# Understanding Error Propagation in GPGPU Applications

Guanpeng Li, Karthik Pattabiraman
University of British Columbia
Vancouver, Canada
gpli@ece.ubc.ca, karthikp@ece.ubc.ca

Chen-Yong Cher, Pradip Bose
IBM T.J. Watson Research Center
New York, USA
chenyong@us.ibm.com, pbose@us.ibm.com

***Abstract*—GPUs have emerged as general-purpose accelerators in high-performance computing (HPC) and scientific applications. However, the reliability characteristics of GPU applications have not been investigated in depth. While error propagation has been extensively investigated for non-GPU applications, GPU applications have a very different programming model which can have a significant effect on error propagation in them. We perform an empirical study to understand and characterize error propagation in GPU applications. We build a compiler-based fault-injection tool for GPU applications to track error propagation, and define metrics to characterize propagation in GPU applications. We find GPU applications exhibit significant error propagation for some kinds of errors, but not others, and the behaviour is highly application specific. We observe the GPU-CPU interaction boundary naturally limits error propagation in these applications compared to traditional non-GPU applications. We also formulate various guidelines for the design of fault-tolerance mechanisms in GPU applications based on our results.**

***Keywords—Fault Injection, Error Resilience, GPGPU, CUDA, Error Propagation***

## I. INTRODUCTION

Graphic Processing Units (GPUs) have found wide adoption as accelerators for scientific and high-performance computing (HPC) applications due to their mass availability and low cost. For example, two of the largest supercomputing clusters in use today, namely Blue-Waters [18] and Titan [51] both use GPUs. GPUs were originally designed for graphics and gaming. However, their use in HPC applications has necessitated the systematic study of their reliability. This is because unlike graphics or gaming applications which are error-tolerant, HPC applications have strict correctness requirements and even a single error can lead to significant deviations in their outcomes. The problem is exacerbated by the lack of standard error detection and correction mechanisms for GPUs, compared to CPUs (Central Processing Units, or the main processor). Recent studies of GPU reliability in the HPC context have found that GPUs can experience significantly higher fault rates compared to CPUs [18], [51], and that GPU applications often experience higher rates of Silent Data Corruption (SDCs), i.e., incorrect outcomes, compared to CPU applications [26], [20].

HPC applications typically run for long periods of time, and hence need to be resilient to faults [44]. Further, in supercomputers, hardware faults have become more and more prevalent due to the shrinking feature sizes and power constraints [22]. One of the most common hardware fault types are transient faults [9], [14], which arise due to cosmic rays or electro-magnetic radiation striking computational and/or memory elements, causing the values computed or stored to be incorrect. To mitigate the effect of transient hardware faults, HPC applications use techniques such as checkpointing and recovery. However, these techniques make an important assumption, namely that faults do not propagate for long periods of time and corrupt the checkpointed state as this would make the checkpoint unrecoverable [24], [35], [53]. Unfortunately, this assumption does not always hold as errors often propagate in real applications [36]. More importantly, unmitigated error propagation can also lead to SDCs, which seriously compromise the applications' correctness.

In this paper, we investigate the error propagation characteristics of general-purpose GPU (GPGPU) applications with the goal of trying to mitigate error propagation. Prior work has investigated error propagation in CPU applications [35], [6], and has developed techniques to mitigate error propagation based on their results [36]. However, it is not clear how applicable are these results to GPGPU applications, which have a very different programming model. Other work has investigated the aggregate error resilience of GPGPU applications [26], [20]. While these are valuable, they do not provide insights into how errors propagate within the GPGPU application. Such insights are necessary for driving the design of low-overhead error detection mechanisms for these applications, which is our long-term goal. Such mechanisms have been demonstrated in the CPU space [36], [6]. *To the best of our knowledge, this is the first study of error propagation in GPGPU applications*.

There are two main challenges with performing studies of error propagation on GPGPU applications. First, because GPGPU applications execute on both the CPU and the GPU (as current GPUs do not provide many of the capabilities needed by applications), it is important to track error propagation across the two entities. Further, GPUs are often invoked multiple times in an application (each such invocation is known as a kernel call), so one needs to track error propagation across these invocations. Second, unlike in the CPU space where there are freely available fault injection tools and frameworks to study error propagation [28], [52], [46], [10], there is a paucity of such tools in the GPU space.

SC16; Salt Lake City, Utah, USA; November 2016
978-1-4673-8815-3/16/$31.00 ©2016 IEEE

We address the first challenge by defining the kernel call as the unit of error propagation, and study error propagation both within kernel calls and across multiple calls. We address the second challenge by building a robust LLVM-based fault injection tool for GPGPU applications. LLVM is a widely-used optimizing compiler [34], and our fault injection tool is written as a module in the LLVM framework. As a result, we are able to leverage the program analysis capabilities provided by LLVM to track error propagation in programs, and correlate it with the program's code.

We make the following contributions in this paper.

- Develop LLFI-GPU, a GPGPU fault injection tool that can operate on the LLVM intermediate representation (IR) of a program and track error propagation in GPGPU programs.
- Define the metrics for tracking and measuring error propagation in GPGPU programs,
- Conduct a comprehensive fault-injection study on how errors propagate in twelve GPGPU applications (including both benchmarks and real-world applications), and how long and how fast such errors propagate and spread in the application,
- Discuss how the results may be leveraged by dependability techniques to provide targeted mitigation of error propagation for GPGPU applications at low cost.

Our main results from the fault-injection study are:

- Only a small fraction of the crash-causing faults that occur in GPUs propagate to the CPU, and only a minuscule fraction of crash-causing faults propagate to other GPU kernels. Thus, it is sufficient to consider checkpointing and recovery techniques at the GPU-CPU boundary.
- Errors do propagate to multiple memory locations, but this behavior is highly application specific. For example, a single fault can contaminate anywhere between 0.0006% locations to more than 60% of total memory locations, depending on the application.
- Unlike CPU programs, most of the memory corruptions in GPU programs lead to data corruptions in program output. Faults in memory that propagate to output data likely do so within the kernel where faults occurred. This allows error detection techniques to operate at the granularity of the GPU kernel call.
- More than 50% of the faults that occur in the GPU are masked within a single kernel execution. Thus, it may be counterproductive to deploy techniques such as Dual Modular Redundancy (DMR) or Error Detection by Duplicated Instructions (EDDI) [41] within the GPU kernel program as they will end up detecting many faults that are eventually masked.

## II. Fault Model and Background

In this section, we first present our fault model, and then define the terms we will use. We then provide a brief introduction of the GPGPU programming model, as well as the LLVM compiler that we use to perform our analysis.

### A. Fault Model

In this paper, we consider transient hardware faults that occur in the computational elements of the GPU processor, including pipeline stages, flip-flops, arithmetic and logic units (ALUs), and register file. We do not consider faults in the GPU memory or cache, as we assume that these are protected with ECC (this is the case for Nvidia GPUs, for example [26]). Likewise, we do not consider faults in the processor's control logic as we assume that it is unlikely to experience errors. This is because control-logic is often a small portion of the processor's total area, and can be protected with low overheads. Finally, we do not consider faults in the instructions' encoding as these can be detected through other means such as error correction codes. Our fault model is in line with other work in the area [27], [38], [21], [50].

### B. Terms

We use the following terms defined in prior work [53], [24]. We modified them slightly for our study on GPUs.

- **Fault Occurrence:** The event corresponding to the occurrence of the hardware fault in the GPU. The fault may or may not result in an error.
- **Fault Activation:** The event corresponding to the manifestation of the fault to the software, i.e., the fault becomes an error and corrupts some portion of the software state (e.g., register, memory location). The error may or may not result in a failure (i.e., SDC, crash or hang).
- **Crash:** The raising of a hardware trap or exception due to the error, because the program attempted to perform an action it should not have (e.g., read outside its memory segments). In GPGPU programs, this can be captured either using a debugger or checking the error code after kernel call.
- **Silent Data Corruption(SDC):** A mismatch between the output of a faulty program run, and an error-free execution of the program. We define SDCs at different levels of the GPGPU program's memory to facilitate error propagation analysis.
- **Benign Faults:** Program output matches that of the error-free execution even though a fault occurred during its execution. This means either the fault was masked, or overwritten by the program. Similar to SDCs, we define benign faults at different levels of the GPU's memory.
- **Error Propagation:** Error propagation means that the fault was activated, and has affected some other portion of the program's state, say 'X'. In this case, we say the fault has propagated to state X. We focus on the faults that affect the program state, and therefore consider error propagation at the application level.

