Using Hardware Performance Counters to support in-field GPU Testing

Juan-David Guerrero-Balaguera, Josie E. Rodriguez Condia, Matteo Sonza Reorda
Department of Control and Computer Engineering, Politecnico di Torino, Torino, Italy

Abstract **—Graphics Processing Units (GPUs) have gained importance in several domains where a high computational effort is required (e.g., where Artificial Intelligence is used). At the same time, their adoption extended to domains (e.g., automotive, robotics, aerospace) where the effects of possible hardware faults can be extremely serious. Hence, it is crucial to identify methods allowing to quickly detect the occurrence of faults in a GPU while it works in the operational phase. In this paper we propose a method based on the adoption of hardware performance counters. We show that the method is able to detect permanent faults occurring in some critical modules, thus allowing to increase the overall reliability of the system. The method does not involve significant costs, since performance counters already exist in all real GPUs, e.g., to support silicon debug, and can be easily accessed in software.**

Keywords—Hardware Performance Counters, Graphics Processing Units, In-field test, Reliability, Safety.

I. Introduction

The GPUs have grown in importance in recent years in highly intensive computation applications such as multimedia, multi-signal analysis, high-performance computing, and Artificial Intelligence. Moreover, the use of these devices has been adopted and extended to other domains with important requirements in terms of reliability and functional safety (i.e., automotive, robotics, and aerospace).

The current cutting-edge technologies using in the manufacturing of GPUs allow improving their performance and reducing their power consumption. However, these semiconductor technologies are susceptible to permanent faults arising during the device's operational life, e.g., by wear-out or aging [1]. These faults may change the functionality of the device and seriously affect the reliability of the system. Accordingly, testing the device periodically for fault detection before producing a catastrophic effect is mandatory.

The in-field test of a GPU involves using specialized hardware structures or mechanisms to guarantee the requirements of safety-critical applications in terms of reliability and operational constraints. Design for Testability (DfT) mechanisms, such as Memory and Logic BIST, can be activated during the power on or in the idle times when the timing constraints are less restricted. Other solutions use software routines to test the functionality of each target module in the GPU [2] [3]. All the above solutions require stopping the application or using their idle time to launch the test.

In this paper, we present a solution for the in-field test of GPUs using the hardware performance counters (PfCs). Although these counters are resources available in the GPUs devoted mainly to silicon debugging and design validation purposes, they can be employed for testing purposes of some critical components inside GPUs. The method allows the quick and inexpensive detection of faults in some critical modules within a GPU (such as the warp scheduler and the divergence management unit) by monitoring the values produced by these counters. Hence, using the proposed solution some faults can be detected resorting to a true concurrent on-line testing, without stopping the application.

The proposed usage of PfCs involves two stages. In a preliminary stage, the target application is run on the device, and the values of some PfCs is gathered to build an invariant fault-free profile. The choice of the PfCs to build the profile depends on the evaluated hardware module of the GPU we consider. Once the expected values of the PfCs for the target application have been calculated, the second stage consists in monitoring whether, during the system operational phase, the application forces the selected PfCs to produce values coherent with the expected ones. In the negative case, the system may force the execution of a detailed test procedure.

In this paper we considered an open-source model of a GPU (named *FlexGripPlus*) mimicking an NVIDIA GPU, where some PfCs are available. Our experiments demonstrate that monitoring the values of PfCs inexpensively allows to detect some critical faults in a few control modules.

II. Background

A. *GPU organization*[1]

Graphic Processing Units (GPUs) are devices composed of scalable arrays of parallel execution units known as *Streaming Multiprocessors* (SMs). Each SM in a GPU uses the *Single-Instruction Multiple-Tread* (SIMT) execution model, where a single instruction executes multiple independent threads concurrently. The execution of one instruction in the SM uses five stages of pipeline (fetch, decode, read, execution/control flow, and writeback).

A scheduler unit controls the execution of one SIMT instruction, fetching, decoding, and distributing the workloads to be executed in the parallel processing cores called *Streaming Processors* (SPs) of the SM. One SM can contain between 8 and 128 SPs depending on the GPU model and the number of parallel threads to be processed concurrently. Additionally, the read and write pipeline stages contain the special controllers

[1] This subsection summarizes the architecture and behavior of NVIDIA GPUs, although the method proposed in the paper easily extends to GPUs of other vendors.

and necessary logic to load and store data into the storage units (General Purpose Registers File (GPRF), Shared Memory (SMem), Global Memory (GMem), and Constant Memory (CMem)). GPRF, SMem, and CMem reside inside the SM, and GMem is external to the device.

