Evaluating the Impact of Execution Parameters on Program Vulnerability in GPU Applications

Fritz G. Previlon, Charu Kalra and David R. Kaeli
Northeastern University
Boston, MA, USA
Email: {previlon, ckalra, kaeli}@ece.neu.edu

Paolo Rech
UFRGS, Universidade Federal do Rio Grande do Sul
Porto Alegre, Brazil
Email: prech@inf.ufrgs.br

***Abstract*—While transient faults continue to be a major concern for the High Performance Computing (HPC) community, we still lack a clear understanding of how these faults propagate in applications. This paper addresses two particular aspects of the vulnerabilities of HPC applications as run on Graphics Processing Units (GPUs): their dependence on input data and on thread-block size.**

To characterize fault propagation as a function of input parameters, we leverage an ISA-level fault injection framework and carry out an extensive fault injection campaign to characterize the vulnerability of a suite of GPU applications. Our results show that the vulnerability of most of the programs studied are insensitive to changes in input values, except in less common cases when input values were highly biased, i.e., values that exhibit a special vulnerability behavior. For example, the multiplication property of any value with a zero value (zero times any number is equal to zero) makes it a biased input for multiplication operations.

Our study also examines the effects of changing the GPU thread-block size and its impact on vulnerability. We found that, similar to performance, the vulnerability of an application can depend on the block size of the kernels in the application. In some applications, we found that the silent data corruption rate can vary by as much as 8% when changing the block size of a kernel.

I. Introduction

With the recent adoption of accelerators, we have witnessed a large number of applications benefiting in terms of performance. Graphics Processing Units (GPUs), in particular, enabled by their improved programmability, have become an attractive parallel processing device in High Performance Computing (HPC). GPUs, in fact, are currently used for compute a growing range of applications that includes scientific computing [1], bio-informatics [2], molecular modeling [3], and financial applications [4].

Unfortunately, since GPUs were originally designed to serve as a graphics rendering device, little attention has been paid to the reliability of these devices. Although there have been significant improvements on GPUs architecture to make them more attractive for a broader set of applications, new work needs to be performed to address inherent design and programming trade-offs that impact their reliability.

Errors that undermine the reliability of an HPC system come from a variety of sources including environmental perturbations, firmware errors, manufacturing process, temperature, and voltage variations. Radiation-induced faults are the most critical class of transient faults affecting the reliability of today's computing systems, as they produce a failure rate that is higher than all the other reliability mechanisms combined [5]. Current HPC systems experience transient faults every few tens of hours [6], [7]. Understanding the relationship between the vulnerability of these devices, and its dependence on the characteristics of the programs running on them, is crucial in order to make progress toward exascale computing.

Recent research on GPU reliability has shown that the vulnerability of applications running on GPUs strongly correlates with the type of the application running on the device [8]. Failure rates for applications running on the GPU can vary wildly. For example, Fang *et al.* studied the error resilience of GPU applications by injecting faults in the hardware of a real GPU [8]. Their results show that depending on the application, 1% to 38% of the faults result in errors at an application's outputs and 6% to 70% of the faults result in system crashes. This level of variability in the resilience of the applications suggests that application-specific measures should be taken to address the reliability of GPUs.

We argue that application developers that are working on reliability-sensitive applications should have guidance on how best to design resilient applications. To address this, Sridharan *et al.* introduced a methodology to quantify the level of vulnerability contained in a piece of software independent of the targeted hardware where it will run [9]. A programmer can use this methodology iteratively and redesign his/her code until the level of vulnerability corresponds to a pre-established reliability target level.

While the vulnerability of a program can be independent of hardware parameters, it is dependent on the input parameters, as well as the specific binary generated by the compiler (and compiler flags). Input values, through logical masking, have the ability to effectively influence fault propagation. A change in the input values of a program can change the vulnerability of this program. In this work, we aim to develop a better understanding of this dependence.

The concern for the programmer is to effectively measure the vulnerability of their program. Accounting for all combinations of the input data and optimization flags is unfeasible. Our study aims at identifying reliability similarities among input values or optimization configurations, eventually limiting the number of cases to test. Our analysis is then beneficial for

978-3-9819263-0-9/DATE18/©2018 EDAA

experimenters or system reliability quantifiers and qualifiers.

