Multi-Level Fault Tolerance: Evaluation and Mitigation on Modern GPUs

CprE 545: Fault Tolerant Computing (Spring 2022)

Matthew Dwyer
dwyer@iastate.edu

May 3, 2022

Abstract

In this report, we will evaluate modern GPU fault-tolerance and mitigation using a multi-leveled approach. This includes approaching the issue from the RTL level, the ABI/ISA level, and the Software/Application level. From each perspective, we will evaluate recent works in each area and consider the issues they investigate and their proposed solutions. We will go on to make an analysis of the current state of GPU preparedness and suggest whether issues should be further considered for High-Performance Computing (HPC) and consumer applications. Finally, we will describe how software-level fault tolerance can be an effective approach for future implementations in our current computing landscape, but ultimately a range of options is likely to be the best solution going forward.

1 Introduction

With the onset of ever-faster computing platforms enabled by cutting-edge specialty hardware, the number of computing problems within modern capabilities has continued to grow. Computational workloads once thought of as out of reach are now easily obtainable on commercial and consumer-grade hardware. With this newfound access to computing power, more research and capabilities have been achieved on lesser hardware, opening the door to Machine Learning (ML) and Robotics to the masses [25]. Greater software and tool support has also worked to ensure that many commercial and open-source software tools can take full advantage of this hardware, continuing to allow greater capabilities on more accessible hardware platforms.

Graphics Processing Units (GPUs) have been at the heart of this renascence of personal and professional computing. Their distinct Single-Instruction-Multiple-Thread (SIMT) architecture allows for the high-throughput nature of running complex tasks in parallel. This has been coupled with recent additions such as Tensor-Cores [13], allowing for greater support and computation of General Matrix-Multiplication (GEMM) operations, the heart of most ML applications. These advancements have been provided to the masses, making their way into the largest supercomputers [1, 19] all the way down to the some of the smallest Internet-of-Things (IoT) devices [11].

2 Problem

While great strides have been made in performance and capabilities for GPUs, fault-tolerance has all but been neglected except for the most expensive of enterprise implementations [14]. This has been done in part by chip designers and manufacturers to distinguish enterprise-level hardware from the consumer versions and further warranting the hefty price tag that entails. Performance and capabilities between enterprise hardware and their commercial counterparts have continued to converge. Some modern Nvidia GPUs even utilize the same GPU architecture and die between both SKUs, the enterprise offering being binned and packaged with the data center in mind. Often, slight changes such as the inclusion of Error Correction Code (ECC) memories, modified server compliant cooling solutions, and fused off display outputs are the primary separations that are seen [6].

Software and driver support is another distinguishing factor often touted. Despite the likely significant overlap in underlying implementations, workstation and enterprise GPUs are often provided with "enterprise-grade" or "professional" drivers, enabling the usage of the ECC memories and other enterprise applications such as allowing multi-tenant isolation of large GPUs. Differences in updates also are apparent, as enterprise drivers for Nvidia GPUs do not receive "Game-Ready" updates for modern graphics video game applications. Finally, support for enterprise cards is often more readily available, allowing users to get priority access to support agents and internal developers to ensure their implementations and applications are running at peak possible performance.

Despite these differences, the underlying hardware implementation remains primarily the same. This has pushed many novel ML and HPC implementations towards using far more affordable consumer-grade hardware rather than paying magnitudes more for marginal performance capabilities. With this greater availability comes better access to individuals at the cost of fault tolerance. As such, novel approaches to applying fault tolerance to consumer-grade hardware are in high demand. These trends are combined with the high performance demands of modern applications leave little room for commonly performance degrading fault-talent additions. Thus, it is important to explore the overall sliding scale of the current state of modern GPUs, where they are most vulnerable, what side-effects can occur, and what techniques we can employ to prevent them in all modern GPUs.

First, we must understand the implementations and architecture of modern GPU hardware and how faults can propagate, occur, and be mitigated. More importantly, we must also understand the syndromes of faults within a GPU system and how those faults in various parts of the GPU can affect the overall operations and output of critical applications. Once we understand the issues with current hardware and their effects, we can then begin to evaluate approaches to mitigate or lessen the likely hood and side-effects of faults, both at the hardware and software level.

3 Background

To better understand some of the primary points of concern within modern GPUs, we must understand their underlying implementations. Most GPUs implement some form of Reduced Instruction Set Computing (RISC) architecture which is used to program many identical shader cores though-out the GPU. These cores utilize a Single Instruction Multiple Data (SIMD) or SIMT approach to process up to 32 or 64 pieces of data at a time. These data computations often share a Program Counter (PC), execution pipeline, large register banks, and arbitrate for resources (such as ALUs, FPUs, Address, and Store Units) with other workloads (Referred to as cliques, warps, or thread-groups). All data is worked on simultaneously by the computing units and sometimes shared across "threads" or "lanes" during computation. Most operations performed by the shader core operate on the local registers, shared memory, or low-level caches. Although operations such as retrieving data from Video Random Access Memory (VRAM), Dynamic Random Access Memory (DRAM), or even mid-to-high level caches are sometime necessary, they are often avoided if possible as they can be costly. These workloads are scheduled and managed generally by a hardware scheduling engine that is being fed by a host system CPU across a high-speed Peripheral Component Interconnect Express (PCIe) bus. This shows that fault-tolerance must be taken into account at all levels of the GPU, as there are many components that data must pass through to perform a basic operation. With that in mind, low-level shader allocated registers and pipeline resources are of particular interest, as faults in these structures can cause significant errors to occur in data computation and affect multiple values at a time.