B. *Hardware performance counters*

The hardware PfCs are mainly used to evaluate the performance, the occupied resources, the debugging, or the application's optimization [4]–[6]. In addition, several works have explored the PfCs usage for other purposes like power consumption monitoring [7], [8], or detecting malicious attacks [9]–[11]. Moreover, PfCs can be employed to detect anomalies or errors caused by faults (transient or permanent) during the execution of any safety-critical application.

Until now, the usage of PfCs for reliability purposes has been evaluated mainly for CPU-based systems, in which the PfCs are employed to characterize microprocessor-based systems for reliability monitoring [12]. Additionally, The PfCs can be exploited to create statistical modeling for online testing and fault tolerance [13] and for Cyber-Physical Systems reliability and security attacks mitigation [14]. Other works use PfCs for the test programs development [15] or to detect control flow errors caused by faults [16].

In GPUs, several PfCs are available as in most advanced microprocessors. Modern GPUs offer between 100 to 200 PfCs [4] grouped into five general categories: memory, instruction, multiprocessor, cache, and texture [10]. We can find their use in different areas, such as power consumption measurement [7], [17], kernel characterization for optimization [5], [18], and security attacks detection and mitigation [10].

Accessing the GPU PfCs requires incorporating software libraries (APIs and Performance Tools) interacting with the low-level software interfaces to access and collect the PfCs' information. AMD has the GPUPerfAPI performance API and NVIDIA offers several performance tools supporting different features of its devices, such as CUPTI and NVIDIA PerfKit.

The NVIDIA profiling APIs and tools access PfCs using a low-level software interface of the GPU. That interface comprises several assembly instructions (SASS) especially developed to configure a performance counter and capture the counting value into a general-purpose register. The counters are available to the GPU itself, or to the Host where the profiler tool analyzes the results. In NVIDIA GPUs [19], the SASS *pmevent* instruction triggers one or more performance monitor events. The PfCs are visible as special read-only registers and accessed using the *mov* or *cvt* instructions.

III. Experimental setup and results

For the purpose of this work, the FlexGripPlus GPU model was employed to evaluate the effectiveness of PfCs in the detection of some critical faults in two GPU modules during the execution of a selected application. The selected modules are i) the WU and ii) the control flow unit. These units are critical for the normal execution flow of any application in the GPU, since faults affecting them are very likely to seriously impact the behavior of the whole GPU. Therefore, the detection of any fault in these modules before they cause a fatal failure is mandatory.

A. *FlexGrip architecture*

FlexGipPlus is an open-source soft-core GPU compliant with the NVIDIA G80 architecture [20]. FlexGripPlus is compatible with the CUDA programming flow and implements 52 assembly instructions. The internal organization of the FlexGripPlus GPU is composed of a scalable array of Streaming Multiprocessors (SMs) and one block scheduler in charge of dispatching the tasks to each SM. Inside each SM, the warp scheduler assigns the warps to the Scalar Processors (SPs).

1) *The Warp Unit*

According to the NVIDIA specifications, the threads assigned to an SM are launched in small groups called *warps*. A warp is composed of 32 or 64 threads that execute the same instruction in parallel using independent data (SIMT). In FlexGripPlus, the Warp Unit (WU) is in charge of managing the execution of threads in each SM.

The WU comprises six functional units that interact between them. These functional units are the warp generator, warp scheduler, warp checker, fence registers, and two memories used to store the information about the execution and status of each thread per warp. The *warp pool memory* stores the warp ID, the program counter, and the tread mask per warp. The *warp state memory* stores the current status per warp, which can either be (*Ready, active, waiting,* or *finished*).

The warp generator takes the configuration parameters from the block scheduler of the GPU, identifies the total number of warp lines in the pool memory, and initializes every warp line.

After the warp initialization finishes, the warp scheduler launch warps in a round-robin fashion, reading the pool and state memories. The PC, tread mask, and state are updated in the memories every time the warp completes the pipeline execution loop. This regular flow of warp scheduling is sometimes interrupted when a barrier synchronization instruction appears. The WU executes this instruction and marks all warps as waiting. For each *waiting* warp, the WU sets one bit in the fence register. Reading the fence register and checking that all bits are set indicates that all arrived warps are in a *waiting* state, so all of them are synchronized, and the barrier is released. In this case, the WU changes the state of all warps to *ready*, and the warp scheduling process continues as usual.

