TOWARDS A SPACEWORTHY COTS GRAPHICS PROCESSING UNIT: HARDWARE PERFORMANCE COUNTER-BASED SYMPTOMATIC FAULT DETECTION

ANTONIO EMILIO TEIJEIRO

Master's Program in Electrical Engineering

Dedication

To my father, whose steadfast belief in a purpose-driven life is an inspiration to me. Time and experience have only increased my appreciation for all you have done for me. For my wife, to whom I hope to make up for time spent on my thesis and graduate school work with quality time.

TOWARDS A SPACEWORTHY COTS GRAPHICS PROCESSING UNIT: HARDWARE PERFORMANCE COUNTER BASED SYMPTOMATIC FAULT DETECTION

by

ANTONIO EMILIO TEIJEIRO, BSEE

THESIS

Presented to the Faculty of the Graduate School of

The University of Texas at El Paso

in Partial Fulfillment

of the Requirements

for the Degree of

MASTER OF SCIENCE

Electrical and Computer Engineering

THE UNIVERSITY OF TEXAS AT EL PASO

December 2023

Acknowledgments

Thank you to Dr. Rodrigo Romero, for sharing thesis topic ideas to focus on at a time when I felt overwhelmed with the complexity of this field. Your ideas set me on course for finishing my master's degree. In addition, thank you to Matthew Breeding and all those at Sandia National Laboratories who helped demystify radiation effects through a summer of hands-on radiation effects work.

Abstract

Ionizing radiation remains an obstacle to bringing graphics processing units (GPU) to space. Since radiation-hardened GPU chips are technically infeasible at the moment, an emphasis has been placed on the adaptation of commercial-off-the-shelf (COTS) GPUs to the space domain. At present, GPU error detection methods require redundant computation. This thesis work explores the utilization of hardware performance counters, special registers useful for monitoring internal GPU hardware events, for symptom-based, lightweight error detection. Hardware performance counters are successfully utilized for the detection of anomalous single event upsets in the L0 instruction cache, the load store unit, the arithmetic and logic unit, the fused multiply add pipeline, and the address divergence unit of a GPU. These upsets are detected using both supervised and unsupervised shallow machine learning models. Results indicate a viable alternative to redundancy-based computational methods for detection and handling of single-event upsets in a subset of components of a GPU architecture.

Table of Contents

List of Tables

List of Figures

Chapter 1: Introduction

The space environment poses significant challenges for modern electronics. The lack of convection found in a vacuum results quite easily in overheating. In addition, the dissipative effect of an atmosphere with respect to extraterrestrial ionizing radiation is no longer present. Finally, the mission critical nature of space operations yields little room for error despite the challenges. Therefore, any circuitry deployed to space must have its thermal issues addressed, be hardened towards ionizing radiation, and be mission critical qualified.

Of particular concern is modern computer technology. In modern computers, information available in a heterogeneous memory hierarchy is processed by one or more processors. Terrestrial-intentioned innovation in memory and processor technology has resulted in a trend towards decreasing feature size, higher clock rates, and higher density in processor transistors and memory cells. This in turn has resulted in more heat, a lower critical charge, and an unchanging probability of particle strike (Baumann, 2017). Together, this combination ostensibly indicates that modern computers are incompatible with space-bound operations.

In order to use computers in space, space mission processor and memory technologies make use of large feature size technology and low clock rates – the opposite of the above trend and the functional equivalent to utilizing antiquated computing technology. The current state-of-the-art single core processor for space missions, the RAD 750, is similar in processing ability to the IBM Power PC 750 from 1997. Similarly, the Juno spacecraft launched in 2011 with only 128 MB of RAM and the Aditya-L1 heliocentric satellite launched in September of 2023 with only 4 GB of solid-state storage are nowhere near the multicore chips, tens of gigabytes of RAM, and terabytes of storage currently available in computers from retail stores.

Onboard data processing is infeasible with such limitations. Instead, data acquired through cameras and sensors of spaceborne vehicles is transmitted to terrestrial data processing sites. However, even electromagnetic signals are too slow for real-time applications in space. Many missions require travel on the order of hundreds of millions of miles – Juno on its mission to Jupiter, for instance. Such missions experience lag times on the order of minutes and cannot rely on terrestrial computers to perform real-time computations.

Considering the communications delay between terrestrial computing facilities and spaceborne vehicles, onboard data processing would enable new mission capabilities. From a national security standpoint, greater processing power would enable real-time image classification for ICBM early alert systems. From an exploration standpoint, additional image processing algorithms could be added to spaceborne telescopes. From an experimental standpoint, greater processing power would enable more options for spaceborne experiments such as those that take place onboard the International Space Station (ISS). Onboard data processing is vital, and enhancing spaceborne computing technology is the key.

Despite the complexity of the problem and the involved physical barrier to conducting such research, much exciting progress on spaceborne computing technology has occurred. Current designs rely on a combination of thermal radiation out to space and thermal conduction between spacecraft components to achieve suitable equilibrium temperatures for all parts of the spacecraft, electronics included. Alternative emerging memory technologies such as CRAM, STT-MRAM, and ReRAM provide no direct mechanism for ionizing radiation to interact with stored information and therefore are inherently radiation-hardened (Marinella, 2021). Finally, mission-critical electronics qualification techniques have been established and are commonly used by organizations like the National Aeronautics and Space Administration (NASA) to qualify parts for use in space. The problem that looms large is the introduction of radiation-hardened processors with high-performance computing capabilities.

There are two ways to improve the performance of a processor: improve its sequential performance or improve its parallelism. Optimally, both approaches may be combined. However, most of the techniques to improve COTS processors also increase their sensitivity to ionizing radiation. For example, the designer can improve sequential performance by increasing the clock rate, but this increases the likelihood of upsets due to the increased frequency of rising edges on the clock signal (Baumann, 2017). The designer can improve parallelism by introducing multiple cores onto a single chip, but the size of current state-of-the-art radiation-hardened process nodes – such Skywater Lab's 90 nm process node – precludes the inclusion of many processors onto a reasonable chip footprint. Thus, at present no COTS processor enhancement techniques can be directly transferred to radiation hardened chip design processes.

With so many obstacles to radiation hardened-by-design multiprocessors, adapting state-of-the-art COTS multiprocessors has become a NASA and Department of Defense (DoD) priority. These attempts largely feature some combination of the following techniques to prevent hard faults (permanent circuit damage) and resolve soft faults (repairable faults): attempt to harden only the most critical parts of a processor to prevent hard faults there, triplicate work and use a majority voting scheme to resolve soft faults, and detect soft faults early in execution so that all or part of the affected operation can be redone. An analysis of the successes and failures of each technique have led to the search for a new technique for detecting soft faults, which is explored in this thesis: leverage the presence of GPU hardware performance counters – special GPU registers capable of tracking the occurrences of various hardware events – to detect erroneous GPU calculations without interfering with execution of GPU kernels. The research question guiding the work presented here is whether GPU hardware performance counter metrics are reliably and detectably affected by the presence of hardware faults induced by single-event upsets (SEU) and whether the information they produce suffices as the basis to construct SEU symptomatic fault detectors.

1.1 Related Work

Iturbe et al. created the APEX-SoC platform to support next-generation space instrumentation using a Zedboard mini-ITX board with a Zynq 7Z100 SoC chip (Iturbe et al., 2015). Only the single event mitigator controller (a Xilinx module with ECC to detect and correct soft errors in FPGA configuration memory), ECC logic, watchdog logic, XADC circuit, DDR controller, DDR interfaces, and a separate memory for holding configuration information benefitted from radiation hardening. Additionally, the APEX-SoC implements dual redundancy in the processing stage for fault detection (if the result from each identical pipeline differs, then a fault has occurred) coupled with dual majority voting performed on three identical copies of input data (triple redundancy) to the processing stage and triple majority voting (triple redundancy) performed on the redundant outputs of the processing stage. The final product demonstrated not only continued functionality under irradiation, but also an improvement in signal-to-noise ratio in their CIRIS photodetector. This result demonstrates that selectively hardening only the most critical components, coupled with computational redundancy, are complete functional solutions to high radiation environments.

As with the Apex-SoC, the detection and correction of faults is in general currently handled through redundancy. A calculation is repeated, and if the result differs then an error must have occurred somewhere. Repeat it three times, and the calculation that occurred twice is the correct one. Sabena et. al. showed that time-based redundancy (repeating a calculation to verify results) and thread-based redundancy (repeating the same calculation concurrently across multiple threads) can be utilized on a GPU to detect up to 87.50% of faults (Sabena et al., 2013). Other examples include the work of So et al. to introduce their FISHER scheme (triplication of threads plus mutual thread error checking instead of majority voting) to CPUs, and the work of Oh et. al to introduce instruction-level duplication to CPUs (So et al., 2019; Oh et al., 2002). Given that processing power in space is limited, the use of redundancy for soft error detection and correction is suboptimal.

Santos et al. showed that, although functional, redundancy in computation is unnecessarily wasteful of resources (Santos et al., 2019). Instead, they argue for the implementation of soft fault detection and correction schemes not reliant on redundancy. Algorithmic detection of faults in matrix multiplication led to not just fault detection, but an 87% fault correction rate in matrix multiplication. Similarly, max pooling layers inserted into convolutional neural networks were able to detect 97% of faults. These results suggest that the entirety of a multiprocessor's hardware should be devoted to useful work, as in terrestrial applications, but with the added use of detection tools.

The techniques applied by Santos et al. are helpful. However, adding extra layers and tailored algorithms to a package requires refactoring. Existing GPU-targeted software would need to be ported for space applications. In addition, what about GPU applications that do not utilize matrix multiplications or neural networks? The ideal method of fault detection would be generalizable to all programs, require very little reengineering of a program, and remain low overhead for the host system. Guerrero-Balaguera et al. recently introduced such a method: fault detection using onboard GPU hardware performance counters (Guerrero-Balaguera et al., 2021).

Guerrero-Balaguera et al. investigated the use of two hardware performance counters on the experimental FlexGripPlus GPU to detect malfunctions in a GPU branch divergence unit and warp scheduler. The branch divergence unit of the FlexGripPlus GPU handles divergent if-statements within a warp, which is a group of 32 threads discussed further in Chapter 2, Theoretical Foundations. The Warp scheduler schedules warps onto streaming multiprocessor (SM) subpartitions, also discussed in Chapter 2. Despite the very small number of counters used and the experimental nature of the GPU, a large difference in counter statistics was determined when malfunctions were introduced to the underlying hardware.

This emerging technique can help enable COTS-level of performance of GPUs in space by rendering computational redundancy unnecessary. Furthermore, the few API calls necessary to enable hardware performance counter tracking do not hinder the host system. This thesis aims to apply Guerrero-Balaguera's approach – both in GPU hardware and hardware performance counters investigated – to the NVIDIA RTX 3090, a modern COTS GPU.

Chapter 2. Theoretical Foundations

For the simulation and computation-centric experimentation performed to complete this thesis work, foundational knowledge in GPU microarchitecture, compilation for heterogeneous architectures, and machine learning are of central importance to the analysis of experimental results. Skills such as CUDA, assembly, and python programming in conjunction with open-source software modification and compilation are essential for implementing or building upon the presented work. The following is a basic introduction to such foundational knowledge.

2.1 GPU Microarchitecture

Using as a reference a typical desktop multiprocessor, a GPU microarchitecture is introduced by comparison. In particular, the GA102 GPU from an NVIDIA GeForce RTX 3090 graphics card – the configuration used in this work – is explored.

Like a desktop multiprocessor, a GPU chip contains multiple individual processors. However, these processors are organized very differently in a GPU. A desktop multiprocessor's cores are capable of executing independent processes or tasks; at least *n* independent tasks can be executed concurrently on a chip with *n* cores. In comparison, grids of threads deployed for execution on GPU cores, called streaming processors (SPs), are organized into blocks called Continuous Thread Arrays (CTAs). CTAs are composed of groups of 32 threads called *warps* which execute in single-instruction, multiple-data (SIMD) mode, i.e., groups of streaming processors execute the same instruction in lockstep while sharing hardware resources. Warps are scheduled onto *Stream Multiprocessor subpartitions* within two *Stream Multiprocessors* (SMs) per *Thread Processing Cluster* (TPC). In turn, groups of eight TPCs are organized into *Graphics Processing Clusters* (GPCs). Each level contains its own scheduler and supporting circuitry. Figure 2.1 depicts this graphically.

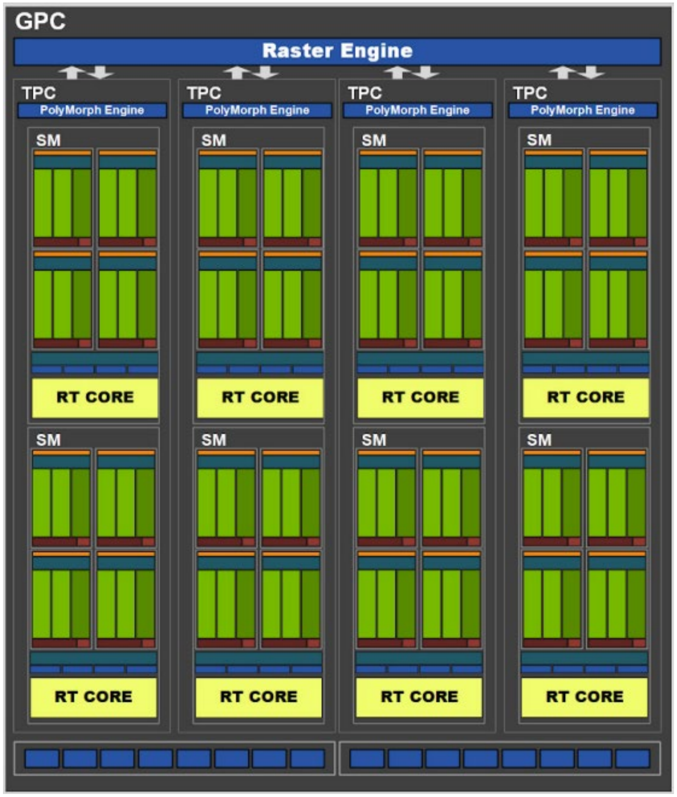


Figure 2.1 Graphics Processing Cluster (GPC) from an NVIDIA GA102 GPU with 84 SMs.

Also like a desktop multiprocessor, a GPU is based on the von Neumann model. Under the original von Neuman model, a processor containing a control unit (CU) and an arithmetic and logic unit (ALU) uses a storage component to fetch instructions to execute and data to process and, subsequently, store the result. However, whereas desktop multiprocessor systems utilize nonvolatile secondary memory for persistent storage, GPUs have no such permanent data storage. Instead, only instructions to be executed on the GPU and associated data are offloaded from the host processor's main memory to the GPU global memory. Then, results are read back to the host main memory after computation. Thus, techniques such as virtual memory are not available directly using the GPU global memory.

An SM is depicted in greater detail in figure 2.2. Here, an SM is revealed to be composed of four SM subpartitions, which warps are scheduled onto. Each SM subpartition shares four load store units (LSUs) between its threads to facilitate the loading of values from memory and the storage of computational results to memory. An extremely fast L0 instruction cache (L0 i-Cache) within each SM subpartition holds instructions that are presumably next in execution order. A special function unit (XU) is utilized for transcendental operations and datatype conversions within each SM subpartition. Depicted also are 32-bit floating point and integer pipelines. Texture (Tex) memory and Ray Tracing (RT) cores are used for graphics and are therefore ignored in this work, which focuses on computational workloads. Tensor cores, which accelerate mixed-precision computation, are also reserved for future work.

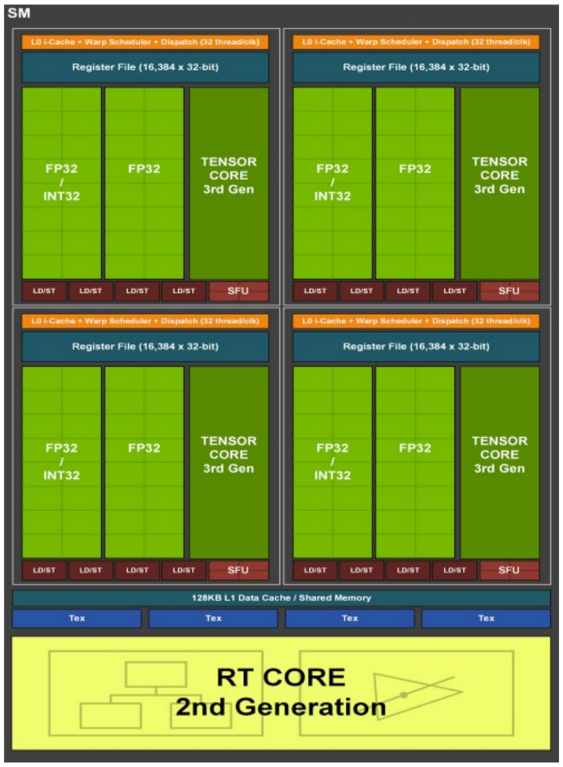

SM
L0 i-Cache + Warp Scheduler + Dispatch (32 thread/clk)
Register File (16,384 x 32-bit)
FP32 / INT32
FP32
TENSOR CORE 3rd Gen
LD/ST
SFU
128KB L1 Data Cache / Shared Memory
Tex
RT CORE 2nd Generation

Figure 2.2 GA102 SM.

Several pipelines are not depicted in figure 2.2. The fused multiply add/accumulate (FMA) pipeline expediently calculates sequences of operations in which a value is added to the result of a multiplication operation. The address divergence unit (ADU) handles branch divergence. The uniform data path (UDP) is utilized to mitigate performance bottlenecks induced by including intermittent integer arithmetic in floating point streams, e.g., updating the iterator of a loop in which floating point calculations are performed, by moving these intermittent integer operations to a separate pipeline for parallel execution, thus keeping the floating point units saturated (Jia et al., 2019). Although not depicted, the FMA and ADU play a vital role in this work.

Digging into the GPU memory hierarchy, each SM contains an L1 cache, an L2 cache is shared by all GPCs, and device DRAM global memory is shared across the GPU. Memory accesses are attempted in the above sequential order to reduce the amount of costly DRAM requests, with misses resulting in accesses to successively slower memory levels.

The L1 cache is part of the L1Tex unit, which is depicted in figure 2.3. An SM's memory input-output (MIO) interface communicates read and write requests to either the L1TEX's load store unit (LSU) or texture unit. Like *Texture* memory*, Surface* memory is utilized for graphics applications and is not considered here due to the computational nature of the benchmark suite chosen for this work. All requests initially arrive at the tag stage, whereupon L1 cache hits are handled by the data stage and misses continue on to the miss stage for a second attempt in L2 cache. Not depicted, but similar to the L1 cache, are the shared memory and constant memory cache. Shared memory is a user software-controlled data cache, whereas the constant memory cache achieves L1 cache/shared memory speeds only if all threads in a warp perform a coalesced access. Beyond the L2 cache, misses are handled in global memory.

