Compiler-based Resilience Prediction and Enhancements for GPU Applications

A Dissertation Presented
by

Charu Kalra

to

The Department of Electrical and Computer Engineering

in partial fulfillment of the requirements
for the degree of

Doctor of Philosophy

in

Computer Engineering

Northeastern University
Boston, Massachusetts

August 2019

ProQuest LLC.
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106 – 1346

To my loving family.

Contents

List of Figures

List of Tables

List of Acronyms

AVF Architectural Vulnerability Factor

CFG Control Flow Graph

CPU Central Processing Unit

CTA Cooperative Thread Arrays

CUDA NVIDIA's Compute Unified Device Architecture Framework

CV Cross Validation

DUE Detected and Unrecoverable Error

ECC Error Correction Codes

FIT Failures-in-Time

GCN Graphics Core Next

GPRs General Purpose Registers

GPU Graphics Processing Unit

IR Intermediate Representation

ISA Instruction Set Architecture

KMeans K-Means Clustering Algorithm

LDS Local Data Store

ML Machine Learning

NRMSE Normalized Root Mean Squared Error

PVF Program Vulnerability Factor

PTX Parallel Thread Execution

RMT Redundant Multithreading

SDC Silent Data Corruption

SIMD Single-Instruction Multiple Data

SIMT Single Instruction Multiple Thread

SMX Streaming Multiprocessor

SoR Sphere of Replication

TLD Thread-level Duplication

TLP Thread-level Parallelism

Acknowledgments

There are so many people who deserve credit for where I stand today. First and foremost, I want to thank my advisor, Prof. David Kaeli, for guiding me and giving me the freedom to explore new ideas. I will always appreciate his patience and kindness.

I am very grateful to have Norm Rubin as my mentor and co-advisor. His wisdom and insights have made a huge positive impact on my research and my perspective towards problem-solving. I want to thank my committee member, Prof. Mi for her feedback that significantly improved the quality of this work.

A big shout-out to my wonderful lab-mates: Saoni, Fritz, Leiming, Xiangyu, Mohammad, Julian, Shi, Nico, Elmira, Amir, Xun, Trinayan, Yifan. I have learned a lot through my collaboration with them. Lab-time would have been extremely boring without these folks.

I thank my dear friends, Sri, Suchi, Nasibeh, Suma, and Shweta, for their positivity and encouragement. I want to express my deepest gratitude to my family for their continuous love and support – especially my adorable niece (Eva) and nephew (Avi) who were a constant source of entertainment during my PhD. Last but not least, I want to thank Sahil Shah for always being there and inspiring me to do better than I ever thought I could.

Abstract of the Dissertation

Compiler-based Resilience Prediction and Enhancements for GPU Applications

by

Charu Kalra

Doctor of Philosophy in Computer Engineering

Northeastern University, August 2019

Dr. David Kaeli, Advisor

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. The scaling of transistor sizes, operating voltages, and design margins has exacerbated the susceptibility of GPU devices to soft errors. Due to the random nature of these faults, predicting whether they will alter program output is a challenging problem. Traditional methods of reliability estimation involve thousands of random fault injections per application, such that one random bit used by the program is flipped during its execution and the outcome is recorded. Despite prior efforts to mitigate errors, we still lack a clear understanding of how resilient these applications are in the presence of transient faults. As a consequence, the same level of fault protection is typically employed for all applications, regardless of their resiliency. This adversely impacts the performance of the GPU applications, many times unnecessarily. Moreover, existing fault protection schemes cannot be tailored based on the reliability requirements of the applications.

In this thesis, we tackle the above limitations of the prior work by building frameworks that can predict and mitigate faults in GPU applications. We develop a toolset that can identify micro-architecture agnostic characteristics in the program, based on the hardware resources they stress during execution. Our study first aims to understand error propagation characteristics when faults are injected in scalar, versus vector, instructions. We then extend this study to build a more sophisticated learning-based framework, called PRISM, which enables us to predict failures in applications without running exhaustive fault injection campaigns, thereby reducing the error estimation effort. We leverage the insights provided by PRISM to develop a framework called ArmorAll, a light-weight, intelligent, adaptive, and portable software solution to protect GPUs against soft errors. ArmorAll consists of a set of pure compiler-based redundancy schemes designed to optimize

instruction duplication on GPUs, thereby enabling much more reliable execution. The choice of the scheme determines the subset of instructions that must be duplicated in an application, allowing adaptable fault coverage for different applications.

Chapter 1

Introduction

The reliability of a system is one of the primary factors that determine its ability to operate correctly and deliver the right data to the user. Any occurrence of a fault may have a detrimental impact on the system if proper fault tolerance techniques are not employed. A fault tolerant system can continue to operate in the event of a failure. The system can achieve this by either preventing the fault, or detecting and correcting the state of data impacted by the fault. However, the criticality of the reliability requirements depends on the application and the user's needs. For example, an error in a single pixel may not be perceivable to the user even if it changes the color from white to black. On the other hand, a single error in a low-order bit in a commercial or scientific application can invalidate a computation.

Today, due to the ever increasing demands for high performance and low power/area overheads, we have witnessed exponential growth in the number of transistors per chip in the current microprocessor technology. Packing more transistors on a chip leads to decreased feature sizes makes circuits more susceptible to permanent (hard) and transient (soft) faults [1, 2, 3]. Permanent faults are caused by physical phenomena affecting the hardware components, such as manufacturing defects, thermal stress, or circuit aging, and may develop over time as a combination of these phenomena. Permanent faults are reproducible and irreversible, and occur in the same micro-architectural location. On the other hand, transient faults can be caused by temperature and voltage variations, electromagnetic interference, crosstalk, and high energy particles in the atmosphere. An alpha or neutron particle strike can invert the state of a logic gate or memory cell, and may drive a wrong value temporarily [4]. Since these faults are temporary, they do not reoccur when the operation is re-executed in the future. Transient faults induced by radiation have received much of the attention, as reliability is claimed to be one of the major challenges for exascale computing [5].

Major supercomputers in the TOP500 list (the fastest 500 supercomputers in the world, published each year) today are taking advantage of the massively parallel floating point capabilities of General Purpose Graphics Processing Units (GPGPUs)[6]. The scope of GPU applications has steadily expanded and they have been introduced in a number of domains that come with significant reliability constraints. These domains include scientific computing [7], bioinformatics [8], molecular modeling [9], and finance [10], where workloads typically demand high precision and correctness; and autonomous vehicles that perform a range of safety-critical tasks such as pedestrian detection and avoidance, and vehicle control [11, 12]. While leveraging the inherent parallelism offered by GPUs is crucial to accelerate these applications, it is equally important to ensure that applications can overcome data corruption caused by transient or permanent faults. Errors in applications from these domains are often very costly, both in financial terms and liability, and potentially in terms of safety. Since GPUs were originally designed for graphics rendering, an operation that is inherently quite resilient to bit flips, little emphasis was placed on the reliability of these devices. Therefore, in order for GPUs to be accepted as the primary choice of processor by the high-performance and automotive communities, vendors will have to address the reliability concerns of graphics processors. Moreover, 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 [13].

In order to detect unwanted changes in the state caused by transient faults, most fault protection techniques ultimately rely on some sort of redundancy (either temporal or spatial). Error tolerance through hardware is usually achieved by using N-modular redundancy across computer nodes. The most popular form of this redundancy is Triple Modular Redundancy (TMR), where the same computation is repeated by three computer nodes and a voting mechanism is used to choose the correct data [14]. The main problem with TMR is that it incurs prohibitive energy and area overheads, which makes it unsuitable for deployment in commodity systems. Other hardware fault-tolerant mechanisms such as parity, Error Correction Code (ECC), residue execution [15], are available, but their benefits rarely come without significant area, power, and cost overheads. Moreover, hardware protection is inflexible and is not necessary for select workloads (e.g., graphics) which are inherently fault-tolerant. Hardware structures which are replicated are said to be within a Sphere of Replication (SoR), and are assumed to be protected. In other words, SoR is a logical boundary that identifies which hardware structures are protected by a fault detection scheme [16]. All values that enter the SoR must be replicated (called input replication) and all redundant values that leave the SoR must be compared (called output comparison) before a single correct copy is allowed to leave.

On the other hand, software-directed approaches such as compiler-based Redundant Multi-Threading (RMT) can be more flexible, but still come with significant slow-down due to high synchronization overhead among threads [17, 18]. RMT accomplishes replication by running two identical redundant threads - a producer and consumer - on replicated input. Whenever state needs to exit the SoR (e.g., on a store to unreplicated global memory), the producer sends its value to the consumer for checking before the consumer is allowed to execute the instruction. Two faults could create simultaneous identical errors in redundant state, but this is considered sufficiently unlikely. Hardware structures outside the SoR must be protected via other means. Figure 1.1 illustrates the performance overhead of three variants of compiler-based RMT on a GPU [17]. The three variants differ in terms of the level of fault coverage (i.e., the Sphere of Replication) 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 clearly, Inter-Group RMT experiences a significant increase in execution time, as compared to execution using Intra-Group RMT. Software-based redundancy can also be applied at a finer grain than a thread, providing selective redundancy on individual instructions. Software instruction-level replication has been studied extensively on CPUs, but it is still an under-explored research area on GPUs. Prior work has leveraged a combination of hardware and software techniques to improve the performance of full instruction duplication on GPUs [19]. However, reliability researchers have not explored the rich data analyses provided by the compiler.

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. Given the random nature of transient faults, predicting whether they will alter the program's output is also a challenging question. 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

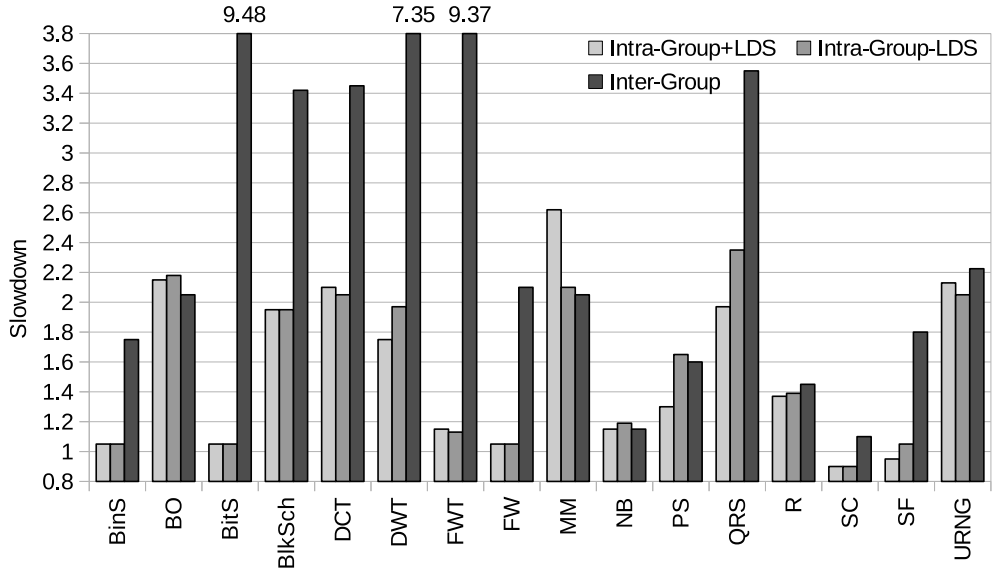


Figure 1.1: Slowdown caused by Intra-Group+LDS, Intra-Group-LDS, and Inter-Group RMT, over baseline, without RMT [17]. Slowdown is calculated by normalizing the runtime of the RMT variant of the kernel by its original runtime.

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. Therefore, understanding the error propagation behavior of an application is an important step towards designing cost-effective fault mitigation schemes.

Traditional methods of reliability estimation involve thousands of random fault injections per application. For each fault injection experiment, one random bit used by the program is flipped during its execution and its outcome (SDC, DUE, or Masked) is recorded. To perform a statistically significant number of injections, an application must be executed thousands of times, such that one random bit is flipped per run of the program. It is not practical to execute an application so many times in order to estimate its vulnerability and then launch it on the GPU. As a result, the same level of fault protection is typically employed for all applications, regardless of their resiliency. The problem with this approach is that not all applications require the same level of fault coverage. This adversely impacts the performance of the GPU applications, many times unnecessarily. There is no existing framework which can predict the vulnerability of GPU applications without performing these

time-consuming fault injections. Moreover, existing fault protection schemes cannot be customized based on the reliability requirements of the applications.

Motivated by the above limitations of the prior work, we develop a research plan to address soft error assessment and remediation in this thesis. We develop a toolset that can identify micro-architecture agnostic characteristics in the program based on the hardware resources they stress during execution. Having micro-architecture agnostic features allows our techniques to be applied to multiple target processors and across different generations. Our study first aims to understand error propagation characteristics when faults are injected in scalar, versus vector, instructions. As mentioned earlier, understanding the fault propagation behavior of different applications is important as it can provide insights into designing cost-effective fault mitigation schemes. We then extend the study to build a more sophisticated learning-based framework named PRISM, which uses a systematic approach to predict the resilience of GPU programs without running exhaustive fault injection campaigns. Once PRISM gives insights into the vulnerability of an application, we must leverage that information to provide high fault coverage to the application and at the same time, minimize the performance overhead incurred by the fault protection scheme. This is possible if the scheme is tailored as per the requirements of the application. The overheads incurred by the hardware and its lack of flexibility prompted us to explore low-cost, portable software solutions for improving reliability on processors. A software-level solution is generally more flexible and can be tuned depending on the reliability characteristics of the application. Therefore, we develop ArmorAll, which is a set of pure compiler-based schemes that exploit redundancy at an instruction level. The redundancy schemes are designed to leverage the vulnerability characteristics of the applications and optimize instruction duplication, thereby providing high fault coverage with low performance overhead. Our work focuses on reliable execution on the processor pipelines, while memory, caches, and register file are assumed to be protected by ECC/parity. This assumption is in line with other work in this domain [19, 20, 21]. We summarize the benefits of both the frameworks in detail, as follows:

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 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.

Our vulnerability characterization of applications has helped us properly design the selective fault protection properties of ArmorAll, which includes multiple compiler-based redundancy schemes that optimize instruction duplication on GPUs. The novel features and benefits of ArmorAll are:

- **Lightweight:** Duplicating all instructions can increase the register usage per thread, adversely impacting the performance and occupancy (number of active threads) of a GPU application [22]. Instruction duplication techniques developed for CPUs do not have to deal with the same issues that GPUs do in terms of the register usage versus occupancy tradeoff. Moreover, CPU redundancy schemes have only been evaluated on a single thread [23, 24, 25]. Therefore, applying CPU-based techniques directly to GPUs without considering these constraints can result in a significant loss in performance. One of the benefits of ArmorAll is that it judiciously applies instruction duplication to protect only those segments of code that are likely to result in user-visible faults when experiencing a soft error. Equipped with ArmorAll, we can produce a GPU binary that leverages our low-cost, high-coverage, solution for addressing soft errors on GPUs.

- **Adaptive:** Prior research on instruction-based duplication replicated a fixed class of instructions in all applications, regardless of their resiliency characteristics [19, 23]. While our work is motivated by their work, exhaustively applying duplication everywhere in the code can adversely impact the performance of GPU applications. Our work differs significantly from this prior approach by recognizing that not all applications are equally vulnerable. Based

on our observations, we propose three redundancy schemes as part of ArmorAll. The extent of duplication can be adapted by the compiler, depending on the choice of the redundancy scheme, which determines the subset of instructions that must be duplicated in an application.

- **Selective:** To provide both high fault coverage and low overhead, ArmorAll first intelligently identifies *critical instructions* in an application that produce program visible outputs. ArmorAll then performs dependence analysis to identify candidate instructions that directly or indirectly influence these critical instructions. A transient fault in any of these instructions is likely to propagate and cause a program-visible corruption. ArmorAll judiciously applies instruction duplication to these instructions to optimize coverage while minimizing performance overhead. On average, the redundancy schemes duplicate between 20.4% to 52.2% of the static instructions, considerably lower than exhaustive duplication.

- **Portable:** The redundancy schemes provided with ArmorAll are implemented in an open-source LLVM compilation framework, which provides independence from the underlying hardware and supports portability [26]. Though our evaluation is done on a specific GPU architecture, ArmorAll can be easily applied to other existing or future GPU architectures.

- **Static:** ArmorAll does not rely on any dynamic profiling counters for assistance in choosing the best redundancy scheme. All of the decision-making by ArmorAll is done by heuristics at compile-time, which eliminates the need of executing an application multiple times on the GPU to gather statistics.

1.1 Contributions of this thesis

The main contributions of this thesis include:

- We identify and extract micro-architecture agnostic vulnerability characteristics from GPU applications that drive the prediction and mitigation of faults.

- We analyze the error propagation characteristics of scalar and vector operations on GPUs.

- We extend the above analysis and explore machine learning models that predict the resiliency of the applications based on the program characteristics.

- We propose three novel reliability-aware compiler-based redundancy schemes that can provide the desired level of fault coverage to an individual GPU application.

- We develop a technique which intelligently selects a redundancy scheme that provides the best coverage to an application, at compile-time.

1.2 Organization of this thesis

In Chapter 2, we describe the possible error outcomes in the event of a transient fault and fundamentals of NVIDIA's Kepler GPU architecture. We will review statistical principles and terminology that we will use throughout this thesis. We will also cover the LLVM IR and optimization framework, and the role of an Intermediate Representation (IR) in the compilation process.

In Chapter 3, we review the body of work dedicated to comprehensive studies of vulnerability and the analytical measurement of vulnerability. Our 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 that have tried to characterize GPU applications. In the second category, we will present the studies on prediction of system vulnerability. Finally, in the third category, we review related work on hardware and software redundancy-based fault mitigation.

In Chapter 4, we describe micro-architecture agnostic features to characterize program resiliency. These features are subsequently used in PRISM and ArmorAll to predict and enhance the resilience of GPU applications. We also study the error propagation characteristics when faults are injected in scalar, versus vector, instructions.

In Chapter 5, we introduce the PRISM framework. PRISM uses SASSIFI, a binary instrumentation tool, to generate dynamic execution and error profiles of the applications [27]. The features generated in Chapter 4 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.

In Chapter 6, we will present three redundancy schemes, as part of ArmorAll, designed to optimize instruction duplication on GPUs.

In Chapter 7, we will summarize the contributions of this thesis, and outline the potential directions for future work.

Chapter 2

Background

In this chapter, we will first provide an overview on compiler basics and LLVM intermediate code representation. Since the target processor of our compiler belongs to the NVIDIA's Kepler family, we will describe its architecture and the compilation flow. We will then describe our profiling and fault injection tool, SASSIFI, and the possible error outcomes when a fault is injected in a program. Lastly, we will review some statistical terminology which we will use in reference to our prediction models in the subsequent chapters.

2.1 Overview of a Compiler

A compiler is a software program that translates source code written in a programming language (the source language) into another computer language (usually the machine code). A program that translates between high-level languages is usually called a source-to-source compiler. A compiler may be comprised of two or more phases, referred to as the frontend, the middle-end, and the backend. In some simpler compilers, the middle-end may be absent. A two phase compiler is comprised of only a frontend and a backend. The frontend of the compiler does lexical analysis, parsing, semantic analysis, and translation to the intermediate representation or IR. The middle-end is usually designed to perform optimizations on a form other than the source code or machine code. This source code/machine code independence is intended to enable generic optimizations to be shared between versions of the compiler supporting different languages and target processors. The backend performs optimizations on the IR and code generation for a particular processor. The structure of a three-phase compiler is shown in Figure 2.1.

