CHAPTER

Addressing hardware reliability challenges in general-purpose GPUs 23

J. Tan, X. Fu
University of Houston, Houston, TX, United States

1 INTRODUCTION

Modern graphics processing units (GPUs) support thousands of concurrent threads and provide remarkably higher computational throughput than CPUs on parallel computing, which is key for the large-scale high-performance computing (HPC) in science and engineering. In addition, the programming models (e.g., NVIDIA CUDA [1], AMD Stream [2], OpenCL [3]) significantly reduce the development effort to parallelize general-purpose applications on GPUs. With these increasing computing power and improved programmability, general-purpose computing on GPUs (GPGPUs) emerge as a highly attractive platform for a wide range of HPC applications exhibiting strong data-level or thread-level parallelism. For example, as the second most powerful supercomputer in the world, Oak Ridge National Laboratory's Titan uses GPUs in addition to the traditional CPUs [4].

This extensive usage of GPGPU makes reliability a critical concern. Traditionally, the dominant workloads on GPUs are graphic processing applications that can effectively mask errors and have relaxed request on computation correctness. For example, Sheaffer et al. [5] observed that errors in video applications are usually masked since they impact only a few pixels, or the corrupted image is quickly recomputed in the next frame. Therefore 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 HPC application computing correlation function [6], 1% of value errors in any of the program output elements is treated as a silent data corruption (SDC) error and cannot be tolerated. This makes reliability a growing concern in GPGPU architecture design. Even worse, the shrinking of feature sizes allows the manufacture of billions of transistors on a single GPGPU processor chip that concurrently runs thousands of threads. This further emphasizes the need for characterizing and addressing reliability in GPGPU architecture design.

There are two paramount hardware reliability challenges in GPGPUs, particle strikes induced soft errors and manufacturing process variations (PVs).

(i) Soft errors, also called transient faults, 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. They may silently corrupt the data and lead to erroneous computation results. Soft-error rate (SER) has been predicted to increase exponentially because of the shrinking of feature sizes and growing integration density [7,8]. GPGPUs with hundreds of cores integrated in a single chip are prone to suffer severe soft-error attacks [9–15]. For example, 8 soft errors were observed in a 72-h testing program on 60 NVIDIA GeForce 8800GTS 512 [10]; and transient faults were found in two-thirds of evaluated commodity GPUs [9]. It was also found that the SDC ratio in commodity GPUs is 16–33% [16], while the ratio is smaller than 2.3% in CPUs that have comparatively developed fault tolerance techniques. The increasing SER becomes the major obstacle to current and future GPGPUs by either preventing them from scaling down to smaller feature sizes or resulting in imprecise operations from these systems.

(ii) PV is the divergence of device parameters from their nominal values, which is caused by the challenging manufacture process at very small feature technologies. PV induces substantial variability in circuit path delay and causes timing errors [17–19]. This impact is further exacerbated in GPGPUs that contain tremendous amounts of parallel critical paths [20,21]. The frequency ratio of the fastest to the slowest computing core in GPGPUs increases up to 1.4 under the impact of PV [20].

In this chapter, we mainly target at the preceding two types of reliability issues. We explore a set of mechanisms to leverage the unique GPGPU architectural features to effectively optimize the GPGPUs' reliability.

This chapter is organized as follows: Section 2 provides the background on the state-of-the-art GPGPUs; Section 3 models and characterizes GPGPU reliability in the presence of soft errors; Section 4 improves the soft-error robustness of GPGPUs; and Section 5 discusses mitigating the susceptibility of GPUs to PVs.

2 GPGPUs ARCHITECTURE

Fig. 1A shows an overview of the state-of-the-art GPUs [1]. They consist of a scalable number of in-order streaming multiprocessors (SM) that can access multiple memory controllers via an on-chip interconnection network. The GPU device has its own off-chip external memory (e.g., global memory) connected to the on-chip memory controllers. Fig. 1B illustrates a zoom-in view of the SM [22]. It contains

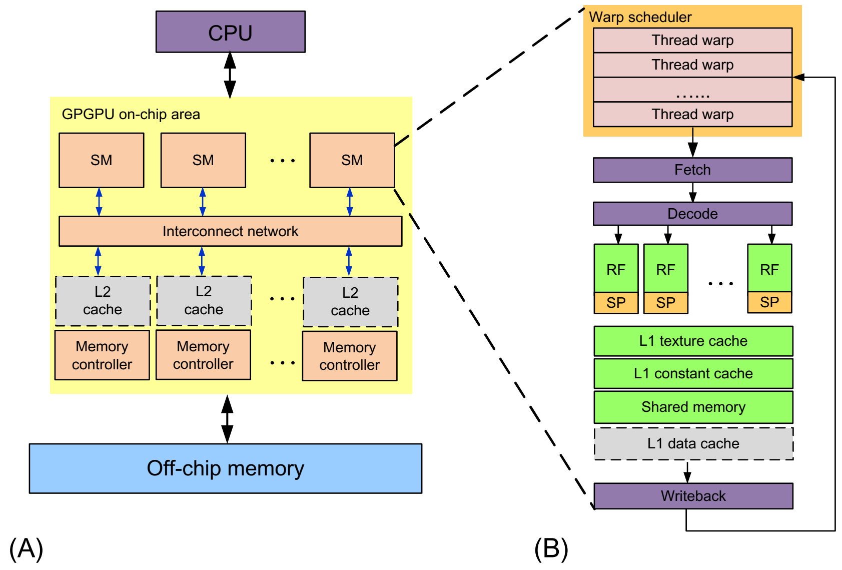


FIG. 1

General-purpose computing on graphic processing units (GPGPU) architecture. (A) An overview. (B) SM microarchitecture.

the warp scheduler, register files (RF), streaming processors (SP), constant cache, texture cache, shared memory, and so on.

To facilitate GPGPUs' application development, several programming models have been developed (e.g., NVIDIA CUDA [1], AMD Stream [2], OpenCL [3]). In this chapter, we take the NVIDIA CUDA programming model as an example; the basic concepts will hold for most programming models. In CUDA, the GPU is treated as a coprocessor that executes highly parallel kernel functions launched by the CPU. The kernel is composed of a grid of lightweight threads; a grid is divided into a set of blocks (referred to as cooperative thread arrays [CTA] 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 shared memory and synchronize at a barrier if needed. Per-block resources, such as registers files, 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 concurrently if there are sufficient per-block resources.

Threads in the SM execute in the SIMD fashion. A number of individual threads (e.g., 32 threads) from the same block are grouped together, which is called a warp. In the pipeline, threads within a warp execute the same instruction but with different data values. Fig. 1B also presents the details of SM microarchitecture. Each SM interleaves multiple warps (e.g., 32) on a cycle-by-cycle basis; the warp scheduler holds those warps, and at every cycle, it selects a warp with a ready instruction (i.e., the same instruction from all the threads within the warp are ready to execute) to feed the pipeline. The execution of a branch instruction in the warp may cause warp divergence when some threads jump while others fall through at the branch. Threads in a diverged warp have to execute in serial fashion, which greatly degrades the performance. Immediate postdominator reconvergence [23] has been widely used to handle the warp divergence. Recently, several mechanisms, such as dynamic warp formation (DWF) [24] and thread block compaction [22], have been applied to further improve the efficiency of branch handling. Because of the SIMD lock-step execution mechanism, a long-latency off-chip memory access from one thread can 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. Memory coalescing improves performance by reducing the requests for memory access.

3 MODELING AND CHARACTERIZING GPGPUs RELIABILITY IN THE PRESENCE OF SOFT ERRORS [25]

3.1 MICROARCHITECTURE-LEVEL SOFT-ERROR VULNERABILITY ANALYSIS

A key observation of soft-error behavior at a microarchitecture level is that a transient fault may not affect processor states required for correct program execution. At a microarchitecture level, the overall hardware structure's SER is decided by two factors [26]: the FIT rate (failures in time, which is the raw SER at circuit level) per bit, mainly determined by circuit design and processing technology; and the architecture vulnerability factor (AVF) [27]. 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's state bits required for architecturally correct execution (ACE). At the instruction-level, an instruction is defined as ACE (un-ACE) instruction if its computation result affects (or does not affect) the program final output, and AVF is primarily determined by the quantity of ACE instructions per cycle and their residency time within the structure [27]. In this chapter, we use AVF as the major metric to estimate structure soft-error vulnerability.

3.2 SOFT-ERROR VULNERABILITY MODELING IN GPGPU MICROARCHITECTURE

In order to characterize and optimize the soft-error vulnerability of emerging GPGPU architecture, a tool to estimate the impact of soft errors on GPGPUs is highly desired. Recently, a Sim-SODA [28] was developed to compute AVF of CPU microarchitecture structures. Compared to the general-purpose CPU cores, GPGPU SM's implement in-order SIMD pipeline and have significantly different architecture and data/control flow from traditional CPUs. Sim-SODA is not applicable to the GPGPU architecture. For instance, in order to calculate AVF of structures buffering in-flight instructions (e.g., issue queue and reorder buffer in the CPU core, the warp scheduler in SM), the framework needs to identify ACE instructions by tracing the data dependence chains on the instruction output(s). Sim-SODA is built upon Alpha ISA, and only supports the analysis on instructions with two inputs and one output. However, instructions in GPGPUs are likely to consume/produce vector input/output, which requires a more complicated analysis considering all the data dependencies among the vector inputs and outputs. Moreover, Sim-SODA mainly targets single-thread uniprocessors in which the AVF computation is relatively simple. On the contrary, GPGPU contains tens of SMs with thousands of parallel threads running simultaneously, and threads within a block share data via the per-core shared memory. Hence error propagation from one thread to another thread is possible. In other words, the instruction output from one thread could affect the correct execution of a different thread. When estimating the GPGPU's microarchitecture vulnerability, a comprehensive methodology is required to consider the vector input/output and thread-level error propagation in ACE instruction identification, which is obviously far beyond the capability of Sim-SODA.

We build GPGPU-SODA (GPGPU SOftware Dependability Analysis) on a cycle-accurate, open-source, and publicly available simulator GPGPU-Sim [29]; it supports CUDA parallel thread execution (PTX) ISA. GPGPU-SODA is capable of estimating the vulnerability of the major microarchitecture structures in each SM, including the warp scheduler, streaming processors, registers, shared memory, and L1 texture and constant caches. The structures in SMs are classified as two types: address-based structures keeping computation data values (e.g., registers, shared memory, and L1 caches) and structures buffering instructions (e.g., warp scheduler). To compute AVF for address-based structure, GPGPU-SODA summarizes the ACE components [30] of each bit's lifetime in the structure. In order to calculate AVF for structures buffering instructions, GPGPU-SODA implements an instruction analysis window [27] for each thread. It explores data dependence chains for instructions with multiple inputs and outputs to perform the comprehensive ACE instruction identification. Note that even threads within a warp share the same PC; an analysis window on one thread for the entire warp is not sufficient. Because it ignores the data dependencies across threads caused by the interthread data sharing, an ACE instruction may be incorrectly identified as un-ACE. In CPU processors, an analysis window with a size of 40,000 instructions is required to determine un-ACE instructions. Since GPGPUs workloads are composed of lightweight threads, the window size is largely reduced to 1000 instructions in GPGPU-SODA. We classify the instruction as vulnerability-unknown if its vulnerability cannot be determined by our 1000-instruction analysis window. The percentage of unknown instructions is mostly lower than 5%. We use synthetic microbenchmarks that have small amounts of instructions to validate the analysis windows. Their reports on un-ACE instructions match our expectations.

3.3 SOFT-ERROR VULNERABILITY ANALYSIS ON GPGPUs

3.3.1 Experimental methodologies

We use the developed GPGPU-SODA to obtain the GPGPU reliability and performance statistics. Our baseline GPGPU configuration models the Nvidia Fermi-style architecture: the GPU contains 28 SMs; the warp size is 32; each SM supports 1024 threads and 8 blocks at most; each SM contains 64 KB registers, and 16 KB shared memory; the scheduler applies the round-robin among the ready warps scheduling policy.

We collected a large set of available GPGPU workloads from Nvidia CUDA SDK [31], Rodinia Benchmark [32], and Parboil Benchmark [6]. We list them as follows: *64H* (64 bin histogram), *BFS* (breadth-first search), *BP* (back propagation), *BS* (Black-Scholes option pricing), *CP* (Columbic potential), *FWT* (fast walsh transform), *HS* (hot spot), *KM* (K-means), *LIB* (LIBOR), *LPS* (3D Laplace solver), *LV* (Levenshtein edit-distance calculation), *MRIF* (magnetic resonance imaging FHD), *MT* (matrix transpose), *NE* (nearest neighbor), *NN* (neural network), *NW* (Needleman Wunsch), *SLA* (scan of large arrays), *SP* (scalar product), *SRAD* (speckle reducing anisotropic diffusion), *ST3D* (stencil 3D), and *STO* (store GPU). The workloads show significant diversities according to their kernel characteristics, divergence characteristics, memory access patterns, and so on. They are compiled into PTX assembly, and GPGPU-SODA takes the produced PTX assembly instructions associated with the information on thread-level registers, shared memory, and memory usages to perform the simulation. We simulate most benchmarks to completion, except a few that causing an extremely long simulation time. AVF is used as the basic metric to estimate how susceptible a GPGPU microarchitecture structure is to soft-error attacks.

3.3.2 Soft-error vulnerability of the GPGPU microarchitecture structures

Fig. 2 profiles the instruction vulnerability characteristics in various benchmarks. It shows the percentage of ACE, un-ACE, and vulnerability-unknown instructions for each workload. The un-ACE instructions are further classified as first dynamically dead (FDD) instructions whose results are not consumed by any other instructions [27], and transitively dynamically dead (TDD) instructions whose results are only read by FDD or TDD instructions [27]. On average, there are 88% ACE, 8% un-ACE, and 4% vulnerability-unknown instructions across the benchmarks. As shown in Fig. 2, most benchmarks have a large number of ACE instructions, with benchmark *SLA* the exception as it contains only 41% ACE instructions. Generally, threads running in GPGPUs are mainly composed of loops, and un-ACE instructions that appear in the loop will be repeatedly executed and will contribute to a large number of un-ACE instructions in the workload. This is the case in *SLA* whose un-ACE instructions mainly exist in the loops. Our GPGPU-SODA framework reports 1% FDD and 8% TDD instructions, which dominate the un-ACE instructions. This is different from the general-purpose workloads in CPU processors that have 10–30%

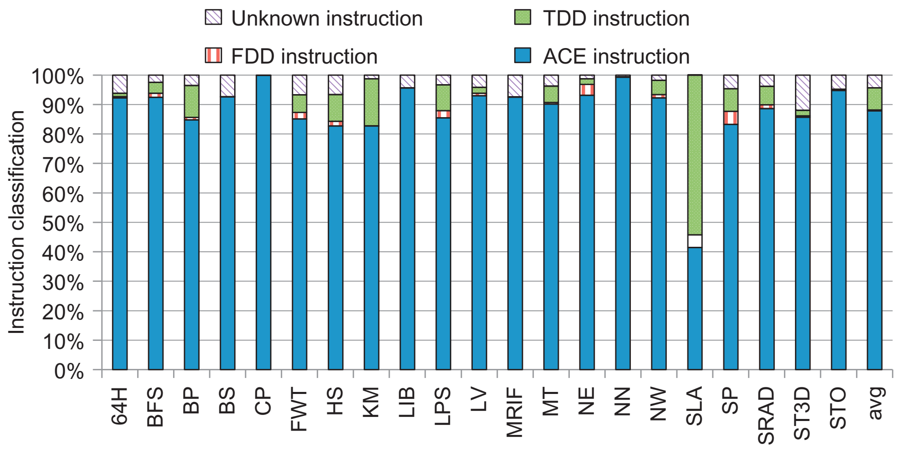


FIG. 2

The percentage of ACE, FDD, TDD, and unknown instructions.

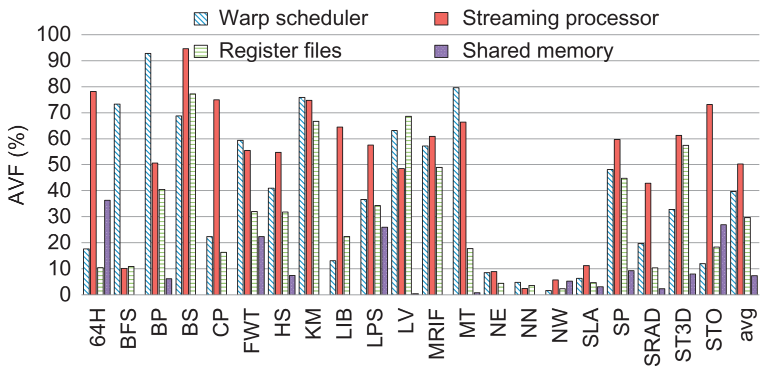


FIG. 3

The AVF of GPGPU SM microarchitecture structures.

NOP (No Operation) instructions, which are usually inserted for pipeline flushing or instruction alignment in VLIW processors [27,28,33].

Fig. 3 shows the soft-error vulnerability of several key structures in GPGPU SMs with the baseline GPGPU configuration. We present the averaged result across the SMs. The AVF of L1, constant, and texture caches are not shown in Fig. 3. This is because the workloads we studied either do not or rarely use those structures, and their AVF is even lower than 4%. As shown in Fig. 3, the primary microarchitecture structures (e.g., warp scheduler, registers) that are heavily utilized exhibit high vulnerability. The AVF of the warp scheduler, registers, and streaming processors increases up to 92%, 77%, and 95%, respectively.

