RISE: Improving the Streaming Processors Reliability Against Soft Errors in GPGPUs

Jingweijia Tan, Xin Fu
Department of Electrical Engineering and Computer Science
University of Kansas
Lawrence, KS 66045 USA
{jtan, xinfu}@ittc.ku.edu

ABSTRACT

With hundreds of cores integrated into a single chip, the general-purpose computing on graphic processing units (GPGPUs) provide high computing power to accelerate parallel applications. However, they are prone to manifest high soft-error vulnerability due to the lack of fault detection and tolerance. Especially, streaming processors become the reliability hot-spot in GPGPUs. This paper explores two opportunistic soft-error detection techniques to cost-effectively improve the streaming processors reliability. Observing that the streaming processors are not fully utilized during the branch divergence and pipeline stalls caused by the long latency operations, we propose to Recycle the streaming processors Idle time for Soft-Error detection (RISE) and obtain the good fault coverage with negligible performance degradation. RISE is composed of full-RISE and partial-RISE. Full-RISE selectively triggers the redundancy for a set of warps so that leverages the fully idled streaming processors during the pipeline stall time for the error detection. Partial-RISE performs the redundancy for a number of threads in certain warps using the partially idled streaming processors during the branch divergence. Our experimental results show that RISE shows strong capability in improving the SPs soft-error reliability by 43% with negligible (e.g. 4%) performance loss.

Categories and Subject Descriptors

B.8.1 [**Performance and Reliability**]: Reliability, Testing, and Fault-Tolerance; I.3.1 [**Computer Graphics**]: Hardware Architecture – *Graphics processors*

General Terms

Performance, Reliability

Keywords

GPGPU, Reliability, Soft Errors, Streaming multiprocessors

1. Introduction

Modern graphic processing unit (GPU) supports thousands of concurrent threads and provides remarkably higher throughput than CPU for parallel applications. In addition, the programming models (e.g. NVIDIA CUDA [1], AMD Stream [2], OpenCL[3]) facilitate the development of data parallel applications on GPU. With their strong computing power and improved programmability, the General-Purpose Computing on GPUs (GPGPUs) emerge as a highly attractive platform for a wide range of HPC applications exhibiting intensive data-level or thread-level parallelism. This extensive usage of GPGPU makes reliability a critical concern. Traditionally, the dominant workloads on GPUs are graphic processing applications which can effectively mask errors and have relaxed request on computation correctness. The error detection and fault tolerance on GPUs receive little attention. However, the newly adopted HPC applications, such as the scientific computing, financial application and medical data processing, have rigorous requirements on execution correctness. For example, in the GPGPU application computing a correlation function [4], 1% of value errors in any of the program output elements is treated as a silent data corruption (SDC) error and cannot be tolerated.

Soft errors, also called transient faults or single upset events, are failures caused by high-energy neutron or alpha particle strikes in integrated circuits. These faults are said to be "soft" in contrast to hard-faults which are permanent in the device. Soft errors may silently corrupt the data and lead to erroneous computation results. Soft error rate (SER) has been predicted to increase exponentially with the shrinking of feature sizes and growing integration density [5, 6], and GPGPUs with hundreds of cores integrated in a single chip are prone to suffer severe soft error attacks. For examples, [7] has already observed eight soft errors in a 72-hour run of testing program on 60 NVIDIA GeForce 8800GTS 512. [8] finds that the silent data corruption ratio in commodity GPU is 16~33%, while the ratio is smaller than 2.3% in CPUs which have comparatively developed fault tolerance techniques. The increasing SER becomes the major obstacle to current and future GPGPU design.

There are several parallel streaming processors (SPs) in each GPGPU core. They perform the fundamental computing operations, and play an important role in exploiting the parallel computing. In the GPGPU with thousands of parallel threads, SPs execute numerous instructions and expose them to neutron and alpha particle strikes, leading to the high SER. We evaluate the SPs soft-error vulnerability by computing its architecture vulnerability factor (AVF) which estimates the possibility that a transient fault in the structure will produce incorrect computation output, and find that the AVF is 53% on average (detailed description on AVF can be found in Section 4). In addition, as the key and representative combinational logic-based structure in the GPGPU, SPs occupy a large fraction of the chip area, the SER of SPs becomes the major contributor to the overall SER of the GPGPU processor. Effectively protecting SPs becomes the essential first step to build the resilient GPGPU architecture.

To overcome the reliability challenge, duplication [9] is the well-known classical technique: all instructions can be redundantly executed in the SPs and two copies are compared for soft-error detection. However, the simple duplication will lead to high performance penalty. Based on our experiments on various benchmarks, we found that on average, the full redundancy results in 58% performance degradation. The high computing throughput is the critical feature provided by GPGPUs. In general, substantially sacrificing the performance to achieve the perfect error coverage is not an efficient solution to the reliability issue in GPGPUs. A good trade-off between performance and reliability is more acceptable and desirable in future commodity GPGPU design.

GPGPU exploits thread-level parallelism (TLP) by grouping a number of threads into a warp which simultaneously executes the same instruction from each thread, warp are interleaved at cycle-by-cycle basis to hide the latency caused by the data dependence between consecutive instructions from a single warp. However, SPs are unused when all warps stall in the pipeline due to the long latency operations (e.g. off-chip memory access). In addition, threads in the same warp may direct to different paths at a branch, they execute sequentially in the diverged warp and partial SPs in the GPU core become idle [10]. We investigate a large set of GPGPU benchmarks, and find that on average, all SPs in a GPU core are idle during 35% of the total execution time, and the case that the SPs are partially idle appears in 11% of the execution time. The large fraction of the SPs idle time provides great opportunities to trigger the partial redundancy and trade-off little performance degradation for maximal reliability improvement. In this paper, we propose RISE (Recycling the streaming processors Idle time for Soft-Error detection) which intelligently leverages the under-utilized SPs to perform the redundancy.

The contributions of this work are as follows:

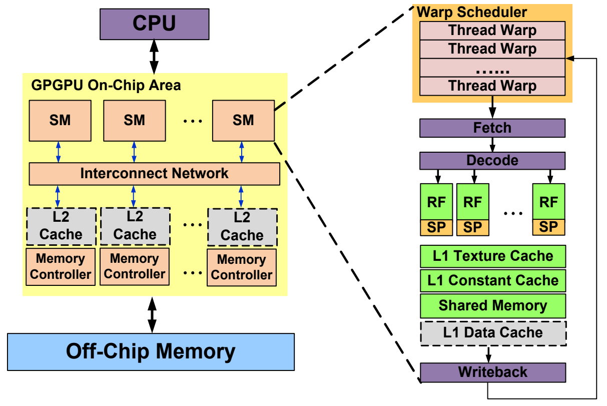
- We propose full-RISE that effectively re-uses the *fully* idled SPs in the GPU core caused by the long-latency memory accesses and the load imbalance among cores for soft-error detection. Full-RISE is composed of the request pending aware Full-RISE (RP-FRISE) and idle streaming multiprocessors (SMs) aware Full-RISE (IS-FRISE). RP-FRISE predicts the warp stall time for its next off-chip memory access and appropriately delays the warp progress via the redundant execution, therefore, successfully recycles the stall time for redundancy without degrading the performance. IS-FRISE estimates the load imbalances among cores, and smartly triggers the redundancy in cores which are predicted to execute fewer threads.
- We propose partial-RISE that redundantly executes a number of threads from a warp by combining their execution with the diverged warp, therefore utilizes the *partially* idled SPs during the branch divergence for reliability optimization.
- Our experiments show that both full-RISE and partial-RISE are capable of optimizing the SPs soft-error robustness substantially with negligible performance penalty. As the integration of the two techniques, RISE is able to enhance the SPs vulnerability by 43% with only 4% performance loss. Based on our sensitivity analysis, RISE is also applicable to GPU architecture designs with various performance optimization schemes.

