GUFI: a Framework for GPUs Reliability Assessment

Sotiris Tselonis Dimitris Gizopoulos

University of Athens
Department of Informatics & Telecommunications
{tseloniss, dgizop}@di.uoa.gr

***Abstract*—Modern many-core Graphics Processing Units (GPUs) are extensively employed in general purpose computing (GPGPU), offering a remarkable execution speedup to inherently data parallel workloads. Unlike graphics computing, GPGPU computing has more stringent reliability requirements. Thus, accurate reliability assessment of GPU hardware structures is important for making informed decisions for error protection.**

In this paper we focus on microarchitecture-level reliability assessment for GPU architectures. The paper makes the following contributions. First, it presents a comprehensive fault injection framework that targets key hardware structures of GPU architectures such as the register file, the shared memory, the SIMT stack and the instruction buffer, which altogether occupy large part of a modern GPU silicon area. Second, it reports our reliability assessment findings for the target structures, when the GPU executes a diverse set of twelve GPGPU applications. Third, it discusses remarkable differences in the results of fault injection when the applications are simulated in the virtual NVIDIA GPUs instruction set (ptx) vs. the actual instruction set (sass). Finally, it discusses how the framework can be employed either by architects in the early stages of design phase or by programmers for a GPU application's error resilience enhancement.

***Keywords*—**reliability assessment, fault injection, GPGPU, microarchitecture simulators

I. Introduction

An impressive set of performance-demanding applications from different research fields – biology, chemistry, finance, numerical analytics, physics, defense and intelligence, etc. – can be accelerated, harnessing the abundant computational power of modern Graphics Processing Units (GPUs). However, the ever increasing rate of hardware faults that follow silicon manufacturing advances jeopardizes the evolution of both CPU and GPU architectures. Transient, intermittent and permanent hardware faults affect the functionality and performance of modern computing systems. Even though the sources of potential failure are well understood (radiation, latent manufacturing defects, operation mode, aging etc. [1]) it is important to accurately quantify their impact on the new GPU architectures. Existing knowledge for CPUs vulnerability to hardware faults can't infer the corresponding vulnerability of GPUs because of the major differences in the design philosophy of the two architectures [2][3].

Metrics for the reliability assessment of classic computer architectures have been defined [4][5] and have already been adapted for GPUs [3][6][7]; for example the Architectural Vulnerability Factor (AVF) quantifies the probability that a transient fault in a hardware component produces a program-visible error taking into account both the hardware and the software masking.

Early and accurate reliability evaluation of GPU hardware components is critical to guide design decisions for error protection, i.e. it indicates the order of importance of structures for protection against hardware faults. The earlier the reliability evaluation is performed the better for the reduction of design time. Several hardware-based methods have been proposed for the protection of GPU hardware components against faults [8]-[14]. Selection of the most suitable protection method significantly relies on early and accurate reliability assessment.

When major design parameters (i.e. size of hardware components, workload etc.) are unknown early in the design, decision-making for protection mechanisms is a difficult task. To reduce the unknowns of this decision-making, CPU and GPU microarchitectural simulators are valuable tools employed for various early assessments of the simulated architectures. Microarchitectural simulators are commonly used for performance, power and reliability studies that can guide important design decisions. Analysis of hardware structures and software workloads employing microarchitectural simulators is an excellent choice for several reasons:

- availability in early stages of design phase,
- configurability of many hardware components,
- very high execution throughput compared to RTL simulators.

Our approach for building the proposed fault injection infrastructure on top of a microarchitectural GPU simulator allows its employment by architects and programmers early in the development stages of a new system. Architects can exploit the framework's easily reconfigurable capabilities to evaluate different GPU hardware design options in terms of both performance and reliability. On the other hand, programmers can also use the proposed framework to characterize accurately the error resilience of the applications. Overall, extending a microarchitectural simulator with detailed fault injection features renders it an appropriate framework for comprehensive reliability assessments and tangible evaluation of several GPUs error detection and correction mechanisms which can be either software or hardware based.

978-1-5090-1953-3/16/$31.00 ©2016 IEEE

II. RELATED WORK

Different aspects of the reliability of GPU architectures have been studied recently: reliability evaluation methods, soft and hard error detection and tolerance solutions that are based either on hardware or software [3][6]-[16].

GPUs reliability evaluation includes approaches that employ microarchitectural simulators to determine the Architectural Vulnerable Factor (AVF) of hardware structures using Architectural Correct Execution (ACE) analysis [3][7][13]. ACE-based analysis has already been made on top of the publicly available GPU simulators for NVIDIA and AMD architectures, GPGPU-sim [17] and Multi2sim [18], respectively. Similarly to CPUs, ACE-based estimation of GPUs AVF is very fast as it usually requires a single run of an application for the reliability characterization of a hardware component. However, ACE analysis is based on tracking of data through the entire architecture and requires substantial extensions of the simulator. Moreover, ACE analysis for CPUs is known to overestimate the vulnerability of the hardware structures [19][20].

Other reliability evaluation approaches are based on fault injection. GPUs have been evaluated in terms of the application level error resilience using fault injection methodology in real NVIDIA GPUs [15][16] but not on a microarchitecture simulator as in our case. While in this paper we characterize the vulnerability of major GPU hardware components both the works of [15] and [16] focus on the error resilience of GPGPU applications (injecting at the source/assembly code level). Both works perform injection of faults only on architectural registers that are accessed during the execution of randomly selected instructions. The framework we describe in this paper allows injection of faults in the target hardware structures at a randomly selected execution cycle. In addition to single transient faults, it supports multiple faults (both in time and space). While source-level approaches deliver useful hints for the Program Vulnerable Factor (PVF) [26], our injection infrastructure provides a complete AVF measurement.

In [6] and [14] faults are also injected in a microarchitectural simulator (Multi2Sim) which models AMD Evergreen GPU architectures running OpenCL workloads. Our framework focuses on the NVIDIA GPU architectures modeled in the GPGPU-sim simulator. To the best of our knowledge this is the first framework that provides reliability evaluation of the hardware components of CUDA enabled GPU devices, through fault injection in a microarchitectural simulator. It can support both reliability and performance measurements of GPU architectures early in the design stages and it can focus on the error resilience of the hardware components and applications.

III. BACKGROUND

A. GPU Architecture

The architecture of modern GPUs offer acceleration to general purpose applications with inherent data-level parallelism. In this paper, we focus on CUDA enabled GPUs that are based on NVIDIA's Fermi architecture [25]. NVIDIA GPU architectures consist of an array of Streaming Multiprocessors (SMs). Fig. 1 shows an overview of the GPGPU architecture and also illustrates the microarchitectural information of the SM. An SM consists of Streaming Processors (SP) for arithmetic operations, Special Function Units (SFU) for transcendental instructions and Load/Store Units (LD/ST) for memory operations. An SM contains its own Register File, Shared Memory and set of L1 Caches (Constant, Data, Instruction and Texture). The SMs work separately from each other based on their own pipeline that is managed by the Single Instruction Multiple Threads Stack (SIMT-Stack) and the Instruction Buffer that are control logic structures.

