Implementation and Experimental Evaluation of a CUDA Core under Single Event Effects

Werner Nedel, Fernanda Kastensmidt
Instituto de Informática - PPGC - PGMICRO
Universidade Federal do Rio Grande do Sul (UFRGS)
Av. Bento Gonçalves 9500, Porto Alegre - RS - Brazil
{wmnedel, fglima} @ inf.ufrgs.br

José Rodrigo Azambuja
Centro de Ciências Computacionais - PPGCOMP
Universidade Federal do Rio Grande (FURG)
Av. Itália Km 8, Rio Grande - RS - Brazil
jrazambuja@furg.br

***Abstract*—Graphic Processing Units have become popular in a broad range of applications due to their high computational power and low prices. Among the applications are the safety critical ones, where fault tolerance is mandatory. This paper presents the implementation of a CUDA core, the main processing core of a GPU and its evaluation under Single Event Transients. Results will be able to help designers to develop the required fault tolerant techniques in an effective fashion.**

Keywords—graphic processing unit, fault tolerance, single event transients

I. INTRODUCTION

Graphic Processing Units (GPUs) [1], [2] are specialized massively parallel many-core processors that take advantage of Thread-Level Parallelism (TLP) to handle computations for computer graphics. GPUs dedicate the majority of the silicon area to data processing with only a small portion assigned to data caching and flow control circuitry, which makes them suitable for solving computer-intensive problems, in spite of general purpose CPUs, which embrace sequential data flow structure with the use of Instruction-Level Parallelism (ILP).

The ability to rapidly manipulate high amounts of memory locations and execute several elementary tasks in parallel at high speeds makes GPUs more effective than CPUs for algorithms in which large blocks of data can be processed in parallel. Examples of algorithms where GPUs excel are oil exploration, analysis of air traffic flow, medical image processing, linear algebra, statistics, 3D reconstruction, and stock options pricing determination [3].

The rapid proliferation of GPUs due to the advent of significant programming support has brought programmers to use them in safety critical applications, like automotive, space and medical [4-6]. In these applications, the use of fault tolerant techniques is mandatory to detect or correct faults, since they must continue to work properly despite the existence of faults. However, the reliability of a GPU system is still an open issue.

The increasing demand for computational has pushed GPUs to be built in cutting-edge technology down to 28nm fabrication process for the latest NVIDIA devices with operating clock frequencies up to 1GHz. The increases in operating frequencies and transistor density combined with the reduction of voltage supplies have made transistors more susceptible to faults caused by radiation interference due to reduced threshold voltages, reduced node capacitances and tightened noise margins [7]. Such faults, mainly caused by energized particles, make the newest GPUs potentially prone to experience radiation-induced errors [8], [9], even on terrestrial applications running at ground level, where neutrons are the main sources of soft errors [10].

Single Event Effect (SEE) is known as one of the major effects that may occur when a single radiation ionizing particles strikes the silicon. This effect can be destructive and non-destructive. An example of destructive effect is Single Event Latchup (SEL) that results in a high operating current, above device specifications, that must be corrected by a power reset. Non-destructive effects are defined as a transient effect fault provoked by the interaction of a single energized particles in drain PN junction of the off-state transistors. This strike may upset a node of the circuit and thus generate a transient voltage pulse that can be interpreted as internal signals and lead to an erroneous result [11]. When the transient pulse occurs in a memory element, such as a register, it is classified as Single Event Upset (SEU). When the particle hits a combinational element, inducing a pulse in the combinational logic, the upset is classified as Single Event Transient (SET).

Compute Unified Device Architecture (CUDA) is a model created by NVIDIA and implemented by its GPUs that gives program developers direct access to the virtual instruction set and memory. The CUDA core, on the other hand, is the hardware module responsible for executing threads in the GPU architecture. A GPU is mainly built on top of a large number of CUDA cores and therefore the CUDA core can be considered as the building block of a GPU.

