Exploration of TMR Fault Masking with Persistent Threads on Tegra GPU SoCs

Andrew Milluzzi, Alan George
NSF CHREC Center
ECE Dept., University of Florida
Room 330, Benton Hall
Gainesville, FL 32611
milluzzi@chrec.org

Alan George
NSF CHREC Center
ECE Dept., University of Pittsburgh
Room 1238D, Benedum Hall
Pittsburgh, PA 1526
george@chrec.org

***Abstract*—Low-power, high-performance, System-on-Chip (SoC) devices, such as the NVIDIA Tegra K1 and Tegra X1, have many potential uses in aerospace applications. Fusing ARM CPUs and a large GPU, Tegra SoCs are well suited for image and signal processing. However, fault masking and tolerance on GPUs is relatively unexplored for harsh environments. With hundreds of GPU cores, a complex caching structure, and a custom task scheduler, Tegra SoCs are vulnerable to a wide range of single-event upsets (SEUs). Triple-modular redundancy (TMR) provides a strong basis for fault masking on a wide range of devices. GPUs pose a unique challenge to a typical TMR implementation. NVIDIA's scheduler assigns tasks based on available resources, but the scheduling process is not publicly documented. As a result, a malfunctioning core could be assigned the same block of code in each TMR module. In this case, a fault could go undetected, impacting the resulting data with an error. Likewise, an upset in the scheduler or cache could have an adverse impact on data integrity.**

In order to mask and mitigate upsets in GPUs, we propose and investigate a new method that features persistent threading and CUDA Streams with TMR. A persistent thread is a new approach to GPU programming where a kernel's threads run indefinitely. CUDA Streams enable multiple kernels to run concurrently on a single GPU. Combining these two programming paradigms, we remove the vulnerability of scheduler faults, and ensure that each iteration is executed concurrently on different cores, with each instance having its own copy of the data. We evaluate our method with an experiment that uses a Sobel filter applied to a 640x480 image on an NVIDIA Tegra X1. In order to inject faults to verify our method, a separate task corrupts a memory location. Using this simple injector, we are able to simulate an upset in a GPU core or memory location. From this experiment, our results confirm that using persistent threading and CUDA Streams with TMR masks the simulated SEUs on the Tegra X1. Furthermore, we provide performance results to quantify the overhead with this new method.

TABLE OF CONTENTS

978-1-5090-1613-6/17/31.00

1. INTRODUCTION

NVIDIA's Tegra K1 and Tegra X1 System-on-Chip (SoC) devices are becoming a popular choice for high-performance embedded computing (HPEC). Combining large Graphics Processing Units (GPUs), capable of running Compute Unified Device Architecture (CUDA) applications, and low-power ARM Central Processing Units (CPUs), Tegra SoCs can achieve over 500 GFLOPS of peak Float32 performance. This compute horsepower, found only in high-power accelerators only a few years ago, is available to HPEC applications with a thermal design power (TDP) of just 10 Watts. While the computational performance is impressive, this horsepower comes at the cost of system complexity. The massively parallel structure of a Tegra X1's GPU combines hundreds of cores, a complex caching structure, a custom task scheduler, and a unified CPU-GPU memory.

Graphics rendering has been a staple of most SoCs for years; however, general-purpose GPUs (GPGPUs) are a relative newcomer. GPGPUs got their start in the data center, accelerating the largest of problems. Until recently, embedded GPUs focused mainly on driving displays. NVIDIA's Tegra SoCs are among the first low-power GPGPUs. Since embedded GPGPUs are just now becoming mainstream, their use in harsh environments is not well understood. One of the main challenges facing embedded GPGPUs in aerospace applications is reliability and fault masking in radiation-rich environments.

A Single-Event Upset (SEU) can affect a GPU in multiple ways. From a data upset in memory or cache to a logic upset in a core or the scheduler, fault mitigation is a complicated task. Much of the existing work in GPU fault tolerance focuses on algorithm-based fault-tolerance (ABFT) to catch data errors. However, ABFT does not account for logic errors in scheduler or core execution and is limited to linear algebra and other similar applications. Triple-Modular Redundancy (TMR) offers an accepted mitigation and masking strategy for most systems. By triplicating system operation and comparing the outputs, a potential error can be identified and prevented.

