2018 51st Annual IEEE/ACM International Symposium on Microarchitecture

Fault Site Pruning for Practical Reliability Analysis of GPGPU Applications

Bin Nie*, Lishan Yang*, Adwait Jog*, and Evgenia Smirni*
*College of William & Mary
Email: {bnie, lyang, esmirni}@cs.wm.edu, ajog@wm.edu

***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 applications is the purpose of this study. To this end, it is imperative to explore the range of application output by injecting faults at all the potential fault sites. This problem 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.**

In this paper, 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 can be practical. The key insight behind our proposed methodology stems from the fact that GPGPU applications spawn a lot of threads, however, 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 and instructions. We identify important features across a set of 10 applications (16 kernels) from Rodinia and Polybench suites and conclude that threads can be first classified based on the number of the dynamic instructions they execute. We 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 and analyzing: a) the dynamic instruction commonalities (and differences) across code blocks within this representative set of threads, b) a subset of loop iterations within the representative threads, and c) 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.

I. 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 now being used in accelerating innovations in various fields such as high-performance computing (HPC) [2]–[9], artificial intelligence, deep learning, virtual/augmented reality, and networking functions such as deep packet inspection [10].

Given the wide-spread adoption of GPUs in many Top500/Green500 supercomputers [11], [12] 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 hardware faults from high-energy particle strikes. One of the popular ways to evaluate general purpose GPU (GPGPU) application error 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 application's output due to data corruption but still execution terminates successfully (i.e., faults are silent), and c) application crashes and hangs. The latter two outcomes are certainly not desirable from the reliability point-of-view and hence a lot of high-overhead protection mechanisms such as check-pointing [13], [14] and error correction codes (ECC) [15]–[17] are employed to strive for reliable executions.

One of the major challenges in evaluating error resilience of applications is to obtain a very high fault coverage, i.e., inject faults in all possible fault sites and record its effect. This procedure is already very time consuming and tedious. In our own analysis of GPGPU applications, we have found that the total number of fault sites can be in the *order of billions*. Assuming a single-bit flip model, Table I quantifies the total number of fault injection sites for a large number of diverse GPGPU application kernels. The tremendous size of 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 I, we only consider soft errors that can occur in functional units (e.g., arithmetic logic unit and load-store unit) [18]. Yet, the number of fault sites is tremendous. Executing one experiment per fault site in such a vast space to collect application error resilience metrics is clearly very difficult and absolutely not practical.

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 [19] and SASSIFI [18] that sample a subset of fault sites to capture a partial view of the overall error resilience characteristics of GPGPU applications. These works claim that experiments on a small and randomly selected set of fault sites is sufficient

978-1-5386-6240-3/18/$31.00 ©2018 IEEE
DOI 10.1109/MICRO.2018.00066

TABLE I: Various metrics (including the total number of possible fault sites) related to considered GPGPU application kernels.

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

for results within 95% confidence intervals and error margins within a 6% range [20]. In this paper, we take an orthogonal approach – our goal is to prune the large amount of fault site space via carefully considering the properties of GPGPU applications. Our pruning mechanisms not only reduce the total number of required fault injections (in some cases to a few hundreds only while still maintaining superior accuracy), but also equivalently reduce the total time to complete the required experiments.

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 an 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) GPGPU applications themselves are error resilient and hence changes in the precision/accuracy of register values do not necessarily change the final output of an application. By leveraging these properties, we propose *progressive* pruning that systematically reduces the number of fault sites while preserving the application error resilience characteristics. Our proposed methodology consists of:

- *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 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:* Our detailed analysis show that many of these selected representative threads still execute subsets of dynamic instructions that are identical across threads. This implies that all instructions are not required to be considered for fault injection, and that the replicated subsets across threads can be considered only once. Therefore, the replicated fault sites are further 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 for further savings while accurately capturing the application error resilience characteristics.

To the best of our knowledge, this is the first work that quantifies the problem of high number of fault sites in GPUs and develops progressive pruning techniques by leveraging GPGPU application-specific properties. Our newly proposed methodology 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.

II. Background and Methodology

This section provides a brief overview of the baseline GPU architecture and applications, followed by a description on the basics of the fault injection methodologies and fault model.

A. GPU Architecture and Applications

Baseline GPU Architecture. A typical GPU consists of multiple cores, also called streaming-multiprocessors (SMs) in NVIDIA terminology [15]. 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 [15]–[17], 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 [15]–[17].

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 [1], [21]. 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 [22] and Polybench [23]) that cover a variety of workloads from different domains. Note that, as kernels of GPGPU applications normally implement independent modules/functions, we perform resilience analysis separately for each kernel. We focus on every static kernel in the application. For static kernels with more than one dynamic invocations, we randomly select one for fault injection experiments. Table I 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.