It is also important to understand how faults occur in a GPU and other electronic components. Energized particles caused by cosmic rays and high-energy protons in short transient pulses can strike hardware components within the GPU logic or supporting circuitry. This can cause temporary logical errors or permanent damage to the hardware depending on the area struck and the severity [23]. Single Event Upsets (SEU) are the most common non-destructive occurrences affecting modern GPUs. These are often referred to as "bit-flip" events when a signal or stored value is altered such that a single bit of data is flipped from one to zero or vice versa. GPUs are especially susceptible to these issues since they contain large arrays of registers located in many register files and also require tight timing and performance during operation. ECC-enabled register files can assist in detecting and mitigating issues occurring in register files, but their implementation is inconsistent from vendor or generation to generation. Their implementations also often present additional drawbacks in performance degradation, reliability, and overall implementation overhead. ECC also does not reduce the number of faults that occur but rather assists in detecting and correcting them when they occur [18, 20].

Higher performance and cutting-edge GPUs also often see higher levels of SEUs due to the higher frequencies, more advanced operating nodes, shrinking transistor size, increased design complexity, rising transistor density, combined with shrinking voltage sup-

plies [5]. GPUs operating on the ground often incur the fewest faults (soft-errors) due to the increased atmospheric protection from radiation [2], but GPUs and other electronics located in HPC environments and those operating in aerospace environments can see fault occurrence frequencies on the order of minutes [26]. This can raise questions about the performance benefits of utilizing GPUs with respect to the high fault rate they can incur [9].

4 Evaluation Methodologies

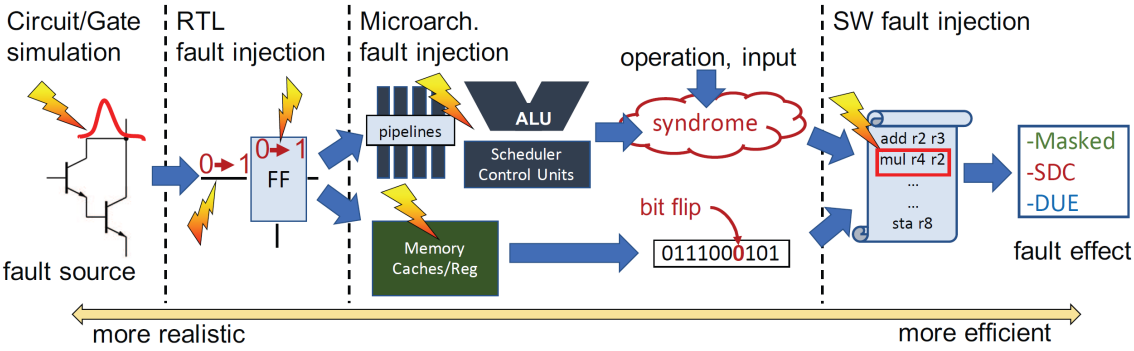

Figure 1: Fault injection propagation levels of clarity. Levels and methods closer to the fault source can provide additional resolution and result in more realistic results, while higher level methods can result in significant simulation speed-ups [21].

As seen in figure 1, fault prorogation characterization can be approached from numerous levels with varying levels of efficiency and realism. Each level presents its own challenges, benefits, and drawbacks. As such, it can at times be challenging to understand the full system from any single perspective. Santos et al. attempted to gain additional visibility while significantly shortening the characteristic generation of GPUs by combine the levels of visibility (See figure 1 for a visualization of fault injection and propagation) and understanding by combining the visibility of RTL based methods and the efficiency of software based simulations. This allowed for a novel holistic understanding of how various structures within the GPU can affect program output and tolerate SEU faults. Often times software methods do not provide the clarity that researchers and developers would hope for but at manageable simulation speeds, while RTL methods can be challenging to implement, require long simulations, but result in high-fidelity data. Combining these two techniques allows a team to produce a reasonable understanding of fault characterizations and symptoms in a simulated system in a far more reasonable amount of time while still providing a detailed picture of its fault-tolerance [21].

When combined, they are able to derive micro-architectural structures within the GPU that may be most susceptible to fault and SEUs. This is important as micro-architectural attribution and characterization can be challenging to achieve since determining fault syndromes from SEU is near impossible with modern hardware since the RTL is close-source

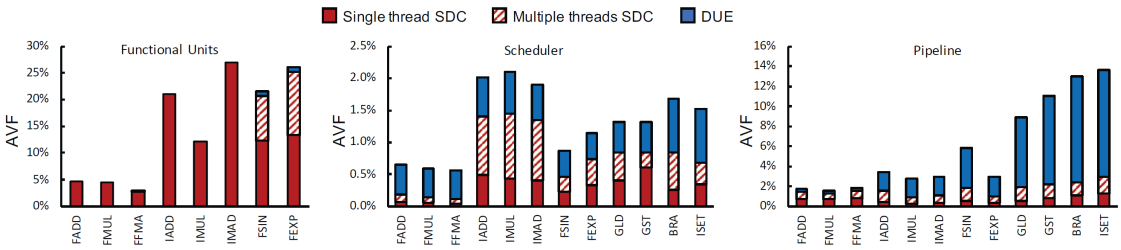

Figure 2: The AVF of RTL level fault injections on functional units (FP32, Int, and SFU), the scheduler, and pipeline registers for various instructions [21].

and the resources most susceptible (such as the hardware scheduler, pipeline control logic, and functional units) are not accessible easily from software. From their experiments, they were able to determine the Architectural Vulnerability Factor (AVF) for multiple micro-architectural structures which are shown in figure 2. AVF is a measure that quantifies if a fault occurs in a architectural structure, how likely that fault will propagate and affect the output of the application [15]. This measure is good for determining how susceptible a specific hardware structure is to faults and SEUs in relation to the entire application and problem being computed.

