Enabling Software Resilience in GPGPU Applications via Partial Thread Protection

Lishan Yang
William & Mary
Williamsburg, VA
lyang11@email.wm.edu

Bin Nie
William & Mary
Williamsburg, VA
bnie@email.wm.edu

Adwait Jog
William & Mary
Williamsburg, VA
ajog@wm.edu

Evgenia Smirni
William & Mary
Williamsburg, VA
esmirni@cs.wm.edu

***Abstract*—Graphics Processing Units (GPUs) are widely used by various applications in a broad variety of fields to accelerate their computation but remain susceptible to transient hardware faults (soft errors) that can easily compromise application output. By taking advantage of a general purpose GPU application hierarchical organization in threads, warps, and cooperative thread arrays, we propose a methodology that identifies the resilience of threads and aims to map threads with the same resilience characteristics to the same warp. This allows engaging partial replication mechanisms for error detection/correction at the warp level. By exploring 12 benchmarks (17 kernels) from 4 benchmark suites, we illustrate that threads can be remapped into reliable or unreliable warps with only 1.63% introduced overhead (on average), and then enable selective protection via replication to those groups of threads that truly need it. Furthermore, we show that thread remapping to different warps does not sacrifice application performance. We show how this remapping facilitates warp replication for error detection and/or correction and achieves an average reduction of 20.61% and 27.15% execution cycles, respectively comparing to standard duplication/triplication.**

***Index Terms*—Reliability, GPGPU application resilience, Transient faults, Thread remapping**

I. Introduction

As general purpose GPUs (GPGPUs) are becoming increasingly susceptible to transient hardware faults (soft errors) often from cosmic radiation [1] or from operating under low voltage [2], their reliable operation is of critical importance. With GPGPUs becoming omnipresent in fields such as high-performance computing (HPC), artificial intelligence, deep learning, virtual/augmented reality, and safety critical systems such as autonomous vehicles [3]–[10], transient hardware faults can lead to bit flips in storage devices including the register file and DRAM. Such bit flips are increasing in frequency as system scales increase especially in the HPC domain [11]–[13]. If bit flips occur during application execution, they may result in application crashes/hangs or even worse in silent data corruption (SDC) where the application successfully completes execution but its output is incorrect. Executions that result in SDC outcomes are the most undesirable as they erroneously provide the user with the illusion of correct output, although cases of SDC output that is within certain user-acceptable ranges may exist [14]. To ensure reliable application execution, several mechanisms are widely employed including error correction codes (ECC) [15]–[17], but ECC cannot still provide protection to datapath errors that originate from unprotected latches in functional units (e.g., arithmetic logic and load-store units) [18].

Reliable execution of GPGPU applications requires high-overhead protection mechanisms such as check-pointing [19], [20] or software solutions that are based on replication. In the GPU domain such replication can be done at different levels: at the kernel, thread, or instruction level. At the thread level, replication is based on using redundant copies of a thread (or block of threads) and then on comparing their results [21]. Different compiler-based implementations of this idea [21], [22] aim to reduce the unavoidable synchronization overhead between the original and redundant threads. If replication is done at the instruction level [23], then the overhead of redundant multi-threading can still be significant. In addition, not all dynamic instructions are typically covered.

In this paper, we offer an orthogonal approach that is based on the fact that thread resilience profiles within a GPGPU application may differ significantly – some threads are inherently resilient, while some are not [24], thread resilience may also depend on application input [25]. Application resilience eventually depends on the thread organization of GPGPU application software. In GPGPU applications threads are arranged at three levels: kernels, thread blocks (or cooperative thread arrays (CTAs) in CUDA terminology), and warps. Each GPU core schedules work at a granularity of warp, which is usually a group of 32 threads. Each group executes the same instruction in a lock-step manner. This is essentially the basis of single-instruction-multiple-thread (SIMT) execution.

Our thesis is that GPGPU software resilience can be achieved via *selective warp replication* provide that threads remapped into warps such that warps consist of threads that are either reliable or unreliable. Therefore, if warps consist of threads that are inherently reliable, these warps (their threads or their instructions) *do not have to be replicated to increase their resilience*. Instead, only warps that contain unreliable threads need to be replicated. The advantage here comes from scheduling of warps in the single-instruction-multiple-data (SIMD) paradigm: as threads within the warps are scheduled in a lock-step way, it is a lot easier to replicate an entire warp of unreliable threads rather than replicate individual threads within warps (or instructions within threads) and reconcile their outcome as the classic redundant multi-threading [21],

[22] advocates.

The process of thread remapping at the warp level is transparent to the software developer and offers a simple way to reorganize code with minimal effort. We stress that the application resilience profile (i.e., the percentage of application executions that result in crashes/hangs, SDC, and correct executions in presence of bit flips) strongly depends on branch divergence and input data taken by different threads [26]. Since application resilience is tied to input, it cannot possibly guide software development. The remapping that we propose in this paper allows the developer to improve application resilience in a transparent way, either by changing the thread-warp mapping to activate replication for a fully transparent approach to the code developers, or by providing guidance to the developer to simply reorganize threads in such a manner that facilitates replication but does not interfere with the parallelization and synchronization logic of the software.

In summary, we make the following contributions:

- Based on individual thread resilience we categorize the warps into three classes: a) Reliable warps where all threads are resilient to single-bit errors, b) Unreliable warps where all threads are unreliable, and c) Mixed warps that contain both reliable and unreliable threads. We show that mixed warps are abundant in kernels.
- We propose a low-overhead partial thread protection mechanism by remapping threads such that the number of mixed warps is minimized. In other words, we change the thread to warp mapping such that distinct reliable and unreliable warp groups are formed. This facilitates the need for protecting *only* unreliable warps as this remapping maintains the per-thread resilience profile.
- We present experiments using 12 benchmarks (17 kernels) from the AxBench, CUDA, PolyBench, and Rodinia suites [27]–[30] and show that 7 of these kernels can benefit from remapping. We show that remapping increases on the average the percentage of reliable warps from 23.40% to 42.08%, while incurring only 1.63% execution overhead due to increased number of stalls in shared memory.
- By duplicating or triplicating the warps, we can easily detect when an error occurs (if duplication is used) or correct the error via triplication [21], [22]. We show that by selectively replicating warps that contain unreliable threads after remapping (i.e., unreliable or mixed warps), we achieve average performance savings 20.61% and 27.15%, for detection and protection, respectively.

The remaining of the paper is organized as follows. Section II describes the background of GPU architecture and the fault model used in this paper. Section III presents characterization regarding various thread resilience patterns observed in the studied benchmarks. Inspired by the characterization results, we propose a partial thread protection mechanism via remapping; the details can be found in Section IV. Section V evaluates the performance gains as well as the overhead of remapping. Then, Section VI discusses related work, and eventually we conclude in Section VII.

II. Background

In this section, we provide a brief introduction on the baseline GPU architecture and the GPGPU execution model. We also discuss the fault model, fault injection method, and application resilience profile.

A. GPUs and GPGPU Application Structure

Baseline GPU Architecture. A GPU typically is equipped with a large number of cores, also known as streaming-multiprocessors (SMs) in NVIDIA terminology [15]. Each core has its private L1 cache, software-managed scratchpad memory, and a large register file. An interconnection network connects all these cores to global memory, which consists of various memory channels (partitions). Every memory channel has a shared L2 cache, and its associated memory requests are handled by a GDDR5 memory controller. There are various protection techniques for single-bit faults in recent commercial GPUs [15]–[17], including single-error-correction double-error-detection (SEC-DED) error correction codes (ECCs) that protect register files, L1/L2 caches, shared memory and DRAM against soft errors. Other structures such as arithmetic logic units (ALUs), thread schedulers, instruction dispatch units, load/store units (LSUs), and interconnection network are not protected [15]–[17].