Sridharan *et al.* investigated the effects of program input values on program vulnerability [10] and concluded that the primary cause for changes in program vulnerability across varying input data values directly relates to the execution profile (i.e., which code regions executed) of the program. The authors argued that an input that causes a program to touch all code regions should provide a fairly accurate measure of the vulnerability of the program. Additionally, Jones *et al.* examined the impact of compiler optimizations on system vulnerability and found that a compiler flag, *-freorder-blocks* can reduce the vulnerability of a system across a number of applications [11].

Prior experiments carried out by Sridharan and Kaeli [10] and Jones [11] were conducted on CPUs. In this work, we evaluate program sensitivity to changes in program inputs for applications that run on GPUs, where the variability in the execution profile tend to be minimized.

We found that, in general, changes in input values for a program do not greatly influence vulnerability. However, an input matrix or array composed of mostly biased values (values that are likely to be logically masked, 0's and 1's for example) can impact the level of robustness of a program by as much as 35% in the presence of transient faults. Additionally, we found that changing the thread-block size of the kernels in a GPU application can change the rate of silent failures by as much as 7% on average for the impacted applications.

This paper provides the following contributions:

1) We provide a characterization of the vulnerability changes based on different input data values for a suite of a popular benchmark. We explore extreme cases, using biased input data values (value always equal to zero or one).
2) We provide a characterization of vulnerability changes incurred in a program when changing the thread-block size of an application.

The paper is organized as follows. In Section II, we describe our methodology used to quantify software vulnerability. as well as how this metric can vary with input data and program parameters. In Section III we present our methodology and experiment setup. In Section IV, we present our findings when performing fault injections on the Rodinia suite of applications. In Section V, we conclude the paper.

II. Background

In this paper, we focus on *transient faults*, which can be generated by various sources, including high-energy neutrons produced by the interaction of cosmic rays within the terrestrial atmosphere. When a transient error in a hardware structure (either memory or logic) is exposed to the software level, there are three possible outcomes. (1) the fault can be *masked* in the executing program either when an operation overwrites the affected bit (e.g., overwriting a register after the flip has occurred and before any instruction reads the faulty value), or when the program does not use the faulty bit. We call this outcome a *masked* outcome. (2) the fault may be detected by the program through sanity checks, or by the operating system (e.g., when a transient fault causes a user program to request access to an unallocated memory location). Outcomes in this category are termed *detected and unrecoverable error (DUE)*. (3) the fault is neither masked nor detected. In this case, the fault produces a *silent data corruption (SDC)*. An SDC causes a program to produce an erroneous output without alerting the user of the incorrect result (silent).

A. Measuring Program Vulnerability

Each application has its proper level of sensitivity to transient faults. In some programs, a fault is more likely to generate a failure (DUE or SDC) than in other programs. Application developers can use one of two methods to measure the vulnerability of their program: the Architecturally Correct Execution (ACE) analysis [12] or software fault injection.

1) ACE analysis: ACE analysis systematically identifies state in a program structure (such as the architectural register file) that is necessary for correct execution of the program. It then computes the proportion of time that the structure contains critical state during program execution. ACE analysis must also consider what portion of the structure holds critical state. For example, to measure the vulnerability of the register file through ACE analysis, one needs to systematically track the instructions that access the register file and are capable of affecting the criticality of each bit in the register file. If two consecutive accesses to a bit in the register file are writes, ACE analysis will mark the first write as non-critical because the first value is overwritten by the second write access, and thus was not used by the program. Any change in the ordering of accesses to the register file will likely change the vulnerability of the program.

2) Software Fault Injection: Software fault injection is a more straightforward and common approach to evaluate the resilience of a program. To measure the vulnerability of the register file through fault injection, a developer can run the application injecting a fault (flipping a bit) in a randomly chosen register. The outcome of an injection is recorded. The ratio of the number of executions for each type of outcome (Masked, DUE or SDC) to the total number of injections gives the probability for a transient fault to produce each outcome. Our study makes use of the fault injection methodology.

B. Input Data and Program Vulnerability

Two features that can influence the resilience that is inherent in a program are: 1) the ordering of instructions in the program and 2) the logical masking performed by the individual instructions of the program [10]. The ordering of instructions allows for identification of dynamically dead instructions and variables, while the type of operations (arithmetic, logical, control flow) performed by each instruction allows for estimating the amount of logical masking present in the program.

