PRISM: Predicting Resilience of GPU Applications Using Statistical Methods

Charu Kalra*, Fritz Previlon*, Xiangyu Li*, Norman Rubin†, and David Kaeli*
*Northeastern University, Boston, MA, †NVIDIA, USA
{ckalra, previlon, xili, kaeli}@ece.neu.edu, {nrubin}@nvidia.com

***Abstract*—As Graphics Processing Units (GPUs) become more pervasive in High Performance Computing (HPC) and safety-critical domains, ensuring that GPU applications can be protected from data corruption grows in importance. Despite prior efforts to mitigate errors, we still lack a clear understanding of how resilient these applications are in the presence of transient faults. Due to the random nature of these faults, predicting whether they will alter program output is a challenging problem. In this paper, we build a framework named PRISM which uses a systematic approach to predict failures in GPU programs. PRISM extracts micro-architecture agnostic features to characterize program resiliency, which serve as predictors to drive our statistical model. PRISM enables us to predict failures in applications without running exhaustive fault injection campaigns, thereby reducing the error estimation effort. PRISM can also be used to gain insight into potential architectural support required to improve the reliability of GPU applications.**

I. Introduction

Due to their massively parallel compute capability, Graphics Processing Units (GPUs) have become ubiquitous in High Performance Computing (HPC). HPC workloads, such as scientific and engineering applications, typically demand high precision and correctness. GPUs are also being used in autonomous vehicles to perform a range of tasks such as pedestrian detection and avoidance, vehicle control, and visualization [3]. As the trends toward exascale computing and autonomous vehicles continue to grow, the ability of these technologies to deliver heavily depends on the reliability of hardware and software components [31], especially in safety-critical applications.

Most GPU architectures today employ a Single Instruction Multiple Data (SIMD) model. Many state-of-the-art techniques such as Error Correction Codes (ECC), Redundant Multi-threading (RMT), and other techniques have been used to improve the reliability of various hardware structures on the GPU [20], [35], [50]. However, these solutions come with significant overhead in terms of area, power, and performance. For example, Figure 1 illustrates the performance overhead of three variants of compiler-based RMT on a GPU [50]. The three variants differ in terms of the level of fault coverage (i.e., the Sphere of Replication [40]) provided on the device. *Intra-Group*-LDS RMT protects the SIMD functional units and Vector General Purpose Registers (GPRs), whereas *Intra-Group*+LDS, in addition, protects the Local Data Store (LDS). Inter-Group RMT has the largest coverage and protects SIMD units, Vector GPRs, Scalar GPRs, Scalar Unit, LDS, Instruction Fetch (IF), and Instruction Decode (ID). But

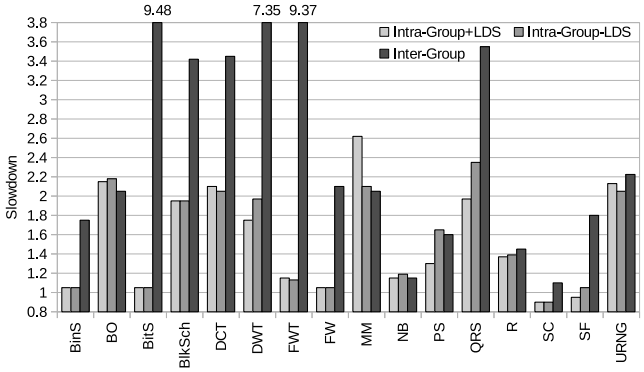


Fig. 1: Slowdown caused by Intra-Group+LDS, Intra-Group-LDS, and Inter-Group RMT, over baseline, without RMT [50]. Slowdown is calculated by normalizing the runtime of the RMT variant of the kernel by its original runtime.

clearly, Inter-Group RMT experiences a significant increase in execution time, as compared to execution using Intra-Group RMT. Despite these attempts to protect different hardware structures on GPUs, we still lack a clear understanding of how vulnerable they are in the presence of transient faults, and whether we truly need to employ such expensive solutions to protect them. Moreover, given the random nature of transient faults, predicting whether they will alter the program's output is also a challenging question.

To tackle these challenges, we build a framework named PRISM, which uses a systematic approach to predict errors in GPU programs. PRISM uses SASSIFI, a binary instrumentation tool, to generate dynamic execution and error profiles of the applications [41]. Our framework extracts micro-architecture agnostic features to characterize program resiliency. These features undergo a dimensionality reduction process in order to identify the program features that have the highest impact on program correctness. The selected features serve as predictors to drive our statistical model. Our model is trained using a diverse set of CUDA applications from a variety of application domains. PRISM provides many benefits, including:

- PRISM enables us to predict error outcomes in applications without running exhaustive fault injection campaigns on a GPU, thereby reducing the error estimation effort. The application is only required to be executed once, in order to capture its dynamic execution profile.
- PRISM can also be used to gain insight into potential hardware and software support required to improve the

SC18, November 11-16, 2018, Dallas, Texas, USA
978-1-5386-8384-2/18/$31.00 ©2018 IEEE

reliability of GPU applications. We can then design more cost-effective solutions to mitigate faults. As a result, PRISM can be deployed as an intelligent module that generates the error profile of an application before it is scheduled for execution on the GPU. Based on the error profile, the system can either recompile the application by activating optimizations such as selective instruction duplication; or enable ECC for specific hardware structures, thereby improving performance and power efficiency.
- Finally, programmers can leverage this framework to write more robust code. PRISM can guide programmers to write or choose more error-resilient algorithms. They can also insert informed 'checks' in their programs to ensure correct execution or graceful exits. PRISM will allow programmers to better understand how their coding choices translate into more reliable code.

To our knowledge, this is the first work to predict resiliency of GPU applications using statistical methods. The key contributions of this paper and the findings of our study are:

- We add new capability to the SASSIFI tool to inject faults randomly in *any* destination register during application execution. We use this feature to generate error profiles of the applications.
- We implement an analysis pass in the profiling phase of SASSIFI to identify scalar and vector instructions. We characterize the GPU workloads based on the hardware resources they stress during execution to accurately capture their behavior. We use these parameters as our *feature set*.
- We identify interaction between features, and perform feature selection based on their contributions to program correctness. We use this reduced feature set to drive our model.
- A key property of PRISM is its flexibility that allows users to plug in their choice of regression model. In this paper, we explore two prediction models - one based on Linear Regression and the other based on similarity analysis.
- Using Linear regression, PRISM is able to predict Masked and Unmasked errors with an accuracy of 90% on our test applications. We also identify that Floating Point instructions contribute significantly towards masking, whereas Integer Arithmetic and a set of Scalar Instructions are the biggest contributors to Detected and Unrecoverable Errors (DUEs). We also provide insights into potential architectural modifications and prospective research that could be performed as a result of our analysis.

The remainder of this paper is organized as follows. In Section II, we describe the fault models used in our work and some of the fundamentals of NVIDIA's Kepler GPU architecture. In Section III, we describe the PRISM framework in detail and report the accuracy of both models. We then review prior work done to estimate the reliability of GPU programs in Section V and present our conclusions in Section VI.

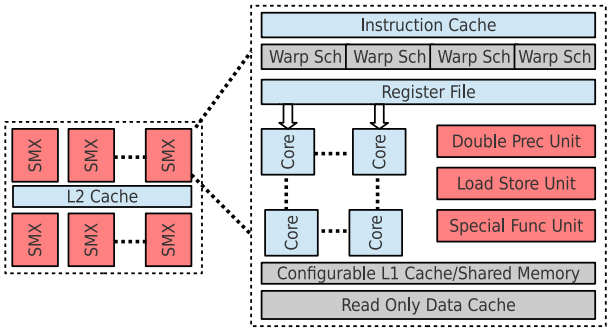


Fig. 2: Overview of NVIDIA Kepler GK110 architecture.

II. Background

A. Fault Modes

Reducing transistor sizes and operating voltage levels make circuits more susceptible to transient and permanent faults. Manufacturing defects, thermal stress, and circuit aging are just a few of the factors that cause permanent faults on a device. Alternatively, transient faults are caused by temperature and voltage variations, electromagnetic interference, crosstalk, and high energy particles in the atmosphere. An alpha or neutron particle strike may be sufficient to invert the state of a logic gate or memory cell, and may drive a wrong value temporarily [33]. Since these faults are temporary, they do not reoccur when the operation is re-executed in the future. These temporary upsets in a transistor's state are called single-event upsets (SEUs) or, if more than one transistor is affected, single-event multi-bit upsets (SEMUs). Although a *fault* causes an undesired change of state in the hardware, it may or may not cause an *error* in the outcome of the program. For instance, if the location that was impacted by a transient fault is never read by the program, or was over-written by a subsequent operation before the faulty value was read, the fault will not manifest into an error. The fault may also be corrected by a redundancy mechanism, such as ECC, before it propagates through the application and produces an undesirable output. When a fault does not manifest into an error, it is said to be *Masked*.

