Characterizing Accuracy-Aware Resilience of GPGPU Applications

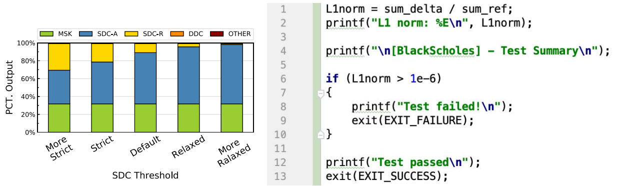
Bin Nie, Adwait Jog, and Evgenia Smirni
William & Mary
Email: {bnie, esmirni}@cs.wm.edu, ajog@wm.edu

Abstract*—Graphics Processing Units (GPUs) have rapidly evolved to enable energy-efficient data-parallel computing. In addition to achieving exascale performance at a stringent power budget, it is imperative for GPUs to provide reliable computing guarantees to the end user. In current commodity systems, such guarantees are often achieved by incurring high protection cost in terms of performance, power, and hardware resources. However, we argue that these strict guarantees are often not required (and that the associated protected overheads can be significantly reduced) because several GPGPU applications are either fault-tolerant or can accept a quantifiable loss in output quality. To this end, this paper characterizes in a hierarchical manner the accuracy-aware resilience of GPGPU applications consisting of thousands of threads. This characterization study shows that accuracy-aware error resilience exhibits several interesting patterns across threads at different hierarchies (i.e., kernel/thread-block/warp). The insights from this characterization study can be used to reduce the overheads of expensive protection or recovery mechanisms that are typically used by GPUs to ensure application reliability.

I. Introduction

Graphics Processing Units (GPUs) have rapidly evolved to enable energy-efficient data-parallel computing for a broad range of areas such as science, engineering, medicine, social media, gaming, and finance [8], [11], [31], [38]. Large-scale data-parallel general purpose GPU (GPGPU) applications from the aforementioned domains are typically long-running, from hours to days. Discarded application outputs due to software or hardware faults could significantly undermine the operational efficiency of large-scale systems [40], making it imperative to provide mechanisms to cope with different types of runtime failures and faults.

In order to provide insights into ways to achieve reliable GPU computing, many previous works have focused on characterizing GPU error resilience [27]–[29]. Typically, these works leverage fault-injection models to evaluate the tolerance of specific applications in the presence of GPU faults. GPU-Qin [9], SASSIFI [14], LLFI-GPU [23] and most recently Nie et al. [30], [46] propose methodologies to systematically inject faults to application threads, record their effects on the application output, and evaluate fault propagation. Although the above works are insightful and useful, their approaches are rather conservative in assuming that a slightest change in the application output, which is referred to as Silent Data Corruption (SDC) [9], [14], [23], is always *unacceptable* to the end user. Consequently, these works conclude that various kinds of protection mechanisms such as frequent check-pointing of necessary application states, re-computation of vulnerable codes, or fault protection mechanisms are necessary to ensure reliable GPU computation [19], [22], [48]. However, these methods often incur high overhead in terms of performance, power, and hardware resources [19], [20], [48], [49].



```
1   L1norm = sum_delta / sum_ref;
2   printf("L1 norm: %E\n", L1norm);
3
4   printf("\n[BlackScholes] - Test Summary\n");
5
6   if (L1norm > 1e-6)
7   {
8       printf("Test failed!\n");
9       exit(EXIT_FAILURE);
10  }
11
12  printf("Test passed\n");
13  exit(EXIT_SUCCESS);
```

(a) (b)

Fig. 1. (a) Effect of a single bit fault on the BlackScholes application output shows that a significant percentage of the fault injection runs lead to silent data corruption (SDC), which can be acceptable to a user (SDC-Accept). The percentage of SDC-Accept increases as the user-defined acceptability threshold becomes less conservative. (b) Code snippet from BlackScholes shows the default evaluation metric (line 1) and threshold value (line 6) to determine the correctness of application output.

To reduce these overheads, we acknowledge that not all faults result in unacceptable loss in application *output quality*. Therefore, if a user is willing to tolerate a *quantifiable* loss in output quality, the overhead to achieve high resilience can be avoided. We call this concept *accuracy-aware resilience*. In order to understand the interplay between quantifiable loss in output quality and the application resilience profile, we show the outcome of over 15K single-bit[1] fault-injection experiments on the output of the BlackScholes application [32] in Figure 1. We assume that a user can measure the acceptability of the output quality with five different thresholds ranging from very strict to very relaxed. We observe that with the default threshold (i.e., the value specified by the CUDA benchmark), a total of 89.1% faults are *benign*: 31.8% are masked (i.e., there is no change in the application output) and 57.3% can be accepted (*SDC-Accept*). Only 10.9% of the outputs are badly corrupted, either beyond user acceptability (*SDC-Reject*) or easily detectable (detectable data corruption – *DDC*), or result in crashes or hangs. If the user accepts more relaxed thresholds for the application output, the application resilience coverage, i.e., the percentage of benign faults (*masked* and *SDC-Accept*), further increases. Motivated by the above observations, we ask the following two questions:

1) How can we systematically analyze the accuracy-aware error resilience of GPGPU applications?

[1] We focus on single-bit faults, which are the most commonly observed faults in GPGPUs [41] and are shown to be sufficient in capturing the reliability characteristics of GPGPU applications [37].

978-1-7281-6095-5/20/$31.00 ©2020 IEEE
DOI 10.1109/CCGrid49817.2020.00-82

2) How can we leverage this analysis to enable low-overhead reliable GPU computing?

One of the major challenges in answering these questions is to come up with a methodical approach that captures the execution flows and resource usage of thousands of concurrently executing threads in GPGPU applications. To this end, we adopt a hierarchical approach, which is based on the inherent GPGPU application hierarchy that arranges threads at three levels:[2] kernels, thread-blocks (or cooperative thread arrays (CTAs) in CUDA terminology), and warps. Note that resource allocation in GPUs happens in the same order [18]. The kernel(s) are first launched on the GPUs, followed by per-core resource allocation across CTAs. The warps inside each CTA are then launched in a lock-step fashion on the single-instruction-multiple-thread (SIMT) execution lanes. We associate this resource allocation procedure with our proposed *accuracy-aware resilience* characterization, which we show to be an effective way to determine *which resources at what levels of the application hierarchy contribute to the user acceptable (SDC-Accept) faults and hence can be protection-free*.

As a use case of the proposed hierarchical analysis of GPGPU resilience, we consider the popular re-computation model [44] as a way to provide protection and assure reliable computing: if the application outputs of the actual computation and re-computation match, then execution can be declared fault-free. Clearly, the overhead of re-computation can be severe. For example, if re-computation is performed in parallel to the actual execution, double hardware resources (e.g., core/memory/register files) would be required. If performed sequentially, the total execution time (including re-computation) doubles. Our hierarchical approach first analyzes the error resilience of representative threads of a kernel to determine if the kernel has the level of resilience that is required by the user. If it is the case, the re-computation of the entire kernel is not required and its associated overheads are saved. On the other hand, if the resilience coverage is not adequate, we perform the error resilience analysis at a finer granularity (i.e., at the CTA-level). If this analysis determines that only a fraction of CTAs do not meet the resilience coverage requirements, we require only the re-computation of such vulnerable CTAs. Consequently, the overall re-computation overhead is reduced because not all CTAs need to be re-computed. We show that our statistically-validated hierarchical approach can provide significant reduction in re-computation overhead while still meet user requirements for application output accuracy and resilience coverage.