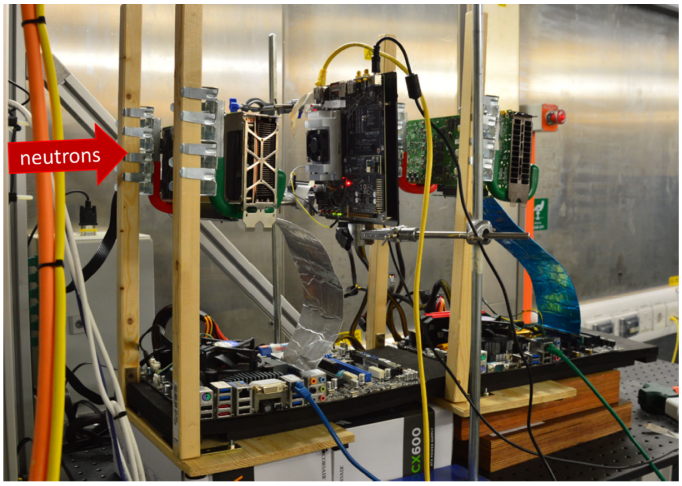

Figure 3: The radiation test setup a the ChipIR facility at the Rutherford Appleton Laboratory. Here a GPU is mounted next to a directed radiation neutron accelerator to target charged particles at the GPU while it is operating. Note the foil on some components to shield them from radiation when not under evaluation [22].

Santos et al. explore this further for Convolutional Neural Networks (CNN) on real hardware by irradiating modern Nvidia GPUs with charged neutron particles while running CNN benchmarks to determine how these effects resulted in changes in output of the networks [22] (See figure 3 for their irradiation setup). Workloads and applications in the ML and CNN space can be challenging to characterize, as they contain many layers of computation, and it can be hard to determine how much affect any single node has on the

output of the entire model. ML models are also lossy in nature, with no model achieving 100% accurate, thus issues can be hard to discern from faults without comparison of known values or recomputation on the same inputs. Some faults are also tolerable at this level, as confidence values may still be within reasonable values.

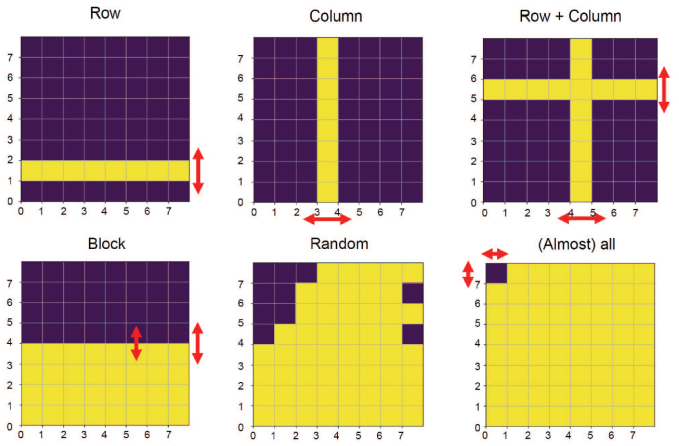

Figure 4: GEMM error patterns evaluated [21].

The benchmarks they chose centered around vehicle guidance systems and GPU utilization, obvious fault critical applications, specifically object detection and classification networks. If a GPU is to be used in in a vehicle guidance system, the system must meet rigorous standards governed by ISO26262. Object detection systems are also required to meet Automotive Safety Integration Level D (ASIL-D) standards as well. ASIL-D requires any components in the system to be able to detect 99% of faults that occur and have a failure rate of no more than 10 failures in time (FIT) over 10^9 hours of operation [10]. There are many possible sources of errors that can undermine the tolerant nature of the GPU guidance system, including environmental changes, software faults, processor, temperature, and voltage variation. Particularly, radiation induced soft (non-permanent) errors dominate the majority of issues GPU systems see in commercial setups [3].

In particular, Santos et al. focused on the General Matrix Multiplication (GEMM) and max pooling operations (significant operations in CNNs) to then extrapolate to overall CNN susceptibility. They considered a wide range of fault patterns shown in figure 4. They show the collected AVF and Program Vulnerability Factors (PVF) (the probability that a transient error of an instruction will result in program output variation/error) for each layer of the You Only Look Once (YOLO) object detection model based on the Darknet CNN and discern by fault type in figure 5.

Figure 6 describes how errors can be propagated throughout a model as inputs progress through each layer. This can be seen by the almost exponential increase in critical faults that slowly build up until almost the entirety of the output has been corrupted. Thus, ML models can be severely susceptible to minor initial faults that can have significant impacts at the final output if not caught early on.

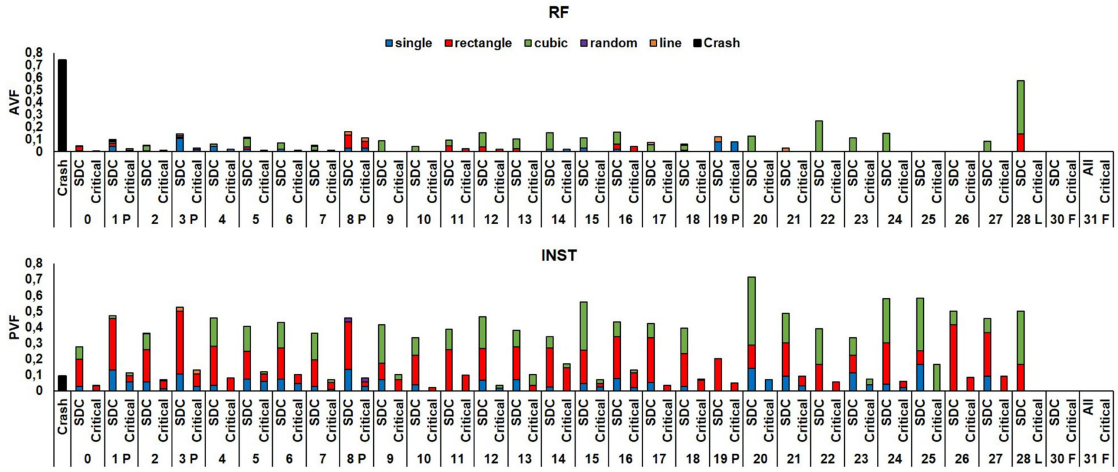

Figure 5: The AVF and PVF of each layer in the YOLO object detection model based on the Darknet for critical and SDC errors partitioned by error type [22].