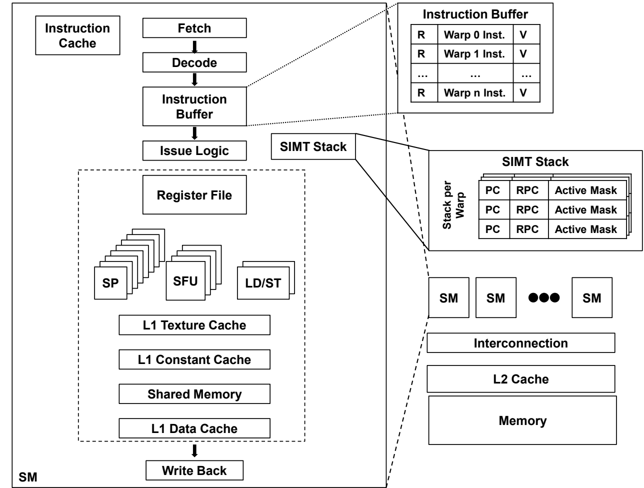

Fig. 1. NVIDIA GPGPU Architecture and a Streaming Multiprocessor

The CUDA kernels that are executed in a GPGPU Architecture consist of threads that are organized in blocks of threads or Common Thread Arrays (CTAs). CTAs are assigned to SMs during the execution of a kernel. SMs carry out the execution of a large number of threads employing the Single Instruction Multiple Threads (SIMT) Architecture which leverages instruction level parallelism for a single thread with thread level parallelism. In particular, the SMs process groups of 32 parallel threads known as warps. The threads of a warp execute the same instruction using the SPs, or the SFUs or the LD/ST units but with different data values per thread. In case of a branch instruction the threads of a warp may follow different paths. In such cases a branch divergence occurs, and threads execution is serialized with a corresponding performance impact. The SMs handle branch divergence by employing SIMT stacks dedicated to each warp. The architecture state of the threads assigned to an SM is kept on-chip during the entire lifetime of the warp. The data cache and the shared memory are partitioned among CTAs. Threads belonging to the same CTA can exchange data through the shared memory while threads of different CTAs can exchange data through the global memory. The number of CTAs and warps that run concurrently on an SM depend on: the requirements of a kernel and the compute capability of the architecture [21].

IV. METHODOLOGY

We measure the Architectural Vulnerable Factor (AVF) of the hardware structures of a NVIDIA GPU architecture, exploiting the features of the developed framework which supports fault injection in the GPU register file, the shared memory, the instruction buffer (Valid Bit) and the SIMT stack (PC field, RPC field, Active Mask). Of course, the framework can be extended to support injections in other hardware components as well.

The AVF measurements reported by fault injection framework are calculated by dividing the number of fault injection experiments on a hardware component that result in application failure (the types of failing behavior are explained later) by the total number of injected faults.

$$AVF = \frac{\#\text{Fault Injections leading to Failure}}{\#\text{Total Fault Injections}}$$

A slightly modified form of the measurement applies to our framework for the register file and the shared memory considering the particular modeling of these components in the simulator. In the GPGPU-sim model each thread of a kernel constructs and accesses its own register file and doesn't reserve a set of registers from a physical register file that would be constructed once for each SM (this would have been a more convenient model for reliability assessment). Moreover, in GPGPU-sim each CTA that is assigned to an SM uses its own instance of shared memory and doesn't occupy a subset of a unified shared memory within an SM (this would have been also a better model for injections). To overcome these two modeling issues of GPGPU-sim, in our analysis for the register file and the shared memory, we define a derating factor for each structure **df_reg** and **df_smem.** In order to estimate the final AVF of the register file and the shared memory, we have to multiply each factor with the relative percentage of failures as explained below. (Alternatively, we could have re-implemented register file and shared memory for the needs of our evaluation; but we wanted to minimally intrude in the simulator and reduce development time. The AVF estimation result would have been the same.)

The **df_reg** is an intuitive quantification of the fraction of a GPU physical register file that we can target in a given cycle during the execution of a given kernel. It depends on:

- #REGS_PER_THREAD that is the number of registers that a thread uses during the execution of a kernel,
- #THREADS that is the number of running threads in an SM during the execution of a given kernel,
- #REGFILE_SIZE_SM that is the number of registers in the register file of an SM.

$$df_reg = \frac{\#\text{REGS_PER_THREAD} \times \#\text{THREADS}}{\#\text{REGFILE_SIZE_SM}}$$

The **df_smem** is an intuitive quantification of fraction of shared memory that we can target in a given cycle during the execution of a given kernel. It depends on:

- #CTA_SMEM_SIZE that is the size of shared memory that is used by a CTA of a kernel,
- #CTAS that is the number of running CTAs in an SM during the execution of a given kernel,
- #SMEM_SIZE that is the size of shared memory in an SM.

$$df_smem = \frac{\#\text{CTA_SMEM_SIZE} \times \#\text{CTAS}}{\#\text{SMEM_SIZE}}$$

We note that the derating-factor for the register file is applied only to the failure rate of register file that comes from fault injection when the simulator runs actual GPU assembly language programs (the SASS ISA of NVIDIA GPUs). If it runs PTX code that is an intermediate virtual ISA then the number of virtual registers that a thread uses are more than the architectural registers that would be assigned to the given thread.

In addition to the AVF of the target hardware components, we compute the Failures In Time (FIT) for a given structure based on its size in bits, the AVF and a fixed intrinsic FIT rate (which we assume to be 0.001 FIT/bit; any arbitrary rate can be used).

$$FIT_{STRUCTURE} = AVF \times \text{INTRINSIC FIT} \times \text{SIZE}$$

Thus, we can quantify the FIT rate of a GPU architecture based on the FIT rates for all structures in it.

$$FIT_{GPU} = \sum_i FIT_i$$

Working at the microarchitecture level, our framework can be employed to characterize any given GPGPU architecture both in terms of reliability and performance. An application's Instructions Per Cycle (IPC – the usual performance metric) that GPGPU-sim reports can be divided with our FIT measurements (reliability metric); together IPC and FIT can provide a two-in-one metric for performance and reliability the Mean Instructions To Failure (MITF). A similar combined metric is used in [7] but instead of FIT they use AVF which doesn't consider the size of a structure which our approach does.

$$MITF \sim \frac{IPC}{FIT}$$

We believe that this combined information is very useful for architects in early stages of design phase because they can compare different GPGPU architectures both in terms of performance and reliability, and singling out the one with the higher IPC to FIT ratio. Programmers can also use the proposed framework to identify the error resilience of CUDA kernels that are executed in the context of an application. Afterwards, they can apply software based techniques in order of priority to enhance the error resilience of more vulnerable kernels while maintaining the required performance levels.

