GPUburn: A System to Test and Mitigate GPU Hardware Failures

Eric Petit, David Defour

GPUburn: A System to Test and Mitigate GPU Hardware Failures

David Defour
Université de Perpignan Via Domitia,
Laboratoire DALI - 54 avenue Paul Alduy
64000 Perpignan- France
david.defour@univ-perp.fr

Eric Petit
Université de Versailles Saint-Quentin,
Laboratoire PRISM - 45 avenue des Etats-Unis
78035 Versailles - France
eric.petit@uvsq.fr

***Abstract*—Due to many factors such as, high transistor density, high frequency, and low voltage, today's processors are more than ever subject to hardware failures. These errors have various impacts depending on the location of the error and the type of processor. Because of the hierarchical structure of the compute units and work scheduling, the hardware failure on GPUs affect only part of the application. In this paper we present a new methodology to characterize the hardware failures of Nvidia GPUs based on a software micro-benchmarking platform implemented in OpenCL. We also present which hardware part of TESLA architecture is more sensitive to intermittent errors, which usually appears when the processor is aging. We obtained these results by accelerating the aging process by running the processors at high temperature. We show that on GPUs, intermittent errors impact is limited to a localized architecture tile. Finally, we propose a methodology to detect, record location of defective units in order to avoid them to ensure the program correctness on such architectures, improving the GPU fault-tolerance capability and lifespan.**

I. Introduction

Hardware failure in computing processors is a known fact [1], [2], [3]. Massively parallel hardware co-processors such as GPU or Xeon Phi (MIC) suffer from the same problem [4]. Different factors such as low voltage, high frequency, thin engraving technology, numerous and dense transistors can significantly increase the hardware failure rate [5], [6], [7]. Using simpler cores with lower frequency and smaller standard-cell architectures, many-cores such as GPUs have a higher transistor density than current CPUs. For instance, with a die size of 550 mm2 and 7.1 billion transistors, the Kepler GPU has 12.1 M transistor / mm2 while the SandyBridge E CPU has a die size of 435 mm2 and 2.27 Billion transistors that is equal to only 5 M transistor / mm2. This corresponds to a ratio of nearly 2.5 between CPU and GPU. Hardware failures can be classified in three main categories [6]:

- **Permanent errors** are caused by the manufacturing process or a permanent damage due to aging.
- **Transient errors** are discrete and non-reproducible. They result from external factors such as radiation or interference.
- **Intermittent errors** are temporary but reproducible within the same context. They result from aging or a local variation of the quality of the manufacturing process.

As transient errors depend on external factors, they are not considered in this work. We focus on permanent and intermittent errors. Even if these problems are common to all processors, our intuition is that the impact on the GPU co-processor will be different due to a large number of independent cores. When an error occurs it can be accurately localized and the contamination can be limited to the faulty independent core unit. When using a GPU for computer graphics, errors usually have minor visual impact which limits the need to handle them. When it comes to scientific computing, especially High Performance Computing, one must ensure the results are reliable and stable in time [8], [9].

Some research works have already investigated the hardware error arising in the memory of GPU systems [8]. To limit the impact of such errors on memory hierarchy, Nvidia introduced Error-Correcting Code, ECC, up to the register files for Fermi and Kepler architecture. However this technology does not provide total resiliency and the error rate remains significantly high for large scale systems [10]. In order to improve the fault tolerance of the GPU based system, it is necessary to address all possible sources of error. To our knowledge, no previous study demonstrated and characterized the hardware failure impact on GPU computation.

To study the intermittent and permanent hardware failures on different GPUs, we design and implement a micro-benchmark platform in OpenCL [11], a portable high-level language. Our benchmarks platform helps the user to detect and localize the intermittent and permanent hardware errors on GPUs. Using the error location, it is possible to modify the hardware, the driver, or the user code to schedule useful work exclusively on trusted parts and avoid the defective ones. To validate the hypothesis of the error self-containement to a particular tile in the GPU hardware hierarchy, we design a hardware apparatus to accelerate aging of the GPU chip [2] while executing our benchmarks. We modify the cooling system of the target architecture and enclose the GPU card in a thermal chamber. The result is an accurate control of temperature increase to accelerate the GPU aging.

Section II of this paper presents the background, and related works on hardware failures, testing protocols, existing benchmarks for the GPU, and previous studies on runtime error detection on the GPU. Section III describes the targeted GPU used for this study. Section IV presents the benchmark platform for GPU self-testing. Section V presents the experiments and results for error characterization. In section VI, we propose

a new methodology to test production GPUs and quarantine the defective hardware. Finally, we discuss the current limitation of our methodology to test and improve the fault tolerance of the GPU.

II. State of the art

A. Source of Hardware Failure

The earliest source of error is the manufacturing process. Some engraving or conception errors can occur. Moreover the engraving process quality may vary inside a single chip. Processors can be locally more sensitive to the voltage and frequency variation which impacts transistors charge and switching delay [6]. Testing the integral circuit is intractable on such a big chip[12], [13]. Thus some failures remain uncovered by factory tests. They are progressively appearing to end-users [14], [15].

A statistical study based on the data from the Folding@home project [8] shows that two thirds of the GPU cards are producing detectable errors. About 2% produce errors with a probability higher than 10^{-4}. However, this rate becomes negligible when it comes to the Nvidia TESLA series.