5 Hardening Approaches

There exist multiple ways to approach the task of mitigating faults in GPU systems. First, modern GPU fault susceptibility must be addressed and evaluated in order to better understand the primary hardware structures susceptible to failures and critical system and workload syndromes (described in the previous section). Next, we can begin to evaluate how to best harden and mitigate those structures to ensure as many fault cases are covered as possible. These approaches include hardware implementation changes to allow for greater support for fault-tolerance at a hardware level through Instruction Set Architecture (ISA) changes to include fault-tolerant instructions for reading, writing, and computing, and software approaches to best ensure reliable results are computed on hardware, susceptible or not. These can be combined with other common methods such as ECC correcting VRAM and register files to provide full coverage from a multitude of SEU faults and propagated errors.

5.1 Software

From their inception, GPUs were primarily fixed-function processing pipelines optimized for graphical applications. As the name suggests, this was their primary intended purpose, yet that purpose has now expanded to include general computing tasks, often referred to as General Purpose GPU (GPGPU) computing. This change has been primarily enabled by the highly programmable nature of modern GPU shader cores. Many methods of programming GPUs have been popularized since CUDA has become the primary method for targeting Nvidia GPUs, HIPP has been used to program AMD GPUs, and other open-source tools such as Open Multi-Processing (OpenMP), Open Compute Language (OpenCL), and Open Acceleration (OpenACC) are able to target wide ranges of the GPU implications[16]. Yet, many of these models are designed with ease of implementation

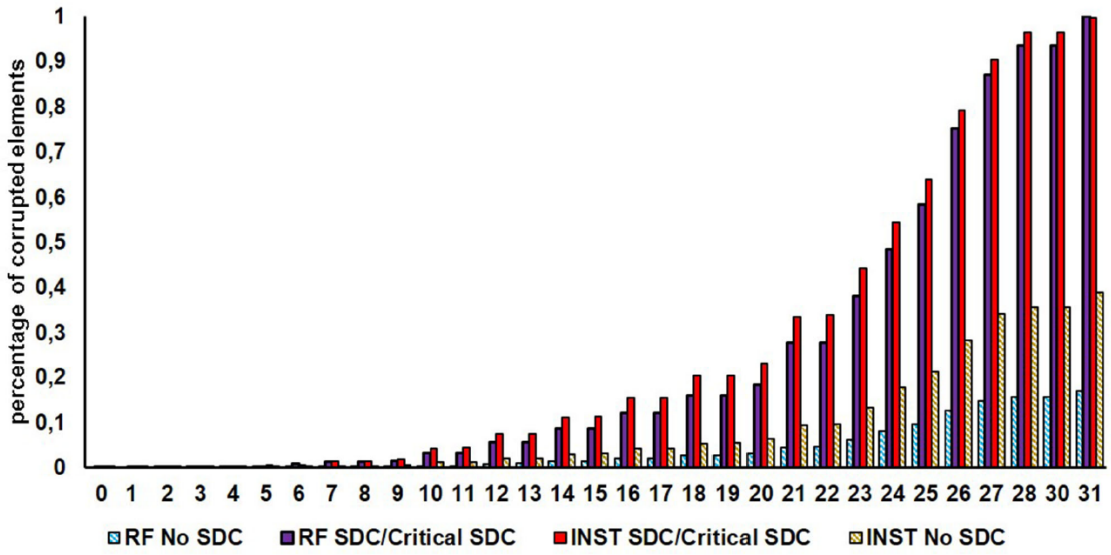

Figure 6: Average percent of corrupted elements at the output of each layer of the YOLO model based on the Darknet CNN [22].

and performance in mind. In order to ensure that modern safety-critical applications and workloads will remain fault-tolerant of modern GPU hardware, consumer, or enterprise, we must ensure that their software implementations are designed with fault-tolerance in mind.

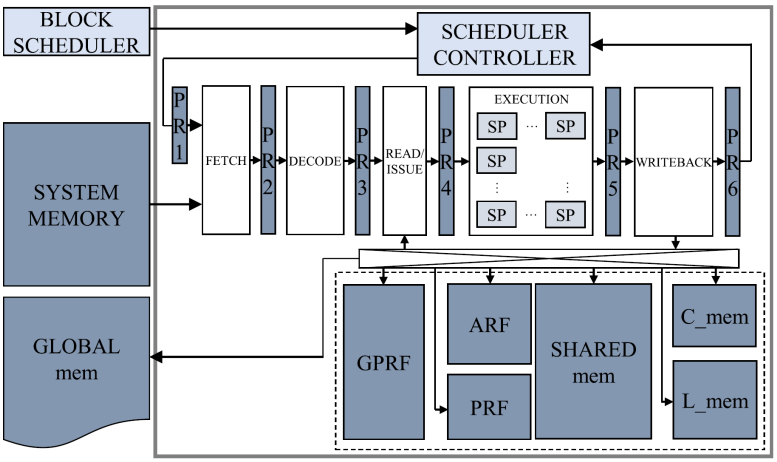

Figure 7: The FlexGripPlus open-source GPU (modeling the Nvidia G80) Simultaneous Multiprocessor (SM) architecture diagram [4].

Goncalves et al. [7] explore this idea in Evaluating Low-Level Software Hardening Techniques for Configurable GPU Architectures. They echo previous findings that many safety-critical applications have been attracted to GPU computing as a way of achieving high-performance and also quick recoverability from transient faults during operation (often through recomuptation). This is challenging, though, as many also require little to

no effect in performance due to those mitigations. To do so, they evaluate a soft-core GPU implementation utilized widely in GPU fault-tolerance work called the FlexGrip-Plus architecture, primarily due to its open-source nature. This GPU implementation has been modeled after the G80 Nvidia Tesla micro-architecture and can even execute CUDA compiled code using the Simultaneous Multiprocessor (SM) 1.0 compatibility environment (See Figure 7 for the architecture diagram). This allows for realistic workload simulation and compatibility to be tested at a Register-Transfer Level (RTL) scope and inject faults to determine fault syndromes and their repercussions [4].