V. EXPERIMENTAL FRAMEWORK

We present GUFI (**GP**GPU-sim **F**ault **I**njector) a framework for reliability studies of GPU architectures hardware and workloads.

A. Fault Injection Infrastructure

GUFI is a complete framework for reliability evaluation of GPU architectures that runs over a well-known simulator of GPUs architectures: GPGPU-sim [17]. The simulator provides detailed microarchitectural models of NVIDIA GPUs architectures and runs CUDA as well as OpenCL workloads. We focused our study on CUDA applications. Moreover, we exploit the capability of the simulator to run either Parallel Thread Execution assembly (PTX) or SASS assembly [21] and we can inject faults in the hardware components of any simulated architecture for both instruction sets.

In order to carry out fault injection campaign in a hardware structure we have to go through the following steps. Initially, we run the application once in order to profile it; after completion a set of files are produced. These files contain useful information about kernels that run in the context of an application like:

- the time interval of a kernel's execution during the entire application,
- information about the registers (name, size) that a thread requires,
- the number of threads that may run concurrently for a given kernel's execution on an SM,
- the number of CTAs that may be assigned concurrently for a given kernel's execution on an SM,
- the size of shared memory that each CTA may occupy in the context of its execution.

The outcome of an application's profiling step is organized on the basis of invocations of each kernel and is the input of the second step that is the creation of masks for all target structures from a **mask-generator**. At this point, fault masks are generated for all targeted structures and for all invocations of all kernels in the context of the execution of an application. The number of masks is defined by the user of GUFI and is the same across all invocations of any kernel.

The fault injection campaign in a hardware component can be set either for a user defined kernel invocation (mode 1) or for the whole application (mode 2). Mode 1 focuses on the reliability evaluation of a specific invocation of a kernel that is launched in the context of an application while mode 2 enables users to make a comprehensive reliability evaluation of the full application in any structure and architecture. The user needs to follow the above steps once for any architecture. Overall, the generation of masks for all target hardware components is scalable to all architectures that the simulator supports, either on PTX or SASS mode.

A GUFI user can do massive fault injection experiments working either on mode 1 or mode 2 as soon as fault masks have been created and kept on **mask_dir** as illustrated in Fig. 2. Both modes employ a bash script – **Campaigner** – that is responsible for the golden run of the application and all fault injection experiments. The Campaigner and all the required files for the simulation (i.e. configuration files of the simulator, executable of application etc.) reside in the campaign's root directory – **app_dir**.

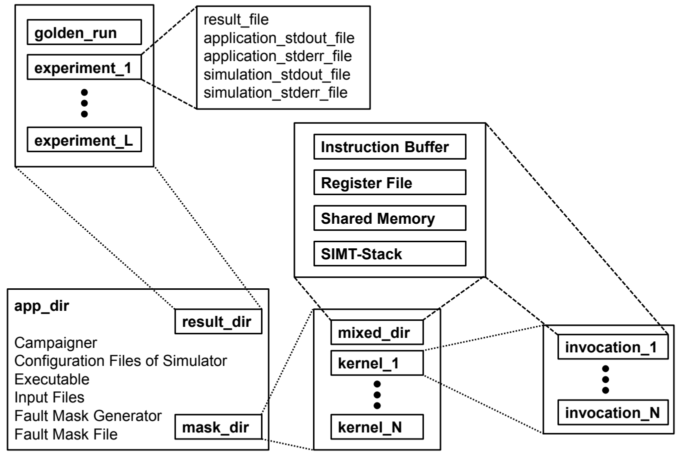

Fig. 2. The directory of fault injection campaign

Initially, the campaigner launches a golden run and moves all output files to a dedicated directory – **golden_run**. Subsequently, it runs a (user defined) number of fault injection experiments that cannot exceed the number of available fault masks. In the context of each fault injection experiment, the Campaigner copies a fault file from the proper directory within **mask_dir** to **app_dir**. Subsequently, the campaigner activates the fault injection experiment. The simulator reads the fault file at the beginning of simulation and injects the fault in the target hardware component at the activation cycle. The fault can be either in a single bit or in multiple bits. The fault mask and activation cycle are defined in the fault file. All files that are produced in each fault injection experiment are moved to a dedicated directory for the given fault injection experiment – **experiment_1** to **experiment_L**.

The files produced in the context of an application are the following:

- **result file**: Output that an application produces and writes in a file.
- **application stdout file**: Messages that an application writes in stdout.
- **application stderr file**: Messages that an application writes in stderr.
- **simulation stdout file**: Messages that an application and simulator write in stdout.
- **simulation stderr file**: Messages that an application and simulator write in stderr.

After completion of a fault injection campaign, a parser processes the output of all experiments one by one. The parser classifies each experiment as Masked, Silent Data Corruption (SDC), or Detectable Unrecoverable Error (DUE), comparing the files of an experiment with the ones of golden run. The classes of fault effects are:

- **SDC**: The application is completed but the result file (if any) or the application stdout file differ from the golden ones and there's no indication that an error occurred in application stderr and simulation stderr files.

- **DUE**: Simulator or application reaches an abnormal state and experiment's execution is forced to completion, printing a warning in application stderr or simulation stderr files.
- **Masked**: The application is completed, there's no indication that an error occurred and the result file (if any) or the application stdout file are identical with the golden ones. Moreover, if a fault strikes an unused resource then the fault injection experiment is characterized as Masked i.e. the registers of an idle SM at the time of fault injection. Simulator has been extended to detect such cases and a related message is printed on the simulation stderr file. Thus parser can distinguish such cases from DUE.

In mode 1, all experiments are done in a specific kernel invocation that is defined by the user. Thus, an experiment results in a set of effects for the injected faults: the Masked, SDC and DUE measurements. In mode 2, each kernel execution has its set of Masked, SDC and DUE counters. Thus, an experiment delivers the appropriate set of measurements, based on the invocation of the kernel that it belongs to. In this paper, we report the results for mode 2 because it provides a broader view: it enables us to evaluate the reliability of the targeted hardware component for an entire application but it also delivers interesting findings across all its kernels and their invocations.

B. Applications

In the context of our reliability evaluation, we use 12 different applications from ispass2009-benchmarks [17], NVIDIA CUDA SDK package [21] and Rodinia benchmark suite [22]. In TABLE I we report the simulation time of the applications, the breakdown of applications into kernels, the number of invocations of each kernel, the number of CTAs (gridDim) and the number of threads per CTA (blockDim) in each kernel. We shortly describe each application and we also report the input data set below.

Breadth-First Search (BFS): BFS is a breadth-first search algorithm which traverses all the connected components in a graph. We use BFS with the default input of 4096 nodes.

Compute Coulombic Potential (CCP): CCP computes the coulombic potential at each grid point over one plane in a 3D grid in which point charges have been distributed. CCP processes the default data that are produced at the beginning of its execution.

Gaussian Elimination (GE): GE is an algorithm for solving systems of linear equations. We employ GE to solve a system of 30 linear equations.