To the best of our knowledge, this is the first work to systematically and comprehensively analyze the *accuracy-aware resilience* for a diverse set of GPGPU applications. We study a total of 15 benchmarks (26 kernels) and launch over 330K fault-injection runs (with an average of 10K runs per kernel), leading to the following key contributions:

- We introduce the concept of *accuracy-aware resilience* to GPGPU applications, which provides more opportunities for exploring low-overhead reliable GPU computing.
- We observe that the resilience of a diverse set of GPGPU applications can be classified hierarchically at different levels:
  a) Kernel-level: Accuracy-aware error-resilience can increase significantly if the user is able to tolerate a limited amount of inaccuracy in the application outputs.
  b) CTA-level: Accuracy-aware error-resilience can vary significantly across *groups* of CTAs. Studying a few CTAs per group is enough to represent the overall accuracy-aware error-resilience of GPGPU applications.
  c) Warp-level: Accuracy-aware error-resilience is similar across warps within a *group* of CTAs. Therefore, it is sufficient to perform accuracy-aware error-resilience analysis only at the CTA-level.
- As a case study, we show that the proposed hierarchical approach can reduce protection overheads related to re-computation based on user-defined fault tolerance and resilience coverage. Specifically, we observe that: **a)** The physical resources allocated to the entire kernel for re-computation are saved (and potentially be used for other useful work or be turned-off for power savings) if a user is able to accept a certain resilience coverage and output quality. **b)** Under stricter user-defined requirements, the re-computation overhead can still be reduced by enabling re-computation at a finer granularity (e.g., at CTA/warp level). Overall, the proposed hierarchical approach is able to reduce re-computation overhead while satisfying user-defined output quality and resilience coverage.

II. Background and Methodology

This section provides a brief overview of the baseline GPU architecture and GPGPU application hierarchy, followed by a description of the fault injection methodology and the evaluated applications.

A. GPU Architecture and GPGPU Application Hierarchy

Baseline GPU Architecture. A typical GPU consists of multiple cores, also called streaming-multiprocessors (SMs) in NVIDIA terminology [33]. Each core is associated with private L1 data, texture and constant caches, software-managed scratchpad memory, and register files. 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 use single-error-correction double-error-detection (SECDED) error correction codes to protect register files, L1/L2 caches, shared memory and DRAM from soft errors, and use parity to protect the read-only data cache. Other structures like arithmetic logic units (ALUs), load control units (LCUs), thread schedulers, instruction dispatch units, and interconnect network are not protected [1]–[3].

GPGPU Applications and Execution Model. GPUs rely on the single-instruction-multiple-thread (SIMT) philosophy and

[2] Section II provides a detailed background on the GPGPU application hierarchy.

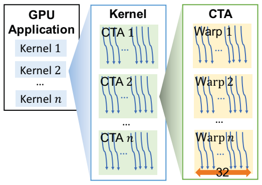


Fig. 2. A simplified overview of GPGPU application hierarchy.

concurrently execute thousands of threads over a large amount of data to achieve high throughput. Figure 2 presents a simplified overview of thread hierarchy in GPGPU applications. A typical GPGPU application launches several kernels. 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 [18], [21]. Such abstraction allows the underlying hardware to relax the execution order of the CTAs to maximize parallelism. The underlying architecture sub-divides each CTA into groups of 32 threads (called *warps*). Each warp is essentially a collection of individual threads that execute a single instruction on the functional units in lock step. This sub-division is an architectural abstraction and is transparent to the application programmer. Like CTAs, warps can also be executed in any order.

We selected benchmarks that cover various workloads from diverse areas, such as image processing, finance, linear algebra, physics, molecular dynamics, and data mining. In total, we evaluated 15 GPGPU applications from widely-used benchmark suites, including CUDA [32], SHOC [7], Rodinia [6], and Mars [15], see Table I. As kernels of GPGPU applications normally implement independent modules, we perform resilience coverage analysis separately for each kernel. For benchmarks with more than one kernel, we randomly select at most four kernels for fault injection experiments. In the rest of this paper, if the kernel index is not specified, it implies that the benchmark only contains one kernel.

B. Fault Injection Methodology

Framework. We employed a fault injection methodology using GPGPU-Sim [5]. Consistent with prior works [9], [14], [43], we inject a single fault per application run and record the detailed execution information provided by GPGPU-Sim, including the fault site, the original and impacted values in the register, and the final output. For all chosen benchmarks (see Table I), we launch 330K fault-injected runs, with an average of 10K runs per benchmark kernel. In Section IV-D we show that 10K runs are beyond sufficient to obtain results of statistical significance with 95% confidence intervals.

Fault Injection Outcome Classification and Analysis. For each fault injection experiment, we examine the application output to understand the effect of an injected fault. There are several possible outcomes. If faults do not lead to any difference between the fault-injected and the fault-free outputs, they are classified as *Masked* faults. In some cases, although faults allow the application to execute completely, the application output is incorrect. Such faults are typically classified as *Silent Data Corruption* (SDC) faults. In certain circumstances, users *may* accept the application output. Therefore, we further classify SDC faults into *SDC-Accept* and *SDC-Reject* according to the user acceptability threshold. Furthermore, some corrupted results can be easily detected (e.g., irregular negative value, infinite or NaN value), we classify them as *Detectable Data Corruption* (DDC). Lastly, faults can also result in crashes or hangs. In summary, we classify fault-injected outputs into five categories: (1) *Masked*, (2) *SDC-Accept*, (3) *SDC-Reject*, (4) *DDC*, and (5) *OTHER*. The first two faults are *benign* and define the resilience coverage of the application, while the rest are *non-benign*.

TABLE I
LIST OF APPLICATIONS WITH EVALUATION METRICS AND THRESHOLDS.

Suite	Benchmark	Evaluation Metric	Default Threshold
CUDA	BlackScholes (BS)	L1 norm*	$1e-6$
	Ray Tracing (RAY)	MSE of images	0.1
	Convolution Separable (CONS)	Relative L2 norm*	$1e-6$
	Fast Walsh Transform (FWT)	L2 norm*	$1e-6$
	JPEG	MSE of images	0.1
	Kmeans (KMN)	MSE of centroid coordinates	0.01
	Laplace3D (LPS)	RMS error*	$1e-6$
	Neural Network (NN)	Difference in prediction accuracy	1%
	Scalar Product (SCP)	L1 norm*	$1e-6$
	Scan Large Array (SLA)	PercLoss	1%
Mars	WordCount (WC)	Difference in word count	1
Rodinia	HotSpot (HS)	MSE of output temperature list	0.01
SHOC	Breadth-First Search (BFS)	PercLoss	1%
	Molecular Dynamics (MD)	PercLoss	1%
	Sort	Ranked Biased Overlap	0.01

* indicates metrics and thresholds as provided by benchmarks. Alternatively, the threshold values can be provided by the user or administrator.