Goncalves et al. also propose assembly level hardening techniques such as instruction and data duplication to allow for increased hardening capabilities. Doing so at the assembly level allows for these methods to be compiler agnostic and also still allow for all compiler optimizations to occur prior to the redundancy techniques to be employed. These code transformations cans also target data and registers at a higher level of granularity, since specific registers and data can be targeted rather than variables. Their work builds off the techniques proposed by Oh et al. [17] and focus on the performance degradation caused by these software hardening techniques. These are targeted primarily at super scalar Central Processing Unit (CPU) architectures and duplicating datapath operations, checking for their result consistency, and notifying the system if irregularities or if value divergence is detected. While adapted from CPUs, their implementation results apply to GPUs as well. Pipeline registers and resources are not access able from the assembly or program level, thus, redundancy in the data and instructions must be utilized to second-handily increase redundancy for the pipeline resources.

The process goes as follows. First, static code analysis is performed to determine the registers in use by the application. From this information we can deduce "spare" registers and duplicate data accordingly on a one-to-one basis for our duplication instruction operations. In the event that there are too few spare registers, two tactics can be employed. Either you must spill register values to memory and incur a performance loss due to the added memory operations, or evaluate the importance of each register in use and accept partial hardening/fault tolerance. In the workloads analyzed by Goncalves et al., sufficient spare registers were always available [7].

5.2 Hardening Techniques

Following register data duplication, the following three transformations are performed: *datapath duplication*, *consistency checking*, and *host notification*. *Datapath duplication* consists of duplicating all computational operations. These are done on the duplicated data registers as to allow for as much isolation as possible. Doing so allows for the advantage of Instruction Level Parallelism (ILP) to be exploited by the GPU hardware, increasing performance relative to that of running a workload twice. It is assumed that memory is separately hardened (ECC for example) and thus stores and memory addresses are not duplicated. *Consistency checking* ensures that the data computed by the duplicated instructions matches the original. It does this by inserting a comparison and a conditional

branch instruction to an error sub-routine. This creates an undesired dependency between the two computation paths, but is unavoidable in this basic implementation. Goncalves et al. particularly evaluates this consistency checking for memory access and execution predicate setting operations, since they determine what data is operated on throughout the execution of the program. Finally, *host notification* occurs when a fault has been detected. This is often in the form a trap instruction seen by the host signaling a an error has been encountered in the GPU code, but can also happen though the use of a memory write to a global memory address accessible by the host. These notifications should not occur on correct program execution, but also depend on the *consistency checking* of the predicate register to ensure that the data execution was valid in the first place. Figure 8 demonstrates these techniques on some basic GPU code (see the Unhardened code compared to the Non-optimized Hardened code).

	Unhardened Code	Non-optimized Hardened Code	Optimized Hardened Code			
			Move	Traceback MEM	Traceback PRED	Delayed Notification
1:	MOV R3, 4	MOV R3, 4	MOV R3, 4	MOV R3, 4	MOV R3, 4	MOV R3, 4
2:		MOV R3', 4;	MOV R3', 4;		MOV R3', 4;	MOV R3', 4;
3:	ADD R1, R1, 1;	ADD R1, R1, 1;	ADD R1, R1, 1;	ADD R1, R1, 1;	ADD R1, R1, 1;	ADD R1, R1, 1;
4:		ADD R1', R1', 1;	ADD R1', R1', 1;	ADD R1', R1', 1;		ADD R1', R1', 1;
5:	LOAD R2, [R1];	LOAD R2, [R1];	LOAD R2, [R1];	LOAD R2, [R1];	LOAD R2, [R1];	LOAD R2, [R1];
6:		LOAD R2', [R1'];	MOV R2', R2;	LOAD R2', [R1'];		LOAD R2', [R1'];
7:		SETP.NE PE, R1, R1';	SETP.NE PE, R1, R1';	SETP.NE PE, R1, R1';		@!PE SETP.NE PE, R1, R1';
8:		@PE ERROR;	@PE ERROR;	@PE ERROR;		
9:	SETP.NE P0, R3, R0;	SETP.NE P0, R3, R0;	SETP.NE P0, R3, R0;	SETP.NE P0, R3, R0;	SETP.NE P0, R3, R0;	SETP.NE P0, R3, R0;
10:		SETP.NE PE, R3, R3';	SETP.NE PE, R3, R3';		SETP.NE PE, R3, R3';	@!PE SETP.NE PE, R2, R2';
11:		@PE ERROR;	@PE ERROR;		@PE ERROR;	
12:		SETP.NE PE, R0, R0';	SETP.NE PE, R0, R0';		SETP.NE PE, R0, R0';	@!PE SETP.NE PE, R3, R3';
13:		@PE ERROR;	@PE ERROR;		@PE ERROR;	
14:	@P0 BRA 1;	@P0 BRA 1;	@P0 BRA 1;	@P0 BRA 1;	@P0 BRA 1;	@P0 BRA 1;
15:	STORE [R4], R1;	STORE [R4], R1;	STORE [R4], R1;	STORE [R4], R1;	STORE [R4], R1;	STORE [R4], R1;
16:		SETP.NE PE, R1, R1';	SETP.NE PE, R1, R1';	SETP.NE PE, R1, R1';		@!PE SETP.NE PE, R1, R1';
17:		@PE ERROR;	@PE ERROR;	@PE ERROR;		
18:		SETP.NE PE, R4, R4';	SETP.NE PE, R4, R4';	SETP.NE PE, R4, R4';		@!PE SETP.NE PE, R4, R4';
19:		@PE ERROR;	@PE ERROR;	@PE ERROR;		
20:						@PE ERROR;

Figure 8: Software hardening techniques presented in Goncalves et al. Datapath duplication instructions are shown in green, consistency checking instructions are highlighted in blue, and host notification operations in red. [7]

5.3 Optimizations

Due to the large amount of additional instructions that are inserted due to the software hardening, the execution time of an application would increase significantly, thus the following optimizations were proposed: *move optimizations*, *traceback optimizations*, and *delayed notification optimizations*. These optimizations trade reliability, the detection of some faults, and a delay in host notification for additional performance, attempting to strike a middle ground.

