HAUBERK: Lightweight Silent Data Corruption Error Detector for GPGPU

Keun Soo Yim, Cuong Pham, Mushfiq Saleheen, Zbigniew Kalbarczyk, and Ravishankar Iyer
Center for Reliable and High Performance Computing
Coordinated Science Laboratory
University of Illinois at Urbana-Champaign
Urbana, IL, 61801, USA
{yim6, pham9, msalehe2, kalbarcz, rkiyer}@illinois.edu

Abstract— **High performance and relatively low cost of GPU-based platforms provide an attractive alternative for general purpose high performance computing (HPC). However, the emerging HPC applications have usually stricter output correctness requirements than typical GPU applications (i.e., 3D graphics). This paper first analyzes the error resiliency of GPGPU platforms using a fault injection tool we have developed for commodity GPU devices. On average, 16-33% of injected faults cause silent data corruption (SDC) errors in the HPC programs executing on GPU. This SDC ratio is significantly higher than that measured in CPU programs (<2.3%). In order to tolerate SDC errors, customized error detectors are strategically placed in the source code of target GPU programs so as to minimize performance impact and error propagation and maximize recoverability. The presented *HAUBERK* technique is deployed in seven HPC benchmark programs and evaluated using a fault injection. The results show a high average error detection coverage (~87%) with a small performance overhead (~15%).**

Keywords-GPGPU; fault tolerance; silent data corruption

I. INTRODUCTION

Graphics processing units (GPUs) are surfacing as a compelling platform for processing general-purpose HPC programs. HPC programs typically process large volumes of data using many collaborating computation tasks (e.g., science simulation or medical data processing). Modern GPU devices are effective at processing these large volumes of data because of their use of multiple cores, wide memory bandwidth, large-size register files, and many arithmetic units. This rich hardware resource lessens structural hazards, and the throughput-driven design of GPU core architecture addresses both data and control hazards (i.e., main hurdles at exploiting instruction-level parallelisms in CPU designs) [1]. Furthermore, GPU hardware resources are directly exposed to the programmer by the programming model (e.g., [2][3]).

HPC programs have strong output correctness requirements. This is in contrast to graphics programs where errors in computing colors of a few pixels may go unnoticed. Many HPC programs have quantifiable correctness requirements for their outputs. For example, in an HPC program computing a correlation function [4], more than 1% of value errors in any of the program output elements (e.g., a floating point number) compared with that of a golden run is treated as a silent data corruption (SDC) error. In this paper, an SDC is defined as an undetected data error in program output that violates correctness requirement of the program. SDC errors are serious problem in many HPC programs because of their long execution times and resulting high likelihood of experiencing hardware faults.

GPU devices targeting graphics applications usually do not need strong fault-tolerance techniques, e.g., these devices do not have any error correcting codes for memory protection [26]. As a result, a relatively high hardware fault rate was observed in such devices. For example, evaluation of commodity GPU devices found at least one permanent fault in 1.8% devices [5] and transient memory fault in 66% of evaluated GPUs [6]. Note that these transient errors are due to soft errors and/or software bugs in GPU device drivers. Recent versions of GPUs for HPC applications support memory fault tolerance techniques (e.g., CRC [8] in GDDR5 or SEC-DED ECC [7]). This is an important step building dependable GPU platforms for HPC domain.

Regardless of added memory error protection, HPC programs are still vulnerable to certain types of GPU hardware faults. For example, it is hard to detect faults in a GPU core (e.g., ALU, FPU, or register file) due to the irregularity and high operational speed of GPU core logic, i.e., constitutes a large portion of the silicon area in the GPU chip [7]. Furthermore, the high-density of transistors on the die increases the likelihood of multi-bit errors [9], and the integration of cores and memories contributes to an increase in hardware fault rate especially for intermittent fault [10].

Designing a technique to tolerate faults in GPU cores is challenging especially for HPC GPU programs because of their strong performance and cost requirements. The success of HPC applications on GPU platforms depends on the achieved computation efficiency in terms of performance versus cost or performance versus energy consumption, i.e., including the overhead for fault tolerance. From this perspective, the HPC GPU program opens a new design space different from traditional fault tolerance (e.g., for mission critical systems). Software-implemented error detection can provide lightweight cost-effective solution for this design space.

In this context, software-implemented full duplication (i.e., well-known techniques) can be an effective approach to detect SDC errors in GPU platforms. However, duplication usually doubles the program execution time. A naïve full duplication simply executes the same GPU kernel twice and compares the results from the two executions. Note that a GPU program consists of CPU- and GPU-side codes, and a GPU kernel is a part of the GPU-side code with an entry function callable from the CPU-side code. Considering the fact GPU kernels form a majority of total program execution time, the doubled execution time of GPU kernels can easily break the performance requirement of HPC GPU programs.

Optimizing naïve full duplication has achieved a limited success in GPU programs. Two optimization techniques have been studied by exploiting underutilized data- and thread-level parallelisms. The reported performance overhead is more than 84% [11]. This overhead is much higher as compared with similar techniques employed for CPU programs (e.g., [16] reports average overhead of 41%). This is because GPU programs are typically already heavily optimized and consume most of useable parallelism and computing resources in GPU.

This paper presents *HAUBERK*, a software technique to derive lightweight error detection and recovery customized for target GPU programs. The derived error detectors are strategically placed and customized by considering their performance and error propagation characteristics. The main contributions of this paper can be summarized as follows:

- A mutation-based, software-implemented fault injector (SWIFI) for evaluation of commodity GPU devices.
- Characterization of sensitivity of GPGPU applications to SDC errors. Our fault injection experiments show that single event upsets (SEU) (emulated by injection of single-bit errors) can seriously harm reliability and data integrity of GPU kernels. For example, 18-45% of data faults cause SDC errors in evaluated GPU programs.
- Design and evaluation of two types of error detectors: (i) *duplication and checksums* to protect *non-loop* GPU kernel codes and (ii) *accumulation-based value range checking* to protect *loop* portions of GPU kernels. Our profiling results indicate that loops form a majority of GPU kernel execution time (>98% in 5 out of 7 benchmark programs).
- Design of a guardian program which reexecutes GPU program in order to tolerate errors and to identify false alarms.
- Evaluation of the *HAUBERK* approach on seven HPC GPU programs. The evaluation results show that the average performance overhead is 15.3% (83% reduction as compared with an optimized full duplication) and the average error detection coverage is 86.8% for injected faults.

II. MEASUREMENT

This section evaluates the error sensitivity of HPC and graphics programs executing on GPU and performance characteristics of the used HPC GPU programs. The parboil benchmark suite [4] is used as the source of HPC programs (six are floating-point programs and one is an integer program). Two applications (ray-trace and ocean-flow simulation) from a GPU SDK [2] are used as 3D graphics programs.

A. Error Sensitivity

Figure 1 shows the error sensitivity of HPC GPU programs, graphics GPU programs, and CPU programs. The GPU program state is classified into three data types – pointer, integer, and FP (float) data – based on the type of data where faults are injected. We inject a single-bit error into each variable in benchmark program by using the fault injection tool described in Section VII.

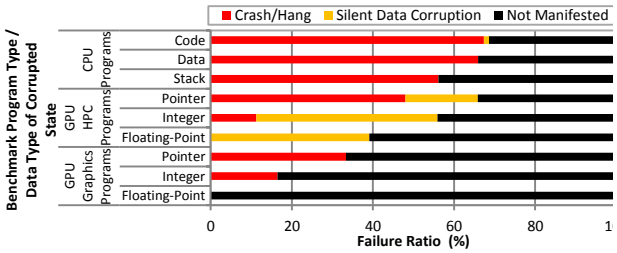


Figure 1. Comparison of average error sensitivity of HPC GPU program, graphics GPU programs, and CPU programs.

Observation 1: *An SEU (or single-bit error) in the pointer, integer, and FP data leads to an SDC error with 18%, 45%, and 39% average probability, respectively.*

The fault injection results indicate that a large portion of injected faults lead to an SDC error in the HPC GPU programs. This shows the importance of detecting SDC errors in GPGPU.

In the HPC GPU programs, the SDC error ratio (18-45%) is higher than that observed in CPU programs (<2.3% according to [14]). On the other hand, the failure (application crash/hang) ratio in the HPC GPU programs is lower than that in CPU programs (see Figure 1). The observed differences have two causes. *(a) The lack of fine-grained error protection in GPUs*. Unlike to modern CPUs, GPUs do not have a page-granularity memory access permission checking that can detect many errors (e.g., corruption of a memory address). This is because of shared memory model, hardware cost, and simplicity of runtime software in GPU. *(b) The massive use of FP data in HPC programs*. In the benchmark HPC programs, FP data occupy 3-6 orders of magnitudes larger memory space than the pointer and integer data taken together (see Figure 2). Moreover, corrupted FP values are seldom detected by basic hardware protection mechanisms, e.g., divide-by-zero in FP value does not lead to an exception but returns an infinite value.

Observation 2: *A fault in an FP variable rarely leads to a GPU program failure, while faults (e.g., 16-33%) in pointer or integer variables are likely to cause program failures*.

In our measurements, we did not observe a GPU kernel failure due to corrupted FP value[1]. Pointer and integer data are highly fault sensitive. This is true not only in HPC and graphics GPU programs but also in CPU programs [13]. This is because many pointer and integer variables are used as a control data (e.g., to decide program control flow or to compute memory address). Thus, if such variables are corrupted, it can make a large drastic change in the program execution flow, i.e., likely to be detected by basic hardware protections.