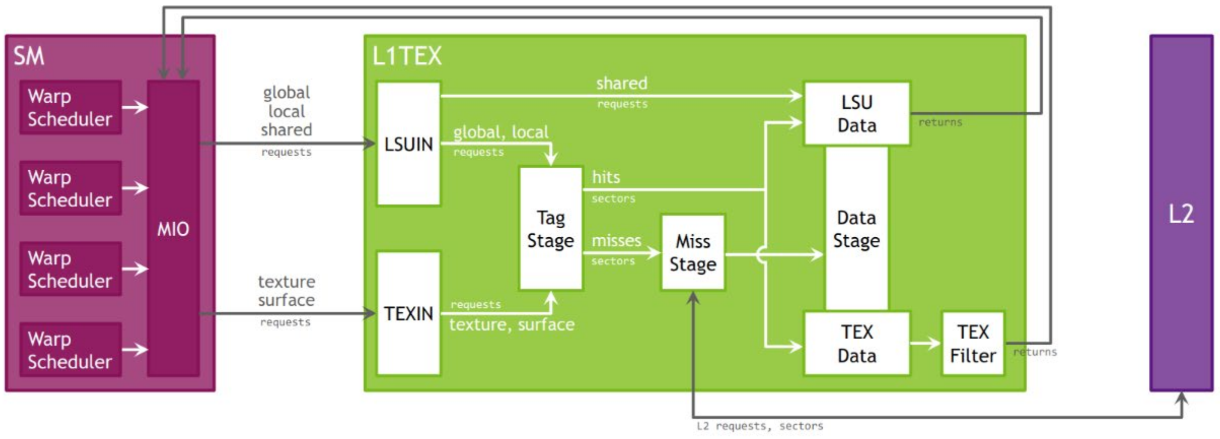


Figure 2.3 Cache Interconnects.

Within each SP are several *pipelines* which serve to enhance computational efficiency. The ALU performs logical operations and integer arithmetic, excluding multiplication and multiply-and-add.

Finally, each SP also contains several pipelines not directly related to arithmetic, but instead to control flow. The address divergence unit (ADU) handles divergent branch and jump addresses, which occur when threads within a warp perform an uncoalesced access, as well as loading constants and implementing block-level barriers, which serve to synchronize thread blocks.

2.2 Hardware Profiling

NVIDIA provides nearly 250,000 GPU *hardware metrics* for system diagnostics and application performance tuning purposes in the GA102. This is not counting the *driver metrics* intended to measure API calls, which per NVIDIA documentation, can skew results sufficiently that they become nondeterministic and are therefore not used in this thesis. Contrary to being overwhelming, when the metric hierarchy is properly understood these metrics are extremely useful for the work presented here.

NVIDIA provides several hardware events whose occurrences can be counted. The number of occurrences of each event are kept in special registers called performance counters. For example, the user can choose to configure a profiling tool to count the occurrences of L1 cache misses in a particular SM, where the cache misses are the events and the register that stores the number of cache misses is the corresponding performance counter. It is important to make this distinction as there are only 16 performance counters in the GA102, with the realistic limit of simultaneous event counting experimentally determined to be about ten events.

Events can be counted in multiple ways. The simplest way is to configure the event and counter using either the CUPTI Profiling Interface, PAPI, or NSight Compute, then reading the counter value when desired. By default, the counter value is reset upon each read, so each read produces the number of events since the last read. Optionally, the user can configure the counter to maintain the count throughout the lifetime of a kernel. The user can also configure the counter in sampling mode, in which the user collects counter values over multiple passes of the kernel. Either way, a raw count is the fundamental information returned from a profiling tool.

Users often need to aggregate event data, e.g., to calculate an average or a percentage. For this reason, CUPTI, over which the other profiling tools are built, provides *metrics* that may be applied to raw event counts for automatic tabulation. Every event has at least the sum, average, minimum, and maximum metrics – aptly categorized as *roll-ups* since they aggregate data from across all unit instances, e.g., the sum of L1 cache misses across all SMs in a GPU. Each roll-up supports 18 sub-metrics (NVIDIA metrics, 2023). Thus, in actuality there are under 2800 events to choose from for the GA102. Of these events, many are used for graphics, reducing the amount useful to this thesis further. This reduced amount is divided into 17 units. Therefore, on average, there are less than 160 events to choose from to cover the entirety of each unit.

In this thesis, PAPI was utilized to extract event count information. PAPI is built on the CUPTI Profiling Interface (not to be confused with its predecessor, CUPTI or its deprecated successor, PerfWorks API) provides low-level control over the placement of hardware performance counter reads in a program.

2.3 Fault Injection

Fault injection is the explicit introduction of faults into a system in order to analyze its behavior under fault conditions. This can be useful for driving complete functional coverage of a system, or as in the case of this thesis, for generating training data representative of the GA102 operating in a simulated space environment. By utilizing fault injection to generate training data, a machine learning model capable of detecting erroneous calculations may be developed.

Fault injection takes many forms. In the case of radiation effects, the best type of fault injection is physical fault injection by an ion beam – a beam of protons, heavy ions, and electrons. High-energy x-rays produced by braking radiation can be used to approximate gamma rays. As this option is expensive, the next best option for testing the GA102 is software-based fault injection, in which case the effects of physical fault injection are emulated through software processes. Thus, software-based fault injection is utilized in this thesis to create as realistic a testing environment as possible.

Ionizing radiation typically induces transient current spikes in operating processor chips that result in either permanent or transient upsets. In memories, the sudden carrier increases can result in bit flips. In other hardware, a sudden current can alter the behavior of an impacted node in various ways, which are often emulated to first order by bit flips, either transient or stuck-at (Tsai et al., 2021).

2.4 CUDA COMPILATION

Part of the CUDA program compilation process is the generation of a PTX file, which exposes information useful to a fault injector. PTX is a low-level parallel-thread-execution instruction set architecture and virtual machine. PTX code contains low level implementation details of the program, such as the names of the registers where the results of each operation will be stored.

The fault injector developed for this thesis intercepts the compilation of CUDA programs just after the PTX file generation stage. Then, the fault injector introduces conditional, based on the id of executing hardware, bitwise XOR and OR operations on registers to emulate bit-flip and stuck-at bit errors, respectively. For example, applying an XOR operation to the result of every add instruction for a particular SP will emulate a faulty adder. This is similar to SASSIFI behavior, albeit one level higher with PTX code instead of SASS code, which is unavailable to the user.

Figure 2.3 depicts example snippets of PTX code with and without fault injection targeting predicated instructions. Note that the code snippet on the right is considerably longer only to show every detail of the process; all the extra moves and set predicate instructions are only executed once. Thus, all subsequent occurrences of target registers would contain only one extra XOR assertion inserted before or after it. The compilation process then continues on to the generation of the final binary containing the upsets. Hence the program must be recompiled between fault injector and golden (non-injected) test runs. See section 3.2 for additional details.

```
mov.u32        %r47, %ntid.y;
mov.u32        %r48, %ctaid.y;
mul.lo.s32     %r2, %r48, %r47;
mov.u32        %r3, %tid.y;
add.s32        %r4, %r2, %r3;
setp.ge.u32    %p1, %r1, %r41;
setp.ge.u32    %p2, %r4, %r43;
or.pred        %p3, %p1, %p2;
@%p3 bra       $L__BB0_9;
```

```
mov.u32        mystoreCTAx, %ctaid.x;
mov.u32        mystoreCTAy, %ctaid.y;
mov.u32        mystorewarpID, %warpid;
setp.lt.u32    mycondition, mystoreCTAx,1;
@mycondition setp.eq.u32    mycondition, mystoreCTAy,0;
@mycondition setp.eq.u32    mycondition, mystorewarpID,0;
@mycondition xor.pred       %p3, %p3, 1;
@%p3 bra       $L__BB0_9;
```

Figure 2.3 Golden PTX code (left) and injected fault (right).

2.5 Machine Learning

Machine learning (ML) methods have gained popularity in recent years to find trends in large datasets. Where humans struggle to see patterns in large datasets, machine learning models excel. By training a supervised machine learning algorithm – specifically a support vector machine (SVM) – on hardware performance counter metrics from faulty and golden runs, a lightweight model can be developed to classify calculations as erroneous or accurate from hardware performance counter metrics alone.

Supervised ML models started from attempts to algorithmically mimic the learning ability of biological neurons. A neuron will only generate an action potential, i.e., fire, if its internal-external depolarization reaches a certain threshold. In other words, there exists a decision boundary. The difficulty in reaching this threshold is modified by certain neurotransmitters. Thus, in mathematical terms, a transformation is performed on input data and the neuron fires (class 1) if the result surpasses the threshold and does not fire (class 0) elsewise. Through training, a neuron learns the proper transformation, which is emulated in supervised machine learning algorithms via the automated minimization of some objective function. To aid with the visualization of these concepts, an example ML application follows.

Consider an application of a linear SVM to the artificial dataset shown in Figure 2.4. From this figure, the data is observed to consist of two features, where each feature forms a dimension, describing two classes of data. Visual inspection reveals that data points which fall into the inner

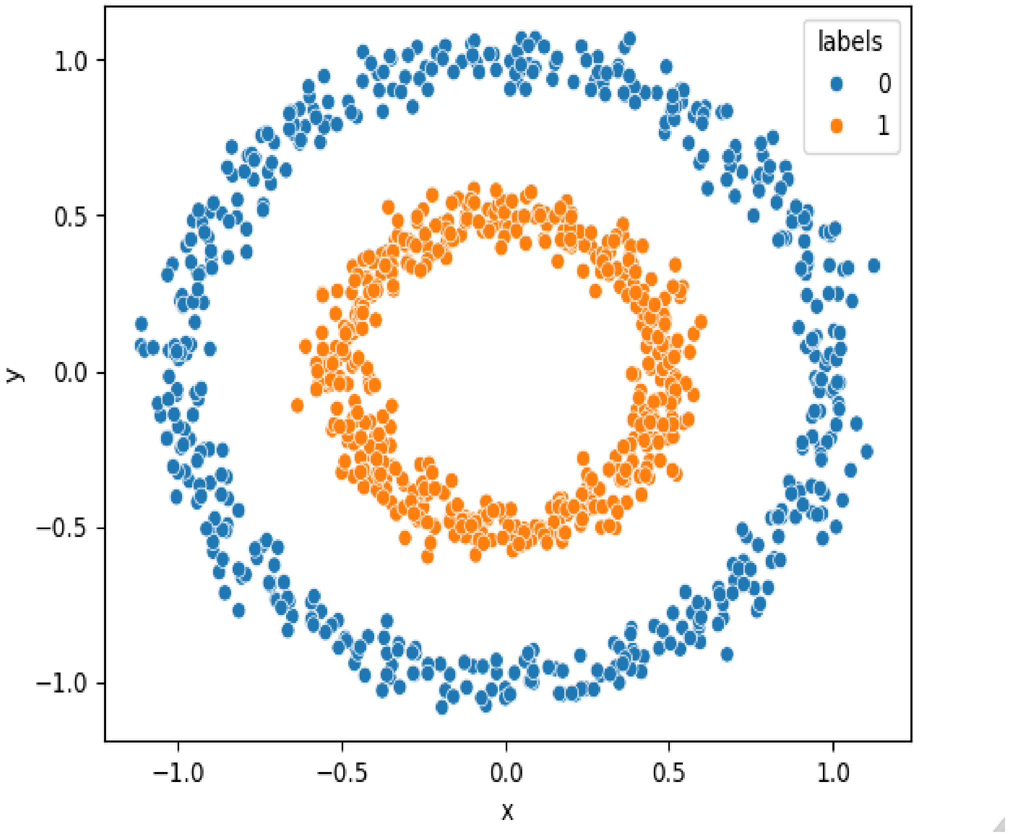


Figure 2.4 Raw Circle Dataset

circle form class 1, whereas data points that fall into the outer circle form class 0. Thus, a circular decision boundary is said to exist between the two regions. Since shallow machine learning models discussed here are capable of learning linear decision boundaries only, a transformation must be found such that a linear decision boundary becomes possible. This is where dataset preprocessing and the machine learning algorithm come into play.

If each data point is given a z-component equal to its distance from the origin, then the two classes become linearly separable in this new three-dimensional subspace. This transformation and decision plane are depicted in figure 2.5. The transformed data is observed to be separable, and the linear SVM classifies each datatype with perfect accuracy using 3-fold cross validation. In practice, such transformations and the subsequent decision boundaries can be found by a combination of manual preprocessing steps, i.e., by applying transformations prior to applying a machine learning model to the data, and automated transformation matrix weight tuning according to objective function-informed gradient descent.

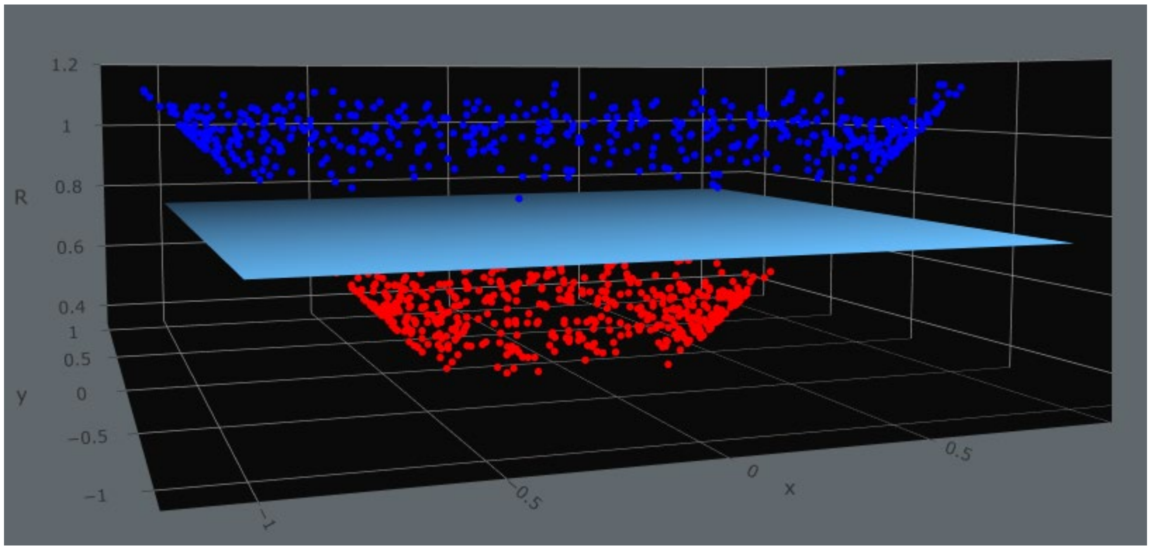


Figure 2.5 Post-transformation dataset with decision plane

Shallow ML is chosen for its computational efficiency and lack of requirement of special hardware. The ability to deploy such pretrained models with little effect on performance precludes the introduction of a bottleneck to already taxed space-grade CPU processing. Furthermore, its compatibility with existing processors, as opposed to neuromorphic models which would require a unique chip design, ensures simpler adaptation to existing host systems.

Unsupervised learning models do not require class labels. Consider again the dataset depicted in figure 2.4. A supervised machine learning model must be informed that the concentric circles belong to different classes before it can begin searching for a subspace with a linear decision boundary – had the left and right halves of the plane been labeled as the two classes, for example, then the model would have declared the decision boundary to be the y-axis. On the other hand, an unsupervised model would be expected to determine the two circles to be of different classes without specific instruction. Thus, whereas a supervised learning model requires a training dataset with labeled datapoints, an unsupervised learning model can extrapolate class information from unlabeled data.

Novelty detectors are a class of unsupervised machine learning algorithm which can learn patterns useful in distinguishing between inlier and outlier samples from training on inlier data. After training, unseen data may be classified as inliers or outliers. Local outlier factor (LOF), specifically, classifies outliers as those datapoints which are in a lower density region compared to other datapoints. LOF can thus be trained on slightly noisy golden data to create a baseline for golden datapoints, after which it should classify fault injected datapoints (which differ in value to an even greater extent than the added noise) as outliers. This removes the need for any software source code modifications. What is needed is to simply train an LOF detector on golden data in a fault-free environment, after which it can be deployed to the faulty environment.

Chapter 3. System Design

This section covers the design aspects of the system developed for this thesis. The benchmark and its modifications, including the integration of PAPI, are presented first. Fault injection and pattern extraction tools are presented next. Finally, their integration into a single system is detailed. Together, these details compose the technical work of this thesis.

3.1 Benchmark Suite

Space applications are often proprietary, which means that their source code is not available to inspection and analysis by the wider community. This lack of transparency makes it difficult for researchers to evaluate their hardware and algorithms. To remedy this problem, the GPU4S project was funded by the European Space Agency (ESA) for the creation of open-source representative benchmarks for space applications (Kosmidis, 2020).

GPU4S includes benchmarks related to machine learning and Digital Signal Processing (DSP) – the two most common space applications. The machine learning subset contains benchmarks for matrix multiplication, 2D convolution, max pooling, ReLU calculations, softmax calculations, Layer-Recurrent Neural Networks (LRNs), and image processing using the CIFAR-10 dataset. The DSP subset contains benchmarks for correlation, FFT, and FIR filter calculations. Also included is a memory bandwidth benchmark. Altogether, these benchmarks cover the gamut of calculations involved in modern machine learning and digital signal processing problems. Expanding on the research presented in (Guererro-Balaguera et al., 2021), this investigation focuses on the matrix_multiplication_bench benchmark.

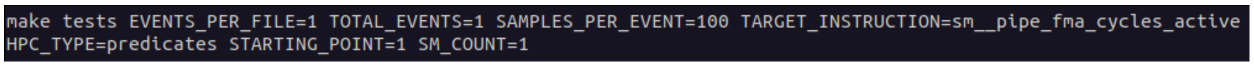
Each benchmark provides several parameters for customization. First is the optimization level. Optimization in the context of the matrix multiplication benchmark refers to the use of GPU tensor cores, which are cores in NVIDIA chips optimized for tensor operations. Although these are worthy fault injection targets, they are reserved for future work.

The next parameter controls the target platform. The supported platforms are CPU, CUDA, OpenCL, OpenMP, or HIP. CPU results in a single-threaded CPU-bound binary. CUDA results in an NVIDIA GPU-enabled binary. OpenCl results in a binary for a heterogeneous environment, such as systems with a combination of CPUs, GPUs, DSPs, and so on. OpenMP results in a multiprocessor/CPU-cluster bound binary. Finally, HIP results in an AMD or NVIDIA GPU-bound binary. Since this thesis utilizes an NVIDIA GPU, CUDA is selected.

The final parameters are the data type, the matrix size, and the block size. The supported datatypes are integer, float, and double. For example, selecting the integer datatype would result in all matrix elements being integer datatypes. Matrix size, on the other hand, controls the size of square matrices utilized in the benchmark. Block size controls the organization of work deployed onto the GPU. Not every benchmark utilizes these parameters – for example, the matrix_multiplication_bench_fp16 benchmark automatically selects the float datatype. However, they are utilized by the matrix_multiplication_bench benchmark.

