PAIS: Parallelization Aware Instruction Scheduling for Improving Soft-error Reliability of GPU-based Systems

Haeseung Lee, Hsinchung Chen, and Mohammad Abdullah Al Faruque
Department of Electrical Engineering and Computer Science
University of California, Irvine, California, USA
{haeseunl, hsinchuc, alfaruqu}@uci.edu

***Abstract*—For decades the semiconductor industry has been driven by Moore's Law and performed aggressive technology scaling to achieve low-power and high-performance. Meanwhile, the semiconductor industry has faced severe reliability challenges like soft-error. Many methodologies (such as redundancy methodologies) have been proposed to improve the soft-error reliability of GPU based systems. However, the GPU compiler has yet to be considered for improving the soft-error reliability of the GPU. In this paper, we propose a novel GPU architecture-aware compilation methodology to further improve the soft-error reliability. The proposed methodology jointly considers the parallel behavior of the GPU and the applications and minimizes the vulnerability of the GPU applications during instruction scheduling. The experimental results show that our methodology is able to perform the scheduling within 5.88 seconds on average and achieves soft-error reliability improvement up to 40% compared to the state-of-the-art compilation techniques. The results show that the performance and power overheads of our methodology are less than 10% in most of the cases.**

I. Introduction

Nowadays, GPUs are widely used in devices ranging from small handheld devices to real-time embedded systems in order to satisfy the growing demand for low-power and high-performance computing [10][20]. In order to provide low-power and high-performance GPU-based systems, the semiconductor industry has done aggressive technology scaling. However, such scaling has also led to severe reliability issues like soft-errors. Since the GPU performs most of the computation-intensive workload, reliability of the GPU is essential for the functionality of the GPU-based systems [25]. Moreover, in [7], it is shown that the real-world GPUs are susceptible to the soft-errors even under normal conditions.

Research has been conducted to improve the soft-error reliability of semiconductor devices including the GPUs. Work in [18] uses instruction scheduling to improve the soft-error reliability of applications. However, since this instruction scheduling is tailored for the RISC processors (SPARC-V8 architecture), it is not scalable to the GPU. Works in [4], [9], and [22] show the problem of soft-error reliability on the GPU by using various methodologies (i.e. GPU memory test), but they do not provide specific solutions to improve the soft-error reliability of the GPU. One proposed technique to optimize soft-error reliability is an application framework for detecting DRAM errors in GPUs and recovering from the checkpoints [13]. Work in [21] has proposed a redundancy methodology, where duplication of the critical parts of the GPU pipeline is used to recompute erroneous results when error is detected. In [24], an application level redundancy methodology has been proposed for minimizing performance overhead of error detection and recomputation by recycling the idle time of the GPU. However, since these research works are based on redundancy or recomputation, the amount of performance and power overhead required to implement their strategies can be significant.

Research has been conducted to leverage the parallel behavior of the GPU to improve the soft-error reliability. The relation between the parallelism of the GPU and reliability is discussed extensively in [6]. Work in [17] shows that modifying the $grid$ and the $block\ size$ of a GPGPU application will affect the parallel execution of the GPU and the Mean Executions Between Failures (MEBF). However, the method to find the reliable $grid$ and $block\ size$ is not proposed.

Research has been conducted to identify and protect the vulnerable parts of GPGPU applications. Work in [5] identifies vulnerable parts from a GPGPU application and inserts checker functions to protect these parts. Work in [12] has proposed a memory access protection methodology by inserting checker instructions into a GPGPU application during compile-time. However, since the soft-errors can occur in any part of a GPGPU application, the improvement of GPU soft-error reliability from these works is limited.

In summary, the above-mentioned state-of-the-art methodologies suffer from the following two limitations:

1) They consider the parallel execution flow of GPU at the application level but they do not provide efficient compilation methodologies to further improve soft-error reliability.
2) They are based on redundancy or recomputation which can lead to significant performance and power overhead.

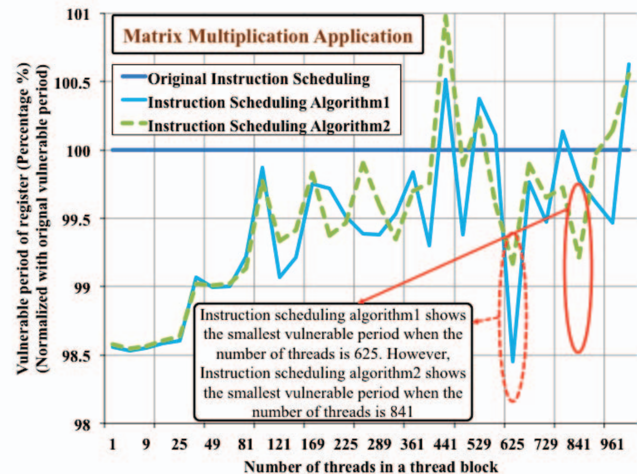

Fig. 1: Motivational Example to Illustrate the Relation Between Vulnerable Period[1] and Instruction Schedule.