Analysis on structure's AVF in SM

As shown in Fig. 3, structure vulnerability varies dramatically across the workloads. For instance, the warp scheduler's AVF is 2% in *NW* but 92% in *BP*. In general, the GPUs' microarchitecture structures show comparatively low AVF in workloads containing a large number of un-ACE instructions that are immune to the soft-error strikes, such as *SLA*. However, the AVF of structures is not always high even in programs with a significant amount of ACE instructions. For example, the AVF of the warp scheduler, streaming processors, and register files is around 5% in *NE*, *NN*, and *NW* benchmarks, although more than 90% of the instructions are ACE. This is because they contain quite a limited number of threads, and most pipeline resources are idle leading to low soft-error vulnerability. As introduced in Section 2, the per-block resources (e.g., registers, thread slots in the warp scheduler, and shared memory) will not be released until the block completes execution, which limits the maximum number of blocks that can be simultaneously assigned to an SM. Different per-block resources become the bottleneck for kernels that have different resource requirements. The bottleneck structure is prone to be fully utilized and manifests high vulnerability, while other structures are usually underutilized and show strong capability in fault tolerance. In *ST3D*, the register file size determines the number of blocks, and the warp scheduler has numerous idle thread slots at runtime. Hence it has low AVF even though most of the instructions sitting there are ACE. For the same reason, some benchmarks (e.g., *BP*) have reliable registers but vulnerable the warp scheduler, which is the resource bottleneck.

In the SM, shared memory is mainly designed for interthread communication, and it is not used in kernels without thread communication/synchronization. Therefore the AVF of shared memory is zero in *BFS*, *BS*, *CP*, *KM*, *LIB*, *MRIF*, *NE*, and *NN*, as shown in Fig. 3. To improve the performance, program developers use the shared memory as software managed on-chip cache for the global memory and reduce the off-chip memory accesses. Intuitively, the shared memory turns out to be vulnerable when it is used for thread communication or performance enhancement purposes, and becomes the resource bottleneck during the block allocation. However, its AVF still stays as low as 12% on average across the benchmarks that use the shared memory. Shared memory is highly banked. The bank selected to hold a data value is determined by the data address, which leads to the unbalanced bank usage in a block. The number of blocks that each bank can support is different, and the minimum number finally limits the number of blocks the shared memory can support. Even though shared memory becomes the resource bottleneck, most banks in it may be underutilized, and the vulnerability of the entire structure is low. Fig. 4 presents the percentage of used entries in the shared memory for each benchmark. On average, it is only 24%. In workloads whose block resource allocation is limited by the share memory (e.g., *64H*, *LPS*, *NW*, *SP*, *SRAD*, and *STO*), more than 40% of entries are never used during the entire execution time. The used entries are written/read in a

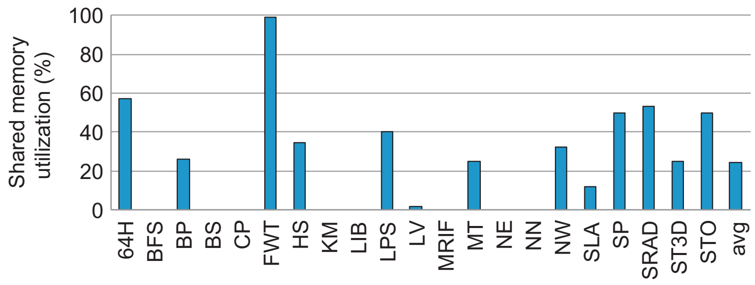


FIG. 4

Shared memory utilization in percentage.

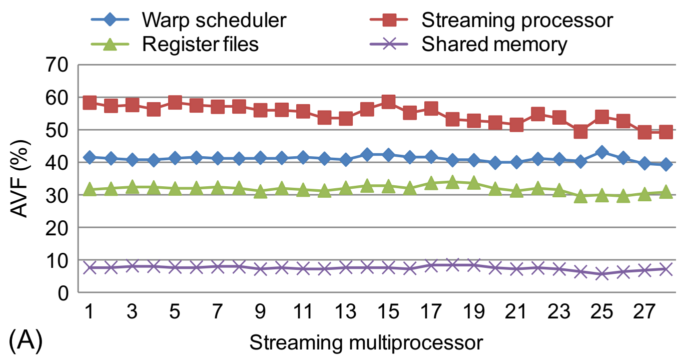
very short period and become free the majority of the time. Therefore the shared memory has very low vulnerability.

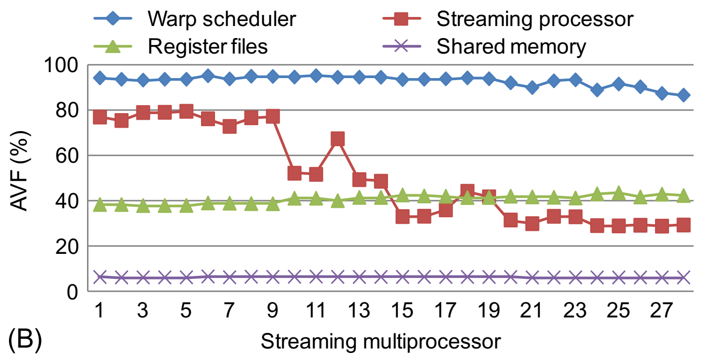
As shown in Fig. 3, the streaming processors' AVF has strong correlation to the number of ACE instructions in most benchmarks. Since the instruction execution time in streaming processors is constant, the ACE instruction quantity per cycle becomes the major factor to determine the AVF. However, streaming processors' AVF is only 10% in *BFS* even 92% instructions in it are ACE. BFS has heavy branch divergences and long-latency off-chip memory accesses. The diverged threads within the warp have to execute sequentially, which causes tremendous performance penalty. In our baseline machine configuration, the immediate postdominator-based reconvergence is applied to reconverge threads at the immediate postdominator (detailed explanation can be found in Ref. [23]) to handle the branch divergences. However, it has limited capability to efficiently recover the performance loss. Furthermore, the long-latency memory accesses increase the streaming processors' idle time. Accordingly, the parallelized streaming processors are highly underutilized in *BFS*, and the AVF is low.

Vulnerability variations at streaming multiprocessor level

A GPGPU processor contains tens of SMs. Generally, they are equally utilized and exhibit similar vulnerability behavior. Fig. 5A shows the AVF of the major microarchitecture structures in each SM when executing *HS*. The figure shows a small AVF divergence among the SMs. However, this is not the case for some benchmarks.

Fig. 5B demonstrates an example of *BP* with its streaming processors' AVF varying significantly across SMs. For example, the streaming processors' AVF is 80% in SM 9, but drops to 29% in SM 27. This happens because the threads running in each SM have a different memory access pattern, which determines the quantity of memory requests that can get coalesced (more coalesced requests implies fewer off-chip memory transactions), and the severity of memory contentions among requests affects the memory access latency. SM 27 experiences more off-chip memory accesses compared to SM 9; the streaming processors in SM 27 are idle in





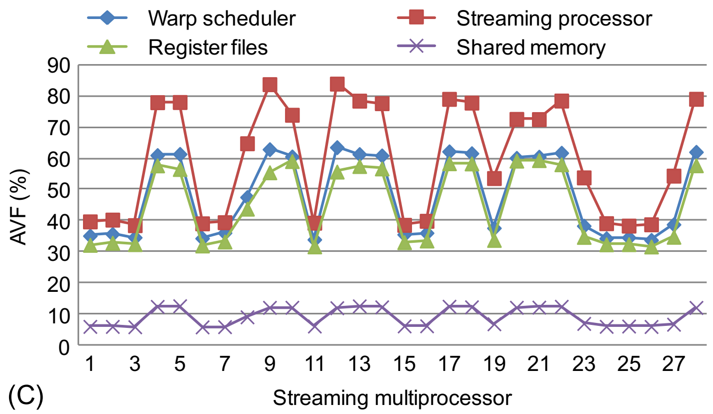


FIG. 5

The AVF of microarchitecture structures in each SM while running (A) FWT, (B) LPS, and (C) SP.

majority of the time, and correspondingly, they are more robust against the soft-error strikes. Interestingly, the AVF of warp scheduler, register files, and shared memory does not show much difference among SMs. During the kernel launch process, all its blocks may not be able to distribute to the SMs and start the execution simultaneously since the maximum number of blocks an SM can support is limited. The block that cannot be allocated to the SM at the kernel launch time has to wait until one block finishes and its resources in the SM are released. By doing this, the execution time of each SM tends to be balanced. Furthermore, SMs with a long block execution time caused by the frequent memory accesses execute fewer blocks and have a smaller quantity of ACE instructions in the storage-based pipeline structures (e.g., warp scheduler) through the entire execution time. However, instructions have to remain in those structures during the off-chip memory accesses, and their residency time increases. As a result, the AVF of the warp scheduler, register files, and shared memory are nearly the same across SMs.

Fig. 5C shows another example of *SP*; the structure of the AVF differs substantially across SMs, and structures in the same SM show similar vulnerability behaviors. Take SM 7 and 9 for example; the vulnerability of the major structures including the warp scheduler, streaming processors, and registers is low in SM 7 but is much higher in SM 9. This is caused by the uneven distribution of blocks among SMs. SM 7 is assigned 3 blocks for execution, while the number of blocks allocated to SM 9 is doubled, leading to a much larger number of ACE instructions. Moreover, different from *BP*, the execution time of SMs in *SP* varies significantly; SM 7 finishes its workloads much earlier than SM 9, which implies that the residency time of ACE instructions in both SMs are similar. Therefore structures in SM 9 are more vulnerable to soft errors because of their numerous ACE instructions.

When a thread encounters a barrier instruction, it has to wait until the synchronization in the block finishes. In other words, warps will stall upon the synchronization requests. Similar to the long-latency off-chip memory accesses, the barrier instructions may increase execution time, thus affecting the structure of soft-error robustness. Since warps are issued to the pipeline in a round-robin order, in our baseline GPGPU's machine, threads within the block usually proceed at a similar rate; the warp waiting time at the barrier is short compared to that for the off-chip memory access. Furthermore, barrier instructions account for only 2.5% of the total instructions in most benchmarks. Their impact on structure vulnerability is trivial.

In summary, we observe that the GPGPU's microarchitecture vulnerability is highly related to workload characteristics such as the percentage of un-ACE instructions, the per-block resource requirements, the branch divergences, the memory access pattern (e.g., memory coalescing, memory resource contentions, and memory access latency), the block allocation that determines the execution time, and the amount of workload executed in each SM.

3.3.3 Impact of architecture optimizations on GPGPU microarchitecture-level soft-error vulnerability

In this section, we evaluate the impact of several popular architectural design optimizations on the soft-error vulnerability of GPGPUs microarchitecture.

Dynamic warp formation

Branch divergence is a major cause for performance degradation in GPGPUs. As we discussed earlier, the immediate postdominator (PDOM) lacks the capability to reconverge threads at the beginning for branch divergence to further improve the performance. DWF is proposed in Ref. [24] to efficiently handle the threads' divergence. It groups threads from multiple warps but branching to the same target into a new and complete warp, and issues it into the SIMD pipeline. Therefore the parallel streaming processors in the SM are fully utilized, and their performance is enhanced.

In our baseline GPGPUs, a warp splits into multiple warps at the occurrence of the branch divergence; those spawned warps have to consume the issued bandwidth to leave the warp scheduler. They delay the issue time of other warps and increase the warp residency time. DWF regroups the divergent threads into new warps. It reduces the number of warps competing for the issue bandwidth and shrinks the warp waiting time. Meanwhile, DWF aims to improve the performance and reduce the kernel execution time. Note that reducing the warp waiting time and correspondingly the ACE instruction residency time can help to decrease AVF, but shrinking the execution time will increase the AVF as it increases the ACE instruction quantity per cycle [27]. Therefore the impact of DWF on the warp scheduler's vulnerability can be negative or positive, depending on the warp residency time and the kernel execution time. When the reduction of warp residency time dominates that of the kernel execution time, the warp scheduler AVF decreases, and vice versa. Since the warps enter the SIMD pipeline earlier, registers are written/read faster, which helps to reduce their lifetime. Similar to the warp scheduler, the DWF could increase/decrease the registers' AVF.

Fig. 6 presents the structures' AVF under the impact of DWF; they are normalized to the baseline case with PDOM applied for comparison. Benchmarks *KM*, *LIB*, *MRIF*, *NN*, and *ST3D* cannot execute when DWF is enabled, so we eliminated those benchmarks from this figure. The results of the L1 caches and shared memory are not shown in Fig. 6 and the following figures since they are always robust to the soft-error strikes under different architectural optimizations. On average, DWF optimizes the warp scheduler and the registers' reliability by 5% and 5%, respectively. Obviously, most benchmarks studied here fall into the category in which DWF has positive effect on those two structures reliability. Note that the capability of DWF to optimize a structure's reliability is significantly affected by the percentage of branch divergence operations in the benchmark. When there are few branch divergences, DWF has limited opportunities to explore the warp formation, whose effect on reliability is trivial. For example, the warp scheduler AVF decreases only 1% in *CP* with 2% branch divergence operations. On the other hand, DWF

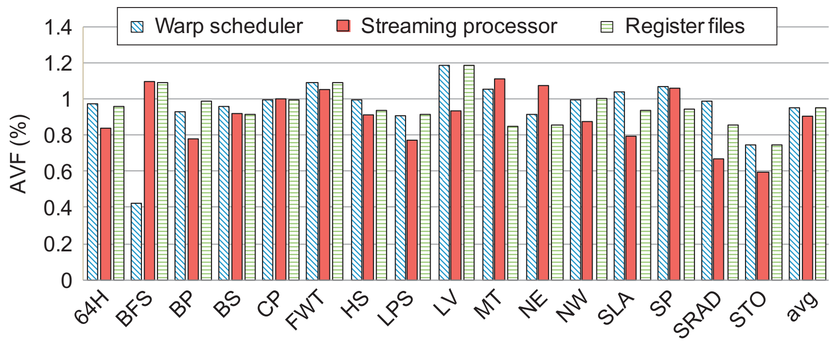


FIG. 6

The normalized AVF of GPGPU streaming multiprocessor microarchitecture structures under the impact of DWF.

helps to improve the warps scheduler's reliability by 60% in *BFS* with 40% branch divergence operations.

The streaming processors' AVF is highly related to the kernel execution time. Since the instruction computation time is mainly determined by the instruction type, a shorter kernel execution time implies an increased quantity of ACE instructions per cycle in the streaming processors. And the streaming processors are more susceptible to soft errors when the execution time shrinks. Intuitively, DWF would hurt the streaming processors' soft-error vulnerability because it targets on optimizing performance and reduces the kernel execution time. Interestingly, DWF decreases streaming processors' AVF by 10% on average as shown in Fig. 6. This is because DWF decreases the IPC in a number of benchmarks (as shown in Fig. 7). When randomly regrouping threads into new warps, DWF may lose opportunities to combine memory accesses originally from the same warp, and introduce extra off-chip memory transactions, which negatively affect the performance. When the

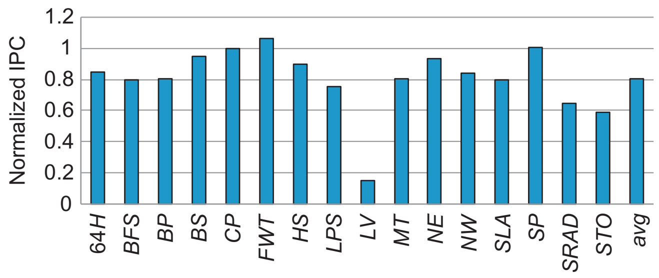


FIG. 7

The normalized IPC under DWF.

benefit of DWF in branch divergences is outweighed by the increased memory access requests, performance starts to degrade.

Off-chip memory access

In the GPGPUs' applications, all threads in a kernel execute the same code and exhibit similar execution progress in the fine-grained multithreading environment. When one thread encounters an off-chip memory access, other threads are likely to issue requests at approximately the same time, leading to severe resource contentions on both the on-chip network and the memory controllers. The effect of interconnect topologies and request scheduling policies in the memory controller on GPU throughput was studied in Ref. [29]. However, their impact on GPU microarchitecture soft-error vulnerability is largely ignored.

Fig. 8A shows the vulnerability of SM structures when applying the FIFO scheduling policy in the memory controller and two interconnect network topologies (i.e., Torus and Fly), respectively. Results are normalized to our baseline configuration with mesh topology and first-ready first-come first-serve (FR-FCFS) scheduling policy. As it shows, even though the performance is hurt considerably while using the Torus network topology (Fig. 8B also presents the normalized IPC for the vulnerability analysis), in general the interconnect network topology has little impact on the vulnerability of the warp scheduler and register files. For instance, the two structures' AVF is similar to the baseline case, while the IPC decreases 23% in NE under the Torus topology. The reduced IPC implies fewer ACE instructions per cycle in the warp scheduler; however, the instructions spend longer time in the structure, and its vulnerability remains almost the same as the baseline case. A similar case occurs in the register files. On the other hand, the streaming processors' AVF varies noticeably in NE under the Torus topology, because it exhibits strong correlation with the GPU performance.