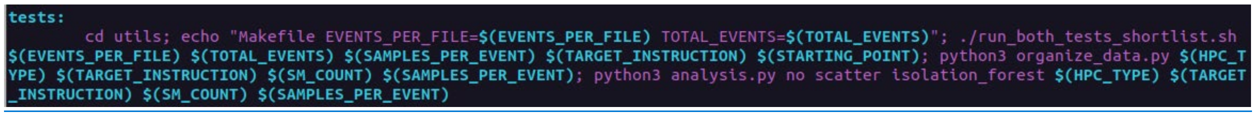
The Performance Application Programming Interface (PAPI), a tool from the University of Tennessee that affords ease of access to hardware performance counter metrics, was integrated with each benchmark via three extra function calls inserted before and after the kernel launch function within each benchmark's main.cpp file. The details of these custom function calls are available in the project's github repository at PAPI/before_launch.c and PAPI/after_launch.c (Teijeiro, 2023). One function – PAPI_pre – initializes the event and performance counter configuration. Another – PAPI_post – destroys the event and counter configuration. Another – read_now – prints the current value of all counters in the configured event set. Together, these three functions simplify the adaptation of a benchmark to PAPI profiling to a few extra lines of code.

Each benchmark has a makefile that is parsed by the Linux make command to build the corresponding libraries and executables from source code. A makefile contains many targets, or designated sets of commands, among which make selects from according to command line arguments. Additional targets were added to the makefile of the matrix_multiplication_bench benchmark for seamless integration with the fault injector and pattern extraction tools. Figure 3.1 depicts an example target selection from the command line. Figure 3.2 depicts the specification for this target inside the corresponding makefile. Figure 3.3 depicts all of the makefile changes added to support these tools.



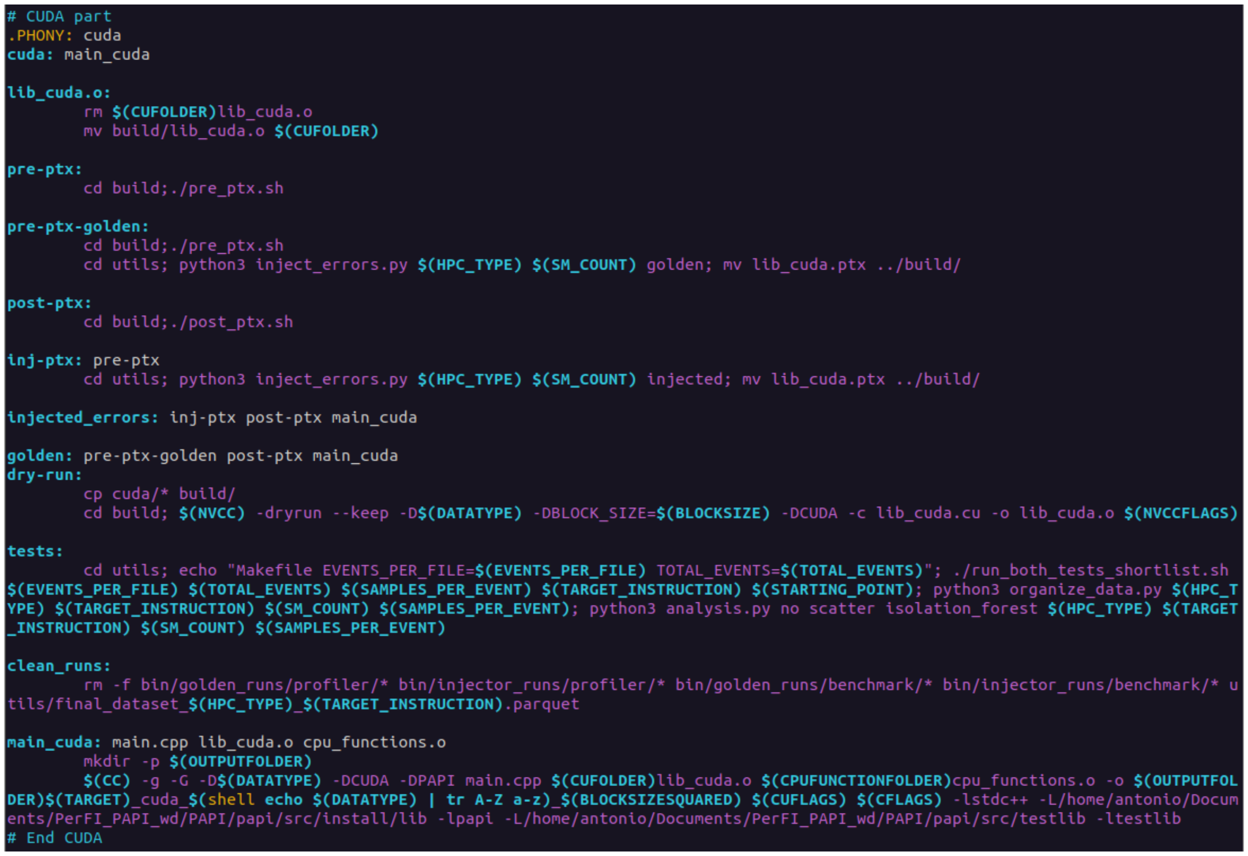
```
make tests EVENTS_PER_FILE=1 TOTAL_EVENTS=1 SAMPLES_PER_EVENT=100 TARGET_INSTRUCTION=sm__pipe_fma_cycles_active
HPC_TYPE=predicates STARTING_POINT=1 SM_COUNT=1
```

Figure 3.1 Linux make command syntax to target the tests rule, along with various parameters.



```
tests:
	cd utils; echo "Makefile EVENTS_PER_FILE=$(EVENTS_PER_FILE) TOTAL_EVENTS=$(TOTAL_EVENTS)"; ./run_both_tests_shortlist.sh
$(EVENTS_PER_FILE) $(TOTAL_EVENTS) $(SAMPLES_PER_EVENT) $(TARGET_INSTRUCTION) $(STARTING_POINT); python3 organize_data.py $(HPC_T
YPE) $(TARGET_INSTRUCTION) $(SM_COUNT) $(SAMPLES_PER_EVENT); python3 analysis.py no scatter isolation_forest $(HPC_TYPE) $(TARGET
_INSTRUCTION) $(SM_COUNT) $(SAMPLES_PER_EVENT)
```

Figure 3.2 Target inside makefile for the tests make command in Figure 3.1



```
# CUDA part
.PHONY: cuda
cuda: main_cuda

lib_cuda.o:
        rm $(CUFOLDER)lib_cuda.o
        mv build/lib_cuda.o $(CUFOLDER)

pre-ptx:
        cd build;./pre_ptx.sh

pre-ptx-golden:
        cd build;./pre_ptx.sh
        cd utils; python3 inject_errors.py $(HPC_TYPE) $(SM_COUNT) golden; mv lib_cuda.ptx ../build/

post-ptx:
        cd build;./post_ptx.sh

inj-ptx: pre-ptx
        cd utils; python3 inject_errors.py $(HPC_TYPE) $(SM_COUNT) injected; mv lib_cuda.ptx ../build/

injected_errors: inj-ptx post-ptx main_cuda

golden: pre-ptx-golden post-ptx main_cuda
dry-run:
        cp cuda/* build/
        cd build; $(NVCC) -dryrun --keep -D$(DATATYPE) -DBLOCK_SIZE=$(BLOCKSIZE) -DCUDA -c lib_cuda.cu -o lib_cuda.o $(NVCCFLAGS)

tests:
        cd utils; echo "Makefile EVENTS_PER_FILE=$(EVENTS_PER_FILE) TOTAL_EVENTS=$(TOTAL_EVENTS)"; ./run_both_tests_shortlist.sh
$(EVENTS_PER_FILE) $(TOTAL_EVENTS) $(SAMPLES_PER_EVENT) $(TARGET_INSTRUCTION) $(STARTING_POINT); python3 organize_data.py $(HPC_T
YPE) $(TARGET_INSTRUCTION) $(SM_COUNT) $(SAMPLES_PER_EVENT); python3 analysis.py no scatter isolation_forest $(HPC_TYPE) $(TARGET
_INSTRUCTION) $(SM_COUNT) $(SAMPLES_PER_EVENT)

clean_runs:
        rm -f bin/golden_runs/profiler/* bin/injector_runs/profiler/* bin/golden_runs/benchmark/* bin/injector_runs/benchmark/* u
tils/final_dataset_$(HPC_TYPE)_$(TARGET_INSTRUCTION).parquet

main_cuda: main.cpp lib_cuda.o cpu_functions.o
        mkdir -p $(OUTPUTFOLDER)
        $(CC) -g -G -D$(DATATYPE) -DCUDA -DPAPI main.cpp $(CUFOLDER)lib_cuda.o $(CPUFUNCTIONFOLDER)cpu_functions.o -o $(OUTPUTFOL
DER)$(TARGET)_cuda_$(shell echo $(DATATYPE) | tr A-Z a-z)_$(BLOCKSIZESQUARED) $(CUFLAGS) $(CFLAGS) -lstdc++ -L/home/antonio/Docum
ents/PerFI_PAPI_wd/PAPI/papi/src/install/lib -lpapi -L/home/antonio/Documents/PerFI_PAPI_wd/PAPI/papi/src/testlib -ltestlib
# End CUDA
```

Figure 3.3 Updated makefile.

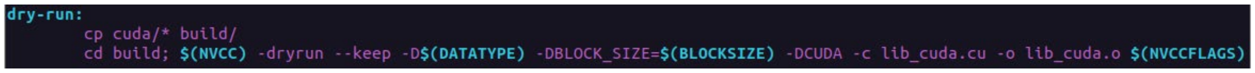
3.2 Fault Injector

NVIDIA provides a fault injector called NVBitFI to inject faults down at the SP-level. Guererro-Balaguera et al. created NVBitPerFI, built upon NVBitFI, in order to inject faults in parallelism management units (Guerrero-Balaguera et al., 2021). However, both require subscribing to the CUPTI Profiling API, which is also required by other profiling tools. Since the CUPTI Profiling API allows only one subscriber, with no way for the user to modify the source code, a new fault injector based upon the principles of NVIDIA's deprecated SASSIFI fault injector was created for this thesis.

The build process for CUDA code is not so dissimilar to that for CPU-bound code. CPU-bound code is converted to assembly, then to object code, then linked to become a binary. CUDA code faces an additional step in the generation of PTX code (platform-independent assembly code) prior to generating SASS code (platform-specific assembly code). SASS code is hidden from the end user. Thus, the lowest level available for fault injection is at the level of PTX code.

Low level fault injection has the advantage that more of the fundamental mechanisms of program implementation are exposed. In the case of PTX code, the final destination registers are abstracted by a virtual register name. This allows the low-overhead introduction of bit flips (both temporary and stuck-at) to the destination address of each operation involving a targeted hardware unit, thus emulating a malfunctioning unit. This is the basis of the fault injector: first, intercept the compilation process just after the generation of the PTX file; second, insert the necessary bit flip operations; and third, continue the compilation process with this modified PTX file. The fault injector is implemented through the makefile "tests" target and its dependencies.

The PTX code was intercepted by splitting up the NVIDIA compilation process and keeping the intermediate files. This was done by adding the -dry-run and --keep commands to the normal nvcc build command found originally in the lib_cuda.o target in the benchmark's makefile, as shown in Figure 3.4.



```
dry-run:
	cp cuda/* build/
	cd build; $(NVCC) -dryrun --keep -D$(DATATYPE) -DBLOCK_SIZE=$(BLOCKSIZE) -DCUDA -c lib_cuda.cu -o lib_cuda.o $(NVCCFLAGS)
```

Figure 3.4 Makefile Dry Run Target

This command outputs a file containing all the commands run in the background by the usual NVIDIA compilation using the nvcc command. Two scripts were then made: pre_ptx.sh contains the commands up to and including PTX generation, and post_ptx.sh contains the remaining commands. Since the CUDA code for these benchmarks is in a source file called lib_cuda.cu, the output of the final step of pre_ptx.sh is called lib_cuda.ptx. This file is parsed by inject_errors.py, which inserts fault injection instructions before or after the appropriate PTX instruction according to the targeted hardware unit. Once this is finished, post_ptx.sh runs.

Several considerations influenced the development of inject_errors.py. The first consideration is the effect of NVIDIA GPU branch divergence on unit activity. Branch divergence refers to threads within a grid executing different sequences of instructions within a program due to differing evaluations of a conditional statement. Since all threads within a warp share hardware necessary for executing instructions, an intra-warp branch divergence requires multiple passes of the problematic section of code to satisfy all threads, causing a massive performance penalty. Thus, intra-warp branch divergence always results in increased unit activity. From figure 2.3, it is clear that this fault injector relies on the conditional execution of fault injection instructions according to SM, SM subpartition, and SP IDs. Hence, branch divergence activity must be accounted for.

Consider the snippet of code from the matrix_multiplication_bench lib_cuda.ptx file in figure 3.5. In this code, the thread index within the grid is determined – a step present in every GPU kernel. Once determined, the thread index is compared with an upper bound to ensure that threads which exceed the boundary of the problem exit early. The thread index is calculated using an Integer Multiply Add (IMAD) statement, which is executed on the FMA pipeline. Therefore, should an SEU affect an FMA pipeline so as to skew the thread index calculation, then abnormal GPU activity may be expected due to anomalous thread exit activity.

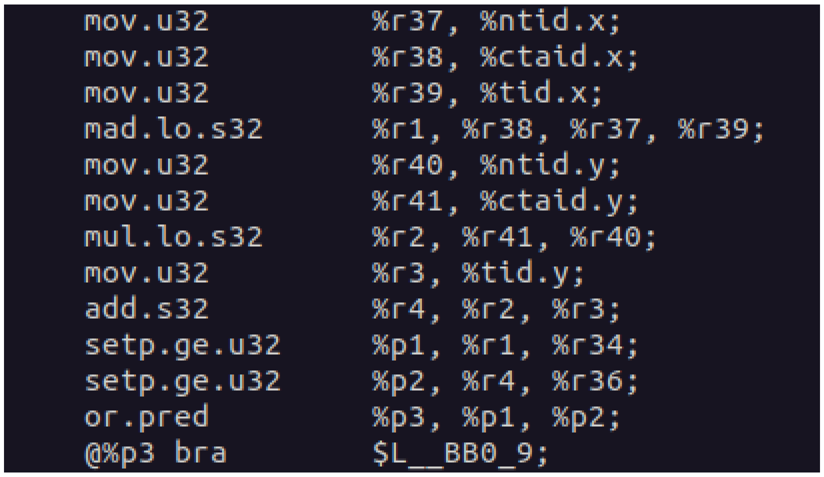
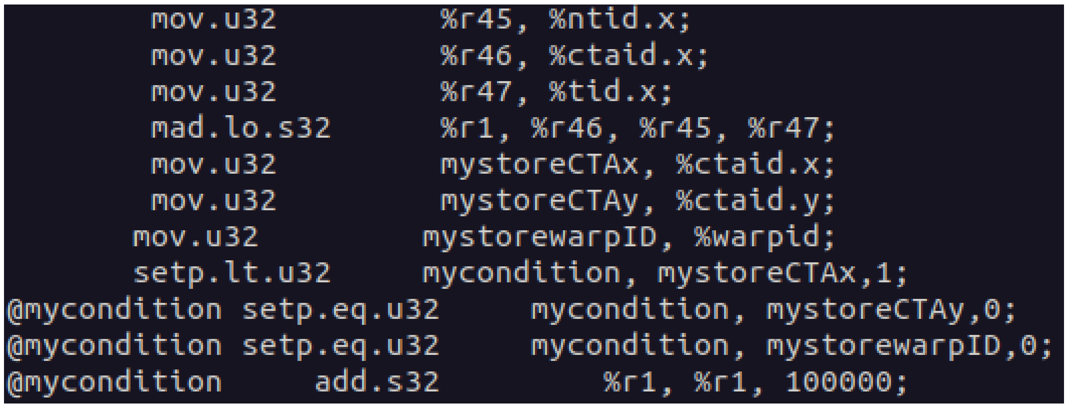

```
mov.u32         %r37, %ntid.x;
mov.u32         %r38, %ctaid.x;
mov.u32         %r39, %tid.x;
mad.lo.s32      %r1, %r38, %r37, %r39;
mov.u32         %r40, %ntid.y;
mov.u32         %r41, %ctaid.y;
mul.lo.s32      %r2, %r41, %r40;
mov.u32         %r3, %tid.y;
add.s32         %r4, %r2, %r3;
setp.ge.u32     %p1, %r1, %r34;
setp.ge.u32     %p2, %r4, %r36;
or.pred         %p3, %p1, %p2;
@%p3 bra        $L__BB0_9;
```

Figure 3.5 Thread Index Calculation

Since the matrix size used in this work was fixed at 104x104=10,816, valid thread indices range from zero to 10,815. The injected fault adds 100,000 to the output of the thread index calculation, so any affected SP should fall outside of the valid range and exit early. Since subsequent instructions in the code branch that fails the fault condition check involve the FMA unit, it is reasonable to measure anomalous thread exit activity via FMA activity. Thus, a drop in FMA activity was expected. In reality, either a large increase or decrease in FMA activity is observed according to the fault condition applied; an intra-warp branch divergence, caused by targeting a specific SP, elicits an *increase* in FMA activity due to multiple passes of the code. An inter-warp branch divergence, caused by targeting a specific warp within a cooperative thread array, elicits the desired decrease. Figure 3.6 displays this SEU simulation via an injected fault in the referenced IMAD operation’s result register (%r1). Table 3.1 displays FMA activity from before and after this fault injection.



```
        mov.u32         %r45, %ntid.x;
        mov.u32         %r46, %ctaid.x;
        mov.u32         %r47, %tid.x;
        mad.lo.s32      %r1, %r46, %r45, %r47;
        mov.u32         mystoreCTAx, %ctaid.x;
        mov.u32         mystoreCTAy, %ctaid.y;
       mov.u32         mystorewarpID, %warpid;
       setp.lt.u32     mycondition, mystoreCTAx,1;
@mycondition setp.eq.u32     mycondition, mystoreCTAy,0;
@mycondition setp.eq.u32     mycondition, mystorewarpID,0;
@mycondition     add.s32         %r1, %r1, 100000;
```

Figure 3.6 Injected Thread Index Calculation

Table 3.1: FMA Activity Before and After Fault Injection

	Metric	Value	Label
1.	sm__pipe_fma_cycles_active.sum	1759744	golden
2.	sm__pipe_fma_cycles_active.sum	1759744	golden
3.	sm__pipe_fma_cycles_active.sum	1759744	golden
4.	sm__pipe_fma_cycles_active.sum	1759744	golden
5.	sm__pipe_fma_cycles_active.sum	1758528	injected
6.	sm__pipe_fma_cycles_active.sum	1758528	injected
7.	sm__pipe_fma_cycles_active.sum	1758528	injected
8.	sm__pipe_fma_cycles_active.sum	1758528	injected

From the preceding discussion, it is clear that intra-warp branch divergence alters GPU activity levels significantly. Therefore, deciding the hierarchical level at which to inject an error is critical for the correct interpretation of results. The FMA pipeline is shared by all threads within an SM subpartition, so errors should be injected at the SM subpartition level and not the SP level. Other hardware, such as local registers, are local to an SP and would merit SP-level injection. The specific injection methodology utilized for each simulated SEU in this work is discussed in section 4.2.