1) Program Instruction Ordering: The dynamic instruction ordering in a program dictates how a program manipulates its data, which contributes to its resilience properties. While the instructions grouped into a basic block always execute in the

same order, changing the input parameters of a program has the potential to influence the execution order of the individual basic blocks, modifying the control flow of the program. If this occurs, different input values can cause a program to have a different dynamic execution pattern, changing the traversal order of the basic blocks, resulting in varying vulnerability behaviors.

2) Logical Masking: Different input values will change the output of arithmetic and logical operations of a program. This has direct impact on the ability of a program to mask potential faults that may happen during its execution. For example, if a MUL (multiplication) instruction receives a value of zero as one of its input register operands, a fault in the second input register will always be masked and will not affect the correct execution of the program. However, if we change the input value of the program in such a way that the same MUL instruction does not receive a zero at its input registers, the amount of masking present in the program will be changed. If the application developers want to develop more resilient and reliable applications, they have to be able to estimate the reliability of their code with every possible combination of input data values and program parameters, which is not always possible. We intend to analyze if there are subsets of input data combinations which provide similar reliability estimations. Such a result would significantly reduce the time needed to evaluate a program reliability.

III. Methodology

A. Error Model

This paper evaluates the resilience of key elements in the instruction execution datapath (e.g., the Arithmetic Logic Units, the Load-Store Units) of a GPU in the presence of single-bit transient faults. We carry out this evaluation through an extensive fault injection campaign in the destination registers of all instructions that write to a general purpose register in the GPU. For every dynamic instruction, our goal is to determine the likelihood of having a silent data corruption (SDC), or a detected and unrecoverable error (DUE) if this instruction does not execute correctly. A fault in the datapath of an instruction will likely produce an incorrect instruction output. If the instruction writes to a register, this register will contain a wrong value until it is overwritten by another instruction.

B. SASSIFI: Fault Injection Framework

In this section, we briefly describe the framework used for our fault injection experiments. We use SASSIFI [13] (SASSI Fault Injection), a tool based on SASSI [14], to inject faults into instruction outputs of programs running on NVIDIA GPUs.

SASSI [14] is a compiler-based tool that can instrument GPU programs. After a program has been compiled to SASS (NVIDIA assembly language), SASSI uses *ptxas*, the compiler back-end and assembler developed from NVIDIA, to insert SASS instrumentation code into the program's SASS assembly. The SASS instrumentation code that SASSI will insert into an application has knowledge about the instruction that is executing, its inputs, outputs and memory used. Additionally, the instrumentation code is able to change the value of an input or output of an instruction.

To use SASSI, developers must first write their own instrumentation code (or SASSI handler) and recompile their application to use the instrumentation code. In a SASSI handler, the developer directs SASSI on which type of instruction to instrument, and where (either before or after the instruction) to add the instrumentation code. SASSI is modeled after the Intel Pin toolset [15], which was modeled after the Digital Equipment Corporation ATOM toolset [16].

SASSIFI leverages SASSI in order to inject bit flips at the output of instructions. It is worth noting that SASSI injects the instrumentation code after the program's SASS code has been generated. Therefore, SASSI has minimal interference with the original application, its instruction schedule, or register usage.

C. Fault Injection with SASSIFI

SASSIFI is a framework that supports the automation of SASSI-based fault injection studies. The framework is composed of a set of scripts, SASSI handlers and application suites.

Carrying out a fault injection study with SASSIFI is a three step process. First, SASSI is used to profile the application. Second, information obtained from the profiling stage is then used to statically generate a number of faults. Third, the program is executed with each fault. SASSIFI also produces statistical reports on the outcome of each injection.

SASSIFI can inject single-bit, double-bit, random, and zero value faults into two types of destinations:

1) *The destination registers of executing instructions*. In this mode, the user can analyze the effects of transient faults that are not masked by the microarchitecture. We use this injection mode in this paper.
2) *The register file of the GPU*. In this mode, SASSIFI will inject faults into any allocated registers of the program. This mode can be used to calculate the AVF of the Register File.

In this work, as stated in III-A, we aim to evaluate the resilience of key structures in the execution datapath that do not typically have error protection. For this reason, we inject faults into the destination registers of executing instructions. In addition, we note that our fault injection campaign is run on a live GPU, while introducing minimal interference with the executing program. Our framework gives us visibility into which parts of a program are most vulnerable. We are able to conduct a comprehensive resilience study and understand how faults propagate through an application.