Hot Spot (HS): HS estimates processor temperature based on an architectural floor plan and simulated power measurements. We try HS with the default input for temperature and power values that are organized on two individual 512x512 matrices.

K-Means (KM): KM is a data-mining algorithm that features high degree of data parallelism. We run KM with 1000 objects and each object consists of 34 features.

Laplace (LPS): LPS implements a Laplace discretization on a 3D structured grid. We run LPS with the default input data that are produced at the beginning of its execution.

Lower Upper Decomposition (LUD): LUD is an algorithm that calculates the solutions of a set of linear equations. We run LUD with the default input data that are produced at the beginning of its execution.

Merge Sort (MS): MS implements a merge-sort for sorting batches of short- to mid-sized (key, value) array pairs. We run MS with for an array of 16384 pairs.

Needleman-Wunsch (NW): NW is a nonlinear global optimization method for DNA sequence alignments. We run NW with the default input dataset that is produced at the beginning of its execution.

Pathfinder (PATHF): PATHF finds a path on a grid from the bottom to the top with the smallest accumulated weights and each step of the path moves straight ahead or diagonally ahead. We run PATHF with 10000 rows, 100 columns and 20 height.

Scalar Product (SP): SP computes the scalar product of vectors. We run SP with the default input dataset produced at

TABLE I. Applications

Application	Suite	Simulation time (s)		Invocations	gridDim	blockDim
BFS	[22]	7	_Z6KernelP4NodePiPbS2_S2_S1_i	8	8	512
			_Z7Kernel2PbS_S_S_i	8	8	512
CCP	[17]	87	_Z7cenergyifPf	1	8x32	16x8
GE	[22]	5	_Z4Fan1PfS_ii	29	1	512
			_Z4Fan2PfS_S_iii	29	8x8	4x4
HS	[22]	112	_Z14calculate_tempiPfS_S_iiiiffffff	1	43x43	16x16
KM	[22]	69	_Z11kmeansPointPfiiiPiS_S_S0_	13	2x2	256
			_Z14invert_mappingPfS_ii	1	4	256
LPS	[17]	74	_Z13GPU_laplace3diiiiPfS_	1	4x25	32x4
LUD	[22]	172	_Z12lud_diagonalPfii	16	1	16
			_Z12lud_internalPfii	15	15x15 to 1x1	16x16
			_Z13lud_perimeterPfii	15	15x1 to 1x1	32
MS	[21]	32	_Z21mergeSortSharedKernelILj1EEvPjS0_S0_S0_j 1 1	1	16	512
			_Z25generateSampleRanksKernelILj1EEvPjS0_S0_jjj	4	1	256
			_Z26mergeRanksAndIndicesKernelPjS_jjj	8	1	256
			_Z30mergeElementaryIntervalsKernelILj1EEvPjS0_S0_S0_S0_S0_jj	4	128	128
NW	[22]	67	_Z20needle_cuda_shared_1PiS_iiii	32	1x1 to 32x1	16
			_Z20needle_cuda_shared_2PiS_iiii	31	31x1 to 1x1	16
PATHF	[22]	48	_Z14dynproc_kerneliPiS_S_iiii	5	47	256
SP	[21]	31	_Z13scalarProdGPUPfS_S_ii	1	128	256
VADD	[21]	2	_Z6VecAddPKfS0_Pfi	1	204	256

the beginning of its execution.

Vector Add (VADD): VADD adds two vectors. We run VADD with the default input dataset produced at the beginning of its execution.

VI. EXPERIMENTAL RESULTS

In this section we present the results of our reliability and performance evaluation for all applications of the experimental analysis. Apart from reporting the overall application vulnerability, we also break it down into kernels' vulnerability for all the hardware structures of our study.

We use GUFI for fault injection of transient faults in the hardware components of a CUDA enabled GPU architecture. We run fault injection campaigns for the register file (regfile), the shared memory (smem), the active mask of SIMT-stack (simt-ams), and the valid bit of instruction buffer entry (ibuffer-entry) for the 12 GPGPU applications discussed in the previous section[1]. For the register file, we study its reliability when the simulated architecture runs PTX ISA (regfile - ptx) and when it runs SASS ISA (regfile - sass). We study the error resiliency of shared memory only for the applications that use it to exchange data among the threads of the same CTA (block) in a kernel. Overall, we carried out 55 fault injection campaigns (mode 2) and each fault injection campaign consists of 2000 injections experiments (fault injection campaigns on regfile-ptx, regfile-sass, simt-stack, ibuffer for 12 applications and fault injection campaigns on smem for 7 applications that use it)[2]. Fault injection experiments were uniformly distributed to kernels' executions. Our GUFI framework can be used for fault injection in any architecture that the GPGPU-Sim simulator models; the following results come from experiments in the simulated model that resembles GeForce GTX480 graphics processor configuration based on Fermi Architecture.

Even though commercial NVIDIA GPU chips of Fermi Architecture incorporate ECC protection in the register file and the shared memory, GPGPU-Sim does not model it. The results of our fault injection campaigns for single-bit transient faults on the unprotected GPU architecture provide insights about its inherent fault tolerance. Such information is very important for the architects in early design stages. Knowledge about the contribution of each hardware component in the overall FIT rate of an unprotected GPU along with area and power costs (provided from other tools) assists the architects in making informed decisions about the most suitable error protection schemes. For example, our measurements (see Fig. 9) using GUFI (for a fixed intrinsic FIT rate of 0.001 FIT/bit) report a GPU FIT rate of 1760 FIT for the HS benchmark with all studied components unprotected. When the shared memory gets protected for single bit errors the overall GPU FIT rate for HS benchmark is reduced to 1611 FIT but when the register file gets protected the GPU FIT rate is only 163. If the designer decides to protect both then the overall GPU FIT rate will be only 14 FIT. On the average across all benchmarks, the FIT rate of an unprotected GPU is 1063 FIT; one with only the shared memory protected is 963 FIT; when only the register file is protected it is 108 FIT. If both components are protected the average FIT rate for the GPU across all benchmarks is only 8 FIT.

A further step is such an analysis can be the modeling of an ECC scheme such as the typical SECDED (Single Error Correction Double Error Detection) in a hardware structure of GPGPU-sim (current version of GUFI does not support any ECC scheme). This can be easily realized and obviously an injection campaign for single-bit transient faults will result in all faults Masked (actually the injection campaign is not needed since the single-bit faults are by design corrected by the ECC hardware). In such a case, GUFI can be used for multiple (double, triple, etc.) bit transient faults injections to see how many stay uncorrected by the SECDED ECC and what is their effect; multi-bit injection is a feature that our framework also supports.

The configuration of the simulated architecture is shown in TABLE II.