The next consideration is the fact that PTX code undergoes Just-In-Time (JIT) compilation – compilation which occurs at runtime – to SASS code, which is then assembled to the final binary, onboard the GPU prior to execution. Hence, just as high-level source code can differ in implementation from its PTX code, PTX implementation of source code can differ from its final binary implementation on the GPU. Thus, the presence of additional fault injection instructions themselves may alter the implementation of the source code in an unpredictable manner. This in turn may induce sufficient anomalous GPU activity to produce false positives upon analysis of the metrics if unaccounted for.

Consider running this benchmark with a pseudo fault injection. Here, an impossible condition – an SM ID of less than zero – needs to be satisfied for the SM to be injected. Despite no SM being injected, FMA metric activity was observed to differ significantly between the two trials. Without access to the underlying SASS code, it is impossible to specify exactly what changed in the compilation of PTX to SASS. However, it is clear that some significant differences emerged. Thus, even a few extra lines of PTX code can result in a significantly different SASS implementation. A mitigation step, therefore, is to minimize the difference between PTX implementations by inserting the same fault injection instructions into the lib_cuda.ptx files for both golden and injected runs, changing only the fault condition each time to include or exclude hardware. This approach can then be double checked using trend information. Figure 3.7 displays the FMA activity trend for repeated upsets observed from applying this fault injection methodology on the FMA pipeline.

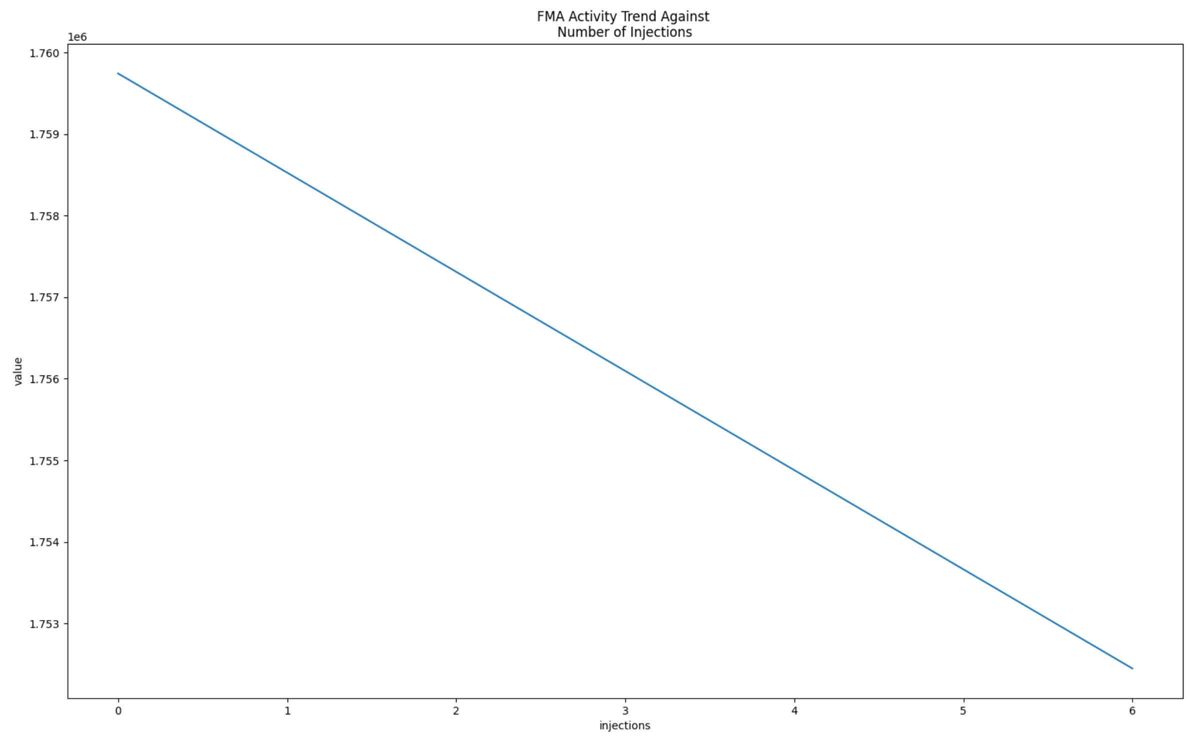


Figure 3.7 FMA Activity Trend vs. Number of Injections

Both Single Event Upsets (SEUs) and Multi Event Upsets (MEUs) were initially considered for this work. In radiation effects, an SEU refers to an error state induced by a single particle strike. Conversely, a multievent upset refers to an error state induced by multiple simultaneous particle strikes. Since a chip in space can be presumed to be bathed in radiation, MEU investigation is clearly of importance. However, skipping an SEU analysis precludes the study of counter activity from MEUs as a linear phenomenon in future work. Furthermore, SEUs are harder to detect and therefore can be argued to satisfy a higher burden of proof. Therefore, this work focuses exclusively on SEUs.

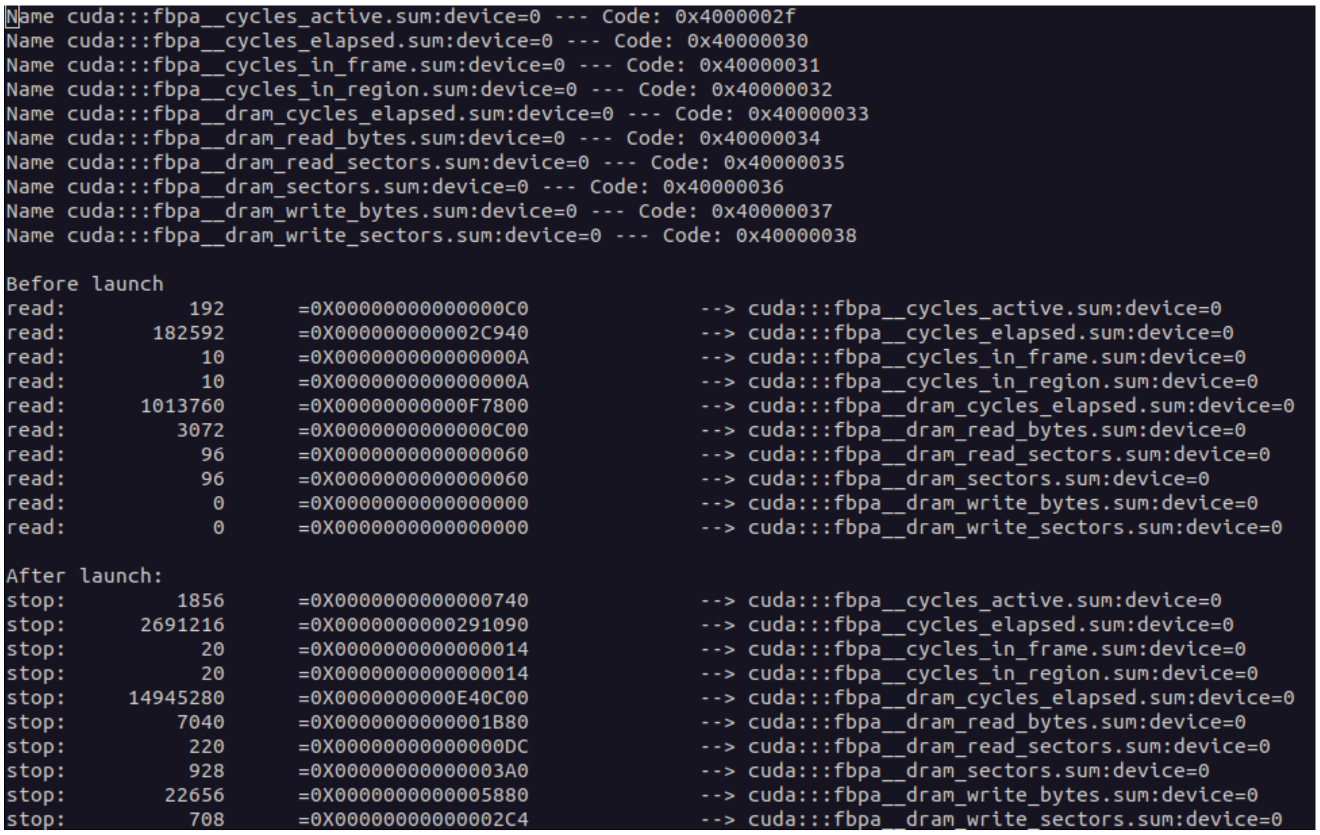
In some cases, SEU simulation implies foreknowledge of the manner in which work is deployed on a GPU. However, in practice this is not feasible. Although specific SMs can be targeted (which is sufficient for L1 cache injection), SM subpartitions cannot – only warps within a cooperative thread array may be targeted. Further complicating matters is the fact that warps can be renumbered during preemption. Thus, a targeted warp may be preempted prior to reaching a fault condition check, thus failing the condition upon resuming execution. The only reasonable solution is to monitor results for this phenomenon by verifying a significant difference between golden and injected trials. Fortunately, this did not become an issue during this work, and SEUs were successfully simulated by targeting CTA coordinate and warp ID combinations. The SEU thus gets simulated wherever this combination is scheduled onto the GPU for a particular trial, introducing an element of hardware locality independence to this procedure. Those SEUs which affect an entire SM at once, i.e. L1 cache faults, are injected at the SM level instead of the warp level.

Finally, the benchmark itself influenced the development of inject_errors.py. For example, the uniform datapath is not used in the execution of these benchmarks, and thus cannot be targeted. Furthermore, since small changes in source code can result in large differences in PTX implementations and subsequent SASS implementations, differences between benchmark PTX files are usually too significant to write a general-purpose parsing script. Since PTX code is assembly language code, its inspection is laborious. For all of these reasons, inject_errors.py is particular to the matrix_multiplication_bench benchmark.

Each SEU simulation is implemented via a call to the corresponding fault injection function in inject_errors.py. For reasons discussed in section 4.1, "Scope," the FMA pipeline, ALU, ADU, LSU, and L1 instruction cache are targeted in this work. For aforementioned reasons, the implementation of each SEU simulation is particular to the matrix_multiplication_bench benchmark and the targeted hardware. As such, the details of the functions targeting these hardware units are discussed in section 4.3, "Experimental Setup and Procedure."

3.3 Pattern Extraction Tools

Two python scripts compose the pattern extraction tool. One script reads and reformats PAPI output, and another utilizes these readings to evaluate a support vector machine or LOF model's success with detecting upsets. PAPI output from each uninjected benchmark is written to $benchmark_directory/bin/golden_runs, but to $benchmark_directory/bin/injector_runs for each injected benchmark. This is done automatically through the rule "tests" added to the benchmark's makefile. The data lists the value of each metric at the time it was read. Figure 3.8 displays the contents of an example output file. Each run contains its own output file. This output is parsed by organize_data.py, which reformats the data into a polars data frame and saves it to a parquet file. In this way, kernel profile metrics are organized for easier parsing by analysis.py.



```
Name cuda:::fbpa__cycles_active.sum:device=0 --- Code: 0x4000002f
Name cuda:::fbpa__cycles_elapsed.sum:device=0 --- Code: 0x40000030
Name cuda:::fbpa__cycles_in_frame.sum:device=0 --- Code: 0x40000031
Name cuda:::fbpa__cycles_in_region.sum:device=0 --- Code: 0x40000032
Name cuda:::fbpa__dram_cycles_elapsed.sum:device=0 --- Code: 0x40000033
Name cuda:::fbpa__dram_read_bytes.sum:device=0 --- Code: 0x40000034
Name cuda:::fbpa__dram_read_sectors.sum:device=0 --- Code: 0x40000035
Name cuda:::fbpa__dram_sectors.sum:device=0 --- Code: 0x40000036
Name cuda:::fbpa__dram_write_bytes.sum:device=0 --- Code: 0x40000037
Name cuda:::fbpa__dram_write_sectors.sum:device=0 --- Code: 0x40000038

Before launch
read:          192      =0X00000000000000C0             --> cuda:::fbpa__cycles_active.sum:device=0
read:       182592      =0X000000000002C940             --> cuda:::fbpa__cycles_elapsed.sum:device=0
read:           10      =0X000000000000000A             --> cuda:::fbpa__cycles_in_frame.sum:device=0
read:           10      =0X000000000000000A             --> cuda:::fbpa__cycles_in_region.sum:device=0
read:      1013760      =0X00000000000F7800             --> cuda:::fbpa__dram_cycles_elapsed.sum:device=0
read:         3072      =0X0000000000000C00             --> cuda:::fbpa__dram_read_bytes.sum:device=0
read:           96      =0X0000000000000060             --> cuda:::fbpa__dram_read_sectors.sum:device=0
read:           96      =0X0000000000000060             --> cuda:::fbpa__dram_sectors.sum:device=0
read:            0      =0X0000000000000000             --> cuda:::fbpa__dram_write_bytes.sum:device=0
read:            0      =0X0000000000000000             --> cuda:::fbpa__dram_write_sectors.sum:device=0

After launch:
stop:         1856      =0X0000000000000740             --> cuda:::fbpa__cycles_active.sum:device=0
stop:      2691216      =0X0000000000291090             --> cuda:::fbpa__cycles_elapsed.sum:device=0
stop:           20      =0X0000000000000014             --> cuda:::fbpa__cycles_in_frame.sum:device=0
stop:           20      =0X0000000000000014             --> cuda:::fbpa__cycles_in_region.sum:device=0
stop:     14945280      =0X0000000000E40C00             --> cuda:::fbpa__dram_cycles_elapsed.sum:device=0
stop:         7040      =0X0000000000001B80             --> cuda:::fbpa__dram_read_bytes.sum:device=0
stop:          220      =0X00000000000000DC             --> cuda:::fbpa__dram_read_sectors.sum:device=0
stop:          928      =0X00000000000003A0             --> cuda:::fbpa__dram_sectors.sum:device=0
stop:        22656      =0X0000000000005880             --> cuda:::fbpa__dram_write_bytes.sum:device=0
stop:          708      =0X00000000000002C4             --> cuda:::fbpa__dram_write_sectors.sum:device=0
```

Figure 3.8 Example profiling output

A python script called analysis.py reads the parquet file created by organize_data.py into a polars data frame and utilizes this data to enable machine learning and visualization capabilities. Using command line arguments, the user can choose to view either scatterplots or Kernel Density Estimation (KDE) distributions of the data, as well as between evaluating an SVM or LOF model.

Metric values are not combined in this work to form a high-dimensional dataset. Instead, their suitability for upset detection is evaluated in isolation. However, a scatterplot of one-dimensional data will appear cluttered. Thus, a small noise y-component is added to each metric to facilitate its scatterplot. By allowing the user to observe all datapoints graphically, the possibility of a linear decision boundary can be deduced for troubleshooting purposes.

Another method of visual inspection for linear decision boundaries is to use KDE estimation. KDE estimation infers a random variable's distribution from a finite number of samples. This is helpful in inferring the general characteristics of a metric, which in turns aids with developing generalized conclusions about its suitability for error detection. However, its application requires the data to have nonzero variance. Thus, metrics which remain constant across program trials are unsuitable for KDE estimation. This can be remedied by the addition of a small noise component, a non-zero positive value less than 1.0, to all metric samples. Since metrics value realizations typically occur on the scale of two to six orders of magnitude, this small noise component does little to alter the information present in the data. Figure 3.9 depicts an example estimated distribution set from injecting errors into cached instructions.

The SVM process begins by shuffling the dataset prior to training. Sequentially similar samples will influence the gradient descent process to optimize for the consistent classification of one class over another; if there are many golden or injected datapoints in a row, then the algorithm's transformation weights will be influenced such that everything is classified as golden or injected, respectively. Three-fold cross-validation with an SVM, meaning that the model is retrained 3 times on a different 2/3 of the shuffled dataset, is utilized to verify metric suitability for detecting errors. This also helps to ensure the model learns an informative pattern.

LOF application begins with the shuffling of the dataset. However, since the objective is to train a model capable of upset detection using only golden data, injected data is dropped for training. Metrics which do not exhibit any variance across trials also suffer in LOF application, but a similar solution to the KDE problem is applied. Here. A noise component ranging from -20 to 20 is added to the data to encourage the LOF model to learn an informative pattern for distinguishing between inliers and outliers. Next, the trained LOF model is applied for novelty detection to the complete dataset – injected data included. A comparison between the labels and LOF predictions results in a reported accuracy as a percentage of correctly classified samples – with golden data classified as inliers and injected data classified as outliers.

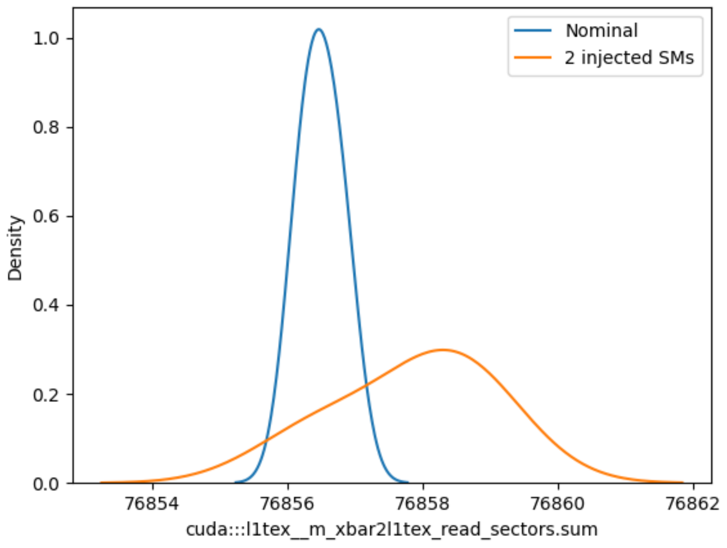


Figure 3.9 Example golden vs. injected metrics distribution.

Chapter 4. Experimentation

This thesis investigates the application of hardware performance counter metrics to error detection. In the following, experimental work is described. Results and discussion are presented in the next chapter.

4.1 SCOPE

From the Nsight compute profiling guide, there are 17 units, 15 subunits, and 13 pipelines available for targeting. Of the 17 units, many are too general to be considered targets. For example, an SM subpartition unit consists of many subunits and pipelines. Injecting an SM subpartition therefore corresponds to injecting at these lower levels in the hierarchy. The remaining fault injection candidates are the DRAM, L2 cache, L2 cache fabric, L1/Texture cache, indexed constant cache, and FPBA units. The DRAM units are not injected due to the presumption of ECC availability. L2 cache and L2 cache fabric injections are not performed due to the inability to directly control L2 cache. Indexed Constant Cache is ignored due to the inability to stimulate changes in its metrics for this benchmark– the instructions that are utilizing it appear to be generated during compilation to SASS. The FPBA unit is also ignored due to the absence of a mechanism for its direct control. Not mentioned in the profiling guide, but of consequence, is the L0 instruction cache within each SM subpartition. Thus, the remaining target is the L0 instruction cache.