The warp checker is in charge of checking each warp's status, performing a comparison between the current state of each warp coming from the SM and the previous values generated in the warp pool memory. According to the comparison results, the warp checker updates the warp pool and the status memory. Both the warp scheduler and the warp checker correspond to Finite State Machines (FSMs) and control the execution of every warp in the SM.

2) *The Control Flow Unit*

The *Divergence Management Unit* (DMU) in the control/execute pipeline stage of the FlexGripPlus GPU controls and tracks the Intra-Warp Divergences (IWDs), which appear in programs when the threads in the same warp execute different instructions, generating multiple execution paths. IWD occurs during the execution of all control flow instructions such as conditional and unconditional branches, barrier synchronization, kernel return, and set synchronization point. Additionally, the DMU interacts directly with the *warp stack* memory, which stores the starting (*divergency*) and ending points (*convergency*) of different execution paths for the same warp.

The operation of the DMU for a warp executing branch instruction or conditional branch instruction follows four main steps. First, the synchronization point is reached, and the DMU pushes in the stack the synchronization point information for the current warp. In the second stage, the execution reaches the branch instruction. In this step, the threads in the “taken” path are executed first. At the same time, the thread mask of the “not-taken” path and the current PC are pushed in the stack. In the third stage, the first path of threads reaches the reconvergence point, the stack is popped, and the threads in the “not taken” path are loaded in the pipeline to be executed. In the fourth stage, all threads in the warp reach the reconvergence point, the synchronization information is popped from the stack, and all threads continue the parallel execution.

The DMU in the FlexGripPlus GPU is an FSM that generates the control signals needed to manage the warp stack memory and the thread execution on each SP. Additionally, the DMU directly interacts with the WU, sending the thread masks and the warp state for the currently executed warp. The DMU generates and propagates the masks and state of the current warp through the pipeline stages up to when it arrives at the WU where the next warp will be scheduled.

B. *The FlexGrip GPU Performance Monitoring Unit*

The FlexGripPlus model has been enriched with a Performance Monitoring Unit (PMU). Each SM of the GPU includes a PMU in charge of counting events produced by several modules during any kernel execution. The PfCs implemented by the PMU correspond to counters existing in real NVIDIA GPUs. Additionally, the GPU includes the necessary SASS-compliant instructions allowing to access the PMU. The devised PMU is composed of a set of 2 PfCs. One counter calculates the number of launched inactive threads (INACT). This measurement takes the thread mask output port of the WU and increments the counter by a factor *k*, where *k* is the number of inactive threads per launched warp. The other counter increments on each warp stack accesses (STACK). This counter increments by one on each push and pop operation.

These counters allow detecting critical faults affecting the WU and the DMU. In the WU, the faults present in the warp pool memory can be detected by resorting to the INACT counter. In fact, this counter allows to directly monitor the thread mask port of the WU, then any fault in that field of the *warp pool memory* can be detected because there will be a variation in the number of active or inactive treads during the application execution. Similarly, any fault in the program counter field of the warp pool memory can be detected by the INACT counter or by the STACK counter when a branch instruction producing divergences is present in the application. This detection is possible because any fault in this field will always produce a wrong instruction execution for the launched warp skipping some instructions or by repeating others.

When a fault induces a malfunction in the FSM of the warp scheduler or the warp checker, it can be detected using any of the PfCs previously mentioned because of an erroneous warp execution either by a missing warp or a wrong warp sequence.

TABLE I. NORMALIZED BOUND LIMITS FOR CHARACTERISTIC OPERATIONAL REGIONS

PfC		**Applications**						
		FFT	**GRAY**	**MM**	**RED**	**SOBEL**	**SORT**	**TRAN**
INACT	**Max**	0.245	0.0	0.0	0.0	0.060	0.018	0.0
	Min	0.0	0.0	0.0	0.0	0.0	0.015	0.0
STACK	**Max**	0.917	0.0	0.500	0.500	0.0	1.014	1.0
	Min	0.0	0.0	0.0	0.0	0.0	1.018	0.0

Faults affecting the DMU can be detected when control flow instructions are executed. So, a fault in this unit affecting the correct generation of the outputs, *current thread mask*, and *warp state* will affect the number of active and inactive launched threads per warp, so the INACT counter is able to detect those faults. Similarly, any fault in the stack memory can be detected by resorting to the same events. In addition, any fault in the FSM of the DMU that affects the control signals used by conditional branches, in turn, changes the number of divergences generated and directly affects the stack accesses. Hence the number of push and pop operations (counted by the STACK counter) requested to the stack memory can be used to detect faults in these locations of the DMU.