Other possible outcome categories are *Detected and Unrecoverable Errors* (DUEs) or *Silent Data Corruptions* (SDCs). DUEs occur when a system is able to detect an error and was unable to recover from it. This could happen when a fault causes the program to take an incorrect execution path which results in a system hang, crash, or other unexpected behavior. For instance, a fault may alter an address and cause the user program to access an unallocated memory location. SDCs occur when a faulty bit is used by the program, and which results in the wrong output. Error rates in this domain are commonly measured using Failures-in-Time or FIT rate, which is the expected number of failures in 10^9 hours of operation.

B. GPU architecture

Next, we describe the NVIDIA Kepler architecture, given that we have chosen it to serve as our evaluation platform [38].

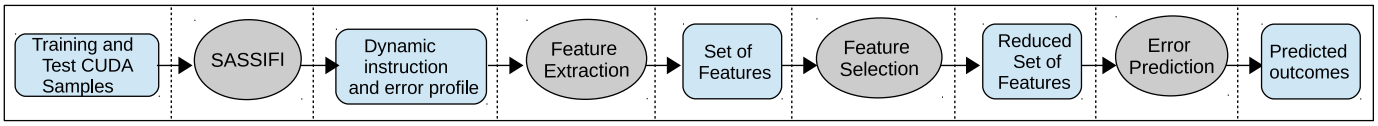


Fig. 3: The PRISM Framework. The ovals represent the processing nodes/phases, whereas the rectangles represent input/output to/from the processing nodes.

Streaming Multiprocessors (SMX) are the fundamental units of computation on NVIDIA GPU architectures, as shown in Figure 2. The smallest unit of execution is a *thread*, which executes on one CUDA core of an SMX on the device. There are 192 CUDA cores per SMX on the Kepler architecture. Each thread has access to 255 registers. Each thread can only access its own private register file, but register values can be shared with other threads via special instructions. Each CUDA core is equipped with a floating point and integer arithmetic and logic unit (ALU). The SMX schedules work in groups of 32 parallel threads, called *warps*. Threads within a warp execute in a Single Instruction Multiple Data (SIMD) fashion. Each SMX has four warp schedulers and eight instruction dispatchers, which allows four warps and eight independent instructions (two per warp) to be issued and executed concurrently. An instruction is said to be *scalar* if all active threads in a warp operate on the same data, otherwise it is *vector*.

Each SMX has 64 KB of on-chip memory that can be configured as 48 KB shared memory with 16KB of L1 cache, or 32 KB of shared memory with 32 KB of L1 cache, or a 16KB/48KB split between shared memory and L1 cache. In addition to L1 cache, Kepler has a 48 KB cache for read-only data, and an L2 cache which serves as the primary point of data unification between the SMX units. Kepler's register file, shared memory, L1 cache, L2 cache, and DRAM memory are protected by a Single-Error Correct Double-Error Detect (SECDED) ECC code, whereas, the Read-Only Data Cache supports single-error correction through a parity check. Programs that run on NVIDIA GPUs are written in the CUDA C/C++ language and compiled using NVIDIA's LLVM-based CUDA Compiler, *nvcc* [36]. We run our fault injection campaign at the SASS level, which is a low-level assembly language for NVIDIA GPUs.

C. Statistical Terminology

We next review the statistical terminology that we will use throughout this paper [25].

- The term *sample* refers to a single, independent unit of data. In our study, each CUDA application is a *sample*.
- The *training set* consists of samples used to develop a model, while the *test set* contains samples used solely for evaluating the performance of the model. It must be noted that training and test sets are mutually exclusive.
- The *predictors* are the independent variables that are used as input for the prediction equation. In our study, predictors are the program features that we derive through dynamic profiling.

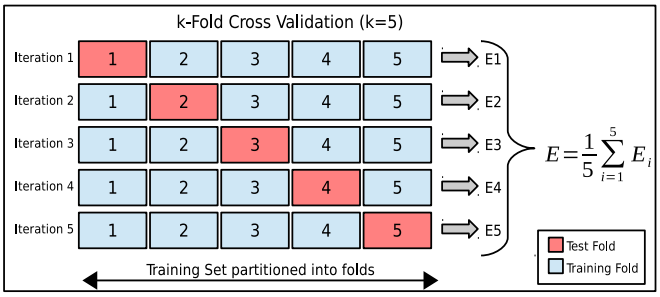


Fig. 4: Demonstration of 5-fold Cross Validation (CV) technique. CV is used to prevent the problem of overfitting.

- *Outcome* refers to the outcome event that is being predicted. Our focus in this paper is to predict resiliency of GPU applications, therefore our *outcomes* are categorized into Masked, DUE, and SDC.
- *Model Training* or *Parameter Estimation* refers to the process of using data to determine the values of model equation.
- *Cross Validation* is a technique to generalize the model to an independent data set. Training and testing the model on the same data will give it a perfect score, but will perform poorly on yet unforeseen data. This situation is called *overfitting*. The solution to this problem is to use cross-validation (CV). We use a k-fold CV approach, in which the training set is divided into k smaller subsets or *folds*, as shown in Figure 4. The model is trained on $k-1$ of the folds and tested on the remaining part of the training data. This process is repeated for each of the k-folds. The final performance is measured by taking the average of the values computed in the loop. In our work, we use 5-fold cross validation to prevent overfitting.

III. PRISM Framework

PRISM framework includes four different phases, as shown in Figure 3. In this section, we will describe each phase in more detail. The first step is to collect samples required for our study. We have selected a diverse set of regular and irregular applications from a variety of domains, as shown in Table I. These workloads have been taken from the CUDA SDK, Lonestar, NUPAR, Parboil, and Rodinia benchmark suites [9], [26], [37], [46], [49]. All of the applications that we were able to support with the SASSIFI tool were included in this study. We modified the source code for several applications because they were tuned for performance benchmarking. For example: some applications had a warm-up pre-execution of the kernel to avoid cold start misses. Any error that occurs in the warm-up kernel will never be captured as the output is usually

Domain	Application
Linear Algebra	Vector Addition (vadd), Single Precision Floating General Matrix Multiplication (sgemm), Matrix Transpose, Scalar Product (sProd), LU Decomposition (lud), Gaussian Elimination, Scan
Graph Traversal	Breadth-First Search (bfs), Survey Propagation (sp), Pathfinder, Minimum Spanning Tree (mst)
Image Processing	Magnetic Resonance Imaging - Gridding (mri-g), Saturating Histogram (histo), MRI-Q, Sum of Absolute Differences (sad), Leukocyte, HeartWall, Speckle Reducing Anisotropic Diffusion (srad)
Physics Simulation	Hotspot
Thermodynamics	Stencil
Fluid and Molecular Dynamics	Lattice-Boltzmann Method (lbm), Computational Fluid Dynamics (cfd), LavaMD
Signal Processing	Infinite Impulse Response (IIR), Discrete Walsh Transform 1D & 2D (dwt1D&2D), Fast Walsh Transform (fwt)
Financial Computation	BlackScholes (blkSch), Binomial Options (binOpt), SobolQRNG (SQRNG)
Electromagnetics	Finite-difference Time-domain (FDTD-3D)
Data reduction and Sorting	Merge Sort (mSort), Radix Sort (rSort), Hyrbid Sort (hSort), Reduction
Data Mining and Pattern Recognition	Kmeans, Nearest Neighbor (nn), Back-propagation (backprop)
Bioinformatics	Needleman-Wunsch (nw)
Astrophysics	Barnes Hut (bh), Two Point Angular Correlation Function (tpacf)

TABLE I: Applications used in our training and test samples, along with their domains.

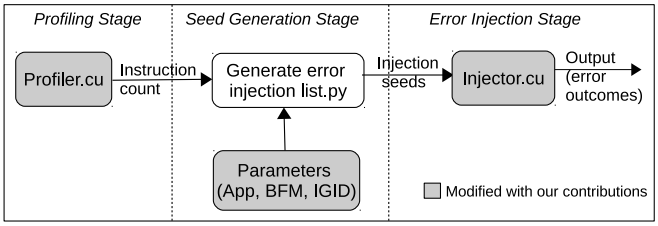


Fig. 5: Different stages in the SASSIFI tool.

over-written by the actual kernel execution. We, therefore, eliminated all warm-up kernel executions. For checking errors, we compared the output of the kernel when the fault is injected with the golden output (without any fault injection). For precision-based applications, we used the default values of the L1 and L2-Norm provided in the benchmark, which were typically around 1e-6. We instrument these application binaries with the SASSIFI handlers, which we will describe in the next section. Our study has been done on a live NVIDIA Kepler K20 GPU.

A. SASSIFI

The SASSIFI tool is based on SASSI, which is a dynamic instrumentation tool for GPUs, similar to the Pin tool for CPUs [32], [41]. SASSI instruments the SASS code at runtime by linking user-written instrumentation handlers with the application binary [45]. SASSI instrumentation handlers are written in CUDA C/C++ and can be inserted before, after, or both before and after an instruction executes.

SASSIFI mainly comprises three stages, as shown in Figure 5. In the first stage *(Profiling Stage)*, the profile handler profiles the applications and generates a population count of different instruction groups (IG). It is important to note that this handler is inserted *before* the instruction executes, which allows us to scan the values of its source registers (the importance of this is described in the next section).