B. Baseline Fault Injection Methodology

We employed a robust fault injection methodology based on GPGPU-Sim [24], 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 [24], [25]) for reliability assessment is validated by GUFI [25], a GPGPU-Sim based framework. In this work, we inject faults using GPGPU-Sim with the PTXPlus mode.

For each experiment, we examine the application output to understand the effect of an injected fault. We classify the outcome of a fault injection into one of the three categories: (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 (or characteristics) of a GPGPU application.

C. 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)) [18], [26]. The destination registers and associated storage are identified by thread id, instruction id, and bit position. Table I shows a few characteristics of various application kernels, including 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 as per 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 and is given by:

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

This number for the GPGPU kernels that we consider in this paper is reported in the rightmost column of Table I.

D. Statistical Considerations

Looking at the number of exhaustive fault sites shown in Table I, 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, which is especially true for production software or workloads executing in data centers [27]). Taking GEMM from Polybench as an example and assuming that it takes a (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 II). 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 II-B) of the considered application kernel. To this end, prior work [20] 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 given by:

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

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 [28]:

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

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 [20], then

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

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 II 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

TABLE II: 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%

profile results of 60K experiments (with 99.8% confidence interval and an error margin of $e = 0.63\%$) as the *ground truth* [29]. Clearly, there is a significant discrepancy between the percentage of *masked* outputs for 60K versus 1K fault injections (see last column). The goal of our fault site pruning mechanism is to achieve the accuracy of the 60K results but with a much reduced number of experiments.

III. Progressive Fault Site Pruning

In this section, we explain the proposed error site pruning techniques while providing intuition along the steps.

A. Overview

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. Yet, thread classification is challenging. In Section III-B, 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.

In the next pruning stage, we perform *b) instruction-wise pruning*, which leverages common blocks of code that are shared among the selected representative threads of the previous pruning stage. We find that 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 characteristics (discussed further in Section III-C), 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 pruning 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 III-D). Within each loop, we are able

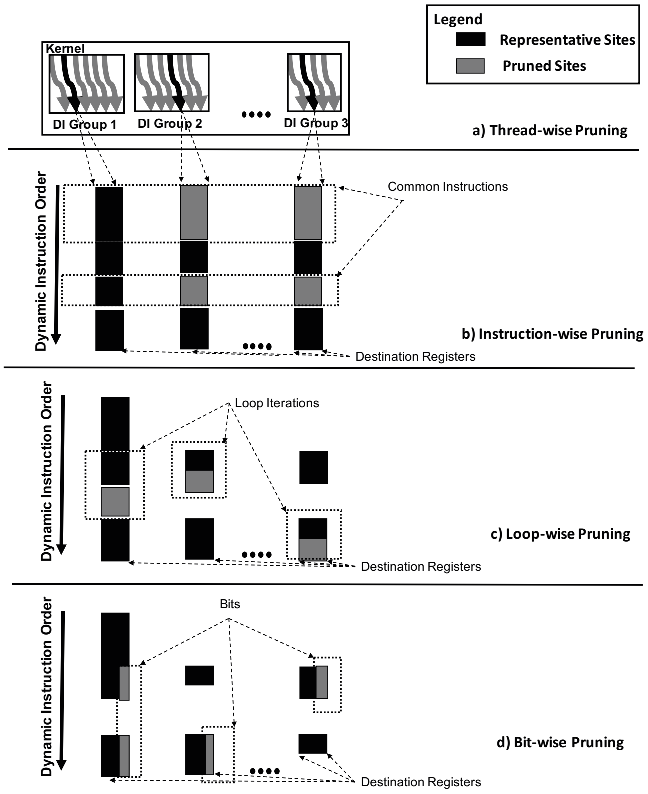


Fig. 1: Overview of the 4-stage Fault Site Pruning Mechanism.

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 III-E 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 that are discussed in detail in the following subsections.

B. Thread-Wise Pruning

As discussed in Section II, 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.

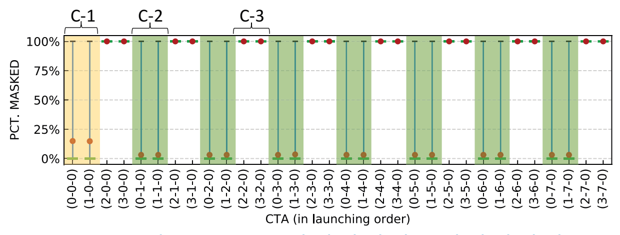
In order to develop a classification process, we study the error resilience characteristics of CTAs and threads of a kernel through a large fault injection campaign (i.e., over 2 million fault injection runs). We investigate the fault resilience features *hierarchically*, starting from CTA-, thread-, and instruction-level. Our analysis illustrates that:

- A few representative CTAs and threads can capture the error resilience characteristics of the entire kernel.

