Error Resilient GPU Accelerated Image Processing for Space Applications

R. L. Davidson, *Student Member, IEEE*, C. P. Bridges, *Member, IEEE*

Abstract— Significant advances in spaceborne imaging payloads have resulted in new big data problems in the Earth Observation (EO) field. These challenges are compounded onboard satellites due to a lack of equivalent advancement in onboard data processing and downlink technologies. We have previously proposed a new GPU accelerated onboard data processing architecture and developed parallelised image processing software to demonstrate the achievable data processing throughput and compression performance. However, the environmental characteristics are distinctly different to those on Earth, such as available power and the probability of adverse single event radiation effects. In this paper, we analyse new performance results for a low power embedded GPU platform, investigate the error resilience of our GPU image processing application and offer two new error resilient versions of the application. We utilise software based error injection testing to evaluate data corruption and functional interrupts. These results inform the new error resilient methods that also leverages GPU characteristics to minimise time and memory overheads. The key results show that our targeted redundancy techniques reduce the data corruption from a probability of up to 46% to now less than 2% for all test cases, with a typical execution time overhead of 130%.

Index Terms— GPU, Parallel Processing, Error Resilience, EO, Image Processing, Image Compression, Onboard, Satellites

1 INTRODUCTION

Today, Earth Observation (EO) data is utilised by a growing number of applications from a wide range of fields. The growing demand for EO imagery has driven significant technological advances in the imaging instruments. As a result, the achievable spatial, spectral, temporal and radiometric resolutions of the remotely acquired data are continually increasing. The growing data dimensionality and subsequent increase in data volume is creating new big data challenges for space applications. This issue is enhanced onboard due to the saturation in satellite downlink performance; which is due to limitations on antenna size, transmission power, pointing abilities and the restricted availability of transmission frequencies. The resulting disparity between payload data volumes and downlink capabilities has resulted in the formation of a large onboard data bottleneck. To allow the continued delivery of high quality imagery in a timely manner this data bottleneck must be alleviated.

This research, explores onboard data processing as a solution to this challenge. Previous work conducted has focused on identifying suitable state-of-the-art image compression algorithms and the proposal of a new GPU accelerated onboard data processing system architecture. An extensive review of image compression algorithms was conducted and is summarised in [1]. In this review, the lossless image compression algorithm CCSDS-123 was demonstrated to be one of the most suitable for onboard implementation. It provides an appropriate trade-off between minimising the required computational resources and maximising compression ratio performance for raw image data [2]. In [1], a new GPU accelerated scalable onboard data processing architecture for satellites was also

proposed. To determine the feasibility and capabilities of a GPU based onboard data processing architecture, a new GPU accelerated parallel CCSDS-123 application has been developed [3]. The results from this work have demonstrated the increase in data processing throughput performance that can be achieved by exploiting GPU hardware and the massively parallel programming paradigm for real-time onboard data processing.

Outside of the Earth's protective atmosphere, the characteristics of space are distinctly different to the environment on Earth. Firstly, power consumption is heavily constrained. Traditionally, to meet the low power requirements of space systems, the devices used often trade-off computational performance. Low power GPU solutions will need to be explored for viable high performance onboard data processing solutions in this field. Additionally, for modern computing technologies, one of the most relevant and challenging differences is the radiation environment. High-energy particles ejected from our sun, supernova explosions outside of our solar system and the Earth's magnetospheres (Van Allen Belts) ability to capture and accelerate energetic particles, all contribute to the harsh radiation environment of near-Earth space [4]. Radiation can cause many different types of effects, of varying severity, in digital electronic and semiconductor devices. There are two major types of radiation effects relevant to modern electronic devices; short-term single event effects (SEE) and long-term total ionising dose (TID). Long-term TID effects are due to a build-up in ionisation over time which can result in undesired component behaviour, eventually leading to total device failure. TID can be effectively mitigated by ensuring the device is sufficiently shielded. SEE are instantaneous errors and for semiconductor based devices commonly result in the changing of a transistor

R. L. Davidson and C. P. Bridges are at Surrey Space Centre, University of Surrey, Guildford, Surrey, United Kingdom, GU2 7XH.
E-mail: rebecca.davidson@surrey.ac.uk, c.p.bridges@surrey.ac.uk

state. This can be either be temporary or permanent, therefore SEE are much more complex to model and difficult to mitigate against.

Currently, in the space sector the field programmable gate array (FPGA) is the established standard choice hardware for onboard data processing [5]. In addition to providing flexible computing with often a low power, volume and mass package, FPGAs can be manufactured to provide SEE resilience at a hardware level. Additionally, due to the wide community adoption of these devices a wide range of software based fault tolerance (SBFT) techniques have also been developed. Conversely, GPUs are not currently manufactured to specifically provide radiation tolerance and the development and testing of appropriate SBFT techniques for GPU hardware and parallel application is relatively unexplored. This is because in many traditional GPU applications the probability of experiencing a radiation induced error is low and the consequence of such an error is even lower. This poses a potential barrier in the adoption of GPU based onboard data processing systems within the space industry.

1.1 Overview

In this paper, we discuss and address the key challenges related to the deployment of a GPU based onboard data processing system in space applications. Concepts specifically relating to GPU processing and the radiation environment of space, are introduced in Section 2. In Section 3, our representative onboard space application for image processing is presented. This includes discussions on the GPU acceleration approach and reports new data processing performance results. These data processing throughput results are given for two GPU computing platforms, the first is a traditional desktop NVIDIA GPU, the GTX 750 Ti [6]. The second is the low power embedded NVIDIA GPU platform, the Jetson TX1 [7]. The Jetson TX1 is used in this research to provide a suitable reference GPU computing platform which is indicative of the low power constraints of space applications. In Section 4, a practical study to assess the error resilience of our GPU image processing application is assessed using software based error injection testing. These results are then used to develop two error resilient versions of the application. Both versions leverage a targeted SBFT scheme and exploit characteristics of the GPU hardware and software model to reduce the introduced execution and memory overhead. Finally, software based error injection experiments are repeated with our modified applications and the results are presented. These are used to quantify the improvements in error resilience of the techniques used. Additionally, we perform performance testing to measure the execution time and memory usage overheads of implementing these error mitigation techniques. Section 5 provides the authors concluding remarks and Section 6 details proposed future research.

2 GPUS AND THE SPACE ENVIRONMENT

This section introduces important concepts relating to GPUs, the massively parallel programming paradigm and discusses the challenges surrounding the deployment of these devices within the environment of space.

Whilst many of the principles of parallel and GPU computing are translatable and non-vendor specific, our research leverages NVIDIA GPU hardware and the NVIDIA CUDA (Compute Unified Device Architecture) programming API. Therefore, several details and terminology specific to NVIDIA hardware and software are also discussed.

2.1 NVIDIA GPU Hardware

NVIDIA is a leading hardware design company that develops state-of-the-art GPUs, tailored specifically to several application groups including gaming, high-end graphical processing, data processing and mobile computing. Whilst changes are made to the underlying hardware typology with each new architecture generation, the major hardware blocks and hierarchy have remained constant [6]. The GPU can be broadly classified as containing three types of structures; control, memory, and computational blocks, as depicted in Fig. 1. Control blocks are those that provide management capabilities, determining the behaviour of other functional blocks of the GPU. Memory structures are those that primarily store information and computational structures are those that execute instructions.