Applying TMR to a GPU application can be achieved through either a time-based or a concurrent solution. A time-based approach would execute a kernel three times sequentially and evaluate the results. A concurrent approach would leverage CUDA streams to run three instances of the kernel concurrently. Both approaches would easily mask data faults, but could still produce errors due to logic faults in a core or scheduler. Since all operations on a GPU rely on the scheduler to manage each thread, a logic fault in the task scheduler could go undetected. Furthermore, since the NVIDIA scheduler does not guarantee determinism in how tasks are assigned

to cores, a malfunctioning core could be used for multiple different threads, leading to incoherent results.

In order to prevent scheduler errors, we propose a new approach to GPU fault masking using *Persistent Threads (PTs)*. A Persistent Thread is a device-specific implementation of a task that is designed to run forever. This dataflow approach is similar to Field-Programmable Gate Array (FPGA) programming in Verilog. Like an FPGA, a GPU could run multiple PTs via CUDA Streams, removing much of the scheduler workload. Using PTs and CUDA Streams on an NVIDIA GPU, we can triplicate a kernel and execute it in parallel, similar to a typical TMR solution on an FPGA.

In order to evaluate Persistent Threads for fault masking, we implement this approach with a Sobel filter on a 640×460 image on a Tegra X1 SoC. We simulate SEUs via a concurrent CUDA kernel corrupting user-specified data streams. We also compare our TMR approach to a CPU-only implementation to evaluate performance. Using this approach, we are able to correct for 100% of injected faults and achieve a $1.5\times$ speedup over a CPU approach.

2. Background

NVIDIA GPUs are massively parallel devices. According to [1], the Tegra X1's GPU contains two Streaming Multiprocessors (SM). These SMs are designed according to the NVIDIA Maxwell Architecture outlined in [2]. Each SM contains 128 Float32 cores. The SM is broken up into four sections of 32 cores each. Each block of 32 cores is managed by its own Warp Scheduler. Each Warp can handle up to 32 threads. The exact method of Warp scheduling is not published; however [3] provides an overview of the threading model.

In order to effectively develop software for an NVIDIA GPU, we must first understand the threading hierarchy. As seen in Figure 1 and outlined in [3], a *thread* is the lowest level scheduled on a single core. Threads are logically grouped together into a *thread block*. A thread block can have up to 1024 threads. Thread blocks are assigned to an SM, sharing memory between threads. While only 128 threads will be executing at a time, the *Warp scheduler* will swap threads inside an SM based on memory stalls. If there are more thread blocks than SMs, then the excess blocks are enqueued until an SM is free. The grouping of thread blocks is called a *grid*.

At any given instant, 256 threads could be executing on a Tegra X1. In addition, hundreds of threads could be enqueued, waiting to execute. The typical approach to CUDA programming exploits this massive parallelism, trading determinism for high performance. Our approach to fault mitigation seeks to balance this programming paradigm with lessons learned in fault tolerance. This balance requires trading some degree of parallelism with the determinism afforded by CUDA Streams and persistent threads.

GPU Fault-Tolerance

Prior to the Tegra K1, CUDA-capable GPUs consumed tens of Watts and were not common in embedded systems. Thus, there is limited prior work in GPU fault tolerance. The authors of [4] and [5] note the overhead of implementing a TMR solution. Focusing on Cholesky Decomposition, their ABFT approach seeks to focus on correcting data errors, during both computation and storage, on high-power accelerators. This new ABFT approach introduces only 4% to 6% overhead for