The FIFO scheduling policy simply serves the memory requests in the order of their arrival time in the memory controller. It causes a frequent switch among DRAM rows, which increases the request service time and the memory access latency. FR-FCFS gives the memory requests that hit the open DRAM row higher priority to access the DRAM. Therefore the IPC decreases 22% under the FIFO scheduling policy compared to the baseline case that uses FR-FCFS. Similar to the case with different network topologies, the warp scheduler and registers' vulnerability changes slightly under FIFO compared to the baseline case, while the impact of the FIFO policy on SM vulnerability is considerable, as shown in Fig. 8A.

Number of threads per SM

In previous works, the effect of altering the number of simultaneously running threads on performance was well explored [29], but its impact on GPGPU microarchitecture soft-error vulnerability is still unknown. In order to change the number of parallel threads per SM, we scale the per-block resources (the amount of maximum blocks, shared memory, and registers) between 50% and 200% to the baseline case, which supports 1024 threads.

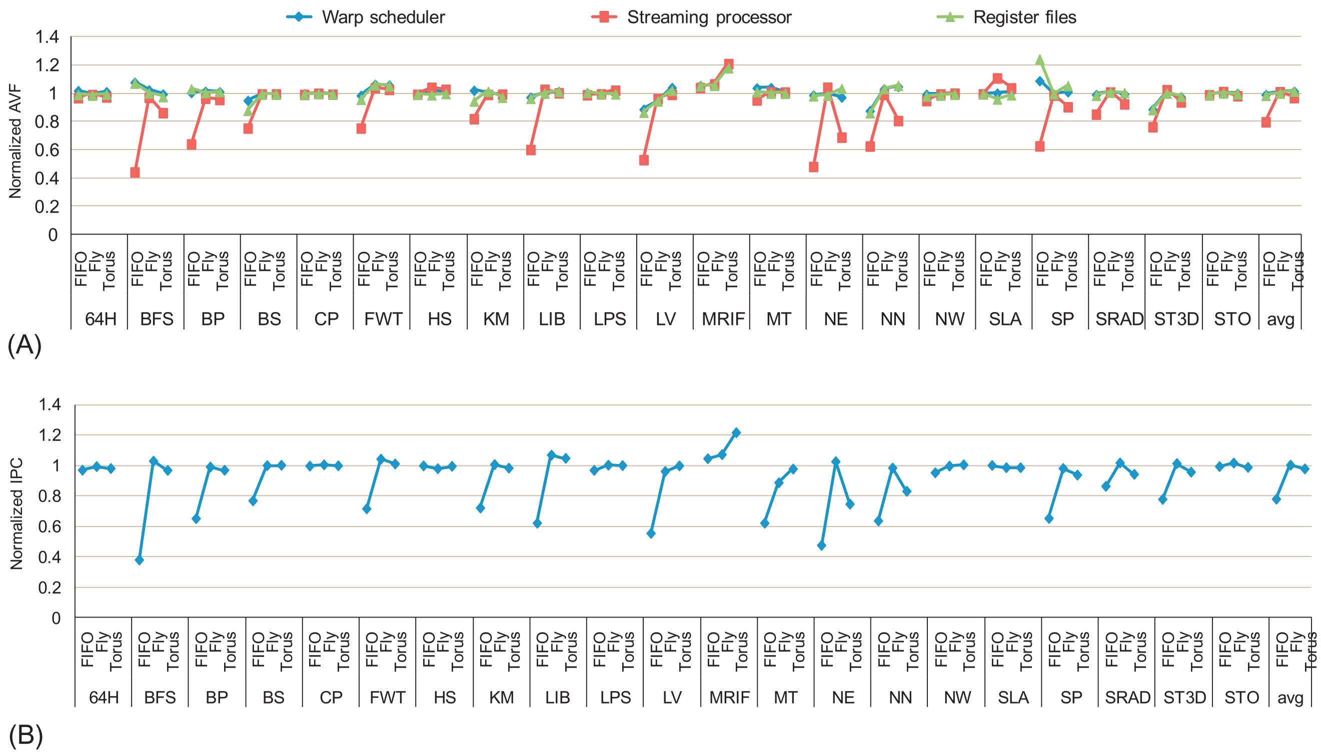


FIG. 8

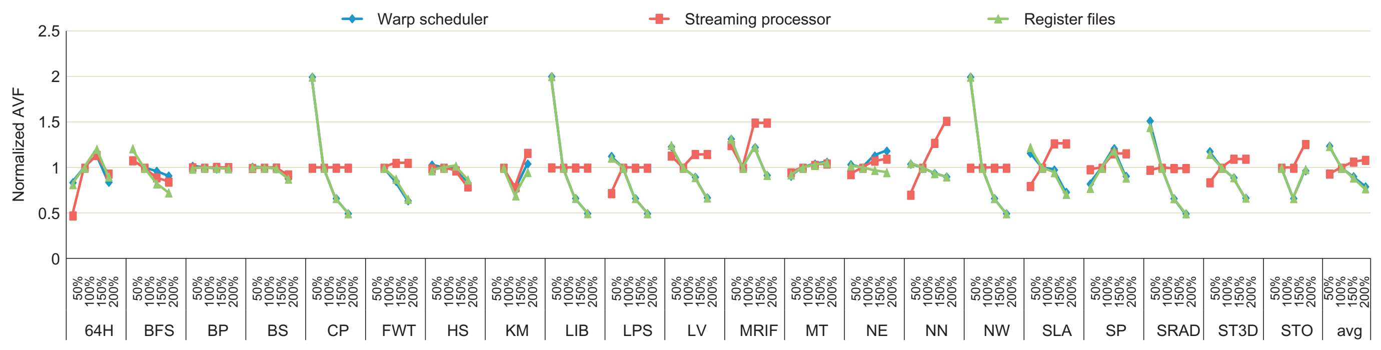
The normalized (A) AVF and (B) IPC of GPGPU microarchitecture structures when applying the FIFO scheduling policy in the memory controller, the Fly, and the Torus network topology, respectively. Both memory controller scheduling and network topology have considerable impact on streaming processors' AVF and have little impact on the vulnerability of the warp scheduler and register files.

Fig. 9 shows the AVF and IPC results with different amounts of threads per SM. The results are normalized to the baseline case. When the per-block resources shrink to 50%, the kernels in *FWT*, *KM*, and *STO* fail to launch in the GPGPU processor because of the insufficient resources for even one block. Therefore the results for *FWT*, *KM*, and *STO* are missing from the 50% reduced thread count (512) in Fig. 9. As Fig. 9A and B shows, both AVF and IPC vary significantly across the benchmarks with the increasing number of threads per SM. We characterize the benchmarks based on their reliability behaviors as we describe next.

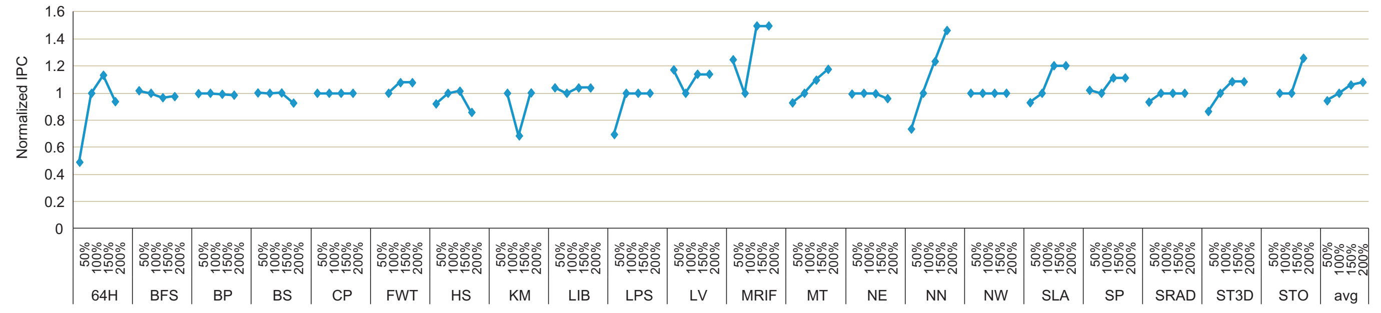
In *CP*, *LIB*, and *NW*, the 50% thread configuration is already sufficient for all the blocks to execute simultaneously; additional thread slots and resources are not used and do not contribute to the performance. Therefore the performance of these benchmarks remains the same as shown in Fig. 9B. In that case, the size of the warp scheduler and registers scales up with the increasing thread count, but their utilization reduces because of the increasing idle entries. Therefore, as Fig. 9A demonstrates, the AVF of the warp scheduler and registers decreases as the number of threads increases. On the other hand, the streaming processors' size and utilization do not change with thread count; hence there is little change in the streaming processors' AVF.

Similarly, *LPS* and *SRAD* are unable to leverage the additional resources beyond the baseline configuration; they show the same behavior as *CP* from the 100% to 200% configurations. However, in the 50% configuration, there are insufficient resources to hold blocks and feed the pipeline. Streaming processors are highly underutilized, and performance and the streaming processors' AVF both decrease. The performance of these two benchmarks is the same from 100% to 200% configuration and much higher than 50% configuration, as shown in Fig. 9B. For *FWT*, *SLA*, *SP*, and *ST3D*, the performance reaches the peak for the 150% configuration; they show the same behaviors as *CP* from the 150% to 200% configurations. *NN* is able to utilize the increased resources, so the performance is improved as the configuration increases from 50% to 200%. Correspondingly, the streaming processors' AVF increases. Moreover, the vulnerability of the warp scheduler and register files decreases slightly, because their usage decreases to some degree with regard to the continuously increased structure size from 50% to 200%.

In *BFS*, *BP*, *BS*, and *NE*, there are enough blocks for the simultaneous execution with the increasing number of threads supported by the SM. However, the performance of these benchmarks remains nearly the same from 50% to 200% configurations (shown in Fig. 9B). The contentions on the interconnect network and memory resources among threads become more severe as the number of concurrent threads increases in the SM, and instructions spend a longer time in the pipeline structures, which induces a negative effect GPU performance. As a consequence, the structure utilization is similar across the different configurations, and the configuration variation has a negligible impact on the SM structure's vulnerability in those benchmarks, except *BFS* as shown in Fig. 9A. In *BFS* the majority of the SMs are idle, while only few still execute blocks at the end of the kernel execution time. As the configuration changes from 50% to 200%, there are more free resources



(A)



(B)

FIG. 9

The normalized (A) AVF and (B) IPC of GPGPU microarchitecture structures when the number of threads per SM is 50% (512 threads), 150% (1536 threads), and 200% (2048 threads) compare to the baseline case. Increasing the quantity of threads improves the warp scheduler and registers' soft-error robustness, but degrades the streaming processors' reliability.

in SMs, which helps to decrease the structures' AVF. Moreover, *KM* exhibits similar behavior as *BP*, except the performance and vulnerability decease in the 150% configuration compared to the baseline case because of the serious workload imbalance among SMs.

As Fig. 9B shows, the IPC increases from 50% to 150% configurations in 64*H* and *HS*, which implies that the increasing resources are able to support more concurrent threads and improve the throughput. Take *64H* as an example, the 50% configuration (especially the 50% shared memory) significantly limits the number of blocks executed in the SM and warp scheduler, and streaming processors are underutilized and exhibit extremely low soft-error vulnerability. However, their performance degrades at 200% configuration because of the more severe resource contentions among threads.

In *STO* the IPC remains the same because the number of threads increases from 100% to 150%. Because the 150% configuration introduces uneven block distribution among SMs, several SMs are heavily loaded, which determines the total kernel execution time. Moreover, the vulnerability of the warp scheduler and register files decreases in the 150% configuration because their overall utilization (considering the increased structure size) across SMs decreases. The performance further improves as the number of threads increase up to 200%, meanwhile the structure vulnerability increases as well. Because the SMs with 200% configuration have balanced workloads, the execution time is greatly reduced. The impact of the workload imbalance also exists in *LV* and *MRIF*. As Fig. 9B shows, the IPC is low in the baseline case (i.e., 100% configuration) compared with other configurations. This is because the 100% configuration results in the serious workload imbalance among SMs.

In *MT*, both the performance and structure vulnerability increases smoothly as the number of threads increases from 50% to 200%. This is because the utilization of each structure increases as the number of threads increase in each SM, which improves the throughput but hurts the SM soft-error robustness.

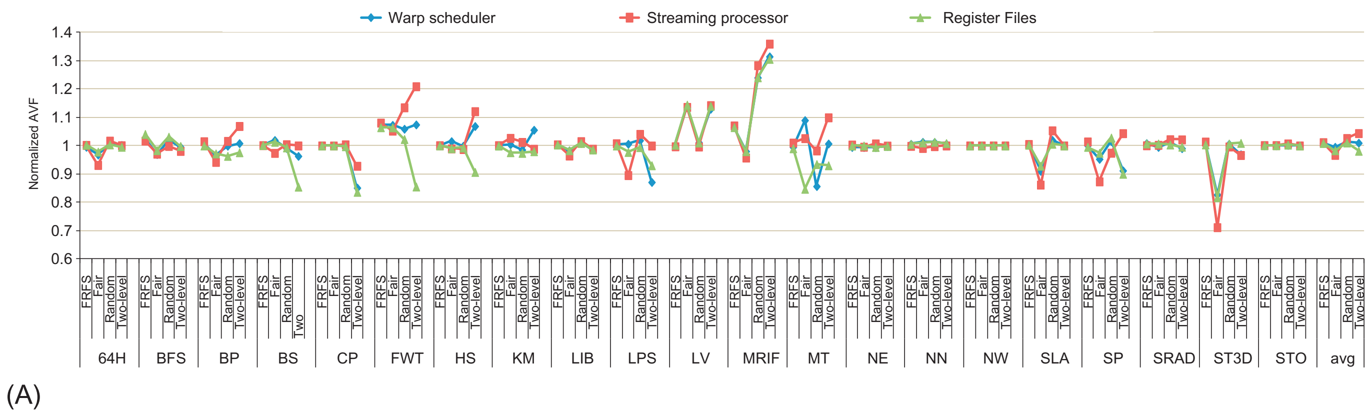
In summary, on average, across the benchmarks increasing the quantity of threads greatly affects the GPGPU microarchitecture soft-error robustness. It improves the warp scheduler and registers' soft-error robustness, but it degrades the streaming processors' reliability.

Warp scheduling

The warp scheduling policy greatly affects GPGPU throughput. In previous work, Lakshminarayana et al. [34] explored several policies and evaluated their impact on performance. In this section, we analyze their effectiveness from the reliability perspective. We investigate four warp scheduling policies: first-ready first-serve (FRFS), Fair (issuing a warp with the minimum instructions executed), Random, and two-level, round-robin scheduling (two-level) that distributes warps into several fetch groups and applies round-robin policy at the group level and the warp level in each group to select appropriate warps for issuing [35]. They are compared to the baseline case of round-robin scheduling. Fig. 10 presents the normalized AVF and IPC under the impact of various policies.

As Fig. 10 shows, on average, across the benchmarks, the structure AVF varies less than 8% under different scheduling policies when compared to the baseline case, and the IPC variation is even less than 5%. The structure vulnerability under the FRFS and Random scheduling policies is similar to the baseline case, but the vulnerability variation is more noticeable in several benchmarks (e.g., *LPS*, *LV*, *MRIF*, *ST3D*, and so on) when Fair and two-level polices are enabled. The Fair policy keeps the uniform progress among warps. It may increase the opportunities for interwarp memory coalescing when threads across warps have similar memory access patterns, and it reduces the warp waiting time. However, when it fails to merge the interwarp memory accesses, a large number of memory requests are generated in a very short time span, which results in heavy memory resource contention and extends the warp stall time. Hence the warp scheduler's AVF increases when the total execution time has a minor increase, such as *MT* shown in Fig. 10A. When a significant number of off-chip memory transactions fail to get coalesced, and the workload execution time severely extends, the significantly reduced number of ACE instructions per cycle dominate the increased instruction residency time in the pipeline structure, and as a result, the structure AVF decreases considerably (e.g., *SLA*, *SP*, and *ST3D*). The two-level scheduling policy aims to hide the long-latency memory accesses and improve performance; therefore, the number of ACE instructions per cycle increases and structures become more susceptible to soft errors (e.g., *LV* and *MRIF*).

To conclude, in this section, we first develop GPGPU-SODA to evaluate the GPGPU microarchitecture soft-error vulnerability. As our GPGPU-SODA suggests, the majority of the structures in GPGPUs (e.g., warp scheduler, streaming processors, and register files) are highly vulnerable, so a comprehensive technique is needed to protect them against the soft-error attacks. We observe that the structure's vulnerability is largely affected by the workload characteristics. For example, both branch divergences and long-latency memory accesses decrease the streaming processors' vulnerability. Moreover, SMs in a single GPGPU processor may manifest significantly different reliability characteristics. We further analyze the impact of several architecture optimizations on GPGPUs' microarchitecture vulnerability. For example, increasing the thread quantity per SM improves the warp scheduler and register files' reliability, but degrades the streaming processors' soft-error robustness. We also found that the structure vulnerability is insensitive to the warp scheduling polices. The observations of the microarchitecture structure's vulnerability characteristics made in this section provide useful guidance in building future resilient GPGPUs. Based on these observations, we suggest the GPGPU reliability-optimization technique should consider all the GPGPUs' microarchitecture structures and workload characteristics and that a technique focusing on one particular structure will not be an efficient, resilient solution for GPGPUs.