Fig. 1. Simplified block diagram of the hierarchical hardware architecture of a typical NVIDIA GPU

As shown in Fig. 1, at the top-level the GPU contains several control blocks. This includes dedicated memory controllers, which allow the GPU to access off-chip global DRAM, and a host interface, which provides a direct communication channel between the GPU and its host. GPUs are often deployed in a heterogeneous computing system, where specific work is offloaded from a host, often a CPU, to the GPU which acts as an accelerator. In this configuration, the host is responsible for invoking work on the GPU. Additionally, at the top level, the GPU features L2 hardware controlled cache memory, which caches off-chip global DRAM memory transactions and is shared by all computation blocks. It also has a GigaThread Engine, which is responsible for scheduling and distributing blocks of work to the streaming multiprocessors (SMs).

The architecture of an SM can be further described as containing several control, memory and computational functional blocks. Inside a SM, the warp scheduler and dispatch units are responsible for managing and distributing threads in groups, called a warp, to the execution units (CUDA cores). A warp is a group of 32 threads and all threads in a warp execute the same instruction simultaneously. For computational work a thread is either mapped

to a single CUDA core, a special function unit (SFU) or a Load/Store unit. SFUs allow efficient execution of functions such as square root, reciprocal and trigonometric functions. Load/Store units are responsible for issuing memory operations to the appropriate memory structures. In addition to top-level Global and L2 cache memories, each SM also provides several memory structures. The register file is shared by all threads on the SM. Each thread has ownership of its own registers which can be accessed only by the owner thread. All threads on a SM also have access to hardware managed L1 cache and a shared memory structure. Shared memory is user managed whilst L1 cache is hardware managed, however, both provide much higher bandwidth and lower latency compared to global off-chip global DRAM. Shared memory enables memory to be shared between threads in the same block, increasing opportunity for threads within the same block to cooperate and increase memory reuse to reduce off-chip memory access traffic.

2.2 NVIDIA GPU Software

To accompany their GPU hardware, NVIDIA have also developed the CUDA API [8]. CUDA is a relatively mature parallel programming API based on the C programming language. The CUDA programming model follows a hierarchical structure, similar to the underlying hardware. At the top level the software developer organises functional work into kernels. A kernel describes the sequential execution path for a single thread on the GPU. Parallelisation is then achieved as the programmer declares the number of threads that will be invoked to concurrently execute the work described in the kernel. Threads can identify themselves using index numbers which can also be used to allow each thread to access different data. In the CUDA software model, the programmer can also organise groups of threads into blocks. All threads within a block have visibility of shared memory resources allowing them to cooperate; threads from different blocks do not have this ability. The CUDA API also allows multiple kernels to be invoked on the GPU concurrently, using the CUDA streams construct. When a single kernel does not provide enough work to fill the GPU, additional kernels can thus be initiated and executed concurrently.

2.3 Traditional Computing in Space

For onboard computing in the space environment, characterising the SEE propagation and error resilience of the devices used can be key to successful system design. The impact of SEE on the system can vary greatly, depending upon the exact functionality of the underlying hardware and how it is used by an application.

Whilst there is growing availability of radiation hardened by design components for use in space, there has been a shift towards the greater utilisation of commercial-off-the-shelf (COTS) components in the commercial satellite sector in recent years. When compared with traditional radiation hardened components, COTS components can provide up to several orders of magnitude increase in comparative performance. Therefore there is often a trade-off to be made between the risk of SEE and

the computing performance.

Error resilience is defined as the ability of a system to withstand errors should they occur. An interaction between a device and a radiation particle may or may not result in a hardware error. Subsequently, a hardware error may or may not result in an error in the application output. This is due to many different error masking effects, which can be inherent or designed, and induced at the different system layers [9] [10]. Understanding the inherent error resilience of a system is key to designing and selecting appropriate mitigation techniques. For COTS components, there is very little that can be done by a third-party to increase error resilience at the hardware level. Therefore, many different methodologies and principles to mitigate the risks posed by the radiation environment on COTS components have been researched. Of these, SBFT techniques are among the most popular. There are two major types of SBFT techniques which are deployed; algorithm based fault tolerance (ABFT) and generic fault tolerance. ABFT principles are based on the encoding of the data used by the algorithm and modification of the algorithm to work with the encoded data to provide greater error resilience. Whilst ABFT incurs only moderate memory and computational overheads they are not applicable for all algorithms. Generic techniques have gained wider spread use in recent years, due to their ease of implementation and wide spread applicability and fault protection coverage [11].

Generic techniques include the implementation of error detection and correction codes (EDAC) [12], checkpointing and rollback for execution pipelines [13] and redundancy based mechanisms for both memory and computational blocks [14]. Redundancy based software techniques, such as dual or triple modular redundancy (DMR, TMR) can be deployed in software for both memory and computation blocks. For computational pipelines DMR and TMR techniques can be employed either in a spatial or temporal manner at different levels such as the instruction, procedural or program level. TMR is one of the most popular techniques as it can be used to both detect and correct errors in memory and computational elements of a design. In TMR the memory or computational pipeline is replicated three times. The results from each, can then be compared to detect errors and using a majority voter an error in one of the replicated streams can be corrected. Whilst its generic approach allows it to be implemented at different architectural levels and easily adapted to a wide range of algorithms, implementing TMR often results in high memory and execution time overheads.

2.4 Radiation Effects on GPUs

Unlike FPGA fabric, the GPU architecture is heterogeneous and constructed from several different types of specialised hardware. Additionally, the exact construction of many commercial GPUs is not openly published and such information is highly proprietary in nature. This makes it extremely difficult to analytically predict how these devices will behave in a radiation environment.

However, the effects of an SEE on a GPU application

can be broadly classified as one of three outcomes; a functional interrupt (FI), a silent data corruption (SDC) or a masked event [9]. A FI is defined as an error that causes an application to hang or malfunction so it does not successfully complete. FI's can be identified by either a non-zero exit status or the timeout of a watchdog timer. Understanding and mitigating FI's is particularly important in space data processing systems which often have very strict and deterministic timing requirements. An SDC is defined as occurring when the application successfully completes but the output data is incorrect. Protecting against SDC can also be very important for our application, as any data corruption caused in the onboard data processing chain will result in the permanent loss of data. Additionally, SDC errors are extremely difficult to detect and correct for in later stages of the processing chain. A masked event is when the application completes successfully and no data errors are found in the output.

From a theoretical understanding of the hardware architecture, described in Fig. 1, we can speculate the type of errors induced in different GPU structures. For instance, we would expect error induced in control blocks, such as the schedulers, to result in a FI. Whilst errors in memory and control blocks could be more vulnerable to SDC affects. We can also postulate that memory structures which are shared by multiple SM's and multiple threads within an SM, will likely have a greater error propagation, in comparison with SDC caused by errors in computation blocks, such as the CUDA cores, which are only responsible for the execution of a single thread. However, due to the distinctive hardware and software abstraction layers and the highly-threaded nature of the software, error propagation characteristics can be extremely complicated to accurately model. Error masking can occur at each abstraction layer, making determining the overall error resilience extremely difficult [10].

3 HIGH DATA THROUGHPUT PROCESSING ON GPUS

A key requirement for next generation onboard data processing is to achieve high data processing throughput. Achieving high onboard data processing throughput is important to ensure real-time data processing constraints are met. Therefore, we have developed a parallelised GPU accelerated application, which represents a typical onboard data processing algorithm. This section introduces the specific algorithm studied, the approach to GPU accelerated parallelisation and new performance results.