TABLE II. Simulated NVIDIA GPU Architecture (GeForce GTX480)

SMs	15
Warp size	32
Maximum Threads per SM	1536
Maximum CTAs per SM	8
Registers per SM	32768
Shared Memory per SM	48 KB
Memory Controllers	6

The IPC as well as the average warp occupancy of the applications under study are shown in TABLE III. The average warp occupancy is an intuitive metric for an application that consists of multiple kernels. We estimate the warp occupancy of a Streaming Multiprocessor for each kernel that runs on the GPU. Afterwards, we weight the warp occupancy with the ratio of the kernel's execution time over the application's execution time and then add the individual weighted warp occupancies of all kernels.

TABLE III. Performance metrics

Application	IPC	Average Warp Occupancy
BFS	20.57	33.00%
CCP	396.41	67.00%
GE	16.44	20.74%
HS	591.45	50.00%
KM	91.99	17.00%
LPS	445.01	58.00%
LUD	53.56	14.08%
MS	207.88	44.58%
NW	23.16	3.19%
PATHF	475.03	67.00%
SP	329.75	100.00%
VADD	101.46	100.00%

The effects of faults across the application differ significantly. Applications BFS, GE and NW have a higher tolerance than the others in all hardware structures because

[1] The configuration of the machine which we use for all our experiments is the following: processor - Intel Core i7-4771 @ 3.50GH, memory 16 GB, operating system – Ubuntu 15.04.

[2] This number comes from the formula of [23] and results in a statistical safe number of fault injection with confidence level 99% and error margin less than 3%.

their input dataset results in small pressure of the hardware components. Thus, the majority of fault injection experiments hit idle resources. In TABLE III, the low average warp occupancy and the noticeable low IPC of BFS, GE and NW indicate that their input data sets impose low pressure to the hardware components. Namely, their IPC is low because for the given input data sets they execute kernels that have just a few threads that are grouped in a few warps. Therefore, the small number of concurrently running warps cannot hide the latency of long operations; thus the low IPC values. BFS application consists of 8 invocations of two kernels and neither of them stretches the architecture as the average warp occupancy is 33% and the IPC is only 20.57. GE application consists of 29 invocations of two kernels and neither of them stretches the architecture as the average warp occupancy is 20.74% and the IPC is 16.44. NW application consists of 32 and 31 invocations of two kernels and neither of them stretches the architecture as the average warp occupancy is only 3% and the IPC is only 23.16.

In Fig. 3 - Fig. 6, NW and SP stand out among applications because they feature respectively the highest and the lowest Masked faults rates: regfile – ptx runs (100%, 92.15%), regfile – sass runs (99.35%, 70.65%), simt – ams (100%, 97.75%) and ibuffer - entry (97.10% , 19.85%).

The SDC rates are higher than DUE rates in all applications except for KM and MS in regfile – ptx (Fig. 3) and regfile – sass (Fig. 4). This is because some faults in KM were detected as they caused segmentation fault and some faults in MS were detected because the application uses a software detection mechanism that can detect some errors. In [16] MS is also reported with higher DUE than SDC rates.

There is a common trend in the behavior of applications when we inject faults in register file running ptx Fig. 3 and sass mode Fig. 4. However the Masked rates in all applications are always higher in the case of ptx mode. The difference ranges from 0.65 percentile units (p.u.) for NW to 21.50 p.u. for SP and is due to the fact that a thread in ptx mode of simulation uses more registers than in sass mode of simulation and thus a fault in a register in ptx mode is easier to be Masked than one in sass mode. Thus, ptx based reliability measurements induce some underestimation of the actual hardware vulnerability; this is expected since ptx is a virtual layer above the actual hardware architecture.

Fig. 5 illustrates the results of fault injection in the active mask of SIMT-Stack. The injected faults can strike every entry of the stack that is dedicated to a warp and not just the entries that are certainly used. This justifies the high Masked rates that the active mask of SIMT-Stack features. Actually, the usage of entries of SIMT-Stack is related to the branch divergence that hosted warps feature. For example, given a uniform distribution of targeted entries a warp with no branch divergence would have higher Masked rates than another warp with higher branch divergence. We also observe that none of the faults in SIMT-Stack causes SDC (only DUEs happen) and this is expected since SIMT-Stack involves control logic.

Fig. 6 shows the results of fault injection in the valid bit of the instruction buffer. In the instruction buffer the Masked rates are less dominant than in the other structures. In the case of real hardware, a fault in the valid bit of instruction buffer may result in SDC error (i.e. some arithmetic or memory instruction skip their issue), yet the simulator usually reaches an abnormal state and raises an assertion (which we classify in the DUE class). In any case, the faults in instruction buffer entries are more severe than in other structures because there is a very small probability of masking.

Fig. 7 illustrates the results of fault injection in the shared memory. We inject faults in the shared memory only for the applications that use it for communication among threads of the same CTA (block); only the 7 of the 12 applications that use shared memory for this purpose are shown in Fig. 7. In Fig. 7, PATHF application features the highest Masked rates (98.55%) while the Masked rates of LUD (75.50%) and SP (75.95%) are very close and make up the bottom boundary.

The SP application is the most vulnerable in all structures and this is expected because it stretches the overall architecture according to average warp occupancy (100%) in TABLE III. The application VADD also follows this trend for regfile – ptx, regfile – sass mode and ibuffer-entry.

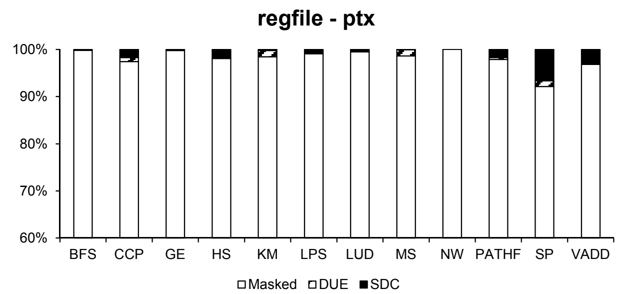

Fig. 3. Fault effects classification results for fault injection in regfile in ptx mode.

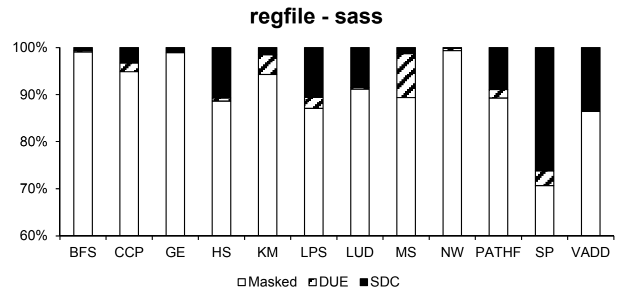

Fig. 4. Fault effects classification results for fault injection in regfile in sass mode.

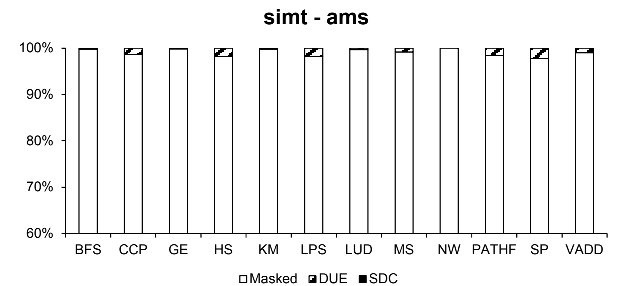

Fig. 5. Fault effect classification results for fault injection in active mask of simt-stack in sass mode.

