SUGAR: Speeding Up GPGPU Application Resilience Estimation with Input Sizing

LISHAN YANG, William & Mary, USA
BIN NIE, William & Mary, USA
ADWAIT JOG, William & Mary, USA
EVGENIA SMIRNI, William & Mary, USA

As Graphics Processing Units (GPUs) are becoming a de facto solution for accelerating a wide range of applications, their reliable operation is becoming increasingly important. One of the major challenges in the domain of GPU reliability is to accurately measure GPGPU application error resilience. This challenge stems from the fact that a typical GPGPU application spawns a huge number of threads and then utilizes a large amount of potentially unreliable compute and memory resources available on the GPUs. As the number of possible fault locations can be in the billions, evaluating every fault and examining its effect on the application error resilience is impractical. Application resilience is evaluated via extensive fault injection campaigns based on sampling of an extensive fault site space. Typically, the larger the input of the GPGPU application, the longer the experimental campaign.

In this work, we devise a methodology, SUGAR (Speeding Up GPGPU Application Resilience Estimation with input sizing), that dramatically speeds up the evaluation of GPGPU application error resilience by judicious input sizing. We show how analyzing a small fraction of the input is sufficient to estimate the application resilience with high accuracy and dramatically reduce the duration of experimentation. Key of our estimation methodology is the discovery of repeating patterns as a function of the input size. Using the well-established fact that error resilience in GPGPU applications is mostly determined by the dynamic instruction count at the thread level, we identify the patterns that allow us to accurately predict application error resilience for arbitrarily large inputs. For the cases that we examine in this paper, this new resilience estimation mechanism provides significant speedups (up to 1336 times) and 97.0 on the average, while keeping estimation errors to less than 1%.

CCS Concepts: • **Hardware → Transient errors and upsets**; • **Computer systems organization → Reliability**; **Single instruction, multiple data**; **Multicore architectures**; • **Software and its engineering → Software reliability**.

Additional Key Words and Phrases: GPGPUs; GPGPU application resilience; Input-aware application resilience

ACM Reference Format:
Lishan Yang, Bin Nie, Adwait Jog, and Evgenia Smirni. 2021. SUGAR: Speeding Up GPGPU Application Resilience Estimation with Input Sizing. *Proc. ACM Meas. Anal. Comput. Syst.* 5, 1, Article 1 (March 2021), 29 pages. https://doi.org/10.1145/3447375

Authors' addresses: Lishan Yang, William & Mary, Williamsburg, VA, USA, lyang11@email.wm.edu; Bin Nie, William & Mary, Williamsburg, VA, USA, bnie@email.wm.edu; Adwait Jog, William & Mary, Williamsburg, VA, USA, ajog@wm.edu; Evgenia Smirni, William & Mary, Williamsburg, VA, USA, esmirni@cs.wm.edu.

2476-1249/2021/3-ART1 $15.00
https://doi.org/10.1145/3447375

1 INTRODUCTION

Graphics Processing Units (GPUs) are widely used for accelerating applications from domains such as high performance computing (HPC) [13, 15, 37, 38, 40, 41, 45, 48], deep learning [10], virtual/augmented reality, autonomous vehicles [7] and network functions [17]. As GPUs are gaining popularity from data centers to mobile systems to self-driving cars, it is increasingly important to develop tools and techniques to evaluate GPU application resilience, especially since GPUs (as well as custom ML processors) are susceptible to transient faults [16, 22, 29, 33–35].

A typical approach to evaluate application resilience is to conduct a systematic fault injection campaign. An architectural register is selected, a bit is flipped, and the output of the application is compared to the correct application output without a bit flip [32]. For GPGPU applications, fault injection campaigns are typically based on statistical sampling since the entire fault site space is in the order of billions [36]. Typically, 1000 experiments (one per different fault site, each randomly selected) are done to obtain results with 95% confidence intervals and ±3% error margins for a view of the application resilience. More accurate results (99.8% confidence intervals and ±0.63% error margins) are given with 60,000 experiments. No matter the case, these numbers are daunting: for inputs that typically represent actual applications, a single execution of the application can take from several minutes to days, such time is further exacerbated if architecture simulators are used [21, 52], effectively rendering the above approaches not practical as 1000 and 60,000 runs are simply not possible. For the above reasons, existing methods are practical for small inputs only, even if with state-of-the-art fault site pruning techniques [36, 54] that further reduce the sampling space to even a few hundred sites while maintaining the same accuracy as with 60,000 runs.

In this paper, we propose a solution to the above problem: analyze the GPGPU application resilience with the *smallest possible input subset* that the application can admit and accurately project application resilience for a target larger input of practical significance. It is important to untangle here two different but related concepts: *input type* and *input size*. Input type refers to the actual input that the application admits: e.g., integers or floats and their range of values. Input size refers to the actual number of input elements.

We introduce the concept of *asymptotic application resilience*: we show that resilience patterns within the coordinated thread arrays (CTAs), the building blocks of GPGPU parallelism, can project application resilience for larger inputs of arbitrary size. For asymptotically larger inputs of the same type, the number of CTAs naturally increases but follows distinct patterns as dictated by the parallelization logic. CTA pattern identification is key to calculate application resilience of larger inputs of the same input type.

In this paper, we propose SUGAR, a new technique for Speeding Up GPGPU Application Resilience Estimation with input sizing. SUGAR is based on the well-established fact [14, 36] that GPGPU application error resilience is determined by the thread dynamic instructions (DI) count. Since GPU organizes threads in a hierarchical way by grouping threads in an organized manner, we leverage this thread grouping feature to identify CTA patterns.

The *first key insight* is that as the input size scales up, CTA patterns remain the same, and only the *number of patterns* changes. We therefore extrapolate application resilience using the observed CTA patterns that evolve as a function of the input size and show how to accurately project application resilience from the *smallest possible* input to *any arbitrary larger input of the same type*. The smallest possible input is defined by the application itself: its size is dictated by having enough parallelism to fill a CTA. Therefore, as long as the larger input does not change the sequence of dynamic instructions, the application error resilience can be predicted remarkably accurately, with an average error less than 1% for arbitrarily larger inputs. For applications where larger inputs affect branch outcomes (and consequently affect the number of instructions), we

require only one additional dynamic instruction profiling which is achieved in just an additional run to estimate error resilience.

The *second key insight* is that application resilience depends heavily on the application input. We identify application input by its *type* as determined by the range of its possible values and its *size* as determined by the number of its elements. Given a certain input type, application resilience can be determined for *any* input size of the same type. We identify three trends of application resilience as a function of the input size: it can improve, decrease, or remain flat, and this depends on the resilience of the dominant CTA pattern of the benchmark. For the cases that the resilience trend is not flat, as input size increases (but input type remains the same), application resilience reaches an asymptotic state that remains unchanged for arbitrarily larger inputs.

In summary, we make the following research contributions:

- We perform a deep analysis of different inputs to understand the impact of input type and input size on GPGPU application error resilience.
- We show that resilience can be easily predicted for applications with inputs that do not change the dynamic instruction count. For other applications, a low overhead profiling with the larger input (essentially just one additional run) is sufficient for accurate resilience estimation.
- We identify the smallest input size (given a certain input type) that profiles application characteristics that can be used to accurately project application asymptotic resilience for arbitrarily large input sizes of the same type.

Overall, our new input-aware resilience estimation mechanism for the two large inputs that we examine here can reduce the overall resilience evaluation time by achieving speedups from 1.2 to 1335. Collectively, for the cases considered here, the reduction of overall evaluation time for medium input sizes is 7.3 and 186.6 for large sizes, while being remarkably accurate as it consistently introduces errors of no more than 1%.

We stress that the above savings/speedups are conservative estimates as they are tied to the specific input sizes for prediction that we consider here paper and commensurate to the size of medium and large inputs used. Larger inputs would invariantly result in higher speedups.

2 BACKGROUND

We give a brief introduction to the baseline GPU architecture and GPGPU execution model. Then, we introduce the fault model, fault injection methods, and benchmarks used.

2.1 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 [4]. Each core has its private L1 cache, software-managed scratchpad memory, and a large register file. An interconnection network connects all cores to global memory, which consists of various memory channels. Every memory channel has a shared L2 cache, and its associated memory requests are handled by a GDDR5 memory controller.

Since GPUs are susceptible to transient faults from high-energy particle strikes [16, 33, 34]. protection techniques are omnipresent in recent commercial GPUs [2, 4, 5]. Such techniques include single-error-correction double-error-detection (SEC-DED) error correction codes (ECCs) to protect register files, L1/L2 caches, shared memory and DRAM against soft errors, and using parity to protect the read-only data cache. Other structures such as arithmetic logic units (ALUs), thread schedulers, instruction dispatch units, and the interconnect network are not protected [2, 4, 5].

GPGPU Execution Model. Following the single-instruction-multiple-thread (SIMT) philosophy [25], GPGPU applications execute thousands of threads concurrently, over large amounts of data. A typical GPGPU application hierarchy is as follows: the application launches kernels on the GPU. Each kernel is divided into groups of threads, known as *thread blocks*, also called *Cooperative Thread Arrays* (CTAs) in NVIDIA terminology. A CTA encapsulates all synchronization and barrier primitives among a group of threads [23, 25]. The CTA formation enables the GPU hardware to relax the execution order of the CTAs, for the purpose of maximizing parallelism. CTAs can be organized in 1-dimension, 2-dimensions, or 3-dimensions, depending on how application programmers organize data and algorithm development. Threads inside one CTA can be further divided into groups of 32 individual threads, known as warps. Warps execute a single instruction on the functional units in lock step. This sub-division of warps is an architectural abstraction, which is transparent to the application programmer.

2.2 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 therefore consider here only commonly occurring computation-related errors due to transient single-bit faults (known also as soft errors) in ALUs/LSUs, i.e., in components that cannot be protected by ECC. These faults can lead to wrong ALU output which would then be stored in destination registers. This erroneous computing operation is what we emulate by injecting a single fault directly to destination register values. This is standard experimental methodology for GPGPU reliability studies [14, 20, 27, 36, 43].

The fault injection methodology used here closely follows the one used in GUFI [49]: flip a bit at a destination register identified by the thread id, the instruction id, and a bit position. We use the single-bit fault model, considering more than one bit flips within a single execution is out of scope of this work.

We perform reliability evaluations on GPGPU-Sim [6] with PTXPlus mode as other works that estimate GPGPU reliability [36]. 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 [6, 49]. Although we use GPGPU-Sim [6] in this work to evaluate our methodology, SUGAR does not depend on this architecture simulator. SUGAR can be used with fault injectors that operate on real hardware, e.g., SASSIFI [20] or NVBitFI [3] for a small input, and estimate application resilience for larger input sizes.

For each fault injection experiment, there are three possible outcomes:

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

To obtain the reliability profile of an application run given an input, we conduct an experimental campaign using the state-of-the-art fault site pruning methodology proposed by Nie et al. [36]. We aggregate the outcome of fault injection experiments as **masked**, **SDC**, and **other** to obtain the application *resilience profile*. Note that this methodology, as others in the literature [14, 20, 49, 53], provides the application resilience profile for the given input only. *To the best of our knowledge, this is the first study to accurately estimate input-dependent GPGPU application resilience.*

2.3 Benchmarks and Inputs

We select applications from several commonly-used benchmark suites (i.e., CUDA [39], Polybench [18], Rodinia [9] and AxBench [55]). Note that, as kernels of GPGPU applications normally

Table 1. Selected Benchmarks and Inputs.

Suite	Benchmark	Kernel Name	Kernel ID	Input Size (#Elements)	#Threads	#Fault Sites
CUDA	BlackScholes	BlackScholesGPU	K1	{384, 12288, 98304}	{61440, 61440, 61440}	{7.49E+06, 1.89E+07, 1.01E+08}
	NN	executeFirstLayer	K1	{784, 25088, 100352}	{1014, 32448, 129792}	{1.05E+07, 3.37E+08, 1.35E+09}
		executeFourthLayer	K4	{784, 25088, 100352}	{10, 320, 1280}	{3.04E+05, 9.74E+06, 3.90E+07}
Polybench	2DCONV	Convolution2D_kernel	K1	{1024, 65536, 4194304}	{1024, 65536, 4194304}	{1.90E+06, 1.34E+08, 8.71E+09}
	GEMM	gemm_kernel	K1	{192, 49152, 196608}	{768, 16384, 65536}	{7.91E+07, 6.23E+08, 4.94E+09}
	MVT	mvt_kernel1	K1	{65792, 1049600, 16781312}	{256, 1024, 4096}	{1.71E+07, 2.73E+08, 4.36E+09}
	SYRK	syrk_kernel	K1	{256, 4096, 65536}	{256, 4096, 65536}	{1.02E+07, 7.93E+07, 4.94E+09}
Rodinia	BFS	Kernel	K7	{4096, 65536, 1000448}	{4096, 65536, 1000448}	{1.99E+06, 1.18E+07, 1.60E+08}
		Kernel2	K8	{4096, 65536, 1000448}	{4096, 65536, 1000448}	{8.88E+05, 1.08E+07, 1.60E+08}
	Gaussian	Fan1	K1	{1088, 4224, 65536}	{512, 512, 512}	{1.35E+05, 2.19E+05, 3.30E+05}
		Fan2	K2	{1088, 4224, 65536}	{2048, 32768, 131072}	{1.57E+06, 2.54E+07, 1.02E+08}
	HotSpot	calculate_temp	K1	{8192, 18432, 131072}	{9216, 30976, 123904}	{3.43E+07, 8.43E+07, 3.37E+08}
	K-Means	invert_mapping	K1	{17408, 34816, 69632}	{1024, 1024, 2304}	{2.44E+07, 4.83E+07, 9.67E+07}
		kmeansPoint	K2	{17408, 34816, 69632}	{2304, 4096, 6400}	{3.73E+06, 7.34E+06, 1.47E+07}
	PathFinder	dynproc_kernel	K1	{64800, 410400, 3283200}	{768, 4864, 38912}	{1.63E+07, 1.15E+08, 9.23E+08}
			K5	{64800, 410400, 3283200}	{768, 4864, 38912}	{1.55E+07, 1.09E+08, 8.77E+08}
AxBench	Jmeint	Jmeint_kernel	K1	{18432, 184320, 1843200}	{4096, 8192, 16384}	{6.87E+06, 6.89E+07, 6.93E+08}

implement independent modules/functions, resilience analysis for each kernel is performed separately. We focus on every static kernel in a benchmark. For static kernels with more than one dynamic invocations, we randomly select one invocation for the fault injection experiments.