A. Motivational Case Study on Vulnerable Period Analysis

In order to observe the impact of instruction scheduling in a vulnerable period[1], we have modified the instruction schedule in a matrix multiplication application and created two additional applications with different instruction schedules. We then measured the vulnerable period of these three matrix multiplication applications (including the original instruction schedule) by using GPGPU-Sim [2] simulator and demonstrate the experimental results in Figure 1. We provide some observations of these experimental results as follows.

[1]The vulnerable period is the metric to measure the soft-error reliability of a GPU application. The vulnerable period is the time from the moment that the data is produced until the last moment that the data is consumed [14]. See section II for more detailed definition and Section III for the vulnerable period estimation technique.

978-3-9815370-6-2/DATE16/©2016 EDAA

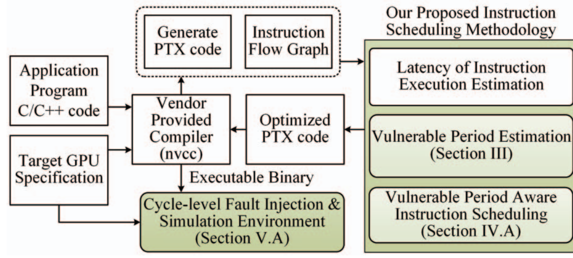

Fig. 2: Overview of the Proposed Soft-error Reliability Improvement Methodology.

Observations:

1) The vulnerable period is fluctuating instead of clearly uptrending/downtrending.
2) A certain instruction schedule does not always have the smallest vulnerable period. For example, *Scheduling Algorithm 1* shows the smallest vulnerable period when the number of threads is 625. However, when the number of threads is 841, *Scheduling Algorithm 2* shows the smallest vulnerable period.
3) Instruction scheduling algorithms may lead to *insufficient protection* if only application behavior is taken into account.

The experimental results imply that the vulnerable period of an application is affected by both the instruction scheduling and the parallel behavior of the GPU. Therefore, an instruction schedule needs to take parallel behavior of the GPU into consideration to further improve the soft-error reliability.

B. Problem and Research Challenges

The problem of compile-time vulnerable period minimization to improve soft-error reliability poses the following *research challenges:*

1) To minimize the vulnerable period of an application during compile-time, it is important to estimate the vulnerable period of GPU applications by considering the accurate GPU execution model.
2) Efficiently decide which instruction should be chosen and where it should be scheduled to minimize the total vulnerable period of an application. Moreover, the solution must ensure that the application programmer does not experience additional compilation overhead.

C. Our Novel Contributions

To address the above-mentioned challenges, we propose a novel methodology to minimize the vulnerable period of an application on a GPU platform which employs:

1) A **Parallelization Aware Instruction Scheduling (PAIS) Algorithm (Section IV-A)** that minimizes the total vulnerable period of a GPU application by considering the impact of the parallel behavior of the GPU. To support this algorithm, we require:
2) **Estimation of the vulnerable period during compile-time (Section III)** of a GPU application, which provides the thread-level information for the GPU.
3) **A fine-grained clock cycle-level fault injection tool (Section V-A)** to verify our methodology. The fault injection tool that can randomly inject faults at the granularity of clock cycle-level is integrated with the state-of-the-art GPGPU-Sim simulator.

Figure 2 shows the overview of the methodology in this paper. To the best of our knowledge, this work is the first GPU compiler work that uses instruction scheduling to improve the soft-error reliability. The proposed compilation methodology bears strong potential for reliability improvement for GPU and can additionally be applied with hardware oriented techniques such as [1], [3], and [26].

II. System Models

We consider a GPGPU application as a target GPU application. A typical GPGPU application is composed of a host code and a kernel code. The former is executed on a CPU and the latter is on a GPU. The proposed instruction scheduling and vulnerable period estimation algorithms are designed based on architecture similar to the Fermi[2] architecture which is proposed by NVIDIA [15]. Our target GPU contains several *Streaming Multiprocessors* (SMs), which consist of many stream processors.

Application Model: The host orchestrates overall behavior of a GPGPU application and the kernel handles most of the computation in the GPGPU application. When the host launches a GPU kernel, the host must set the $configuration$ for the GPU kernel to create the thread hierarchy, where the $grid$ indicates the dimension of the thread block and the $blocksize$ indicates the number of threads in the thread block.

GPU and Fault Model: During execution, based on the $configuration$, thread blocks are assigned to SMs and the threads in the thread blocks are grouped into several $warps$ which are the smallest execution unit of a GPU. A single instruction is fetched for each $warp$, and every thread in the same $warp$ executes the same instruction in lock-step unless there is a divergent control flow. The execution of the thread blocks is possible in any order; in parallel or in sequence. Due to the complexity of predicting the dynamic behavior of a GPU, we assume that the GPU uses the round robin algorithm to schedule the warps as has been adopted in [2] as well.