IV. Results

Next, we present results of our experiments in two categories. In the first category, we present the outcome for each program when we randomly change the input values, as well as when we provide biased input values of all 0's and all 1's. In the second category, we examine the effects of adjusting the size of a thread block on the resilience of the program.

TABLE I
BENCHMARKS

Benchmark	Description	Input Type and Size
BFS	Find the minimum number of edges needed to reach every vertex in an undirected graph	Graph of 4K nodes with the list of edges and their weights
Gaussian	Solves a system of equations using the Gaussian elimination method	16x16 Matrix and a 1x16 vector
Hotspot	Estimates processor temperature based on an architectural floorplan and simulated power measurements	64x64 temperature measurements and 64x64 power measurements
KMeans	Performs fuzzy k-means clustering on a set of data points	100 data points of 34 features per data point
LavaMD	Calculates particle potential and relocation due to mutual forces between particles within a large 3D space	4x4x4 boxes with their respective charges and distances between them
NW (Needleman-Wunsch)	Nonlinear global optimization method for DNA sequence alignments	2 sequences of length 2048

TABLE II
OUTCOME CATEGORIES FOR INJECTIONS

Outcome Category	Explanation
Masked	All output files match the correct output files
DUE	Detected and Unrecoverable Errors (Crash), or App does not terminate in the allocated time
Potential DUE	App exits with non-zero status, error messages were recorded
SDC	Error values at the program output

For each application, we inject 10,000 single-bit faults at the output of instructions that write to a General Purpose Register, which are sufficient to guarantee the worst case statistical error bars at 95% confidence level to be at most 1.96%. The benchmarks selected are from the Rodinia suite and are representative across a range of application domains: linear algebra, physics simulation, and fluid dynamics. A more detailed description of the benchmarks, along with the type of input that is used, is provided in Table I. The description for each outcome of an injection is provided in Table II.

A. Impact of Input Values

Figure 1 shows how the vulnerability of applications change when the input values change. We first randomly change the input values (shown by *RDM 1* and *RDM 2*, then we consider extreme cases of biased inputs, such as a matrix or an array of all 0 (*ZERO*) or 1 (*ONE*) values. We did not notice any change in vulnerability with changing input sizes of RDM 1 and RDM 2.

The input values for *RDM 1* and *RDM 2* were obtained by using the programs' own input generator. Our programs allow users to change their input values by re-running the input generator scripts. We generated the new input values and verified that both sets of values in *RDM 1* and *RDM 2* were completely different from each other.

The results show that random changes in the input values do not alter the vulnerability behavior of the applications. The difference between outcome rates using *RDM 1* and *RDM 2* is less than 2%, which is an extremely interesting result. Based on our results we can claim that, when evaluating the reliability of generic GPU applications, it is reasonably sufficient to use randomly generated inputs. However, it is worth noting that, whenever available, realistic inputs and configurations are preferred.

Our investigation shows that changing the input values for the programs under investigation does not significantly alter their control flow. In other words, when the input values are changed, but within a limited range of values, we found that the control flow graph of our applications was not altered enough to significantly affect their vulnerability. A similar trend was observed for programs running on CPUs [10]. This work reported that different program inputs lead to different vulnerability behaviors for a program, because different program inputs were inducing different execution profiles (different instruction streams) for that program.

For ZERO and ONE inputs in **bfs**, we gave all the nodes in the input graph 0 and 1 edges, respectively. Consequently, all the nodes in the graph are visited with one iteration of the GPU kernels. For regular inputs, with an average of 8 edges per node, the program converges after 8 iterations of the kernels. Faults in **bfs** are more likely to be propagated with more iterations of these kernels. To obtain a worst case result for SDC's, an experimenter should select the upper bound on the realistic number of edges per node for bfs.

For ZERO and ONE inputs in **gaussian**, we set the values in the input matrix and vector to zero and one, respectively. For these input values, there exist an infinite number of solutions to the system of equations. A fault in this program will not likely produce an incorrect result when the input is ZERO or ONE. Biased inputs for gaussian include the set of inputs for which there exist an infinite number of solutions.

