Data-centric Reliability Management in GPUs

Gurunath Kadam, Evgenia Smirni, and Adwait Jog
Department of Computer Science
William & Mary, Williamsburg, VA, USA
Email: gakadam@email.wm.edu, esmirni@cs.wm.edu, ajog@wm.edu

***Abstract*—Graphics Processing Units (GPUs) have become the default choice of acceleration in a wide range of application domains. To keep up with computational demands, the GPU memory system is constantly being innovated from both the cache and DRAM perspectives. Such innovations can adversely affect GPU reliability and in fact, can lead to an increase in the number of multi-bit faults. To address this problem, we systematically study a wide range of GPGPU applications and find that usually, only a small percentage of data needs protection to increase application resilience. This data is highly accessed and shared (constitutes *hot memory*), which implies that faults in this space can often lead to incorrect application output. An in-depth analysis of application code shows that information of such data can be passed on to the hardware to guide low-overhead detection/correction schemes. In this vein, we developed low-overhead partial data replication schemes that exploit latency tolerance in GPUs. Overall, this data-centric approach dramatically improves GPGPU application resilience, with a minimal additional average performance overhead of 1.2% for detection-only and 3.4% for detection-and-correction.**

***Keywords*—GPUs; Reliability; Multi-bit Faults; Application Resilience**

I. Introduction

Graphics Processing Units (GPUs) have become an inevitable part of every computing system due to their ability to provide large improvements in performance and energy efficiency compared to CPUs [2]–[4], [7], [28], [30]–[32], [37], [57], [62], [65]. Consequently, they have become the default choice for accelerating innovations in various fields such as high-performance computing (HPC), artificial intelligence (AI), and even reliability-critical autonomous vehicle software [14], [49]–[51], [53], [55], [59], [61], [65]. The emerging computing needs of these domains have fueled the growth of GPU architectures. Especially, the growing focus on deep learning has increased GPU demands tremendously. Almost every year AMD and NVIDIA unveil new GPU designs that incorporate significant innovations to their GPUs leading to improved performance and energy efficiency. For example, the latest Ampere architecture [54] has an L2 cache size that is 10x larger comparing to previous generations and new high bandwidth memories are being incorporated into almost all new GPUs.

The effect of the above innovations on GPU reliability is not yet well-understood. For example, advanced DRAM architectures make single-bit and multi-bit faults more common [44]–[46], [64], [66]. Similarly, low voltage cache design proposals (i.e., AMD Killi [17] or IBM Dante [6]) for managing power consumption of large last-level caches in GPUs [17], [54] can cause an increased number of multi-bit faults. These multi-bit faults can lead to catastrophic failures, such as accidents of autonomous vehicles [10], [24], [35], [56]. Unfortunately, the existing ECC mechanisms cannot correct multi-bit faults. SECDED is only capable of detecting up to two-bit faults and of correcting one-bit fault only. Other mechanisms such as ChipKill [11] are currently not feasible in GPUs [29]. Popular methods such as check-pointing [19], [29], [48] come with significant overhead costs due to the large amounts of data the GPGPU applications typically process [25]. Similarly, redundant computation techniques, if not carefully performed, can lead to significant overheads in terms of both performance and energy [13], [20], [40], [69], [70].

In order to provide low-overhead reliability in GPUs, especially in the context of multi-bit faults, we take a data-centric approach. Based on our extensive application-level analysis, we find that for a large number of applications, only a limited amount of data needs additional reliability protection compared to the baseline SECDED. Such data constitutes a small fraction of the entire application memory, is read-only, and is highly accessed and shared across the majority of concurrently executing warps. We show that if this data is subject to multi-bit faults, it can lead to incorrect application output (e.g., high mis-classifications errors in the case of neural networks) as the faulty data is accessed by multiple thread instructions across the majority of warps. Interestingly, we observe that this critical portion of the data can be profiled and this information can then be passed on to hardware for developing low-overhead correction and detection mechanisms.

To the best of our knowledge, this is the first work that takes a data-centric approach towards improving GPU reliability while incurring low overhead. In summary, this paper makes the following contributions:

- We perform detailed application-level analysis to show that a small fraction of critical data (hot memory blocks) used by a large number of GPGPU application threads can dramatically increase thread vulnerability to multi-bit faults. This data is usually read-only and can be profiled with low-overhead.
- We develop both detection and correction schemes for application resilience that prioritize reliability fortification of this identified critical data. Our resilience schemes leverage data information obtained from the application source code and access pattern for replicating *only* the hot memory blocks.
- Our reliability management schemes exhibit very limited overhead due to the small fraction of data that gets replicated and to the fact that the performance overhead of additional checks (and associated memory accesses) is largely hidden

thanks to the latency tolerance property of GPUs.

Quantitatively, our resilience schemes significantly improve GPU reliability by dropping the number of silent data corruption (SDC) outcomes in the application runs by 98.97% on average, while incurring a low average performance overhead of 1.2% for detection and 3.4% for detection-and-correction scheme.

II. Background and Fault Injection Setup

In this section, we present a brief overview of the baseline GPU architecture and the sources of faults in the caches and memory. Finally, we describe the fault injection model and the error metrics for each application used throughout the paper.

A. Baseline GPU Architecture

Figure 1 shows a generic GPU architecture. It is composed of a set of cores, known as streaming multiprocessors (SMs) in NVIDIA terminology. Each SM consists of an array of processing elements (PEs) and several load/store (LD/ST) units. Furthermore, each SM is associated with an L1 cache shared across the PEs. Next, all SMs on the GPU share multiple L2 cache banks which are connected through an interconnection network. Each L2 cache bank is connected to a separate memory channel. Finally, all SMs are supported by high-bandwidth off-chip global memory (DRAM). Throughout the paper, we evaluate the proposed techniques on a cycle-level GPU simulator – GPGPU-Sim [5]. Note that we assume that caches and memory are already protected by SECDED ECC and hence we focus only on the effect of multi-bit errors on application output. Table I provides more details on the simulated architecture.

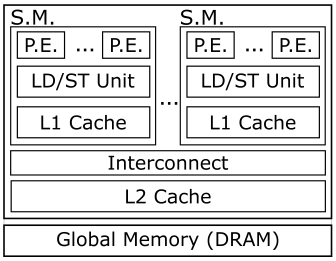

Fig. 1: A Schematic of the Baseline GPU Architecture.

TABLE I: Key configuration parameters of the simulated GPU.

Core Features	1400MHz core clock, SIMT width = 32 (16 × 2)
Resources / Core	32KB shared memory, 32KB register file, 15 SMs
L1 Caches / Core	16KB 4-way L1 data cache, 2KB 4-way I-cache 128B cache block size
L2 Caches	16-way 256 KB/memory channel (1536 KB in total), 128B cache block size
Memory Model	6 GDDR5 Memory Controllers, FR-FCFS scheduling 16 DRAM-banks, 924 MHz memory clock
Interconnect	1400MHz interconnect clock

Program Execution Model. A CUDA program consists of multiple functions known as kernels. A kernel is launched across many threads, where each thread is responsible for a set of instructions to be processed on the PEs. The threads are organized in groups, known as *Co-operative* Thread Arrays (CTAs). The number of CTAs and their size (i.e., the number of threads per CTA) are configured by the programmer at the kernel launch time. Each CTA is assigned to one SM. The number of CTAs assigned per SM is governed by the resources available per SM. Threads of a CTA are executed on the PEs at a granularity of warps, where each warp is usually a group of 32 threads. Within a warp, all 32 threads are processed in lockstep, executing the same instructions on different data.

B. Data Memory Faults in GPUs

Hardware faults arise due to particle strikes, temperature/-voltage fluctuations, or process variations [42]. Prior work has shown that GPUs are susceptible to a variety of faults [17], [41], [44], [45], [63], [64], [66]. In this work, we focus on faults occurring in the GPU memory hierarchy. Single-bit and multi-bit faults in the storage cells or the read logic of the SRAM (cache) and DRAM may cause errors in the stored data [8], [12], [26], [27]. Consequently, the application may read erroneous data resulting in silent data corruption (SDC) in its output.

The impact of the data memory faults on the application output depends on the application usage. For example, a memory fault in the GPU while executing a convolution neural network (CNN) can result in image mis-classification. If the CNN is employed in self-driving automobiles, then such mis-classification can cause catastrophic results, including loss of lives. We provide more details on the effects of faults on applications in Section III-C.

GPUs use (SECDED)-based error checking and correction (ECC) codes to address the faults in GPU caches and memory [29]. SECDED ECC detects and corrects single-bit faults, and detects double-bit faults in the application memory. However, the growing multi-bit faults are harder and expensive to detect and correct. This is the focus of this paper.

C. Fault Injection Setup

1) Fault Model: To emulate data memory errors caused by faults in caches and DRAM, we follow the error emulation framework described by Luo et al. [39]. To this end, we inject faults in the data memory blocks allocated by the application address space, irrespective of how they are mapped in the caches and DRAM. To clearly show the impact of increasing bit faults, we run two sets of fault injection experiments: first, we inject faults only in a single memory block. Second, we inject faults in 5 different memory blocks. For brevity, additional options are not shown.

1 data memory block: We select one 128B data memory block from the application address space. The memory block selection is determined by the objective of the fault injection experiment (refer to Sections III-C and V-B for details).

5 data memory blocks: Here, we select 5 128B data memory blocks from the application address space. As in the 1 memory block case, the block selection depends on the objective of the fault injection experiment.

Within the selected memory block(s), we randomly target a word to inject faults. The injected faults are modeled as

TABLE II: Output Error Metrics for Applications.