The out-of-core memory bus on GPU works at a relatively high frequency and very high data rates. It makes them more sensitive to the interference caused by radiation or voltage instability [16]. The solution proposed by Nvidia for its TESLA GPGPU cards is to use lower frequency and to introduce ECC. Despite the positive impact on the error rate, the penalty on the memory bandwidth is significantly high, nearly 25% of the total bandwidth on a TESLA 2050 or 2070. Since the GPU performance is very dependent on the memory bandwidth, the execution time vary with the same order of magnitude. Therefore this functionality is deactivated on many systems and can be replaced by on-line self-testing and redundancy [17], [18].

In the recent years, the voltage has decreased slower than the transistors shrink. The result is an acceleration of processors aging [6], [19] due to electromigration. It consists of the migration of constituent materials of the circuit. With a transistor size getting close to a few atoms, the circuits are getting more and more sensitive to this phenomenon.

B. Standard Protocols for Hardware Error Detection

In the industry, the test protocols for Integrated Circuit (IC), the results classification, and the unit system are defined by the *Joint Electron Device Engineering Council*, JEDEC. The test protocols can be accelerated by changing physical parameters such as voltage or temperature [1], [2], [3].

The standardization permits the constructor to provide comparable data sheets to help the architect engineer to accurately choose the system components. For instance, Xilinx summarizes in [20] the test results for some FPGA components.

JEDEC defines temperature as a good parameter to accelerate IC aging [2]. For the intermittent errors we target in our study, the target temperature for the stress-tests is between 120°C and 160°C.

C. Related Work on GPU

1) GPGPU benchmarking: Even if GPGPU is a relatively recent domain for HPC, there are some benchmark suites such as RODINIA [21], SHOC [22], or PARBOIL [23]. These benchmarks are built around compute intensive kernels of applications. They are useful to measure or compare the performance of a system or the efficiency of an optimization. However they do not feature error detection and characterization mechanisms. Indeed in these benchmarks, an intermittent error can remain silent on the final result.

Some synthetic micro-benchmarks target precise architectural units [24]. However these computations have generally no particular meaning and therefore cannot integrate any error checking. Furthermore we haven't found any of them written in OpenCL which is not compliant with our portability objective.

GPU simulators like barra [25] cannot characterize hardware failures because it requires the real hardware. However they can be useful to evaluate the coverage and cost of new error detection and correction mechanism. Another usage of the simulator can be found in GPGPU-SODA [26]. This is a model based soft-error vulnerabiliity characterization for GPGPU architecture. The objective is to determine with part of the architecture are susceptible to soft-error in function of the workload characteristic. This analytic approach has not been validated by experimental data and the approach is not providing any solution to enhance the GPU reliability.

2) Detection and correction of hardware failure on GPU: The study of Haque et al. [8] of the Folding@home statistics does not characterize the error location, context, and frequency precisely. However it shows that errors can happen not only in memory but also potentially in the GPU processor chip. To our knowledge, there are no available studies targeting intermittent error on GPUs in desktop and HPC systems.

However, some previous works have already addressed the computational errors on the GPU. There are two main approaches to deal with hardware failure: prevention or detection-correction.

In order to prevent an error, it is possible to constrain the GPU usage parameters: working temperature, frequency, cooling devices, or maximum Thermal Design Power (TDP). Nvidia is already using some of these levers as illustrated by this two patents [27], [28]. They control temperature and voltage to avoid failure and prevent aging phenomenon. The test we present in section V-B shows the characteristics we have measured for these mechanisms.

Another solution to leverage the reliability of the system is the error detection and correction. The method can rely on hardware or software mechanism. An example of hardware mechanism can be found in Shaeffer et al. [29]. They are using redundancy and comparator to detect errors and correct with a majority vote. The hauberk tool [30] is a software method to detect and correct errors based on duplication and checksum on the code outside of the loops.

III. Nvidia GPU accelerator description

TESLA architecture is the first CUDA/OpenCL capable architecture. These coprocessors, denoted *devices* in the following, are used as an on-demand support unit by the *host*

Table I. CUDA AND OPENCL TERMINOLOGY

CUDA	OpenCL
Thread	Work-item
Thread block	Work-group
Grid dimension	Local-work-size and global-work-size
Global memory	Global memory
Constant memory	Constant memory
Shared memory	Local memory
Local memory	Private memory

system. We should differentiate the hardware and software organization of threads. Indeed, it is Nvidia driver and hardware responsibility to group and schedule the user threads onto the GPU.

In this section we present the GPU programing models followed by the GPU architecture.

A. GPU programing model

There are currently two main alternatives to program Nvidia GPU devices: CUDA and OpenCL. In order to ease the description of TESLA architecture, we use CUDA terminology. However, Table I shows the correspondence between the terminology of both languages.

From the programmer's point of view, the thread hierarchy is divided in three levels: `threads`, `blocks` and `grid`. The same device code is executed by many `thread` operating in parallel on different data. `Threads` are packed in sets of `block_size` threads to form a so called `blocks`. `Threads` can be synchronized if they belong to the same `block`. `Threads` of a `block` and `blocks` of a `grid` are virtually launched in parallel. This means that no assumption can be done on their execution order. The `grid` is defined by its dimension in a number of blocks. A `block` is identified by its unique coordinate in the `grid`, and a `thread` by its coordinate in the `block`. Therefore, the programmer can use these identification numbers to specify the work done by each `thread` in each `block`.