For **kmeans**, ZERO and ONE inputs are data points where the values of the features are all 0 and 1, respectively. A large portion of this algorithm iteratively computes the distance of each data point to its assigned cluster center. For a ZERO input set, the distance is always 0, as are the coordinates of all the cluster centers. A fault is not likely to change the membership of the data points.

A similar case arises for **lavaMD**, where an input of ZERO means that the distances and charges of the particles are all zero. Consequently, there are no relocations of any particles and it is unlikely for a fault to change that result. The ONE

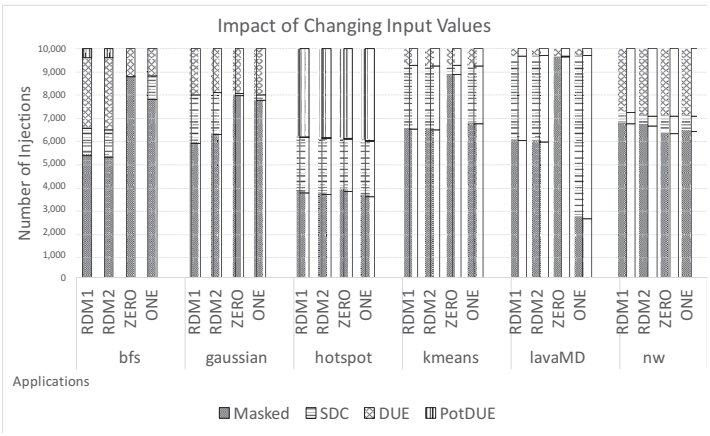

Fig. 1. Fault injection outcomes for applications with different input values. *RDM 1* and *RDM 2* are randomly generated; *ZERO* and *ONE* are input arrays/matrices where all the elements are 0 or 1, respectively.

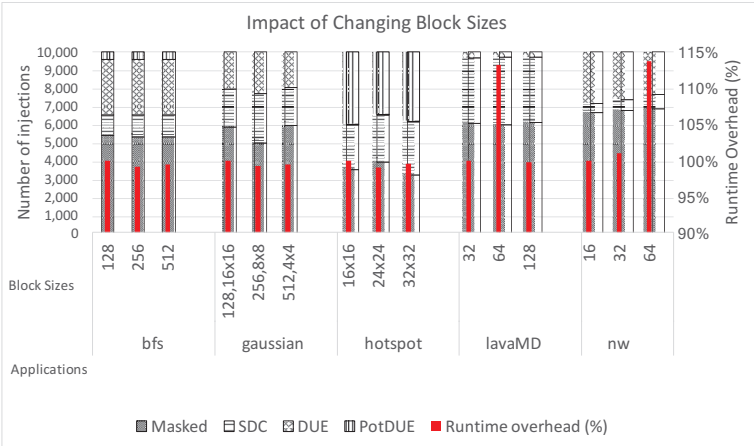

Fig. 2. Fault injection outcomes for applications when block sizes are changed. The x-axis represents the thread-block sizes (1D and 2D) for the kernels. Runtime overhead for respective block sizes over the baseline configuration (configuration with the smallest block size) is also shown for each block size.

input set, however, which assigns the distances and charges of all the particles to 1, corresponds to the maximum value for both distances and charges. This leads to a very large displacement of the particles, and more opportunities for fault propagation.

Overall, the changes in resilience caused by our biased inputs, with the exception of the ONE input set for lavaMD, correspond to an increase in the masked outcomes. This is because the biased input sets are extreme corner cases for our applications, and they happen to maximize the fault masking.

B. Effects of changes in kernel block sizes

A common practice for GPU programmers is to change the number of threads that each instance of their kernel can launch, also called the block size. The block size can significantly influence the performance of the program. A programmer often chooses the optimal block size depending on the complexity of their code, the pressure placed on system resources (e.g., registers and memory) and the compute capability of their device.

Some of the applications used in this experiment allow a user to adjust the block size of the kernel. A user can then experiment with the program by increasing (decreasing) the number of threads in each block while decreasing (increasing) the number of blocks in a grid.

Changing the block size may improve the performance of a program, depending on the capabilities of the hardware. Block size has already been demonstrated to significantly impact the reliability of GPUs [17]. Using our architectural-level fault injection framework, we aim at better understanding the reasons for the dependence of reliability on block size.