Application	Output Format	Error Metric
C-NN	Vector Classifications	Percentage of Misclassifications in output.
P-BICG	Result Vector	Percentage of output vector elements with different values than the baseline.
P-GESUMMV	Result Vector	
P-MVT	Result Vector	
A-Laplacian	Filtered Image	Normalized Root Mean Square Error compared to the baseline image.
A-Meanfilter	Filtered Image	
A-Sobel	Edge Detected Image	
A-SRAD	Image	

permanent and *stuck-at* faults. Furthermore, for stuck-at faults, we assumed that a faulty bit is stuck at either a logical 0 or 1 with equal probability. To study the effect of multi-bit faults on the application output, we inject either 2-bit, 3-bit, or 4-bit faults at random bit locations within the target word. For each setting, we execute 1000 runs to achieve statistically significant results [22], [33], [47].

2) Error Metric Selection: The faults in the data memory blocks may go undetected by SECDED-ECC in GPUs and cause an incorrect application output. This is a case of silent data corruption (SDC). To identify whether a fault-injected application run results in an SDC output, we adopt metrics tailored to each application. For the applications from the Polybench suite, the output is either a single- or multi-dimensional vector. To determine whether the output is an SDC, we note how many vector elements deviate from the fault-free baseline output vector. Applications from the Axbench suite generate images as output. Therefore, we compare the output image from a fault-injected run with the output image from the fault-free baseline run. Table II details the error metric selected for each application. For each application, we set a threshold value (either directly provided by the benchmark suite or reasonably set based on the application behavior) to determine output quality, that is, whether the application run resulted in an *SDC outcome*.

III. Motivation and Application Analysis

In this section, we first highlight the problem of increasing memory faults in GPUs. Next, we analyze the application memory access pattern and illustrate that a small fraction of data memory (hot memory) in GPGPU applications is highly accessed and shared across multiple warps. Finally, we demonstrate the vulnerability of GPGPU applications to faults in hot memory.

A. Problem Definition and Goals of This Work

Current Trends. Innovations in GPU memory systems lead to tremendous growth in performance and energy efficiency. For example, the on-chip GPU cache sizes are consistently increasing across GPU generations to accommodate increasing working data sets, see Figure 2. From the DRAM perspective, memory bandwidth and capacity are growing consistently. Advanced high-bandwidth memories (HBM) in GPUs now have up to sustained bandwidth of 1-2 TB/sec with capacities of 16-32GB [54].

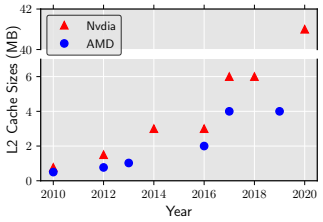

Fig. 2: L2 Cache size trends for NVIDIA and AMD GPUs.

Unfortunately, the effect of memory innovations on GPU reliability is not well understood. Recent efforts are directed towards reducing on-chip power mainly by operating the caches at near-threshold voltage [17], [18] but such reduction leads to a significant increase in multi-bit faults [18]. Previous studies have demonstrated that SECDED ECC is not sufficient to mitigate faults in DRAM [41], [63], [64]. Sridharan et al. [63], [64] showed through a field study that DRAM failures are dominated by permanent faults rather than transient faults and result in faulty data. Another field study by Martino et al. [41] compares CPU and GPU error rates and demonstrates that GPUs are three-orders of magnitude more susceptible to errors than CPUs.

Our goals. In this work, we aim to devise performance-efficient resilience schemes to address the multi-bit faults in L2-caches and DRAM. Since GPUs operate on large amounts of data in parallel, addressing the faults in the entire memory space incurs high performance and storage overhead [13]. Aiming to minimize this overhead, we propose a selective memory protection technique that is based on the observation that protecting *only a small fraction* of the data memory against multi-bit faults is sufficient to provide high reliability. To illustrate the above, we first analyze the memory access pattern of applications to identify if there is a fraction of memory with a high number of accesses comparing to the rest of the memory blocks. We show that this data is highly accessed and shared across multiple warps. We term the memory blocks of this highly accessed and shared fraction of the memory *hot memory blocks*. Next, we show that faults in hot memory blocks can result in silent data corruption (SDC) of the application output. Finally, we develop mechanisms to identify the hot memory blocks. Our analysis of several application codes shows that the hot memory blocks are usually read-only, constitute a very small fraction of the total application memory, and can be identified quickly. Using these insights, we propose two selective memory protection mechanisms, where we duplicate/triplicate the hot memory blocks to achieve low-overhead detection/correction schemes. We also show how reliability-performance trade-offs can be achieved by adjusting the amount of replication.

B. Application Access Pattern Analysis

Application Selection. To evaluate the impact of the proposed resilience schemes, we focus on applications with a clear and quantifiable output. Applications are drawn from popular benchmark suites including CUDA-SDK [52], Polybench [58],

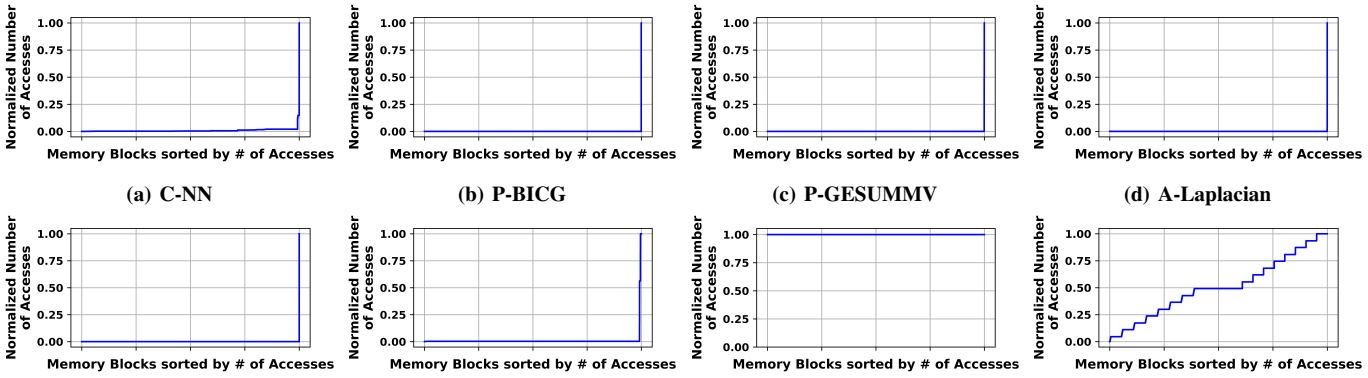

Fig. 3: Normalized number of accesses to data memory blocks. For (a)-(f), we note that very few memory blocks experience a very high number of RD accesses compared to other blocks.

and Axbench [73]. We also ensure that the selected applications show variability in terms of memory access patterns.

Application Classification based on Access Pattern. We examine the read (RD) accesses to the data memory blocks from the application address space. We focus on the RD accesses as they are the most dominant ones in the memory access pattern. We analyze the access count to each memory block of the application under observation. In Figure 3, we show example plots for 8 applications, where the RD access counts to each memory block are sorted from low to high. Based on the access patterns in Figure 3, we split the applications into two primary categories. First, for applications in Figure 3(a)-(f), we note that *few memory blocks account for a high number of RD accesses*. Specifically, for C-NN, the memory block with the highest number of RD accesses has 4732-times more RD accesses than the memory block with the least number of RD accesses. On the other hand, for applications in Figure 3(g)-(h), we note that no memory blocks have high RD accesses compared to the rest of the memory blocks. For example, for C-BlackScholes, the numbers of accesses across different memory blocks are equal. Lastly, for P-GRAMSCHM, the number of accesses increases in small steps, and therefore, there are no memory blocks with a disproportionally high number of RD accesses.

Here, we focus on applications demonstrating access profiles similar to those shown in Figure 3(a)-(f), where a small number of memory blocks accounts for a very high number of RD accesses compared to the rest of the memory blocks. Table II lists the selected applications.

Observation I: For several GPGPU applications, a small number of data memory blocks incurs a very high number of read (RD) accesses as compared to the rest of the memory blocks.

Warp-level Spread of Highly Accessed Data. Next, we profile the RD accesses of applications listed in Table II to see if the highly accessed data memory blocks are *always shared* across multiple warps. To this end, we plot the number of warps accessing the data memory blocks, with memory blocks sorted by the total number of RD accesses from low to high, see Figure 4.

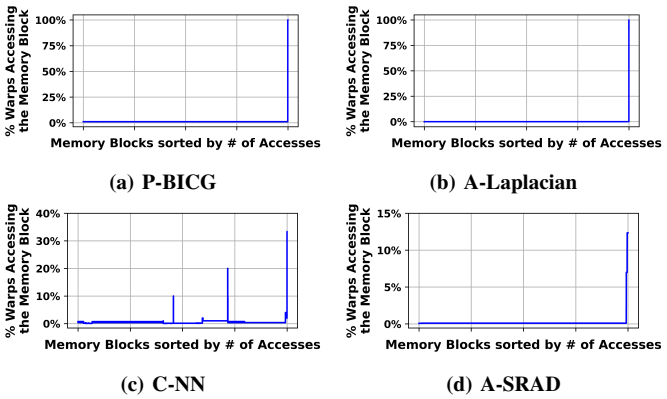

Fig. 4: Percentage of active warps accessing the data memory blocks.

Figure 4(a)-(b) (P-BICG and A-Laplacian), show that the highly accessed memory blocks are also shared across all the active warps. This trend is representative of all applications in this study except for C-NN and A-SRAD. For C-NN and A-SRAD, see Figure 4(c)-(d), we note that while the highly accessed data memory blocks are not shared across all warps, they are still highly shared across multiple warps when compared to the rest of the memory blocks.

Observation II: Highly accessed data memory blocks are typically shared across a large number of warps compared to the rest of the memory blocks accessed by the applications. Therefore, an error in the *hot memory blocks* (that are highly accessed and shared) can spread across a large number of warps, making output degradation increasingly likely.