Outcome Analysis. In order to determine whether a certain SDC output is acceptable or not, we need a metric and threshold value to quantitatively measure the difference between the outputs of fault-injected and fault-free runs. Choosing the most appropriate metric and threshold requires domain knowledge. *We anticipate that the evaluation metrics/thresholds are provided by the user.* In addition, we also provide several choices of commonly used metrics and threshold default values. For most applications, we choose widely-used metrics such as mean squared error (MSE)[3] and percentage loss (PercLoss)[4]. For certain applications, we use domain-specified metrics. RAY and JPEG, which are image processing applications, are evaluated by the MSE of images, pixel by pixel. For Neural Network (NN), provided that the prediction accuracy for the

[3] $MSE = \frac{1}{n}\sum_{i=1}^{n}(X_i - Y_i)^2$, where X and Y are two vectors of size n.

[4] $PercLoss = \frac{\#\ miss\_match}{\#\ total} \times 100\%$, where $\#\ miss\_match$ is the number of different values in the fault-free and fault-injected outputs, and $\#\ total$ is the total count of values.

fault-free run is 100%, we use the difference to this value as the evaluation metric. For Sort, the result of which is a ranked list, we use the commonly used Ranked Biased Overlap (RBO) [45] to quantify the difference between fault-free and fault-injected outputs. Table I shows the evaluation metrics and their respective default thresholds. Besides the default threshold, we also evaluate application output with two more strict and two more relaxed threshold values, yielding to a total of five levels of SDC threshold values (as shown in Figure 1).

C. Baseline Fault Model

We used the popular single-bit fault injection model [9], [14], [30] to evaluate the effect of soft errors in GPUs. These faults affect the functional units such as arithmetic-logic units (ALUs) and the load-store units (LSUs), which are *not* ECC protected in commercial GPUs. A fault site contains three aspects of data: (1) tid indicates the candidate thread, (2) $inst\_id$ and sim_cycle identify the instruction and its simulation cycle (sim_cycle is necessary because the same instruction can be executed many times if inside a loop), (3) bit_pos tells which bit location is affected. To mimic a single-bit fault in a functional unit, we inject faults to data values of the destination registers, which is standard practice in this line of work [9], [14], [30].

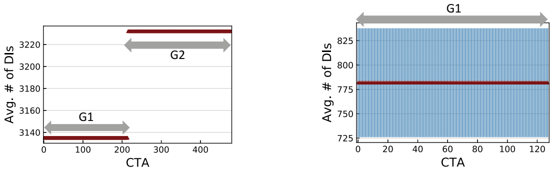
The goal is to determine representative *fault sites*. We first start with selecting representative threads (details in the next Section). Each representative thread has a tid. Next, we profile such a representative thread with the GPGPU-Sim simulator to collect instruction-related execution details, including the instruction type, its simulation cycle, and the destination register type. There can be tens to thousands of dynamic instructions in one thread. In order to control the number of fault sites, we randomly sample a few iterations for instructions inside *loop* blocks. We also select all instructions outside *loop* blocks to make sure we cover all types of instructions. Finally, for every selected instruction, we flip one bit in its destination register. The bit_pos is chosen from a set of pre-selected bit position candidates that are evenly spread in the register. Those bit positions are selected to cover a range of positions in registers, as it is impractical to conduct experiments on every single bit.

III. A Hierarchical Approach to Thread Classification

GPGPU applications can contain a massive number of threads. Therefore, it is unrealistic to perform fault-injection runs on every single thread. Consequently, we have to identify a fraction of representative threads, which is a challenging open problem. We realize this with a hierarchical classification and thread selection method.

A. Multi-level Classification and Thread Selection

We propose and evaluate a hierarchical approach for thread grouping. Following the hierarchy of GPGPU applications, we classify threads at the CTA and warp levels. We group CTAs (or warps) based on the distribution of thread dynamic instruction (DI) counts[5], which has been shown to be an effective proxy for accurately capturing the error resilience of threads [9], [30]. The rationale is that threads with the same DI count are likely to execute the exact same set of instructions, thus resulting in similar error resilience behavior. Such rationale is also confirmed with millions of fault injection experiments [30]. We consider the mean and one standard deviation of DI counts to quantitatively compare different CTAs (or warps). Then, from each CTA (or warp) group, we randomly select a limited number of threads for fault injection.



(a) BlackScholes (b) SCP

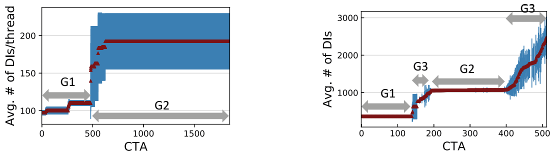
Fig. 3. Distribution of thread dynamic instruction (DI) counts at the CTA level for regular benchmarks (a) BlackScholes and (b) SCP. **The red triangle indicates the average and the blue error bar indicates one std.**

1) CTA-level classification: **Regular CTA Analysis.** First, we illustrate the distribution of DIs at the CTA level for different benchmarks. Figure 3 focuses on two regular benchmarks: BlackScholes and SCP. The x-axis indicates the index of CTAs, while the red triangle and the blue error bar correspond to mean and one standard deviation of DI counts, respectively. CTAs are sorted by the average DI counts in the ascending order along the x-axis. Since we classify CTAs with similar DI distribution as one group, there exist two distinct CTA groups for BlackScholes (see Figure 3(a)). In addition, each group only contains one type of thread (i.e., the standard deviation of DI counts is 0). For such regular benchmarks, we only group at the CTA level. Figure 3(b) reports on SCP, another regular benchmark. All CTAs share the same mean and standard deviation of DI counts while their standard deviation is higher than that in Figure 3(a). For kernel-level and CTA-level analysis, we classify all CTAs in SCP as a single group. Because of the high variance in DI counts in certain CTAs, we also perform warp-level analysis (see Section III-A2).

Irregular CTA Analysis. Figure 4 illustrates two irregular benchmarks: HotSpot and RAY, which exhibit divergence in the DI distributions due to branch instructions. We classify CTAs in HotSpot into two groups (see Figure 3(a)): group *G1* (regular group) contains CTAs with low standard deviation of DI counts while group *G2* (irregular group) contains CTAs with diverse threads. Likewise, in RAY (see Figure 4(b)), CTAs with no variance in DI counts are grouped into regular groups *G1* and *G2*, while all other CTAs are classified into the irregular group *G3*.

Effect of Input. We explore the question: does the CTA grouping method change with application input? If not, this implies that it is possible to profile the kernel once and the resulted grouping is applicable to other inputs. To explore this, we feed HotSpot and RAY with three inputs: Small, Medium, and Large. Table II shows the effect of various

[5] Detailed kernel/CTA/warp/thread information can be obtained through the GPGPU-Sim simulator [5].