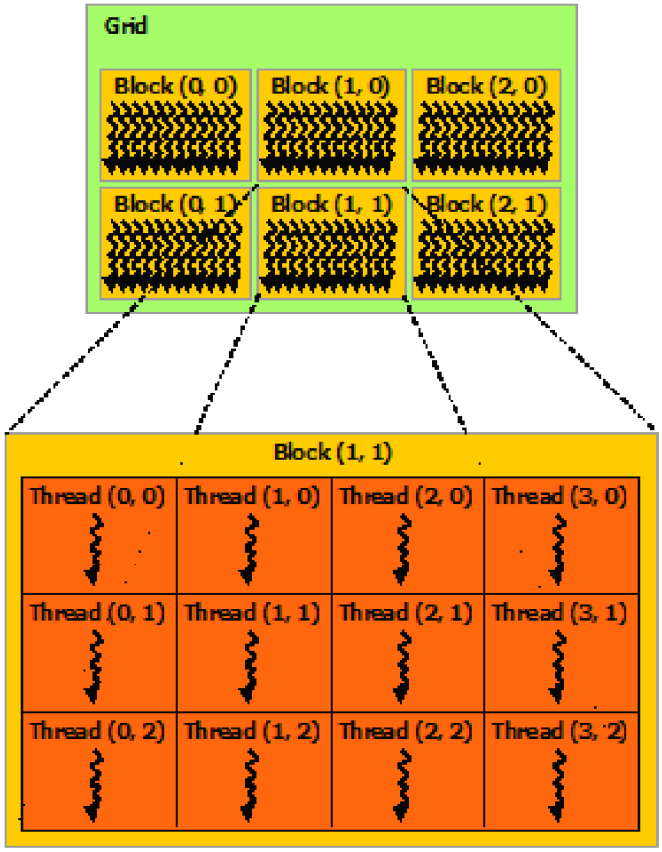

Figure 1. Overview of GPU threading model [3].

Cholesky Decomposition, making their approach comparable performance to non-fault-tolerant solutions.

Investigating ABFT with SEUs due to radiation, [6] concluded that GPU schedulers are prone to upsets. The authors investigated several benchmarks on a Tesla C2050 and a GTX480 GPU in a radiation-rich environment. They observed an increase in faults as the degree of parallelism increases. For the Matrix Multiplication benchmark, the authors enabled Error-Correcting Code (ECC) memory. While ECC memory on the GPUs was able to mitigate many of the data errors, enabling ECC reduced the total cache by over 10%. Cache size can have a profound impact on performance. Thus, the authors looked to an ABFT solution to mitigate errors, while not enabling the ECC feature. During radiation testing, the authors of [6] observed an inverse relationship between the number of multiple random errors and the number of threads assigned to a block, with the same degree of parallelism. In order to investigate the impact of parallelism and the device scheduler, they explored two simple kernels that perform 10,000 adds or multiplies, respectively. The authors found that maximum parallelism results in the largest number of errors and reducing either blocks or threads improves performance. From these experiments, the authors concluded that the GPU scheduler, managing a high degree of parallelism, can have profound impacts on fault tolerance.

Selecting the best strategy for fault masking or tolerance on a GPU is a complex process. As the authors of [7] identified, TMR provides a robust, portable solution, while incurring performance and power overhead. ABFT has the potential to increase performance, detecting and correcting single errors.

However, experimental results suggested multiple errors are likely to happen on a GPU in a radiation environment. As a result, the kernel would need to be recalculated, incurring similar overhead to TMR. The authors proposed an extension to ABFT, coined ExtABFT, to correct for multiple errors without recalculating the kernel. Applying this new approach to Matrix Multiplication, the authors achieved more efficient performance for matrices larger than 256×256. For example, a 2048×2048 ExtABFT solution achieved nearly the same performance as a non-fault-tolerant implementation. However, ABFT and ExtABFT can only be applied to a limited set of algorithms, requiring a unique implementation for each kernel.

TMR is an obvious choice for fault masking and tolerance. However, the overhead required for TMR can have adverse impacts in device performance. The authors of [8] investigate implicit Double Modular Redundancy (DMR) and TMR in GPU code execution. Since GPUs are massively parallel devices, the authors exploit two common issues with large, complex systems: underutilization and redundancy. Using redundant calculations in applications, the authors leverage an implicit form of DMR to detect errors and then force TMR once an error is detected. This approach reduces the overhead of doubling or tripling a complete system, incurring only a 8.4% and 29% overhead for DMR and TMR, respectively. If an application has good resource utilization and the algorithm minimizes redundant calculations, this approach may incur additional overhead.

CUDA Streams

Maximizing device utilization often requires a GPU to concurrently execute more than one kernel at any given instant. CUDA handles concurrency through Streams [3]. Kernels can be assigned to a stream and executed sequentially; however, multiple streams can execute on a GPU concurrently. Multiple streams enable concurrency between kernel instances. The authors of [9] provide a strong case for the benefits of concurrent kernel scheduling with regards to power efficiency. Ensuring all device resources are used is key to maximizing energy efficiency. Using CUDA Streams and a custom scheduler, they were able to achieve a 34.5% increase in operations per Watt.

Persistent Threads

Concurrency can improve power efficiency, but each kernel launch still requires runtime overhead. Persistent Threading on NVIDIA GPUs is a programming paradigm where kernel threads execute the same task until the device is reset or the program is complete. PTs adds some determinism to an application, as the scheduled block will be assigned to an SM and the threads will continue to remain active. PT was first detailed in [10]. The authors outlined four use cases of PT: CPU-GPU Synchronization; Load Balancing; Maintaining Active State; and Global Synchronization. Each use case addressed a problem where a kernel must either complete and be restarted, requiring overheard from the CUDA runtime API, or challenges within the CUDA programming model. PTs addressed these issues by creating a software scheduler that can have more insight into program dataflow than NVIDIA's hardware scheduler. However, a kernel containing PTs must also be designed to not overflow the hardware scheduler. Too many PTs will result in tasks failing to be scheduled and the system locking up. Lastly, the authors of [10] identified that PTs do not always result in better performance or improved productivity. Since there is no NVIDIA support for PTs, an efficient implementation can be complex and time-consuming.

