes for all 32 CTAs in 2DCONV. CTAs are listed in the order of their launching time along the x-axis. For every CTA, we calculate the percentage of *masked* outputs (percentage of *SDC* and *other* outputs are not shown) for each of its 256 threads and show the distribution of *masked* outputs using boxplots (i.e., one boxplot for each CTA to illustrate salient points in the distribution of masked outputs, including the 25th and 75th percentiles, and the mean and median). We


(a) 2DCONV (line=34, opcode=*mad*)

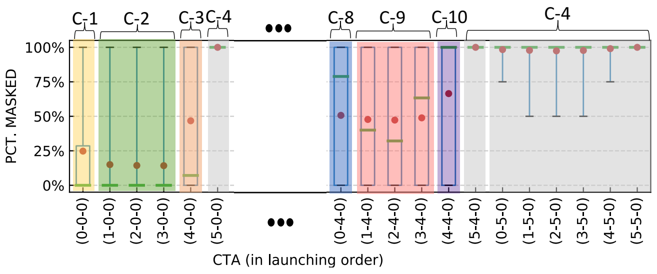
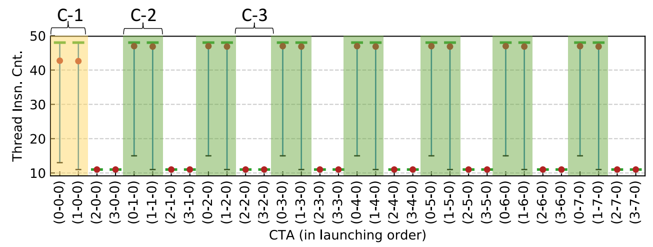

(b) HotSpot (line=52, opcode=*add*)

Fig. 2: CTA grouping after $60K$ fault injection runs of one target instruction for (a) 2DCONV and (b) HotSpot. CTAs with the same color are classified into the same group.


(a) 2DCONV

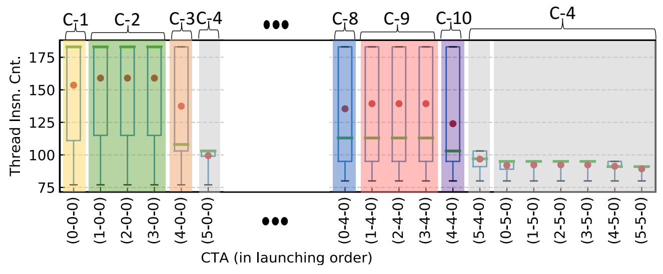

(b) HotSpot

Fig. 3: CTA grouping given by average dynamic thread instruction count (*iCnt*) per CTA for (a) 2DCONV and (b) HotSpot. CTAs with the same color are classified into the same group. The CTA classification is the same as the one observed in Figure 2.

observe that CTAs exhibit three distinct distributions as given by the different shapes of boxplots. Each group is marked by a different color. Therefore, 3 CTAs (one per group) is sufficient to represent the entire kernel. Similarly, Figure 2(b) shows the CTA grouping results for HotSpot. There are 36 CTAs in total, each containing 256 threads. For clarity, we show a few CTAs only. We observe that HotSpot has more diverse CTAs than 2DCONV and hence we classify its CTAs into 10 groups (C-1 to C-10).

Although the experiments illustrated in Figure 2 point to a promising methodology to obtain a first-order CTA grouping, it is obtained with $300K$ fault injection runs per kernel. This is still not always practical, as one can always opt to the random fault injection campaign [10], which requires $60K$ runs. Therefore, it is imperative to find an effective metric to guide pruning. We show that the number of dynamic instructions per thread (*iCnt*) is an alternative good measure for thread classification. This is encouraging as only *one* fault-free execution is sufficient to collect all the required *iCnt* information.

Figure 3(a)-(b) shows the distribution of thread *iCnt* per CTA for 2DCONV and Hotspot. Recall that each boxplot in Figure 2 represents the distribution of percentage of *masked* outputs. Similarly here, we are able to classify the CTAs into the same groups as in Figure 2 (both Figure 2 and 3 use the same color-code). Tables 3 and 4 report the grouping results guided by the average thread *iCnt* per CTA (given by Figure 3) for 2DCONV and HotSpot, respectively (see the left three columns). The above results confirm that *iCnt* is effective in capturing the error resilience characteristics at the CTA-level. Based on the grouping guided by *iCnt*, only a few CTAs per kernel are sufficient to capture the entire picture. We have conducted similar experiments for other application kernels (not shown here due to lack of space) that overwhelmingly support the above conclusion.

TABLE 3: CTA and thread groups for 2DCONV.

CTA Grp.	Avg. iCnt	CTA Proportion	Thd. Grp.	Thd. iCnt	Thd. Proportion*
C-1	43	6.25%	T-11	13	12.50%
			T-12	15	2.73%
			T-13	48	84.77%
C-2	47	43.75%	T-21	15	3.13%
			T-22	48	96.87%
C-3	11	50.00%	T-31	11	100.00%

* For each CTA group, we show its percentage of threads belonging to the corresponding thread group.

Observation-1: A few CTAs are enough to capture the error resilience characteristics of a kernel. These CTAs are selected based on the average thread dynamic instruction count (*iCnt*).

3.1.2 Thread-wise Pruning

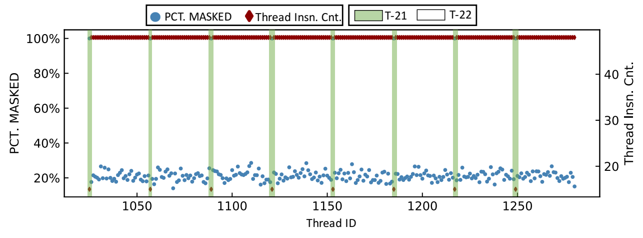
By narrowing down to only a few CTAs in a kernel, we are able to significantly reduce the number of fault sites. Yet, an exhaustive fault injection campaign using all threads in selected CTA representatives is still not viable. For example, for a CTA with 256 threads, if each thread executes an average of 100 dynamic instructions and if all destination registers are 32-bit wide, then a total of $819,200$ runs are needed. We continue the thread classification within each CTA in order to select only a few representative threads. We classify threads inside a CTA using (1) a large number of fault injection runs and (2) *iCnt* and confirm that the two methods lead to the same thread grouping results, see Figure 4. In other words, thread *iCnt* is also effective within a CTA to classify threads.

Figure 4(a) shows results for 2DCONV. Each blue dot represents the percentage of *masked* outputs in that thread (left y-axis) and each red dot indicates the corresponding thread *iCnt* (right y-axis). We mark threads in the same group with the same color. We observe a clear repeating pat-

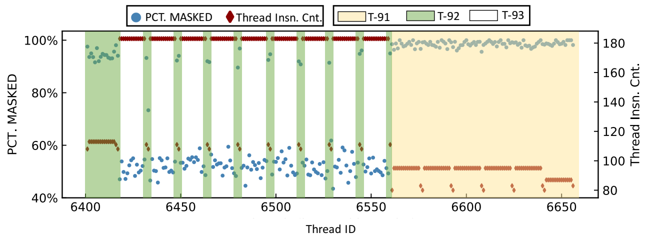
TABLE 4: CTA and thread groups for HotSpot.

CTA Grp.	Avg. iCnt	CTA Proportion	Thd. Grp.	Thd. iCnt Range	Thd. Proportion*
C-1	154	2.78%	T-11	77 − 98	23.44%
			T-12	111 − 115	10.55%
			T-13	183	66.02%
C-2	159	8.33%	T-21	77 − 90	12.50%
			T-22	108 − 115	16.41%
			T-23	183	71.09%
C-3	137	2.78%	T-31	77 − 103	45.31%
			T-32	108 − 115	8.98%
			T-33	183	45.70%
C-4	99	30.56%	T-41	77 − 99	28.91%
			T-42	103	71.09%
C-5	160	8.33%	T-51	89 − 111	18.75%
			T-52	113	5.08%
			T-53	115	5.08%
			T-54	183	71.09%
C-6	166	25.00%	T-61	108	6.25%
			T-62	111	6.25%
			T-63	113 − 115	10.94%
			T-64	183	76.56%
C-7	143	8.33%	T-71	95 − 108	43.75%
			T-72	113 − 115	7.03%
			T-73	183	49.22%
C-8	135	2.78%	T-81	80 − 98	45.31%
			T-82	111 − 113	8.98%
			T-83	183	45.70%
C-9	139	8.33%	T-91	80 − 95	37.50%
			T-92	108 − 113	13.28%
			T-93	183	49.22%
C-10	124	2.78%	T-101	80 − 103	60.94%
			T-102	108 − 113	7.42%
			T-103	183	31.64%

* For each CTA group, we show its percentage of threads belonging to the corresponding thread group.