3.1 CCSDS-123 Algorithm Overview

The algorithm studied in this work is CCSDS-123, a lossless predictive image compression algorithm. The algorithm was originally developed by NASA JPL (National Aeronautics and Space Administration, Jet Propulsion Laboratory) and was specifically designed for processing multispectral and hyperspectral data sets. Due to its ability to achieve competitive compression ratio performance whilst minimising computational requirements it has been subsequently adopted and standardised by the Consultative Committee for Space Data System (CCSDS). In-depth technical details of the CCSDS-123 algorithm have been

covered thoroughly in several publications including the original algorithm publication and the CCSDS standard document. Therefore, only an overview of the algorithm is provided in this work and the reader is referred to these publications for further details on the algorithm [2][15][16].

CCSDS-123 has been chosen as a case study for this research because it is the state-of-the-art lossless image compression algorithm for next generation onboard image processing, as discussed in [1]. Additionally, it is characteristically representative of other typical onboard data processing algorithms, featuring elements which are highly sequential in nature, which can be challenging to implement efficiently on GPU hardware. The algorithm is also composed of a wide range of computationally intense mathematical functions such as multiplication, addition and vector cross products which are common elements of many data and image processing algorithms. Algorithm 1, provides a simplified structural overview of CCSDS-123 highlighting key functional blocks and showing the overall data flow. It emits control blocks and user defined parameter usage for simplicity.

Algorithm 1 – Serial Simplified CCSDS-123

```

1. Procedure CCSDS-123 (*IN, *OUT)
2.   for z = 1; IM_BANDS; z++
3.     for y = 1; IM_HEIGHT; y++
4.       for x = 1; IM_WIDTH; x++
5.         if (y==1) && (x==1) then
6.           Initialise_Weights(*W)
7.         end
8.         LS=Calc_Local_Sum(*IN)
9.         *LD=Calc_Local_Differences(LS, *IN)
10.        PRED_LD=Dot_Product(*W, *LD)
11.        PRED=Calc_Prediction(PRED_LD, LS)
12.        Update_Weights(*W, *LD, PRED, *IN)
13.        RESIDUAL(x,y)=Map_Residual(PRED, *IN)
14.      end
15.    end
16.  end
17.  *OUT=Entropy_Encode(*RESIDUAL)

```

The algorithm exploits spectral data redundancies in addition to traditional spatial redundancies by utilising information from a small 3D neighbourhood of previously encoded pixels, from up to 15 adjacent image bands, to influence the prediction. Ultimately, CCSDS-123 operates in an iterative manner over each pixel sequentially, as represented by the nested loops in lines 2-4. A key novelty of this scheme is the use of the sign algorithm, a low complexity variation of the least mean square algorithm, to produce optimised band weightings for accurate prediction. These weightings need to be initialised for the first pixel in each band as shown in lines 5-7. The weight initialisation values are determined by a simple calculation based on a user defined parameter. The sign algorithm is used in combination with a local difference method to improve convergence speed and prediction accuracy. This is achieved firstly by the calculation of a local sum value (line 8). The local sum is a weighted sum of previous neighbouring pixels. This integer value is then used to construct a local difference vector (line 9). The local differences represent the scaled

difference between the local sum and actual pixel values in both spatial and spectral dimensions. The dot product of the local difference and the updated weight vectors, line 10, are provided to the prediction function, line 11. The weightings for each spectral band used in prediction are then updated inside a feedback loop to minimise the prediction error (line 12). The prediction residual, the difference between the predicted value and the actual pixel value, is then calculated and mapped to an integer range for reversible decoding (line 13). The residuals are then entropy encoded using a variable length scheme to produce a final compressed bitstream representation of the data (line 17).

3.2 Parallel Implementation Approaches

The CCSDS-123 algorithm has not been specifically designed to provide inherent parallelism. Initially, it may seem there is little opportunity to exploit any thread level parallelism (TLP). However, for parallelisation it is important to understand data dependencies in additional to functional dependencies. Fig. 2 shows a function block diagram of the CCSDS-123 algorithm, arrows indicate functional dependencies and highlights the different data dependencies of each functional block. Several functional blocks of this algorithm contain no internal data dependencies (blocks with no background shading in Fig. 2) meaning these blocks could be fully parallelised exposing TLP equal to the number of pixels in the image. However, several blocks (shaded in grey in Fig. 2) contain spatial dependencies limiting the amount of parallelism that can be exploited to be equal to the number of spectral bands in the image. This dependency is induced by the spatial serial dependency of the weight update feedback loop and subsequent dependencies of the weight vector itself.

An initial parallelism approach could be to classify and then separate the constituent functions of the algorithm based on the amount of parallelism available at each stage. Whilst this implementation strategy allows the maximum level of parallelism to be exploited, it often leads to a heavily memory bound application. On the GPU, memory operations have proportionally much larger associated latencies. Therefore, GPUs are better suited to hiding instruction latency and alleviating computationally bound problems. Additionally, kernel and memory initialisation overheads are not trivial in GPU computing, therefore to achieve high

performance for multi-kernel applications it is essential to ensure enough work is conducted in each kernel to mitigate these overheads.

As a result, a parallelisation approach which is typically better suited for memory bound problems on GPUs is one that limits the degree of parallelism exploited. Firstly, by reducing the number of kernels we can reduce the kernel and memory initialisation overhead. Secondly, this increases the opportunity for the GPU to exploit low-level instruction level parallelism and context switching to effectively hide instruction latencies. Finally, this also allows low latency on-chip memory to be utilised, minimising the memory induced latencies of the application. This approach has been taken in the development of our CCSDS-123 implementation and is demonstrated in Algorithm 2.

Algorithm 2 – GPU Accelerated Parallel Simplified CCSDS-123

```

1. Threads = IM_BANDS
2. Kernel CCSDS-123 (*IN, *OUT)
3.   for y = 1; IM_HEIGHT; y++
4.     for x = 1; IM_WIDTH; x++
5.       if (y==1) && (x==1) then
6.         Initialise_Weights(*W)
7.       end
8.       LS=Calc_Local_Sum(*IN)
9.       *LD=Calc_Local_Differences(LS, *IN)
10.      PRED_LD=Dot_Product(*W, *LD)
11.      PRED=Calc_Prediction(PRED_LD, LS)
12.      Update_Weights(*W, *LD, PRED, *IN)
13.      RES(x,y)=Map_Residual(PRED, *IN)
14.      CWRD(x,y), LEN(x,y)=Calc_E_Prm(RES(x,y))
15.     end
16.   end

17. Threads = IM_BANDS(log(IM_BANDS))
18. Kernel INCLUSIVE_SUM(*LEN)
19.   *LEN=Inclusive_Sum(*LEN)

20. Threads = IM_BANDS x IM_HEIGHT x IM_WIDTH
21. Kernel BIT_PACKER(*RES, *EN_PRMS)
22.   *OUT=Entropy_Encode(*RES, *EN_PRMS)

```

As shown in Fig. 2, there are no spectral dependencies throughout all stages of the algorithm, this gives one consistent potential axis of parallelism for all these functional blocks, as shown by lines 1 – 16 of the CCSDS-123 kernel

Fig. 2. A block diagram of the CCSDS-123 algorithm which highlights the differing data dependencies of the constituent functional blocks. Those blocks that are shown to have spatial data dependencies will consequently form the critical path for parallel implementation.

in Algorithm 2. Intermediate results are now passed between functional blocks using fast on-chip memories. Thread local integer values, such as local sums (line 8), can be stored using registers. Additionally, user managed shared memory can be used to store shared array values, such as local difference and weight vectors (lines 9 and 12). Not only is shared memory lower latency than global memory but it also provides opportunity for greater data re-use and collaboration between threads. The CCSDS-123 kernel takes the input image pixel information, adheres to the CCSDS-123 standard compressions scheme and outputs the final compressed codeword in integer format and the length of the binary codeword. This information then needs to be combined to create the final compressed binary bitstream. This is achieved by first performing an inclusive sum on the codeword length to obtain the offset locations for each binary codeword in the final compressed bit stream (lines 17 – 19). This is then passed to a fully parallelised bit packer kernel which is responsible for converting the integer codewords to a binary bitstream (lines 20 – 22).