### C. GPU Architecture and Programming Model

In this study, we focus on GPGPU applications that are implemented on the Nvidia Compute Unified Device Architecture (CUDA), a widely adopted programming model and toolset for GPUs. In the CUDA programming model, a GPGPU application consists of a control program running on the CPU, and a computation program called the kernel running on the GPU(s) without interfering with CPU. The kernel is implemented as a collection of functions in a C-like language, but with CUDA-specific annotations.

CUDA kernels adopt the single instruction multiple threads (SIMT) model to exploit the massive parallelism of GPU devices. From a software perspective, the CUDA programming model abstracts the SIMT model into a hierarchy of kernels, blocks and threads. The CUDA kernels consist of blocks, which consist of threads. This hierarchy allows various levels of parallelism such as fine-grained data parallelism, coarse-grained data parallelism and task parallelism. From a hardware perspective, blocks of threads run on streaming mutiprocessors (SMs) which have on-chip shared memory for threads inside the same block. Within a block, threads are launched in fixed groups of 32 threads, which are called warps. Threads in a warp execute the same sequence of instructions but with different data values.

GPU has its own memory space that is distinct from and not synchronized with the host CPU's memory. In the CUDA programming model, there are four kinds of memory: (1) global, (2) constant, (3) texture, and (4) shared. Global, constant, and texture memory accesses are generally slower from large device memory. Shared memory space, which can be software managed, is much smaller and built on chip, hence it is much faster to access. CUDA applications need to be aware of the memory hierarchy to access GPU memory efficiently.

### *D. LLVM Compiler*

In this paper, we use the LLVM compiler [34] for building the fault injector. LLVM is a compiler infrastructure that lets us statically analyze programs and optimize them. We choose LLVM as it provides support for a wide variety of language constructs including CUDA, and is equipped with a variety of program analysis and transformation techniques. Further, LLVM uses a typed intermediate representation (IR), in which source-level constructs can be easily represented. This enables identification of the code structures in a GPGPU program that are responsible for error propagation. Finally, LLVM allows us to abstract out details of the GPU hardware, which is essential as our goal is to understand program-level error propagation characteristics. Note however that our methodology is not tied to LLVM. Any platform that allows us to analyze GPU code and correlate it with the fault injection results will suffice.

## III. GPU FAULT INJECTOR

We build a fault injector for GPUs based on the open-source LLFI fault injector [52] , which has been extensively used for error propagation studies on the CPU [6], [35]. However, LLFI does not inherently support GPUs. Furthermore, performing error propagation analysis (EPA) for GPUs is much more intricate than on CPUs. We therefore extended LLFI to perform both fault injection and EPA on GPUs. We refer to the extended version of LLFI as LLFI-GPU[1] to distinguish it from the existing LLFI infrastructure.

[1]Available at: *https://github.com/DependableSystemsLab/LLFI-GPU*

Fault injection can be done at different levels of the system such as at the gate-level, circuit-level, architecture level and application level. Prior work [13] has found that there may be significant differences in the raw rates of faults exposed to the software layer when fault injections are performed in the hardware. However, we are interested in faults that are not masked by the hardware and make their way to the application. Therefore, we inject faults directly at the application level.

### *A. Design Overview*

LLFI is a compiler-based fault injection framework, and uses the LLVM compiler to instrument the program to inject faults. The CUDA Nvidia compiler NVCC is also based on LLVM, and compiles LLVM IR to a PTX representation, which then gets compiled to the SAS machine code by Nvidia's backend compiler. So at first glance, it seems trivial to integrate LLFI and NVCC to build a GPU-based fault injector. However, there are two challenges that arise in practice. First, NVCC does not expose the LLVM IR code and directly transforms it to the PTX code. LLFI relies on the IR code to perform instrumentation for fault injection, and hence cannot inject faults into the IR used by NVCC. Second, GPU programs are multi-threaded, often consisting of hundreds of threads, and hence we need to inject faults into a random thread at runtime. However, LLFI does not support injecting faults into multi-threaded programs.

We address the first challenge by attaching a dynamic library to NVCC which can intercept its call to the LLVM compilation module [1]. At that point, we invoke the instrumentation passes of LLFI to perform the instrumentation of the program. We then return the instrumented LLVM IR to NVCC, which proceeds with the rest of the compilation process to transform it to PTX code. We address the second challenge by adding a *threadID* field to the profiling data collected by LLFI to identify each thread uniquely. We then choose a thread at random to inject into at runtime from the set of all threads in the program. We also add information on the kernel call executed and the total number of kernel calls to the profiling data. These are used to choose kernel calls to inject faults into.

LLFI-GPU works as follows. First, LLFI-GPU profiles the program and obtains the total number of kernel calls, the number of threads per kernel call, and the total number of instructions executed by each kernel thread. It then creates an instrumented version of the program with the fault injection functions inserted into the CUDA portion of the program's code (this is similar to what LLFI does, except that we restrict the instrumentation to the CUDA portion of the program). LLFI-GPU then chooses a random thread in a random kernel call, and a random dynamic instruction executed by it, based on the profiling data gathered (the instruction is chosen uniformly from the set of all instructions executed). For the chosen instruction, LLFI-GPU overwrites the result value of the instruction with a faulty version of the result (e.g., by flipping a single bit in it), and continues the application.

Thus, LLFI-GPU directly executes the program on the GPU hardware after instrumenting it, unlike prior approaches such as GPU-Qin [20] which use debuggers for fault injection. Debugger-based fault injection has the advantage that it offers more control over the program, but is often significantly slower. As a point of comparison, we ran both GPU-Qin[2] and LLFI-GPU on a simple matrix multiplication benchmark, MAT from NVIDIA SDK Sample [4]. Similar to LLFI-GPU, GPU-Qin operates in two phases: profiling and fault injection. We measured the average time taken by GPU-Qin for these two phases to be 2 hours (=7200 seconds), and 82 seconds per run respectively. In contrast, LLFI-GPU takes only 6 seconds for profiling this benchmark and 2 seconds per run for the fault injection phase for the same set of inputs. The significant speedup of over 1000x obtained by LLFI-GPU is because GPUs execute significantly slower in debug mode [2], and GPU-Qin single steps through every instruction in the program in the profiling phase. We have confirmed that the above execution times are fairly typical depending on the number of dynamic instructions of the programs executed using GPU-Qin[3], and hence we did not run any of the other benchmarks with GPU-Qin. Therefore, LLFI-GPU is significantly faster and more scalable than prior techniques, making it feasible for studying realistic HPC workloads.

## B. Error Propagation Analysis (EPA)

After injecting a fault, LLFI-GPU tracks memory data at every kernel boundary for the analysis of error propagation. This is because we are interested in error propagation at the GPU kernel boundary, rather than within a kernel. This is different from what LLFI does as it tracks the error propagation using memory and registers after each LLVM instruction. Although we can leverage the existing EPA mechanism in LLFI for tracking error propagation at the kernel boundaries, we found that this incurs very high overheads, and often results in the kernel running substantially slower. Therefore, we decided to build our own error tracking mechanism in LLFI-GPU that is optimized for our use case.

Figure 1 illustrates how the EPA mechanism works for a simple GPU kernel. The code fragment is from *bfs*. It allocates memory on device through *cudaMalloc()* before launching kernels, and it deallocates the memory on device through *cudaFree()* at the end of the program. After each kernel invocation, LLFI-GPU saves all memory data allocated on the GPU to disk. This step corresponds to line 6-13. Later, we compare the saved data after each kernel call with that from a golden run and mark any differences as a result of the error propagation. Because we perform this comparison at the kernel boundaries, we do not need to worry about non-determinism introduced by thread interleaving within the GPU.