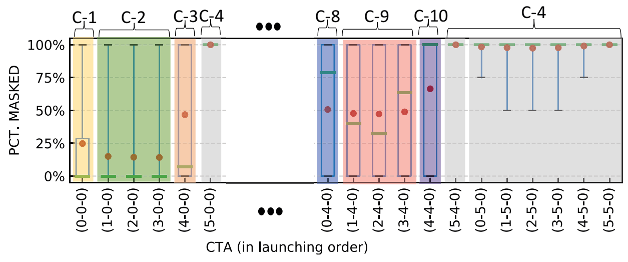
- The number of dynamic instructions (short as *iCnt*) per thread can be used as an effective classifier to identify representative threads and guide the first pruning step.

1) CTA-wise Pruning: We first focus on understanding the error resilience characteristics at the CTA level. Although it is not practical 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.

Instead, 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 nature 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.



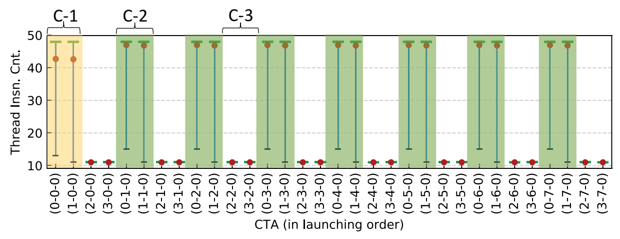
(a) 2DCONV (line=34, opcode=*mad*)



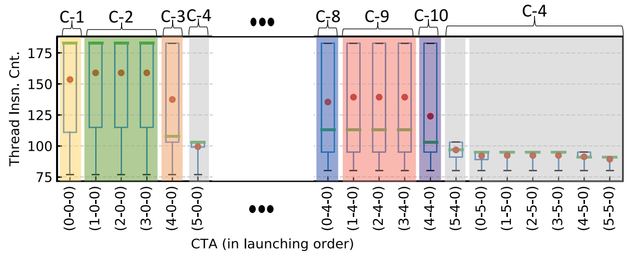
(b) HotSpot (line=52, opcode=*add*)

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. In the box plot, the horizontal green lines represent the median and red dots represent the mean.

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



(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. A significant similarity is is observed with Figure 2.

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 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 from the boxplots 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 [20], which requires $60K$ runs. Therefore, it is imperative to find an effective metric that can further prune the fault space. 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 results for 2DCONV and Hotspot. Each boxplot shows the distribution of thread *iCnt* per CTA. 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). Table III and IV 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).

To summarize, 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 III: CTA and threads 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*).

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 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. Therefore, we continue the thread classification within each CTA in order to select only a few representative threads. As done previously, we classify threads inside a CTA using (1) a large number of fault injection runs and (2) *iCnt*. We 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 pattern that allows for classifying all threads into two distinct groups (one marked with green color, the other one is uncolored, see Figure 4(a)):

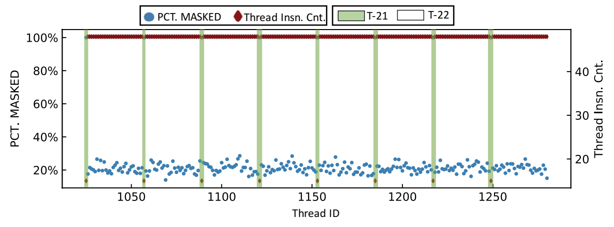
1) T-21: threads with *iCnt*=15 and percentage of *masked* outputs at around 100%.
2) T-22: threads with *iCnt*=48 and percentage of *masked* outputs between 20% to 30%.

Table III 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 III-C.

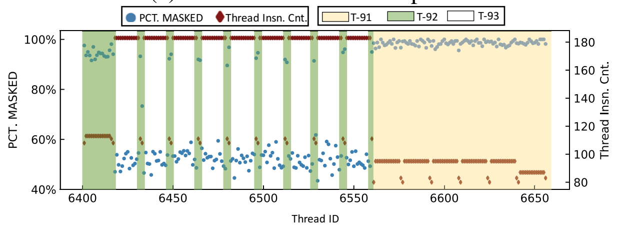
TABLE IV: CTA and threads groups for HotSpot.

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.

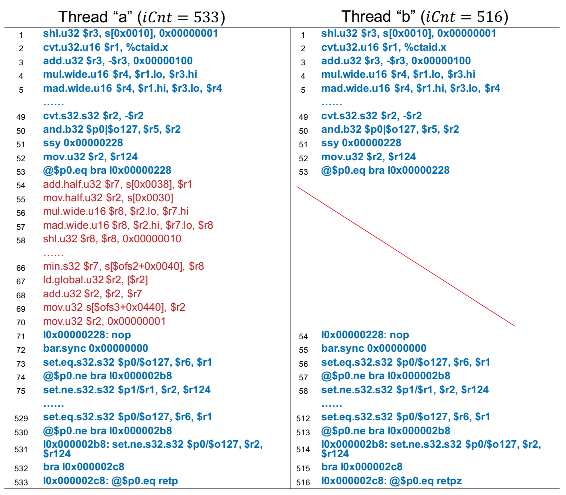
Figure 4(b) shows that threads in HotSpot can be also classified into several groups (Table IV). Due to the complexity of this kernel, we merge thread groups with similar *iCnt* together for visualization purposes, resulting in 3 distinct groups: one marked in green, one marked in yellow, and the third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

We find that 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.

