Towards Building Error Resilient GPGPU Applications

Bo Fang, Jiesheng Wei, Karthik Pattabiraman, Matei Ripeanu
Department of Electrical and Computer Engineering
University of British Columbia
Vancouver, Canada
Email: {bof, jwei, karthikp, matei}@ece.ubc.ca

***Abstract*—GPUs (Graphics Processing Units) have gained wide adoption as accelerators for general purpose computing. They are widely used in error-sensitive applications, i.e. General Purpose GPU (GPGPU) applications However, the reliability implications of using GPUs are unclear. This paper presents a fault injection study to investigate the end-to-end reliability characteristics of GPGPU applications. The investigation showed that 8% to 40% of the faults result in Silent Data Corruption (SDC). To reduce the percentage of SDCs, we propose heuristics to selectively protect specific elements of the application and design fault detectors based on heuristics. We evaluate the efficacy of the detectors in reducing SDCs and measure performance overheads of the detectors. Our results show that the heuristics are able to reduce the SDC causing faults by 60% on average, while incurring reasonable performance overheads (35% to 95%).**

I. Introduction

GPUs (Graphics Processing Units) have gained wide adoption as accelerators for general purpose computing. However, the reliability implications of using GPUs are unclear. GPUs were originally designed for applications that are intrinsically fault-tolerant: for example image rendering applications where a few wrong pixels are not noticeable by human eyes. As GPUs are used to accelerate a wider class of applications (i.e., General Purpose GPU applications) such as DNA sequencing and linear algebra, it becomes critical to understand the behaviour of these applications in the presence of hardware faults. This is especially important as hardware faults become more and more common in commodity systems due to the effects of technology scaling and manufacturing variations [1]. For example, Haque et al. [2] show that two third out of over 50,000 GPUs exhibit a detectable, pattern of memory soft errors. This rate can be as high as four failures per week for some GPUs.

GPU manufacturers have invested significant effort on improving GPU reliability. For instance, starting with Fermi models, NVIDIA GPUs use Error Correcting Code (ECC) to protect register files, DRAM, cache and on-chip memory space from transient faults. However, hardware faults can occur in the computational or control data paths, and can propagate to registers and/or memory. Such faults would not be detected by ECC in registers and/or memory, as they would cause the correct ECC to be calculated on faulty data. Therefore, GPU applications can still be affected by hardware faults in spite of these mechanisms.

The long-term goal of our work is to develop low-overhead software-based fault-tolerance mechanisms that are tuned to the characteristics of GPGPU applications. As a first step towards this goal, we investigate the end-to-end reliability characteristics of GPGPU applications through fault injection experiments. Our investigation shows that 8% to 40% of the faults result in Silent Data Corruptions (SDCs), or incorrect outputs. This is higher than the fractions of SDCs reported for CPU applications, thus suggesting that GPGPU applications may be more sensitive to errors.

We further design error detectors to selectively protect specific elements of the application, with the goal of reducing the percentages of SDCs. Our results show that the error detectors we designed are able to reduce the number of SDCs by 60% on average, while incurring performance overheads of 55% on average, thus pointing to the potential of using software-based techniques for protecting GPGPU applications.

Other work has proposed the use of hardware redundancy to protect the GPU's computational units [3]. However, hardware approaches have significant performance and energy overheads. An alternative to hardware redundancy is software redundancy, which has the advantage that it can be selectively applied depending on the needs of the application. Software approaches have been extensively explored in the context of CPU-based applications [4]–[6]. However, due to fundamental differences between the CPU and GPU programming models, and the strict performance requirements of GPU applications, these approaches cannot be extended to GPUs in a straightforward manner. The work closest to ours is by Yim et al. [7], who also derive detectors for GPGPU applications with the aim of reducing the SDCs in such applications. Our work differs from theirs in two aspects. First, they perform fault injections at the source code level, while we do so at the executable code level. Because many hardware faults cannot be modelled at the source code level, our injections are more representative of hardware faults. Secondly, Yim et al. focus on "virtual variables" in the GPGPU applications and place the detectors based on a coarse-granularity abstraction (loop and non-loop). In contrast, our detectors are based on generic properties of the GPGPU program's structure. We came up with heuristics that are specific to GPGPU applications and we place error detectors accordingly in the program code.