(a) HotSpot (b) RAY

Fig. 4. Distribution of thread dynamic instruction (DI) counts at the CTA level for irregular benchmarks (a) HotSpot and (b) RAY. **The red triangle indicates the average and the blue error bar indicates one std.**

TABLE II
THE IMPACT OF DIFFERENT INPUTS ON CTA GROUP POPULARITY FOR HOTSPOT AND RAY. **NOTATION: GRP-S/M/L=THE PERCENTAGE OF CTAS IN THAT GROUP WITH SMALL/MEDIUM/LARGE INPUT, R=REGULAR, IR=IRREGULAR.**

Benchmark	Grp.	Type	GRP-S	GRP-M	GRP-L
HotSpot	*G1*	R	31%	26%	26%
	G2	IR	69%	74%	74%
RAY	*G1*	R	25%	25%	27%
	G2	R	9%	22%	28%
	G3	IR	66%	53%	45%

* R: regular group; IR: irregular group.

inputs on group "popularity" (i.e., the percentage of CTAs in that group). We observe that for both benchmarks, the number of CTA groups, as well as their types (regular or irregular), is the same in all three inputs, implying that the CTA grouping strategy is input-independent. Additionally, we notice that group popularity changes with different inputs. For example, for HotSpot, the popularity of *G1* starts from 31% for *Small* input, then decreases but stabilizes at 26% as the input size increases. For RAY, the popularity of *G1* is quite stable but that of *G2* increases significantly from 9% to 28% with larger input sizes. We also explore the impact of input size on CTA grouping strategy for other benchmarks and this observation persists. For brevity, we do not show those results. In sum, the number of groups persists across different inputs while their popularity may change.

2) Warp-level classification: We now focus on the warp level to explore whether heterogeneity in terms of dynamic instruction counts exists. Figure 5 shows the mean and one standard deviation of DI counts at the warp level for SCP, which is different from the CTA-level (compare to Figure 3(b)). At the warp level, we are able to classify warps into four groups: regular groups *G1*, *G2*, and *G3* with no variance in their DI counts, and the irregular group *G4*.

We also investigate whether warp-level grouping is input-independent and observe that this holds for all benchmark kernels except MD. Figure 6(a) and (b) show the warp-level plots for MD k1 using Small and Large inputs, respectively. For Small input, we classify all warps into one irregular group while for Large input, we classify warps into two groups: the regular *G1* and the irregular *G2*. However, if we further explore the warp-level DI counts in MD k1 with Large input, we find that all warps look very similar. In fact, almost all (i.e, $\geq 94\%$) threads in most (i.e., $\geq 98\%$) warps in *G2* share the same DI count as threads in *G1*. That is, only 1 or 2 threads out of all 32 threads per warp are different. Therefore, all warps in MD k1 with Large input can also be classified as one group, just as in the classification for MD k1 with Small input. Consequently, even though different inputs may change standard deviation of the DI count, it does not truly impact the resulting warp grouping strategy. In other words, we can still apply the same warp-level grouping method derived from one input to others.

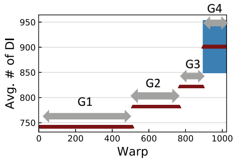
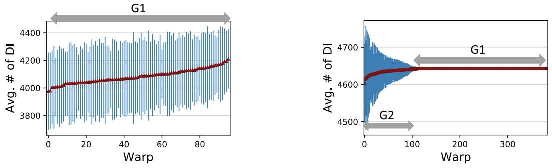


Fig. 5. Distribution of thread dynamic instruction counts at the warp level for SCP. **The red triangle indicates the average and the blue error bar indicates one std.**



(a) Small input (b) Large input

Fig. 6. Distribution of thread dynamic instruction counts at the warp level for MD k1, using two inputs: (a) Small and (b) Large. **The red triangle indicates the average and the blue error bar indicates one std.**

TABLE III
CTA-LEVEL AND WARP-LEVEL CLASSIFICATION FOR BENCHMARK KERNELS. **NOTATION: *%R-Grp.*= % REGULAR GROUPS OVER ALL GROUPS, *# DI Grp.*=# OF GROUPS CLASSIFIED BY DYNAMIC INSTRUCTION COUNTS, *# ErrDist Grp.*=# OF GROUPS REFINED BY FAULT DISTRIBUTION.**

Benchmark	# CTA	# Warp	Grp. Level	% R-Grp	# DI Grp.	# ErrDist Grp.
BlackScholes	480	1920	CTA	100%	2	1
RAY	512	2048	CTA	55%	3	3
CONS k6	1152	4608	CTA	100%	3	1
CONS k7	2304	4608	CTA	100%	2	1
FWT k6	1024	16384	CTA	100%	1	1
FWT k13	128	1024	CTA	100%	1	1
JPEG	512	1024	CTA	100%	1	1
KMN k1	121	968	CTA	100%	2	2
KMN k2	121	968	CTA	100%	2	2
LPS	128	512	CTA	0%	3	1
NN k4	1000	1000	CTA	100%	2	2
HotSpot	1849	14792	CTA	26%	2	2
BFS k3	20	320	CTA	90%	3	3
BFS k9	20	320	CTA	90%	3	3
BFS k11	20	320	CTA	85%	3	3
WC k114	512	2048	CTA	94%	2	2
WC k5	1	8	Warp	75%	2	2
WC k91	32	256	Warp	85%	5	2
SCP	128	1024	Warp	87.5%	4	1
SLA k256	8	64	Warp	87.5%	4	1
SLA k258	8	64	Warp	0%	1	1
MD k1	48	384	Warp	100%	1	1
MD k3	48	384	Warp	100%	1	1
Sort k8	512	4096	Warp	88%	2	1
Sort k20	512	4096	Warp	87.5%	2	1
Sort k24	512	4096	Warp	87.5%	2	2

3) Classification result and thread selection: We apply the CTA-level and warp-level grouping method described above to every benchmark kernel. Table III shows the classification results. Column *Grp. Level* indicates the classification level, i.e., CTA or warp. Recall that we only consider the warp level for SCP-like benchmark kernels. Column *# DI Grp.* shows the number of groups classified by the distribution of DIs in CTA or warp, while Column *% R-Grp* points out the percentage of regular groups (i.e., groups with low to no variance in DI counts). Naturally, due to the simplicity in thread selection, regular groups are preferable. Fortunately, we observe a significant percentage of regular groups in most benchmark kernels, varying from 26% to 100% with an average of 82%. We also explore the group-wise error resilience and further combine groups that share similar resilience characteristics. Column *# ErrDist Grp* indicates such refined group counts. More details are given in Section IV.

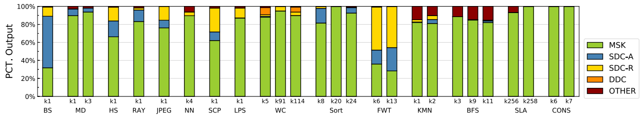


Fig. 7. Distribution of fault injection outcomes at benchmark kernel level. (SDC faults are evaluated with the default threshold values.)

Having determined the grouping level and strategy, the next step is to select a limited number of threads per group for fault injection runs. For regular groups, where all threads share similar dynamic instruction counts, it is straightforward to randomly select one thread per group. For irregular groups, which contain a variety of different threads, we randomly select a limited number of threads based on the frequency of their DI counts.