This profiling information is processed by the *Seed Generator* that generates a statistically significant number of injection seeds using a uniform random distribution. A *Seed* specifies the location where the fault will be injected. It is a combination of kernel ID, kernel invocation ID, and dynamic instruction ID of the specified instruction type. Each seed is equivalent to one fault. To generate the seeds, the Seed Generator requires other information from the user, such as injection mode, bit flip model (BFM), Instruction Group ID (IGID), and the number of injections. Our selection of these parameters uses the following process:

- **Injection mode:**
  SASSIFI currently supports three injection modes - Register File (RF), Instruction Output Address (IOA), and Instruction Output Value (IOV). RF mode supports the injection of faults randomly across all registers that are used by the program. IOA mode supports error injection into the register indices and store addresses. Lastly, IOV support injections into the destination register of an instruction after it has executed. In our study, we use IOV mode to inject errors in the instructions.
- **Instruction Groups:**
  SASSIFI identifies different types of instructions and allows the user to select them in order to study how the errors injected propagate to an application output. Some examples of Instruction Groups include:
  GPR - Instructions that write to a destination register,
  ST - Store Instructions,
  CC - Instructions that write to a condition code,
  PR - Instructions that write to Predicate Registers, etc. We added capability to SASSIFI to inject a fault randomly in any kind of destination register (GPR, Predicate, or CC). Before this modification, a user could select only one of the three kinds of destination register during a fault injection run.
- **Bit Flip Model (BFM):**
  For IOV mode, we use Single Bit Flip mode in which one bit in one register in one thread is flipped.
- **Number of Injections:**
  Here, we specify the number of unique seeds that we would like to generate for our fault injection campaign.

Finally, in the *Injector Stage*, we perform error injection based on the seeds generated by the previous stage. The injector keeps track of the kernel ID, kernel invocation ID, and Instruction Group ID, and when their value matches with the seed, a fault is injected. We generate 1000 uniformly random

seeds while injecting one seed per run. The results have an error margin of 3.1%, with a 95% confidence level [28]. Unlike the profiler handler, the injector handler is inserted *after* the instruction executes and flips a bit in the destination register, implying that a fault has occurred in the datapath of the instruction. Since we only inject faults in the instruction outputs, our analysis takes into account only the live architectural state. Once a fault injection is done, the outcome of the program can fall into one of the three categories - Masked, DUE, or SDC, as described in Section II-A. We perform our fault injection campaign on a real NVIDIA Kepler K20 GPU.

B. Feature Extraction

Once injections are run on all applications, SASSIFI generates the instruction and error profile for each application. In this section, we describe the type of instructions that we use to characterize application behavior. These instructions allow us to capture the overall workload stress on the underlying microarchitecture.

- *Data Movement Instructions*: are responsible for moving data between registers, such as MOV, SHFL, etc.
- *Integer Arithmetic Instructions*: perform arithmetic instructions on integer data type.
- *Floating Point Instructions*: perform computation on floating point data types.
- *Logic Instructions*: comprised of logical operations, such as AND, OR, and shift operations.
- *Load Instructions*: load data from global, shared, constant, and texture memory.
- *Store Instructions*: store data into the global, shared or local memory.
- *Control Flow instructions*: branch and jump instructions that determine the control flow of the program.
- *Predicate Instructions*: for example ISETP and FSETP, instructions that write to predicate registers.
- *Kernel Register Usage*: the number of general purpose registers used by the kernel.

As we mentioned earlier, most GPU architectures today employ a Single Instruction Multiple Data (SIMD) model. Prior work has shown that a significant portion of SIMD instructions demonstrate scalar characteristics (i.e., all the active threads operate on the same data) [10], [27]. In other words, an instruction is said to be *scalar* if all active threads in a warp operate on the same input data, otherwise it is *vector*.

Based on this differentiation, we implement an analysis pass in the SASSIFI profiler to identify dynamically scalar SASS instructions. To achieve this, we check the value of all source operands and the operation performed on those operands (opcode) for every SASS instruction. This check must be done *before* every instruction because their values might change after the instruction has executed.

As shown in Algorithm 1, we first select a leader thread by using two CUDA intrinsic functions - __ffs() and __ballot(). Threads within a warp are also called *lanes*; the simplest way to elect a leader is to use the active lane with the lowest number. ballot() returns the active mask (1 for an active lane and 0 for an inactive lane). ffs() returns the 1-based index of the lowest set bit in the active mask. Subtracting 1 gives us a 0-based index of the lowest active lane id, which is our leader thread. Once we identify the leader, all threads first scan one source register used by the SASS instruction. The value of this source register is *shuffled* across all threads in the warp. The SHFL instruction allows a thread to read a register from another thread in the same warp, without using shared memory. If the value of the source register is the same across all threads in the warp, then the register is marked as *scalar*. This process is repeated for all source registers. If all source registers used by the instruction are found to be scalar, the leader thread marks the instruction as scalar ($S \leftarrow S \cap S$). Even if one of the operands is found to be vector, an instruction is marked as *vector* ($V \leftarrow V \cap S$, or $V \leftarrow V \cap V$)[1]. Atomic operations are, by default, marked as vector, whereas unconditional control flow instructions are marked as scalar. This is done for all dynamic SASS instructions in the program. We record scalar and vector instances of every opcode using appropriate counters.

Algorithm 1 Pseudo-code for detecting scalar instructions in the kernel. All threads execute this code concurrently before executing each SASS instruction.

```
1: Input: SASS instruction
2: Output: Instruction Scalar or Vector
3: int threadIdxInWarp = get_laneid();
4: int firstActiveThread = (__ffs(__ballot(1))-1);          ▷ leader
5: for all Source Registers R_i in an instruction do
6:     GPRRegValue regVal = GetRegValue(R_i);
7:     // shuffle the leader's value across all threads
8:     int leaderValue = __shfl(regVal.asInt, firstActiveThread);
9:     // true when all threads' value equal to leaderValue
10:    int allSame = (all(regVal.asInt == leaderValue) != 0);
11:    // warp leader writes the results
12:    if threadIdxInWarp == firstActiveThread then
13:        is_scalar &= allSame;
14:    end if
15: end for
16: if is_scalar == 1 then
17:    atomicAdd(&CountersInstType[SCALAR], 1)
18: else
19:    atomicAdd(&CountersInstType[VECTOR], 1)
20: end if
```

If there are multiple executions of either the same kernel or different kernels within the same application, the values of dynamic scalar, vector, and total instructions are averaged out across the kernel runs. Given the nature of GPU applications, one of the challenges with multiple executions is that an instruction which is scalar during one instance may not be scalar in the next instance. Our algorithm is able to capture this dynamic behavior of every instruction.

Next, we derive metrics to quantify the kernel characteristics using the instruction mix and their scalar/vector instances. Metrics such as Integer Arithmetic Intensity, Floating Point Intensity, and Logic Intensity give us an insight into the usage of different functional units on the GPU. Control Flow Intensity, as the name suggests, allows us to capture the control

[1] S and V represent Scalar and Vector operands, respectively.

Metric	Kind	Synopsis	Example of opcodes included [2]
Control Flow Intensity	Scalar	Total number of dynamic Control Flow Instructions/N	BRA, JMP, BRX, JCAL
Data Movement Intensity	Scalar, Vector	Total number of dynamic Data Movement Instructions/N	MOV, SHFL, PRMT
Floating Point Intensity	Scalar, Vector	Total number of dynamic Floating Point Instructions/N	FADD, FMUL, FMAD
Integer Arithmetic Intensity	Scalar, Vector	Total number of dynamic Integer Arithmetic Instructions/N	IADD, IMUL, MAD
Logic Intensity	Scalar, Vector	Total number of dynamic Logic Instructions/N	LOP, SHL, SHR
Load Intensity	Scalar, Vector	Total number of dynamic Load Instructions/N	LD, LDS, LDC, LDG
Predicate Intensity	Scalar, Vector	Total number of dynamic Predicate Instructions/N	ISETP, FSETP, PSETP
Store Intensity	Scalar, Vector	Total number of dynamic Store Instructions/N	ST, STS, STL
Register Usage	-	Number of General Purpose Registers used by the kernel	All opcodes
Scalar Intensity	Scalar	Total number of dynamic scalar instructions/N	Scalar instances of all opcodes
Vector Intensity	Vector	Total number of dynamic vector instructions/N	Vector Instances of all opcodes
Total features	**18**		

TABLE II: Description of metrics derived using kernel characteristics. We use these metrics as features in our model. N = Total number of dynamic instructions executed by the application.

flow graph, which plays an important role in determining how an error propagates through the kernel. Information about thread level resource utilization is retrieved using the kernel register usage metric. Memory type based classification is captured using load and store intensities. These metrics are summarized in Table II.