(a) 2DCONV: CTA Group C-2



(b) HotSpot: CTA Group C-9

Fig. 4: Thread Grouping inside one CTA.

tern that allows for classifying all threads into two distinct groups (green and white), see Figure 4(a)):

1) T-21: threads with *iCnt*=15 and percentage of *masked* outputs at around 100%.
2) T-22: threads with *iCnt*=48 and percentage of *masked* outputs between 20% to 30%.

Table 3 reports the thread grouping details for 2DCONV (right three columns). A potential reason for such similarity in the distribution of fault injection outcomes among threads with different *iCnt* is the fact that these threads share large common code blocks, this is further discussed in Section 3.2.

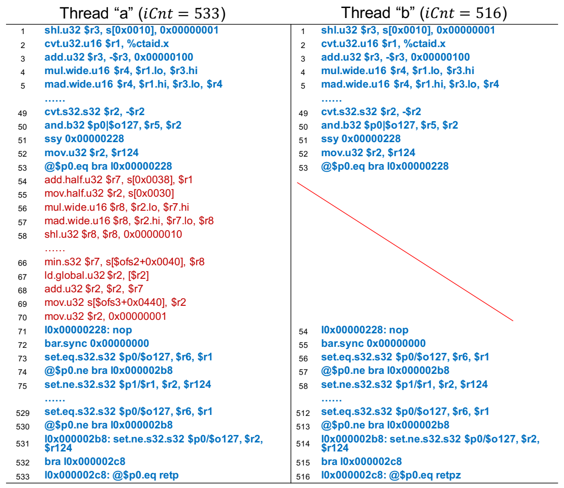
Figure 4(b) shows that threads in HotSpot can be also classified into several groups (Table 4). Due to the complexity of this kernel, we merge thread groups with similar *iCnt* together for visualization purposes, resulting in 3 distinct groups: green, yellow, and white. During the actual fault injection campaign, we still classify threads based on their exact *iCnt* (a total of 87 thread groups across selected CTAs) and select one representative thread per group.

It is important to perform the grouping in two steps: first at the CTA level and then at the thread level. Through our fault injection runs, we find that threads with the same *iCnt* from different CTAs could have different instructions and thus show different distribution of fault injection outcomes (this is observed in HotSpot and Gaussian K2). Therefore, the step of CTA-wise grouping cannot be skipped.

Observation-2: Threads can be further classified within a CTA. A few threads within a CTA are able to represent the CTA's error resilience characteristics.

3.2 Instruction-Wise Pruning

Our analysis shows that different threads normally share a large portion of common instructions. We aim to further prune the fault sites by finding common instruction blocks among the resulted set of thread representatives after the *thread-wise pruning* stage. We illustrate this observation using PathFinder. Figure 5 shows instruction snippets of its two representative threads ("a" and "b") chosen from the previous pruning stage. Comparing their PTXPlus code, lines 1 to 53 are the same; thread "a" has 17 more instructions in the middle; at the end, all the remaining 463 lines across the two threads are also the same.



Thread "a" ($iCnt = 533$)

```
1    shl.u32 $r3, s[0x0010], 0x00000001
2    cvt.u32.u16 $r1, %ctaid.x
3    add.u32 $r3, -$r3, 0x00000100
4    mul.wide.u16 $r4, $r1.lo, $r3.hi
5    mad.wide.u16 $r4, $r1.hi, $r3.lo, $r4
     ......
49   cvt.s32.s32 $r2, -$r2
50   and.b32 $p0|$o127, $r5, $r2
51   ssy 0x00000228
52   mov.u32 $r2, $r124
53   @$p0.eq bra l0x00000228
54   add.half.u32 $r7, s[0x0038], $r1
55   mov.half.u32 $r2, s[0x0030]
56   mul.wide.u16 $r8, $r2.lo, $r7.hi
57   mad.wide.u16 $r8, $r2.hi, $r7.lo, $r8
58   shl.u32 $r8, $r8, 0x00000010
     ......
66   min.s32 $r7, s[$ofs2+0x0040], $r8
67   ld.global.u32 $r2, [$r2]
68   add.u32 $r2, $r2, $r7
69   mov.u32 s[$ofs3+0x0440], $r2
70   mov.u32 $r2, 0x00000001
71   l0x00000228: nop
72   bar.sync 0x00000000
73   set.eq.s32.s32 $p0/$o127, $r6, $r1
74   @$p0.ne bra l0x000002b8
75   set.ne.s32.s32 $p1/$r1, $r2, $r124
     ......
529  set.eq.s32.s32 $p0/$o127, $r6, $r1
530  @$p0.ne bra l0x000002b8
531  l0x000002b8: set.ne.s32.s32 $p0/$o127, $r2,
     $r124
532  bra l0x000002c8
533  l0x000002c8: @$p0.eq retp
```

Thread "b" ($iCnt = 516$)

```
1    shl.u32 $r3, s[0x0010], 0x00000001
2    cvt.u32.u16 $r1, %ctaid.x
3    add.u32 $r3, -$r3, 0x00000100
4    mul.wide.u16 $r4, $r1.lo, $r3.hi
5    mad.wide.u16 $r4, $r1.hi, $r3.lo, $r4
     ......
49   cvt.s32.s32 $r2, -$r2
50   and.b32 $p0|$o127, $r5, $r2
51   ssy 0x00000228
52   mov.u32 $r2, $r124
53   @$p0.eq bra l0x00000228
54   l0x00000228: nop
55   bar.sync 0x00000000
56   set.eq.s32.s32 $p0/$o127, $r6, $r1
57   @$p0.ne bra l0x000002b8
58   set.ne.s32.s32 $p1/$r1, $r2, $r124
     ......
512  set.eq.s32.s32 $p0/$o127, $r6, $r1
513  @$p0.ne bra l0x000002b8
514  l0x000002b8: set.ne.s32.s32 $p0/$o127, $r2,
     $r124
515  bra l0x000002c8
516  l0x000002c8: @$p0.eq retpz
```

Fig. 5: PTXplus code of two representative threads for PathFinder. Blue lines indicate common instructions.

Table 5 shows the percentage of *masked* and *SDC* outputs for PathFinder *if soft errors are injected in their common portion only*. The distributions of fault injection outcomes that stem from this common block are quite close (see columns 4 and 5 in the table). Naturally, fault injections have to occur in the entire body of thread "a" to calculate its resilience, but since there is a common code block across the two threads, it can be used to extrapolate the distribution of fault injection outcomes of thread "b". This eliminates the need to inject faults in thread "b" and essentially prunes the fault sites generated for this thread. We introduce −0.078% error for the percentage of *masked* outputs and −0.031% error for the

TABLE 5: Effect of instruction-wise pruning for two threads.

Application	Thread	% Common Insn.	% MSK	% SDC
PathFinder	a	92.1%	89.4%	0.0%
	b	100.0%	90.1%	0.4%

percentage of *SDC* outputs (both minimal variations), but with a significant reduction of $12,344$ fault sites.

To confirm that this behavior persists across kernels, we conduct exhaustive experiments across the fault site space after *CTA-wise* and *thread-wise* pruning and confirm that common blocks of instructions across threads share a surprisingly similar distribution of fault injection outcomes (Table 6). The third column of Table 6 shows the percentage of pruned common instructions, and the 4th and 5th columns show the error of pruned results, compared to the exhaustive experiments before pruning common instruction blocks. Table 6 shows that the percentage of common instructions pruned in applications kernels ranges from 42.86% to 92.81% and that the error introduced by pruning common instruction blocks for *masked* and *SDC* outputs is -0.15% and -0.1%, respectively.