The rest of this paper is organized as follows: Section 2 provides background on state-of-the-art GPGPU architecture. Section 3 presents our full-RISE and partial-RISE techniques. Section 4 describes our experimental methodologies. Section 5 evaluates the proposed techniques. We discuss the related work in Section 6, and conclude the paper in Section 7.

2. Background: General-purpose computing on graphic processing units (GPGPU) architecture

Figure 1(a) shows an overview of the state-of-the-art GPU Architecture [1]. It consists of a scalable number of in-order streaming multiprocessors (SM) that can access to multiple memory controllers via an on-chip interconnection network. Figure 1(b) illustrates a zoom-in view of the SM. It contains the warp scheduler, register files, streaming processors (SPs), constant and texture caches, and shared memory. In addition to CPU main memory, the GPU device has its own off-chip external memory (e.g. global memory) connected to the on-chip memory controllers. Some high-end GPUs also have L2 cache (shown as dotted line in Figure 1), and error-checking-correction unit for both on-chip and off-chip memory [11].

In this paper, we study the NVIDIA CUDA programming model but some of the basic constructs will hold for most GPU programming models. In CUDA, the GPU is treated as a co-processor that executes highly-parallel kernel functions launched by the CPU. The kernel is composed of a grid of light-weighted threads; a grid is divided into a set of blocks (referred as cooperative thread arrays in CUDA); each block is composed of hundreds of threads. Threads are distributed to the SMs at the granularity of blocks, and threads within a single block communicate via the shared memory and synchronize at a barrier if desired. Per-block resources, such as registers, shared memory, and thread slots in an SM are not released until all the threads in the block finish execution. More than one block can be assigned to a single SM if resources are available.



(a) An overview **(b) SM microarchitecture**

Figure 1. General-Purpose Computing on Graphic Processing Units (GPGPU) architecture

Threads in the SM execute on the single-program multiple-data model. A number of individual threads (e.g. 32 threads) from the same block are grouped together, called warp. In the pipeline, threads within a warp execute the same instruction but with different data values. Figure 1(b) also presents the details of SM microarchitecture. Each warp has a dedicated slot in the scheduler which records the warp ID (WID), active mask describing the active threads in the warp, and a single PC.

At every cycle, a ready warp is selected by the scheduler to feed the pipeline. The execution of a branch instruction in the warp may cause warp divergence, threads in a diverged warp have to execute in serial fashion which greatly degrades the performance. In the pipeline, a long latency off-chip memory access from one thread would stall all the threads within a warp, and the warp cannot proceed until all the memory transactions complete. The load/store requests issued by different threads can get coalesced into fewer memory requests according to the access pattern.

3. RISE: Recycling the Streaming Processors Idle Time for Soft-Error Detection

In this section, we propose to intelligently Recycle the stream processors Idle time for Soft-Error detection (RISE). It leverages the SPs idle time to execute the redundant threads and substantially improve the SPs reliability with negligible performance loss. We present the full-RISE and partial-RISE in Section 3.1 and 3.2, respectively.

When implementing the redundant multithreading technique in CPUs, the main and redundant threads share the pipeline resources, no extra hardware resources are required to support the redundant thread. However, every parallel thread in GPGPU has statically allocated resources including the register files and on-chip shared memory. Those per-thread resources have to be double-sized to successfully launch the main and redundant threads simultaneously into the GPU core, which leads to extremely high resource usage and power consumption [12]. In order to avoid this high overhead, the redundant thread in our study will use the same per-thread resources with the main thread. In other words, the redundant thread follows the main thread immediately, they interleave at the instruction-by-instruction basis and execute at the same speed.

In this study, we focus on the single-bit error model which has the first order impact on the failure rate in processors [35]. Note that SPs will operate the computations from main and redundant threads in two consecutive cycles, our technique is sufficient to detect the single-bit errors in SPs. Moreover, it is also able to catch the multiple-bit errors occurring simultaneously in the SPs resulting from either single upset event or multiple, independent upsets. The particle-strikes which could last for more than a cycle in a single bit are not considered in this paper.

3.1 Full-RISE

In previous work, the opportunistic transient-fault detection in CPUs has been proposed by Gomma et al. [13]. It keeps the progress difference between the main and redundant threads, and triggers the redundant thread when the main thread stalls for the long-latency operations. Therefore, the redundant thread can efficiently leverage the under-utilized pipeline resources to perform the redundancy without degrading the performance. However, the technique is not applicable to the SPs soft-error detection in GPGPUs since both main and redundant threads in GPUs have to keep the same progress (as we described above), and they stall at the same time. A novel technique is desired to intelligently trigger the redundant thread ahead of the pipeline stalls in the SM. In this subsection, we propose Full-RISE which is composed of two methodologies: request pending aware full-RISE and idle SMs aware full-RISE.

3.1.1. Request Pending Aware Full-RISE

3.1.1.1. The Observation on Resource Contentions among Memory Requests

In CUDA programming model, all threads in a kernel execute the same code. Moreover, warps in the GPU SM interleave at cycle-by-cycle basis and exhibit similar execution progress. When one warp sends out a request for the off-chip memory access, other warps are likely to issue the requests at approximately the same time. The sudden burst of the memory requests will cause the congestion in the interconnect network and the memory controller, leading to the severe resource contentions. Especially, the input buffer in the memory controller will be quickly filled up by the requests. Generally, the out-of-order (OoO) first-ready first-come first-serve scheduling policy is applied in the memory controller. It grants higher priority to the requests which hit an open row in the DRAM, and save the time spent on precharge and activation to open a new row. However, its impact on alleviating the request congestion is ambiguous since a limited number of requests (e.g. only one) can be serviced at a time, and most requests are experiencing the long waiting time.

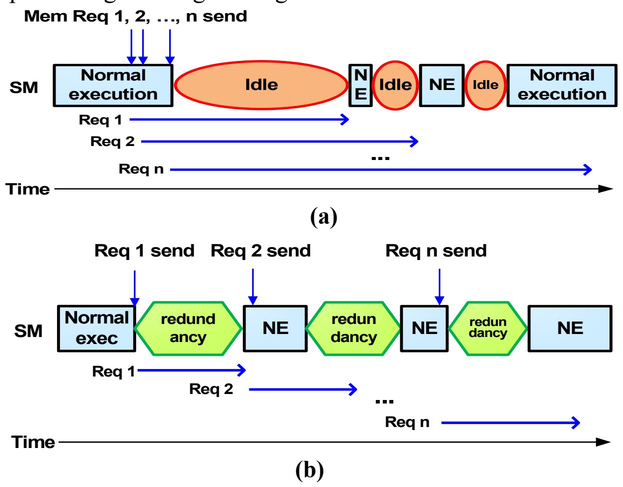


Figure 2. SPs idle time (a) caused by resource contentions among memory requests, and (b) recycled for partial Redundancy (NE: normal execution)

Figure 2 shows a case observed in a CUDA benchmark - *BP*. A large number of memory requests from the SM are routed to the memory controller in a short period. Due to the serious resource contentions, the off-chip memory access takes up to thousands of cycles. Correspondingly, the SM suffers extremely long pipeline stall, and SPs remain idle (as shown in Figure 2(a)). Instead of sending the memory requests at similar time and stalling afterwards for a long period, warps can run at slower pace which prolongs their requests issues (e.g. req2 ~ reqn in Figure 2(b)) and avoids the possible resource contentions, therefore, their requests are serviced without any delay in the input buffer. Due to the dramatically reduced memory operation latency, the total warp execution time still keeps the same or even decreases. This provides the great opportunity for temporal redundancy, whose negative effect on performance can just be positively used to postpone the warp execution progress and their requests issue time. The SPs idle time caused by the request pending in the memory controller (highlighted with red color in Figure 2(a)) is successfully recycled for the redundancy (highlighted with green color in Figure 2(b)). Note that the redundant and normal executions are interleaved at cycle level. We mark the redundancy separately from the normal execution in Figure 2(b) for the easy understanding of the impact of redundancy. By running a large number of benchmarks, we observe that the request pending contributes to 65% of the time that all SPs in the SM are free. It implies that the SPs soft-error reliability can be substantially improved with no performance penalty.