In this paper, we consider neutron-induced soft-errors at a given rate and assume that neutron-induced soft-errors are evenly distributed over the GPU area. Among several soft-error reliability quantification methodologies, the *Architectural Vulnerability Factor* (AVF) [14] and the *Instruction Vulnerability Index* (IVI) [19] are two of the most well-known metrics used for quantifying the soft-error reliability. The AVF is the probability of visible errors caused by the soft-errors happened in hardware components. Basically, the AVF is estimated according to the life time of data (from the time it is produced until the last time it is consumed, a temporal effect), i.e. its **vulnerable period**. Instead of solely considering the temporal effect, the IVI further considers the spatial effect, where each hardware component has a different probability of a soft-error occurrence. We consider AVF in this paper due to the fact that a detailed Register Transfer Level (RTL) model of the GPU is not available to the research community. However, our methodology can also be scalable to the IVI metric if the spatial effect can be estimated for GPUs as it has been done for the single-core SPARC-V8 CPU in [19].

III. Estimation of Vulnerable Period on the GPU

During run-time, each $warp$ runs instructions in lock-step and it is assumed that the GPU hardware schedules the $warps$ in a round-robin manner [2]. The latency of the instruction execution is related to the parallel behavior of $warps$ on the pipeline stages of the GPU, where the access latency in the memory system is jointly taken into account [8]. Below we provide more details about the latency of the instruction execution.

A. Latency of Instruction Execution

The $(i+1)^{th}$ instruction will be issued after all $warps$ have issued the ith instruction. Therefore, instruction issue latency, t_{issue},

[2]Both Kepler and Fermi architectures contain similar hardware units to execute the GPU kernel [16].

shown in Figure 3 is represented by the following equation:

$$t_{issue} = \frac{n_{warp} \times sizeof(warp)}{n_{core} \times r_{issue}} \tag{1}$$

where n_{warp} represents the number of $warps$ in a thread block. n_{core} represents the number of streaming processors in a single SM, and r_{issue} represents the instruction issue rate.

For the pipeline stages of a GPU and the number of instructions, n, the execution time is determined by t_{issue} and the instruction execution latency, $L_{exe}(I_n)$. Moreover, a data dependent instruction will be held back at the issue stage until its operands are ready. L_i represents the latency of the instruction that I_i depends on. Therefore, the complete execution time of the n instructions may be described by the following equation:

$$Exec(I_n) = \sum_{i=1}^{n} \Big(L_i + t_{issue} \Big) + L_{exe}(I_n) \tag{2}$$

To accurately measure the latency of the data dependent instructions, the latency of the arithmetic and memory access instructions needs to be considered..

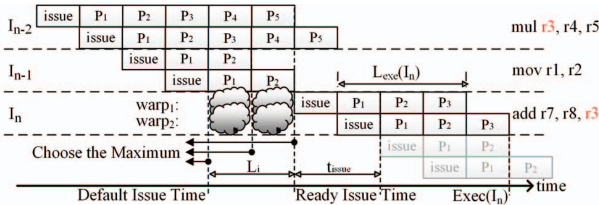

Fig. 3: Example Pipeline Stages of Arithmetic Instructions.

For arithmetic instructions, different execution latencies will lead to different data dependent latencies. The data dependent latency is the period between the default issue time and the ready issue time of a dependent instruction. Figure 3 shows an example of the default and the ready issue time. The default issue time is the time that all the previous instruction is issued. The default issue time could be estimated by the summation of data dependent latency and the t_{issue} for all the instructions preceding the instruction, I_n. The ready issue time is the time that all the data dependency for the instruction I_n is resolved. Note that the ready issue time is always greater and equal to the default issue time. The ready issue time may be estimated by the instruction whose execution time is the longest and has finished after the default issue time of the dependent instructions. Here, we need to look more closely since the issue time always overlaps with the execution latency. Therefore, subtraction of the overlapped issue time from the execution latency is the data dependent latency. In Figure 3, the instruction I_{n-2} actually dominates the dependent latency since it has the longest execution time. The dependent latency is two clock cycles for I_n because one overlapped issue cycle of I_{n-2} and one overlapped issue cycle of I_{n-1} are subtracted from the longest execution time.

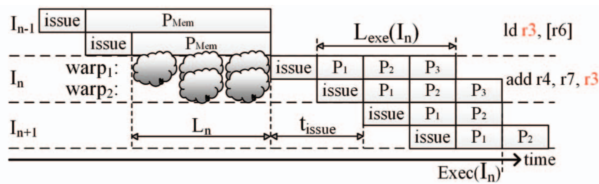

Fig. 4: Example Pipeline Stages of Memory Access Instructions.