C. Impact of Faults in Data Memory

Having identified the hot memory blocks in Section III-B, here we test our hypothesis that faults in these hot memory blocks likely cause an SDC of the application output. Figure 5 illustrates our fault injection setup to demonstrate the effect of

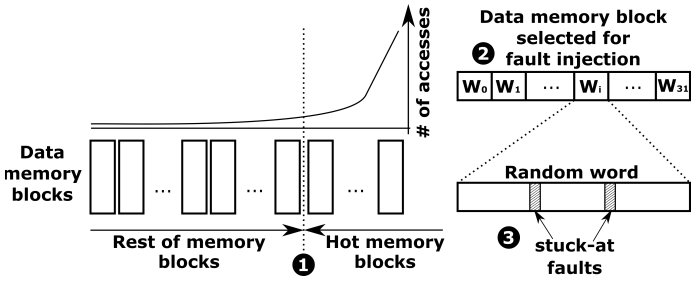

Fig. 5: Fault injection methodology to evaluate application vulnerability of hot memory blocks to the memory faults. The data memory blocks are sorted based on the total number of accesses to each block.

faults in the hot memory blocks as compared to the rest of the memory blocks. The data memory blocks are divided into two categories based on the access count profile shown in Figure 3: hot memory blocks and the rest of the memory blocks (❶). As explained in Section II-C, we do two distinct experiments: 1) with 1 block for fault injections per run and 2) with 5 blocks for fault injections per run. To demonstrate the likelihood of SDC output if faults occur in the hot memory blocks, we select random data memory blocks only from the hot memory blocks. We randomly target a word within each selected block (❷) and then inject faults at random bit locations in the target word (❸). Next, to show the likelihood of SDC output if faults are injected in the rest of the memory blocks, we select random data memory blocks only from that space for fault injection.

We compare the SDC of the application output in both cases. Figure 6 shows the number of SDC outcomes for the hot memory blocks and the rest of the memory blocks. For all applications, we notice a clear trend: the number of SDC outcomes increases as the number of faults for the selected data memory blocks increase. Furthermore, as the number of faulty data memory blocks increase (5 faulty blocks vs 1 faulty block), the number of SDC outcomes further increases.

Observation III: Faults in the hot memory blocks likely result in more SDCs in the application output comparing to faults in the rest of the memory blocks. Furthermore, as the number of faulty data memory blocks and/or the number of bit faults per data memory block increases, the probability of an SDC output increases.

IV. Data-centric Reliability Management: Analysis, Design, and Implementation

In this section, we describe the application source code analysis to identify hot data objects. Based on this analysis, we introduce two resilience schemes that prioritize the hot memory for reliability protection to minimize SDCs while incurring low performance loss.

A. Application Source Code Analysis

As noted in Observation III, the hot memory blocks must be prioritized for reliability protection. Therefore, we first identify to which input data objects in the application source code these hot memory blocks belong. We start by examining the read-only data objects in the application source code against the load instructions in the corresponding PTX code. Here, we provide the analysis for the three representative applications – namely, P-BICG, C-NN, and A-Laplacian.

We begin with P-BICG. The relatively simple source code of this application facilitates understanding the access pattern to the data objects of interest. P-BICG application accepts three read-only input data objects– `A`, `r` and `q` – for two CUDA kernel functions. Listing 1 shows the source code for the first kernel, `bicg_kernel1`, which accepts `A` and `r`. From Listing 1, we note that `A` and `r` are accessed by each thread of a kernel in a for-loop on line 14. After examining the PTX code for P-BICG in relation to the addresses of the hot memory blocks, we note that only the memory blocks of data object `r` are highly accessed. This can be seen by examining the access patterns of `A` and `r` with respect to their access indices, `[i * NY + j]` and `[i]`, respectively. Note that the offset for the index of the data objects `A` increases by a large value of `i * NY + j`. Consequently, the data memory blocks of `A` show low locality, and hence are not highly accessed and shared. On the other hand, the index of `r` increases by a small value of `i`, which results in uniformly strided accesses with a high locality. Consequently, the data memory blocks of `r` are highly accessed. We notice a similar access pattern for `q` in the second kernel of P-BICG. Out of three read-only input data objects to P-BICG, `r` and `q` experience a very high number of accesses and are shared across multiple warps.

Listing 1: First Kernel in P-BICG.

```
1  #define NX 3072
2  #define NY 3072
3
4  __global__ void bicg_kernel1(float *A, float *r, float *s)
5  {
6          int j = blockIdx.x * blockDim.x + threadIdx.x;
7
8          if (j < NY)
9          {
10                 s[j] = 0.0f;
11                 int i;
12                 for(i = 0; i < NX; i++)
13                 {
14                         s[j] += A[i * NY + j] * r[i];
15                 }
16         }
17 }
```

Next, we analyze source code and the corresponding PTX code of C-NN and observe that the data objects `Layer1_Weights` and `Layer2_Weights`, which are inputs to the kernel functions of the first (shown in Listing 2) and second (not shown here) layers of C-NN, are highly accessed and shared across different warps. Here, we focus only on `Layer1_Weights` which incurs the highest number of accesses. Note that `Layer1_Weights` is accessed on lines 11 and 15 in `FirstLayer` kernel of C-NN, see Listing 2. The access generated by a thread of a block on line 11 is to the same data element of `Layer1_Weights` across threads of a cooperative thread array (CTA)-block. As a result, the corresponding data memory blocks of `Layer1_Weights` experience a high number of accesses from a large number of threads from different warps. The next access to `Layer1_Weights` on line 15 is inside a for-loop. Additionally, the offset to the index is regular and small (`[weightBegin+i]`). Consequently, as noted for P-BICG, this results in uniformly strided accesses

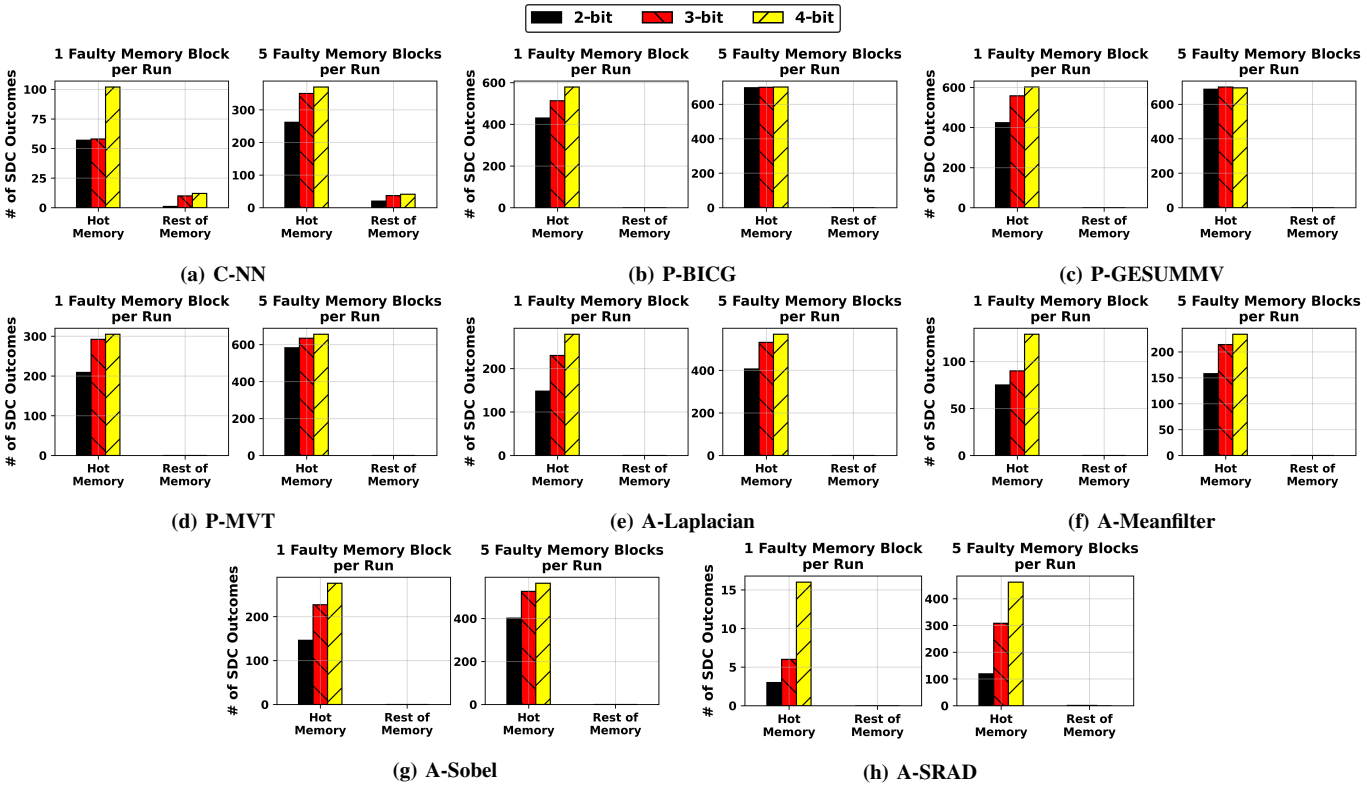

Fig. 6: Effect of faults in the hot (highly accessed/shared) memory blocks versus the rest of the memory blocks on the application output.

to the data memory blocks of `Layer1_Weights`. Since a large number of threads execute the corresponding for-loop, the data memory blocks of `Layer1_Weights` get a very high number of accesses. This is also true for the data memory blocks of `Layer2_Weights` in the next kernel of C-NN.