3.1.1.2. The Concept of Request Pending Aware Full-RISE

As discussed above, the partial redundancy (relative to the full redundancy) can be treated as the knob to control the warp progress. It has to be carefully tuned: the over adjustment will result in the excessive redundancy and unnecessarily delay, consequentially, degrades the performance significantly; on the other hand, the insufficient adjustment cannot effectively leverage the SPs idle time for reliability enhancement. Moreover, different warps should spend different amount of time on redundancy in order to separate their memory requests. In one sentence, the warp progress should well adapt to its memory access pattern to achieve the optimal SPs soft-error robustness. Obviously, statically determining the period of performing the partial redundancy fails to meet the goal since workloads have various memory access patterns. In this study, we propose RP-FRISE, as the abbreviation of request pending aware full-RISE, which dynamically tunes the knob (i.e. partial redundancy) per warp and recycles the SPs idle time to maximize the error coverage in SPs.

Since the major resource contentions occur in the memory controller, the request waiting time in the input buffer is a good indicator to the necessity of slowing down the warp. A long waiting time implies a serious resource contention, the warp which issues the request should have been delayed. While a short waiting time means that the request is issued appropriately, postponing the warp progress may degrade the performance. In the ideal case, a memory request gets serviced once it arrives at the memory controller: the period that allows a warp to perform the redundant execution should be equal to its request waiting time. In RP-FRISE, we use the previous request pending time to predict the delay in the following memory transaction and tune the partial redundancy in the warp.

Note that the warp progress has already been postponed somehow when finishing the previous memory access, further slowing it down as the same amount of the previous request waiting time may serious prolong the warp computation. An example is shown in Figure 3 to illustrate the challenge: warp0 and warp1 exhibit different execution progress after the first memory access, and their following memory requests have less interference. Using the previous request waiting time leads to excessive redundancy and degrades the performance. On the other hand, although warps run at different rate after the long latency memory access, the interference is still severe. Figure 4 demonstrates the case. When the second memory request is issued from warp0, it interferes with the unfinished memory access from warp1 and the waiting time increases. The extended memory access in warp0 further affects the warp1, and setting the redundancy time as the preceding request waiting time for warp1 is appropriate. Note that warp1 may not delay as long as the scheduled redundancy time even performing the full redundancy for the computation between its two memory accesses (highlighted by the pink color in Figure 4). In that case, it will be stalled after finishing the redundancy due to the interference with warp0. Therefore, the required redundancy time still accurately predicts the necessary delay in warp1. As can be seen from Figures 3 and 4, generally, the excessive redundancy occurs when the kernel is more computation-intensive and the time spent on the normal computation alleviates the memory contentions among threads. While the short normal computation in the memory-intensive kernel cannot effectively separate the memory requests, a longer redundant execution is desired.

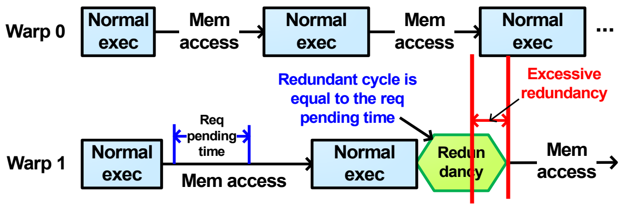


Figure 3. An Example of the Excessive Redundancy

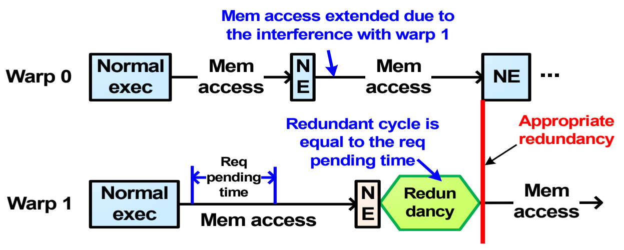


Figure 4. An Example of the Appropriate Redundancy

In our study, we sample the memory access latency periodically when running a kernel, and use it to adjust the amount of cycles that the warp executes the redundant threads. Eq.1 describes the analytical model to dynamically determine the redundancy cycles (represented by RC) in the warp based on its previous request pending time (represented by T_P) and the latest sampled memory access latency (represented by T_{acc_lat}).

$$RC = \begin{cases} 0, & \text{if } T_P \le T_{thr\_pend} \\ \left\lceil \frac{T_{acc\_lat}}{T_{ref\_lat}} \times T_P \right\rceil, & \text{if } T_{acc\_lat} < T_{ref\_lat},\ T_P > T_{thr\_pend} \\ T_P, & \text{if } T_{acc\_lat} \ge T_{ref\_lat},\ T_P > T_{thr\_pend} \end{cases} \quad \text{Eq.1}$$

where T_{thr_pend} is the threshold of the request pending time. It is possible that the previous memory access is over delayed by the redundancy, the improper delay (implied in the prior request pending time) cannot further propagate to the following execution. T_{thr_pend} plays an important role to filter it out. Only when T_P is longer than T_{thr_pend}, the warp redundancy would be enabled, otherwise, it is disabled to maintain the performance. T_{ref_lat} is the referred memory access latency describing the memory access time with moderate resource contentions. When T_{acc_lat} is higher than T_{ref_lat}, it implies that the kernel currently exhibits the memory-intensive characteristics and the aggressive redundancy (i.e. directly setting the redundancy cycles as the request pending time) should be applied. To the contrary, a lower T_{acc_lat} means that the kernel involves heavier computation, the redundancy period should be scaled down according to the ratio of T_{acc_lat} to T_{ref_lat}.

Recall that threads in a warp execute the same instruction, triggering the redundancy at the warp level becomes the first choice in RP-FRISE. However, all threads in the block have to synchronize at barriers if exist, large progress difference among warps belonging to the same block will extend the faster warps' waiting time at the barrier and hurt performance. Additionally, GPGPU programmers are encouraged to make consecutive threads access consecutive memory locations, warps in a block tend to show the strong spatial locality [1, 14]. The memory requests issued by those warps are likely to be directed to the same row in the DRAM, called row locality [10]. When they are sent out simultaneously, the pending time in the input buffer of the memory controller tends to be similar under the out-of-order memory scheduling policy. On the other hand, it will even take longer time to complete the memory transactions if issued separately, because the row locality among warps is broken and the row switches more frequently. In order to maintain the performance, RP-FRISE performs the block level redundancy. Since the redundant time is computed on the basis of memory request, a single warp may have more than one option on setting the redundancy cycles, and there are numerous choices when extending to a block. In RP-FRISE, the redundancy time will be incorporated into the response packet from the memory to the SM. To a block, the first arrived packet after it finishes the previous assigned redundant execution sets the new redundancy cycles, which should be applied to all its warps.