CPU-GPU Synchronization was one of the main use cases for PTs. The authors of [11] used PTs to minimize CPU-GPU communication for genetic algorithms. They directly compared the *traditional* CUDA programming paradigm against their PT approach, resulting in a $2\times$ speedup. Highlighting the complexities involved with PTs, the authors identified the potential to simplify and automate some of the development process.

3. APPROACH

TMR on GPUs, leveraging concurrency with CUDA Streams and the determinism afforded by PT, is unexplored. While ABFT can easily detect data errors, the massively parallel nature of a GPU presents many opportunities for a logic SEU. NVIDIA documentation does not guarantee any deterministic approach to process scheduling, making it impossible to identify the source of an error. Concurrent PTs enable a more deterministic approach to scheduling while shifting additional responsibilities to the developer.

Concurrent TMR with Persistent Threading

A software TMR solution has two potential approaches on a multi-core processor: time-based and concurrent. Time-based TMR executes three times in a row, comparing the results. The time-based approach mitigates transient errors in computation, but can be affected by logic errors in the scheduler. In a modern GPU, a scheduler error could keep a thread from being completely executed. Furthermore, should a core suffer a more-permanent upset, the scheduler may still assign it tasks, leading to incoherent data.

Concurrent TMR ensures each redundant calculation is completed by a different core. In a GPU, concurrency is several groups of cores, minimizing the effects of a single malfunctioning core. However, concurrency reduces the overall parallelism of the device, potentially resulting in degraded performance. If a kernel's parallelism is not carefully calculated, the partitioned GPU may be forced to enqueue threads, resulting in potential scheduling errors.

In order to mitigate the potential scheduler upsets, PT reduces the load on the hardware scheduler. PT kernels, once started, will continue to run until the reset. By designing the grid and thread blocks of a PT kernel to map as close as possible to the device architecture, we can reduce the task switching performed by the Warp schedulers. However, miscalculating the correct grid and thread block topology could lock up the device, with PTs never getting scheduled. Memory management is another major drawback of PTs. Without a definite endpoint, it is impossible for the CUDA runtime API to know when a kernel finishes computation. Thus, it is up to the developer to create their own method of synchronization to copy data to and from the device.

TMR VGA Sobel Filter on Tegra X1

The NVIDIA Tegra X1 is computationally suited for image-processing applications and thus provides the perfect testbed for our approach. A Sobel filter on a VGA (640×480) image is a common kernel for many vision systems. Requiring two convolutions, this kernel is highly parallel and compute intensive. These characteristics ensure the device is well utilized, while still enabling scaling to allow for concurrent execution.

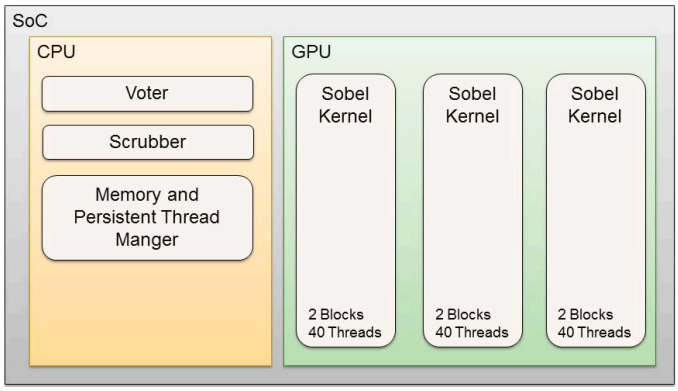

Figure 2. Concept Diagram for proposed Sobel TMR approach.

Figure 2 presents an overview of how concurrent TMR with PT can be applied to our Sobel Filtering application. The hybrid application leverages both the Tegra's CPUs and GPU to complete computation. While the massively parallel GPU performs the image processing, the CPU handles control code, scrubbing, and voting.