In Figure 2, we show the results of a fault injection campaign for our applications while changing the block sizes of their kernels. We also show the runtime overhead for, the respective block sizes, over the baseline configuration (i.e., the configuration with the smallest block size). Our results show that, just as performance of an application is affected

by adjusting the block size for a kernel, the resilience of an application is also affected by this configuration parameter.

In Figure 2, we see how the vulnerability of selected benchmarks changes when the thread block size changes. Similar to performance, the vulnerability of our applications does not always change with the thread block size. However, our results show that the SDC rate can change by as much as 8% for (*hotspot* and *gaussian*). These results show that changing a block size in an application can affect an application's robustness.

Our results suggest that the changes observed in reliability when we modify the block size are not only due to corruption of the scheduler/dispatcher, as proposed by Rech et al. [17]. Our fault model (Section III-A) does not account for faults occurring in the scheduler, nor the dispatcher. Changing the thread block size in a program changes the distribution of data across the Streaming Multiprocessors. This means that program vulnerability factors associated with the thread block size are related to the distribution of data in the Streaming Multiprocessors (SMs).

The degree of error propagation displayed is application dependent. In gaussian, for example, *512,4x4* means that the first kernel has 512 threads per block, while the second kernel has a thread block of 2 dimensions (4x4). We found that the thread divergence present in the first kernel code made it more likely for a fault to be masked in this kernel with a larger block size.

Our results also suggest that in reliability estimation, experimenters should not only use realistic input values, but also be mindful of the block size that is to be used with the application. In our programs, there is no direct correlation between the change in performance and the change in resilience. We plan on investigating this relationship further in our future work.

V. Conclusion

This study examines the impact of changes in the input of a program on its vulnerability. Our study also evaluates the impact of changing the thread-block sizes of a program, a common practice among developers when tuning an application for performance.

Our results show that, generally, changes in input data do not affect the vulnerability behavior of a program, as long as the data values do not represent extreme biased values such as all one's or all zero's. Our experiments on the variation of the thread block size show that the SDC rates can change by as much as 8% for some applications. A programmer needs to carefully evaluate the reliability trade-offs of tuning the performance of an application when modifying the block size of the kernels. In the future, we plan on evaluating the trade-offs between performance and reliability by adjusting the block size of our applications.

References

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

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

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

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

[5] R. Baumann, "Radiation-induced soft errors in advanced semiconductor technologies," *Device and Materials Reliability, IEEE Transactions on*, vol. 5, no. 3, pp. 305–316, Sept 2005.

[6] L. B. Gomez, F. Cappello, L. Carro, N. DeBardeleben, B. Fang, S. Gurumurthi, K. Pattabiraman, P. Rech, and M. S. Reorda, "Gpgpus: How to combine high computational power with high reliability," in *2014 Design, Automation Test in Europe Conference Exhibition (DATE)*, March 2014, pp. 1–9.

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

[8] B. Fang, K. Pattabiraman, M. Ripeanu, and S. Gurumurthi, "Gpu-qin: A methodology for evaluating the error resilience of gpgpu applications," in *Performance Analysis of Systems and Software (ISPASS), 2014 IEEE International Symposium on*, March 2014, pp. 221–230.

[9] V. Sridharan and D. R. Kaeli, "Eliminating microarchitectural dependency from architectural vulnerability," in *Int'l Symposium on High Performance Computer Architecture (HPCA-15)*, 2009, pp. 117–128.

[10] V. Sridharan and D. Kaeli, "The effect of input data on program vulnerability," 2009.

[11] T. M. Jones, M. F. P. O. 'boyle, and O. G. Ergin, "Evaluating the effects of compiler optimisations on avf," in *Workshop on Interaction Between Compilers and Computer Architecture (INTERACT-12)*, 2008.

[12] 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 of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 36. Washington, DC, USA: IEEE Computer Society, 2003, pp. 29–. [Online]. Available: http://dl.acm.org/citation.cfm?id=956417.956570

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

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

[15] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building customized program analysis tools with dynamic instrumentation," in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '05. New York, NY, USA: ACM, 2005, pp. 190–200.

[16] A. Srivastava and A. Eustace, "Atom: A system for building customized program analysis tools," *SIGPLAN Not.*, vol. 29, no. 6, pp. 196–205, Jun. 1994. [Online]. Available: http://doi.acm.org/10.1145/773473.178260

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