3.3 GPU Implementation Performance Results

This section details the performance testing results of our CUDA CCSDS-123 application. The application testing was conducted on non-tiled and tiled versions of two images, one multispectral (“Landsat Agriculture”) and one hyperspectral (“AVIRIS Hawaii”). Both of these data sets are available online via the CCSDS image corpus [17]. Details on the images, tiling and compression parameters used for these performance testing results are given in Appendix A, and are the same as used in [3].

Data processing throughput has also been measured for two different GPUs, a traditional desktop grade NVIDIA GPU, the GTX 750 Ti and the low power embedded Jetson TX1 platform. The Jetson TX1 is based on the NVIDIA Tegra X1 system-on-chip (SoC) device which features four ARM Cortex-A57 CPU cores and four ARM Cortex-A53 CPU cores in addition to the NVIDIA GPU silicon. Both the GTX 750 Ti and Jetson TX1 GPUs are based on the same NVIDIA Maxwell generation architecture, however a number of characteristics differ between the two platforms; most importantly in this work the number of GPU computing cores and the thermal design power (TDP). The GTX 750 Ti features 640 cores and a TDP of 60W, whilst the Jetson TX1 has 256 cores and a maximum TDP of 15W and 10W nominal TDP. Due to its lower power consumption, the Jetson TX1 can be considered a suitable reference platform for space applications and therefore included in this research to highlight the computing performance differences between the different GPU platforms.

Table 1 provides the average data processing throughput results for our GPU application for original and tiled versions of both test data sets and both GPU platforms, in both Mb/s and MS/s. These throughput results include all kernel executions, memory allocations, freeing and data transfers. To clearly illustrate the difference in data processing throughput the results in Mb/s are also graphed and given Fig. 3.

TABLE 1
GPU ACCELERATED CCSDS-123 DATA PROCESSING
THROUGHPUT (TP) PERFORMANCE

Image	Config.	GPU	TP (Mb/s)	TP (MS/s)
AVIRIS Hawaii	Original	GTX 750 Ti	869	72.4
		Jetson TX1	588	49.0
	Tiled	GTX 750 Ti	4818	401.5
		Jetson TX1	1394	116.2
Landsat Agriculture	Original	GTX 750 Ti	23	2.9
		Jetson TX1	14	1.8
	Tiled	GTX 750 Ti	1271	158.9
		Jetson TX1	426	53.3
	Tiled & MS Optimised	GTX 750 Ti	3061	382.6
		Jetson TX1	994	124.3

Fig. 3. GPU accelerated CCSDS-123 lossless image compression application data processing throughput comparison in Mb/s between different image test case and two GPU platforms.

The data processing throughput results highlight the inherent difference between multispectral and hyperspectral images. Hyperspectral images feature a large number of spectral bands, in the order of several hundreds, whilst multispectral images are composed of a much smaller number of bands, typically up to 10. In our CCSDS-123 application, parallelism is firstly exploited by defining a number of threads equal to the number of bands. Therefore, reducing the inherent number of bands has the effect of reducing the achievable data processing performance of the application, making high throughput compression of multispectral images a challenge. This is demonstrated by the comparison of results for original “AVIRIS Hawaii” and “Landsat Agriculture” images, where for both platforms there is significant slowdown of approximately 38 times. However, this is reduced to 3.7 times when additional parallelism is exposed through tiling and exploited by instantiating increased number of concurrent thread

blocks. To further increase the data processing throughput of multispectral imagery, we have developed an additional multispectral (MS) optimised version of our application. In this version, the main kernel is modified so that the additional parallelism induced by image tiling can be exploited at the thread level to increase the warp execution efficiency. This approach is discussed in detailed in [3]. The effective increase in data processing throughput is also detailed in Table 1 and Fig. 3, where the slowdown between hyperspectral and multispectral imagery has been reduced to only 1.7 times.

Secondly, from Table 1 we can see the increase in data processing throughput that is achieved by exploiting the additional parallelism induced by performing image tiling. Image tiling alone increased the throughput by approximately 55 times for the Landsat Agriculture image and 5.5 times for “AVIRIS Hawaii” image, on the desktop GPU. However, this increase in performance was not consistent among the two GPU devices, where an increase of only 19 and 1.8 times was observed for the embedded GPU, for “Landsat Agriculture” and “AVIRIS Hawaii” images respectively. This reduction is expected; whilst the Jetson TX1 is based on the same generation architecture as the GTX 750 Ti, it features 3 less SM’s and therefore 384 less CUDA cores. This indicates that the increased parallelism made available via image tiling has saturated the hardware resources of the Jetson TX1 GPU.

Additionally, we can compare the results given in Table 1 with performance testing results for previous FPGA and GPU implementations of the CCSDS-123 algorithm, as they utilise the same “AVIRIS Hawaii” test image. The previous FPGA implementation achieved a data processing throughput of 58 MS/s (MSamples/s) [15] and the GPU achieved 321.91 MS/s [16]. Comparing these performances with our own given in Table 1, our implementation is at least competitive with the prior state-of-the-art GPU implementation. It is however difficult to make a direct comparison as a different generation GPU has been used in [16]. Additionally, the authors did not publish the compression parameters used for testing.

When comparing the data processing throughput of our application with the FPGA implementation, we consistently exceed previous throughput achieved with a speedup of 6.9 and 1.3 for GTX 750 Ti and Jetson TX1 hardware respectively. Considering that the CCSDS-123 algorithm was specifically designed for implementation on FPGA hardware, this is a significant result. Showing that despite not being designed to provide high level of parallelism suited for GPU hardware, greater data processing throughput can be achieved, even when performed on the embedded low power Jetson TX1. This ultimately demonstrates that GPU based onboard data processing presents a suitable alternative to an FPGA based system, in terms of data processing throughput.

4 ERROR RESILIENT PROCESSING ON GPUS

One of the most common and comprehensive methodologies for characterising the error resilience of a system is to perform beam testing. Beam testing uses a radiation source to radiate a physical device or system under test. This aims to represent a realistic demonstration of the error resilience of the system in a certain radiation environment. The probability that a gate-level error in hardware propagates to the software application and its output can thus be quantitatively measured and this is often referred to as the architectural vulnerability factor (AVF) [10]. However, radiation rates are difficult to control and experiments can be extremely costly and time consuming to conduct. For practicality, often the error resilience of the hardware and software are evaluated separately [18].

Software based error injection provides a cheap and time efficient framework to replicate the effects of physical hardware faults, by intentionally altering instructions in a controlled manner, to assess the error resilience of a software application. Software based error injection is a useful technique to initially classify the error resilience of a software application. The results of which can help with the selection and development of efficient error mitigation techniques and a more error resilient application. The limitation of software based error injection is that it is only able to manipulate architecturally visible states and software accessible blocks. This is important to note as recent beam testing research has shown that GPU hardware blocks, such as the schedulers, can have a significant influence on the error resilience of an application, however these blocks cannot be tested using software based error injection [19].