We did not notice any SDC errors caused by a single-bit error in 3D graphics programs. In graphics program, SDC error is defined as a user-noticeable corruption in video output data. This is because graphics program has a high frame rate (e.g., 30fps) and a transient fault typically makes a small

[1] While perhaps rare such scenario is still possible. For example, if there is a data-flow from an FP variable to an integer or a pointer variable (e.g., FP data is used to calculate memory address), a corrupted FP value can propagate to a control data and cause a failure.

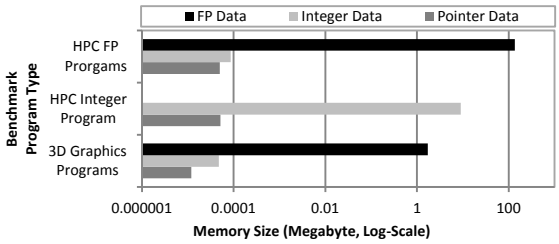


Figure 2. Data type vs. Memory size.

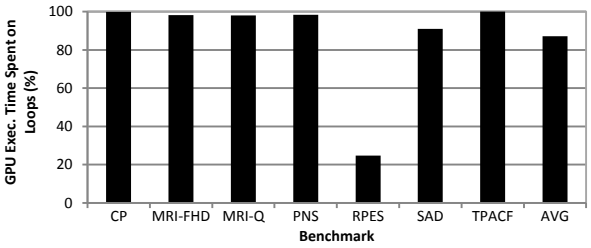
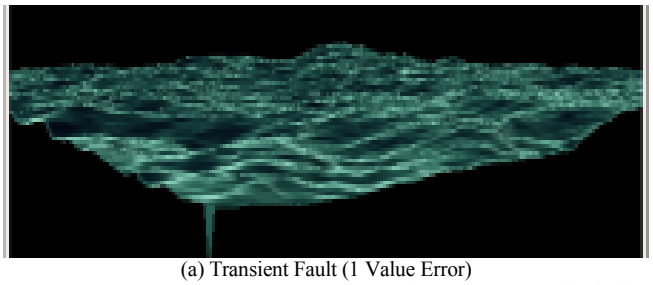


Figure 4. Percent of execution time on loops in HPC GPU programs.



(a) Transient Fault (1 Value Error)

(b) Intermittent Fault (10,000 Value Errors)

Figure 3. Impact of faults in a 3D graphics program on GPU.

change in just one frame. Figure 3(a) shows a video frame of the ocean-flow program that is corrupted by a single-bit fault in its input data stream (a spike in the image is due to the injected fault).

Observation 3: *3D graphics programs can experience SDC errors when exposed to a longer duration fault in GPU.*

The impact of an intermittent fault having a long duration time can be significant even in 3D graphics programs. In the ocean-flow program, corruptions of 10,000 values form a prominent stripe pattern in the rendered frame image (see Figure 3(b)). These injected 10,000 errors emulate an intermittent fault lasting 80µs on an FPU of a 250MHz GPU with 1 instruction per cycle and 50% of execution instructions using the FPU. Note that the injected errors can also reflect impact of an intermittent fault in a memory module or bus. Adding a detection for SDC errors in 3D graphics programs would allow eliminate the corrupted frames and provide better QoS.

B. Performance

This section characterizes the execution times of loop and non-loop portions of GPU kernels (see Figure 4). This data is obtained by measuring the execution time of GPU kernel with and without loops.

Observation 4: *Loops (for, while, and do-while) form a large portion (>98% in 5 out of 7 programs and 87% on average) of the total execution time spent on GPU.*

Note that many GPU kernels are implementation of loops in original CPU codes. These loops executing on GPU typically have many iterations (e.g., proportional to the input data size) and consequently form relatively larger portions of total execution time. In contrast, non-loop codes are executing in parallel by exploiting thread-level parallelism.

The profiling data suggests that a special care is required when placing error detectors inside loop body. A small increase in the execution time of a loop can largely increase the total execution time (in accordance with Amdahl's law). For example, adding just 5 instructions inside a loop body can degrade the performance of a GPU-side code by 22% if the loop has 20 instructions and the loop forms 90% of the total GPU kernel execution time. For GPU programmers, loops are one of the main optimization targets and thus often have a small number of instructions.

III. Related Work

This section classifies and analyzes existing error detection techniques potentially applicable in the context of this study (see Figure 5). The design goal is to find a high coverage detector without compromising performance.

(i) Naïve full duplication. This basic technique has high SDC error detection ratio (close to 100%) but almost doubles the execution time. Duplication uses either temporal or spatial redundancy. Spatial redundancy is achieved by duplicating GPU hardware. The technique can quickly detect errors; however, synchronizing original hardware and its replica brings ~50% of extra performance overhead together with doubled hardware cost [15]. Software technique can easily create temporal redundancy. *R-Naïve* [11] executes same GPU kernel twice by using two different copies of memory data. *R-Naïve* has a good SDC error detection ratio (~100%) but it also almost doubles the GPU execution time and CPU memory space used to keep input and output data. We found that real GPU failure examples where these existing full duplication approaches cannot detect and tolerate (Section IX).

(ii) Optimized full duplication. This approach utilizes idle hardware resources for processing extra computation brought by the duplication. SWIFT [16] extends and applies an instruction duplication technique (EDDI [17]) to a VLIW-type CPU processor by duplicating backward computation slices for address and data values of all memory write operations. These duplicated instructions are reordered by compiler (or hardware scheduler) before execution to exploit the instruc-

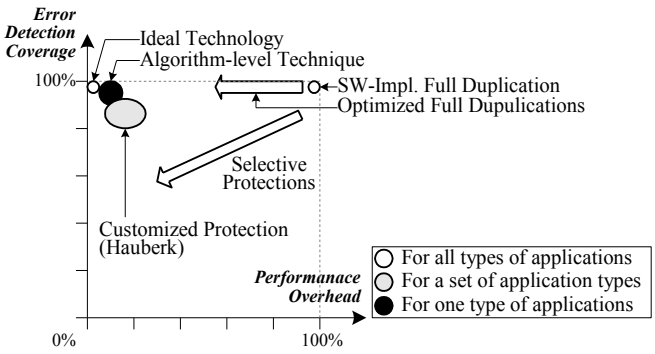


Figure 5. Spectrum of various types of error detection techniques.

tion-level parallelism in VLIW (or superscalar) processors. On an Itanium CPU, SWIFT shows ~100% data error detection ratio with ~41% performance overhead, on average.

The study reported in [11] employs optimized full duplication and shows that the approach is not highly effective for GPU programs in the way it is for CPU programs. After optimizations by exploiting data- or thread-level parallelism in GPU, >84% of performance overhead is shown in widely-used GPU programs. This is because GPU programs are heavily optimized such that original program already uses most of the usable hardware resources in GPU, while the duplicated computation seeks same types of hardware resources or parallelism as the original one.

(iii) Selective protection. This approach selectively protects parts of the program state in order to reduce the amount of extra computation to detect errors. Error detectors are strategically placed in highly error sensitive state. This is motivated by fault injection results, which indicate that many faults in lower-layers of the system are masked and do not manifest in applications [12][13][14].

(a) Fault injection. Fault injection can be used to find error sensitive program state [18]. This is most effective if the size of the program state (e.g., code and data) is small. Otherwise, it can take a long time to analyze the error sensitivity of a large-size program. The large volume of GPU program state (e.g., several gigabytes memory data and several 100 or 1,000 threads) can make the fault injection approach impractical if fine-grained (e.g., a data word) sensitivity analysis is needed.

(b) Static complier analysis. Compiler-based heuristic algorithms can quickly select protection target state even in large-size programs by using static source code analysis. For example, an early technique [19] analyzes the number of possible uses of each program variable and selects a certain number of variables from one with the largest possible uses. This technique detects 41% of SDC errors with 33% performance overhead when applied to CPU programs. Another technique [20] excludes program states from the protection if errors in the state can quickly lead to the program crash.

(c) Dynamic program analysis. This derives and selects likely program invariants by profiling and monitors selected invariants at runtime. For example, if a variable always contains a value between *min* and *max* during profiling, this generates an error detector to check whether the value of this variable is in the identified boundaries [21]. Because profiling uses a limited number of input data, the derived detectors may lead to false positives and that can be addressed by an on-line diagnosis [22].

(iv) Algorithm-level techniques. Error detection techniques designed and optimized for a particular type of algorithm or program are usually highly efficient in terms of error detection coverage and performance overhead. For example, a technique [23] customized for matrix multiplication algorithms detected ~99% of SDC errors with only a small amount of overhead. A software technique [24] customized for GPU global memory errors can detect memory errors with a negligible overhead in compute-intensive applications.

IV. GPU *HAUBERK*

HAUBERK generates customized error detection and recovery routines for GPU programs by leveraging common performance and reliability characteristics of GPU programs.

A. Design Principles

The key principles used to design error detector derivation algorithms in *HAUBERK* are:

Principle 1: *HAUBERK customizes error detectors by using profiling information of common HPC GPU programs in order to minimize the impact on performance.*

HAUBERK uses two types of error detectors for loop and non-loop portions of codes in GPU program. This is motived by the aforementioned loop execution time measurement data. Considering the small contribution of non-loop codes to the overall execution time, strong error detection techniques are designed for non-loop codes. On the other hand, error detection techniques for loop codes are designed and optimized to minimize the performance impact (e.g., by adding only two addition instructions inside a loop).