3.1.2. Idle SMs Aware Full-RISE

While a kernel is launched into the GPU, its blocks may not be even distributed across the SMs. A number of SMs are free when approaching the end of the kernel execution, and they have to wait for other busy SMs to finish their tasks. In other words, the SPs become idle when no more blocks can be assigned to the SM. We found that this case contributes to 35% of the time that all SPs are free in the SM. Those free SPs can be leveraged for redundant execution. One straightforward method is to redundantly execute the blocks which are currently running in other SMs. It will cause the challenges for memory synchronization between the original and redundant blocks and introduce more memory transactions from the redundant blocks.

Instead of implementing the redundancy when the SM is free, we propose to do it at the beginning of the kernel execution so that the SMs execution time is aligned and the SPs idle time is effectively recycled for redundancy. This technique is named as IS-FRISE, as the abbreviation of idle SMs aware Full-RISE. Based on the information obtained during the kernel launch process (e.g. the total number of blocks, the maximum concurrent blocks a single SM supports when running the specific kernel), a simple calculation is done to roughly estimate the total number of blocks assigned to each SM through the entire kernel execution ignoring the possible memory access delay. We conservatively assume that there is only one block difference among SMs, and divide the block quantity by the number of SMs. If there is a remainder, we expect that some SMs may be free at the end of the kernel execution, and the remainder determines the quantity of SMs (which are randomly selected among all the SMs) running an additional block. Once the kernel is launched, the full redundancy is applied to one of the currently executed blocks in SMs which are predicted to run fewer blocks, thus, the total loads including the redundancy are balanced across SMs and performance remains the same. Note that the load imbalance among SMs can be larger than one block since faster SMs will take more blocks, but predicting a large load difference is likely to cause the overestimate of the SMs idle time which leads to the aggressive redundancy and hurt the performance.

When integrating the two full-RISE techniques simultaneously, some blocks may perform the redundancy twice which is a waste of resources. Considering that IS-FRISE only takes in effect on a small set of blocks, the redundant execution scheduled by RP-FRISE on those blocks will be ignored. Although there is an overlapped effect between the two techniques, their positive interaction minimizes the potential of excessive redundancy in RP-FRISE. Recall that RP-FRISE depends on the previous request pending time to determine the redundancy cycles for the following execution, it loses the opportunities to perform redundant execution before the blocks issue their first memory requests. The memory contentions among the first memory transactions tend to be severe. Using the first request's pending time for redundancy cycles calculation would over delay the block progress, and this negative effect is likely to propagate towards the end of the kernel execution. IS-FRISE triggers the full redundancy on certain blocks at the very beginning of the kernel execution, it differentiates the block progress across SMs and mitigates the resource contentions even before the first memory requests, hence, effectively reduces the possibility of the unnecessary redundancy.

3.1.3. The Implementation of Full-RISE

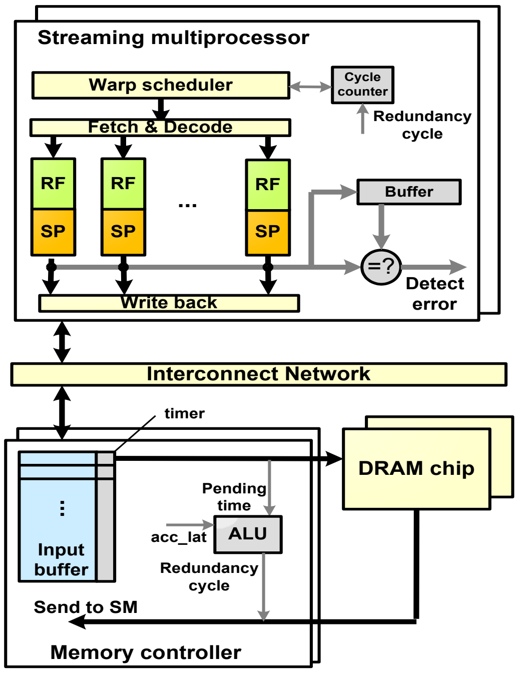


Figure 5. The Implementation of Request Pending aware Full-RISE

Figure 5 shows the implementation of the request pending aware full-RISE in the GPGPU architecture. In the memory controller, a timer is attached to each input buffer entry, and an arithmetic logic unit (ALU) is added to calculate the redundancy cycles (in our study, we assume that only one request is serviced at a time, and one ALU is sufficient to perform the calculation). When a memory request is written into the input buffer, the timer is set as zero and automatically increments every cycle, it records the request pending cycles which will be sent to the ALU when the request is issued out for DRAM access. The sampled memory access latency, threshold pending time and referred access latency are used as the inputs to the ALU as well. Its output is combined with the response packet and sent back to the SM. Considering that ALU operation has to be done per memory request, and the major computation in it is the division (as shown in Eq.1.) which lasts for tens of cycles, we set the referred access latency as 2 to power of n and translate the division into logical shift. It will operate based on the product of the average access latency and the pending time. We performed the detailed sensitivity analysis on the referred access latency, and found that RP-FRISE achieves optimal trade-off between reliability and performance when setting it as 2^7=128 cycles. Note that the redundancy time computation occurs in parallel with the data access in DRAM, it does not introduce any extra delay to the critical path in the memory controller.

As Figure 5 shows, each block in the SM is allocated a cycle counter which keeps the redundancy cycles. The amount of counters per SM is determined by the maximum number of concurrent blocks an SM supports (e.g. 8 in our default configuration). When a response packet arrives at the SM, we can find out its corresponding cycle counter based on the warp it returns to. If the counter is larger than zero, it implies that the block is already under the redundancy mode, the new redundancy time (in cycles) will be ignored. Otherwise, it is multiplied by the number of warps in the block, and the result will be written into the counter, which implies the desired total amount of redundancy cycles applied to all warps in the block. In this study, instead of controlling each warp in a block to spend the same amount of time in redundancy mode, we apply a relaxed mechanism to manage the total redundancy cycles at the block level. When a ready warp is selected to feed the pipeline, the counter is accessed, a larger-than-zero value suggests a redundant execution. The warp will remain in the warp scheduler and be granted a high issue priority to ensure that the same redundant warp is executed in the following cycle. Meanwhile, the counter decreases by one. During the write back stage, the SPs outputs of the original warp are written into the destination registers and an attached buffer; while the redundant warp's outputs will skip the write operation, and directly compare with the data just written into the buffer for the error detection. A mis-match will raise the recovery signal. The warp PC will be fetched again to start additional redundant execution, three copies' comparison will correct the error and resume the warp computation.

The idle SMs aware full-RISE requires modulo operation to determine the number of SMs running fewer blocks. A simple AND logic operation can compute out the remainder. It performs simultaneously with the kernel launch process, and no extra delay is introduced to the normal kernel execution. When combining the two full-RISE mechanisms, the block with full redundancy selected by the IS-FRISE will set its counter as one and disable the value decrease function until it finishes, ensuring the full redundant execution for its warps.

As can be seen in Figure 5, the major hardware added into the memory controller includes the unit performing simple integer arithmetic and logic operations, and a number of 10-bit timers (we set the maximum request pending time as 1024 cycles), it causes around 3% area overhead to the memory controller. In the SM, the added hardware contains 8 cycle counters, thirty-two 32-bit buffers for warp outputs (the SM pipeline width is 32 in the default configuration, each SP output 32-bit value), and some combinational logics, totally resulting in 1% area overhead to the SM.