The Tegra X1's CPU and GPU share the same main memory. In order to use this feature, we use CUDA Mapped Memory to avoid copying data to and from the GPU. The compute aspects of TMR are well suited for a GPU; however, the comparison and voting on the outputs would not take advantage of GPU resources. As a result, we can partition our TMR approach to use the CPU cores for voting and the GPU cores for computation. Since the memory between the devices is shared, the CPU can operate directly on the GPU output, improving performance. Using double buffering, the processing and voting can happen concurrently, resulting in a pipelined application. This characteristic is important for streaming applications.

As noted in [6], minimizing scheduler workload can reduce errors in the system. For this experiment, the GPU on the Tegra X1 is partitioned into three equal sets of eighty threads for a VGA image. While sixteen cores will remain unused, the VGA image is easily divided in subframes without exceeding the total number of cores on the device. Evenly sharing the workload requires each kernel to have two thread blocks of forty threads each, as seen in Figure 2, ensuring each SM receives 120 threads. Using only 240 threads reduces the degree of parallelism, minimizing the load on the scheduler. Thus, each thread processes a 12×320 subframe of the image as seen in basic CUDA kernel pseudocode in Algorithm 1.

Algorithm 1 GPU Sobel Filter Image Partitioning

```
1:  threads = 40
2:  blocks = 2
3:  procedure SOBELFILTER(*IMAGEIN, *IMAGEOUT)
4:      lines = 480/threads
5:      pixels = 640/blocks
6:      for y = threadId×lines; y < lines; y++ do
7:          for x = blockId×pixels; x < pixels; x++ do
8:              Convolution(imageIn[x][y], filter);
```

Converting a standard CUDA kernel to a PT requires the addition of several loops and conditions as noted in Algorithm 2. The first structure to add is the main *while* loop. This loop surrounds all of the computation. Only initial configuration and memory declarations should be outside the loop. Next add a conditional statement to determine if memory is synchronized and the thread is commanded to run. If the thread is idle, it should periodically poll to see if things change. Flags monitored by the loop and conditional structures are CUDA Mapped Memory locations, shared by both the CPU and GPU, enabling the CPU to control execution.

Algorithm 2 GPU Persistent Thread Sobel Filter

```
1:  procedure PTSOBELFILTER(*IMAGEIN, *IMAGEOUT,
    *START, *STOP)
2:      DeclareMemory();
3:      CalculateThreadOffsets();
4:      CalculateBlockOffsets();
5:      while !*stop do
6:          if *start then
7:              SobelFilter(*imageIn, *imageOut);
8:              *start = 0;
9:          else
10:             Sleep();
```

Algorithms 1 and 2 outline much of the CUDA kernel. However, other than using the grid and thread block dimensions, there is little to indicate concurrency. In order to use CUDA Streams, the CPU must first create three streams. The NVIDIA kernel launch parameters have a field for a stream assignment. Since each CUDA kernel launch is asynchronous, the CPU code can sequentially start each kernel. Once the kernels are started, they will wait until the CPU sets the start flag.

Algorithm 3 CPU Sobel Filter Control

```
1:  procedure MAIN()
2:      DeclareMemory();
3:      CreateStreams(3);
4:      <<<2, 40, 0, 1>>>PtSobelFilter(args[1]);
5:      <<<2, 40, 0, 1>>>PtSobelFilter(args[2]);
6:      <<<2, 40, 0, 1>>>PtSobelFilter(args[3]);
7:      while validImages do
8:          SetMemory();
9:          *start1=1;
10:         *start2=1;
11:         *start3=1;
12:         VoteOnPreviousOutputs();
13:         while *start1 || *start2 || *start3 do
14:             Sleep();
15:     *stopAll=1;
16:     FreeMemory();
```

Before the CPU sets the start flag, the CPU must first ensure that the image is in memory. Since we are using CUDA Mapped Memory, there is no need to explicitly copy the memory to the GPU memory space. This feature is unique to the Tegra K1 and X1. Likewise, when computation is complete, there is no need to copy the memory from the device. Using double buffering, we can pipeline the application, hiding the voting behind the kernel execution. The CPU implementation is summarized in Algorithm 3.