C. Feature Selection

In this phase, we perform *feature selection*, with the goal of selecting a subset of relevant features without incurring much loss in information. Feature selection simplifies our model and makes it easier to comprehend by researchers/users. It also reduces the training and data acquisition time. Fewer features increase the generality of the model and prevent overfitting. There are different ways in which a user can minimize the number of features. One way is to eliminate features that are of little or no interest to the user. For example, if the user does not wish to distinguish between scalar and vector instances in their study, they can combine Scalar and Vector Intensities for all features. Hence, Floating Point Intensity will now be the sum of Scalar Floating Point Intensity and Vector Floating Point Intensity, Integer Arithmetic Intensity will be the sum of Scalar Integer intensity and Vector Integer Intensity, and so forth.

In this paper, we use a forward selection wrapper method to select a subset of relevant features [24]. This method begins with no features in the model, and on every iteration, adds the feature which best improves the performance of the model. This continues until adding a new feature no longer improves the performance of the model. The advantage of using a wrapper method is that it is able to detect possible interaction between features during feature selection. Other options include filter methods, such as Normalized Mutual Information (NMI) and Pearson Correlation Coefficient [23], [54]. Although filter methods are computationally less intensive than wrappers, they have lower prediction performance than wrappers because they are not tuned for any specific model [56]. Latent factor based dimensionality reduction techniques, such as Factor Analysis (FA) and Principal Component Analysis (PCA), and supervised techniques, such as Partial Least Square (PLS), could also be used. A caveat with using FA or PCA is that they do not take into account the labels/error outcomes during feature reduction. They generate a single set of features for all outcomes, which might work well in predicting one kind of outcome, but not for another. Since we have three possible outcomes, a separate feature set for each outcome might provide better accuracy. We describe the selected features after introducing the models in the next section.

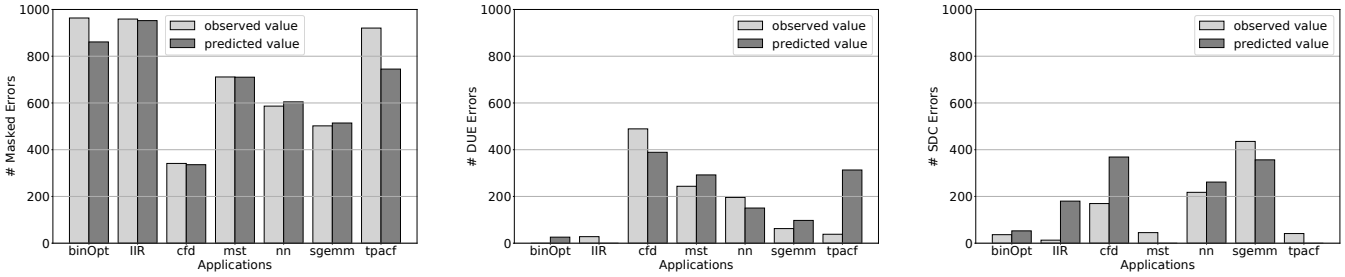
Outcome	Ridge (Coefficients)	K-Nearest Neighbor
SDC	Vector Predicate (-0.59) Scalar Float (-0.27) Register Usage (-0.24)	Vector Predicate Vector Integer Register Usage Vector Logic
DUE	Vector Float (-0.42) Scalar Float (-0.28) Scalar Logic (0.26) Scalar Store (0.28)	Vector Float Vector Store Scalar Logic Vector Predicate
Masked	Vector Float (0.47) Scalar Float (0.43) Vector Predicate (0.49)	Vector Float, Scalar Store, Scalar Load, Vector Predicate, Vector Integer, Vector Load, Scalar Float, Scalar Move

TABLE III: Features selected using the forward selection wrapper method for both Ridge Linear Regression and K-NN. The coefficient of the feature is provided in parentheses for the Ridge Regression model. K-NN is a non-parametric model, hence does not require coefficients.

D. Error Prediction

Statistical Regression Analysis is a set of techniques to estimate the relationship between a single dependent variable and multiple independent variables. In this paper, we explore two models: Ridge Linear Regression and K-Nearest Neighbor [11], [18]. As mentioned earlier, the wrapper method for feature selection is tuned for a particular model. Therefore, the set of features selected for Linear Regression and K-NN are different. Out of the 50 CUDA samples studied, we randomly select 43 samples (85%) for training and 7 samples (15%) for testing our model. The wrapper executes only on the training set and uses 5-fold cross-validation to select features having the highest correlation with the error outcomes. We summarize the features for both models in Table III.

1) Model 1: Linear Regression: Among the various forms of regression analysis, Linear Regression is most commonly used due to its well-behaved and well-studied properties [4].



(a) Masked **(b) DUE** **(c) SDC**

Fig. 6: Observed and Predicted values of Masked, DUE, and SDC outcomes using the Ridge Regression Model. The accuracy of prediction for Masked errors is 90%. For DUEs, with the exception of *tpacf*, our predicted values are close to the observed values.

The model can be stated as follows:

$$y = b_0 + b_1x_1 + b_2x_2 + ... + b_ix_i + e_i \quad (1)$$

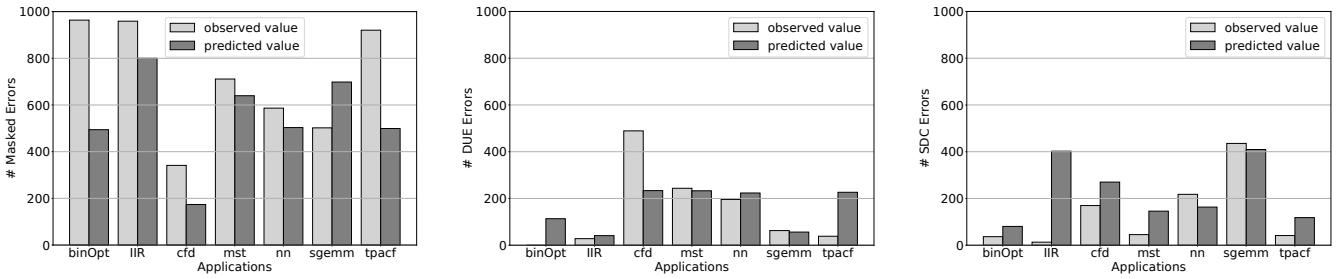
The terms x_i are the independent variables. Their changing values cause the dependent variable, y, to vary as a response. The terms b_i are the parameters or *regression coefficients*. e_i represents error, which is derived by comparing the predicted and observed values of y. This model is *linear* because no parameter appears as an exponent or is multiplied/divided by another parameter. The goal of the regression is to accurately predict the values of the coefficients, b_i, from the observed measurements of x_i and y. Given that the focus in this paper has been GPU error modeling, y then represents error outcomes, and x_i are the kernel characteristics such as Integer Arithmetic Intensity, Register Usage, etc. Once a model and its coefficients are proposed, we use Normalized Root Mean Square Error (NRMSE), a commonly-used statistical metric, for measuring the quality of the model [42].

Next, we apply Ridge Regression, a type of linear regression model, on our test samples using the selected features, and report our results in Figure 6. For Masked, we observe that Ridge is able to predict masking with a NRMSE as low as 10%, which means its prediction accuracy is 90%. The coefficients for Scalar Float, Vector Predicate, and Vector Float were found to be 0.43, 0.49, and 0.47, respectively. In the case of DUEs, the overall predicted values are close to the observed values, except for *tpacf*, which is an outlier. The coefficients for DUEs were found to be -0.42, -0.28, 0.26 and 0.285 for Vector Float, Scalar Float, Scalar Logic, Scalar Store, respectively. For SDCs, the model seems to be reasonably accurate for a few applications, but has a couple of outliers such as *IIR* and *cfd*. The coefficients for the predictors were found to be -0.59, -0.27, -0.24 for Vector Predicate, Scalar Float, and Register Usage, respectively. We summarize the key takeaways from this information as follows:

Key takeaways:

- In Masked, the coefficients for the three selected features were found to be quite uniform. This means all three features contribute almost equally towards Masking. Note that two out of three features are floating-point intensive. This suggests that floating point intensity plays a significant role in masking errors. The level of masking may vary, depending on the value of the L1 or L2 Norm selected by the user in the application. For floating point intensive applications, the L1 or L2 Norm could be added as an additional feature, or a sensitivity analysis could be done by varying their precision, but this is beyond the scope of this work.
- For DUEs, three out of four features were identified as Scalar, which means Scalar instructions seem to be significant contributors to DUEs. Also, the coefficients for floating point features were found to be negative. Lower floating point intensity could conversely imply higher integer intensity. Hence, it could be said that faults in integer operations are more likely to result in a DUE.
- Moreover, from our experimentation, we observed that most DUEs are generated when an instruction tries to access an illegal memory location. This happens when the register used for holding or computing a memory address gets corrupted. Therefore, identifying and replicating instructions that feed into load/store instructions could assist in detecting sources of DUEs. These instructions can be identified using static data-flow analysis, such as *reaching definitions*, to create use-def chains [5].
- A linear model for SDCs does not perform as well as it does for predicting Masked errors. A possible explanation for this could be that a linear relationship is insufficient to predict SDCs, and there could be some underlying non-linearity that must be exploited. However, if the user is interested in predicting the number of unmasked errors (SDC+DUE) for his/her application, they could use (1000 - # of predicted Masked errors) as an indirect way to estimate Unmasked errors, since a linear model predicts masking quite accurately.