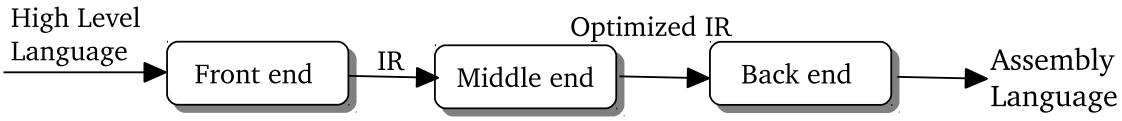


Figure 2.1: Structure of a three-phase compiler.

2.2 Intermediate Representation

Intermediate Representations (IRs) are low level, machine independent languages that compilers use to analyze and optimize code in a retargetable way. IRs allow compilers to maintain a generic set of optimizations for multiple ISAs, rather than duplicating these optimizations for each supported ISA. As an example, the intermediate representation of the LLVM compiler infrastructure can support many languages (C, Objective C, C++, OpenCL, Python, Ruby, Haskell, etc.) and ISAs (x86, ARM, MIPS, PowerPC, SPARC, etc.) with a mature set of analyses and optimizations developed in terms of the IR [26].

2.2.1 Types of Intermediate Representations

IRs broadly fall into three structural categories [28]:

- *Graphical IRs* encode the compiler's knowledge in the form of a graph. All algorithms are expressed in terms of nodes, edges, trees, or lists. An example of graphical IR is an Abstract Syntax Tree (AST).
- *Linear IRs* resemble assembly code for some abstract machine. The algorithms iterate over simple, linear, sequences of instructions. LLVM IR is an example of a linear IR.
- *Hybrid IRs* are a combination of both linear and graphical IRs. A common hybrid representation uses a linear IR to represent blocks of straight code and a graph to represent the control flow among those blocks. A Control Flow Graph (CFG) is an example of a hybrid IR.

2.2.2 Role of the Intermediate Representation

Intermediate Representations are used by the compilers for two purposes:

- **Portability:** Without an IR, compiling N different source languages to M different machines would require N x M compilers, as shown in Figure 2.2a. A portable compiler translates the

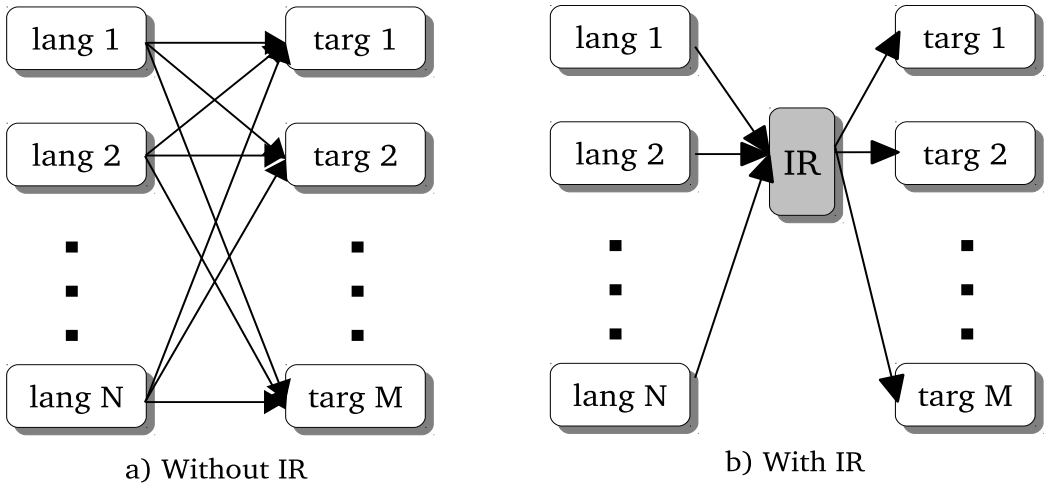


Figure 2.2: Compilation support from N source languages to M targets.

source language into an IR, that then translates the IR into machine language, as shown in Figure 2.2b. This way only N + M compilers have to be built [29].

- **Modularity:** When developing a compiler, an IR can keep the frontend of the compiler independent of machine specific details. At the same time, an IR can help the backend compiler to be independent of the peculiarities of the source language, therefore making the development of the compiler more modularized.

2.2.3 LLVM Intermediate Representation

LLVM IR is a RISC-like instruction set, but without any notion of the source language constructs, such as classes, inheritence and other semantics [26]. It has no knowledge about machine specific details such as processor pipeline, the number of registers, etc. LLVM is a load/store architecture: programs transfer values between registers and memory solely via load and store operations using typed pointers. The entire LLVM instruction set consists of only 31 opcodes. Most opcodes in LLVM are overloaded (for example, the add instruction can operate on integer or floating point operand types). Most instructions, including all arithmetic and logical operations, are in one of three address forms: they take one or two operands and produce a single result. LLVM uses SSA form as its primary code representation, i.e., each virtual register is written in only a single instruction, and each use of a register is dominated by its definition [30]. Memory locations in LLVM are not in SSA form. SSA form requires that each variable is assigned exactly once, and

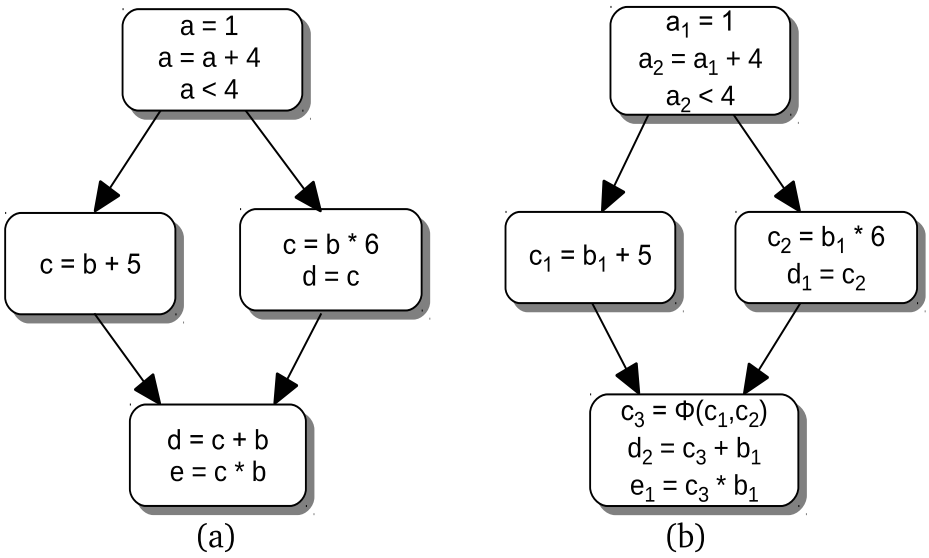


Figure 2.3: (a) A CFG in non-SSA form (b) CFG converted to SSA form by inserting phi instructions.

every variable is defined before it is used, as shown in Figure 2.3. SSA improves the results of a variety of compiler optimizations by simplifying the properties of variables. SSA form is achieved by inserting phi nodes, which are represented by using phi instructions. The syntax of phi instruction is as follows [31]:

```
<result> = phi <ty> [ <val0>, <label0>], ...
```

The type field determines the type of the incoming values. The phi instruction takes a list of pairs as arguments, with one pair for each predecessor basic block of the current block. Only values of the first class type may be used as the value arguments to the phi node, and labels can only be used as the label arguments. The phi instructions must always be first in a basic block.

LLVM has received attention for many years. A variety of compiler frontends and backends have been developed to support a diverse range of high level languages and modern architectures using the LLVM code representation and transformations. The advanced and robust features offered by LLVM led us to utilize it's intermediate code representation in our compiler infrastructure.

2.3 Graphics Processing Units

GPUs have become the most widely used accelerators available in the market today. The use of GPUs is no longer limited to graphics, but has been extended to support a broad

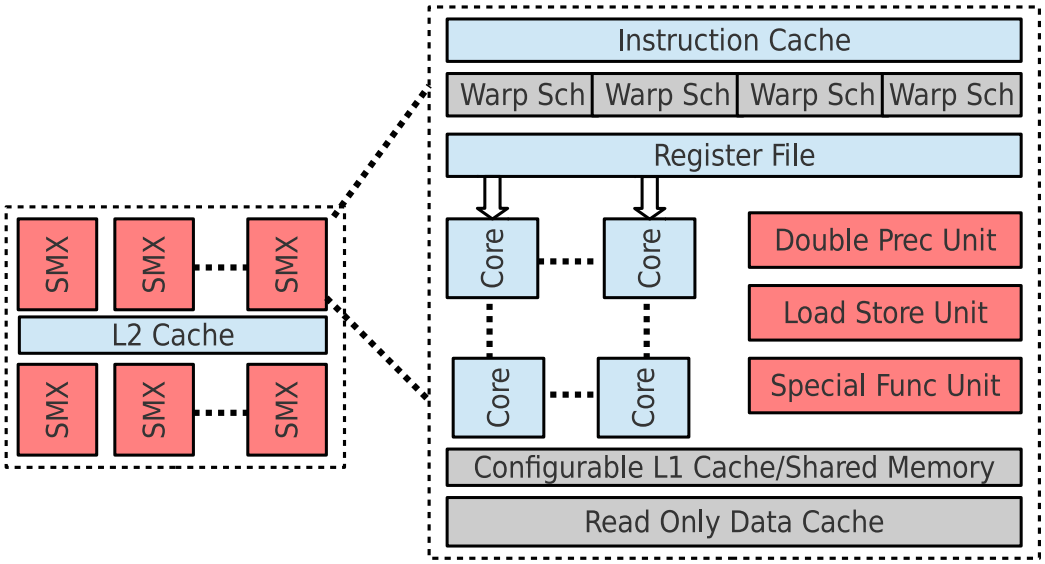


Figure 2.4: Overview of NVIDIA Kepler GK110 architecture.

range of compute-based applications. Effective exploitation of the massively parallel resources of a GPU have resulted in large speedups for applications in a number of computational domains including scientific computing, biomedical imaging, global positioning system, and signal processing applications [32, 33, 34]. The popularity of GPUs has led to extensive research in developing workloads for GPUs and optimizing them to enhance their performance. CUDA and OpenCL are the most commonly used parallel programming frameworks to create applications that can execute on GPUs [35, 36]. CUDA is currently supported on NVIDIA GPUs and multicore CPUs [37], whereas OpenCL is supported on different heterogeneous devices including GPUs, multi-core CPUs and FPGAs. Many techniques have been developed to exploit the potential parallelism in applications by understanding their interaction with the GPU architecture [38, 39, 40, 41].

Architecturally, a GPU is composed of thousands of cores that can handle many thousands of concurrent threads. We describe the NVIDIA Kepler architecture, given that we have chosen it to serve as our evaluation platform [42]. Streaming Multiprocessors (SMX) are the fundamental units of computation on NVIDIA GPU architectures, as shown in Figure 2.4. 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

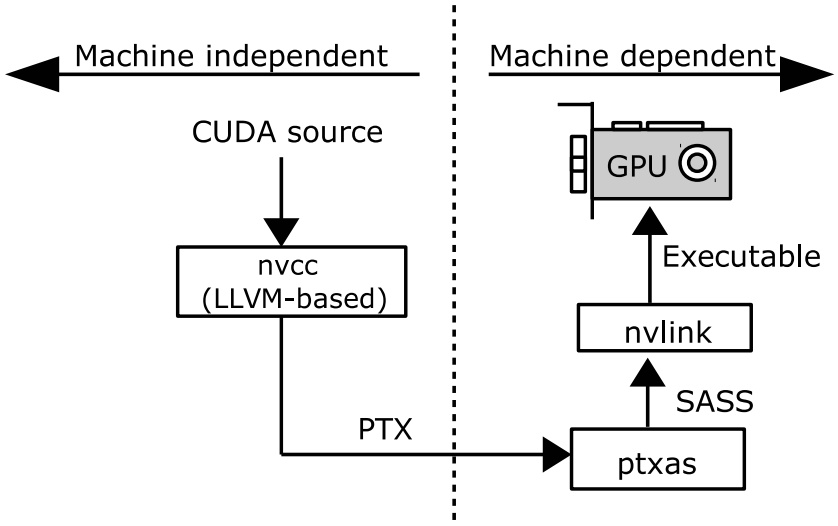


Figure 2.5: The flow of the NVIDIA *nvcc* CUDA compiler.

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* [43]. The front-end converts CUDA code into an intermediate representation, LLVM-IR, and subsequently to a virtual instruction set architecture (ISA) called PTX (Parallel Thread Execution), as shown in Figure 2.5. The backend compiler, *ptxas*, translates PTX to machine code, called SASS, that runs on the GPU. We run our profiling and fault injection campaign at the LLVM IR and SASS level. We will now describe NVIDIA's dynamic instrumentation and fault injection framework, SASSIFI, in further detail.

2.4 Fault Injections for Reliability Estimation

Fault injection is one of the most commonly used methods to evaluate the dependability of a computer system. Researchers and engineers have created many novel methods to inject faults at both the hardware level and the software level. The difference between hardware and software methods mainly lie in the fault injection points they can access, the cost and the level of perturbation. Hardware methods can inject faults into chip pins and internal components, such as combinational circuits and registers that are not software-addressable. On the other hand, software methods are convenient for directly producing changes at the architecturally visible state. Thus, hardware methods are used to evaluate low-level error detection and masking mechanisms, and software methods to test higher level mechanisms [44]. Software fault injection methods are attractive as they do not require expensive hardware.

Software fault injection can be performed either at compile-time or during run-time. To inject a fault at compile-time, the source or the assembly of a program is modified to emulate the effect of a transient fault. The modification in the code alters the target instructions causing the fault to be activated when the program executes on the device. Compile-time fault injection is very simple, but does not enable a fault to be injected at runtime because the fault is hard coded. The second option is to perform fault injection at runtime. In this technique, instructions are inserted into the code that trigger fault injection before a particular instruction. Additional instructions are inserted in the code for run-time fault injection, rather than modifying the original instructions. In our work, we use a run-time fault injection tool called SASSIFI, to inject faults at the GPU assembly level. Injecting faults at the assembly level allows us to take advantage of the ISA-specific optimizations performed prior to producing the binary. Since the final binary executes on the hardware, working at this level provides higher fidelity in terms of reliability metrics. We will now describe the SASSIFI tool in more detail.

2.4.1 SASSIFI

The SASSIFI tool is based on SASSI, which is a dynamic instrumentation tool for GPUs, similar to the Pin tool for CPUs [27, 45]. SASSI instruments the SASS code at runtime by linking user-written instrumentation handlers with the application binary [46]. SASSI instrumentation handlers are written in CUDA C/C++ and can be inserted before, after, or both before and after an instruction executes.

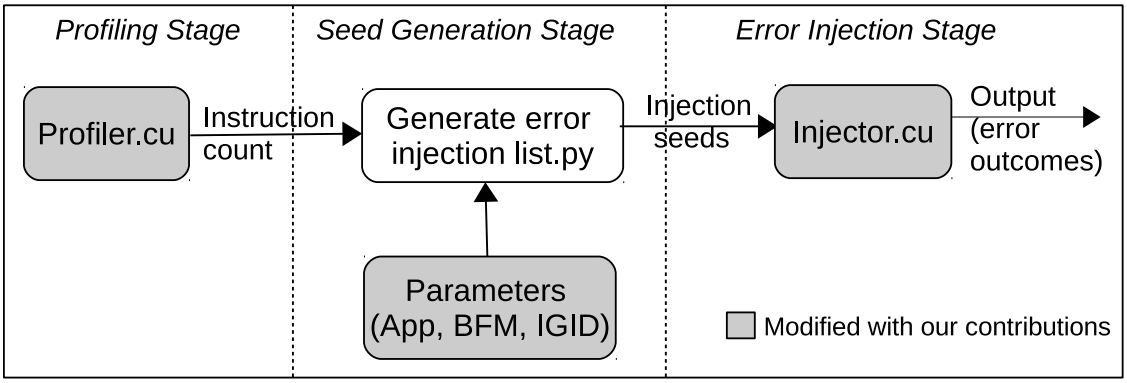


Figure 2.6: Different stages in the SASSIFI tool.

SASSIFI mainly comprises three stages, as shown in Figure 2.6. 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 [47]. 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. We perform our fault injection campaign on a real NVIDIA Kepler K20 GPU. which is significantly faster than using a simulator. 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 Chapter 1.

2.5 Statistical Terminology

We next review the statistical terminology that we will use throughout this thesis [48].

- The term *sample* refers to a single, independent unit of data. In our study, each CUDA application is a *sample*.

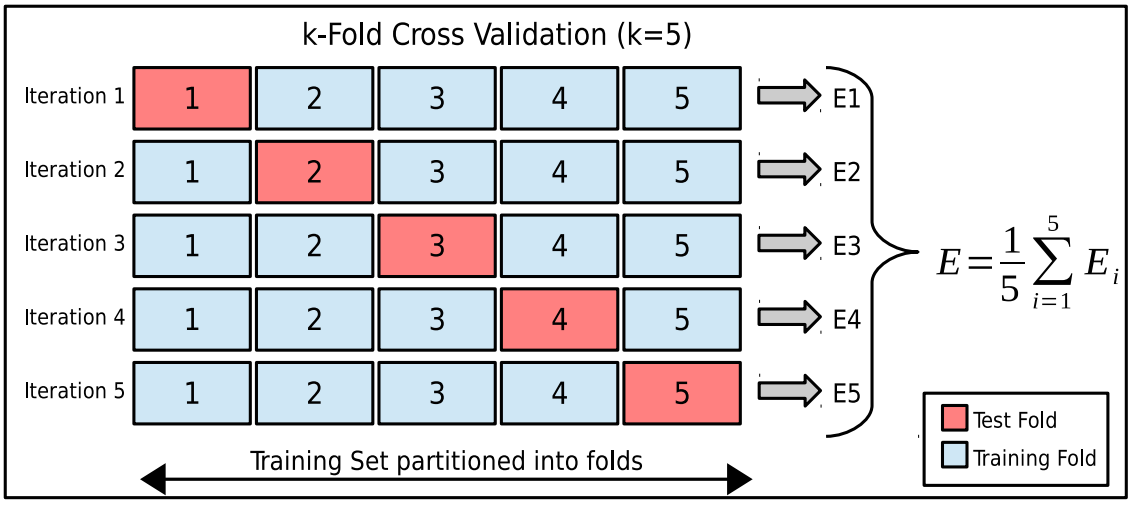


Figure 2.7: Demonstration of 5-fold Cross Validation (CV) technique. CV is used to prevent the problem of overfitting.

- 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.

- *Outcome* refers to the outcome event that is being predicted. Our focus in this thesis 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 2.7. 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.

- A *Confidence Interval* is the probability that the value of the parameter-of-interest falls within the given range of values.
- *Error Margin* is the range of values above and below the sample value in a confidence level.

