2012 IEEE/ACM 45th Annual International Symposium on Microarchitecture

Warped-DMR: Light-weight Error Detection for GPGPU

Hyeran Jeon Murali Annavaram
University of Southern California
{hyeranje,annavara}@usc.edu

Abstract

General purpose graphics processing units (GPGPUs) are feature rich GPUs that provide general purpose computing ability with massive number of parallel threads. The massive parallelism combined with programmability made GPGPUs the most attractive choice in supercomputing centers. Unsurprisingly, most of the GPGPU-based studies have been focusing on performance improvement leveraging GPGPU's high degree of parallelism. However, for many scientific applications that commonly run on supercomputers, program correctness is as important as performance. Few soft or hard errors could lead to corrupt results and can potentially waste days or even months of computing effort. In this research we exploit unique architectural characteristics of GPGPUs to propose a light weight error detection method, called Warped Dual Modular Redundancy *(Warped-DMR). Warped-DMR detects errors in computation by relying on opportunistic spatial and temporal dual-modular execution of code. Warped-DMR is light weight because it exploits the underutilized parallelism in GPGPU computing for error detection. Error detection spans both within a warp as well as between warps, called* intra-warp *and* inter-warp DMR*, respectively. Warped-DMR achieves 96% error coverage while incurring a worst-case 16% performance overhead without extra execution units or programmer's effort.*

1. Introduction

Recently GPU architectures have been enhanced with several microarchitectural features that allow GPUs to be used not only for graphics applications but also for general purpose computing. Nowhere else is this trend more visible than in super-computing centers which are adopting GPUs as processing engines to achieve massive parallel execution. Named GPGPU, the new GPU variant is known to derive superior performance over multi-core CPUs by allowing thousands of concurrent threads to run efficiently within a limited power budget. Many application developers have been attracted to this new powerful and relatively cheap parallel execution paradigm. Significant effort has been expended to port applications to GPGPU paradigm in order to achieve better performance by efficiently leveraging abundant parallel computational resources.

GPGPUs now run business-critical applications, long running scientific codes, and financial software. These new application domains demand strict program correctness [7]. A few erroneous computations or a corrupt value could have severe negative repercussions. CMOS technology scaling, while provided power and performance benefits, is also leading to significant number of reliability concerns. GPGPU will be vulnerable to soft/hard error and the vulnerability is predicted to grow exponentially [6]. Since GPGPUs evolved from GPUs, the primary focus of GPGPU design has been to increase parallel performance. In particular, reliability is considered as a secondary issue in GPU computations since traditional graphics applications have been shown to be inherently fault tolerant [7]. However, to support business critical application domains on GPGPUs, there is a need to provide architectural support for at least error detection. As a first step, error detection can translate the most harmful silent data corruption (SDCs) errors to detectable but unrecoverable errors (DUEs). In fact, commercial GPGPU designers have already started addressing reliability concerns. Recently NVIDIA's Fermi GPGPU added ECC for the memory components [16].

GPGPUs have hundreds of hardware thread contexts today and in the near future they will have thousands of contexts. Each thread context contains a relatively simple processor pipeline with very minimal resources to support speculation, if any. Hence the vast majority of the chip area is dedicated to execution units, such as ALUs. In the presence of hundreds (or even thousands) of thread contexts, even a tiny probability of a logic error in each thread context adds up to an exponentially high probability of errors at the chip level. Recognizing this concern, several researchers have been focusing on improving reliability of GPGPU computation. They are mostly software approaches [6] [22]. Software approaches can be more flexible but demand programmers to re-write their applications with focus on fault tolerance, which is quite undesirable given that writing a GPGPU application itself is non trivial [11]. Compilers could reduce some of the burden on the programmer by automatically providing redundant code execution [22]. However, the error coverage is limited by the granularity of compiler's code insertion and has the *hidden error* problem: even though each line of code is executed twice for verification purpose, if the two instances are executed on the same processing core, some hardware defects, such as stuck-at faults, cannot be detected. Note that software does not decide which thread is mapped to which core in current GPGPU architectures. Furthermore, in software approaches the results from redundant execution are mostly compared at the end of the program execution. Hence faults are likely to be discovered too late to take quick corrective action.

In this paper we propose a low-overhead hardware approach for detecting computation errors in GPGPUs. The approach uses dual modular redundancy (DMR) [13] but opportunistically switches between spatial and temporal redundancy to improve error coverage while reducing the error detection overhead. We call this approach Warped-DMR. In this paper, we assume that only execution units are vulnerable. Memory is assumed to be protected by ECC, as is done in many industrial GPU designs [16]. Hence, for the memory operations, we only verify the address computations and assume that the loaded data is always error free.

1.1. Exploiting Opportunity

Before presenting the details of Warped-DMR, we first present motivating data that shows the utilization of thread contexts on a GPGPU. Figure 1 shows the execution time breakdown in terms of the number of active threads for a subset of benchmarks selected from NVIDIA CUDA SDK [4], Parboil Benchmark Suite [5], and ERCBench [1]. These results were generated by simulating a modern NVIDIA-style GPGPU architecture using *GPGPU-Sim* [3]. More details of the simulated architecture and benchmarks are provided later in Section 5. NVIDIA GPGPU executes instructions in a batch of threads unit

1072-4451/12 $26.00 © 2012 IEEE
DOI 10.1109/MICRO.2012.13

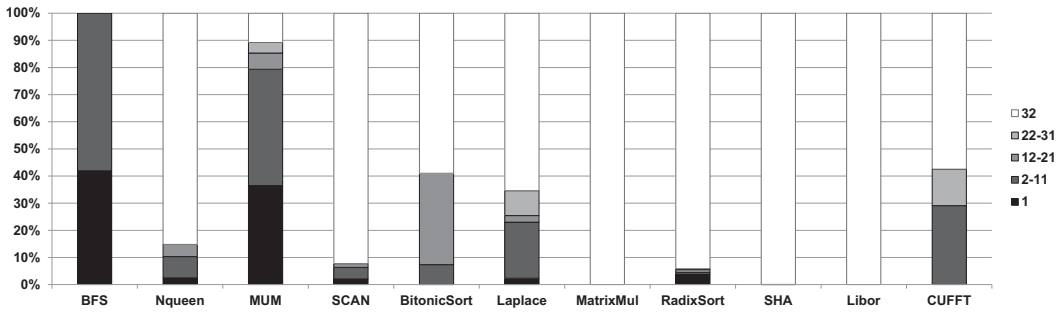


Figure 1: Execution time breakdown with respect to the number of active threads

(a.k.a. *Warp*) which consists of 32 threads. Each color in the bar chart denotes the fraction of cycles that the corresponding number of threads are actively executing the code. As can be seen, majority of applications do not have 32 active threads all the time. For example, over 40% of BFS instructions are executed by only single thread. In other words, the remaining 31 threads within the warp are idle while the single active thread is executing instructions. The processing cores associated with the idle threads remain unused during the idle period. The reasons for this underutilization are described later in Section 2.2.

The underutilization of GPGPU resources provides us an opportunity to provide error detection capability by exploiting the underutilized resources without incurring performance overheads. In this paper we present Warped-DMR, which consists of two techniques for error detection.

(1) *Intra-warp DMR:* Our first error detection method is called *intra-warp DMR*. Intra-warp DMR simply uses the inactive threads to verify execution of active threads in the same warp by using dual modular spatial redundancy. The underutilized or idle cores are used as computational checkers for a subset of active threads. Hence the overhead of intra-warp DMR is nearly zero, with the exception of a negligible area overhead needed to compare the computation results and duplicate the input data.