Table 1 outlines the 11 benchmarks (15 kernels) studied here. In the rest of this paper, if the kernel index is not specified, it implies that the benchmark contains one kernel only. Three input sizes per benchmark are examined: small (S), medium (M), large (L), see the fifth column of Table 1 for the exact sizes (in terms of the number of input elements) used per benchmark. Typically, a small input is obtained as a subset of the larger target input such that there is enough thread parallelism to fill one CTA.

The sixth and seventh columns correspond to the number of threads and the exhaustive number of fault sites (provided that single-bit faults can occur in ALU/LSU units only, where no ECC is available) for small, medium, and large inputs, respectively. The number of threads and fault sites[1] are indicative of the tremendous complexity of the problem for these benchmark/input combinations. Note that from a small input to medium and large, the number of exhaustive fault sites increases by orders of magnitude (from millions to billions), highlighting the magnitude of the difficulty of the problem. Considering that in practice application resilience needs to be re-evaluated for every different *input type* and *size*, providing a methodology to accurately estimate application reliability from small to arbitrarily large inputs of the same type can result in significant savings.

Below we give an overview of the input selection of benchmarks. The ranges of input values are typically given by the benchmarks themselves.

BlackScholes: This benchmark simulates the Black-Scholes model that captures the dynamics of a financial market. The input consists of three parts: 1) *stock price* is provided by random float numbers generated under a $[5, 30]$ Uniform distribution; 2) *exercise price*, consists of random float numbers generated with a $[1, 100]$ Uniform distribution; 3) *time*, that consists of random float numbers generated with a $[0.25, 10]$ Uniform distribution. Here, the smallest input size to fill at least one CTA is 384 (total number of elements). Input sizes of 12288 and 98304 elements correspond to medium and large sizes, respectively.

2DCONV: 2DCONV performs a 2-Dimensional convolution on a matrix of Uniformly-distributed floating-point numbers between 0 and 1. The smallest input size is 1024, medium and large inputs are 65536 and 4194304 elements.

[1]The exhaustive fault sites have been calculated using the methodology in [36].

GEMM: GEMM does general matrix-matrix multiplication and accepts as input three matrices of floating-point elements sampled from Uniform $[0, 1]$. Inputs have 12288, 49152, and 196608 elements.
MVT: MVT performs matrix-vector multiplication. The elements of the matrix and vector are floating-point and sampled from a Uniform distribution ranging from 0 to 10. Three inputs of size 65792, 1049600, and 16781312 are generated.
SYRK: SYRK performs symmetric rank-k operations. Three inputs of 1024, 4096, and 65536 floating-point elements are generated using a $[0, 1]$ Uniform distribution.
BFS: The input of this benchmark is a graph with vertices and edges. The original benchmark provides 3 graphs with the number of vertices equal to {4096, 65536, 1000448}.
Gaussian: Gaussian solves systems of equations using Gaussian elimination. The system of equations to be solved is represented in a matrix format. The matrix is populated by floating-point numbers generated by a [10000, 10001] Uniform distribution. The three inputs have matrices of 1088, 4224, and 65536 elements and all cases are solvable.
HotSpot: HotSpot simulates the temperature on a chip. This benchmark uses two input files in a matrix form: temperature and power. In contrast to the inputs of the benchmarks already described, HotSpot's inputs are not generated by any distribution. The benchmark provides a script to scale up the inputs. We scale up both temperature and power matrices from input size 8192 to 18432 and 131072. The range of temperature is $(322, 345)$ and of power is $(0, 0.002823]$.
K-Means: k-means (KMN) implements a clustering algorithm. We use the script provided by the benchmark to generate different integers ranging from 0 to 255. The generated integers are organized in a matrix format with 17920, 35840, and 71680 elements.
PathFinder: PathFinder does dynamic programming on a 2D grid to find the shortest weighted path. The 2D grid is represented by path weights that are uniformly distributed in $[0, 1]$. The weights are stored in a matrix format. Three matrices of 51200, 409600, and 3276800 elements are used.
Jmeint: Jmeint performs triangle intersection detection, which is widely used in 3D gaming. The input is pairs of 3D triangle coordinates, which are all Equilikely random variates (integers) ranging from -50 to 50. We use three input sizes here of 18432, 184320, and 1483200 integers.
NN: NN implements the testing phase of a pre-trained four-layer neural network model. The input is images provided by CUDA benchmark suite. The number of images used in input S, M, and L is 1, 32, and 128, corresponding to 784, 25088, and 100352 input elements.

3 INPUT-WISE RESILIENCE

We present a detailed characterization of the effect of input size on the application dynamic instruction (DI) count. We first show that error resilience is primarily determined by the thread DI count, which is consistent with other works [14, 36]. We link thread (and consequently kernel) resilience under different inputs by examining how the thread DI count and CTA patterns are affected by the input size. We then show how to perform fast resilience estimation by identifying resilience patterns at the CTA level. We stress that we have done experiments with *many* inputs for each of the benchmarks in Table 1 but in the exposition below we present studies of specific cases that are indicative examples of the identified resilience patterns.

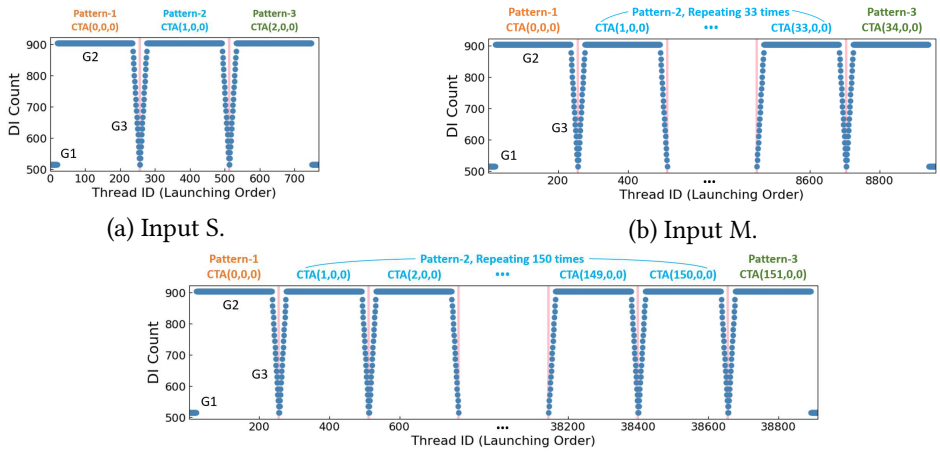
3.1 DI-insensitive benchmarks

We first focus on applications where the DI count is not sensitive to input size. Following the CTA structure in GPU applications, we start with analyzing the DI behavior of the application and classify DI-insensitive benchmarks according to their CTA organization as 1-Dimensional or 2-Dimensional.

3.1.1 *Resilience Patterns with 1-Dimensional (1D) Structure.*

DI Analysis. We use PathFinder K1 to show the effect of input size on DI count and resilience patterns. We collect the DI profile using small, medium, and large inputs. This analysis helps classify each thread by its DI count. Here, we plot the DI count as the function of thread id, see Figure 1.

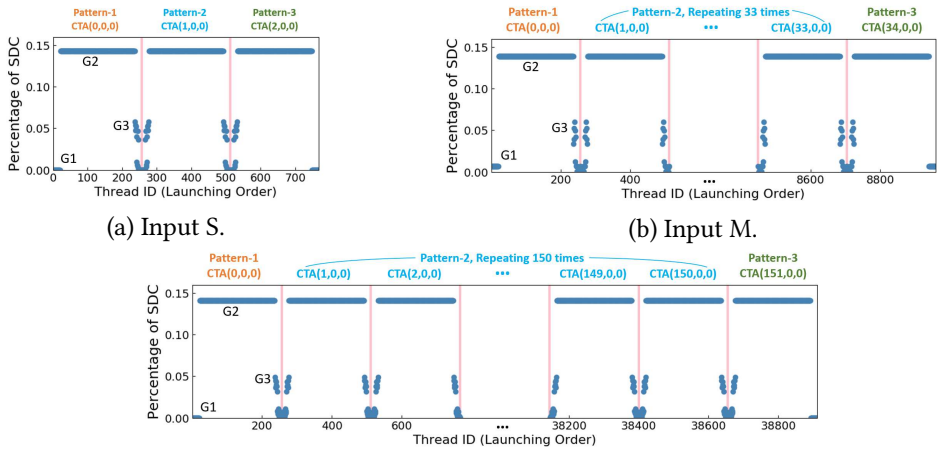
Comparing threads in CTA(0,0,0) across the three inputs, it is clear that threads with the same id share the same DI count. We mark this as Pattern-1. Across the three inputs, we observe a repeating pattern, marked as Pattern-2, which occurs 1, 33, and 150 times for inputs S, M, and L, respectively.



(a) Input S. (b) Input M.

(c) Input L.

Fig. 1. DI count scatter plots for S, M, L inputs of PathFinder K1. Three distinct CTA patterns are clearly discerned. There is only one Pattern-1 and one Pattern-3 in each of the inputs, while Pattern-2 occurs 1, 33, and 150 times for input S, M, and L, respectively. PathFinder K1 is identified as a DI-insensitive benchmark.



(a) Input S. (b) Input M.

(c) Input L.

Fig. 2. Thread resilience (in terms of SDC outputs, y-axis) for S, M, and L inputs of PathFinder K1. CTA resilience patterns follow the thread DI counts in Figure 1.

The last CTA is marked as Pattern-3. Because of the above regularities, PathFinder K1 is categorized as DI-insensitive.

Resilience Analysis. We group threads according to DI count, and perform fault injection experiments for every thread group to obtain results of statistical significance according to the methodology presented in [36]. We aggregate the fault injection results to get the resilience of thread groups, i.e., distribution of masked, SDC, and other outputs. Figure 2 shows thread resilience in terms of the percentage of SDC outputs which are anyway the most perilous being silent (similar results exist for masked and other outputs but are not shown here due to lack of space). Comparing the DI count scatter plot in Figure 1(a) and the resilience pattern in Figure 2(a) for input S, there are stark similarities: threads with the same DI count have overwhelmingly similar resilience. In addition, the one-to-one relationship between the thread DI count and resilience pattern discovered for input S persists for inputs M and L. Within CTA(0,0,0) in Figure 2(a)-(c), we can clearly discern 3 groups of threads with different resilience behaviors:

- G1: Threads 1 – 20 and 256 are the most resilient threads, with close to 0% SDC outputs;
- G2: Threads 21 – 236 have SDC outputs near 15%;
- G3: Threads 237 – 255 form the last group.

Here, we opt to cluster threads 237 – 255 into a single group (G3) despite the fact that they have a different DI-count for ease of presentation. In fact, each thread from the G3 group is treated separately when their resilience profile is used for kernel estimation.

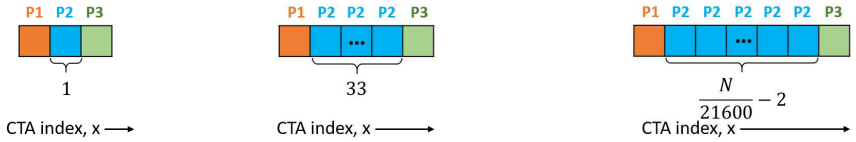
The resilience of each thread group is similar for different inputs. Most of PathFinder K1 threads belong to G2, the dominant thread group in this kernel, this is consistent across inputs S, M, and L. Note that for the G3 cluster, thread resilience is slightly different for inputs M and L comparing to input S, see Figure 2(a)-(c). Since our purpose is to estimate the resilience of a large input from a smaller subset (from input S, in our case), this slight difference will unavoidably contribute to an error in estimations.