Of the 15 subunits, many are not targetable for this work. For example, many have a pipeline analog; the LSU subunit also falls into the category of a pipeline. Others are unavailable for direct control, such as the aperture_device subunit, which is a memory interface to device

DRAM. Those which are unique and available for direct control are the local and shared memory. However, these two subunits are left for future work as well.

Of the 13 pipelines, the FMA logical pipeline envelopes the FMAHeavy, FMALite, and FP16 pipelines. Since this work focuses on computational workloads, a further two – the tensor memory access and texture units – can be ignored due to the decision to leave tensor processors and graphics workloads for future work. Of the remaining eight pipelines, the uniform, special function, and FP64 pipelines are not used in the execution of this benchmark (as learned from observing corresponding metric values of zero). Therefore, the remaining subunit targets are the ADU, ALU, CBU, LSU, and FMA units. Of these, CBU SEU simulation is left for future work.

Thus, one unit and four pipelines remain for targeting. These five hardware units are the L1 cache, ALU, LSU, ADU, and FMA. Complete SEU simulation coverage in these hardware units is out of the scope of this work; instead, one SEU simulation in each hardware component type will be experimented in order to evaluate the potential of this topic for further research.

4.2 Using Hardware Performance Counters

Since radiation hardened-by-design GPUs, i.e., GPUs that are inherently insensitive to ionizing radiation, are not feasible using current technology, upset detection and correction are vital to bringing COTs GPUs to space. At present, upsets are detected and corrected using duplication and triplication of work, respectively. These methods are computationally expensive. The question to explore is whether GPU hardware performance counter metrics are reliably affected by the presence of hardware faults induced by single-event upsets. If so, the next question is whether they produce enough information to use them as symptomatic fault detectors.

Hardware performance counters, as indicators of GPU activity, inherently contain information relevant to error detection. However, since NVIDIA GPU hardware performance counters are only capable of measuring unit activity, it is not clear at this time whether every kind of error will be detectable. For example, if a single event upset results in the incorrect calculation of one element a matrix but does not result in a cache miss, a different execution time, etc. then it will not be possible to detect this error using performance counters. However, the faulty hardware performing this calculation may perform another operation that does affect unit activity. Regardless, there are certainly types of upsets that are likely to disturb unit activity levels and therefore will be detectable from performance counter metrics.

From Chapter 2, Theoretical Foundations, it is observed that cache misses will result in increased M-Stage, frame buffer, and device DRAM activity. This is because The L1/Texture cache M-stage handles L1 cache misses, passing on the request to L2 cache. The Frame buffer handles L2 cache misses, passing on the request to device DRAM. Then, the data will be retrieved from or stored to device DRAM. Thus, upsets which result in a cache miss will be detectable from metrics concerning the M-stage, frame buffer, and device DRAM.

The L0 i-Cache, LSU, and ALU are involved in the storage of data to the proper location. The instruction containing the correct base address and offset information resides in L0 instruction cache. The ALU uses this information, summing the destination matrix base address and element offset in an add instruction and storing the result in a pointer register for use in a succeeding store instruction. The LSU then executes the store instruction. Thus, an SEU in any of these three hardware components can result in a cache miss, which is detectable from the aforementioned metrics.

From Chapter 3.2, it is observed that thread index miscalculations can result in anomalous thread exit activity. Every GPU kernel must include instructions for calculating each thread's index within the grid of threads. This calculation is usually performed using an IMAD instruction, as with the matrix_multiplication_bench benchmark. Since the IMAD instruction is executed in the FMA pipeline, upsets affecting an FMA pipeline will affect thread index calculations. These miscalculations can result in anomalous thread exit activity, which can be detected from the activity of pipelines utilized to execute instructions succeeding the thread index calculation. As deduced from the presence of additional IMAD instructions in subsequent PTX code, the FMA pipeline is used several times following the thread index calculation in the matrix_multiplication_bench benchmark. Therefore, an SEU affecting the FMA pipeline can be expected to induce anomalous FMA activity. Thus, SEUs affecting the FMA pipeline will be detectable from FMA metrics.

Guerrero-Balaguera et al. present their findings on the monitoring of Divergence Management Unit (DMU) activity for fault detection on the FlexGripPlus GPU (Guerrero-Balaguera et al., 2021). In their research, they showed that DMU activity could detect upsets which resulted in intra-warp branch divergence. The equivalent to the DMU in modern NVIDIA GPUs is the ADU. Since the ADU handles divergence, a faulty ADU would cause anomalous branch divergence activity. Since the FMA pipeline is utilized to execute instructions throughout the benchmark, FMA activity would also indicate anomalous ADU activity.

L0 instruction cache, LSU, and ALU SEUs which result in cache misses should be detectable from miss, hit, M-stage, frame buffer, and device DRAM metrics. FMA SEUs that result in thread index miscalculations should be detectable from FMA activity. ADU SEUs which result in abnormal branch divergence activity should be also detectable from FMA activity. Although the latter two assertions are applicable only for the matrix_multiplication_bench benchmark, the first is generally applicable. Thus, the SEUs to simulate and the metrics to monitor have been determined.

4.2 EXPERIMENTAL SETUP AND PROCEDURE

All work was performed in a system with the following hardware: 1. GPU: RTX 3090; 2. Motherboard: ASUS Z590-E motherboard; 3. CPU: Intel i9-10850k; 4. RAM: 32 GB. Simulations of the SEUs described in section 4.2 were implemented in inject_errors.py.

Instructions to be executed are often held in L0 instruction cache. These instructions vary in format but can in general be expected to contain fields for an opcode, source, and destination address. Thus, if an SEU occurs in L0 instruction cache at the destination address portion of a store instruction, which is used to write back to memory, then the data will be stored in a memory location likely not cached. The ALU applies the offset for each matrix element to the aforementioned base address in order to determine its final store destination address. Thus, an SEU in the ALU during this computation can also result in a cache miss. Finally, the LSU is involved in the correct relocation of information to memory. Thus, an SEU in the LSU during the store operation can also result in a cache miss.

These three SEUs all result in the calculation of an invalid destination address for use with the store instruction. Therefore, they can all be approximated in inject_errors.py by toggling a bit within the set index portion of the destination address, albeit at different levels within the GPU hierarchy. In lieu of official NVIDIA cache policy documentation, trial and error was required to find the set index location within the destination address. From this procedure, the 2^{20} bit position was determined to form part of the set index. Thus, the “inject_L0” function within inject_errors.py implements a bitwise XOR operation of 2^{20} on the registered destination matrix element address in lib_cuda.ptx at the warp level. The “inject_LSU” function performs the same operation at the quarter-warp level, since there are four LSUs per SM subpartition. Finally, the “inject_ALU” operation performs this operation at the SP level. In this manner, all three SEUs are simulated with the same operation at different levels of the GPU hardware hierarchy.

In PTX code, branching is controlled through predicate guards. These predicate guards start with an “@” symbol, possibly followed by a logic symbol, followed by a specially declared predicate register. For example, “@!%p1” means “if predicate register p1 is Boolean false.” Predicate guards precede lines of code to be conditionally executed. Predicate registers are set to true or false according to a preceding “setp” (set predicate)_instruction. For example, “setp.lt.s32 %p1,%r2, %r3” means “set predicate register %p1 to true if %r2 is less than %r3, else set it to false. Thus, abnormal ADU activity can be approximated by changing the state of predicate registers just before they are used in predicate guards. This functionality is implemented in the function “inject_predicates” within inject_errors.py.

In every GPU kernel, each thread must determine its index in order to determine its responsible area of work – if any – within the kernel. This calculation occurs in the form of an Integer Multiply Add (IMAD) instruction, whose operands come from special ID registers called *predefined identifiers* within the GPU. IMAD instructions are executed on the FMA pipeline, which is shared at the SM subpartition level. The IMAD instruction result is placed in a general-purpose register, which can be injected just after calculation to simulate an SEU in the FMA pipeline. This functionality is implemented in the function “inject_fma” within inject_errors.py.

The grid of threads is laid out as follows. In each test, the matrix size is set to 104x104 elements to ensure that work is scheduled onto all SPs, as there are 10496 SPs in the GA102. A block size of shape 16x16 is chosen, for a total of 8 warps per block. Thus, a grid of 7x7 blocks of 16x16 threads each is chosen. From this information, CTA coordinates (0,0) and a warp ID of 0 is selected for targeting. In each test, 1000 golden runs and 1000 fault injected runs were conducted to ensure the statistical significance of results. The data from each test is combined into a dataset that includes labels to distinguish between golden and injected data points. An SVM's ability to distinguish between these two classes is then evaluated, with success determined by the accuracy of the SVM under 3-fold cross validation. As a second method of evaluation, an LOF model's generalization ability to classify previously unseen injected samples as outliers is experimented.

Chapter 5. Results

Table 5.1 displays the SEU detection results using those metrics experimentally determined to be the most useful for detecting each type of SEU. Displayed in each row is the targeted hardware, the name of the metric, an SVM's accuracy on its data under K-fold cross validation, and an LOF's accuracy on its data. Table 5.2 presents SEU detection results for a sampling of less sensitive metrics which were hypothesized to be informative. Table 5.3 presents results from repeating SEUs simulated in table 5.2 once per CTA.

Table 5.1: Best SEU Experimental Results

Targeted Hardware	Metric	3-fold CV Accuracy	LOF Accuracy
L0 i-Cache	lts__t_requests_aperture_device_evict_normal_lookup_miss.sum	[0.96701649 0.97601199 0.96696697]	91.55%
LSU	lts__t_requests_aperture_device_evict_normal_lookup_miss.sum	[0.93553223 0.93703148 0.95495495]	72.6%
ALU	lts__t_requests_aperture_device_evict_normal_lookup_miss.sum	[0.86656672 0.87556222 0.87987988]	49.95%
FMA	sm__pipe_fma_cycles_active.sum	[1. 1. 1.]	100.0
ADU	sm__pipe_fma_cycles_active.sum	[1. 1. 1.]	100.0

Table 5.2: Less Sensitive Metrics (SEU)

Targeted Hardware	Metric	3-fold CV Accuracy	LOF Accuracy
L0 i-Cache	l1tex__t_bytes_pipe_lsu_lookup_miss.sum	[0.79910045 0.77961019 0.75825826]	69.8%
L0 i-Cache	fbpa__dram_write_bytes.sum	[0.48275862 0.48125937 0.51201201]	48.8%

LSU	l1tex__t_bytes_pipe_lsu_lookup_miss.sum	[0.63568216 0.63118441 0.63963964]	53.45%
LSU	fbpa__dram_write_bytes.sum	[0.48575712 0.47826087 0.503003]	51.05%
ALU	l1tex__t_bytes_pipe_lsu_lookup_miss.sum	[0.57721139 0.5892054 0.56606607]	49.1%
ALU	fbpa__dram_write_bytes.sum	[0.55922039 0.51124438 0.51951952]	50.25%

Table 5.3: Less Sensitive Metrics, Repeated SEU

Targeted Hardware	Metric	3-fold CV Accuracy	LOF Accuracy
L0 i-Cache	l1tex__t_bytes_pipe_lsu_lookup_miss.sum	[1. 1. 1.]	93.85%
L0 i-Cache	fbpa__dram_write_bytes.sum	[0.98650675 0.98350825 0.98048048]	91.6%
LSU	l1tex__t_bytes_pipe_lsu_lookup_miss.sum	[1. 1. 1.]	98.85%
LSU	fbpa__dram_write_bytes.sum	[0.95352324 0.93853073 0.95195195]	86.75%
ALU	l1tex__t_bytes_pipe_lsu_lookup_miss.sum	[0.98650675 0.988006 0.996997]]	98.35%
ALU	fbpa__dram_write_bytes.sum	[0.7946027 0.79610195 0.80930931]	70.3.%

5.2 Discussion

The performed L0 instruction cache, LSU, and ALU fault injections were expected to cause a cache miss, resulting in increased L1Tex M-stage, Frame Buffer, and DRAM activity in the affected unit. It was posited that this increased activity would be reflected in the corresponding performance counter metrics, which could then be utilized to detect the error. The performed FMA fault injection was expected to cause excessive early thread exit activity, resulting in decreased FMA activity. Likewise, the performed ADU fault injection was expected to alter FMA activity. Thus, it was posited that these fault injections would be detectable from FMA metrics. The results generally confirmed the hypothesis.

An important part of machine learning, and more generally data analytics, is ensuring the general applicability of learned patterns. It is not uncommon for high accuracies to be reported using one set of data, only for the same model to be unacceptably inaccurate on a new dataset. This phenomenon is called overfitting, and occurs when a pattern appears in the training dataset but is not present in general. In such a case, the model is said to have memorized the training set, rather than extracting a useful pattern for general purpose classification. The most informative metrics in this work result in SVMs and LOFs with reported 100% accuracy scores. A further analysis of these results is warranted to eliminate the possibility of overfitting.

It is often said that real data is messy. This is the case regarding Frame Buffer (FBPA) and DRAM activity, neither of which are able to inform an SVM or LOF sufficiently to surpass 70% accuracy for an SEU. This is due to the fact that these metrics are probabilistic in nature; since the CUPTI Profiling Interface provides metric functionality beginning at the roll-up level, it is not possible to isolate metric values from the targeted SM. Thus, SEUs do not disturb unit activity sufficiently to separate golden and injected distributions.

The results from repeating a fault injection, displayed in figure 5.1, support this theory. Here, the l1tex__m_xbar2l1tex_read_sectors.sum metric was selected for further analysis. Figure 5.1 plots the injected l1tex__m_xbar2l1tex_read_sectors.sum distribution for several injection conditions. For one SM injection, the golden and injected distributions nearly overlap. Consequently, machine learning models struggle to find a difference between the two classes, resulting in an accuracy of just over 50%. However, as the upset is repeated across multiple locations, the distributions continue to shift. Eventually, 100% accuracy is reached at 60 SM-level upsets per trial.

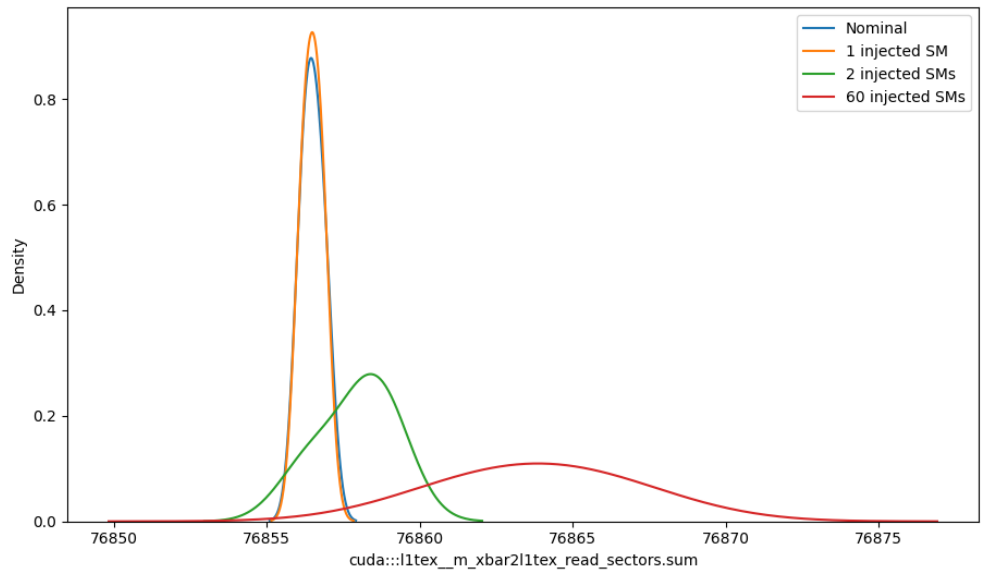


Figure 5.1 Distributions for a selection of injected SM amounts.
Note that noise of magnitude less than one was added to aid with kernel density estimation.

Table 5.3 displays the results from utilizing the metrics in table 5.2 to detect repeated upsets. Here, an injection is performed once per CTA instead of once per kernel launch. L0 i-Cache, LSU, and ALU injection campaign results reveal a significant increase in error detection accuracy. With repeated upsets, as high as 100% accuracy is achieved using metrics that were suboptimal for SEU detection. This confirms metric distribution separation to be the culprit for suboptimal results in table 5.2. Thus, a clear trend of increasing machine learning model accuracy against increased golden and injected metric distribution separation exists.

Other metrics present a deterministic nature, in which the events being counted display invariant activity across benchmark trials. Thus, any small disturbance in their activity will be discernible to a machine learning model. These are not process errors, but rather a characteristic of running this benchmark in isolation. Since the benchmark contains no variability in execution between injected trials, an unchanging amount of FMA instructions will be executed each time. Thus, the sm__pipe_fma_cycles_active.sum metric will be invariant across golden or injected trials in this situation, as shown in figure 5.2. Still other metrics are not strictly deterministic, but instead present narrow distributions that are similarly sensitive and specific to the SEUs that were injected in this work. The lts__t_requests_aperture_device_evict_normal_lookup_miss.sum metric in table 5.1 is one such metric, as shown in figure 5.3. From these observations, it is clear that the reported high accuracies in table 5.1 are due to sensitive and specific metrics, rather than overfitting.

Thus, the results have been determined to be reliable and generally applicable. In this work, the SVM was utilized to verify the existence of two distinct classes of data – golden and injected – after sampling metrics from golden and injected trials. However, as a supervised machine learning algorithm, commercial application of an SVM requires the user to inject faults into source code to generate training data. Since commercial software is often closed-source, this presents a serious limitation. Therefore, LOF performance is also examined. The LOF model was trained using noisy golden data, then tested on the entirety of the 2000 sample golden and injected dataset for each metric. The LOF also reported up to 100% accuracy. Therefore, it is possible to treat SEU detection as an anomaly detection problem by profiling closed source software and training a novelty detection model, such as LOF, on the resulting golden data. Then the model may be deployed to its intended environment.