(2) *Inter-warp DMR:* The second error detection approach is called *inter-warp DMR*. When a warp is fully utilized, all 32 threads are active in a warp, and hence there is no opportunity for intra-warp DMR. In this scenario we use dual modular temporal redundancy. A duplicated execution of each fully utilized warp is scheduled for execution later whenever the associated execution unit becomes idle. A special purpose *Replay Queue*(ReplayQ) is used for the purpose of buffering the duplicate execution warps. We also shuffle the execution of redundant threads onto different cores, compared to the original thread-to-core assignment, to reduce the hidden error problem. Inter-warp DMR when combined with the ReplayQ mechanism reduces the need for unnecessary stalls in the pipeline to execute redundant instructions, thus significantly lowering the performance overhead of temporal redundancy.

Simulation results on several GPGPU applications shows that intra-warp and inter-warp DMR complement each other to provide 96.43% error coverage with 16% worst case performance overhead.

The remainder of this paper is organized as follows. Section 2 provides background on the design of contemporary GPGPUs and error detection methods. Section 3 describes Warped-DMR. Section 4 describes the architectural modification for supporting Warped-DMR. Section 5 shows our evaluation methodology and results. Section 6 discuss related work and we conclude in Section 7.

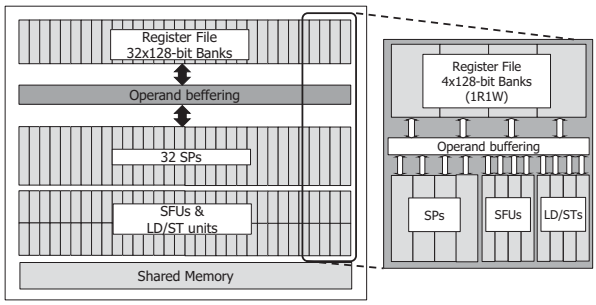


Figure 2: GPGPU Chip architecture and a SIMT cluster(modified after borrowing from [8])

2. Background

2.1. GPGPU Architecture

The GPGPU architecture varies depending on vendors and models. In this paper, we use the basic architecture of NVIDIA's Fermi [16] as our GPGPU model. A GPGPU consists of a scalable number of *streaming multiprocessor*(SM)s, each comprising of *shader processor*(SP) cores for arithmetic operations, *LD/ST units* for memory operations, *Special Function Units* (SFUs) for instructions such as sine, cosine, and square root, several register file banks, and shared memory. In this paper, we assume that each SM has 32 SPs, 32 register file banks and 64KB of shared memory.

Fig 2 shows the internals of one SM. The shared memory is accessible by all the SPs within a SM and part of this memory is configurable as an L1 cache. Each SM schedules threads in a unit of thread group of 32 threads, called a Warp. The threads within a warp execute the same code in a lock step manner. They all share one *program counter*(PC) but access different data operands. Such an execution approach is called *Single Instruction Multiple Threads*(SIMT) execution. Each thread in a warp may use a different register bank within a SM to access its data operands. Each individual thread execution is referred to as *SIMT lane*.

Within each SM, we further assume that four SIMT lanes make a *SIMT cluster* as in [8]. Thus each warp has eight SIMT clusters. As shown in Fig 2, each SIMT cluster has 4 SPs and 4 banks of register files. Each entry of a register bank is 128-bit wide and contains four 32-bit registers, each associated to one SIMT lane [8]. As each entry of the register bank consists of 4 registers having the same name but associated with 4 different threads, loading an entry from a register bank can feed all 4 SIMT lanes at once. Most common instructions that read 2 operands, write 1 result (*2R1W*), as well as the special instruction like MULADD that read 3 operands, write 1 result (*3R1W*) can access the four register banks to read their input operands and write output data concurrently without any register port

stalls most of the time. However, if an instruction fetches operands from the same bank, the operands cannot be fetched concurrently. To handle bank conflicts, GPGPUs use operand buffering logic that hides the latency of multi-cycle register fetch.

2.2. Underutilization of GPGPU Resources

Underutilization of GPGPU's computational resources can be due to two reasons (1) underutilization within homogeneous execution units and (2) underutilization among heterogeneous execution units.

Underutilization of homogeneous units: Underutilization within homogeneous execution units is caused by lock-step execution in GPGPUs. All 32 threads in a warp share a single PC. Whenever a branch instruction is encountered, some of the threads within the warp may take the branch while others may not depending on the data operands. Due to a single PC-constraint, threads with not-taken branch are executed first followed by threads with taken branches (or vice-versa). While the not-taken path instructions are executed, the core assigned to the taken path threads are idled. This is called the branch divergence problem.

To execute the divergent instructions, GPGPU hardware scheduler uses an *active mask* which consists of 32 bits indicating the active state of each thread within a warp. At each cycle, the threads whose active bit is set to '1' are allowed to execute the issued instruction, while the threads whose active bit is set to '0' wait. We call the thread as an *active thread* if it has '1' in the corresponding bit of the active mask and as an *inactive thread* otherwise. Note that there is one active mask per each warp.

A simple example of a branch divergence is illustrated in Figure 3. Let us assume that an if-then-else statement(shown in Figure 3(a)) is executed by a warp of two threads. When both threads reach the conditional branch instruction and if the condition is true for both threads then the two threads are concurrently executed (shown in Figure 3.(b)). However, if the two threads take different branch paths then only one thread can execute at a time (shown in Figure 3.(c)). In this example, the utilization of the system while executing the if-else statement becomes only 75% since among 8 cycles (2 cores $\times$ 4 cycles each), 6 cycles are actually used for the execution. Underutilization is even worse in real applications as shown in Figure 1: ranging from 7% in BFS to up to 77% in Bitonic Sort.

Underutilization of heterogeneous units: The underutilization among heterogeneous execution units is caused by the limitations in the scheduler feeding three different execution units. GPGPUs have three different types of execution units: SPs, LD/ST units, and SFUs. All three different types of execution units are fed by a single warp scheduler and an instruction dispatcher unit [16]. Hence, during any given cycle, only one instruction can be issued to one of the three execution units which leads to idle units. Heterogeneous unit underutilization has not been considered as severe as the underutilization caused by control divergence. However, if a code segment executes the same type instructions in a burst fashion then the scheduler will schedule instructions to just one type of execution unit while the rest two execution units remain idle.

Some state-of-the-art GPGPUs such as NVIDIA Fermi and Kepler [17] have multiple schedulers per SM. For example, Fermi has two schedulers in a SM which can issue instructions concurrently. The two schedulers share LD/ST units and SFUs while having their own SPs. Hence, instructions can be simultaneously issued to two different type execution units among three if the two schedulers issue different type operations. Even in this case there is still an underutilization of heterogeneous units since not all three execution units are used, but the degree of underutilization is decreased. Furthermore, due to several scheduling issues such as data dependency among the instructions, schedulers are not likely to be able to issue instructions to all the execution units.

3. Warped-DMR

Warped-DMR exploits the two types of underutilized resources to execute code redundantly and opportunistically. Different execution strategy is used for each of them.

3.1. Intra-warp DMR

To detect errors in execution units, intra-warp DMR relies on DMR execution approach. In traditional DMR there are as many *verification cores* as the number of *monitored cores*. A verification core executes the same instruction stream of the associated monitored core and the two execution outputs are compared. Error is detected if the results on the two cores differ. Every single instruction is thus executed twice providing 100% error coverage. However, the area or performance overhead of DMR exceeds 100% since at least one verification core should be added for each monitored core.

Intra-warp DMR uses the cores idled by underutilization within homogeneous execution units to execute the code redundantly, instead of adding extra cores for verification purpose. Whenever a partially utilized warp is scheduled, the operands of an active thread within the warp are forwarded to an inactive thread. The inactive thread thus can DMR an active thread's execution. The execution results of the inactive thread and the active thread are compared at the end of execution. If the two results are not identical, the hardware scheduler will be notified of an error occurrence. The necessary microarchitectural support for intra-warp DMR are discussed in Section 4.