Move optimizations eliminate the duplicated load instruction inserted during hardening and instead performs a move operation from the loaded register to the duplicate. This in theory provides an additional speed-up since the value is no longer being loaded twice but rather moved, providing an additional point of failure that would not be detected

during the consistency check if the loaded value were altered before it was moved.

Traceback optimizations focus on duplication and consistency checking hardening (along with host notification in some cases) on a specific set of the most critical instructions in a program rather than all of them. Since previous works have shown that determining the sections of code must vulnerable or critical is still an open problem, Goncalves et al. chose to focus on predicate setting instruction sequences (control flow) and memory operations (data flow). These sequences were determined based on control flow analysis stemming from the list of target instructions and recursively tracing back to build up a set of dependent instructions.

Delayed notification optimizations trade host notification time for removing the dependency of each consistency check on the previous. Since every consistency check overwrites a flag in the predicate register, the following consistency check could overwrite the value written previously if it is not completed and checked prior to computing the next check. As such, there is a dependency on the previous check before the next can continue. By using conditional comparison (where the flag is only set if it is resulting in a true/high value) and only checking the flag at the end of the program execution, we can then remove the dependency on the previous check at the sacrifice of slower notification time and lower accuracy of when the fault occurred.

5.4 ISA

Another approach promoted by Goncalves et al. is to take the software methods previously described (duplication, consistency checking, and host notification) and provide hardware support encapsulated in a few additional atomic instructions. These would focus again on memory access, register writing, and predicate setting instructions and would combine the duplication, checking, and notification into hardware instructions. Including all of these operations into a few single hardware encoded instructions would allow for fewer hardware instructions and a performance benefit when used [8]. These instructions would also make fault-tolerance a first-class citizen within the GPU shader pipeline, allowing for less tangential methods through software by providing real support for them directly in hardware.

Hardware implementation overhead.

	Original design	Hardware overhead (%)		
		ISAset	ISAls	ISAsetls
Area (mm^2)	196,338	0.047	0.030	0.179
Cells (#)	377,798	0.152	0.031	0.369
Power (mW)	95.074	0.053	0.056	0.125
Delay (ns)	1.99205	−0.002	−0.001	−0.008

Figure 9: Atomic resilient instruction implementation overhead in area (mm^2 + # of cells), timing, and power [8].

These additions were again tested on the FlexGripPlus open-source GPU. Changes were made to the instruction encoding to allow for the inclusion of an additional encoded source register (in the case of comparisons) and an additional destination register location (for stores). There were not enough additional bits available to allow for full register file access of the 128 registers available to a thread, thus only a subset (load and store used 5 bits for the first replica and 6 for the second, while predicate setting uses 6 offset bits) was chosen to be allocated for the duplicate operations. These atomic resilient instructions were implemented in addition to their original counterparts and were distinguished with a trailing 1 in the encoded op code. A host notification system interrupt was also created to allow for hardware supported notifications back to the host processor from the GPU. The implementation cost of each of the 3 proposed instruction can be seen in the table of figure 9.

5.5 Comparison of Approaches

Execution time overhead.

Application	Original (µs)	Hardening technique overhead (%)			
			Memory	Set predicate	All
FFT	964	SW	93.1	88.7	103.2
		ISA	56.3	24.5	65.6
Matrix Mult.	320	SW	95.9	87.1	104.9
		ISA	41.9	9.3	46.9
Vector Sum	141	SW	105.2	–	–
		ISA	45.3	–	–
Bitonic Sort	824	SW	83.2	79.0	104.4
		ISA	30.3	55.9	36.6
Edge Detect.	1096	SW	–	–	–
		ISA	49.5	75.1	66.1

Figure 10: Hardware atomic instruction (ISA) implementation execution comparison with software hardening alone (execution time overhead) [8].

ISA hardening techniques built off the software hardening approaches provided the best results in reduced performance overhead when compared to software alone. As seen in figure 10 the execution time overheads can be significantly reduced. Figure 11 showcases its performance in Silent Data Corruption (SDC) occurrences and figure 12 shows the differences in Detected Unrecoverable Errors (DUE) such as infinite loops caused by a SEU.

Again, since we are applying the duplication of data we must also ensure that we can allocate registers efficiently to available locations to ensure no significant degradation in performance. This is one of the primary issues of the both software and hardware (ISA) hardening approaches. Also, while the implementation overheads are small, these instructions can be implemented in software and were shown not to be needed in all scenarios. While their speed-up is significant when used, the additional overhead of adding these operations would out weight the cost of implementing them in software due to the optimization approaches that could still be take and the flexibility that they provide in

SDC Reduction.

Application	SDC effects	Hardening technique (%)			
			Memory	Set predicate	All
FFT	1452	SW	89.1	39.9	100.0
		ISA	84.9	17.5	96.1
Matrix Mult.	3461	SW	100.0	- 4.6	100.0
		ISA	100.0	- 5.6	100.0
Vector Sum	3164	SW	100.0	–	–
		ISA	100.0	–	–
Bitonic Sort	1739	SW	61.1	87.5	100.0
		ISA	92.6	71.4	100.0
Edge Detect.	258	SW	–	–	–
		ISA	99.6	80.2	100.0

Figure 11: Silent Data Corruption (SDC) reductions in both software and ISA hardening for memory and set predicate instructions [8].

most cases. Both of these methods were also tested on RTL rather than on real hardware with simulated faults inserted. While you are able to achieve low-level visibility into the hardware at this level, physical tests could have also been conducted to further develop convincing evidence that these methods would map to realistic implementations.

DUE reduction.