In general, we conclude that it is sufficient to use DI count as a proxy for thread resilience. While *individual* thread resilience does not change from one input size to a larger input (or may change marginally, as in the G3 case above), the percentage of each thread group in the kernel may be different. For example, the percentage of G1 for input S, M, and L is 5.7%, 1.2%, and 0.9%, respectively, and for G2 the percentage is always 84.4%. This difference may result in different kernel resilience for different input sizes. We conclude that it is possible to infer *individual* thread resilience from one input to a larger input, as changing the input for DI-insensitive benchmarks does not have a significant effect on *individual* thread resilience.

The above patterns can be presented visually at the CTA level, see Figure 3. In this figure, every cell is a CTA. Each unique color represents a unique CTA pattern. In this figure, P1, P2, and P3 correspond to Pattern-1, Pattern-2, and Pattern-3, respectively in Figure 1 and Figure 2. Figure 3 visualizes the 1D structure in the CTA pattern across different input sizes. The pattern repetitions can be calculated according to input size. In fact, CTA patterns can be easily identified by looking at the source code.

Code and Input Analysis. To further explain the structure of the observed patterns, we turn into the source code. A code snippet of PathFinder K1 is shown in Figure 4. PathFinder leverages dynamic programming to find the shortest weighted path given a certain input. The input data is broken into chunks, and every chunk of data is assigned to a CTA. Although every CTA has the same amount of data to process, from the chunk start position *blkX* (Line 4) till the end position

[2]Source code Line 3 in Figure 4 computes $small\_block\_cols$. Given $BLOCK\_SIZE = 256$, $iteration = 10$, and $HALO = 2$, we can get $small\_block\_cols = 216$. In addition, the number of rows in the input data is 100, so each CTA processes 21600 elements.



(a) Input S. (b) Input M. (c) Input size=N.

Fig. 3. CTA patterns for different inputs for PathFinder K1.

```
1  int bx = blockIdx.x;
2  int tx = threadIdx.x;
3  int small_block_cols = BLOCK_SIZE-iteration*HALO*2;  // Data chunk size
4  int blkX = small_block_cols*bx-border;  // Chunk start position
5  int blkXmax = blkX+BLOCK_SIZE-1;  // Chunk end position
6  int xidx = blkX+tx;  // Global thread ID
7  int validXmin = (blkX < 0) ? -blkX : 0;  // Valid chunk start position
8  int validXmax = (blkXmax > cols-1) ? \
9      BLOCK_SIZE-1-(blkXmax-cols+1) : BLOCK_SIZE-1;  // Valid chunk end position
10 ......
11 bool isValid = IN_RANGE(tx, validXmin, validXmax);  // Checking Validation
12 ......
13 for (int i=0; i<iteration ; i++){
14   ......
15   if( IN_RANGE(tx, i+1, BLOCK_SIZE-i-2) && isValid){
16     // If valid, take the branch.
17     ......
18     }
19 }
```

Fig. 4. Code snippet of PathFinder K1. Input has no effect on branch divergence, hence CTA resilience patterns across inputs of different size persist.

blkXmax (Line 5), this range may cover some invalid data at the boundary. Therefore, a boundary check is performed to check whether the thread is inside the valid range [$validXmin, validXmax$] (Line 11). A loop starts at Line 13, and its condition check depends on the number of iterations, which is a pre-defined parameter for each kernel. Only threads whose thread id is inside the valid range execute the *if* condition in Line 15 and process the data. In both cases, the input value does not affect branch divergence. Consequently, the instruction execution context persists across different inputs. Therefore, even if input changes, resilience patterns remain similar, and the benchmark is identified as DI-insensitive.

Table 2. Parameters of different CTAs from PathFinder K1 with input M. Some threads from the boundary CTAs are filtered due to boundary checking, resulting in different CTA resilience patterns.

Pattern	CTA	start_pos	end_pos	validXmin	validXmax
P1	(0,0,0)	-20	235	20	255
P2	(1,0,0)	196	451	0	255
P2	(2,0,0)	412	667	0	255
...					
P2	(33,0,0)	7108	7363	0	255
P3	(34,0,0)	7324	7579	0	235

Following the code snippet shown in Figure 4, we calculate the start and end positions of each CTA's data chunk[2](see Line 3-5), as well as the valid chunk intervals (see Lines 7-9) using the default CTA size 256. An example using input M is shown in Table 2. Note that the CTAs with

Table 3. Benchmarks with 1-Dimensional CTA structure

Benchmark	Pattern	Number of CTAs	Dominant Pattern
BlackScholes	P1	$\lceil \frac{input\_size}{num\_thd\_per\_cta} \rceil$	✓
	P2	$512 - \lceil \frac{input\_size}{num\_thd\_per\_cta} \rceil$	
MVT	P1	$\lceil \frac{\lfloor \sqrt{input\_size} \rfloor}{num\_thd\_per\_cta} \rceil$	✓
Gaussian K1	P1	$\lfloor \frac{\lfloor \sqrt{input\_size} \rfloor}{num\_thd\_per\_cta} - 1 \rfloor$	✓
K-Means K1	P1	$\frac{input\_size/34}{num\_thd\_per\_cta}$	✓
	P2	$(\lceil \sqrt{\frac{input\_size/34}{num\_thd\_per\_cta}} \rceil)^2 - \frac{input\_size/34}{num\_thd\_per\_cta}$	
K-Means K2	P1	$\frac{input\_size/34}{num\_thd\_per\_cta}$	✓
	P2	$(\lceil \sqrt{\frac{input\_size/34}{num\_thd\_per\_cta}} \rceil)^2 - \frac{input\_size/34}{num\_thd\_per\_cta}$	
PathFinder K1[2]	P1	1	
	P2	$\lceil \frac{input\_size}{21600} \rceil - 2$	✓
	P3	1	
PathFinder K5[2]	P1	1	
	P2	$\lceil \frac{input\_size}{21600} \rceil - 2$	✓
	P3	1	

pattern P2 have the same $[validXmin, validXmax]$ intervals, because they are in the middle part of the data. The $validXmin$ of the starting CTA (0,0,0) with pattern P1 is different from the others, and the $validXmax$ of the last CTA (34,0,0) with pattern P3 is also different. This validation check filters some of the threads at the boundary from the first CTA and the last CTA, leading to a different control flow path that results in a different DI set (thus, different DI count). This is the reason that these two boundary CTAs exhibit different resilience patterns. Since these filtered threads do not touch any input data, they are very resilient, with close to 0% SDC outputs shown in Figure 2 (G1 group).

Based on the source code, the number of each repeating CTA pattern can be calculated. Pattern P1 and P3 are the boundary CTAs (one CTA). For pattern P2, the number of CTAs is given by[2]:

$$num\_repeating = \lceil \frac{input\_size}{21600} \rceil - 2 \tag{1}$$

Based on Equation 1, the CTA pattern shown in Figure 3(c) and the detailed resilience pattern shown in Figure 2, we can calculate the number of CTAs within each pattern and then extrapolate their resilience for a larger input.

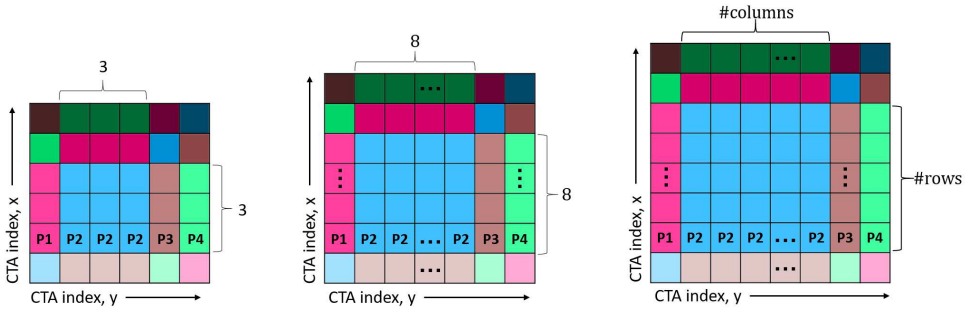
Across all benchmarks in this study, if branch divergence is not affected by input values, then thread DI count does not change with larger inputs and the CTA resilience pattern persists. Similar benchmarks with 1-Dimension patterns include BlackScholes, MVT, Gaussian K1, K-Means K1 and K2, and PathFinder K1 and K5. For the above benchmarks, we followed the same steps as those depicted above for Pathfinder K1 and determined their CTA patterns. A summary of salient features of CTA resilience patterns for the above benchmarks is shown in Table 3, and the CTA patterns using the input sizes used in this paper can be found in Appendix A, Table 7. Note that different benchmarks have different CTA resilience patterns (e.g., P1 in BlackScholes is different from P1 in MVT). We also identify the dominant pattern of each kernel, which can be used to estimate the asymptotic resilience, see Section 4.1.

Summary: DI-insensitive benchmarks exhibit repeating CTA resilience patterns across inputs of different size. Benchmark resilience can be derived using the 1-Dimensional structure of CTA patterns, which is identified using code inspection.

3.1.2 Resilience Patterns with a 2-Dimensional (2D) Structure.

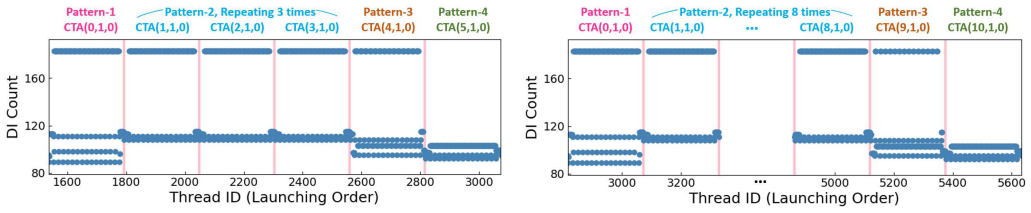
We now focus on benchmarks with CTAs organized in 2-Dimensions and use HotSpot as an example to illustrate our findings. HotSpot computes the temperature on a single chip. CTAs are grouped across both x and y dimensions, see Figure 5. For different inputs, the CTA patterns are similar. The boundary check of HotSpot is more complicated compared to PathFinder K1, resulting in 16 different CTA patterns within a single kernel. For input S shown in Figure 5(a), the blue cells at the center occur 3×3 times. We revisit later in the section how this repeating pattern evolves as a function of input size.

DI and Resilience Analysis. The scatter plots of DI count and thread resilience are given in Figure 6 and Figure 7, respectively. Due to space limitations, we only show here a part of the pattern, CTAs from the second bottom row in Figure 5(a). There are 6 CTAs at the second bottom row for input S, with 4 different CTA patterns, repeating 1,3,1,1 times, respectively. Comparing input S and another larger input shown in Figure 5(b), Pattern-1 is exactly the same across the two inputs. Similar observations can also be made for Pattern-2, Pattern-3, and Pattern-4. The only difference in Figures 7(a) and (b) is the number of repetitions for Pattern-2: 3 for input S and 8 for the larger input M. Clear similarities emerge when comparing the DI count in Figure 6 and the resilience patterns in Figure 7.



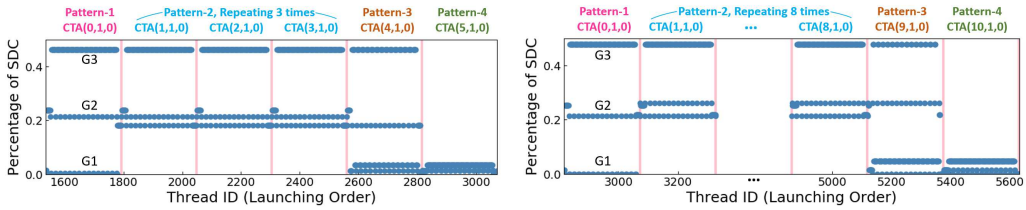
(a) Input S. (b) Input M. (c) Larger input.

Fig. 5. HotSpot CTA structure. The number of columns and rows the center can be calculated using Equations 2 and 3.



(a) Input S. (b) Input M.

Fig. 6. DI count scatter plot for inputs S and M of HotSpot for the CTAs in the bottom row of Figure 5. DI counts are consistent across CTA patterns for S and M.



(a) Input S. (b) Input M.