C. 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 application. Figure 5 shows instruction snippets of its two representative threads ("a" and "b") chosen from the previous pruning stage. Comparing their PTXPlus code, dynamic instructions from the first line till line number 53 are all the same; thread "a" has 17 more instructions in the middle; at the end, all the remaining 463 instructions 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 comparison of two representative threads for PathFinder. Blue bold lines indicate common instructions.

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

TABLE V: 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%

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 VI). The third column of Table VI 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. This pruning technique is useful for complicated applications such as PathFinder and HotSpot, with the reduction of 92.81% instructions and 92.80% instructions, respectively. Table VI shows that the percentage of common instructions pruned in applications kernels ranges from 42.86% to 92.81% and the error introduced by pruning common instruction blocks for *masked* and *SDC* outputs is -0.15% and -0.1%, respectively.

TABLE VI: Summary of instruction-wise pruning for selected kernels. Other kernels do not exhibit instruction commonality.

Application	Kernel	% Pruned Common Insn.	Introduced Error	
			MSK	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.

D. Loop-Wise Pruning

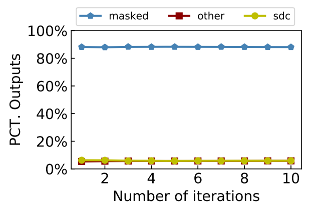
Table VII shows the total number of instructions and the number of loop iterations per kernel. The kernels are sorted in increasing order by the portion of instructions in loops (after the loop is unrolled). Excluding kernels with no loops, a large portion of instructions in a kernel come from loop iterations,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

TABLE VII: Statistics related to loops.

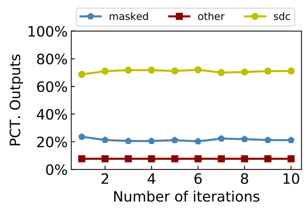
Application	Kernel	# Thd.	# Loop Iter.	% Insn. in Loop
HotSpot	K1	9216	0	0.0%
2DCONV	K1	8192	0	0.0%
NN	K1	43008	0	0.0%
Gaussian	K1	512	0	0.0%
	K2	4096	0	0.0%
	K125	512	0	0.0%
	K126	4096	0	0.0%
LUD	K45	256	0	0.0%
	K46	16	120	65.79%
	K44	32	120	78.75%
K-Means	K1	2304	34	82.42%
	K2	2304	170	87.6%
PathFinder	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%

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 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. We observe that 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 different loop iterations, thus sampling is sufficient for obtaining the distribution of fault injection outcomes. These observations hold true for the evaluated applications, but may not be true for other applications.

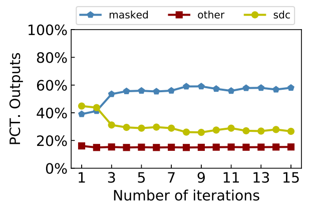
Figure 6 shows that different applications require different numbers of sampled loop iterations to reach stability for the percentage of masked, SDC, and other outputs. Figure 6(a) shows that PathFinder requires 3 sampled loop iterations. Figure 6(b) shows that the output of SYRK becomes stable after 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



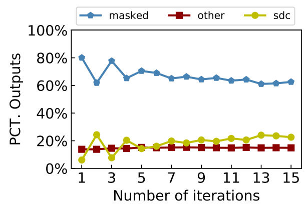
(a) PathFinder (Max # of Loop Iterations=20)



(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.

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).

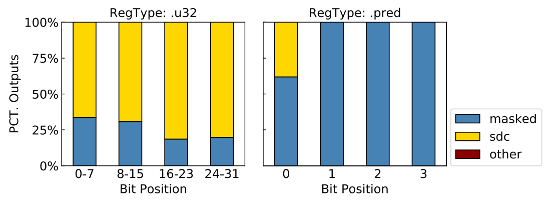
To summarize, Figure 6 suggests that randomly sampling a few iterations is generally sufficient in capturing the distribution of fault injection outcomes of application kernels. This offers another way to further reduce the fault sites within a thread. Similar experiments are done for all other applications and result in the same conclusion. 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.

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.