## C. Limitations

Our fault injections are performed at the LLVM IR level rather than at the SASS or PTX code levels. One potential drawback of this approach is that downstream compiler optimizations may change both the number and order of instructions, or even remove the fault injection code we inserted. To mitigate this effect, we made sure that our fault injection pass is applied after various optimization passes in the LLVM IR code. Further, LLFI-GPU gathers all executed instructions in the profiling phase as described above, and we made sure that all

```
 1 cudaMalloc(gpu_graph_visited, int*512);
 2 cudaMalloc(gpu_graph_edge, int*512);
 3 ...
 4 kernel1()
 5 // Track device data
 6 foreach(gpu_graph_visited){
 7   // Save to disk
 8   dump(each_visited);
 9 }
10 foreach(gpu_graph_edge){
11   // Save to disk
12   dump(each_edge);
13 }
14 ...
15 kernel2()
16 // Track device data agin, same as above.
17 ...
18 cudaFree(gpu_graph_visited);
19 cudaFree(gpu_graph_edge);
```

Fig. 1: Example of the error propagation code inserted by LLFI-GPU

the target instructions as fault injection candidates are gathered and injects faults only into these instructions - this ensures that faults are not injected into dead instructions that are optimized out by the backend compiler. Due to backend optimizations after the IR is generated, the mapping of instructions may be changed at the machine assembly levels (e.g., SASS level). This may result in different absolute values of the SDC rate for fault injections performed at different levels. However, as we said before, we are interested in obtaining insights into error propagation intrinsic to applications instead of deriving derated SDC rates. Finally, a previous study on CPU applications showed that there is negligible difference in SDC rates between fault injections performed at the LLVM IR level and the assembly code level [52].

## IV. Metrics for Error Propagation

In this section, we define the metrics for measuring error propagation in our experiments. We measure error propagations along two axes, namely (1) execution time, which captures the temporal nature of the propagation, and (2) memory states, which captures the spatial nature of the propagation. We examine these in further detail below.

## A. Execution time

A fault can propagate in the program corrupting data values until it either causes program termination (e.g., by a crash), or it is masked. The former happens if the fault crashes the program, or the program finishes execution successfully (program hangs are handled through a watchdog timer). The latter happens if the faulty data is overwritten, or if the values to which the fault propagates are discarded by the program. The execution time metric measures the time between the fault's occurrence and the masking or termination events.

We use kernel invocations to measure the propagation time of an error. For example, if an error occurs in a certain kernel invocation K1, and the program crashes after two more invocations of the kernel, say K2 and K3, we label the execution time of this fault to be 2. There are three reasons for using kernel invocations as the unit of propagation. First, we are often interested in knowing if an error propagates across

[2]The only other GPU fault injector that we know of, SASSIFI [26], was not publicly available at the time the paper was written, and hence could not be used for comparison.

[3]Based on personal communication with the developers of GPU-Qin.

the CPU-GPU boundary or across multiple kernel invocations on the GPU. The number of kernel invocation captures this value. The second reason is that unlike other metrics such as wall-clock time (e.g., seconds), or the number of executed instructions which are platform dependent, the number of kernel invocations depends only on the GPU application. Thus, it captures the application-level semantics of error propagation without being affected by platform-specific details. This is important as our goal is to design application level error-resilience mechanisms. Finally, unlike CPU applications which have a few long-living threads, GPU applications typically have a large number of short-lived threads executing on the GPU as they focus on throughput. When these threads terminate, the control is passed back to the CPU, thereby resulting in frequent CPU-GPU boundary crossings. We found that the average kernel invocation time in our benchmarks is usually less than one minute - this is in line with prior work [49].

```
 1 // Allocate for all being used in device
 2 cudaMalloc(); ...
 3 // Total Memory
 4 cudaMemcpy(gpu_graph_nodes, cpu_graph_nodes);
 5 // Total Memory
 6 cudaMemcpy(gpu_graph_visited, cpu_graph_visited);
 7 // Total Memory
 8 cudaMemcpy(gpu_graph_edges, cpu_graph_edges);
 9 // Result Memory
10 cudaMemcpy(gpu_result_cost, cpu_result_cost);
11 ...
12 // kernel invocations
13 kernel <<<...>>>, kernel2 <<<...>>>, ... ...
14 ...
15 // Result Memory
16 cudaMemcpy(cpu_graph_nodes, gpu_graph_nodes);
17 // Output Memory
18 foreach(cpu_result_cost):
19 {if(cost<0} dumpCostForResult();}
20 // Deallocate memory used in device
21 cudaFree() ...
```

Fig. 2: Code Example of a Kernel Cycle from Benchmark *bfs*

In addition to kernel invocations, GPU applications also need to transfer data from CPU memory to GPU memory and back. Typical GPU applications perform multiple kernel invocations in between transfers to amortize the latency of transfers. We define a *kernel cycle* as a sequence of kernel invocations by the application that is prefixed by memory transfer from the CPU to the GPU, and suffixed by memory transfer from the GPU back to the CPU. All applications in our study except LULESH and NMF had only a single kernel cycle. However, all of them have multiple kernel invocations within a single kernel cycle.

Figure 2 shows an example of a kernel cycle in the *bfs* application. The first phase of the kernel cycle (lines 1-10) consists of memory allocations, and data movement from the host (CPU) to the device (GPU) memory. The second phase consists of kernel invocations (line 13), which perform computations on the data copied to the GPU memory. Finally, in the third phase, after the kernels finish their work, the CPU collects data from the GPU and processes it (lines 15-20).

## B. Memory States

To better understand error propagation, we examine which parts of a GPU program's memory have been affected by an error after fault injection. We divide memory into three categories, namely Total Memory(TM), Result Memory(RM) and Output Memory(OM). TM is a superset of RM, which in turn is a superset of OM. This is shown in Figure 3. TM refers to the entire memory space allocated for the program on device. The allocations are usually done through *cudaMalloc* calls, and through global variables declared by kernels. RM refers to the memory locations containing the computation results that the CPU transfers from the GPU at the end of a kernel cycle. OM refers to the memory locations containing the data that the CPU actually processes for computing the program output.

In the example in Figure 2, the transfer occurs at line 16. So *gpu_result_cost* is the pointer to the RM in this example. Further, the processing phase occurs in lines 18-19. So the OM consists of the results of the *dumpCostForResult* function. Note that applications may choose only certain parts of the RM to copy into their output. For example, a floating point application may use only the two most significant digits from the result in RM to compute the output, in which case the OM consists of only these two digits.

We use SDC to refer to corruption of the above three categories of memory, as all of these pertain to data corruptions. We use the memory type as a subscript to denote corruptions of different memory categories, e.g., $SDC_{RM}$. Note that $SDC_{OM}$ is what is typically defined as an SDC in prior work [20], [54], [26] on GPUs, as they only study the effect of faults on the final output of the application. However, our aim is to study error propagation and hence we study data corruption in the memory states of the application. We also refer to corruption of the memory that is in TM but not RM as (TM-RM), and that in RM but not OM as (RM-OM).

For example, in Figure 3, assume that an error occurs during a kernel invocation (K1) and affects the memory location (L1) in the TM. However, it does not propagate to the RM. In the next kernel invocation (K2), the faulty value in L1 is read and affects a value at another location L2 in RM. Hence we say the error propagates from the TM to RM during K2. In this case, the error causes an $SDC_{TM}$ after kernel K1, and an $SDC_{RM}$ after kernel K2. However, the error does not propagate to the OM, and hence does not result in an $SDC_{OM}$.

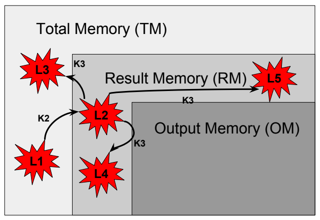


Fig. 3: Memory State Layout for CUDA Programming Model (K2 and K3 are kernel invocations)

We also measure what fraction of the memory is contaminated by error propagation. We define this as the *spread* of an error. For example, in Figure 3, at K3, the faulty value in location L2 is assigned to different memory locations (L3, L4 and L5), and hence propagates to these locations. In this case, the fault value has propagated to a total of 5 locations at the end of K3. Assuming TM consists of 100 memory locations, the error spread after K3 in this case is 5/100=5%.