In this paper, the authors will implement a CUDA core prototype described in Hardware Description Language (HDL) and will perform an experimental fault injection campaign to simulate the effects of SEUs and SETs on it. Results will show the most sensitive parts of the CUDA core that must be protected by fault tolerance techniques in order to be used in safety critical applications, even at ground level.

II. Related Works

The literature has a few works that have implemented a GPU in HDL languages for FPGAs. The FlexGrip project [14] examines the implementation of a soft G80 GPU in a Virtex-6 FPGA. This architecture is able to run CUDA compiled objects without hardware recompilation. The results presented in the work indicated a considerable gain, up to 30x, when compared to MicroBlaze processor. However, the G80 architecture was the first GPU developed with general computation purpose, still in 2006, and several improvements are taken place in new GPUs architectures [1], [2], such as Fermi, which is the architecture used as reference in the present work.

Some projects have evaluated the neutron sensitivity of GPUs. In [15] the authors analyzed the results of neutrons radiation campaigns on a GPU, presenting a complete guide for accelerated radiation experiments, while in [16] the experiments are focused on the sensitivity of integer and floating point operations executed in GPUs. It shows that there is a strong dependence of the neutron-induced error rate of different codes on data type, and operations are more reliable when executed on floating point data with respect to integer.

Radiation injection campaigns are very important to evaluate the sensitive of GPU internal modules and instructions, but to the best of our knowledge, so far no fault injection campaign has been performed in the HW implementation. This work presents the implementation of a CUDA core and a simulation fault injection campaign in the integer and floating point units.

III. Graphic Processing Units Internal Structure

A GPU is a many-core device with a substantial parallel processing capability. The GPU consists of an array of multiprocessors enabling the device to execute lots of threads in parallel. The majority of the silicon area is dedicated to data processing units with only a small portion assigned to data caching and flow control circuitry. Its data processing units can be seen in Fig. 1.

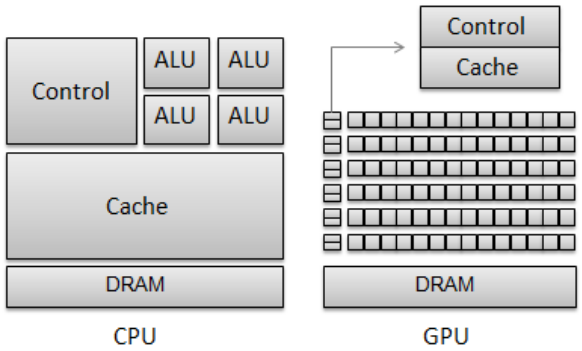

Figure 1: Data processing units in a GPU

Such architecture enables the GPU to solve streaming compute-intensive problems [1] more efficiently than when compared to General Purpose Processors (GPPs). On the other hand, GPUs are designed to optimize the execution time of a given task, and to do so they need robust control units to schedule thread execution sequentially. Large cache memories are also needed to minimize the latency of instruction and data accesses. Conversely, GPUs have several computing units which instantiates and executes a large amount of instructions at the same time, so a very simple control unit is sufficient to schedule the work between them.

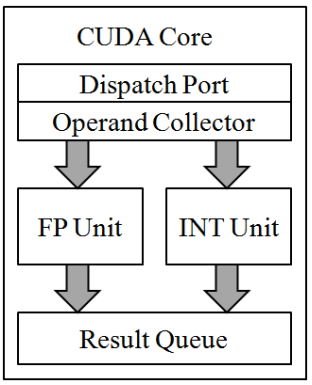

Figure 2: CUDA Core Structure

CUDA is the name of the programming model created by NVIDIA that enables the GPU to be programmed with a variety of high level programming languages. This programming model is the key to use GPUs as General Purpose Graphics Processing Units (GPGPUs). The host program launches several kernels organized as a grid of thread blocks. The thread blocks are partitioned and grouped into warps, which is the smallest set of simultaneous operations.