Observation#1: Only a few groups of CTAs are different in terms of the number of dynamic instructions they execute.
Observation#2: Only a few warps within the selected groups of CTAs are different in terms of the number of dynamic instructions they execute.
Observation#3: Hierarchical grouping is not sensitive to the type or size of the input.

IV. Hierarchical Approach to Error Resilience Characterization

Having demonstrated the rationale and methodology for the hierarchical thread classification and selection method, in this section we characterize and analyze benchmark resilience.

A. Application Kernel Level Characteristics

We evaluate t resilience by computing the distribution of fault injection outcomes, which is the percentage of each type of fault (i.e., *Masked, SDC-Accept, SDC-Reject, DDC, and Others*) among all fault-injection runs. We launch over 330K fault-injection runs, with an average of 10K runs per kernel. In Section IV-D, we validate statistically that 10K runs are sufficient to obtain the error resilience profile of GPGPU applications.

1) Scope of accuracy-aware resilience: Figure 7 presents the distribution of fault injection outcomes evaluated with the default SDC threshold of every benchmark kernel listed in Table I. Every stacked bar represents the fault distribution of one benchmark kernel. The first impression is that for all benchmarks, the majority of soft-errors are masked, i.e., they are imperceptible to the end user. The actual percentage numbers of *Masked* faults vary from 31.8% in BlackScholes to 100% in Sort k20, SLA k258, and CONS k7. For CONS k6, there are very few non-masked faults (i.e., $\leq 1\%$), which are barely visible in Figure 7. We check the number of loop iterations in those benchmarks with close to 100% *Masked* outputs and the low number (≤ 6) confirms that the results are not biased due to sampling. Such a large portion of *Masked* faults implies that the protection effort for these runs is perhaps not necessary.

Secondly, we notice that the majority of the benchmark kernels present a non-negligible percentage of *SDC* faults. In previous works, these faults are deemed unacceptable but users who embrace approximate computing may be willing to trade corrupted output with lower resilience overhead and better performance, as long as the "degree of corruption" is within expected ranges. For this reason, we further divide the *SDC* results into *SDC-Accept* and *SDC-Reject*. We observe that benchmark kernels with a large portion of *SDC* faults also exhibit a significant percentage of *SDC-Accept* faults. Note that the fault distribution in Figure 7 is evaluated with the application default threshold values. The percentage of *SDC-Accept* is expected to increase when the benchmark is evaluated with relaxed threshold values (see also Figure 1). The percentage of *SDC-Accept* faults can be very high in some benchmarks, such as 12.8% in RAY, 17.7% in HotSpot, 15.5% in FWT k6, 25.9% in FWT k13, and even 57.3% in BlackScholes. In some other benchmarks (i.e., LPS, SLA, and CONS), there are little to no *SDC-Accept* faults. Note also that these benchmarks have a significant percentage ($\geq 89\%$) of *Masked* faults.

From the domain perspective, image processing applications such as RAY and JPEG are resilient to soft-errors, as minor changes in output images are barely distinguishable by the end users. NN also digests single bit flips well. Those soft errors slightly impact the weights of trained neural networks, thus barely result in wrong outputs. In contrast, SCP and FWT are more sensitive to soft errors, see the percentage of *benign* faults (*Masked* and *SDC-Accept*) in Figure 7.

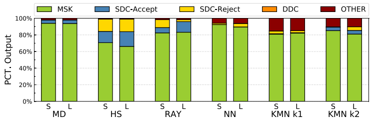


Fig. 8. Impact of Small and Large inputs on fault distribution.

2) Sensitivity to input size: To examine the impact of different inputs on the resilience profile of an application, we apply two choices of inputs, i.e., Small vs. Large, on

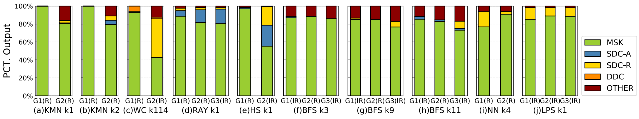


Fig. 9. Error resilience characteristics at CTA level. **Each bar is distinguished by its group name and whether it is regular (R) or irregular (IR).**

five of the benchmark kernels, see Figure 8. For NN and KMN k1, whose scope of *SDC-Accept* is negligible (i.e., $\leq$ 1%), we observe high similarity in the fault distribution when using different input sizes. For other benchmark kernels, using a large input leads to a decrease in the percentage of *Masked* faults, specifically to 4.1% for KMN k2, 0.3% for MD, and 4.5% for HotSpot. Fortunately, for these kernels, the percentage of *SDC-Accept* increases correspondingly with large input, resulting in a similar scope of *benign* faults comparing to the small case. KMN k2 is the only exception, where the increase of *SDC-Accept* is less than the increase of *SDC-Reject*. Moreover, for RAY, whose scope of *Masked* faults is not impacted by the input size, the large input makes the kernel more error resilient by having a larger percentage (i.e., 6.6% more) of *SDC-Accept* faults.

Observation#4: There is an ample scope of *SDC-Accept* faults in some GPGPU applications.
Observation#5: Using large input typically preserves or increases the scope of resilience coverage, i.e., *benign* outputs.

B. CTA Level Characteristics

Consistent with the hierarchical classification at the CTA level (see Section III), we perform fault injection runs for every CTA group, in order to explore whether fault distributions vary across different CTA groups.

In Table III, we show the number of CTA groups based on the distribution of dynamic instructions in CTAs of the benchmark kernel (see Column "*# DI Grp*"). We further combine groups that share similar fault distribution and the final number of groups is shown in Column "*# ErrDist Grp.*". Clearly, $\# \; ErrDist \; Grp. \leq \# \; DI \; Grp$. We observe that for BlackScholes, CONS, and LPS, all *DI* groups are combined into one *ErrDist* group due to the similarity in the group fault distribution. For BlackScholes, though there are two *DI* groups (both regular), their average numbers of dynamic instructions are very close (3135 and 3232), yielding to similar fault resilience characteristics. The same applies to CONS. For LPS, all its three *DI* groups are irregular, and although they have different average number of dynamic instructions, the major composite threads have the same dynamic instruction counts, resulting in similar resilience characteristics.

Except for the three aforementioned benchmarks, the rest of benchmark kernels share different fault distribution at the CTA level. Figure 9 shows the stacked bar plots for the fault distribution of every *ErrDist* group for 10 benchmark kernels (others are not shown due to limited space). We observe that the fault distribution can be significantly different among *ErrDist* groups. First, the composition of fault distribution can be different. In KMN k2, almost all soft errors are masked in *G1* while there is large portion of *SDC-Accept*, *SDC-Reject*, and *Other* faults (i.e., 4.4%, 4.4%, and 10.2%, respectively) in *G2*. Such observation also exists for KMN k1 and WC k114. Second, for some other kernels, certain *ErrDist* groups can have more percentage of *SDC-Accept* faults, including RAY *G2* and *G3* (14.1% and 15.9%, respectively), HotSpot *G2* (23.4%), and BFS k11 *G1* (3.0%). Furthermore, NN k4 *G1* and BFS k11 *G3* present a notable larger percentage of *SDC-Reject* faults (16.9% and 8.2%, respectively), which have the potential to be converted to acceptable output with relaxed threshold values. In contrast, the difference between the fault distribution of *ErrDist* groups in some benchmarks (e.g., BFS k3) can be small, but the percentage of *benign* faults in their *ErrDist* groups is high ($\geq$ 85%).