In summary, our paper makes the following contributions:

1) Evaluates the end to end behaviour of GPGPU applications through fault-injection experiments done at the assembly code level,
2) Develops heuristics for selectively protecting GPGPU applications from SDC(Silent Data Corruption)-causing faults by placing error-detectors in the program,
3) Evaluates the efficacy of the detectors in reducing SDCs and measures performance overheads of the detectors.

II. Reliability Characteristic Study

This section describes the empirical study for evaluating the error resilience of GPU applications. We start by introducing our fault model and then outline the design of the fault injector.

A. Fault model

We consider transient faults in the functional units of the processor. Examples are faults in the ALU and the load-store unit. We do not consider faults in cache, memory and register files, as we assume that these are protected by Error Correcting Code (ECC). We use the single bit flip model to simulate transient faults as done in prior work [8]. We inject faults into the destination register of instructions to simulate an error in the ALU or load-store unit (depending on the instruction). For vector instructions that have multiple destination registers, we randomly choose a destination register to inject.

B. Fault Injector

We have designed a fault injector with the following goals (Figure 1 shows a schematic overview):

1) The injector should have visibility to runtime information of executed instruction stream, to make sure that the injected fault correctly simulates the hardware faults.
2) The injector should interfere minimally with the executed applications. This guarantees that the fault injector itself does not affect the way hardware faults are propagated.
3) The injector should inject faults uniformly in dynamic instructions of applications. This reflects the uniformity of the actual hardware faults on the program's execution.

We achieve these goals by building a fault injector (1) based on the CUDA GPU debugging tool, namely *cuda-gdb* [1]. The fault injector comprises two main phases. First, we profile applications to get the run-time information of different threads (goal 1). Second, we randomly choose one of the executed instructions for fault-injection. The injection is done uniformly over the space of *executed* instructions; thus, we simulate the occurrence of transient errors that occur uniformly over time. The fault injector only injects faults when the chosen instruction is executed (goal 3). This is realized by setting a conditional breakpoint before running the application. When the application hits this breakpoint, a fault is injected into the application (goal 2). Only one fault is injected in each run, as hardware faults are relatively rare events. Our approach is similar to that taken by CPU-based fault injectors such as NFTAPE [9] except that we target GPUs.

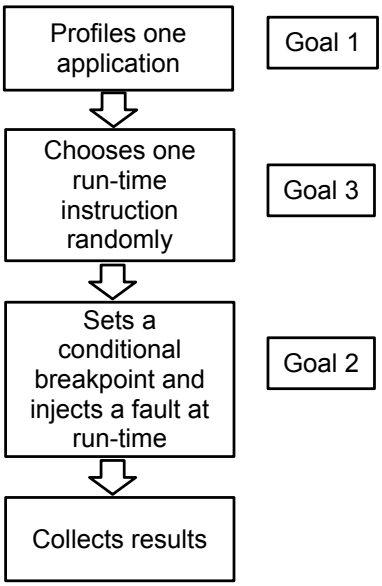

Fig. 1. Design of Fault Injector

Once a fault is injected, we check to make sure that the faulty location is used in the system (i.e., activated). Only activated faults are considered in the evaluation. We then monitor the application to see if it throws an exception (crashes), times out by going into an infinite loop (hangs), or prints incorrect outputs (SDCs). Note that not all activated faults will have an effect on the applications' output - such faults are called benign faults.

We defer the results of the fault-injection to section IV. An interesting result of the evaluation is that silent data corruption (SDC) constitutes 8% to 40% of total outcome across benchmarks. This is consistent with what Yim et al. [7] found, namely that 18 to 45% of faults lead to SDCs, although the injection methodology in our work is different [2]. Such high SDC rates indicate the necessity of reducing SDCs for GPGPU applications, which we consider next.

III. Heuristic-based fault detection

Faults in GPGPU applications lead to high rate of SDCs [7], [8]. One way to reduce the SDCs is to duplicate the entire program. However, this approach leads to considerable power and performance overheads. In contrast, we attempt to selectively protect "important" portions of the application, from the perspective of reducing SDCs. While it is possible that all sections of a program are equally likely to lead to SDCs, in practice we have found that most SDCs are caused by a select few code sections. We call these code sections as *reliability hotspots*.