The host controls the execution steps: create an execution context associated to the target device, allocate memory on the device, initiate the transfer of the input dataset, call the kernel on the device, and transfer the result to the host. When calling the kernel, the host fixes the grid and block dimensions. They have to be chosen carefully by the user to be valid and efficient on the target.

B. The TESLA Architecture

Graphic processors have throughput oriented design exploiting data-parallelism. They integrate a large number of computing units working as a vector processor, i.e. all units are executing the Same Instruction at the same time but on Multiple Data (SIMD). GPUs keep thousands of threads to hide long-latency operations, such as memory references, with parallel concurrency. Figure 1 describes such a device.

TESLA GPUs have up to 128 processing cores called streaming processors (SP), supporting single precision floating point operations, organized in a hierarchical manner. There are organized in 16 multiprocessors of 8 SP each. The thread hierarchy contains 4 levels, threads, warps, cooperative thread

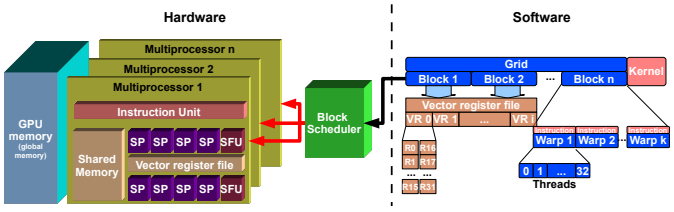


Figure 1. Description of the G80 architecture

array (CTA), and grids. CTAs in TESLA architecture correspond to `blocks` in CUDA. The additional level in TESLA, called warps, is transparent to the programmer. A warp is a group of 32 threads created, managed, scheduled, and executed by a Single-Instruction Multiple-Thread (SIMT) unit. Up to 24 warps (groups of 32 threads) are active per Multiprocessor. The multiprocessor maintains a scoreboard for each warp to prioritize ready warps, according to warp type, instruction type, and fairness. SM maps one warp of 32 threads on 8 SPs. It executes 16 threads (half-warp) every two cycles.

There are several memory devices. For instance the G80 has register banks, a shared memory and a read-only texture cache. Multiprocessor features many different computing units such as fused multiplication and addition (FMA), intrinsic functions (cos,sin, exp, log, sort), interpolation unit, texturing and filter units. These units address integer and floating point.

IV. THE GPU SELF-TESTING BENCHMARKS

In this section, we describe the proposed software infrastructure based on self-testing. The software stresses specific architectural elements of the accelerator, and detects and stores the errors occurring during the execution. The goal is to characterize the errors by type, frequency of occurrence, location, and consequences. The self testing benchmark suite is composed of a host part and a device part.

A. Host System

It is essential in our protocol to minimize the interference between the device program and the control of the experiment. To do so, we use the OpenCL philosophy and delegate the control to the host code. The C++ code running on the host is generic and can be reused in experiments on various platforms. The host is in charge of:

- Selecting the target device in case of multiple possible OpenCL target devices in the system.
- Allocating memory on the device and transfer the initial input datasets.
- Defining each kernel scheduling parameters, such as local work size and global work size.
- Selecting the pattern of micro-benchmark kernels to execute in sequence.
- Iterating over kernels sequence and repeat.
- Controling the test duration.
- Controling the number of iterations.

- Performing some additional software controls of the working temperature by modulating the charge of the GPU.

The execution parameters are set such that on the targeted devices, each software thread is associated with a unique identifier corresponding to the physical ID of the processing element executing it. The additional software control of temperature calibrate a small latency between kernels call, to lower the average load of the GPU and thus let the cooling system evacuate enough calories. This last step is only useful when the GPU is running at high temperature when for example the aging process has to be accelerated. This was the case for our test as described in the next section.

B. GPU Device

The benchmark is built from multiple micro-benchmarks. Micro-benchmarks are compiled without any optimization, and the generated assembly validity is checked with a disassembler. Among the specific targeted architectural elements, we focus on register banks (*REGkernel*), shared memory (*SHMKernel*) and fused add-multiply (*MADKernel*). Each of these kernels is built from a particular sequence of instructions in a loop. The instructions and the input data are built such as single error propagate up to the final result. This way, errors will be detected be comparing results with the reference results computed before kernel launch. We provide a generic OpenCL template to build new micro-benchmarks. Each new kernel must provide an input dataset, a reference result, and an OpenCL kernel conforming to the signature template. The output produced by kernel executions has to be deterministic.

Algorithm 1 presents the *SHMKernel*. The kernel loads and stores in sequence different memory patterns from the shared memory. If an error occurred, it appears in `vecA` with its position. This code is not using the `vecB` reference result since the result is the input vector itself. Even though the framework is written in OpenCL with portability in mind, each device micro-benchmark has to be architecture dependent to take into account their internal specificities. In this work, each proposed OpenCL micro-kernel targets a very specific part of Nvidia architecture. We demonstrate our protocol for TESLA architecture.

```
#define NBPATTERN 4
__constant uint pattern[NBPATTERN]={0xFFFFFFFF,
                          0xAAAAAAAA, 0x55555555,
                          0x00000000};

__kernel void ShMemKernel(
     __global int * vecA,// Error vector
     __global   float * vecB,// Reference result
     __global   float * vecC,// Input Number
     __global   float * vecD,// Input Number
     const uint nbiter)
{
    uint gid   = get_global_id(0);
    uint lid   = get_local_id(0);
    uint lsize = get_local_size(0);
    int i, j, k;
    __local int localBuffer[SIZE_OF_SH_MEM];

    vecA[gid] = 0;

    for(i=0; i<nbiter; i++){
      for(k=0; k < NBPATTERN; k++){
        // 1: Write the value to the shared memory
        for(j=lid; j<SIZE_OF_SH_MEM; j+=lsize)
          localBuffer[j]=pattern[k];

        // 2: Read the value from shared memory
        for(j=lid; j<SIZE_OF_SH_MEM; j+=lsize){
          if(localBuffer[j] != pattern[k])
            vecA[gid] = j;
        }
      }
    }
}
```