Application	DUE effects	Hardening technique (%)			
			Memory	Set predicate	All
FFT	2321	SW	30.1	100.0	99.9
		ISA	32.3	100.0	100.0
Matrix Mult.	1755	SW	0.2	99.9	99.8
		ISA	4.0	100.0	100.0
Vector Sum	1	SW	100.0	–	–
		ISA	100.0	–	–
Bitonic Sort	1368	SW	1.9	99.9	99.9
		ISA	5.3	100.0	100.0
Edge Detect.	1952	SW	–	–	–
		ISA	12.7	99.9	99.9

Figure 12: Detected Unrecoverable Error (DUE) reductions in both software and ISA hardening for memory and set predicate instructions [8].

6 Future Approachs

6.1 General Thoughts

For future work in hardening commercial and enterprise GPUs for fault-critical scenarios, it is pertinent not to underestimate the importance of software hardening techniques. Not only do these techniques hold for any GPU available, hardened or not, but some optimization techniques can be utilized to reduce their effect on performance. While almost all techniques will cause some degradation in performance when compared to their optimized, non-hardened counterparts, the differences in performance are likely still viable for most applications. If the performance differences are unacceptable, it is possible the class of GPU platform being targeted is not correct for the use case. The focus on software

hardening techniques is dual pronged. First, regardless of implementation, in some environments such as aerospace and HPC faults are a regular occurring expectation. Software hardening can be implemented following an evaluation of performance and tolerance on a per-implementation level and be added as a second layer of protection. This leads to my second point. Software based redundancy can be tuned according to performance to cost benefit dynamically. This is the only method which puts the power of weighing performance vs. cost in the hands of the application developer rather than the hardware designer. This freedom allows for far more flexibility in wide ranges of applications, scenarios, and use case environments.

These reasons combine to build a compelling argument for focusing on software hardening techniques. It goes without saying though that providing users with the ability to choose is not a poor approach. Providing an application developer or compiler with the ability to pick between instruction implementations which provide different levels of tolerance with respective trad-offs would allow for even greater flexibility. Yet, these features are again often locked behind costly enterprise grade pay-walls, hardware, drivers, and applications. This leads me to my next point.

GPU fault tolerance needs to be treated as a first class citizen during hardware design, and should be standard on all commercial grade GPUs. As previously mentioned, more commercially viable, and yet critical, applications are gaining traction in the research and soon to be commercial product phases of development. It has been shown that by utilizing the same GPU dies for enterprise and commercial applications that some features are baked in by nature of the silicon design process. Enabling many of these features for commercial GPUs would likely come at no cost to design firms, and would allow for the significant improvement in overall industry adoption of GPUs in fault tolerant areas. As previously discussed, in some critical scenarios, the fault risk of using GPUs can outweigh their performance benefits. As such, ensuring that fault tolerance is a standard feature for GPUs and that their tolerance levels are provided at product launch would go a long way to allowing applications and system designers to make informed decisions about their ability to include GPUs in their applications and projects. Testing and fully understanding the syndromes and susceptibility of GPUs to faults can really only be reasonably accomplished when the RTL source is available and at significant cost. Designers and manufacturers therefore should either allow researchers access to study these characteristics of their hardware or provide the in-depth analysis themselves.

6.2 Lock-Step/Collaborative GPU Computing

Another area of GPU fault tolerance that has yet to see considered is the combination of multiple GPUs. It is not uncommon for CPU processors to be arraigned in a lock-step computational mode where each preforms the same operations in lock-step (as the name would suggest) and compares results. This can significantly reduce the SEU and temporal failure suitability that GPUs are particularly susceptible to and allow for possibly fewer changes to be made to the hardware and software to achieve the same tolerance levels.

This idea has been shown to be successful in aerospace environments with commercial grade, non-hardened CPU system implementations and could allow for even greater levels of tolerance for the most critical of applications. The likes of SpaceX and other space agencies have turned to Triple Modular Redundancy (TRM) and x86 commercial processors to achieve greater performance, lower cost, and high tolerances for space flight hardware [12, 24]. This method of duplicating hardware resources in not new and could also be applied to GPUs in a similar context, allowing for fewer changes to be made to harden hardware. Support would still likely be required by hardware manufacturers though for those specific use cases.

This could also be implemented as a collaborative effort rather than lock-step by certifying the results and duplicating some computations across two or more GPUs. This would allow a lesser loss of performance due to the nature of lock-step duplication of computation while still allowing for a dynamic range of tolerance levels. While some additional performance loss would be induced by the required sharing of information across multiple GPU cards and also allowing for a larger fault surface, it would provide a sliding scale of opportunities from double the computational power (assuming two GPUs in tandem) to almost double the tolerance in application computation. Again, this goes back to providing the ability to determine the fault tolerance of the hardware at the application level rather than being determined based off the hardware chosen to implement the system. Having this additional freedom allows for hardware to be used in a larger variety of ways all with little hardware costs.

7 Conclusion

With the rapid growth of GPU development and adoption in a wide range of critical applications and systems, the need for fault tolerant GPU hardware and software implementations is ever growing. As these applications continue to expand in their adoption and spread into the consumer space in commercial products and systems, it is vital that all modern GPUs carry support for multiple variable levels of fault detection, correction, and avoidance. While the current state of GPUs is every improving in performance, it is still vital to consider architectural changes' affects on fault tolerance of the GPU.

With the aforementioned shift in adoption of consumer and commercial hardware to implement critical systems, it has only grown the need to support fault-tolerant solutions and approaches for consumer hardware and platforms. As crucial applications such as medical devices and imaging, robotics, self-driving cars, and ML recognition tasks continue to push for greater consumer usage, we must ensure that fault-tolerant implementations and technologies are present on the platforms they are implemented on. These have and will continue to include consumer-grade hardware, which necessitates either the addition of fault-tolerant features on consumer hardware or the cost of enterprise hardware to decrease.