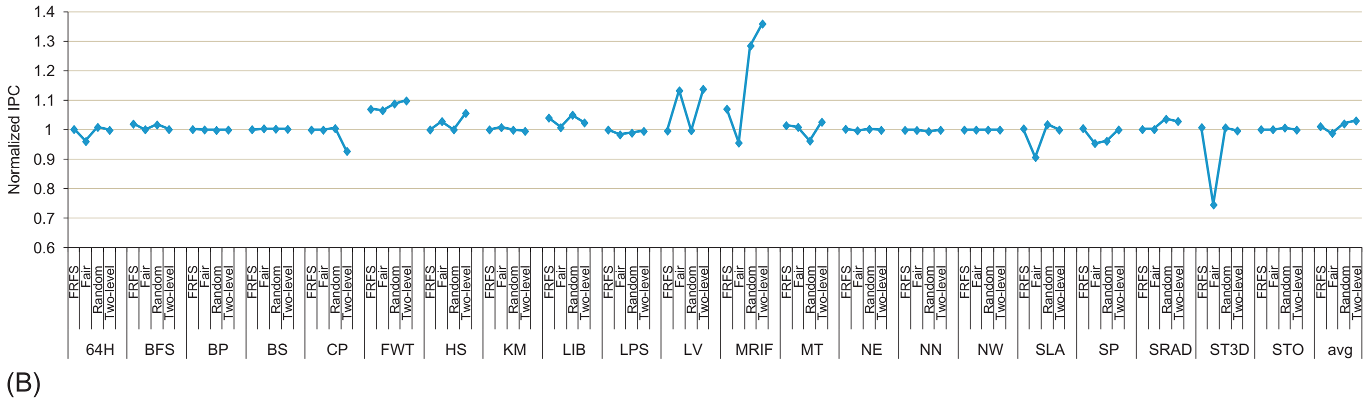


FIG. 10

The normalized (A) AVF and (B) IPC of GPGPU microarchitecture structures under the impact of FRFS, Fair, Random, and two-level warp scheduling policies, respectively. The structure vulnerability is insensitive to the warp scheduling polices.

4 RISE: IMPROVING THE STREAMING PROCESSORS' RELIABILITY AGAINST SOFT ERRORS IN GPGPUS [36]

In each GPGPU streaming multiprocessor, there are several parallel streaming processors (SPs). They perform the fundamental computing operations and play an important role in exploiting the parallel computing. In the GPGPUs 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 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 3.1). In addition, as the key and representative combinational logic-based structure in the GPGPUs, SPs occupy a large portion of the chip area, and the SER of SPs becomes the major contributor to the overall SER of a GPGPU processor. Effectively protecting SPs becomes the essential first step to building a the resilient GPGPU architecture.

To overcome the reliability challenge, duplication [37] 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 a high-performance penalty. For example, software duplication causes more than 100% a performance penalties [38]. In our experiments, we also evaluate a simple hardware mechanism that duplicates all warp instructions (we refer it as full redundancy). Based on the 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.

As introduced in Section 2, GPGPU exploits TLP by grouping a number of threads into a warp that simultaneously executes the same instruction from each thread, and warps are interleaved on a 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 because of the long-latency operations (e.g., off-chip memory access). In addition, threads in the same warp may take different paths at a branch. They execute sequentially in the diverged warp, and partial SPs in the GPU core become idle [35]. 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 11% of the execution time. The large portion of the SPs' idle time provides great opportunities to trigger the partial redundancy and to trade a little performance degradation for maximal reliability improvement. In this section, we propose *RISE (Recycling the streaming processors Idle time for Soft-Error detection)*, which intelligently leverages the underutilized SPs to perform the redundancy and to substantially improve the SPs, reliability with negligible performance loss.

The details of this work are as follows:

- We propose Full-RISE, which effectively reuses 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 comprises the request pending aware Full-RISE (RP-FRISE) and the idle 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, thereby successfully recycling the stall time for redundancy without degrading the performance. IS-FRISE estimates the load imbalances among cores and smartly triggers the redundancy in cores that 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, thereby utilizing 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 reduce 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 section is organized as follows: Sections 4.1 and 4.2 present our Full-RISE and Partial-RISE techniques, respectively. Section 4.3 introduces the integrate RISE technique. Section 4.4 describes our experimental methodologies and evaluates the proposed techniques.

4.1 FULL-RISE

When implementing the redundant multithreading (RMT) technique in CPUs, the main and redundant threads share the pipeline resources, and 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 in the GPU core, which leads to high use of resources and high power consumption. For example, Wadden et al. evaluate the software RMT in GPUs and observe that the intragroup RMT causes more than 100% slowdown in several applications [38]. 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 on an instruction-by-instruction basis and execute at the same speed.

In previous work, the opportunistic transient-fault detection in CPUs was proposed by Gomma et al. [39]. 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 underutilized pipeline resources to perform the redundancy without degrading the performance. However, the technique is not applicable to the SPs' soft-error detection in the GPGPUs in our study. As previously described, both main and redundant threads in our study share the same resources thus they keep the same execution progress and 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 section, we propose Full-RISE, which is composed of two methodologies: request pending aware Full-RISE and idle SMs aware Full-RISE.

In this study, we focus on the single-bit error model, which has the first-order impact on the failure rate in processors [40]. 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 a 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 section.

4.1.1 Request pending aware Full-RISE

The observation on resource contentions among memory requests

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

Fig. 11 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. Because of the serious resource contentions, the off-chip memory access takes up to thousands of cycles. At the same time, the SM suffers extremely long pipeline stall, and SPs remain idle (as shown in Fig. 11A). Instead of sending the memory requests at a similar time and then stalling for a long time, warps can run at a slower pace, which prolongs their request issues (e.g., req2 $\sim$ reqn in Fig. 11B) and avoids possible resource contentions and thereby having their requests serviced without any delay in the input buffer. Because of the dramatically reduced memory operation latency, the total warp execution time remains the same or even decreases. This provides a great opportunity for temporal redundancy, whose negative effect on performance can be positively used to postpone the warp execution progress and

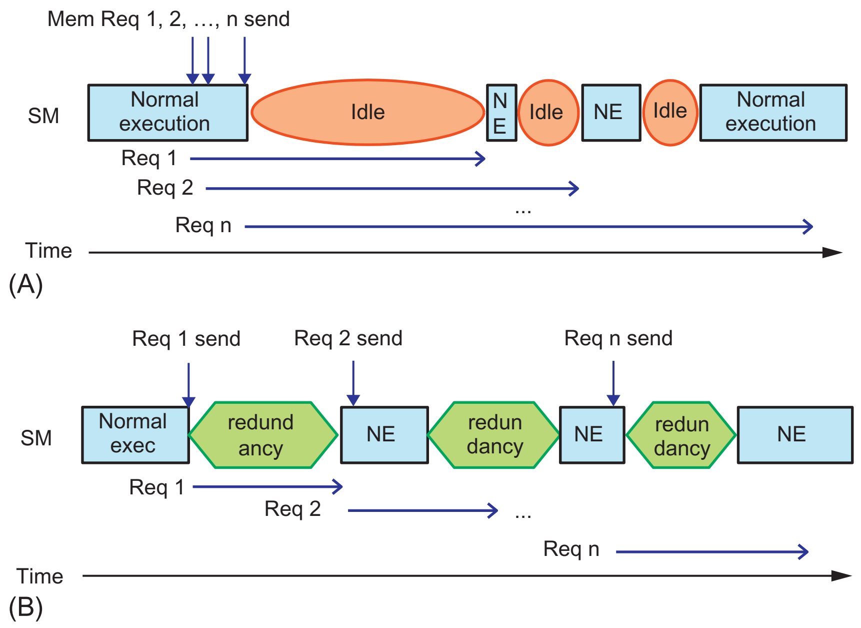


FIG. 11

SPs' idle time (A) caused by resource contentions among memory requests, and (B) recycled for partial redundancy. *NE*, normal execution.

their requests' issue time. The SPs' idle time caused by the request pending in the memory controller (highlighted in oval in Fig. 11A) is successfully recycled for the redundancy (highlighted with Hexagon in Fig. 11B). Note that the redundant and normal executions are interleaved at cycle level. We mark the redundancy separately from the normal execution in Fig. 11B for easy understanding of the impact of redundancy. By running a large number of benchmarks, we observe that the request pending contributes 65% of the time that all SPs in the SM are free. This implies that the SPs soft-error reliability can be substantially improved with no performance penalty.

The concept of request pending aware Full-RISE

As just discussed, the partial redundancy (relative to the full redundancy) can be treated as the knob to control the warp progress. In order to use the SPs' idle time without hurting performance, the partial redundancy has to be carefully tuned: overadjustment will result in excessive redundancy and unnecessary delay, consequentially degrading 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 amounts of time on redundancy in order to separate their memory requests. That is, the warp progress should adapt well to its memory access pattern to achieve the SPs' optimal soft-error robustness. Obviously, statically determining the period for performing the partial redundancy fails to meet the goal since workloads have various memory access patterns. In this study, we propose RP-FRISE (for 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 for the necessity of slowing down the warp. A long waiting time implies a serious resource contention and that the warp that 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 was already postponed after finishing the previous memory access, further slowing it down by the same amount of the previous request waiting time may serious prolong the warp computation. The example in Fig. 12 illustrates the challenge: warp0 and warp1 exhibit a 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 rates after a long-latency memory access, the interference is still severe. Fig. 13 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 the scheduled redundancy time as long, even when performing the full redundancy for the computation between its two memory accesses (highlighted by the light gray rectangular in Fig. 13). In that case, it will be stalled after finishing the redundancy because of the interference with warp0. Therefore the required redundancy time still accurately predicts the necessary delay in warp1. As can be seen in Figs. 12 and 13, 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 brief normal computation in the memory-intensive kernel cannot effectively separate the memory requests, a longer redundant execution is desired.

In our study, we sample the memory access latency periodically when running a kernel and use it to adjust the number 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}).

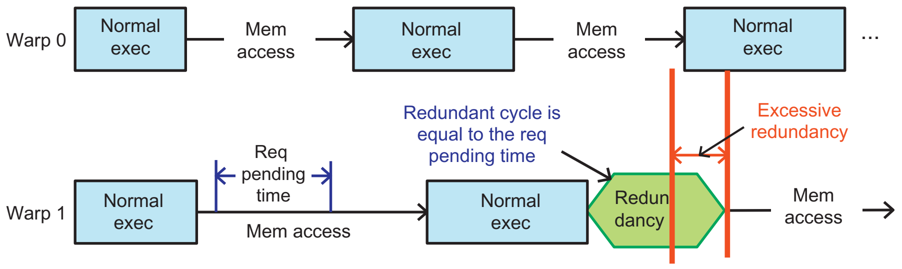


FIG. 12

An example of excessive redundancy.

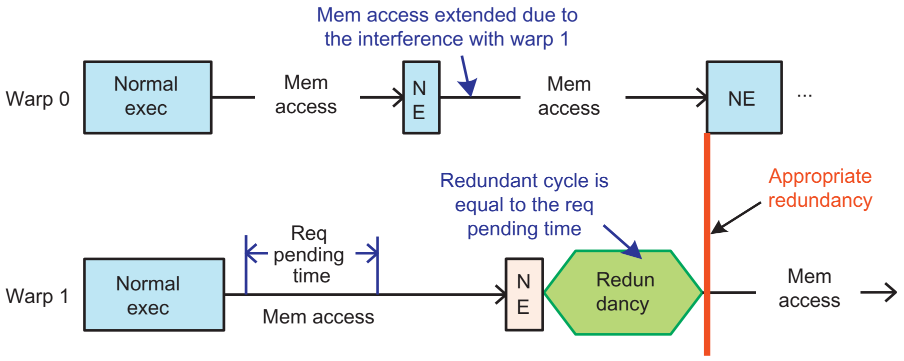


FIG. 13

An example of appropriate redundancy.

$$RC = \begin{cases} 0, & \text{if } T_P \le T_{thr\_pend} \\ \left[\frac{T_{acc\_lat}}{T_{ref\_lat}} \times T_P\right], & \text{if } T_{acc\_lat} < T_{ref\_lat}, T_P > T_{thr\_pend} \\ T_P, & \text{if } T_{acc\_lat} \ge T_{ref\_lat}, T_P > T_{thr\_pend} \end{cases} \qquad (1)$$

Here T_{thr_pend} is the threshold of the request pending time. It is possible that the previous memory access is delayed by the redundancy and that the improper delay (implied in the prior request pending time) cannot further propagate to the following execution. T_{thr_pend} plays an important role in filtering the excessive redundancy out. Only when T_P is longer than T_{thr_pend} is warp redundancy enabled; otherwise, it is disabled to maintain 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 memory-intensive characteristics and aggressive redundancy (i.e., directly setting the redundancy cycles as the request pending time) should be applied to it. To the contrary, a lower T_{acc_lat}

means that the kernel involves heavier computation and that 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, and 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 there exist, and large progress differences 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, and warps in a block tend to show the strong spatial locality [1,41]. The memory requests issued by those warps are likely to be directed to the same row in the DRAM, which is called row locality [35]. When they are sent out simultaneously, the pending time in the input buffer of the memory controller tends to be similar under the OoO memory scheduling policy. On the other hand, it will take even 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 performance, RP-FRISE performs block-level redundancy. Since the redundant time is computed on the basis of memory requests, a single warp may have more than one option for setting the redundancy cycles, and there are numerous choices when extending to a block. With RP-FRISE the redundancy time will be incorporated into the response packet from the memory to the SM. For a block, the first packet arriving after it finishes the previously assigned redundant execution sets the new redundancy cycles, which should be applied to all its warps.

4.1.2 Idle SMs aware Full-RISE

While a kernel is launched into the GPUs, its blocks may not be evenly 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, an SP becomes 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 that are currently running in other SMs. This 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 so 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 called IS-FRISE, as the abbreviation for 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 that 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 an overestimate of the SMs' idle time, which leads to aggressive redundancy and will hurt 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 has an effect on only 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 requests pending time to determine the redundancy cycles for the following execution, and 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 toward the end of the kernel execution. IS-FRISE triggers the full redundancy on certain blocks at the very beginning of the kernel execution, and it differentiates the block progress across SMs and mitigates the resource contentions even before the first memory requests, effectively reducing the possibility of unnecessary redundancy.

4.1.3 The implementation of Full-RISE

Fig. 14 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 that one ALU is sufficient to perform the calculation). When a memory request is written into the input buffer, the timer is set to zero and automatically increments every cycle. It records the request pending cycles that 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 input 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) that lasts for tens of cycles, we set the referred access latency at 2 to the power of n and translate the division into logical shift. This shift operation 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 it is set at $2^7 = 128$ cycles. Note that the

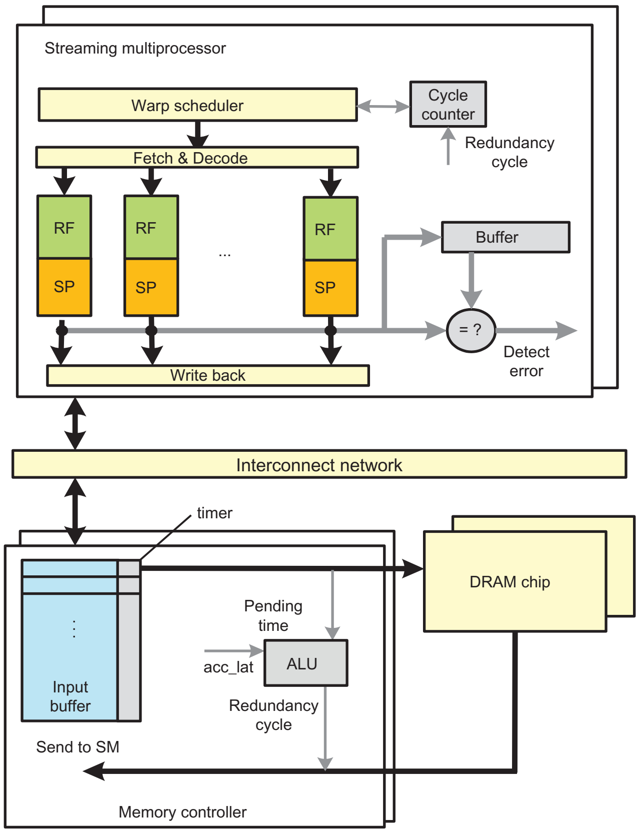


FIG. 14

Implementation of request pending aware Full-RISE.

redundancy time computation occurs in parallel with the data access in DRAM, and it does not introduce any extra delay to the critical path in the memory controller.

As Fig. 14 shows, each block in the SM is allocated a cycle counter that keeps the redundancy cycles. The amount of counters per SM is determined by the maximum number of concurrent blocks an SM supports (e.g., eight in our default configuration). When a response packet arrives at the SM, we can determine 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, and 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, and 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' output of the original warp are written into the destination registers and an attached buffer, while the redundant warp's output will skip the write operation and directly compare it to the data just written into the buffer for error detection. A mismatch will raise the recovery signal. The warp PC will be fetched again to start additional redundant execution, and three copies' comparison will correct the error and resume the warp computation.

The idle SMs' aware Full-RISE requires a 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 a full redundant execution for its warps.