In general, we observe that regular groups tend to have a larger portion of *benign* faults than irregular ones. Furthermore, if we only focus on *Masked* faults, which are by definition always *benign*, the regular groups always have a large portion as compared to irregular ones. For example, the percentage of *Masked* faults in WordCount k114 is 93.1% and 42.6% in regular *G1* and irregular *G2*, respectively. In HotSpot, the percentages are 96.9% in the regular group and 55.4% in the irregular one.

Observation#6: A significant percentage of CTA groups are more resilient (i.e., have high percentage of *SDC-Accept* outputs) than other groups.

C. Warp Level Characteristics

Previously, we show that CTA groups have distinct fault distribution, especially when comparing regular with irregular groups. Here, we are interested in whether such heterogeneity persists in the warp level. We use the warp-level grouping strategy (described in Section III) to classify warps within the same CTA group. Figure 10 shows the fault distribution of every warp-level *DI* group for SCP, Sort k24, WordCount k5, and WordCount k91. For SCP (see Figure 10(a)), all four DI groups share similar percentage of faults, while only G1 contains slightly more SDC-Accept outputs than the others. Such similarity in the fault distribution among warp groups is also observed for Sort k24 in Figure 10(b). Consequently, for both kernels, there is only one *ErrDist* group.

For WordCount k5 and k91 (see Figure 10 (c) and (d), we observe significant difference in error-resilience among warp groups. Some groups (i.e., G1 for WordCount k5 and G1 and G2 for WordCount k91) are more resilient to injected faults than others. Recall that results in Figure 10 are evaluated with default threshold values. By varying levels of thresholds (results are not presented here due to lack of space), we observe that warp groups exhibit different sensitivity to the threshold values. For SCP, all the warp groups increase the percentage of SDC-Accept almost at the same amount as we relax the SDC threshold values. In contrast, for WordCount k91, G5 is more sensitive to relaxed threshold values (i.e., we observe an

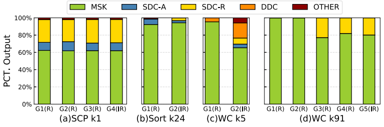
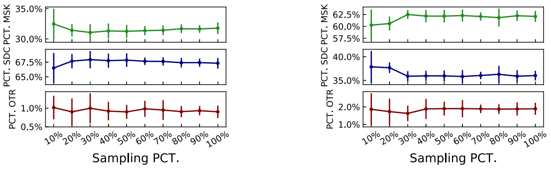


Fig. 10. Error resilience characteristics at warp level. **Each bar is distinguished by its group name and whether it is regular(R) or irregular(IR).**



(a) BlackScholes (b) SCP

Fig. 11. Changes in the percentage of faults with increasing sample size for (a) BlackScholes and (b) SCP. *PCT.MSK*, *PCT.SDC*, and *PCT. OTR* indicate the percentage of masked, SDC, and other (including DDC, crashed, and hangs) faults, respectively. **Error bars give the 95% confidence intervals.**

increase in the percentage of SDC-Accept) comparing to other warp groups.

Observation#7: Similar to CTA-level analysis, some warps are more resilient than others.

D. Statistical Validation

We have shown the resilience characteristics for benchmarks at the kernel, CTA, and warp levels. The vast parallelization of the GPGPU applications makes the generation of all possible fault sites not possible. To evaluate the statistical significance of our result, for every benchmark kernel, we randomly sample 10% of the entire space of generated fault sites (10K for each experiment) and gradually add 10% until we reach 100% (i.e., all generated sites). For every increment, we calculate the 95% confidence interval. Figure 11 reveals resilience changes over increasing the sample sizes. It is clear that the fault percentage fluctuates significantly in the initial increments, indicating that the sample space in insufficient to reach results of statistical significance, but becomes steady after the sampling percentage exceeds 80%. Moreover, we see significant overlaps across the confidence intervals, which suggests that our experiments do capture the "unknown" means of the fault distributions. In fact, we observe that the average error margin is 1.27%, 0.75%, and 0.75% for the percentage of *Masked*, *SDC*, and other faults (including *DDC*, crashes, and hangs), respectively (see the ranges of the y-axes of the graphs in Figure 11). The above results are consistent with the fact that 1K experiments are enough to obtain 95% confidence intervals with 6% error margins and 60K experiments are necessary for 99.8% confidence intervals with 1.26% error margins, see [30] for an overview.

V. Use Case: Reducing Protection Overhead

In this section, we leverage on the various observations of our characterization study to improve on application resilience while maintaining reduced overhead. We first discuss the trade-off among the following metrics.

1) **Resilience Coverage (RC) and Output Quality (OQ)**: The *perfect output quality* refers to only accepting *Masked* outputs. However, as shown in Section IV, there exists a large scope of *SDC-Accept* outputs of GPGPU applications. These tolerable outputs provide the opportunity of improving the Resilience Coverage (RC), which is defined as the percentage of runs with *benign* faults (i.e., *Masked* and *SDC-Accept*). Acceptable resilience coverage is application and user dependent [44].
2) **Overhead Reduction (OR)**: To improve GPGPU application resilience, we consider a re-computation model that computes the kernel again and compares its output with the actual execution output for any anomalies. As a baseline, we assume all CTAs of the kernel are vulnerable (i.e., do not meet the Resilience Coverage requirement) and hence need to be re-computed at the expense of additional physical resources. In the worst case, these resources are twice of the total resource required for the actual computation.

We focus on how our accuracy-aware resilience characterization can help in reducing the physical resource requirements. For example, if our characterization shows that 50% of CTAs are not vulnerable, then only 50% additional physical resources are required for re-computation. In the remaining section, we consider two different output quality (OQ) thresholds:

1) *Perfect OQ*: includes *Masked* outputs only.
2) *Default OQ*: includes *Masked* outputs and *SDC-Accept* outputs (evaluated with default thresholds, see Table I).

Table IV shows the trade-off between resilience coverage and re-computation overhead for different benchmark kernels. Under "Kernel-Level", the "Perfect OQ (OR)" column provides resilience coverage and (protection overhead reduction) that considers Perfect OQ while the "Default OQ (OR)" column provides resilience coverage and (protection overhead reduction) that considers Default OQ. For some benchmark kernels, we can further gain on overhead reduction by considering thread groups at a finer granularity (see column "Default OQ (OR)" under "CTA-/Warp-Level".)

Coarse-grain Protection Overhead Analysis. We first show analysis at the kernel level (see the two columns under "Kernel-Level" in Table IV). We observe that resilience coverage increases as we start to relax the output quality requirement, which results in increasing overhead reduction (OR). For BlackScholes, for example, the resilience coverage is very low (31.8%) when users desire perfect output quality. With such low resilience coverage, it is necessary to protect the entire kernel (i.e., 0% overhead reduction). However, if users are able to accept some inaccuracy in output (i.e., accept the default output quality), the resilience coverage increases to 89.1%. If this is agreeable with the user, then the re-computation cost of the entire kernel can be avoided (i.e., 100% overhead reduction).