3.2 Partial-RISE

3.2.1. The Concept of Partial-RISE

When the warp diverges at the branch instruction, several SPs in the SM are idle. In the workload encountering frequent branch divergences, SPs are partially free in majority of the time. For instance, the case occurs during 52% of the total kernel execution time in a CUDA benchmark - *HS*. Unfortunately, full-RISE fails to leverage such large portion of SPs idle time for reliability optimization due to its nature of performing the redundancy via using all SPs. In this subsection, we propose partial-RISE which intelligently combines the redundant threads into the diverged warp to utilize the partially idled SPs, thus, improves the SPs error coverage and maintains the performance.

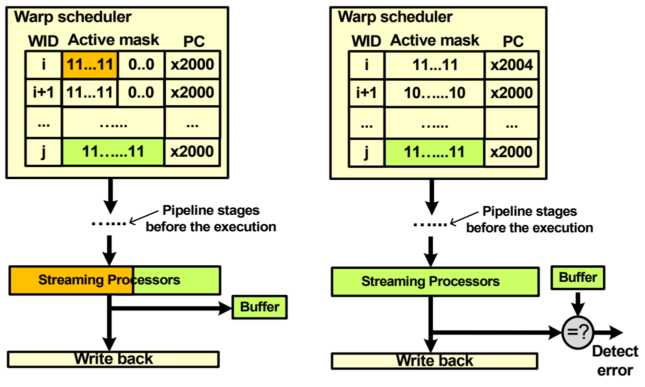


Figure 6. An Example of Partial-RISE

As described in Section 3.1, GPGPU has the unique microarchitecture characteristic (e.g. warps interleave at cycle level, and all threads execute the same code), and typically, there are numerous warps in the SM warp scheduler. When a ready warp is issued into the pipeline, it is highly possible that another ready warp sharing the same PC is sitting in the scheduler. Similarly, the diverged ready warp can usually find another ready warp (not necessarily diverges), and both are going to perform the same operation in SPs. A number of threads from a warp with the same PC can join the execution of the diverged warp. Therefore, the warp is partially protected as the idle SPs in the diverged warp are effectively utilized to execute the redundant threads for it. Moreover, the warp will be issued in the following cycle so that both main and redundant threads output can be compared immediately for the error detection. We name this technique as partial-RISE. Figure 6 demonstrates an example of it. In Figure 6(a), at cycle *m*, the warp *i* and *j* have the same PC, and the active mask shows that warp *i* diverges. When it is issued into the pipeline, threads from warp *j* will take the free slots in warp *i* based on its active mask. When the outputs of warp *j* are available, they will be sent to the buffer instead of writing to the registers. During the cycle *m*+1(shown in Figure 6(b)), warp *j* is issued, it outputs are compared with the data saved in the buffer in previous cycle.

Finding the warp with the same PC is critical to partial-RISE. We investigate various benchmarks, and Figure 7 plots the ratio of the number of cycles that at least one ready warp has the identical PC with the currently issued diverged warp to the total warp divergence cycles (shown as the red bars). The round-robin policy is applied for the warp scheduling. The benchmarks with quite few branch divergences are not shown in the figure since the partial-RISE is rarely triggered in that case. As it demonstrates, in more than half of the time, the diverged warp has the opportunity to combine with another warp by searching across the warp scheduler. However, the two warps are likely to use the SP and registers belonging to the same lane and encounter the lane conflict. It becomes the major obstacle to partial-RISE. As shown in Figure 6(a), although warp *i*+1 has the identical PC with warp *i*, partial-RISE cannot be used as both them have the same active mask. Figure 7 also shows the possibility that a diverged warp can be combined with another warp without lane conflict (shown in the yellow bars). As can be seen, it decreases significantly down to 16%, and even becomes zero in some benchmarks (e.g. *BP*, *SLA*, and *ST3D*). Since threads in a warp are independent but their operations are the same, there is no requirement to bind a thread to a certain lane. We propose to randomly shuffle the threads while sending them to the SMs, it will be performed in parallel with the kernel launch process and no extra delay to the entire kernel execution. By doing that, the possibility of lane conflicts between two warps decreases and partial-RISE can be triggered more frequently. The black bar in Figure 7 shows that re-arranging threads successfully brings the possibility of finding a ready warp with the same PC back to 48%.

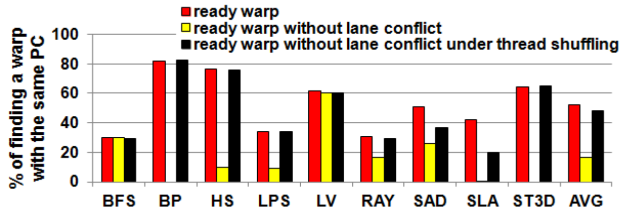


Figure 7. Possibility that a Diverged Warp Finding a Warp with Identical PC

Recently, several mechanisms (e.g. the dynamic warp formation [15], thread block compaction [16], the large warp microarchitecture (LWM) [10]) have been proposed to improve the efficiency of branch handling. Take the LWM as an example, it implements larger warp so that the sub-warps can be formed from the active threads in a large warp, and when they are executed in the pipeline, the SPs idle time reduces under branch divergences. One possible concern is that partial-RISE has trivial benefit for reliability enhancement when LWM is applied in the GPU. This is not the case. A large warp contains much smaller number of threads (e.g. 256) compared to the entire warp scheduler. It is unable to find active threads out of the warp, and it does not provide any mechanism to avoid the lane conflicts. LWM does not always fully utilize the idled SPs during branch divergences. While partial-RISE searches warps across the entire scheduler, and is equipped with the thread shuffling technique, it can efficiently use those idled SPs in LWM for redundant execution. We observe that partial-RISE takes in effect in 20% of the time when LWM fails to fully utilize the SPs.

In some benchmarks (e.g. *NN*, *NW*), the block contains few threads (e.g. 16) so that it has only one warp and all threads in that warp cannot fill up the SIMT pipeline width. Since several lanes keep idle through the entire kernel execution, partial-RISE will perform the intra-warp duplication which leverages those idle lanes to provide the spatial redundancy for some threads contained in the warp.

3.2.2. The Implementation of Partial-RISE

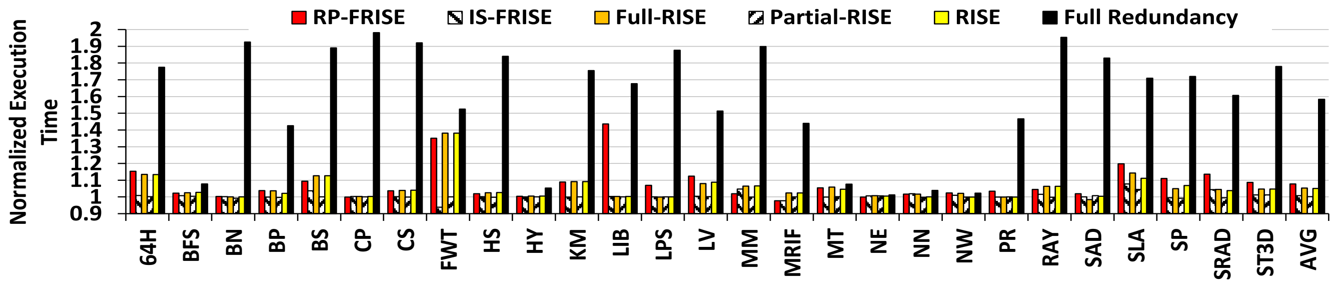
In the SM pipeline, when the warp enters to the final pipeline stage (i.e. write back stage), its PC and active mask are updated, and its status turns to be ready for issue in the following cycle. To implement the partial-RISE technique, a comparator is attached to each warp entry in the warp scheduler. While updating the warp status, its PC (active mask) will be compared with other ready warp PCs (active masks) in the scheduler to seek a warp for joined execution. The comparison is executed along with the write back stage, and no impact to the critical path delay. If succeeds, the scheduler will be notified to issue the diverged warp and threads from its matched warp simultaneously in the next cycle, followed by the normal issue of the matched warp. Partial-RISE will re-use the hardware (e.g. buffer, comparator) in Full-RISE to perform the error detection. In total, partial-RISE leads to additional 1% area overhead to the SM equipped with full-RISE.

