GPU4S Bench: Design and Implementation of an Open GPU Benchmarking Suite for Space On-board Processing

Iván Rodriguez*†, Leonidas Kosmidis†*, Jérôme Lachaize‡
Olivier Notebaert‡, David Steenari§

*Universitat Politècnica de Catalunya
†Barcelona Supercomputing Center (BSC), Spain
‡Airbus Defence and Space, Toulouse, France
§European Space Agency, The Netherlands

***Abstract*—The constant demand for increased on-board performance in future space missions calls for the exploration and adoption of new hardware architectures, able to provide high performance in a low power envelope. In the GPU4S (GPU for Space) project, funded by the European Space Agency (ESA), we study the applicability of embedded Graphics Processing Units (GPUs) in space, in order to show whether current and future on-board processing algorithms can be benefited from them, as well as to select the appropriate embedded GPU which can satisfy the performance needs of future missions. However, in the absence of relevant benchmarking solutions for space applications and GPUs, a new benchmark suite had to be designed. In this work, we describe the design and implementation of the GPU4S Bench benchmark suite, which has been specifically designed to achieve the goals of our project, accompanied by some indicative results.**

I. Introduction and Motivation

The space industry is facing a dramatic increase in the performance required by future missions as future spacecraft require to acquire orders of magnitude more data compared to existing ones, supporting much higher resolutions, precision and sampling frequencies. Moreover, other types of space missions like robotic exploration such as the Rosalind Franklin ExoMars rover [1] and new types of space missions and concepts like the space tug [2] and active debris removal [3] require highly autonomous operations, which need significant on-board processing capabilities.

Embedded Graphics Processing Units (GPUs) have shown a great potential in high performance processing in temperature and battery-constrained devices, following the widespread and successful use of GPUs in high-performance domain. The GPU4S (GPU for Space) [4] project funded by the European Space Agency (ESA) aims at evaluating the potential of embedded GPUs for use in space, for the first time and define the roadmap of GPU adoption in space. In order to achieve this long term goal, several intermediate steps have to be achieved.

First, we need to confirm that the existing and mainly future aerospace software can be effectively parallelised in order to exploit the performance advantage of GPUs. This is because GPUs are known to work well with certain types of algorithms which require massively parallel processing, but also exhibit regular behaviour in memory accesses and branching.

Second, we need to ensure that programming of this new architecture can be mastered with reasonable effort by industry, and therefore highly efficient software versions can be achieved without excessive investment on the development cost. For example, the Cell Broadband Engine [5] (CBE) jointly designed by IBM, Sony and Toshiba was one of the most powerful and energy efficient architectures of its time, but it was proven notoriously difficult to program [6], reducing its industry adoption beyond the gaming sector.

Finally, we need to experimentally test various embedded GPUs to identify the most promising candidates for space use, primarily based on their performance and energy efficiency, as well as their software tools and libraries. This will result in sa selection of the most promising IP (intellectual property) GPU solution as well as the most promising COTS (commercial off-the-shelf) GPU solution. The former will be used for the development of a radiation hardened version of an embedded GPU IP based on European technology, which is the leader in embedded GPU designs, to support Europe's non-dependence in the space domain. The latter can be used in the shorter term to enable the fast adoption of GPUs in space.

In order to achieve the aforementioned necessary steps within the project, there is a clear need for a benchmarking suite which can provide all the required information to take the correct decisions. However, we notice a considerable lack of standard benchmarking solutions in the space domain, especially regarding GPUs.

In addition to the theoretical MIPS (Million instructions per second) metric of a given processing technology and MFLOPS (Million Floating Point Instructions per second), in the single core domain, we notice the use of proprietary representative benchmarks known as ESTEC-mix and Rudstone [7], as well as the Euclid mission software provided by ESA [8]. Multi-core evaluations include microbenchmarks [9], non-space representative parallel benchmarks such as SPLASH [10] and proprietary space applications like GAIA's mission [11].

This gap in benchmarking solutions has been acknowledged by ESA, which addressed this issue by developing the NGDSP (Next Generation Digital Signal Processing) Benchmark [12] [13]. However, this benchmark suite focuses only on signal processing, which is just one of the available space domains.

In general GPU benchmarking, several open benchmark suites exist such as Rodinia [14] and Parboil [15], however their algorithms are not representative of the ones used in space processing. In critical domains the only existing solution is the EEMBC ADASmark [16], however it is only available for a single GPU programming model (OpenCL) and it is not representative of on-board processing algorithms either.

In order to cover this gap which was essential for our project, we have developed an open source GPU benchmark suite focusing on space on-board processing. Our suite, GPU4S Bench has the following characteristics: it covers all space domains, it is portable to any embedded GPU and it is easily extendible to cover new programming models. Moreover, it is configurable, supporting different endianess and several data types and input sizes, in order to cover both existing and future missions requirements. In addition to randomised inputs and reference CPU implementations of each benchmark, it is also accompanied by representative inputs and outputs and a comparison framework, which allow not only to check for the equivalence of the output but also to evaluate the precision of the computation. Finally, at the end of the project its source will be released as open source.