## V. Experimental Setup

We describe the benchmarks used, and the fault injection procedure. We also provide details of the hardware and software platform used for the measurements, followed by the research questions.

### A. *Benchmarks Used*

We choose twelve GPGPU applications in total for our experiments. These are drawn from standard GPU benchmark suites such as Rodinia [11] and Parboil [47], as well as real world applications. We choose five programs from Rodinia, and two programs from Parboil. The applications from Rodinia were chosen based on two criteria: (1) compatibility with our toolset (i.e., we could compile them with NVCC), and (2) suitability for our experiments. For the latter criteria, we discard small applications that had too few kernel invocations (as it is uninteresting to measure error propagation in such applications), and applications in which the outputs were non-deterministic (as it is difficult to classify the results of an injection or error propagation in such applications). For Parboil, we randomly choose two applications from the suite to balance time with representativeness.

In addition to the standard benchmarks, we pick five real-world HPC GPGPU applications, Lulesh [33], Barnes-Hut [45], Fiber [55], Circuit [5] and NMF [8]. These applications perform n-body simulation, hydrodynamics modeling, fiber scattering simulation, circuit solving and audio source processing respectively.

Table I shows the details of the applications used in our study and the input used. The number of kernel invocations ranges from 4 to 8567 in these applications. The lines of C code of these applications ranges from 222 to 5684. We configured our benchmarks to run on a single GPU as our goal is to study error propagation between kernels. Multi-GPU programs also transfer data and synchronize at kernel boundaries [3], and we hypothesize that our results generalize to such programs - validating this hypothesis is a subject of future work.

Similar to prior work in the area [20], [26], [6], [53], [27], we run each benchmark application with a single input. However, questions remain on whether multiple inputs may affect error propagation behaviors. We hypothesize that different inputs have limited effect on error propagation as the propagation is primarily dominated by the application's algorithm, rather than problem size. This is because different inputs likely only scale the execution times of certain code sections, rather than change the underlying program structure. We will further validate this hypothesis in future work.

### B. *Fault Injection Method*

As mentioned before, we use LLFI-GPU to perform the fault injection experiments. We consider only one fault per run as hardware transient faults are rare events relative to the program execution times. For each application, we inject

TABLE I: Characteristics of GPGPU Programs in our Study

| **Benchmark** | **Benchmark Suite/Author** | **Description** | **Kernel Invocations** | **LOC** | **Input** |
|---|---|---|---|---|---|
| BFS | Rodinia (v2.1) | An algorithm for traversing or searching tree or graph data structures | 15 | 342 | 4096 |
| LUD | Rodinia (v2.1) | An algorithm to calculate the solutions of a set of linear equations | 9 | 564 | 64 |
| PathFinder | Rodinia (v2.1) | Use dynamic programming to find a path on a 2-D grid | 4 | 236 | 100000 100 20 |
| Gaussian | Rodinia (v2.1) | Compute result row by row, solving for all of the variables in a linear system | 29 | 394 | 16 |
| HotSpot | Rodinia (v2.1) | Estimate processor temperature based on an architectural floorplan and simulated power measurements | 15 | 328 | 512 2 32 |
| cutcp | PARBOIL (v0.2) | Computes the short-range component of Coulombic potential at each grid point | 10 | 1540 | watbox. sl40.pqr |
| stencil | PARBOIL (v0.2) | An iterative Jacobi stencil operation on a regular 3-D grid | 99 | 1584 | 128 128 32 100 |
| Barnes-Hut | Texas State Univ. San Marcos (v2.1) | An approximation algorithm for performing an n-body simulation developed by Texas State Univ. San Marcos | 20 | 965 | 4 4 |
| Lulesh | Lawrence Livermore National Laboratory (v1.0) | Science and engineering problems that use modelling hydrodynamics | 8567 | 5684 | edgeNodes =2 |
| Fiber | Northeastern University (v1.5) | High Performance Computing of Fiber Scattering Simulation application | 2881 | 1437 | 480 4 20 |
| Circuit | Rice University | Parallel circuit solver for solving 2D circuit grid using Jacobi method | 450 | 222 | 0.00001 1 |
| NMF | UC Berkley | Audio analysis and source separation. | 409 | 2398 | default |

10,000 faults in total - this yields error bars ranging from 0.22% to 1.11% depending on the application for the 98% confidence intervals. Further, we use the single bit-flip model for injecting faults as it is the de-facto fault model used in studies of transient faults. Although recent work [13] has found that hardware faults may manifest as both single and multiple bit flips at the software level, other studies have shown that there is very little difference in failure rates due to single and multiple bit flips [37], [7], [12]. Therefore, we stick to the single bit flip model in this study.

To obtain a golden run for error propagation analysis, we first run the program without any fault injections. We then gather the output of the program and memory data stored after each kernel invocation as described in Section III-B. We measure error propagation and error spreading by comparing data from fault injection runs with the golden run. This comparison is done on a bit-wise basis, except for floating point numbers, which are compared using 40 digits of precision. As we omit benchmarks that have random values in program output, the golden runs of the chosen benchmarks are deterministic. We manually verified that this was the case for our benchmarks.

There are three kinds of failures that can occur due to an injected fault: Crashes, SDCs and Hangs. Crashes are found by using the CUDA API call *cudaGetLastError()* after every kernel invocation. SDCs are found by comparing the program's output with the golden run for each memory type ($TM$, $RM$, and $OM$). Hangs are found by setting a watchdog timer for 5000 seconds when the program starts - this is much larger than the time taken by each application run.

## C. Hardware and Software Platform

Fault injection experiments are performed on host PCs with an Intel Xeon CPU and 32GB DDR3 memory. We use two GPU platforms both from Nvidia, namely Tesla K20 the GTX960, for running our experiments. The results were similar on both platforms - this is not surprising as our experiments were at the application level. We therefore report only the results on the K20 platform in this paper. We will further detail our comparison in RQ7. The operating system running on the host is Ubuntu Linux 12.04 64bit, and the CUDA driver and Toolkit used is V6.0.1.

## D. Research questions (RQs)

We answer the following questions in our study.

RQ1: What is the percentage of SDCs in different memory states?

RQ2: How long do errors take to propagate to the RM?

RQ3: Do errors spread into different memory states and why?

RQ4: How many faults are masked within the GPU kernel and not allowed to propagate?

RQ5: Do crash-causing faults propagate to the host CPU before they cause crashes?

RQ6: Do crash-causing faults propagate across kernels before they cause crashes?

RQ7: Are resilience characteristics of applications different on different GPU platforms?

## VI. Results

The results are organized by the research questions asked in Section V. We first present the aggregate results of the fault injections across all the benchmarks.

## A. Aggregate Fault Injections

Figure 4 shows the aggregate results for our fault injection experiments. SDCs and Benign are measured by comparing the programs' final outputs with the golden run. This corresponds to data recored in OM after the programs finish their executions. So SDCs here correspond to $SDC_{OM}$s, and benign to $Benign_{OM}$. On average, crashes constitute 17.52%, SDCs constitute 18.98% and Benign faults constitute 63.35% of the injections. In our experiments, hangs are negligible, and are hence not reported.

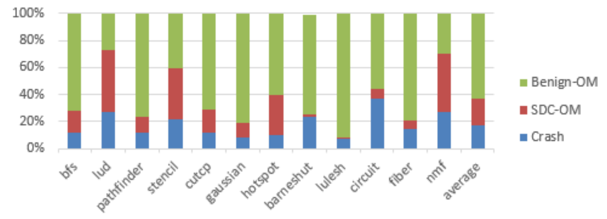


Fig. 4: Aggregate Fault Injection Results across the 12 Programs

## B. Error Propagation

**RQ1: What's the percentage of SDCs in different memory states?**

We analyze the memory data in each memory type after the last kernel invocation in kernel cycle, and compare with the golden run. Table II shows SDCs measured at different memory locations at the end of the kernel cycle. On average, $SDC_{OM}$, $SDC_{RM}$ and $SDC_{TM}$ are 19.67%, 25.70% and 25.79%. As can be seen, the values of $SDC_{RM}$ and $SDC_{TM}$ are very similar across applications. In other words, most faults in the TM propagate to the RM. This is surprising as it suggests that there is little to no masking of errors in the TM. On the other hand, CPU applications are known to exhibit significant error masking in memory [6]. One possible reason is that unlike CPUs, GPUs perform highly specialized computations, and hence all the results produced are important to the application and hence they propagate to the output.