For memory access instructions, the execution latency is the same as the memory access latency without the instruction issue latency. Figure 4 shows an example of the memory access latency. Assuming that the system has adopted DDR2 memory, the maximum memory bandwidth, BW_{MEM}, should be the bandwidth of the bus multiplied by the clock frequency of memory. Since it may be impossible to predict the exact cache behavior during compile-time, we use average memory access time to estimate the latency of the memory access instruction.

$$BW_{MEM} = Bus_width \times Memory_freq \times 2 \tag{3}$$

During run-time, each memory access may consume some amount of memory bandwidth, which may be estimated by the following equation:

$$bw = Fetch_size \times Core_freq \tag{4}$$

Therefore, we can obtain the number of concurrent memory accesses in one memory transaction from the following equation:

$$N_{mem_req} = \lfloor BW_{MEM}/bw \rfloor \tag{5}$$

The total number of memory accesses, n_{mem_req}, may be estimated by the following equation:

$$n_{mem_req} = n_{warp} \times n_{SM} \tag{6}$$

We find the total number of memory transactions sent from SMs to memory and then multiply it by the average memory access latency, L_{avg_mem}, since the memory operation is handled sequentially. By using this information, the memory access latency L_{mem} may be estimated by using the following equation:

$$L_{mem} = \lceil \frac{n_{mem_req}}{N_{mem_req}} \rceil \times L_{avg_mem} - \frac{(n_{warp} - 1)}{n_{warp}} \times t_{issue} \tag{7}$$

From this equation we can observe that in order to estimate the execution latency of a memory access instruction, it is necessary to subtract its issue latency from the memory access latency.

Algorithm 1: Algorithm for Computing Vulnerable Period.

```
   Input: Instruction Flow Graph G, configuration c, # of SM N_SM
   Output: Total vulnerable period TotVulnPeriod
 1 Function EstimateVulnPeriod (G, c, N_SM)
 2 begin
 3     InitializeGraph(); // Initialize the node and the edge
       variables
 4     TotVulnPeriod ← 0;
 5     ProgMissMatchFlag ← 0;
 6     G ← DuplicateGraph(G, c, N_SM);
 7     foreach N ∈ G do
 8         t_issue ← GetIssueLatency(N); // Equation 1
 9         SetNodeInfo(t_issue);
10         foreach E_out ∈ N do
11             if E_out ← ∅ then
12                 if N = MemOp then
13                     n_concurrent ← FindConcurrentAccess(N) ;
                       // Get the number of concurrent access
14                     L_exe ← EstimateMemLatency(N, n_concurrent) ;
                       // Equation 3-7
15                 else
16                     L_exe ← GetPipeLatency(N);
17             SetOutgoingEdge(N, E_out, L_exe) ;
18     foreach G ∈ G do
19         Edges ← Null ;
20         foreach E ∈ G do
21             E_longest ← Edges.find(E) ;
22             if E_longest = NULL then
23                 Edges.add(E);
24             else
25                 SetLongest(Edges, E) ;
26     TotVulnPeriod = GetSummation(G, Edges) ;
27     return TotVulnPeriod ;
```

Algorithm 1 shows the pseudo-code used to estimate the total vulnerable period of the application. As mentioned in Section III, we consider the parallel behavior of the GPU to estimate the vulnerable period. The instruction flow graph of the kernel code, G, the configuration of the application, c, and the number of SMs, N_{SM}, are generated as inputs for our algorithm (Line 1). In the graph,

the instructions are represented as nodes whose outgoing edges, if existing, represent data dependencies between the instructions. Afterwards, the graph is updated while considering the degree of parallelism provided by the configuration c and N_{SM} of the target GPU architecture (Line 6). The loop (Lines 7-17) starts to update the weight of the edges which will store L_{exe} from a node to its dependent nodes. For each node in the graph, t_{issue} is estimated (Line 8) and then for each outgoing edge of the node N, the execution latency L_{exe} is determined according to the node type (Lines 10-17). In the second loop (Lines 18-25), the total vulnerable period is explored by searching and adding the longest edge from each node.

IV. Parallelization Aware Instruction Scheduling

As mentioned in Section I-C, our goal is to find the instruction schedule that minimizes the vulnerable period of a GPU application. In order to find an optimal instruction schedule, it is necessary to verify every instruction schedule. However, the finding of an optimal instruction schedule is known to be an **NP-hard** problem [23]. The overhead of finding an optimal instruction schedule that minimizes the vulnerable period could be significant. Therefore, in order to find the best instruction schedule while minimizing the compilation overhead, we propose a heuristic function that schedules the instructions based on the data dependency.