GPU4S Bench differs with respect to other benchmark suites in the fact that in addition to performance and energy efficiency evaluation, it can also provide additional insights which are crucial for our project such as the ease of development and optimisation for a given GPU programming model, as well as the programming efficiency compared to optimal implementations provided by GPU vendors.

II. Design Principles

The GPU4S Bench design covers these requirements:

Open Source: The benchmarks need to be free of company IP rights. Although the development is funded by ESA, which owns the rights of the software and the purpose of its funded projects is to use their outcomes for the benefit of its member states, an open source benchmark suite maximises the potential of becoming the de-facto means of performance and energy efficiency comparison between embedded GPUs for space, as well as to allow reproducibility and crowdsourcing results from new architectures. Being able to directly compare results from the same suite among different targets saves time and reduces costs, while it enables taking more straightforward decisions for the hardware of future space programmes. Such a benefit has been already observed with the NIR HAWAII-2RG BM algorithm [8], which has been used in several ESA-funded and internal ESA activities for benchmarking of numerous platforms. Such an algorithm is much more useful compared to an advanced and complete but proprietary processing space application e.g. [11], which cannot be reproduced in future studies performed by different contractors.

Space Relevance and Space Domain Coverage: The benchmarks contained in GPU4S Bench need to be relevant to the space domain and accompanied with representative input data, while they need to cover as many space domains as possible. The benchmarks need to cover not only existing on-board processing in terms of both algorithms and input configurations but also future cases envisioned in the different space domains and their performance requirements.

Portability and fairness of comparison: The benchmarks should not be limited to a given embedded GPU e.g. supporting a single programming model or architecture, but to be able to run in a variety of GPUs. If the benchmark code should be differentiated to support different platforms, this should be limited to the absolutely minimum. The non-platform dependent portion of the benchmark should be identical for all platforms, in order to allow fair comparison. If support for a new platform needs to be added in the future, the benchmark structure should allow its extension in an easy manner.

Configurability: The software needs to support multiple configurations in terms of inputs sizes, execution parameters and data types used in the computation, as well as different endianess, for comparison with existing on-board processors.

Correctness and Computation Precision: The software should be functionally correct, following a standard implementation if available. A reference CPU implementation should be provided. The software should be accompanied with reference outputs in order to check for the correctness of the target platform outputs. If the outputs do not match, which can happen in parallel algorithms using floating point arithmetic [17], the precision of the computation should be evaluated.

Reproducibility: The benchmarks results should be reproducible. If random input generation is supported, replication of the results should be possible to be performed if needed.

Ease of Development and Optimisation Evaluation: The development of the benchmark suite should allow the assessment of the difficulty of software development and optimisation for GPUs. Moreover, it should enable the evaluation of the performance of the optimised version in comparison with the optimal implementation of the software.

III. Design of GPU4S Bench

In order to design a highly relevant benchmark suite covering as many space domains as possible, we have performed a survey of on-board software across all Airbus Defence and Space divisions, collecting requirements and types of software used in each space domain for current and future missions. In order to maximise our domain coverage and also comply with our freedom of company IP restrictions, we decided to identify common algorithm building blocks between different space domains. Table I provides an overview of the identified algorithms which according to our analysis span more than one domain and are included in GPU4S Bench. An additional reason for the selection of the particular building blocks is the availability of near optimal implementations from GPU vendors, which allow to compare the ease of development and evaluation as discussed later. In addition to the individual

TABLE I
FUNDAMENTAL BUILDING BLOCKS EXTRACTED FROM CURRENT AND FUTURE ON-BOARD APPLICATIONS ACROSS ALL SPACE DOMAINS.

Domains / Building Blocks	Compression	Vision Based Navigation	Image Processing	Neural Network Processing	Signal Processing
Fast Fourier Transform			GENEVIS [18]		ADS-B [19], NGDSP [13]
Finite Impulse Response Filter					ADS-B [19], NGDSP [13]
Discrete Wavelet Transform	CCSDS 122 [20]				
Pairwise Orthogonal Transform	CCSDS 122 [20]				
Predictor	CCSDS 122 [20]				
Matrix Computation		GENEVIS [18]		Inference	
Convolution Kernel		OpenCV	GO3S [21], GENEVIS [18]	Inference	
Max Detection			GO3S [21]	Inference	ADS-B [19]
Synchronization Mechanism		GENEVIS [18]	EUCLID NIR [8], GO3S [21]	TensorFlow	ADS-B [19], NGDSP [13]
Memory Allocation		CERES [22], OpenCV	EUCLID NIR [8], GO3S [21]	TensorFlow	ADS-B [19], NGDSP [13]

building blocks, we have also selected to implement two space-relevant complex applications, in order to be able to demonstrate additional effects that are only visible when performing a chain of GPU operations. In Section IV we provide a detailed information of the GPU4S benchmarks.

In addition to the benchmark selection, we have selected the appropriate parameters e.g. input sizes which match existing and future missions requirements, and we defined representative inputs for them. Each algorithm is implemented in a parametric way to support multiple data types: single precision floating point (`float`), double precision floating point (`double`), and 32-bit integer (`int`). For the GPU implementation of the benchmarks we also support half precision (16-bit) floating point (`half`) in order to evaluate their performance impact compared with the rest of the data types. However, since this data type is not supported natively in CPUs, we don't provide any means for functional validation.