[1] https://developer.nvidia.com/cuda-gdb

[2] Our injectors are at the assembly code level (NVIDIA SASS), while Yim et al. [7] injects faults at the source code level

More formally, a reliability hotspot is a code section in which a fault is highly likely to lead to an SDC. Our goal is to discover the characteristics of reliability hotspots in terms of the program's structure, so that we can identify such hotspots without relying on fault injection. While fault injection is useful, it is a time consuming process and ideally, one would want to come up with a priori heuristics of what portions of the program to protect based only on code structure. We introduce three heuristic categories to identify the reliability hotspots based on program structure (later we show the error detectors corresponding to these categories).

- Category I: Loop conditions.
- Category II: Branches with block or thread identifier
- Category III: Computation statements that pertain to any of the following:
  1) Initialization of block and thread identifier
  2) Computation involving block or thread id
  3) Computation involving loop iterator variables
  4) Data movement between global memory and other memory regions

Figure 2 illustrates the hotspots that belong to categories I, II and III with CUDA C code [3]. We came up with these categories by analyzing the results of the fault-injection experiment in Section II. We will use LOOP, BRANCH and COMP to denote the three categories respectively. Note that all three categories are based on program structure only and do not need any dynamic information for identifying the hot-spot.

```
1  // Category I
2  for ( int a = aBegin, b= bBegin;
3   a <= aEnd; a+= aStep, b += bStep){
4      //Do something
5  }
6  // Category II
7  // tid is the thread identifier
8  if ( tid<no_of_nodes && g_graph_mask[tid]) {
9      //Do something
10 }
11 // Category III-(1)
12 unsigned int tid =
13 blockIdx.x*MAX_THREADS_PER_BLOCK + threadIdx.x;
14 // Category III-(2)
15 int aBegin = wA * BLOCK_SIZE * by;
16 // Category III-(3)
17 for (int k = 0; k < BLOCK_SIZE; ++k) {
18     Csub  + = AS(ty,k)*BS(k,tx);
19 }
20 // Category III-(4)
21 // C is an array allocated in global memory
22 C[c + wB * ty + tx] = Csub;
```

Fig. 2. Illustration of code example of hotspots

For each category, we designed the corresponding fault detection mechanism based on the characteristics of the program structure corresponding to the category. Table I provides an overview of the error detectors we add to the program for each category. Figure 3 illustrates the error detectors we added to the example of Figure 2; we represent the detectors through *cudaAssert* macros. Note that although we add fault injectors manually, it is feasible to automate the instrumenting process. This is a topic for future work.

[3]Our current implementation works on CUDA, but it is straightforward to extend to OpenCL

TABLE I
FAULT DETECTION MECHANISM

Category	Description	Mechanism
LOOP	Loop condition	Check if the loop iterator is always inbound in the loop body and the loop condition invariants remain the same at the end of the loop
BRANCH	Branch with block or thread identifier	Check if the ID is inbound of the branch condition
COMP	Computation statements	Duplicate the computation and check for a match

```
1  // Category I
2  for ( int a = aBegin, b= bBegin;
3   a <= aEnd; a+= aStep, b += bStep) {
4      //Do something
       //Detector:  cudaAssert(a <= aEnd);
5  }
   //Detector:  cudaAssert(aEnd_check == aEnd)
6  // Category II
7  // tid is the thread identifier
8  if ( tid<no_of_nodes && g_graph_mask[tid]) {
9      //Do something
       //Detector:
       cudaAssert(tid < no_of_nodes);
       cudaAssert(g_graph_mask[tid] == true);
10 }
11 // Category III-(1)
12 unsigned int tid =
13 blockIdx.x*MAX_THREADS_PER_BLOCK + threadIdx.x;
   //Detector:
   cudaAssert(tid ==
   blockIdx.x*MAX_THREADS_PER_BLOCK+thredIdx.x);
14 // Category III-(2)
15 int aBegin = wA * BLOCK_SIZE * by;
   //Detector:
   cudaAssert(aBegin == wA*BLOCK_SIZE*by);
16 // Category III-(3)
17 for (int k = 0; k < BLOCK_SIZE; ++k) {
       //Detector:
       temp = Csub;
18     Csub  + = AS(ty,k)*BS(k,tx);
       //Detctor:
       cudaAssert(Csub == temp+AS(ty,k)*BS(k,tx));
19 }
20 // Category III-(4)
21 // C is an array allocated in global memory
22 C[c + wB * ty + tx] = Csub;
   //Detector: cudaAssert(C[c+WB*ty+tx] == Csub);
```

Fig. 3. Illustration of the error detectors on the code example of Figure 2. The detectors we inserted are indicated as cudaAssert().