Listing 1. Shared memory micro-benchmark kernel

V. Experimental Results

In this section we describe our experiments on GPUs to analyze their behavior and reliability when used under stress. We focus on intermittent errors due to the IC aging. During 3 months we have tested different Nvidia graphic GPUs from different generations (G80, G92, GT200, GF100, GF114, GK104). We apply internal stress, using intensive micro-benchmarks execution, and external stress using high temperature.

A. Hardware set up

By using the hardware mechanism, we control the physical properties of the experiment, such as the architectural set-up and the chip temperature. The hardware configuration used during the experiment is described in figure 2. The host machine is an HP Z800 desktop with a Xeon E5645@2.4GHz and 3x4GB of RAM. The CPU is connected to the system with an Intel X58 and ICH10 controllers. The host operating system is Ubuntu 12.04 with OpenCL 1.1, CUDA 4.2.1, and Nvida driver 295.41. We set the GPU in an external TESLA D870 case connected to the system using a dedicated Nforce100 bridge. This provides us a convenient way to tune the case without interfering with the host system environment (temperature).

For some tests, we needed to control the temperature (ie. Tests of protection mechanism, acceleration of IC aging). Some of the targeted GPU cards use the ONSEMI ADT7473 [31] chip to monitor temperature and control the fan. For these cards, some of the parameters related to the temperature can be modified by a BIOS update using overclocking tools such as nvflash or nibitor. Unfortunately, the modification cannot be applied to all GPUs. Even by modifying the BIOS, the GPUs cannot reach the range of 160-170°C without physical modification. Therefore, we build a thermal chamber around the TESLA case and modified the fan control. The temperature is monitored using internal probes and an external P3400 thermocouple set as close as possible to the GPU processor chip. The strength of such an approach is the minimum intrusion of measurement on the device by doing the monitoring on the host machine outside the thermal chamber.

B. Frequency Scaling

Manufacturers embed some mechanisms to automatically shut it down or scale down the frequency of the GPU when a certain temperature threshold is reached [27]. In order to

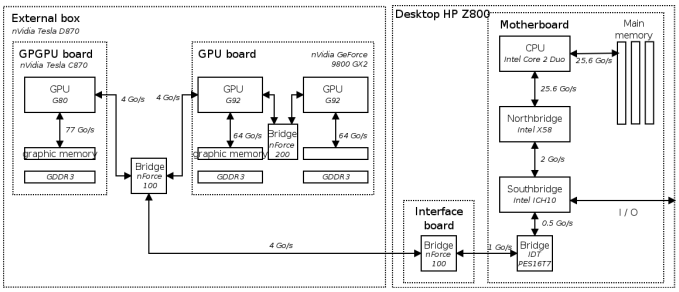


Figure 2. Testing device description.

Table II. THERMAL PROTECTION MECHANISMS CHARACTERISTICS.

GPU : Manufacturer	Fab. Process (nm)	Throttling (°C: MHz)	TSP (°C)
G80 : Nvidia C870	90	*105/110* : 600/300/150/75	-
G92 : Nvidia 9800GX2	65	*105/115* : 600/300/37	130
GT200 : Nvidia T10P	65	*105* : 900/475	115
GF100 : Nvidia GTX480	40	*100* : 700/350	110
GF114 : ASUS GTX560	40	*100* : 810/405	105
GK104 : PNY GTX670	28	*100* : 915/457	100

characterize these mechanisms, we tested the performance of different graphic cards against temperature, without modifying the original firmware. Table II summarizes the results. For each card, we measure the temperatures triggering the frequency scaling and the scaling factor. We run the test up to the Thermal Shutdown Protection. Indeed, TSP cuts the power supply of the card above a certain temperature threshold.

In addition, starting from GF11x, a software mechanism in the driver can scale down the frequency for certain executable (e.g. Furmark, OCCT) to prevent damage on GPU due to overheating. Changing the name of the executable is enough to overrun the protection. From GTX 570/580, a third mechanism monitors the 12v rail for power and stability and scale the frequency accordingly. It ensures a maximum TDP for the GPU and limits the consumption if the system power supply is overloaded.

Figure 3 represents the evolution in time of the temperature for the MAD kernel execution in an infinite loop. The results are obtained on C870 (G80) and 9800GX2 (G92) cards. These graphics show different levels of performance. The C870 has three levels for which the frequency is divided by two with the previous level. Contrary to the TSP, our experiments show that the frequency scaling is reversible. Indeed, when the temperature is getting lower than the activation threshold, the frequency scales back to the previous level. We can observe that the temperature threshold which activate the TSP is getting lower with each generation of graphic card up to the point where it is equivalent to the temperature threshold of throttling. This indicates that the manufacturer is seriously taking into account the IC aging and reliability problem linked with high temperature and prefers to shut down the circuit rather than let it run under stress caused by temperature.