Fig. 7. Thread resilience for inputs S and M of HotSpot. CTA resilience patterns across the two inputs are similar and follow the DI count trends of Figure 6.

```
1  int bx = blockIdx.x;
2  int by = blockIdx.y;
3  int tx=threadIdx.x;
4  int ty=threadIdx.y;
5  ......
6  // data chunk size
7  int small_block_rows = BLOCK_SIZE-iteration*2;
8  int small_block_cols = BLOCK_SIZE-iteration*2;
9  // chunk boundary
10 int blkY = small_block_rows*by-border_rows;
11 int blkX = small_block_cols*bx-border_cols;
12 int blkYmax = blkY+BLOCK_SIZE-1;
13 int blkXmax = blkX+BLOCK_SIZE-1;
14 // calculate the global thread coordination
15 int yidx = blkY+ty;
16 int xidx = blkX+tx;
```

```
17 // load data if it is within the valid range
18 int loadYidx=yidx, loadXidx=xidx;
19 int index = grid_cols*loadYidx+loadXidx;
20 if( IN_RANGE(loadYidx, 0, grid_rows-1) &&
21     IN_RANGE(loadXidx, 0, grid_cols-1)){
22     temp_on_cuda[ty][tx] = temp_src[index];
23     power_on_cuda[ty][tx] = power[index];
24 }
25 ......
26 for (int i=0; i<iteration ; i++){
27     ......
28     // only perform compation if within the valid range
29     if( IN_RANGE(tx, i+1, BLOCK_SIZE-i-2) &&     \
30         IN_RANGE(ty, i+1, BLOCK_SIZE-i-2) &&     \
31         IN_RANGE(tx, validXmin, validXmax) && \
32         IN_RANGE(ty, validYmin, validYmax) ){
33             ......
34     }
35 }
```

Fig. 8. Code snippet of HotSpot. Branch divergence only depends on CTA ID and thread ID, but not input data itself, thus HotSpot is DI-insensitive.

We use the second row from the bottom in Figures 5(a) and 5(b) to show the results of fault injection experiments on different threads. In CTA(0,1,0) there are in total 3 different resilience groups of threads, also marked as G1–G3 in Figure 7:

- G1: threads at the bottom, with around 0% SDCs;
- G2: threads in the middle, with about 20% SDCs;
- G3: threads at the top, with more than 40% SDCs.

The grouping in G1, G2, and G3 is done to facilitate the discussion by providing a higher-level overview of their resilience profile. For each thread group, the distributions of masked, SDC, and other outputs for different inputs remain remarkably close. We conclude that we can directly use the resilience of the smallest input to infer the resilience of larger ones as the dominant thread group, G3, has the most SDC outputs. Note that this is also repeating in CTA(1,1,0), CTA(2,1,0), CTA(3,1,0), CTA(4,1,0), and CTA(5,1,0). The percentage of G3 increases as input size gets larger (as the repeating times of the middle CTA pattern increase), which indicates that HotSpot becomes less resilient when input size increases. We also mark the repeating CTA patterns, P1 (Pattern-1) to P4 (Pattern-4) in Figure 5. Note that for HotSpot the specific input values for sizes S and M are not exactly the same (although they are derived from the same distribution), this is the reason that there are some changes in the SDC values from S to M. Since the S values are used to estimate the resilience for input M, this discrepancy leads to a small estimation error for the larger input.

Code and Input Analysis. We turn to code analysis to understand why the resilience profile of the above groups persists across different inputs. A code snippet of HotSpot is shown in Figure 8. Similar to PathFinder K1, the logic is to first calculate the data chunk size and chunk boundary, then load

Table 4. Benchmarks with 2-Dimensional CTA Structure

Benchmark	Pattern	Number of CTAs	Dominant Pattern
2DCONV	P1	1	
	P2	1	
	P3	$\frac{\lfloor\sqrt{input\_size}\rfloor}{32}$	✓
	P4	$\frac{\lfloor\sqrt{input\_size}\rfloor}{8}$	✓
GEMM	P1	$\frac{\lfloor\sqrt{input\_size/3}\rfloor^2}{num\_thd\_per\_cta}$	✓
SYRK	P1	$\lceil\frac{input\_size}{num\_thd\_per\_cta}\rceil$	✓
Gaussian K2	P1	$\frac{\lfloor\sqrt{input\_size}\rfloor}{8} - 1$	
	P2	1	
	P3	$(\frac{\lfloor\sqrt{input\_size}\rfloor}{16} - 1) \times (\frac{\lfloor\sqrt{input\_size}\rfloor}{8} - 1)$	✓
	P4	$\frac{\lfloor\sqrt{input\_size}\rfloor}{16} - 1$	
	P5	$\frac{\lfloor\sqrt{input\_size}\rfloor}{16} \times (\frac{\lfloor\sqrt{input\_size}\rfloor}{8} - 1)$	✓
	P6	$\frac{\lfloor\sqrt{input\_size}\rfloor}{16}$	
HotSpot		16 patterns, shown in Figure 5(b).	

data and perform computation only if the current thread falls in the valid chunk boundaries. This is only related to CTA ID and thread ID, but not input data itself, so thread resilience persists in the context of different inputs. HotSpot is therefore classified as DI-insensitive, exhibiting remarkably similar resilience patterns across different input sizes.

Looking at Figure 5 where the CTA patterns are visually illustrated, we note that the center blue CTAs are the dominant group. Given the user-defined parameters *num_thd_per_cta*, *pyramid_height* and *Expand_rate*, we can calculate the number of the center blue CTAs as a function of the input size $NX \times NY$ as follows:

$$\#rows = \lceil \frac{NX}{num\_thd\_per\_cta - pyramid\_height \times Expand\_rate} \rceil - 3 \tag{2}$$

$$\#columns = \lceil \frac{NY}{num\_thd\_per\_cta - pyramid\_height \times Expand\_rate} \rceil - 3 \tag{3}$$

Equations 2 and 3, as well as Figure 5 and Figure 7, can be used to infer the resilience of a larger (or smaller) inputs of the same type for HotSpot.

We have examined all benchmarks with a 2D organization in their CTAs, they include 2DCONV, GEMM, SYRK, Gaussian K2, and HotSpot. For the above benchmarks, branch divergence has no effect on the eventual DI count, they are therefore classified as DI-insensitive. The number of CTAs for these benchmarks as a function of the input size is shown in Table 4, while the CTA patterns using the input sizes used in this paper can be found in Appendix A, Table 8. We also point out the dominant patterns of each kernel.

Summary: For DI-insensitive benchmarks with 2-Dimensional patterns and if branch divergence is not affected by input values, it is feasible to extract resilience trends for larger inputs of the same type from smaller ones.

3.2 DI-sensitive benchmarks

Some benchmarks have branch divergence which is affected by input data. Such benchmarks include BFS K7, BFS K8, Jmeint, NN K1, and NN K4. Thread DI count in these benchmarks changes when input size changes, thus these benchmarks are DI-sensitive. We use BFS K8 as an example.

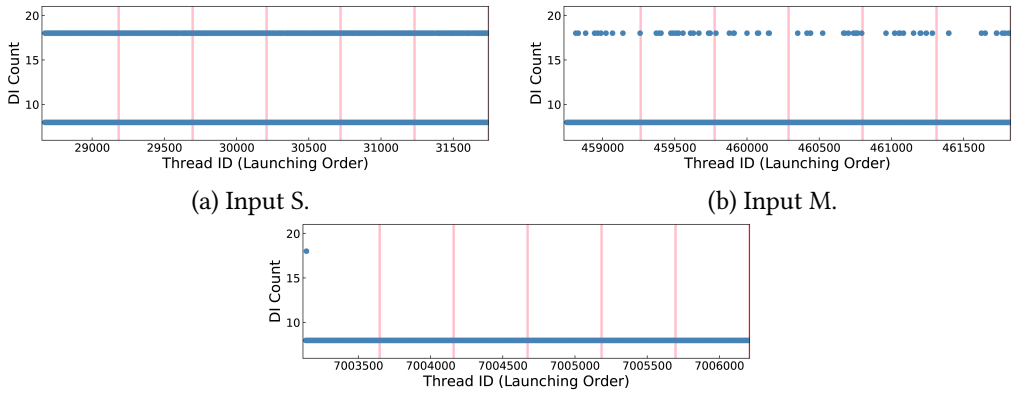


Fig. 9. DI count scatter plot for S, M, and L inputs of BFS K8 (first 6 CTAs only). Although all inputs have only 2 DI groups, there is no clear trend.

DI Analysis. The DI count scatter plot for S, M, and L inputs[3] of BFS K8 is shown in Figure 9. Due to space constraints, we only show the first 6 CTAs. Different from the one-to-one mapping in DI-insensitive benchmarks, there are no clear trends across the three input sizes. Nevertheless, some similarities still exist: the three inputs still have the same DI groups.

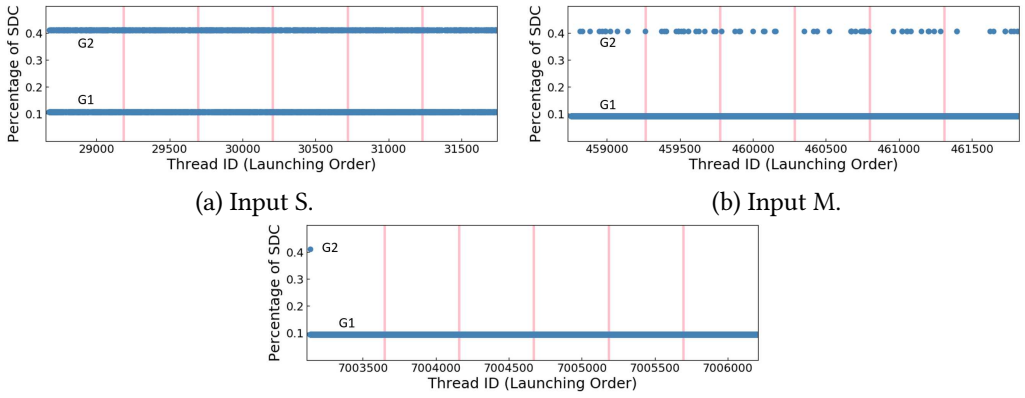
Resilience Analysis. After fault injection experiments on all thread groups, we obtain the resilience patterns, see Figure 10. Comparing the three different inputs in Figure 10(a), (b), and (c), the resilience of each thread group is similar, i.e., they all have one horizontal line at the bottom (dominant group), and another set of threads at the top (minority group). Comparing with Figure 9, the dominant/minority grouping is the same as the one implied by the DI pattern. Therefore, DI count can still be used to link these inputs.

Figure 10 shows the resilience of different inputs for G1 (dominant thread group) and G2 (minority group). Irrespective of input size, the resilience of G1 and G2 remains the same. G1 is more resilient, and its percentage in the whole kernel is increasing as input size increases, directly implying that in BFS K8 larger inputs are more resilient.

We can still extrapolate thread resilience from a small input to larger inputs for DI-sensitive benchmarks. Compared to the deterministic model that we use for extrapolating the resilience profile of DI-insensitive benchmarks, here we need *one more profiling run* to get the DI profile, and add fault injection experiments if threads execute a new DI set that has not been observed with the small input. Still, there is no need to repeat experiments for the dominant group, as its resilience profile persists.

Code and Input Analysis. The source code reveals the reason for this DI-sensitivity, see the snippet in Figure 11. Input to BFS K8 is *g_updating_graph_mask*, which is given by the previous kernel, BFS K7(Lines 2-3). *g_updating_graph_mask* is calculated and updated in the previous kernels based on the input of BFS, which is the nodes and edges of the studied graph. Input for BFS consists of two parts: *g_graph_nodes* for vertices in the graph (Line 5) and *g_graph_edges* for edges in the graph (Line 8). Note that the input *g_graph_visited* in BFS K7 is given by the previous kernel, BFS K6 (Line 9, in K6). When an edge in the input graph changes, *g_updating_graph_mask* is different and the branch divergence in Line 2 is affected. The context of both threads and their instructions

[3]Note that for BFS input S is a subgraph of input M, and input M is a subgraph of input L.



(a) Input S. (b) Input M.

(c) Input L.

Fig. 10. Resilience scatter plot for S, M, and L inputs of BFS K8 (first 6 CTAs). Similar to the DI count scatter plot in Figure 9, there is no clear trend across inputs.

```
1 int tid = blockIdx.x*MAX_THREADS_PER_BLOCK + threadIdx.x;
2 if( tid<no_of_nodes && g_updating_graph_mask[tid])
3 // g_updating_graph_mask is calculated by the previous kernel based on the input.
4 {
5     g_graph_mask[tid]=true;
6     g_graph_visited[tid]=true;
7     *g_over=true;
8     g_updating_graph_mask[tid]=false;
9 }
```

Fig. 11. Code snippet of BFS K8. Input data affects the branch divergence, therefore results in DI-sensitivity.

changes, causing the change of resilience patterns. Depending on input size, the dominant thread group shown in Figure 9 occupies 64.45% – 99.997% of the whole kernel. These threads skip the *if* condition at Line 2 with very limited access to input data; thus are more resilient.

Benchmarks that are sensitive to input when the branch divergence is affected by input values are classified as DI-sensitive benchmarks. In addition to BFS K8, DI-sensitive kernels are BFS K7, Jmeint, NN K1, and NN K4. The discovery of CTA patterns of the above DI-sensitive benchmarks for the three input sizes used in this paper is outlined in Appendix B.

We stress that even for the DI-sensitive case, if the majority group persists across input sizes, its resilience dominates, and we can still estimate kernel resilience from a subset of input to the target one. What we need to do for the larger input, is to estimate the percentage of the majority group in the mix of threads. An additional DI profiling run with the larger/target input is needed in this case to carefully calculate the the percentage of the majority/minority groups. Note that this is just a DI profiling run. No fault injection runs are required to evaluate thread resilience profile, which is already known.

Summary: Benchmarks with branch conditions affected by input values are DI-sensitive. We need one additional DI-profiling run for the resilience estimation of the target (large) input.