In this section, we present and analyse the results from our software based error resilience study of our previously described CCSDS-123 GPU application. The software based error injection testing has been conducted to allow us to simulate how our software application behaves when an error occurs, causing values in instructions and memory to be altered. The errors induced are representative of the behaviours of a GPU due to SEE induced by the space environment. The results from the error injection experiments help increase our understanding of the error resilience of our application, in terms of SDC and FI errors. Using these results, we then describe developments made towards two new error resilient versions of the CCSDS-123 application. Software based error injection testing is then repeated on the new applications. The results are then assessed to quantify the improvements made in error resilience. Additionally, the associated computational processing throughput and memory usage overheads are measured and presented.

4.1 GPU Application Error Resilience Testing

In this research, we utilise the SASSIFI framework to perform software based error injections. SASSIFI is a software framework developed by NVIDIA for GPU applications [20]. It allows users to practically assess the impact of errors on GPU applications. SASSIFI is based on the SASSI GPU assembly language instrumentation tool also devel-

oped by the NVIDIA Architecture Research Group [21]. Although not an official part of the CUDA software toolkit SASSI and SASSIFI are research prototypes which provide a selective instrumentation framework for NVIDIA GPU applications. The framework allows instrumentation to be inserted into the NVIDIA native Instruction Set Architecture (ISA) known as SASS. The advantage of this framework is its open availability, via GitHub [22], proven functionality, detailed documentation, flexible and wide coverage error model and low execution overhead. SASSIFI provides three modes of operation, which inject errors into the instruction output value (IOV), instruction output address (IOA) and register file (RF). Together, the three injection modes, allow us to assess the error resilience, in terms of SDC, FI and masked error rates, of the register file memory and different instruction types. Consequently, this enables us to identify high-level trends in error resilience between different kernels or different run-time and compile-time configurations.

For IOV and IOA modes we inject single bit errors into a single thread per warp for each possible instruction group and for the RF mode we injected a single bit flip error for a single register. For each injection type, at least 370 injections were performed per test. Using profiling results to determine the total number of instructions in our application we calculated that this allows us to achieve a 95% confidence level with a confidence interval of 5%.

A) Kernel Error Resilience Comparison

There are two user kernels in our application, CCSDS-123 and Bit Packer. The SASSIFI framework allows us to evaluate the impact of error injections for each kernel independently. This kernel-level interpretation of the results allows us to assess which of the kernels has the greatest contribution towards the overall error resilience of the application. This understanding helps select efficient and effective error mitigation techniques for each kernel and the overall application whilst minimising the introduced overhead.

An overview of the error injection results for the three injection modes (IOV, IOA and RF), for two image compression test cases (Agri, Hawaii), are given in Fig. 4. For each test case in Fig. 4, each stacked bar represents the proportion of SDC, FI and masked error effects as a percentage of the total injected errors for the two major kernels in the GPU application (Bit Packer and CCSDS-123). RF mode is inherently able to achieve 100% error injection coverage of all registers used in the application. However, for IOV test mode a small proportion of miscellaneous instructions were not covered by the error injections to minimise experiment execution time. For IOA conditional code and predicate instructions are also not covered as these instructions do not have address components, therefore this error mode is irrelevant for these instructions. For clarity, using profiling results the proportion of instructions not covered by these IOV and IOA test modes are also shown in Fig. 4 as uncharacterised bars.

Fig. 4. Error probabilities for each kernel as a percentage of our application for the three injections modes.

The results shown in Fig. 4 highlight the significant influence the CCSDS-123 kernel has on the overall application error resilience, compared to the Bit Packer kernel, for all injection modes. This is an expected trend, as it reflects the respective differences between the algorithmic and computational workloads of the individual kernels; where the CCSDS-123 kernel has almost 10 times the computational workload compared to the Bit Packer kernel. This is an important application characteristic to identify when looking towards the selection of efficient error mitigation schemes. Implementing appropriate error mitigation techniques for the Bit Packer kernel can only increase the overall application error resilience by less than 10%. Whilst, mitigating both SDC and FI errors in the CCSDS-123 kernel can improve the application error resilience by up to 60%. Therefore, these results suggest that a targeted approach to firstly protect the CCSDS-123 kernel will have a significant impact on improving the error resilience of the whole application.

The results given in Fig. 4 also show variation in error probabilities with error injection mode and hence error location. This is reflected by the results for the different injection modes. Therefore, the following sections will discuss the results for each error injection mode in greater detail.

B) Register File Error Resilience

Using the RF specific error injection mode, we can classify the error resilience of the GPU register file memory structure. Fig. 5 details the error injections results for the register file mode for each kernel (Bit Packer, CCSDS-123) and two image test cases (Hawaii, Agri).

Fig. 5. Results from error injection test of the GPU register file (RF Mode), showing percentages of SDC, FI and masks for each kernel of the GPU CCSDS-123 application for two test images; AVIRIS Hawaii and Landsat Agriculture.

These results show there is a relatively small probability that a single bit-flip in an allocated register results in an SDC, only between 15-20%. Meanwhile, between 40-50%, of errors result in a FI of the kernel execution. These results demonstrate the importance of developing and introducing appropriate mitigating techniques to deal with FI error effects in addition to data corruption effects.

As RF error injections are performed randomly with respect to both run-time and physical location for allocated registers in each kernel, we can utilise the knowledge of the amount of allocated registers and the size of the register file, to calculate AVFs. To calculate the SDC and FI AVF for both kernels we derate the error probabilities, shown in Fig. 5, with the average fraction of occupied registers for each kernel. The formulas used are given in equations 1 and 2. The resulting SDC and FI AVF for both kernels are given in Table 2. Due to the variation in application throughput and therefore run-time for the different test cases, the resulting error rate simulated by our error injections is between 5.7 and 62.5 errors per second, for Agriculture and Hawaii test images respectively.

$$\text{Average \# of allocated registers} = \text{Registers Usage Per Thread} * \text{Threads Per SM} \quad (1)$$

$$\text{Average fraction of allocated registers} = \frac{\text{Average \# of Allocated Registers}}{\text{Registers per SM}} \quad (2)$$

TABLE 2
KERNEL REGISTER USAGE AND AVF's

Kernel	Register Usage Per Thread	SDC AVF	FI AVF
CCSDS-123	72	0.31 %	6.12 %
Bit Packer	19	0.06 %	0.19 %

These AVF results highlight the overall very low vulnerability factor of the Bit Packer kernel with regards to both SDC and FI effects, this is inherently due to the proportionally low register usage. The low SDC AVF for both kernels indicate that the addition of extensive error correction codes (ECC) to the register file will likely not achieve a good trade-off between error mitigation impact and processing throughput overhead cost. Due to the low probability of SDC effects occurring, low overhead detection and

re-computation based techniques may provide a better trade-off between error resilience and execution time overhead. However, the FI AVF for CCSDS-123, whilst below 10%, poses the greatest error risk for the register file in our application. Therefore, research into low overhead FI protection mechanisms for memory structures is recommended for future work.

C) Instruction Error Resilience

Using the SASSIFI framework we are also able to analyse the error resilience of different instruction types within our algorithm. This can provide useful information for the later development of targeted or compiler level error mitigation techniques. To interpret how error resilience differs between instruction types and how they contribute to the overall error resilience of our application, we performed instruction level profiling. This provides in-depth information on instruction occurrence rates and allows us to normalise the error injection results against their occurrence for accurate interpretation of the error injection results. Firstly, we have classified and assessed instructions in our application based on where they output data to; namely general purpose registers (GPR), predicate registers (PR), conditional code (CC), or store instruction which output data to memory structures such as shared memory and global memory. The composition of our application with respect to these instruction groupings are given in Table 3.