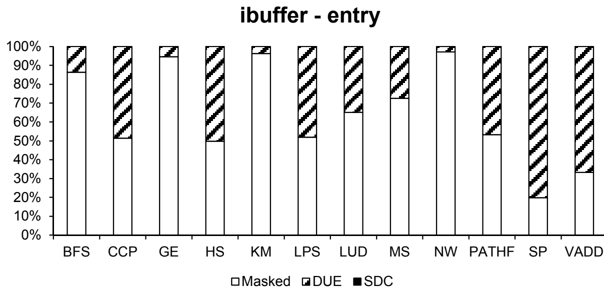

Fig. 6. Fault effects classification results for fault injection in valid bit of instruction buffer entry in sass mode.

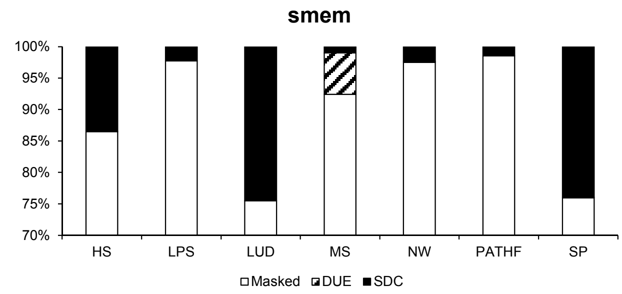

Fig. 7. Fault effects classification results for fault injection in shared memory in sass mode (the remaining applications BFS, CCP, GE, KM and VADD do not use the shared memory).

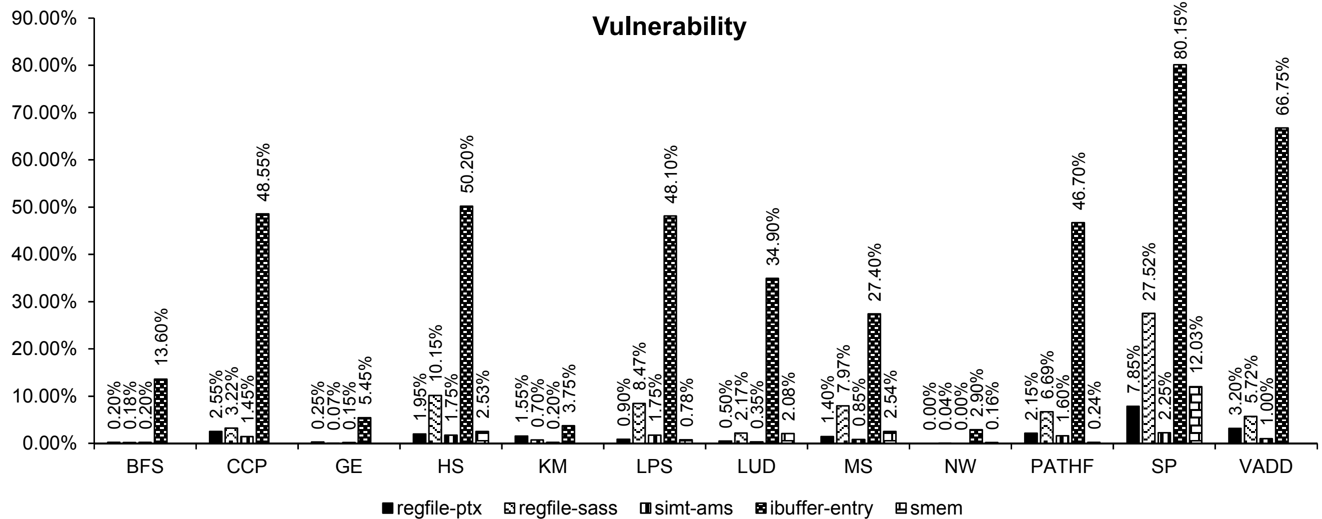

Fig. 8. Vulnerability of the hardware components for all applications (sum of the SDC and DUE classes). This measurement corresponds to the AVF of each component for each application.

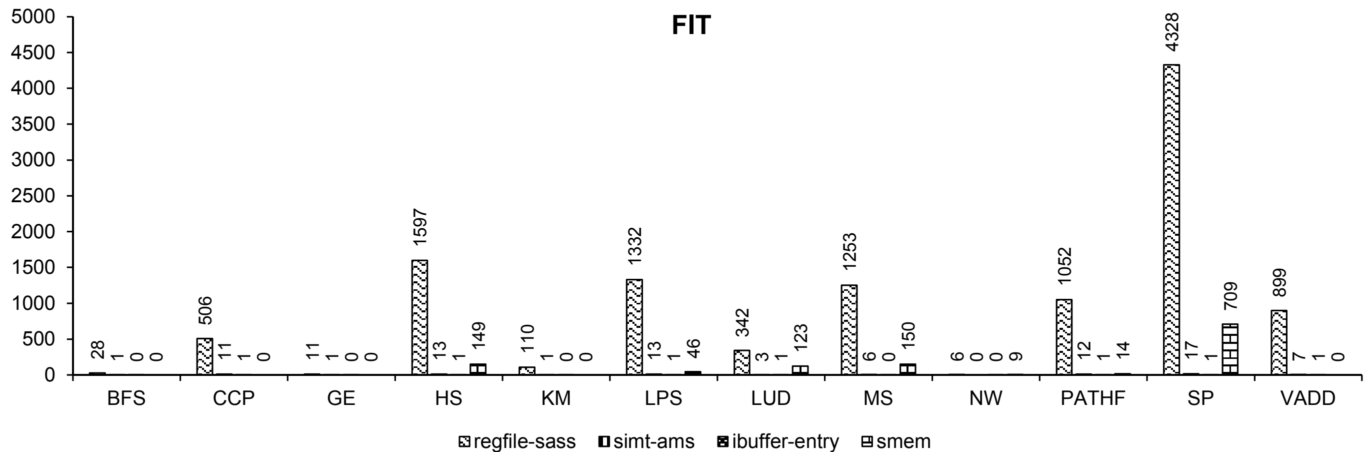

Fig. 9. Failures In Time (FIT) due to faults in hardware components for each application (0 denotes small FIT rates; we round the rates to the closest integer).