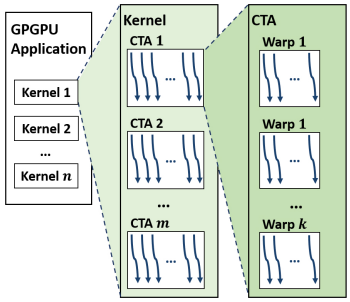


Fig. 1. GPU Software Execution Model.

GPGPU Software Execution Model. Following the single-instruction-multiple-thread (SIMT) philosophy [31], GPGPU applications execute thousands of threads concurrently over large amounts of data. This helps in masking the latency and achieving high throughput. A typical GPGPU application launches various kernels on the GPUs, see Figure 1. Each kernel is divided into groups of threads, known as *thread blocks*, which are called *Cooperative Thread Arrays* (CTAs) in CUDA terminology. A CTA encapsulates all synchronization and barrier primitives among a group of threads [31], [32]. This CTA formation enables the GPU hardware to relax the execution order of the CTAs, for the purpose of maximizing parallelism. Threads inside one CTA can be further divided into groups of 32 individual threads, known as warps. As the most fine-grained level in terms of scheduling, warps execute a single instruction on the functional units in lock step. This

TABLE I
SELECTED BENCHMARKS.

Suite	Benchmark	Kernel Name	Kernel ID	Pct. of reliable warps	Pct. of reliable threads
AxBench	Jmeint	Jmeint_kernel	K1	0.00%	55.15%
	Laplacian	LaplacianFilter	K1	49.38%	54.08%
	MeanFilter	AverageFilter	K1	17.19%	26.55%
CUDA	NN (NeuralNetwork)	executeFirstLayer	K1	100.00%	100.00%
		executeSecondLayer	K2	100.00%	100.00%
		executeThirdLayer	K3	100.00%	100.00%
		executeFourthLayer	K4	100.00%	100.00%
	SCP	scalarProdGPU	K1	0.00%	0.00%
PolyBench	2DCONV	Convolution2D_kernel	K1	0.00%	12.11%
	MVT	mvt_kernel1	K1	0.00%	0.00%
Rodinia	Gaussian	Fan1	K1	87.50%	90.62%
		Fan2	K2	63.89%	95.87%
	HotSpot	calculate_temp	K1	25.00%	43.75%
	NearestNeighbor	euclid	K1	0.56%	0.57%
	PathFinder	dynproc_kernel	K1	8.33%	19.79%
	SRAD	reduce	K3	100.00%	100.00%
		srad	K4	100.00%	100.00%

sub-division of warps is an architectural abstraction, which is transparent to the application programmer.

B. Fault Model

We assume that register files and other components such as caches and memory are protected by ECC (which is the case in almost all GPUs). We simulate commonly occurring computation-related errors due to transient faults (known as soft errors) in ALUs/LSUs. These faults can lead to wrong ALU output which would then be stored in destination registers, or corrupted variables loaded by an LSU. This erroneous computing operation is what we emulate by injecting faults directly to destination register values. This is a standard experimental methodology for GPGPU reliability studies [18], [24], [33]–[35].

The fault injection methodology used here closely follows the one used in [24], [36]: we flip a bit at a destination register identified by the thread id, the instruction id, and a bit position. We perform our reliability evaluations on GPGPU-Sim [37] with PTXPlus mode. GPGPU-Sim is a widely-used cycle-level GPU architectural simulator, and its PTXPlus mode provides a one-to-one mapping of instructions to actual ISA for GPUs [36], [37]. Any fault injection tool or technique. (e.g., SASSIFI [18] or NVBitFI [38]) can be used for evaluating the application reliability, i.e., the technique presented in this paper does not depend on GPGPU-Sim.

GPGPU Application Resilience Profile. For each fault injection experiment, there are three possible outcomes:

- **masked** output: the application output is identical to that of fault-free execution.
- **silent data corruption (SDC)** output: the fault injection run exits successfully without any error, but the output is incorrect.
- **other**: the fault injection run results in a crash or hang.

To obtain the resilience profile of an application run, we conduct an experimental campaign using the state-of-the-art fault injection methodology proposed by Nie et al. [24] that aggressively prunes the fault space while achieving accuracy that is remarkably close to the ground truth. Within the pruned fault space, we conduct one run per fault location (one single bit flip) and evaluate the application outcome as **masked, SDC** or **other**. We aggregate the outcome of all experiments to obtain the application *resilience profile*, i.e., what percentage of the runs are expected to result in masked, SDC, or other outputs. The lower the SDC percentage, the higher the application resilience. In this paper, we focus on reducing the percentage of SDC outputs. Faults that lead to masked outputs can be ignored, while faults that lead to a crash or hang are easily detected.

In this work we focus on how to improve application resilience protection when a single bit fault occurs. The proposed methodology can be readily extended to multi-bit fault models [39].

III. CHARACTERIZATION

We conducted experiments across 12 benchmarks (17 kernels) selected from 4 benchmark suites [27]–[30], listed in Table I. The selected benchmarks cover different application domains including 3D gaming, image processing, and scientific computations. Past work has established that different

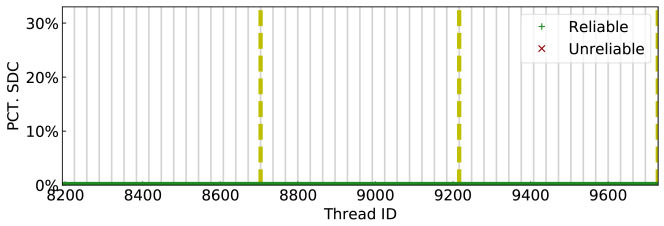


Fig. 2. All the threads are reliable in SRAD K3. Gray solid lines separate different warps, and yellow dashed lines separate different CTAs. Due to space constraint, here we only show the first 3 CTAs in the kernel. There are in total 8 CTAs in SRAD K3.

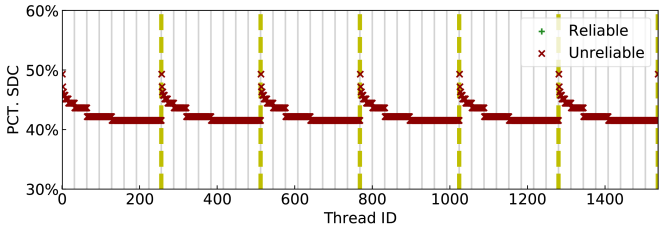


Fig. 3. All threads are unreliable in SCP. Gray solid lines separate different warps, and yellow dashed lines separate different CTAs. There are 128 CTAs in total, but due to space constraint, we only show the first 6 CTAs.

GPU threads have different resilience profiles and that the thread dynamic instruction count (iCnt) can be used as a proxy of individual thread resilience [24], [33]. Indeed, fault site pruning [24] is based on this exact concept: it demonstrates that threads with the same dynamic instruction count have the same resilience profile, therefore it is sufficient to select one thread from each group with the same iCnt for fault injection and extrapolate the thread resilience of the entire group from a single thread. Our experiments further corroborate here what past work has also shown: different threads have typically different resilience. Understanding the patterns of thread resilience is helpful for scheduling purposes to improve application resilience. We categorize benchmarks into three cases: reliable, unreliable, and mixed, based on the thread resilience within each warp. We focus on warps because a warp is the smallest scheduling unit. In addition, it is not desirable to change the thread-CTA mapping. If done so, it breaks the synchronization and thread communication within a CTA, requiring significant effort in redesigning the parallel software logic.