4 RESILIENCE ESTIMATION METHODOLOGY

Based on the parallel code organization (CTA patterns and DI counts) and their resilience as identified in the previous section, we present SUGAR (Speeding Up GPGPU Application Resilience

Estimation with input sizing), a methodology for the resilience estimation of a target application with a target input. While the main ideas behind the methodology are established in Section 3, we present here the sequence of steps to evaluate application resilience. The overview of the methodology is given in Figure 12.

Step ❶: Classify Application Type (DI-Insensitive or DI-Sensitive) and CTA Pattern (1D or 2D). As a first step, we check the effect of input data on branch statements that could potentially affect the DI set, to determine whether the target application is DI-insensitive or DI-sensitive. This is obtained by examining if there is any branch divergence that depends on the actual input. If the DI sets of thread groups remain the same for different inputs, then the application is deemed DI-insensitive. Otherwise, if branch divergence is present, then the application is DI-sensitive. DI-insensitive applications are further categorized into 1- or 2-Dimensional based on their CTA organizational structure. For the benchmarks examined in this work, their categorization is shown in Table 5.

CTA Pattern Extraction. Naturally, for DI-insensitive benchmarks, the DI sets of thread groups can be captured deterministically, by the organization of threads in their launching order. This allows the discovery of repeating patterns and resilience trends. The repeating patterns are an outcome of the thread organization in the software development process. For the benchmarks considered in this paper, pattern discovery is fairly straightforward, it is essentially a high-level view of the parallel code and is done by code inspection. As established in Section 3, using the size of the target input, it is possible to calculate the number of the CTA patterns, see Tables 3 and 4 that illustrate the CTA patterns for the DI-insensitive benchmarks examined here.

DI-sensitive benchmarks do not have deterministic DI patterns for different input sizes. Determining how the DI patterns change as a function of input size is typically not easy to discern with just code review. For such cases, an additional DI profiling using the target input is needed to obtain the updated DI counts and corresponding patterns. This step is necessary to determine the updated pattern, i.e., the updated percentages of different thread groups and especially the percentage of the majority thread group that dominates resilience.

Step ❷: Form Input S. To this end, we perform exploratory runs where we explore a few small input sizes. These exploratory runs are used to obtain the DI patterns of small input sizes to determine the smallest input size that fills at least one CTA for each pattern. These are exploratory runs without any fault injection, consequently their cost is negligible. Typically, inputs are in the form of matrices, vectors, or graphs. Given a target input, we use random sampling on the matrix and/or vector elements to form the smallest size S. For applications such as BFS that have graphs as input, random sampling is not possible as it breaks the original features of the input data structure. For graph-based inputs, we select a subgraph that maintains similar characteristics as the target input, e.g., similar degree.

Step ❸: Perform Experiments. Fault injection experiments are performed with input S (formed in **Step ❷**) to measure the kernel resilience profile, i.e., the percentage of expected SDC, masked, and other outputs if a single bit flip occurs. The fault injection campaign is the most expensive part of the resilience estimation as it is based on a large number of experiments in order to reach a resilience profile that is statistically significant [14, 36]. Yet, this step provides pivotal information for the subsequent step, where the resilience of the target input is estimated as it provides the resilience $Res(X)$ of X, where X is a benchmark kernel, a CTA pattern P_i, or a thread group G_i. Note that for DI-sensitive applications, if a thread group with a new DI count is found while obtaining the DI profile of the target input, then additional fault injection experiments for this thread group are needed.

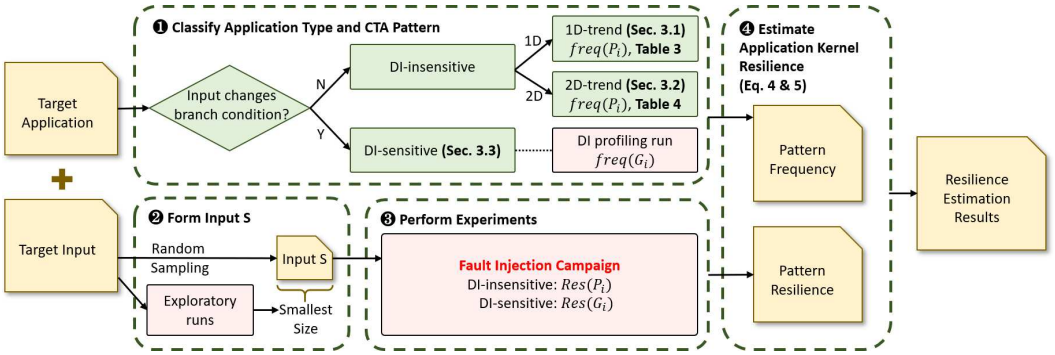


Fig. 12. Methodology Overview: SUGAR

Table 5. Benchmark Classification

Criterion		Benchmarks
DI-insensitive	1D structure	BlackScholes, MVT, Gaussian K1, K-Means K1, K-Means K2, PathFinder K1, PathFinder K5.
	2D structure	2DCONV, GEMM, SYRK, Gaussian K2, HotSpot.
DI-sensitive		BFS K7, BFS K8, Jmeint, NN (K1, K4)

Step ❹: Estimate Application Kernel Resilience. This last step estimates application resilience for the target input. For DI-insensitive applications, assuming that there are n different CTA patterns in the whole kernel, let $Res(P_i)$ be the resilience of CTA pattern P_i for $i < n$ (calculated in **Step ❸**) , where $i < n$, and let its frequency $freq(P_i)$ be calculated by **Step ❶**, see Tables 3 and 4. To calculate kernel resilience, each CTA pattern is weighted by the frequency of each pattern, and the effect of scaling the input size is captured:

$$Res(kernel) = \sum_{i=1}^{n} Res(P_i) \times freq(P_i). \tag{4}$$

For DI-sensitive workloads, we estimate the kernel resilience *at the granularity of a thread*. Because of their non-deterministic DI patterns, we use the additional profiling of **Step ❶** (note that this is just a regular, fault-free run) to obtain the frequency of each thread group G_i to calculate kernel resilience:

$$Res(kernel) = \sum_{i=1}^{n} Res(G_i) \times freq(G_i), \tag{5}$$

where n is the number of thread groups inside the target kernel, and $Res(G_i)$ and $freq(G_i)$ are the resilience and frequency of thread group G_i. We stress that the above formulas are key to extrapolate kernel resilience for larger inputs.

4.1 Arbitrary Large Input: Asymptotic Resilience for DI-insensitive benchmarks

For DI-insensitive kernels, based on the resilience of a small input subset and the extrapolated CTA patterns, our methodology can estimate application resilience for any input size, including

asymptotically large ones. We first use Gaussian K1 as an example, then generalize resilience estimation for asymptotically large inputs.

Gaussian K1 has 2 CTA patterns, see Table 3. P1 occurs $\lceil \frac{\lfloor \sqrt{input\_size} \rfloor}{num\_thd\_per\_cta} \rceil - 1$ times, while P2 occurs only once (tail). Let P to be the total number of patterns in Gaussian K1:

$$P = \lceil \frac{\lfloor \sqrt{input\_size} \rfloor}{num\_thd\_per\_cta} \rceil \tag{6}$$

The percentages of the two composing CTA patterns are:

$$Pct(P1) = \frac{P-1}{P},\ Pct(P2) = \frac{1}{P} \tag{7}$$

and the kernel resilience of Gaussian K1 is obtained by:

$$Res(Kernel) = Pct(P1) \times Res(P1) + Pct(P2) \times Res(P2) \tag{8}$$

For Gaussian, *num_thd_per_cta* in Eq.6 is a predefined number which does not change by the input size. Therefore, when *input_size* becomes asymptotically large, the percentage of each pattern becomes:

$$\begin{aligned} \lim_{input\_size \to \infty} Pct(P1) &= \lim_{P \to \infty} Pct(P1) = 1 \\ \lim_{input\_size \to \infty} Pct(P2) &= \lim_{P \to \infty} Pct(P2) = 0 \end{aligned} \tag{9}$$

and the resilience of Gaussian K1 depends only on the dominant pattern $P1$:

$$\lim_{input\_size \to \infty} Res(Kernel) = 1 \times Res(P1) + 0 \times Res(P2) = Res(P1) \tag{10}$$

The dominant pattern for each kernel is marked on Tables 3 and 4 and is therefore used to derive the asymptotic resilience for arbitrarily large inputs as follows:

$$\lim_{input\_size \to \infty} Res(Kernel) = Res(Dominant\_Pattern) \tag{11}$$

Summary: When input size is asymptotically large, only the dominant patterns (the CTA resilience patterns for DI-insensitive benchmarks or the thread resilience patterns for DI-sensitive benchmarks) contribute to the final kernel resilience. The resilience of the whole kernel is dominated by the resilience of the dominant pattern.

5 EVALUATION

We perform the fault injection campaign using GPGPU-Sim [6]. The detailed configurations of the simulated GPGPU are shown in Table 6. We use fault site pruning [36], the state-of-the-art fault injection methodology, to obtain the percentage of masked, SDC, and other outputs. Higher percentage of masked outputs corresponds to higher application resilience.

Table 6. Detailed configurations of the simulated GPGPU.

Core Features	1400MHz core clock, 15 cores, 32 SIMD width
Resources per Core	48KB shared memory, 32768 registers, Max. 1536 threads (48 wavefronts × 32)
L1 Caches per Core	16KB 4-way L1 data cache, 2KB 4-way I-cache, 12KB 24-way texture cache, 8KB 2-way constant cache, 128B cache block size
L2 Cache	8KB 8-way per sub partition; 786KB in total. 128B cache block size
Memory Model	6 GDDR5 memory controllers, 16 DRAM-banks, FR-FCFS scheduling, 924 MHz memory clock
Interconnect	350MHz interconnect clock

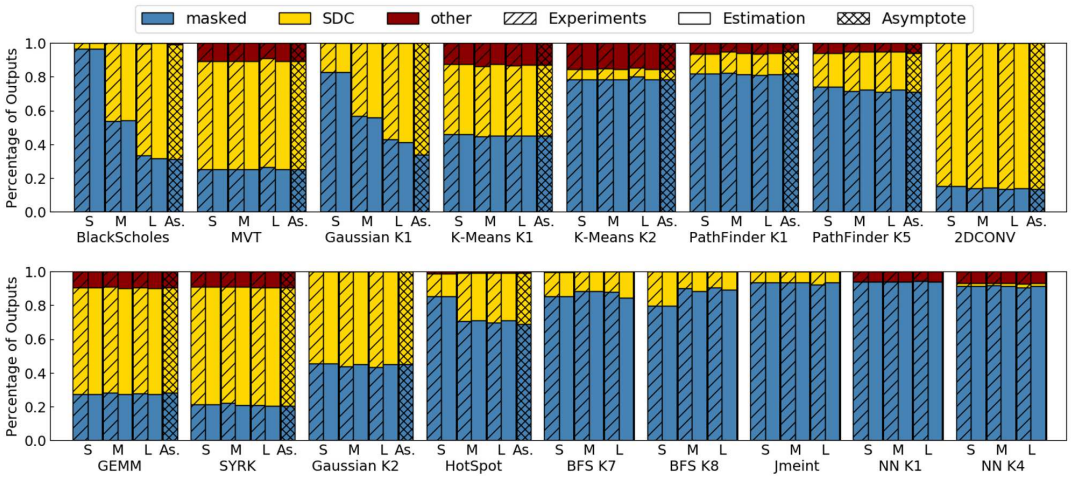


Fig. 13. Resilience profile for different benchmark kernels with medium and large inputs. We use the small input to extrapolate the application resilience for medium and large inputs, this extrapolation is close to ground truth (experimentation).

Accuracy. We use the results of the fault injection campaign of the smallest possible input that each benchmark admits predicting kernel resilience of larger inputs of the same type. Recall that the small input is a subset of the larger one and is defined as the smallest input that can fill a CTA. The smallest input depends on the parallelization choices of the software developer. We evaluate SUGAR's prediction accuracy for two target sizes (medium and large) by comparing to results of a detailed experimental campaign according to the existing state of the art (the fault site pruning methodology) [36].

Figure 13 shows the comparison between estimation (solid bars) and experimentation (shaded bars). For the small input, estimation and experimentation are identical, so it is pointless to evaluate any accuracy. Across all benchmarks and for medium input, the average differences between estimation and experimentation in terms of masked, SDC, and other outputs are 0.68%, 0.64%, and 0.25%, respectively. Accuracy remains excellent for large input: average errors are as low as 1.14%, 1.07%, and 0.31% for masked, SDC, and other outputs. Average errors across all cases (M and L), for masked, SDC, and other outputs become 0.89%, 0.83%, and 0.28%, respectively.

The figure also presents the asymptotic resilience of DI-insensitive benchmarks (bar with the tiled pattern, rightmost bar for each kernel). Naturally, for asymptotically large inputs, we only report the estimation curve as the input size is practically infinity. For DI-sensitive benchmarks, since an additional DI profiling is needed, estimating their asymptotic resilience is not possible. For several kernels, the L input already reaches asymptotic resilience.