TABLE 6: Summary of instruction-wise pruning.

Application	Kernel	% Pruned Common Insn.	Introduced Error MSK	Introduced Error SDC
HotSpot	K1	92.81%	-0.14%	0.14%
PathFinder	K1	92.80%	0.03%	-0.09%
LUD	k46	80.00%	-0.78%	-0.70%
2DCONV	k1	66.67%	0.09%	-0.09%
Gaussian	K2	62.50%	-0.13%	0.13%
Gaussian	K126	42.86%	0.00%	0.00%
Average		72.94%	-0.15%	-0.10%

Note that several application kernels (e.g., 2MM, MVT, SYRK, and GEMM) after thread-wise pruning end up with only one representative thread. These kernels are not suitable for instruction-wise pruning and are therefore not included in the table. For Gaussian K1 and K2, and K-Means K1, instruction-wise pruning is also not applicable. For these application kernels, there are two representative threads, one with very few instructions (i.e., less than 10) and other with many (i.e., hundreds or thousands), leaving few opportunities to explore code commonality.

Observation-3: Different representative threads may share significant portions of common instructions. Therefore, distributions of fault injection outcomes of these common portions are similar. Consequently, a large number of fault sites can be pruned while achieving significant accuracy.

3.3 Loop-Wise Pruning

Table 7 shows the total number of instructions and the number of loop iterations. The kernels are sorted in increasing order by the portion of instructions in loops (after the loop is unrolled). Excluding kernels with no loops, a large portion of instructions in a kernel come from loop iterations, ranging from 65.79% in LUD K46 to 99.71% in MVT. We aim to discover whether the distribution of fault injection outcomes can be captured by a subset of loop iterations.

Towards this goal, we consider a number of randomly sampled iterations for fault injections. We present results for different fault site sizes, defined by the total number

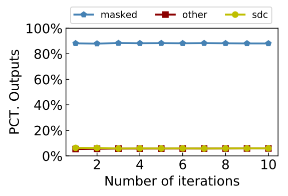
TABLE 7: Statistics related to loops.

Application	Kernel	# Thd.	# Loop Iter.	% Insn. in Loop
HotSpot	K1	9216	0	0.0%
2DCONV	K1	8192	0	0.0%
NN	K1	43008	0	0.0%
Gaussian	K1	512	0	0.0%
	K2	4096	0	0.0%
	K125	512	0	0.0%
	K126	4096	0	0.0%
LUD	K45	256	0	0.0%
	K46	16	120	65.79%
	K44	32	120	78.75%
K-Means	K1	2304	34	82.42%
	K2	2304	170	87.6%
PathFinder	K1	1280	20	92.84%
SYRK	K1	16384	128	98.13%
2MM	K1	16384	128	98.18%
GEMM	K1	16384	128	98.21%
MVT	K1	512	512	99.71%

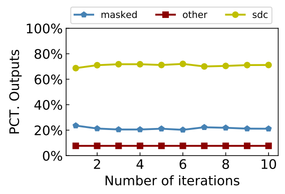
of sampled iterations (num_iter) ranging from 1 to 15. Figure 6 shows the impact of num_iter on the distribution of fault injection outcomes for PathFinder, SYRK, and K-Means K1. For K-Means K1, we show the effect of two different random seeds for sampling the loop iterations. The distribution of fault injection outcomes is stable after a certain number of sampled loop iterations. Looking closer into the application source code, we observe that: 1) several loop conditions are controlled by constants and not variables that are changed within the loop and 2) there is no data communication among different loop iterations. Therefore, there is no error propagation among loop iterations, thus sampling is sufficient for obtaining the distribution of fault injection outcomes.

Figure 6 shows that different applications require different numbers of sampled loop iterations to reach stability for the percentage of masked, SDC, and other outputs. Figure 6(a) shows that PathFinder requires 3 sampled loop iterations. Figure 6(b) shows that the output of SYRK becomes stable after 8 sampled loop iterations. In both cases the trend is clear. For K-Means K1 (Figure 6(c)), there is no clear trend with a few sampled iterations but results stabilize when the number of sampled loop iterations reaches 15. To further explore the behavior of this kernel, we sample the loop iterations of K-Means K1 using another random seed. Figure 6(d) reports the results and shows that stability is again achieved with 15 loop iterations, as shown in Figure 6(c). Even with different seeds, stability occurs with the same number of loop iterations for K-Means K1. Since different loop iterations process similar data, changing the seed does not make a significantly affect the application reliability profile. In general, different benchmark kernels need different number of loop iterations; this shows the differences of data among different applications.

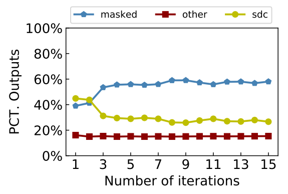
To summarize, Figure 6 suggests that randomly sampling a few iterations is generally sufficient in capturing the distribution of fault injection outcomes of application kernels. This offers another way to further reduce the fault sites within a thread. Therefore, we randomly add iterations one by one, until the result is stable. For the examined kernels, the number of iterations sampled among loops differs from a minimum of 3, to a maximum of 15, with an average of 7.22 iterations across all application kernels.



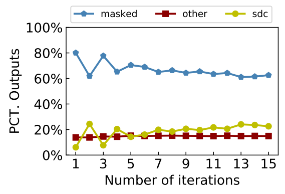
(a) PathFinder (Max # of Loop Iterations=20)



(b) SYRK (Max # of Loop Iterations=128)



(c) K-Means K1 (Max # of Loop Iterations=34)



(d) K-Means K1, using a different seed

Fig. 6: Impact of loop-wise pruning on distribution of fault injection outcomes for (a) PathFinder, (b) SYRK, and (c)-(d) for K-Means K1 with different random seeds.

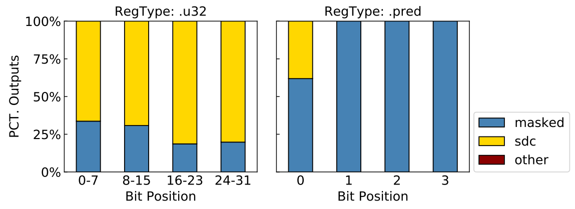
Observation-4: Distribution of fault injection outcomes in a kernel can be captured by a subset of iterations in the loop. This provides an opportunity for fault site pruning thanks to the abundance of instructions in a loop.

3.4 Bit-Wise Pruning

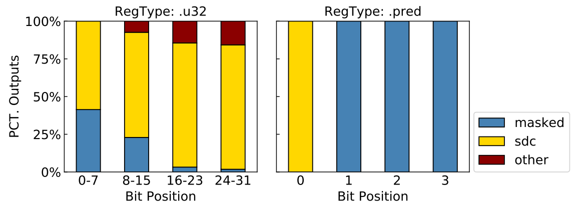
We explore whether we can further prune the fault site space from the perspective of bit positions. The intuition is that not all bit positions contribute equally to incorrect outputs. One may assume that bit flips in higher bit positions would produce more problematic outputs as the difference between the original value and flipped value tends to be larger. However, this intuition does not always hold true. The error pattern depends on application kernels and register types.

Figure 7 presents the distribution of fault injection outcomes for two major types of registers (i.e., *.u32* and *.pred*) for 2DCONV and MVT. We evenly partition bit positions in a register into 4 sections and show the distribution of fault injection outcomes for every section. First, we notice that for register type *.u32*, the intuition of higher bit sections having more problematic outputs holds for both application kernels. For MVT, the percentage of *masked* outputs decreases with increasing bit positions and becomes almost invisible in the higher two bit sections. For register type *.pred* that has 4 bits, we observe that for both applications, the lowest bit position results in output errors, while the higher three bit positions are very error resilient (they result only in *masked* outputs). This is the nature of 4-bit predicate system [19]: the highest three bits in register type *.pred* are used for the overflow flag, carry flag, and sign flag, respectively, while the lowest bit represents the zero flag. Within the context of the applications we study in this work, only the zero flag is used for branch conditions, so we can confidently prune the other three bit positions within *.pred*.