The percentage of SDC outcomes across the various numbers of experiments can characterize one thread as reliable or unreliable. For example, if the percentage of fault-injected runs that result in SDC outcomes is smaller than a small number (typically in the range from zero to 5%, essentially if its resilience coverage is 95%), then we characterize the thread as reliable, otherwise it is deemed unreliable. In the following, we show some example cases.

1. All threads are reliable. Some applications are very resilient to faults. Figure 2 shows the resilience scatter plot of different threads in SRAD K3. Threads are organized in thread launching order. We use the gray solid lines to separate different warps, and use yellow dashed lines to separate different CTAs. Due to space constraint, here we only show the first 3 CTAs in SRAD K3, but the same pattern repeats across all CTAs: all threads in SRAD K3 are reliable. Similar to SRAD K3, SRAD K4 and all NN kernels are highly resilient to soft errors.

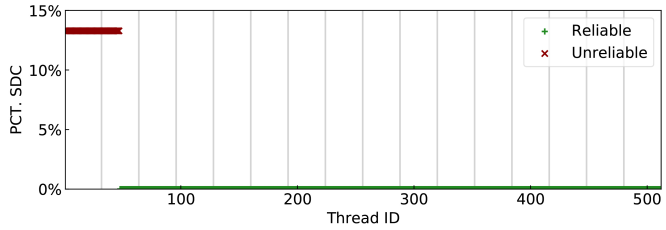
2. All threads are unreliable. Some applications have a high probability of SDC outputs when faults are injected. Figure 3 shows the percentage of SDC outputs per thread for SCP. Here, all threads have more than 40% SDC outputs. An application with similar resilience behavior is MVT, with 63.82% SDC outputs for all of its threads.

3. Mixed Reliable and Unreliable Threads within Warps. Reliable and unreliable threads can co-exist in the same warp (and consequently CTA), see Figure 4. Reliable threads are marked with a green '+', while red 'x' represents unreliable threads and marks their SDC probability. We start from two simple benchmarks: Gaussian K1 and NearestNeighbor (Figure 4(a) and (b), respectively). For Gaussian K1, there are in total 512 threads organized in one CTA only. The first 48 threads are unreliable, and the remaining threads are very resilient (their percentage of SDC outputs is 0%). NearestNeighbor shows a similar resilience pattern: threads at the beginning are unreliable, those that are launched later are reliable. There are in total 168 CTAs in NearestNeighbor. Due to the space constraint, here we only show the last 6 CTAs which can best express the idea of well-organized warps. For both Gaussian K1 and NearestNeighbor, reliable threads and unreliable threads are already organized separately within different warps (with the exception of a single warp either at the start for Gaussian or at the tail for NearestNeighbor that contains both reliable and unreliable threads).

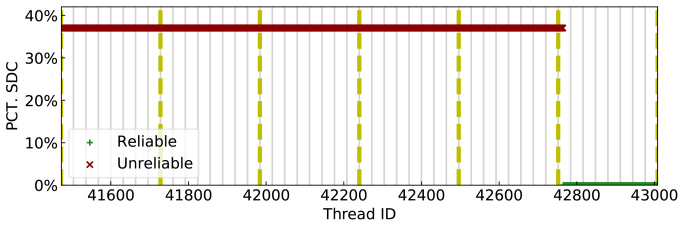
However, there are benchmarks where their threads are not that well-organized. For HotSpot, shown in Figure 4(c), reliable and unreliable threads are mixed within different warps. Due to space constraint, here we only show the first 6 CTAs at the beginning for HotSpot. As shown in Table I, the percentage of reliable warps (warps containing only reliable threads) is 25% for HotSpot, but there are in total 43.75% reliable threads.

Similarly, in Jmeint, there is no reliable warp, because all of the warps have both reliable and unreliable threads, as shown in Figure 4(c). For Jmeint more than half (55.15%) of the threads are reliable. However, since they are mixed in CTAs with the remaining 44.85% unreliable threads, protecting via replication would require replication of the entire kernel, i.e., every warp. Similar observations apply to Laplacian, MeanFilter, 2DCONV, Gaussian K2, and PathFinder, see Figure 4(e)-(h).

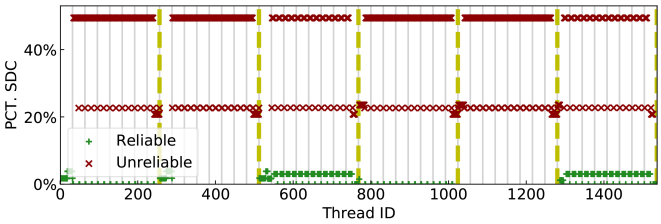
Figure 4 clearly illustrates that there is ample scope for partial protection: If we group threads judiciously, then we can



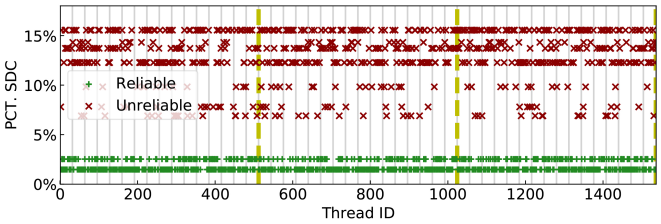
(a) Gaussian K1.



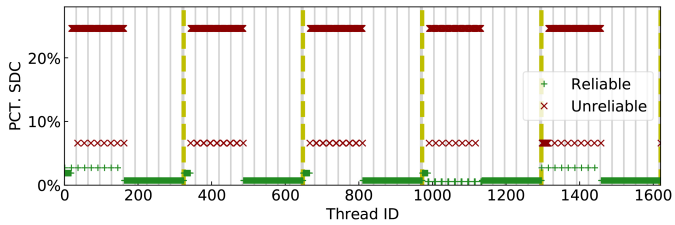
(b) NearestNeighbor (last 6 CTAs).



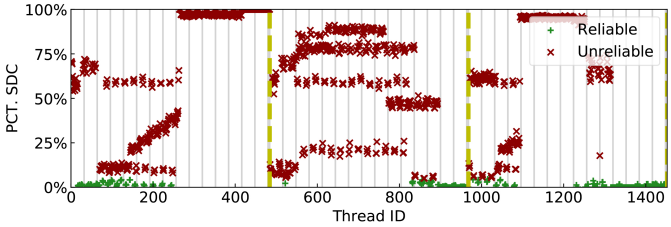
(c) HotSpot (first 6 CTAs).



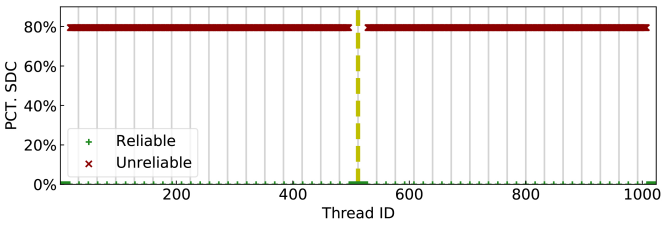
(d) Jmeint (first 3 CTAs).



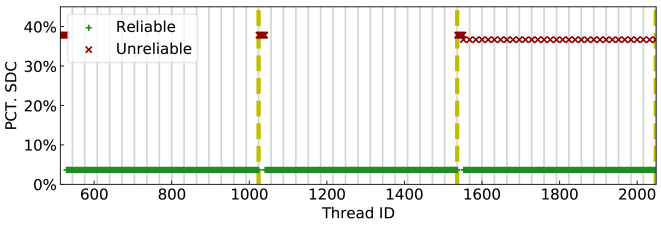
(e) Laplacian (first 5 CTAs).