C. Reliability stress Test

We have seen in the previous section that only 9800GX2 and C870 can reach 130°C. First we took two cards of each

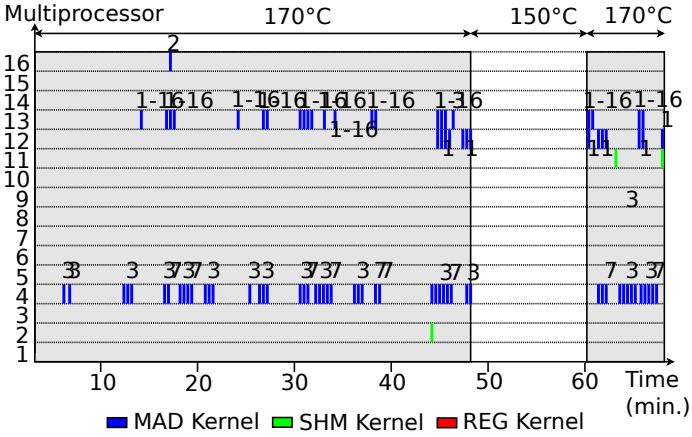


Figure 4. Error localisation in function of time and temperature on the GPU C870 number 1. Single numbers indicate the faulty SP unit. Range numbers such as 1-16 corresponds to a vector error.

generation and made them run at 130°C with our micro-benchmarks running without interruption during 3 weeks. We do not observe any error.

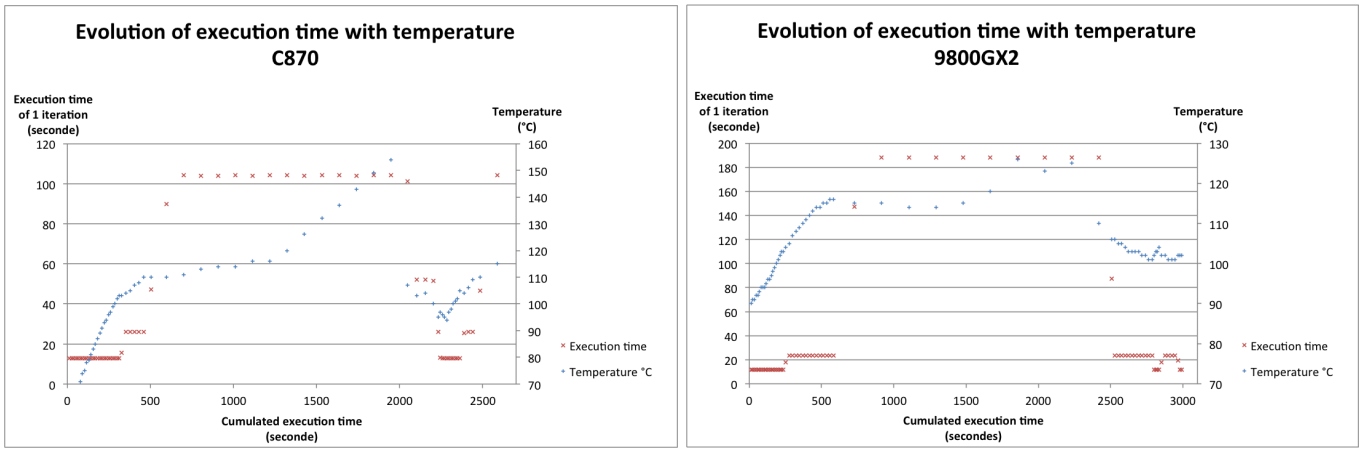
For the two C870, we set the temperature to 160°C. At this temperature we had a maximum variation of +/-5°C. These tests ran for 15 days. After the eleventh day, one of the two cards had a permanent failure, the card was unable to restart. We can not determine if this failure was due to the GPU itself or from a component present on the board. We then took another C870 to continue the test. We then raised the temperature to 170°C and ran the test for 70 minutes. We finally got intermittent errors. The type and distribution of errors are reported in figure 4 and 5.
We can observe that none of the two cards presents error on the register bank. Almost all the errors occurred in the MAD kernel. These errors are of two types:

- **Vectorial errors** affect the results of a complete half-warp (e.g. Results 1 to 16). This means that error arises either at the half-warp instruction dispatch or at the register level. Since the register micro-benchmark shows no errors during all the test, we assume that these errors occurred in the instruction exception pipeline.
- **Scalar errors** imply only a single PE of a TPC. As stated in the literature, these errors occurs in burst mode, i.e. very close in time on a single PE.

While decreasing the temperature from 170°C to 150°C, the intermittent error immediately disappear. At 170°C they reappear immediately. This is due to an irreversible circuit aging phenomenon.

VI. PROPOSED METHODOLOGY TO QUARANTINE HW ERRORS

We have seen in section V that we are able using the proposed micro-benchmarks, to locate units that could return a faulty result. As we have noticed, these errors are silent and except the corrupted result, do not impact the rest of the GPU as the execution continues. In addition, they are confined to a specific area, involving either a single SP or a vector



(a) C870 (b) 9800GX2

Figure 3. Evolution of the cycle per iteration agains temperature.