However, there is a difference of about 6% on average between $SDC_{OM}$ and $SDC_{RM}$ (see in $SDC_{(RM-OM)}$). This is due to the application masking the error either through type-casting or selective truncation of floating point data. An example of this is *cutcp*, which exhibits a difference of nearly 30% between the $SDC_{OM}$ and the $SDC_{RM}$. Overall, about 76% of the faults propagate from the RM to the OM, which is again much higher than observed in CPU applications [6].

We note that the *lulesh* application comes equipped with application-level algorithm correctness checks (i.e., residual check). For example, the documentation for *lulesh* states that the correct output consists of the correct number of iterations, and six most-significant digits in the *final origin energy* variable [32]. Therefore, $SDC_{OM}$ here is measured based on these correctness checks. None of the other applications however come with such checks, and hence we consider the entire output in these cases for comparison with the fault-injected run.

**RQ2: How long do errors take to propagate to the RM?**

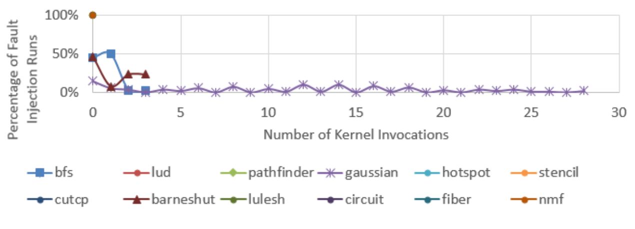
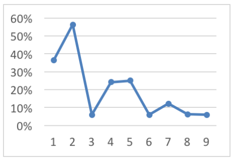


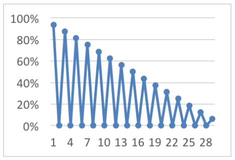
Fig. 5: Detection Latency of faults that result in $SDC_{RM}$

TABLE II: SDCs that occur in the different memory types

| | bfs | lud | pathfinder | stencil | cutcp | gaussian | hotspot | barneshut | lulesh | circuit | fiber | nmf | average |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $SDC_{OM}$ | 17.01% | 45.58% | 20.60% | 37.00% | 16.50% | 10.90% | 29.40% | 1.30% | 0.97% | 7.50% | 7.01% | 43.20% | 19.67% |
| $SDC_{(RM-OM)}$ | 0.00% | 0.00% | 1.30% | 0.00% | 28.90% | 0.00% | 6.00% | 2.30% | 0.10% | 14.80% | 12.69% | 10.10% | 6.35% |
| $SDC_{RM}$ | 17.01% | 45.58% | 21.90% | 37.00% | 45.40% | 10.90% | 31.50% | 3.60% | 1.07% | 22.30% | 19.70% | 53.30% | 25.70% |
| $SDC_{(TM-RM)}$ | 0.00% | 0.00% | 0.00% | 0.00% | 0.00% | 0.80% | 0.00% | 0.20% | 0.00% | 0.10% | 0.00% | 0.00% | 0.09% |
| $SDC_{TM}$ | 17.01% | 45.58% | 21.90% | 37.00% | 45.40% | 11.70% | 31.50% | 3.80% | 1.07% | 22.40% | 19.70% | 53.30% | 25.79% |



(a) lud



(b) gaussian

Fig. 6: Percentage of RM Updated by Each Kernel Invocation. Y-axis is the percentage of RM locations that are updated during each kernel invocation. X-axis represents timeline in terms of kernel invocations.

Once a fault is injected in a kernel, we measure the number of kernel calls after which it propagates to RM (if it does). Figure 5 shows the results. As shown in the figure, most faults that affect the RM do so within a single kernel invocation after injection. This means that errors propagate relatively soon to the RM after their occurrence. The exceptions are *bfs*, *gaussian* and *barneshut*.

Figure 6 shows the percentage of RM updated after each kernel invocation. For the sake of space, we only show the results for two applications, namely *lud* and *gaussian*. As we can see, *lud* updates the RM in every kernel invocation, whereas *gaussian* does not. This explains why the propagation occurred within one kernel execution for *lud*, but not *gaussian*.

## C. Error Spreading

We defined error spreading as the percentage of memory locations contaminated due to an error (Section IV). Unlike the previous section where we considered any difference from the golden run as an SDC for that memory type, here we only consider the amount of memory that is different.

**RQ3: Do errors spread into different memory states and why?**

From our fault injection experiments, we examine how errors spread as a function of time (i.e., kernel invocations). Because we performed 10,000 injections per application, we cannot show all the data. Therefore, we only show representative injections for each application. Figure 7 shows the percentage of memory locations in TM and RM that are contaminated by the injected faults at the first dynamic kernel invocation. The spread is calculated as (*Contaminated TM or RM Locations* / *Total TM Locations*) * 100.

Our main findings are: (1) error-spreading is very application-specific. For example, a single fault can contaminate nearly 60% of TM memory locations in *lud*, whereas the number is as low as 0.0006% in *cutcp*. (2) only a very small amount of memory locations are affected in the same kernel where the fault was injected. Rather, most faults propagate into memory locations in later kernel invocations. (3) trends of error spreading between RM and TM of the same application are rather similar, though the absolute values may be different. In other words, applications that have extensive error spread in the TM have extensive error spread in the RM as well.

Finally, in almost all applications the error spreading either increases or remains constant as the number of kernel calls increase. The exception to this is the *stencil* application in which the error spreading decreases significantly as the number of kernel calls increase. This is because the algorithm of *stencil* takes neighbours' values and keeps averaging them. The errors may be finally masked as the averaging process progresses. We also observed that there is a small decrease in error spreading in the *bfs* application after it reaches its peak in TM. This is because some of locations in TM are reassigned during program execution, and faulty data in these locations can be overwritten with correct data, thereby masking the errors.

Because *lud* exhibits the highest error spread, we study its code structure to understand the reasons. Figure 8 shows the code structure leading to extensive error spread in *lud*. The code exhibits a cyclic data flow from global memory to shared memory, and then back to global memory. Note that shared memory is used to transfer data between threads only in the same block. In the example, *dia* is a pointer to shared memory, and *m* points to global memory. At line 5, a portion of shared memory in *dia* is initialized by global memory *m*. This portion of data in *dia* is shared by other threads in its block. After the shared data is consumed, it writes data back to global memory *m* at line 9 again. If a fault occurs in *m* at line 5, *dia* will be first compromised and the data processed by other threads in the same block may be affected after reading the corrupted data from *dia*. And then at the end of the kernel invocation, a different part of *m* may be corrupted by reading data from *dia*(line 9). In the next invocation of this kernel, faulty values in *m* may be used in the initialization of *dia* again at line 5. But this time, it may be initialized to a different portion of *dia* that are used by threads in a different block, because there are different parts of *m* that were corrupted in the previous kernel invocation. This leads to extensive error spread for this application.

## D. Fault Masking

In the previous two sections, we examined how faults propagate across different memory locations and kernel executions. We now ask the complementary question: how many faults are masked within a single kernel invocation of their occurrence.