Principle 2: *HAUBERK selectively protects the program state where errors in other states are likely to propagate*.

The *HAUBERK* loop error detector selectively protects program states where computation of the state directly or indirectly uses many other variables. This means errors in these variables are likely to propagate to protected states and thus are likely to be detected by strategically placed error detectors. The detection accuracy is improved by customizing loop error detectors for common patterns in FP value distributions.

Principle 3: *HAUBERK places error detectors by considering the recoverability of errors (i.e., urgency in error detection to enable and support safe error recovery).*

HAUBERK defers placements of error detectors as long as possible by taking advantage of inherent hardware-enforced error isolation between GPU and CPU (i.e., provided by private memory and explicit communications of GPU and CPU states, see Figure 6). Many errors occurring in GPU-side programs are detected by basic hardware protection before propagating and harming the availability of CPU-side control software (e.g., OS). For example, GPU runtime can detect all GPU kernel crashes by default. Thus, we focus on detecting SDC errors in the output of GPU kernels because this is a practically feasible error propagation path from GPU kernel state to CPU-side program state. Then, in order to further

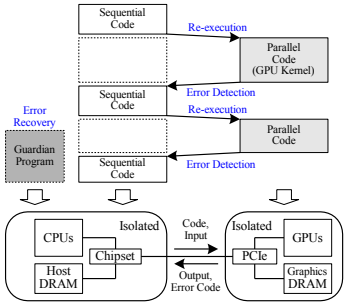


Figure 6. Isolation execution and deferred checking model of *Hauberk*.

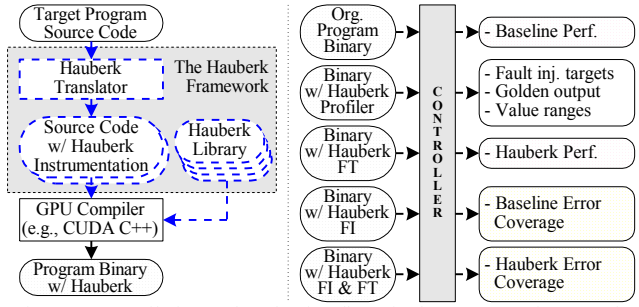


Figure 7. Compilation and evaluation flows in the *HAUBERK* framework.

reduce the performance overhead, we defer the placement of error detectors in GPU kernel code as long as possible that avoids or reduces coverage overlap between the basic hardware-enforced detectors and *HAUBERK*-generated detectors.

Like any other error detector checking intermediate program state, error detectors in *HAUBERK* can suffer of false alarms (i.e., an error in intermediate program state does not propagate to final program output nor makes an observable change in the output). In order to enable a diagnosis of false alarms, we defer reporting detection of suspicious behavior until the end of a GPU kernel execution. If the kernel completes without a failure, its output data is reported to the CPU code together with the error detection result. Any reported error triggers a recovery process in the CPU codes that can identify false alarms by reexecuting the GPU kernel and comparing the returned outputs.

B. Framework

Figure 7 depicts a compile flow of the *HAUBERK* framework. This framework uses the source code of a target GPU program (in CUDA C++) as an input and places hooks in the source code by using a source-to-source translator (an extension of CETUS [27]). This instrumented source code is compiled by using a GPU compiler (CUDA CC) with a proper *HAUBERK* library. This source code mutation makes *HAUBERK* easily applicable to other types of programs (e.g., OpenCL). Note that this mutation can also be efficiently done by the programmer (e.g., testing or releasing engineer) even if he does not have a good understanding of the semantic of target program as long as he is familiar with the syntax of the programming language used and *HAUBERK* instrumentation rules.

A *HAUBERK* library is a user-level C library. This library defines a collection of variables and functions for codes added by the *HAUBERK* translator. Four types of libraries exist: profiler, FT (fault tolerance), FI (fault injector), and FI&FT. We generate a program binary file using each of these libraries (see Figure 7). Specifically, the original program binary is used to measure baseline performance. A program binary with the *HAUBERK* profiler profiles value ranges of variables protected by loop error detectors, derives all fault injection targets, and gets the output of the golden run. A program binary with *HAUBERK* FT is used to evaluate the performance overhead of placed *HAUBERK* error detection and recovery routines. A program binary with *HAUBERK* FI is used to analyze error detection coverage and error sensitivity of baseline program. Finally, a program with *HAUBERK* FI&FT is used to evaluate the error detection coverage of placed *HAUBERK* fault tolerance routines. We use a GUI-based controller program to automate this evaluation process when many experiments are needed (e.g., for fault injection).

Places where *HAUBERK* translator adds or mutates source codes are summarized in Table I. It shows the exact changes made by the *HAUBERK* translator depending on the type of used library. More specific descriptions on these transformations are provided in Section V, VI, and VII.

V. ERROR DETECTION

This section describes the error detector derivation algorithms for non-loop and loop codes of GPU kernel. Note that a GPU kernel can have one or more loops with non-loop codes before, after, and between these loops. Many variables defined in non-loop codes are control data (e.g., pointers, constant input data, and data for control-flow conditions), while many variables manipulated inside loops are streamed input and output data.

A. For Non-Loop Code

The definitions of all virtual variables defined in non-loop codes are duplicated in source code. In this paper, virtual variable means a subset of the live range of program state where the subset has one definition and multiple uses.

A naïve variable-granularity duplication can duplicate the definition of virtual variable and check the original and duplicated variables after the last use (or the immediate post-dominator of last uses) as exemplified in Figure 8(b). This can largely increase the register pressure (e.g., by two times) because the duplicated variable has the same live range as the original variable. Note that in the GPU, if physical registers allocated to each thread are insufficient, register spill operations occur, which slows down the performance due to memory accesses (e.g., to an on-chip cache or an off-chip DRAM).

HAUBERK duplicates the definition of the virtual variable and immediately checks the original and duplicated variables (see Figure 8(c)). This check is done to detect errors that may occur during the computation (e.g., in ALU or FPU). To detect errors occurring after this computation (e.g., errors in register file), we update the checksum variable by XORing the original variable value to it. This checksum update is done right before the comparison operation to prevent losing errors that occurred between the comparison and the check-

TABLE I. DESCRIPTIONS OF INSTRUMENTATIONS USED FOR *HAUBERK*.

Location \ Lib.	*FI (Section VII)*	*Profiler (Section V.B)*	*FT (Section V.A., V.B., VI)*
[CPU] Top of the main file	Includes a header file for *HAUBERK* libraries		
[CPU] Entry of main()	Initializes the control block		
	The control block is for the location, time, and type of fault injection target	The control block is for profiled value ranges and execution counts	The control block is for value ranges, detection results, and outliers
[CPU] Exit of main()	Stores fault activation result to a file	Stores profiling results to a file	Stores updated value ranges to a file
[CPU] Before launching GPU kernel	Copies the control block from CPU to GPU		
	-		Notifies this to guardian process and calls a checkpoint library (option)
[CPU] After GPU kernel launch	Waits until the kernel completion and copies the control block back from GPU to CPU		
	-		Calls an error recovery function
[CPU] GPU kernel function	Adds a pointer variable for the control block as a function parameter in GPU kernel function prototype and its caller(s)		
[GPU] After definition of virtual variable in GPU non-loop	Calls a library function with an identifier, pointer, type, and used hardware components of variable defined in previous statement		Updates a checksum variable, duplicates the definition, and checks original and duplicated variables
	To inject a fault into a defined variable at a designated time of execution	To count execution count per variable	
[GPU] After def. of virtual variable in GPU loop	Same as "After definition of virtual variable in GPU non-loop" field	Adds two addition statements for each protected target virtual variable (one for target variable and the other for counter) and merges the counters if possible	
[GPU] Before loop in GPU kernel	-	Defines accumulator and counter variables for each protected loop variable	
		-	Updates the checksum var. if needed
[GPU] After loop in GPU kernel	-	Profiles value ranges of accumulated variables divided by their counter	Checks accumulated variable value ranges and updates the checksum var.
[GPU] Exit of GPU kernel	-	-	Checks the checksum variable

sum update (see Figure 8(c)). The checksum variable is updated (i.e., XOR) once again using the original variable after its last use or the immediate post-dominator of last uses (e.g., after the loop in Figure 8(c)). The checksum variable is 4 bytes. If a variable size is not 4 bytes, it is aligned by four-bytes for XOR operations. This checksum variable is shared for all duplicated virtual variables in same kernel (i.e., even in nested functions). This variable still shall be zero at the kernel exit because it is XORed twice with each and every duplicated virtual variable value.

This checksum-based duplication avoids a large increase in the register pressure. Only one checksum variable is added per kernel because one checksum variable is used for multiple virtual variables. The duplicated variables are alive only for two statements (i.e., one for its definition and the other for checking). The increase in the live range of the original variable in the presented technique (e.g., from last uses to the immediate post-dominator if there are multiple last uses) is same as that of the naïve duplication technique. Register pressure control in this duplication and checksum technique efficiently leverages a common characteristic in GPU architecture that memory operations are more expensive than computation operations.