Chapter 3

Related Work

In this chapter, we perform a comprehensive review of the prior work related to our research area. In the first section, we will cover research in the field of GPU workload characterization and its application towards performance and power optimizations. In the following sections, we will discuss work carried out in the area of GPU vulnerability and vulnerability prediction. In the last section, we will present research conducted in the area of hardware and software redundancy-based fault mitigation.

3.1 GPU application characterization

As GPUs have gained popularity in high performance computing domains, many studies have tried to characterize the performance of GPU workloads and identify the bottlenecks. Characterization is typically done by either profiling or instrumenting an application which helps understand its architectural and microarchitectural behavior. Alternatively, benchmarking is also used to understand the interaction between the hardware and the application. The data collected through characterization is then analyzed in order to optimize the performance, power, or reliability of an application or to improve the hardware.

Kerr *et al.* introduce a set of metrics for GPU workloads and utilize these metrics to analyze the behavior of GPU workloads [49]. They characterize applications at the PTX level based on control flow, data flow, parallelism, and memory behavior. They use this information to guide compiler optimizations, application re-structuring, and micro-architecture design. Goswami *et al.* propose a set of microarchitecture-agnostic workload characteristics to capture the behavior of GPU applications [50]. The authors use statistical methods, such as dimensionality reduction and clustering,

to characterize workload behavior. Burtscher *et al.* quantitatively study the behavior of irregular GPU programs in which the control flow and memory access patterns are data-dependent and unpredictable at compile-time [51]. The authors use performance counters to measure the control-flow and memory-access irregularity and identify that irregular codes more frequently underutilize GPU resources due to branch divergence, load imbalance, and synchronization overheads, as compared to regular applications. In their subsequent work, the authors accelerate irregular CUDA applications and assess the impact of their optimizations on caches, DRAM bandwidth, and execution latency. Che *et al.* present a detailed characterization of the Rodinia benchmark suite on a Fermi architecture GPU and contrast its performance with the Parsec suite [52]. Seo *et al.* characterize and compare the performance of the NAS Parallel Benchmark suite written in OpenCL and OpenMP [53]. The above papers study characterize GPU applications in terms of their performance, whereas our work focuses on characterization based on the reliability of GPU applications.

Sherwood *et al.* characterize the time-varying behavior of the program using a hardware independent metric called *Basic Block Vector (BBV)* [54]. The authors use BBVs to identify common patterns and codes that repeat during program execution. They cluster the BBVs and identify the representative models for each cluster. These representatives form simulation points (called SimPoints), which, when combined, accurately capture the behavior of the entire program. Unlike our work, the goal of SimPoints is to reduce the overall simulation time by executing single or multiple SimPoints. Previlon *et al.* have exploited the time varying behavior of the program to reduce the fault injection time [55].

Prior research has also characterized power and energy efficiency of GPU applications. Jiao *et al.* identified the correlation between power consumption and voltage/frequency levels for applications with different memory and compute behavior [56]. Abe *et al.* characterize the power and performance of GPU and provide a statistical model that can predict them across multiple generations of GPU [57]. Hong *et al.* propose a system that takes the GPU kernel as an input and predicts its power and performance together without using hardware performance counters [58]. Wu *et al.* use machine learning techniques to predict the power and performance of GPU applications [59]. With their trained model, they are able to estimate the power and performance of new kernels within 15% and 10% of real hardware, respectively. However, none of the prior work has attempted to predict the reliability of GPUs using statistical methods, which is a major our focus area in this thesis.

3.2 GPU Vulnerability Analysis

Vulnerability analysis plays an important role in determining which faults matter and which do not. This allows us to make informed decisions about which structures to protect. 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 [60, 61]. Studies have measured GPU application resilience using Program Vulnerability Factor (PVF) or related metrics using both fault injection [62, 63, 27] and ACE analysis [64, 65]. PVF is a vulnerability metric that only considers program-level effects on vulnerability [66]. PVF can be measured with either statistical fault injection or through Architecturally Correct Execution (ACE) analysis [67]. 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 [68]. Farazmand *et al.* quantify the Architectural Vulnerability Factor (AVF) of the register file, local memory, and active mask on the GPU via statistical fault injection [69]. AVF is the measure of the percentage of injected faults that result in program failures [67].

In addition, Li *et al.* have investigated the propagation of errors across GPU kernel calls using fault injection experiments [70]. They have also developed a comprehensive fault injection tool, LLFI-GPU, which allows users to perform fault injection experiments at the intermediate assembly level. Although the LLVM IR is not directly exposed to the user, LLFI-GPU attaches a dynamic library to *nvcc* which can intercept the call to the LLVM compilation module [71]. At this point, the passes of LLFI-GPU are invoked to instrument the program. LLFI-GPU then returns the instrumented LLVM IR to *nvcc*, which proceeds with the rest of the compilation process to transform it into PTX code. Hari *et al.* proposed SASSIFI which is a Pin-like tool for injecting faults at the GPU assembly level (SASS) and analyzing the propagation of faults during program execution [27, 45]. Li *et al.* introduced Trident, a model that captures error propagation at an instruction level using a static analysis [72]. Their model tries to predict application vulnerability by evaluating the probability for a fault to propagate, measured at the granularity of an instruction. However, their work is limited to CPUs and has only been evaluated on single-threaded programs.

Previlon *et al.* have studied the impact of input data and execution parameters on the resiliency of GPU applications [73, 74]. They have reported that random changes in the input values do not alter the vulnerability behavior of the applications significantly, unless the inputs are extremely

biased. Their work, *PCFI*, exploits the predictability in fault-injection outcome based on the program counter that is impacted by soft-errors [75], thereby reducing the number of fault injections. Our work in this thesis 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.

3.3 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. 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 [76]. 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 [62]. 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.* [77]. However, their work is focused on understanding the resilience of GPU applications, and does not use the categorization for resilience prediction.

X. Li *et al.* use tainted analysis on microarchitectural registers to approximate the effects of faults injected into these registers [78]. 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 [79]. Walcott *et al.* [80], Biswas *et al.* [81], and Duan *et al.* [82], 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 [83]. Moreover, Farahani *et al.* dynamically predict the vulnerability of a program during its execution [84]. 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 are suited for GPU applications.

3.4 Redundancy-based Fault Mitigation on GPUs and CPUs

In order to detect errors in the execution state caused by transient faults, most fault protection techniques ultimately rely on some sort of redundancy (either temporal or spatial). We reflect on our approach in the context of prior research in the field of software and hardware-based fault mitigation on GPUs and CPUs.

Software Redundant Multithreading (RMT): Wadden et al. proposed RMT, which is implemented at a software-level targeting GPUs. RMT creates two copies of a thread, a leading thread and a trailing thread [17]. The leading thread places the value in a local or global buffer at every store instruction, and the trailing thread verifies the value and commits the executed instructions. This inter-thread communication requires synchronization for correct operation, which causes significant performance and power overheads. Subsequent work by Gupta et al. proposed compiler optimizations to reduce the high synchronization overhead of RMT on GPUs [85]. Typically, software-based instruction-level duplication does not incur the high synchronization overhead experienced by RMT.

Moreover, programmer intervention is needed in RMT to modify the source code to ensure spare hardware resources are available for the redundant threads.

Software Instruction Duplication: SInRG [19] implements instruction duplication at a compiler level. Unlike our work, SInRG is not selective, as it does not employ heuristics to identify instructions eligible for duplication. Instead, SInRG duplicates the same classes of instructions in every application, regardless of their resiliency. This approach can cause unnecessary duplication and verification overhead in applications. Since SInRG was implemented in NVIDIA's proprietary CUDA backend compiler, it provides tighter control over the SASS instructions generated in the final binary. The authors proposed specialized ISA instructions to optimize the duplication and verification processs in order to limit any extra performance overhead. SInRG and ArmorAll focus on optimizing different aspects of redundancy through the compiler – SInRG optimizes the verification process, whereas ArmorAll optimizes instruction selection. Therefore, integrating SInRG with ArmorAll's proposed techniques should further improve performance and paves a path for future work.

On the CPU front, SWIFT duplicates all computational instructions and no memory instructions, thereby reducing memory pressure and improving performance [23]. It also proposes a control flow checking (CFC) mechanism to detect control flow errors. Trident calculates the SDC probability on a per-instruction basis using the values of the input operands and instruction opcodes, only duplicating those instructions that have a high SDC probability [72]. In Shoestring [24], the authors propose a compiler technique that combines symptom-based error detection with selective instruction duplication. Their approach statically identifies *Symptom Generating* and *High Value* instructions and duplicates all instructions that appear between *Safe* and *High Value* instructions. In order to determine the probability of whether an instruction is safe, they use a *distance* metric, which is the number of statically scheduled instructions between producer and consumer instructions. However, the distance between instructions could vary, depending on the code scheduler, and is therefore prone to inaccuracies.

Moreover, these CPU-based techniques are evaluated on single-threaded applications. Applying these techniques directly on GPUs without modifying them would lead to a significant loss in performance. This is because GPUs and CPUs are architecturally quite different. On a GPU, the register usage can impact the number of concurrent threads that can be executed on the SM. CPUs do not experience many of these constraints, and hence, these techniques must be optimized before applying them to GPUs.

Data Flow analysis has been explored in test-suite reduction algorithms to identify a subset of the test data that would maximize the error detection rate in software [86]. However, such analysis

has not been explored in prior work for protecting hardware against transient faults. Therefore, unlike previous studies, our work attempts to use information flow in order to make a more informed decision about which instructions should be selected for duplication, without any additional hardware support.

Hardware Redundancy Schemes: Error tolerance through hardware can be achieved by using N-modular redundancy across computer nodes. The most popular forms of this redundancy are Dual or Triple Modular Redundancy, as deployed in business-class systems such as the IBM Z Series [87]. The main problem with DMR and TMR is that they incur prohibitive energy and area overheads, which makes them unsuitable for deployment in many classes of systems. Hardware mechanisms, such as RISE and Warped-DMR, take advantage of under-utilized resources on GPUs for error detection. RISE includes techniques to predict and leverage idle SM cycles and compute cores to execute redundant work [21]. Warped-DMR provides error detection capabilities within a warp (Intra-warp DMR) and across warps (Inter-warp DMR) by exploiting underutilized resources on the GPU [20]. Both RISE and Warped-DMR require complex hardware changes and are not directly comparable with our approach.

Memory protection schemes: For protecting the contents of storage elements such as the register file, caches, and main memory, ECC is applied to ensure error-free execution of the program. Schemes such as ECC and parity can detect faults only *after* valid data is written to the register file/memory and correction codes are properly generated (if applicable). In contrast, our work focuses on error detection in the datapath of the instruction *before* the output is written back to the register or memory.

Chapter 4

Vulnerability Characterization

Modern GPUs exploit data parallelism in application kernels to achieve high performance and efficiency. However, there is a loss in efficiency due to redundant execution when the threads perform the same operations on the same data. Researchers have reported that, on average, about 42% of dynamic instructions in a warp are scalar (i.e., all active threads operate on the same data [88]). To take advantage of these scalar instructions, AMD's Graphics Core Next (GCN) architecture has a scalar co-processor in each compute core, along with a separate scalar register file [89]. Lee *et al.* proposed a scalarizing compiler that uses convergence and variance analysis to statically identify scalar values and instructions [90]. Because there is no information about the input during compile time, static analysis is only able to identify instructions that are *strictly scalar* or guaranteed to be scalar during execution. While their compiler is able to identify two-thirds of the dynamic scalar opportunities, more opportunities can be extracted dynamically. In this thesis, we use dynamic analysis for identifying scalar opportunities in GPU kernels. The goal of this analysis is to understand how error propagation differs when a transient fault occurs in a scalar instruction, versus vector.

In this Chapter, we will first characterize the workloads in terms of their scalar and vector intensities. We will then perform fault injections using SASSIFI in scalar and vector instructions to analyze their vulnerability. This analysis will provide insight into potential architectural support required to improve the reliability of scalar instruction execution on GPUs. We will further expand our set of characteristics which will allow us to build a more sophisticated learning-based framework to predict the reliability of GPU applications. Finally, we will describe dependence analysis that allows the compiler to identify a subset of instructions based on program characteristics and facilitate selective fault protection in GPU apps.

4.1 Analyzing the Vulnerability of Vector-Scalar Execution

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) [88, 90]. 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.