Listing 2: First layer in C-NN.

```
1
2  __global__ void FirstLayer(float *Layer1_Neurons,float *Layer1_Weights,float *
       Layer2_Neurons)
3  {
4    int blockID=blockIdx.x;
5    int pixelX=threadIdx.x;
6    int pixelY=threadIdx.y;
7    int weightBegin=blockID*26;
8    int windowX=pixelX*2;
9    int windowY=pixelY*2;
10   float result=0;
11   result+=Layer1_Weights[weightBegin];
12   ++weightBegin;
13   for(int i=0;i<25;++i)
14   {
15     result+=Layer1_Neurons[(windowY*29+windowX+kernelTemplate[i])+(29*29*
           blockIdx.y)]*Layer1_Weights[weightBegin+i];
16   }
17   result=(1.7159*tanhf(0.66666667*result));
18   Layer2_Neurons[(13*13*blockID+pixelY*13+pixelX)+(13*13*6*blockIdx.y)]=result;
19 }
```

Lastly, we examine the source code of A-Laplacian shown in Listing 3. The access profile (Figure 3(d)) identifies the data memory blocks of the filter data object `d_LaplacianMatrix` as the most highly accessed (see line 24) In contrast to P-BICG and C-NN, the index offset of `d_LaplacianMatrix` does not change linearly. However, since the entire `d_LaplacianMatrix` fits in one memory block, its accesses converge to that memory block. Consequently, the data memory block of `d_LaplacianMatrix` is highly accessed and shared across multiple warps. Following `d_LaplacianMatrix`, `width`, and `height` are the next most highly accessed and shared data objects.

We performed similar application source code analysis for all applications studied here. Table III lists the read-only input data objects for the GPU kernel functions of each application along with their respective sizes. The data objects are ordered from high to low in terms of the number of accesses, those identified as hot data objects are in bold and can be identified by examining the application source code. Lastly, from Table III, we note that while the hot data objects are highly accessed and shared, they occupy significantly less space than the rest of the data objects combined. For example, in C-NN, the hot data objects, that is `Layer1_weights` and `Layer1_Weights`, occupy only 2.15% of the total application data memory. We notice a similar trend across all applications.

We also performed runtime temporal analysis on the accesses to the hot data objects. Since, in most applications, accesses to the data objects are uniformly strided with small offset, the accesses have high temporal locality (e.g., for P-BICG). For other applications such as A-Laplacian, since the hot data objects are small enough to fit in few data memory blocks, accesses to these exhibit high temporal locality.

Observation IV: Through offline application source code analysis, the hot data objects forming the hot memory blocks can be identified. Furthermore, these hot data objects have a very small memory footprint (at the most 2.15%) compared to the rest of the data objects. Lastly, the hot data objects experience high temporal locality.

TABLE III: Input data objects to the GPU applications. Data objects are sorted based on the number of accesses incurred (Highest to Lowest). The emboldened data objects are classified as hot data objects (highly accessed and shared).

Application	Input Data Objects	Size of hot memory blocks normalized to the total application memory (in percentage)	Percentage of accesses to hot memory blocks w.r.t. the total number of accesses
C-NN	**Layer1_Weights**, **Layer2_Weights**, Layer3_Weights, Layer4_Weights, Images	2.15	34.99
P-BICG	**p**, **r**, A	0.064	5.7
P-GESUMMV	**x**, A, B	0.025	4.8
P-MVT	**y1**, **y2**, a	0.048	5.8
A-Laplacian	**Filter**, **Filter_Height**, **Filter_Width**, Image	0.001	73
A-Meanfilter	**Filter_Height**, **Filter_Width**, Image	0.0001	39.89
A-Sobel	**Filter**, **Filter_Height**, **Filter_Width**, Image	0.001	73
A-SRAD	**i_N**, **i_S**, **i_E**, **i_W**, Image	0.86	39.67

Note that this analysis can be adapted for other applications using available binary instrumentation tools for GPUs [1], [68]. The binary instrumentation tools offer two useful functionalities: First, the memory tracing functionality can be extended to identify the hot memory blocks. Second, the application instruction profiling at the binary level can help to identify the hot data objects. The access pattern and source code analyses are done once offline, and therefore, have no runtime overhead.

Listing 3: Filter Kernel in A-Laplacian.

```
__global__ void LaplacianFilter(Pixel* g_DataIn, Pixel* g_DataOut,
                int* width, int* height, float* d_LaplacianMatrix)
{
        __shared__ Pixel sharedMem[BLOCK_HEIGHT*BLOCK_WIDTH];
        int x = blockIdx.x * TILE_WIDTH + threadIdx.x;
        int y = blockIdx.y * TILE_HEIGHT + threadIdx.y;
        if( x < FILTER_RADIUS || x > *width  - FILTER_RADIUS - 1 || y <
              FILTER_RADIUS || y > *height - FILTER_RADIUS - 1)
        {
            int index = y * (*width) + x;
            g_DataOut[index] = g_DataIn[index];
            return;
        }
        int index = y * (*width) + x;
        int sharedIndex = threadIdx.y * blockDim.y + threadIdx.x;
        sharedMem[sharedIndex] = g_DataIn[index];
        __syncthreads();
        if( threadIdx.x >= FILTER_RADIUS && threadIdx.x < BLOCK_WIDTH -
              FILTER_RADIUS &&  threadIdx.y >= FILTER_RADIUS && threadIdx.y <
              BLOCK_HEIGHT - FILTER_RADIUS)
        {
            float sum = 0;
            for(int dy = -FILTER_RADIUS; dy <= FILTER_RADIUS; ++dy)
                for(int dx = -FILTER_RADIUS; dx <= FILTER_RADIUS; ++dx)
                {
                    float centerPixel = (float)(sharedMem[sharedIndex + (dy *
                          blockDim.x + dx)]);
                    sum += centerPixel * d_LaplacianMatrix[(dy + FILTER_RADIUS)
                          * FILTER_DIAMETER + (dx+FILTER_RADIUS)];
                }
                Pixel res = max(0, min((Pixel)sum, 255));
                g_DataOut[index] = res;
        }
}
```

B. Detection and Correction Resilience Schemes

We leverage the information related to hot memory blocks (Observations I, II, and IV) to devise detection/correction schemes. We particularly focus on Observation III that demonstrates that the hot data objects must be prioritized for protection against multi-bit faults. As discussed in Section III, the proposed resilience schemes target multi-bit faults in L2-cache and DRAM. Our resilience schemes complement the existing SECDED-ECC protection.

1) Multi-bit Fault Detection: As the read-only hot data objects prioritized for protection are smaller in size compared to the total application memory (refer to Table III), we replicate the hot data objects for "protection". Replication allows to easily identify the multi-bit faults by comparing their two copies.

Given an application, we first sort the data objects based on the number of their accesses and identify the hot data objects (this is done with a one-time offline source code analysis as described in Section IV-A). For the applications studied in this work, Table III lists all the data objects per application sorted from high to low number of accesses. The hot data objects to be prioritized for reliability protection are emboldened.

Next, we duplicate the selected data objects in the GPU DRAM at two distinct locations. During the application execution, if a memory access to the data memory blocks of one of the reliability-protected data objects is an L1-cache hit, then the normal operation takes place where the data is returned to the corresponding SM core. However, if the access is an L1-cache miss, then the LD/ST unit at the L1-cache generates two accesses, each to one of the two copies of the data memory block. Once both accesses return data to L1-cache, the copies of data are compared bit-wise to identify any multi-bit faults. If a bit mismatch is identified, then our reliability scheme generates a *terminate* signal to the GPU application causing the application to exit early and notify the user. In this case, the user is expected to rerun the application.

Since the detection-only scheme duplicates the L1-cache missed accesses to the data memory blocks of selected read-only data objects, the main source of performance loss is the additional accesses going to the L2-cache and DRAM. To minimize performance loss, we leverage the fact that this is a detection-only scheme: if the protected data is corrupted, then the application is terminated. Therefore, it is not necessary to wait for both copies of the data to arrive before proceeding with the application execution. Instead, we devise a *lazy* bit comparison: once we receive the first data copy for a corresponding load instruction, the execution moves forward. As soon as the second copy is received, then the *lazy* comparison is performed to check for multi-bit faults. Consequently, any performance loss is minimized as the execution is not stalled.

2) Multi-bit Fault Detection-and-Correction: We next describe the second resilience scheme which not only detects multi-bit faults but also corrects them. To detect and correct the multi-bit faults, we employ a majority vote mechanism that is implemented via data triplication. Each copy is stored at a distinct location in the GPU DRAM with distinct memory addresses. For each L1-cache missed access for the data object covered under the reliability scheme, we generate three accesses

to the L2-cache. Once all three accesses are returned to the LD/ST unit at the L1-cache, we perform a three-way bitwise comparison on the received data copies. During the comparison, if all the data copies have the same bits indicating no bit fault, then the application execution moves forward. If a bit mismatch is observed in one of the copies indicating a bit fault, then based on the majority vote the offending bit is changed to the correct value. The corrected bit value is used for the computation. Since the data copies are stored at distinct locations, the probability of the same bit fault occurring in all three data copies is minimal.

In this detection and correction scheme, we wait for all three data copies to be received in order to perform the three-way comparison for data correction. Consequently, the two sources of performance loss are 1) the increased number of memory accesses due to the three accesses to the data and 2) the stall times when the LD/ST unit at L1-cache waits for all three accesses to return with data. For the set of applications examined here, we do not observe a significant performance loss, because the size of the input data objects prioritized for the reliability improvement is small as shown in Table III.

C. Implementation Overhead

In Section IV-A, we identified the hot data objects via manual application of source code analysis. For an unknown application, the same access pattern analysis can be automated with the assistance of binary instrumentation tools, such as NVBit [68]. Note that this information collection is a one-time process and typically done offline. Based on the profiled information, the following steps are performed.