2) Model 2: K-Nearest Neighbor: Next, we apply a popular non-parametric model called K-Nearest Neighbor (K-NN). Our hypothesis is that similar applications might show similar resiliency behavior. The rationale behind choosing K-NN to explore similarity between applications is that it does not make any assumption about the underlying distribution of the data,



(a) Masked **(b) DUE** **(c) SDC**

Fig. 7: Observed and Predicted values of Masked, DUE, and SDC outcomes using a K-Nearest Neighbor Model. The value of K was found to be 4, 3, and 2, respectively.

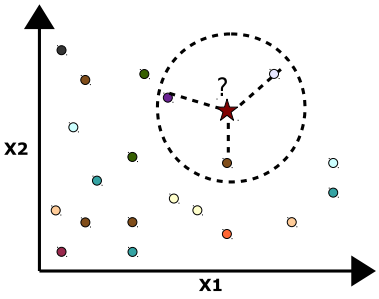


Fig. 8: Demonstration of K-Nearest Neighbor (K = 3).

which makes it very robust.

Using K-NN, an application's resilience can be predicted by using its K closest neighbors. Here, 'K' represents the number of neighbors (here, training samples) closest to a test sample that will be used to make a prediction for that test sample. The closeness between the test and training sample is measured by using Euclidean distance metric. To illustrate this model, all samples correspond to points in an n-dimensional feature space, as shown in Figure 8. For simplicity, we only use two dimensions/features, X_1 and X_2. To make a prediction for a test sample, we first locate its three nearest training samples (i.e., K=3). We predict the outcome for the test sample by using an average of the values of its three nearest neighbors. The value of K is chosen through cross validation. Figure 9 provides a visual representation of our intuition. We use Cosine similarity on the feature set to generate this application similarity heatmap. In the figure, shades of red show similarity between applications, whereas shades of blue represent dissimilarity. We try to leverage this similarity information in our methodology.

We use the features selected by the forward selection wrapper for K-NN to compare similarity between applications. The value of K was found to be 4, 3, and 2 for Masked, DUEs and SDCs, respectively, through cross-validation. Figure 7 shows the accuracy of K-NN Model for all three outcomes. One can notice that Ridge Regression clearly provides better prediction accuracy for Masked than K-NN. For Masked, the value of K is 4, which means that every sample must find four similar samples in its vicinity. If the four chosen samples are not similar, it may end up smoothing things out too much and eliminating some important details in the

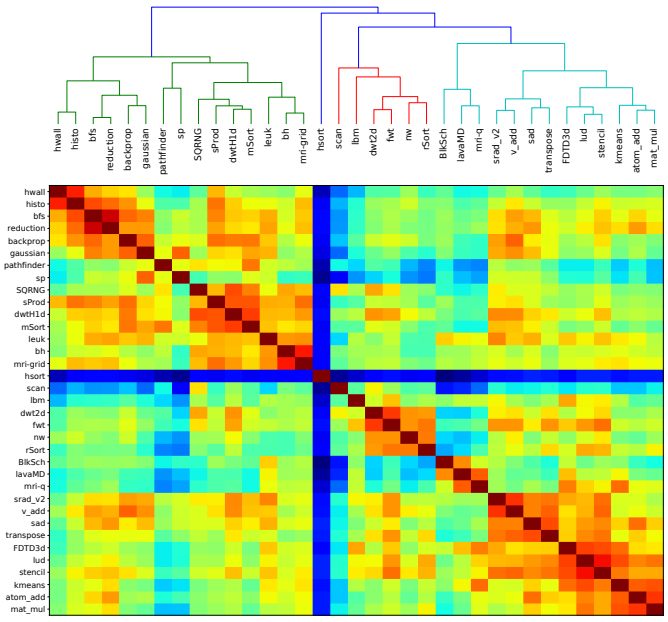


Fig. 9: Visualization of similarities between applications using a dendrogram at the top. In the heatmap, Red represents similarity, whereas Blue represents dissimilarity. In the dendrogram, the vertical distance is a measure of similarity. Shorter distance means more similarity.

distribution. Moreover, K-NN uses 8 selected features to check similarity for Masked outcomes. As the number of features/dimensions increases, the sparseness of the training data in the n-dimensional feature space also increases. This causes the distance metric to lose significance as it becomes difficult to accurately identify the neighbors. This phenomenon is also known as the *curse of dimensionality* [7]. A combination of the above two reasons might impact the prediction accuracy for Masked outcomes.

In case of DUEs, K-NN has accuracy comparable to Ridge. Besides the *cfd* and *tpacf* applications, the model predicts the outcomes quite accurately. This suggests that these two outliers were not able to find three similar neighbors. A solution to this problem is to increase the size of the training data. While there are small clusters in the heatmap, having a large sample space might create more concrete and dense clusters. This can eventually result in a better prediction for

K-NN. A small value of K is more susceptible to noise, as might be the case seen in SDC prediction. Finding the right value of K is also a challenging research problem.

Key Takeways:

- Overall, we find that K-NN does not work as well as Ridge. This may be a side effect of working with a small training sample space (43 CUDA applications), as K-NN has a tendency to perform better with large data samples. In addition, if the number of selected features is large as compared to the sample space, then K-NN might suffer from the *curse of dimensionality*, as observed in the case of Masked outcomes.
- However, K-NN is a more robust model as it does not rely on the underlying distribution of the data; unlike a Linear model which assumes a linear relationship between features and outcomes. We anticipate that the results for K-NN would improve if we increased the number of training samples. Hence, it might be worth revisiting K-NN by supporting more CUDA applications on SASSIFI. The user must consider the above trade-offs and choose a model which best suits their data.
- Both Ridge and K-NN select register usage as one of the features impacting SDCs. A negative coefficient of Ridge could imply that increasing the number of registers could reduce the number of observed SDCs. This could make for an interesting case to study the impact of compiler optimizations, such as loop unrolling, on the resiliency of the application.

IV. Discussion

A. Support for Other Modern Architectures

Modern GPUs exploit data parallelism in application kernels to achieve high performance and efficiency. However, there can be a loss in efficiency due to redundant execution whenever threads perform the same operations on the same data. Scalar instructions reduce energy by eliminating replicated work. Moreover, scalarization reduces overall register file capacity by eliminating redundant operand storage, or additionally allows a register file of a given size to hold the context of more threads. The execution of the scalar instructions is enabled by architectural and microarchitectural support provided next to the parallel datapaths. For example: AMD's Graphics Core Next (GCN) architecture has a scalar co-processor in each compute core, along with a separate scalar register file [1]. A warp (or *wavefront*, in OpenCL terminology), is mapped across the SIMD lanes with one thread per SIMD [44]. In contrast, the scalar unit has a single lane with a scalar register file to execute scalar instructions. To support a GCN-like architecture (as shown in Figure 10) in PRISM, the count of scalar instructions must be recorded only once per warp, whereas the count for vector instruction depends on the number of active threads in a warp executing that instruction. This distinction must be accounted for while calculating scalar and vector instances, as described in Algorithm 1.

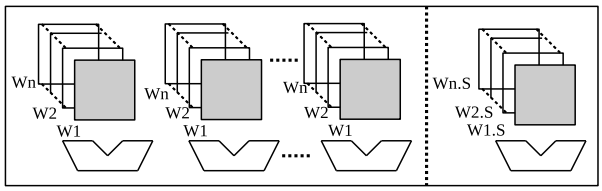


Fig. 10: Spatial SIMT Architecture with a separate Scalar Unit.

In our study, we have used Kepler because the SASSIFI tool is optimized for the CUDA v7.0 software stack and a compute capability of 3.5. To take full advantage of architectures supporting a compute capability greater than 3.5 (e.g., Pascal and Volta), SASSIFI must first be optimized to run on these newer architectures. Unlike Kepler, Pascal and Volta have an extended Instruction Set (ISA) with explicit support for half-precision floating point instructions. These additional opcodes must be incorporated in the intensities, described in Table II, for PRISM to provide better accuracy. Adding half-precision FP instructions might impact the high masking effects of floating point instructions that we observed in the Ridge Regression Model. This requires further investigation in order to understand the contribution of half-precision floating point instructions to program correctness, which we plan to explore in our future work.

V. Related Work

This study spans both the fields of GPU architecture/reliability and machine learning. We identify and present three categories of research related to our study. The first category includes the studies on prediction of system vulnerability. In the second category, we review the body of work dedicated to comprehensive studies of vulnerability and the analytical measurement of vulnerability. Finally, in the third category, we will present the studies that have tried to characterize GPU applications.

A. Vulnerability Prediction

Numerous studies have been conducted on the prediction of vulnerability. These studies propose methods to estimate the vulnerability of a system during its execution, and is called *online vulnerability*. Such a technique allows a system administrator to adapt any vulnerability protection scheme to the current vulnerability state of the system.