As can be seen in Fig. 14, the major hardware added to 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). We use CACTI [42] to analyze the area of the added storage structure under 32 nm technology. We estimate the logic area as $3\times$ of the storage. Thus Full-RISE creates about a 3% area overhead in 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, resulting in 1% area overhead for the SM.

4.2 PARTIAL-RISE

4.2.1 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 the 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 a large portion of SPs' idle time for reliability optimization because of its nature of performing the redundancy via using all SPs. In this section, we propose Partial-RISE, which intelligently combines redundant threads in a diverged warp to utilize the partially idled SPs, thus improving the SPs' error coverage and maintaining performance.

As described in Section 4.1, GPGPUs have a 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 redundant threads. Moreover, the warp will be issued in the following cycle so that the output for both the main and the redundant threads can be compared immediately for the error detection. We name this technique Partial-RISE. Fig. 15 shows an example. In Fig. 15A, at cycle *m*, the warps *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 being written to the registers. During the cycle $m + 1$ (shown in Fig. 15B), warp *j* is issued, and it outputs are compared to the data saved in the buffer in the previous cycle.

Finding the warp with the same PC is critical to Partial-RISE. We investigate various benchmarks, and Fig. 16 plots the percentage of warp divergence cycles that at least one ready warp has the identical PC with the current issued diverged warp (shown as the left bars). The round-robin policy is applied to the warp scheduling. The benchmarks with quite a few branch divergences are not shown in the Fig. 16 because the Partial-RISE is rarely triggered in that case. As the figure shows, more

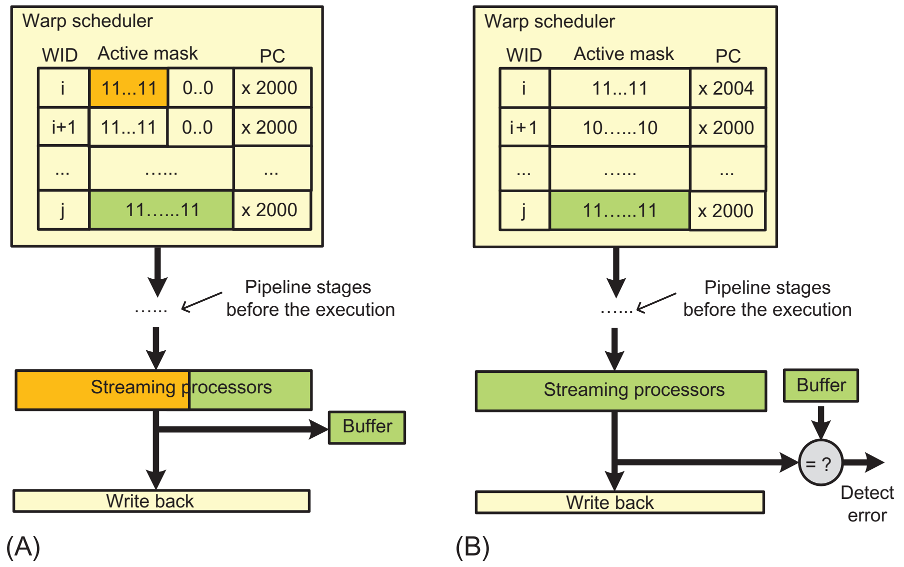


FIG. 15

An example of Partial-RISE. (A) warp *j* is issued together with warp *i*. (B) warp *j* is issued in the next cycle.

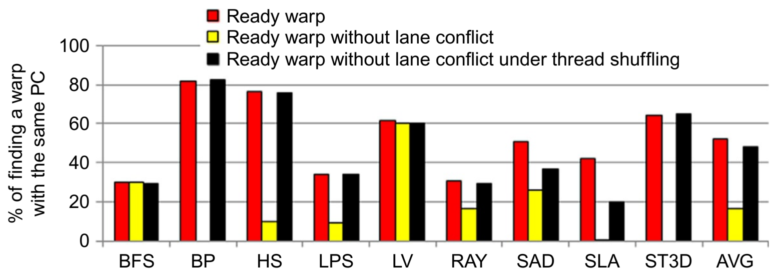


FIG. 16

Possibility of a diverged warp finding a warp with identical PC.

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 lane conflict. This becomes a major obstacle for Partial-RISE. As shown in Fig. 15A, although warp $i+1$ has the identical PC with warp i, Partial-RISE cannot be used because both of them have the same active mask. Fig. 16 also shows the possibility that a diverged warp can be combined with another warp without lane conflict (shown in the middle bars). As can be seen, the possibility decreases significantly down to 16%, and even becomes zero in some benchmarks (e.g., *BP*, *SLA*, *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, which will be performed in parallel with the kernel launch process and with no extra delay to the entire kernel execution. In this way, the possibility of lane conflicts between two warps decreases and Partial-RISE can be triggered more frequently. The black bar in Fig. 16 shows that rearranging threads successfully brings the possibility of finding a ready warp with the same PC back to 48%.

Several recent mechanisms (e.g., the DWF [24], the thread block compaction [22], and the large warp microarchitecture (LWM) [35]) are proposed to improve the efficiency of branch handling. For example, the LWM as an example, it implements larger warps so that the subwarps 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 was that Partial-RISE would have little benefit for reliability enhancement when LWM is applied in the GPU. This is not the case. A large warp contains a much smaller number of threads (e.g., 256) compared to the entire warp scheduler. It is unable to find active threads in the warp, and it does not provide a 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 has an effect 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 intrawarp duplication that leverages those idle lanes to provide the spatial redundancy for some threads contained in the warp.

4.2.2 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 becomes 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, with no impact on the critical path delay. If successful, 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 reuse the hardware (e.g., buffer, comparator) in Full-RISE to perform the error detection. In total, Partial-RISE adds an additional 1% area overhead to the SM equipped with Full-RISE.

4.3 RISE: PUTTING IT ALL TOGETHER

Because Full-RISE and Partial-RISE aim 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 warps for redundancy. We find that the Partial-RISE trigger time even increases by 2% when combined with Full-RISE. This happens because the row locality leads to similar redundancy cycles and consequently to similar progress among blocks in the same SM under Full-RISE. Thus the block progress difference generally happens at the SM level.

4.4 EVALUATIONS

4.4.1 Experimental setup

We use AVF [27] 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 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. [27] to identify the

ACE bits and their residency time in the structure and to compute the AVF of GPGPU microarchitecture structures. We build our vulnerability estimation framework based on the cycle-accurate, open-source, and publicly available simulator GPGPU-Sim [29] and to 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 16384 32-bit registers, 16 KB shared memory, 8 KB constant cache, and 64 KB texture cache; the warp scheduler applies the round-robin scheduling policy; the immediate postdominator reconvergence [23] is used to handle the branch divergences; the GPU includes 8 DRAM controllers, each controller has a 32-entry input buffer and applies OoO FR-FCFS scheduling policy; the interconnect topologies are 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 [31], Rodinia Benchmark [32], Parboil Benchmark [6], and some third-party applications. The workloads show significant diversity according to their kernel characteristics, divergence characteristics, memory access patterns, and so on.

4.4.2 Effectiveness of RISE

To better analyze the technique's effectiveness, we classify the benchmarks in 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 that 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 computation-intensive benchmarks such as *BS*, *CS*, *MM*, and *RAY*. Note that the preceding categories are not exclusive (i.e., one benchmark can be classified into different categories).

Fig. 17 shows (A) the 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 Fig. 17A as well for comparison. As can be seen, on average, full redundancy results in 58% performance degradation. Because the full redundancy achieves 100% error coverage (i.e., AVF is zero), its results are not shown in Fig. 17B. We present the averaged results across SMs, and the results are normalized to the baseline case without optimization. As Fig. 17 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, which 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

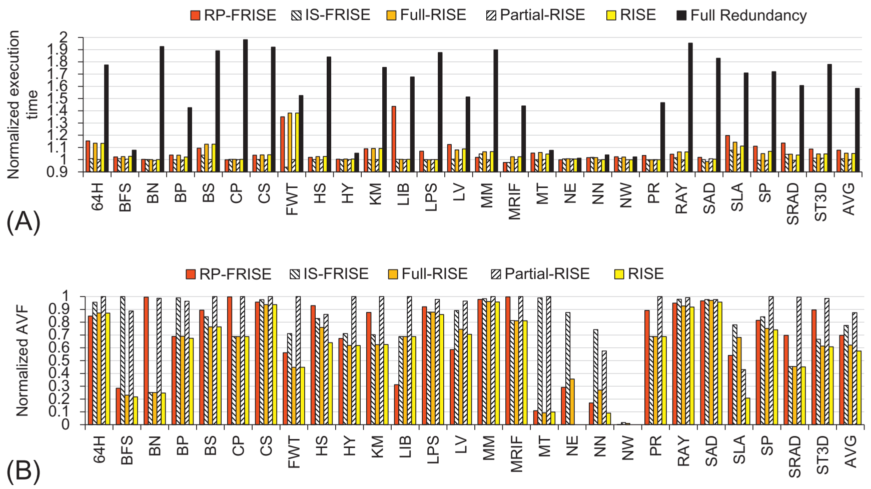


FIG. 17

The normalized (A) execution time and (B) SPs' AVF under IS-FRISE, RP-FRISE, Full-RISE, Partial-RISE, RISE, and full redundancy techniques.

less obvious in *PR*, although the warps spend 17% of the execution time waiting for the memory transactions. This is because their memory requests were already well separated during the execution phase. Also, the memory access latency is generally short, and the pipeline only stalls for a couple of cycles for the memory transaction. Postponing the warp progress would easily cause substantial damage to the performance; 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 to determine the corresponding redundancy cycles, its prediction accuracy is affected when the next memory access pattern differs greatly from the last one. As a result, excessive redundancy is applied, which hurts the performance. On average, across the 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 a certain SM. It is possible that an SM commits multiple blocks at similar times and that 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 until 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, but it induces a complicated hardware design that 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), which may not be an issue when configuring the machine differently, but another set of benchmarks may encounter this problem. Therefore IS-FRISE is applicable to various GPU architecture designs. As a combination of RP-FRISE and IS-FRISE, Full-RISE maintains their positive effects for substantially optimizing soft-error robustness, and more important, their interaction effectively mitigates the performance penalty caused by RP-FRISE. As Fig. 17 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 in the third category. As shown in Fig. 17, on average across benchmarks in the third category, Partial-RISE reduces SPs' AVF by 32% without any performance penalty. Finally, when putting it all together, RISE integrates the benefit of all the proposed techniques by achieving 43% of SPs' soft-error reliability enhancement and minimizes (only 4%) the performance loss. In the computation-intensive benchmarks, the effect of RISE is less impressive because of the SPs' limited idle time. Note that RISE optimizes the SPs' vulnerability via redundancy; the SPs' AVF does not migrate to any other microarchitecture structures.

4.4.3 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 a minimum number of instructions executed [34], FRFS, LWM, and two-level round-robin warp scheduling that effectively hides long memory access latency and improves the SPs utilization [35]. Fig. 18 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 Fig. 18 for comparison. As the figure shows, the effectiveness of RISE is not affected when running with different schemes; on average, the SPs' AVF reduction stays 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, so 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 to around 12%. As we described in Section 4.2.1, LWM finds only 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

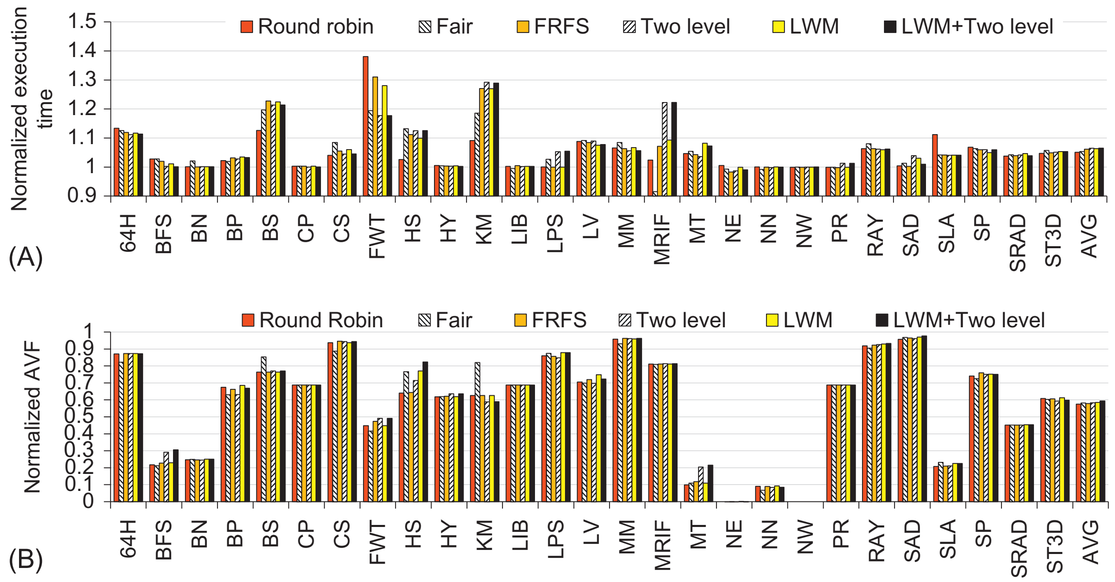


FIG. 18

The normalized (A) execution time and (B) SPs' AVF under various scheduling policies and performance optimizations.

the load imbalance across SMs; therefore, the Full-RISE in RISE can be frequently triggered to recycle the SPs' free time for reliability enhancement.

To conclude, in this section, 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.

5 MITIGATING THE SUSCEPTIBILITY OF GPGPUs TO PVs [43]

As process technology keeps scaling down, the increasing PV have become a growing threat to processor design and fabrication [44]. PV is the divergence of device parameters from their nominal values, which is caused by challenging manufacturing processes at very small feature technologies. PV induces delay variations among critical paths and causes timing errors. To ensure that processors run as expected, the maximum clock frequency (FMAX) has to be limited by the worst critical path delay, leading to substantial performance loss. For example, the chip frequency degrades as much as 22% in 45 nm process technology because of PVs [44]. And the frequency degradation becomes more significant with the continuous shrinking in feature size.

Unfortunately, the PV impact is exacerbated in modern GPGPUs. This is because FMAX is strongly related to the number of critical paths in a chip [45]. GPGPUs contain a tremendous amount of parallel critical paths to deliver high computing throughput. Thus the possibility that one path fails to meet the timing speculations increases substantially, which forces a severe decrease on FMAX in GPGPUs compared to that in CPUs [21,46]. The negative effect of PV on GPGPUs' frequency has recently motivated several architecture-level proposals for performance boosting under PV [21,46–52].

GPGPUs support a great number of parallel threads and implement a zero overhead context switch among threads to hide the long-latency operations. This requires an extremely large RF to keep the states and contents of all active threads. For instance, the register file size is 2 MB in NVIDIA Fermi [53] and 6 MB in AMD Cayman [54]. Such large register files include numerous parallel critical paths and are quite sensitive to PVs. Even worse, to afford a greater number of threads executing simultaneously in the SM, the register file size has continuously increased in recent GPUs' product generations [53,55]. Therefore register files have become one of the major units that affect frequency and performance [56], and it is crucial to mitigate the PV impact in them.

The unique, that is, highly banked, register architecture design in GPGPUs provides a promising direction to efficiently tolerate the PV effect. However, generic

PV mitigation techniques, such as adaptive body biasing (ABB) [57] and gate sizing [58], fail to exploit this unique feature and could cause considerable power and area overhead when directly applied to GPGPUs' register file. In this section, we propose to characterize and further leverage this highly banked architecture feature to effectively mitigate the susceptibility of GPGPUs' register file to PVs. Note that we assume other PV-sensitive structures, such as streaming processors, are handled by conventional PV tolerance mechanisms (e.g., ABB).

There have been several PV tolerant techniques explored to optimize the multiported register file in CPUs [59,60]. For instance, Liang et al. [59] proposed $n\%$ variable-latency RF ($n\%$ VL-RF) to put all registers' read ports in fast and slow categories. The slowest $(100 - n)\%$ ports are marked as slow and are accessed in two cycles. They are not considered in determining the frequency so that the chip frequency increases in the presence of PV. When a slow port is assigned to read a register, a port switching technique is triggered to switch to a fast port attached to the same register and to avoid the extra cycle delay. However, port switching is not applicable to modern GPGPUs' register file. This is because implementing multiple ports to a sizeable register file in GPGPUs is not practical, and the highly banked register architecture has become a widely accepted design to provide high register access bandwidth [29,61–63]. Although the VL technique can be simply extended to GPGPUs' registers by exploring the fast and slow registers based on their access delay to attain the optimal frequency improvement, such a fine-grain register classification causes extremely high IPC degradation. This is because multiple (e.g., 32) same-named registers are accessed simultaneously in GPGPUs, and the access latency increases when one slow register gets involved.

In this section, we first develop a novel fast and slow register classification mechanism to maximize the frequency improvement in the highly banked register architecture. We then exploit the unique features in GPGPU applications to intelligently tolerate the extra access delay to the slow registers.