TABLE 3
INSTRUCTION TYPE OCCURRENCE FOR GPU CCSDS-123
APPLICATION BASED ON OUTPUT LOCATION

Instruction Type	Occurrence (%)
General Purpose Registers (GPR)	86.4
Predicate Register (PR)	6.9
Conditional Code (CC)	4.0
Memory Store (ST)	2.6

Table 3 shows that the majority of instructions in our application output data to GPR instructions. Therefore, we expect the error resilience of these instructions will have the greatest influence on the overall error resilience of our application. To assess this hypothesis, the error injection results for the instruction groupings detailed in Table 3 are given in Fig. 6. The bars in Fig. 6 represent the percentage of SDC, FI and masked errors weighted by the instruction group occurrence rate detailed in Table 3. The different percentages between the two kernels (Bit Packer and CCSDS-123) are also represented as per the key in Fig. 6. For IOA mode it is only possible and relevant to inject an error into GPR and ST instructions.

Fig. 6. Results of error injection into output value and address components (IOV and IOA modes). These results are weighted against the instruction type occurrence ratios shown in Table 3.

As predicted, based on the occurrence rates of the instructions, Fig. 6 shows that GPR instructions clearly have the greatest influence, in terms of both SDC and FI rates, on the whole application. Whilst, store instruction account for the smallest percentage of instructions (Table 3), they have a significant influence on error resilience compared to more common PR or CC instructions. We postulate that this is likely due to the impact of algorithmic characteristics on error resilience. In our application, we only utilise non-register memory for the storage of shared intermediate result vectors and for the storage of the final results of each kernel. Therefore, there will be very little algorithmic masking effects occurring for these operations. These results highlight the importance of protecting memory instructions, which whilst having the lowest occurrence rate, appear to have significantly reduced error resilience, when compared to other instruction types.

The open source nature of the SASSIFI framework also allows users to define their own custom instruction grouping for low-level instruction error resilience analysis. Using this functionality, we have assessed the error resilience of 91% of instructions in our application. The major instruction groups in our application are global and shared memory load instructions (LD, LDS), integer addition and multiplication (IADD_IMUL & MAD) or logical operations (SHUFF_LOP). The specific instruction occurrence rates for these instruction groups are given in Table 4.

The corresponding error injection results for these instruction groupings are given in Fig. 7. In Fig. 7 the bars

represent the percentage of SDC, FI and masked errors weighted against the instruction occurrence rates of each instruction group as per Table 4.

The results given in Fig. 7 highlight that the three computational instruction groups (IADD_IMUL, MAD and SHUFF_LOP) have a large contribution to the overall application SDC vulnerability. By potentially protecting these three instruction groups (combined) we could reduce the SDC probability by approximately 42%. The results also

TABLE 4
INSTRUCTION GROUPING DEFINITION & OCCURRENCE FOR GPU CCSDS-123 APPLICATION

Group Name	Opcodes	Occurrence (%)
IADD_IMUL	IMNMX, ISCADD, IADD, IADD3, IADD32I	26.4
MAD	XMAD	22.7
SHUFF_LOP	SHE, SHL, SHR, LOP	12.2
LD	LD, LDC, LDG, LDL	4.8
LDS	LDS	3.7

Fig. 7. Results of error injection tests into different instruction types for GPR instructions in IOV mode. These results are weighted against the occurrence ratios of the instructions groups as shown in Table 4.

show that despite the relatively low occurrence rate of the LD and LDS instructions, shown in Table 4, their error resilience is relatively low. This mirrors the results we also observed with store operations assessed in Fig. 6.

Whilst we do not currently have the capability to target the protection of specific instructions, these results highlight the potential opportunities and advantages of developing a compiler based solution for error resilient application development. A compiler based solution that could target specific instructions and types could assess and achieve a suitable trade-off between error resilience and induced overhead, in terms of both data processing throughput and memory usage.

4.2 Error Resilient GPU Application Development

As an initial step towards the research and development of more error resilient GPU software applications, we have developed two new versions of our application which aim to mitigate SDC effects. The modified applications leverage TMR principles to allow for the detection and correction of SDC errors. We have chosen to implement a targeted scheme which solely protects the CCSDS-123 kernel in our applications. This is because the previous error resilience results have shown that this kernel has the largest influence over the overall error resilience of our application.

TMR can be implemented at several different architectural levels, in this work we have investigated two implementations for the TMR protection at a kernel and thread level, referred to hence forth as K-TMR and T-TMR. Both approaches execute the complete instruction pipeline of the CCSDS-123 kernel three times, resulting in three copies of output data being generated. A new TMR comparator kernel has also been developed which compares the three copies of output data to detect errors and using a majority voter also corrects data discrepancies. The difference between the K-TMR and T-TMR versions of our application is how the instruction pipeline is triplicated. Both implementations aim to utilise the parallel nature of the GPU and characteristics of the CCSDS-123 kernel to minimise the overhead induced by the additional workload. In the K-TMR version, we triplicate the kernel execution and utilise the CUDA streams construct to allow for concurrent kernel execution. For the T-TMR version, we modified the original kernel so that upon declaring triple the number of threads each instruction is executed three times and subsequently output data is stored in separate memory locations for later comparison. When the CUDA cores of the GPU are not fully utilised, this version exploits underutilisation to hide the execution overhead.

4.3 TMR GPU Application Error Resilience Testing

To quantify changes in error resilience of our modified applications, we repeated the SASSIFI based error injection testing. We utilised the same testing parameters so we can directly compare the error resilience of the two new applications against each other and the original unprotected application. A top-level view of the results for IOV, IOA and RF injection modes are given in Fig. 8. The bars in Fig. 8 represent the percentage of SDC, SDC corrected, FI and masked errors which occurred for the original (No TMR), K-TMR and T-TMR GPU applications.

The results in Fig. 8 show the significant reduction in SDC error effects achieved by both K-TMR and T-TMR versions. For IOV and IOA modes SDC's were reduced by at least 40% achieving SDC probabilities below 2%. For the register file injection mode, the SDC outcome was reduced by over 15% achieving an SDC probability of 0.5% or below. We can also see from Fig. 8 how the overall FI rate has been affected by our TMR implementations. For both IOV and IOA modes the FI rate remains statistically similar with no significant change to the observed FI rates.

Fig. 8. Comparative error injection results for original unprotected application, K-TMR and T-TMR.

To gain a deeper insight into the impact of SDC errors on the modified application version, Fig. 9 shows the same experimental results as Fig. 8, but with the error probabilities broken down at the kernel level. The bars in Fig. 9 represent the percentage of SDC, corrected SDC, FI and masked errors which occur for each kernel of the application which includes the new TMR Compare kernel for the K-TMR and T-TMR applications.

Fig. 9. Comparative error injection results for kernels of non-protected application, K-TMR and T-TMR.

The kernel analysis, shown in Fig. 9, shows that the K-TMR version eliminated all SDC errors in the protected CCSDS-123 kernel and the only remaining SDC's occurred in the new comparator or unprotected Bit Packer kernels. However, the T-TMR version did not eliminate all SDC's, a very small proportion, 0.5%, still occurred in the CCSDS-123 Kernel. We propose that this likely due to the corruption of shared registers and resources. In this initial implementation, all three duplications of the instruction pipeline are executed within the same thread block. Therefore, errors in shared resources may propagate to all three of the TMR instructions and will not be detected, or corrected.