Note that since the *.pred* register is not a common one, the scope of pruning is not significant. For *.u32* (see Figure 7) there is a consistent pattern as a function of the bit position. Therefore, we evenly select bit positions across the whole

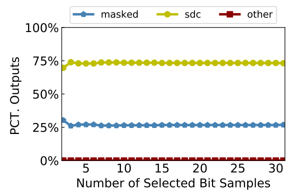


(a) 2DCONV

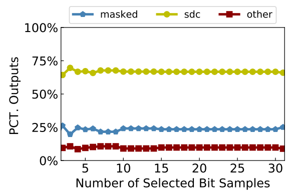


(b) MVT

Fig. 7: Distribution of fault injection outcomes of different bit position sections of two major register types (*.u32* and *.pred*) for (a) 2DCONV and (b) MVT.



(a) 2DCONV



(b) MVT

Fig. 8: Impact of bit-wise pruning on distribution of fault injection outcomes for (a) 2DCONV and (b) MVT (all registers). Percentage of outputs stabilizes at 16 bits.

register, increasing the number of selected bit samples from 1 to all 32 bit positions. Note that the selected bits are separated by equal intervals. For instance, for a 32-bit register and selecting 8 bit samples, we focus on bits in the following positions $\{3, 7, 11, 15, 19, 23, 27, 31\}$.

Figure 8 shows the results. For 2DCONV (see Figure 8 (a)), the change in distribution of fault injection outcomes changes as the number of sampled bits increase. This behavior persists in Figure 8 (b) for MVT. Overall, sampling 16 bits is sufficient and the space can be further pruned.

Observation-5: It is possible to reduce the number of fault sites by examining only a subset of bit positions.

4 Evaluation

In this section, we evaluate the proposed progressive pruning methodology by comparing with $60K$ random experiments (baseline case, see Section 2.3). The error margin and confidence interval of *baseline* are set to 0.63% and 99.8%, respectively. Figure 9 shows the comparison results. We observe that our pruning method produces very accurate error resilience estimations for several benchmark kernels including Hotspot, K-Means K2, Gaussian K2, Gaussian K126, PathFinder, LUD K44, LUD K46, 2DCONV, GEMM, and SYRK. For these kernels, the difference in terms of the percentage of *masked* outputs comparing with *baseline* is always less than 1%. For the remaining kernels, there is no significant mismatch from the *baseline*. On average, the differences in terms of *masked*, *SDC*, and *other* outputs are 1.68%, 1.90%, and 1.04%, respectively.

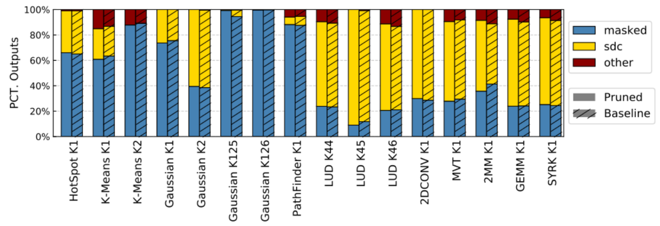


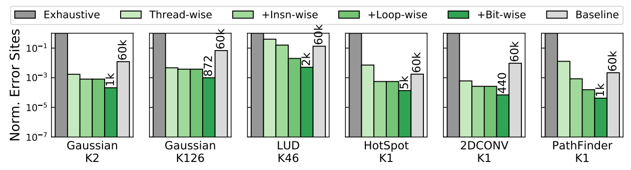
Fig. 9: Error resilience comparison of progressive fault site pruning techniques against the ground truth (baseline).

Next, we compare the effectiveness of the proposed progressive feature-based pruning in terms of fault site reduction. Figure 10 shows the comparison results. Note that we use log scale with a base of 10 for the y-axis. The number of fault sites left after each pruning step is normalized by the original exhaustive fault sites for every application kernel for cross-kernel comparison. The height of each bar represents the normalized number of fault sites after each step and the decrease in bar height from the previous bar indicates the reduction in fault site space. The last two bars in each sub-figure report also a number that indicates the fault site size of the fully pruned space versus the $60K$ baseline case which is the closest to the ground truth. Note that our pruning technique needs one-time offline profiling to collect the application features needed for pruning.

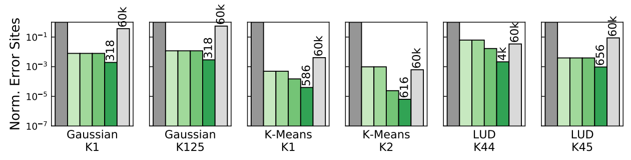
We observe from Figure 10(a) that *Thread-wise pruning* is most effective, as it reduces the magnitude of the number of fault sites by up to 5 orders of magnitude and use only a few representative threads (i.e., less than 10) per application kernel. This is a significant reduction compared to the original number of threads per kernel, e.g., 1 representative out of 16384 threads for GEMM, SYRK, and 2MM, and 6 representatives out of 8192 threads for 2DCONV. Such efficient first-order thread-wise pruning lays a substantial base for the following steps. One important clarification is that any later pruning is performed on the selected thread representatives, therefore further reductions after this step are expected to be modest.

Instruction-wise pruning exploits the commonality among the thread representatives selected in the previous step. It is important to clarify that kernels in the second row (see Figure 10 (b)) are not suitable for *Instruction-wise pruning*, because their representative threads do not have many common instruction blocks. Kernels in Figure 10 (c) are not applicable to *Instruction-wise pruning* as there is only one thread group per kernel, i.e., they only have a single representative thread. Comparing results within the first row of Figure 10, we observe that *Instruction-wise pruning* is most effective for HotSpot and PathFinder, with a reduction of 92.81% and 92.80% instructions, respectively.

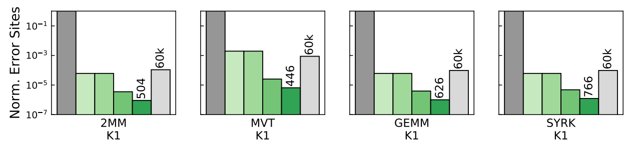
Loop-wise and *Bit-wise pruning* progressively contribute to further reduction. The effectiveness of *Loop-wise pruning* depends on the percentage of loop instructions in the fault sites left by the previous step. We observe a large reduction in K-Means K2, LUD K46 and matrix-related applications including 2MM, GEMM, SYRK, and MVT. This matches the fact that there is a large portion of loop instructions in these kernels (see Table 7). On the other hand, the effectiveness



(a) Kernels with instruction-wise commonality



(b) Kernels without instruction-wise commonality



(c) Kernels not applicable to instruction-wise commonality

Fig. 10: Fault site reduction comparison based on various feature-based pruning techniques. “+” indicates that each pruning stage is progressively built upon the pruned sites resulted from the previous stage.

of *Bit-wise pruning* is relative stable, i.e., the percentage of reduction in fault sites obtained by *Bit-wise pruning* is consistent across kernels.

Summary: The proposed pruning produces comparable distribution numbers of fault injection outcomes against a comprehensive baseline injection of $60K$ experiments which we use here as a statistically sound approximation of ground truth. For each step of feature-based progressive fault site pruning, we observe significant progressive reduction in the number of fault sites, ending up with only a few hundreds of fault sites in several kernels.

5 Multi-Bit Fault Models

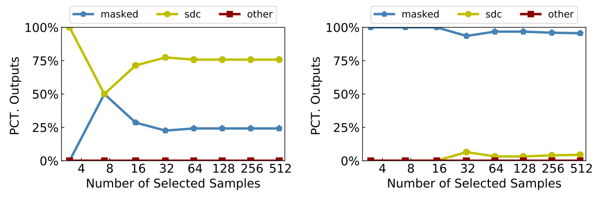
In this section, we extend our progressive pruning technique for *multi-bit faults* during a single run. As presented in Table 1, the number of fault sites is already tremendous even for single-bit fault injection. Extending the fault space to multi-bit faults, the number can grow exponentially. For example, the number of fault sites for single-bit fault injection for GEMM is 6.23E+08 and this number grows by 16 orders of magnitude of when injecting two single-bit faults. Equation (5) shows a general fault-space calculation, where x represents the number of faults and the inner part $\sum_{t=1}^{N}\sum_{i=1}^{M(t)} bit(t,i)$ is essentially Equation (1) (i.e., the fault space for a single-bit fault).

$$FaultCoverage\_Multi(x) = [\sum_{t=1}^{N}\sum_{i=1}^{M(t)} bit(t,i)]^{x} \tag{5}$$

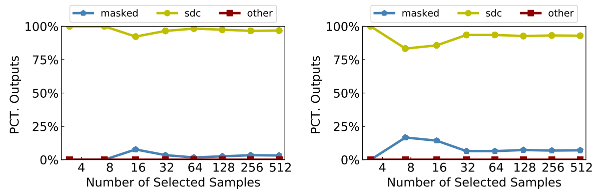
We consider the following multi-bit fault models:

1) Multi-bit faults in the same word;
2) Multiple single-bit faults in *different words* accessed by the same thread;
3) Multiple single-bit faults in different threads.