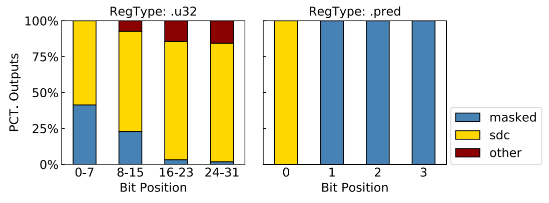
E. Bit-Wise Pruning

Beyond instruction-wise pruning, we explore whether it is possible to 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. Intuitively, 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(a)–(b) presents the distribution of fault injection outcomes for two major types of registers (i.e., *.u32* and *.pred*) for 2DCONV and MVT, respectively. We evenly partition bit



(a) 2DCONV



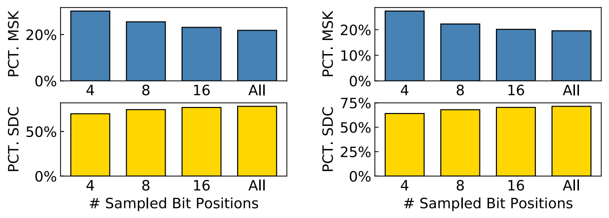
(b) MVT

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.

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 [30]: the highest three bits in register type *.pred* are used for 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 in register type *.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 select several bit positions from each register section resulting in a total of 4, 8, and 16 bit samples (at most, depending on the register size) and compare the distribution of fault injection outcomes with that of all bit positions. Note that the selected bits are separated by equal intervals. For instance, for a 32-bit register and selecting 2 bit positions per section, 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 promising as fault site space can be significantly pruned.

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



(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.

IV. Evaluation

In this section, we evaluate the proposed progressive pruning methodology by comparing with $60K$ random experiments (baseline case, see Section II-D).

We calculate the distribution of fault injection outcomes for every application kernel and compare with the percentage numbers given by the *baseline* (the closest approximation to ground truth as discussed in Section II-B). 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*, *other* outputs are 1.68%, 1.90%, and 1.64%, respectively.

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 subfigure 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 the most effective, as it reduces the magnitude of the number of fault sites by up to 5 orders of magnitude. With *Thread-wise pruning*, we only use 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

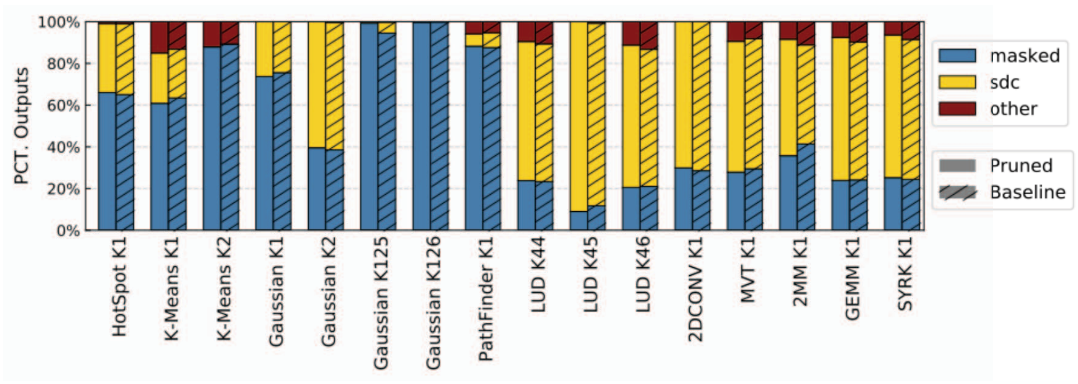


Fig. 9: Error resilience comparison of progressive fault site pruning techniques against the ground truth (baseline).

thread-wise pruning lays a substantial base for the following steps. One important clarification that needs to be stated 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 the reduction of 92.81% and 92.80% instructions, respectively.

Loop-wise and *Bit-wise pruning* progressively contribute to the reduction of the fault sites for each application kernel. 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 VII). On the other hand, the effectiveness 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.

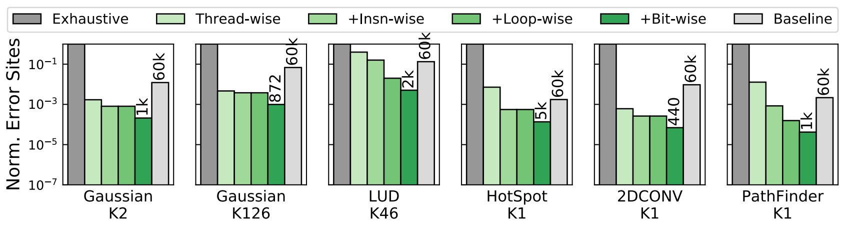
Summary: We present results of the 10 applications (16 kernels) using the pruned fault site subspaces outlined above to seek the distribution of application outputs (*masked*, *SDC*, and *other*). Our proposed mechanism is able to produce comparable distribution numbers of fault injection outcomes against 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.