The contributions of this work are as follows:

- We observe that PV exhibits much stronger systematic effects in the vertical direction than in the horizontal direction within each RF bank in the state-of-the-art GPGPU register file floor plan [61,62]. We then propose a coarse-grain register classification mechanism by vertically dividing each RF bank into subbanks, and applying the variable-latency technique at the subbank level (VL-SB). VL-SB is able to attain the same frequency improvement as the fine-grain classification at the register level.
- We further propose RF bank reorganization (RF-BRO) to virtually combine subbanks with the same speed type (i.e., fast and slow). Therefore the same-named registers in an RF bank entry share the uniform access delay, and the newly formed RF banks can be classified in fast and slow categories. We also show that the proposed VL-SB and RF-BRO techniques are applicable to other register file architecture designs [64,65].
- In order to mitigate the IPC loss caused by the slow RF banks' access under VL-SB + RF-BRO, we propose to grant warps that heavily use fast RF banks a

higher issue priority (in GPGPUs, threads are executed in warps). This forces fast RF banks to serve more threads and minimizes the use of slow RF banks, and also appropriately enlarges the progress difference among warps to effectively hide stalls caused by long-latency operations.

By combining all the explored techniques, we achieve 15% frequency improvement compared to the baseline case without optimization under PV, and 24% IPC improvement compared to the fine-grain register classification mechanism (i.e., VL-RF technique).

The rest of this section is organized as follows: Section 5.1 provides background on the highly banked register file in GPGPUs. Section 5.2 presents the register classification mechanism to achieve the most desirable frequency optimization. Section 5.3 proposes a set of techniques to hide the access delay overhead on slow registers. Section 5.4 evaluates the proposed mechanisms.

5.1 BACKGROUND: HIGHLY BANKED REGISTER FILE IN GPGPUs

In Nvidia PTX standard, an instruction can read up to four registers and write to one register. Therefore the register file in the SM is heavily banked (e.g., 16 or 32 banks) instead of multiported to provide high bandwidth, and multiple register operands required by one instruction can be read from different banks concurrently [1,29,61–65]. Each RF bank is equipped with dual ports to support one read and one write per cycle. Each entry in the bank is 128 bytes wide to hold 32 same-named registers [61,62]. During the register access, the RF bank ID is obtained based on the warp ID and register ID, and the port attached to that RF bank is activated to serve the access request.

Ideally, the register access for an instruction warp finishes in one cycle [61]. This is not the case when multiple register access requests map to the same bank and cause a bank conflict. In that case, requests have to be served sequentially, which extends the register access time to multiple cycles and hurts the performance. In order to reduce the possibility of bank conflicts, registers in a warp are distributed across the RF banks. Since multiple source operands for an instruction warp may not be read at the same cycle due to the bank conflicts, operand collectors are applied to buffer the operands. One instruction warp will be allocated one operand collector once issued by the scheduler. When all required operands are ready in their assigned operand collector, the instruction warp proceeds to the execution stage and releases its operand collector resources.

5.2 FREQUENCY OPTIMIZATION FOR GPGPUs’ REGISTER FILE UNDER PVs

5.2.1 Modeling PV impact on GPGPUs’ register file

PVs are a combination of random effects (e.g., due to random dopant fluctuations) and systematic effects (e.g., due to lithographic lens aberrations) that occur during transistor manufacturing. Random variations refer to random fluctuations in parameters from die to die and device to device. Systematic variations refer to layout-dependent variations that cause nearby devices to share similar parameters. Die-to-die (D2D) variations mainly exhibit as random variations, and within-die (WID) variations consist of both random and systematic variations. We focus on WID variations since D2D effect can be modeled as an offset value to all the devices in the chip. Among the design parameters, effective channel length (L_{eff}) and threshold voltage (V_{th}) are two key parameters subject to large variations [66]. The high V_{th} and L_{eff} variations cause high variations in transistor switching speed.

An architectural model of PVs, VARIUS [66], has been developed to quantitatively characterize the frequency variation in CPUs. In this study, we leverage the SRAM timing error model in VARIUS and modify it to model the PV effects on GPGPUs' register file. We focus on the 32 nm process technology that is generally used in the state-of-the-art GPUs [30]. We set the WID correlation distance coefficient ϕ as 0.5, and assume V_{th}'s $\sigma/\mu = 12\%$, L_{eff}'s $\sigma/\mu = 6\%$ [19,66], and the random and systematic components have equal variances for both V_{th} and L_{eff} [19,66,67]. Fig. 19 shows an example of V_{th} variation map for GPGPUs' register file. We generate 100 chips for statistical analysis and present the averaged result.

Note that each SM in GPGPUs exhibits different FMAX under PV. We model SM-to-SM variations as the ratio of frequencies of the fastest and the slowest SM in a GPGPUs' chip, and model the within-SM variations as the ratio of frequencies of the fastest and slowest critical path. Based on our experimental results, within-SM variations are 1.7, which is larger than SM-to-SM variations that are around 1.3. This is because each SM has numerous parallel critical paths, while there are only tens of SMs. SM-to-SM variations have been smoothed out as the FMAX of each SM is determined by the slowest critical path in it. Moreover, there are techniques letting each SM in GPGPUs run at their own FMAX for PV mitigation [46]. We thus perform the variability analysis within the SM. As previously mentioned, the register file is one of the major structures that limit the SM frequency, and we assume other PV-sensitive structures are handled by conventional PV tolerance mechanisms. Thus the SM frequency is determined by the register file frequency, which is the reciprocal of the slowest register access time. Fig. 20A demonstrates the register file frequency distribution over 100 chips. There are 15 SMs in the Fermi architecture, and we use the averaged register file frequency across SMs to represent the frequency for one chip. Therefore Fig. 20A mainly shows the impact of within-SM variations on frequency. As the figure shows, the mean frequency degradation is 40% compared to

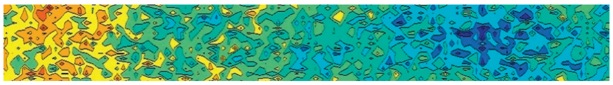


FIG. 19

V_{th} variation map of GPGPUs' register file.

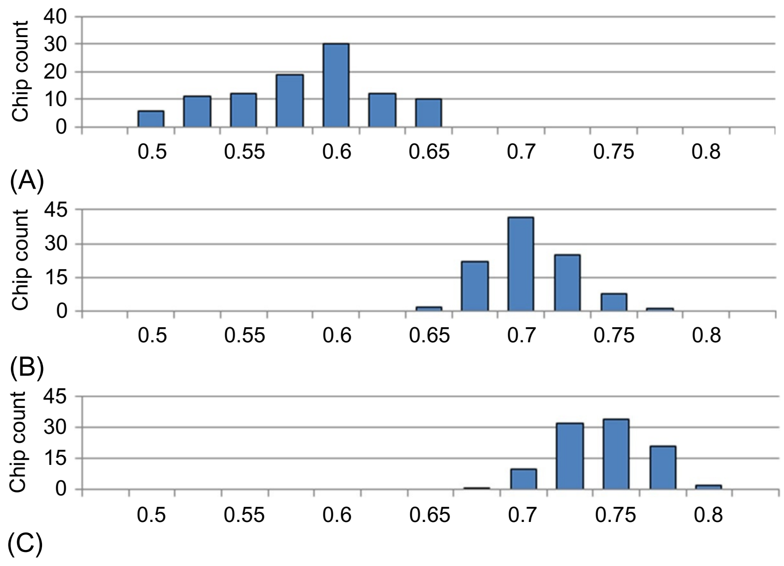


FIG. 20

Register file frequency distribution. (A) Frequency distribution (baseline without optimization). (B) Frequency distribution (70% VL register vectors). (C) Frequency distribution (70% VL sub-banks).

GPUs without PV. This is because numerous critical paths in GPGPUs' register file lead to the large within-SM variation, which significantly decreases the frequency.

5.2.2 *Variable-latency subbanks (VL-SB) in GPGPUs register file*

Extending VL-RF to GPGPUs RF

In our baseline RF design, each SM is equipped with a 128 KB register file that is composed of 32 KB 32-bit registers. In order to reduce the impact of the extremely slow register and boost the frequency under PV, one can apply the *n*% variable-latency RF technique [29] to divide those 32 KB registers contained in the SM into fast and slow categories depending on their access delay. Based on our sensitivity analysis, setting *n*% as 70% delivers the optimal trade-off between frequency and the amount of slow registers. We implement the 70% variable-latency register file (70% VL-RF); the slowest 30% registers are classified in the slow category and will take two cycles to finish the read/write operation; frequency is determined by the slowest register of the remaining 70% registers in the fast category.

However, the variable-latency RF causes serious IPC loss. Recall that one operand access in an instruction warp involves the parallel accesses to 32 same-named registers from all threads within the warp. As long as there is 1 slow register among those 32 registers, the operand access latency is 2 cycles. In our baseline RF design [61,62], although the 32 same-named registers are implemented close to each other and included in a single entry of the RF bank, the PV exhibits weak systematic effects for such a 1024-bit wide entry. As a result, most 32 same-named registers contain at least 1 slow register, and the operand access latency is generally extended. We observe 23% IPC degradation under 70% VL-RF compared to the baseline case without optimization under PV. Moreover, the variable-latency technique requires an extra bit per register to record the speed information as fast or slow, leading to high power and area overhead.

An alternative design to avoid the large IPC degradation is to consider each 32 same-named registers as a group, called register vector, and apply the 70% variable-latency technique at the register vector level, namely, 70% VL-RV. Similar to VL-RF, VL-RV requires an extra bit per register vector and causes considerable power and area overhead. More importantly, there is large delay variability among registers within a register vector, but the slowest one determines its access delay. In other words, the variations at register level are significantly smoothed out at the register vector level. Thus dividing register vectors into fast and slow categories has a limited effect on frequency optimization.

Variable-latency subbanks (VL-SB) in RF

In our baseline RF design [61,62], each RF bank holds sixty-four 1024-bit wide entries. Therefore PV exhibits much stronger systematic effects in the vertical direction than that in the horizontal direction within each RF bank. In this study, we focus on this wide-entry RF architecture containing 16 banks, and the explored techniques perfectly fit to other RF architecture, which is discussed in Section 5.2.4.

We propose *n*% variable-latency subbanks in GPGPUs' register file (named VL-SB) that vertically divides each RF bank into several subbanks, and registers within each subbank share the same access speed that is constrained by the slowest one. Subbanks exhibit distinct access delay, and the slowest $(100 - n)\%$ ones are marked as slow. There is small delay variability among registers contained in a subbank because of the systematic effects; therefore the large variations at register level are well maintained at the subbank level in VL-SB, which maximizes the frequency improvement under the VL technique.

Note that both read and write delay are considered in VL-SB. We observe that subbanks with long (short) read delay are highly likely to exhibit long (short) write delay under the impact of systematic variations. This makes the subbanks' classification quite straightforward, and leads to only two categories that are fast read + fast write (i.e., fast subbank) and slow read + slow write (i.e., slow subbank). We choose to divide each RF bank into two subbanks because further aggressively performing the fine-grain partition (i.e., four subbanks or more) does not lead to an obvious frequency increase based on our sensitivity analysis. As can be seen, only 32 bits are required to keep the speed information for the 32 subbanks in VL-SB, which causes negligible area overhead.

Since PV also causes delay variability among SMs in the same GPGPUs chip, one can change the value of *n*% during the subbanks' partition in different SMs to ensure the uniform frequency across SMs. In that case, each SM has a distinct number of fast subbanks that may affect the IPC. It is a good idea to employ the per-SM clocking as discussed in Ref. [46]; we thus keep the uniform partition criterion (i.e., 70%)

for each SM, and our techniques are orthogonal to the previously explored inter-SM level PV mitigation mechanisms [46].

Fig. 20B and C justifies the effectiveness of 70% VL register vectors and 70% VL subbanks by showing the RF frequency distribution when the two techniques are enabled, respectively. Every 32 nearby register vectors in VL-RV are grouped into an array to keep the same area overhead as VL-SB for a fair comparison. This makes 32 arrays in total, and the slowest one in the fastest 70% arrays decides the frequency. As Fig. 20B demonstrates, the mean frequency in 70% VL-RV increases 10% compared to the baseline case presented in Fig. 20A, while 70% VL-SB in Fig. 20C is able to boost the mean frequency by 15%.

Note that the structural redundancy technique [21], which adds redundant structures to the processor as spares, is not applicable to eliminate the slowest $x\%$ critical paths in GPGPUs' RF for frequency boosting. If just applying redundancy to replace the slowest $x\%$ register vectors, which may distribute across all the RF banks, the register mapping becomes extremely complicated and impractical. Applying redundancy to replace the slowest $x\%$ RF banks will cause considerable area overhead. Moreover, selective word-line voltage boosting [68], which was applied to reduce the cache line access latency under the PV effect, is not applicable to GPGPUs' RF. Since RF is accessed more frequently than caches, selective word-line voltage boosting will cause considerable energy overhead to GPGPUs' RF.

5.2.3 Register file bank reorganization

The VL-SB technique faces the same challenge as VL-RF, because it distributes registers belonging to the same register entry (i.e., register vector) to two subbanks, which may be classified in different categories. We further propose RF-BRO on top of VL-SB that virtually combines two subbanks from the same category to form a new RF bank. RF-BRO ensures the uniform access latency during the parallel accesses to 32 same-named registers. An operand access from an instruction warp is able to finish in one cycle as long as it is mapped to the newly formed RF bank that is composed of two fast subbanks.

Fig. 21 shows an example of applying 70% VL-SB with RF-BRO on GPGPUs RF. When VL-SB is applied (step I in Fig. 21), the 16 RF banks are vertically divided into 32 subbanks, and 22 of them are fast based on the 70% partition criterion (only 5 RF banks are shown in Fig. 21 for the illustration purpose). However, 10 RF banks still need two cycles to finish the read/write operation because they all contain slow subbanks. With the help of RF-BRO (step II in Fig. 21), subbanks with the same type are virtually grouped to rebuild the RF banks. For example, the new RF bank 0 is composed of subbank 0 and 5, and its access delay decreases to one cycle as it gets rid of the slow subbank 1. As a result, there are only five RF banks exhibiting a two-cycle access delay. In other words, the percentage of slow RF banks reduces from 62.5% to 31.25% under RF-BRO.

5.2.4 Implementation

We divide the word-line in each RF bank into two segments to obtain the subbanks. This is a widely used method in SRAM-based structures to reduce the delay because

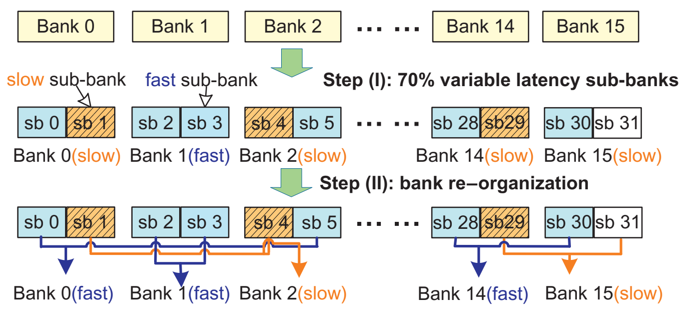


FIG. 21

An example of applying 70% VL-SB (step I) and RF-BRO (step II). *sb*, subbank.

of word-line [42] or to save dynamic power when a single word needs to be accessed in a large cache [69]. Each word-line segment in the subbank is equipped with a local decoder in our VL-SB.

The implementation of VL-SB is similar to that of the VL-RF technique [59]: the speed information for each RF subbank will be collected by using BSIT [70] at chip test time. This information is used to mark each subbank as fast or slow, and also to set an appropriate SM frequency. Note that the bank reorganization does not physically move any subbank during the chip fabrication. It virtually rebuilds the RF banks by introducing a 16-entry bank organization table; each entry in the table records the IDs of 2 subbanks that are assigned to the newly formed RF bank. In order to implement the bank reorganization technique, the IDs and the type (fast or slow) of every two same-type subbanks are configured into a ROM at the chip test time. They will be loaded from the ROM and written into the SRAM-based bank organization table once GPUs are powered on. Based on our gate-level modeling, the access latency of the bank organization table is negligible because of its small size, thus it does not increase the cycle time of the pipeline stage.

Fig. 22 depicts the implementation of the two proposed techniques. During an operand access to the 32 same-named registers, the RF bank ID obtained from the warp and register IDs is used to index the bank organization table and retrieve IDs and the speed type of corresponding subbanks. The same entry in those two subbanks will be activated simultaneously for operand access. For example, when RF bank 2 is accessed, subbanks 1 and 4 are enabled as shown in Fig. 22. Meanwhile, the speed type obtained from the table will be ANDed with a busy signal, and a slow type leads to a distribution of the signal to all subbanks. Only subbanks that are activated to fulfill this operand access will receive the busy signal and save it into the attached latch. This is used to prevent the precharge at the next cycle to ensure that the register read lasts for two cycles and finishes correctly. In GPGPUs RF, an arbitrator is applied to select a group of nonconflicting accesses and send it to the RF banks at every cycle [29,61,62]. Therefore the busy signal is also sent to the RF arbitrator to stall the following register read/write to the same RF bank.