Since the focus of this work is to detect errors, error handling is out of scope of this paper. But one can use simple techniques that allow the scheduler to either re-schedule the warp (in case of transient errors) or to stop running the program and raise an exception to the system (in case of a permanent fault).

3.2. Inter-warp DMR

Intra-warp DMR is an opportunistic approach that exploits idle cores. But when a warp utilizes all the cores, intra-warp DMR is unable to provide error detection coverage. To handle this case we present the second error detection method, called inter-warp DMR. Inter-warp DMR exploits resource underutilization caused due to heterogeneous execution units. As mentioned earlier, NVIDIA GPGPU uses SPs for arithmetic operations, LD/ST units for memory instructions, and SFUs for complex GPGPU operations such *sine* and *cosine*. In any given cycle the instruction issue logic issues instructions to only one of the three execution units. Hence, when an instruction is issued to SFUs or LD/ST units, the SPs may become idle. If different instruction types are issued in an interleaved manner, then each instruction's verification is done at the following cycle of the original execution. For instance, if an arithmetic instruction is followed by a LD/ST instruction, then the arithmetic instruction will be redundantly executed in the next cycle on SPs when the primary LD/ST instruction is being executed.

Figure 4 shows a simplified execution of a code segment which has several interleaved *add* and *load* instructions. As the two instructions use different type of execution units(*add* uses SPs and *load* executes on LD/ST units), whenever an instruction is issued onto the corresponding execution units, the other type of execution units become

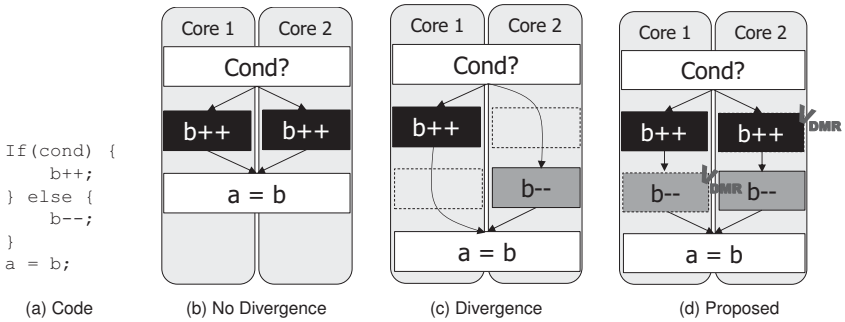


Figure 3: Example of underutilization of homogeneous units and Intra-Warp DMR

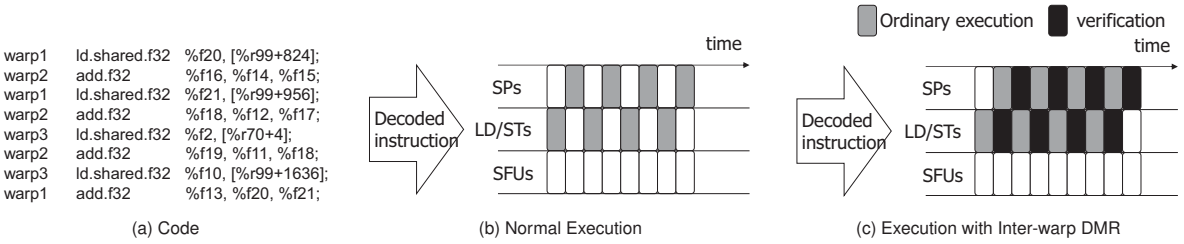


Figure 4: Example of underutilization of heterogeneous units and Inter-Warp DMR

idle. If both units take only one cycle to execute, inter-warp DMR allows the *add* and *load* instructions to be DMRed one cycle later than the original execution cycle on the associated execution units, without the need for stealing many cycles from regular program execution. Figure 4(c) depicts the operation of inter-warp DMR which only adds one extra cycle at the end of the eight cycle execution. Inter-warp DMR does not interfere with the execution scheduling of the primary *add* and *load* instructions.

Even with inter-warp DMR, there are scenarios when it is not possible to completely eliminate the overhead of DMR. In the example shown above, instructions that require different types of execution units are interleaved. But when the same type of instructions are scheduled for several cycles in a row, a new microarchitectural structure called ReplayQ is used to buffer the unverified instructions so that the instructions can be dequeued and re-executed whenever the corresponding execution unit becomes available. Note that we do not allow an instruction to consume unverified instruction results that are still buffered in the ReplayQ. Hence, whenever there is a RAW dependency on an unverified result, the dependent instruction is forced to wait and the ReplayQ gives priority to verify the source instruction.

During intra-warp DMR, the original code and verification code are guaranteed to be executed on different SIMT lanes. However, during inter-warp DMR, no such guarantee can be provided by default since contemporary GPGPUs may use core affinity that assigns a thread to the same core when redundantly executed. If an execution is DMRed on the same core, hardware defect on the core cannot be detected. For example, if core *i* has stuck-at-zero error, the result of the verification and original execution both will be 0, which leads to a hidden error. To avoid such hidden errors, inter-warp DMR associates a verification thread to a different SIMT lane than the original SIMT lane. We call this approach *Lane Shuffling*. Lane shuffling is operated within a SIMT cluster to minimize the wiring overhead. The microarchitectural enhancements for inter-warp DMR

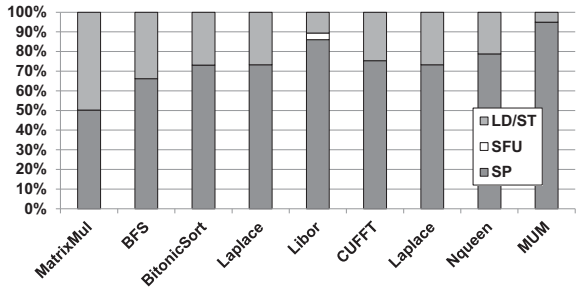


Figure 5: Execution time breakdown with respect to the instruction type

are discussed in Section 4.

3.3. Error Coverage

The theoretical error checking coverage of intra-warp DMR is 100% when the number of the active threads is less than half of the warp size. In this scenario every active thread's execution can be verified by at least one of the inactive threads. If the active thread count is greater than half of the warp size, the coverage is $\frac{\#inactive\_threads * 100}{\#active\_threads}\%$. The overhead of intra-warp DMR is almost zero as verification is done on the existing idle cores concurrently with the active threads. Only minimal hardware logic is added for register forwarding and results comparison.

The theoretical error checking coverage of inter-warp DMR is 100% as each fully occupied warp's execution is re-executed a few cycles later. The best case execution overhead of inter-warp DMR is zero as the redundant execution is done only when the corresponding execution unit is idle. In reality, due to the capacity of the ReplayQ and the unbalanced instruction distribution(see Figure 5 for instruction type distribution), there will be some overhead. Our results show that the worst case overhead is 16%, which is well below the theoretical overhead of 100%.

Priority	MUX0	MUX1	MUX2	MUX3
1st	0	1	2	3
2nd	1	0	3	2
3rd	2	3	0	1
4th	3	2	1	0

Table 1: Priority table of RFU MUXs

3.4. Advantages of Warped-DMR

Warped-DMR verifies the computations at individual execution unit level(i.e. SP). The DMR can be done at a coarser granularity, such as at the entire SM level by duplicating a thread block[1] onto two different SMs or at the chip level by invoking two copies of a kernel function onto two GPGPUs. The coarser method might be simpler to implement. However, the finer method allows for more aggressive error detection. For example, when there is a faulty SP, a SM-level or a chip-level error checking cannot isolate which core has the defect. Hence, the only option to fix the problem is to disable the entire SM even though the remaining 31 SPs in the SM as well as the other logic blocks including scheduler, dispatcher, and the local memory are fault-free. Similarly, when using chip level checking, one has to disable an entire GPGPU chip even with just one failed SP. With Warped-DMR we can monitor the reliability at the granularity of a SP. In the previous examples, we can still use the SM even though a SP has a defect by using a core re-routing approach as suggested in [23].