In the following sections, we illustrate how to extend the proposed progressive fault site pruning technique to each of the above models.



(a) 2DCONV



(b) MVT

Fig. 11: Impact of bit-wise pruning on 2-bit fault injection outcomes of different instructions from (a) 2DCONV and (b) MVT. Percentage of outputs stabilizes at 32 combinations.

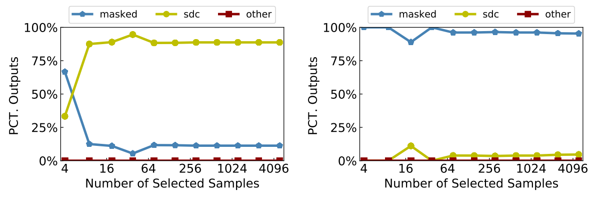
5.1 Multi-bit Faults in the same word

We first examine whether the four steps of progressive pruning can also apply when multi-bit faults occur in the same word. To this end, we followed the steps outlined in Sections 3.1 to 3.3 with 2-bit and 3-bit faults in the same word. The results are remarkably similar to those with a single bit faults and are not shown here in the interest of brevity. We conclude that the first three pruning steps directly apply for the single-word multi-bit model. Bit-wise pruning, the fourth step needs to be adjusted to consider the effect of injecting two (or three) faults within a word.

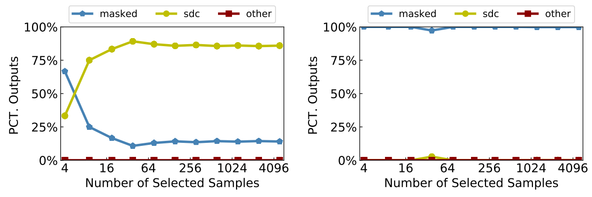
We start with considering 2-bit faults in the same word. For a 32-bit register, there are in total $\binom{32}{2}=$ 496 different combinations of bit flips. This change dramatically increases the fault site space. Following the steps outlined in Section 3.4, we aim to identify the number of 2-bit samples that can capture the reliability profile. For a 32-bit register, the exhaustive combination set of 2-bit faults (lexicographically ordered) is $\{(0,1),(0,2),\ldots(0,31),(1,2),\ldots(1,31),\ldots(30,31)\}$. Note that combinations as those shown above, do include consecutive (burst) bit flips. Within this space, we evenly select samples such that the selected combinations are separated by equal bit intervals. For example, when setting the number of samples to be 16 per word, then we inject faults in the $\{(0,16),(1,17),(2,19),\ldots,(15,31)\}$ locations.

For each of the applications, we select a set of representative instructions covering different operand types and different threads to perform 2-bit fault injection on the entire space, i.e., 496 combinations. We show the results of 4 different instructions from 2DCONV and MVT in Figure 11, when the number of selected samples ranges from 3 to all 496 possible combinations. Typically, after sampling 32 combinations, resilience stabilizes.

Similarly, in the case of 3-bit fault in the same word, there are in total $\binom{32}{3}=$ 4960 different combinations. Figure 12 shows the percentage of different outputs in various instructions using 3-bit fault in the same word model when the number of selected samples varies from 4 to 4960. For 3-bit faults, 64 samples can capture the resilience profile.



(a) 2DCONV



(b) MVT

Fig. 12: Impact of bit-wise pruning on distribution of 3-bit fault injection outcomes of different instructions from (a) 2DCONV and (b) MVT. Distribution stabilizes at 64 bits.

In general, when multiple bits are flipped within the same word, the same steps can be used compared to single-bit errors but bit-wise pruning requires more combinations (and subsequently more experiments). In general, the number of exhaustive fault sites increases one order of magnitude for 2-bit faults, then another order of magnitude for 3-bit, but bit-wise pruning only doubles and triples the number of fault sites for 2- and 3-bit, respectively (details per benchmark are not shown in the interest of brevity).

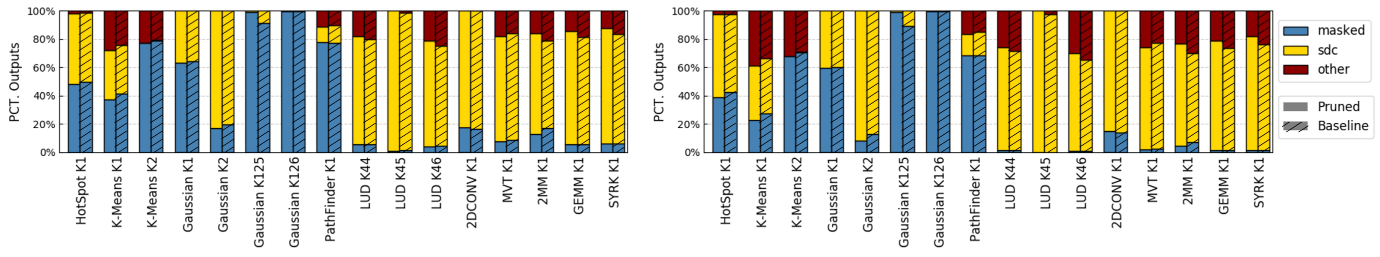
Observation-6: Fault site pruning can be extended to multi-bit faults in the same word model:
1) Thread-wise, instruction-wise, and loop-wise pruning steps stay unchanged.
2) Bit-wise pruning requires more samples of bit positions comparing to the single-bit case.

5.2 Multiple Single-Bit Faults

Multiple Single-Bit Faults in the Same Thread: We first state a necessary assumption to extend the result of single-bit fault injection to the context of multiple single bit faults in the same thread.

Assumption-1 *We assume that multiple single bit errors in a single thread are independent, i.e., they do not interact with each other.* After injecting one fault, an error may propagate to the dynamic instructions that are executed next. Injecting a subsequent fault to a register of an upcoming dynamic instruction that is already affected by the previous fault injection has very low probability to revert the register to its original correct state.

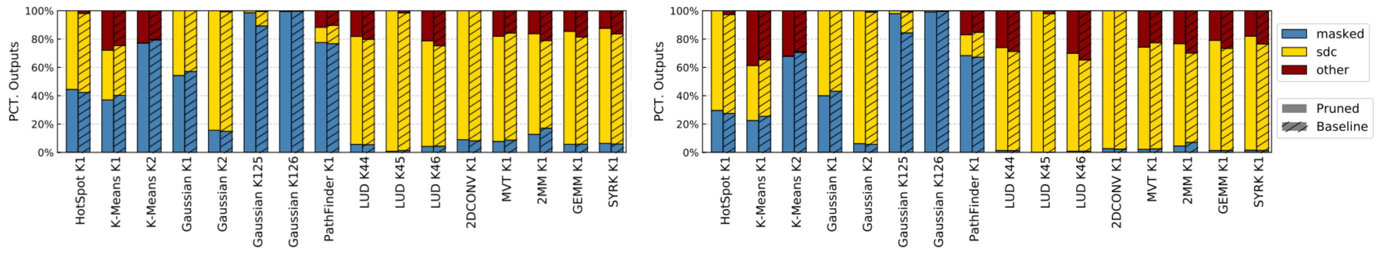
We first estimate the thread resilience with two-bit faults, then use the different thread resilience profiles to calculate the kernel resilience. For a particular thread, to determine the outcome of injecting two faults, we first select one fault site either randomly (i.e., as in the baseline case) or by using progressive pruning (see Section 3). If the first fault does not cause the program to crash, then we select a second fault site in the same thread. Assuming that the distribution of single-bit fault injection outcomes of one thread is $x\%$ *masked*, $y\%$ *SDC*, and $z\%$ *other* outputs such that $x\%+y\%+z\%=100\%$, then the outcome of a two-bit fault injection can be calculated as follows:



(a) Outcome of injecting two single-bit faults

(b) Outcome of injecting three single-bit faults

Fig. 13: Comparison of pruning against ground truth (baseline) for (a) two faults and (b) three faults in the same thread.