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
```

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 (a 1 for an active lane and a 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.

Using these counters, we quantify the application's scalar and vector intensities as follows:

$$\text{Scalar Intensity} = \frac{\text{\# of dynamic scalar instructions}}{\text{Total \# of dynamic instructions}} \tag{4.1}$$

$$\text{Vector Intensity} = \frac{\text{\# of dynamic vector instructions}}{\text{Total \# of dynamic instructions}} \tag{4.2}$$

$$= 1 - \text{Scalar Intensity} \tag{4.3}$$

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.

4.1.1 Evaluation Methodology

We inject 1000 random single bit flips in the destination register value (IOV) of each Scalar and Vector instruction (a total of 2000 injections per application). Once a fault is injected, the outcome of the program can fall into one of the three categories - Masked, DUE, or SDC, as

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

Acronym	Application
atomic add	Integer Atomic Vector Addition
bfs	Breadth-First Search
histo	Saturating Histogram
lbm	Lattice-Boltzmann Method Fluid Dynamics
mri-gridding	Magnetic Resonance Imaging - Gridding
mri-q	Magnetic Resonance Imaging - Q
sad	Sum of Absolute Differences
sgemm	Dense Matrix-Matrix Multiply
tpacf	Two Point Angular Correlation Function

Table 4.1: Applications selected from Parboil used in our evaluation.

described in Chapter 1. For a fair comparison, we scale the number of outcomes for each instruction type by their intensities. For example: error outcomes caused by 1000 scalar injections [Scalar Masked, Scalar DUEs, Scalar SDCs] are multiplied by the factor of Scalar Intensity (from Equation 4.1) and error outcomes caused by 1000 vector injections [Vector Masked, Vector DUEs, Vector SDCs] are multiplied by the factor of Vector Intensity (from Equation 4.2). Due to this scaling, our total injections are reduced to 1000 random injections, which will be reflected in our results described in the next Section. We use nine applications from Parboil benchmark suite for our evaluation [91], as shown in Table 4.1.

4.1.2 Results

In this section, we first present the overall error profile resulting from scalar and vector fault injections. To better understand the source of each outcome, we evaluate the correlation between program opcodes (for the instructions where the faults are injected) and the error outcomes.

4.1.2.1 Overall Outcomes

Figure 4.1 shows the outcomes for the studied GPU applications, along with the scalar intensity of each application. These outcomes (out of 1000) are calculated as follows:

$$\text{Masked} = \text{Scalar Masked} + \text{Vector Masked}$$

$$\text{SDCs} = \text{Scalar SDCs} + \text{Vector SDCs}$$

$$\text{DUEs} = \text{Scalar DUEs} + \text{Vector DUEs}$$

$$\text{Overall Outcomes} = \text{Masked} + \text{SDCs} + \text{DUEs}$$

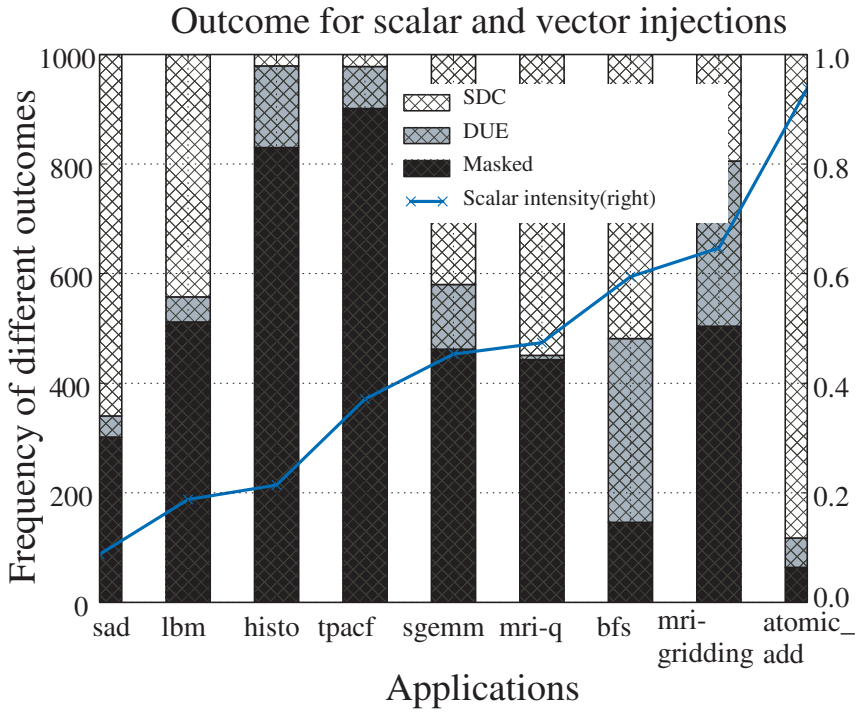


Figure 4.1: Fault outcome frequency for our GPU applications, along with their scalar intensities.

Scalar SDCs, Scalar DUEs, and Scalar Masked are SDCs, DUEs, and Masked outcomes that occur due to faults in scalar instructions, respectively, whereas Vector SDCs, Vector DUEs, and Vector Masked are the SDCs, DUEs and Masked outcomes that occur due to faults in vector instructions. As mentioned earlier, the original number of outcomes are scaled by their associated intensity. When the scalar intensity is high, the probability of a fault occurring in a scalar instruction is high. When the scalar intensity is low, the probability is much higher that any particle strike will occur in a vector instruction. From Figure 4.1, we observe that applications that have higher scalar intensity, such as atomic add, mri-q, sgemm, and mri-gridding, experience a high number of SDCs and DUEs. Applications such as histo and tpacf have relatively low scalar intensity, and show high masking of errors. The probability of a particle strike occurring in a scalar instruction is low for these applications.

We can quantify the relationship between different outcomes with scalar intensity by measuring the correlation. We use Pearson's Correlation Coefficient [92], r, to calculate the linear correlation between two variables x and y, as follows:

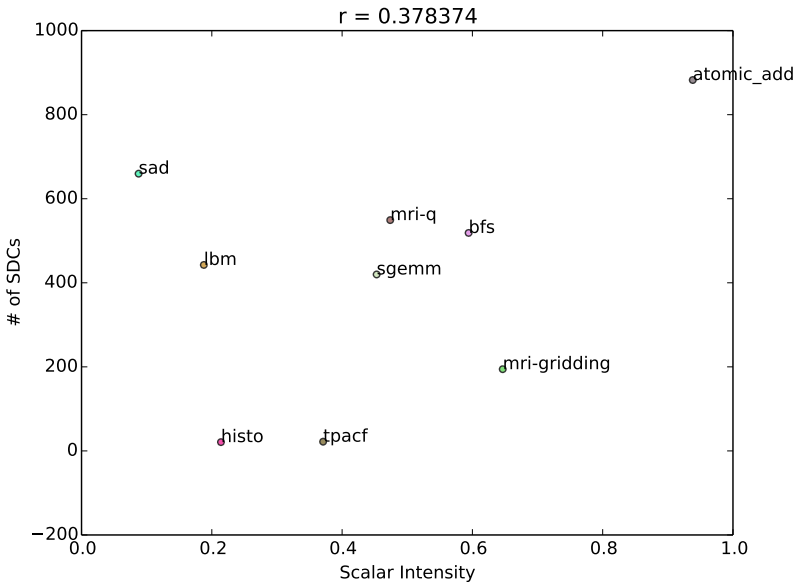


Figure 4.2: Correlation between scalar intensity and the overall SDCs.

$$r_{xy} = \frac{\sum_{i=1}^{n} (x_i - \overline{x})(y_i - \overline{y})}{\sqrt{\sum_{i=1}^{n} (x_i - \overline{x})^2 \sum_{i=1}^{n} (y_i - \overline{y})^2}} \tag{4.4}$$

The values for r range between -1 and 1, where 1 means a high positive correlation, 0 means no correlation, and -1 means a high negative correlation.

$$r_{xy} = \begin{cases} \text{Negative Correlation}, & \text{if } r = -1 \\ \text{No Correlation}, & \text{if } r = 0 \\ \text{Positive Correlation}, & \text{if } r = 1 \end{cases}$$

The coefficient of correlation, r, between scalar intensity and total SDCs is found to be 0.37, as shown in Figure 4.2. This indicates a low positive correlation. Furthermore, the value of r is 0.31 for scalar intensity and DUEs, and -0.5 for scalar intensity and Masked, as shown in Figures 4.3 and 4.4, respectively. This could signify that SDCs and DUEs are expected to increase as the scalar intensity of the application increases, while the likelihood of masking is reduced with increased scalar intensity.

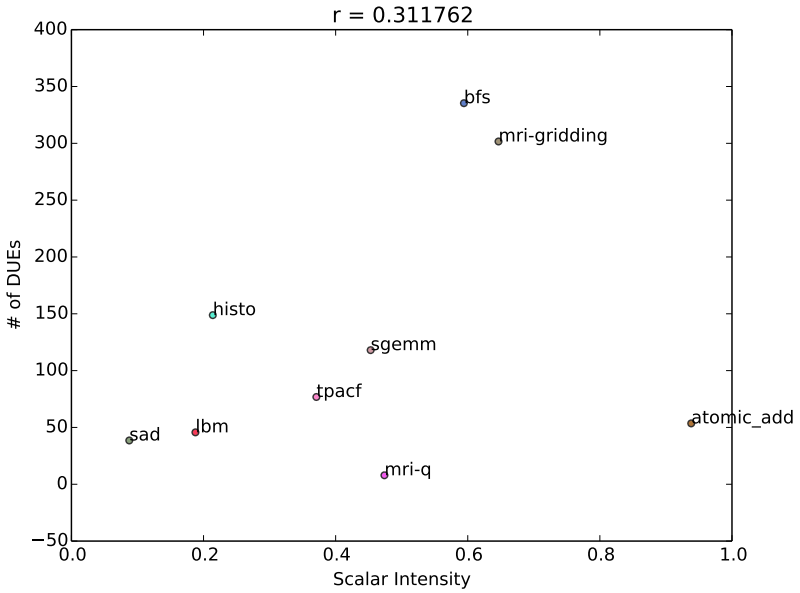


Figure 4.3: Correlation between scalar intensity and the overall DUEs.

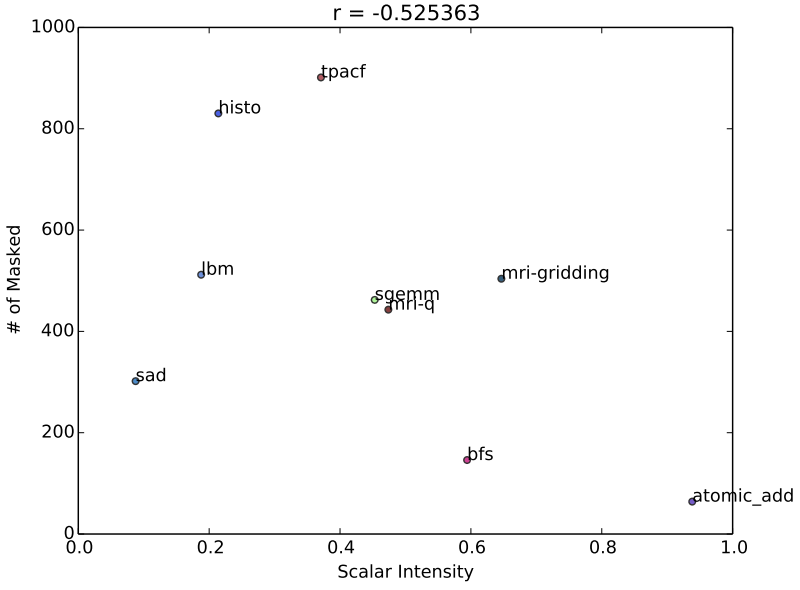


Figure 4.4: Correlation between scalar intensity and the overall Masked.

Opcode	Description
IADD	Integer Add
ISETP	Integer Set Predicate
MOV	Data Movement

Table 4.2: Instruction opcodes from NVIDIA Kepler instruction set.

Scalar Opcode	SDC	DUE	Masked
IADD	-0.5464	0.393	0.401
ISETP	-0.596	0.122	0.567
MOV	-0.599	0.0656	0.596

Table 4.3: Correlation between scalar instances of the opcodes with overall SDCs, DUEs, and Masked.

4.1.2.2 Correlation between outcomes and opcodes

Next, we select three commonly occurring opcodes among the set of chosen applications (described in Table 4.1) and calculate their correlation with the outcomes. These opcodes are described in Table 4.2. The ISETP instruction sets a predicate register, the value of which determines whether an instruction will be executed (or not). As mentioned earlier, when an instruction executes multiple times, it could be scalar on one instance and vector on another, depending on the nature of application and the input operands. We capture the scalar and vector instances of these three opcodes from the program and calculate their correlation with different outcomes. As shown in Figures 4.5 and 4.6, scalar instances of IADD present in our applications show low positive correlation with DUEs, whereas vector instances of the same opcode show no or low negative correlation with DUEs. This could mean that a fault in a scalar IADD instruction is more likely to crash versus a vector IADD instruction. This prompts us to calculate the value of r for all other opcodes and outcomes to verify whether a similar trend exists. We report these results in Tables 4.3 and 4.4.

We observe a similar trend with the scalar ISETP and MOV opcodes, which exhibit low positive correlation with the DUEs, and vector instances show slightly higher masking. However, the number of SDCs seem to be unaffected by the nature of the opcode. This could mean that a linear correlation may not be sufficient to characterize the relationship between the opcodes and the SDCs, and other non-linear correlation mechanisms must be explored.

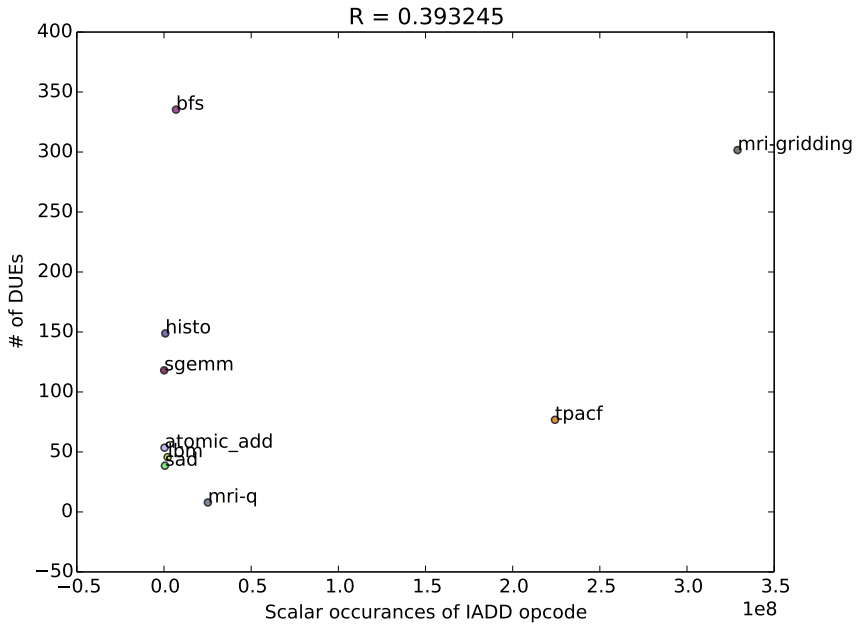


Figure 4.5: Correlation between scalar instances of IADD with overall DUEs.

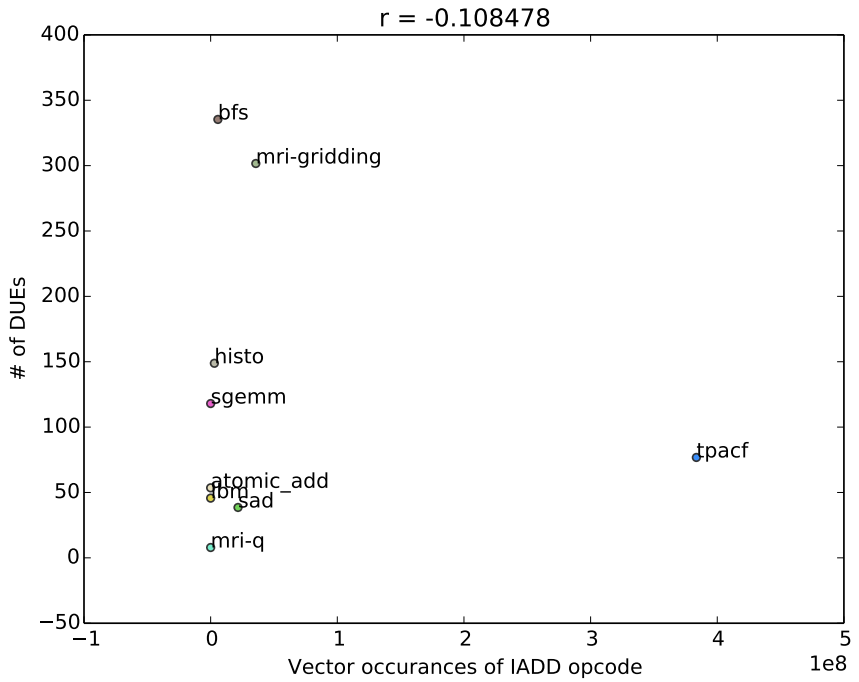


Figure 4.6: Correlation between vector instances of IADD with overall DUE.

Vector Opcode	SDC	DUE	Masked
IADD	-0.526	-0.1085	0.592
ISETP	-0.556	-0.0380	0.5934
MOV	-0.578	-0.0208	0.610

Table 4.4: Correlation between vector instances of the opcodes with overall SDCs, DUEs, and Masked outcomes.

4.2 Workload Characterization for Resilience Prediction

As we mentioned earlier, once injections are run on all applications, SASSIFI generates the instruction and error profile for each application. In this section, we extend the type of instructions, beyond scalar and vector instructions, to characterize the 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, are instructions that write to predicate registers.
- *Kernel Register Usage*: the number of general purpose registers used by the kernel.

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 flow graph, which plays an

Metric	**Kind**	**Synopsis**	**Example of opcodes included** [93]
Control Flow Intensity	Scalar	Total # of dynamic Control Flow Instructions/N	BRA, JMP, BRX, JCAL
Data Movement Intensity	Scalar, Vector	Total # of dynamic Data Movement Instructions/N	MOV, SHFL, PRMT
Floating Point Intensity	Scalar, Vector	Total # of dynamic Floating Point Instructions/N	FADD, FMUL, FMAD
Integer Arithmetic Intensity	Scalar, Vector	Total # of dynamic Integer Arithmetic Instructions/N	IADD, IMUL, MAD
Logic Intensity	Scalar, Vector	Total # of dynamic Logic Instructions/N	LOP, SHL, SHR
Load Intensity	Scalar, Vector	Total # of dynamic Load Instructions/N	LD, LDS, LDC, LDG
Predicate Intensity	Scalar, Vector	Total # of dynamic Predicate Instructions/N	ISETP, FSETP, PSETP
Store Intensity	Scalar, Vector	Total # of dynamic Store Instructions/N	ST, STS, STL
Register Usage	-	# of General Purpose Registers used by the kernel	All opcodes
Scalar Intensity	Scalar	Total # of dynamic scalar instructions/N	Scalar instances of all opcodes
Vector Intensity	Vector	Total # of dynamic vector instructions/N	Vector Instances of all opcodes
Total features	**18**		

Table 4.5: 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.

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 4.5.

4.3 Workload Characterization for Fault Mitigation

In this section, we describe 'dependence analysis' that allows the compiler to identify a subset of instructions based on program characteristics and facilitate selective fault protection in GPU apps. Our redundancy schemes leverage dependence analysis, which generates constraints on the execution order of instructions [94]. Dependences represent two different kinds of constraints on

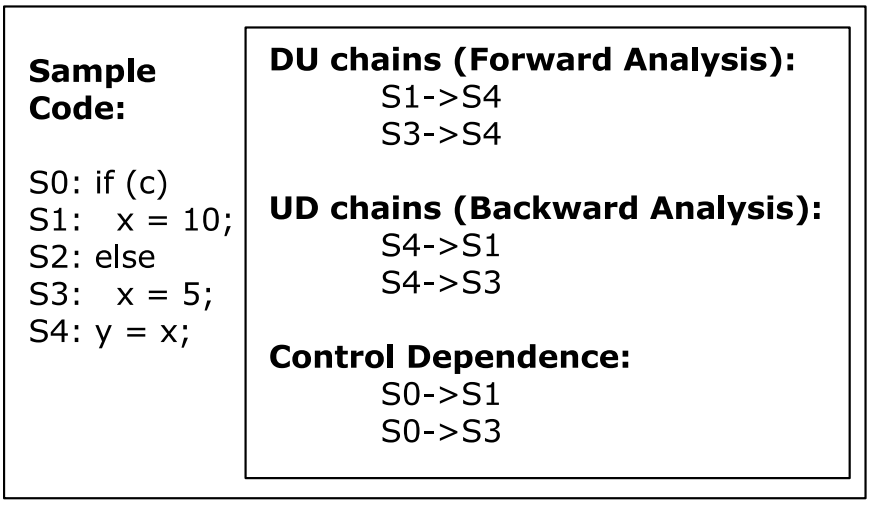


Figure 4.7: An example of Data Dependence and Control Dependence Evaluation.

program transformation – 1) data dependence and 2) control dependence. Data dependence arises due to constraints designed to ensure that data is produced and consumed in the correct order. Control dependence arises when an instruction's execution depends on the outcome of a preceding instruction. We track both data and control dependence in the program, since maintaining these constraints is essential in guaranteeing program correctness.

Data Dependence: Data Dependence is usually performed by tracking the data flow in the CFG and constructing a data dependence graph (DDG). This Data Flow Analysis (DFA) can be done either statically or dynamically. Unlike dynamic DFA, static DFA does not require the program to be executed at all on a machine. Depending on the direction of the analysis, DFA can be performed as *forward* analysis or *backward* analysis.

One of the auxiliary data structures constructed during DFA are Use-Def (UD) chains (and their counterpart, Def-Use (DU) chains). We describe how to construct UD chains and DU chains using the example shown in Figure 4.7. In the code provided, variable x is 'defined' at statement S1 and S3, while it's value is 'used' in statement S4. A DU chain links each *definition* of *x* to those *uses* that a definition can reach. Therefore, they are generated through a forward traversal from S1 (the *def* of x) to S4 (the *use* of x), and similarly S3 to S4, given that no other definition of variable *x* exists on the path between S1→S4 and S3→S4. Alternatively, UD chains link each *use* of variable x to the *definition* which reaches that *use*. Therefore, they are generated through a backward traversal from S4 (the *use* of x) to S1 and S3 (the *def* of x), given that no other definition of variable x exists between S4→S1 and S4→S3.

LLVM uses Static Single Assignment (SSA) form as its primary code representation, which ensures that each variable is written only once, and each use of a register is dominated by its definition [30]. To maintain the SSA form, a phi (Φ) node is often used to select a value depending on the predecessor of the current block. A Φ node is an IR instruction, which is not implemented in the ISA for most architectures. The SSA property of LLVM simplifies the creation of UD chains, which are a key component in our implementation.

Control Dependence: An instruction, *x*, is control dependent on another instruction, *y*, if *y* controls whether or not *x* is executed. Such instructions include simple branching instructions if_icmpeq, if_icmpne, if_icmplt, if_icmple, if_icmpgt if_icmpge, etc. Since these instructions can control which instructions execute, they are the direct source of a control dependence. As shown in Figure 4.7, S3 and S1 are control dependent on S0. However, S4 does not depend on S0 because S4 is always executed irrespective of the outcome of S0. We ignore some sources of control dependences mentioned such as unconditional branching instructions, eg: br label %bb_label. LLVM adds an unconditional branch instruction at the end of each basic block. These instructions are an artifact of LLVM and do not appear in the final assembly code generated by the backend. Moreover, though these instructions can change the flow of control for instruction execution, they are usually used with conditional branching instructions, and hence, were left out of consideration.

The Use-Def Chains created using DFA and Control Dependence Analysis will be leveraged by ArmorAll for fault mitigation, described in Chapter 6.

4.4 Summary of Workload Characteristics for Reliability

- We implement an analysis pass in the profiling phase of SASSIFI to identify scalar and vector instructions.
- We find the correlation of three most commonly occurring scalar and vector opcodes with different outcomes. We observe that while some scalar opcodes show positive correlation with the outcomes, others do not.
- We then extend the set of workload characteristics, beyond scalar and vector, based on the hardware resources they stress during execution to accurately capture their behavior. These characteristics will be used to drive our prediction models described in the next chapter.
- We describe dependence analysis that allows the compiler to identify a subset of critical instructions, based on the vulnerability behavior of the application.

Chapter 5

Error Prediction Using PRISM

In this Chapter, we extend our analysis of scalar-vector vulnerability, from the previous chapter, to develop a more sophisticated learning-based framework called PRISM. We will describe the phases of PRISM in more detail. PRISM framework includes four different phases, as shown in Figure 5.1.

5.1 Data Collection Using SASSIFI

The first step required to build any learning-based model is to collect the data. For our study, we have selected a diverse set of 50 regular and irregular applications from a variety of domains, as shown in Table 5.1. These workloads have been taken from the CUDA SDK, Lonestar, NUPAR, Parboil, and Rodinia benchmark suites [91, 95, 96, 97, 98]. 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 over-written by the actual kernel execution. We, therefore, eliminated all warm-up kernel executions. For error checking,

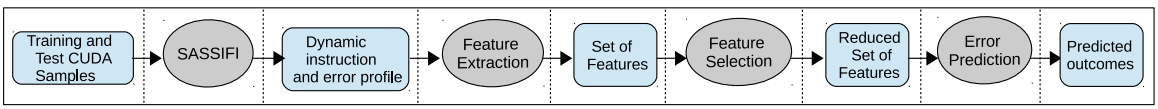


Figure 5.1: The PRISM Framework. The ovals represent the processing nodes/phases, whereas the rectangles represent input/output to/from the processing nodes.

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