The derivation algorithm of non-loop error detectors has five steps:

(i) Update checksum. After the definition of each virtual variable in non-loop codes, the algorithm inserts a statement to update the checksum by using the defined virtual variable value. *(ii) Duplicate computation*. This step duplicates the definition statement of the target virtual variable. Another variable (i.e., temporally allocated in a register) is used to keep the duplicated computation result. *(iii) Check computation result*. This step inserts an *if*-statement to compare the original and duplicated virtual variable values. The live range of this duplicated variable ends here. Although this *if*-statement is a point of control-flow divergence, because all threads in a same warp (i.e., unit of thread scheduling in GPU) make the same control-flow decision if there is no

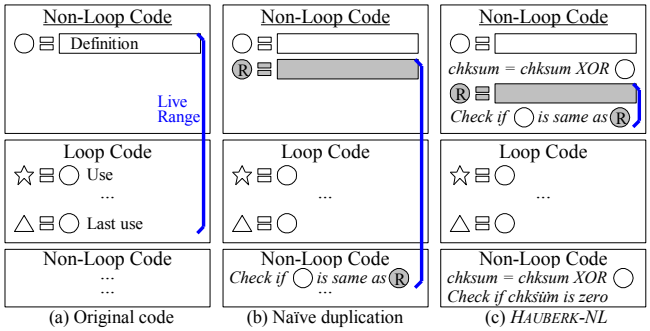


Figure 8. Duplication techniques for non-loop codes where statements marked as gray symbols or italic texts are added for error detection.

fault, this does not introduce a large performance or scheduling overhead. *(iv) Update checksum*. This step inserts an XOR statement to update the checksum variable again by using the original virtual variable. The inserted location depends on the number of uses of the original virtual variable. For example, if the variable is used but not updated inside a loop, this algorithm inserts an XOR statement after the loop. If the variable is updated inside a loop, this XOR statement is inserted right before the loop (i.e., introducing an uncovered window). Note that variables updated inside loops are protected by error detectors for loops described in Section V.B. *(v) Validate checksum*. This step inserts an *if*-statement as the last statement of the GPU kernel to check whether the checksum is zero. A statement is added to set an SDC error bit at runtime if the checksum is non-zero.

The described algorithm repeats the steps from (i) to (iv) for all virtual variables defined outside of the loops. In the case of function parameters, the checksum is updated only (i.e., without duplication) at the entry and exit of the kernel function if the parameter is not modified inside the kernel. This can detect corruptions in parameters. If a parameter is updated inside the function, its second checksum update is done before the update statement, and the updated parameter is treated as another virtual variable (i.e., protected separate-

ly). The same derivation rule applies to memory load expressions and statements.

The delivery of potential error detection report from GPU to CPU is done by using an object in memory (namely, control block). CPU-side program allocates a control block in its memory, copies the allocated object to GPU memory, and delivers the pointer of copied object as a parameter of GPU kernel. Placed error detectors (i.e., added *if*-statement) use this passed control block and marks detection results. If the GPU kernel completes normally, the CPU-side code copies this control block back to CPU memory and tosses it to the error recovery engine described in Section VI. This control block also delivers other information between CPU and GPU (e.g., to configure loop error detectors) as described in Section V.B.

B. For Loop Code

We present value-accumulation-based range checking for loop codes. Derivation of this error detector has four steps:

(i) Select target variable for protection. Among all virtual variables defined inside a target loop, we first select self-accumulating virtual variables. This is because these variables do not need any extra code added inside the loop for protection. We then exclude virtual variables that have forward dataflow dependency to these selected variables from the dataflow graph of all virtual variables inside the loop.

Among the reminder of virtual variables, we select a virtual variable with the largest cumulative backward dataflow dependency. As shown in Figure 9, a cumulative backward dataflow dependency means the number of virtual variables defined inside a loop and unprotected by non-loop error detectors that can directly or indirectly be used to compute the target virtual variable. Thus, a larger cumulative backward dataflow dependency means a higher chance of propagation of errors in other system states to the target virtual variable. If a technique is available that can detect even small corruption in the target variable, this can cover errors in the program state by only checking a few variables.

Figure 9 exemplifies a data-flow graph of a loop in a GPU kernel that is computing a *coulombic potential function*. A circle means a binary or unary operator, and both box and ellipse mean either a virtual variable or a temporary variable where the name of temporary variable starts with T. A temporary variable is used for virtual variable defined by using multiple binary or unary operations, and each operation has an intermediate program state in register or memory. In this example, two output variables (i.e., either live after the loop or written to memory) exist that are marked as black boxes. The cumulative backward dataflow dependency of *energyx1* and *energyx2* are 12 and 13, respectively, including the memory load data but not the constant (i.e., 1.0 in the figure). Here, we exclude five virtual variables that are not modified inside the loop and are protected by non-loop error detectors (i.e., black ellipses in the figure). Thus, we first select *energyx2* for protection.

Users can specify the maximum number of virtual variables (Max_{var}) that can be protected by these loop error detectors. Note that Max_{var} counts self-accumulating variables. If Max_{var} is higher than one, this selection is repeated Max_{var} times. Before repeating this selection process, we remove the previously selected virtual variable(s) and other virtual variable(s) having forward data dependency to the previously selected ones from the dataflow graph. This is to select and protect another virtual variable that can cover the largest number of previously unprotected (either directly or indirectly) virtual variables. Note that this repetition eventually terminates because there is only a finite number of virtual variables in any loop.

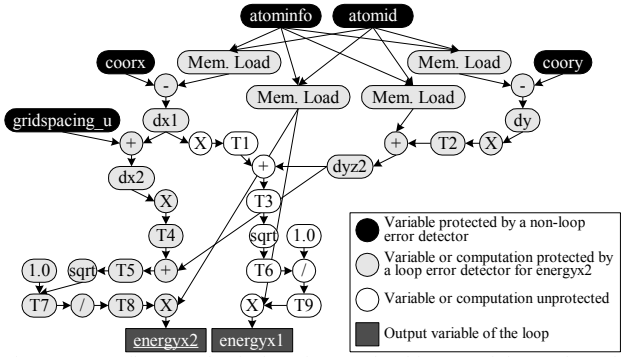


Figure 9. Dataflow graph of a loop in a coulombic potential GPU kernel.

(ii) Generate value accumulator code. The placed error detector accumulates the data value of each protected virtual variable in every loop iteration. This step is skipped if a protected variable is a self-accumulator. For each protected virtual variable, another variable is defined with the initial value of zero (e.g., *float accumulator = 0.0;*) right before the loop. Using this accumulator variable, an accumulation statement is added right after the definition of the protected virtual variable (e.g., *accumulator += energyx2;* if the protected variable name is *energyx2*) inside the loop.

(iii) Generate accumulation counter code. An addition statement is added to count the number of accumulation operations for each accumulator variable. The *HAUBERK* translator defines an integer variable right before the loop (e.g., *int iterator = 0;*) and adds an integer addition statement (e.g., *iterator++;*) inside the loop right after the placed accumulator(s). Even when many accumulator variables are used, these variables often have an identical control-flow path (e.g., common case is accumulation count is same as the loop iteration count). In this case, these variables can share one accumulation counter. If the accumulation count is expected to be same as the loop iteration count, we maintain this custom accumulation counter because this is also used to detect some loop control-flow errors (i.e., errors in loop iterator, termination condition, or iterator manipulation operation).

(iv) Generate error checking code. An error checking routine is added right after the loop code. This added routine calls a function defined in the *HAUBERK* FT library (i.e., *HauberkCheckRange(...)*) by using the averaged accumulator value (e.g., *accumulator*/*iterator*) and the pointer to control block. The called function checks whether the current accumulation value is within the profiled value ranges (i.e., specified in the control block). If the value is outside of ranges, this function calculates new ranges (i.e., assuming it is a false positive) and stores this to control block together with setting an SDC error bit. The updated ranges are used by the recovery engine

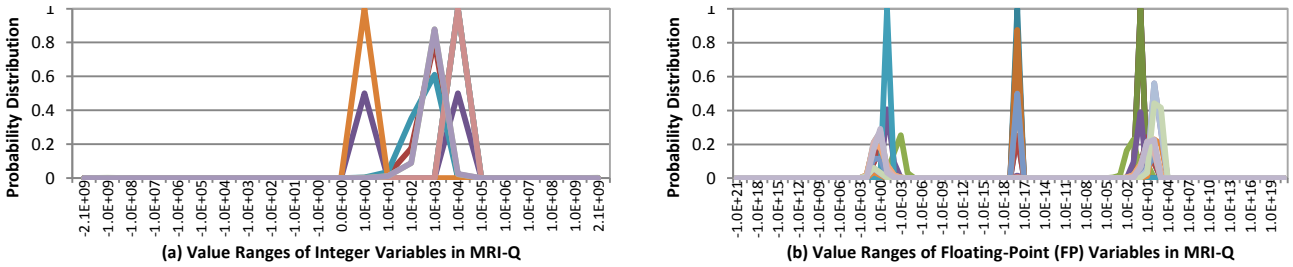


Figure 10. Value range distributions of integer (a) and FP (b) variables in the *MRI-Q* program executing on a GPU device.

as a part of its on-line learning process. In the FT library, the function called at the entry of *main()* loads the profiled value range from a file and configures the control block for loop error detectors. Another function called at the exit of *main()* stores the updated value ranges to the same file if false alarm is detected.

