An On-board Algorithm Implementation on an Embedded GPU: A Space Case Study

Iván Rodriguez*† Leonidas Kosmidis†* Olivier Notebaert‡ Francisco J. Cazorla† David Steenari§

*Universitat Politècnica de Catalunya
‡Airbus Defence and Space, Toulouse, France

†Barcelona Supercomputing Center (BSC), Spain
§European Space Agency, The Netherlands

***Abstract*—On-board processing requirements of future space missions are constantly increasing, calling for new hardware than the traditional ones used in space. Embedded GPUs are an attractive candidate offering both high performance capabilities and low power consumption, but there are no complex industrial case studies from the space domain demonstrating these advantages. In this paper we present the GPU parallelisation of an on-board algorithm, as well as its performance on a promising embedded GPU COTS platform targeting critical systems.**

***Index Terms*—embedded GPUs, aerospace, case study**

I. Introduction

The current trend in aerospace is the constant increase of performance requirements for on-board processing, to support the higher data rates and autonomy required across all space domains including science and robotic exploration, telecom and navigation [1]. Such performance cannot be provided by existing space CPUs such as the NGMP [2] or PowerPC 750 [3], so new technologies including COTS devices from other critical domains like automotive are currently explored.

Embedded GPUs are considered among the most promising enabling technologies, since they can offer both high performance software processing (as opposed to existing high-performance hardware processing provided by FPGAs for space) as well as low power consumption below 10W [4], which is the limit of feasible thermal dissipation of a single component in a space system. However, so far there is limited practical indication in industry on whether existing on-board software is a good fit for the massively parallel GPU programming model and more importantly whether embedded GPUs can provide the processing requirements of future missions. In fact, until now only the standalone lossless compression algorithm CCSDS-123 [5] has been demonstrated in a non-embedded GPU, as well as simple, non-space specific image processing algorithms from existing GPU libraries [6] on a custom designed embedded GPU platform for space. However, there is a lack of complex case studies in the literature.

We bridge this gap by presenting for the first time a demonstration of a GPU implementation of a complex on-board algorithm, the H2RG infrared detector [7] used in several existing and future space missions such as NASA's Hubble Space Telescope, its successor James Webb Space Telescope (JWST) and ESA's Euclid astronomy and astrophysics space mission [8]. We demonstrate not only the feasibility of porting such a case study on a GPU, but also its performance benefits by executing it on an embedded COTS GPU platform designed for the automotive domain, which has similar reliability requirements with low earth orbit (LEO) missions.

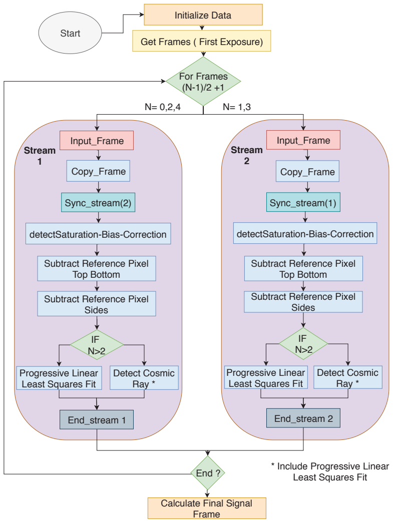
Start
Initialize Data
Get Frames (First Exposure)
For Frames (N-1)/2 +1
N= 0,2,4
N= 1,3
Stream 1
Input_Frame
Copy_Frame
Sync_stream(2)
detectSaturation-Bias-Correction
Subtract Reference Pixel Top Bottom
Subtract Reference Pixel Sides
IF N>2
Progressive Linear Least Squares Fit
Detect Cosmic Ray *
End_stream 1
Stream 2
Input_Frame
Copy_Frame
Sync_stream(1)
detectSaturation-Bias-Correction
Subtract Reference Pixel Top Bottom
Subtract Reference Pixel Sides
IF N>2
Progressive Linear Least Squares Fit
Detect Cosmic Ray *
End_stream 2
End ?
* Include Progressive Linear Least Squares Fit
Calculate Final Signal Frame

Fig. 1. High level view of the H2RG case study GPU implementation.

II. Application and GPU parallelisation

The baseline sequential version of the Near Infrared (NIR) H2RG BM case study is based on ESA's implementation for the Euclid mission. The application processes *(*N) raw images acquired from the HAWAII-2 sensor – an integrated circuit for visible and infrared astronomy applications – to produce a single output image. More details are provided in [7].

Although the algorithm operates over image matrices, the GPU parallelisation of the H2RG algorithm is not straightforward, since a significant part of the application requires accessing data with complex loop dependencies, complicating its GPU implementation. Despite this challenge, we managed to port the entire application to the GPU, so that the CPU is only responsible to offload computations to the GPU.

An instrumental step towards the preparation for the GPU implementation has been the code refactoring of the sequential version, in order to eliminate global and static variables from the application functions. This permits their porting for execution on the GPU becoming *kernels*, functions which cannot retain state between calls, unless they are passed as function arguments. This helped also to identify which variables had to be allocated in the GPU memory, since GPUs have their own address space which is distinct from the CPU one, despite the fact that in embedded GPUs such as ours, both devices share the same physical memory. Since our implementation targeted an NVIDIA embedded GPU as we explain in the next Section, we have performed our implementation in CUDA. Although

978-3-9819263-4-7/DATE20/©2020 EDAA

CUDA supports *Unified Memory*, a feature which allows to share memory between the CPU and GPU in order to simplify GPU programming at the expense of reduced execution control and therefore suboptimal performance, we have opted not to use this feature, in order be able to fully utilise our platform.

In Figure 1 we present a high level view of the parallelisation strategy we have followed. In order to optimise our implementation, we took advantage of parallelism in several levels: at frame level, application stage level and intra-kernel.