GPUs are divided into various computing units, called Streaming Multiprocessors (SM), which are responsible for executing several threads in parallel, in a Single Instruction Multiple Data (SIMD) fashion. Every thread executes with dedicated memory locations, thus avoiding complex resource sharing or long pipelines. A very simple control unit is then sufficient to schedule threads execution [3]. The structure of a GPU is different from the typically CPU one, where sophisticated control units are needed to schedule complex thread execution sequentially or in parallel with other executions, eventually taking advantage of the presence of multiple arithmetic logic units. Moreover, on a CPU, large cache memories must be provided to minimize the instruction and data access latencies of large complex applications.

The SM is composed of various computing cores named CUDA cores, load/store units and Special Function Units (SFU), which can be seen in Fig. 3 as Core, LD/ST and SFU, respectively. Each CUDA core has a pipeline integer Arithmetic Logic Unit (ALU) and a Floating Point Unit (FPU), which can be seen in Fig. 2 as INT Unit and FP Unit, respectively. The load/store units supply memory access to all the threads running in the SM. The SFUs execute special instructions such as sin, cosine, reciprocal, and square root. It is decoupled from the dispatch unit, and therefore does not interfere with the other execution units. These processing cores are then connected by an interconnect network and fed with data from an instruction cache and a register bank.

As one can see in Fig. 2, the CUDA core is the main processing node of a GPU. It is composed not only by the

Integer and FP units, but also by a dispatch port, operand collector, and result queue. The dispatch port is responsible for distributing the control signals from the SM into both processing units or, in other words, designating each instruction to its processing unit. The operand collector is a hardware module responsible for interface with the register bank and thus providing the required data to be processed. The result queue performs a sort of temporal register cache for the functional units. The INT unit executes all the integer operations, while the FP unit deals with the floating point instructions, using the IEEE 754-2008 floating-point standard [1]. Finally, the result queue avoids conflicts of writes in the register file. Also, some operations do not need to write back into the register file, producing only an intermediate result which is consumed by another operation. So, in that situation, if the two operations are scheduled close enough, the output value can be forwarded to the input of next operation, avoiding an extra access into the register file.

In a GPU, each thread may dispose of up to tens of megabits of internal registers and has access to shared memory, which is necessary to avoid multiple accesses to the DRAM performed by threads executing operation on the same data. Usually, on a GPU, threads do not interact with each other to minimize latency and waiting delays and to take full advantage of the GPU parallelism. Thus, a small stand-alone computing unit executes each thread, and the GPU scheduler is needed just to synchronize all threads and to check if execution has been completed. The GPU physical design may then be viewed as a group of several isolated elementary computing units (each one executing a thread) whose corruption will generate an output error in the threads that is being executed but will not affect other units.

IV. CUDA Core Implementation

Field Programmable Gate Arrays (FPGAs) are highly specialized devices that offer application-specific customization to designers, which includes Block RAM (BRAM) memory, Digital Signal Processors (DSPs), and Combinational Logic Blocks (CLBs) configured through Look-Up Tables (LUTs). Such devices offer low time-to-market and have been used in safety-critical applications in the past with success [12]. Due to these characteristics, we have chosen the XC5VLX110T part Virtex5 FPGA to implement the CUDA core, which contains a total of 69,120 slice LUTs and 5,905 slice registers.

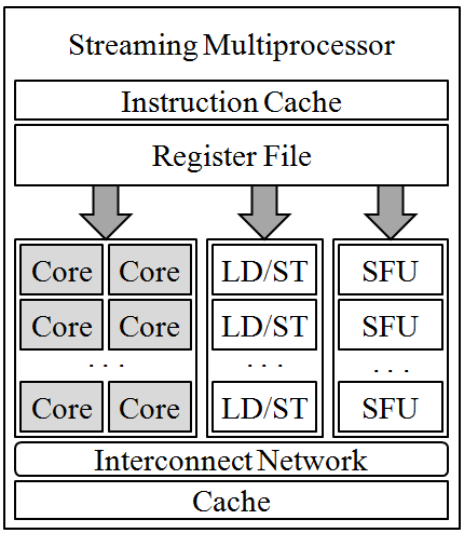

Figure 3: Streaming Multiprocessor Structure

In order to implement the CUDA core, we started by selecting the instruction set to be supported. Taking [13] into consideration, we chose for the INT unit the following instructions: IADD, IADD32I, IMUL, IMUL32I, IMAD, ISCADD, ISETP, and ICMP. For the FP unit, we have chosen the following instructions: FADD, FADD32I, FMUL, FMUL32I, FCMP, MUFU, DADD, DMUL, and DFMA. By implementing this set of instructions, we are able to support a large set of applications to be executed, by means of providing addition and multiplication with different types of address mode and floating point precision. The INT unit was implemented in a 5-stage pipeline, while the FP unit was implemented in a 3-stage pipeline.