The detection code for an example in Figure 9 is as follows where bold texts are added for *HAUBERK* protection. It also has an added code after the loop to check the loop iteration count (i.e., *HauberkCheckEqual(...)*). Often, we can calculate the loop iteration count (e.g., loop iteration count is *MAX* for a loop, i.e., *for(int i=0; i<MAX; i++)* { … }). This loop iteration count is treated and checked as an invariant of the program. Even if the calculation of the loop iteration count is complex (e.g., using two conditions), we find that it is still feasible in many cases to drive a statement that can dynamically calculate the iteration count. For example, for a loop *for(int x=0, y=0; x<A && y<B; x++, y++)* { … }, the loop iteration count is same as the minimum of A and B. Also if a condition variable can be changed inside the loop, the iteration count is computed and stored in a variable before the loop.

```
float accumulator = 0; int iterator = 0;
for (atomid=0; atomid < numatoms; atomid++) {
  …
  accumulator += energyx2;
  iterator++;
}
HauberkCheckRange(controlblock, 0, accumulator / iterator);
HauberkCheckEqual(controlblock, 0, iterator, numatoms);
```

If a hardware fault makes a large change in the averaged accumulated value, this is likely to be detected by this value range checking. On the other hand, if an error makes only a small change in the value of protected variable, this will not be detected by the checking as far as the corrupted value is within the checked value ranges. Note that this can also be a case that the error also did not significantly impact the program output so to cause an SDC error.

The use of value range checking in GPU programs is motivated by our measurement data. A strong correlation is observed in values stored in or computed for a same program variable in many HPC GPU programs. This strong correlation is observed in both integer and FP data (see Figure 10). Figure 10 shows the value distribution of integer and FP variables in an HPC GPU program (*MRI-Q*). Each graph line represents the value distribution of a single variable, and *x-axis* means integer (or FP) numbers that can be encoded by 32-bit integer (or FP) variable where *1.0E+N* means 1^N and *1.0E-N* means 1^{-N}. Integer values computed by the same code fragments are likely to be in adjacent two units of power of 10s. Most of these graph lines have a sharp peak higher than >0.5. This means that >50% of values computed for the same variable are likely to be in a single unit of power of 10s. Similar characteristics are observed in other HPC GPU programs.

Note that variables in the same program have relatively similar correlation points in both FP and integer data. This is because these variables have direct or indirect data flows to each other and thus their values are correlated.

An important finding is that many FP variables have three correlation points. Two correlation points are in negative and positive numbers with a similar magnitude, and the other point is in close to zero. Values in each correlation point are strongly correlated to each other (e.g., most of correlation values have same order of magnitude). Considering the wide value space that an FP variable can encode (e.g., approximately $2^{-126} \sim 2^{128}$ for single-precision positive FP numbers), a typical FP program uses a small fraction of the available FP value space, making this value range checking effective in FP data.

Based on this finding, the value range profiling algorithm is specifically designed to detect up to three correlation points. We set two default threshold points (e.g., at -10^{-5} and 10^{-5}) and treat any value observed between these points as a value correlated to the correlation point at zero. Other values outside of these threshold points are correlated to the correlation point in positive or negative numbers. We sum up the sizes of value spaces of all value ranges identified by this profiling. We then change the two threshold points (i.e., by multiplying either 10 or 0.1 to examine its neighbors) and repeat the same profiling. This process is repeated if the calculated total value space is smaller than that measured in the previous run.

If a potential SDC error is detected, this error detector does not terminate the GPU kernel. This instead defers error reporting until the kernel completion. If the kernel causes a failure, it validates that this detection is not a false alarm.

VI. ERROR RECOVERY

This section describes retry-based error recovery in *HAUBERK*, which can diagnose and tolerate errors in GPU.

(i) Guardian Program. A guardian program is used as a parent process of program instrumented by using the *HAUBERK* framework (Figure 6). This is because a failure in the GPU kernel can make its host-side CPU program crash depending on the failure type (i.e., as a result of conservative fail-stop policy for crucial failures). When the GPU program terminates, this event is communicated to its parent guardian process by the OS kernel. For example, the *SIGCHLD* signal

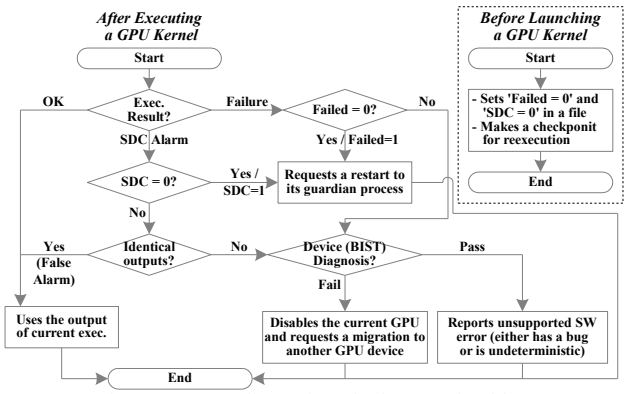


Figure 11. Error diagnosis and tolerance algorithm.

is sent in Linux OS. After checking the return value of its child process, the guardian process marks this failure information in a file and restarts the GPU program. If the failure is repeated twice in the same GPU kernel using the same input data (see Figure 11), the guardian process runs a program to diagnose intermittent or permanent faults in GPU device.

Optionally, instrumented GPU program can use a checkpoint library [25] in order to reduce the recovery time. A checkpoint can be made before launching a GPU kernel, and the guardian process can restore the latest checkpoint upon detection of a GPU program failure.

The guardian also offers a preemptive detection of GPU kernel hang (or execution delay). For example, if the execution time of a GPU kernel is not only *T* (i.e., 10) times longer than its previous execution time but also a certain time interval (e.g., 1 minute), the guardian process assumes this as a hang or time delay error and preemptively sends a kill signal to its child process. *HAUBERK* FT library functions called before and after launching GPU kernel control a timer to report the measured execution time of a GPU kernel to its guardian process using an inter-process communication primitive. Our experiment data in Section IX identifies many cases where this preemptive detection is useful.

(ii) Diagnosis of False Alarms. *HAUBERK* loop error detectors may result in both false positives and negatives. (a) A *false positive* occurs when a new input data produces a value for accumulator variable that is not in the profiled value ranges. This is because used value ranges are derived by profiling that only uses a limited number of input samples. Using many representative samples in profiling can reduce the likelihood of false positives but it cannot guarantee complete removal of false positives in many real world applications. (b) A *false negative* occurs when the averaged accumulated value is within the profiled value ranges after the program experiences a fault, while the program output is largely corrupted and violates its correctness requirement.

False SDC detection alarms are identified by reexecution. When loop error detector reports a potential SDC error, the recovery assumes this is a false positive and reexexcutes the GPU kernel for diagnosis (see Figure 11).

(a) False alarm. If the reexecution also raises an SDC alarm and its output is identical to the original output, these two are likely to be false alarms (i.e., false positive). Here, *identical* means each value in the output of one execution is same as the corresponding value of the output of the other execution if the output of GPU program is always deterministic. If a nondeterministic GPU program is used, output values showing a certain degree of difference (i.e., more than twice of the output correctness requirement – a conservative approach is used because the golden run output is not available) are still treated as identical. Up on a detection of a false positive, we store the updated value ranges to a file (i.e., a part of on-line learning process).

(b) SDC error due to transient or short intermittent fault. If the reexecution terminates normally and does not raise an SDC alarm, we assume the alarm raised in the first execution is due to transient or intermittent fault (i.e., removed before the second execution). In this case, the reexecution result is taken.

(c) SDC error due to long intermittent or permanent fault. If the reexecution also raises an SDC alarm but its output is not identical to the original execution output, we execute a GPU program that is specifically designed to produce multiple sets of output data by examining various parts of GPU hardware. The functionality of this program is similar to built-in self-test (BIST). If this program detects a hardware fault, the current GPU device is disabled and another device in the node or cluster is used for reexecuting the current GPU program. A daemon process is periodically running this program on disabled GPU devices with a time delay ($T_{backoff}$). Here, $T_{backoff}$ is doubled after every execution of this program. If the error was due to an intermittent fault, this configuration reduces the utilization of GPU device, and once the fault is removed, this program can re-enable the GPU device.

(iii) Configuring Loop Error Detector. This false alarm diagnosis can calculate the false positive ratio. If the current false positive ratio of a *HAUBERK* loop error detector is higher than a threshold (e.g., 10%), the recovery engine increases the parameter *alpha* (e.g., by multiplying 10) for the error detector. If the false positive ratio is smaller than another threshold (e.g., 5%), it reduces the *alpha* (e.g., divides by 10) as far as *alpha* is larger than or equal to 1. Specifically, the maximum value of each value range is multiplied by *alpha*, and the minimum value of each value range is divided by *alpha* if these maximum and minimum values are positive numbers. The use of loose value ranges can reduce false positives but at the same time can increase false negatives. This tradeoff between false positives (i.e., performance overhead due to reexecution) and false negatives (i.e., detection accuracy) is analyzed in Section IX.

VII. DEPENDABILITY EVALUATION FRAMEWORK

This section presents a SWIFI framework for commodity GPU devices. In order to optimize and evaluate the dependability of programs instrumented with protection mechanisms derived by *HAUBERK*, a dependability benchmarking tool is required. However, there is no published fault injection tool for real GPU hardware [26]. Thus, we built a SWIFI toolset to emulate single- and multi-bit transient faults in GPU processor and memory. The fault injection framework does not

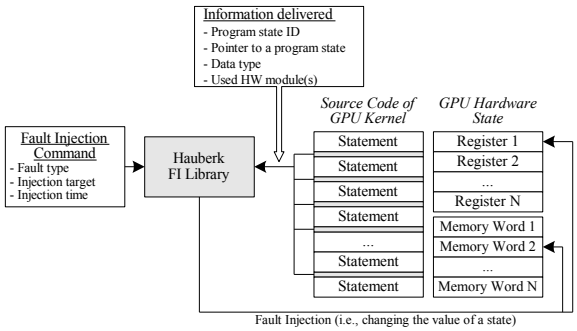


Figure 12. A GPU kernel with *HAUBERK* fault injection codes.

require any modification in GPU hardware and hence is applicable to commodity GPUs.