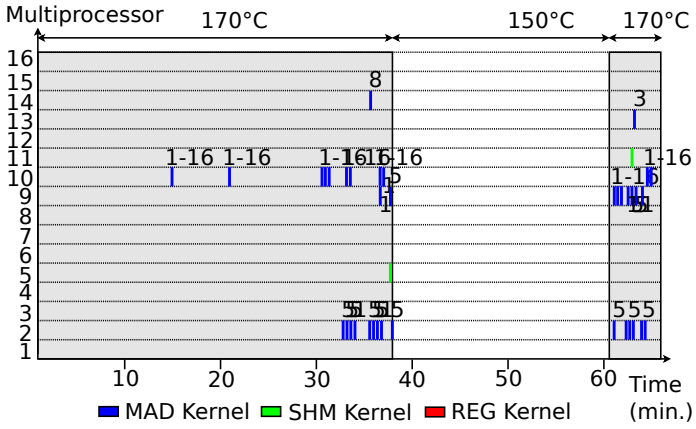


Figure 5. Error localistion in function of time and temperature on the GPU C870 number 2. Single numbers indicate the faulty SP unit. Range numbers such as 1-16 corresponds to a vector error.

(entire SM) and appear sporadically potentially impacting results return by such a GPU. Therefore, it is legitimate to quarantine units that could return a faulty result. There are two options to achieve quarantine, a hardware solution and a software solution. The hardware solution is usually the one adopted by the manufacturers during the factory process who disconnect entire multiprocessor and sells them as model with less compute unit. To our knowledge, there are no proposed solutions that do the same job in software. In this section, we show after describing the scheduling algorithm in use in TESLA architecture, how we are able to quarantine hardware errors in software at both SP and block level. By tuning the scheduling with our methodology, the manufacturers can, after-sales, make the GPU reliable again and prolongate the lifespan of the faulty GPU.

A. TESLA Scheduling Algorithm

To ease the understanding of the link between the hardware and the software, we use the CUDA terminology for the execution parameter set in software. Without loss of generality we only consider execution parameter in one dimension, even though it is possible to set some of these parameters in higher dimension where,

- $GridDim$ is the number of block within the grid
- $BlockDim$ is the number of thread per block
- $BlockID.x$ is the block id x within the grid
- $ThreadID.x$ is the thread id x within the block

In a similar manner, we define the following hardware identifier for a given GPU:

- $NumSM$ is the number of multiprocessor
- $SM.x$ is the multiprocessor number x
- $SP.x$ is the Streaming Processor number x

In this definition, we omit the Thread Processor Cluster level, present in the TESLA architecture. It has been shown in [32] that the scheduling of block regarding TPC and SM is static and depends only of the graphic cards and/or driver version. During our test, we noticed that when an error appears for a given SP at the hardware level, it corresponds to the same $ThreadID.x$ and $BlockID.x$ for an identical execution parameter at software level. Out of this observation, we deduce that block number b_x is executed by the multiprocessor number mp_y such that:

$$mp_y = b_x \mod NumSM$$

and within a given SM, thread number t_z is executed by SP number sp_w such that

$$sp_w = t_z \mod 8$$

where 8 corresponds to the number of SP per SM.

B. Impact of a corrupted unit

In section V-C, we have shown how to detect and locate a faulty Streaming Processor of a multiprocessor. Let $SP_{(i,j)}$ be the faulty SP number i $(0 \leq i \leq 7)$ of the multiprocessor number k $(0 \leq k \leq 15)$.

We now have to determine which software threads can be impacted by a hardware failure. Out of the scheduling algorithm we know that within a SM, the faulty $SP.i$ executes every software thread $ThreadID.x$ such that $(ThreadID.x) \mod 8 == SP.i$. Therefore, out of the $BlockDim$ threads launched on this multiprocessor, $\lceil BlockDim/8 \rceil$ can be potentially corrupted.

In a similar fashion, based on the scheduling results presented in [32] and in this paper, we deduce that the faulty $SM.k$ will execute every $BlockID.x$ such that $(BlockID.x) \mod NumSM == BlockID.x$.

C. Solutions to Quarantine Defective Hardware

To avoid faulty units, we propose to exploit a characteristic of the data parallel programming of GPUs such as CUDA or OpenCL. In this programming model, threads are independent and no communication is allowed between two threads of two different blocks. In addition, it is not possible to make any assumption about the scheduling of software threads of a given block as threads can be executed in any order. However, it is possible to define synchronization barrier between threads of a given block. We propose to avoid corrupted units using this important property. There are two solutions, either making correct SP do the job of corrupted SP and increasing ILP or by launching extra threads and/or extra blocks. We propose to detail the second solution which implies modifications of the execution parameters of the kernel as well as the kernel itself.

Let call an *instruction block*, a block of instructions located between two synchronization barriers. Let N be the original number of Threads per block, and M the number of blocks for the execution parameter of the kernel. For one faulty SP indexed $corr\_SP$ of the multiprocessor indexed $corr\_SM$, the modified kernel will be launched with the same number M of blocks and $N + \lceil N/8 \rceil + \lceil N/64 \rceil$ threads. One can notice that this modification is working as long as the new configuration still meets the requirement of the hardware (512 threads per block for TESLA architecture). Each instruction block of listings 2 can automatically be translated into modified instruction blocks as shown in listings 3. The modification exploits the SIMT principle, where each thread can follow its own execution path leading to divergent threads. However, one can observe that divergence is contained. When a block is executed on a *normal* SM, only the code from line 2 to 6 is executed. In this case the overhead is limited to the test (line 2) and the management of the extra warps that are not doing any work (line 4). When a block is executed on a SM having a corrupted SP of index $Corr\_SP$ (line 7 to 18), only correct SP are doing work (line 9), and the extra launched threads are used to make the others SP do the work that should have been done by the corrupted one.

```
1  int idx = ThreadIdx.x;
2   Instruction_Block1(idx);
3  __syncthreads();
4   Instruction Block2(idx);
5  __syncthreads();
6   ...
```