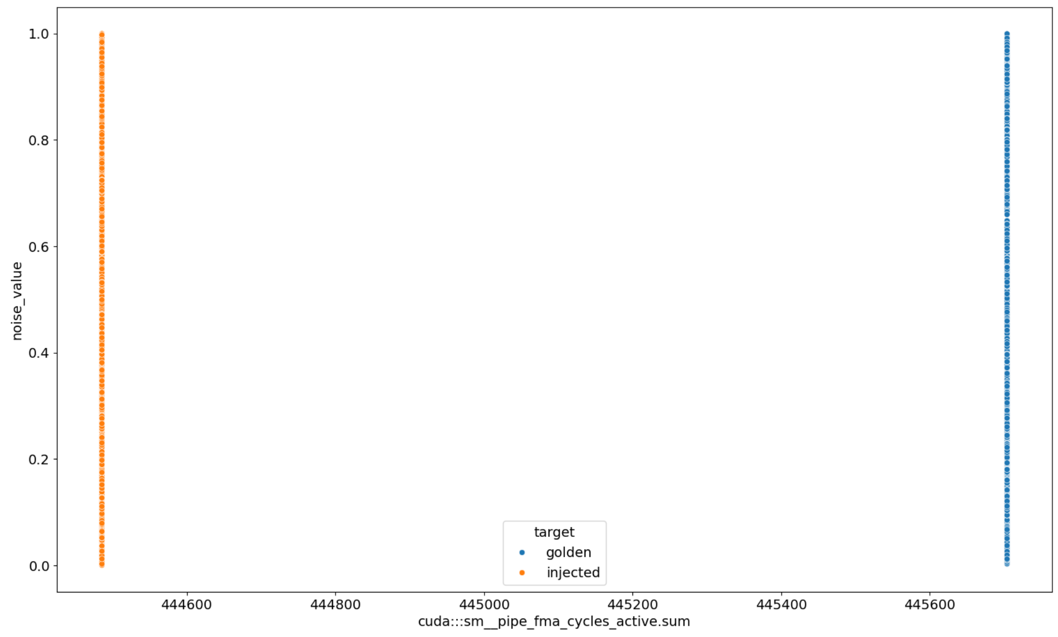


Figure 5.2 Scatterplot of sm__pipe_fma_cycles_active for FMA SEU injection. Note that a noisy y-component was added to enable two-dimensional scatterplot visualization.

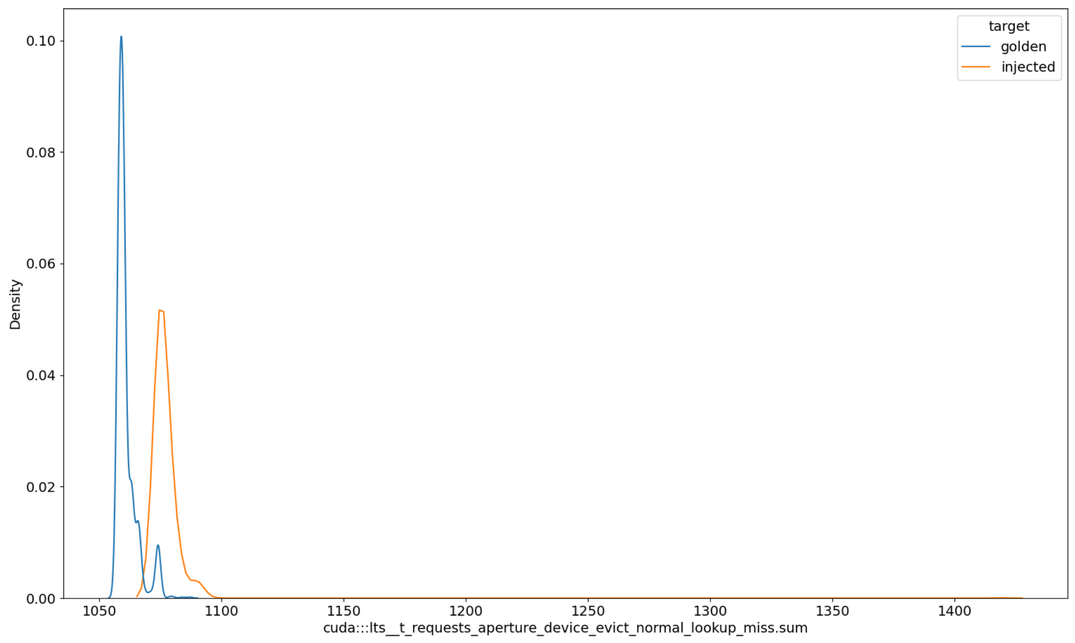


Figure 5.3 Narrow golden and injected kernel density estimated distributions for L0 i-Cache SEU injection.

Chapter 6. Conclusions and Future Work

This work investigates the feasibility of single event upset detection in a modern COTS GPU without the use of computational redundancy for the purpose of furthering space exploration capabilities. Hardware performance counters were utilized for this purpose, promising a computationally efficient solution if functional. Results confirm the feasibility of such an approach, with some metrics useful for achieving perfect accuracy.

Some metrics require a multitude of upsets for error detection due to the CUPTI Profiling Interface's lack of support for individual unit metric collection. However, the older CUPTI interface did provide such functionality. Metric collection at the unit level would inherently contain less noise and thus result in narrower distributions. These narrow distributions may be more easily separated than their roll-up counterparts. Thus, a regression test to CUPTI and a compatible older-model GPU for the detection of SEUs using these metrics would be insightful. If successful, perhaps NVIDIA could be convinced to reintroduce this functionality. Regardless, metrics capable of satisfactory error detection were found even in probabilistic cases, as shown in table 5.1.

The next step is to evaluate this algorithm under low levels of radiation. This would likely have to occur in the powered off state at first or under low flux on an unhardened chip. Furthermore, the beam would need to be tuned onto only the processing chip. Protons, heavy ions, electrons, and high energy x-rays, as a surrogate for gamma rays, could be utilized for the test. Without the presence of an atmosphere, neutrons are unlikely to threaten chips, rendering a neutron test unnecessary for deep-space applications. On the other hand, tests for applications involving commercial airline-altitude flight, self-driving cars, or other robotics for extreme environments could benefit from a neutron test.

As each metric's activity is shown to correspond to particular upsets in hardware, this approach can also likely be utilized as a diagnostic technique for progressively defective hardware, enabling longer mission duration through the progressive deactivation of SMs – a feat currently not feasible through user-available software, but likely implementable by researchers at NVIDIA. Many authors have introduced the application of progressive deactivation from the standpoint of preserving a particular computational processing ability level (Al-Allaf et al., 2023; Strollo & Trifiletti, 2016). However, lightweight upset detection combined with progressive deactivation of SMs in a GPU could result in an acceptable processing ability level for mission duration. Thus, future work could also include investigating the combination of performance counter based upset detection with progressive deactivation under radiation.

Finally, the same upsets caused by ionizing radiation can be caused by other physical phenomena. Thus, although this technique was developed for space applications, it can be extended seamlessly to other mission critical applications. Autonomous vehicles, for example, could benefit from the reallocation of resources freed from redundant calculation work. Calculations determined to be erroneous may be quickly redone. Repeated erroneous calculations could trigger a self-diagnostic test to inform progressive deactivation. Thus, performance counter based symptomatic fault detection presents interesting research across domains.

References

Baumann, R., & Kruckmeyer, K. (2017, August 18). Radiation Handbook for Electronics (rev. A) - texas instruments india. https://www.ti.com/seclit/eb/sgzy002a/sgzy002a.pdf

Di Carlo, S., Condia, J. E., & Reorda, M. S. (2020). An on-line testing technique for the Scheduler Memory of a GPGPU. *IEEE Access*, *8*, 16893–16912. https://doi.org/10.1109/access.2020.2968139

Di Carlo, S., Gambardella, G., Indaco, M., Martella, I., Prinetto, P., Rolfo, D., & Trotta, P. (2013). A software-based self test of Cuda Fermi gpus. *2013 18TH IEEE EUROPEAN TEST SYMPOSIUM (ETS)*. https://doi.org/10.1109/ets.2013.6569353

Guerrero Balaguera, J. D., Rodriguez Condia, J. E., Fernandes Dos Santos, F., Sonza Reorda, M., & Rech, P. (2023). Understanding the effects of permanent faults in GPU's parallelism management and Control Units. *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. https://doi.org/10.1145/3581784.3607086

Guerrero-Balaguera, J.-D., Condia, J. E., & Reorda, M. S. (2021). Using hardware performance counters to support infield GPU testing. *2021 28th IEEE International Conference on Electronics, Circuits, and Systems (ICECS)*. https://doi.org/10.1109/icecs53924.2021.9665511

Iturbe, X., Keymeulen, D., Ozer, E., Yiu, P., Berisford, D., Hand, K., & Carlson, R. (2015). An integrated SOC for Science Data Processing in next-generation space flight instruments avionics. *2015 IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC)*. https://doi.org/10.1109/vlsi-soc.2015.7314405

Jia, Z., Maggioni, M., Smith, J., & Paolo Scarpazza, D. (2019). Dissecting the NVidia Turing T4 GPU via Microbenchmarking. arXiv preprint arXiv:1903.07486.

Kernel Profiling Guide. NVIDIA Documentation Hub - NVIDIA Docs. (n.d.-a). https://docs.nvidia.com/nsight-compute/ProfilingGuide/index.html#metrics-structure

Kernel Profiling Guide. NVIDIA Documentation Hub - NVIDIA Docs. (n.d.-b). https://docs.nvidia.com/nsight-compute/ProfilingGuide/index.html#metrics-hw-model

Kosmidis, L., Rodriguez, I., Jover, Á., Alcaide, S., Lachaize, J., Abella, J., Notebaert, O., Cazorla, F. J., & Steenari, D. (2020). GPU4S: Embedded gpus in space - latest project updates. *Microprocessors and Microsystems*, *77*, 103143. https://doi.org/10.1016/j.micpro.2020.103143

Marinella, M. J. (2021). Radiation effects in advanced and emerging nonvolatile memories. *IEEE Transactions on Nuclear Science*, *68*(5), 546–572. https://doi.org/10.1109/tns.2021.3074139

NSRL User Guide. BNL. (n.d.). https://www.bnl.gov/nsrl/userguide/gcrsim.php

NVIDIA. (n.d.). NVIDIA AMPERE GA102 GPU ARCHITECTURE.
https://www.nvidia.com/content/PDF/nvidia-ampere-ga-102-gpu-architecture-whitepaper-v2.pdf

Oh, N., Shirvani, P. P., & McCluskey, E. J. (2002). Error detection by duplicated instructions in super-scalar processors. *IEEE Transactions on Reliability*, *51*(1), 63–75. https://doi.org/10.1109/24.994913

Sabena, D., Reorda, M. S., Sterpone, L., Rech, P., & Carro, L. (2013). On the evaluation of soft-errors detection techniques for gpgpus. *2013 8th IEEE Design and Test Symposium*. https://doi.org/10.1109/idt.2013.6727092

Santos, F. F., Pimenta, P. F., Lunardi, C., Draghetti, L., Carro, L., Kaeli, D., & Rech, P. (2019a). Analyzing and increasing the reliability of convolutional neural networks on gpus. *IEEE Transactions on Reliability*, *68*(2), 663–677. https://doi.org/10.1109/tr.2018.2878387

Santos, F. F., Pimenta, P. F., Lunardi, C., Draghetti, L., Carro, L., Kaeli, D., & Rech, P. (2019b). Analyzing and increasing the reliability of convolutional neural networks on gpus. *IEEE Transactions on Reliability*, *68*(2), 663–677. https://doi.org/10.1109/tr.2018.2878387

So, H., Didehban, M., Shrivastava, A., & Lee, K. (2019). A software-level redundant multithreading for soft/hard error detection and recovery. *2019 Design, Automation &amp; Test in Europe Conference &amp; Exhibition (DATE)*. https://doi.org/10.23919/date.2019.8715089

Strollo, E., & Trifiletti, A. (2016). A fault-tolerant real-time microcontroller with multiprocessor architecture. *2016 MIXDES - 23rd International Conference Mixed Design of Integrated Circuits and Systems*. https://doi.org/10.1109/mixdes.2016.7529781

Teijeiro, A. E. (2023, November 25). GitHub. https://github.com/aeteijeiro/ThesisWork

Tsai, T., Hari, S. K., Sullivan, M., Villa, O., & Keckler, S. W. (2021). NVBITFI: Dynamic Fault Injection for gpus. *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. https://doi.org/10.1109/dsn48987.2021.00041

Appendix

This appendix includes source files for compiling, executing, and analyzing the impact of using hardware performance counters of GPUs for symptomatic fault detection.