A source code mutation (i.e., embedding error injection code) technique is used to efficiently control fault injection target (i.e., a state of a thread running on one of several 100 GPU cores). Although our implementation is done for CUDA C++, the framework can be easily ported to other parallel programming languages (e.g., OpenCL).

For each GPU kernel statement that can change a program state, the *HAUBERK* source-to-source translator derives the symbol name and data type of a variable or program state that could be changed by the previous statement. A function call statement is added after each program statement[2] to call a fault injector library function (see Figure 12). The arguments of the added function call include: the variable identifier, pointer to the derived variable, identifier of the data type corresponding to the target variable, and hardware components used by the preceding statement. The hardware components used are statically derived by analyzing the operation types, e.g., ALU and FPU for integer and FP expressions, respectively. The *HAUBERK* provided fault injection (FI) library changes the value stored in the derived variable as specified by the fault type. If the derived variable is for a FPU register, this register value is copied to an ALU register. This is because the fault injection uses, for example, a logical XOR operation that is only supported by ALU. The changed value in the ALU register is then copied back to the original FPU register.

The emulated hardware faults can be classified by using two metrics: fault location and type.

(i) Fault location. Hardware faults can occur in any transistor (or component) in GPU. A fault is either masked (e.g., when the faulty transistor is not in use) [12] or it propagates to a software-visible architecture state. A fault propagated to an architecture state is a failure in the microarchitecture layer and can be treated as a fault in the architecture or software layer. The SWIFI toolset emulates faults in this architecture-layer and evaluates their impacts on the reliability and correctness (i.e., data integrity) of the application software.

By using the location of the fault in the software-visible architecture state, hardware faults are classified into several types: (a) faults in the core ALU, (b) faults in the core FPU, (c) faults in an SM (streaming multiprocessor in NVIDIA GPU [7]) register, and (d) faults in SM scheduler. These faults are emulated as errors in the architecture state, i.e., program variables or control-flow decision. For example, a fault in ALU is emulated by changing the computation result stored in a program variable of an operation that uses ALU. We assume memory data transfers between GPU core and its cache and memory are reliably done because memory data and data paths to on-chip and off-chip memory are protected in the latest versions of GPU devices [7].

(ii) Fault type. Fault can corrupt one or more transistors, and an error in a single transistor can propagate and corrupt multiple bits in its architecture state. We model both single- and multi-bit errors in the architecture state. For example, we emulate multi-bit errors in GPU register file. Although the latest GPUs support a SEC-DED ECC for register file, multi-bit errors can occur in register file and propagate to program states without being detected by a SEC-DED ECC. In practice, supporting stronger ECC has hardware cost issue. For example, while a SEC-DED ECC causes ~22% extra space overhead when the protection unit is 32bits (e.g., register), a DEC-TED ECC (i.e., correcting double-bit errors) introduces ~41% space overhead if the protection unit is the same.

VIII. EXPERIMENTAL METHODOLOGY

We use a GPU cluster where each node has an NVIDIA Tesla S1070 (4 GT200 GPU and 4GB memory per GPU) for the experiments. The benchmark programs (descriptions are in [4]) used for the measurements reported in Section II are used to evaluate application dependability.

In order to assess the performance overhead, we measure the time spent on GPU kernels, memory copies, and CPU-side codes. GPUs operate in synchronous mode when conducting this measurement. In practice, the measurement focuses on the GPU kernel execution time because the time spent on executing the CPU code and memory copy operations is similar regardless of used error detection technique. Note that even if the memory copy traffic is doubled, this does not increase the DMA time largely as long as the data size is not excessively large as compared with the memory copy bandwidth between CPU and GPU memory (e.g., 4GB/s in PCI-E v2.0 with 8 lanes).

For the dependability evaluation, we emulate hardware faults (in various parts of hardware components) that propagate and corrupt single or multiple bits in an architectural state (register or memory). 20-50 virtual variables are selected in each benchmark program and faults are injected into each of the selected virtual variables. Fifty different error masks (randomly generated) are used for each variable in order to emulate single and multi-bit errors. In total, about 10,000 faults are injected into seven benchmark programs. Specifically, we perform 10,000 different fault injection ex-

[2] Although not common, adding many call statements in the source code of GPU kernel can cause a GPU runtime error if used GPU device does not have sufficient hardware resources. In this case, fault injection target is selected at compile-time. Specifically, the variable identifier of a fault injection target is given as input of the *HAUBERK* translator that adds only one call statement in GPU kernel source code where the added call statement has the given variable identifier as its parameter. This, however, increases the total fault injection experiment time because the target program shall be instrumented and compiled again for each fault injection target.

periments per application where each experiment runs a program and injects only one fault (either single- or multi-bit). Thus, in our experiments, error detection coverage *p* means that a fault in the used GPU programs can be either detected or masked with the probability of *p* if the characteristic of the fault is same as that of the used 10,000 faults.

The observed fault injection outcomes are classified into five types: (i) *failure*, a GPU kernel crash detected by the GPU runtime environment or a GPU kernel hang detected by the guardian process, (ii) *masked*, the output of a GPU kernel satisfies its correctness requirement regardless of the injected fault, (iii) *detected & masked*, the injected fault is masked but error detectors raise an SDC alarm, (iv) *detected*, the output of GPU kernel does not satisfy the correctness specifications and an alarm is raised by error detectors, and (v) *undetected*, if the output does not satisfy the correctness specifications but is not detected by error detectors.

IX. Experimental Results

This section evaluates the performance and coverage of *HAUBERK* in comparison with (i) *Baseline*, without any custom error detection, (ii) *R-Naïve*, full duplication based on reexecuting a GPU kernel twice, (iii) *R-Scatter*, an optimized full duplication exploiting data-level parallelism [11], (iv) *HAUBERK-NL*, *HAUBERK* only for non-loop codes, and (v) *HAUBERK-L*, *HAUBERK* only for loop codes.

A. Performance Overhead

Figure 13 shows the performance overhead of GPU kernels of seven HPC GPU programs (i.e., normalized to the baseline performance) when a same data set is used for training and testing. The average overhead of *HAUBERK* is 15.3%.

HAUBERK shows a significant performance overhead reduction as compared with *R-Naïve* and *R-Scatter*. The average overheads of *R-Naïve* and *R-Scatter* are 100% and 89%, respectively. This shows that the evaluation data reported in [11] holds even in more complex GPU programs. *R-Scatter* has a larger overhead in GPU than similar techniques for CPU program because its duplicated computation seeks same types of hardware resources or parallelism as the original computation, which is already heavily optimized in terms of used resources and parallelism. Note that statement duplication used in *R-Scatter* does not always double the performance overhead because the duplicated statements can be processed by using previously unused resources.

R-Naïve and *R-Scatter* have larger memory overheads than *HAUBERK*, which has only a small memory overhead (i.e., typically <10KB in both CPU and GPU memory spaces). *R-Naïve* doubles the CPU memory space to keep output data of the first and second executions of GPU kernel. *R-Scatter* doubles used GPU memory space and resources (e.g., global/shared memory and partly registers). This means *R-Scatter* is not directly applicable to programs that use more than half of one of these resources. For example, *TPACF* uses more than half of the GPU shared memory (e.g., 16KB total in the used GPU). Thus, we could not compile this program using the *R-Scatter* error detectors.

The average performance overhead of *HAUBERK* on the used benchmark is 15.3%. A large variation is observed in

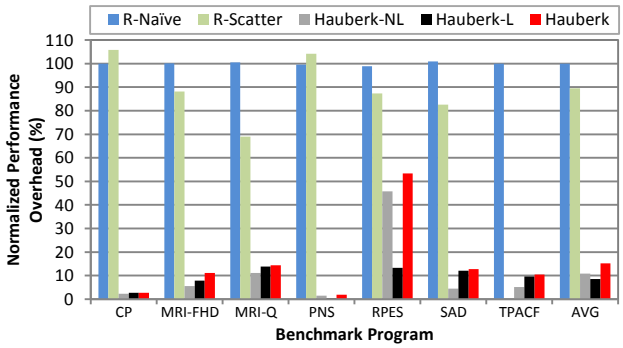


Figure 13. Performance overhead.

the performance overhead of *HAUBERK* on a particular program (*RPES*) where a large portion of GPU codes is sequential (i.e., non-loop). Excluding the performance overhead for *RPES*[3], the average overhead of *HAUBERK* is 8.9% where the minimum and maximum are 1.9% and 14.3%, respectively.

The performance overhead of *HAUBERK* is similar but not a straight sum of performance overheads of *HAUBERK-NL* and *HAUBERK-L*. This is because of common performance overheads (e.g., to deliver the control block between CPU and GPU and to manipulate the control block by placed error detectors).

The overhead of *HAUBERK-NL* depends on the portion of execution time spent on loops. For example, the overhead of *HAUBERK-NL* is exceptionally high in *RPES* because non-loop codes in this program form 75% of total execution time. In some benchmarks (i.e., *MRI-Q* and *MRI-FHD*), the overhead of *HAUBERK-NL* is larger than the contribution of non-loop codes to the total execution time because the duplication increases the register pressure (i.e., consequently increasing memory spill operations) and can interfere with the memory coalescing patterns in original program.