It is also worth noting that the static power consumption of GPGPUs is nearly 60% of the total power consumption. To reduce static power consumption [9] showed that it is best to provide power gating at the SM level. Idle SM periods can be long and hence they can be completely turned off . But providing power gating at the SP level does not provide enough benefits. While SPs are idle for a significant fraction of the time, the idleness is finely interspersed with periods of activity. The latency of power gating outweighs the benefits of turning off idle SPs. Warped-DMR is thus an ideal choice for repurposing idle SPs to provide reliability since power gating idle SPs is not beneficial.

4. Architectural support for Warped-DMR

4.1. Register Forwarding Unit

For intra-warp DMR each inactive thread that is going to verify the computation should be able to either access an active thread's register file or get the active thread's register values using data forwarding. As adding an extra port to the register file is expensive, we added a *Register Forwarding Unit*(RFU) at the end of each register bank. RFU consists of four 4 32-bit input MUXs as can be seen in Figure 6. A 128-bit entry of a register bank is divided into 4 32-bit data and forwarded to all the 4 MUXs.

To enable intra-warp DMR the four MUXs in the RFU pair active threads with inactive threads based on a priority. The priority configuration of the 4 MUXs within a SIMT cluster is shown in Table 1. Each column indicates the priority ordering for each MUX. As a first priority every MUX delivers the input data to its associated SIMT lane if that SIMT lane's active mask is set. MUX0 provides input data to SIMT lane 0, MUX1 to SIMT lane 1 and so on. If a SIMT lane's active mask is reset, then that SIMT lane is idle and can be used for DMR. Hence, every idle SIMT lane looks for an active SIMT lane whose computation can be redundantly executed on the idle SIMT lane. To find an active SIMT lane which can be redundantly executed on an idle SIMT lane, each MUX looks for the active SIMT lane according to the priority listed in the table. For instance, if SIMT lane 0 is idle, then MUX0 looks at SIMT lane 1 to see if it is active as lane 1 is the 2nd priority for MUX0. If so, then the inputs from SIMT lane 1 are then simply directed by MUX0 to run on SIMT lane 0. If SIMT lane 1 is also inactive then SIMT lane 2 active mask bit is checked followed by SIMT lane 3 active mask to find an active thread. As can be seen from Table 1, each MUX runs through a different priority sequence to allow uniform pairing possibilities between active and idle SIMT lanes. In this algorithm, if there is only one active lane, the lane is redundantly executed on the rest three idle lanes, which results in more than dual modular redundancy. We simply allow such a scenario to occur rather than to add additional hardware logic in the MUX to prevent this scenario since it does not lower the error coverage.

In Figure 6, the bold lines inside of RFU illustrates a simple example of an intra-warp DMR when an active mask for an instruction is 4'b0011. As each bit of an active mask indicates each corresponding thread's activeness, 4'b0011 means that thread 0 and 1 are active and thread 2 and 3 are inactive for the instruction. Thread 2 and 3 will perform DMR for the execution of thread 0 and 1 according to intra-warp DMR assignment from Table 1.

We implemented a RFU design and a 128-bit comparator by using *Synopsis Design Compiler v.Y-2006.06-SP4* [21]. The respective area overhead is $390\mu m^2$ and $622\mu m^2$ and the timing overhead is 0.08ns and 0.068ns. The timing overhead of the MUX is thus less than 0.06% compared to a typical cycle period(1.25ns) of GPGPU of 40nm technology and 800MHz core clock. [2]

4.2. Thread-Core Mapping

Since the register forwarding is limited to within a SIMT cluster of just four SPs, in intra-warp DMR the verification and monitored core are restricted to be within the same SIMT cluster. This limitation minimizes wire delays and complex routing paths. With this mapping restriction, however, some SIMT clusters might not be able to use intra-warp DMR if all the SIMT lanes within a cluster are fully utilized, even when some SIMT lanes across clusters are idle. Based on preliminary experiments we found that many applications are likely to have unbalanced active thread distribution within a warp.

To improve the availability of idle SPs within a SIMT cluster, we modified the thread to core affinity scheduling algorithm. There is only sparse documentation on how current GPGPUs assign threads to SPs within a warp. It is believed that the threads are mapped to cores in order: for example, thread 0 is always executed on core 0, thread 1 is mapped to core 1 and so on. Our modified scheduling algorithm assigns threads to SIMT clusters in a round-robin fashion. Thus thread 0 is assigned to cluster 0, thread 1 is assigned cluster 1 and so on. We show later in our results section that this simple scheduler change increased error detection opportunities by 9.6% compared to the default in order mapping of threads to core.

4.3. ReplayQ

As discussed earlier, inter-warp DMR relies on re-executing an instruction at a later time if the corresponding execution units are not free. Instead of stalling program execution, inter-warp DMR buffers

[1] A thread block in NVIDIA CUDA programming is a logical partition of a program. Any thread can communicate with other threads only when they are in the same thread block. A thread block is launched onto a SM.

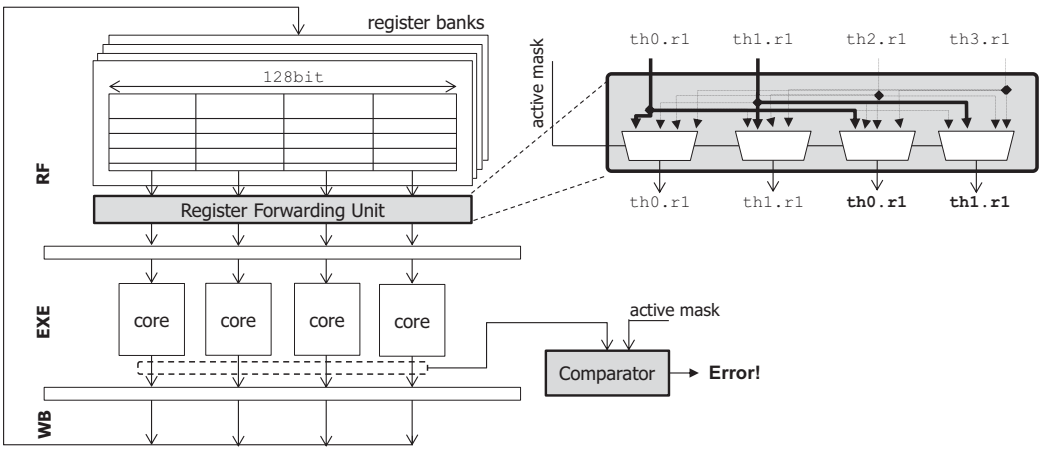


Figure 6: Register Forwarding Unit and Comparator for Intra-Warp DMR

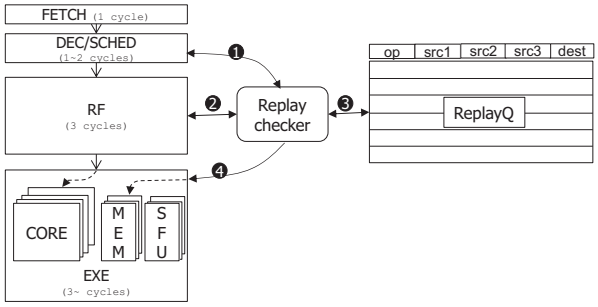


Figure 7: ReplayQ and Replay Checker for Inter-Warp DMR

unverified instructions into a ReplayQ whenever the corresponding execution unit is not available. A *Replay Checker* engine is designed to manage the ReplayQ. There is one Replay Checker and ReplayQ per SM.