X. Li *et al.* use tainted analysis on microarchitectural registers to approximate the effects of faults injected into these registers [30]. The taint analysis is able to identify all the detectable faults in registers that would later be used in stores, branches or system calls. Fu *et al.* proposed a correlation between vulnerability of core microarchitectural structures and performance counters [15]. Walcott *et al.* [51], Biswas *et al.* [8], and Duan *et al.* [12], propose training-based models that use performance variables to estimate Architectural Vulnerability Factor (AVF) at runtime.

As these studies only estimate vulnerability at a hardware level, they do not take into account the resilience properties from a programming perspective. Wibowo *et al.* use a cross-layer approach which accounts for, not only the

microarchitecture-level vulnerability, but also for the inherent resilience present in the algorithms being executed [53]. Moreover, Farahani *et al.* dynamically predict the vulnerability of a program during its execution [14]. They utilize machine learning to predict program vulnerability. Their algorithm learns from performance features at both architecture and microarchitecture levels.

All of these approaches differ from our work in that:

1) The vulnerability is evaluated at different layers of the system stack, while our work only targets the ISA level. Moreover, they predict vulnerability at runtime in order to allow a dynamic vulnerability protection scheme to save energy, while only enabling soft error protection when necessary. Since this kind of support is not available in many systems, a developer would need to be responsible for managing resilience of their own programs. With our framework, a programmer is able to evaluate the robustness of his/her program, regardless of the hardware that it is running on.
2) They target CPU applications, while our work focuses on GPU applications. The GPU architecture implements an in-order SIMD pipeline, with thousands of threads concurrently executing. Given the execution model of the GPU, error propagation within a GPU application is different from a CPU. It is possible for errors to propagate across threads in a block, or across invocations of a GPU kernel. Furthermore, the control flow changes in GPU applications are minimized, so that the application can fully take advantage of the parallel hardware. The features that we consider in our work, as described in Table II, are suited for GPU applications.

B. GPU Vulnerability

Recently, there have been a number of studies that have considered GPU reliability, given the growth in popularity of GPUs in HPC and safety-critical applications [39], [48]. In addition, Li *et al.* have investigated the propagation of errors across GPU kernel calls using fault injection experiments [29]. They have also developed a comprehensive fault injection tool, LLFI-GPU, which allows users to perform fault injection experiments at the intermediate assembly level.

Program Vulnerability Factor (PVF) is a vulnerability metric that only considers program level effects on vulnerability [43]. PVF can be measured with either statistical fault injection or through Architecturally Correct Execution (ACE) analysis [34]. ACE analysis systematically identifies state in a program structure (such as the architectural register file) that is necessary for correct execution of the program. Because ACE analysis conservatively assumes that all bits in an architectural structure are important until proven otherwise, the vulnerability estimation obtained from ACE analysis is often overestimated [52].

Studies have measured GPU application resilience using the PVF or related metric using both fault injection [13], [17], [41] and ACE analysis [19], [47]. Our work aims at simplifying the process of vulnerability estimation, avoiding lengthy and exhaustive fault injection experimentation, as well as the inaccuracy and overestimation of the ACE analysis.

C. GPU application characterization

As GPUs have gained popularity in high performance computing domains, many studies have tried to characterize performance of GPU workloads. Kerr *et al.* introduce a set of metrics for GPU workloads and utilize these metrics to analyze the behavior of GPU workloads [22]. Goswami *et al.* propose a set of microarchitecture-agnostic workload characteristics to capture the behavior of GPU applications [16]. These study characterization of GPU applications in terms of their performance. Our work focuses on characterization based on the reliability of GPU applications.

Kalra *et al.* quantify the linear correlation between the proportion of scalar instructions and different outcomes. However, their work solely focuses on understanding the vulnerability of scalar and vector opcodes, and does not predict the resilience of applications [21]. Fang *et al.* introduce a fault injection methodology, and present some error resilience characteristics of GPU application kernels based on the results observed from a fault injection study [13]. They found that program behavior can influence application resilience. They categorized the applications according to their respective patterns of computations. Their categorization follows the 13 dwarfs of parallelism, as presented by Asanovic *et al.* [6]. However, their work is focused on understanding the resilience of GPU applications, and does not use the categorization for resilience prediction.

To our knowledge, our work is the first to make use of microarchitecture agnostic features to predict vulnerabilities in GPU applications. Unlike prior work, we approach the problem of reliability from a non-traditional perspective by bringing together statistical learning methods to predict failure in applications.

VI. Conclusion

In this paper, we propose a framework named PRISM to predict resiliency of GPU applications. As part of PRISM, we explore two prediction models based on different hypotheses. Linear regression model tries to capture any linear relation between program characteristics and outcomes, whereas K-NN tried to identify similarities between applications to predict the outcomes. To use PRISM, the application needs to be executed only once to collect an instruction profile of the application, instead of running an exhaustive fault injection campaign. PRISM can help us predict the resiliency profile for an application based on its instruction mix, which can aid architects to selectively protect the hardware structures, and potentially avoid the overhead introduced by RMT and ECC.

In the future, we plan to explore other models such as Tree-based Regression. We also plan to characterize neural network and machine learning workloads to further expand our GPU application suite.

REFERENCES

[1] *AMD Graphics Core Next (GCN) architecture*, https://www.amd.com/Documents/Compute_Cores_Whitepaper.pdf.

[2] *CUDA Binary Utilities*, http://docs.nvidia.com/cuda/pdf/CUDA_Binary_Utilities.pdf.

[3] "NVIDIA Tegra K1 techinical reference manual."

[4] *Multivariate Regression*. Wiley-Blackwell, 2012. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/9781118391686.ch10

[5] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1986.

[6] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick, "The landscape of parallel computing research: A view from berkeley," TECHNICAL REPORT, UC BERKELEY, Tech. Rep., 2006.

[7] R. E. Bellman, *Adaptive control processes: a guided tour*. Princeton university press, 2015, vol. 2045.

[8] A. Biswas, N. Soundararajan, S. S. Mukherjee, and S. Gurumurthi, "Quantized avf: A means of capturing vulnerability variations over small windows of time," 2009.

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

[10] Z. Chen and D. Kaeli, "Balancing scalar and vector execution on gpu architectures," in *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, May 2016, pp. 973–982.

[11] T. Cover and P. Hart, "Nearest neighbor pattern classification," *IEEE Trans. Inf. Theor.*, vol. 13, no. 1, pp. 21–27, Sep. 2006. [Online]. Available: http://dx.doi.org/10.1109/TIT.1967.1053964

[12] L. Duan, B. Li, and L. Peng, "Versatile prediction and fast estimation of architectural vulnerability factor from processor performance metrics," in *2009 IEEE 15th International Symposium on High Performance Computer Architecture*, Feb 2009, pp. 129–140.

[13] B. Fang, K. Pattabiraman, M. Ripeanu, and S. Gurumurthi, "A systematic methodology for evaluating the error resilience of gpgpu applications," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 12, pp. 3397–3411, Dec 2016.

[14] B. Farahani and S. Safari, "A cross-layer approach to online adaptive reliability prediction of transient faults," in *2015 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFTS)*, Oct 2015, pp. 215–220.

[15] X. Fu, J. Poe, T. Li, and J. A. B. Fortes, "Characterizing microarchitecture soft error vulnerability phase behavior," in *14th IEEE International Symposium on Modeling, Analysis, and Simulation*, Sept 2006, pp. 147–155.

[16] N. Goswami, R. Shankar, M. Joshi, and T. Li, "Exploring gpgpu workloads: Characterization methodology, analysis and microarchitecture evaluation implications," in *Workload Characterization (IISWC), 2010 IEEE International Symposium on*, Dec 2010, pp. 1–10.

[17] Q. Guan, N. Debardeleben, S. Blanchard, and S. Fu, "F-sefi: A fine-grained soft error fault injection tool for profiling application vulnerability," in *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, May 2014, pp. 1245–1254.

[18] A. E. Hoerl and R. W. Kennard, "Ridge regression: Biased estimation for nonorthogonal problems," *Technometrics*, vol. 12, no. 1, pp. 55–67, 1970.

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

[20] C. Kalra, D. Lowell, J. Kalamatianos, V. Sridharan, and D. Kaeli, "Performance evaluation of compiler-based software rmt in an hsa environment," in *The 12th Workshop on Silicon Errors in Logic - System Effects, SELSE*, April 2016.

[21] C. Kalra, F. Previlon, X. Li, N. Rubin, and D. Kaeli, "Analyzing the Vulnerability of Vector-Scalar Execution on Data-Parallel Architectures," in *The 14th Workshop on Silicon Errors in Logic - System Effects, SELSE*, April 2018.

[22] A. Kerr, G. Diamos, and S. Yalamanchili, "A characterization and analysis of ptx kernels," in *2009 IEEE International Symposium on Workload Characterization (IISWC)*, Oct 2009, pp. 3–12.