(a) Outcome of injecting two single-bit faults

(b) Outcome of injecting three single-bit faults

Fig. 14: Comparison of pruning against ground truth (baseline) for (a) two faults and (b) three faults in different threads.

1) $\textit{masked}\% = x\% \times x\%$,
2) $\textit{SDC}\% = x\% \times y\% + y\% \times x\% + y\% \times y\%$,
3) $\textit{other}\% = 1 - \textit{masked}\% - \textit{SDC}\%$.

Multi-bit fault injection model: For every thread, we obtain the outcome of multi-bit fault injection recursively. Injecting m faults can be decomposed into two steps. First, inject m-1 faults. If m-1 faults do not cause the program crash, then inject one more fault. Assuming that the thread resilience profile with a single fault injection is $x_1\%$ *masked*, $y_1\%$ *SDC*, and $z_1\%$ *other* outputs and the resilience profile of m-1 fault injection outcomes is $x_{m-1}\%$ *masked*, $y_{m-1}\%$ *SDC*, and $z_{m-1}\%$ *other* outputs, then the outcome of m faults is

1) $\textit{masked}\% = x_{m-1}\% \times x_1\%$,
2) $\textit{SDC}\% = x_{m-1}\% \times y_1\% + y_{m-1}\% \times x_1\% + y_{m-1}\% \times y_1\%$,
3) $\textit{other}\% = 1 - \textit{masked}\% - \textit{SDC}\%$.

Multiple Single-Bit Faults in Different Threads: When multiple single-bit faults occur in different threads, thread communication needs to be considered.

Assumption-2 *We assume that threads do not interact with each other.* Threads in the same thread block communicate using either shuffle instructions or shared memory. The benchmarks studied in this work do not have shuffle instructions. To understand the shared memory usage, we looked into the source code of the evaluated benchmarks (Table 1) and observe that only a few benchmarks (HotSpot, PathFinder and some kernels from LU Decomposition) use shared memory for thread communication. In these cases, we are able to provide an upper bound of the error resilience profile for the following reason: if threads communicate, the outcome of multi-bit faults should be the same or worse as compared to no communication.

When thread resilience is calculated, the outcome of multi-bit fault injections in different threads is calculated as in the case of multiple single-bit faults in the same thread. Note that when kernel resilience is calculated, the overall result is different because the final outcome depends on combining the resilience of *different* threads.

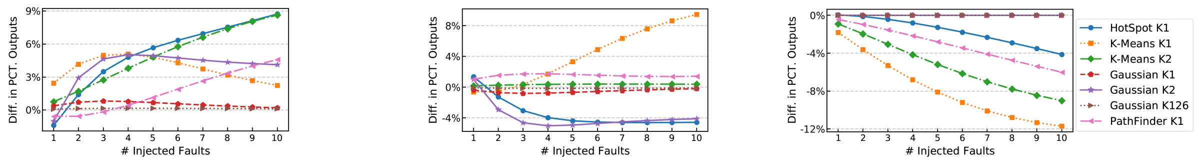
5.2.1 Evaluation

We compare the error distribution of injecting x-bit faults with pruning against using the random 60K method ($x \in [1, 10]$), for the two cases 1) where faults occur in the same thread and 2) in different threads. Then, we present how the error resilience profile of an application changes with an increasing number of injected faults.

Accuracy: We start with comparing the outcomes obtained using the proposed progressive fault site pruning technique against *baseline*, the closest approximation to ground truth as discussed in Section 2.1. Figure 13(a) shows the distribution of two single-bit fault injection outcomes for every benchmark kernel, when faults are injected into the same thread. The results obtained by pruning technique are still close to baseline, for most benchmark kernels. On average, the differences in terms of the percentage of *masked*, *SDC*, and *other* outputs are 1.52%, 2.65%, and 1.64%, respectively. In addition to double-bit, we also present the comparison of three single-bit fault injection outcomes obtained by the two techniques, see Figure 13(b). Differences with baseline start to become more visible when compared to the single-bit case (see Figure 9) and double-bit case (see Figure 13(a)). On average, the differences in terms of the percentage of *masked*, *SDC*, and *other* outputs are 1.94%, 3.42%, and 2.58%, respectively.

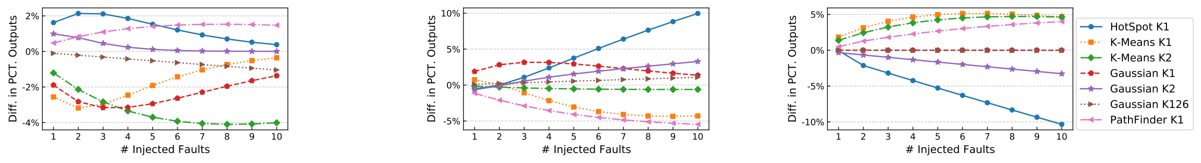
We present the distribution of two single-bit fault injection outcomes in different threads for every benchmark kernel, see Figure 14(a). We observe that the pruning method still produces accurate estimations of error resilience for most of the benchmark kernels. On average, the differences in terms of the percentage of *masked*, *SDC*, and *other* outputs are 1.81%, 2.58%, and 2.03%, respectively.

We also present the comparison of three single-bit fault injection outcomes obtained by the two techniques in Figure 14(b). We observe that differences with baseline start to become more visible when compared to the single-bit case (see Figure 9) and two bit case (see Figure 14(a)). The



(a) *masked* outputs (b) *SDC* outputs (c) *other* outputs

Fig. 15: Impact of the increasing number of injected faults in the same thread on the discrepancy with baseline for (a) *masked*, (b) *SDC*, and (c) *other* outputs.



(a) *masked* outputs (b) *SDC* outputs (c) *other* outputs

Fig. 16: Impact of the increasing number of injected faults in different threads on the discrepancy with baseline for (a) *masked*, (b) *SDC*, and (c) *other* outputs.

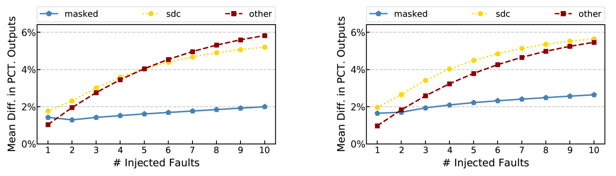
average differences in terms of the percentage of *masked*, *SDC*, and *other* outputs increase to 1.89%, 3.45%, and 2.79%, respectively.

Furthermore, we notice that for some benchmark kernels with three fault injections (including LUD K44, LUD K45, LUD K46, MVT, 2MM, GEMM, and SYRK), there are already almost no *masked* outputs (i.e., *masked*% $\leq$ 5% in Figure 13(b)), ditto for Figure 14(b)). For these kernels, it is clear that there is no need to inject more faults to them, they clearly do not have any resilience to more faults as all outcomes are *SDC* or *other*.

Figures 15 and 16 show that generally the difference with baseline increases as we inject more faults into the same thread or different threads. For most kernels, the difference is always within $\pm 5\%$ for all three types of outputs when injecting multiple faults. For HotSpot, the difference in terms of *SDC* and *other* outputs starts exceeding $\pm 5\%$ after injecting 6 and 5 faults, respectively. The difference of the percentage of *masked* outputs for HotSpot is always less than 2%, which shows that the pruning method is still able to provide a good estimation of the HotSpot's resilience even with multiple faults.