we compared the output of the kernel when the fault is injected with the golden output (without any fault injection). We perform 1000 fault injections in each application to attain an error margin of 3.1%, with a 95% confidence level [47]. 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, as described in Chapter 4, to collect the execution and error profiles of each application.

5.2 Feature Extraction

Once we have the data for all applications in the form of their execution and error profiles, we process them to generate micro-architectural agnostic characteristics or features, as described

in Table 4.5. These features characterize the GPU workloads based on the hardware resources they stress during execution to accurately capture their behavior. We leverage these characteristics and use them as our *feature set*.

5.3 Feature Selection

In this phase, we prune the above *feature set* by performing *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 thesis, we use a forward selection wrapper method to select a subset of relevant features [99]. 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 [92, 100]. 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 [101]. 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 5.2: 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.

5.4 Error Prediction

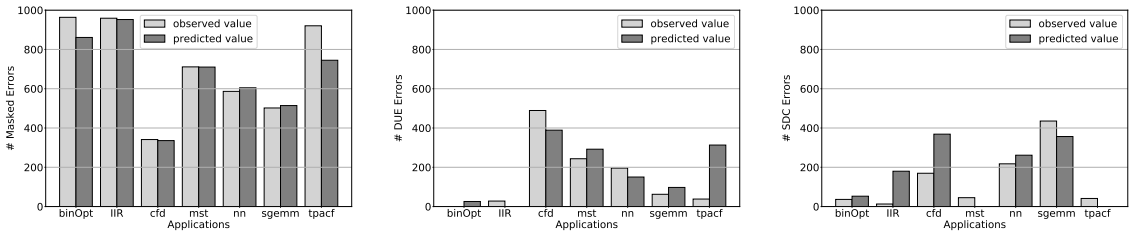
Statistical Regression Analysis is a set of techniques to estimate the relationship between a single dependent variable and multiple independent variables. In this chapter, we explore two models: Ridge Linear Regression and K-Nearest Neighbor [102, 103]. As mentioned earlier, the wrapper method for feature 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 5.2.

5.4.1 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 [104]. The model can be stated as follows:

$$y = b_0 + b_1 x_1 + b_2 x_2 + ... + b_i x_i + e_i \tag{5.1}$$

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



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

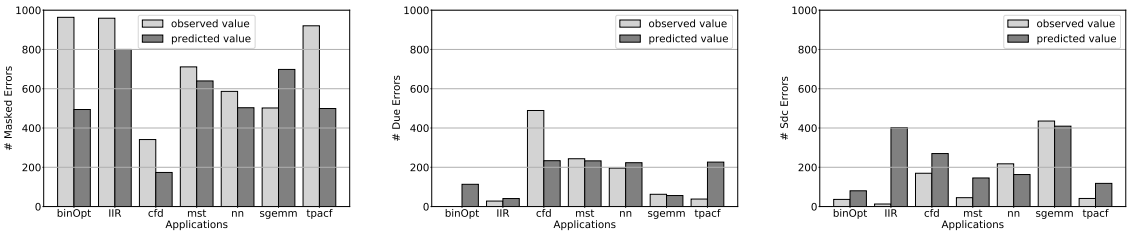
Figure 5.2: Observed and Predicted values of Masked, DUE, and SDC outcomes using the Ridge Regression Model. The accuracy of prediction for Masked errors is 90%. For DUEs, with the exception of *tpacf*, our predicted values are close to the observed values.

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 thesis 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 [105].

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 5.2. 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 findings as follows:

Key findings:

- 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



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

Figure 5.3: 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.

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 thesis.

- 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 [94].

- 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.

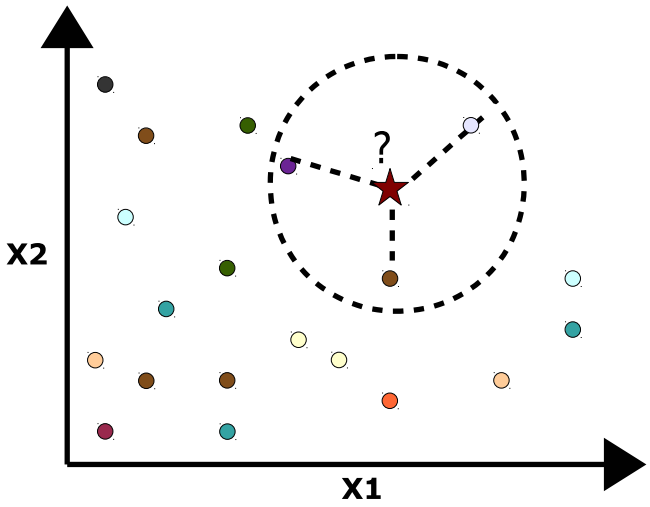


Figure 5.4: Demonstration of K-Nearest Neighbor (K = 3).

5.4.2 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, 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 5.4. 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 5.5 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

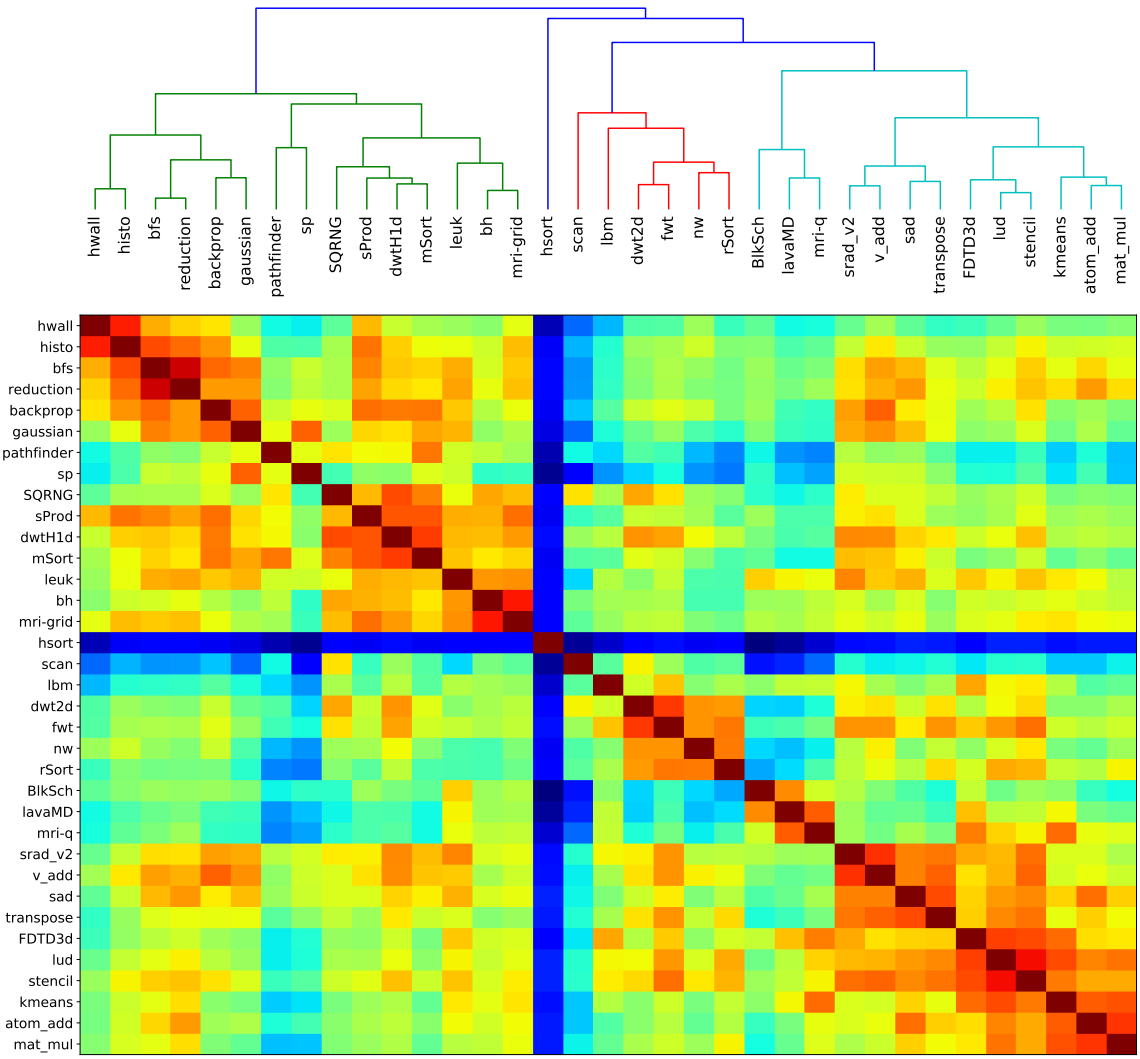


Figure 5.5: Visualization of similarities between applications using a dendrogram at the top. In the heatmap, Red represents similarity, whereas Blue represents dissimilarity. In the dendrogram, the vertical distance is a measure of similarity. Shorter distance means more similarity.

SDCs, respectively, through cross-validation. Figure 5.3 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 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* [106]. 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

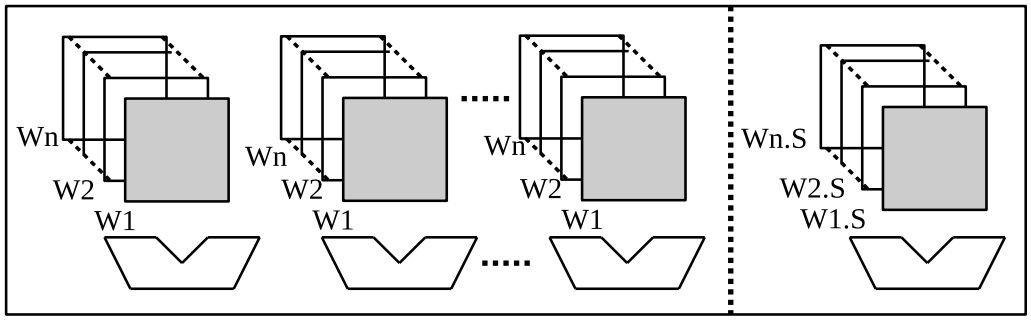


Figure 5.6: Spatial SIMT Architecture with a separate Scalar Unit.

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.

5.5 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 [89]. A warp (or *wavefront*, in OpenCL terminology), is mapped across the SIMD lanes with one thread per SIMD [107]. 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 5.6) 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.

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 4.5, 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.

5.6 Summary of PRISM

To our knowledge, this is the first work to predict resiliency of GPU applications using statistical methods. We summarize the contributions and findings of this chapter as follows:

- 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 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 chapter, we explore two prediction models - one based on Linear Regression and the other based on similarity analysis (K-NN).
- 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.

Chapter 6

Fault Mitigation using ArmorAll

In this chapter, we will introduce three compiler-based redundancy schemes that constitute ArmorAll [108]. All three schemes leverage dependence analysis and UD chains, covered in Section 4.3, to create a sequence of instructions, which we refer to as a *path*. A path, for instance, contains a sequence of instructions – $\{u_n, u_{n-1}, u_{n-2}, ..., u_1, u_0\}$, where:

- u_i: *use* of the value v_i, which is *defined* in u_{i-1}, where $i \in \mathbb{N}$ or, u_i: is an instruction which is control dependent on u_{i-1}, where $i \in \mathbb{N}$.
- The last instruction in the path is u_0: *def* of the value *used* in u_1.

Since the path is generated through a *backward* walk, we use descending order to enumerate the instructions.

A path always begins from the *root* instruction, u_n, and terminates when no more instructions can be traversed in that path. However, an application may contain multiple paths, depending on the number of root instructions identified. It is possible for an instruction to be part of multiple paths; in such cases, the instruction is only considered once. The two factors that distinguish the three schemes we propose are: 1) their root instruction, and 2) the final worklist constructed. The root varies depending on the entity that is being protected. The entity could be an address, a value, or both. The worklist includes an unordered set of instructions that appear on the path being traversed, starting from the root.

Once the final worklist is created, all instructions in the worklist (with a few exceptions) are considered *duplication-eligible* and subsequently duplicated. The duplicate instruction reads the same source operands as the original instruction, but writes to a new destination register. The destination operands of the instruction are compared with their clones via additional verification

instructions. If the verification fails, a mismatch warning is issued to the user. We also do not duplicate Φ operations (if they appear in the worklist) and unconditional branch instructions, as these instructions do not translate into actual instructions in the final binary, as mentioned in Chapter 4. Next, we describe our three schemes, and how they protect these entities, in greater detail.

6.1 Address Armor

Key Idea: As the name suggests, Address Armor protects the *addresses* used by memory instructions. Illegal memory accesses and addressing exceptions can be caused by transient faults [109]. They are often the result of an address being corrupted either due to bit flips in registers that hold addresses, or the propagation of a fault into one of those registers. Therefore, the goal of Address Armor (AA) is to track all instructions that participate in memory address computation. Since both load and store instructions access memory, they naturally become the root for this scheme. Address Armor *does not* protect the value loaded or stored by the instructions, only their addresses.

Implementation: In LLVM, the instruction *getelementptr* is typically used to compute addresses for both load and store instructions. Therefore, the Address Armor first initializes a worklist with all root/*getelementptr* instructions present in the program. For each *getelementptr* instruction in the worklist, it tracks all instructions that appear in the data and control dependence path and adds them to the worklist, if not previously added. This process continues until the path for all *getelementptr* instructions is completely traversed, and no further instructions can be added to the worklist. The final worklist, S_{AA} is a set of instructions, expressed as the union of all paths $\bigcup P_j$, where P_j is the sequence of instructions traversed, starting from the root $j \in \mathbb{N}$.

In Address Armor, we do not duplicate load and store instructions themselves. Since the addresses of these instructions are protected by Address Armor, there is no need perform each operation twice. One might argue that corruption in their value can still lead to an incorrect output. However, protecting the value is not the objective of Address Armor. We relax the constraints by assuming that when an application is protected using Address Armor, the likelihood of the values getting corrupted is low.

Summary of Address Armor:

- Goal: To protect *addresses* used by load and store instructions.
- Root: Load and Store instructions.

- Which instructions from the worklist are duplicated? ALU, Floating point, Compare, and getelementptr instructions.
- Which instructions are *not* duplicated? Loads, Stores, Barriers, Phi, and Unconditional Branch instructions.

6.2 Value Armor

Key Idea: This scheme, as its name suggests, protects the *values* written to memory by store instructions. An incorrect output is often a result of corruption caused by either a bit flip in a register that holds an output value, or propagation of a fault into one of those registers. Therefore, the goal of Value Armor (VA) is to track all instructions that contribute to the computation of the output value. Since the output value is written to memory using store instructions, they become the root instruction for this scheme. Value Armor does not protect the addresses used by a store (or a load) instruction, only the values.

Implementation: Value Armor first initializes a worklist with all the root instructions (here, store instructions). For each root instruction in the worklist, it tracks all instructions that appear in the data and control dependence path and adds them to the worklist. A path terminates when it either encounters a load instruction, or if there are no further instructions that can be added to the path. If Value Armor keeps tracking beyond load instructions (Syntax: %val = load i32* %ptr), then its path will include instructions that compute the address of the load, which is contrary to the goal of the Value Armor. Therefore, all instructions are added to the worklist until one of the termination conditions (described above) is encountered. The final worklist, S_{VA} is a set of instructions, expressed as the union of all paths $\bigcup P_k$, where P_k is the sequence of instructions traversed, starting from the root $k \in \mathbb{N}$.

Since the value must be protected by Value Armor, we also duplicate the load instructions that appear on the worklist, unlike Address Armor. In this case, we assume that when an application is protected using Value Armor, the likelihood of the address being corrupted is low.

Summary of Value Armor:

- Goal: To protect *values* used by load and store instructions.
- Root: Store instructions.

- Which instructions from the worklist are duplicated? ALU, Floating point, Compare, Load, and getelementptr instructions.
- Which instructions are *not* duplicated? Stores, Phi, Barriers, and Unconditional Branch instructions.

6.3 Hybrid Armor

Key Idea: This scheme, as its name suggests, protects both – the *values and addresses* of load and store instructions. The goal of Hybrid Armor (HA) is to track all instructions that contribute to the computation of the output value *and* addresses. Since the output value is written to memory using store instructions, they are also the root for this scheme.

Implementation: The Hybrid Armor scheme first initializes a worklist with all of the root instructions (i.e., store instructions). Similar to AA and VA, for each root instruction in the worklist, Hybrid Armor tracks (using a backward traversal) all instructions that appear in the data and control dependence path and adds them to the worklist. The major difference between Hybrid Armor and Value Armor is that Hybrid Armor does not terminate at load instructions. The path for Hybrid Armor must include instructions that compute the address of the load and store instructions, as well as their values. This process continues until the path for all *store* instructions is completely traversed, and no further instructions can be added to the worklist. The final worklist, S_{HA} can be expressed as the union of set of instructions in Address Armor and Value Armor, $S_{AA} \cup S_{VA}$, where S_{AA} and S_{VA} are the final worklists generated by Address Armor and Value Armor, respectively. Since both the value and address of a store are protected by Hybrid Armor, there is no need to duplicate the store instruction.

Summary of Hybrid Armor:

- Goal: To protect both – *values and addresses* used by load and store instructions.
- Root: Store instructions.
- Which instructions from the worklist are duplicated? ALU, Floating point, Compare, Load, and getelementptr instructions.
- Which instructions are *not* duplicated? Stores, Phi, Barriers, and Unconditional Branch instructions.

6.4 Evaluation Methodology

6.4.1 Fault Injection Framework:

In this chapter, we use an LLVM-based fault injection framework called LLFI-GPU [70]. The benefits of using LLVM are that it offers portability and modularity by isolating itself from the high-level language, as well as from the ISA and other hardware-specific details. LLFI-GPU captures the faults that are visible at the application level and are not masked by the hardware. Though we use a NVIDIA Kepler K20 to perform our fault injection campaign, it is important to note that our redundancy schemes are not tied to any specific GPU architecture. The choice of the Kepler is tied to the LLFI-GPU toolset.

LLFI-GPU is hosted in the nvcc compiler stack, labeled as the 'fault injector' in Figure 2.5. The LLVM IR is not directly exposed to the user; LLFI-GPU attaches a dynamic library to *nvcc* which can intercept the call to the LLVM compilation module [71]. At this point, the passes of LLFI-GPU are invoked to instrument the program. LLFI-GPU then returns the instrumented LLVM IR to *nvcc*, which proceeds with the rest of the compilation process to transform it into PTX code. Our compilation passes are implemented as the last step in the LLFI-GPU framework, in order to leverage the existing LLVM optimizations and to ensure that no LLVM optimization can further modify the code. We further analyzed the SASS code of all CUDA applications using the NVIDIA tools, *cuobjdump* and *nvdisasm*, to ensure that our code was not optimized by the *ptxas* backend [93]. The output of the duplicate instructions are verified with the original instructions to notify the user of a mismatch, and hence they are not treated as dead code by the backend. The fault model of LLFI-GPU considers transient hardware faults that occur in the computational elements of the GPU, including pipeline stages, flip-flops, arithmetic and logic units (ALUs), and the register file [70]. It assumes GPU memory or cache, its control logic, and instruction encodings are protected with ECC.

6.4.2 Benchmarks Evaluation

We evaluate the error detection capability of ArmorAll by performing an extensive fault injection campaign on a diverse set of CUDA applications from the CUDA SDK [97]. We target a 95% confidence interval and a 3.1% error margin, which can be achieved by performing 1000 injections in each application [47]. In prior work, a 95% confidence level with $<$5% error margin has been deemed accurate [27, 110]. We inject one fault per run of the application. When checking for errors, we compared the output of the kernel when the fault is injected with the golden output

Outcome	Synopsis
Masked:undetected	Fault was masked and no mismatch was detected