Figure 7 shows how the Replay Checker works within the context of current GPGPU pipeline. A GPGPU pipeline in the figure has the following stages: instruction fetch stage (FETCH), decode & schedule stage (DEC/SCHED), register fetch stage (RF), and execution stage (EXE). The write back stage (WB) is omitted in the figure for simplicity. The latency of each pipeline stage is also shown in the figure. The stages having multiple cycle latency consist of multiple sub-stages. For example, RF is comprising of RF0, RF1, and RF2. These latencies reflect the pipeline latencies of current GPGPUs that we modeled [8].

If active mask of the instruction in the first *RF* stage is all active, the Replay Checker is activated. If the active mask has some idle slots intra-warp DMR will verify the warp's execution. During intra-warp DMR execution Replay Checker and ReplayQ do not play any role in managing the warp's redundant execution. Once Replay Checker is active it compares the instruction type of the warp in *RF*(❷) and that of the warp in $DEC/SCHED$ stage(❶). If the instruction type is different, then the Replay Checker creates a DMR copy of the *RF* instruction to be co-executed with the instruction in $DEC/SCHED$ (❹). DMR copy consists of the values of the input operands and opcode. Note that even though an instruction takes several cycles of the execution, the next instruction can be issued at the following cycle to the execution unit as the EXE stage itself is super-pipelined. If *RF* and $DEC/SCHED$ instruction type are the same then the instruction type in *RF* stage (a two bit value indicating SP, LD/ST or SFU instruction) is compared against all the queued entries in the ReplayQ (❸). The instruction decoder would have already marked each instruction based on its execution resource demand into either SP, LD/ST or SFU instruction type. In our experiments the maximum size of ReplayQ is 10 entries. Hence, 10 two bit XORs are used for this comparison. If any instruction in the ReplayQ has different type than the instruction in *RF* then the Replay Checker dequeues that instruction and pairs it with the instruction in *RF* stage (❹) for co-execution in the next cycle. When multiple ReplayQ instructions are available for co-execution then one instruction is picked at random. The instruction in *RF* stage is then enqueued in the ReplayQ. When an instruction *RF* is enqueued into ReplayQ it simply implies that the instruction that is one cycle behind *RF* is going to use the same type of execution units as the instruction in *RF*. Hence, there will be no opportunity to verify the *RF* instruction in the next cycle following its execution cycle. Hence, that instruction needs to be buffered for future verification.

Also to distinguish a DMR execution from an original execution, a single *dmr* bit is added. *dmr* is set by Replay Checker when creating a DMR copy. By using this value, RFU can apply the lane shuffling only to the fully utilized DMR executions.

If there is no instruction in the ReplayQ whose instruction type is different than the instruction in the *RF* stage, the Replay Checker checks if the ReplayQ is full. If the ReplayQ has empty slots, the *RF* instruction is enqueued to the ReplayQ (❸). If the ReplayQ is full, a stall cycle is inserted into the pipeline immediately after the instruction in *RF* finishes the first *EXE* stage and then the instruction is re-executed by using the operand values that are still available in the pipeline. This eager re-execution reduces unnecessary register reads but adds one cycle performance penalty. Note that this penalty is applicable only in the rare case that ReplayQ has no instruction that is different than *RF* stage instruction *and* ReplayQ is full.

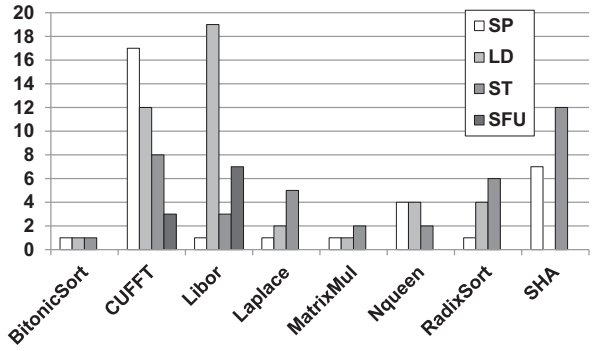
Whenever a new instruction is scheduled in the pipeline which is going to consume (RAW dependency) data from an unverified instruction that is buffered in the ReplayQ then the Replay Checker stalls the pipeline and executes the verification of the source instruction before allowing the consumer instruction to execute.

Algorithm 1 shows the pseudo code of the Inter-Warp DMR with ReplayQ.

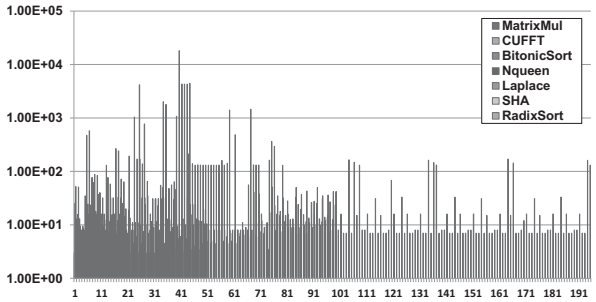
4.3.1. Components and Effective Size of ReplayQ Since ReplayQ buffers instructions only when there is no available resource, it is

Algorithm 1 Inter-Warp DMR with ReplayQ

i_{rf} := *instruction in RF stage*
i_{dec} := *instruction in DEC/SCHED stage*
op_{rf} := *instruction type of* i_{rf}
op_{dec} := *instruction type of* i_{dec}
if $op_{rf} \neq op_{dec}$ **then**
  Coexecute DMR of i_{rf} *with* i_{dec} *execution*
else
  if $\exists i_{rq} : i_{rq} \in ReplayQ$ *and instruction type of* $i_{rq} \neq op_{rf}$ **then**
    Dequeue i_{rq} *from ReplayQ*
    Enqueue i_{rf} *to ReplayQ*
    Coexecute DMR of i_{rq} *with* i_{rf} *execution*
  else
    if *ReplayQ is full* **then**
      Insert a Stall cycle
      DMR i_{rf} *one cycle later the original execution*
    else
      Enqueue i_{rf} *to ReplayQ*
    end if
  end if
end if



(a) Instruction type switching distances within 1000 cycles



(b) RAW dependency distances of the registers of warp1 thread 32(warp0 thread1 for SHA)

Figure 8: Two key factors to determine effective ReplayQ size

OS	Ubuntu Linux kernel v2.6.38
CPU	Intel Core i7(quad core) @ 2.67 GHz
Compiler	nvcc-2.3 / gcc-4.3.4

Table 2: Experimental Environment

Parameter	Value
Execution Model	In-order
Execution Width	32 wide SIMT
Warp Size	32
# Threads/Core	1024
Register Size	64 KB
# Register Banks	32
# Core(SP)s/Multiprocessor(SM)	32
# SMs	30

Table 3: Simulation Parameters

critical to quantify how often such a scenario occurs in GPGPUs. ReplayQ also stalls the pipeline whenever there is RAW dependency on an unverified instruction.

Figure 8(a) shows the average cycle distance before an instruction type is switched to another. In most of the applications, normally less than 6 instructions of the same type are consecutively issued. CUFFT, Libor, and SHA have longer distances between different instruction types but it is also bounded to a maximum of 20. Hence, the ReplayQ only needs to buffer 20 instructions in the worst case, but an average size of 6 will suffice for most applications.

Figure 8(b) shows the number of cycles between when a register is written to the time when that register is read by another instruction. In this figure we only show the RAW dependency distance for warp1 thread 32. But the data is quite similar for all warps and all threads within each warp. The RAW dependency distance is at least 8 cycles and almost half of the registers have greater than 100 cycles of distance and some have even longer than 1000 cycles of distance. Hence, the RAW dependency related pipeline stalls are likely to be just a few in Warped-DMR.