C. *Performance counters precomputation and monitoring*

Once the PfCs of interest are identified, several experiments using representative applications in order to characterize each one in terms of the PfCs values under fault-free conditions are performed. The experiments were developed using the FlexGripPlus configured with 1 SM and 8 SPs per SM. The representative applications employed for the experiments were carefully selected to cover all parallel communications patterns presented in GPU programming. Those applications are Fast Fourier Transform (FFT), Matrix Multiplication (MM), Reduction (RED), Transpose of a Matrix (TRAN), Sobel Filter (SOBEL), RGB to Gray conversion (GRAY).

Each application was executed five times, employing different input data each time. For each experiment, the evolution of the selected PfCs was collected during the execution of each application. Each performance counter was sampled ten times during the execution of the kernel for each application. The sampling of the PfCs consists in reading each performance counter on a given sampling time t_x during the kernel execution and then calculating the difference between the current counter value PFC(t_x) and the previous one PFC(t_{x-1}). This sampling data allows identifying the *characteristic operational region* in which the values of the PfCs guarantee that the application is executing correctly. Otherwise, when any performance counter value lies outside this region, this is considered as a possible fault presence indication.

Table I presents the normalized bounds of the operational region calculated for each application using two representatives PfCs. One counter (INACT) is used to detect faults in the already mentioned locations in the WU and in the DMU, and the other counter (STACK) for detecting faults in the DMU. The INACT counter is normalized, corresponding to the number of inactive threads divided by the total number of launched threads. For the STACK counter, the normalization is calculated as the number of Push and Pop operations divided by the number of divergences. The normalization process is performed for each individual sampling period.

TABLE II. FAULT EFFECTS IN THE WARP SCHEDULER MEMORY

PfC		Applications						
		FFT	GRAY	MM	RED	SOBEL	SORT	TRAN
INACT	**Max**	+0.003	+0.030	+0.031	+0.031	+0.031	+0.029	+0.031
	Min	0.0	0.0	+0.028	0.0	0.0	0.0	0.0
STACK	**Max**	+0.042	0.0	0.0	0.0	0.0	0.0	0.0
	Min	0.0	0.0	0.0	0.0	0.0	-0.036	0.0

TABLE III. FAULT EFFECTS IN THE DMU

PfC		Applications						
		FFT	GRAY	MM	RED	SOBEL	SORT	TRAN
INACT	**Max**	-0.088	0.0	0.0	0.0	0.0	0.0	0.0
	Min	0.0	0.0	0.0	0.0	0.0	0.0	0.0
STACK	**Max**	+0.458	0.0	+0.984	+0.500	0.0	+0.700	+0.875
	Min	0.0	0.0	0.0	0.0	0.0	-0.429	0.0

As shown in Table I, the characteristic operational region for the INACT counter is particular for each application. For example, in the FFT, SOBEL and SORT applications the inactive threads reach a maximum ratio 0.245, 0.060 and 0.018, respectively. The minimum ratio of 0.0 means that during the execution of the application, the performance counter does not change between two consecutive sampling times. For the case of GRAY, MM, RED, and TRAN, the maximum and minimum values are 0 because these applications do not have inactive threads during normal execution.

In the case of the STACK, the GRAY and SOBEL applications do not produce any divergence and push/pop operations. Therefore, those applications are excluded from the analysis using this counter, because they do not use the DMU.

The values of the selected PfCs are affected by the presence of a stuck-at fault in the selected locations of the WU and DMU. Table II shows the changes in the normalized selected PfCs when a fault is present (highlighted in gray) in the thread mask of the *warp pool memory*. According to these results, the INACT counter overcomes the maximum value allowable in all applications, more precisely by 0.003 for the FFT application up to 0.031 for MM, RED, SOBEL, and TRAN. On the other hand, the STACK counter can detect this fault as well, but only for FFT and SORT applications.

Table III presents the affected performance counter ratio results in the faulty scenario where one fault is present in the selected locations of the DMU to control the stack memory (highlighted in gray). According to the results, this kind of faults affects the maximum number of push/pop operations, increasing the STACK counter value during the execution of any application. In all applications, the STACK counter overflows the maximum region value by 0.5 for RED up to 0.984 in MM. In the case of SORT, the STACK value is lower than the minimum value by 0.429. We observe in Table III that the STACK counter detects this kind of faults independently of the considered application.