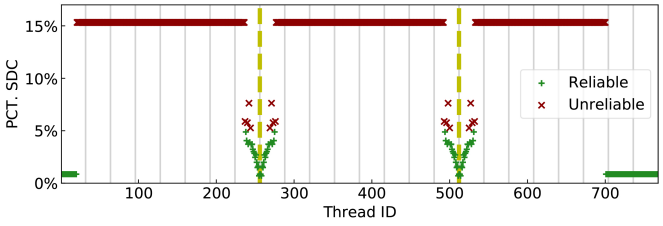
(f) MeanFilter (first 3 CTAs).



(g) 2DCONV.



(h) Gaussian K2 (first 3 CTAs).



(i) PathFinder.

Fig. 4. Reliable and unreliable threads exist together in the same warp. Due to space constraints, we only show the a part of the CTAs for NearestNeighbor, HotSpot, Jmeint, Laplacian, MeanFilter, and Gaussian K2.

increase the percentage of reliable warps, and avoid redundant protection of warps (threads) that are anyway resilient. Table II summarizes the benchmark categorization.

Summary. From the reliability perspective, there is no need to protect reliable threads. Benchmarks where all threads are reliable, result in reliable kernel executions. Similarly, for benchmarks that have warps that are all unreliable, protection needs to be applied to the entire kernel. Approaching kernel reliability from the scheduling perspective, threads are grouped and scheduled in units of warps, which is transparent to software developers. For benchmarks that consist of both reliable and unreliable threads, we explore ways to *remap* threads into warps such that warps consist of *only* reliable or unreliable threads. If this is done, then it is not necessary

TABLE II
BENCHMARK CATEGORIES

Category		Benchmark
All threads are reliable		NN K1, NN K2, NN K3, NN K4, SRAD K3, SRAD K4
All threads are unreliable		SCP, MVT
Mixed warps	Well-organized	Gaussian K1, NearestNeighbor
	Need Remapping	Jmeint, MeanFilter, 2DCONV, HotSpot, Gaussian K2, PathFinder, Laplacian

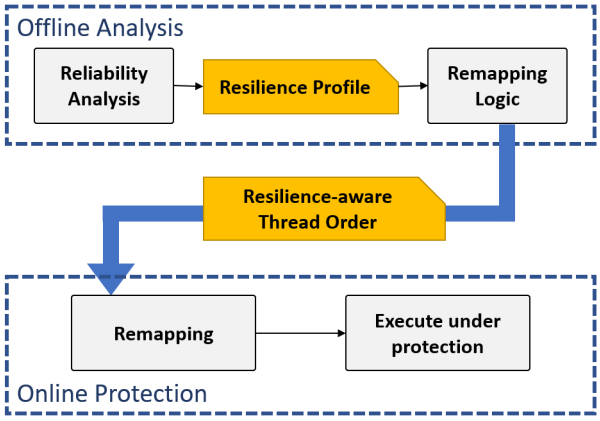


Fig. 5. Workflow.

to protect the application fully but instead focus on protecting unreliable warps only.

IV. Resilient Software Protection via Remapping

In [24], threads are identified as the most important GPGPU component, and the resilience pattern of an application can be derived from thread resilience. Here, we propose a low-overhead partial protection mechanism that leverages thread resilience patterns via remapping. The main idea is to remap threads to warps for the purpose of separating reliable and unreliable threads, as scheduling of threads can be done at the warp granularity. By addressing the problem at the warp level, we propose to recompute warps that contain unreliable threads, essentially offering *partial protection* to a subset of warps and not the entire kernel, without compromising application reliability.

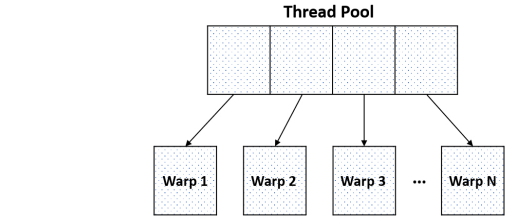
Figure 5 shows the workflow of the proposed protection mechanism. There are two components: 1) Offline analysis, to obtain the resilience-aware thread order, and 2) Online protection, which uses this resilience-aware thread order to achieve low-overhead protection.

1) **Offline Analysis.** For any target kernel and for a specific input, the resilience of each thread needs to be first obtained. There is no restriction on which method of reliability analysis is used. Fault injection campaigns [18], [24], [33], ACE (Architecturally Correct Execution) analysis, or a combined method leveraging both fault injection and ACE analysis [40] can be used. The only requirement is that the resilience of every thread needs to be evaluated. This is not difficult to do, despite the fact that most GPGPU applications have tens of thousands of threads, because for most benchmarks threads with the same DI count have the same resilience behavior [24], [33]. This reduces the number of experiments that need to be done to obtain the thread resilience profile. In this work, we evaluate thread and kernel resilience using the fault site pruning technique [24].
We first identify the resilience profile of threads in their launching order (see Figure 4). Then, threads are re-ordered into warps following the remapping logic: threads with similar resilience are remapped into the same warp. Detailed explanation is in Subsection IV-A.
2) **Online Protection.** Based on the resilience-aware thread order obtained from offline analysis, we can remap threads before execution. The actual remapping idea can be implemented in various ways. In this work, we directly change the thread-warp mapping (see Subsection IV-A for implementation details). After remapping, threads are executed, and error detection/correction is applied to unreliable warps only. Error detection/correction can be implemented and applied in various ways. Here, we use warp duplication for error detection, and warp triplication for error correction. Details are given in Subsection IV-B.

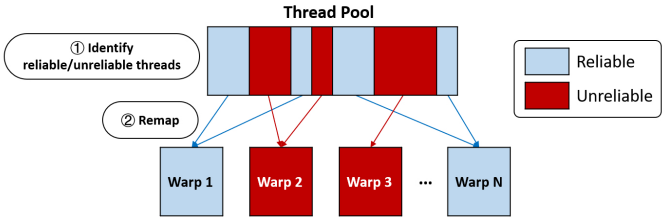
A. Remapping

CTAs are collections of threads defined by the CUDA programmer. The thread to warp mapping is done linearly by default (i.e., threads are allocated to warps in groups of 32 as shown in Figure 6(a)). To change the mapping between thread to warp, we first identify reliable/unreliable threads in offline analysis, then remap threads, see Figure 6(b). By changing the linear thread order, we can group threads with the same resilience (i.e., percentage of SDC outputs) into the same warp, and use different resilience protection at the warp level according to their reliability profile. Note that remapping is done within each CTA, but not across CTAs. This is because synchronization is ensured inside each CTA. Remapping across different CTAs can affect their synchronization, hence introduce errors in the software logic.

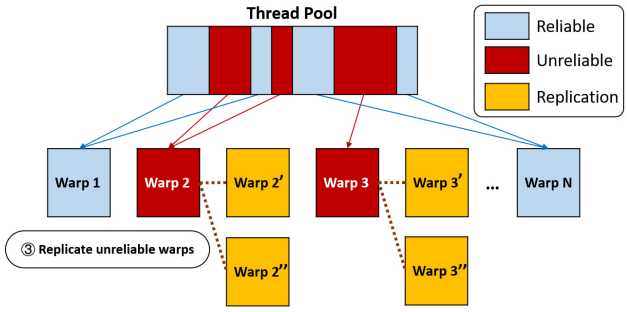
We use GPGPU-Sim [37] to implement the above. At the initialization phase, CTAs are constructed. Without remapping, threads are launched linearly. With remapping, we fill each CTA according to its resilience-based launching order. This resilience-based launching order is decided based on offline profiling. We start fetching threads from the beginning of the linear launching order, and put reliable threads into a warp. Meanwhile, we organize unreliable threads into another warp, if any. When a warp is filled (32 threads), the warp is ready for execution, and a new warp is formed for the upcoming threads. When all the threads are remapped into warps, if there are any partially filled reliable or unreliable warps, they are combined



(a) Thread-warp mapping.



(b) Remapping.