Algorithm 2: Algorithm for Instruction Scheduling.

```
   Input: Instruction Flow Graph G_in
   Output: New Instruction Flow Graph G_new
 1 Function BuildFromBottom (G_in)
 2 begin
 3   G_new.clear();
 4   Pos ← |G_in|;
 5   n_tgt ← ∅; RegSet ← ∅;
 6   while G_in ≠ ∅ do
 7     CandidateSet ← ∅;
 8     pass ← true;
 9     if Pos = |G_in| then
10       n_tgt ← GetSinkNode(); G_new.push(n_tgt);
11       G_in.remove(n_tgt);
12       Pos ← Pos - 1;
13     else
14       for idx = 0; idx < Pos; idx++ do
15         for cnt = idx; cnt < Pos; cnt++ do
16           if G_in[cnt].is_sync() then
17             pass ← false;
18           if pass = true and
             G_in[cnt].is_consume(G_in[idx].GetDest()) then
19             pass ← false;
20         if pass = true and G_in[idx].is_in_loop() then
21           if G_in[idx].GetLastLoopPos() < Pos then
22             pass ← false;
23         if pass = true and RegSet.exist(G_in[idx].GetDest()) then
24           CandidateSet.push(G_in[idx]);
25       if CandidateSet = ∅ then
26         n_tgt ← G_in.GetLastNode();
27       else
28         n_tgt ← CandidateSet.front();
29       G_new.push(n_tgt);
30       G_in.remove(n_tgt);
31       RegSet.RemoveDestRegs(n_tgt);
32       RegSet.AddSrcRegs(n_tgt);
33       Pos ← Pos - 1;
```

A. Vulnerable Period Aware Instruction Scheduling

The vulnerable period of a register will reach a minimum if the distance between the producer and the consumer instructions is zero. The main object of the proposed heuristic is to place the producer instruction as close to the consumer instruction as possible. The proposed heuristic generates an instruction schedule by using a bottom-up strategy that starts from the last instruction of the application. The last instruction is selected and placed by the heuristic at the beginning of the instruction scheduling process. Afterward, the producer instructions corresponding to the last instruction are examined based on the following three conditions:

1) There is no synchronization instruction between the producer and the consumer instruction.
2) There is no other instruction that consumes the same data produced by the producer instruction.
3) The loop depth value of the producer instruction and the consumer instruction must be identical.

If the producer instruction satisfies all three conditions, then this instruction is eligible for scheduling and the heuristic places this instruction near the consumer instruction to minimize the vulnerable period. After that, the placed producer instruction becomes the consumer instruction and the heuristic iterates the above mentioned process until all the instructions are scheduled. **Algorithm 2** shows the pseudo-code for the proposed instruction scheduling algorithm. At the beginning, the last instruction is selected and placed in Lines 10-13. For each pair of producer and consumer instructions, the heuristic examines the three conditions in Lines 15-25. If the pair of instructions satisfies these three conditions, it will be placed near the consumer instruction in Lines 30-34. The overall complexity of the instruction scheduling algorithm is given by $O(n \times n \times n) = O(n^3)$ because of the three levels loop hierarchy.

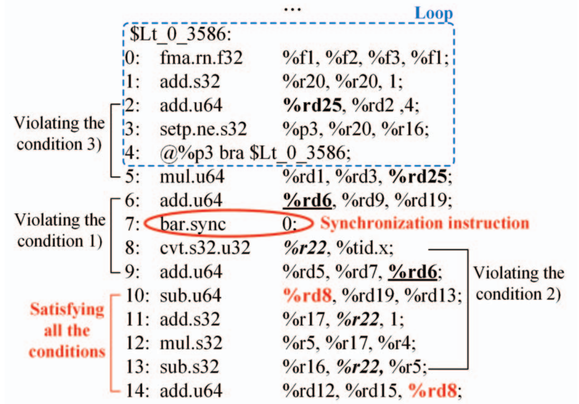

Fig. 5: Example Code for the Proposed Instruction Scheduling.

B. Example of the Proposed Instruction Scheduling

Figure 5 shows an example of the proposed instruction scheduling. In the figure, there are four producer and consumer instruction pairs: instructions in Lines 2 and 5, in Lines 6 and 9, in Lines 8 and 13, and in Lines 10 and 14.

The first pair of instructions, in Lines 2 and 5, violates the third condition because these instructions are not in the same loop. The second pair of instructions, in Lines 6 and 9, violates the first condition due to the synchronization instruction in Line 7. The third pair of instructions, in Lines 8 and 13, violates the second condition, because there is another consumption of the register $\%r22$ in Line 11. However, the last pair of instructions, in Lines 10 and 14, satisfies all three conditions, and therefore the proposed heuristic is able to place the instruction in Line 10 around its consumer instruction, in Line 14.