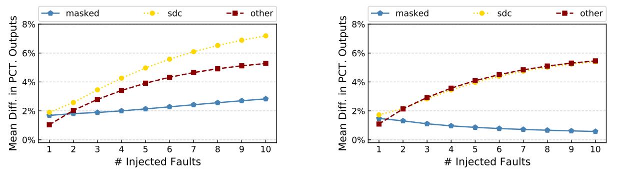
We summarize how the average discrepancy (across all kernels) changes over an increasing number of faults in Figures 17 and 18, for multiple single-bit fault injection in the same thread and different threads, respectively. For the fault model of injecting in the same thread, the average differences of *masked*, *SDC*, and *other* outputs for 10 faults are all below 6%: 2.64%, 5.64%, and 5.46%, respectively. For injecting faults in different threads, the average values increase by injecting more faults, end up to be as high as 2.83%, 7.19%, and 5.28% for *masked*, *SDC*, and *other* outputs, respectively, for 10 faults. If we exclude Gaussian K125, the most challenging kernel, and re-calculate the mean values, the average differences are less than 2% for *masked* outputs and less than 6% for *SDC* and *other* outputs.

Observation-7: The difference between the distribution of fault injection outcomes obtained by the proposed fault site pruning technique and *baseline* is acceptable, i.e., within $\pm 3\%$ for up to 3-bit fault injection and within $\pm 6\%$ for up to 10-bit fault injection.



(a) All kernels (b) All but Gaussian K125

Fig. 17: Mean error versus baseline for multiple single bit faults in the same thread calculated (a) across all kernels and (b) across all kernels but excluding Gaussian K125.

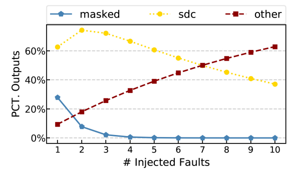


(a) All kernels (b) All but Gaussian K125

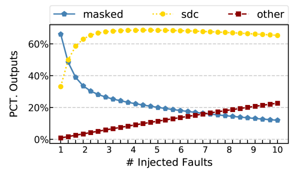
Fig. 18: Mean error versus baseline for multiple single bit faults in different threads calculated (a) across all kernels and (b) across all kernels but excluding Gaussian K125.

Impact of multiple single-bit faults: So far, we have shown that the distribution of fault injection outcomes obtained through the proposed progressive fault site pruning technique is close to the distribution achieved by *baseline* under the context of multiple single-bit fault injection. Therefore, in this section, we present how error resilience characteristics change over increasing number of faults using the result given by the proposed pruning method.

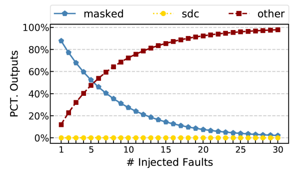
Figure 19 presents the distribution of fault injection outcomes of four representative benchmark kernels for multiple single-bit faults in the same thread. The percentage of *masked* outputs reduces significantly as we inject more faults and ends up at 0% within 10 injected faults for most benchmark kernels, see Figure 19 (a) for MVT. There are three exceptions: HotSpot (see Figure 19 (b)), K-Means K2 (see Figure 19 (c)) and PathFinder K1. These kernels are more error resilient than the others. Another exception is Gaussian K126 (see Figure 19 (d)), whose percentage of



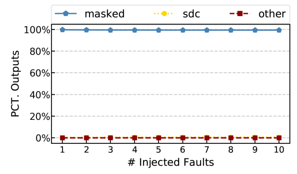
(a) MVT



(b) HotSpot

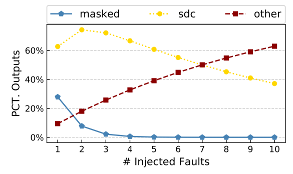


(c) K-Means K2

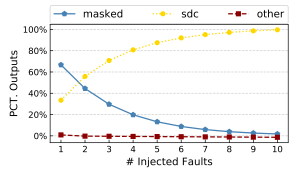


(d) Gaussian K126

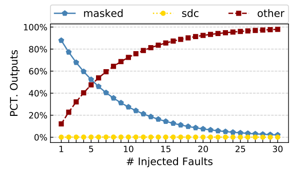
Fig. 19: Error resilience changes over increasing number of injected faults in the same thread for representative benchmark kernels.



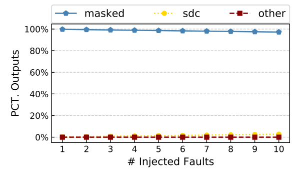
(a) MVT



(b) HotSpot



(c) K-Means K2



(d) Gaussian K126

Fig. 20: Error resilience changes over increasing number of injected faults for representative benchmark kernels.

masked outputs is still over 99% with 10 faults injected, implying that this kernel is even more error resilient.

The observations for injecting faults into different threads are similar, see Figure 20 with the exception of HotSpot: the percentage of *masked* outputs is close to 0% for 10 faults.

Observation-8: For the majority of benchmark kernels, the percentage of *masked* outputs stabilizes at 0% with 10 injected faults.

6 Related Work

High-level Reliability Analysis. Simulation-based analysis is employed widely in characterizing critical hardware structures for the purpose of finding vulnerabilities introduced by soft errors. Prior work [20], [21], [22] performed architectural vulnerability analysis (AVF) by performing exhaustive fault injection experiments. Faults are injected at various levels (e.g., application- or micro-architecture-level) and the effects of bit flips are measured by analyzing the application output. Application-level fault injection techniques are widely used in evaluating error-resilience characteristics for both CPU [23], [24] and GPU applications [25]. They are generally fast and still can provide detailed information. Another option is performing neutron-beam experiments [2], which is not always feasible. In this paper, we follow the process of studying reliability via fault injection, at PTXPlus-level, which is much faster and feasible than beam injection and is also reasonably accurate [15].

Fault Analysis. Although much work has been done on fault analysis and fault prediction models in real systems [26], [27], [28], [29], [30], [31], there are only a limited number of fault injection models designed specifically for GPUs. Fang et al. [16] proposed GPU-Qin to understand how faults affect application output in GPUs. A GPU debugging tool *cuda-gdb* [32] is leveraged by GPU-Qin to inject single bit errors into the destination operands. Similarly, Hari et al. [8] developed a fault injection tool, called SASSIFI, which injects different kinds of faults into destination register values, destination register indices and store addresses, and register files. The aforementioned studies adopt the commonly used single-bit fault model. Sangchoolie et al. [33] consider the impact of multi-bit faults for CPU applications. Here, we propose a new single-bit fault model for GPGPU applications and extend it to multi-bit fault models.

Fault-site Pruning. Within the CPU context, Relyzer [34] and MeRLiN [18] group fault sites into equivalence classes and select one or more pilots per class for fault injection. Directly transferring such techniques to GPU applications is not straightforward because GPU applications spawn hundreds to thousands of threads, leading to an enormous fault site space. The work presented here, extends the methodology in [35] for multi-bit faults.

Input-dependent resilience analysis. A common limitation of the fault injection works in both the GPU and CPU domains (included the work presented here) is that they are input-dependent, i.e., fault injection experiments have to be redone for different inputs. Minotaur [36] and vTrivent [37] leverage techniques from the software engineering and compiler domains, respectively, for reliability analysis of CPU applications with multiple inputs. To the best of our knowledge, there is no related work in the literature on this topic for GPGPU applications and is subject of our future work.

7 Conclusions

We demonstrate that fault sites in GPUs are very large and hence it is impractical to inject faults at every site to gain a comprehensive understanding of the GPGPU application error resilience. To address this, we present a progressive fault site reduction methodology based on GPGPU application-specific features. The key insight stems from the fact that while GPGPU applications spawn a lot of threads, many of them execute the same set of instructions. Therefore, several fault sites are redundant and can be pruned by a careful analysis of faults across threads, instructions, loop iterations within the same thread, and register bit positions. Across a set of 10 GPGPU applications (16 kernels in total) from the Rodinia and Polybench suites, we achieve a significant reduction in the number of fault-injection experiments (up to seven orders of magnitude) needed for a remarkably accurate GPU reliability assessment. In addition, we show how the proposed fault site pruning can be used in the more challenging case of multi-bit fault injections.

Acknowledgments

We thank the anonymous reviewers for their insightful comments. This work is supported by the National Science

Foundation (NSF) grants (#1717532 and #1750667) and performed in part using computing facilities at William & Mary.