(c) Remapping and protection.
Unreliable warps are replicated once for detection.
If applying error correction, there are two replicas.

Fig. 6. Logic of mapping, remapping, and protection.

into one mixed warp and ready for execution. It is important to note that thread remapping to different warps does not affect their reliability profile because thread resilience is typically determined by branch divergence and input data and not the order of thread execution [41].

B. Partial Protection

In addition to remapping, error detection/correction can be applied to unreliable warps. Here, we use warp replication/triplication, to demonstrate how partial protection works. During remapping, when an unreliable warp is filled (32 threads), we replicate it into another warp and send both warps to execute. After these two warps finish execution, thread outputs (usually the outputs are the computation results to be written into memory by *store* instructions) are compared to detect whether there is any difference, see Figure 6(c). Since warps are the smallest unit for scheduling at the GPU level, duplication at the warp level is transparent for the programmer to handle than at the thread level. Duplication at the CTA level would require redoing the logic of communication/synchronization among threads, a far more challenging software effort. This is fully avoided by handling replication at the warp level.

If error correction is applied, then each unreliable warp is triplicated, according to triple modular redundancy (TMR). In Figure 6(c), warp-2 is triplicated into warp-2' as well as warp-2" for error correction. Since reliable warps do not need error detection/correction, they are not replicated/triplicated. Note that, mixed warps that have both reliable and unreliable threads also need to be duplicated or triplicated for error detection/correction.

V. Evaluation

In this section we present a detailed evaluation of thread remapping (Section V-A). Then, we discuss the overhead magnitude when applying protection via remapping (Section V-B).

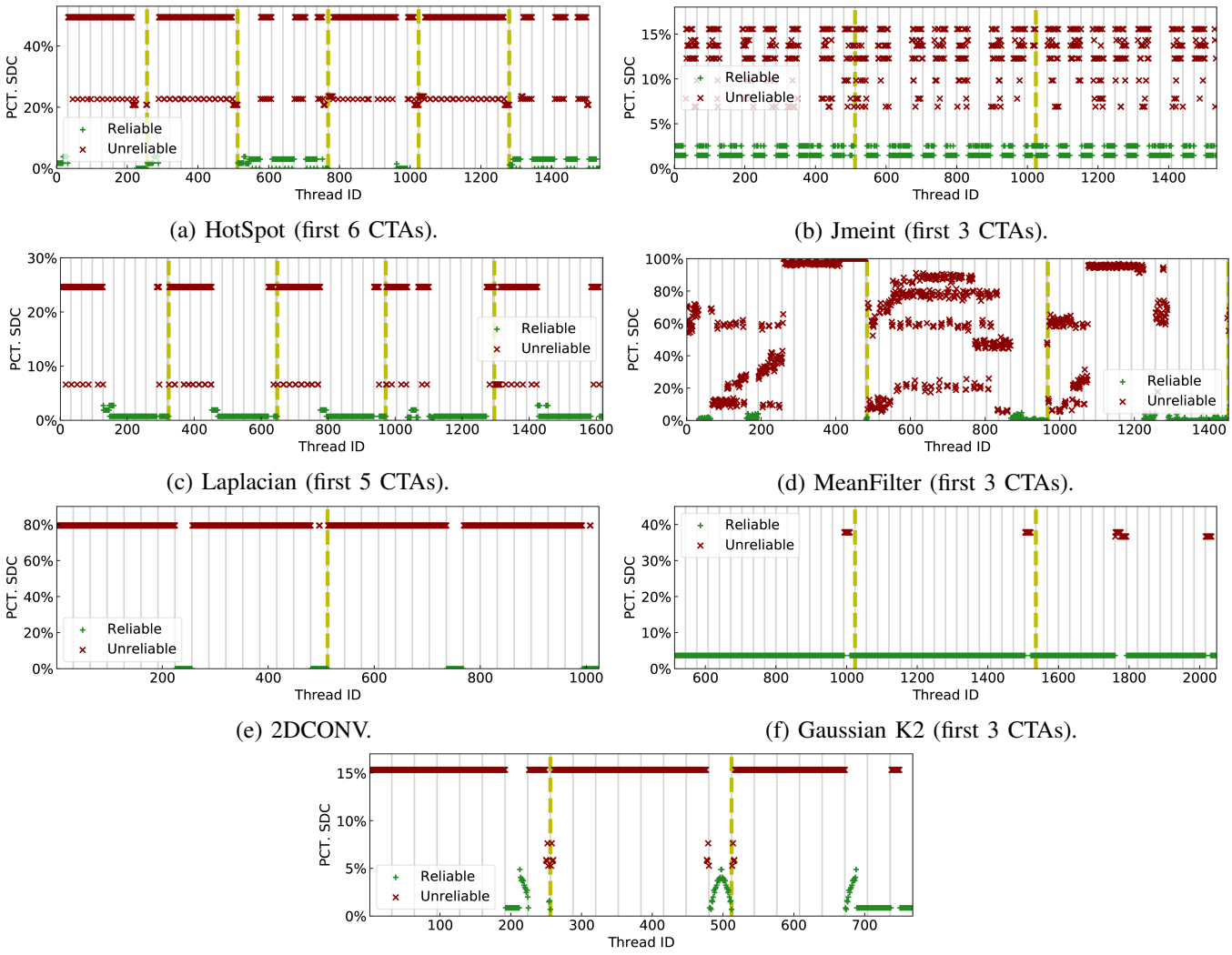
A. Effectiveness of Thread Remapping

We first show the resilience pattern of different benchmarks after remapping, see Figure 7. In this figure, if a thread has an SDC probability less or equal to 5%, it is considered reliable, and remapping is performed based on this 5% SDC threshold, i.e., our goal is a 95% reliability coverage. Gray solid lines in Figure 7 separate different warps, and yellow dashed lines separate different CTAs. For HotSpot, in the first CTA, originally all the reliable threads in Figure 4(c) are distributed across all warps. After remapping, these reliable threads are gathered and scheduled in the first and last warp of the CTA. We end up with 1, 5, 1, and 5 reliable warps for the second, third, fourth, and sixth CTA, respectively. There is a mixed warp at the end of the third CTA. Since there are still unreliable threads in this mixed warp, it still needs protection. For the fifth CTA, all threads are unreliable, therefore the thread resilience pattern is the same before and after remapping. The resilience patterns after remapping for Jmeint, Laplacian, MeanFilter, 2DCONV, Gaussian K2, and PathFinder are shown in Figure 7(b)-(f).

The improvement in terms of the percentage of reliable warps for applications is shown in Figure 8. On average, originally the percentage of reliable warps is 23.40%. By remapping, the percentage increases to 42.08%. The biggest improvement happens on Jmeint, where there are 52% reliable warps after remapping from the original 0%.

Changing the SDC tolerance threshold can result in different remappings. Figure 9 shows how remapping changes the resilience pattern when different SDC thresholds are applied in Jmeint. If the SDC threshold is set to 2%, there are a few reliable threads to be remapped, see Figure 9(a). For increased SDC thresholds, remapping results in more reliable warps, see the changes of resilience patterns in Figure 9(a) to (d).

For Hotspot, even for SDC threshold equal to 0%, there are still several reliable warps (two warps in Figure 10(a) within the first 6 CTAs and in total 25% for the whole kernel). With the SDC threshold increasing, see Figure 10(b)-(d), remapping changes the resilience pattern, and more reliable threads are gathered together.



(a) HotSpot (first 6 CTAs).

(b) Jmeint (first 3 CTAs).

(c) Laplacian (first 5 CTAs).

(d) MeanFilter (first 3 CTAs).

(e) 2DCONV.

(f) Gaussian K2 (first 3 CTAs).

(g) PathFinder.