Table I shows the area of the complete CUDA core. The INT Unit requires 2,098 slice LUTs and 1,812 slice registers and, resulting in 2.6% and 3.0% in the FPGA's fabric occupation, respectively. Additionally, the FP Unit requires 1,258 slice LUTs and 1,306 slice registers, resulting in 1.8% and 1.9% in the FPGA's fabric occupation, respectively. The whole system has been implemented using only fabric logic, meaning that no hardware accelerator, such as DSPs, has been used. The complete CUDA core, including dispatch port, operand collector and a two-word result queue requires 3,656 slice LUTs and 3,417 slice registers, which represent 5.3% and 4.9% occupation of the FPGA fabric. Such results mean that the currently used FPGA could support up to 9 CUDA cores.

TABLE I. Area and Frequency Results for the Virtex5 FPGA in Slice LUTs and Registers

	INT Unit (occupation)	**FP Unit (occupation)**	**CUDA Core (occupation)**
Slice LUTs	2,098 (2.6%)	1,258 (1.8%)	3,656 (5.3%)
Slice Registers	1,812 (3.0%)	1,306 (1.9%)	3,417 (4.9%)
Fequency (MHz)	100	300	100

V. Fault Injection Experimental Results

In order to evaluate both the effectiveness and the feasibility of the CUDA core, we implemented a benchmark containing all the instruction implemented and supported by the CUDA core with all possible operating modes (IADD with a normal and negated operand, IMULT with signed and unsigned, etc). Instruction's internal signals have been translated to our implementation and used as inputs. It is important to notice the program memory feeds the SM with instruction, which then provides the CUDA core operand and control signals.

In order to perform the fault injection campaign, faults were injected in the signals of the implemented CUDA core (including registered signals), one fault at each program execution. The SEU and SET types of faults were injected directly in the CUDA core VHDL code by using the Xilinx's

iSim simulator. SEUs were injected in registered signals and SETs will be injected in combinational signals. Fault durations will last in the signal for one and a half clock cycle, so that we can avoid latch window masking. The fault injection campaign was performed automatically. At the end of each execution, the results were compared to the expected correct values.

The fault injection campaign comprehended 40,000 faults and analyzed the results by dividing faults between the faults that caused an error in the CUDA core (*Incorrect Result*) and the ones masked by the logic. The system was simulated with a clock frequency of 100MHz (period of 10ns) and most of the 5,841 signals describing were be upset.

Results from the fault injection campaign are presented in Table II. As one can see, 18.4% of the faults injected into the integer unit (INT Unit) have resulted in a wrong result, while 21.2% of the faults injected into the floating point unit (FP Unit) have resulted in a wrong result. When combined, 19.8% of fault injected in the CUDA core have corrupted the output results. Such results prove that GPUs must be hardened by means of fault tolerance techniques in order to be used in harsh environments.

TABLE II. INTEGER AND FLOATING POINT UNITS UNDER SET AND SEU FAULTS

	Faults Injected	**Correct Results**	**Incorrect Results**
INT Unit	20,000	16,313 (81.6%)	3,687 (18.4%)
FP Unit	20,000	15,746 (78.8%)	4,254 (21.2%)
Total	40,000	32,059 (80.2%)	7,491 (19.8%)

VI. CONCLUSIONS AND FUTURE WORK

In this paper, the authors presented the implementation of a CUDA core in a SRAM-based FPGA with an occupation of 3.2% slice LUTs and 10.1% slice registers of a Virtex5 FPGA. Such results allow future works to implement up to 9 CUDA cores in a single FPGA. At the cost of 5 CUDA cores, one could still implement a GPU with 2 SMs, each with 2 CUDA cores, being able to test case-study applications parallelized and adapted to GPUs.