Finally, it is interesting to note three distinctive trends:

a) As input size increases, resilience drops for BlackScholes, Gaussian K1, and HotSpot. For these benchmarks, the dominant CTA pattern is the most vulnerable one with the least masked outputs. As size increases, the percentage of dominant CTA pattern also increases, consequently the percentage of masked outputs decreases.
b) BFS K8 has the opposite trend: as input increases, reliability increases. For this kernel, the dominant thread group is the more resilient one, and this is reflected with larger inputs.

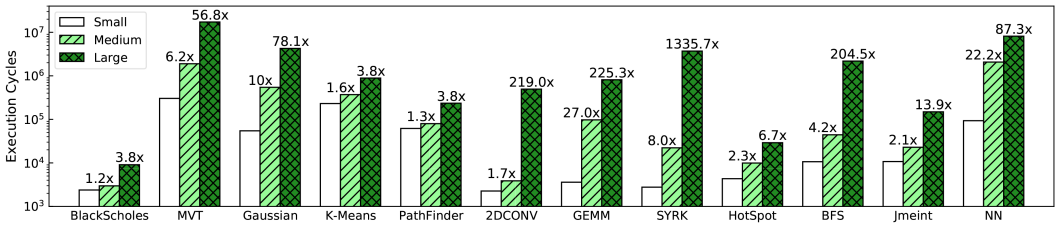


Fig. 14. Raw number of execution cycles for S, M, and L inputs. The resilience estimation speedup for M and L is also given as a raw number for the M and L bars. On average, we achieve a speed-up of 7.3 for medium inputs, a speedup of 186.6 for large inputs.

c) For the remaining benchmarks, resilience remains flat. For MVT, K-Means K1 and K2, PathFinder K1 and K5, 2DCONV, GEMM, SYRK, and BFS K7, even for the smallest input, the percentage of dominant pattern is already large enough to capture the asymptotic resilience.

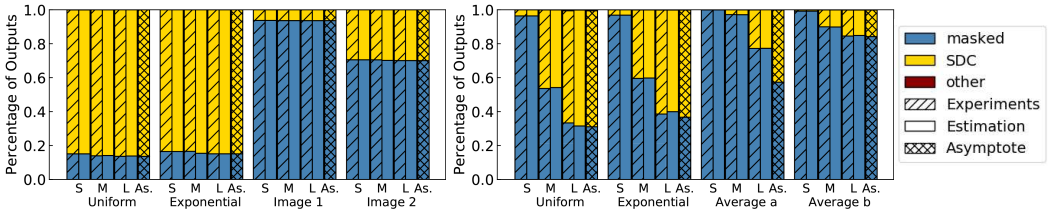
Efficiency. Because of the shared cluster environment we are using, pure timing measurements are not accurate, we therefore evaluate the performance of SUGAR as a function of instruction cycles, see Figure 14. Although we analyze the resilience of kernels separately, to perform the fault injection run, the whole benchmark including all kernels needs to be executed. We therefore measure the number of execution cycles for the entire application, rather than for each kernel. We show the raw number of execution cycles in the bar plot of Figure 14, note that the y-axis is in logscale. On top of the M and L bars, we also report the actual speedup achieved with SUGAR.

For DI-insensitive benchmarks, the estimation overhead is only the fault injection campaign for input S. For DI-sensitive benchmarks (i.e., BFS, Jmeint, and NN), the estimation overhead is the fault injection campaign for input S, plus 1 additional dynamic instruction profiling run for each larger input. We achieve an average speed-up of 97.0 across all benchmarks and both M and L sizes. Note that this speed-up is tied to the two input sizes that we consider (M and L). If we consider only the input L (essentially if we ignore the input M), the average speed up increases by 186.6. If larger inputs than those used here were to be considered, then speed ups would have been higher.

Sensitivity to input type. The above discussion focuses on kernel resilience for different input size but for the same input *type*. Here, we try to answer the following question: How does benchmark resilience change depending on the *input type*? We select two benchmarks: one with flat resilience profile (2DCONV) and one with a decreasing resilience trend (Blacksholes) as input size increases.

For 2DCONV we generate four inputs types: using a Uniform distribution with mean 0.5, using an Exponential distribution with mean 0.5, and two inputs of two different images. Recall that BlackScholes's input has three parts (stock price, exercise price, and time, as described in Section 2.3). The first input type uses data that are generated with a Uniform distribution, the second type is generated with an Exponential distribution with the same mean as the Uniform, and two more types are generated under Uniform distributions with means that are two and four orders of magnitude larger than the first two input types.

Results are presented in Figure 15. *The figure shows that trends (flat, decreasing) persist across different input types but application resilience is strongly tied to the range of input values*: the difference in resilience between the Exponential and Uniform inputs with the same mean is insignificant. When inputs with dramatically different ranges or values are used (lower values for 2DCONV, and higher values for BlackScholes), resilience dramatically increases. Consistent with the results presented in Figure 13, SUGAR's estimation is in excellent agreement with experimentation.



(a) 2DCONV. (b) BlackScholes.

Fig. 15. Influence of different input types.

The above experiments illustrate that the *input type* can have a tremendous effect on GPGPU application resilience. The topic deserves more detailed analysis but it is out of scope here.

6 RELATED WORK

GPGPU application resilience analysis: Fault-injection is commonly used to evaluate the reliability of GPGPU applications [14, 20, 32, 36, 42, 44, 56]. SWIFI [56] injects faults by modifying programs at the source level. The process is simple and fast but is too coarse-grained to accurately capture resilience features at finer levels. GPU-Qin [14] injects faults at the micro-architecture level. The authors leverage the GPU debugging tool *cuda-gdb* [1] to inject single bit errors into destination registers. SASSIFI [20] directly injects faults into low-level SASS instructions, also with the help of *cuda-gdb*. Unlike the compiler-based methods used in GPU-Qin and SASSIFI, Tselonis et al. [49] propose GUFI to validate the feasibility of using the commonly used GPGPU simulator, GPGPU-Sim [6] to study the reliability of GPGPU applications. Nie et al. [36] leverage GPGPU-specific features to dramatically prune the fault site space without sacrificing accuracy. This pruned fault site space can be used for evaluating how application resilience deteriorates in the presence of multi-bit faults [54]. PCFI [42] accelerates fault injection guided by program counters to predict fault injection outcomes. NVBitFI [3] is a fault injector for NVIDIA GPUs built on top of NVBit [51].

CPU resilience analysis: At the CPU domain, application resilience is measured using the Architectural Vulnerability Factor (AVF) estimation or fault injection. Duan et al. [12] use Boosted Regression Trees to model the relationship between AVF and various performance metrics. Nair et al. [31] introduce a first-order mechanistic model for AVF, with inexpensive profiling to calculate AVF. This model can also be used to explore factors that affect AVF. In addition to AVF, Sridharan et al. [46] introduce the concept of Program Vulnerability Factor (PVF) and provide the method of calculating PVF. Another detailed analysis of PVF to explain AVF behaviors is presented in [47].

Fault injection techniques are applied in CPU domain at different hardware and software levels[8, 11, 19, 24, 28, 50]. Relyzer [19] analyzes applications to generate a subset of fault sites for fault injection. Approxilyzer [50] is built on top of Relyzer[19] to identify vulnerable instructions which lead to SDC outputs. MeRLiN [24] accelerates the fault injection campaign for reliability assessment by performing ACE-like analysis, grouping similar fault sites, and pruning fault sites. Trident [28] analyzes error propagation at different levels to predict the percentage of SDC outputs for the whole application and its instructions.

Input-dependent resilience analysis: A common limitation of the above works in both the CPU and GPU domains is that they are input-dependent. In other words, fault injection experiments have to be redone for different inputs. In the CPU domain, there are some works that consider the effect of input on reliability. Li et al. [26] study the impact of different inputs on the probabilities of silent data corruption (SDC) for CPU applications. Their solution, vTrident is compiler-based

and bounds the SDC probability under multiple inputs using fault injection results obtained by only one input. Inspired by software testing, Minotaur [30] leverages techniques from the software engineering domain to speed up the reliability analysis for CPU applications. Specifically, they use the test-case minimization concept to minimize the inputs selected for reliability assessment and use binary search to find inputs that are small but representative. The above two solutions focus on sequential execution and while well-suited to the CPU domain, they cannot be directly applied to GPU applications as analysis needs to be performed for every single thread. Given the number of threads and huge exhaustive fault sites shown in Table 1, straight-forward application of the above techniques is not viable.

To our best knowledge, SUGAR is the first methodology in the GPU domain for estimating application resilience for arbitrary large inputs given the smallest possible input of the same type. This is achieved by identifying how CTA patterns are organized (and evolve) as a function of the input size.

7 CONCLUSIONS

We deeply analyze the impact of different input sizes on application resilience and propose a new resilience estimation methodology for large inputs that result in significant speedups (up to 1336 for the cases considered here and 97.0 on the average) while being remarkably accurate with average errors less than 1%. We show that the thread dynamic instruction (DI) count and CTA patterns have a significant impact on the overall application resilience. The proposed methodology, SUGAR, allows for the use of a small subset of input to extrapolate the resilience of arbitrarily large inputs of the same type. As an immediate extension of this work, we are looking into analyzing the effects of multi-bit faults by applying the methodology outlined in [54], the input-dependent resilience of benchmarks with more thread communications, and the effect of the input type (especially the range of input values) to application resilience.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their insightful comments that significantly improved the presentation of SUGAR. 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] [n. d.]. CUDA-GDB. http://docs.nvidia.com/cuda/cuda-gdb/#axzz4PHxjHEUB
[2] [n. d.]. GP100 Pascal Whitepaper. https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf
[3] [n. d.]. NVBitFI. https://github.com/NVlabs/nvbitfi.
[4] [n. d.]. NVIDIA Fermi Architecture Whitepaper. http://www.nvidia.com/content/pdf/fermi_white_papers/nvidia_fermi_compute_architecture_whitepaper.pdf
[5] [n. d.]. NVIDIA Kepler GK110 Architecture Whitepaper.
[6] Ali Bakhoda, George L Yuan, Wilson WL Fung, Henry Wong, and Tor M Aamodt. 2009. Analyzing CUDA workloads using a detailed GPU simulator. In *Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on*. IEEE, 163–174.
[7] Subho S Banerjee, Saurabh Jha, James Cyriac, Zbigniew T Kalbarczyk, and Ravishankar K Iyer. 2018. Hands off the wheel in autonomous vehicles?: A systems perspective on over a million miles of field data. In *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 586–597.
[8] Jon Calhoun, Luke Olson, and Marc Snir. 2014. FlipIt: An LLVM based fault injector for HPC. In *European Conference on Parallel Processing*. Springer, 547–558.

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

[10] Zitao Chen, Guanpeng Li, Karthik Pattabiraman, and Nathan DeBardeleben. 2019. BinFI: an efficient fault injector for safety-critical machine learning systems. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–23.

[11] Hyungmin Cho, Shahrzad Mirkhani, Chen-Yong Cher, Jacob A Abraham, and Subhasish Mitra. 2013. Quantitative evaluation of soft error injection techniques for robust system design. In *Proceedings of the 50th Annual Design Automation Conference*. ACM, 101.

[12] Lide Duan, Bin Li, and Lu Peng. 2009. Versatile prediction and fast estimation of architectural vulnerability factor from processor performance metrics. In *2009 IEEE 15th International Symposium on High Performance Computer Architecture*. IEEE, 129–140.

[13] Anders Eklund, Paul Dufort, Daniel Forsberg, and Stephen M LaConte. 2013. Medical image processing on the GPU–Past, present and future. *Medical image analysis* 17, 8 (2013), 1073–1094.

[14] Bo Fang, Karthik Pattabiraman, Matei Ripeanu, and Sudhanva Gurumurthi. 2014. 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*. IEEE, 221–230.

[15] R Foster. 2012. How to harness big data for improving public health. *Government Health IT* (2012).

[16] Vinicius Fratin, Daniel Oliveira, Caio Lunardi, Fernando Santos, Gennaro Rodrigues, and Paolo Rech. 2018. Code-dependent and architecture-dependent reliability behaviors. In *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 13–26.

[17] Qian Gong, Phil DeMar, and Wenji Wu. 2017. *Deep Packet/Flow Analysis using GPUs*. Technical Report.

[18] Scott Grauer-Gray, Lifan Xu, Robert Searles, Sudhee Ayalasomayajula, and John Cavazos. 2012. Auto-tuning a high-level language targeted to GPU codes. In *Innovative Parallel Computing (InPar), 2012*. IEEE, 1–10.

[19] Siva Kumar Sastry Hari, Sarita V Adve, Helia Naeimi, and Pradeep Ramachandran. 2012. Relyzer: Exploiting application-level fault equivalence to analyze application resiliency to transient faults. In *ACM SIGPLAN Notices*, Vol. 47. ACM, 123–134.

[20] Siva Kumar Sastry Hari, Timothy Tsai, Mark Stephenson, Stephen W Keckler, and Joel Emer. 2015. SASSIFI: Evaluating resilience of GPU applications. In *Proceedings of the Workshop on Silicon Errors in Logic-System Effects*.