V. Related Work

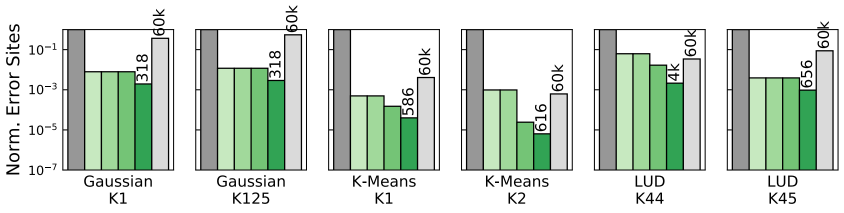
To the best of our knowledge, this is the first work that identifies the problem of large number of fault sites that make GPU reliability assessment impractical and proposes ways to efficiently address it. In this section, we briefly discuss works that are most relevant to this study.

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 [31]–[33] performed architectural vulnerability analysis (AVF) by performing exhaustive fault injection experiments. For the analysis purposes, 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 [34], [35] and GPU applications [36]. They are generally fast and still can provide detailed information. However, Cho et al. [37] pointed out that application-level methods can be inaccurate as compared to flip-flop-level methods for CPU applications. Another option is performing neutron-beam experiments [38], which is not always feasible. We acknowledge the aforementioned pros and cons of various techniques for reliability analysis. 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 [25].

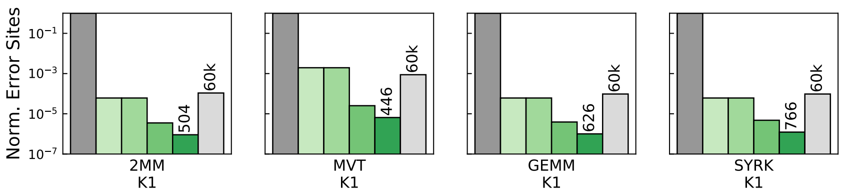
Fault Injection Analysis. Although much work has been done on fault injector models/frameworks [39]–[52] in the CPU domain, there are only a limited number of fault injection models designed specifically for GPUs. For example, to evaluate application error resilience in GPUs, Fang et al. [26] proposed GPU-Qin to understand how faults affect an application's output in GPUs. A GPU debugging tool *cuda-gdb* [53] is leveraged by GPU-Qin to inject single bit errors into the destination operands. Similarly, Hari et al. [18] 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.



(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. The height of the pruned fault sites bar is normalized by the original exhaustive fault sites for each application kernel, see last column of Table I. The effectiveness of progressive fault site pruning is compared against comprehensive baseline injection ($60K$ random experiments). The exact numbers are shown on the top of the last two columns for the proposed method and the baseline case, respectively.

Fault-site Pruning. One of the major concerns of aforementioned fault injection works, both in CPU and GPU domain, is the space complexity of possible fault sites. Within the CPU context, major works by Relyzer [54] and MeRLiN [29] grouped fault sites into equivalence classes and select one or more pilots per class for fault injection. They showed significant benefits of employing their mechanisms in the workloads typically executed on CPUs. We believe directly transferring such pruning techniques to GPU applications is not straightforward because GPU applications typically spawn hundreds to thousands of threads, leading to enormous fault site space. Our work identifies fruitful features that play a role in the final error resilience characteristics of an application and leverage them to carefully prune the fault site space. Finally, to illustrate the effectiveness of our pruning mechanisms, we performed exhaustive experiments on the pruned space and compared the results to the ones closest to the ground truth.

VI. 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 behind this methodology stems from the fact that GPGPU applications spawn a lot of threads, however, 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 and instructions. For additional benefits, we also considered loop iterations within the same thread and register bit positions. We pruned the associated redundant fault sites that are not necessary to capture the GPGPU application error resilience. 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 an accurate GPU reliability assessment.

Acknowledgment

The authors would like to thank the anonymous reviewers. This material is based upon work supported by the National Science Foundation (NSF) grants (#1717532 and #1750667) and a summer grant from the College of William and Mary. This work was performed in part using computing facilities at the College of William and Mary which were provided by contributions from the NSF, the Commonwealth of Virginia Equipment Trust Fund and the Office of Naval Research.

References

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

[2] A. Eklund, P. Dufort, D. Forsberg, and S. M. LaConte, "Medical image processing on the GPU–past, present and future," *Medical image analysis*, vol. 17, 2013.

[3] G. Pratx and L. Xing, "GPU computing in medical physics: A review," *Medical physics*, vol. 38, 2011.

[4] S. S. Stone, J. P. Haldar, S. C. Tsao, W.-m. W. Hwu, Z.-P. Liang, and B. P. Sutton, "Accelerating advanced MRI reconstructions on GPUs," in *Proceedings of the 5th conference on Computing frontiers*. ACM, 2008.