V. Results and Evaluation

A. Experimental Setup and Fault Injection Tool

Our methodology integrates the NVCC compiler and is capable of generating the kernel code with a minimum vulnerable period. During

the experiment, the proposed PAIS uses timing information that is extracted from the GPGPU-Sim [2] simulator in order to estimate vulnerable period and generate a reliable instruction schedule. The performance and average power consumption is measured by using the GPGPU-Sim and the GPUWattch [11].

For evaluating the soft-error reliability of the proposed PAIS, we have implemented a fine-grained clock cycle-level fault injection tool which is integrated with the GPGPU-Sim, as shown in Figure 6. In our fault injection tool, the kernel is executed twice in the GPGPU-Sim, where the effective fault log is generated during the first execution and the simulation result is returned as feedback for the second execution.

During the first execution, the fault injection tool periodically generates a list of injected faults that includes the faulty components and the clock cycle numbers. In every clock cycle, the fault injection tool checks the information in the injected fault list. An injected fault is considered as an effective fault if the following conditions are satisfied:

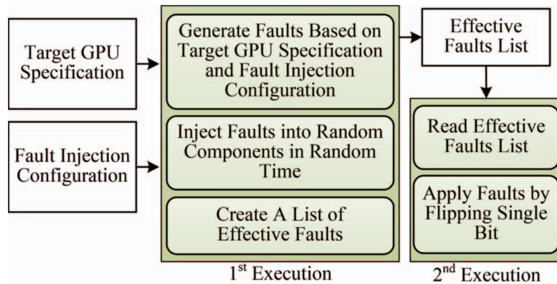

Fig. 6: Experimental Setup for Fault Injection Flow.

- Current clock cycle number is matched with the clock cycle number in the injected fault list.
- The status of the faulty component in the injected fault list is active.

If there is an effective fault, then the fault injection tool writes the following information into the effective fault log: the clock cycle number, SM number, thread id, instruction string, and the faulty component.

During the second execution, the fault injection tool obtains information from the effective fault log and flips one bit from the faulty component. After the second execution, the output is compared to the correct output which is produced without the fault injection. The output is categorized into *correct output*, *incorrect output*, and *application crash*. For our experiments, we chose six benchmark applications for their intensive usage in GPU platforms. These benchmark applications are: Backprop, BFS, Srad, Kmeans, Matrix Multiplication, and Hotspot for basic arithmetic operation. During the experiment, each benchmark application is executed 15 times with various fault injection rates (10, 50, and 100 faults/1 Million (M) cycles).

B. Experimental Results

The PAIS algorithm execution time is obtained using an Intel Core^{TM} i7 Quad-core processor at 3.5 GHz. During the experiment, our PAIS algorithm is able to perform the scheduling within 5.88 seconds on average. Figure 7 shows the vulnerable period improvement of the proposed PAIS. From the result, we observe a significant decrease in the vulnerable period of the applications except for the *Hotspot* application. This is because our algorithm could not cope with the Hotspot application's heavy usage of synchronization instructions in the kernel function (Section IV-A).

Figure 8 compares the soft-error reliability improvement between PAIS, two reliability enhancement methodologies proposed in [5], and

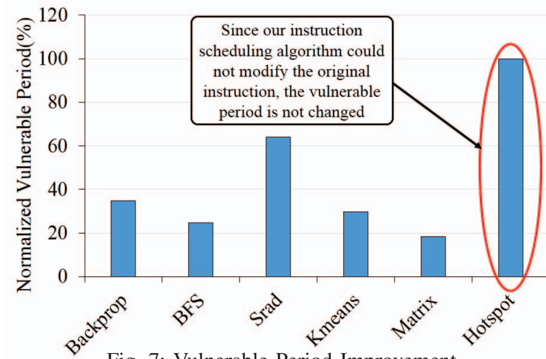

Fig. 7: Vulnerable Period Improvement.

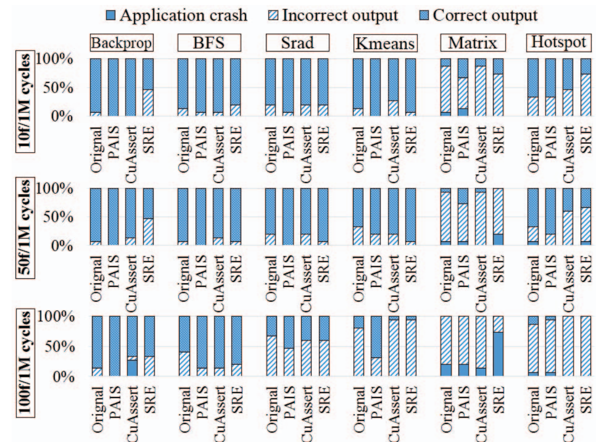

Fig. 8: Soft-error Reliability Improvement Compared to [5], [12]. Each Application is Executed 15 Times with Different Fault Injection Rates.

array bounds checker in [12], respectively. The results demonstrate that in comparison to the other methods, PAIS is generally better and can further improve the soft-error reliability up to 40%.