IV. EXPERIMENTAL EVALUATION

We use an NVIDIA Telsa C2075 graphic card with CUDA toolkit 4.1 for our experiments. We choose five benchmarks, namely AES encryption (AES) [10], matrix multiplication (MAT) [11], MUMmerGPU (MUM) [12], Breadth First Search (BFS) [13] and LIBOR Monte Carlo (LIB) [14].

We describe the benchmarks and their corresponding configurations.

AES encryption (AES): This application supports both encryption and decryption. We encrypt a 256 KB file with a 256-bit key.

Matrix Multiplication (MAT): MAT is taken from NVIDIAs CUDA SDK 4.1. As a common building block, MAT is widely used in many linear algebra algorithms. We modify the code so that MAT launches the CUDA kernel code only once, to ensure that subsequent runs do not overwrite the results. We multiply two 192*128 floating-point matrices.

MUMmerGPU (MUM): MUM is a parallel sequence alignment program used for processing DNA queries. We use the Bacillus anthracis str. Ames complete genome as the reference and 1000 25-character queries.

Breadth First Search (BFS): BFS performs a breadth-first search on a graph using CUDA programming model. We perform BFS on a graph with 4096 nodes.

LIBOR Monte Carlo (LIB): LIB performs Monte Carlo simulation based on London Interbank Offered Rate Market Model that calculates interest rate for financial business between banks. We simulate 4096 paths for 15 options.

We inject faults into each benchmark 2500 times on average with the same input to ensure that we have a sufficient number of activated faults. Only one fault is injected per run. Overall we have approximately 1500 runs that have activated faults for each benchmark, i.e., the faulty values are used in the program, for each benchmark. The activation rates vary from 30% to 60% among benchmarks. Only activated faults are considered in our results. We categorize the fault based on the application's behaviour as Benign or correct output, Silent Data Corruption (SDC) or incorrect output, Crash, and Hang.

Characteristic study results. The results in Figure 4 show the overall results of the reliability characteristic study. Across the five benchmarks, crashes constitute between 18% and 50% of the outcomes and are the dominant outcomes. Silent data corruptions (SDC) are the second most frequent failure outcomes, observed from 8% to 40% depending on benchmark. The reason for the high number of SDCs could be that a high degree of parallelism of GPGPU applications lowers the complexity of a single thread, which decreases the probability that a fault is masked by the application behaviour.

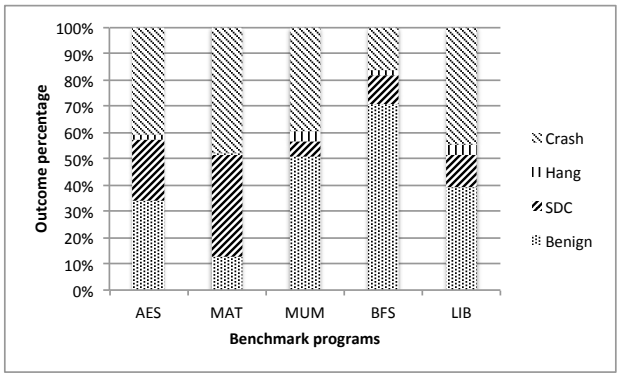

Fig. 4. Overview of fault injection results

Overall detection results The result of the characteristic study shows that SDCs are a serious problem for GPGPU applications. In order to reduce the SDC frequency, as we describe in section III, we embed error detectors in three benchmarks including BFS, MAT and LIB using our heuristics. We select these benchmarks because MAT has the highest SDC rate while BFS has the lowest SDC rate and LIB is in the middle across five benchmarks based on the results of characteristic study. Table II show the number of detectors we inserted for each category.

TABLE II
NUMBER OF DETECTORS INSERTED IN EACH BENCHMARK

Benchmark	LOC of kernel	Number of detectors for each category			
		LOOP	*BRANCH*	*COMP*	*Total*
BFS	44	2	7	9	17
MAT	91	3	0	11	14
LIB	392	36	0	47	83

We run the fault injection experiment again with instrumented benchmarks and evaluate the effectiveness and performance overhead of the fault detectors. To ensure repeatability, we do not stop the program's execution when a detector detects an error, but rather log the heuristic category that this detector belongs to. We then run the program to completion and classify the efficacy of the detector based on the type of failure it would have averted had it stopped the program. Note that we inject faults uniformly in the applications' execution, not just confined to the detectors.