Scrubbing

If one of the three kernels suffers a non-transient upset, the kernel may need to be reset. The proposed approach enables basic scrubbing by halting execution on one kernel, forcing the threads to complete. This operation can be triggered by

Algorithm 4 CPU Sobel Filter Control with Scrubbing

```
1:  procedure MAIN()
2:      DeclareMemory();
3:      CreateStreams(3);
4:      <<<2, 40, 0, 1>>>PtSobelFilter(args[1]);
5:      <<<2, 40, 0, 2>>>PtSobelFilter(args[2]);
6:      <<<2, 40, 0, 3>>>PtSobelFilter(args[3]);
7:      while validImages do
8:          SetMemory();
9:          *start1=1;
10:         *start2=1;
11:         *start3=1;
12:         VoteOnPreviousOutputs();
13:         while *start1 || *start2 || *start3 do
14:             Sleep();
15:     if persistentErrorDetected() then
16:         bad = getAffectedKernelNumber();
17:         stopAffectedKernel(bad);
18:         <<<2, 40, 0, bad>>>PtSobelFilter(args[bad]);
19:     *stopAll=1;
20:     FreeMemory();
```

the CPU control code as seen in Algorithm 4. Modification to support scrubbing requires a simple conditional statement if a persistent error is detected. Once the kernel is identified, it can be stopped and relaunched. Identification of a persistent fault is handled by the voter. When comparing output values, the voter tracks fault locations. Should a fault be present in the same location in three successive iterations of the kernel, the system assumes there is a persistent fault with the offending thread. Scrubbing avoids a full GPU reset, minimizing overhead to the TMR application.

Fault Injector

In order to evaluate the proposed TMR Sobel filter, we must inject faults into the Tegra X1. While the CUDA debugger can enable a step-by-step execution of a kernel, simulating repeated upsets would be a slow process. To help automate this process, a tool called SASSI [12] was created. SASSI enables instrumentation and profiling of an NVIDIA GPU to better understand kernel execution. It can be used to inject faults. However, SASSI is not currently supported on Tegra SoCs. As a result, we are forced to create our own simple fault injector.

Algorithm 5 GPU Fault Injector

```
1:  threads = 1
2:  blocks = 1
3:  procedure FAULTINJECTOR(*IMAGE, *X, *Y, ITERATIONS, *VALUE)
4:      for i =0; i ¡ iterations; i++ do
5:          image[x[i]][y[i]] = value[i];
```

To simulate a fault, we simply change a user-specified location in memory. This upset can either be injected by a CUDA kernel or the CPU, acting on mapped memory. We chose to implement our injector on the GPU to enable portability to a device without unified memory. This simpler kernel receives a pointer to the buffer to be upset, an array of the coordinates of the memory location to change, total number of upsets, and an array of new values for the memory location. The kernel can be launched in its own stream by the CPU and inject the upset. To simulate a failed thread, we inject a fault to a specific subframe for multiple iterations of the Sobel Filter.

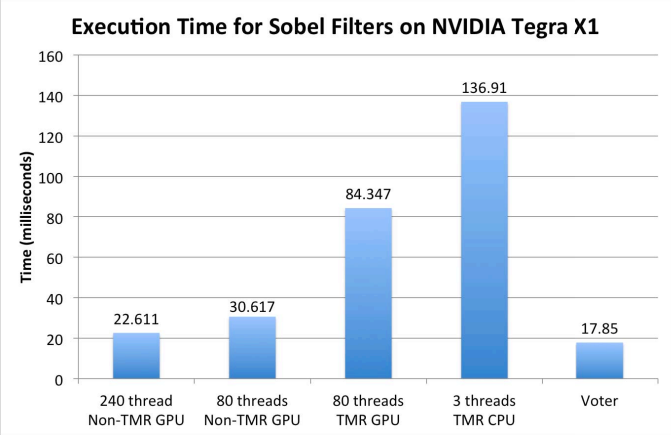

Figure 3. Comparsion of Sobel Filtering on Tegra X1.

The fault injector is outlined in Algorithm 5, enabling both single and multiple fault upsets for a given iteration.

4. RESULTS

The proposed TMR approach was implemented in C as outlined in the previous section. The code was tested to ensure it operated correctly without the introduction of vaults. Next, both transient and persistent faults were introduced into the system. GPU execution correctly produced two valid datasets and one containing errors. The CPU voter evaluated each pixel and logged masked faults. TMR correctly handled and masked all faults, and after 3 iterations, the scrubber relaunched the problem kernel. We also investigated device-wide transient faults, such that no kernel produced the correct result. While no single output image was perfect, the voter was able to mask the faults to produce the expected results. From these observations, we established that the implementation is functional.