Fig. 7. Resilience patterns after remapping. If a thread has SDC probability less or equal to 5%, it is considered reliable. Due to space constraint, we only show the first several CTAs for HotSpot, Jmeint, Laplacian, MeanFilter, and Gaussian K2.

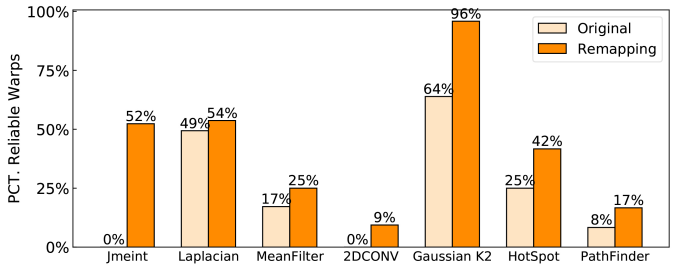
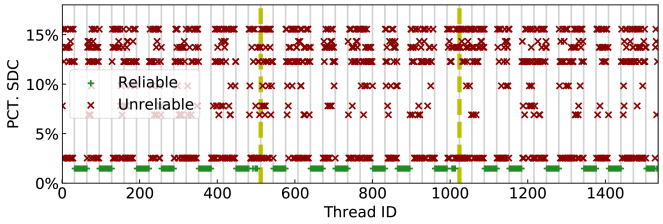


Fig. 8. Percentage of reliable warps before and after remapping.

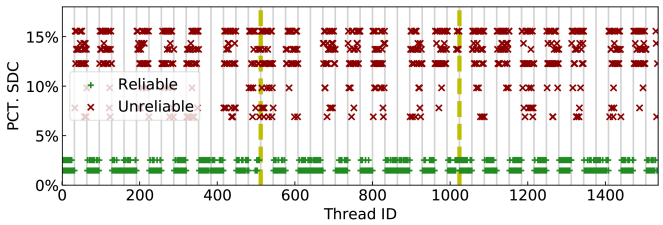
Figure 11 shows how the percentage of SDC outputs changes when the SDC threshold increases for all 7 benchmarks eligible for remapping. Jmeint and PathFinder are the first two benchmarks reaching 100% reliable warps, with SDC threshold less than 20%. Gaussian K2 has 94% reliable warps when the SDC threshold is 3.6% only. This is because the SDC percentage of its major thread group is 3.6%. HotSpot reaches 100% reliable when SDC threshold is about 50%, and 2DCONV requires SDC threshold to be 80% to get 100% reliable warps. MeanFilter is the most complicated benchmark, and it reaches 100% reliable only when SDC threshold is set to 100%, because 15% of the threads have 100% SDC outputs. In general, we see that if we set the SDC threshold to 5% only, there is still ample room for remapping for most benchmarks, as shown in Figure 8.

B. Overhead Introduced by Remapping and Protection

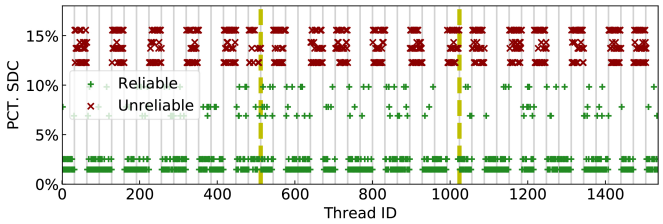
Thread remapping (and protection) may affect the performance of program execution. Because of the shared cluster environment we are using, pure timing measurement is not accurate. Instead, we use the number of instruction cycles measured using GPGPU-Sim performance mode to reflect the execution performance.



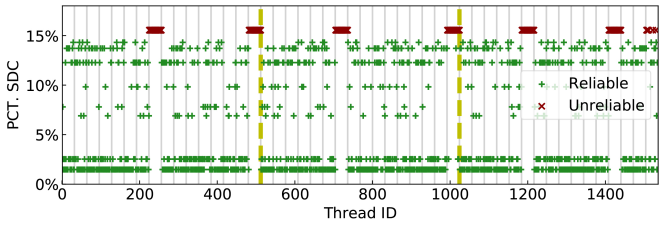
(a) threshold = 2%



(b) threshold = 5%

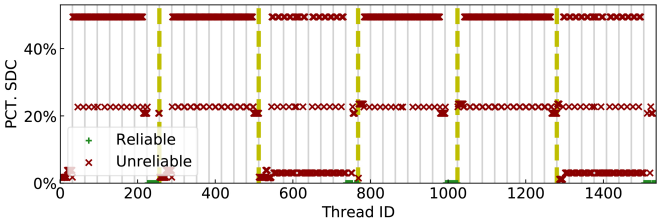


(c) threshold = 12%

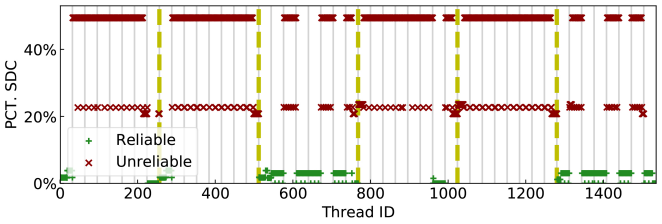


(d) threshold = 15%

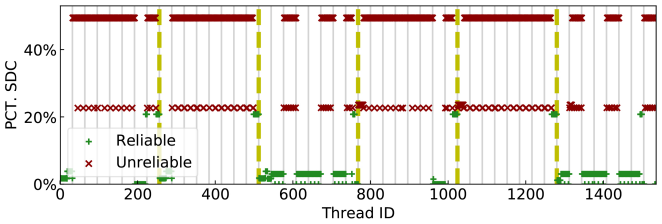
Fig. 9. Remapped resilience patterns of Jmeint under different SDC threshold. Due to space constraint, we only show the first 3 CTAs.



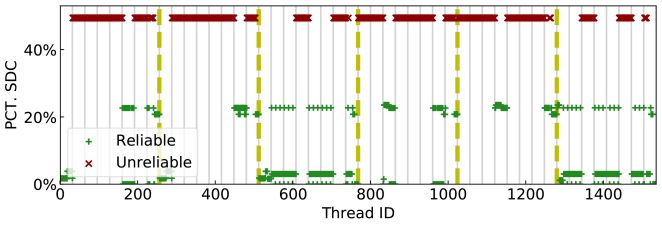
(a) threshold = 0%



(b) threshold = 5%



(c) threshold = 22%



(d) threshold = 25%

Fig. 10. Remapped resilience patterns of HotSpot under different SDC threshold. Due to space constraint, we only show the first 6 CTAs.

For overhead analysis, we first present the performance overhead due to remapping. The remapping overhead of each benchmark kernel is shown in Figure 12. On average, the remapping overhead is only 1.63%. Note that there are some benchmarks with negative overhead in Figure 12, such as Laplacian, 2DCONV, HotSpot, and PathFinder, in these cases execution cycles reduce with remapping.