4.4 TMR GPU Application Performance Testing

In addition to assessing the error resilience of the K-TMR and T-TMR applications we have also conducted performance testing and analysis of the memory usage of these applications to establish the overhead requirements induced. These results are given in Table 5 which shows the execution time, execution overhead and memory requirements for the original and modified applications under several kernel configurations for the Landsat Agriculture test image. The different kernel configurations tested (A-D) represent the same degree of parallelism but increasing levels of TLP for the CCSDS-123 kernel; as the number of threads is increased and the number of blocks declared decreases. The execution overhead for the TMR implementations are calculated in comparison with the equivalent configuration for the original (no TMR) version.

TABLE 5
TMR TECHNIQUE OVERHEAD COMPARISON

TMR Mode	Config. [# Kernels, # Blocks, # Threads]	Execution Time			
		GTX 750 TI		Jetson TX1	
		ms	Overhead	ms	Overhead
No TMR	A= [1,1024,6]	61	-	118	-
	B= [1,512,12]	36	-	73	-
	C= [1,256,24]	24	-	51	-
	D= [1,128,48]	23	-	51	-
K-TMR	A= [3,1024,6]	145	138 %	314	166 %
	B= [3,512,12]	88	144 %	185	152 %
	C= [3,256,24]	54	130 %	118	131 %
	D= [3,128,48]	53	130 %	120	135 %
T-TMR	A= [1,1024,18]	68.5	12 %	147	24 %
	B= [1,512,36]	66.5	85 %	146	99 %
	C= [1,256,72]	54.5	132 %	126	149 %
	D= [1,128,144]	52	126 %	116	126 %

$$\text{Shared Memory (Bytes)} = 8 * \# \text{Bands} * \text{Wlen} * (\# \text{Threads} / \# \text{Bands}) \quad (3)$$

$$\text{Global Memory (Bytes)} \begin{cases} \text{If TMR} \approx 38 * \# \text{Pixels} \\ \text{Else} \approx 14 * \# \text{Pixels} \end{cases} \quad (4)$$

The execution time and overhead results given in Table 5 show that for the original application as the TLP increases, execution times decreases with diminishing returns. This is also mirrored by the K-TMR version of our application, where the overhead of the K-TMR application remains relatively constant across all kernel configurations, with a slight decrease in overhead with increasing level of TLP exploited. However, this trend is not observed by the T-TMR implementation, which has a significantly increasing overhead with increasing TLP. We attribute this to the relationship between TLP and warp execution efficiency. For T-TMR configuration A, representing the lowest level of TLP, an almost negligible overhead is introduced. For this case, the warp execution efficiency is increased without increasing the number of required warps and the additional GPU workload is almost completely hidden. However, as we continue to increase the explicit TLP of the kernel (configurations B – D), additional warps are required to implement the TMR strategy and a larger overhead is induced.

When we consider the configuration, which achieves the minimised execution time, both K-TMR and T-TMR implementations incur a significant but similar overhead of 130% and 126% respectively. This shows that when the explicit application configuration is highly optimised for low execution time there is less opportunity to effectively hide the execution overhead.

Equations 3 and 4 provide the relationships for shared and global memory. Shared memory is closely related to the number of threads initialised per kernel. Due to this relationship, the shared memory usage overhead is increased for the T-TMR implementation when compared to the K-TMR and original applications. The global memory overhead introduced for K-TMR and T-TMR applications are the same and approximately equated to an overhead of 170%. We reduce the global memory overhead introduced by only applying the TMR techniques to the CCSDS-123 kernel rather than the whole application.

5 CONCLUSION

This paper has presented new contributions to help improve the technology readiness of GPU based computing for space applications. The paper is based on the recent research which resulted in the development of our own GPU implementation of the space application representative CCSDS-123 image compression algorithm. The application was originally created to demonstrate the data processing and compression capabilities of a new GPU based onboard data processing system for EO satellites. Data processing throughput results are given for the desktop GTX 750 Ti and low power embedded Jetson TX1. The Jetson TX1 has been tested as a representative platform for onboard suitable GPU based computing. These results highlight the inherent differences in data processing throughput between the GTX 750 Ti and the low power Jetson TX1, with a reduction in throughput of up to 3 times between the platforms. However, despite these differences the Jetson TX1 GPU application exceeded the processing throughput achieved by a previous space representative FPGA solution [15].

The major focus and contributions of this paper relate to the new work presented in the area of GPU and parallel application error resilience. We utilised the recently released NVIDIA SASSIFI software based error injection framework, to quantitatively measure the error resilience of our image compression GPU application. Using these results, we analysed the error resilience of the register file memory and different GPU instructions groups for our application. We found that both SDC and FI AVFs for the register file were relatively low, below 1% for SDC effects and approximately 6% for FI in the worst case. However, instruction level error injections resulted in much higher SDC and FI error probabilities, between 40 -50% and 10-20% respectively. These results indicate that the implementation of ECC will have little influence in reducing the overall SDC rate of our application. However, due to the large impact of instruction error resilience an instruction based protection scheme will likely be much more suitable. Kernel level analysis has also highlighted that a single kernel in the multi-kernel application had a major influence on the overall error resilience of the entire application. Therefore, the application is well suited to a targeted protection scheme.

To enable further advancements towards the practical demonstration of a GPU based computing platform in a space environment, we have developed two error resilient versions of our GPU accelerated image compression application. We have initially explored the use of targeted TMR schemes and developed two implementations which aim to exploit different architectural and software model properties of the GPU to minimise the execution and memory overheads introduced. Specifically, our K-TMR application exploits CUDA streams to concurrently execute multiple kernels and T-TMR utilises underutilisation of TLP, where applicable, to hide the additional execution overheads of TMR. To test and prove the error correcting capabilities of our newly developed TMR based applications we conducted software based error injection testing. These results showed that by deploying TMR on just a single kernel in our application we reduced the SDC effects by at least 40% for instruction induced errors and to less than 0.5% for the register file. However, the TMR protection scheme does not protect against FI effects. Therefore, the development of additional protection techniques to reduce or mitigate FI effects is necessary before the application can be adopted as part of a practical system for onboard EO applications.

In addition, we provided quantitative results on the execution time and memory overheads introduced by our TMR applications and their relationships with exposed TLP. This analysis shows that for low TLP, T-TMR is effectively able to hide the increased computational load of TMR achieving a low execution time overhead of just 12%. However, for high degrees of TLP, execution overhead is more difficult to hide and both T-TMR and K-TMR applications introduce similar execution overheads of between 126 and 130% respectively.

6 FUTURE RESEARCH

We hope to further expand on work outlined in this paper by performing additional development to ensure duplicated instructions in the T-TMR application are executed in separate physical hardware, so all SDCs can be mitigated. Secondly, develop appropriate techniques to mitigate or reduce the impact of FI error effects.

Furthermore, whilst software based error injection is a relatively cheap and quick methodology for assessing error resilience it cannot completely characterise an applications behaviour in an error prone environment. One particular factor which cannot be assessed through software based error injection is the impact of computing workload and GPU configurations on the scheduler and how this subsequently impacts error resilience. Whilst it is well documented how kernel execution parameters impact data processing throughput, how they impact error resilience is not well understood. Recent research conducted in [23], has shown, through experimental beam testing, there are relationship between TLP, scheduler workload and error resilience on GPUs. It would be valuable to compare and assess our application using beam testing to firstly validate the results obtained via software based error injection and also to determine the impact of GPU schedulers on application error resilience.