[23] W. Kirch, Ed., *Pearson's Correlation Coefficient*. Dordrecht: Springer Netherlands, 2008, pp. 1090–1091. [Online]. Available: https://doi.org/10.1007/978-1-4020-5614-7_2569

[24] R. Kohavi and G. H. John, "Wrappers for feature subset selection," *Artificial Intelligence*, vol. 97, no. 1, pp. 273 – 324, 1997, relevance. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S000437029700043X

[25] M. Kuhn and K. Johnson, *Applied Predictive Modeling*. New York, Heidelberg, Dordrecht, London: Springer, 2013. [Online]. Available: https://dl.dropboxusercontent.com/u/108263707/_book/KuhnJohnson2013apm.pdf

[26] M. Kulkarni, M. Burtscher, C. Casçaval, and K. Pingali, "Lonestar: A suite of parallel irregular programs," in *Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on*. IEEE, 2009, pp. 65–76.

[27] Y. Lee, R. Krashinsky, V. Grover, S. W. Keckler, and K. Asanovi, "Convergence and scalarization for data-parallel architectures," in *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, Feb 2013, pp. 1–11.

[28] R. Leveugle, A. Calvez, P. Maistri, and P. Vanhauwaert, "Statistical fault injection: Quantified error and confidence," in *Design, Automation Test in Europe Conference Exhibition, 2009. DATE '09.*, April 2009, pp. 502–506.

[29] G. Li, K. Pattabiraman, C. Y. Cher, and P. Bose, "Understanding error propagation in gpgpu applications," in *SC16: International Conference for High Performance Computing, Networking, Storage and Analysis*, Nov 2016, pp. 240–251.

[30] X. Li, S. V. Adve, P. Bose, and J. A. Rivers, "Online estimation of architectural vulnerability factor for soft errors," in *2008 International Symposium on Computer Architecture*, June 2008, pp. 341–352.

[31] R. Lucas, J. Ang, K. Bergman, and S. e. a. Borkar, "Top ten exascale research challenges," Feb 2014. [Online]. Available: http://science.energy.gov/~/media/ascr/ascac/pdf/meetings/20140210/Top10reportFEB14.pdf

[32] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building customized program analysis tools with dynamic instrumentation," in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '05. New York, NY, USA: ACM, 2005.

[33] J. Maiz, S. Hareland, K. Zhang, and P. Armstrong, "Characterization of multi-bit soft error events in advanced SRAMs," in *Electron Devices Meeting, 2003. IEDM'03 Technical Digest. IEEE International*. IEEE, 2003, pp. 21–4.

[34] S. S. Mukherjee, C. Weaver, J. Emer, S. K. Reinhardt, and T. Austin, "A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor," in *Proceedings. 36th Annual IEEE/ACM International Symposium on Microarchitecture, 2003. MICRO-36.*, Dec 2003, pp. 29–40.

[35] S. Mukherjee, J. Emer, and S. Reinhardt, "The soft error problem: an architectural perspective," in *High-Performance Computer Architecture, 2005. HPCA-11. 11th International Symposium on*, Feb 2005, pp. 243–247.

[36] NVIDIA, "NVCC, https://developer.nvidia.com/cuda-llvm-compiler."

[37] ——, "NVIDIA, CUDA SDK, V7.0."

[38] ——, "NVIDIA Kepler GK110 Architecture White Paper."

[39] P. Rech, L. L. Pilla, P. O. A. Navaux, and L. Carro, "Impact of GPUs Parallelism Management on Safety-Critical and HPC Applications Reliability," in *IEEE International Conference on Dependable Systems and Networks (DSN 2014)*, Atlanta, USA, 2014.

[40] S. K. Reinhardt and S. S. Mukherjee, "Transient fault detection via simultaneous multithreading," in *Proceedings of the 27th Annual International Symposium on Computer Architecture*, ser. ISCA '00. New York, NY, USA: ACM, 2000.

[41] S. K. Sastry Hari, T. Tsai, M. Stephenson, S. Keckler, and J. Emer, "Sassifi: An architecture-level fault injection tool for gpu application resilience evaluation," in *2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, April 2017, pp. 249–258.

[42] M. V. Shcherbakov, A. Brebels, N. L. Shcherbakova, A. P. Tyukov, T. A. Janovsky, and V. A. Kamaev, "A survey of forecast error measures," *World Applied Sciences Journal*, vol. 24, pp. 171–176, 2013.

[43] V. Sridharan and D. R. Kaeli, "Eliminating microarchitectural dependency from architectural vulnerability," in *IEEE 15th International Symposium on High Performance Computer Architecture, 2009. HPCA 2009.* IEEE, 2009, pp. 117–128.

[44] A. Staff, "OpenCL and the AMD APP SDK v2. 4," 2011.

[45] M. Stephenson, S. K. Sastry Hari, Y. Lee, E. Ebrahimi, D. R. Johnson, D. Nellans, M. O'Connor, and S. W. Keckler, "Flexible software profiling of gpu architectures," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, ser. ISCA '15. New York, NY, USA: ACM, 2015, pp. 185–197.

[46] J. A. Stratton, C. Rodrigues, I.-J. Sung, N. Obeid, L.-W. Chang, N. Anssari, G. D. Liu, and W.-m. W. Hwu, "Parboil: A revised benchmark suite for scientific and commercial throughput computing," *Center for Reliable and High-Performance Computing*, 2012.

[47] J. Tan, N. Goswami, T. Li, and X. Fu, "Analyzing soft-error vulnerability on gpgpu microarchitecture," in *2011 IEEE International Symposium on Workload Characterization (IISWC)*, Nov 2011, pp. 226–235.

[48] D. Tiwari, S. Gupta, J. Rogers, D. Maxwell, P. Rech, S. Vazhkudai, D. Oliveira, D. Londo, N. DeBardeleben, P. Navaux, L. Carro, and A. Bland, "Understanding gpu errors on large-scale hpc systems and the implications for system design and operation," in *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, Feb 2015, pp. 331–342.

[49] Y. Ukidave, F. N. Paravecino, L. Yu, C. Kalra, A. Momeni, Z. Chen, N. Materise, B. Daley, P. Mistry, and D. Kaeli, "Nupar: A benchmark suite for modern gpu architectures," in *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering*, ser. ICPE '15. New York, NY, USA: ACM, 2015, pp. 253–264. [Online]. Available: http://doi.acm.org/10.1145/2668930.2688046

[50] J. Wadden, A. Lyashevsky, S. Gurumurthi, V. Sridharan, and K. Skadron, "Real-world design and evaluation of compiler-managed GPU redundant multithreading," in *Proceeding of the 41st annual international symposium on Computer architecture*. IEEE Press, 2014, pp. 73–84.

[51] K. R. Walcott, G. Humphreys, and S. Gurumurthi, "Dynamic prediction of architectural vulnerability from microarchitectural state," in *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ser. ISCA '07. New York, NY, USA: ACM, 2007, pp. 516–527. [Online]. Available: http://doi.acm.org/10.1145/1250662.1250726

[52] N. J. Wang, A. Mahesri, and S. J. Patel, "Examining ace analysis reliability estimates using fault-injection," in *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ser. ISCA '07. New York, NY, USA: ACM, 2007, pp. 460–469. [Online]. Available: http://doi.acm.org/10.1145/1250662.1250719

[53] B. Wibowo, A. Agrawal, T. Stanton, and J. Tuck, "An accurate cross-layer approach for online architectural vulnerability estimation," *ACM Trans. Archit. Code Optim.*, vol. 13, no. 3, pp. 30:1–30:27, Sep. 2016. [Online]. Available: http://doi.acm.org/10.1145/2975588

[54] Y. Yang and J. O. Pedersen, "A comparative study on feature selection in text categorization," in *Proceedings of the Fourteenth International Conference on Machine Learning*, ser. ICML '97. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1997, pp. 412–420. [Online]. Available: http://dl.acm.org/citation.cfm?id=645526.657137

[55] J. H. Zar, *Biostatistical Analysis (5th Edition)*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 2007.

[56] Y. Zhang, S. Li, T. Wang, and Z. Zhang, "Divergence-based feature selection for separate classes," *Neurocomputing*, vol. 101, pp. 32 – 42, 2013. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0925231212006054

APPENDIX
ARTIFACT DESCRIPTION

In this Appendix, we describe how to reproduce the results reported in this paper.