[21] Jen-Cheng Huang, Joo Hwan Lee, Hyesoon Kim, and Hsien-Hsin S Lee. 2014. GPUMech: GPU performance modeling technique based on interval analysis. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE, 268–279.

[22] Saurabh Jha, Subho S. Banerjee, Timothy Tsai, Siva Kumar Sastry Hari, Michael B. Sullivan, Zbigniew T. Kalbarczyk, Stephen W. Keckler, and Ravishankar K. Iyer. 2019. ML-Based Fault Injection for Autonomous Vehicles: A Case for Bayesian Fault Injection. In *49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2019, Portland, OR, USA, June 24-27, 2019*. IEEE, 112–124. https://doi.org/10.1109/DSN.2019.00025

[23] Adwait Jog, Onur Kayiran, Nachiappan Chidambaram Nachiappan, Asit K Mishra, Mahmut T Kandemir, Onur Mutlu, Ravishankar Iyer, and Chita R Das. 2013. OWL: cooperative thread array aware scheduling techniques for improving GPGPU performance. In *ACM SIGPLAN Notices*, Vol. 48. ACM, 395–406.

[24] Manolis Kaliorakis, Dimitris Gizopoulos, Ramon Canal, and Antonio Gonzalez. 2017. MeRLiN: Exploiting Dynamic Instruction Behavior for Fast and Accurate Microarchitecture Level Reliability Assessment. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*. ACM, 241–254.

[25] David B Kirk and W Hwu Wen-Mei. 2016. *Programming massively parallel processors: a hands-on approach*. Morgan Kaufmann.

[26] Guanpeng Li and Karthik Pattabiraman. 2018. Modeling input-dependent error propagation in programs. In *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 279–290.

[27] Guanpeng Li, Karthik Pattabiraman, Chen-Yang Cher, and Pradip Bose. 2016. Understanding error propagation in GPGPU applications. In *High Performance Computing, Networking, Storage and Analysis, SC16: International Conference for*. IEEE, 240–251.

[28] Guanpeng Li, Karthik Pattabiraman, Siva Kumar Sastry Hari, Michael Sullivan, and Timothy Tsai. 2018. Modeling soft-error propagation in programs. In *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 27–38.

[29] Abdulrahman Mahmoud, Neeraj Aggarwal, Alex Nobbe, Jose Vicarte, Sarita Adve, Christopher Fletcher, Iuri Frosio, and Siva Hari. 2020. PyTorchFI: A Runtime Perturbation Tool for DNNs. 25–31. https://doi.org/10.1109/DSN-W50199.2020.00014

[30] Abdulrahman Mahmoud, Radha Venkatagiri, Khalique Ahmed, Sasa Misailovic, Darko Marinov, Christopher W Fletcher, and Sarita V Adve. 2019. Minotaur: Adapting Software Testing Techniques for Hardware Errors. In *Proceedings of the*

Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems. ACM, 1087–1103.

[31] Arun Arvind Nair, Stijn Eyerman, Lieven Eeckhout, and Lizy Kurian John. 2012. A first-order mechanistic model for architectural vulnerability factor. In *2012 39th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 273–284.

[32] Bin Nie, Adwait Jog, and Evgenia Smirni. 2020. 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*. IEEE, 111–120.

[33] Bin Nie, Devesh Tiwari, Saurabh Gupta, Evgenia Smirni, and James H Rogers. 2016. A large-scale study of soft-errors on GPUs in the field. In *High Performance Computer Architecture (HPCA), 2016 IEEE International Symposium on*. IEEE, 519–530.

[34] Bin Nie, Ji Xue, Saurabh Gupta, Christian Engelmann, Evgenia Smirni, and Devesh Tiwari. [n. d.]. Characterizing Temperature, Power, and Soft-Error Behaviors in Data Center Systems: Insights, Challenges, and Opportunities. In *MASCOTS 2017*. 22–31.

[35] Bin Nie, Ji Xue, Saurabh Gupta, Tirthak Patel, Christian Engelmann, Evgenia Smirni, and Devesh Tiwari. [n. d.]. Machine Learning Models for GPU Error Prediction in a Large Scale HPC System. In *DSN 2018*. 95–106.

[36] Bin Nie, Lishan Yang, Adwait Jog, and Evgenia Smirni. 2018. Fault Site Pruning for Practical Reliability Analysis of GPGPU Applications. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 749–761.

[37] NVIDIA. [n. d.]. Computational Finance. http://www.nvidia.com/object/computational_finance.html

[38] NVIDIA. [n. d.]. Researchers Deploy GPUs to Build World's Largest Artificial Neural Network. https://nvidianews.nvidia.com/news/researchers-deploy-gpus-to-build-world-s-largest-artificial-neural-network

[39] NVIDIA. 2011. CUDA C/C++ SDK Code Samples. http://developer.nvidia.com/cuda-cc-sdk-code-samples

[40] Jin-Hong Park, Munehiro Tada, Duygu Kuzum, Pawan Kapur, Hyun-Yong Yu, Krishna C Saraswat, et al. 2008. Low temperature ($\lesssim 380^\circ$ 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*. IEEE, 1–4.

[41] Guillem Pratx and Lei Xing. 2011. GPU computing in medical physics: A review. *Medical physics* 38, 5 (2011), 2685–2697.

[42] Fritz G Previlon, Charu Kalra, Devesh Tiwari, and David R Kaeli. 2019. PCFI: Program Counter Guided Fault Injection for Accelerating GPU Reliability Assessment. In *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 308–311.

[43] Behrooz Sangchoolie, Karthik Pattabiraman, and Johan Karlsson. 2017. 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*. IEEE Computer Society, 97–108. https://doi.org/10.1109/DSN.2017.30

[44] Hamid Sarbazi-Azad. 2016. *Advances in GPU Research and Practice*. Morgan Kaufmann.

[45] I Schmerken. 2009. Wall street accelerates options analysis with GPU technology. *Wall Street Technology* 11 (2009).

[46] Vilas Sridharan and David R Kaeli. 2008. Quantifying software vulnerability. In *Proceedings of the 2008 Workshop on Radiation Effects and Fault Tolerance in Nanometer Technologies*. ACM, 323–328.

[47] Vilas Sridharan and David R Kaeli. 2009. Eliminating microarchitectural dependency from architectural vulnerability. In *2009 IEEE 15th International Symposium on High Performance Computer Architecture*. IEEE, 117–128.

[48] Sam S. Stone, Justin P. Haldar, Stephanie C. Tsao, Wen mei W. Hwu, Bradley P. Sutton, and Zhi-Pei Liang. 2008. Accelerating advanced MRI reconstructions on GPUs. *J. Parallel Distrib. Comput.* 68, 10 (2008), 1307–1318.

[49] Sotiris Tselonis and Dimitris Gizopoulos. 2016. GUFI: A framework for GPUs reliability assessment. In *Performance Analysis of Systems and Software (ISPASS), 2016 IEEE International Symposium on*. IEEE, 90–100.

[50] Radha Venkatagiri, Abdulrahman Mahmoud, Siva Kumar Sastry Hari, and Sarita V Adve. 2016. 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*. IEEE, 1–14.

[51] Oreste Villa, Mark Stephenson, David Nellans, and Stephen W Keckler. 2019. NVBit: A Dynamic Binary Instrumentation Framework for NVIDIA GPUs. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. 372–383.

[52] Xiebing Wang, Kai Huang, Alois Knoll, and Xuehai Qian. 2019. A Hybrid Framework for Fast and Accurate GPU Performance Estimation through Source-Level Analysis and Trace-Based Simulation. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 506–518.

[53] Lishan Yang, Bin Nie, Adwait Jog, and Evgenia Smirni. 2021. Enabling Software Resilience in GPGPU Applications via Partial Thread Protection. In *43rd International Conference on Software Engineering, 23-29 May 2021 (to appear)*.

[54] Lishan Yang, Bin Nie, Adwait Jog, and Evgenia Smirni. 2021. Practical Resilience Analysis of GPGPU Applications in the Presence of Single- and Multi-Bit Faults. *IEEE Trans. Comput.* 70, 1 (2021), 30–44.
[55] Amir Yazdanbakhsh, Divya Mahajan, Hadi Esmaeilzadeh, and Pejman Lotfi-Kamran. 2017. Axbench: A multiplatform benchmark suite for approximate computing. *IEEE Design & Test* 34, 2 (2017), 60–68.
[56] Keun Soo Yim, Cuong Pham, Mushfiq Saleheen, Zbigniew Kalbarczyk, and Ravishankar Iyer. 2011. Hauberk: Lightweight silent data corruption error detector for GPGPU. In *2011 IEEE International Parallel & Distributed Processing Symposium (IPDPS)*. IEEE, 287–300.

A PATTERNS OF DI-INSENSITIVE BENCHMARKS.

In Section 3.1 we use PathFinder K1 and HotSpot to showcase how to derive the patterns of DI-insensitive kernels. The number of patterns of 1D and 2D benchmarks with input size N are summarized in Table 3 and 4. Tables 7 and 8 illustrate the patterns of the specific input sizes used in this paper (their generic formulas are provided in Table 1, column 5).

Table 7. 1D Patterns.

Benchmark	Pattern	Number of CTAs		
		S	M	L
BlackScholes	P1	1	32	256
	P2	479	448	224
MVT	P1	1	1	1
Gaussian K1	P1	1	2	8
K-Means K1	P1	2	4	8
	P2	2	0	1
K-Means K2	P1	2	4	8
	P2	2	0	1
PathFinder K1	P1	1	1	1
	P2	1	33	150
	P3	1	1	1
PathFinder K2	P1	1	1	1
	P2	1	33	150
	P3	1	1	1

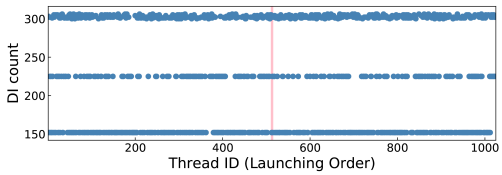
Table 8. 2D Patterns.

Benchmark	Pattern	Number of CTAs		
		S	M	L
2DCONV	P1	1	1	1
	P2	1	1	1
	P3	1	8	64
	P4	4	32	256
GEMM	P1	1	64	256
SYRK	P1	1	16	256
Gaussian K2	P1	3	7	31
	P2	1	1	1
	P3	3	21	465
	P4	1	3	15
	P5	6	28	496
	P6	2	4	16
HotSpot	$\#row$	3	8	19
	$\#column$	3	8	19

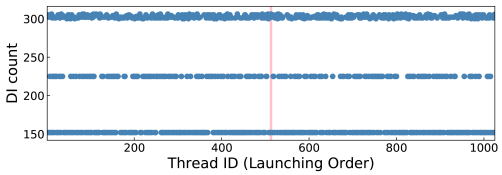
B PATTERNS OF DI-SENSITIVE BENCHMARKS

In this appendix, we show the CTA patterns of all DI-sensitive benchmarks used in this paper and show how to derive their resilience for different inputs.

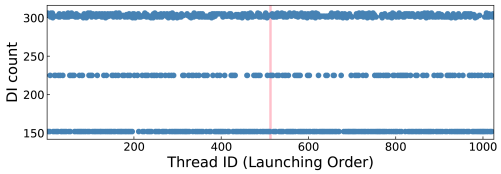
Jmeint. Figure 16 shows the DI patterns of Jmeint for different inputs. There are mainly three DI groups clearly shown in Figure 16: one at the bottom with 152 dynamic instructions per thread, one in the middle with 225 dynamic instructions per thread, and one at the top with DI count around 300. However, these patterns are not deterministic. A code snippet of Jmeint is presented in Figure 17. The input of Jmeint is several arrays stores the 3D triangle coordinates, and the arrays initialized in Line 5 are assigned with input data (Line 10–27). All these arrays, *v0, v1, v2, u0, u1*, and *u2* are defined using input data. After several hops of variable calculations, *du0du1* and *du0du2* are compared to 0.0 in Line 58 (if condition). Because these two variables are derived using input data, there is branch divergence due to input data. Therefore, Jmeint is DI-sensitive. The resilience patterns of Jmeint are shown in Figure 18. Since Jmeint is DI-sensitive, one additional DI profiling run with the target input is needed to determine its resilience.



(a) Input S.



(b) Input M (First 2 CTAs, 20 CTAs in total).

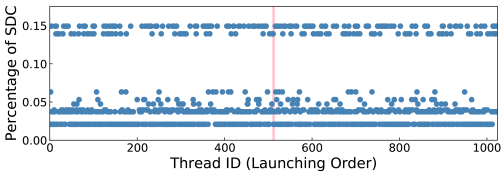


(c) Input L (First 2 CTAs, 200 CTAs in total).