Each entry of the ReplayQ should maintain opcode and the original execution result as well as the source register values. The original execution result is for verifying the original execution. Each SM has one ReplayQ which covers all the SIMT Clusters. Each entry of a ReplayQ contains 32 *lanes* $\times$ 3 *operands*(each instruction can have up to 3 operands)$\times$4 *bytes* for the source register values, 32 *lanes* $\times$ 4 *bytes* for the original execution result and $2 \sim 4$ *bytes* for the opcode so total of $514 \sim 516$ *bytes*. Therefore, the ReplayQ size with 10 entries is around 5KB. This is only 4% of the register file size which is assumed to have 128KB in [8].

5. Evaluation

5.1. Settings and Workloads

We used *GPGPU-Sim v3.0.2* [3] to evaluate the proposed Warped-DMR approach. The simulation environment is described in Table 2 and the simulation parameters are set as listed in Table 3. The simulation parameters model our baseline GPGPU architecture as illustrated in Fig 2. A GPGPU chip has 30 SMs and each SM is comprising of 32 SIMT lanes. 4 SIMT lanes build a SIMT cluster which consists of 4 register banks, 4 SPs, 4 LD/ST units and 4 SFUs.

For the workloads, we used several applications from NVIDIA CUDA SDK [4], Parboil Benchmark Suite [5], and ERCBench [1].

Category	Benchmark	Parameter
Scientific	Laplace transform	$gridDim = 25 \times 4$, $blockDim = 32 \times 4$
	Mummer	$input files$: $NC\_003997.20k.fna$ and $NC\_003997\_q25bp.50k.fna$
	FFT	$gridDim = 32$, $blockDim = 25$
Linear Algebra/Primitives	BFS	$input file$: $graph65536.txt$, $gridDim = 256$, $blockDim = 256$
	Matrix Multiply	$gridDim = 8 \times 5$, $blockDim = 16 \times 16$
	Scan Array	$gridDim = 10000$, $blockDim = 256$
Financial	Libor	$gridDim = 64$, $blockDim = 64$
Compression/Encryption	SHA	$directmode$, $inputsize$: 99614720, $gridDim = 1539$, $blockDim = 64$
Sorting	Radix Sort	$–n = 4194304$ $–iterations = 1$ $–keysonly$
	Bitonic Sort	$gridDim = 1$, $blockDim = 512$
AI/Simulation	NQueen	$gridDim = 256$, $blockDim = 96$

Table 4: Workloads

As mentioned earlier, our main target applications are those needing strict accuracy such as scientific computing or financial applications. Hence, we excluded some applications that are inherently fault tolerant, such as graphics applications. We picked 6 categories of applications: scientific computing, linear algebra/primitives, financial, compression/encryption, sorting, and AI/simulation. The applications that are included in the 6 categories are listed in Table 4.

5.2. Error Coverage and Overhead

Figure 9(a) shows the percentage of executed instructions covered by Warped-DMR. We compare three different implementations. The baseline implementation is the 4 SIMT lane cluster with no enhanced thread-core mapping. The second bar shows the impact of increasing the cluster size to 8 SIMT lanes and allowing register forwarding within a larger cluster size. The last bar shows the results using the enhanced thread-core mapping as stated in the Section 4.2. Warped-DMR with enhanced thread mapping provide an average of 96.43% error coverage compared to 91.91% error coverage with a more hardware intensive 8 SIMT lane cluster. The gaps in error coverage are primarily due to intra-warp DMR when the number of idle cores is fewer than the number of active cores. For instance, BFS is almost exclusively covered by only intra-warp DMR as all the warps are underutilized. The utilization of all the warps in BFS is less than 50% as illustrated in Figure 1. Hence, every single active thread's execution can be verified by inactive threads without any ReplayQ involvement. Such applications also have negligible performance overhead (almost zero)as seen in Figure 9(b) while the error coverage is 100%. CUFFT derived the lowest error coverage, 90.167%, as most of the underutilized warps' utilization is greater than 80%, implying only 25% of active threads' executions can be verified.

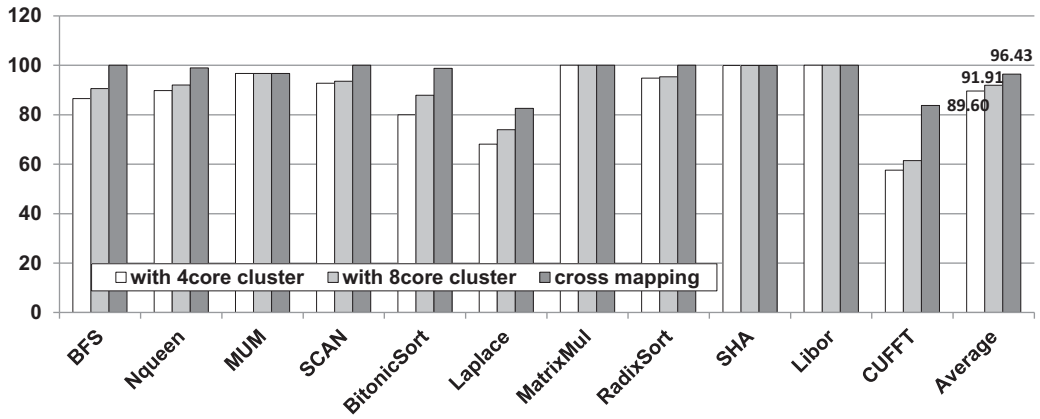
Applications that are well parallelized like Libor, MatrixMul, and SHA are mostly covered by inter-warp DMR as most of the warps are fully utilized. In such applications, the error coverage is almost 100% but the performance overhead is higher than the other applications as shown in Figure 9(b). There are four bars per benchmark in Figure 9(b). Each bar is normalized to the kernel execution cycles of the base machine with zero error detection support. Using the data presented in Section 4.3.1, we varied the ReplayQ from 0 to 10 entries. As the ReplayQ size increased to a maximum of just 10 entries the average performance overhead reduced to 16%. In some applications that are mostly covered by inter-warp DMR such as MatrixMul, performance overhead without ReplayQ exceeds 70%. However, by using 10 entries of ReplayQ, the overhead drops to 18%.

5.3. Comparison with SW approach

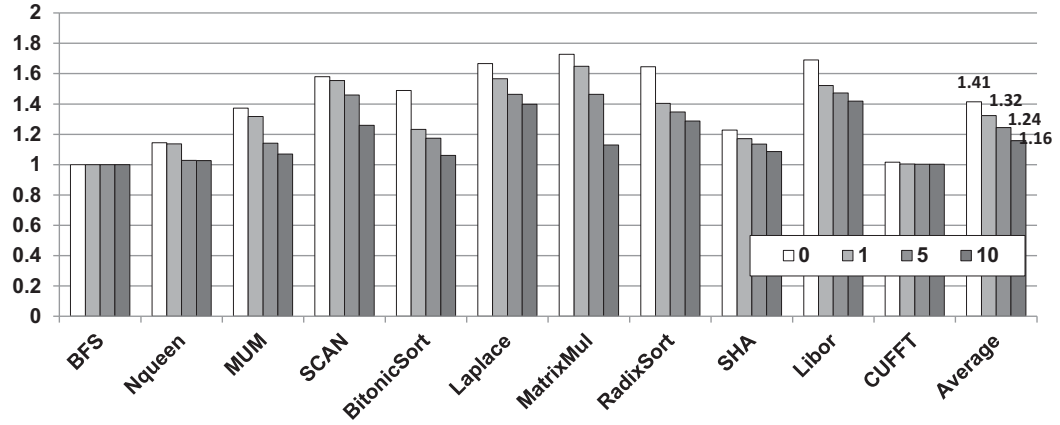
In this section, we compare Warped-DMR with other approaches. We modified applications to provide error detection capabilities through R-Naive and R-Thread approaches as described in [6]. R-Naive simply invokes the kernel function twice and then compares the output data. As each kernel should use the same input data, the memory copy between CPU and GPGPU also should be duplicated. It is relatively simple to implement but has more than 100% time overhead [6]. R-Thread duplicates the thread blocks within a kernel and then compares the output of the original and the duplicated thread block. The duplicated thread block refers to the original thread block's input data by modifying index calculations. If there is any SM that is not used for the kernel execution, the redundant thread block can be executed on it simultaneously with the original thread blocks. Otherwise, R-Thread also can have quite high execution overhead as the execution time for redundant thread blocks cannot be hidden. R-Thread does not call expensive CUDA APIs redundantly like R-Naive but it still requires twice as much data transfer from GPGPU to CPU resulting in significant data transfer overhead. R-Scatter in [6] is not used in this experiment as it is for VLIW architecture and it was shown to be inferior to R-Thread. We also implemented a dual modular temporal redundancy(DMTR) which verifies every single instruction in the following cycle. It is a simplified version of SRT with 1 cycle of slack time [19].