Next we compared our approach to several Sobel Filtering implementations as seen in Figure 3. The baseline comparison is a non-TMR approach using 240 threads. Implementing the subframe approach as outlined in Algorithm 1, the grid was defined as two blocks of 120 threads. A 240-thread kernel represents the best-case implementation, while minimizing scheduler load by not exceeding the number of cores on the GPU. Performance is low due to the large subframes. As a result, we were not able to leverage the on-chip shared memory to improve device performance. In addition, since we are using a small number of threads, the GPU is unable to hide thread stalls due to memory accesses. This limited parallelism results in sub-optimal kernel performance as the kernel is highly memory-bound.

Scaling the kernel down to two blocks of forty threads, we observe almost a 50% increase in execution time. This performance is much better than expected, confirming the initial implementation is highly memory-bound. Scaling up to the our concurrent TMR approach with PTs, we observe nearly a $3\times$ increase in execution time. Since each kernel must process the entire VGA image and memory bandwidth is suspected to be the bottleneck, this result is expected.

We compared the proposed fault-masking approach to an OpenMP Sobel Filter on the CPU cores, mirroring the implementation on the GPU. While each Sobel Filter is serial on

three concurrent threads, we observe a substantially longer execution time. The proposed GPU TMR approach achieves $1.58\times$ speedup over the CPU implementation. Furthermore, the GPU implementation enables the 17.85 millisecond execution time of the voter to be hidden behind GPU computation, resulting in additional speedup.

5. Conclusions

Low-power GPUs are a growing trend in HPEC. The NVIDIA Tegra K1 and X1 enable CUDA programming and over 500 GFLOPS of peak performance at just 10 Watts TDP. While these devices are becoming more mainstream, the affects of harsh environments, such as radiation, are not well understood. With no published work on Tegra SoC beam testing, we must rely on existing work with high-power GPUs to estimate vulnerability.

Prior work on GPU fault tolerance focuses on ABFT to minimize overhead. While TMR provides a flexible approach that works with any application, ABFT requires a specialized implementation and testing. In addition, an ABFT approach can only be applied to linear algebra and other similar kernels. Evaluation of ABFT on high-power GPUs in radiation environments reveals multiple upsets in execution, requiring the GPU to recompute and compare the results. Furthermore, highly parallel applications result in more errors due to load on the hardware scheduler. As a result of this prior work, we seek to investigate a new approach to fault masking and tolerance on GPUs that is both portable and minimizes load on the device scheduler.

Building on prior work, we present a concurrent TMR approach leveraging persistent threads. PTs reduce scheduler overhead by shifting process control to the CUDA kernel. While this programming paradigm results in additional coordination for the CUDA kernel and host CPU, PTs add determinism to the scheduling process. CUDA Streams provide a method for adding concurrency to independent kernels. Combining PTs and concurrency requires careful planning to avoid exceeding device cores. The combination of PTs, TMR, and CUDA Streams results in a robust, flexible, fault-tolerant strategy for GPUs.

Evaluating the proposed TMR approach, we confirm TMR effectively identifies and corrects for faults while the scrubber detects and restarts malfunctioning threads. Performance of a fault-masking GPU Sobel filter achieves $1.58\times$ speedup compared to a CPU implementation. Furthermore, offloading computation to the GPU enables pipelining for steaming image processing, hiding the voting latency behind the GPU computation.

6. Future Work

While the proposed approach is functional, it is unclear if there is room for optimization. While PTs minimizes scheduling load, it also limits the degree of parallelism. Additional study of different data types, such as Float16, degree of parallelism, and upset rates for Tegra SoCs is required to improve performance. Warp schedulers can swap PTs, enabling more threads and blocks; however, it is unclear what impact this switching might have on application upset rates. GPUs use a high degree of parallelism and task switching to hide latency of memory accesses. Reducing block sizes enables the use of shared memory while increasing thread count. This simple optimization is key to achieving maximum device performance, but drastically increases the load on GPU schedulers.

In addition to improving performance, future work would also investigate how the proposed approach scales to the older Tegra K1 as well as higher-power GPU accelerators. The NVIDIA Tegra K1 leverages 192 Kepler architecture cores in a single multiprocessor and a unified CPU-GPU memory. This change in architecture could have adverse effects on performance as well as device utilization. Scaling up to larger devices enables more parallelism, while also introducing communication overhead to the non-unified memory.

PTs have the potential to improve performance for certain applications. While fault-tolerance benefits from the added determinism and reduced scheduling load, PTs also remove the CUDA runtime API launch overhead. CUDA runtime overhead is a major factor when launching many small kernels. Consider a low-latency, line-based, image-processing application. Kernel execution time may be shorter than the overhead in copying memory and launching the kernel. PTs could mitigate the CUDA runtime overhead, improving performance. Additional work is needed to identify other HPEC applications that could benefit from PTs.