A.1 Makefile

```
# CONFIGURATION DIRECTIVES
# Compilers
#CC = /usr/bin/nvcc
#NVCC = /usr/bin/nvcc
CC=nvcc
NVCC=nvcc
HIP = /opt/rocm/hip/bin/hipcc
# the build target executable:
TARGET = matrix_multiplication
# FLAGS
# CC FUNCTIONS compiler flags:
CFLAGS  = -O0
# CPU compiler flags:
CPUFLAGS  = -O0
# NVCC compiler flags
NVCCFLAGS = -arch compute_86 -code sm_86 -O0
# CUDA FLAGS
CUFLAGS = -I/usr/local/cuda/include/ -L/usr/local/cuda/lib64 -lcuda -
lcudart
# OPENCL FLAGS
OPFLAGS = -I/usr/local/cuda/include/   -L/oldroot/root/usr/lib/x86_64-
linux-gnu/ -lOpenCL
# OPENMP FLAGS
OMPFLAGS = -fopenmp -lm
# HIP FLAGS
HIPFLAGS = -I/opt/rocm/hip/include -L/opt/rocm/hip/lib
# OPENBLAS INCLUDE
OBLASFLAG = -I/opt/OpenBLAS/include/ -L/opt/OpenBLAS/lib -lopenblas
# ATLAS INCLUDE
ATLASFLAG = -I/usr/local/atlas/include -L/usr/local/atlas/lib -lcblas -
latlas
# LIBRARY: ATLAS or OPENBLAS
LIBRARY=ATLAS
LIBFLAGS=$(ATLASFLAG)
# Littelendian and Bigendian flags, by default if value is not set is
Littelendian if value is set to -DBIGENDIAN is Bigendian
# -DBIGENDIAN
ENDIANFLAGS =
# Data type can be INT FLOAT HALF and DOUBLE
DATATYPE =
# By default BLOCKSIZE equals 4.
BLOCKSIZE = 4
BLOCKSIZESQUARED = $(shell echo $(BLOCKSIZE)\*$(BLOCKSIZE) | bc)
```

```
NUMEVENTS =
# FOLDERS
# CPU FUNCTIONS FOLDER
CPUFUNCTIONFOLDER = ./cpu_functions/
# CUDA FOLDER
CUFOLDER = ./cuda/
# OPENCL FOLDER
OPFOLDER = ./opencl/
# OPENMP FOLDER
OMPFOLDER = ./openmp/
# HIP FOLDER
HIPFOLDER = ./hip/
# CPU FOLDER
CPUFOLDER = ./cpu/
# OUTPUT FOLDER
OUTPUTFOLDER = ./bin/
# COMPILER MAIN
all:
      @echo "YOU NEED TO SELECT A PLATFORM -> make CPU / make CUDA / make
OpenCL / make CUDA-opt / make OpenCL-opt / make CUDA-lib / make OpenCL-
lib"
# End Main
# Shortcuts
.PHONY: all-cpu
all-cpu: cpu
.PHONY: all-cuda
all-cuda: cuda cuda-opt cuda-lib
.PHONY: all-opencl
all-opencl: opencl opencl-opt opencl-lib
.PHONY: all-openmp
all-openmp: openmp openmp-opt openmp-lib
.PHONY: all-hip
all-hip: hip hip-opt
.PHONY: CPU
CPU: cpu
.PHONY: CUDA
CUDA: cuda
.PHONY: OpenCL
OpenCL: opencl
.PHONY: OpenMP
OpenMP: openmp
.PHONY: Hip
Hip: hip
.PHONY: CUDA-opt
CUDA-opt: cuda-opt
.PHONY: OpenCL-opt
OpenCL-opt: opencl-opt
.PHONY: OpenMP-opt
OpenMP-opt: openmp-opt
.PHONY: Hip-opt
Hip-opt: hip-opt
.PHONY: CUDA-lib
CUDA-lib: cuda-lib
.PHONY: OpenCL-lib
```

```
OpenCL-lib: opencl-lib
.PHONY: OpenMP-lib
OpenMP-lib: openmp-lib
# End Shortcuts
# CPU FUNCTIONS part
cpu_functions.o: $(CPUFUNCTIONFOLDER)cpu_functions.cpp
	$(CC) $(ENDIANFLAGS) -D$(DATATYPE) -c
$(CPUFUNCTIONFOLDER)cpu_functions.cpp -o
$(CPUFUNCTIONFOLDER)cpu_functions.o $(CFLAGS)
# End CPU
# CPU part
.PHONY: cpu
cpu: main_cpu
lib_cpu.o: $(CPUFOLDER)lib_cpu.cpp
	$(CC) -D$(DATATYPE) -DBLOCK_SIZE=$(BLOCKSIZE) -c
$(CPUFOLDER)lib_cpu.cpp -o $(CPUFOLDER)lib_cpu.o $(CPUFLAGS)
main_cpu: main.cpp lib_cpu.o cpu_functions.o
	mkdir -p $(OUTPUTFOLDER)
	$(CC) -D$(DATATYPE) main.cpp $(CPUFOLDER)lib_cpu.o
$(CPUFUNCTIONFOLDER)cpu_functions.o -o
$(OUTPUTFOLDER)$(TARGET)_cpu_$(shell echo $(DATATYPE) | tr A-Z a-
z)_$(BLOCKSIZESQUARED) $(CPUFLAGS) $(CFLAGS)
# End CPU
# CUDA part
.PHONY: cuda
cuda: main_cuda
lib_cuda.o:
	rm $(CUFOLDER)lib_cuda.o
	mv build/lib_cuda.o $(CUFOLDER)
pre-ptx:
	cd build;./pre_ptx.sh
pre-ptx-golden:
	cd build;./pre_ptx.sh
	cd utils; python3 inject_errors.py $(HPC_TYPE) $(SM_COUNT) golden;
mv lib_cuda.ptx ../build/
post-ptx:
	cd build;./post_ptx.sh
inj-ptx: pre-ptx
	cd utils; python3 inject_errors.py $(HPC_TYPE) $(SM_COUNT) injected;
mv lib_cuda.ptx ../build/
injected_errors: inj-ptx post-ptx main_cuda
golden: pre-ptx-golden post-ptx main_cuda
dry-run:
	cp cuda/* build/
	cd build; $(NVCC) -dryrun --keep -D$(DATATYPE) -
DBLOCK_SIZE=$(BLOCKSIZE) -DCUDA -c lib_cuda.cu -o lib_cuda.o $(NVCCFLAGS)
tests:
	cd utils; echo "Makefile EVENTS_PER_FILE=$(EVENTS_PER_FILE)
TOTAL_EVENTS=$(TOTAL_EVENTS)"; ./run_both_tests_shortlist.sh
$(EVENTS_PER_FILE) $(TOTAL_EVENTS) $(SAMPLES_PER_EVENT)
$(TARGET_INSTRUCTION) $(STARTING_POINT); python3 organize_data.py
$(HPC_TYPE) $(TARGET_INSTRUCTION) $(SM_COUNT) $(SAMPLES_PER_EVENT);
python3 analysis.py no scatter svm $(HPC_TYPE) $(TARGET_INSTRUCTION)
$(SM_COUNT) $(SAMPLES_PER_EVENT)
```

```
clean_runs:
	rm -f bin/golden_runs/profiler/* bin/injector_runs/profiler/*
bin/golden_runs/benchmark/* bin/injector_runs/benchmark/*
utils/final_dataset_$(HPC_TYPE)_$(TARGET_INSTRUCTION).parquet
main_cuda: main.cpp lib_cuda.o cpu_functions.o
	mkdir -p $(OUTPUTFOLDER)
	$(CC) -g -G -D$(DATATYPE) -DCUDA -DPAPI main.cpp
$(CUFOLDER)lib_cuda.o $(CPUFUNCTIONFOLDER)cpu_functions.o -o
$(OUTPUTFOLDER)$(TARGET)_cuda_$(shell echo $(DATATYPE) | tr A-Z a-
z)_$(BLOCKSIZESQUARED) $(CUFLAGS) $(CFLAGS) -lstdc++ -
L/home/antonio/Documents/PerFI_PAPI_wd/PAPI/papi/src/install/lib -lpapi -
L/home/antonio/Documents/PerFI_PAPI_wd/PAPI/papi/src/testlib -ltestlib
# End CUDA
# OpenCL Part
opencl:  main_opencl
lib_opencl.o: $(OPFOLDER)lib_opencl.cpp
	$(CC) -D$(DATATYPE) -DBLOCK_SIZE=$(BLOCKSIZE) -DOPENCL -c
$(OPFOLDER)lib_opencl.cpp -o $(OPFOLDER)lib_opencl.o $(CFLAGS) $(OPFLAGS)
main_opencl: main.cpp lib_opencl.o cpu_functions.o
	mkdir -p $(OUTPUTFOLDER)
	$(CC) -D$(DATATYPE) -DOPENCL main.cpp $(OPFOLDER)lib_opencl.o
$(CPUFUNCTIONFOLDER)cpu_functions.o -o
$(OUTPUTFOLDER)$(TARGET)_opencl_$(shell echo $(DATATYPE) | tr A-Z a-
z)_$(BLOCKSIZESQUARED) $(CFLAGS) $(OPFLAGS)
# End OpenCL
# OpenMP Part
openmp:  main_openmp
lib_omp.o: $(OMPFOLDER)lib_omp.cpp
	export OMP_NUM_THREADS=8
	$(CC) -D$(DATATYPE) -DBLOCK_SIZE=$(BLOCKSIZE) -DOPENMP -c
$(OMPFOLDER)lib_omp.cpp -o $(OMPFOLDER)lib_omp.o $(CFLAGS) $(OMPFLAGS)
main_openmp: main.cpp lib_omp.o cpu_functions.o
	mkdir -p $(OUTPUTFOLDER)
	$(CC) -D$(DATATYPE) -DOPENMP main.cpp $(OMPFOLDER)lib_omp.o
$(CPUFUNCTIONFOLDER)cpu_functions.o -o
$(OUTPUTFOLDER)$(TARGET)_omp_$(shell echo $(DATATYPE) | tr A-Z a-z)
$(CFLAGS) $(OMPFLAGS)
# End OpenMP
# Hip part
hip: main_hip
lib_hip.o: $(HIPFOLDER)lib_hip.cpp
	$(HIP) -D$(DATATYPE) -DBLOCK_SIZE=$(BLOCKSIZE) -DHIP -c
$(HIPFOLDER)lib_hip.cpp -o $(HIPFOLDER)lib_hip.o $(CFLAGS) $(HIPFLAGS)
main_hip: main.cpp lib_hip.o cpu_functions.o
	mkdir -p $(OUTPUTFOLDER)
	$(HIP) -D$(DATATYPE) -DHIP main.cpp -x none $(HIPFOLDER)lib_hip.o
$(CPUFUNCTIONFOLDER)cpu_functions.o -o
$(OUTPUTFOLDER)$(TARGET)_hip_$(shell echo $(DATATYPE) | tr A-Z a-
z)_$(BLOCKSIZESQUARED) $(CFLAGS) $(HIPFLAGS)
# End Hip
# CUDA part optimized
.PHONY: cuda
cuda-opt: main_cuda_opt
lib_cuda_opt.o: $(CUFOLDER)lib_cuda_opt.cu
```

```
      $(NVCC) -D$(DATATYPE) -DBLOCK_SIZE=$(BLOCKSIZE) -DPAPI -DCUDA -c
$(CUFOLDER)lib_cuda_opt.cu -o $(CUFOLDER)lib_cuda_opt.o $(NVCCFLAGS)
main_cuda_opt: main.cpp lib_cuda_opt.o cpu_functions.o
      mkdir -p $(OUTPUTFOLDER)
      $(CC) -D$(DATATYPE) -DCUDA main.cpp $(CUFOLDER)lib_cuda_opt.o
$(CPUFUNCTIONFOLDER)cpu_functions.o -o
$(OUTPUTFOLDER)$(TARGET)_cuda_opt_$(shell echo $(DATATYPE) | tr A-Z a-
z)_$(BLOCKSIZESQUARED) $(CUFLAGS) $(CFLAGS) -lstdc++
# End CUDA optimized
# OpenCL Part optimized
opencl-opt:  main_opencl_opt
lib_opencl_opt.o: $(OPFOLDER)lib_opencl_opt.cpp
      $(CC) -D$(DATATYPE) -DBLOCK_SIZE=$(BLOCKSIZE) -DOPENCL -c
$(OPFOLDER)lib_opencl_opt.cpp -o $(OPFOLDER)lib_opencl_opt.o $(CFLAGS)
$(OPFLAGS)
main_opencl_opt: main.cpp lib_opencl_opt.o cpu_functions.o
      mkdir -p $(OUTPUTFOLDER)
      $(CC) -D$(DATATYPE) -DOPENCL main.cpp $(OPFOLDER)lib_opencl_opt.o
$(CPUFUNCTIONFOLDER)cpu_functions.o -o
$(OUTPUTFOLDER)$(TARGET)_opencl_opt_$(shell echo $(DATATYPE) | tr A-Z a-
z)_$(BLOCKSIZESQUARED) $(CFLAGS) $(OPFLAGS)
# End OpenCL optimized
# OpenMP Part optimized
openmp-opt:  main_openmp_opt
lib_omp_opt.o: $(OMPFOLDER)lib_omp_opt.cpp
      export OMP_NUM_THREADS=8
      $(CC) -D$(DATATYPE) -DBLOCK_SIZE=$(BLOCKSIZE) -DOPENMP -c
$(OMPFOLDER)lib_omp_opt.cpp -o $(OMPFOLDER)lib_omp_opt.o $(CFLAGS)
$(OMPFLAGS)
main_openmp_opt: main.cpp lib_omp_opt.o cpu_functions.o
      mkdir -p $(OUTPUTFOLDER)
      $(CC) -D$(DATATYPE) -DOPENMP main.cpp $(OMPFOLDER)lib_omp_opt.o
$(CPUFUNCTIONFOLDER)cpu_functions.o -o
$(OUTPUTFOLDER)$(TARGET)_omp_opt_$(shell echo $(DATATYPE) | tr A-Z a-z)
$(CFLAGS) $(OMPFLAGS)
# End OpenMP optimized
# Hip part
hip-opt: main_hip_opt
lib_hip_opt.o: $(HIPFOLDER)lib_hip_opt.cpp
      $(HIP) -D$(DATATYPE) -DBLOCK_SIZE=$(BLOCKSIZE) -DHIP -c
$(HIPFOLDER)lib_hip_opt.cpp -o $(HIPFOLDER)lib_hip_opt.o $(CFLAGS)
$(HIPFLAGS)
main_hip_opt: main.cpp lib_hip_opt.o cpu_functions.o
      mkdir -p $(OUTPUTFOLDER)
      $(HIP) -D$(DATATYPE) -DHIP main.cpp -x none
$(HIPFOLDER)lib_hip_opt.o  $(CPUFUNCTIONFOLDER)cpu_functions.o -o
$(OUTPUTFOLDER)$(TARGET)_hip_opt_$(shell echo $(DATATYPE) | tr A-Z a-
z)_$(BLOCKSIZESQUARED) $(CFLAGS) $(HIPFLAGS)
# End Hip
# CUDA part library
.PHONY: cuda
cuda-lib: main_cuda_lib
lib_cuda_lib.o: $(CUFOLDER)lib_cuda_lib.cu
```

```
      $(NVCC) -D$(DATATYPE) -DCUDA -c $(CUFOLDER)lib_cuda_lib.cu -o
$(CUFOLDER)lib_cuda_lib.o $(NVCCFLAGS)
main_cuda_lib: main.cpp lib_cuda_lib.o cpu_functions.o
      mkdir -p $(OUTPUTFOLDER)
      $(CC) -D$(DATATYPE) -DCUDA main.cpp $(CUFOLDER)lib_cuda_lib.o
$(CPUFUNCTIONFOLDER)cpu_functions.o -o
$(OUTPUTFOLDER)$(TARGET)_cuda_lib_$(shell echo $(DATATYPE) | tr A-Z a-z)
$(CUFLAGS) $(CFLAGS) -lstdc++ -lcublas
# End CUDA library
# OpenCL Part library
opencl-lib:  main_opencl_lib
lib_opencl_lib.o: $(OPFOLDER)lib_opencl_lib.cpp
      $(CC) -D$(DATATYPE) -DOPENCL -c $(OPFOLDER)lib_opencl_lib.cpp -o
$(OPFOLDER)lib_opencl_lib.o $(CFLAGS) $(OPFLAGS) -
I/home/irodrig/clBlast/include/ -L/home/irodrig/clBlast/lib/ -lclblast
main_opencl_lib: main.cpp lib_opencl_lib.o cpu_functions.o
      mkdir -p $(OUTPUTFOLDER)
      $(CC) -D$(DATATYPE) -DOPENCL main.cpp $(OPFOLDER)lib_opencl_lib.o
$(CPUFUNCTIONFOLDER)cpu_functions.o -o
$(OUTPUTFOLDER)$(TARGET)_opencl_lib_$(shell echo $(DATATYPE) | tr A-Z a-z)
$(CFLAGS) $(OPFLAGS)  -I/home/irodrig/clBlast/include/ -
L/home/irodrig/clBlast/lib/ -lclblast
# End OpenCL library
# OpenMP Part library
openmp-lib:  main_openmp_lib
lib_omp_lib.o: $(OMPFOLDER)lib_omp_lib.cpp
      $(CC) -D$(DATATYPE) -D$(LIBRARY) -DBLOCK_SIZE=$(BLOCKSIZE) -DOPENMP
-c $(OMPFOLDER)lib_omp_lib.cpp -o $(OMPFOLDER)lib_omp_lib.o $(CFLAGS)
$(OMPFLAGS) $(LIBFLAGS)
main_openmp_lib: main.cpp lib_omp_lib.o cpu_functions.o
      mkdir -p $(OUTPUTFOLDER)
      $(CC) -D$(DATATYPE) -D$(LIBRARY) -DOPENMP main.cpp
$(OMPFOLDER)lib_omp_lib.o  $(CPUFUNCTIONFOLDER)cpu_functions.o -o
$(OUTPUTFOLDER)$(TARGET)_omp_lib_$(shell echo $(DATATYPE) | tr A-Z a-z)
$(CFLAGS) $(OMPFLAGS) $(LIBFLAGS)
# End OpenMP library
# Clean
.PHONY: clean
clean:
      rm -rf *.o
      rm -rf $(CPUFUNCTIONFOLDER)*.o
      rm -rf $(OPFOLDER)*.o
      rm -rf $(OMPFOLDER)*.o
      rm -rf $(HIPFOLDER)*.o
      rm -rf $(CUFOLDER)*.o
      rm -rf $(CPUFOLDER)*.o
      rm -rf $(OUTPUTFOLDER)$(TARGET)_*
```

A.2 INJECT_ERRORS.PY

```
import sys

def add_regs(newfile,newRegsAdded,line):
    if ".reg " in line:
        print(f"in Reg")
        newfile.write(line)
        newline="    .reg .pred      mycondition;"
        newline+="\n    .reg .u32       mystoreSMID;"
        newline+="\n    .reg .u32       mystorelaneID;"
        newline+="\n    .reg .u32       mystoreCTAx;"
        newline+="\n    .reg .u32       mystoreCTAy;"
        newline+="\n    .reg .u32       mystorewarpID;\n"
        print(newline)
        newfile.write(newline)
        newRegsAdded=1
    return newRegsAdded

def inject_predicates(newfile, idsMoved,line,sm_count):
    if (('@' in line) and (idsMoved==0)):
        newline="        mov.u32         mystoreCTAx, %ctaid.x;"
        newline+="\n        mov.u32         mystoreCTAy, %ctaid.y;"
        newline+="\n       mov.u32         mystorewarpID, %warpid;"

        newline+="\n       setp.lt.u32     mycondition,
mystoreCTAx,"+sm_count+";"
        if GOLDEN:
            newline+="\n@mycondition setp.eq.u32     mycondition,
mystoreCTAy,0;"
            newline+="\n@mycondition setp.eq.u32     mycondition,
mystorewarpID,100;"
        else:
            newline+="\n@mycondition setp.eq.u32     mycondition,
mystoreCTAy,0;"
            newline+="\n@mycondition setp.eq.u32     mycondition,
mystorewarpID,0;"
        predicate_name=line.split(" ")[0].strip().strip('@')
        newline+="\n@mycondition xor.pred" + "          " +
predicate_name.strip('\t') + ", " + predicate_name.strip('\t') + ", 1;\n"
        print(newline)
        newfile.write(newline)
        idsMoved=1
    newfile.write(line)
    return idsMoved


def inject_L0(newfile, idsMoved,line,sm_count):
    if (('st.global.u32' in line) and (idsMoved==0)):
        print('in idsMoved')
        newline="        mov.u32         mystoreCTAx, %ctaid.x;"
        newline+="\n        mov.u32         mystoreCTAy, %ctaid.y;"
        newline+="\n       mov.u32         mystorewarpID, %warpid;"
        newline+="\n       setp.lt.u32     mycondition,
mystoreCTAx,"+sm_count+";"
        newline+="\n@mycondition setp.lt.u32     mycondition,
mystoreCTAy,"+sm_count+";"
        if GOLDEN:
            newline+="\n@mycondition setp.eq.u32     mycondition,
mystorewarpID,1000;"
        else:
            newline+="\n@mycondition setp.eq.u32     mycondition,
mystorewarpID,0;"

        target_variable=line.split('[')[1].split(']')[0]
        if idsMoved==0:
            newline+="\n@mycondition    xor.b64" + "         " +
target_variable + ", " + target_variable + ", 1048576;\n"
        print(newline)
        newfile.write(newline)
        idsMoved=1
    newfile.write(line)
    return idsMoved

def inject_LSU(newfile, idsMoved,line,sm_count):
    if (('st.global.u32' in line) and (idsMoved==0)):
        print('in idsMoved')
        newline="       mov.u32         mystoreCTAx, %ctaid.x;"
        newline+="\n        mov.u32         mystoreCTAy, %ctaid.y;"
        newline+="\n       mov.u32         mystorewarpID, %warpid;"
        newline+="\n       mov.u32         mystorelaneID, %laneid;"

        newline+="\n       setp.lt.u32     mycondition,
mystoreCTAx,"+sm_count+";"
        newline+="\n@mycondition setp.lt.u32     mycondition,
mystoreCTAy,"+sm_count+";"
        newline+="\n@mycondition setp.eq.u32     mycondition,
mystorewarpID,0;"
        if GOLDEN:
            newline+="\n@mycondition setp.lt.u32     mycondition,
mystorelaneID,0;"
        else:
            newline+="\n@mycondition setp.lt.u32     mycondition,
mystorelaneID,8;"

        target_variable=line.split('[')[1].split(']')[0]
        if idsMoved==0:
            newline+="\n@mycondition    xor.b64" + "         " +
target_variable + ", " + target_variable + ", 1048576;\n"
        print(newline)
        newfile.write(newline)
        idsMoved=1
    newfile.write(line)
    return idsMoved

def inject_ALU(newfile, idsMoved,line,sm_count):
    if (('st.global.u32' in line) and (idsMoved==0)):
        print('in idsMoved')
        newline="        mov.u32         mystoreCTAx, %ctaid.x;"
        newline+="\n        mov.u32         mystoreCTAy, %ctaid.y;"
        newline+="\n       mov.u32         mystorewarpID, %warpid;"
        newline+="\n       mov.u32         mystorelaneID, %laneid;"

        newline+="\n       setp.lt.u32     mycondition, 
mystoreCTAx,"+sm_count+";"
        newline+="\n@mycondition setp.lt.u32     mycondition, 
mystoreCTAy,"+sm_count+";"
        newline+="\n@mycondition setp.eq.u32     mycondition, 
mystorewarpID,0;"
        if GOLDEN:
            newline+="\n@mycondition setp.eq.u32     mycondition, 
mystorelaneID,1000;"
        else:
            newline+="\n@mycondition setp.eq.u32     mycondition, 
mystorelaneID,0;"

        target_variable=line.split('[')[1].split(']')[0]
        if idsMoved==0:
            newline+="\n@mycondition    xor.b64" + "          " + 
target_variable + ", " + target_variable + ", 1048576;\n"
        print(newline)
        newfile.write(newline)
        idsMoved=1
    newfile.write(line)
    return idsMoved

def inject_fma(newfile, idsMoved,line,sm_count):
    if 'mad.lo.s32' in line:
        newfile.write(line)
        if(idsMoved==0):
            print('in idsMoved')
            newline="        mov.u32         mystoreCTAx, %ctaid.x;"
            newline+="\n        mov.u32         mystoreCTAy, %ctaid.y;"
            newline+="\n       mov.u32         mystorewarpID, %warpid;"
            newline+="\n       setp.lt.u32     mycondition, 
mystoreCTAx,"+sm_count+";"
            if GOLDEN:
                newline+="\n@mycondition setp.eq.u32     mycondition, 
mystoreCTAy,0;"
                newline+="\n@mycondition setp.eq.u32     mycondition, 
mystorewarpID,100;"
            else:
                newline+="\n@mycondition setp.eq.u32     mycondition, 
mystoreCTAy,0;"
                newline+="\n@mycondition setp.eq.u32     mycondition, 
mystorewarpID,0;"

            target_variable=line.split(',')[0].split(' ')[-1].strip()
            newline+="\n@mycondition     add.s32" + "          " + 
target_variable + ", " + target_variable + ", 100000;\n"
            print(newline)
            newfile.write(newline);
            idsMoved=1
    else:
        newfile.write(line)
    return idsMoved

with open("lib_cuda.ptx",'w') as newfile:
    with open("../build/lib_cuda.ptx",'r') as originalfile:
        f=originalfile.readlines()
        newRegsAdded=0
        idsMoved=0;
        target=sys.argv[1]+'.'
        SM_COUNT=sys.argv[2]
        RUN_TYPE=sys.argv[3]
        if RUN_TYPE=='golden':
            GOLDEN=1
        elif RUN_TYPE=='injected':
            GOLDEN=0
        else:
            raise Exception("RUN_TYPE not set properly")

        if len(target)==0:
            print("ERROR: no target passed"); input()
        for line in f:
            if ((".reg " in line) and (newRegsAdded==0)):
                newRegsAdded=add_regs(newfile, newRegsAdded, line)
            elif target=='predicates.':
                idsMoved=inject_predicates(newfile,
idsMoved,line,SM_COUNT)

            elif target=='LSU.':
                idsMoved=inject_LSU(newfile, idsMoved,line,SM_COUNT)

            elif target=='L0.':
                idsMoved=inject_L0(newfile, idsMoved,line,SM_COUNT)

            elif target=='ALU.':
                idsMoved=inject_ALU(newfile, idsMoved,line,SM_COUNT)

            elif target=='fma.':
                idsMoved=inject_fma(newfile, idsMoved,line,SM_COUNT)
            else:
                idsMoved=inject_other(newfile, idsMoved,line,target)
```