To better understand why remapping may result in better performance, we look into various performance measures that are normalized over the original thread mapping. Figure 13 shows the normalized L1 data cache miss rate and the number of stalls caused by accessing shared memory. The numbers are normalized by the execution without remapping. On the one hand, from Figure 13(a), we observe that the L1 data cache miss rate is decreasing. On the other hand, Figure 13(b) shows that the number of stalls increases for 2DCONV, Jmeint, HotSpot, and PathFinder; for Laplacian, MeanFilter,

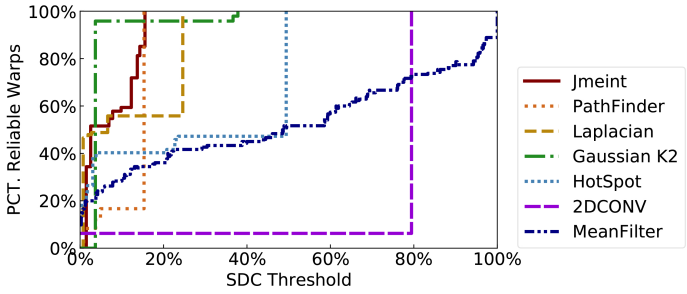


Fig. 11. Percentage of reliable warps grows as the SDC tolerance threshold increases.

and Gaussian K2, the number of stalls decreases. Trends are not consistent across benchmarks, therefore some gain and some lose performance with remapping. In sum, we claim that remapping does not significantly affect performance which

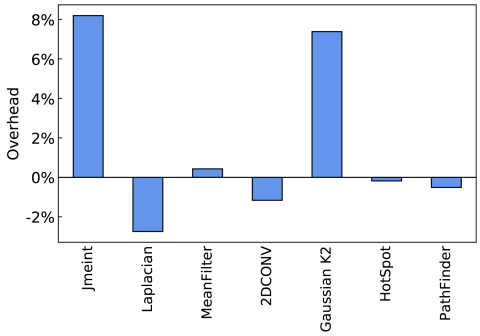
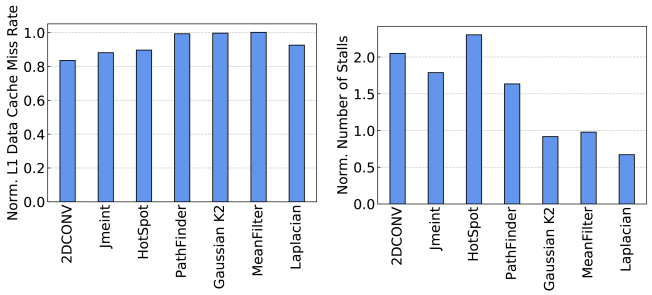


Fig. 12. Overhead of remapping.



(a) L1 Data Cache Miss Rate (b) Number of Stalls

Fig. 13. Detailed metrics of remapping overhead. All numbers are normalized by the execution without remapping.

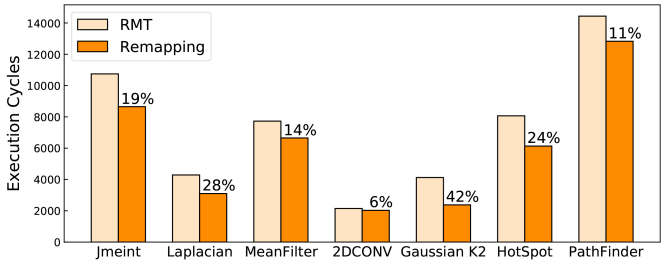


Fig. 14. Comparison of execution performance using duplication for error detection between remapping and RMT.

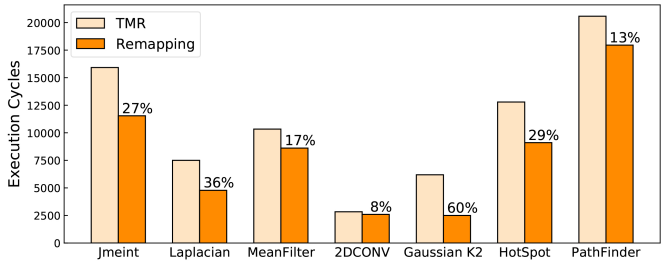


Fig. 15. Comparison of execution performance using triplication for error correction between remapping and TMR.

remains in the same ballpark as the original cases.

Last but not least, we show the performance savings of applying error detection/correction after remapping. In the case of error detection, we compare our technique with RMT (Redundant Multi-Threading), where all the threads (all warps) are duplicated for error detection. Figure 14 shows the execution performance of our remapping technique in execution cycles, comparing to RMT. We also present the percentage of saved execution cycles at the top of each application bar. On average, the percentage of saved execution cycles for error detection is 20.61%, while Gaussian K2 achieves a significant 42.39% savings.

In addition, we compare partial protection via remapping with TMR (Triple Modular Redundancy), results are shown in Figure 15. The average saving in terms of execution cycles is 27.15%, and again, Gaussian K2 has the highest savings of 60.02%. Generally performance results are similar for both error detection and correction, and the saving of error correction is always higher for every benchmark. This is expected, since partial protection using triplication avoids the execution of two copies for all reliable warps, while for error detection with duplication, we only save one copy execution of reliable warps. The per benchmark savings are related to the benchmark resilience profile, i.e., the percentage of reliable threads in each CTA. Since 95.87% of threads in Gaussian K2 is reliable, this benchmark achieves the highest savings.

Summary. We show the effectiveness of remapping by analyzing the percentage of reliable warps, which increases on average from 23.40% to 42.08% with remapping. Remapping introduces moderate to insignificant overhead. After applying remapping with protection, an average saving of 20.61% and 27.15% execution cycles for error detection (duplication of CTAs without remapping) and correction (triplication of CTAs without remapping), respectively.

VI. Related Work

Several works address reliability within the software engineering domain. Chiminey [42] provides a reliable platform for cloud computing. Bleser et al. [43] presents an automated approach to analyze the resilience of actor programs in distributed systems. Chan et al. [44] uses invariants to study error propagation in multi-threading applications using software fault injection. Yang et al. [45] uses a software fault injection tool to evaluate different anomaly detectors. However, none of these works are applied in the context of GPUs.

Redundancy-based solutions are used to protect GPGPU applications from errors. Such solutions rely on double execution [21], [23], [46] for error detection, called *dual-modular redundancy* (DMR) and triple execution [47], [48] for error correction, called *triple-modular redundancy* (TMR). Dimitrov et al. [46] first evaluate the overhead of introduced redundancy at kernel level, thread level, and instruction level and show that at all levels the overhead can be over 90%. Wadden et al. [21] take a deeper look at two different ways of applying redundant multithreading (RMT) at the granularity of CTAs (i.e., intergroup RMT and intragroup RMT) and present the trade-off between overhead and resilience coverage. Mahmoud et al. [23] choose instruction-level redundancy as it is transparent to programmers and propose SInRG, a collection of

several software and hardware optimizations, to further reduce overhead. In addition to those works targeting error detection, researchers also propose various solutions to correct errors with reduced overhead [47], [48] as compared to a naive implementation of TMR with triple overhead.

While the aforementioned solutions all focus on comparing and analyzing various redundancy-based protection solutions and seeking opportunities to reduce redundancy overhead, the "partial protection" methodology approaches this problem from a totally different perspective by focusing on reducing the portion of threads that require protection and on organizing the threads in such a manner that result in more reliable software.

VII. Conclusions

We presented a methodology to remap threads into warps according to their resilience profile. Looking into 12 benchmarks (17 kernels) from four benchmark suites, we identified that 7 of them are amenable to remapping for resilience. The proposed solution reduces overhead by identifying the portion of threads that are unreliable and by applying any detection/protection mechanism only on them instead of the entire kernel. In other words, our solution reduces overhead by identifying the portion of threads that are unreliable and by organizing them into warps that consist of threads that are either reliable or unreliable. Then, any detection/protection technique (including RMT and TMR) can be applied upon the identified unreliable warps only, instead of the entire kernel. Even with the simplest error detection and correction technique (warp duplication and triplication), we achieve an average saving of 20.61% and 27.15% execution cycles for error detection and error correction, respectively.

Data Availability

This paper is based on already available open-source benchmark data and existing fault injection tools. The proposed technique offers a methodology for re-organizing threads after evaluating thread resilience using the fault site pruning methodology [24]. Any resilience evaluation technique in the literature can be also used to derive thread resilience [18], [33], [36], [38] to guide remapping.