Masked:detected	Fault that would have potentially been masked, but a mismatch was detected by the Armor
SDC:undetected	SDC that was not detected by the Armor
SDC:detected	SDC that was detected by the Armor
DUE:undetected	Crash/DUE that was not detected by the Armor
DUE:detected	Crash/DUE that was detected by the Armor

Table 6.1: Possible error outcomes for ArmorAll redundancy schemes. We allow the program to continue execution, even if a mismatch is detected. This enables us to evaluate the accuracy of the detection scheme.

(without any fault injection). For precision-based applications, we used the default values of the L1 and L2-Norm provided in the benchmark, which typically ranged between 5e-4 to 1e-6. To gather GPU performance and occupancy statistics, we used the *nvprof* tool and measured the application runtime on a real Kepler K20.

6.4.3 Outcome Classification

Before we dive deeper into the error detection capabilities of each redundancy scheme, we will describe the error outcomes that are plausible when a fault is injected. As mentioned earlier in this chapter, the duplicate instruction reads the same source operands as the original instruction, but writes to a new destination register. The destination operands of the instructions are verified (with their clones) via additional verification instructions. If the verification fails, a mismatch warning is issued to the user. However, we allow the program to continue execution, even if a mismatch is detected. This enables us to evaluate the accuracy of the detection scheme, and if we were being too conservative.

When a fault is injected in the program, one of the outcomes described in Table 6.1 is possible. When a fault does not manifest into an error upon program completion, and no mismatch is detected during its execution by ArmorAll, we categorize it as *Masked:undetected*. This is an ideal

case that leverages the inherent resiliency of the application. However, since our redundancy schemes duplicate the instructions and verify their output with the clones immediately, it does not provide much opportunity for masking. Sometimes a fault occurring in a duplication-eligible instruction could have been masked if it was allowed to propagate, but instead a mismatch was detected early on due to the conservative nature of ArmorAll. Such a category is called *Masked:detected*.

When a faulty bit is read by the program and causes a wrong program output (i.e., a visible error), an SDC is said to have occurred. If the propagation of the fault is detected by ArmorAll before it corrupts the output of the program, it is considered an *SDC:detected*. The higher the number of SDC:detected outcomes means better error protection. If the fault corrupts the output value and the fault was not detected during its propagation, it is referred to as *SDC:undetected*.

Finally, DUEs occur when a system is able to detect an error, but is unable to recover from the error. Our goal is to detect errors that can cause these DUEs *before* they occur, so that they can be prevented. We refer to the errors that were detected by ArmorAll before the DUE occurred as *DUE:detected*. If ArmorAll was unable to detect them and the system eventually crashed or completed execution with any other undesirable behavior, it is called *DUE:undetected*.

While our focus in this work is Error Detection, our proposed redundancy schemes can be augmented with any appropriate Error Recovery mechanism such as checkpointing, to allow applications to rollback to the checkpoint and restart execution when a mismatch is detected [111]. Error correction techniques, such as instruction triplication, can also be used when a mismatch is detected. Error recovery or correction are interesting avenues for future research, but are currently beyond the scope of this work.

6.5 Results

Next, we will evaluate the detection coverage and the associated overhead of the three redundancy schemes which constitute ArmorAll. We will also propose a set of heuristics that will allow the compiler to choose the best redundancy scheme out of the three.

6.5.1 Instruction Duplication Overhead

Static: We first measure the instruction coverage as a fraction of the number of static instructions in the program that are assumed to be protected via instruction duplication. As shown in Figure 6.1, the range of static instructions duplicated for Address Armor varies from 8-54%,

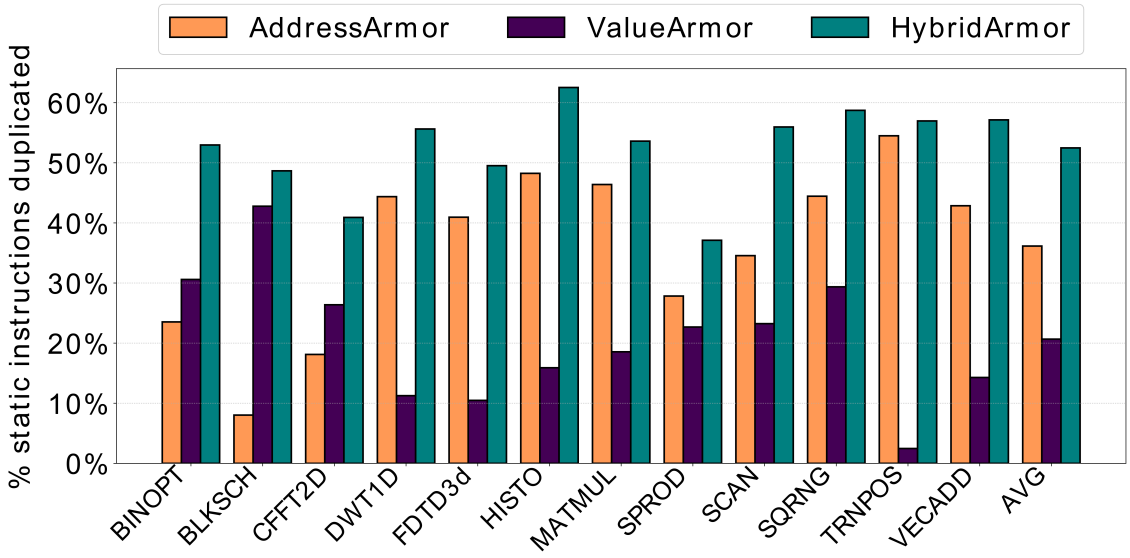


Figure 6.1: Percentage of static instructions duplicated for each Armor.

Armor	Min	Max	Average
Address Armor	8.2%	53.3%	26.2%
Value Armor	7.4%	27.6%	19.2%
Hybrid Armor	9.2%	54.8%	27.5%

Table 6.2: Min, Max, and Average increases in the program binary size for ArmorAll redundancy schemes, as compared to the baseline with no duplication. The program binary includes the non-duplicated host code.

with an average coverage of 35.9%. Value Armor duplicates between 2%-42% the instructions, with an average of 20.4% of the static instructions. Since Hybrid Armor protects both values and addresses of load and store instructions, the duplication rate is higher than with Address Armor and Value Armor. It's coverage ranges between 38%-62%, with an average of 52.2% of the static instructions duplicated. For each instruction duplicated, there is an additional verification instruction to compare the outputs of the original instruction and the duplicated instruction. The average code bloat (increase in the size of the program binary, including the unduplicated host code) for Address Armor, Value Armor, and Hybrid Armor is 26.2%, 19.2%, and 27.5%, respectively. The baseline used for comparison does not perform any instruction duplication. For all Armor types, the minimum increase in binary size was observed in VECADD, whereas the maximum increase was observed in TRNPOS applications, as shown in Table 6.2.

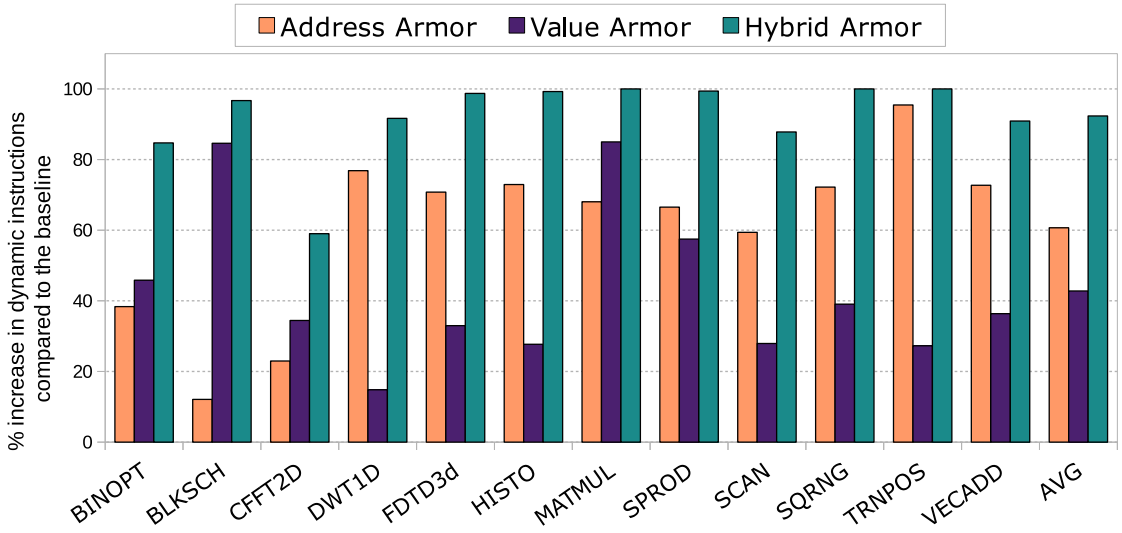


Figure 6.2: Percentage increase in dynamic instruction count compared to the baseline.

Dynamic: Next, we evaluate how the static coverage translates into dynamic instruction overhead. Figure 6.2 shows the percent increase in the dynamic instructions executed, as compared to the baseline. The dynamic instructions executed in the Armors include the duplicated instructions, verification instructions, as well the notification instructions. The baseline does not provide any duplication. The percent increase in the dynamic instruction count varies across applications, with an average increase of 60.7%, 42.8% and 90.1% for Address Armor, Value Armor, and Hybrid Armor, respectively. As expected, Hybrid Armor protects both addresses and values, hence it adds more duplication and verification to the kernel. The number of verification instructions can be reduced by delaying the verification after a few instructions by using a signature register [19]. However, there is a risk that an application may crash/hang before the delayed check is encountered. Therefore, Value Armor might benefit more from delaying verification than Address Armor or Hybrid Armor as the risk of DUEs is assumed to be lower in case of applications protected by Value Armor.

6.5.2 Error Detection Capability of ArmorAll

As mentioned earlier, we allow an application to continue execution even if a mismatch is detected. This allows us to evaluate the accuracy of the detection, and know whether we were too conservative. When a fault is injected, one of the outcomes listed in Table 6.1 is possible. We discuss the detection capability of each Armor for DUEs, SDCs, and Masked, in this order.

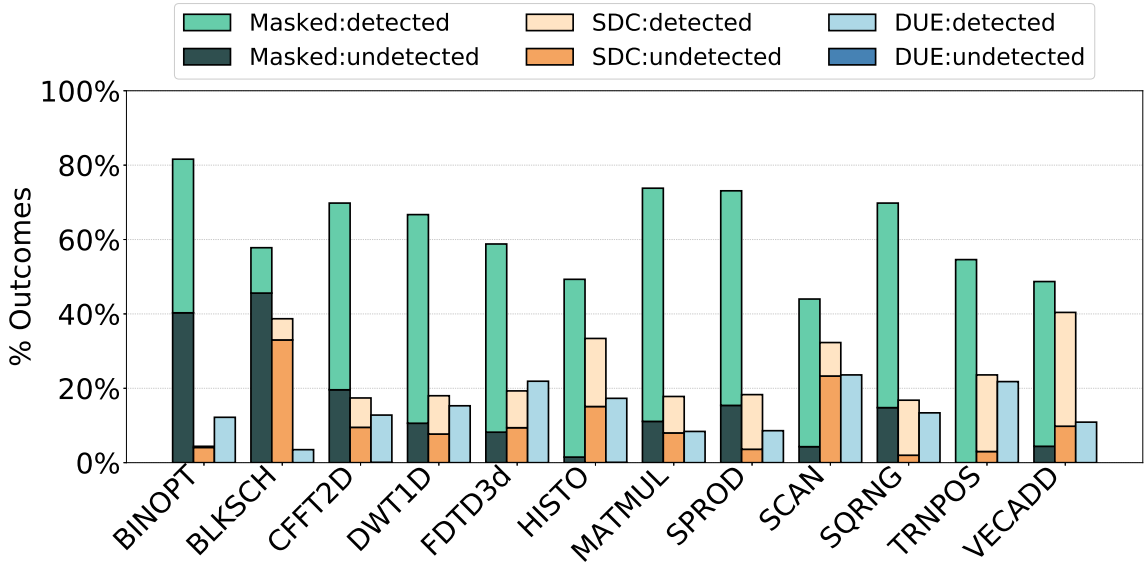


Figure 6.3: Fault injection results showing error detection capability of Address Armor. AA is able to detect all DUEs and some SDCs.

6.5.2.1 Address Armor

In this section, we will focus on the error detection capability of Address Armor. The goal of Address Armor is to protect the addresses used by load and store instructions. Typically, illegal memory accesses can cause DUEs, which can be prevented if the instructions computing memory addresses are protected. As shown in Figure 6.3, Address Armor is highly efficient and is able to detect 100% of the DUEs (right-most bars), despite duplicating only 35.9% (on average) of the static instructions (see Figure 6.1).

Interestingly, one of the unexpected behaviors in Address Armor is that it is also able to detect SDCs (middle bars), along with DUEs. This is because Address Armor was able to prevent threads from accessing incorrect (but not illegal) memory addresses and reading/writing wrong values, which can eventually lead to an SDC. Therefore, Address Armor is able to protect more than what it was originally designed for.

Due to the conservative nature of ArmorAll, we are able to detect some of the outcomes which would have been masked if the error was allowed to propagate. However, Address Armor is still able to provide some undetected error masking in all applications. We will discuss the masking offered by ArmorAll in further detail in Section 6.6.

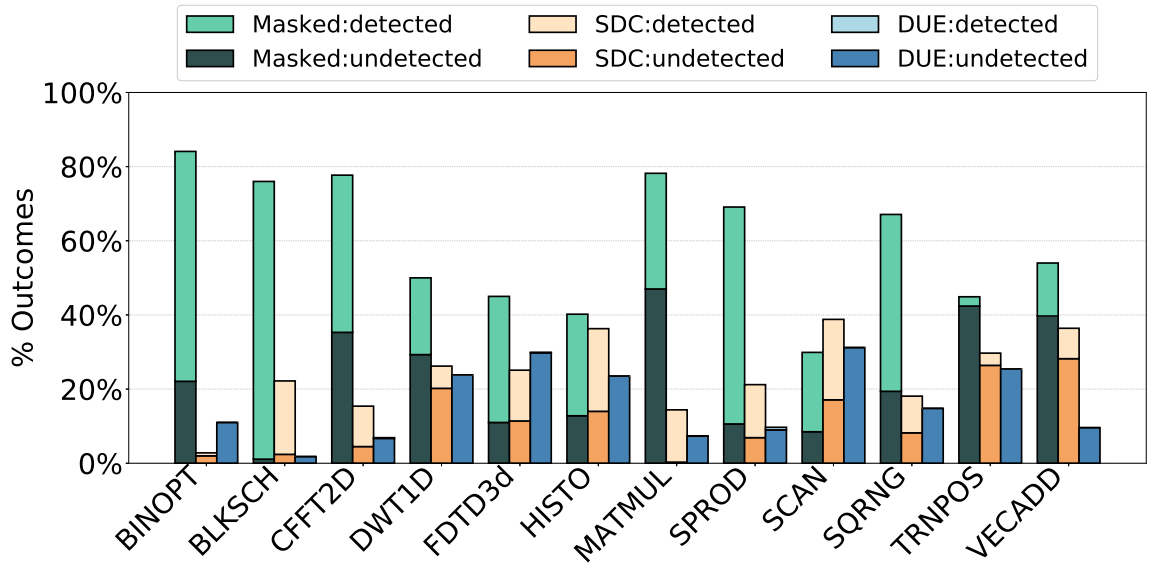


Figure 6.4: Fault injection results demonstrarting the error detection capability of Value Armor. VA can detect some SDCs, but no DUEs.

6.5.2.2 Value Armor

Figure 6.4 shows the error detection capability of Value Armor. The goal of Value Armor is to protect the values used by load and store instructions.

As shown in the Figure 6.4, Value Armor is unable to detect any DUEs, which is an expected behavior because protecting against DUEs is not the primary goal of Value Armor. Protecting output values can help detect SDCs, as seen in many applications. In case of BLKSCH (BlackScholes) and MATMUL (Matrix Multiplication), Value Armor is able to detect more than 95% SDCs, with about 42% and 19% instruction coverage, respectively, for the two applications. Similar behavior can be seen in the case of CFFT2D (ConvolutionFFT2D), FDTD3D (Finite Difference Time Domain), HISTO (Histogram), SPROD (Scalar Product), SCAN (Scan), and SQRNG (Sobol Quasi Random Number Generator), where Value Armor can detect over 70% the SDCs, with an average instruction coverage of 26%.

For the DWT1D (Discrete Walsh Transform), TRNPOS (Matrix Transpose), and VECADD (Vector Addition) workloads, Value Armor does not perform as well. This is because these applications are mainly address-bound (computing addresses). In TRNPOS, the kernel performs a large number of matrix-based moves (loading from one location in the matrix and storing to a different location). All other instructions in this kernel compute addresses. This behavior can also be observed

in Figure 6.1 which shows that only 2% of the instructions are duplicated by Value Armor. Therefore, TRNPOS does not benefit from Value Armor.

In the case of VECADD, the dominant operation is vector addition of two vectors A and B, expressed as: $C[i] = A[i] + B[i]$, where $i = blockDim.x * blockIdx.x + threadIdx.x$. Here, i holds the offset from the base address of each vector. The effective address is computed using the LLVM *getelementptr* instruction, which returns a pointer to the value at (base address + offset). If a fault corrupts the value of i, it changes the offset. Thus, a thread could end up reading the value from another index belonging to another thread. On GPUs, all threads can access the global memory space and nothing prohibits a thread from reading another global memory location. In addition, a fault in the value of i can also cause DUEs if the effective address computed is beyond the legal memory space assigned to the application. This is because bit flips in more significant (i.e., high order) bits can cause larger deviation in the offset resulting in memory exceptions. Since Value Armor does not protect addresses, it will be unable to capture this behavior.

Lastly, Value Armor shows high Masked:undetected for applications which are address-bound, or have a mix of addresses and data values, such as MATMUL. This is because most of the instructions in these applications are not duplicated using Value Armor, hence there is more opportunity for Masking. Moreover, applications such as BLKSCH, SQRNG, and BINOPT are floating-point applications. Therefore, any slight change in their values will be detected by Value Armor, reducing the likelihood of Masking.

6.5.2.3 Hybrid Armor

Figure 6.4 shows the error detection capabilities of Value Armor. Hybrid Armor protects both values and addresses used by load and store instructions.

Similar to Address Armor, Hybrid Armor is able to detect all DUEs. In terms of SDC detection, Hybrid Armor provides the combined detection capability of Address Armor and Value Armor, and can detect over 98% of the SDCs across all applications. However, Hybrid Armor is more conservative than Address Armor and Value Armor, leaving little room for any potential masking opportunity. This can be seen in the high percentage of Masked:detected outputs across all applications. This overhead comes with high coverage requirements.

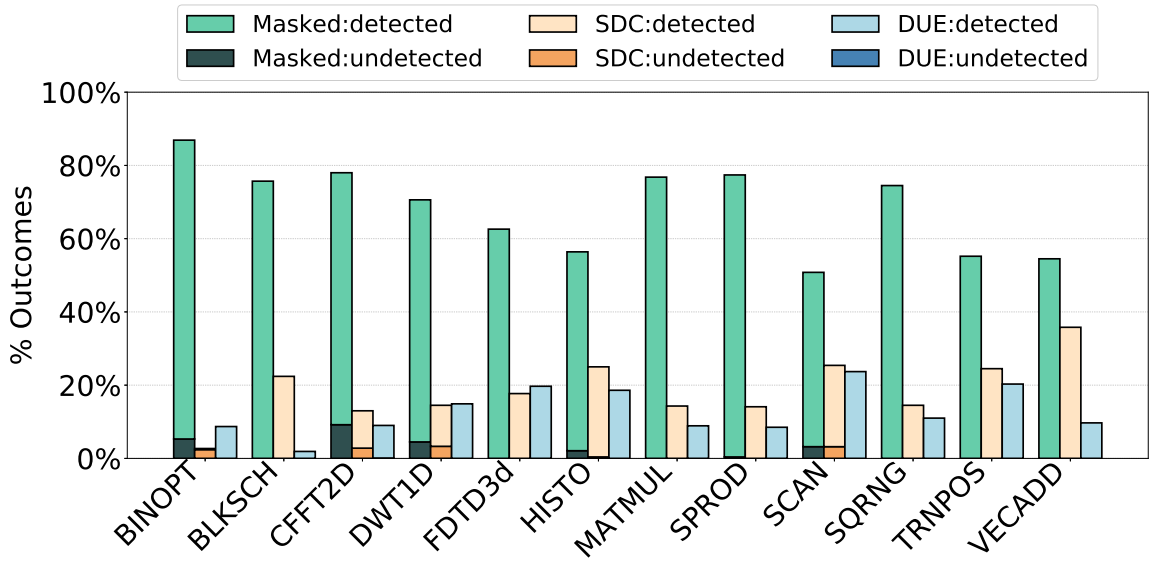


Figure 6.5: Fault injection results, demonstrating error detection capability of Hybrid Armor. HA can detect almost every error, but is also quite conservative (high percentage of Masked:detected outcomes).

6.5.3 Performance Overhead

In this section, we analyze the performance overhead of our three schemes as compared to a baseline that uses no instruction duplication. In addition to comparison with the baseline, we also quantitatively compare our results with a prior state-of-the-art software-based approach that protects all execution units using thread-level duplication (TLD) [17]. We recognize that TLD provides additional coverage to the register file and local memory, which we assume is protected by ECC. Figure 6.6 shows the execution time of applications run with ArmorAll and with TLD, relative to the baseline (without any duplication). The average increase in runtime for Address Armor, Value Armor, and Hybrid Armor versus the *uninstrumented baseline* is 1.7X, 1.39X, and 2X, respectively. Compared to TLD, the average improvement in the runtime is 42.6%, 53.2% and 32.2% for AA, VA, and HA, respectively. We discuss the sources of slowdown in both - TLD and ArmorAll.

State-of-the-art: 1. TLD creates a duplicate copy of each thread, identified as a leading thread and a trailing thread. At every store instruction, the leading thread places the value in a global buffer, and the trailing thread reads the value from the buffer and verifies it with its private copy before committing the store instruction. This process requires explicit synchronization between the leading and trailing thread to ensure that the values are written and read in the correct order. The

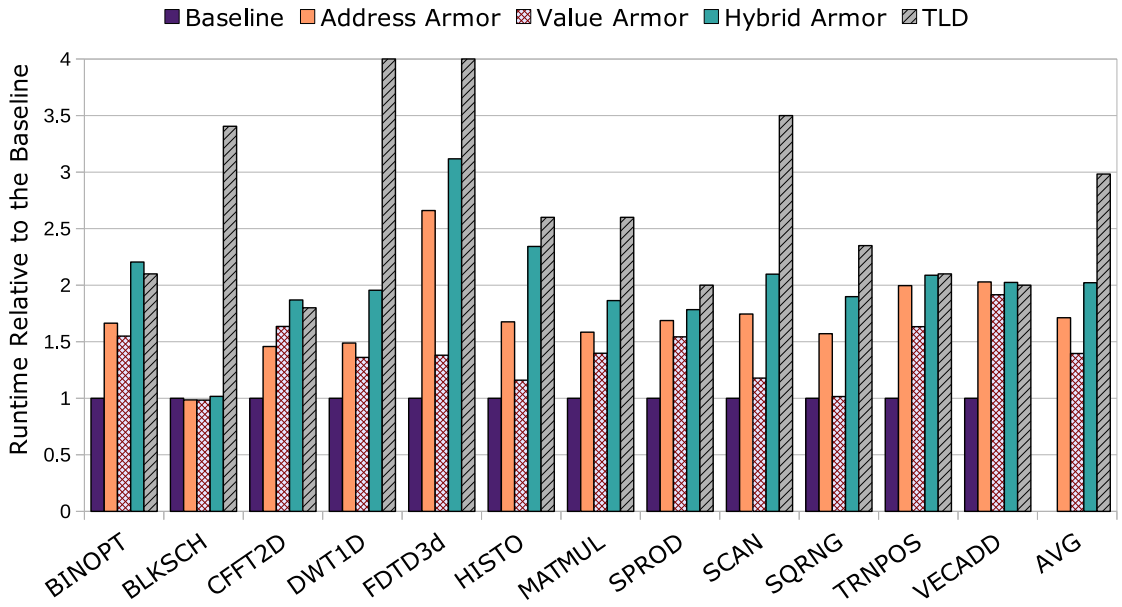


Figure 6.6: Runtime of applications equipped with ArmorAll and with TLD, relative to the baseline (no duplication). The average improvement in the runtime, as compared to TLD, is 42.6%, 53.2% and 32.2% for AA, VA, and HA, respectively.

high overhead in TLD can mainly be attributed to inter-block communication and synchronization between threads during store instructions.

2. Unlike ArmorAll, programmer intervention is needed in TLD to modify the source code to ensure spare compute resources are available for the redundant threads. Workloads can benefit from TLD if they have spare resources available (e.g., lower than 50% occupancy); however, many of the GPU applications we studied have an occupancy greater than 50%, as shown in Figure 6.7. Despite the availability of spare compute resources, the synchronization overhead with TLD can adversely impact the performance of an application.

Our approach: For ArmorAll, there are three sources of slowdown across applications. First, due to additional instructions added to the kernel (for duplication, verification and notification), there are more dynamic instructions executed per thread, which increases the runtime. Second, the addition of these new instructions can introduce new dependencies in the kernel and limit opportunities to schedule and software pipeline instruction sequences. These added dependencies can contribute significantly to the performance overhead.