A.3 ANALYSIS.PY

```
import seaborn as sns
import polars as pl
import pandas as pd
import sys
from sklearn import svm
from sklearn.model_selection import cross_val_score
from sklearn.utils import shuffle
from sklearn.neighbors import LocalOutlierFactor
from sklearn.ensemble import IsolationForest
import matplotlib.pyplot as plt
import numpy as np

def print_separator(message):
    print(f"####################\n {message}\n####################")

def remove_outliers(DataFrame):
    df_local=DataFrame
    df_np=df_local.drop('target').to_numpy()
    #noise_clf=LocalOutlierFactor(n_neighbors=2,algorithm='brute')
    noise_clf=IsolationForest(n_estimators=200)
    inline=noise_clf.fit_predict(df_np)
    inliers=[idx for idx,res in enumerate(inline) if res==1]
    df_local=df_local.with_row_count("row_number")

df_local=df_local.filter(pl.col('row_number').is_in(pl.Series("inliers",in
liers)))
    #print(df_local); input()
    return df_local.drop("row_number")

if len(sys.argv)!=8:
    raise Exception("must supply VERBOSITY, GRAPHICS, ML, HPC_TYPE,
TARGET_INSTRUCTION, SM_COUNT, NUM_RUNS arguments")

VERBOSITY=sys.argv[1]; GRAPHICS=sys.argv[2]; ML=sys.argv[3];
HPC_TYPE=sys.argv[4];
TARGET_INSTRUCTION=sys.argv[5];SM_COUNT=sys.argv[6];NUM_RUNS=sys.argv[7]

if VERBOSITY=='verbose':
    VERBOSE=True
else:
    VERBOSE=False

if GRAPHICS=='distribution':
    DISTPLOT=True
    SCATTERPLOT=False
elif GRAPHICS=='scatter':
    SCATTERPLOT=True
    DISTPLOT=False
elif GRAPHICS=='both':
    DISTPLOT=True
    SCATTERPLOT=True
else:
    DISTPLOT=False
    SCATTERPLOT=False

if ML=='svm':
    SVM=True
    LOF=False
elif ML=='LOF':
    SVM=False
    LOF=True
else:
    SVM=False
    LOF=False

#initial processing of original dataset
#print_separator('Original Dataset')
df =
pl.read_parquet(f"results/final_dataset_{HPC_TYPE}_{TARGET_INSTRUCTION}_{S
M_COUNT}_{NUM_RUNS}.parquet")
df_golden=df.filter(pl.col('target')=='golden')
df_injected=df.filter(pl.col('target')=='injected')
#print(df)


#remove all columns with 0 standard deviation, since they probably aren't
telling me anything anyways.
#print_separator('Reduced Dataset')
for col in df.columns:
    #print(df[col]);print(df[col].std());input("waiting")
    if df[col].std()==0:
        df=df.drop(col)

#print(df)

df_pd=df.to_pandas()
if DISTPLOT:
    for col in df.columns:
        if col != 'target':
            sns.kdeplot(data=df_pd,x=col,hue='target')
            plt.show()

if VERBOSE:
    for col in df_mean.columns:
        print_separator()
        print("column name: "+col)
        print(df_mean[col])
        print_separator()
        input()

if SCATTERPLOT:
    for col in df.columns:
        if col != 'target':

noise_column=pd.DataFrame(data=np.random.rand(df_pd.shape[0],1),columns=["
noise_value"])
```

```
            df_pd_with_noise = pd.concat([df_pd,noise_column],axis=1)
            print(df_pd_with_noise)

sns.scatterplot(data=df_pd_with_noise,x=col,y='noise_value',hue='target')
            plt.show()


if SVM:
    for col in df.columns:
        if col != 'target':
            X=df.select(col).to_pandas()
            y=df.select('target')
            y=np.ravel(y)
            X_shuffled,y_shuffled=shuffle(X,y)#,random_state=42)
            clf=svm.SVC()
            scores=cross_val_score(clf,X_shuffled,y_shuffled,cv=3)
            print(f"{HPC_TYPE} {TARGET_INSTRUCTION} 3-CV accuracy:
{scores}")
            #print(f"{col}, {HPC_TYPE}, {TARGET_INSTRUCTION} scores:
{scores}")

if LOF:
    df_pl_with_noise =
(df_golden.select(df_golden.columns[0])+pl.from_numpy(np.random.uniform(-
5,5,(df_golden.shape[0],1)))).with_columns(df_golden['target'])#.rename({'
column_0':'target'})
    X_shuffled,y_shuffled =
shuffle(df_pl_with_noise.drop('target'),df_pl_with_noise.select('target'))
    noise_clf=LocalOutlierFactor(novelty=True,n_neighbors=60)
    noise_clf.fit(X_shuffled)

    apply=pl.DataFrame(noise_clf.predict(X_shuffled.to_pandas()))
    result=X_shuffled.with_columns(apply)
    result=result.with_columns(y_shuffled['target'])

    zeros = pl.from_numpy(np.zeros((df.shape[0],1)))

apply=pl.DataFrame(noise_clf.predict(df.select(df.columns[0]).drop('target
')))
    classified_df=df.with_columns(apply)

    correct=0
    for row in classified_df.rows():
        if 'golden' in row:
            if row[row.index('golden')+1]==1:
                correct+=1
        elif 'injected' in row:
            if row[row.index('injected')+1]==-1:
                correct+=1
    print(f"{HPC_TYPE} {TARGET_INSTRUCTION} LOF accuracy:
{(correct/df.shape[0])*100}%")
```

A.4 CROSS_ANALYSIS.PY

```
import seaborn as sns
import polars as pl
import pandas as pd
import sys
from sklearn import svm
from sklearn.model_selection import cross_val_score
from sklearn.utils import shuffle
from sklearn.neighbors import LocalOutlierFactor
from sklearn.ensemble import IsolationForest
import matplotlib.pyplot as plt
import numpy as np



VERBOSITY='no'
GRAPHICS='distribution'
SVM='no'
HPC_TYPE='l1tex'
TARGET_INSTRUCTION='store'
NUM_RUNS=100
folder='l1tex'




def print_separator(message):
    print(f"####################\n {message}\n####################")

def remove_outliers(DataFrame):
    df_local=DataFrame
    df_np=df_local.drop('target').to_numpy()
    #noise_clf=LocalOutlierFactor(n_neighbors=2,algorithm='brute')
    noise_clf=IsolationForest(n_estimators=200)
    inline=noise_clf.fit_predict(df_np)
    inliers=[idx for idx,res in enumerate(inline) if res==1]
    df_local=df_local.with_row_count("row_number")

df_local=df_local.filter(pl.col('row_number').is_in(pl.Series("inliers",in
liers)))
    return df_local.drop("row_number")


if VERBOSITY=='verbose':
    VERBOSE=True
else:
    VERBOSE=False

if GRAPHICS=='distribution':
    DISTPLOT=True
    SCATTERPLOT=False
elif GRAPHICS=='scatter':
    SCATTERPLOT=True
    DISTPLOT=False
elif GRAPHICS=='both':
    DISTPLOT=True
    SCATTERPLOT=True
else:
    DISTPLOT=False
    SCATTERPLOT=False

if SVM=='svm':
    SVM=True
else:
    SVM=False



num_affected_SMs_list=[1,2,60]
df = pl.DataFrame()

for count in num_affected_SMs_list:
    df_part =
pl.read_parquet(f"{folder}/final_dataset_{HPC_TYPE}_{TARGET_INSTRUCTION}_{
count}_{NUM_RUNS}.parquet")
    sm_count_list = np.zeros(df_part.shape[0]).tolist()
    for idx,value in enumerate(sm_count_list):
        sm_count_list[idx]=count
    df_part=df_part.with_columns(pl.Series('sm_count',sm_count_list))
    ########
    # Add noise for KDE
    ########


noise_df=pl.DataFrame(np.random.rand(df_part.shape[0],df_part.shape[1]-
2))#,1)})

#testing=df_part[col]+pl.DataFrame({"noise_value":np.random.rand(df_part.s
hape[0],1)})
        #input(noise_df.head())

df_part=(df_part.drop(['target','sm_count'])+noise_df).with_columns([df_pa
rt['target'],df_part['sm_count']])

    ########

    df=pl.concat([df,df_part])
df_golden=df.filter(pl.col('target')=='golden')
df_injected=df.filter(pl.col('target')=='injected')
print(df)

for col in df.columns:
    if col=='target':
        continue
    if col=='sm_count':
        continue
    sns.kdeplot(df_golden.filter(pl.col('sm_count')==count),x=col,bw=1)
    for count in num_affected_SMs_list:

sns.kdeplot(data=df_injected.filter(pl.col('sm_count')==count),x=col,bw=1)

    plt.legend(['Nominal','1 injected SM','2 injected SMs','60 injected
SMs'])
    plt.show()
import polars as pl
import pandas as pd
import sys
from sklearn import svm
from sklearn.model_selection import cross_val_score
from sklearn.utils import shuffle
from sklearn.neighbors import LocalOutlierFactor
from sklearn.ensemble import IsolationForest
import matplotlib.pyplot as plt
import numpy as np



VERBOSITY='no'
GRAPHICS='distribution'
SVM='no'
HPC_TYPE='l1tex'
TARGET_INSTRUCTION='store'
NUM_RUNS=100
folder='l1tex'




def print_separator(message):
    print(f"####################\n {message}\n####################")

def remove_outliers(DataFrame):
    df_local=DataFrame
    df_np=df_local.drop('target').to_numpy()
    #noise_clf=LocalOutlierFactor(n_neighbors=2,algorithm='brute')
    noise_clf=IsolationForest(n_estimators=200)
    inline=noise_clf.fit_predict(df_np)
    inliers=[idx for idx,res in enumerate(inline) if res==1]
    df_local=df_local.with_row_count("row_number")

df_local=df_local.filter(pl.col('row_number').is_in(pl.Series("inliers",in
liers)))
    return df_local.drop("row_number")


if VERBOSITY=='verbose':
    VERBOSE=True
else:
    VERBOSE=False
```

```
if GRAPHICS=='distribution':
    DISTPLOT=True
    SCATTERPLOT=False
elif GRAPHICS=='scatter':
    SCATTERPLOT=True
    DISTPLOT=False
elif GRAPHICS=='both':
    DISTPLOT=True
    SCATTERPLOT=True
else:
    DISTPLOT=False
    SCATTERPLOT=False

if SVM=='svm':
    SVM=True
else:
    SVM=False



num_affected_SMs_list=[1,2,60]
df = pl.DataFrame()

for count in num_affected_SMs_list:
    df_part =
pl.read_parquet(f"{folder}/final_dataset_{HPC_TYPE}_{TARGET_INSTRUCTION}_{
count}_{NUM_RUNS}.parquet")
    sm_count_list = np.zeros(df_part.shape[0]).tolist()
    for idx,value in enumerate(sm_count_list):
        sm_count_list[idx]=count
    df_part=df_part.with_columns(pl.Series('sm_count',sm_count_list))
    ########
    # Add noise for KDE
    ########


noise_df=pl.DataFrame(np.random.rand(df_part.shape[0],df_part.shape[1]-
2))#,1)})

#testing=df_part[col]+pl.DataFrame({"noise_value":np.random.rand(df_part.s
hape[0],1)})
        #input(noise_df.head())

df_part=(df_part.drop(['target','sm_count'])+noise_df).with_columns([df_pa
rt['target'],df_part['sm_count']])

    ########

    df=pl.concat([df,df_part])
df_golden=df.filter(pl.col('target')=='golden')
df_injected=df.filter(pl.col('target')=='injected')
print(df)

for col in df.columns:
    if col=='target':
        continue
    if col=='sm_count':
        continue
    sns.kdeplot(df_golden.filter(pl.col('sm_count')==count),x=col,bw=1)

    for count in num_affected_SMs_list:

sns.kdeplot(data=df_injected.filter(pl.col('sm_count')==count),x=col,bw=1)

    plt.legend(['Nominal','1 injected SM','2 injected SMs','60 injected
SMs'])
    plt.show()
```

A.5 ORGANIZE_DATA.PY

```
import os
import polars as pl
import sys
HPC_TYPE=sys.argv[1]
TARGET_INSTRUCTION=sys.argv[2]
SM_COUNT=sys.argv[3]
NUM_RUNS=sys.argv[4]
directory = '../bin/golden_runs/profiler'
df = pl.DataFrame()
for filename in os.listdir(directory):
    if filename.endswith('.out'):
        with open(os.path.join(directory, filename)) as f:
            print(filename)
            lines=f.readlines()
            counts = {}
            for line in lines:
                if 'stop' in line:
                    count=int(line.split(":")[1].split('=')[0])
                    label="cuda:::"+line.split(":::")[1].split(':')[0]
                    counts[label]=count
            df=pl.concat([df,pl.DataFrame(counts)])

df_golden=df.with_columns(pl.lit("golden").alias("target"))
directory = '../bin/injector_runs/profiler'
df = pl.DataFrame()
for filename in os.listdir(directory):
    if filename.endswith('.out'):
        with open(os.path.join(directory, filename)) as f:
            print(filename)
            lines=f.readlines()
            counts = {}
            for line in lines:
                if 'stop' in line:
                    count=int(line.split(":")[1].split('=')[0])
                    label="cuda:::"+line.split(":::")[1].split(':')[0]
                    counts[label]=count #**4


            df=pl.concat([df,pl.DataFrame(counts)])

df_injected=df.with_columns(pl.lit("injected").alias("target"))
df_final=pl.concat([df_golden,df_injected])
df_final.write_parquet(f"results/final_dataset_{HPC_TYPE}_{TARGET_INSTRUCT
ION}_{SM_COUNT}_{NUM_RUNS}.parquet")
```

A.6 EXTRACT_SHORTLIST_EVENTS.PY

```
import sys
import os
EVENTS_PER_FILE=int(sys.argv[1])
if sys.argv[2]=='all':
    TOTAL_EVENTS='all'
else:
    TOTAL_EVENTS=int(sys.argv[2])
HPC_TYPE=sys.argv[3]
STARTING_POINT=int(sys.argv[4])
with open("ncu_cuda_avail2",'r') as f:
    my_num_events=0
    block=0
    fnew=open(f"cuda_events/block{block}",'w')
    event_num=0
    print(f"type of TOTAL_EVENTS: {type(TOTAL_EVENTS)}")
    while(True):
        line=f.readline()
        if TOTAL_EVENTS=='all':
            if "End of file" in line:
                if((my_num_events%EVENTS_PER_FILE)==0):
                    os.system(f"rm cuda_events/block{block}")
                break
        else:
            if(("End of file" in line) or (my_num_events>=TOTAL_EVENTS)):
                if((my_num_events%EVENTS_PER_FILE)==0):
                    os.system(f"rm cuda_events/block{block}")
                break
        if ((HPC_TYPE in line) or (HPC_TYPE=='all')):
            if "Counter" in line:
                event_num+=1
                if event_num<STARTING_POINT:
                    continue
                my_num_events+=1
                HPC=line.split(' ')[0]
                line="cuda:::"+HPC+".sum:device=0 "
                fnew.write(line)
                print(line)
                if ((my_num_events%EVENTS_PER_FILE)==0):
                    block+=1
                    fnew.close()
                    fnew=open(f"cuda_events/block{block}",'w')

    fnew.close()

    print(my_num_events)
```

A.7 RUN_BOTH_TESTS.SHORTLIST.SH

```
#!/bin/bash
CWD=`pwd`
#without injections
cd ..
make clean_runs
make golden DATATYPE=INT BLOCKSIZE=16 &&
cd $CWD
rm cuda_events/*
python3 extract_shortlist_events.py $1 $2 $4 $5
for i in $(seq 1 $3)
do
      for eventsfile in cuda_events/*
      do
            events_in_file=`wc -w $eventsfile | awk '{print $1}'`
            sudo ./benchmark_command.sh
golden_runs/profiler/golden_run$i.out $events_in_file $eventsfile
golden_runs/benchmark/golden_run$i.out
      done
done
cd ..
#with injections
make injected_errors DATATYPE=INT BLOCKSIZE=16
cd $CWD
for i in $(seq 1 $3)
do
      for eventsfile in cuda_events/*
      do
            events_in_file=`wc -w $eventsfile | awk '{print $1}'`
            sudo ./benchmark_command.sh
injector_runs/profiler/injector_run$i.out $events_in_file $eventsfile
injector_runs/benchmark/injector_run$i.out
      done
done
```

A.8 BENCHMARK_COMMAND.SH

```
#!/bin/bash
cd ../bin
echo "eventsfile: $3, size: $2, metric destination: $1, benchmark
destination: $4"
./matrix_multiplication_cuda_int_256 $2 -s 104 $(cat ../utils/$3) 2>> $1
>> $4
cd ../utils
```

Vita

Antonio Teijeiro developed a deep passion for electrical engineering following a high school co-op in the IT department of a credit union. During his time as an undergraduate, he undertook networking and robotics internships at NASA's Kennedy Space Center and worked at school on the application of machine learning techniques to the prediction of epileptic seizures. During his senior year, his capstone project team's work on epileptic seizure prediction earned them a 1st place finish at the national ExCEllence in Senior Design competition hosted at UT Dallas. He finished off his undergraduate career with *suma cum laude* honors in Electrical Engineering and an Outstanding Senior award from the University of Texas at El Paso.

As a master's student, Antonio has TA'd Senior Design, Microprocessor Systems, and GPU Programming courses. He also led and personally participated in two Winter Classic Invitational Cluster Computing Competition supercomputing teams, resulting in 2nd and 4th place national finishes. He is proud of the personal advancement in the field of operating systems, signal processing, computer architecture, and radiation effects he has achieved during this time. He recently began receiving support from Sandia National Laboratories as a year-round intern in the radiation effects experimentation group as he finishes his master's and subsequent Ph.D. work in the field of radiation effects on electronics.

Outside of engineering, the author enjoys training horses, flying, playing music, and learning languages. He also enjoys reading across a myriad of topics in STEM, history, political science, and psychology. For questions about this work or other general inquiry please feel free to contact the author at aeteijeiro@gmail.com.