3.3 RISE: Putting It All Together

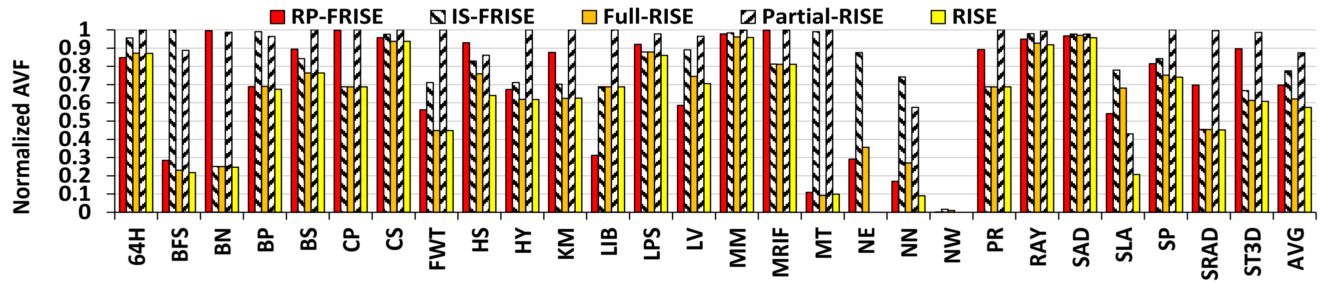
Since Full-RISE and Partial-RISE target to recycle the SPs idle time caused by two different cases for reliability improvement, they can be integrated into RISE. While implementing RISE, the warp under the redundancy mode due to the full-RISE will not be considered in partial-RISE. Although full-RISE differentiates the block execution progress, it does not degrade the efficiency of partial-RISE in finding appropriate warp for redundancy. We find that the partial-RISE trigger time even increases by 2% when combined with full-RISE. It is because that the row locality leads to the similar redundancy cycles, and consequently, similar progress among blocks in the same SM under full-RISE, and the block progress difference generally happens at the SM level.

4. Experimental Setup

We use architecture vulnerability factor (AVF) [17] to evaluate the error coverage of our proposed RISE. A hardware structure's AVF refers to the probability that a transient fault in that hardware structure will result in incorrect program results. Therefore, the AVF, which can be used as a metric to estimate how vulnerable the hardware is to soft errors during program execution, is determined by the processor state bits required for architecturally correct execution (ACE). The structure's AVF in a given cycle is the percentage of ACE bits that the structure holds, and its overall AVF during program execution is the average AVF at any point in time. We apply the methodology proposed by Mukherjee et al. [17] to identify the ACE bits and their residency time in the structure and compute the AVF of GPGPU microarchitecture structures. We build our vulnerability estimation framework based on the



(a) Normalized Execution Time



(b) Normalized AVF of Streaming Processors

Figure 8. The normalized (a) Execution Time and (b) SPs' AVF under IS-FRISE, RP-FRISE, Full-RISE, Partial-RISE, RISE, and Full Redundancy techniques

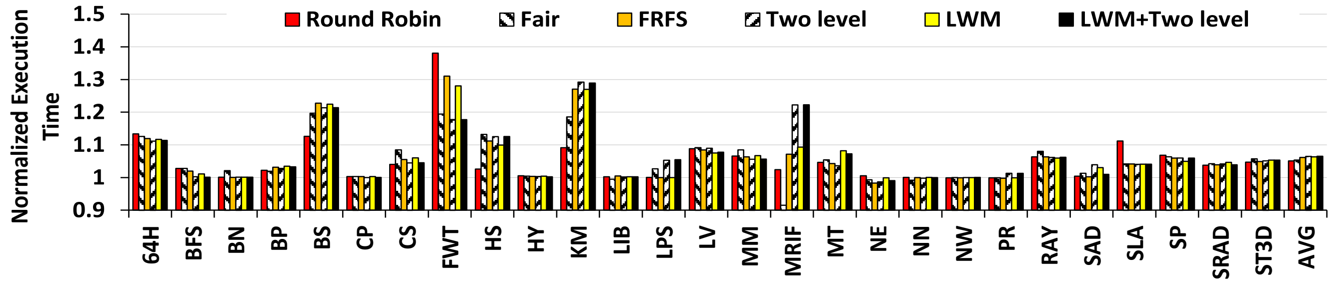
cycle-accurate, open-source, and publicly available simulator GPGPU-Sim [18], and obtain the GPGPU reliability and performance statistics. Our baseline GPGPU configuration is set as follows: there are 28 SMs in the GPU, SM pipeline width is 32, warp size is 32, each SM supports 1024 threads and 8 blocks at most, each SM contains 16K 32-bit registers, 16KB shared memory, 8KB constant cache, and 64KB texture cache, the warp scheduler applies the round robin scheduling policy, the immediate post-dominator reconvergence [19] is used to handle the branch divergences; the GPU includes 8 DRAM controllers, each controller has a 32-entry input buffer, and applies out-of-order first-ready first-come first-serve scheduling policy; the interconnect topologies is Mesh, and the dimension order routing algorithm is used in the interconnect. We collect a large set of available GPGPU workloads from Nvidia CUDA SDK [20], Rodinia Benchmark [21], Parboil Benchmark [4] and some third party applications. The workloads show significant diversity according to their kernel characteristics, divergence characteristics, memory access patterns, and so on.

5. Evaluation

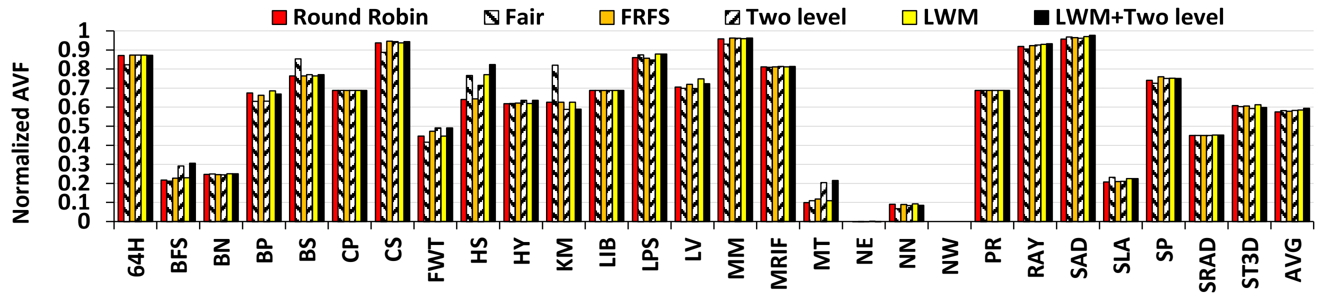
5.1 Effectiveness of RISE

To better analyze the technique effectiveness, we classify the benchmarks into four categories based on their workload characteristics. The first category includes memory-intensive benchmarks such as *64H, BFS, BP, HY, LIB, LV, MT, NE, NN, NW, PR,* and *SLA*. The second category contains benchmarks which cause load imbalance across SMs in our baseline GPGPU configuration, they are *BN, CP, FWT, HY, KM, LV, MRIF, NW, PR, SLA, SP, SRAD,* and *ST3D*. The third category includes benchmarks usually utilizing partial SPs (caused by the frequent branch divergences or the partially full warps) such as *BFS, BP, HS, LPS, LV, NN, NE, NW, SAD, SLA,* and *ST3D*. The last category includes the computation-intensive benchmarks such as *BS, CS, MM,* and *RAY*. Note that the categories described above are not exclusive (i.e. one benchmark can be classified into different categories).