The overhead of *HAUBERK-L* has a relatively small variation because the same number (i.e., Max_{var} = 1) of variables is protected in each loop. The smallest overhead is observed when the protected variable is in integer type (i.e., *PNS*) thanks to the fast integer arithmetic speed. Note also that the overhead of *HAUBERK-L* is relatively small if the program (i.e., *CP*) has a self-accumulating variable (i.e., FP variable) in its loops. The fact that *CP* has larger overhead than *PNS* implies that value-range checker for FP data placed outside of a loop is an expensive operation in terms of performance overhead because FP variable has up to three value ranges to check.

B. Error Detection Coverage

Figure 14 shows the error detection coverage of *HAUBERK* for the benchmark programs and number of error bits when the same input data set is used for training and test runs. On average, 13.2% of injected faults can escape *HAUBERK* error detectors and lead to SDC errors. In other

[3] We find that *RPES* is removed in the recent release of the Parboil benchmark suite because this type of program is not widely used as GPU program (i.e., inefficient due to the large portion of sequential code).

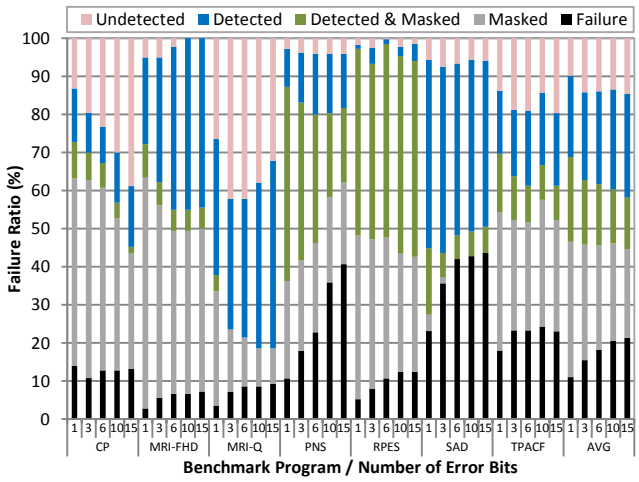


Figure 14. Error detection coverage of *HAUBERK*.

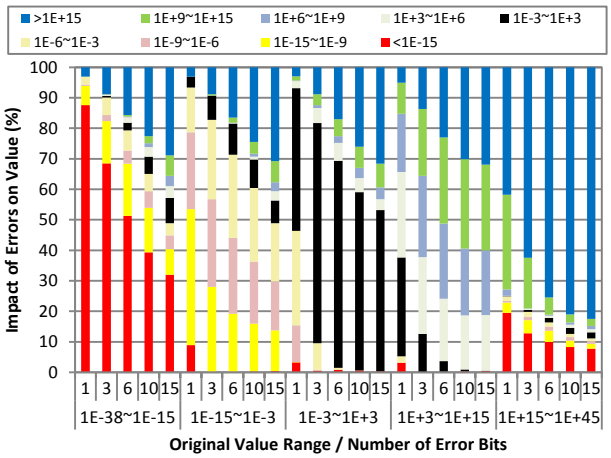


Figure 15. Changes in the magnitude of values after experiencing a fault depending on the orginal value range (of FP data) and error bit count.

words the average detection coverage is 86.8%. If a system experiences two faults during its execution, the coverage of *HAUBERK* would be $(1-(1-0.868)^2) = \sim 98.3\%$, assuming the two faults are independent. In the case of single-bit errors, on average, 35.6% of the errors are masked, 11.0% lead to a GPU program failure, and 21.4% are detected by the introduced error detectors. Out of the remaining 32%, 22.2% are detected but did not violate the application correctness, and 9.8% lead to SDC errors by bypassing the embedded error detectors.

The *detected & masked* error type is an error that changes an intermediate program state but does not make a large corruption in the output significant enough so as to violate the correctness requirements. Both the *detected* and *detected & masked* errors need a reexecution of the GPU kernel to diagnose false alarm because the golden output is not available in practice. This reexecution has relatively small impact on the average execution time of the GPU kernel because this fault can only happen if the GPU faces a hardware fault. In practice, the hardware fault rate is often low enough and the error recovery time is not long enough to impact performance.

The ratio of *detected & masked* type directly depends on the degree of strictness of output correctness requirement of application. For example, this ratio is low in *SAD* (i.e., an integer program) because it does not allow value errors in the output. This ratio is relatively high in *PNS* and *RPES*, where correctness requirements are relatively loose: $Max\{0.01, 1\%|GR_i|\}$ and $2\%|GR_i|+10^{-9}$, respectively. Here, $|GR_i|$ is *i*-th element of the golden output. Note that *MRI-Q* has stricter requirement than these two: $Max\{10^{-4}\text{Max}\{|GR|\}, 0.2\%|GR_i|\}$ where |GR| means all elements in the golden output.

Multi-bit errors typically increase the percentage of program failures and decrease the percentage of masked errors (see Figure 14). This is because when many bits are corrupted, this is likely to make a large value change in both FP and integer data as far as the number of corrupted bits is less than half of available bits. In Figure 15, regardless of an original value range, if the number of corrupted bits increases, the portion for >1E+15 (i.e., value errors more than 10^{15}) gradually increases. This data is obtained by injecting faults into 33 million randomly-generated FP value samples. The same characteristic is observed in integer values.

Note that multi-bit errors do not always bring higher error detection coverage than single-bit errors when *HAUBERK*-generated error detectors are used. Some programs (e.g., *CP*) have lower coverage when many more bits are flipped. This is because multi-bit errors generally have higher non-benign error ratio (i.e., smaller masked error ratio), while many of these non-benign errors in these programs evade the provided loop error detectors. For example, if a corrupted variable is used as a divisor operand that computes another variable protected by a loop detector, a multi-bit error in the divisor operand variable can eventually reduce the protected variable value, i.e., less likely to be detected by the loop detector.

We find multiple cases where the GPU kernel hangs or faces a long execution time delay (i.e., a part of failure type in Figure 14). These failures are undetected by either *R-Naïve* or *R-Scatter*. An example case is when a loop iterator is corrupted (e.g., to a large negative number and the loop terminates if the iterator is bigger than a positive number) and the corrupted iterator does not cause a crash. Another example is specific to the *TPACF* implementation that uses a loop and performs a memory write operation until the write is successfully done and not overwritten by another thread (i.e., checked by reading the data back). If the address of memory write is corrupted to specific address ranges, the loop does not terminate because the corrupted address never returns the write requested value. Failures in these two cases are detected by the guardian process in *HAUBERK*.

C. False Positive

We evaluate the false positives of *HAUBERK* loop error detectors by using different training and test data sets. Four benchmark programs are selected for this evaluation based on the availability of multiple data sets and their representativeness with respect to other programs. Out of 52 datasets prepared for each program, 50 are randomly selected and used for training and the remaining two are used for evaluating the derived detectors. This process is repeated 10 times to calculate average false positive ratio (see Figure 16).

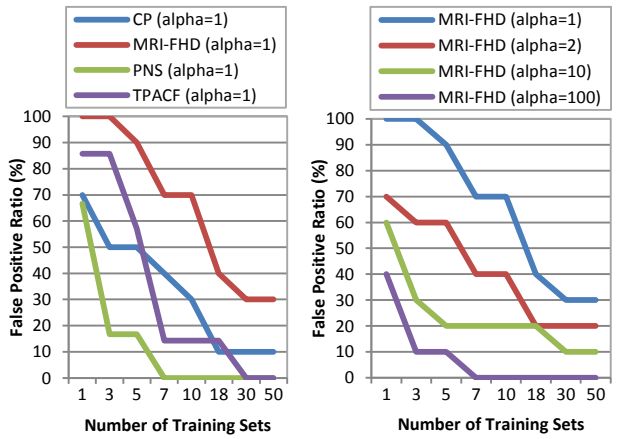


Figure 16. False positive ratio vs. Training count.

We find that the false positive ratio largely varies depending on the program. For example, the measured false positive ratio of *PNS* becomes close to zero after executing seven training sets but that of *MRI-FHD* remains 30% even after running 50 training sets. This is because in the case of *PNS*, the program input represents parameters of a fixed simulation model and thus accurate detectors can be relatively easy to derive. In the case of *MRI-FHD*, the inputs are vectors and the output computation involves multiplication of the different vectors; thus, range-based detectors are not that precise.

In order to address detector imprecision, we investigate dynamic recalibration of the bounds (i.e., *min* and *max* values defining the bounds) used in the range detectors. The approach (described in Section VI(iii)) multiplies the bounds by *alpha*, a multiplication factor derived based on the monitored current false positive ratio.

(a) If alpha = 1. Even when *alpha* is 1, the false positive ratio quickly converges to a ratio less than 10% in the three out of the four evaluated benchmark programs (see Figure 16(left)). These three programs do not need to use the *alpha* larger than 1 (i.e., used 50 training input sets are sufficient). The detection ratio reported in Figure 14 corresponds to the *alpha* equal to 1.

(b) If alpha > 1. In the case of *MRI-FHD*, the false positive ratio does not quickly converge to below 10% if *alpha* is 1 (Figure 16(left)). Using a larger alpha value is needed even after processing more than 50 training input sets. Figure 16 (right) shows the false positive ratio of *MRI-FHD* where the four curves are derived for *alpha* values of 1, 2, 10, and 100. When a large multiplication factor (i.e., *alpha*) is used, the false positive ratio decreases quickly after a small number of training sets. For example, for the *MRI-FHD* application applying *alpha* = 100, the false positive ratio becomes zero after executing 7 training sets. This shows that the adaptive technique to control value ranges used in the detector can efficiently manage the false positive ratios and consequently reduce the performance overhead.