Our Evaluation Platform includes:

- Graphics Processing Unit: Kepler K20
- Operating Sytem: Linux Ubuntu 14.04 LTS
- CUDA version: CUDA v7.0
- Python version: 2.7

We will provide step-by-step modifications required in every phase of the PRISM framework, shown in Figure 3. The description is provided as follows:

SASSIFI:

The first step is to download SASSIFI from resource (1) and then make the following changes to the tool:

1) We created a new Instruction Group called 'DEST_REG' that allows a user to inject faults in any destination register. The guidelines for adding a new IGID has been provided in the SASSIFI user guide.
2) Next, we implemented the scalarization algorithm described in Algorithm 1 in the profiler handler to identify scalar instructions.
3) We support 50 CUDA applications from the CUDA SDK, Parboil, Lonestar, Rodinia, and NUPAR benchmark suites on SASSIFI. A sample example (atomic add) has been provided in the github repository to demonstrate how to instrument a kernel with SASSIFI handlers (both - profiler and error injector). If an application has multiple kernels, each kernel must be instrumented with the handlers and statically linked using the linker, *nvlink*.
4) We then classify the outcomes into different categories. A DUE occurs when the program crashes or hangs even before it finishes execution. This behavior is automatically recorded by the tool and classified as a DUE. To check for SDCs or Masked outcomes, we developed a script (unique for every application) which executes after the program completes execution. We compare the *stdout*, *stderr*, and the *result* with the golden output. If there is a discrepancy in any of the three, the outcome is classified as an *SDC*, otherwise it is classified as *Masked*.
5) After implementing all of the above changes in the SASSIFI tool, we run 1000 injections on each sample (a total of 50,000 injections) to gather our raw data. We have used the IOV injection mode as it allows us to flip bits in a destination register, implying that a fault has occurred in the datapath of a random instruction. RF and IOA modes are specifically used for studying the vulnerability of the register file and memory accesses, which can be modeled using our methodology.

Rest of the framework:

The remainder of our framework has been written in Python. For feature extraction, we process the raw data generated by SASSIFI and derive the metrics mentioned in Table II. For feature selection, we implement our algorithm for the forward wrapper method with cross validation, as provided in the book [25]. Lastly, we feed the tuned parameters (such as K) and selected features into both models to generate our final results. These models can either be written by the user, or leverage existing python libraries. This framework can also be written in R programming language.

Choosing the Number of Injections:

As mentioned above, we have chosen 1000 injections per application for our study. According to Leveugle *et al.*, this number of injections should be sufficient to provide an error margin of 3.1%, with a 95% confidence level [28]. The confidence level is the probability that the value of the parameter-of-interest falls within the given range of values. A 95% confidence level is the most commonly used confidence level by researchers [55], and has been used in various prior work [13], [41]. The error margin is the range of values above and below the sample value in a confidence level.

Next, we observe the changes in all three outcomes by running 10K injections on our applications. Statistically, the difference between 1K and 10K injections is that 10K experiences a lower error margin for the same confidence level. Figure 11, 12, and 13 show the observed values for Masked, DUE, and SDC, respectively, for 1K and 10K injections. We report the Min, Max, and Mean Absolute difference (or error) as percentages comparing the observed values for 1K and 10K injections in Table IV. The maximum error is within our expected error margin of 3.1%.

In order to evaluate the robustness of our approach, we also report the results for 100K injections for the DWT2D application from Rodinia, randomly selected from our workload suite. As shown in Table V, we compare the results of 100K, 10K, and 1K injections for DWT2D. The observed difference in the distribution of outcomes for 1K and 100K injections varied by 2.26% for Masked, 2.71% for DUE, and 0.45% for SDC. Users can verify the number of injections required to achieve the desired confidence interval and error margin by using resource (2).

	Masked	**DUE**	**SDC**
Min Error	0.01 (lavaMD)	0.01 (lavaMD)	0 (Vector Add)
Max Error	3.00 (lbm)	3.09 (pathfinder)	3.06 (BlkSch)
Mean Absolute Error	1.51	1.15	1.39

TABLE IV: Min, Max, and Mean Absolute Error observed between 1K and 10K injections for Masked, DUE, and SDC.

	Masked	**DUE**	**SDC**
DWT2D (100K)	23.35%	23.34%	53.31%
DWT2D (10K)	22.96%	24.15%	52.89%
DWT2D (1K)	21.09%	26.05%	52.86%

TABLE V: Percentage Masked, DUE, and SDC for 100K, 10K, and 1K injections in DWT2D application.

Resources

1) *https://github.com/NVlabs/sassifi/*
2) *https://www.surveysystem.com/sscalc.htm*

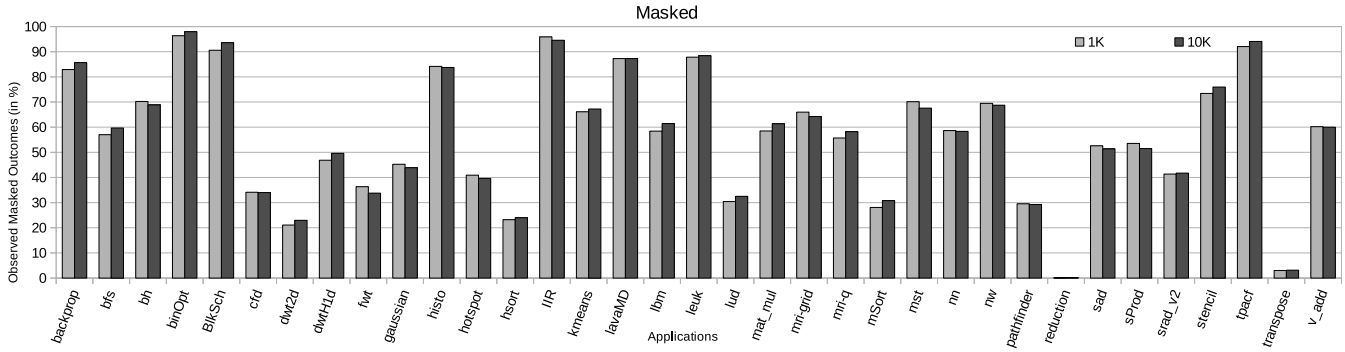


Fig. 11: Masked outcomes (in %) for 1K and 10K injections

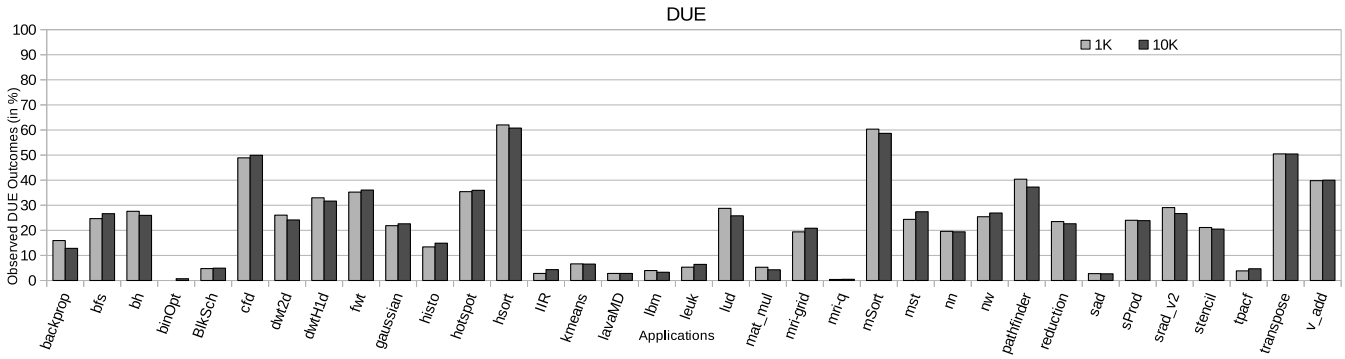


Fig. 12: DUE outcomes (in %) for 1K and 10K injections

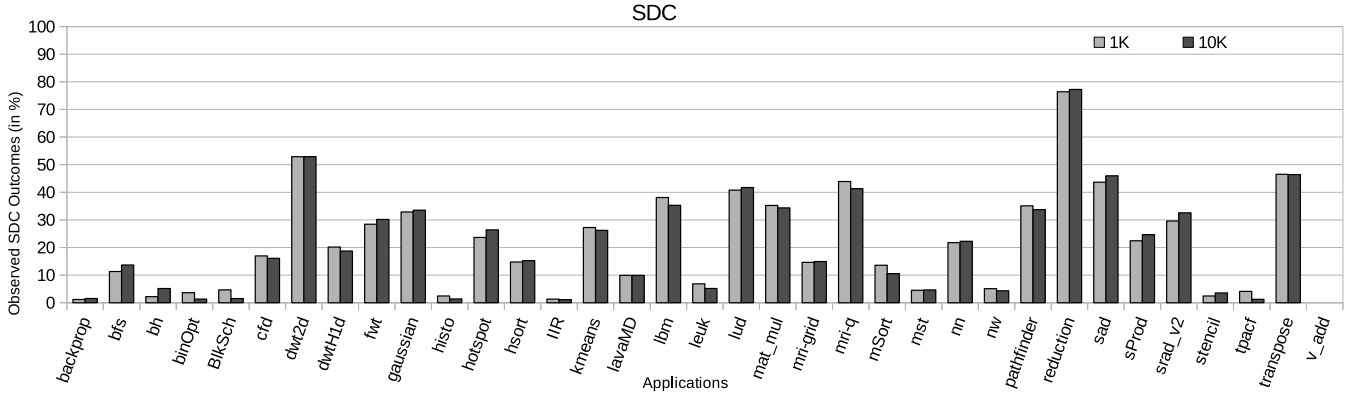


Fig. 13: SDC outcomes (in %) for 1K and 10K injections