Figure 5 shows the effectiveness of fault detectors on the SDC rates of the three benchmarks [4]. As we can see, the detectors manage to significantly reduce the SDC rates. For BFS, the SDC rate drops from 11.1% to less than 1%; for MAT, the SDC rate drops from 24.7% to 11.6%; and for LIB, the SDC rate drops from 9.1% to 2.7%. On average the fault detectors succeeded in reducing SDC rate from 14.5% to 5.8%, which corresponds to an average coverage of 60% (percentage of SDCs detected). These results demonstrate that the error detectors are able to achieve significant coverage.

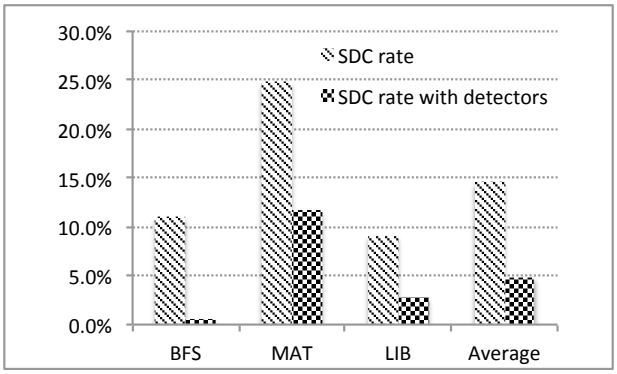

Fig. 5. Fault detection experimental results for instrumented benchmarks.

[4]The baseline SDC rates vary from those in the characteristic study due to the insertion of extra code for the detectors

Breakdown of the coverage and performance overhead To better understand the effectiveness of our fault detectors, we present for each combination of heuristics, the break down the coverage in Figure 6 and the performance overhead in Figure 7. We measure the coverage as the percentage of SDCs caught by each combination of categories. We measure the performance overhead by timing the execution of the GPU kernel ten times and compute the average.

Overall, our detectors achieved the coverages of 94% for BFS, 48% for MAT and 80% for LIB. In particular, for BFS, the most effective category of detectors is COMP, which covers 85% of SDCs, followed by BRANCH, which covers 70% of SDCs. For MAT and LIB, we only present results of two categories since there is no detector of category BRANCH in the code. COMP is also the most effective category for both MAT and LIB as it achieves 47% coverage and 80% coverage respectively. LOOP is ineffective in providing any coverage for all the three programs.

The performance overhead is about 55% on average for all three benchmarks, with BFS and MAT incurring about 35% performance overhead and LIB incurring 95% overhead (the 'ALL' category'). BRANCH incurs only 3% performance overhead, but is able to detect 70% of SDCs for BFS. On the other hand, LOOP is more expensive than BRANCH in terms of performance overhead. However, it is not effective on detecting errors across three benchmarks. COMP is the most expensive type of detectors as shown in the Figure 7 and on average incurs 43% performance overhead, with a wide range from 18% for MAT to 85% for LIB.

As mentioned above, the performance overhead of instrumented LIB is around 95%, primarily due to the COMP category, which contributes to 85% of the overhead. We believe that this is related to a large number of detectors we inserted in this category. In particular, most of the error detectors in COMP category are computation statements involving loop iterators. However, this is not common for GPGPU applications as they do not typically have expensive loops within a single thread, which is how the LIB is implemented. We therefore believe that LIB is an anomalous case and that the overheads are likely to be similar to those of MAT and BFS. Nonetheless, we will investigate other ways to reduce the performance overheads of benchmarks such as LIB. This is a direction for future work.