Listing 2. Original instruction blocks of a given kernel

```
1  int idx;
2  if ((BlockID.x) mod numSM) != corr_SM){
3     // This is a trusted SM
4     if (ThreadIdx.x <N){
5        idx = ThreadIdx.x;
6        Instruction_Block1(idx);
7  } }else {
8     // This block is executed by a
   SM having a corrupted SP
9     if (ThreadID.x mod 8 != corr_SP){
10       if (ThreadIdx.x < N){
11          idx = ThreadIdx.x;
12          Instruction_Block1(idx);
13       }else if (ThreadIdx.x <(N+N/8)){
14          idx =  (ThreadIdx.x - N)*8 + Corr_SP;
15          Instruction_Block1(idx);
16       }else {
17          idx = (ThreadIdx.x - N - N/8)*8 + Corr_SP;
18          Instruction_Block1(idx);
19       }
20    }
21 }
22 __syncthreads();
```

Listing 3. Modified instruction block of a given kernel when there is one faulty SP indexed $corr\_SP$ in the multiprocessor indexed $corr\_SM$

The described mechanism works as long as no assumptions are done on the thread scheduling as mentioned in CUDA or OpenCL specifications. However, we noticed during our tests, that this assumption is not always respected. Some developers are using the fact that all the threads of the same warp execute in one single step. For example, this is the case for the last few steps of the reduction program provided with the OpenCL/CUDA Nvidia sdk. A solution would be to force developer to respect the philosophy of the data parallel programing model and do not make such assumptions about the hardware.

In a similar manner it is possible to alleviate the impact of vector error. In this case, the entire multiprocessor were those errors arise are unreliable and should be avoided. It is possible to do so by launching extra $M/NumSM$ blocks and using the same test as described in listing 3, line 2 to avoid the faulty multiprocessor.

VII. Conclusion

Current processors are more and more sensitive to hardware failures and intermittent errors. One of the key factors of this error is temperature. During our tests, we have observed that GPUs also suffer from intermittent errors. It may explain why manufacturers have lowered the threshold of thermal protection mechanism at each generation over the past 6 years. However, we believe that the internal structure of these processors makes GPU behave differently when such errors arise. First, because GPUs are made by duplicating many times compute units and an error in one unit does not imply that the entire processor is corrupted. Second, because these units are highly structured and the programming model which addresses these processors make it possible to quarantine faulty units.

In this article, we have seen three major contributions improving the reliability of GPUs. The first one, is GPUBurn, a framework implemented in OpenCL made of several micro-benchmarks. This framework addresses three different hardware units of the TESLA architecture. GPUBurn can be easily extended to other units and architectures. The second contribution of this paper is to investigate which units of TESLA architecture are the most sensitive to IC aging. This

is possible with a thermal chamber where GPUs are running at high temperature. We observe that most of the errors were concentrated on streaming processor or multiprocessor level. The third contribution is a set of software modifications of the execution kernel to quarantine faulty units at either streaming processor or multiprocessor level when detected. Our final result is a methodology to detect hardware errors, and mitigate them by avoiding defective units. This ensure the program correctness on GPU architectures, improving the processors fault-tolerance capability and lifespan.

In this work we have observed GPU's behaviour ragarding IC aging due to high temperature. Comparing these errors with errors when others parameters are modified such as voltage and frequency remains to be done. As other future works, we plan to measure the impact in terms of performance and reliability of the proposed solution to avoid defective units with the help of a simulator.

ACKNOWLEDGMENT

We thank the Itea2-H4H project for its support and funding.

REFERENCES

[1] JEDEC, "Foundry process qualification guidelines," *JEDEC technical report*, 2004.

[2] JEDEC, "Accelerated moisture resistance-unbiased autoclave," *JEDEC technical report*, 2008.

[3] JEDEC, "Stress-test-driven qualification of integrated circuits," *JEDEC technical report*, 2011.

[4] P. Rech, C. Aguiar, R. Ferreira, M. Silvestri, A. Griffoni, C. Frost, and L. Carro, "Neutron-induced soft errors in graphic processing units," in *Radiation Effects Data Workshop (REDW), 2012 IEEE*, 2012, pp. 1–6.

[5] J. Guilhemsang, O. Héron, N. Ventroux, O. Goncalves, and A. Giulieri, "Impact of the application activity on intermittent faults in embedded systems," in *VTS*. IEEE Computer Society, 2011, pp. 191–196.

[6] J. Guilhemsang, "Test en ligne pour la détection des fautes intermittentes dans les architectures multiprocesseurs embarquées," THESE, Université de Nice Sophia-Antipolis, Apr. 2011. [Online]. Available: http://tel.archives-ouvertes.fr/tel-00640599

[7] O. Héron, J. Guilhemsang, N. Ventroux, and A. Giulieri, "Analysis of on-line self-testing policies for real-time embedded multiprocessors in dsm technologies," in *IOLTS*. IEEE, 2010, pp. 49–55.

[8] I. S. Haque and V. S. Pande, "Hard data on soft errors: A large-scale assessment of real-world error rates in gpgpu," in *Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, ser. CCGRID '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 691–696.