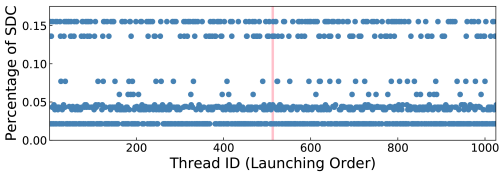
Fig. 16. DI count scatter plot for S, M, and L inputs of Jmeint.

```
__global__ void jmeint_kernel(float *v0_d, float *v1_d,
   float *v2_d, float *u0_d, float*u1_d, float*u2_d, ......)
{
    ......
    float v0[3], v1[3], v2[3], u0[3], u1[3], u2[3];
    int blockId =  blockIdx.x + blockIdx.y * gridDim.x;
    int idx = blockId * blockDim.x + threadIdx.x;
    ......
    // Array v0, v1, v2, u0, u1, u2 all depend on input data.
    v0[0] = v0_d[idx * 3 + 0];
    v0[1] = v0_d[idx * 3 + 1];
    v0[2] = v0_d[idx * 3 + 2];
    v1[0] = v1_d[idx * 3 + 0];
    v1[1] = v1_d[idx * 3 + 1];
    v1[2] = v1_d[idx * 3 + 2];
    v2[0] = v2_d[idx * 3 + 0];
    v2[1] = v2_d[idx * 3 + 1];
    v2[2] = v2_d[idx * 3 + 2];
    u0[0] = u0_d[idx * 3 + 0];
    u0[1] = u0_d[idx * 3 + 1];
    u0[2] = u0_d[idx * 3 + 2];
    u1[0] = u1_d[idx * 3 + 0];
    u1[1] = u1_d[idx * 3 + 1];
    u1[2] = u1_d[idx * 3 + 2];
    u2[0] = u2_d[idx * 3 + 0];
    u2[1] = u2_d[idx * 3 + 1];
    u2[2] = u2_d[idx * 3 + 2];
    ......
    float e1[3], e2[3], n1[3], n2[3], d[3];
    float d1, du0, du1, du2, du0du1, du0du2;
    ......
    // e1 depends on v1 and v0
    e1[0] = v1[0] - v0[0];
    e1[1] = v1[1] - v0[1];
    e1[2] = v1[2] - v0[2];
    // e2 depends on v2 and v0
    e2[0] = v2[0] - v0[0];
    e2[1] = v2[1] - v0[1];
    e2[2] = v2[2] - v0[2];
    // n1 depends on e1 and e2
    n1[0] = (e1[1] * e2[2]) - (e1[2] * e2[1]);
    n1[1] = (e1[2] * e2[0]) - (e1[0] * e2[2]);
    n1[2] = (e1[0] * e2[1]) - (e1[1] * e2[0]);
    // d1 depends on n1, v0
    d1 = -(n1[0] * v0[0] + n1[1] * v0[1] + n1[2] * v0[2]);
    // du0 depends on n1, u0, d1
    du0 = (n1[0] * u0[0] + n1[1] * u0[1] + n1[2] * u0[2]) + d1;
    // du1 depends on n1, u1, d1
    du1 = (n1[0] * u1[0] + n1[1] * u1[1] + n1[2] * u1[2]) + d1;
    // du2 depends on n1, u2, d1
    du2 = (n1[0] * u2[0] + n1[1] * u2[1] + n1[2] * u2[2]) + d1;
    ......
    // du0du1 depends on du0, du1
    du0du1 = du0 * du1;
    // du0du2 depends on du0, du2
    du0du2 = du0 * du2;
    // This if condition depends on du0du1, du0du2
    if (du0du1 > 0.0f && du0du2 > 0.0f) {
        return false; // no intersection
    }
    ......
}
```

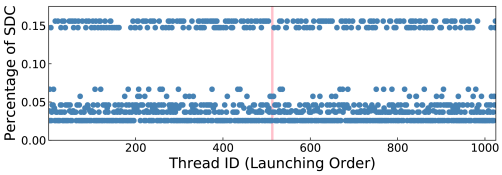
Fig. 17. Code snippet of Jmeint.



(a) Input S.

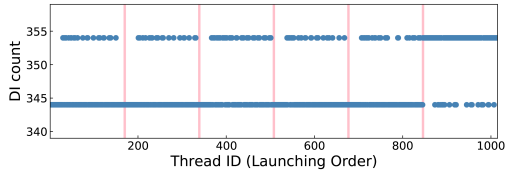


(b) Input M (First 2 CTAs, 20 CTAs in total).

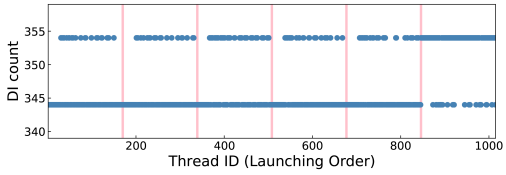


(c) Input L (First 2 CTAs, 200 CTAs in total).

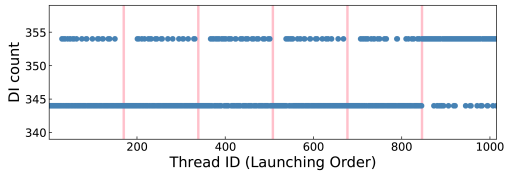
Fig. 18. Resilience scatter plot in terms of the percentage of SDC outcomes for S, M, and L inputs of Jmeint.



(a) Input S.

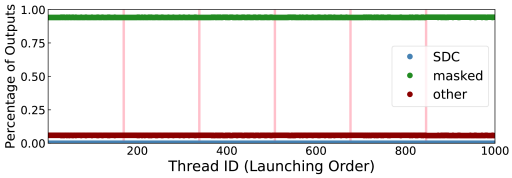


(b) Input M (First 6 CTAs, 192 CTAs in total).

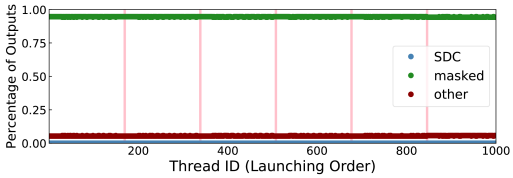


(c) Input L (First 6 CTAs, 768 CTAs in total).

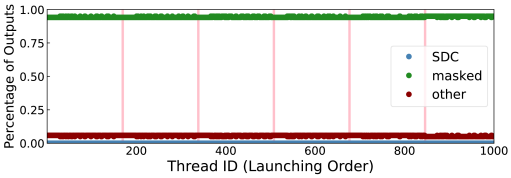
Fig. 19. DI count scatter plot for S, M, and L inputs of NN K1.



(a) Input S.



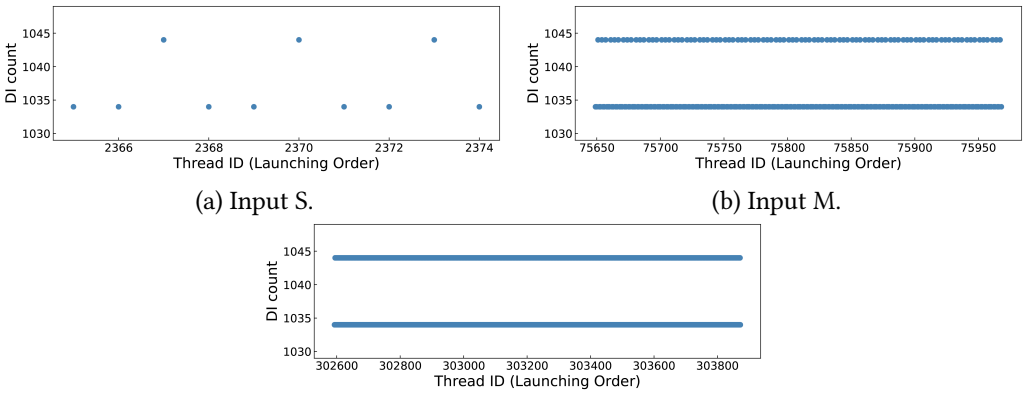
(b) Input M (First 6 CTAs, 192 CTAs in total).



(c) Input L (First 6 CTAs, 768 CTAs in total).

Fig. 20. Resilience scatter plot for S, M, and L inputs of NN K1.

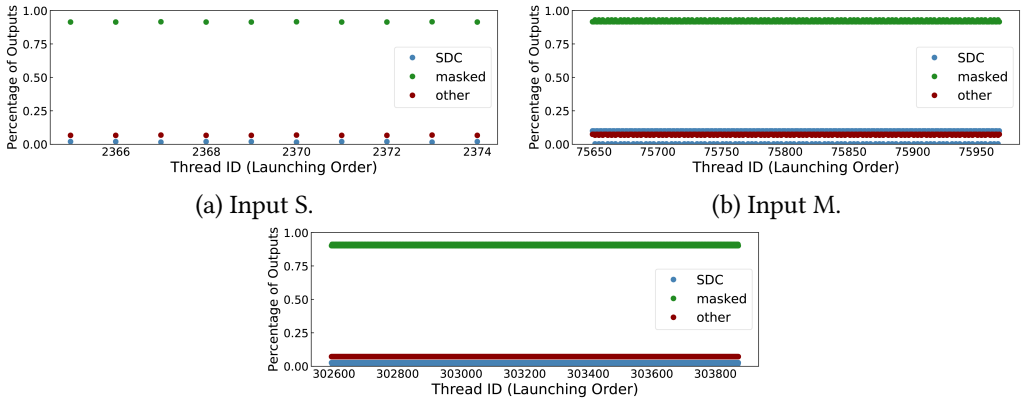
NN K1. There are two DI groups in NN K1, see Figure 19, the resilience scatter plot is in Figure 20. Because there are nearly no SDC outcomes in NN K1, we take this opportunity to also plot the percentage of masked, SDC, and other outcomes (it is not possible to do the same in the remaining of the benchmarks because SDCs have a strong presence across most CTAs and if masked and other outcomes are also plotted, the figures become visually unattractive).



(a) Input S. (b) Input M.

(c) Input L.

Fig. 21. DI count scatter plot for S, M, and L inputs of NN K4. Here we don't plot the CTA lines, because every CTA has only one thread.

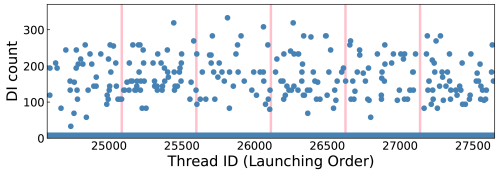


(a) Input S. (b) Input M.

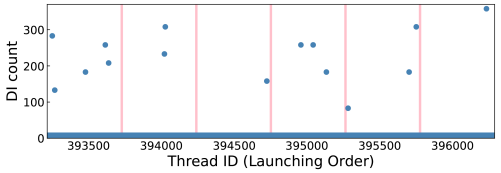
(c) Input L.

Fig. 22. Resilience scatter plot in terms of the percentage of masked outcomes for S, M, and L inputs of NN K4. Because there is no SDC outcome in NN K4, we use the percentage of masked outcomes to show the resilience patterns. Here we don't plot the CTA lines, because every CTA has only one thread.

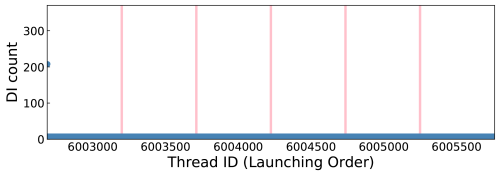
NN K4. The DI count and resilience scatter plots are shown in Figure 21 and Figure 22, respectively. Here we do not plot the CTA lines (vertical pink lines), because every CTA has only one thread. This is the original implementation and setup of NN K4 in the CUDA benchmark suite.



(a) Input S (First 6 CTAs, 8 CTAs in total).

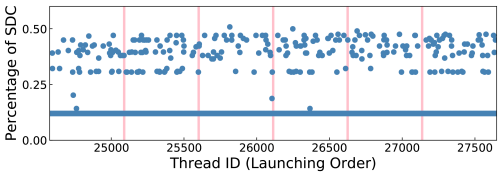


(b) Input M (First 6 CTAs, 128 CTAs in total).

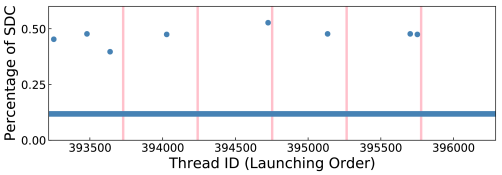


(c) Input L (First 6 CTAs, 1954 CTAs in total).

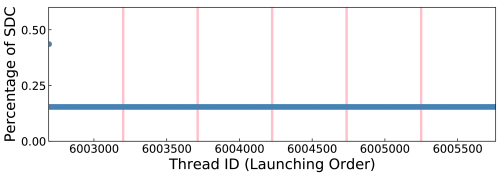
Fig. 23. Resilience scatter plot in terms of the percentage of masked outcomes for S, M, and L inputs of BFS K7.



(a) Input S (First 6 CTAs, 8 CTAs in total).



(b) Input M (First 6 CTAs, 128 CTAs in total).



(c) Input L (First 6 CTAs, 1954 CTAs in total).

Fig. 24. Resilience scatter plot in terms of the percentage of masked outcomes for S, M, and L inputs of BFS K7.

BFS K7. Figure 23 and Figure 24 shows the DI count patterns and resilience patterns of BFS K7, respectively. Similar to BFS K8, there is a main DI group at the bottom, and minor DI groups with more dynamic instructions.

Received October 2020; revised December 2020; accepted January 2021