Figure 8 shows the (a) execution time and (b) the AVF of streaming processors when running various benchmarks under the impact of IS-FRISE, RP-FRISE, Full-RISE, Partial-RISE, and RISE, respectively. The execution time of full redundancy is demonstrated in Figure 8(a) as well for comparison. As can be seen, on average, it results in 58% performance degradation. Since the full redundancy achieves 100% error coverage (i.e. AVF is zero), its results are not shown in Figure 8(b). We present the averaged results across SMs, and the results are normalized to the baseline case without any optimization. As Figure 8 shows, RP-FRISE exhibits strong capability in improving the SPs soft-error vulnerability with little performance loss when executing the memory-intensive benchmarks (classified as the first category). Take the *MT* as an example, the SPs' AVF decreases by 90% with only 4% performance penalty. It implies that the redundancy cycles are properly set by RP-FRISE and the SPs idle time in the baseline cases is effectively recycled for redundant execution. One may notice that the AVF reduction under RP-FRISE is less obvious in *PR*, although the warps spend 17% of the execution time in waiting for the memory transactions. It is because their memory requests have already well separated during the execution, the memory access latency is generally short, and the pipeline only stalls couple of cycles for the memory transaction. Postponing the warp progress would easily hurt the performance substantially, therefore, RP-FRISE is seldom triggered. While improving the SPs' AVF, RP-FRISE degrades performance in some benchmarks such as *FWT, LIB,* and *SLA*. Because RP-FRISE uses the last request pending time to predict the next request waiting time and determine the redundancy cycles correspondingly, its prediction accuracy is affected when the next memory access pattern differs greatly from the last one. As a result, the excessive redundancy is applied which hurts the performance. On average across the



(a) Normalized Execution Time



(b) Normalized AVF of Streaming Processors

Figure 9. The normalized (a) Execution Time and (b) SPs' AVF under Various Scheduling Policies and Performance Optimizations

benchmarks in the first category, RP-FRISE enhances the SPs soft-error robustness by 57% with 8% performance loss.

The impact of IS-FRISE on improving the SPs' error coverage is impressive in benchmarks belonging to the second category. For example, the SPs free time caused by the imbalanced block distribution is completely recycled for redundancy in *BN* and the AVF decreases 75% with 0.1% performance loss. Recall that IS-FRISE conservatively assumes one block difference among SMs to maintain the kernel execution time, while the difference is larger in some benchmarks (e.g. *LV*). In the GPU, a block is assigned once there is an empty slot in certain SM. It is possible that an SM commits multiple blocks at similar time and the remaining unexecuted blocks are all allocated to it. Although other SMs finish the block execution in the very near future, they have to remain idle till the kernel completes. In that case, IS-FRISE does not fully leverage the idle SPs to perform redundancy. Monitoring and predicting the block progress in each SM and dynamically controlling the number of redundant blocks may lead to better reliability improvement, but it induces complicated hardware design which should be avoided. On average across benchmarks in the second category, IS-FRISE reduces SPs' AVF 37% with no performance loss. Note that the load imbalance of a benchmark is related to the GPU configuration (e.g. number of SMs), it may not be an issue when configuring the machine differently, but meanwhile, another set of benchmarks may encounter this problem. Therefore, IS-FRISE is applicable to various GPU architecture designs. As the combination of RP-FRISE and IS-FRISE, Full-RISE keeps their positive effects on optimizing the soft-error robustness substantially, and more important, their interaction effectively mitigates the performance penalty caused by the RP-FRISE. As Figure 8 shows, Full-RISE improves SPs AVF 39% with 4% performance loss on average across all investigated benchmarks.

The major benefit of Partial-RISE is observed in benchmarks classified into the third category. As shown in Figure 8, on average across benchmarks in the third category, it reduces SPs AVF by 32% without any performance penalty. Finally, when putting all together, RISE integrates the benefit of all the proposed techniques by achieving 43% SPs soft-error reliability enhancement and minimizes (only 4%) the performance loss. In the computation-intensive benchmarks, the effect of RISE is less impressive due to the limited SPs idle time. Note that, RISE optimizes the SPs vulnerability via redundancy, the SPs AVF does not migrate to any other microarchitecture structures.

5.2 Sensitivity Analysis

Various techniques have been proposed on warp scheduling and warp formation to improve the GPGPU throughput, such as Fair which issues the instruction for the warp with minimum number of instructions executed [22], First-Ready First-Served (FRFS), large warp microarchitecture (LWM), and two-level round-robin warp scheduling that effectively hides long memory access latency and improves the SPs utilization [10]. Figure 9 shows (a) the execution time and (b) SPs' AVF under RISE when those optimization schemes are enabled. Results are normalized to the baseline case with the corresponding optimizations, respectively. The results obtained when using the default scheduling policy (i.e. Round Robin) is also shown in the figure for comparisons. As it shows, the effectiveness of RISE is not affected when running with different schemes: on average, the SPs AVF reduction keeps around 42% with 6% performance loss. Take the scheme of LWM+two_level as an example, it reduces the SPs idle time to some degree to shrink the kernel execution time, one would expect that it largely diminishes the opportunities to trigger RISE. However, this only occurs in a limited number of benchmarks (i.e. *BFS, HS, MT*) which shrinks the reliability optimization by around 12%. As we described in Section 3.2.1., LWM only finds active threads in the warp scope and cannot avoid the lane conflicts, it leaves sufficient room for Partial-RISE in RISE to further use the partially idled SPs for redundant execution. Moreover, the two-level round robin scheduling cannot totally avoid the memory contentions and the load imbalance across SMs, therefore, the Full-RISE in RISE can be frequently triggered to recycle the SPs free time for reliability enhancement.

6. Related Work

There have been various studies on protecting CPUs via software/hardware-based duplication. For instance, Qureshi et al. [34] appropriately trigger redundant threads to minimize performance loss and provide full soft error coverage for the entire pipeline. Slick [23] avoids the redundancy for results predictable instructions to improve the performance when running redundant multithreading. Feng et al. [24] leverage the symptom based detection and selective instruction duplication to minimize user-visible failures induced by soft errors. Sun et al. [25] combine the C-element based error detection techniques with idle resources exploited in functional units at the circuit level to improve their fault tolerance capability. However, they mainly target on CPUs and largely ignore the GPGPU architecture.

Soft-errors on GPU have been investigated in [26 ~ 28], and recovered via redundancy and checkpointing [9, 29 ~ 32]. For example, Sheaffer et al. [30] explore the concept of the sphere of replication on GPGPU processors, and present a hardware redundancy-based approach to create a reliable GPU with no performance loss. Dimitrov [31] investigate three software approaches to perform redundant execution for GPGPU reliability. Maruyama et al. [9] propose a high-performance software framework to enhance GPU with DRAM fault tolerance. It leverages light-weight data coding for error detection and checkpointing for recovery. Solano-Quinde et al. [32] propose an application-level checkpoint scheme for GPGPU systems, and explore the computation-communication overlapping to reduce the checkpoint overhead. In our study, we exploit the SPs idle time and recycle it to perform the redundancy for error detection to maximize the SPs reliability with little performance loss. Recently, Nathan et al. [33] develop Argus-G, it implements control flow, dataflow and computation checkers in the SPs for low cost error detection. Yim et al. [8] strategically places customized error detectors in the source code of GPU applications to tolerate errors. Both the two schemes are orthogonal to our proposed techniques.