In the remaining discussion, we assume a 85% resilience coverage requirement set by the user, as many kernels satisfy it at the default output quality threshold. For example, we

TABLE IV
RESILIENCE COVERAGE VS. OVERHEAD REDUCTION.

	Kernel-Level		CTA-/Warp-Level
Benchmark	**Perfect OQ (OR)**	**Default OQ (OR)**	**Default OQ (OR)**
BlackScholes	**31.8% (0%)**	89.0% (100%)	–
RAY	**83.2% (0%)**	96.0% (100%)	–
Sort k8	**81.5% (0%)**	97.8% (100%)	–
JPEG	**76.1% (0%)**	**84.6% (0%)**	–
SCP	**62.1% (0%)**	**71.6% (0%)**	–
FWT k6	**36.0% (0%)**	**51.5% (0%)**	–
FWT k13	**28.3% (0%)**	**54.2% (0%)**	–
HotSpot	**66.2% (0%)**	**83.8% (0%)**	99.6% (26%)
NN k4	89.6% (100%)	89.6% (100%)	91.9% (92%)
WC k5	87.9% (100%)	88.9% (100%)	96.6% (75%)
WC k91	94.9% (100%)	94.9% (100%)	100% (75%)
WC k114	89.8% (100%)	89.8% (100%)	93.4% (94%)
BFS k3	88.5% (100%)	88.5% (100%)	100% (100%)
BFS k9	**84.9% (0%)**	**84.9% (0%)**	86.1% (95%)
BFS k11	**82.1% (0%)**	**83.9% (0%)**	99.2% (90%)
KMN k1	**82.2% (0%)**	**82.6% (0%)**	100% (7%)
KMN k2	**81.0% (0%)**	85.4% (100%)	100% (7%)

* The resilience coverage requirement is set to be 85%.
* Kernels with no values in the fourth column only contain one fault distribution group, thus are not applicable for fine-grain analysis.

find that for kernels such as RAY, Sort k8, and KMN k2, the resilience coverage requirement of 85% is met and hence its re-computation can be completely avoided leading to 100% reduction in protection overhead by accepting Default OQ instead the Perfect OQ. However, we also find that some other kernels (see cells in bold in Table IV) do not meet the 85% resilience coverage requirement even at the Default OQ. For such kernels, we have to resort to fine-grain analysis to seek opportunities of overhead reduction.

Fine-grain Protection Overhead Analysis. If the kernels do not meet the resilience coverage requirement, the protection overhead can still be reduced by exploiting the fact that some CTAs or warp groups are significantly more error-resilient than others (see Observations #6 and #7). We propose not to re-compute such groups and hence reduce the associated protection overhead. As CTAs are independent of each other, output of only those CTAs will be required to be compared that have lower resilience coverage. After applying our resilience characterization (Section IV), we find that the resilience coverage has increased significantly for most kernels (see "Default OQ (OR)" column under "CTA-/Warp-Level") and still with a significant overhead reduction. For example, for HotSpot, at the CTA level, users can obtain 99.6% resilience coverage while still reducing overhead by 26% (i.e., G1 in Figure 9(e) can be protection-free). In addition, for kernels with over 85% resilience coverage (i.e., NN k4, WC k5, WC k91, WC k114, and BFS k3), it is still possible to further improve their resilience coverage at a finer granularity (see fourth column). Although the above analysis is for the 85% resilience coverage requirement, similar analysis can be performed for any other threshold.

Observation#8: Hierarchical error-resilience analysis offers flexibility for resilience coverage and overhead reduction.

VI. RELATED WORK

There is a large number of studies that focus on leveraging simulation-based analysis to detect critical hardware structures that are more vulnerable to soft errors. Prior works [10], [17] have conducted architectural vulnerability factor (AVF) analysis, which tracks every bit during the application run and calculates the likelihood of the output being affected. Although there is a large body of work on fault injection models/frameworks [4], [12], [13], [34]–[36], [39] in the context of CPUs, only a limited set of fault injector models have been proposed for GPUs [9], [14], [23], [30], [46], [47]. Yim et al. [47] build a source-to-source translator, SWIFI, to investigate error resilience in GPUs and demonstrate that the ratio of silent data corruption (SDC) in GPUs is much higher than that observed in CPUs.

Prior works [9], [30], [46] use the number of dynamic instructions (DI) per thread as a proxy for thread behavior. The rationale is that threads with the same dynamic instruction count are likely to execute the exact same set of instructions, thus resulting in similar error resilience behavior. In this paper, we typically put threads with the different DI count in the same group based on additional hierarchical information (Section IV-B). This allows higher overhead reduction compared to GPU-Qin because vulnerable threads are encapsulated into fewer CTAs, which then can be recomputed.

While the purpose of fault protection is to completely avoid faults, approximate computing explores the trade-off between accuracy, performance, and energy efficiency. Prior studies have considered this trade-off in specific areas including bioinformatics [16], [25], performance analysis [42], data mining [26], and image recognition [24]. Approxilyzer [44] has been proposed to evaluate the three-way trade-off among output quality, resilience coverage, and overhead reduction. It is built for single-threaded CPU applications and is not clear how it can be extended for GPGPU applications with thousands of threads and billions of fault sites.

VII. CONCLUSIONS

In this paper, we characterize the of *accuracy-aware resilience* of GPGPU applications. We propose a hierarchical thread classification and selection approach to understand the application resilience coverage. Through a large number of fault injection runs ($330,000$ in total) on a variety of GPGPU applications, we obtain several interesting observations. First, the error resilience of GPGPU application kernels can significantly increase by embracing some loss in output quality. Second, the accuracy-aware error-resilience of a kernel can be captured by analyzing threads of only a few thread-blocks. Third, the proposed hierarchical approach facilitates the overhead reduction of protection/recovery mechanisms to ensure reliable output.

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).

References

[1] GP100 Pascal Whitepaper.

[2] NVIDIA Fermi Architecture Whitepaper.

[3] NVIDIA Kepler GK110 Architecture Whitepaper.

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

[5] A. Bakhoda, G. L. Yuan, W. W. Fung, H. Wong, and T. M. Aamodt. Analyzing CUDA workloads using a detailed GPU simulator. In *Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on*, pages 163–174. IEEE, 2009.

[6] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing, 2009.

[7] A. Danalis, G. Marin, C. McCurdy, J. S. Meredith, P. C. Roth, K. Spafford, V. Tipparaju, and J. S. Vetter. The scalable heterogeneous computing (SHOC) benchmark suite. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, pages 63–74. ACM, 2010.

[8] A. Eklund, P. Dufort, D. Forsberg, and S. M. LaConte. Medical image processing on the GPU–past, present and future. *Medical image analysis*, 17(8):1073–1094, 2013.

[9] B. Fang, K. Pattabiraman, M. Ripeanu, and S. Gurumurthi. GPU-Qin: A methodology for evaluating the error resilience of GPGPU applications. In *Performance Analysis of Systems and Software (ISPASS), 2014 IEEE International Symposium on*, pages 221–230. IEEE, 2014.