First, we replicate the data objects protected by our resilience schemes in GPU DRAM (either two or three times, depending on our target). We store the start addresses of each copy of the data object. These start addresses are used to generate the replication accesses to the required data index within the data object. To do so, we add the memory offset calculated for the original memory access to the respective start address. For each data object, we need either 32 bits or $(2 \times 32 =)$ 64 bits to store the start addresses in the detection-only and detection-and-correction, respectively. We allocate 128 bytes for the start address storage, which accommodates $(128B/(32 \times 4) =)$ 32 and $(128B/(2 \times 32 \times 4) =)$ 16 data objects for detection and detection/correction, respectively. In our analysis, the maximum number of data objects to a GPU application never exceeds five (Table III). We use a 32-bit adder to compute the data index mentioned above for the copy accesses.

Second, to replicate the L1-cache missed accesses to the protected data objects, we track their respective load instructions. To do so, we store the addresses of load instructions to the corresponding data objects in the LD/ST unit near L1-cache. Each load instruction needs 32 bits to store its address. We allocate 128 bytes for the instruction address storage, which accommodates $(128B/(32 \times 4) =)$ 32 load instruction addresses. In our applications, the number of load instructions does not exceed 22. The LD/ST unit checks the program counter to see if one of the load instructions to the protected data objects experiences a miss, in which case, additional accesses are generated to the copies of data objects. To compare the data copies, we use a 256-bit wide comparator for comparing the data at 32B granularity. Lastly, we allocate 128 bytes to store at most 32 load instructions awaiting the comparison of their data copies at the LD/ST unit. Note that all overheads associated with the data movement and stalls are modeled and final results already include these overheads.

V. Experimental Results

In this section, we experimentally evaluate the proposed detection-only and detection-and-correction resilience schemes using the applications listed in Table II.

A. Performance Evaluation

Note that the results presented in this subsection come from one profiling run only. Figure 7 plots for each application a) the execution time for each application and b) the L1-cache missed accesses. All metrics in Figure 7 are plotted normalized to the baseline case (i.e., the baseline execution with no resilience scheme). Therefore, the "1.0" value on the y-axis in each plot represents the baseline value. Note that the numeric values on the x-axis correspond to the cumulative number of data objects covered under the resilience schemes. The data objects covered are by their order of importance as shown in Table III. For example, for C-NN, "1" corresponds to `Layer1_weights`, while "2" corresponds to `Layer1_weights` and `Layer2_weights`, and so on.

1) Detection-Only: For evaluating performance, we focus on the overhead due to the duplication of accesses in the detection-only resilience scheme. Therefore, we ignore the cases where data memory errors result in application crashes. From Figure 7, we make the following observations. First, across all applications, as the number of data objects covered by the detection-only resilience scheme increases, the respective application execution times increase. This loss in performance is consistent with the increase in L1-cache missed accesses due to duplication. Second, the L1-cache missed accesses increase fractionally when we cover *only the hot data objects*, which is attributed to their small memory footprint in addition to their spatial and temporal locality (Observation IV). Lastly, the detection-only scheme implements a lazy bit-wise evaluation, where application execution proceeds when any copy of the duplicated data arrives at the LD/ST unit of L1-cache. (Recall that the execution does not stall awaiting both accesses to arrive.) Therefore, when only the hot data objects are protected, the corresponding performance loss on average is only 1.2%, see Figure 7. In contrast, when *all* data objects are protected, the average performance loss becomes 40.65% due to the steep increase in duplicated accesses.

2) Detection-and-Correction: We make the following observations from Figure 7 regarding the detection-and-correction scheme. First, similar to the detection-only scheme, as the number of protected data objects increases, the L1-cache missed accesses increase but this increase is larger comparing to detection-only. This is expected as accesses are now triplicated. In addition, to correct the fault(s), execution is stalled for all

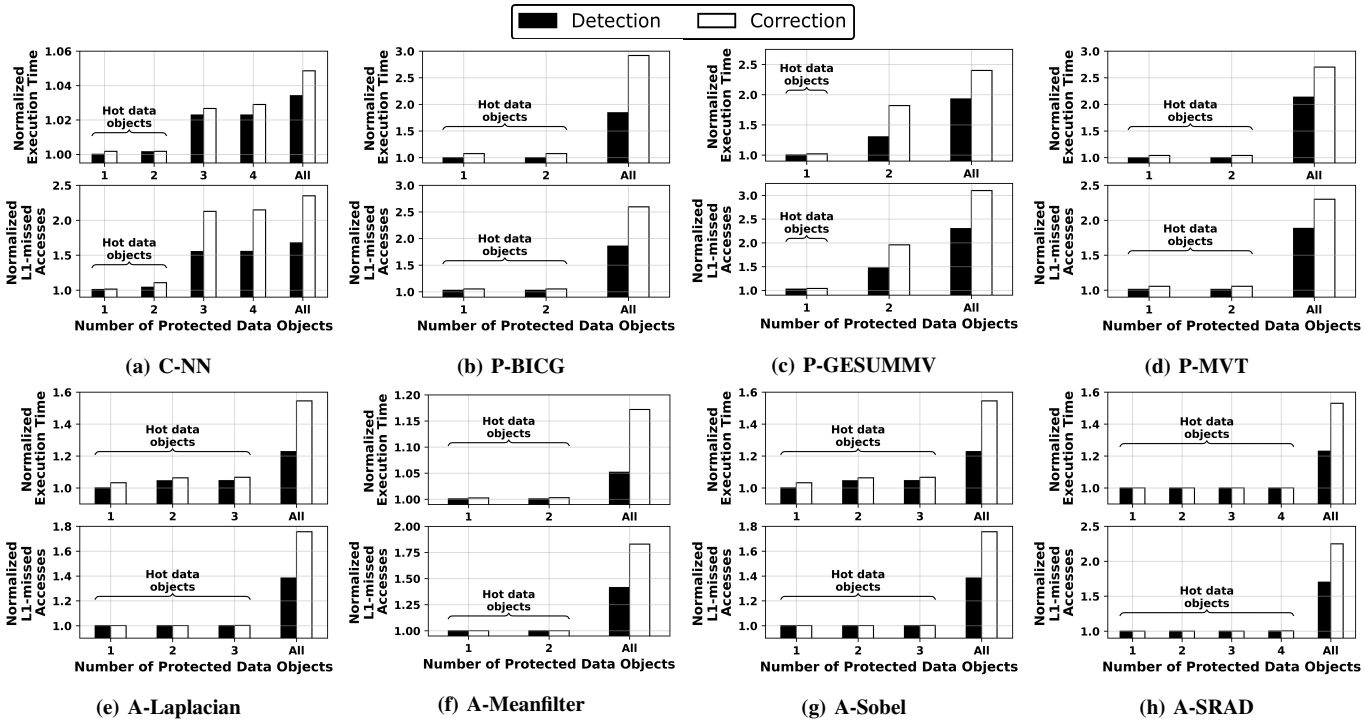

Fig. 7: Performance overhead of Detection-only (dark bar) and Detection-and-Correction (white bar) resilience schemes. All numbers are normalized to the baseline case (no reliability protection, 1.0). The hot data objects reside in hot memory blocks.

three accesses to arrive with data. Consequently, the execution time increases as a function of the volume of the protected data objects.

It is interesting to note that when we enable detection-and-correction for hot data objects only, the corresponding average performance loss is only 3.4% as the increase in the number of L1-cache missed accesses is still minimal (almost at the same level as for the detection-only scheme). If all application data objects were to be triplicated, the average performance loss shoots to 74.24%. Overall, the performance loss triggered by the detection-and-correction is much higher than for the detection-only scheme if all data objects are protected but it is nearly at the same level with detection-only if only hot data objects are protected.

B. Reliability Evaluation

We evaluate the two resilience schemes based on the percentage of SDC outcomes for 1000 fault injection experiments. Recall that this number of experiments is necessary to achieve results of statistical significance (95% confidence intervals with $\pm 3\%$ error margins) [22], [33].

We inject faults in the entire application memory space, see Figure 8. Specifically, for reliability evaluation, we select the data memory block(s) where the faults are to be injected based on its number of L1-missed accesses (a missed access forces bringing data from L2-caches and DRAM which are highly susceptible to faults) during an application run (❶). Recall that we perform two distinct experiments: 1) with 1 block for fault injections per run and 2) with 5 blocks for fault injections per run (see Section II-C). We randomly target a word within

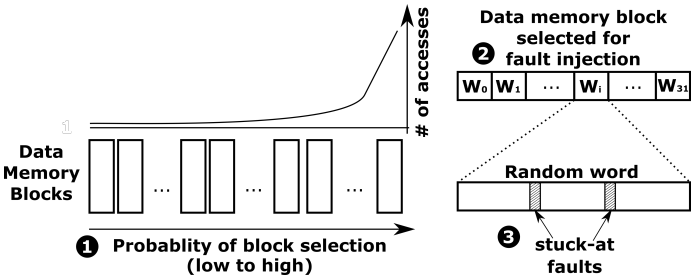

Fig. 8: Fault injection for evaluating fault detection-and-correction: the probability of a memory block selection depends on the number of its L1-missed accesses (since the proposed schemes address faults in L2-caches and DRAM).

the selected memory block(s) for fault injections (❷) and then inject faults at random bit locations in the selected word (❸).

Figure 9 plots the number of SDC outcomes versus the (cumulative) number of data objects protected by the resilience schemes. Across all applications, the baseline case with no enabled resilience scheme is more susceptible to faults. As we cumulatively protect more data objects, the number of SDC outcomes reduces. When either the per memory block bit faults or the number of faulty memory blocks increases, the number of SDC outcomes increases as well. Across all applications, protecting hot data objects with the proposed resilience schemes decreases the number of SDC outcomes significantly (an average drop of 98.97%) across all fault injection configurations.[1]

[1]In some cases, we observe that the number of SDC outcomes is less than 3% (the statistical error margins). However, the majority of cases, especially at higher fault rates, demonstrate clear benefits of our schemes.