7. Conclusions

With their strong computing power and improved programmability, GPGPUs emerge as highly-efficient devices for a wide range of parallel applications. Meanwhile, GPGPU with hundreds of cores integrated in a single chip are highly vulnerable to the soft error strikes. Especially, streaming processors play an important role to exploit the data parallelism and are becoming the vulnerability hot-spots in GPUs. This work explores the first essential step to optimize the SPs soft-error robustness. We propose RISE to effectively recycle the SPs idle time for soft-error detection. RISE is composed of full-RISE and partial-RISE. Full-RISE exploits the fully idled SPs caused by the long-latency memory transactions and imbalanced load assignment among GPU cores, and uses them to perform the redundant execution and enhance the SPs reliability. Partial-RISE combines the redundant execution of a number of threads from a warp with a diverged warp to recycle the SPs idle time during the branch divergence for their reliability optimization. Experiment results show that RISE reduces the SPs AVF by 43% with only 4% performance loss. Our sensitivity analysis also shows that RISE is applicable to GPUs with various performance optimization mechanisms.

Acknowledgement

This work is supported by the National Science Foundation under Award No. EPS - 0903806 and matching support from the State of Kansas through the Kansas Board of Regents.

References

[1] NVIDIA. CUDA Programming Guide Version 3.0., Nvidia Corporation, 2010.

[2] Advanced Micro Devices, Inc. AMD Brook+. http://ati.amd.com/technology/streamcomputing/AMD-Brookplus.pdf.

[3] Khronos. Opencl – the open standard for parallel programming of heterogeneous systems. http://www.khronos.org/opencl/

[4] Parboil Benchmark suite. URL: http://impact.crhc.illinois.edu/ parboil.php.

[5] N. Wang and S. Patel, ReStore: Symptom Based Soft Error Detection in Microprocessors, In Proceedings of DSN, 2005.

[6] C. Weaver, J. Emer, S. Mukherjee, and S. Reinhardt, Techniques to reduce the soft error rate of a high-performance microprocessor, In Proceedings of ISCA, 2004.

[7] N. Maruyama, A. Nukada, and S. Matsuoka, A High Performance Fault-Tolerant Software Framework for Memory on Commodity GPUs, In Proceedings of IPDPS, 2010

[8] K. Yim and R. Iyer. Hauberk: Lightweight silent data corruption error detectors for gpgpu. In Proceedings of the 17th Humantech Thesis Prize (Also in IPDPS 2011), 2011.

[9] S.K. Reinhardt, and S.S. Mukherjee, Transient Fault Detection via Simultaneous Multithreading, In Proceedings of ISCA, 2000.

[10] V. Narasiman, C. J. Lee, M. Shebanow, R. Miftakhutdinov, O. Mutlu, and Y. N. Patt. Improving GPU Performance via Large Warps and Two-Level Warp Scheduling. In Proceedings of MICRO, 2011.

[11] Nvdia's Next Generation CUDA™ Compute Architecture: Fermi™, http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf

[12] X Yang, et. al., TH-1: China's first petaflop supercomputer, Frontiers of Computer Science in China, 2010.

[13] M. A. Gomaa and T. N. Vijaykumar, Opportunistic Transient-Fault Detection, In Proceedings of ISCA, 2005.

[14] D. Kirk and W. W. Hwu. Programming Massively Parallel Processors: A Hands-on Approach. Elsevier Science, 2010.

[15] W. W. L. Fung, I. Sham, G. Yuan, and T. M. Aamodt, Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow, In Proceedings of MICRO, 2007.

[16] W. W. L. Fung and T. Aamodt. Thread Block Compaction for Efficient SIMT Control Flow. In Proceedings of HPCA, 2011.

[17] 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, In Proceedings of MICRO, 2003.

[18] A. Bakhoda, G.L. Yuan, W. W. L. Fung, H. Wong, Tor M. Aamodt, Analyzing CUDA Workloads Using a Detailed GPU Simulator, In Proceedings of ISPASS, 2009.

[19] S. S.Muchnick. Advanced Compiler Design and Implementation. Morgan Kaufmanns, 1997.

[20] http://www.nvidia.com/object/cuda_sdks.html

[21] S. Che, M. Boyer, J. Meng, D. Tarjan, J. Sheaffer, S. Lee, and K. Skadron. Rodinia: A Benchmark Suite for Heterogeneous Computing, In Proceedings of IISWC, 2009.

[22] N. B. Lakshminarayana, H. Kim, Effect of Instruction Fetch and Memory Scheduling on GPU Performance, Workshop on Language, Compiler, and Architecture Support for GPGPU, 2010.

[23] A. Parashar, A. Sivasubramaniam, S. Curumurthi, SlicK: Slice-based Locality Exploitation for Efficient Redundant Multithreading, In Proceedings of ASPLOS, 2006.

[24] S. Feng, S. Gupta, A. Ansari, and S. Mahlke, Shoestring: probabilistic soft error reliability on the cheap, In Proceedings of ASPLOS, 2010.

[25] Y. Sun, S. Li, M. Zhang, C. Song, Exploiting Idle Resources for Reducing SER of Microprocessor Functional Units, In Proceedings of ICCET, 2010.

[26] I. Haque and V. Pande. Hard Data on Soft Errors: A Large-Scale Assessment of Real-World Error Rates in GPGPU. In Proceedings of CCGrid, 2010.

[27] G. Shi, J. Enos, M. Showerman, and V. Kindratenko. On Testing GPU Memory for Hard and Soft Errors. In Proceedings of SAAHPC, 2009.

[28] J. Tan, N. Goswami, T. Li, and X. Fu, Analyzing Soft-Error Vulnerability on GPGPU Microarchitecture, In Proceedings of IISWC, 2011.

[29] N. Maruyama, A. Nukada, and S. Matsuoka. Software-Based ECC for GPUs. In Proceedings of SAAHPC, 2009.

[30] J. Sheaffer, D. Luebke, and K. Skadron, A Hardware Redundancy and Recovery Mechanism for Reliable Scientific Computation on Graphics Processors, In Proceedings of Graphics Hardware 2007.

[31] M. Dimitrov, M. Mantor, and H. Zhou, Understanding Software Approaches for GPGPU Reliability, The 2nd workshop on General-Purpose Computation on Graphics Processing Units (GPGPU-2), 2009.

[32] L. Solano-Quinde, B. Bode, and A. Somani, Coarse Grain Computation-Communication Overlap for Efficient Application-Level Checkpointing for GPUs, In Proceedings of International Conference on Electro/Information Technology (EIT), 2010.

[33] R. Nathan, D. J. Sorin, Argus-G: A Low-Cost Error Detection Scheme for GPGPUs, Workshop on Resilient Architectures (WRA), 2010.

[34] M. K. Qureshi, O. Mutlu, Y. N. Patt, Microarchitecture-Based Introspection: A Technique for Transient-Fault Tolerance in Microprocessors, In Proceedings of DSN, 2005.

[35] S.S.Mukherjee, J. Emer, and S.K.Reinhardt, The Soft Error Problem: An Architectural Perspective, In Proceedings of HPCA, 2005.