Since the other works focus on protecting specific parts of an application (i.e. thread index and loop counter), their improvement for the soft-error reliability is limited. Therefore, the methodologies presented in [5] and [12] may be more successful if soft-errors occur on specific parts of applications they have protected. However, if soft-errors randomly occur throughout the application, these methods generally provide worse results because by forcing the use of additional protection functions, they also increase the application's exposure to soft-errors. Therefore, the methodologies in [5] and [12] may have less soft-error reliability improvement than our PAIS.

In addition, we observe that there is no correct output from the *Matrix* application with 100 faults/1M cycles fault injection rate. This is because the *Matrix* application is sensitive to fault. The failure rate of the *Matrix* application is already higher than 70% with 50 faults/1M cycles fault injection rate. Therefore, with 100 faults/1M cycles fault injection rate, the failure of the *Matrix* application is an expected outcome.

Furthermore, the overheads introduced by our PAIS are compared to [5] and [12], where Figures 9 and 10 show the performance and power overheads, respectively. The results show that the proposed PAIS causes a small amount of performance and power overhead compared to the original code because our PAIS does not insert any additional code into the application. However, in the *Backprop* application, a significant amount of performance overhead is caused because of: 1) the scheduling of the load/store instructions, and 2) the modification of the configuration.

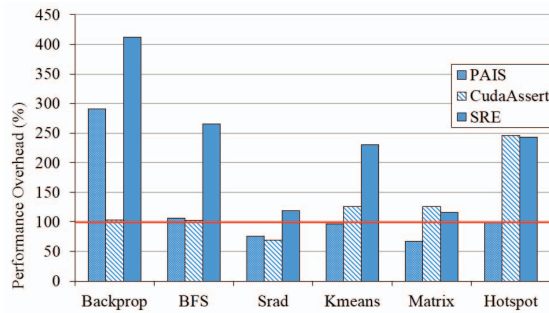

Fig. 9: Performance Overheads Compared to [5], [12].

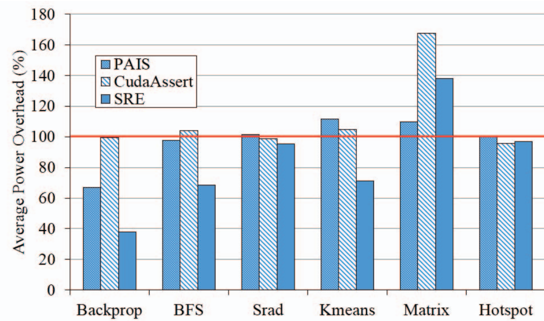

Fig. 10: Average Power Consumption Overheads Compared to [5], [12].

VI. Conclusion

In this paper, we have proposed a GPU architecture-aware compilation methodology to improve the soft-error reliability on the GPU platform. In the methodology, we have developed a novel instruction scheduling algorithm that decreases the vulnerable period to improve the soft-error reliability of hardware registers that hold the application data. We have analyzed the GPU architecture and have constructed the system model to estimate the total vulnerable period of the parallel executing threads on the GPU to find the best configuration. In addition, we have developed a fine-grained fault injection tool that is integrated with the state-of-the-art cycle-level GPU simulator to evaluate the proposed methodology. From the experimental results, we show that our methodology improves soft-error reliability up to 40% on average and our scheduling algorithm can generate a reliable schedule within 5.88 seconds as well. In addition, the results show that the performance and power overheads of our methodology are less than 10% in most cases.

References

[1] N. Avirneni and A. Somani. "Low Overhead Soft Error Mitigation Techniques for High-Performance and Aggressive Designs". *IEEE Transactions on Computers*, 61(4):488–501, 2012.

[2] A. Bakhoda, G. Yuan, W. Fung, H. Wong, and T. Aamodt. "Analyzing CUDA workloads using a detailed GPU simulator". *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'09)*, pages 163–174, 2009.

[3] A. El-Maleh and K. Daud. "Method for synthesizing soft error tolerant combinational circuits", 2014. US Patent 8,640,063.

[4] B. Fang, J. Wei, K. Pattabirama, and M. Ripeanu. "Evaluating Error Resiliency of GPGPU Applications". *High Performance Computing, Networking, Storage and Analysis (SCC'13)*, pages 1502–1503, 2012.

[5] B. Fang, J. Wei, K. Pattabiraman, and M. Ripeanu. "Towards Building Error Resilient GPGPU Applications". *3rd Workshop on Resilient Architecture (WRA'12)*, 2012.

[6] L. B. Gomez, F. Cappello, L. Carro, N. Debardeleben, B. Fang, S. Gurumurthi, K. Pattabiraman, P. Rech, and M. S. Reorda. "GPGPUs: How to Combine High Computational Power with High Reliability". *Design, Automation and Test in Europe Conference and Exhibition (DATE'14)*, pages 1–9, 2014.