Figure 10 shows the execution time of the four different error detection approaches and the original execution without error detection. The execution time includes data transfer time between CPU and GPGPU as well as kernel execution time. As the data transfer between CPU and GPGPU is not included in the simulation, the data transfer time was measured by using CUDA Timer API. As R-Naive should call kernel twice, the data transfer time also becomes twice the original. In R-Thread, twice the size of original output should be copied back to the CPU as the output of redundant and original thread block are compared on the CPU side. On the other hand, both DMTR and Warped-DMR have the same data transfer time as the original execution as they compare the execution results on the GPGPU.

Of the four approaches, R-Naive took the longest time for execution. It is mainly due to two individual kernel executions and the twice the data transfer time. R-Thread can reduce the execution overhead in the presence of idle resources at granularity of a SM. For instance, Bitonic Sort uses fewer than 30 SMs and hence idle SMs are always available. However, in the other applications which use all the SMs for the original execution, R-Thread has no room to hide the execution time of the redundant thread blocks. Also, R-Thread trans-



(a) Error coverage with respect to the SIMT cluster organization and Thread to Core mapping



(b) Normalized Kernel simulation cycles with respect to the ReplayQ size

Figure 9: Error coverage and Overhead of Warped-DMR

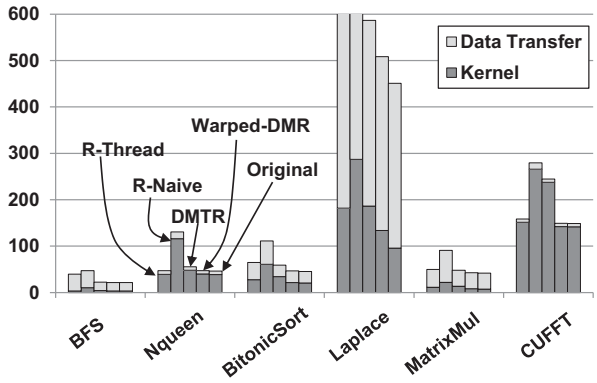


Figure 10: Execution times of different approaches

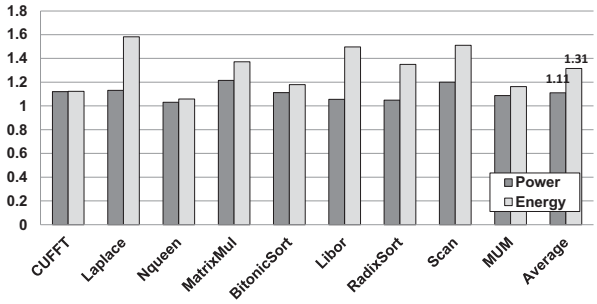


Figure 11: Normalized Power Consumption

fers twice the output data to the CPU. Therefore, R-Thread suffered the second longest execution time in many of the tested applications. Warped-DMR derived the best performance among the four different approaches due to opportunistic error detection. In some applications like Laplace, Warped-DMR has 20% longer kernel execution time compared to the original execution. However, when the data transfer time is included, the overall overhead is reduced to 13%.

5.4. Power Consumption

We measured power and energy consumption by using an analytical model [9]. In [9], the power consumption of processing components are estimated based on the following equation:

$$RP_{comp} = MaxPower_{comp} \times AccessRate_{comp} \quad (1)$$

$$AccessRate_{comp} = \frac{\sum \#Insts\_accessing\_comp\ per\ SM}{Exec\_cycles \div Ints\_sched\_interval} \quad (2)$$

The total power consumption consists of runtime and idle power. The runtime power is again comprising of runtime power of SMs and Memory. Each SM's runtime power is aggregated power consump-

tion of several processing components(SPs, SFUs, caches, shared memory, register file, and fetch/decode/schedule unit) and a constant factor. Each component's runtime power(RP_{comp}) can be calculated by multiplying the access rate of the component($AccessRate_{comp}$) by a power parameter($MaxPower_{comp}$) as shown in Equation 1. $AccessRate_{comp}$ is estimated by dividing the total number of instructions accessing the component(#$Insts\_accessing\_comp$) by the execution cycles($Exec\_cycles$) over the instruction scheduling interval($Ints\_sched\_interval$).

We assume that the architecture dependent parameters(i.e. $MaxPower_{comp}$, idle power, and constant factor) are the same as in [9]. By using the total simulation cycles and the number of executed instructions acquired from our simulation, we estimated the power consumption of each application with and without Warped-DMR. Memory components such as caches and shared memory are not included in the estimation since Warped-DMR only doubles the address calculations for the memory operations and the redundant executions are always conducted on already loaded data. ReplayQ is assumed to have 10 entries.

As shown in Figure 11, the power consumption of Warped-DMR is 11% higher and the overall energy consumption is 31% higher than the baseline with zero error detection capability. Some applications such as Laplace consume the worst case 60% more energy due to timing overhead as shown in Figure 9. Energy consumption is calculated by multiplying the power consumption of each application by its execution time. The execution time is calculated by multiplying the number of simulated cycles by the cycle duration. We assume that the cycle duration is 1.25 ns.

6. Related Work

In this section we describe the most relevant prior work on GPGPU reliability and DMR execution. [6] proposed three software approaches: R-Naive, R-Scatter, and R-Thread. R-Naive simply invokes memory API and kernel function twice to create a software-centric DMR execution within a GPGPU. R-Scatter tries to exploit underutilized VLIW lanes for redundant execution by duplicating kernel code. R-Thread doubles the thread block count for a kernel and uses the newly added thread blocks do the redundant execution. We compared our work with the R-Naive and R-thread approaches and showed that Warped-DMR significantly reduces the overhead of DMR execution.

[14] checks computations, control flow, and data flow of a GPGPU program by inserting signature collector code. After the kernel execution, the collected signatures are compared with statically generated signatures. However, this approach only checks the computation after the kernel code is complete, which can be much later than when the error was first encountered.

[22] is a systematic approach which uses a guardian process that intercepts the crash event and restarts the program using checkpoints. They also instrumented the source code, duplicated non-loop code and inserted range checking code for loops. This approach also relies on extensive software instrumentation and large checkpoints to support redundant execution. Many of these reliability studies for GPGPUs are software approaches. Software approaches are always more flexible compared to hardware approaches. However, the error coverage can be limited by the granularity of programmers (or compiler's) code insertion. Compared to those approaches, Warped-DMR can check 96.43% of all instructions without any programmer's effort.

[15] suggested a sampling DMR in which DMR is conducted only for a short period of time within each epoch rather than doing it for entire execution time. Using this approach the authors state that permanent errors can be eventually detected even though transient errors might be missed. Warped-DMR takes advantage of GPU-specific microarchitectural features to provide high coverage for both transient and permanent faults.