We also analyze the impact of the selected *alpha* value on the detection coverage. The error detection coverage of *MRI-FHD* is 95%, 95%, 82.8%, and 81.6%, when the *alpha* is 1, 1000, 10000, and 100000, respectively. The value of *alpha* only affects the detection coverage of *HAUBERK* loop error detector. None or a small reduction (<0.5% decrease) in the error detection coverage is observed for *MRI-FHD* application when applying a multiplication factor (*alpha*) smaller than 1,000. This implies that the use of a large multiplication factor in the early stage of testing or training does not largely harm the error detection coverage because a fault in an FP or integer value often alters the data by orders of magnitude (e.g., $>10^6$ times, see Figure 15). A large increase (12.2%) in the undetected SDC ratio is observed when the *alpha* is set to 10,000. This threshold *alpha* value is not fixed but depends on multiple factors, including the iteration count of protected loop and the application output correctness requirement.

D. *HAUBERK* Instrumentation Time

We evaluate the instrumentation time of *HAUBERK* error detectors. On average, adding the *HAUBERK* instrumentation takes 81 seconds where the minimum and the maximum are 36 and 112 seconds, respectively, with the Parboil suite. The used machine has two 2.4GHz CPUs and 2GB DRAM and executes a Linux OS. This instrumentation time includes the C preprocessing time, parsing time, analysis time, and transformation time but excludes the time spent on C code beautifier.

The exact time spent on processing the *HAUBERK* transformers (i.e., placing error detectors in the intermediate representation) is 0.7 second, on average. The sizes of total program source code and GPU kernel code of each of the used benchmark programs are 579 and 266 lines, respectively, on average, before C preprocessing. If a source-to-source translator is already used for other purposes (e.g., performance), the *HAUBERK* transformers only add a short delay (e.g., <0.7s, on average, per GPU kernel) to the total compilation time. The *HAUBERK* instrumentation can make small impact even if a target program is big and contains many GPU kernels.

The *HAUBERK* instrumentation is needed only after performance optimization and before testing. When developing an HPC program, most of the development time is spent on optimizing the program performance. After this optimization, developer typically runs an integrated stress testing. Because the *HAUBERK* instrumentation is for runtime fault tolerance, this instrumentation is added just before this final testing.

X. CONCLUSION

This paper analyzed reliability problems in GPGPU platforms, focusing particularly on the design of efficient low-cost detection and recovery mechanisms for handling SDC (silent data corruption) errors. In order to tolerate SDC errors, customized error detection techniques are strategically placed in the source code of target GPU program so as to minimize performance impact and error propagation, and maximize recoverability. The presented *HAUBERK* technique is evaluated using a mutation-based fault injection tool (developed as part of this study) for automated reliability testing of commodity GPU devices. *HAUBERK* offers a high error detection coverage (~87%) with a small performance overhead (~15%).

ACKNOWLEDGEMENT

We would like to thank John Stratton in the IMPACT group who provided data sets for some benchmark programs. This work was supported in part by NSF grant CNS-05-24695; the Gigascale Systems Research Center (GSRC/MARCO); the Department of Energy under Award Number DE-OE0000097; and Boeing Corporation as part of ITI Boeing Trusted Software Center.

REFERENCES

[1] D.B. Kirk and W.W. Hwu, *Programming Massively Parallel Processors: A Hands-on Approach*, Morgan Kaufmann, 2010.

[2] NVADIA CUDA, http://www.nvidia.com/object/cuda_home.html

[3] AMD/ATI Brook+, http://sourceforge.net/projects/brookplus/

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

[5] G. Shi, J. Enos, M. Showerman, and V. Kindratenko, "On Testing GPU Memory for Hard and Soft Errors," *In Proceedings of the Symposium on Application Accelerators in High-Performance Computing (SAAHPC)*, 2009.

[6] I. S. Haque and V. S. Pande, "Hard Data on Soft Errors: A Large-Scale Assessment of Real-World Error Rates in GPGPU," *In Proceedings of the IEEE/ACM International Conference on Cluster, Cloud and Grid Computing, pp. 691-696, 2010.*

[7] NVIDIA Corporation, *NVIDIA's Next Generation CUDA Compute Architecture: Fermi*, White Paper v1.1, 2009.

[8] S.-S. Yoon, B.-K. Kim, Y.-K. Kim, and B. Chung, "A fast GDDR5 read CRC calculation circuit with read DBI operation," *In Proceedings of the IEEE Asian Solid-State Circuits Conference*, pp. 249-252, 2008.

[9] L. Borucki, G. Schindlbeck, and C. Slayman, "Comparison of Accelerated DRAM Soft Error Rates Measured at Component and System Level," *In Proceeding of the International Reliability Physics Symposium*, pp. 482-487, 2008.

[10] C. Constantinescu, "Trends and challenges in VLSI circuit reliability," *IEEE Micro*, 23(4):14–19, 2003.

[11] M. Dimitrov, M. Mantor, and H. Zhou, "Understanding Software Approaches for GPGPU Reliability," *In Proceedings of Workshop on General Purpose Processing on Graphics Processing Units (GPGPU-2)*, pp. 94-104, 2009.

[12] N.J. Wang, J. Quek, T.M. Rafacz, S.J. Patel, "Characterizing the effects of transient faults on a high-performance processor pipeline," *In Proceedings of the IEEE International Conference on Dependable Systems and Networks (DSN)*, pp. 61-70, 2004.

[13] K.S. Yim, Z. Kalbarczyk, and R.K. Iyer, "Measurement-based Analysis of Fault and Error Sensitivities of Dynamic Memory," *In Proceedings of the IEEE International Conference on Dependable Systems and Networks (DSN)*, pp. 431-436, 2010.

[14] W. Gu, Z. Kalbarczyk, R.K. Iyer, "Error sensitivity of the Linux kernel executing on PowerPC G4 and Pentium 4 processors," *In Proceedings of the IEEE International Conference on Dependable Systems and Networks (DSN)*, pp. 887-896, 2004.

[15] J.W. Sheaffer, D.P. Luebke, and K. Skadron, "A Hardware Redundancy and Recovery Mechanism for Reliable Scientific Computation on Graphics Processors," *In Proceedings of the ACM SIGGRAPH Symposium on Graphics Hardware (GH)*, pp. 55-64, 2007.

[16] G.A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D.I. August, "SWIFT: Software Implemented Fault Tolerance," *In Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, pp. 243-254, 2005.

[17] N. Oh, P.P. Shirvani, and E.J. McCluskey, "Error Detection by Duplicated Instructions in Super-Scalar Processors," *IEEE Transactions on Reliability*, 51(1):63-75, 2002.

[18] M. Hiller, A. Jhumka, and N. Suri, "On the Placement of Software Mechanisms for Detection of Data Errors," *In Proceedings of the IEEE International Conference on Dependable Systems and Networks (DSN)*, pp. 135-144, 2002.

[19] K. Pattabiraman, Z. Kalbarczyk, and R.K. Iyer, "Automated Derivation of Application-aware Error Detectors using Static Analysis," *In Proceedings of the International On-Line Testing Symposium (IOLTS)*, pp. 211-216, 2007.

[20] S. Feng, S. Gupta, A. Ansari, and S. Mahlke, "Shoestring: probabilistic soft error reliability on the cheap," *In Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS),* pp. 385-396, 2010.

[21] M.D. Ernst, J. Cockrell, W.G. Griswold, and D. Notkin, "Dynamically Discovering Likely Program Invariants to Support Program Evolution," *IEEE Transactions on Software Engineering*, 27(2):99-123, 2001.

[22] S.K. Sahoo, M.-L. Li, P. Ramachandran, S. Adve, V. Adve, and Y. Zhou, "Using Likely Program Invariants to Detect Hardware Errors," *In Proceedings of the IEEE International Conference on Dependable Systems and Networks (DSN)*, pp. 70-79, 2008.

[23] K. Huang, J. Abraham, "Algorithm-Based Fault Tolerance for Matrix Operations," *IEEE Transactions on Computers*, C-33(6):518-528, 1984.

[24] N. Maruyama, A. Nukada, A., and S. Matsuoka, "A high-performance fault-tolerant software framework for memory on commodity GPUs," *In Proceedings of the International Symposium on Parallel and Distributed Processing (IPDPS)*, pp. 1-12, 2010.

[25] H. Takizawa, K. Sato, K. Komatsu, and H. Kobayashi, "CheCUDA: A Checkpoint/Restart Tool for CUDA Applications," *In Proceedings of International Conference on Parallel and Distributed Computing, Applications, and Technologies (PDCAT)*, pp. 408-413, 2009.

[26] J.W. Sheaffer, D.P. Luebke, and K. Skadron, "The Visual Vulnerability Spectrum: Characterizing Architectural Vulnerability for Graphics Hardware," *In Proceedings of the ACM SIGGRAPH Symposium on Graphics Hardware (GH)*, pp. 9-16, 2006.

[27] C. Dave, H. Bae, S.-J. Min, S. Lee, R. Eigenmann, and S. Midkiff, "Cetus: A Source-to-Source Compiler Infrastructure for Multicores," *IEEE Computer*, 42(12):36-42, 2009.