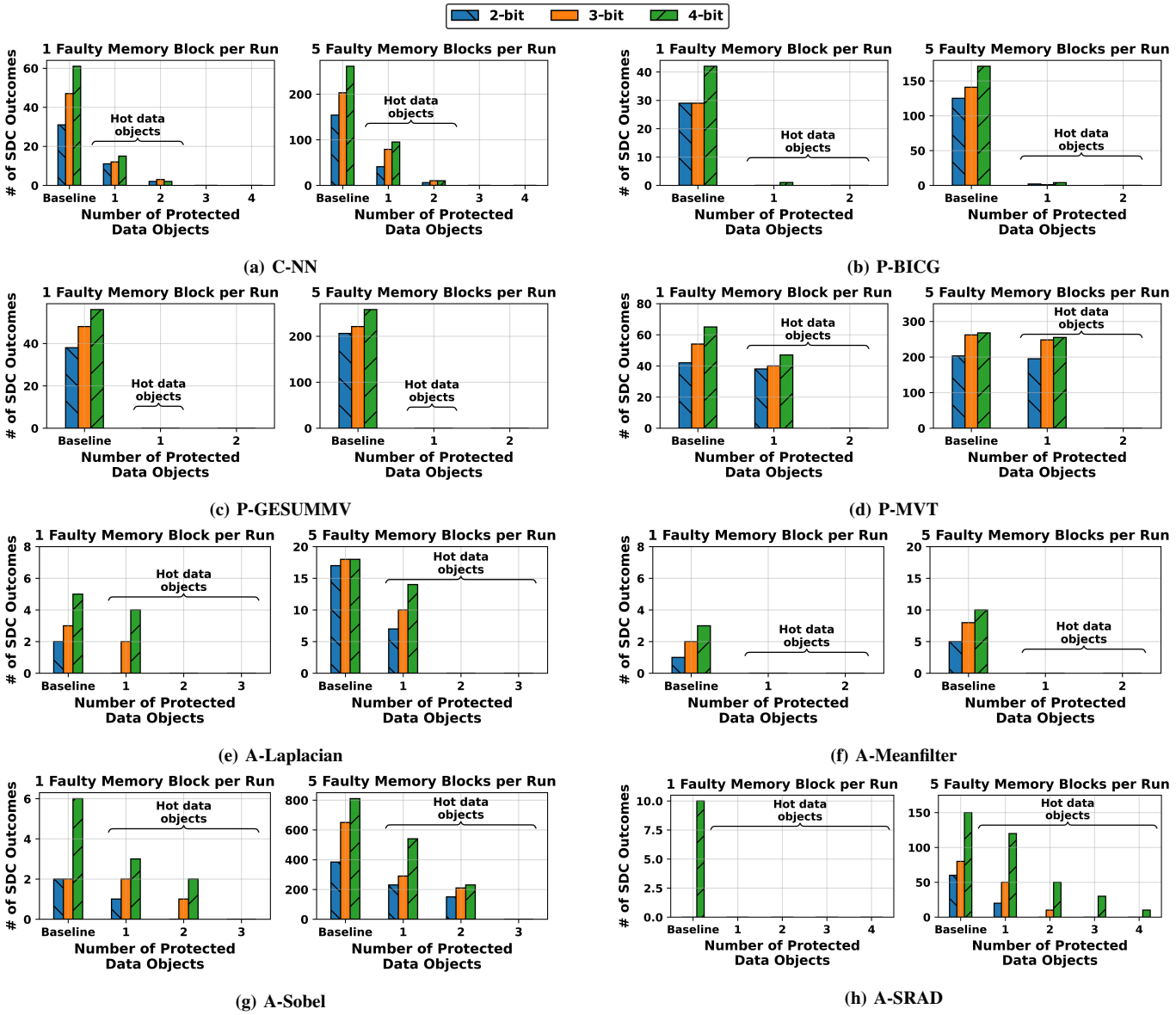

Fig. 9: Silent data corruption due to faults in L2-cache and DRAM: The x-axis represents the number of protected data objects cumulatively increasing, starting from the baseline case (no data objects are protected). The y-axis shows the number of SDC outputs out of 1000 runs for each error injection configuration. The detection-only/dectection-and-correction schemes stop the multi-bit data memory errors caused by the faults from propagating to the output.

C. Reliability and Performance Tradeoff

Figure 9 shows that in the absence of resilience schemes, GPGPU applications are highly vulnerable to multiple memory faults. Yet, as we cumulatively protect an increasing number of data objects, the number of SDC outcomes decreases, but at a small performance cost. Figure 7 shows that since the focus is on protecting a limited number of hot data objects, the performance degradation due to protection is indeed minimal. Across all applications, with the detection-and-correction (the detection-only) scheme, on average hot memory blocks can be protected with a performance loss of only 3.4% (1.2%) resulting in a 98.97% drop in the number of SDC outcomes. On the contrary, if all data are protected the performance loss becomes 74.24% (40.7%). By selecting the number of data objects to be protected and especially when protection is applied to hot data objects only, the desired reliability and performance tradeoff can be achieved.

VI. Related Work

To our knowledge, this is the first work that makes a case for data-centric reliability management in GPUs. In this section, we briefly discuss the works that are most related to ours.

Memory/Cache Errors. Sridharan et al. [64] discovered that almost half of the DRAM faults are multi-bit failures, and more than 50% of the DRAM faults are permanent. Tiwari et al. [66] showed through a large scale GPU study that GPU DRAM is

most vulnerable to multi-bit errors compared to the rest of GPU hardware. Furthermore, two independent studies demonstrate the necessity of improved ECC, such as Chipkill, instead of SECDED due to increasing multi-bit errors in DRAMs [41], [63]. Due to increasing cache sizes, several efforts have been developed to operate caches at low voltage to improve power efficiency. Recent works have demonstrated experimentally that bit faults increase as the operating voltage of the cache reduces [6], [17], [18]. In this paper, we address these multi-bit faults in cache/memory via low-overhead detection/correction mechanisms.

Error Injection Studies. GPU-Qin injects fault at the micro-architecture level to simulate transient faults in GPUs, excluding caches and memory [15], [16]. LLFI [36] is an LLVM compiler-based fault injection framework for GPUs, where an intermediate representation is modified to simulate error injection. SASSIFI [22] directly injects faults into low-level SASS instructions. PCFI [60] inject errors in different parts of instructions to simulate errors in the GPU register files and memory. Unlike the compiler-based methods used in GPU-Qin and SASSIFI, Tselonis et al. [67] propose GUFI to validate the feasibility of using the commonly used GPGPU simulator, GPGPU-Sim [5] to study the reliability of GPGPU applications. Nie et al. [47] propose a fault-site pruning mechanism that dramatically reduces the number of required fault-injection experiments in GPGPU applications to obtain results of high statistical significance, this pruning methodology is also adapted for multi-bit faults [71]. SUGAR [72] speeds up the evaluation of GPGPU application error resilience by judicious input sizing and illustrates how analyzing a small fraction of the input is sufficient to estimate application resilience with high accuracy while dramatically reducing experimentation time.

Reliability Solutions. Redundant computations by modifying source code are explored for fault tolerance as GPUs have a large number of on-chip cores [13]. Thread remapping into reliable and unreliable warps can facilitate partial replication mechanisms for error detection/correction at the warp level and shows superior performance to standard duplication/triplication [70]. Nie et al. [43] show that when a quantifiable loss in output quality is acceptable to the user, one can reduce the overhead of protection/recovery mechanisms by taking advantage of resilience patterns of threads at different hierarchies (i.e., kernel/thread-block/warp). Compiler-based redundant multithreading (RMT) compares the outputs from replicated computations for error detection, albeit with a highly variable performance loss [20], [69]. Mahmoud et al. [40] introduce a replication algorithm to duplicate select GPU instructions while maintaining low performance loss. Another approach for fault-tolerance is checkpoint-restart, where upon the fault occurrence the application restarts from the last checkpoint [19], [48]. However, the associated overhead of the checkpoint-restart mechanism is prohibitive [29].

For caches operating at low voltage, Killi [17] offers a variable ECC mechanism for a subset of L2 cache lines, while disabling the cache lines with more than one fault at the cost of cache capacity. Chandramoorthy et al. [6] implement a boosted SRAM cache, where the cache voltage is boosted for each read and write operation.

Prior works suggest heterogeneous reliability solutions for CPU workloads [21], [23], [34], [38], [39]. Hukerikar et al. [23] devise a software-based parity mechanism to improve the reliability of critical program objects in HPC applications. Luo et al. [39] show that applications exhibit different memory error resiliency based on the error location in DRAM and propose a hardware/software mechanism to enhance memory reliability. Li et al. [34] profile scientific applications to relate changes in application behavior and the location and frequency of error. SDCTune [38] identifies and protects SDC-prone program data based on static and dynamic features. Hari et al. [21] deploy low-cost program detectors in the SDC-crucial section of the program to identify and reduce SDCs. Ranger [9] restricts output values of selected layers in CNNs to minimize error propagation to improve CNN resilience.

The schemes proposed in this work complement the SECDED ECC by detecting and correcting multi-bit faults in the GPU L2 cache and DRAM. To the best of our knowledge, this is the first work that identifies the most vulnerable data in the context of GPGPU application resilience. Based on this information, our schemes protect the highly-used input data objects and provide improved reliability at a low overhead.

VII. Conclusions

Multi-bit faults are typically an unwanted side-effect of GPU memory performance innovations. In this paper, we perform an in-depth application-level analysis of memory access patterns and show that a large number of applications work on a limited number of hot data objects of highly-accessed data, which are also shared by a majority of warps. Such highly accessed and shared data is vulnerable to faults potentially leading to silent data corruption in the application output. We show that as hot data objects constitute a small fraction of the total memory footprint, protecting them against faults is an inexpensive solution that provides high application resilience in the presence of multi-bit faults.