**RQ4: How many faults are masked within the GPU kernel and not allowed to propagate?**

Table III shows the percentage of benign faults measured at the first kernel invocation after the fault injection ($Benign_{TM}$). We measure this value by comparing all memory locations in TM with the golden run right after the first kernel invocation where the fault is injected. If there is no

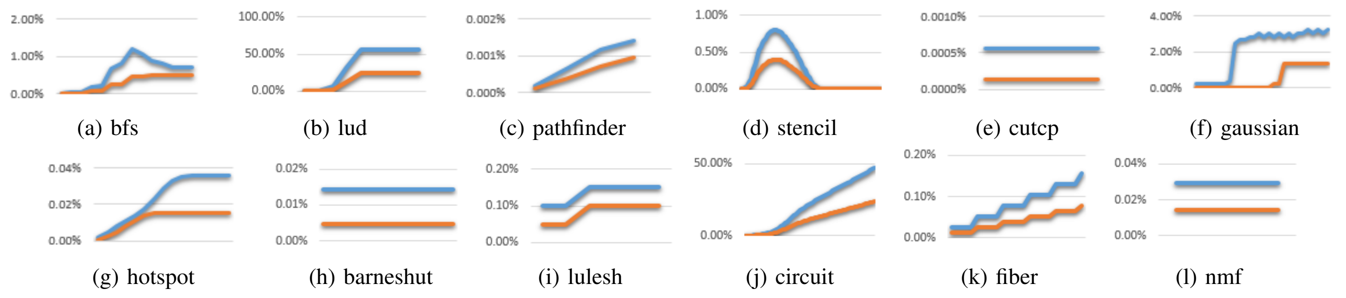


Fig. 7: Percentage of TM and RM Contaminated at Each Kernel Invocation. Y-axis is the percentage of contaminated memory locations, X-axis is timeline in terms of kernel invocations. Blue lines indicate TM, and red lines represent RM.

TABLE III: Percentage of Benign Faults Measured at the First Kernel Invocation after Fault Injection

| bfs | lud | pathfinder | stencil | cutcp | gaussian | hotspot | barneshut | lulesh | circuit | fiber | nmf | average |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 63.5% | 28.5% | 69.9% | 25.0% | 42.7% | 81.1% | 57.2% | 76.4% | 92.5% | 25.6% | 62.9% | 20.0% | 53.4% |

```
...
kernel lud_perimeter(*m){sv
  __shared__ dia[BLOCK_SIZE][BLOCK_SIZE];
  ...
  dia[i][j] = m[array_offset+idx];
  ...
  for(...){
    ...
    m[array_offset+(blockId.x+1)+...] = dia[i][idx];
    ...
  }
  ...
}
...
```

Fig. 8: Code Structure Leading to Extensive Error Spread in *lud*

difference between the two TMs, we count it as a benign fault. We consider TM here as it is a superset of both RM and OM. Therefore if a fault is masked in the TM, it is also masked in the other two types of memory (even in future kernel executions).

As we can see from the table, more than 50% of the faults injected are masked within the same kernel they are injected in, and do not propagate. The maximum masking is achieved in the case of *lulesh* in which nearly 92.5% of the faults are masked. The minimum masking is achieved for *nmf* in which only 20% of the faults are masked. Such variations across applications are because programs contain different amount of code structures leading to masking effects.

We found there are two prevalent patterns leading to error masking in our benchmark programs, namely (1) Comparison, and (2) Truncation. An example of Truncation is shown in Figure 9, on the left. R0 and R1 are initialized at lines 2 and 3, and R2 holds the result of comparing R0 and R1 at line 3. Consider a fault that flips the first bit of R1 - R1 erroneously becomes 1110 from 1111. However, the result of R2 will not be affected since R1 is still greater than R0. An example of Truncation is shown in the right part of Figure 9. At line 2, R0 is initialized. At line 3, value of R1 is truncated from 0001 to 01. Consider a fault that occurs at line 2 and flips either of the left-most 2 bits of R0 - it will not affect the value of R1 at line 3 due to the truncation. Hence the fault will be masked.

```
1 ...
2 R0 = 0000
3 R1 = 1111
4 R2 = cmp R0, R1
5 ...
```

```
1 ...
2 R0 = 0001
3 R1 = cast R0, 2
4 // R1 -> 01
5 ...
```

Fig. 9: Examples of Fault Masking. (a) Comparison, (b) Truncation

## E. *Crash-causing Faults*

The next two research questions have to do with crash-causing faults and their propagation to the CPU (host) or other GPU kernels. We focus on crash-causing faults as prior work has found that long-latency crashes can lead to checkpoint corruptions, and cause unrecoverable failures [36], [53].

**RQ5: Do faults propagate to the host CPU and cause crashes?**

From our experiments, we can observe that there is a very small chance (0.02% on average) for a fault that occurs in a kernel to contaminate the CPU states and lead to a crash eventually. This is because memory address spaces are separate in the CPU and GPU, and hence faulty pointers produced by the GPU are unlikely to be used in the CPU to access memory. Because faulty pointers are responsible for the majority of crash causing errors on the CPU [36], these errors do not lead to crashes.

**RQ6: Do crash-causing faults propagate across kernels before they cause crashes?**

In our experiments, we find that there is no fault that propagates across kernels and causes a crash. In other words, cash-causing faults typically cause crashes within the kernel in which they occur. This is because pointers or memory address offset variables are usually passed between kernels in the constant memory space, which is read-only. Note however that it is possible for faults to propagate across kernels if address offsets of pointers are passed through global variables (we have empirically verified this observation through carefully constructed code samples - we do not present these due to space constraints). However, typical GPGPU programs do not exhibit this behavior, as each thread is responsible for its own memory locations, and hence multiple threads do not read the same offset to calculate the same memory address.

### F. Platform Differences

**RQ7: Are resilience characteristics of applications different on different GPU platforms?**

We use the two platforms, K20 and GTX960 for this experiment. To answer this question, we performed 1,000 fault injections for each benchmark application on the two platforms. We focus on SDCs for this experiment as these are often the most important concern in practice. Note that we have omitted two programs, *fiber* and *nmf*, as we encountered errors when compiling them on GTX960, probably because they use features that are not supported on that platform. To compare the distributions of the SDC values, we ran a t-test between the values obtained on the two platforms. We found that the p-value was 0.991. Thus, our results show that we fail to reject the null hypothesis, indicating that the values are statistically indistinguishable from each other. Therefore, we can conclude that the resilience characteristics of the applications do not vary significantly between the two platforms.

## VII. Implications

In this section, we consider the implications of the results on error detection and recovery techniques. These are organized by the RQs.

**RQ1: Percentages of SDCs in different memory states** In RQ1, we found that most SDCs in the TM propagate to the RM, and more than 76% of the SDCs in the RM propagate to the OM. Table IV shows the size of data in OM, RM and TM in each application. As we can see, the RM is only a small fraction of the TM. This suggests that checking the RM for consistency (e.g., using detectors [42]) may be much more efficient than checking the entire TM. Another possibility is checking the OM which is even smaller than the RM. However, the OM may not be updated until the end of the program and hence checking the OM may incur high detection latency.

**RQ2: Detection Latency of Errors in RM** Error detection latency is critical when designing checkpoint intervals. If the detection latency is too long, errors may propagate to checkpoints before they are detected, thereby corrupting checkpoints [36]. Our results show that errors propagate to the RM relatively soon after their occurrence (i.e., within one kernel call in most cases). Therefore, placing detectors on RM will ensure low-latency error detection.

**RQ3: Error spreading to different memory states** Error spreading is highly application specific in GPGPU applications. Further, only certain code structures in GPGPU programs may lead to extensive error spread. Therefore, one can statically analyze the program to identify such structures to protect. Further, the code can be restructured to avoid error spreading in some circumstances. In some applications (e.g., *Stencil*), there may be natural mechanisms in the code to dilute the effect of error spreading over time.

**RQ4: Effect of error masking** We found that there is substantial error masking within GPU kernels, and that many errors do not even affect the TM after they occur. This means that there may not be a need to deploy expensive error detection mechanisms such as Dual Modular Redundancy (DMR) or Error Detection by Duplicated Instructions (EDDI) [41] within the kernel, unless it is a safety-critical application. Instead one can check the results after the kernel's invocation. For example, ABFT-based detection algorithms [29], [17] can be used at the kernel boundary to detect errors, to determine whether GPU kernel re-execution should be initiated.

**RQ5 & RQ6: Crash-causing Faults and Checkpoint Scheme** Studies on error propagation on CPUs find that crash-causing faults can propagate for a long time before they cause crashes. Hence, checkpoints may be corrupted by these faults if the crash-latency in the program is not bounded [35], [36]. However, on GPGPU programs, we find that crash-causing faults do not propagate outside the kernel where faults occur. In other words, crash-latency of GPGPU programs is naturally bounded within one kernel invocation. Therefore, one can place checkpoints at kernel boundaries for crash recovery. As we find that many kernels do not propagate errors to other kernels, individual kernels could also recover from failures through re-execution at the kernel boundaries of copying. The application can be restarted locally on the same GPU or on a spare GPU. Further, as most kernels have short execution times in the range of milliseconds, the cost of re-executing a kernel would be insignificant.