Finally, regardless of implementation platform, software hardening techniques and op-

timizations can be utilized to ensure that a minimal level of fault tolerance is achieved. While introducing degradation in performance capabilities in most cases, in critical applications this is likely a worth while trade-off. Ultimately it should be up to developers and system integrators to decide the level of fault their system is willing to tolerate. It is therefore important to provide them with cutting edge tools, support, and techniques to work with a multitude of tunable software hardening solutions, and allow them to pick the solutions that fit their needs.

References

[1] Introducing the ai research supercluster — meta's cutting-edge ai supercomputer for ai research, Jan 2022.

[2] José Rodrigo Azambuja, Gabriel Nazar, Paolo Rech, Luigi Carro, Fernanda Lima Kastensmidt, Thomas Fairbanks, and Heather Quinn. Evaluating neutron induced see in sram-based fpga protected by hardware- and software-based fault tolerant techniques. *IEEE Transactions on Nuclear Science*, 60(6):4243–4250, Dec 2013.

[3] R.C. Baumann. Radiation-induced soft errors in advanced semiconductor technologies. *IEEE Transactions on Device and Materials Reliability*, 5(3):305–316, Sep. 2005.

[4] Josie E. Rodriguez Condia, Boyang Du, Matteo Sonza Reorda, and Luca Sterpone. Flexgripplus: An improved gpgpu model to support reliability analysis. *Microelectronics Reliability*, 109:113660, 2020.

[5] Anand Dixit and Alan Wood. The impact of new technology on soft error rates. In *2011 International Reliability Physics Symposium*, pages 5B.4.1–5B.4.7, April 2011.

[6] Alex Glawion. Pro vs. consumer gpus - what's the difference &; why so expensive?, Apr 2022.

[7] Marcio Goncalves, Esteban Rodriguez, Matteo Sonza Reorda, Luca Sterpone, and José Rodrigo Azambuja. Evaluating low-level software-based hardening techniques for configurable gpu architectures. *The Journal of Supercomputing*, 78:1–25, 04 2022.

[8] M.M. Gonçalves, J.E. Rodriguez Condia, M. Sonza Reorda, L. Sterpone, and J.R. Azambuja. Improving gpu register file reliability with a comprehensive isa extension. *Microelectronics Reliability*, 114:113768, 2020. 31st European Symposium on Reliability of Electron Devices, Failure Physics and Analysis, ESREF 2020.

[9] Siva Kumar Sastry Hari, Timothy Tsai, Mark Stephenson, Stephen W. Keckler, and Joel Emer. Sassifi: An architecture-level fault injection tool for gpu application resilience evaluation. In *2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 249–258, April 2017.

[10] Road vehicles — Functional safety — Part 1: Vocabulary. Standard, International Organization for Standardization, Geneva, CH, March 2000.

[11] Leela S Karumbunathan. Nvidia jetson agx orin series technical brief v1.1, Mar 2022.

[12] Jacek Krywko. Space-grade cpus: How do you send more computing power into space?, Nov 2019.

[13] Stefano Markidis, Steven Wei Der Chien, Erwin Laure, Ivy Bo Peng, and Jeffrey S. Vetter. NVIDIA tensor core programmability, performance &amp precision. In *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, may 2018.

[14] Adriano Marques. Consumer vs server-grade hardware and why you can't do business-grade work with the former, Jun 2021.

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

[16] Bogdan Oancea, Tudorel Andrei, and Raluca Mariana Dragoescu. GPGPU computing. *CoRR*, abs/1408.6923, 2014.

[17] N. Oh, P.P. Shirvani, and E.J. McCluskey. Error detection by duplicated instructions in super-scalar processors. *IEEE Transactions on Reliability*, 51(1):63–75, March 2002.

[18] Daniel A. G. Oliveira, Paolo Rech, Heather M. Quinn, Thomas D. Fairbanks, Laura Monroe, Sarah E. Michalak, Christine Anderson-Cook, Philippe O. A. Navaux, and Luigi Carro. Modern gpus radiation sensitivity evaluation and mitigation through duplication with comparison. *IEEE Transactions on Nuclear Science*, 61(6):3115–3122, Dec 2014.

[19] Oliver Peckham. Nvidia announces 'eos' supercomputer, Mar 2022.

[20] L. L. Pilla, P. Rech, F. Silvestri, C. Frost, P. O. A. Navaux, M. Sonza Reorda, and L. Carro. Software-based hardening strategies for neutron sensitive fft algorithms on gpus. *IEEE Transactions on Nuclear Science*, 61(4):1874–1880, Aug 2014.

[21] Fernando F. dos Santos, Josie E. Rodriguez Condia, Luigi Carro, Matteo Sonza Reorda, and Paolo Rech. Revealing gpus vulnerabilities by combining register-transfer and software-level fault injection. In *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 292–304, June 2021.

[22] Fernando Fernandes dos Santos, Pedro Foletto Pimenta, Caio Lunardi, Lucas Draghetti, Luigi Carro, David Kaeli, and Paolo Rech. Analyzing and increasing

the reliability of convolutional neural networks on gpus. *IEEE Transactions on Reliability*, 68(2):663–677, June 2019.

[23] Charles Slayman. *JEDEC Standards on Measurement and Reporting of Alpha Particle and Terrestrial Cosmic Ray Induced Soft Errors*, volume 41, pages 55–76. 09 2010.

[24] Robert Sole. Spacex for the falcon 9 uses linux and x86 processors on three redundant computers, Dec 2020.

[25] Yifan Sun, Nicolas Bohm Agostini, Shi Dong, and David Kaeli. Summarizing cpu and gpu design trends with product data, 2019.

[26] Devesh Tiwari, Saurabh Gupta, James Rogers, Don Maxwell, Paolo Rech, Sudharshan Vazhkudai, Daniel Oliveira, Dave Londo, Nathan DeBardeleben, Philippe Navaux, Luigi Carro, and Arthur Bland. Understanding gpu errors on large-scale hpc systems and the implications for system design and operation. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, pages 331–342, Feb 2015.