Fig. 8 shows the vulnerability of all hardware structures and for all applications (sum of the non-Masked fault effect classes; corresponds to the AVF of each structure). Derating factor applies solely to fault injection results of register file in sass mode and shared memory as explained earlier in the paper. Both the df_reg and df_smem derating factors are the percentage of registers that can be injected during the execution of a kernel. Applications that consist of multiple kernels' executions have a pair of derating factors (df_reg and df_smem) for each kernel execution. The value of each derating factor in the vulnerability of application is proportional to the ratio of execution time of each kernel invocation to the total execution time of application. Moreover, in Fig. 8 we observe that all applications except for BFS and GE feature higher register file vulnerability in the sass mode than in ptx mode. We expect that both BFS and GE with bigger input datasets would stretch the register file and the overall architecture thereby making the rule of underestimated vulnerability of registerfile in ptx mode apply to these applications too. Of course, the actual vulnerability of the register file is defined by the results of fault injection in the sass mode.

Fig. 9 shows (rounded to the closest integer) the FIT rates of all hardware structures for all applications of our study. With respect to the average values the order of structures based on their (per-bit) vulnerability (from Fig. 8) is as follows: instruction buffer (35.70%), register file (6.07%), shared memory (2.91%) and SIMT-stack (0.96%). However, due to the different sizes of the components the order of the average values of FIT differs from the vulnerability order and is as follows: register file (955 FIT), shared memory (100 FIT), simt-stack (7 FIT) and instruction buffer (1 FIT). This is expected since even though both vulnerability and size of a structure impact on FIT, the impact of the size is dominant. Thus a structure with high vulnerability but small size such as the instruction buffer results in less FIT than another structure with less vulnerability and bigger size like register file. The order of structures according to their contribution in FIT explains why newer GPGPU Architectures (i.e. Fermi Architecture) feature ECC protection on register file and shared memory.

In Fig. 10, we summarize the combined metric of IPC over FIT for all applications and the average ratio which is 0.56, for the given set of 12 applications running on the given GPGPU Architecture without any fault protection. Hence, the proposed framework can investigate the trade-off between performance and reliability among different implementations, exploiting GPGPU-Sim that is able to model different GPGPU models (GTX480, QuadroFX5600, QuadroFX5800, TeslaC2050) as well as extra implementations (such as hardware based protection techniques) and the potential of the proposed fault injection infrastructure. If, for example, the register file and shared memory are SECDED-protected then all failures would stem from the simt-stack and the instruction buffer. The average IPC/FIT would then increase from 0.56 shown in Fig. 10 to 72.44.

TABLE IV. Breakdown of the applications' vulnerability (AVF) in the individual kernels' vulnerability.

Application Name	Kernel Name	Breakdown Application Vulnerability				
		regfile ptx	*regfile sasss*	*simt stack*	*ibufffer*	*shared memory*
BFS	_Z6KernelP4NodePiPbS2_S2_S1_i	0.15%	0.18%	0.20%	11.40%	0.00%
	_Z7Kernel2PbS_S_S_i	0.05%	0.00%	0.00%	2.20%	0.00%
CP	_Z7cenergyifPf	2.55%	3.22%	1.45%	48.55%	0.00%
GE	_Z4Fan1PfS_ii	0.00%	0.00%	0.00%	0.75%	0.00%
	_Z4Fan2PfS_S_iii	0.25%	0.07%	0.15%	4.70%	0.00%
HS	_Z14calculate_tempiPfS_S_iiiiffffff	1.95%	10.15%	1.75%	50.20%	2.53%
KM	_Z11kmeansPointPfiiiPiS_S_S0_	1.15%	0.56%	0.20%	2.85%	0.00%
	_Z14invert_mappingPfS_ii	0.40%	0.14%	0.00%	0.90%	0.00%
LPS	_Z13GPU_laplace3diiiiPfS_	0.90%	8.47%	1.75%	48.10%	0.78%
LUD	_Z12lud_diagonalPfii	0.05%	0.00%	0.00%	1.00%	0.03%
	_Z12lud_internalPfii	0.35%	2.04%	0.25%	11.05%	0.85%
	_Z13lud_perimeterPfii	0.10%	0.13%	0.10%	22.85%	1.20%
MS	_Z21mergeSortSharedKernelILj1EEvPjS0_S0_S0_j 1 1	1.30%	6.60%	0.20%	14.30%	1.55%
	_Z25generateSampleRanksKernelILj1EEvPjS0_S0_jjj	0.05%	0.01%	0.00%	1.20%	0.00%
	_Z26mergeRanksAndIndicesKernelPjS_jjj	0.00%	0.01%	0.00%	0.85%	0.00%
	_Z30mergeElementaryIntervalsKernelILj1EEvPjS0_S0_S0_S0_S0_jj	0.05%	1.35%	0.65%	11.05%	0.99%
NW	_Z20needle_cuda_shared_1PiS_iiii	0.00%	0.03%	0.00%	1.95%	0.07%
	_Z20needle_cuda_shared_2PiS_iiii	0.00%	0.01%	0.00%	0.95%	0.09%
PATHF	_Z14dynproc_kerneliPiS_S_iiii	2.15%	6.69%	1.60%	46.70%	0.24%
SP	_Z13scalarProdGPUPfS_S_ii	7.85%	27.52%	2.25%	80.15%	12.03%
VADD	_Z6VecAddPKfS0_Pfi	3.20%	5.72%	1.00%	66.75%	0.00%

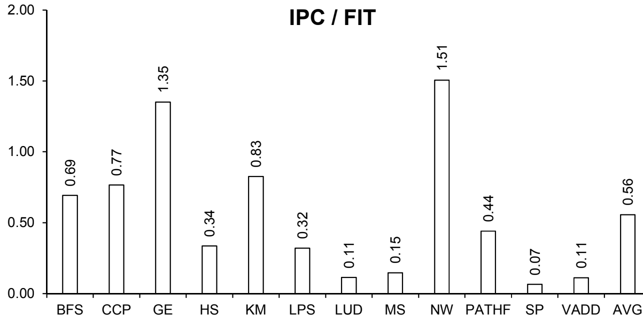

Fig. 10. IPC to FIT

In addition, programmers can use the GUFI framework to quantify the inherent error resiliency of applications. For example, in TABLE IV we break down the vulnerability of each application for all structures into the vulnerability of the individual kernels. Given that the failures mainly manifest due to faults in the register file and the shared memory (due to their bigger sizes) we have to focus on the vulnerability of the register file (sass mode) and the shared memory for each kernel. If an application consists of many kernels then enhancing only the error resilience of the most vulnerable kernels will result in more error resilient application. Hence, in case of applications consisting of many kernels, applying partial protection will enhance the error resilience of the application with less performance degradation than the one if there would be protection for all the kernels of an application.

VII. Discussion

TABLE V summarizes the sizes of the hardware components of the simulated Fermi Architecture and indicates whether GUFI supports fault injection on them. We use ★★★ when GUFI already supports fault injection for a hardware component, ★★ when GUFI partially supports fault injection in a hardware component (a subset of its fields) and ★ when GUFI needs some extra development effort to support fault injection on it.