**RQ7: Differences across platforms** From our findings, it appears that error resilience of GPGPU applications does not depend on the specific hardware platform (we have only validated it on platforms from the same manufacturer, which was Nvidia in our case). This suggests that one can perform resilience characterization on one platform and generalize the results to a different platform.

## VIII. Related Work

**Fault Injectors:** Yim et al. [54] built one of the first fault injectors for GPGPU applications. However, their injector operates at the source code level, and only considers faults that are visible at the source level. Fang et. al. [20] designed a GPGPU fault injector, GPU-Qin, that operates on the CUDA assembly code (SASS). They use the CUDA debugger (CUDA-gdb) to inject faults, which takes significantly longer than performing fault injections by code instrumentation (Section III). In follow up work, Hari et. al. [26] built SASSIFI, a GPGPU fault injector that transforms the SASS code of the program to inject faults similar to LLFI-GPU. Both GPU-Qin and SASSIFI operate on the SASS assembly code level, which makes it difficult to map back the faults to the source code. In contrast, LLFI-GPU operates at the LLVM IR level, which is much closer to the program's source code. This makes it possible to perform program analysis on the program and map the fault injection results back to the source code, thereby helping programmers make their code error resilient.

**Error Propagation in CPU applications:** Ashraf et. al. [6] analyzed how faults propagate in different memory locations in HPC applications. Li. et. al. [36], [35] characterized the source code patterns on how crash-causing faults propagate in applications, while Yim et al. [53] characterized error propagation in different types of memory. Natella et. al. [40] characterized error propagation of software faults from

TABLE IV: Size of OM, RM and TM

| | bfs | lud | pathfinder | stencil | cutcp | gaussian | hotspot | barneshut | lulesh | circuit | fiber | nmf | geo mean |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| OM | 14.29% | 7.50% | 0.99% | 7.50% | 7.50% | 1.67% | 5.00% | 18.75% | 12.50% | 15.00% | 49.98% | 0.03% | 5.02% |
| RM | 14.29% | 50.00% | 1.98% | 50.00% | 50.00% | 3.21% | 66.67% | 37.50% | 18.75% | 50.00% | 49.98% | 0.03% | 13.56% |
| TM | 100% | 100% | 100% | 100% | 100% | 100% | 100% | 100% | 100% | 100% | 100% | 100% | 100% |

one software component to another. Finally, Zilles et. al. [15] study patterns of fault masking at the instruction level, which is the complement of error propagation. All of the above studies focus on CPU programs. GPGPU programs have a very different programming model and semantics compared to CPU programs, and hence it is not clear how to extend these findings for GPGPU programs.

**GPU Fault Tolerance:** There have been many papers on building fault tolerance techniques on GPGPU platforms. Jeon et. al. [30] duplicated kernel threads that have the same input to detect errors. Dimitrov et. al. [19] leverage both instruction-level and thread-level parallelism to duplicate application code. Tan et. al. [48] proposed an analytical method to evaluate the error-resilience of GPU platforms. Peña et. at. [43] explored low-cost data protection and recovery mechanisms for GPGPU platforms based on API interception. Finally, Yim et al. [54] proposed a technique to detect errors by duplicating code at within the loop bodies of GPGPU programs. While these are useful, none of the above papers study error propagation in GPGPU programs, which is our focus.

## IX. Conclusion

In this paper, we study error propagation in GPGPU application with the goal of building targeted error detection and recovery mechanisms for them. We built a fault injection tool LLFI-GPU, and defined metrics for quantifying propagation in GPGPU applications. We empirically studied error propagation across ten GPGPU applications using LLFI-GPU. The main findings are: (1) Crash-causing faults in GPGPU are naturally kernel-bounded, (2) Error spreading in memory is highly application dependent, (3) Most memory data corruptions lead to output corruption unlike what is observed in CPU programs, and (4) The majority of faults are masked within a single kernel execution, and do not propagate across kernels.

As future work, we plan to study propagation in other GPU programming models such as OpenCL, and to build static analyzers to identify error propagation patterns in applications.

## X. Acknowledgements

We thank the anonymous reviewers of SC'16 and Igancio Laguna for their comments that helped improve the paper. This work was supported in part by a Discovery Grant from the Natural Science and Engineering Research Council (NSERC), Canada, an equipment grant from the Canada Foundation for Innovation (CFI) and Defense Advanced Research Projects Agency, Microsystems Technology Office (MTO), under contract no. HR0011-13-C-0022. The views expressed are those of the authors and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

## References

[1] Enabling on-the-fly manipulations with LLVM IR code of CUDA sources. https://github.com/apc-llc/nvcc-llvm-ir. [Online; accessed Apr. 2016].

[2] NVIDIA CUDA-GDB Documentation. http://docs.nvidia.com/cuda/cuda-gdb/#axzz45I7ljdgy. [Online; accessed Apr. 2016].

[3] NVIDIA Multi-GPU Programming. http://www.nvidia.com/docs/IO/116711/sc11-multi-gpu.pdf. [Online; accessed Apr. 2016].

[4] NVIDIA SDK Samples. http://docs.nvidia.com/gameworks/content/artisttools/hairworks/HairWorks_sdkSamples.html. [Online; accessed Apr. 2016].

[5] Parallel circuit solver. https://github.com/glaswep/hpc. [Online; accessed Apr. 2016].

[6] R. Ashraf, R. Gioiosa, G. Kestor, R. DeMara, C.-Y. Cher, and P. Bose. Understanding the propagation of transient errors in HPC applications. In *International Conference for High Performance Computing, Networking, Storage and Analysis(SC)*. IEEE, 2015.

[7] F. Ayatolahi, B. Sangchoolie, R. Johansson, and J. Karlsson. A study of the impact of single bit-flip and double bit-flip errors on program execution. In *Computer Safety, Reliability, and Security*, pages 265–276. Springer, 2013.

[8] E. Battenberg and D. Wessel. Accelerating nonnegative matrix factorization for audio source separation on multi-core and many-core architectures. In *10th International Society for Music Information Retrieval Conference (ISMIR 2009)*, 2009.

[9] S. Borkar. Designing reliable systems from unreliable components: the challenges of transistor variability and degradation. volume 25, pages 10–16. IEEE, 2005.

[10] J. Calhoun, L. Olson, and M. Snir. Flipit: An LLVM based fault injector for HPC. In *Euro-Par: Parallel Processing Workshops*, pages 547–558. Springer, 2014.

[11] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *International Symposium on Workload Characterization (IISWC 2009)*, pages 44–54. IEEE, 2009.

[12] C.-Y. Cher, M. S. Gupta, P. Bose, and K. P. Muller. Understanding soft error resiliency of Blue Gene/Q compute chip through hardware proton irradiation and software fault injection. In *Proceedings of the 2014 International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. IEEE, November 2014.

[13] H. Cho, S. Mirkhani, C.-Y. Cher, J. A. Abraham, and S. Mitra. Quantitative evaluation of soft error injection techniques for robust system design. In *ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–10. IEEE, 2013.

[14] C. Constantinescu. Intermittent faults and effects on reliability of integrated circuits. In *Reliability and Maintainability Symposium*, page 370. IEEE, 2008.

[15] J. J. Cook and C. Zilles. A characterization of instruction-level error derating and its implications for error detection. In *International Conference on Dependable Systems and Networks(DSN)*, pages 482–491. IEEE, 2008.

[16] C. H. A. Costa, Y. Park, B. S. Rosenburg, C.-Y. Cher, and K. D. Ryu. A system software approach to proactive memory-error avoidance. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. IEEE, 2014.

[17] S. Di and F. Cappello. Adaptive impact-driven detection of silent data corruption for hpc applications.

[18] C. Di Martino, W. Kramer, Z. Kalbarczyk, and R. Iyer. Measuring and understanding extreme-scale application resilience: A field study of 5,000,000 HPC application runs. In *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 25–36. IEEE, 2015.