Acknowledgments

We thank the anonymous reviewers for their insightful comments. This material is based upon work supported by the National Science Foundation (NSF) grant (#1717532). This work was performed in part using computing facilities at William & Mary which were provided by contributions from NSF, the Commonwealth of Virginia Equipment Trust Fund, and the Office of Naval Research.

References

[1] V. Fratin, D. A. G. de Oliveira, C. B. Lunardi, F. Santos, G. Rodrigues, and P. Rech, "Code-dependent and architecture-dependent reliability behaviors," in *DSN*, pp. 13–26, 2018.

[2] S. Ganapathy, J. Kalamatianos, B. M. Beckmann, S. Raasch, and L. G. Szafaryn, "Killi: Runtime fault classification to deploy low voltage caches without MBIST," in *25th IEEE International Symposium on High Performance Computer Architecture, HPCA 2019, Washington, DC, USA, February 16-20, 2019*, pp. 304–316, IEEE, 2019.

[3] 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, no. 8, pp. 1073–1094, 2013.

[4] G. Pratx and L. Xing, "GPU computing in medical physics: A review," *Medical physics*, vol. 38, no. 5, pp. 2685–2697, 2011.

[5] S. S. Stone, J. P. Haldar, S. C. Tsao, W. mei W. Hwu, B. P. Sutton, and Z.-P. Liang, "Accelerating advanced MRI reconstructions on GPUs," *J. Parallel Distrib. Comput.*, vol. 68, no. 10, pp. 1307–1318, 2008.

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

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

[8] NVIDIA, "Computational finance."

[9] NVIDIA, "Researchers deploy GPUs to build world's largest artificial neural network."

[10] 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. IEEE International*, pp. 1–4, IEEE, 2008.

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

[12] 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 *25th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems, MASCOTS 2017, Banff, AB, Canada, September 20-22, 2017*, pp. 22–31, 2017.

[13] 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 *48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2018, Luxembourg City, Luxembourg, June 25-28, 2018*, pp. 95–106, 2018.

[14] B. Nie, A. Jog, and E. Smirni, "Characterizing accuracy-aware resilience of GPGPU applications," in *20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing, CCGRID 2020, Melbourne, Australia, May 11-14, 2020*, pp. 111–120, IEEE, 2020.

[15] "NVIDIA Fermi Architecture Whitepaper."

[16] "NVIDIA Kepler GK110 Architecture Whitepaper."

[17] "GP100 Pascal Whitepaper."

[18] 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.

[19] 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*, pp. 408–413, IEEE, 2009.

[20] 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, no. 2010, 2010.

[21] J. Wadden, A. Lyashevsky, S. Gurumurthi, V. Sridharan, and K. Skadron, "Real-world design and evaluation of compiler-managed GPU redundant multithreading," *ACM SIGARCH Computer Architecture News*, vol. 42, no. 3, pp. 73–84, 2014.

[22] M. Gupta, D. Lowell, J. Kalamatianos, S. Raasch, V. Sridharan, D. Tullsen, and R. Gupta, "Compiler techniques to reduce the synchronization overhead of gpu redundant multithreading," in *2017 54th ACM/EDAC/IEEE Design Automation Conference (DAC)*, pp. 1–6, IEEE, 2017.

[23] A. Mahmoud, S. K. S. Hari, M. B. Sullivan, T. Tsai, and S. W. Keckler, "Optimizing software-directed instruction replication for gpu error detection," in *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 842–853, IEEE, 2018.

[24] B. Nie, L. Yang, A. Jog, and E. Smirni, "Fault site pruning for practical reliability analysis of gpgpu applications," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 749–761, IEEE, 2018.

[25] L. Yang, B. Nie, A. Jog, and E. Smirni, "Sugar: Speeding up gpgpu application resilience estimation with input sizing," *Proc. ACM Meas. Anal. Comput. Syst.*, vol. 5, no. 1, pp. 1:1–1:29, 2021.

[26] G. Li and K. Pattabiraman, "Modeling input-dependent error propagation in programs," in *48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2018, Luxembourg City, Luxembourg, June 25-28, 2018*, pp. 279–290, IEEE Computer Society, 2018.

[27] A. Yazdanbakhsh, D. Mahajan, H. Esmaeilzadeh, and P. Lotfi-Kamran, "Axbench: A multiplatform benchmark suite for approximate computing," *IEEE Design & Test*, vol. 34, no. 2, pp. 60–68, 2016.

[28] "CUDA-GDB."

[29] 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*, pp. 1–10, IEEE, 2012.

[30] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *2009 IEEE International Symposium on Workload Characterization (IISWC)*, pp. 44–54, Ieee, 2009.

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

[32] 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*, vol. 48, pp. 395–406, ACM, 2013.

[33] 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*, pp. 221–230, IEEE, 2014.

[34] 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*, pp. 240–251, IEEE, 2016.

[35] 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 *47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2017, Denver, CO, USA, June 26-29, 2017*, pp. 97–108, IEEE Computer Society, 2017.

[36] S. Tselonis and D. Gizopoulos, "Gufi: A framework for gpus reliability assessment," in *Performance Analysis of Systems and Software (ISPASS), 2016 IEEE International Symposium on*, pp. 90–100, IEEE, 2016.

[37] 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*, pp. 163–174, IEEE, 2009.

[38] "Nvbitfi." https://github.com/NVlabs/nvbitfi.

[39] L. Yang, B. Nie, A. Jog, and E. Smirni, "Practical resilience analysis of gpgpu applications in the presence of single-and multi-bit faults," *IEEE Transactions on Computers*, vol. 70, no. 1, pp. 30–44, 2021.

[40] A. Vallero and S. Di Carlo, "Combining cluster sampling and ace analysis to improve fault-injection based reliability evaluation of gpu-based systems," in *2019 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*, pp. 8138–8143, IEEE, 2019.

[41] G. Li and K. Pattabiraman, "Modeling input-dependent error propagation in programs," in *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pp. 279–290, IEEE, 2018.

[42] I. I. Yusuf, I. E. Thomas, M. Spichkova, S. Androulakis, G. R. Meyer, D. W. Drumm, G. Opletal, S. P. Russo, A. M. Buckle, and H. W. Schmidt, "Chiminey: Reliable computing and data management platform in the cloud," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 2, pp. 677–680, IEEE, 2015.

[43] J. De Bleser, D. Di Nucci, and C. De Roover, "A delta-debugging approach to assessing the resilience of actor programs through run-time test perturbations," in *Proceedings of the IEEE/ACM 1st International Conference on Automation of Software Test*, pp. 21–30, 2020.

[44] A. Chan, S. Winter, H. Saissi, K. Pattabiraman, and N. Suri, "Ipa: Error propagation analysis of multi-threaded programs using likely invariants," in *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pp. 184–195, IEEE, 2017.

[45] Y. Yang, Y. Wu, K. Pattabiraman, L. Wang, and Y. Li, "How far have we come in detecting anomalies in distributed systems? an empirical study with a statement-level fault injection method," in *2020 IEEE 31st International Symposium on Software Reliability Engineering (ISSRE)*, pp. 59–69, IEEE, 2020.

[46] M. Dimitrov, M. Mantor, and H. Zhou, "Understanding software approaches for gpgpu reliability," in *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, pp. 94–104, ACM, 2009.

[47] A. Milluzzi and A. George, "Exploration of TMR fault masking with persistent threads on tegra gpu socs," in *2017 IEEE Aerospace Conference*, pp. 1–7, IEEE, 2017.

[48] J. Chen, S. Li, and Z. Chen, "GPU-ABFT: Optimizing algorithm-based fault tolerance for heterogeneous systems with GPUs," in *2016 IEEE International Conference on Networking, Architecture and Storage (NAS)*, pp. 1–2, IEEE, 2016.