[7] I. Haque and V. Pande. "Hard Data on Soft Errors: A Large-Scale Assessment of Real-World Error Rates in GPGPU". *10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing (CCGrid'10)*, pages 691–696, 2010.

[8] S. Hong and H. Kim. "An Integrated GPU Power and Performance Model". *Proceedings of the 37th Annual International Symposium on Computer Architecture (ISCA'10)*, pages 280–289, 2010.

[9] H. Jeon, M. Wilkening, V. Sridharan, S. Gurumurthi, and G. H. Loh. "Architectural Vulnerability Modeling and Analysis of Integrated Graphics Processors". *Workshop on Silicon Errors in Logic-System Effects (SELSE'13)*, pages 1–6, 2013.

[10] H. Lee and M. A. A. Faruque. "GPU-EvR: Run-Time Event Based Real-Time Scheduling Framework on GPGPU Platform". *Design, Automation and Test in Europe Conference and Exhibition (DATE'14)*, pages 1–6, 2014.

[11] J. Leng, T. Hetherington, A. ElTantawy, S. Gilani, N. Kim, T. Aamodt, and V. Reddi. "GPUWattch: Enabling Energy Optimizations in GPGPUs". *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA'13)*, pages 487–498, 2013.

[12] S. Li, N. Farooqui, and S.Yalamanchili. "Software Reliability Enhancements for GPU Applications". *Sixth Workshop on Programmability Issues for Heterogeneous Multicores (MULTIPROG'13)*, 2013.

[13] N. Maruyama, A. Nukada, and S. Matsuoka. "A High-Performance Fault-Tolerant Software Framework for Memory on Commodity GPUs". *IEEE International Symposium on Parallel & Distributed Processing (IPDPS'10)*, pages 1–12, 2010.

[14] S. S. Mukherjee, C. Weaver, J. Emer, S. K. Reinhardt, and T. Austin. "A Systematic Methodology to Compute the Architectural Vulnerability Factors for a High-Performance Microprocessor". *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture (MICRO'03)*, pages 29–40, 2003.

[15] NVIDIA. "NVIDIA's next generation CUDA compute architecture: Fermi". 2009.

[16] NVIDIA. "NVIDIA's next generation CUDA compute architecture: Kepler GK110". 2012.

[17] P. Rech, L. Pilla, P. Navaux, and L. Carro. "Impact of GPUs Parallelism Management on Safety-Critical and HPC Applications Reliability". *44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'14)*, pages 455–466, 2014.

[18] S. Rehman, M. Shafique, and J. Henkel. "Instruction scheduling for reliability-aware compilation". *2012 49th ACM/EDAC/IEEE Design Automation Conference (DAC'12),*, pages 1288–1296, 2012.

[19] S. Rehman, M. Shafique, F. Kriebel, and J. Henkel. "Reliable Software for Unreliable Hardware: Embedded Code Generation Aiming at Reliability". *Proceedings of the seventh IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis (CODES+ISSS'11)*, pages 237–246, 2011.

[20] G. Sadowski. "Design Challenges Facing CPU-GPU-Accelerator Integrated Heterogeneous Systems". *Design Automation Conference (DAC'14)*, 2014.

[21] J. W. Sheaffer, D. P. Luebke, and K. Skadron. "A Hardware Redundancy and Recovery Mechanism for Reliable Scientific Computation on Graphics Processors". *SIGGRAPH/Eurographics Workshop on Graphics Hardware*, pages 55–64, 2007.

[22] G. Shi, J. Enos, M. Showerman, and V. Kindratenko. "On Testing GPU Memory for Hard and Soft Errors". *Proc. Symposium on Application Accelerators in High-Performance Computing (SAAHPC'09)*, pages 1–3, 2009.

[23] G. Shobaki, K. Wilken, and M. Heffernan. "Optimal Trace Scheduling Using Enumeration". *ACM Transactions on Architecture and Code Optimization (TACO'09)*, pages 1–32, 2009.

[24] J. Tan and X. Fu. "RISE: Improving the Streaming Processors Reliability Against Soft Errors in GPGPUs". *Proceedings of the 21st international conference on Parallel architectures and compilation techniques (PACT'12)*, pages 191–200, 2012.

[25] J. Tan, Z. Li, and X. Fu. "Soft-error Reliability and Power Co-optimization for GPGPUS Register File Using Resistive Memory". *Design, Automation and Test in Europe Conference and Exhibition (DATE'15)*, pages 369–374, 2015.

[26] K. Wu and D. Marculescu. "A Low-Cost, Systematic Methodology for Soft Error Robustness of Logic Circuits". *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, pages 367–379, 2013.