Third, the duplicate instruction reads the same input operand and writes to a new destina-

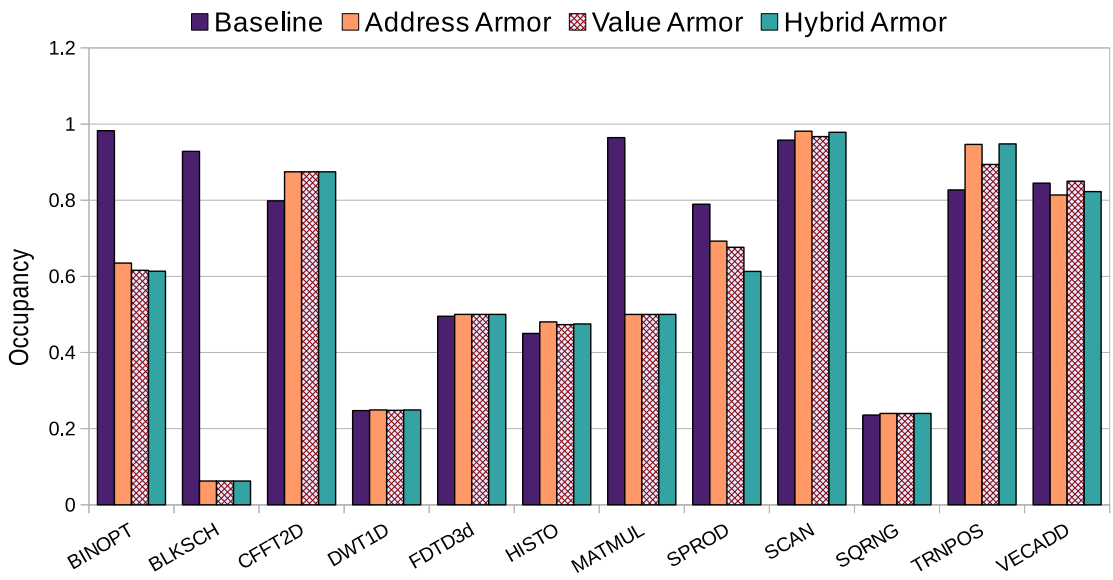


Figure 6.7: Occupancy of applications with and without ArmorAll

tion register. This may increase the overall register pressure, depending on the number of instructions duplicated, and consequently, impact the warp occupancy of the application. From our experimentation, most of the applications did not show much change in their occupancy when executed with the Armors as compared to the baseline. The only exceptions were BINOPT, BLKSCH, and MATMUL. In the case of BINOPT and MATMUL, the reduction in occupancy had an adverse impact on the execution performance. However, reduction in occupancy may not always reduce the execution performance. This can be observed in BLKSCH where there is not much change in the runtime of the application, despite a reduction in occupancy. Sometimes a reduction in occupancy can also improve the performance of an application, although we did not observe this phenomenon in our applications. Running fewer threads on the SM can reduce memory contention and improve the runtime of the application. Therefore, low occupancy does not necessarily imply poor performance.

6.5.4 Choosing the Best Armor for an application

In this section, we propose heuristics to identify the redundancy scheme that is likely to offer the best error protection to an application. We choose not use the commonly-used performance metrics, such as memory intensity or compute intensity, for characterizing reliability, as they can be misleading [112]. Our results have shown that faults in instructions that compute addresses for load/store instructions can cause illegal memory accesses, and not just load and store instructions

themselves. Therefore, solely focusing on the number of loads and stores in an application may not provide the complete picture.

Instead, we measure the *cardinality* of the final worklist generated by each Armor as a metric to choose the best Armor. As mentioned earlier, each Armor constructs a worklist, which is a set of unique instructions that appear on the data and control dependence path, starting from a root instruction. Mathematically, cardinality is defined as the number of elements present in a given set. In this context, cardinality is the number of instructions in the final worklist constructed by the Armor, after removing the duplication-*ineligible* instructions. The remaining instructions are considered highly vulnerable and must be duplicated. We use the following notation to represent cardinality:

$$|S_{AA}| = \text{Cardinality of the worklist generated by AA}$$
$$|S_{VA}| = \text{Cardinality of the worklist generated by VA}$$
$$|S_{HA}| = \text{Cardinality of the worklist generated by HA}$$

If the cardinality of Address Armor is greater than the cardinality of Value Armor, it implies that the application will be better protected if AA is chosen. The rationale behind this scheme is that the longer the total path, the more opportunities there there will for the fault to propagate to the output. Also, a larger number of instructions will have a higher probability of being targeted by transient faults. As these instructions are considered highly vulnerable, the likelihood of an application failing is higher if they are left unprotected. For example: if $|S_{AA}|$ = 80 and $|S_{VA}|$ = 10, it would be more logical to guard the application using Address Armor rather than Value Armor, as 80 instructions are more likely to cause an error than 10 instructions. Similarly, if $|S_{VA}| \ll |S_{AA}|$, it is better to invoke Value Armor. However, if the $|S_{AA}| \approx |S_{VA}|$, then Hybrid Armor is expected to provide better overall coverage.

We summarize the selection criteria as follows:

$$|S_{AA}| \gg |S_{VA}| \rightarrow \text{Address Armor}$$
$$|S_{AA}| \ll |S_{VA}| \rightarrow \text{Value Armor}$$
$$|S_{AA}| \approx |S_{VA}| \rightarrow \text{Hybrid Armor}$$

The cardinality of each application is reflected in Figure 6.1 as a fraction of the total number of instructions in the application. In the case of BLKSCH, the cardinality of VA is clearly much higher than for AA. The error detection results also reflect that AA is not able to protect against

most of the SDCs, whereas VA can successfully overcome 90% of the errors. Therefore, choosing VA will provide better error protection in the case of BLKSCH. For applications DWT1D, FDT3D, HISTO, MATMUL, SQRNG, TRNSPOS, and VECADD, the cardinality for Address Armor is 2X higher than the cardinality of Value Armor. This can be seen in Figure 6.3, showing Address Armor can protect against more than 80% of the DUEs and SDCs in these applications, on average.

For BINOPT, CFFT2D, SPROD, and SCAN, the AA to VA cardinality ratio is close to 1. Therefore, Hybrid Armor is expected to provide better coverage for these applications. However, in the case of SPROD, Hybrid Armor can be too conservative as Address Armor can protect against most of the errors. Overall, our heuristic has been able to choose the Armor correctly for 11 out of 12 of the applications (91.7% accuracy), with SPROD being the only exception. We observe an average improvement of 64.5% in runtime versus TLD when employing the best redundancy scheme chosen using the proposed selection criteria.

In our approach, the cardinality is measured using the number of static instructions in the worklist. It does not take into account the execution frequency of these instructions. However, for the applications we studied, the trend of static and dynamic instructions has remained the same for most applications, as seen in Figures 6.1 and 6.2. Using the dynamic instruction count is an alternative, however, it will not improve the overall prediction accuracy for the current set of applications. Moreover, making a decision at compile-time further saves the overhead of re-executing an application on the GPU. By calculating the cardinality of AA, VA and HA statically, the compiler can choose the most appropriate Armor without relying on dynamic profiling information. The selection criteria can be extended in the future to include the weights of the instructions in the worklist as more applications are added to the suite.

6.6 Approximate Error Detection

The error detection results show that Hybrid Armor is quite conservative, as it detects most of the errors that could have potentially been masked. The drawback of being conservative is that when ArmorAll is augmented with any error recovery scheme (e.g. check-pointing), the application must roll back to a checkpoint and restart from there whenever a mismatch is detected. This can cause significant execution overhead. The more conservative an approach is, the more the application may suffer from rollback-restart overhead; and this may be unnecessary many times.

In this section, we try to relax the detection constraints in ArmorAll. One of the opportunities to relax and improve ArmorAll's detection capabilities is during execution of floating-point

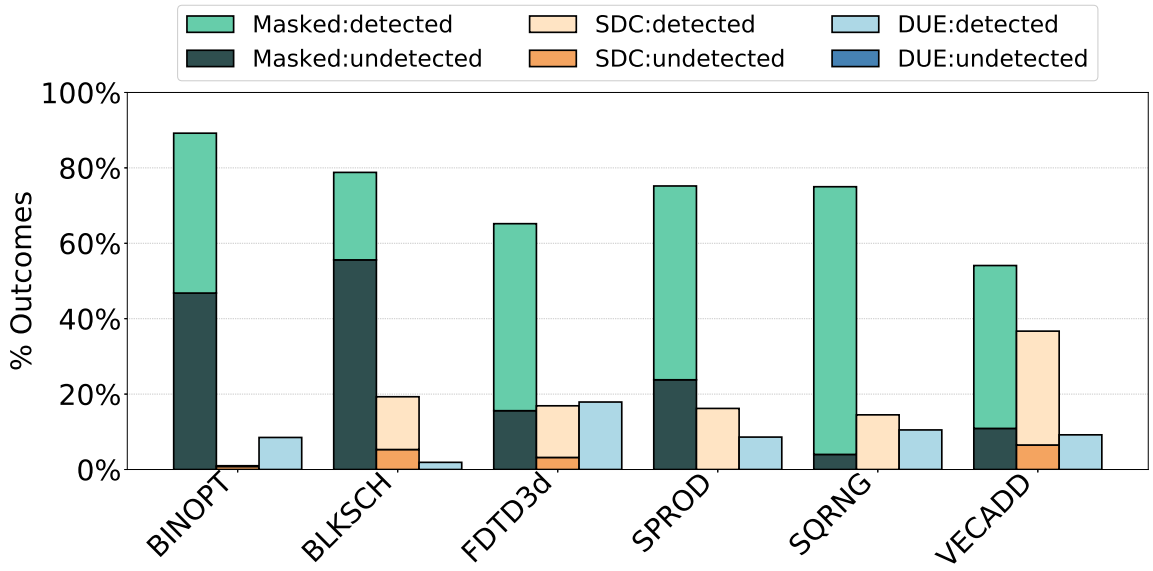


Figure 6.8: Fault injection results by using approximate error detection in Hybrid Armor. The detection precision used here is 0.1. Relaxing error detection increases the percentage of Masked:undetected outcomes in floating point applications, making HA less conservative.

applications/operations. We observed that in floating point operations, detecting a small perturbation in a value may not always impact the final program output. This prompted us to experiment with different *detection precisions*, starting with a difference of 0.1, as shown in Figure 6.8. In this case, we only report a mismatch when the difference in the output of the original and duplicated (floating point) instruction is greater than 0.1. In other words, we allow any difference less than 0.1 to propagate, and observe its impact on the output of the application, as well as the behavior of ArmorAll. We do not modify the default L1 or L2-Norm used to compare the final value(s) with the golden output in these applications. The value of 0.1 was selected experimentally and is used to test the potential of approximate detection. We also experimented with a precision of 0.001, but did not observe any significant change in the results. For the sake of brevity, we only show the results for a detection precision of 0.1. In future work, we plan to leverage Algorithmic Differentiation to help identify the best precision value to use for error detection [113].

To demonstrate the results for approximate detection, we only chose a subset of the applications, focusing on those that use floating point operations. Most applications show an increase in Masked:undetected outcomes, as intended, while BINOPT and BLKSCH show the largest increase. SQRNG primarily uses integer operations and eventually converts those values to floating point,

hence not leaving much room for approximate detection. Some applications, such as BLKSCH, FDTD3D, and VECADD also show a slight increase in the number of undetected SDCs. This side effect is because some of the deviations under 0.1, which were not detected, did not get masked and eventually propagate to the output. In practice, it may not be possible to assure apriori that we can eliminate all causes of unreliability. Therefore, the goal of fault tolerance is to reduce the system failure rate, increasing availability to an acceptable level [1].

6.7 Summary of ArmorAll

- In this chapter, we propose three novel reliability-aware compiler-based redundancy schemes that can provide adjustable levels of fault coverage to individual GPU applications.
- The redundancy schemes are designed with the goal of providing selective coverage to applications by protecting one or both of these entities – address and/or values used by load/store instructions.
- Our evaluation shows that, depending on the redundancy scheme chosen, ArmorAll can achieve an average reduction of 100% of Detected Uncorrectable Errors (DUEs) or 98% Silent Data Corruptions (SDCs) or both, while only requiring to cover 60.7%, 42.8%, or 90.1% of the dynamic instructions, respectively.
- In comparison to a state-of-the-art scheme, the performance overhead of our redundancy schemes shows an average improvement of 64.5% when using the best redundancy scheme across the studied applications.

Chapter 7

Conclusions and Future Work

7.1 Dissertation Summary

Reliability is one of the major challenges faced in exascale supercomputing and safety-critical applications today. While leveraging the inherent parallelism offered by Graphics Processing Units is crucial to accelerate the applications, it is equally important to ensure that applications can overcome data corruption caused by transient or permanent faults. It is therefore necessary to find ways to improve the reliability of execution, with limited execution overhead.

There are several challenges to overcome in order to address the above problems. Existing techniques are not able to predict the resilience of GPU applications without performing time-consuming fault injection campaigns. As a result, the same level of protection is applied to all applications regardless of their resilience. This results in conservative execution and unnecessary overhead since not every application requires the same level of protection. A thorough understanding of application behavior is required before employing expensive techniques to protect an application. In this thesis, we achieve this goal by performing a detailed characterization of the application and creating a statistical model that can predict device reliability. We also develop a set of compiler-based redundancy schemes that can adapt the level of protection provided to an application. Since hardware techniques are inflexible, adding support for reliability in the software (compiler) allows the user to turn on or turn off the optimization depending on the application's behavior and requirement.

In Chapter 4, we implemented an analysis pass in the profiling phase of SASSIFI to identify scalar and vector instructions. This study aims to understand how error propagation differs when a transient fault occurs in a scalar instruction, versus vector. We measured the scalar and vector intensities of the applications and identified linear correlations present between scalarness and the

fault outcomes. We learned that applications that have higher scalar intensity, such as atomic add, bfs, sgemm, and mri-gridding, experience a high number of SDCs and DUEs, while applications such as histo and tpacf have relatively low scalar intensity, show high masking of errors. Since scalar instructions are executed once per warp, faults in these instructions potentially have a higher impact on the application vulnerability. Therefore, applications that have higher scalar intensity should have more protection than applications with higher vector intensity.

We further extended our implementation to extract a set of workload characteristics, beyond scalar and vector, based on the hardware resources they stress during execution to accurately capture their behavior. In Chapter 5, we identified interaction between these program characteristics/features, and perform feature selection based on their contributions to program correctness. We used this reduced feature set to drive our machine learning models. For this work, we explored two models- Linear Regression and K-NN and evaluated their accuracy on randomly selected test data. We observed that Linear model predicts masking probability with 90% accuracy, whereas K-NN suffers from the curse of dimensionality. We learned that floating point intensity plays a significant role in masking while faults in integer and scalar instructions lead to higher DUEs. Even though K-NN did not provide favorable results, we anticipate that expanding the training suite will improve its accuracy.

To provide adaptable fault coverage to applications, in Chapter 6, we leveraged dependence analysis to implement three redundancy schemes - Address Armor, Value Armor, and Hybrid Armor, in LLFI-GPU and evaluated their error detection capability on applications from CUDA SDK. We analyzed the performance overheads of the redundancy schemes in ArmorAll against state-of-the-art and evaluated the instruction duplication overhead (both Static and Dynamic), increase in the program binary size, execution time, change in register pressure and occupancy. We observed that Address Armor is highly efficient and is able to detect 100% of the DUEs, despite duplicating only 35.9% of the static instructions. In addition to protecting addresses which can cause DUEs in the event of a fault, Address Armor is also able to detect SDCs. This is because Address Armor is able to prevent threads from accessing incorrect (but not illegal) memory addresses and reading/writing wrong values, which can eventually lead to an SDC. Similar to Address Armor, Hybrid Armor is able to detect all DUEs. In terms of SDC detection, Hybrid Armor provides the combined detection capability of Address Armor and Value Armor, and can detect over 98% of the SDCs across all applications. However, Hybrid Armor is more conservative than Address Armor and Value Armor, leaving little room for any potential masking opportunity. We also developed heuristics that allows the compiler to select the redundancy scheme that provides the highest fault coverage to an application

without executing it on the GPU. We measure the *cardinality* of the final worklist generated by each Armor as a metric to choose the best Armor achieving an accuracy of 91.7%.

7.2 Future Research Challenges

As part of our PRISM work, we mainly focused on program characteristics that ensure our statistical models are independent of the underlying micro-architecture. The current feature set can be expanded to include algorithm-based features as well. This is because the algorithm also has an impact on how the error propagates to the output. In addition to Linear Regression and K-NN regression models, other models such as Tree-based Regression can also be explored. Once there are sufficient workloads supported on SASSIFI to collect the raw data, neural networks can also be explored.