[19] proposed a Simultaneous and Redundantly Threaded (SRT) processor design. Instead of replicating hardware resources, they used thread level replication. Trailing thread redundantly executes the same program copy that the leading thread executes. The hardware resources are shared between the trailing and the leading threads. [13] proposed a Chip-level Redundantly Threaded (CRT) processor which explicitly disables core-affinity to make sure that two threads execute on different cores. This approach essentially exploits the performance advantage of SRT's loose synchronization as well as the high fault coverage of lockstepping method [20]. [10] proposed a method to reduce the performance overhead of SRT by preventing the trailing thread from redundantly fetching register data. The key idea is to reuse the already fetched data for the trailing thread.

Compared to the hardware based DMR or RMT approaches, Warped-DMR has some domain-specific advantages. Warped-DMR checks every single instruction (but in less than 4% of cases it checks only partial number of inputs). This approach not only detects permanent errors but most transient errors can also be detected. Also, unlike [20], Warped-DMR does not use an entire core just for DMR. Instead, we utilize the idle periods of cores for DMR.

Due to DMR's high area overhead, some self-checking schemes also have been studied. One of the most popular self-checking schemes is residue checking [18] [12]. Instead of duplicating entire execution units, residue checking adds residue operator units which require much less area than the entire execution units. An error in the original operator unit is detected by comparing the residue of the original computation result and the output of the residue operation which is executed on the residue operator unit. Residue checking has small area footprint but residue checking is only applicable for some simple arithmetic operations(it cannot be used for exponent calculations [12]). Warped-DMR can detect errors in any arithmetic operation supported on a GPGPU, including complex operations implemented in an SFU.

7. Conclusion

As GPGPUs play critical role in high performance computing today, reliability should be treated as a first class citizen alongside performance. In this paper, we proposed Warped-DMR a hardware approach to detect computation errors in GPGPUs. We presented the reasons for underutilization of resources in GPGPU applications and then presented inter-warp and intra-warp DMR to exploit the idle resources for error detection. Intra-warp DMR checks the active threads' execution by using idle cores from underutilized warps. For the fully utilized warps, inter-warp DMR verifies computation by using temporal DMR whenever the corresponding execution unit becomes idle. A simple ReplayQ microarchitecture design is used for maintaining instructions in case the corresponding execution unit is not idle for several cycles. To prevent an instruction from being executed and verified on the same core, which may lead to hidden errors, we designed a register forwarding/lane shuffling logic. We presented a detailed state space exploration and showed that Warped-DMR provides 96.43% error coverage with 16% performance overhead.

Acknowledgement

We would like to thank the anonymous reviewers for their valuable comments. This work was supported by DARPA-PERFECT-HR0011-

12-2-0020 and NSF grants NSF-1219186, NSF-CAREER-0954211, NSF-0834798.

References

[1] "Ercbench." [Online]. Available: http://ercbench.ece.wisc.edu/

[2] "Geforce 400 series." [Online]. Available: http://en.wikipedia.org/wiki/GeForce_400_Series

[3] "Gpgpu-sim." [Online]. Available: http://www.ece.ubc.ca/ aamodt/gpgpu-sim/

[4] "Nvidia cuda sdk 2.3." [Online]. Available: http://developer.nvidia.com/cuda-toolkit-23-downloads

[5] "Parboil benchmark suite." [Online]. Available: http://impact.crhc.illinois.edu/parboil.php

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

[7] X. Fu, N. Goswami, and T. Li, "Analyzing soft-error vulnerability on gpgpu microarchitecture," in *Proceedings of the 2011 IEEE International Symposium on Workload Characterization*, November 2011, pp. 226–235.

[8] M. Gebhart, D. R. Johnson, D. Tarjan, S. W. Keckler, W. J. Dally, E. Lindholm, and K. Skadron, "Energy-efficient mechanisms for managing thread context in throughput processors," in *Proceedings of the 38th annual International Symposium on Computer Architecture*, Jun 2011, pp. 235–246.

[9] S. Hong and H. Kim, "An integrated gpu power and performance model," in *Proceedings of the 37th annual International Symposium on Computer Architecture*, Jun 2010, pp. 280–289.

[10] S. Kumar and A. Aggarwal, "Reducing resource redundancy for concurrent error detection techniques in high performance microprocessors," in *Proceedings of the 12th International Symposium on High Performance Computer Architecture*, February 2006, pp. 212–221.

[11] V. W. Lee, C. Kim, J. Chhugani, M. Deisher, D. Kim, A. D. Nguyen, N. Satish, M. Smelyanskiy, S. Chennupaty, P. Hammarlund, R. Singhal, and P. Dubey, "Debunking the 100x gpu vs. cpu myth: an evaluation of throughput computing on cpu and gpu," in *Proceedings of the 37th annual International Symposium on Computer Architecture*, Jun 2010, pp. 451–460.

[12] D. Lipetz and E. Schewarz, "Self checking in current floating-point units," in *Proceedings of the IEEE 20th Symposium on Computer Arithmetic*, July 2011, pp. 73–76.

[13] S. S. Mukherjee, M. Kontz, and S. K. Reinhardt, "Detailed design and evaluation of redundant multithreading alternatives," in *Proceedings of the 29th annual International Symposium on Computer Architecture*, May 2002, pp. 99–110.

[14] R. Nathan and D. J. Sorin, "Argus-g: A low-cost error detection scheme for gpgpus," in *Workshop on Resilient Architectures*, December 2010.

[15] S. Nomura, M. D. Sinclair, C.-H. Ho, V. Govindaraju, and M. de Kruijf, "Sampling + dmr: Practical and low-overhead permanent fault detection," in *Proceedings of the 38th annual International Symposium on Computer Architecture*, Jun 2011, pp. 201–212.

[16] NVIDIA, "Fermi white paper v1.1." Available: http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf

[17] NVIDIA, "Nvidia geforce gtx 680 white paper v1.0." Available: http://www.geforce.com/Active/en_US/en_US/pdf/GeForce-GTX-680-Whitepaper-FINAL.pdf

[18] N. Ohkubo, T. Kawashimo, M. Suzuki, Y. Suzuki, J. Kikuchi, M. Tokoro, R. Yamagata, E. Kamada, T. Yamashita, T. Shimizu, T. Hashimoto, and T. Isobe, "A fault-detecting 400 mhz floating-point unit for a massively-parallel computer," in *International Solid-State Circuits Conference*, February 1999, pp. 368–369.

[19] S. K. Reinhardt and S. S. Mukherjee, "Transient fault detection via simultaneous multithreading," in *Proceedings of the 27th annual International Symposium on Computer Architecture*, Jun 2000, pp. 25–36.

[20] T. J. Slegel, R. M. A. III, M. A. Check, B. C. Giamei, B. W. Krumm, C. A. Krygowski, W. H. Li, J. S. Liptay, and J. D. MacDougall, "Ibm's s/390 g5 microprocessor design," in *IEEE MICRO*, March 1999, pp. 12–23.

[21] Synopsis, "Design compiler user guide," 2010. Available: http://acms.ucsd.edu/info/documents/dc/dcug.pdf

[22] K. S. Yim, C. Pham, M. Saleheen, Z. Kalbarczyk, and R. K. Iyer, "Hauberk: Lightweight silent data corruption error detector for gpgpu," in *Proceedings of 25th IEEE International Parallel & Distributed Processing Symposium*, May 2011, pp. 287–300.

[23] L. Zhang, Y. Han, Q. Xuz, and X. Li, "Defect tolerance in homogeneous manycore processors using core-level redundancy with unified topology," in *Proceedings of the Conference on Design, Automation and Test in Europe*, March 2008, pp. 891–896.