Each of the application stages are implemented as separate kernels, although in some cases such as in detectSaturation-Bias-Correction we have merged three stages in a single kernel in order to increase the number of cache hits due to data reuse. In the Subtract Reference Pixel stage, the opposite transformation was applied since it performed two independent operations which required synchronisation between them.

As shown in the figure, we process consecutive frames in different processing pipelines, which are known as *streams* in CUDA terminology. This allows a processing scheme in which the memory transfer of one frame is overlapped in time with the GPU execution of the application stages of the previous frame. Within each processing pipeline, the stages of the algorithm are executed one after the other. Moreover, when the data copy one of the frames has finished, its stages are also executed on the GPU, since the GPU is capable of executing multiple kernels from different streams concurrently, when both their computational resources can be satisfied. In addition, even when it is not possible to execute two kernels at the same time, the utilisation of the GPU is maximised, since there is always a delay between the CPU offloading of one kernel and its execution and termination, which could leave the GPU underutilised if only one stream was used.

Finally, we have optimised the execution time of each individual kernel, by unrolling their loops and when possible using shared memory to save bandwidth and allow sharing data between the threads of the kernel. The latter was fundamental to implement the most challenging of all kernels, Subtract Reference pixel Sides, by using the shared memory to replace the circular buffer implementation used in the CPU version. This way, with a careful selection of the number of threads working in a group, we were able to communicate between them to compute the mean values required in the kernel.

Our implementation is parametric w.r.t. the above described different parallelism factors, so the same code implementation can be configured to be executed efficiently on other embedded GPUs, too. The long process of parameter tuning has been performed by measuring the end-to-end execution time of the application as well as by using NVIDIA's provided tools, such as the command line profiler (*nvprof*) and the interactive performance guide included in NVIDIA's Visual Profiler (*nvvp*).

In our platform the optimal performance has been achieved by processing 2 frames in parallel. Further increasing the number of parallel frames only increased the memory consumption – since each image pipeline requires its own state for processing a frame such as the input image and its intermediate results – without performance gains or resulting in slowdown.

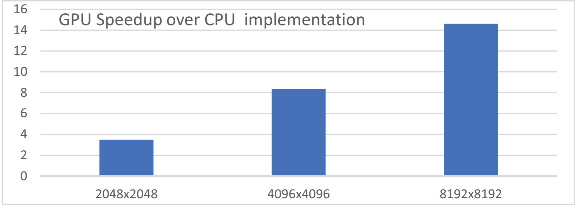

Fig. 2. H2RG GPU version speedup over the CPU on NVIDIA Xavier.

III. Results

We optimised our code on the NVIDIA's Xavier platform, designed to reach the highest certification level in the automotive domain, ASIL-D. As such, it includes reliability features such as ECC and as a consequence, it can be used for space missions up to LEO without any further radiation mitigation techniques. The platform includes ARMv8 Carmel CPUs and a powerful embedded GPU based on the Volta microarchitecture, which are clocked at 1.2 GHz and 520 MHz at its 10W TDP (thermal design power) performance mode.

Figure 2 presents the relative performance of the GPU implementation over the sequential version in the same platform, for various input sizes. The GPU implementation is 3.5× faster than CPU for the smallest size, and up to 15× for larger sizes similar to the requirements of future missions.

As an indication of comparison with existing space processors [7], the CPU implementation for the standard 2K×2K image size is 98× faster than the LEON2 at 80 MHz and 15.5× than the PowerPC 750 running at 800MHz, while our GPU implementation is 806× and 128× faster respectively. Such a massive increase in performance of a currently used image resolution shows clearly that embedded GPUs can satisfy the performance requirements of future space missions.

Acknowledgments

This work is funded by ESA under the GPU4S (GPU for Space) project (ITT AO/1-9010/17/NL/AF) and an ERC grant (No. 772773). It is also partially supported by the Spanish Ministry of Economy and Competitiveness (MINECO) under grants TIN2015-65316-P and FJCI-2017-34095 and HiPEAC.

References

[1] J. Rosello, "ESA Programs Views, Earth Observation," *12th ESA Workshop on Avionics, Data, Control and Software Systems (ADCSS)*, 2018.

[2] M. Hjorth et al, "GR740: Rad-Hard Quad-Core LEON4FT System-on-Chip," in *Data Systems in Aerospace Conference (DASIA)*, 2015.

[3] R. W. Berger et al., "The RAD750 -A Radiation Hardened PowerPC Processor for High Performance Spaceborne Applications," in *IEEE Aerospace Conference Proceedings*, vol. 5, March 2001, pp. 2263–2272.

[4] L. Kosmidis et al., "GPU4S: Embedded GPUs for Space," in *Digital System Design (DSD) Euromicro Conference*, 2019.

[5] R. L. Davidson and C. P. Bridges, "GPU-accelerated Multispectral EO Imagery Optimised CCSDS-123 Lossless Compression Implementation," in *2017 IEEE Aerospace Conference*, March 2017, pp. 1–12.

[6] N. Tsog, M. Behnam, M. Sjdin, and F. Bruhn, "Intelligent Data Processing Using In-orbit Advanced Algorithms on Heterogeneous System Architecture," in *2018 IEEE Aerospace Conference*, March 2018, pp. 1–8.

[7] A. Jung and P.-E. Crouzet, "The H2RG Infrared Detector: Introduction and Results of Data Processing on Different Platforms," European Space Agency (ESA), Presentation, 2012, http://www.esa.int/Our_Activities/Space_Engineering_Technology/Onboard_Data_Processing/General_Benchmarking_and_Specific_Algorithms.

[8] P.-E. Crouzet, "On-board processing for the Euclid mission," *ESA Workshop on Avionics, Data, Control and Software Systems (ADCSS)*, 2011.