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

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

[12] S. Fu and C. Xu. Quantifying temporal and spatial correlation of failure events for proactive management. In *Reliable Distributed Systems, 2007. SRDS 2007. 26th IEEE International Symposium on*, pages 175–184. IEEE, 2007.

[13] A. Gainaru, F. Cappello, M. Snir, and W. Kramer. Fault prediction under the microscope: A closer look into HPC systems. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, page 77. IEEE Computer Society Press, 2012.

[14] S. K. S. Hari, T. Tsai, M. Stephenson, S. W. Keckler, and J. Emer. SASSIFI: Evaluating resilience of GPU applications. In *Proceedings of the Workshop on Silicon Errors in Logic-System Effects*, 2015.

[15] B. He, W. Fang, Q. Luo, N. K. Govindaraju, and T. Wang. Mars: a MapReduce framework on graphics processors. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pages 260–269. ACM, 2008.

[16] R. K. Jena, M. M. Aqel, P. Srivastava, and P. K. Mahanti. Soft computing methodologies in bioinformatics. *European Journal of Scientific Research*, 26(2):189–203, 2009.

[17] 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.

[18] A. Jog, O. Kayiran, N. Chidambaram Nachiappan, A. K. Mishra, M. T. Kandemir, O. Mutlu, R. Iyer, and C. R. Das. OWL: cooperative thread array aware scheduling techniques for improving GPGPU performance. In *ACM SIGPLAN Notices*, volume 48, pages 395–406. ACM, 2013.

[19] J. Kim, M. Sullivan, and M. Erez. Bamboo ECC: Strong, safe, and flexible codes for reliable computer memory. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, pages 101–112. IEEE, 2015.

[20] J. Kim, M. Sullivan, S.-L. Gong, and M. Erez. Frugal ECC: Efficient and versatile memory error protection through fine-grained compression. In *High Performance Computing, Networking, Storage and Analysis, 2015 SC-International Conference for*, pages 1–12. IEEE, 2015.

[21] D. B. Kirk and W. H. Wen-Mei. *Programming massively parallel processors: a hands-on approach*. Morgan kaufmann, 2016.

[22] R. Koo and S. Toueg. Checkpointing and rollback-recovery for distributed systems. *IEEE Transactions on software Engineering*, (1):23–31, 1987.

[23] G. Li, K. Pattabiraman, C.-Y. Cher, and P. Bose. Understanding error propagation in GPGPU applications. In *High Performance Computing, Networking, Storage and Analysis, SC16: International Conference for*, pages 240–251. IEEE, 2016.

[24] J. Meng, S. Chakradhar, and A. Raghunathan. Best-effort parallel execution framework for recognition and mining applications. In *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1–12. IEEE, 2009.

[25] S. Mitra and Y. Hayashi. Bioinformatics with soft computing. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, 36(5):616–635, 2006.

[26] S. Mitra, S. K. Pal, and P. Mitra. Data mining in soft computing framework: a survey. *IEEE transactions on neural networks*, 13(1):3–14, 2002.

[27] 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 *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 519–530. IEEE, 2016.

[28] 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 *2017 IEEE 25th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 22–31. IEEE, 2017.

[29] 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 *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 95–106. IEEE, 2018.

[30] B. Nie, L. Yang, A. Jog, and E. Smirni. Fault site pruning for practical reliability analysis of GPGPU applications. In *51st Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2018, Fukuoka, Japan, October 20-24, 2018*, pages 749–761. IEEE Computer Society, 2018.

[31] NVIDIA. Computational finance.

[32] NVIDIA. CUDA C/C++ SDK Code Samples, 2011.

[33] NVIDIA. Fermi: NVIDIA's Next Generation CUDA Compute Architecture, 2011.

[34] A. Oliner and J. Stearley. What supercomputers say: A study of five system logs. In *Dependable Systems and Networks, 2007. 37th Annual IEEE/IFIP International Conference on*, pages 575–584. IEEE, 2007.

[35] A. Pecchia, D. Cotroneo, Z. Kalbarczyk, and R. K. Iyer. Improving log-based field failure data analysis of multi-node computing systems. In *Dependable Systems & Networks (DSN), 2011 IEEE/IFIP 41st International Conference on*, pages 97–108. IEEE, 2011.

[36] R. K. Sahoo, M. S. Squillante, A. Sivasubramaniam, and Y. Zhang. Failure data analysis of a large-scale heterogeneous server environment. In *Dependable Systems and Networks, 2004 International Conference on*, pages 772–781. IEEE, 2004.

[37] 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 *Dependable Systems and Networks (DSN), 2017 47th Annual IEEE/IFIP International Conference on*, pages 97–108. IEEE, 2017.

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

[39] B. Schroeder and G. Gibson. A large-scale study of failures in high-performance computing systems. *IEEE Transactions on Dependable and Secure Computing*, 7(4):337–350, 2010.

[40] M. Snir, R. W. Wisniewski, J. A. Abraham, S. V. Adve, S. Bagchi, P. Balaji, J. Belak, P. Bose, F. Cappello, B. Carlson, et al. Addressing failures in exascale computing. *The International Journal of High Performance Computing Applications*, 28(2):129–173, 2014.

[41] D. Tiwari, S. Gupta, J. Rogers, D. Maxwell, P. Rech, S. Vazhkudai, D. Oliveira, D. Londo, N. DeBardeleben, P. Navaux, et al. Understanding GPU errors on large-scale HPC systems and the implications for system design and operation. In *High Performance Computer Architecture (HPCA), 2015 IEEE 21st International Symposium on*, pages 331–342. IEEE, 2015.

[42] H.-L. Truong and T. Fahringer. Soft computing approach to performance analysis of parallel and distributed programs. pages 622–622. Springer, 2005.

[43] 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*, pages 90–100.

[44] R. Venkatagiri, A. Mahmoud, S. K. S. Hari, and S. V. Adve. Approxilyzer: Towards a systematic framework for instruction-level approximate computing and its application to hardware resiliency. In *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*, pages 1–14. IEEE, 2016.

[45] W. Webber, A. Moffat, and J. Zobel. A similarity measure for indefinite rankings. *ACM Transactions on Information Systems (TOIS)*, 28(4):20, 2010.

[46] L. Yang, B. Nie, A. Jog, and E. Smirni. Practical resilience analysis of GPGPU applications in the presence of single- and multi-bit faults. In *Transactions on Computers*. IEEE, 2020.

[47] K. S. Yim, C. Pham, M. Saleheen, Z. Kalbarczyk, and R. Iyer. Hauberk: Lightweight silent data corruption error detector for GPGPU. In *Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International*, pages 287–300. IEEE, 2011.

[48] D. H. Yoon and M. Erez. Memory mapped ECC: low-cost error protection for last level caches. In *ACM SIGARCH Computer Architecture News*, volume 37, pages 116–127. ACM, 2009.

[49] D. H. Yoon and M. Erez. Virtualized and flexible ECC for main memory. In *ACM SIGARCH Computer Architecture News*, volume 38, pages 397–408. ACM, 2010.