For most components, the sizes we report come from NVIDIA resources [21]; for those not included in such documents we estimate their sizes from the GPGPU-Sim configuration. For example, the Fermi Architecture's SM can be assigned concurrently with 48 warps and each warp uses one SIMT Stack (24 bits for PC field, 24 bits for RPC field, 32 bits for the Active Mask) while the depth of a stack cannot exceed the warp size. Thus, the SIMT stacks roughly occupy 15.36 KB within an SM and 230.4 KB for the entire GPU chip. In addition, GPUWattch [24] that is integrated into GPGPU-Sim models Memory Coalescing Arrays into memory controllers and all the Memory Coalescing Arrays roughly occupy 6 x 3 KB.

Moreover, in GPGPU-Sim, the fields of data in L1 and L2 caches are not modeled. This is also the case for the fields of data operands in the operand collector units. Although, the functional units cover a substantial area in a GPU [25], GUFI does not support fault injection on them. In total, the GUFI framework covers a significant portion of the array-based hardware components of a GPU: about 64% of the area of hardware components with documented sizes is covered by GUFI. Future extensions of the framework will cover other hardware components (marked with a ★).

TABLE V. Coverage of GUFI on the GPU hardware components

Component	Size	Support
Register file	15 x 128 KB	★★★
Operand Collector	15 x 14 x 3 x 128 B	★
Shared Memory	15 x 48 KB	★★★
L1D	15 x 16 KB	★
Constant Cache	15 x 8KB	★
Texture Cache	15 x 12 KB	★
L2 Cache (shared)	1 x 768 KB	★
Instruction Buffer	15 x 48 x 2 x (2 bits + 128 B)	★★
SIMT Stack	15 x 48 x 32 x 10 B	★★★
Memory Coalescing Arrays	6 x 3 KB	★
Functional Units	480 CUDA cores	★

VIII. CONCLUSIONS

We have introduced GUFI, a detailed fault injection framework built on top of a state of the art microarchitectural simulator of GPGPU architectures, GPGPU-sim. The proposed framework supports reliability evaluation measurements based on fault injection in several critical hardware components of GPU architectures: register file, shared memory, SIMT stack and instruction buffer. We made a comprehensive reliability evaluation of the target hardware components employing 12 GPGPU applications from different suites. Our study highlights meaningful differences on the results of fault injection in register file depending on the running ISA (the virtual ptx vs. the actual sass). The proposed infrastructure can be used extensively by the GPU reliability research community architects and programmers. Architects in early stage of design phase will be able to evaluate different simulated models of GPGPU architecture as well as different hardware based protection techniques both in terms of performance and reliability. Programmers can also use the proposed framework to break the vulnerability of an entire application down to the vulnerability of its kernels. Software based error detection/correction techniques focusing on the most vulnerable kernels can be then employed to improve resilience.

ACKNOWLEDGMENT

This work is supported by the 7th Framework Program of the European Union through the CLERECO Project, under Grant Agreement 611404.

REFERENCES

[1] S.R.Nassif, N.Mehta, Y.Cao, "A resilience roadmap", DATE ‘10.
[2] D. Kirk, W. W. Hwu, “Programming Massive Parallel Processors”, 2nd edition.
[3] H. Jeon, M. Wilkening, V. Sridharan, S. Gurumurthi ,G. Loh, "Architectural vulnerability modeling and analysis of integrated Graphics Processors", SELSE ‘13.
[4] S. S. Mukherjee, C. T. Weaver, J. Emer, S. K. Reinhardt, T. Austin, “A systematic Methodology to Compute the Architectural Vulnerability Factors for a High-Performance Microprocessor”, MICRO ‘03.

[5] A. Biswas, P. Racunas, R. Cheveresan, J. Emer, S. S. Mukherjee, R. Rangan, “Computing architectural vulnerability factors for address-based structures”, ISCA ‘05.
[6] N.Farazmand, R.Ubal, D.Kaeli, "Statistical fault injection-based AVF analysis of a GPU architecture", SELSE ‘12.
[7] J.Tan, N.Goswami, T.Li, X.Fu, "Analyzing soft-error vulnerability of GPGPU microarchitecture", IISWC ‘11.
[8] J.W.Sheaffer, D.P.Luebke, K.Skadron, "A hardware redundancy and recovery mechanism for reliable scientific computation on graphics processors", SIGGRAPH ‘07.
[9] R.Nathan, D.Sorin, "Argus-G: A low-cost error detection scheme for GPGPUs", WRA ‘10.
[10] A.Durytskyy, M.Zahran, R.Karri, "Improving GPU robustness by making use of faulty parts", ICCD ‘11.
[11] J.Tan, X.Fu, "RISE: improving the streaming processors reliability against soft errors in GPGPUs", PACT ‘12.
[12] H. Jeon, M. Annavaram, “Warped-DMR:Light-weight error detection for GPGPU”, MICRO ‘12.
[13] J. Tan, Z. Li, X. Fu, "Cost-effective soft-error protection for SRAM-based structures in GPGPUs", CF ‘13,
[14] R. Shah, M. Choi , B. Jang, "Workload-dependent relative fault sensitivity and error contribution factor of GPU onchip memory structures”, SAMOS ‘13.
[15] S. K. S. Hari, T. Tsai, M. Stephenson, S. W. Keckler, and J. Emer, "SASSIFI: Evaluating resilience of GPU applications", SELSE ‘15
[16] B. Fang, K. Pattabiraman, M. Ripeanu, S. Gurumurthi,“GPU-Qin: A methodology for evaluating the error resilience of GPGPU applications”, ISPASS ‘14.
[17] Ali Bakhoda, George Yuan, Wilson W. L. Fung, Henry Wong, Tor M. Aamodt, "Analyzing CUDA workloads using a detailed GPU simulator", ISPASS ‘09.
[18] R. Ubal, B. Jang, P. Mistry, D. Schaa, D. Kaeli,"Multi2Sim: A Simulation Framework for CPU-GPU Computing", PACT '12
[19] N. George, C. Elks, B. Johnson, J. Lach, “Transient fault models and AVF estimation revisited”, DSN ‘10.
[20] N. J. Wang, A. Mahesri, S. J. Patel, “Examining ACE analysis reliability estimates using fault injection”, ISCA ‘07.
[21] “NVIDIA CUDA SDK 4.2” [Online]. Available: https://developer.nvidia.com/cuda-toolkit-42-archive
[22] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, “Rodinia: A benchmark suite for heterogeneous computing,” IISWC ‘09,.
[23] R.Leveugle, A.Calvez, P.Maistri, P.Vanhauwaert, “Statistical fault injection: Quantified error and confidence”, DATE ‘09.
[24] J. Leng , T. Hetherington , A. ElTantawy , S. Gilani , N. S. Kim , T. Aamodt , Vijay Janapa Reddi, “GPUWattch: enabling energy optimizations in GPGPUs” , ISCA ‘13
[25] P. N. Glaskowsky, "NVIDIA’s Fermi: The First Complete GPU Computing Architecture",
[26] V. Sridharan, D. Kaeli, “Using Hardware Vulnerability Factors to enhanche AVF analysis ”, ISCA ‘10