Acknowledgments

This work was funded by the industry and government members of the NSF CHREC Center, as well as the I/UCRC Program of the National Science Foundation under Grant No. IIP-1161022. The authors would like to thank Hitarth Mehta of the NSF Center for High-Performance Reconfigurable Computing at the University of Florida for assisting with the initial investigation into Persistent Threads.

References

[1] NVIDIA. (2015) Nvidia tegra x1: Nvidia's new mobile superchip, version 1.0. Apress. Accessed: 2016-4-20. [Online]. Available: http://international.download.nvidia.com/pdf/tegra/Tegra-X1-whitepaper-v1.0.pdf

[2] M. Harris. (2014) Maxwell: The most advanced cuda gpu ever made. Accessed: 2016-4-20. [Online]. Available: https://devblogs.nvidia.com/parallelforall/maxwell-most-advanced-cuda-gpu-ever-made/

[3] NVIDIA. (2015) Cuda c programming guide, version 7.5. Accessed: 2016-4-20. [Online]. Available: http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

[4] J. Chen, S. Li, and Z. Chen, "Gpu-abft: Optimizing algorithm-based fault tolerance for heterogeneous systems with gpus," in *2016 IEEE International Conference on Networking, Architecture and Storage (NAS)*, Aug 2016, pp. 1–2.

[5] J. Chen, X. Liang, and Z. Chen, "Online algorithm-based fault tolerance for cholesky decomposition on heterogeneous systems with gpus," in *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, May 2016, pp. 993–1002.

[6] P. Rech and L. Carro, "Experimental evaluation of gpus radiation sensitivity and algorithm-based fault tolerance

efficiency," in *2013 IEEE 19th International On-Line Testing Symposium (IOLTS)*, July 2013, pp. 244–247.

[7] P. Rech, C. Aguiar, C. Frost, and L. Carro, "An efficient and experimentally tuned software-based hardening strategy for matrix multiplication on gpus," *IEEE Transactions on Nuclear Science*, vol. 60, no. 4, pp. 2797–2804, Aug 2013.

[8] M. Abdel-Majeed, W. Dweik, H. Jeon, and M. Annavaram, "Warped-re: Low-cost error detection and correction in gpus," in *2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, June 2015, pp. 331–342.

[9] Q. Jiao, M. Lu, H. P. Huynh, and T. Mitra, "Improving gpgpu energy-efficiency through concurrent kernel execution and dvfs," in *Code Generation and Optimization (CGO), 2015 IEEE/ACM International Symposium on*, Feb 2015, pp. 1–11.

[10] K. Gupta, J. A. Stuart, and J. D. Owens, "A study of persistent threads style gpu programming for gpgpu workloads," in *Innovative Parallel Computing (InPar), 2012*, May 2012, pp. 1–14.

[11] N. Capodieci and P. Burgio, "Efficient implementation of genetic algorithms on gp-gpu with scheduled persistent cuda threads," in *Parallel Architectures, Algorithms and Programming (PAAP), 2015 Seventh International Symposium on*, Dec 2015, pp. 6–12.

[12] M. Stephenson, S. K. S. Hari, Y. Lee, E. Ebrahimi, D. R. Johnson, D. Nellans, M. O'Connor, and S. W. Keckler, "Flexible software profiling of gpu architectures," in *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*, June 2015, pp. 185–197.

BIOGRAPHY

Andrew Milluzzi *received a B.S. in Computer Engineering and a B.S. in Software Engineering from Rose-Hulman Institute of Technology in 2012 and his M.S. degree in Electrical and Computer Engineering from the University of Florida in 2013. He is currently a Doctoral Candidate at the University of Florida, focusing on heterogeneous computing.*

Alan George *is Professor of ECE at the University of Florida, where he serves as Director of the NSF Center for High-performance Reconfigurable Computing (CHREC). He received the B.S. degree in CS and M.S. in ECE from the University of Central Florida, and the Ph.D. in CS from the Florida State University. Dr. George's research interests focus upon high-performance architectures, networks, systems, services, and apps for reconfigurable, parallel, distributed, and fault-tolerant computing. He is a Fellow of the IEEE. In January 2017, the lead site of CHREC, his students, and he will move to the University of Pittsburgh, where Dr. George will serve as Ruth and Howard Mickle Endowed Chair and the Department Chair of Electrical and Computer Engineering in the Swanson School of Engineering at Pitt.*