Acknowledgements

We thank our shepherd, Karthik Pattabiraman, and the anonymous reviewers for their detailed feedback which improved this paper. This work is supported by the National Science Foundation (NSF) grant (#1717532). This work was performed using computing facilities at William & Mary.

References

[1] "The CUDA Profiling Tools Interface (CUPTI)," https://docs.nvidia.com/cuda/cupti/.

[2] M. Abdel-Majeed and M. Annavaram, "Warped register file: A power efficient register file for GPGPUs," in *19th IEEE International Symposium on High Performance Computer Architecture, HPCA 2013, Shenzhen, China, February 23-27, 2013*, pp. 412–423.

[3] M. Abdel-Majeed, D. Wong, and M. Annavaram, "Warped gates: gating aware scheduling and power gating for GPGPUs," in *The 46th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO-46, Davis, CA, USA, December 7-11, 2013*, pp. 111–122.

[4] M. Arora, S. Nath, S. Mazumdar, S. B. Baden, and D. M. Tullsen, "Redefining the Role of the CPU in the Era of CPU-GPU Integration," *IEEE Micro*, vol. 32, no. 6, pp. 4–16, 2012.

[5] A. Bakhoda, G. L. Yuan, W. W. L. Fung, H. Wong, and T. M. Aamodt, "Analyzing CUDA workloads using a detailed GPU simulator," in *IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2009, April 26-28, 2009, Boston, Massachusetts, USA, Proceedings*, pp. 163–174.

[6] N. Chandramoorthy, K. Swaminathan, M. Cochet, A. Paidimarri, S. Eldridge, R. V. Joshi, M. M. Ziegler, A. Buyuktosunoglu, and P. Bose, "Resilient low voltage accelerators for high energy efficiency," in *25th IEEE International Symposium on High Performance Computer Architecture, HPCA 2019, Washington, DC, USA, February 16-20, 2019*, pp. 147–158.

[7] N. Chatterjee, M. O'Connor, G. H. Loh, N. Jayasena, and R. Balasubramonian, "Managing DRAM Latency Divergence in Irregular GPGPU Applications," in *International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2014, New Orleans, LA, USA, November 16-21, 2014*, pp. 128–139.

[8] Q. Chen, H. Mahmoodi-Meimand, S. Bhunia, and K. Roy, "Modeling and Testing of SRAM for New Failure Mechanisms Due to Process Variations in Nanoscale CMOS," in *23rd IEEE VLSI Test Symposium (VTS 2005), 1-5 May 2005, Palm Springs, CA, USA*, pp. 292–297.

[9] Z. Chen, G. Li, and K. Pattabiraman, "A Low-cost Fault Corrector for Deep Neural Networks through Range Restriction," in *51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2021, Taipei, Taiwan, June 21-24, 2021*, 2021.

[10] Z. Chen, G. Li, K. Pattabiraman, and N. DeBardeleben, "*BinFI*: an efficient fault injector for safety-critical machine learning systems," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2019, Denver, Colorado, USA, November 17-19, 2019*, pp. 69:1–69:23.

[11] T. J. Dell, "A white paper on the benefits of Chipkill-correct ECC for PC server main memory," 1997.

[12] L. Dilillo and B. M. Al-Hashimi, "March CRF: an efficient test for complex read faults in SRAM memories," in *Proceedings of the 10th IEEE Workshop on Design & Diagnostics of Electronic Circuits & Systems (DDECS 2007), Kraków, Poland, April 11-13, 2007*, pp. 173–178.

[13] M. Dimitrov, M. Mantor, and H. Zhou, "Understanding software approaches for GPGPU reliability," in *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units, GPGPU 2009, Washington, DC, USA, March 8, 2009*, ser. ACM International Conference Proceeding Series, vol. 383, pp. 94–104.

[14] A. Eklund, P. Dufort, D. Forsberg, and S. LaConte, "Medical image processing on the GPU - Past, present and future," *Medical Image Anal.*, vol. 17, no. 8, pp. 1073–1094, 2013.

[15] B. Fang, K. Pattabiraman, M. Ripeanu, and S. Gurumurthi, "GPU-Qin: A methodology for evaluating the error resilience of GPGPU applications," in *2014 IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2014, Monterey, CA, USA, March 23-25, 2014*, pp. 221–230.

[16] B. Fang, K. Pattabiraman, M. Ripeanu, and S. Gurumurthi, "A Systematic Methodology for Evaluating the Error Resilience of GPGPU Applications," *IEEE Trans. Parallel Distributed Syst.*, vol. 27, no. 12, pp. 3397–3411, 2016.

[17] S. Ganapathy, J. Kalamatianos, B. M. Beckmann, S. Raasch, and L. G. Szafaryn, "Killi: Runtime Fault Classification to Deploy Low Voltage Caches without MBIST," in *25th IEEE International Symposium on High Performance Computer Architecture, HPCA 2019, Washington, DC, USA, February 16-20, 2019*, pp. 304–316.

[18] S. Ganapathy, J. Kalamatianos, K. Kasprak, and S. Raasch, "On Characterizing Near-Threshold SRAM Failures in FinFET Technology," in *Proceedings of the 54th Annual Design Automation Conference, DAC 2017, Austin, TX, USA, June 18-22, 2017*, pp. 53:1–53:6.

[19] R. Garg, A. Mohan, M. B. Sullivan, and G. Cooperman, "CRUM: Checkpoint-Restart Support for CUDA's Unified Memory," in *IEEE International Conference on Cluster Computing, CLUSTER 2018, Belfast, UK, September 10-13, 2018*, pp. 302–313.

[20] M. Gupta, D. Lowell, J. Kalamatianos, S. Raasch, V. Sridharan, D. M. Tullsen, and R. K. Gupta, "Compiler Techniques to Reduce the Synchronization Overhead of GPU Redundant Multithreading," in *Proceedings of the 54th Annual Design Automation Conference, DAC 2017, Austin, TX, USA, June 18-22, 2017*, pp. 65:1–65:6.

[21] S. K. S. Hari, S. V. Adve, and H. Naeimi, "Low-cost program-level detectors for reducing silent data corruptions," in *IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2012, Boston, MA, USA, June 25-28, 2012*, pp. 1–12.

[22] S. K. S. Hari, T. Tsai, M. Stephenson, S. W. Keckler, and J. S. Emer, "SASSIFI: An architecture-level fault injection tool for GPU application resilience evaluation," in *2017 IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2017, Santa Rosa, CA, USA, April 24-25, 2017*, pp. 249–258.

[23] S. Hukerikar and C. Engelmann, "Havens: Explicit reliable memory regions for HPC applications," in *2016 IEEE High Performance Extreme Computing Conference, HPEC 2016, Waltham, MA, USA, September 13-15, 2016*, pp. 1–6.

[24] S. Jha, S. S. Banerjee, T. Tsai, S. K. S. Hari, M. B. Sullivan, Z. T. Kalbarczyk, S. W. Keckler, and R. K. Iyer, "Ml-based fault injection for autonomous vehicles: A case for bayesian fault injection," in *49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2019, Portland, OR, USA, June 24-27, 2019*, pp. 112–124.

[25] A. Jog, O. Kayiran, N. C. Nachiappan, A. K. Mishra, M. T. Kandemir, O. Mutlu, R. R. Iyer, and C. R. Das, "OWL: cooperative thread array aware scheduling techniques for improving GPGPU performance," in *Architectural Support for Programming Languages and Operating Systems, ASPLOS '13, Houston, TX, USA - March 16 - 20, 2013*, pp. 395–406.

[26] S. M. Khan, D. Lee, and O. Mutlu, "PARBOR: An Efficient System-Level Technique to Detect Data-Dependent Failures in DRAM," in *46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2016, Toulouse, France, June 28 - July 1, 2016*, pp. 239–250.

[27] J. Kinseher, M. Völker, and I. Polian, "Improving Testability and Reliability of Advanced SRAM Architectures," *IEEE Trans. Emerg. Top. Comput.*, vol. 7, no. 3, pp. 456–467, 2019.

[28] D. Kirk and W. W. Hwu, *Programming Massively Parallel Processors*. Morgan Kaufmann, 2010.

[29] K. Lee, M. B. Sullivan, S. K. S. Hari, T. Tsai, S. W. Keckler, and M. Erez, "On the trend of resilience for gpu-dense systems," in *49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2019, Portland, OR, USA, June 24-27, 2019, Supplemental Volume*, pp. 29–34.

[30] S. Lee and C. Wu, "Characterizing the latency hiding ability of GPUs," in *2014 IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2014, Monterey, CA, USA, March 23-25, 2014*, pp. 145–146.

[31] V. W. Lee, C. Kim, J. Chhugani, M. Deisher, D. Kim, A. D. Nguyen, N. Satish, M. Smelyanskiy, S. Chennupaty, P. Hammarlund, R. Singhal, and P. Dubey, "Debunking the 100x GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU," in *37th International Symposium on Computer Architecture (ISCA 2010), June 19-23, 2010, Saint-Malo, France*, pp. 451–460.

[32] J. Leng, T. H. Hetherington, A. ElTantawy, S. Z. Gilani, N. S. Kim, T. M. Aamodt, and V. J. Reddi, "GPUWattch: enabling energy optimizations in GPGPUs," in *The 40th Annual International Symposium on Computer Architecture, ISCA'13, Tel-Aviv, Israel, June 23-27, 2013*, pp. 487–498.

[33] R. Leveugle, A. Calvez, P. Maistri, and P. Vanhauwaert, "Statistical fault injection: Quantified error and confidence," in *Proceedings of the Conference on Design, Automation and Test in Europe*. European Design and Automation Association, 2009, pp. 502–506.