[9] X. Yang, X. Liao, W. Xu, J. Song, Q. Hu, J. Su, L. Xiao, K. Lu, Q. Dou, J. Jiang, and C. Yang, "Th-1: China's first petaflop supercomputer," *Frontiers of Computer Science in China*, vol. 4, pp. 445–455, 2010, 10.1007/s11704-010-0383-x.

[10] B. Schroeder, E. Pinheiro, and W.-D. Weber, "Dram errors in the wild: a large-scale field study," *Commun. ACM*, vol. 54, no. 2, pp. 100–107, 2011.

[11] KHRONOS, "Opencl official website," *http://www.khronos.org/opencl/*, 2012.

[12] A. Aharon, D. Goodman, M. Levinger, Y. Lichtenstein, Y. Malka, C. Metzger, M. Molcho, G. Shurek, and Y. M. C. Metzger, "Test program generation for functional verification of powerpc processors in ibm," 1995.

[13] I. Wagner and V. Bertacco, *Post-Silicon and Runtime Verification for Modern Processors*. New York, USA: Springer, 2011.

[14] D. Koncaliev, "Intel fdiv bug," *http://www.cs.earlham.edu/ dusko/cs63/fdiv.html*, 1995.

[15] D. Koncaliev, "Intel fpu bug," *http://www.cs.earlham.edu/ dusko/cs63/fpu.html*, 1997.

[16] B. Greskamp, S. R. Sarangi, and J. Torrellas, "Threshold voltage variation effects on aging-related hard failure rates," in *ISCAS*. IEEE, 2007, pp. 1261–1264.

[17] J. Kraus and M. Förster, "Facing the multicore-challenge ii," R. Keller, D. Kramer, and J.-P. Weiss, Eds. Berlin, Heidelberg: Springer-Verlag, 2012, ch. Efficient AMG on heterogeneous systems, pp. 133–146.

[18] J. Lobeiras, M. Amor, and R. Doallo, "Influence of memory access patterns to small-scale fft performance," *The Journal of Supercomputing*, pp. 1–12, 10.1007/s11227-012-0807-5.

[19] E. B. Nightingale, J. R. Douceur, and V. Orgovan, "Cycles, cells and platters: an empirical analysisof hardware failures on a million consumer pcs," in *Proceedings of the sixth conference on Computer systems*, ser. EuroSys '11. New York, NY, USA: ACM, 2011, pp. 343–356.

[20] XILINX, "Device reliability report (ug116)," *http://www.xilinx.com/support/documentation/user_guides/ug116.pdf*, 2010.

[21] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC)*, ser. IISWC '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 44–54.

[22] A. Danalis, G. Marin, C. McCurdy, J. S. Meredith, P. C. Roth, K. Spafford, V. Tipparaju, and J. S. Vetter, "The scalable heterogeneous computing (shoc) benchmark suite," in *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, ser. GPGPU '10. New York, NY, USA: ACM, 2010, pp. 63–74. [Online]. Available: http://doi.acm.org/10.1145/1735688.1735702

[23] J. Stratton, C. Rodrigues, I. Sung, N. Obeid, L. Chang, N. Anssari, G. Liu, and W. Hwu, "Parboil: A revised benchmark suite for scientific and commercial throughput computing," *Center for Reliable and High-Performance Computing*, 2012.

[24] H. Wong, M.-M. Papadopoulou, M. Sadooghi-Alvandi, and A. Moshovos, "Demystifying gpu microarchitecture through microbenchmarking," in *ISPASS*. IEEE Computer Society, 2010.

[25] S. Collange, M. Daumas, D. Defour, and D. Parello, "Barra: A parallel functional simulator for gpgpu," in *Proceedings of the 2010 IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, ser. MASCOTS '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 351–360.

[26] J. Tan, N. Goswami, T. Li, and X. Fu, "Analyzing soft-error vulnerability on gpgpu microarchitecture," in *Workload Characterization (IISWC), 2011 IEEE International Symposium on*, 2011, pp. 226–235.

[27] L. Mimberg, B. Wagner, and M. Lao, "Method and system for dynamic power supply voltage adjustment for a semiconductor integrated circuit device ," *Patent*, no. 10/078,292, February 15 2002.

[28] C. W. Davies and B. M. Kelleher, "Apparatus, system, and method for managing aging of an integrated circuit ," *Patent*, no. 10/882,140, June 29 2004.

[29] J. W. Sheaffer, D. P. Luebke, and K. Skadron, "A hardware redundancy and recovery mechanism for reliable scientific computation on graphics processors," in *Proceedings of the 22nd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*, ser. GH '07. Aire-la-Ville, Switzerland, Switzerland: Eurographics Association, 2007, pp. 55–64.

[30] K. S. Yim, C. Pham, M. Saleheen, Z. Kalbarczyk, and R. Iyer, "Hauberk: Lightweight silent data corruption error detector for gpgpu," in *Proceedings of the 2011 IEEE International Parallel & Distributed Processing Symposium*, ser. IPDPS '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 287–300.

[31] ONSEMI, "dbcool remote thermal monitor and fan control," ONSEMI, http://www.onsemi.com/pub_link/Collateral/ADT7473-D.PDF, Tech. Rep., 2010.

[32] S. Collange, D. Defour, and A. Tisserand, "Power Consuption of GPUs from a Software Perspective," in *ICCS 2009*, ser. Lecture Notes in Computer Science, vol. 5544. Springer, 2009, pp. 922–931. [Online]. Available: http://hal.archives-ouvertes.fr/hal-00348672/en/