[5] R. Foster, "How to harness big data for improving public health," *Government Health IT*, 2012.

[6] I. Schmerken, "Wall street accelerates options analysis with GPU technology," *Wall Street Technology*, vol. 11, 2009.

[7] NVIDIA. Computational finance. [Online]. Available: http://www.nvidia.com/object/computational_finance.html

[8] NVIDIA. Researchers deploy GPUs to build world's largest artificial neural network. [Online]. Available: https://nvidianews.nvidia.com/news/researchers-deploy-gpus-to-build-world-s-largest-artificial-neural-network

[9] J.-H. Park, M. Tada, D. Kuzum, P. Kapur, H.-Y. Yu, K. C. Saraswat *et al.*, "Low temperature ($\leq 380°$C) and high performance Ge CMOS technology with novel source/drain by metal-induced dopants activation and high-k/metal gate stack for monolithic 3D integration," in *Electron Devices Meeting, 2008. IEDM 2008.*

[10] Q. Gong, P. DeMar, and W. Wu, "Deep Packet/Flow Analysis using GPUs," Tech. Rep., 2017.

[11] The Green500 List - June 2015. [Online]. Available: http://www.green500.org/lists/green201506

[12] Top500 Supercomputer Sites - June 2015. [Online]. Available: http://www.top500.org/lists/2015/06/

[13] H. Takizawa, K. Sato, K. Komatsu, and H. Kobayashi, "CheCUDA: A checkpoint/restart tool for CUDA applications," in *2009 International Conference on Parallel and Distributed Computing, Applications and Technologies, PDCAT 2009, Higashi Hiroshima, Japan, 8-11 December 2009*.

[14] S. Laosooksathit, N. Naksinehaboon, C. Leangsuksan, A. Dhungana, C. Chandler, K. Chanchio, and A. Farbin, "Lightweight checkpoint mechanism and modeling in GPGPU environment," *Computing (HPC Syst)*, vol. 12, 2010.

[15] NVIDIA Fermi Architecture Whitepaper. [Online]. Available: http://www.nvidia.com/content/pdf/fermi_white_papers/nvidia_fermi_compute_architecture_whitepaper.pdf

[16] NVIDIA Kepler GK110 Architecture Whitepaper. [Online]. Available: https://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf

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

[18] 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 *2017 IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2017, Santa Rosa, CA, USA, April 24-25, 2017*.

[19] G. Li, K. Pattabiraman, C. Cher, and P. Bose, "Understanding error propagation in GPGPU applications," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2016, Salt Lake City, UT, USA, November 13-18, 2016*.

[20] R. Leveugle, A. Calvez, P. Maistri, and P. Vanhauwaert, "Statistical fault injection: Quantified error and confidence," in *Design, Automation and Test in Europe, DATE 2009, Nice, France, April 20-24, 2009*.

[21] 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 *Architectural Support for Programming Languages and Operating Systems, ASPLOS '13, Houston, TX, USA - March 16 - 20, 2013*.

[22] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *Proceedings of the 2009 IEEE International Symposium on Workload Characterization, IISWC 2009, October 4-6, 2009, Austin, TX, USA*.

[23] 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*.

[24] A. Bakhoda, G. L. Yuan, W. W. L. Fung, H. Wong, and T. M. Aamodt, "Analyzing CUDA workloads using a detailed GPU simulator," in *IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2009, April 26-28, 2009, Boston, Massachusetts, USA, Proceedings*.

[25] S. Tselonis and D. Gizopoulos, "GUFI: A framework for gpus reliability assessment," in *2016 IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2016, Uppsala, Sweden, April 17-19, 2016*.

[26] B. Fang, K. Pattabiraman, M. Ripeanu, and S. Gurumurthi, "GPU-Qin: A methodology for evaluating the error resilience of GPGPU applications," in *2014 IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2014, Monterey, CA, USA, March 23-25, 2014*.

[27] G. P. Rodrigo Álvarez, P.-O. Östberg, E. Elmroth, K. Antypas, R. Gerber, and L. Ramakrishnan, "Hpc system lifetime story: Workload characterization and evolutionary analyses on nersc systems," in *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*. ACM, 2015.

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

[29] M. Kaliorakis, D. Gizopoulos, R. Canal, and A. González, "MeRLiN: Exploiting dynamic instruction behavior for fast and accurate microarchitecture level reliability assessment," in *Proceedings of the 44th Annual International Symposium on Computer Architecture, ISCA 2017, Toronto, ON, Canada, June 24-28, 2017*.

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

[31] N. Farazmand, R. Ubal, and D. Kaeli, "Statistical fault injection-based AVF analysis of a GPU architecture," *Proceedings of SELSE*, vol. 12, 2012.