In the case of ArmorAll, a deferred verification technique (proposed by Gupta et al. [85]) could further improve its performance. Moreover, ArmorAll's toolset can also be integrated with checkpoint-recovery mechanisms. For floating point applications, we used the same detection precision throughout the application. This can increase the number of Masked:detected outcomes and potentially increase the rollback-recovery overhead. To avoid this, techniques such as algorithmic differentiation can be exploited to find the detection precision to be used for different instructions in a floating point application.

Bibliography

[1] S. Mukherjee, *Architecture design for soft errors*. Morgan Kaufmann, 2011.

[2] S. Shah and J. Hasler, "Soc fpaa hardware implementation of a vmm+ wta embedded learning classifier," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 8, no. 1, pp. 28–37, 2017.

[3] L. G. Szafaryn, "Understanding and optimizing heterogeneous soft-error protection," Ph.D. dissertation, University of Virginia, 2015.

[4] 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.

[5] J. Dongarra, P. Beckman, T. Moore, P. Aerts, G. Aloisio, J.-C. Andre, D. Barkai, J.-Y. Berthou, T. Boku, B. Braunschweig, F. Cappello, B. Chapman, X. Chi, A. Choudhary, S. Dosanjh, T. Dunning, S. Fiore, A. Geist, B. Gropp, R. Harrison, M. Hereld, M. Heroux, A. Hoisie, K. Hotta, Z. Jin, Y. Ishikawa, F. Johnson, S. Kale, R. Kenway, D. Keyes, B. Kramer, J. Labarta, A. Lichnewsky, T. Lippert, B. Lucas, B. Maccabe, S. Matsuoka, P. Messina, P. Michielse, B. Mohr, M. S. Mueller, W. E. Nagel, H. Nakashima, M. E. Papka, D. Reed, M. Sato, E. Seidel, J. Shalf, D. Skinner, M. Snir, T. Sterling, R. Stevens, F. Streitz, B. Sugar, S. Sumimoto, W. Tang, J. Taylor, R. Thakur, A. Trefethen, M. Valero, A. Van Der Steen, J. Vetter, P. Williams, R. Wisniewski, and K. Yelick, "The international exascale software project roadmap," *Int. J. High Perform. Comput. Appl.*, vol. 25, no. 1, pp. 3–60, Feb. 2011. [Online]. Available: http://dx.doi.org/10.1177/1094342010391989

[6] TOP500, "TOP500.ORG, https://www.top500.org/list/2018/06/."

[7] E. Alerstam, T. Svensson, and S. Andersson-Engels, "Parallel computing with graphics processing units for high-speed monte carlo simulation of photon migration," *Journal of biomedical optics*, vol. 13, p. 060504, 2008.

[8] M. C. Schatz, C. Trapnell, A. L. Delcher, and A. Varshney, "High-throughput sequence alignment using graphics processing units," *BMC Bioinformatics*, vol. 8, no. 1, pp. 1–10, 2007. [Online]. Available: http://dx.doi.org/10.1186/1471-2105-8-474

[9] J. E. Stone, J. C. Philips, P. L. Freddolino, D. J. Hardy, L. G. Trabuco, and K. Schulten, "Accelerating molecular modeling applications with graphics processors," *Journal of Computational Chemistry*, vol. 28, pp. 2618–2640, 2007.

[10] S. Grauer-Gray, W. Killian, R. Searles, and J. Cavazos, "Accelerating financial applications on the gpu," in *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units*, ser. GPGPU-6. New York, NY, USA: ACM, 2013, pp. 127–136. [Online]. Available: http://doi.acm.org/10.1145/2458523.2458536

[11] NVIDIA, "Tegra K1 techinical reference manual."

[12] J. Kim, H. Kim, K. Lakshmanan, and R. Rajkumar, "Parallel scheduling for cyber-physical systems: Analysis and case study on a self-driving car," in *Cyber-Physical Systems (ICCPS), ACM/IEEE International Conference on*, 2013.

[13] 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

[14] R. E. Lyons and W. Vanderkulk, "The use of triple-modular redundancy to improve computer reliability," *IBM Journal of Research and Development*, vol. 6, no. 2, pp. 200–209, April 1962.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] A. Mahmoud, S. Hari, M. Sullivan, T. Tsai, and S. Keckler, "Optimizing software-directed instruction replication for gpu error detection," in *International Conference for High-Performance Computing, Networking, Storage and Analysis (SC)*, Nov 2018.

[20] H. Jeon and M. Annavaram, "Warped-dmr: Light-weight error detection for gpgpu," in *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, Dec 2012, pp. 37–47.

[21] J. Tan and X. Fu, "Rise: Improving the streaming processors reliability against soft errors in gpgpus," in *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '12. New York, NY, USA: ACM, 2012, pp. 191–200. [Online]. Available: http://doi.acm.org/10.1145/2370816.2370846

[22] C. Kalra, "Design and evaluation of register allocation on gpus," Master's thesis, Northeastern University Boston, 2015.

[23] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. I. August, "Swift: software implemented fault tolerance," in *International Symposium on Code Generation and Optimization*, March 2005, pp. 243–254.

[24] S. Feng, S. Gupta, A. Ansari, and S. Mahlke, "Shoestring: Probabilistic soft error reliability on the cheap," in *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XV. New York, NY, USA: ACM, 2010, pp. 385–396. [Online]. Available: http://doi.acm.org/10.1145/1736020.1736063

[25] M. Didehban and A. Shrivastava, "Nzdc: A compiler technique for near zero silent data corruption," in *Proceedings of The 53rd Annual Design Automation Conference (DAC)*, 2016.

[26] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* IEEE, 2004, pp. 75–86.

[27] 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.

[28] K. Cooper and L. Torczon, *Engineering a compiler*. Elsevier, 2011.

[29] R. Dominguez, "Dynamic translation of runtime environments for heterogeneous computing," 2013.

[30] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Efficiently computing static single assignment form and the control dependence graph," *ACM Trans. Program. Lang. Syst.*, vol. 13, no. 4, pp. 451–490, Oct. 1991. [Online]. Available: http://doi.acm.org/10.1145/115372.115320

[31] C. Lattner, J. Haberman, and P. Housel, "Llvm language reference manual. 2010."

[32] B. Deschizeaux and J.-Y. Blanc, "Imaging earths subsurface using cuda," *GPU Gems*, vol. 3, pp. 831–850, 2007.

[33] H. Hassanieh, F. Adib, D. Katabi, and P. Indyk, "Faster gps via the sparse fourier transform," in *Proceedings of the 18th annual international conference on Mobile computing and networking*. ACM, 2012, pp. 353–364.

[34] D. Rivera, D. Schaa, M. Moffie, and D. Kaeli, "Exploring novel parallelization technologies for 3-d imaging applications," in *Computer Architecture and High Performance Computing, 2007. SBAC-PAD 2007. 19th International Symposium on*. IEEE, 2007, pp. 26–33.

[35] K. O. W. Group *et al.*, "The opencl specification, version 1.1, 2010," *Document Revision*, vol. 44.

[36] C. Nvidia, "Programming guide," 2008.

[37] J. A. Stratton, S. S. Stone, and W. H. Wen-mei, "Mcuda: An efficient implementation of cuda kernels for multi-core cpus," in *Languages and Compilers for Parallel Computing*. Springer, 2008, pp. 16–30.

[38] Y. Ukidave, C. Kalra, D. Kaeli, P. Mistry, and D. Schaa, "Runtime support for adaptive spatial partitioning and inter-kernel communication on gpus," in *2014 IEEE 26th International Symposium on Computer Architecture and High Performance Computing*. IEEE, 2014, pp. 168–175.

[39] G. S. Murthy, M. Ravishankar, M. M. Baskaran, and P. Sadayappan, "Optimal loop unrolling for gpgpu programs," in *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*. IEEE, 2010, pp. 1–11.

[40] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, and W.-m. W. Hwu, "Optimization principles and application performance evaluation of a multithreaded gpu using cuda," in *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*. ACM, 2008, pp. 73–82.

[41] S. Ryoo, C. I. Rodrigues, S. S. Stone, S. S. Baghsorkhi, S.-Z. Ueng, J. A. Stratton, and W.-m. W. Hwu, "Program optimization space pruning for a multithreaded gpu," in *Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization*. ACM, 2008, pp. 195–204.

[42] NVIDIA, "NVIDIA Kepler GK110 Architecture White Paper."

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

[44] M.-C. Hsueh, T. K. Tsai, and R. K. Iyer, "Fault injection techniques and tools," *Computer*, vol. 30, no. 4, pp. 75–82, April 1997.

[45] 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.

[46] 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.

[47] 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.

[48] 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

[49] 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.

[50] 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.

[51] M. Burtscher, R. Nasre, and K. Pingali, "A quantitative study of irregular programs on gpus," in *2012 IEEE International Symposium on Workload Characterization (IISWC)*, Nov 2012, pp. 141–151.

[52] S. Che, J. W. Sheaffer, M. Boyer, L. G. Szafaryn, Liang Wang, and K. Skadron, "A characterization of the rodinia benchmark suite with comparison to contemporary cmp workloads," in *IEEE International Symposium on Workload Characterization (IISWC'10)*, Dec 2010, pp. 1–11.

[53] S. Seo, G. Jo, and J. Lee, "Performance characterization of the nas parallel benchmarks in opencl," in *2011 IEEE International Symposium on Workload Characterization (IISWC)*, Nov 2011, pp. 137–148.

[54] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, "Automatically characterizing large scale program behavior," in *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS X. New York, NY, USA: ACM, 2002, pp. 45–57. [Online]. Available: http://doi.acm.org/10.1145/605397.605403

[55] F. G. Previlon, C. Kalra, D. Tiwari, and D. R. Kaeli, "Characterizing and exploiting soft error vulnerability phase behavior in gpu applications," in *Under Review*, 2019.

[56] Y. Jiao, H. Lin, P. Balaji, and W. Feng, "Power and performance characterization of computational kernels on the gpu," in *Proceedings of the 2010 IEEE/ACM Int'L Conference on Green Computing and Communications & Int'L Conference on Cyber, Physical and Social Computing*, ser. GREENCOM-CPSCOM '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 221–228. [Online]. Available: http://dx.doi.org/10.1109/GreenCom-CPSCom.2010.143

[57] Y. Abe, H. Sasaki, M. Peres, K. Inoue, K. Murakami, and S. Kato, "Power and performance analysis of gpu-accelerated systems," in *Presented as part of the 2012 Workshop on Power-Aware Computing and Systems*. USENIX, 2012.

[58] S. Hong and H. Kim, "An integrated gpu power and performance model," in *ACM SIGARCH Computer Architecture News*, vol. 38, no. 3. ACM, 2010, pp. 280–289.

[59] G. Wu, J. L. Greathouse, A. Lyashevsky, N. Jayasena, and D. Chiou, "Gpgpu performance and power estimation using machine learning," in *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, Feb 2015, pp. 564–576.

[60] 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.

[61] 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.

[62] 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.

[63] 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.

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

[65] 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.

[66] 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.

[67] 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.

[68] 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

[69] N. Farazmand, R. Ubal, and D. Kaeli, "Statistical fault injection-based analysis of a GPU architecture," in *Workshop on Silicon Errors in Logic - System Effects (SELSE)*, 2012.

[70] 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.

[71] "Enabling on-the-fly manipulations with llvm ir code of cuda sources," https://github.com/apc-llc/nvcc-llvm-ir.

[72] G. Li, K. Pattabiraman, S. K. S. Hari, M. Sullivan, and T. Tsai, "Modeling soft-error propagation in programs," in *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN). IEEE*, 2018.

[73] F. G. Previlon, C. Kalra, D. R. Kaeli, and P. Rech, "Evaluating the impact of execution parameters on program vulnerability in gpu applications," in *Design, Automation and Test in Europe, DATE*, April 2018, pp. 1–6.

[74] ——, "A comprehensive evaluation of the effects of input data on the resilience of gpu applications," in *The 32nd IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*, October 2019, pp. 1–6.

[75] F. G. Previlon, C. Kalra, D. Tiwari, and D. R. Kaeli, "Pcfi: Program counter guided fault injection for accelerating gpu reliability assessment," in *2019 Design, Automation Test in Europe Conference Exhibition (DATE)*, March 2019, pp. 308–311.

[76] 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.

[77] 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.

[78] 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.

[79] 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.

[80] 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

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

[82] 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.

[83] 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

[84] 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.

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

[86] S. Rapps and E. J. Weyuker, "Data flow analysis techniques for test data selection," in *Proceedings of the 6th International Conference on Software Engineering*, ser. ICSE '82. Los Alamitos, CA, USA: IEEE Computer Society Press, 1982, pp. 272–278. [Online]. Available: http://dl.acm.org/citation.cfm?id=800254.807769

[87] W. Bartlett and L. Spainhower, "Commercial fault tolerance: a tale of two systems," *IEEE Transactions on Dependable and Secure Computing*, vol. 1, no. 1, pp. 87–96, Jan 2004.

[88] 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.

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

[90] 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.

[91] 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.

[92] 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

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

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

[95] 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.

[96] 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.

[97] NVIDIA, "NVIDIA, CUDA SDK, V6.0."

[98] 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

[99] 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

[100] 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

[101] 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

[102] 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

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

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

[105] 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.

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

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

[108] C. Kalra, F. Previlon, N. Rubin, and D. Kaeli, "Armorall: Compiler-based resilience targeting gpu applications," in *Under Review*, 2019.

[109] N. J. Wang and S. J. Patel, "Restore: symptom based soft error detection in microprocessors," in *2005 International Conference on Dependable Systems and Networks (DSN'05)*, June 2005, pp. 30–39.

[110] C. Kalra, F. Previlon, X. Li, N. Rubin, and D. Kaeli, "Prism: Predicting resilience of gpu applications using statistical methods," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, ser. SC '18. Piscataway, NJ, USA: IEEE Press, 2018, pp. 69:1–69:14. [Online]. Available: http://dl.acm.org/citation.cfm?id=3291656.3291748

[111] R. Koo and S. Toueg, "Checkpointing and rollback-recovery for distributed systems," *IEEE Transactions on Software Engineering*, vol. SE-13, no. 1, pp. 23–31, Jan 1987.

[112] A. Kerr, G. Diamos, and S. Yalamanchili, "A characterization and analysis of ptx kernels," in *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC)*, ser. IISWC '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 3–12. [Online]. Available: https://doi.org/10.1109/IISWC.2009.5306801

[113] H. Menon, M. O. Lam, D. Osei-Kuffuor, M. Schordan, S. Lloyd, K. Mohror, and J. Hittinger, "Adapt: Algorithmic differentiation applied to floating-point precision tuning," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, ser. SC '18. Piscataway, NJ, USA: IEEE Press, 2018, pp. 48:1–48:13. [Online]. Available: http://dl.acm.org/citation.cfm?id=3291656.3291720