IV. Conclusions

This paper proposes the usage of PfCs as a support for in-field test of GPUs. These counters are available in GPUs mainly to support silicon debugging and for design validation. However, they can be used for testing purposes of some critical components inside the GPUs (in this paper we considered the WU and the DMU). The method allows the quick and inexpensive detection of critical faults in the target components by monitoring the values produced by these counters. Any deviation in the counter's values outside the operational region for the selected application corresponds to a fault detection and can be used to stop the application and launch an in-field test procedure. Therefore, this work demonstrates that the selected PfCs are effective for the detection of faults in critical units during the runtime of any application in a GPU.

As future work we plan to extend the use of PfCs to evaluate other units in the GPU, such as the SP or the memory controllers.

References

[1] S. Hamdioui, *et al.*, "Reliability challenges of real-time systems in forthcoming technology nodes," *Design, Automation and Test in Europe,* 2013, pp. 129–134.

[2] J. D. Guerrero-Balaguera, *et al.*, "On the Functional Test of Special Function Units in GPUs," *International Symposium on Design and Diagnostics of Electronic Circuits and Systems,* 2021, pp. 81–86.

[3] J. E. R. Condia, *et al.*, "On the testing of special memories in GPGPUs," *International Symposium on On-Line Testing and Robust System Design,* 2020.

[4] B. Zigon, *et al.*, "Utilizing gpu performance counters to characterize gpu kernels via machine learning," *International Conference on Computational Science,* 2020, pp. 88–101.

[5] S. C. Tsai, *et al.*, "Kernel Aware Warp Scheduler," *IEEE International Symposium on Circuits and Systems*, 2018.

[6] B. Sprunt, "The basics of performance-monitoring hardware," *IEEE Micro*, vol. 22, no. 4, pp. 64–71, 2002.

[7] R. A. Bridges, *et al.*, "Understanding GPU power: A survey of profiling, modeling, and simulation methods," *Computing Surveys*, vol. 49, no. 3. Association for Computing Machinery, 2016.

[8] Y. Kim, *et al.*, "System-level online power estimation using an on-chip bus performance monitoring unit," *IEEE Trans. Comput. Des. Integr. Circuits Syst.*, vol. 30, no. 11, pp. 1585–1598, 2011.

[9] S. Das, *et al.*, "SoK: The challenges, pitfalls, and perils of using hardware performance counters for security," *IEEE Symposium on Security and Privacy*, 2019, pp. 20–38.

[10] H. Naghibijouybari, *et al.*, "Rendered Insecure: GPU Side Channel Attacks are Practical," *Conference on Computer and Communications Security,* 2018, vol. 15.

[11] X. Wang *et al.*, "Malicious Firmware Detection with Hardware Performance Counters," *IEEE Trans. Multi-Scale Comput. Syst.*, vol. 2, no. 3, pp. 160–173, 2016.

[12] E. W. L. Leng, *et al.*, "Hardware performance counters for system reliability monitoring," *International Verification and Security Workshop*, 2017, pp. 76–81.

[13] S. Esposito, *et al.*, "A novel method for online detection of faults affecting execution-time in multicore-based systems," *ACM Trans. Embed. Comput. Syst.*, vol. 16, no. 4, 2017.

[14] A. Carelli, *et al.*, "Performance monitor counters: Interplay between safety and security in complex cyber-physical systems," *IEEE Trans. Device Mater. Reliab.*, vol. 19, no. 1, pp. 73–82, 2019.

[15] W. Lindsay, *et al.*, "Automatic test programs generation driven by internal performance counters," *International Workshop on Microprocessor Test and Verification*, 2005, pp. 8–13.

[16] H. A. H. Ahmad, *et al.*, "A performance counter-based control flow checking technique for multi-core processors,"*International Conference on Computer and Knowledge Engineering,* 2017, pp. 461–466.

[17] J. Guerreiro, *et al.*, "GPGPU Power Modeling for Multi-domain Voltage-Frequency Scaling," *International Symposium on High-Performance Computer Architecture*, 2018, pp. 789–800.

[18] J. Filipovič, *et al.*, "Using hardware performance counters to speed up autotuning convergence on GPUs," arXiv:2102.05297 [cs.DC], 2021.

[19] NVIDIA Corporation, "PTX ISA :: CUDA Toolkit Documentation," 2021. [Online]. Available: https://docs.nvidia.com/cuda/parallel-thread-execution/index.html. [Accessed: 30-Jun-2021].

[20] J. E. R. Condia et al., "Flexgripplus: An improved gpgpu model to support reliability analysis," Microelectronics Reliability, vol. 109, 2020.