[32] H. Jeon, M. Wilkening, V. Sridharan, S. Gurumurthi, and G. Loh, "Architectural vulnerability modeling and analysis of integrated graphics processors," in *Workshop on Silicon Errors in Logic-System Effects (SELSE), Stanford, CA*, 2012.

[33] J. Tan, N. Goswami, T. Li, and X. Fu, "Analyzing soft-error vulnerability on GPGPU microarchitecture," in *Proceedings of the 2011 IEEE International Symposium on Workload Characterization, IISWC 2011, Austin, TX, USA, November 6-8, 2011*.

[34] 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 *Dependable Computing, 2008. PRDC'08. 14th IEEE Pacific Rim International Symposium on*. IEEE, 2008.

[35] K. S. Yim, Z. Kalbarczyk, and R. K. Iyer, "Measurement-based analysis of fault and error sensitivities of dynamic memory," in *Dependable Systems and Networks (DSN), 2010 IEEE/IFIP International Conference on*. IEEE, 2010.

[36] K. S. Yim, C. M. Pham, M. Saleheen, Z. Kalbarczyk, and R. K. Iyer, "Hauberk: Lightweight silent data corruption error detector for GPGPU," in *25th IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2011, Anchorage, Alaska, USA, 16-20 May, 2011*.

[37] H. Cho, S. Mirkhani, C. Cher, J. A. Abraham, and S. Mitra, "Quantitative evaluation of soft error injection techniques for robust system design," in *The 50th Annual Design Automation Conference 2013, DAC '13, Austin, TX, USA, May 29 - June 07, 2013*.

[38] 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 *48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2018, Luxembourg City, Luxembourg, June 25-28, 2018*.

[39] 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 *44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2014, Atlanta, GA, USA, June 23-26, 2014*.

[40] N. El-Sayed and B. Schroeder, "Reading between the lines of failure logs: Understanding how HPC systems fail."

[41] Y. Liang, Y. Zhang, A. Sivasubramaniam, M. Jette, and R. K. Sahoo, "BlueGene/L failure analysis and prediction models," in *2006 International Conference on Dependable Systems and Networks (DSN 2006), 25-28 June 2006, Philadelphia, Pennsylvania, USA*.

[42] Y. Liang, Y. Zhang, A. Sivasubramaniam, R. K. Sahoo, J. E. Moreira, and M. Gupta, "Filtering failure logs for a BlueGene/L prototype," in *2005 International Conference on Dependable Systems and Networks (DSN 2005), 28 June - 1 July 2005, Yokohama, Japan*.

[43] A. J. Oliner and J. Stearley, "What supercomputers say: A study of five system logs," in *37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2007, Edinburgh, UK, June 25-28, 2007*.

[44] A. Pecchia, D. Cotroneo, Z. Kalbarczyk, and R. K. Iyer, "Improving log-based field failure data analysis of multi-node computing systems," in *41st Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2011, Hong Kong, China, June 27-30, 2011*.

[45] R. K. Sahoo, A. Sivasubramaniam, M. S. Squillante, and Y. Zhang, "Failure data analysis of a large-scale heterogeneous server environment," in *2004 International Conference on Dependable Systems and Networks (DSN 2004), 28 June - 1 July 2004, Florence, Italy*.

[46] B. Schroeder and G. A. Gibson, "A large-scale study of failures in high-performance computing systems," *IEEE Trans. Dependable Sec. Comput.*, vol. 7, 2010.

[47] B. Schroeder and G. A. Gibson, "Disk failures in the real world: What does an MTTF of 1, 000, 000 hours mean to you?" in *5th USENIX Conference on File and Storage Technologies, FAST 2007, February 13-16, 2007, San Jose, CA, USA*.

[48] L. N. Bairavasundaram, *Characteristics, impact, and tolerance of partial disk failures*. ProQuest, 2008.

[49] J. Meza, Q. Wu, S. Kumar, and O. Mutlu, "A large-scale study of flash memory failures in the field."

[50] S. Fu and C. Xu, "Quantifying temporal and spatial correlation of failure events for proactive management," in *26th IEEE Symposium on Reliable Distributed Systems (SRDS 2007), Beijing, China, October 10-12, 2007*.

[51] A. Gainaru, F. Cappello, J. Fullop, S. Trausan-Matu, and W. Kramer, "Adaptive event prediction strategy with dynamic time window for large-scale HPC systems," in *Managing Large-scale Systems via the Analysis of System Logs and the Application of Machine Learning Techniques*. ACM, 2011.

[52] A. Gainaru, F. Cappello, M. Snir, and W. Kramer, "Fault prediction under the microscope: a closer look into HPC systems," in *SC Conference on High Performance Computing Networking, Storage and Analysis, SC '12, Salt Lake City, UT, USA - November 11 - 15, 2012*.

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

[54] 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 *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2012, London, UK, March 3-7, 2012*.