A fault injection campaign has been proposed to test the proposed implementation under SETs and SEUs with a case-study benchmark. The results will be able to guide designers into developing fault tolerant techniques for GPUs with increased efficiency and lower costs in time and performance degradation.

In the future, the authors intend to implement the entire SM structure and perform several fault injection campaigns, including a fault injection by simulation at RTL level, by injecting faults in the FPGA's configuration bitstream, and by irradiating the FPGA with neutron and Cobalt-60. By doing so, we will be able to run CUDA compiled code and verify how the GPU responds to the effects of radiation.

REFERENCES

[1] NVIDIA Next Generation CUDA Compute Architecture: Fermi [online]. Available: http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_architecture_Whitepaper.pdf

[2] AMD Graphics Cores Next Architecture [online]. Available: http://www.amd.com/la/Documents/GCN_Architecture_whitepaper.pdf

[3] J. Kruger and R. Westermann, "Linear algebra operators for GPU implementation of numerical algorithms," ACM Trans. Graph, vol. 3, no. 22, pp. 908–916, 39–55, 2003.

[4] Q. Hu, B. Xiao, and M. Friswell, "Robust fault-tolerant control for spacecraft attitude stabilisation subject to input saturation," Control Theory Applications, vol. 5, no. 2, pp. 271–282, 2011.

[5] N. Zhang, "Investigation of Fault-Tolerant Adaptive Filtering for Noisy ECG Signals," in IEEE Symposium on Computational Intelligence in Image and Signal Processing, pp. 177–182, 2007.

[6] A. Strano, D. Bertozzi, A. Grasset, S. Yehia, "Exploiting structural redundancy of SIMD accelerators for their built-in self-testing/diagnosis and reconfiguration," in IEEE Int. Conf. on App.-Specific Systems, Architectures and Processors, 2011.

[7] R. C. Baumann, "Soft errors in advanced semiconductor evices-part I: the three radiation sources," in IEEE Transactions on Device and Materials Reliability. March 2001.

[8] C. Slayman and O. A. La Carte, "Soft errors-past history and recent discoveries," in Proc. IEEE Int. Integr. Reliab. Workhop, 2010, pp. 25–30, Invited Paper.

[9] A. Dixit and A. Wood, "The impact of new technology on soft error rates," in IEEE Int. Reliab. Phys. Symp. 2011.

[10] P. Rech, C. Aguiar, C. Frost, L. Carro, "An efficient and experimentally Tuned Software-Based Hardening Strategy for matrix multiplication on GPUs," IEEE Transactions on Nuclear Science, Vol. 60, pp. 2797-2804, 2013.

[11] P. Dodd, L. W. Massengill, "Basic mechanism and modeling of single-event upset in digital microelectronics," IEEE Transactions on Nuclear Science, Vol. 50, pp. 583-602, 2003.

[12] H. Quinn, P. Graham, K. Morgan, Z. Baker, M. Caffrey, D. Smith, R. Bell, "On-orbit results for the xilinx virtex-4 FPGA," IEEE Radiation Effects Data Workshop, pp. 1-8, 2012.

[13] Asfermi: an assembler for the NVIDIA Fermi instruction set [online]. Available: https://code.google.com/p/asfermi/

[14] K. Andryc, M. Merchant, "FlexGrip: A soft GPGPU for FPGAs", IEEE Field-Programmable Technology (FPT), pp. 230-237, 2013.

[15] P. Rech, C. Aguiar, R. Ferreira, C. Frost, L. Carro, "Neutron radiation test of graphic processing units", IEEE On-Line Testing Symposium (IOLTS), pp. 55-60, 2012.

[16] P. Rech, C. Aguiar, C. Frost, L. Carro, "Neutron sensitivity of integer and floating point operations executed in GPUs", IEEE Test Workshop (LATW), pp. 1-6, 2013.