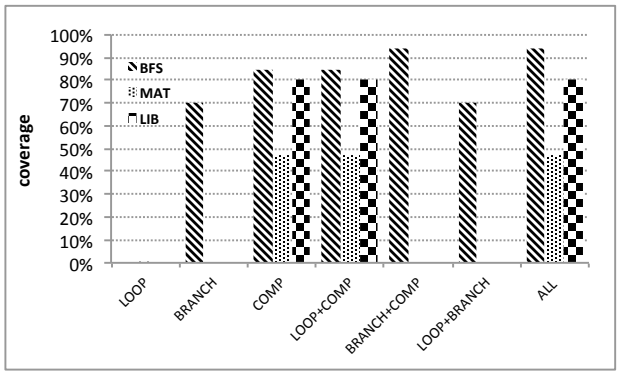

Fig. 6. The breakdown of fault detection coverage of combinations of categories for three benchmarks

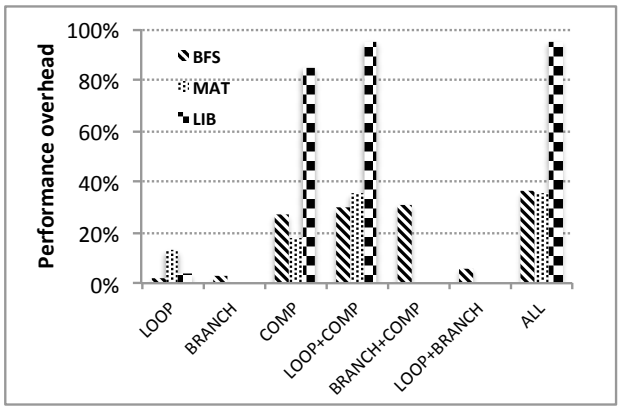

Fig. 7. The breakdown of fault detection performance overhead of combinations of categories for three benchmarks

V. Related Work

Several studies have attempted to characterize the vulnerability of different micro architectural structures in GPUs through AVF analysis [15], [16]. These studies identify micro architectural structures that must be protected from faults in order to achieve high coverage. However, these approaches do not consider the end-to-end impact of faults in applications, nor do they attempt to understand the reliability characteristics of GPGPU applications. AVF analysis has been shown to have significant inaccuracies compared to fault injection [17].

Dimitrov et al. [18] proposes software redundancy approaches to increase the applications' reliability on both NIVIDIA and AMD GPUs. They propose three approaches for GPGPU reliability that leverage both instruction-level parallelism and thread-level parallelism to reduce the overheads. Despite these optimizations, their approach incurs performance overheads of 85 to 100%, and they conclude that understanding both the application characteristics and the hardware platform is necessary for efficient protection.

SWIFT [4] is among the earliest techniques to provide comprehensive error detection coverage at the software level. It uses instruction-level duplication coupled with control-flow checking to ensure high coverage from hardware errors. However, it can incur substantial performance overheads due to the amount of error detection code added, and as such, does not attempt to selectively trade-off coverage for performance. This is the main difference with our work. Further, it focuses on CPU applications and not on GPU applications.

Shoestring [6] aims to reduce SDCs by selectively protecting program instructions that potentially lead to SDCs. It identifies high value instructions that write to global memory or produce function call arguments in the program and apply vulnerability analysis heuristics to the program instruction level for selective duplication. At a high-level, Shoestring is similar to our work in terms of developing heuristics to selectively protect parts of the program. However, it differs

from our work in that it focuses on CPU applications so that the heuristics they develop are not well suited for GPGPU applications.

Thaker et al. [19] observes that errors in control-data are more likely to lead to egregious outputs and catastrophic failures. Wei et al. [20] propose BlockWatch, a compiler-based technique to add detectors to parallel programs with the focus on control data of the program. BlockWatch statically infers the similarity of the program's control-data across threads, and checks their conformance to the inferred similarity at runtime. Our work also uses control-data checks on the thread ID in a manner similar to BlockWatch. However, we consider many other kinds of data in addition to control data, as GPU applications have only a small fraction of control data.

Hari et al. [5] presents a low-cost, program-level fault detection mechanism for reducing SDCs. They use their prior work, Relyzer [21] to profile applications and select a small portion of the program fault site to identify static instructions that are responsible for SDCs. Then by placing program level fault detectors on those SDC-causing sections, they can achieve high SDC coverage at low cost. It is noteworthy that application-specific behaviours are major contributors of SDCs for half of their benchmarks, which makes it difficult to extend their technique to other applications, especially GPU applications which have different behaviours from CPU applications.

As discussed in section I, Yim et al. [7] proposes a technique to detect errors through data duplication at the programming language level (loop code and non-loop code) for GPGPU applications. Their main contribution is to develop application-specific detectors for different code segments in order to reduce performance overhead while achieving high error detection coverage. However, they do not provide any insight into how to choose the program locations at which to place detectors that can achieve the highest coverage, which is our focus. This is important in order to provide high coverage under a performance overhead constraint.