[19] M. Dimitrov, M. Mantor, and H. Zhou. Understanding software approaches for GPGPU reliability. In *Workshop on General Purpose Processing on Graphics Processing Units*, pages 94–104. ACM, 2009.

[20] B. Fang, K. Pattabiraman, M. Ripeanu, and S. Gurumurthi. Gpu-qin: A methodology for evaluating the error resilience of GPGPU applications. In *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 221–230. IEEE, 2014.

[21] S. Feng, S. Gupta, A. Ansari, and S. Mahlke. Shoestring: probabilistic soft error reliability on the cheap. In *ACM SIGARCH Computer Architecture News*, volume 38, page 385. ACM, 2010.

[22] A. Geist. How to kill a supercomputer: Dirty power, cosmic rays, and bad solder. *IEEE Spectrum*, 2016.

[23] R. Gioiosa, J. C. Sancho, S. Jiang, and F. Petrini. Transparent, incremental checkpointing at kernel level: a foundation for fault tolerance for parallel

computers. In *Proceedings of the ACM/IEEE SC2005 Conference on High Performance Networking and Computing, 2005*, pages 9–9. ACM/IEEE, Nov 2005.

[24] W. Gu, Z. Kalbarczyk, R. K. Iyer, and Z. Yang. Characterization of linux kernel behavior under errors. In *International Conference on Dependable Systems and Networks(DSN)*, page 459. IEEE, 2003.

[25] P. H. Hargrove and J. C. Duell. Berkeley lab checkpoint/restart (BLCR) for linux clusters. *Journal of Physics: Conference Series*, 46(1), 2006.

[26] S. Hari, T. Tsai, M. Stephenson, S. Keckler, and J. Emer. Sassifi: Evaluating resilience of GPU applications. In *SELSE: IEEE Workshop of Silicon Errors in Logic*. IEEE, 2015.

[27] S. K. S. Hari, S. V. Adve, H. Naeimi, and P. Ramachandran. Relyzer: exploiting application-level fault equivalence to analyze application resiliency to transient faults. In *ACM SIGARCH Computer Architecture News*, volume 40, page 123. ACM, 2012.

[28] M. Hiller, A. Jhumka, and N. Suri. Propane: an environment for examining the propagation of errors in software. In *ACM SIGSOFT Software Engineering Notes*, volume 27, pages 81–85. ACM, 2002.

[29] K.-H. Huang and J. A. Abraham. Algorithm-based fault tolerance for matrix operations. *Computers, IEEE Transactions on*, 100(6):518–528, 1984.

[30] H. Jeon and M. Annavaram. Warped-DMR: Light-weight error detection for GPUGPU. In *IEEE/ACM International Symposium on Microarchitecture*, pages 37–47. IEEE Computer Society, 2012.

[31] W. M. Jones, J. T. Daly, and N. DeBardeleben. Impact of sub-optimal checkpoint intervals on application efficiency in computational clusters. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, HPDC '10. ACM, 2010.

[32] I. Karlin. Lulesh programming model and performance ports overview. https://codesign.llnl.gov/pdfs/lulesh_Ports.pdf. [Online; accessed Apr. 2016].

[33] I. Karlin, A. Bhatele, B. L. Chamberlain, J. Cohen, Z. Devito, M. Gokhale, R. Haque, R. Hornung, J. Keasler, D. Laney, et al. Lulesh programming model and performance ports overview. *Lawrence Livermore National Laboratory (LLNL), Livermore, CA, Tech. Rep*, 2012.

[34] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization*, page 75. IEEE, 2004.

[35] G. Li, Q. Lu, and K. Pattabiraman. Fine-grained characterization of faults causing long latency crashes in programs. In *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2015.

[36] G. Li, K. Pattabiraman, C.-Y. Cher, and P. Bose. An application-specific checkpointing technique for minimizing checkpoint corruption. In *International Symposium on Software Reliability Engineering(ISSRE)*. IEEE, 2015.

[37] Q. Lu, M. Farahani, J. Wei, A. Thomas, and K. Pattabiraman. LLFI: An intermediate code level fault injector for hardware faults. In *International Conference on Quality, Reliability and Security (QRS)*. IEEE, 2015.

[38] Q. Lu, K. Pattabiraman, M. S. Gupta, and J. A. Rivers. SDCTune: a model for predicting the SDC proneness of an application for configurable protection. In *International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, page 23. ACM, 2014.

[39] A. Moody, G. Bronevetsky, K. Mohror, and B. R. de Supinski. Design, modeling, and evaluation of a scalable multi-level checkpointing system. In *Proceedings of the 2010 International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. IEEE, 2010.

[40] R. Natella, D. Cotroneo, J. Duraes, H. S. Madeira, et al. On fault representativeness of software fault injection. volume 39, pages 80–96. IEEE, 2013.

[41] 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.

[42] K. Pattabiraman, G. P. Saggese, D. Chen, Z. Kalbarczyk, and R. K. Iyer. Automated derivation of application-specific error detectors using dynamic analysis. *IEEE Transactions on Dependable and Secure Computing*, 8(5):640–655, 2011.

[43] A. J. Peña, W. Bland, and P. Balaji. Vocl-ft introducing techniques for efficient soft error coprocessor recovery. In *International Conference for High Performance Computing, Networking, Storage and Analysis(SC)*, page 71. ACM, 2015.

[44] D. A. Reed and J. Dongarra. Exascale computing and big data. *Communications of the ACM*, 58(7):56–68, 2015.

[45] V. Reva. An efficient cuda implementation of the tree-based barnes hut n-body algorithm. 2014.

[46] V. C. Sharma, A. Haran, Z. Rakamaric, and G. Gopalakrishnan. Towards formal approaches to system resilience. In *Pacific Rim International Symposium on Dependable Computing (PRDC)*, pages 41–50. IEEE, 2013.

[47] J. A. Stratton, C. Rodrigues, I.-J. Sung, N. Obeid, L.-W. Chang, N. Anssari, G. D. Liu, and W.-m. W. Hwu. Parboil: A revised benchmark suite for scientific and commercial throughput computing. *Center for Reliable and High-Performance Computing*, 2012.

[48] J. Tan, N. Goswami, T. Li, and X. Fu. Analyzing soft-error vulnerability on GPGPU microarchitecture. In *IEEE International Symposium on Workload Characterization (IISWC)*, pages 226–235. IEEE, 2011.

[49] I. Tanasic, I. Gelado, J. Cabezas, A. Ramirez, N. Navarro, and M. Valero. Enabling preemptive multiprogramming on GPUs. In *ACM SIGARCH Computer Architecture News*, volume 42, pages 193–204. IEEE Press, 2014.

[50] A. Thomas and K. Pattabiraman. Error detector placement for soft computation. In *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 1–12. IEEE, 2013.

[51] D. Tiwari, S. Gupta, J. Rogers, D. Maxwell, P. Rech, S. Vazhkudai, D. Oliveira, D. Londo, N. DeBardeleben, P. Navaux, et al. Understanding GPU errors on large-scale HPC systems and the implications for system design and operation. In *International Symposium on High Performance Computer Architecture (HPCA)*, pages 331–342. IEEE, 2015.

[52] J. Wei, A. Thomas, G. Li, and K. Pattabiraman. Quantifying the accuracy of high-level fault injection techniques for hardware faults. In *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2014.

[53] K. S. Yim, Z. T. Kalbarczyk, and R. K. Iyer. Quantitative analysis of long-latency failures in system software. In *Pacific Rim International Symposium on Dependable Computing(PRDC)*, pages 23–30. IEEE, 2009.

[54] K. S. Yim, C. Pham, M. Saleheen, Z. Kalbarczyk, and R. Iyer. Hauberk: Lightweight silent data corruption error detector for GPGPU. In *International Parallel & Distributed Processing Symposium (IPDPS)*, page 287. IEEE, 2011.

[55] L. Yu, Y. Zhang, X. Gong, N. Roy, L. Makowski, and D. Kaeli. High performance computing of fiber scattering simulation. In *Proceedings of the 8th Workshop on General Purpose Processing using GPUs*, pages 90–98. ACM, 2015.