ACRONYMS

EO	– Earth Observation
CCSDS	– Consultative Committee on Space Data Systems
SEE	– Single Event Effects
TID	– Total Ionising Dose
FPGA	– Field Programmable Gate Array
SBFT	– Software Based Fault Tolerance
CUDA	– Compute Unified Device Architecture
SM	– Streaming Multiprocessor
SFU	– Special Function Unit
DRAM	– Dynamic Random Access Memory
FI	– Functional Interrupt
SDC	– Silent Data Corruption
AVF	– Application Vulnerability Factor
COTS	– Commercial Off The Shelf
ABFT	– Algorithm Based Fault Tolerance
EDAC	– Error Detection And Correction
DMR	– Double Modular Redundancy
TMR	– Triple Modular Redundancy
NASA	– National Aeronautics and Space Administration
JPL	– Jet Propulsion Laboratory
TLP	– Thread Level Parallelism
SoC	– System on Chip
TDP	– Thermal Design Power
IOV	– Instruction Output Value
IOA	– Instruction Output Address
RF	– Register File
ECC	– Error Correction Code
GPR	– General Purpose Registers
PR	– Predicate Registers
CC	– Conditional Code
ST	– Store instructions

REFERENCES

- [1] R. L. Davidson, C. P. Bridges, "Adaptive Multispectral GPU Accelerated Architecture for Earth Observation Satellites", Proc. of 2016 IEEE Int. Conf. on Imaging Systems and Techniques.
- [2] Consultative Committee for Space Data Systems, Lossless Multispectral & Hyperspectral Image Compression CCSDS 123.0-B-1, Blue Book, [Online] <http://public.ccsds.org/publications/archive/123x0b1ec1.pdf> (Accessed September 2017)
- [3] R. L. Davidson and C. P. Bridges. "GPU Accelerated Multispectral EO Imagery Optimised CCSDS-123 Lossless Compression Implementation." 2017 IEEE Aerospace Conference Proceedings.
- [4] E. G. Stassinopoulos, J. P. Raymond, "The space radiation environment for electronics", Proc. IEEE, vol. 76, no. 11, pp. 1423-1442, Nov. 1988.
- [5] S. Lopez, T. Vladimirova, C. Gonzalez, J. Resano, D. Mozos and A. Plaza, "The Promise of Reconfigurable Computing for Hyperspectral Imaging Onboard Systems: A Review and Trends," In Proceedings of the IEEE, vol. 101, no. 3, pp. 698-722, March 2013. DOI: 10.1109/JPROC.2012.2231391
- [6] NVIDIA GeForce GTX 750 Ti Whitepaper V1.1. [Online] <https://international.download.nvidia.com/geforce-com/international/pdfs/GeForce-GTX-750-Ti-Whitepaper.pdf> (Accessed January 2018)
- [7] NVIDIA Jetson TX1 Information [Online] <https://developer.nvidia.com/embedded/buy/jetson-tx1-devkit> (Accessed January 2018)
- [8] NVIDIA CUDA Toolkit Documentation V7.0. [Online] http://developer.download.nvidia.com/compute/cuda/7_0/Prod/doc/CUDA_Toolkit_Release_Notes.pdf. (Accessed January 2018)
- [9] Kastensmidt, Fernanda, and Paolo Rech. "FPGAs and parallel architectures for aerospace applications: soft errors and fault-tolerant design". Springer, 2015.
- [10] S. S. Mukherjee, C. T. Weaver, J. Emer, S. K. Reinhardt, T. Austin, "A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor", Proc. 36th Annu. IEEE/ACM Int. Symp. Microarchitecture, pp. 29-42 December 2003.
- [11] P. Rech, C. Aguiar, C. Frost, L. Carro, "An Efficient and Experimentally Tuned Software-Based Hardening Strategy for Matrix Multiplication on GPUs", Nuclear Science IEEE Transactions on, vol. 60, no. 4, pp. 2797-2804, 2013.
- [12] Y. Bentoutou, "A real time EDAC system for applications onboard earth observation small satellites", IEEE Trans. Aerosp. Electron. Syst., vol. 48, no. 1, pp. 648-657, Jan. 2012.
- [13] I. P. Egwuotuoha, D. Levy, B. Selic, and S. Chen, "A survey of fault tolerance mechanisms and checkpoint/restart implementations for high performance computing systems," The Journal of Supercomputing (2013): 1-25, 2013.
- [14] P. K. Samudrala, J. Ramos, S. Katkooi, "Selective Triple Modular Redundancy (STMR) Based Single-Event Upset (SEU) Tolerant Synthesis for FPGAs", IEEE Transactions on Nuclear Science, Vol. 51, No. 5, pp. 2957-2969, Oct. 2004.
- [15] N. Aranki, D. Keymeulen, A. Bakhshi, and M. Klimesh, "Hardware implementation of lossless adaptive and scalable hyperspectral data compression for space," in Adaptive Hardware and Systems, 2009. AHS 2009. NASA/ESA Conference on, August 2009, pp. 315-322.
- [16] B. Hopson, K. Benkrid, D. Keymeulen, N. Aranki, "Real-time CCSDS lossless adaptive hyperspectral image compression on parallel GPGPU & multicore processor systems", Proc. 2012 NASA/ESA Conf. Adaptive Hardware and Systems (AHS), pp. 107-114, 2012-Jun.
- [17] CCSDS Test Image Corpus [Online] <https://cwe.ccsds.org/sls/docs/SLS-DC/123.0-B-Info/Test-Data> (Accessed January 2018)
- [18] L. A. B. Gomez, F. Cappello, L. Carro, N. DeBardeleben, B. Fang, S. Gurumurthi, S. Keckler, K. Pattabiraman, R. Rech, M. S. Reorda, "Gpgpus: How to combine high computational power with high reliability", 2014 Design Automation and Test in Europe Conference and Exhibition, 2014.
- [19] P. Rech, C. Aguiar, T. Fairbanks, H. Quinn, L. Carro, "Threads distribution Effects on Graphics Processing Units Neutron Sensitivity", IEEE Transactions on Nuclear Science, vol. 60, no. 6, pp. 4220-4225, 2013.
- [20] S. Hari, T. Tsai, M. Stephenson, S. Keckler, J. Emer, "SASSIFI: An architecture-level fault injection tool for GPU application resilience evaluation", Performance Analysis of Systems and Software (ISPASS) 2017 IEEE International Symposium on, 2017.
- [21] M. Stephenson, S. K. S. Hari, Y. Lee, E. Ebrahimi, D. R. Johnson, D. Nellans, M. O'Connor, S. W. Keckler, "Flexible Software Profiling of GPU Architectures", Proc. of the Int'l Symp. on Computer Architecture (ISCA), 2015.
- [22] An architecture-level fault injection tool for GPU application resilience evaluations, NVlabs, [Online], <https://github.com/NVlabs/sassifi> (Accessed January 2018)
- [23] P. Rech, 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.

BIOGRAPHIES

Rebecca L. Davidson received a BEng in Electronic Engineering from the University of Surrey, in 2014. She is currently a PhD student at Surrey Space Centre, University of Surrey. Her current research is in onboard data processing of EO payload data for satellite downlink optimisation. Her current research focus is on efficient hardware-software co-design for space suitable high throughput and high compression ratio image processing using commercial-off-the-shelf GPU processors.

Dr Christopher P. Bridges (BEng, 2005; PhD 2009) leads the On-Board Data Handling (OBDH) research group within Surrey Space Centre (SSC). He researches software defined radios, real-time embedded systems, agent computing, Java processing, multi-core processing in FPGAs, and astrodynamics computing methods in many spaceflight payloads. In 2013, he designed, built and still operates the UK's first CubeSat (STRaND-1) with SSTL and now contributes towards computing hardware and software in missions with SSTL, on ESA's ESEO mission and also the NASA-JPL/CalTech AAReST mission.