VI. Conclusion

This paper presents a fault injection study to investigate the end-to-end reliability characteristics of GPGPU applications. The investigation showed that 8% to 40% of the faults result in SDCs. To reduce the percentage of SDCs, we propose heuristics to selectively protect specific elements of the application. Our results show that the heuristics are able to reduce the SDC causing faults by 60% on average, while incurring reasonable performance overheads (35% to 95%). These results demonstrate the potential of using software-based techniques for protecting GPGPU applications.

Future work will consist of validating the heuristics with more GPGPU applications and developing more efficient error detectors to further reduce the performance overhead. We will also investigate automated techniques to instrument the program with error detectors.

Acknowledgment

The authors would like to thank Wilson Fung for his help with getting familiar with GPU architectures. We would also like to thank Lauro Beltrao Costa, Elizeu Santos-Neto and Abdullah Gharaibeh for their generous feedback and suggestions on the different phases of this project.

References

[1] C. Constantinescu, "Trends and challenges in vlsi circuit reliability," in *IEEE MICRO*, 2003.

[2] 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, 2010.

[3] J. W. Sheaffer, "A hardware redundancy and recovery mechanism for reliable scientific computation on graphics processors," in *Proceedings of the GH*, 2007.

[4] G. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. August, "SWIFT: Software implemented fault tolerance," in *Intl Symposium on Code Generation and Optimization*, 2005, pp. 243–254.

[5] S. K. S. Hari, S. V. Adve, and H. Naeimi, "Low-cost program-level detectors for reducing silent data corruptions," in *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2012.

[6] S. Feng, "Shoestring: probabilistic soft error reliability on the cheap," in *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, 2010.

[7] K. S. Yim, "Hauberk: Lightweight silent data corruption error detector for gpgpu," in *IEEE International Parallel and Distributed Processing Symposium*, 2011.

[8] B. Fang, J. Wei, K. Pattabiraman, and M. Ripeanu, "Evaluating the error resiliency of gpgpu applications," in *To appear in the poster section of 2012 ACM/IEEE Conference on Supercomputing*, 2012.

[9] D. T. Stott, P. H. Jones, M. Hamman, Z. Kalbarczyk, and R. K. Iyer, "Nftape: Networked fault tolerance and performance evaluator," in *Proceedings of the 2002 International Conference on Dependable Systems and Networks*, ser. DSN '02, Washington, DC, USA, 2002.

[10] S. A. Manavski, "Cuda compatible gpu as an efcient hardware accelerator for aes cryptography," in *IEEE Intl Conf. on Signal Processing and Communication*, 2007.

[11] [Online]. Available: http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Programming_Guide.pdf

[12] M. Schatz, "High-throughput sequence alignment using graphics processing units," in *BMC Bioinformatics*, 2007.

[13] P. Harish, "Accelerating large graph algorithms on the gpu using cuda," in *HiPC*, 2007.

[14] M. Giles and S. Xiaoke, "Notes on using the nvidia 8800 gtx graphics card." [Online]. Available: http://people.maths.ox.ac.uk/œgilesm/hpc/

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

[16] D. K. N. Farazmand, R. Ubal, "Statistical fault injection-based avf analysis of a gpu architecure," in *IEEE Workshop on Silicon Errors in Logic*, 2012.

[17] N. J. Wang, A. Mahesri, and S. J. Patel, "Examining ace analysis reliability estimates using fault-injection," in *Proceedings of the 34th annual international symposium on Computer architecture*, ser. ISCA '07, 2007.

[18] M. Dimitrov, M. Mantor, and H. Zhou, "Understanding software approaches for gpgpu reliability," in *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, 2009, pp. 94–104.

[19] D. Thaker, D. Franklin, J. Oliver, S. Biswas, D. Lockhart, T. Metodi, and F. Chong, "Characterization of error-tolerant applications when protecting control data," in *Proc. IISWC*, 2006.

[20] J. Wei and K. Pattabiraman, "BLOCKWATCH: Leveraging similarity in parallel programs for error detection," in *Proceedings of the IEEE International Conference on Dependable Systems and Networks (DSN)*, 2012.

[21] H. N. Siva Kumar Sastry Hari, Sarita V. Adve and P. Ramachandran, "Relyzer: exploiting application-level fault equivalence to analyze application resiliency to transient faults," in *ACM ASPLOS*, 2012.