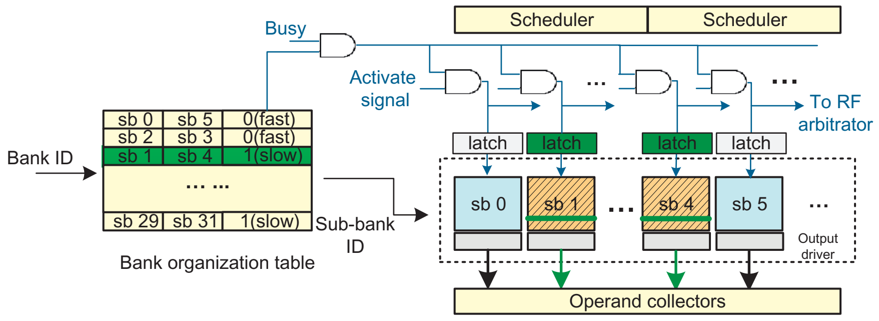


FIG. 22

Hardware implementation of variable-latency subbanks (VL-SB) and register file bank reorganization (RF-BRO).

5.2.5 Feasibility in alternative register file architecture

Alternative register file architectures are used in contemporary GPGPUs. One example is to group four SIMD lanes in an SM into a cluster, and eight clusters form a complete 32-wide SM [64,65]. In this case, each cluster contains four register banks, and each entry in a bank is only 16 bytes wide and contains the register values for four threads in a warp (i.e., four same-named registers). Thus 32 same-named registers are distributed into eight banks (one bank per cluster), and the same entries from eight RF banks are accessed simultaneously for one operand access. As can be seen, the systematic effects for those 32 same-named registers are weak since they are evenly distributed to different clusters. Neither the VL-RF technique nor the VL-RV technique could deliver good frequency improvement. We can consider this narrow-width style RF architecture as vertically dividing the 1024-bit wide register vectors (i.e., 32 same-named registers) into eight subbanks. Thus the proposed RF-BRO technique can be directly applied for the performance optimization under PV; we will not subdivide each register bank, instead we adopt the VL technique to those 16-byte wide banks, and virtually reorganize eight banks with the same speed to form the 32 same-named registers. In summary, our VL-SB RF design and the RF-BRO mechanism built upon it are applicable to other GPGPUs RF design.

5.3 MITIGATING THE IPC DEGRADATION UNDER VL-SB AND RF-BRO

Although the VL-SB + RF-BRO technique explored in Section 5.3 largely optimizes the GPGPUs RF frequency under PV impacts, there are still about 30% slow banks among the virtually reorganized RF banks, leading to around 9% IPC degradation based on our experimental results shown in Section 5.4. We further propose a set of techniques that harness the unique characteristics in GPGPUs' applications to minimize the IPC loss. Note that the slow (fast) RF banks mentioned in this section are RF banks that are virtually composed of two slow (fast) subbanks under RF-BRO.

5.3.1 Register mapping under VL-SB and RF-BRO

The SM in Fermi-style architecture is armed with two warp schedulers. Warps with odd and even IDs are dispatched into those two schedulers, respectively. At every cycle, two instruction warps are issued based on the round-robin scheduling policy, and they are likely to have identical PC since all threads in a kernel execute the same code. Mapping the same-ID registers from different warps into the same bank seriously exacerbates the bank conflicts, because different entries within a bank may be requested by the two simultaneously issued instruction warps. Therefore the register-to-bank mapping mechanism follows Eq. (2)

$$\text{bank\_ID} = (\text{warp\_ ID} + \text{register\_ ID})\% \quad (\text{the number of banks}) \tag{2}$$

to ensure that different banks hold the same-ID registers across the warps. As Eq. (2) shows, consecutive warps tend to map their same-ID registers into nearby RF banks. For instance, R1 from warp0 and warp1 are mapped to bank 1 and bank 2, respectively. Generally, consecutive warps exhibit strong data locality [35], so their same-ID registers should be allocated to RF banks with the same speed type to ensure that they execute at a similar progress. We propose to save IDs of same-type subbanks into consecutive entries in the ROM at the chip test time; therefore bank0–bank10 are fast and bank11–bank15 are slow in the bank organization table under VL-SB and RF-BRO techniques. By using Eq. (2), there are a number of registers per warp mapping to the slow RF banks. And the slow bank keeps registers with different IDs at the warp level. Fig. 23 demonstrates an example of register mapping: R11–R15 from warp0 while R0–R4 from warp11 are assigned to the slow RF bank11–bank15.

5.3.2 Fast-bank aware register mapping

It has been observed that around 50% of registers are not even allocated by the compiler for the application execution [63]. This unique feature can be leveraged to minimize the use of slow RF banks by mapping registers to fast banks to the maximum degree during the kernel launch time. For the benchmarks that have high RF utilization, slow banks are used to ensure high level TLP. We find that a small set of registers have much higher access frequency than other registers allocated to the same warp, and they are usually the registers with small IDs. For instance, each warp in benchmark BN (detailed experiment methodologies are in Section 5.4) is assigned 14 register vectors, and R0–R2 are used 250% more frequently than R3–R13. This is because the compiler tends to reuse the small-ID registers. We explore a novel fast-bank aware register mapping mechanism (named FBA-RM) that consists of two steps: (1) obtaining the register resource requirements at the kernel launch time, and mapping registers only to fast banks if they are large enough to hold all registers needed by the parallel threads (i.e., reducing the number of RF banks to 11 in Eq. (1) during the mapping); (2) allocating the large-ID registers to slow RF banks when they have to be used, which means that the slow banks are rarely accessed. In

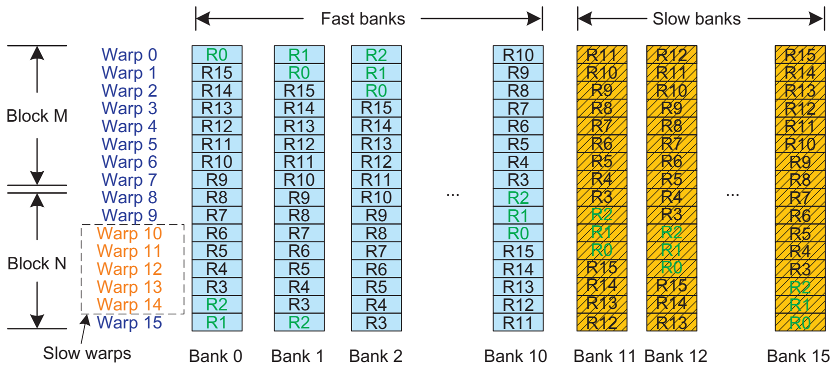


FIG. 23

The idea of FWAS. R0–R2 are frequently accessed small-ID registers. Warp 10–14 in block N are slow warps.

that case, Eq. (1) is used for small-ID registers to fast banks mapping and large-ID registers to slow banks mapping, respectively.

The major disadvantage of the FBA-RM technique is the increased bank conflicts because most RF accesses are limited to fast banks, and the technique fails to effectively mitigate the IPC degradation. In Section 5.4, we perform the detailed evaluation about FBA-RM by comparing it to other proposed techniques in the following sections.

5.3.3 *Fast-warp aware scheduling (FWAS) policy*

Considering that modifying the register mapping mechanism to minimize the use of slow banks has little impact on performance optimization, we thus adopt the default mapping mechanism in Fermi and propose a set of methods to hide the extra access delay on slow banks.

As Fig. 23 shows, the frequently accessed small-ID registers in a number of warps (e.g., warp 11) are mapped to slow RF banks, which seriously delays their execution progress. And we define this kind of warp as slow warp. We further define warps whose frequent register accesses in fast banks as fast warps (e.g., warp 0). The execution progress for fast warps is delayed somehow when the default warp scheduling policy (i.e., round-robin) is applied. This is because the round-robin policy gives each warp the same issue priority, and when the slow warps occupy the pipeline resources (e.g., the issue and write slot), the ready fast warps cannot leverage those resources for execution. As a result, there is a small progress difference between fast and slow warps within the same SM.

We propose the fast-warp aware scheduling policy (named FWAS) that assigns fast warps higher issue priority than slow warps to maximize the progress difference between them. This explores the unique opportunities to mitigate the IPC loss as follows:

(1) The fast warps have a shorter execution time, thus the RF resources allocated to these warps are able to serve more warps during execution. This is effective for kernels including a large number of blocks that cannot be fully distributed to SMs at one time.

Fig. 23 explains this opportunity in detail. Generally, there are multiple blocks executing concurrently in an SM. Fig. 23 shows an example SM with two blocks: block M and block N, each block contains eight warps. The frequently accessed registers (i.e., R0–R2) of all warps in block M (i.e., warp 0–7) are mapped to fast banks, so all warps belonging to block M are fast warps. On the contrary, in block N, most of their frequently accessed registers in warp 10–14 are mapped to slow banks, thus these warps are considered as slow warps. During program execution, FWAS prioritizes fast warps and allows them to finish earlier. As a result, warp 10–14 will left behind and become the bottleneck for block N. When all warps in block M finish execution, the incoming block within the same kernel will be assigned to take the resources (e.g., warp slots, registers) just released by block M. It also contains more fast warps (i.e., use those fast banks) because its warps will be assigned the same warp IDs as those warps in block M. On the other hand, block N will not release its resource until all its slow warps finish execution. As a result, the number of blocks that contain a larger amount of fast warps increases (in other words, fast banks are able to serve more warps) during the entire kernel execution, leading to the IPC improvement.

(2) The fast warps are able to start their off-chip memory accesses earlier, which alleviates the memory contention under the round-robin policy and reduces the pipeline stall time. This is quite effective in memory-intensive benchmarks.

In order to implement the FWAS policy, the fast warps have to be identified at the issue stage. However, there is no clear boundary between fast and slow warps, because the frequently accessed registers in a few warps are allocated to both fast and slow banks under the default mapping mechanism, for example, the heavily used R0–R2 in warp9–warp10 in Fig. 23. Although slow warps also access fast banks, it is highly possible that an instruction with fast bank access belongs to a fast warp. Instead of performing the accurate fast and slow warp identification, we choose to simply give the instruction warp that requires fast bank accesses higher issue priority. At the decode stage an instruction warp is marked as fast if it has fast RF read/write. Note that only when all its operands reads are mapped to fast RF banks is an instruction warp considered as possessing fast RF read. The one-bit fast/slow information combined with the ready bit is sent to the selection logic during the issue stage to perform the FWAS policy.

5.4 EVALUATIONS

We use a cycle-accurate, open-source, and publicly available simulator GPGPU-Sim (v3.1.0) [29] to evaluate the IPC optimization under our proposed methodologies. Note that our 70% variable-latency technique causes the frequency variations among

SMs, and we model that SM level frequency difference into the simulator as well. Our baseline GPGPU configuration models the Nvidia Fermi-style architecture: the GPU contains 15 SMs; the warp size is 32; each SM supports 1536 threads and 8 blocks at most; each SM contains 128 KB registers, 16 KB L1 data cache, and 48 KB shared memory; L2 cache size is 768 KB; the scheduler applies the round-robin among ready warps scheduling policy. The experimental methodology to model the PV impact and evaluate the frequency optimization under various variable-latency techniques is described in Section 5.2.1.

We collect a large set of GPGPU workloads from Nvidia CUDA SDK [31], Rodinia Benchmark [32], and Parboil Benchmark [6]. The benchmarks show significant diversity according to their kernel characteristics, divergence characteristics, memory access patterns, and so on.

5.4.1 IPC improvement

Evaluation of VL-SB with RF-BRO

To evaluate the effectiveness of VL-SBs with RF-BRO, we compare it to the VL-RF technique explored by Liang et al. [59]. Fig. 24 shows the IPC of the investigated benchmarks when those two PV mitigation techniques are enabled. The results are normalized to the baseline case without optimization (i.e., frequency is determined by the slowest critical path). As discussed in Section 5.3.1, bank0–bank10 are always fast while bank11–bank15 are always slow in all chips under RF-BRO. The IPC results are identical for all chips and only one chip's IPC results for RF-BRO-based techniques are shown in Figs. 24 and 25. The averaged IPC results across all 100 chips are shown for 70% of the VL-RF technique. In the baseline case, all registers have one-cycle access latency, and it has the same IPC as the ideal case without PV. Note that the 70% variable-latency register vector (VL-RV) technique discussed in Section 5.2.2 can also partition the banks into fast and slow subbanks, thus achieving the same IPC results as VL-SB + RF-BRO. However, the frequency improvement of VL-RV is lower than VL-SB + RF-BRO. Therefore VL-RV is not included in Fig. 24. As the figure shows, when compared with the VL-RF mechanism, VL-SB + RF-BRO successfully reduces the IPC loss from 23% to 9% on average across all the benchmarks because it reorganizes subbanks to deliver a considerable number of fast RF banks. VL-RF focuses at a quite fine-grain register-level classification, making it

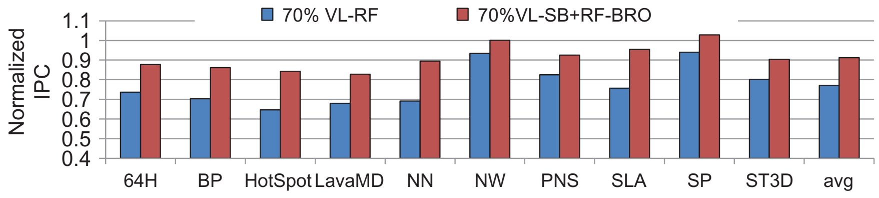


FIG. 24

Normalized IPC results under 70% VL-RF and 70% VL-SB + RF-BRO.

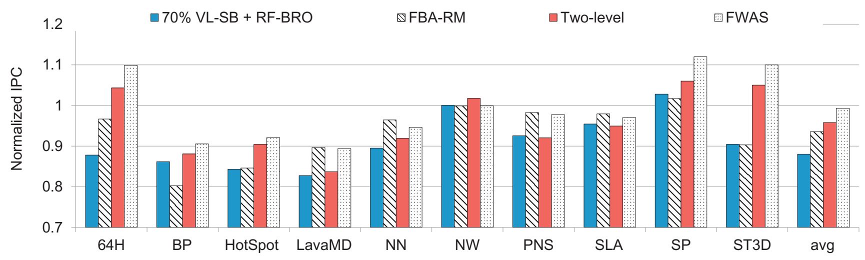


FIG. 25

Normalized IPC results under FBA-RM, two-level, and FWAS.

impossible to apply the RF-BRO for fast RF bank reorganization, and almost all the register vectors accesses take two cycles.

Interestingly, the IPCs of benchmark NW under both VL-RF and VL-SB + RF-BRO are already approaching 1. This is because NW includes very few threads, and there are insufficient warps running concurrently in the SM to hide the stalls for true data dependencies between consecutive instructions from a single warp (in absence of the long memory operations). As a result, the extra register access time is well absorbed by those stall cycles and has little impact on the IPC. On the other hand, the IPC decreases considerably in several computation-intensive benchmarks under VL-SB + RF-BRO, because few stall cycles are helping to hide long RF access delays. For instance, the IPC reduction is 18% in benchmark LavaMD that makes SM active 80% of the total execution time.

Evaluation of FWAS

Fig. 25 presents the normalized IPC results when the fast-warp aware scheduling policy (FWAS) is enabled. We compare FWAS with the fast-bank aware register mapping (named FBA-RM) discussed in Section 5.3.2. A two-level scheduling policy was proposed to boost performance [35]. This policy splits warps into groups and triggers the round-robin warp scheduling policy at intragroup and intergroup levels, respectively. By doing this, each group's warps reach a long-latency instruction at different points in time, which can also alleviate the memory contentions and tolerate the latency. We thus introduce the two-level policy into VL-SB + RF-BRO and compare it with our proposed FWAS policy. As shown in Fig. 25, all three techniques are able to mitigate the IPC reduction compared to the VL-SB+RF-BRO technique; the IPC losses under FBA-RM, two-level, and FWAS are 7%, 5%, and 1%, respectively.

The effectiveness of FBA-RM is highly related to the register utilization. For instance, NN uses less than 10% of the register file through the entire execution. The fast RF banks are by far enough to support the requirement. Moreover, since very few registers are utilized per warp in NN, pushing them to the fast banks negligibly increases the bank conflicts. As a result, the IPC loss of NN decreases from 11% under VL-SB + RF-BRO to only 4% under FBA-RM. Note that VL-SB + RF-BRO applies the default register mapping mechanism, so the slow RF banks have the same utilization as the fast ones no matter how many registers are needed, leading to a considerable IPC loss for NN. On the other hand, the performance penalty is severely exacerbated when benchmarks need more register resources. Take HotSpot as an example, it requires around 80% of the register file. When FBA-RM allocates the frequently used register to fast banks, the negative impact caused by the increased bank conflicts outweighs the positive effect of the decreased slow banks accesses, and results in a 16% IPC degradation.

As Fig. 25 shows, the two-level technique integrated with VL-SB + RF-BRO can effectively improve IPC on multiple memory-intensive benchmarks, such as *64H* and *ST3D*, because it decreases stall cycles caused by long-latency memory operations. For example, the IPC of *ST3D* is even 1% higher than that in the baseline case. As an exception, its effect on PNS is quite limited. The group size is fixed (i.e., eight warps) in the two-level policy, and there are only eight warps—that is, one group—in most SMs when executing PNS. The intergroup level round-robin in the two-level is inactive.

Similar to the two-level technique, FWAS shows the strong capability to mitigate the IPC loss for memory-intensive benchmarks. Moreover, FWAS does not have any constraint on warp grouping, and it successfully mitigates the IPC loss of PNS to only 2%, which is far better than that under the two-level technique. On average, across the memory-intensive benchmarks (i.e., *64H*, *NW*, *PNS*, and *ST3D*), FWAS boosts the IPC to 105% when normalized to the baseline case, which implies a 12% performance improvement compared to VL-SB + RF-BRO. Additionally, FWAS can effectively optimize the IPC for computation-intensive benchmarks, especially those including numerous blocks, because it makes the fast RF banks support warps at the utmost degree. For instance, it induces 8% IPC gains for both HotSpot and LavaMD compared to VL-SB + RF-BRO.