References

[1] D. B. Kirk and W. W. Hwu, *Programming Massively Parallel Processors - A Hands-on Approach*. Morgan Kaufmann, 2010.

[2] V. Fratin, D. A. G. de Oliveira, C. B. Lunardi, F. Santos, G. Rodrigues, and P. Rech, "Code-dependent and architecture-dependent reliability behaviors," in *DSN 2018*.

[3] B. Nie, D. Tiwari, S. Gupta, E. Smirni, and J. H. Rogers, "A large-scale study of soft-errors on GPUs in the field," in *HPCA 2016*.

[4] B. Nie, J. Xue, S. Gupta, C. Engelmann, E. Smirni, and D. Tiwari, "Characterizing temperature, power, and soft-error behaviors in data center systems: Insights, challenges, and opportunities," in *MASCOTS 2017*.

[5] B. Nie, A. Jog, and E. Smirni, "Characterizing accuracy-aware resilience of GPGPU applications," in *CCGrid 2020*.

[6] H. Takizawa, K. Sato, K. Komatsu, and H. Kobayashi, "CheCUDA: A checkpoint/restart tool for CUDA applications," in *PDCAT 2009*.

[7] GP100 Pascal Whitepaper. [Online]. Available: https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf

[8] S. K. S. Hari, T. Tsai, M. Stephenson, S. W. Keckler, and J. S. Emer, "SASSIFI: an architecture-level fault injection tool for GPU application resilience evaluation," in *ISPASS 2017*.

[9] G. Li, K. Pattabiraman, C. Cher, and P. Bose, "Understanding error propagation in GPGPU applications," in *SC 2016*.

[10] R. Leveugle, A. Calvez, P. Maistri, and P. Vanhauwaert, "Statistical fault injection: Quantified error and confidence," in *DATE 2009*.

[11] A. Jog, O. Kayiran, N. C. Nachiappan, A. K. Mishra, M. T. Kandemir, O. Mutlu, R. R. Iyer, and C. R. Das, "OWL: cooperative thread array aware scheduling techniques for improving GPGPU performance," in *ASPLOS 2013*.

[12] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *IISWC 2009*.

[13] S. Grauer-Gray, L. Xu, R. Searles, S. Ayalasomayajula, and J. Cavazos, "Auto-tuning a high-level language targeted to GPU codes," in *Innovative Parallel Computing (InPar), 2012*.

[14] A. Bakhoda, G. L. Yuan, W. W. L. Fung, H. Wong, and T. M. Aamodt, "Analyzing CUDA workloads using a detailed GPU simulator," in *ISPASS 2009*.

[15] S. Tselonis and D. Gizopoulos, "GUFI: A framework for gpus reliability assessment," in *ISPASS 2016*.

[16] B. Fang, K. Pattabiraman, M. Ripeanu, and S. Gurumurthi, "GPU-Qin: A methodology for evaluating the error resilience of GPGPU applications," in *ISPASS 2014*.

[17] L. M. Leemis and S. K. Park, *Discrete-event simulation: A first course*. Pearson Prentice Hall Upper Saddle River, NJ, 2006, pg. 366.

[18] M. Kaliorakis, D. Gizopoulos, R. Canal, and A. González, "MeRLiN: Exploiting dynamic instruction behavior for fast and accurate microarchitecture level reliability assessment," in *ISCA 2017*.

[19] GPGPU-Sim Instruction Set Architecture. [Online]. Available: http://gpgpu-sim.org/manual/index.php/Main_Page#PTXPlus_Condition_Codes_and_Instruction_Predication

[20] N. Farazmand, R. Ubal, and D. Kaeli, "Statistical fault injection-based AVF analysis of a GPU architecture," in *in SELSE 2012*, 2012.

[21] H. Jeon, M. Wilkening, V. Sridharan, S. Gurumurthi, and G. Loh, "Architectural vulnerability modeling and analysis of integrated graphics processors," in *SELSE 2012*.

[22] J. Tan, N. Goswami, T. Li, and X. Fu, "Analyzing soft-error vulnerability on GPGPU microarchitecture," in *IISWC 2011*.

[23] D. Chen, G. Jacques-Silva, Z. Kalbarczyk, R. K. Iyer, and B. Mealey, "Error behavior comparison of multiple computing systems: A case study using linux on pentium, solaris on sparc, and aix on power," in *PDRC 2008*.

[24] K. S. Yim, Z. Kalbarczyk, and R. K. Iyer, "Measurement-based analysis of fault and error sensitivities of dynamic memory," in *DSN 2010*.

[25] K. S. Yim, C. M. Pham, M. Saleheen, Z. Kalbarczyk, and R. K. Iyer, "Hauberk: Lightweight silent data corruption error detector for GPGPU," in *IPDPS 2011*.

[26] C. D. Martino, Z. T. Kalbarczyk, R. K. Iyer, F. Baccanico, J. Fullop, and W. Kramer, "Lessons learned from the analysis of system failures at petascale: The case of blue waters," in *DSN 2014*.

[27] Y. Liang, Y. Zhang, A. Sivasubramaniam, M. Jette, and R. K. Sahoo, "BlueGene/L failure analysis and prediction models," in *DSN 2006*.

[28] A. J. Oliner and J. Stearley, "What supercomputers say: A study of five system logs," in *in DSN 2007, Edinburgh, UK, June 25-28, 2007*.

[29] A. Pecchia, D. Cotroneo, Z. Kalbarczyk, and R. K. Iyer, "Improving log-based field failure data analysis of multi-node computing systems," in *DSN 2011*.

[30] R. K. Sahoo, A. Sivasubramaniam, M. S. Squillante, and Y. Zhang, "Failure data analysis of a large-scale heterogeneous server environment," in *DSN 2004*.

[31] B. Nie, J. Xue, S. Gupta, T. Patel, C. Engelmann, E. Smirni, and D. Tiwari, "Machine learning models for GPU error prediction in a large scale HPC system," in *DSN 2018*.

[32] CUDA-GDB. [Online]. Available: http://docs.nvidia.com/cuda/cuda-gdb/#axzz4PHxjHEUB

[33] 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 *DSN 2017*.

[34] S. K. S. Hari, S. V. Adve, H. Naeimi, and P. Ramachandran, "Relyzer: exploiting application-level fault equivalence to analyze application resiliency to transient faults," in *ASPLOS 2012*.

[35] B. Nie, L. Yang, A. Jog, and E. Smirni, "Fault site pruning for practical reliability analysis of GPGPU applications," in *MICRO 2018*.

[36] A. Mahmoud, R. Venkatagiri, K. Ahmed, S. Misailovic, D. Marinov, C. W. Fletcher, and S. V. Adve, "Minotaur: Adapting software testing techniques for hardware errors," in *ASPLOS 2019*. ACM.

[37] G. Li and K. Pattabiraman, "Modeling input-dependent error propagation in programs," in *DSN 2018*.

Lishan Yang is a Ph.D. candidate in Computer Science Department at William & Mary. She received her B.S. degree in Computer Science from University of Science and Technology of China (USTC) in 2016. Her research interests are in GPU architecture, reliability analysis, and performance analysis. She is a member of IEEE and ACM.

Bin Nie received her B.S. degree in Software Engineering from Xiamen University, China (2012), her M.S. degree in Computer Science from Fordham University (2014), and her Ph.D. degree in Computer Science from William and Mary (2019). Her research interests include GPGPU reliability analysis, fault injection methodologies and heterogeneous system logs analysis (HPC, Data Centers, IT systems). She is a member of IEEE.

Adwait Jog is an Assistant Professor of Computer Science at William & Mary, Williamsburg, VA. His interests are on designing capable, energy-efficient, reliable, and secure general-purpose GPUs and other accelerators. He is the recipient of the NSF CAREER Award (2018), NSF CRII award (2017), and Penn State Outstanding Graduate Research Assistant Award (2014). He is a member of IEEE and ACM.

Evgenia Smirni is the Sidney P. Chockley professor of computer science at William and Mary, Williamsburg, VA. Her research interests include queuing networks, stochastic modeling, resource allocation, storage systems, cloud computing, workload characterization, and modeling of distributed systems and applications. She is an ACM Distinguished scientist and an IEEE Fellow.