[34] D. Li, J. S. Vetter, and W. Yu, "Classifying soft error vulnerabilities in extreme-scale scientific applications using a binary instrumentation tool," in *SC Conference on High Performance Computing Networking, Storage and Analysis, SC '12, Salt Lake City, UT, USA - November 11 - 15, 2012*, p. 57.

[35] G. Li, S. K. S. Hari, M. B. Sullivan, T. Tsai, K. Pattabiraman, J. S. Emer, and S. W. Keckler, "Understanding error propagation in deep learning neural network (DNN) accelerators and applications," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2017, Denver, CO, USA, November 12 - 17, 2017*, pp. 8:1–8:12.

[36] G. Li, K. Pattabiraman, C. Cher, and P. Bose, "Understanding error propagation in GPGPU applications," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2016, Salt Lake City, UT, USA, November 13-18, 2016*, pp. 240–251.

[37] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, "Nvidia tesla:

A unified graphics and computing architecture," *Micro, IEEE*, vol. 28, no. 2, pp. 39–55, 2008.

[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 *2014 International Conference on Compilers, Architecture and Synthesis for Embedded Systems, CASES 2014, Uttar Pradesh, India, October 12-17, 2014*, pp. 23:1–23:10.

[39] Y. Luo, S. Govindan, B. Sharma, M. Santaniello, J. Meza, A. Kansal, J. Liu, B. Khessib, K. Vaid, and O. Mutlu, "Characterizing application memory error vulnerability to optimize datacenter cost via heterogeneous-reliability memory," in *44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2014, Atlanta, GA, USA, June 23-26, 2014*, pp. 467–478.

[40] A. Mahmoud, S. K. S. Hari, M. B. Sullivan, T. Tsai, and S. W. Keckler, "Optimizing software-directed instruction replication for GPU error detection," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis, SC 2018, Dallas, TX, USA, November 11-16, 2018*, pp. 67:1–67:12.

[41] C. D. Martino, Z. T. Kalbarczyk, R. K. Iyer, F. Baccanico, J. Fullop, and W. Kramer, "Lessons Learned from the Analysis of System Failures at Petascale: The Case of Blue Waters," in *44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2014, Atlanta, GA, USA, June 23-26, 2014*, pp. 610–621.

[42] S. Mukherjee, "Architecture design for soft errors." Morgan Kaufmann, 2008.

[43] B. Nie, A. Jog, and E. Smirni, "Characterizing Accuracy-Aware Resilience of GPGPU Applications," in *20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing, CCGRID 2020, Melbourne, Australia, May 11-14, 2020*, pp. 111–120.

[44] B. Nie, D. Tiwari, S. Gupta, E. Smirni, and J. H. Rogers, "A large-scale study of soft-errors on GPUs in the field," in *2016 IEEE International Symposium on High Performance Computer Architecture, HPCA 2016, Barcelona, Spain, March 12-16, 2016*, pp. 519–530.

[45] B. Nie, J. Xue, S. Gupta, C. Engelmann, E. Smirni, and D. Tiwari, "Characterizing Temperature, Power, and Soft-Error Behaviors in Data Center Systems: Insights, Challenges, and Opportunities," in *25th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems, MASCOTS 2017, Banff, AB, Canada, September 20-22, 2017*, pp. 22–31.

[46] B. Nie, J. Xue, S. Gupta, T. Patel, C. Engelmann, E. Smirni, and D. Tiwari, "Machine Learning Models for GPU Error Prediction in a Large Scale HPC System," in *48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2018, Luxembourg City, Luxembourg, June 25-28, 2018*, pp. 95–106.

[47] B. Nie, L. Yang, A. Jog, and E. Smirni, "Fault Site Pruning for Practical Reliability Analysis of GPGPU Applications," in *51st Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2018, Fukuoka, Japan, October 20-24, 2018*, pp. 749–761.

[48] A. Nukada, H. Takizawa, and S. Matsuoka, "NVCR: A Transparent Checkpoint-Restart Library for NVIDIA CUDA," in *25th IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2011, Anchorage, Alaska, USA, 16-20 May 2011 - Workshop Proceedings*, pp. 104–113.

[49] NVIDIA, "Computational Finance." [Online]. Available: https://www.nvidia.com/en-us/gtc/topics/finance/

[50] NVIDIA, "How to harness big data for improving public health," *http://www.govhealthit.com/news/how-harness-big-data-improving-public-health*.

[51] NVIDIA, "Researchers deploy gpus to build world's largest artificial neural network," *http: //nvidianews.nvidia.com/Releases/Researchers-Deploy-GPUs-to-Build-World-s-Largest-Artificial-Neural-Network-9c7.aspx*.

[52] NVIDIA, "CUDA C/C++ SDK Code Samples," 2011. [Online]. Available: http://developer.nvidia.com/cuda-cc-sdk-code-samples

[53] NVIDIA, "JP MOrgan Speeds Risk Calculations with NVIDIA GPUs," 2011.

[54] NVIDIA, "NVIDIA A100 Tensor Core GPU Architecture," 2020.

[55] S. I. Park, S. P. Ponce, J. Huang, Y. Cao, and F. K. H. Quek, "Low-cost, high-speed computer vision using NVIDIA's CUDA architecture," in

[56] K. Pei, Y. Cao, J. Yang, and S. Jana, "Deepxplore: Automated whitebox testing of deep learning systems," in *Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017*, pp. 1–18.

37th IEEE Applied Imagery Pattern Recognition Workshop, AIPR 2008, Washington, DC, USA, 15-17 October 2008, Proceedings, pp. 1–7.

[57] B. Pichai, L. Hsu, and A. Bhattacharjee, "Architectural support for address translation on GPUs: designing memory management units for CPU/GPUs with unified address spaces," in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '14, Salt Lake City, UT, USA, March 1-5, 2014*, pp. 743–758.

[58] L.-N. Pouchet, "Polybench: the polyhedral benchmark suite," 2012. [Online]. Available: http://web.cs.ucla.edu/~pouchet/software/polybench/

[59] G. Pratx and L. Xing, "GPU computing in medical physics: A review," *Medical physics*, vol. 38, p. 2685, 2011.

[60] F. G. Previlon, C. Kalra, D. Tiwari, and D. R. Kaeli, "PCFI: Program Counter Guided Fault Injection for Accelerating GPU Reliability Assessment," in *Design, Automation & Test in Europe Conference & Exhibition, DATE 2019, Florence, Italy, March 25-29, 2019*, pp. 308–311.

[61] I. Schmerken, "Wall street accelerates options analysis with GPU technology," *(2008-11-07)[2009-11-02]. http://wallstreetandtech.com/technology-risk-management/showArticle. jhtml*, 2009.

[62] A. Sethia, G. S. Dasika, M. Samadi, and S. A. Mahlke, "APOGEE: adaptive prefetching on gpus for energy efficiency," in *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques, Edinburgh, United Kingdom, September 7-11, 2013*, pp. 73–82.

[63] V. Sridharan, N. DeBardeleben, S. Blanchard, K. B. Ferreira, J. Stearley, J. Shalf, and S. Gurumurthi, "Memory Errors in Modern Systems: The Good, The Bad, and The Ugly," in *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '15, Istanbul, Turkey, March 14-18, 2015*, pp. 297–310.

[64] V. Sridharan and D. Liberty, "A study of DRAM failures in the field," in *SC Conference on High Performance Computing Networking, Storage and Analysis, SC '12, Salt Lake City, UT, USA - November 11 - 15, 2012*, p. 76.

[65] S. S. Stone, J. P. Haldar, S. C. Tsao, W. W. Hwu, B. P. Sutton, and Z. Liang, "Accelerating advanced MRI reconstructions on gpus," *J. Parallel Distributed Comput.*, vol. 68, no. 10, pp. 1307–1318, 2008.

[66] D. Tiwari, S. Gupta, G. Gallarno, J. Rogers, and D. Maxwell, "Reliability lessons learned from GPU experience with the Titan supercomputer at Oak Ridge leadership computing facility," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2015, Austin, TX, USA, November 15-20, 2015*, pp. 38:1–38:12.

[67] S. Tselonis and D. Gizopoulos, "GUFI: A framework for GPUs reliability assessment," in *2016 IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2016, Uppsala, Sweden, April 17-19, 2016*, pp. 90–100.

[68] O. Villa, M. Stephenson, D. W. Nellans, and S. W. Keckler, "NVBit: A Dynamic Binary Instrumentation Framework for NVIDIA GPUs," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2019, Columbus, OH, USA, October 12-16, 2019*, pp. 372–383.

[69] J. Wadden, A. Lyashevsky, S. Gurumurthi, V. Sridharan, and K. Skadron, "Real-world design and evaluation of compiler-managed GPU redundant multithreading," in *ACM/IEEE 41st International Symposium on Computer Architecture, ISCA 2014, Minneapolis, MN, USA, June 14-18, 2014*, pp. 73–84.

[70] L. Yang, B. Nie, A. Jog, and E. Smirni, "Enabling Software Resilience in GPGPU Applications via Partial Thread Protection," *CoRR*, vol. abs/2103.02825, 2021.

[71] L. Yang, B. Nie, A. Jog, and E. Smirni, "Practical Resilience Analysis of GPGPU Applications in the Presence of Single- and Multi-Bit Faults," *IEEE Trans. Computers*, vol. 70, no. 1, pp. 30–44, 2021.

[72] L. Yang, B. Nie, A. Jog, and E. Smirni, "SUGAR: Speeding Up GPGPU Application Resilience Estimation with Input Sizing," *Proc. ACM Meas. Anal. Comput. Syst.*, vol. 5, no. 1, pp. 1:1–1:29, 2021.

[73] A. Yazdanbakhsh, D. Mahajan, H. Esmaeilzadeh, and P. Lotfi-Kamran, "AxBench: A Multiplatform Benchmark Suite for Approximate Computing," *IEEE Des. Test*, vol. 34, no. 2, pp. 60–68, 2017.