5.4.2 Overall performance improvement

Fig. 26 compares the overall performance (IPC × frequency) under the baseline case without optimization, 70% VL-RF, and our FWAS mechanism by presenting the performance distribution over the 100 investigated chips, respectively. The performance is normalized to the ideal case without PV impacts. As the figure shows, FWAS significantly improves the mean performance by 15% over the baseline case; 70% VL-RF achieves slightly higher frequency (i.e., 2%) than our FWAS technique because it performs at a fine-grain register-level; however, our technique outperforms the 70% VL-RF by 17% because of the substantial (i.e., 24%) IPC improvement.

To conclude, this section aims to mitigate the susceptibility of GPGPUs' register file to PV. We first propose to vertically divide RF banks into subbanks and explore the variable-latency technique at a subbank level to perform the coarse-grain register classification, which maximizes the frequency optimization under PV. We then propose to reorganize the RF banks by virtually combining the same-type subbanks,

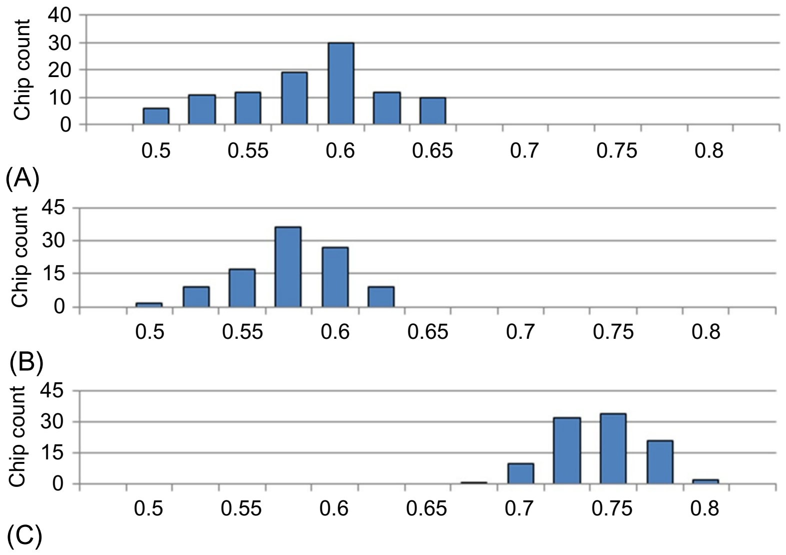


FIG. 26

Overall performance (IPC × frequency) distribution. (A) Overall performance distribution (baseline). (B) Overall performance distribution (70% VL-RF). (C) Overall performance distribution (all-together).

leading to the fast and slow RF bank categories. The IPC degrades when using the slow RF banks during the kernel execution. We further explore the FWAS technique that leverages the unique features in GPGPU applications to minimize the use of slow RF banks and thus mitigate the IPC loss. Our PV mitigation techniques achieves 15% overall performance improvement compared to the baseline case without optimization.

REFERENCES

[1] NVIDIA CUDA Programming Guide Version 3.0, 2010.
[2] Advanced Micro Devices Inc., AMD Brook+, 2016, 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] http://www.top500.org/lists/2015/11/.
[5] J. Sheaffer, D. Luebke, K. Skadron, A hardware redundancy and recovery mechanism for reliable scientific computation on graphics processors, in: Proceedings of Graphics Hardware, 2007.
[6] Parboil Benchmark suite, 2016, http://impact.crhc.illinois.edu/ parboil.php.

[7] N. Wang, S. Patel, ReStore: symptom based soft error detection in microprocessors, in: Proceedings of DSN, 2005.

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

[9] I. Haque, V. Pande, Hard data on soft errors: a large-scale assessment of real-world error rates in GPGPU, in: Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing, IEEE Computer Society, 2010, pp. 691–696.

[10] N. Maruyama, A. Nukada, S. Matsuoka, A high-performance fault tolerant software framework for memory on commodity GPUS, in: 2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS), IEEE, pp. 1–12.

[11] N. Maruyama, A. Nukada, S. Matsuoka, Software-based ECC for GPUs, in: 2009 Symposium on Application Accelerators in High Performance Computing (SAAHPC'09), 2009.

[12] J. Sheaffer, D. Luebke, K. Skadron, A hardware redundancy and recovery mechanism for reliable scientific computation on graphics processors, in: Proceedings of the 22nd ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware, Eurographics Association, 2007, pp. 55–64.

[13] G. Shi, J. Enos, M. Showerman, V. Kindratenko, On testing GPU memory for hard and soft errors, in: Proceedings of Symposium on Application Accelerators in High-Performance Computing, 2009.

[14] K. Yim, R. Iyer, Hauberk: lightweight silent data corruption error detectors for GPGPU, in: Proceedings of the 17th Humantech Thesis Prize (also in IPDPS 2011), 2011.

[15] C. Ding, C. Karlsson, H. Liu, T. Davies, Z. Chen, Matrix multiplication on GPUs with on-line fault tolerance, in: Proceedings of the 9th IEEE International Symposium on Parallel and Distributed Processing with Applications (ISPA 2011), IEEE Computer Society Press, 2011.

[16] K. Yim, R. Iyer, Hauberk: lightweight silent data corruption error detectors for GPGPU, in: Proceedings of the 17th Humantech Thesis Prize (also in IPDPS 2011), 2011.

[17] S. Dighe, S. Vangal, P. Aseron, S. Kumar, T. Jacob, K. Bowman, J. Howard, J. Tschanz, V. Erraguntla, N. Borkar, V. De, S. Borkar, Within-die variation-aware dynamic-voltage-frequency-scaling with optimal core allocation and thread hopping for the 80-core TeraFLOPS processor, J. Solid-State Circuits 46 (1) (2011) 184–193.

[18] R. Teodorescu, J. Torrellas, Variation-aware application scheduling and power management for chip multiprocessors, in: 35th International Symposium on Computer Architecture, 2008 (ISCA'08), June 21–25, 2008, pp. 363–374.

[19] U.R. Karpuzcu, K.B. Kolluru, N.S. Kim, J. Torrellas, VARIUS-NTV: a microarchitectural model to capture the increased sensitivity of many-cores to process variations at near-threshold voltages, in: Proceedings of the 2012 42nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN) (DSN'12), 2012, pp. 1–11.

[20] J. Lee, P. Ajgaonkar, N.S. Kim, Analyzing performance impact of process variations on GPGPU throughput, in: Proceedings of ISPASS, 2011.

[21] S. Seo, R.G. Dreslinski, M. Woh, Y. Park, C. Charkrabari, S. Mahlke, D. Blaauw, T. Mudge, Process variation in near-threshold wide SIMD architectures, in: Proceedings of DAC, 2012.

[22] W.W.L. Fung, T. Aamodt, Thread block compaction for efficient SIMT control flow, in: Proceedings of HPCA, 2011.

[23] S.S. Muchnick, Advanced Compiler Design and Implementation, Morgan Kaufmann, San Francisco, CA, USA, 1997.
[24] W.W.L. Fung, I. Sham, G. Yuan, T.M. Aamodt, Dynamic warp formation and scheduling for efficient GPU control flow, in: Proceedings of MICRO, 2007.
[25] J. Tan, Y. Yi, F. Shen, X. Fu, Modeling and characterizing GPGPU reliability in the presence of soft errors, J. Parallel Comput. 39 (9) (2013) 520–532.
[26] N. Soundararajan, A. Parashar, A. Sivasubramaniam, Mechanisms for bounding vulnerabilities of processor structures, in: Proceedings of ISCA, 2007.
[27] S.S. Mukherjee, C. Weaver, J. Emer, S.K. Reinhardt, T. Austin, A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor, in: Proceedings of MICRO, 2003.
[28] X. Fu, T. Li, J. Fortes, Sim-SODA: a unified framework for architectural level software reliability analysis, in: Workshop on Modeling, Benchmarking and Simulation, 2006.
[29] A. Bakhoda, G.L. Yuan, W.W.L. Fung, H. Wong, T.M. Aamodt, Analyzing CUDA workloads using a detailed GPU simulator, in: Proceedings of ISPASS, 2009.
[30] A. Biswas, R. Cheveresan, J. Emer, S.S. Mukherjee, P.B. Racunas, R. Rangan, Computing architectural vulnerability factors for address-based structures, in: Proceedings of ISCA, 2005.
[31] http://www.nvidia.com/object/cuda_sdks.html.
[32] S. Che, M. Boyer, J. Meng, D. Tarjan, J. Sheaffer, S. Lee, K. Skadron, Rodinia: a benchmark suite for heterogeneous computing, in: Proceedings of IISWC, 2009.
[33] B. Fahs, S. Bose, M. Crum, B. Slechta, F. Spadini, T. Tung, S.J. Patel, S.S. Lumetta, Performance characterization of a hardware mechanism for dynamic optimization, in: Proceedings of MICRO, 2001.
[34] N.B. Lakshminarayana, H. Kim, Effect of instruction fetch and memory scheduling on GPU performance, in: Workshop on Language, Compiler, and Architecture Support for GPGPU, 2010.
[35] V. Narasiman, M. Shebanow, C.J. Lee, R. Miftakhutdinov, O. Mutlu, Y.N. Patt, Improving GPU performance via large warps and two-level warp scheduling, in: Proceedings of MICRO, 2011.
[36] J. Tan, X. Fu, RISE: improving streaming processors reliability against soft errors in GPGPUs, in: International Conference on Parallel Architectures and Compilation Techniques (PACT), September, 2012.
[37] S.K. Reinhardt, S.S. Mukherjee, Transient fault detection via simultaneous multithreading, in: Proceedings of ISCA, 2000.
[38] J. Wadden, L. Alexander, G. Sudhanva, S. Vilas, S. Kevin, Real-world design and evaluation of compiler-managed GPU redundant multithreading, in: Proceeding of the 41st Annual International Symposium on Computer Architecture (ISCA), 2014.
[39] M.A. Gomaa, T.N. Vijaykumar, Opportunistic transient-fault detection, in: Proceedings of ISCA, 2005.
[40] S.S. Mukherjee, J. Emer, S.K. Reinhardt, The soft error problem: an architectural perspective, in: Proceedings of HPCA, 2005.
[41] D. Kirk, W.W. Hwu, Programming Massively Parallel Processors: A Hands-on Approach, in: (1st ed.) Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2010.
[42] N. Muralimanohar, R. Balasubramonian, N.P. Jouppi, Cacti 6.0: a tool to understand large caches, Technical report, University of Utah and Hewlett Packard Laboratories,2007.

[43] J. Tan, X. Fu, Mitigating the susceptibility of GPGPUs register file to process variations, in: International Parallel & Distributed Processing Symposium (IPDPS), May, 2015.

[44] S. Sarangi, B. Greskamp, A. Tiwari, J. Torrellas, EVAL: utilizing processors with variation-induced timing errors, in: 2008 41st IEEE/ACM International Symposium on Microarchitecture, November 8–12, 2008, pp. 423–434.

[45] N.S. Kim, T. Kgil, K. Bowman, V. De, T. Mudge, Total power optimal pipelining and parallel processing under process variations in nanometer technology, in: IEEE/ACM International Conference on Computer-Aided Design, 2005 (ICCAD-2005), November 6–10, 2005, pp. 535–540.

[46] J. Lee, P. Ajgaonkar, N.S. Kim, Analyzing throughput of GPGPUs exploiting within-die core-to-core frequency variation, in: Proceedings of ISPASS, 2011.

[47] E. Krimer, P. Chiang, M. Erez, Lane decoupling for improving the timing-error resiliency of wide-SIMD architectures, in: Proceedings of the 39th Annual International Symposium on Computer Architecture (ISCA'12), 2012, pp. 237–248.

[48] E. Krimer, R. Pawlowski, M. Erez, P. Chiang, Synctium: a near-threshold stream processor for energy-constrained parallel applications, in: IEEE Computer Architecture Letters, 9 (1), 2010, pp. 21–24.

[49] R. Pawlowski, E. Krimer, J. Crop, J. Postman, N. Moezzi-Madani, M. Erez, P. Chiang, A 530 mV 10-lane SIMD processor with variation resiliency in 45 nm SOI, in: 2012 IEEE International Solid-State Circuits Conference, February 19–23, 2012, pp. 492–494.

[50] A. Rahimi, L. Benini, R.K. Gupta, Spatial memoization: concurrent instruction reuse to correct timing errors in SIMD architectures, IEEE Trans. Circuits Syst. Express Briefs 60 (12) (2013) 847–851.

[51] A. Rahimi, L. Benini, R.K. Gupta, Hierarchically focused guardbanding: an adaptive approach to mitigate PVT variations and aging, in: Proceedings of the Conference on Design, Automation and Test in Europe (DATE'13), 2013, pp. 1695–1700.

[52] P. Aguilera, J. Lee, A. Farmahini-Farahani, K. Morrow, M. Schulte, N.S. Kim, Process variation-aware workload partitioning algorithms for GPUs supporting spatial-multi-tasking, in: Proceedings of the conference on Design, Automation & Test in Europe (DATE'14), 2014.

[53] NVIDIA's Next Generation CUDA Compute Architecture: Fermi, 2016, http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf.

[54] D. Kanter, AMD's Cayman GPU architecture, 2010, http://realworldtech.com/page.cfm?ArticleID=RWT121410213827.

[55] NVIDIA's Next Generation CUDA Compute Architecture: KeplerGK110, 2016, http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf.

[56] N. Goswami, B. Cao, T. Li, Power-performance co-optimization of throughput core architecture using resistive memory, in: 2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA2013), February 23–27, 2013, pp. 342–353.

[57] J. Tschanz, J. Kao, S. Narendra, Adaptive body bias for reducing impacts of die–to-die and within-die parameter variations on microprocessor frequency and leakage, J. Solid-State Circuits 37 (11) (2002) 1396–1402.

[58] A. Agarwal, B.C. Paul, H. Mahmoodi, A. Datta, K. Roy, A process-tolerant cache architecture for improved yield in nanoscale technologies, IEEE Trans. Very Large Scale Integr. Syst. 13 (1) (2005) 27–38.

[59] X. Liang, D. Brooks, Mitigating the impact of process variations on processor register files and execution units, in: 2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06), 2006, pp. 504–514.

[60] R. Teodorescu, J. Nakano, A. Tiwari, J. Torrellas, Mitigating parameter variation with dynamic fine-gain body biasing, in: 40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2007), December 1–5, 2007, pp. 27–42.

[61] Y. Lu, S. Ganapathy, Z. Mao, M.I. Guo, R. Canal, N. Jing, Y. Shen, X. Liang, An energy-efficient and scalable eDRAM-based register file architecture for GPGPU, in: Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA'13), 2013, pp. 344–355.

[62] J. Leng, T. Hetherington, A. ElTantawy, S. Gilani, N.S. Kim, T.M. Aamodt, V.J. Reddi, GPUWattch: enabling energy optimizations in GPGPUs, in: Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA'13), 2013, pp. 487–498.

[63] M. Abdel-Majeed, M. Annavaram, Warped register file: a power efficient register file for GPGPUs, in: Proceedings of the 2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA) (HPCA'13), 2013, pp. 412–423.

[64] M. Gebhart, D.R. Johnson, D. Tarjan, S.W. Keckler, W.J. Dally, E. Lindholm, K. Skadron, Energy-efficient mechanisms for managing thread context in throughput processors, in: Proceedings of the 38th Annual International Symposium on Computer Architecture (ISCA'11), 2011, pp. 235–246.

[65] M. Gebhart, S.W. Keckler, W.J. Dally, A compile-time managed multi-level register file hierarchy, in: Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-44), 2011, pp. 465–476.

[66] S. Sarangi, B. Greskamp, R. Teodorescu, J. Nakano, A. Tiwari, J. Torrellas, VARIUS: a model of process variation and resulting timing errors for microarchitects, Trans. Semicond. Manuf. 21 (1) (2008) 3–13.

[67] A. Agrawal, A. Ansari, J. Torrellas, Mosaic: exploiting the spatial locality of process variation to reduce refresh energy in on chip eDRAM modules, in: International Symposium on High Performance Computer Architecture (20th HPCA), Minneapolis, 2014, pp. 84–95.

[68] Y. Pan, J. Kong, S. Ozdemir, G. Memik, S.W. Chung, Selective wordline voltage boosting for caches to manage yield under process variations, in: Proceedings of the 46th Annual Design Automation Conference (DAC'09), 2009, pp. 57–62.

[69] M. Yoshimoto, K. Anami, H. Shinohara, T. Yoshihara, H. Takagi, S. Nagao, S. Kayano, T. Nakano, A divided word-line structure in the static RAM and its application to a 64K full CMOS RAM, IEEE J. Solid-State Circuits 18 (5) (1983) 479–485.

[70] M. Tehranipour, Z. Navabi, S. Falkhrai, An efficient BIST method for testing of embedded SRAMs, in: The 2001 IEEE International Symposium on Circuits and Systems, 2001 (ISCAS 2001), vol. 5, 2001, pp. 73–76.