Furthermore, we added the option of random input generation to increase the potential test cases and ensure that the benchmark implementations behave as expected under different inputs. To guarantee reproducibility, we print the random seed used in an experiment with randomised input, and we support setting a specific seed for repeating experiments.

Since we required a standard implementation for each building block, we have selected to follow the specification of existing widespread software for our reference CPU implementation and our equivalent GPU code. In particular, we follow the implementation of MATLAB/GNU Octave, which are standard tools used frequently for prototyping on-board algorithms and which we have also used for our output validation methodology, explained in Section V. In addition, we follow the implementation of optimised vendor GPU libraries for both functional validation as well as for development and optimisation efficiency comparison, as discussed further in Section VI, which in all but one cases (neural network convolution) match the MATLAB specification.

IV. THE GPU4S BENCH SUITE

The domains we covered in our analysis are the following: Earth and sky observation, as well as science missions are primarily focused on image processing and analysis for processing the acquired data and compression for transmitting them in to ground. In telecommunication satellites on the other hand, the dominant type of computation is signal processing.

Future missions are expected to be highly autonomous in order to support applications such as Active Debris Removal [3] and robotic exploration. These functionalities can be enabled using Vision-based Navigation for Guidance and Navigation (GNC) and AI (Artificial Intelligence) inference solutions based on Deep Neural Networks (DNN).

Next we examine the building blocks and complete applications and how they fit on the above domains.

A. Building Blocks

Fast Fourier Transform: The Fast Fourier Transform (FFT) is a ubiquitous algorithm used across several space domains, mainly in telecommunications and for image analysis, in its 2D form. Moreover, it is an essential part of the ADS-B (Automatic Dependent Surveillance) system [19], a surveillance technology in which an aircraft determines its position via satellite navigation. In the latter case, the FFT is applied in a sliding window of 128 points. For the library implementation of this block, we are using the cuFFT library for NVIDIA targets and the clFFT library for OpenCL targets and for validation we use the MATLAB `fft` function.

Finite Impulse Response: The Finite Impulse Response (FIR) filter is also widely used in signal processing. No vendor-provided library supports this block, so we only use the MATLAB option of the convolution function (`conv`) between the signal and the filter taps.

Matrix Operations: Matrix operations are used in many domains, especially matrix multiplication which is one of the most well studied and understandable benchmarks. In particular it is used in Vision-based navigation for perspective corrections and in neural processing, where it is the fundamental way of implementing inference in a fully connected network. We use the MATLAB matrix multiplication functionality for functional validation, and the cuBLAS and clBLAS libraries for the development efficiency evaluation.

Discrete Wavelet Transform, Pairwise Orthogonal Transform and Predictor: These algorithmic blocks are used in compression within the CCSDS-122 standard [20]. Since these building blocks are contributing in a single domain and it is known from previous ESA studies that this algorithm is

difficult to parallelise due to dependencies, we prioritised the implementation of the rest of the algorithms.

Convolution: Convolution is used in conjunction with images in its 2D form for vision based navigation and in image processing. It is also fundamental for the implementation of Convolutional Neural Networks (CNN). Note however that the former implements convolution with padding, functionally equivalent to MATLAB, while the latter is implemented without padding as defined in cuDNN library from NVIDIA.

Max: Maximum detection is common across several domains: image, neural network and signal processing. It is also included in the softmax and max-pooling of the cuDNN library. Other DNN building blocks included in GPU4S Bench are the Relu and the Local Response Normalisation.

Synchronisation and memory allocation: The effect of these operations are visible in complex processing chains, for this reason they are implemented in the complex applications of our suite. They can be found in all space domains.

B. Complex Applications

CIFAR-10: This application performs inference using a neural network with 10 layers, trained with the CIFAR-10 dataset [23]. Each layer is implemented by reusing the neural network building blocks from the individual benchmarks.

Euclid NIR: This application [8] is widely used in ESA projects, therefore we performed its GPU parallelisation.

In both individual building blocks and applications we support a mode in which multiple frames are processed, in order to amortise the additional overhead of GPU memory allocations and transfers.

V. Functional Validation and Precision

Our benchmarks are configurable to use floating point or integer formats. A given GPU may favour the implementation for a certain format, but their results might not be bit-identical. However, we need to know whether the results of a benchmark execution are functionally correct, and also what is the precision of the computation in the target GPU. For the benchmarks with an available library implementation we validate our benchmark results against the library implementation, for its supported data types e.g. cuFFT supports only single or double precision floating point. However, for the rest of the cases, we designed a methodology to measure the residual error of the target GPU against a double precision computation on a CPU.

For each benchmark, we include a sequential reference implementation written in C, which we execute in the maximum precision format, e.g. double floating point format (64-bit). The benchmark is paired with a representative reference input data set in 64 bits binary format of the same input type. The output of the reference implementation is also stored in binary form of the IEEE-754 double format, producing the reference output, also known as golden standard or ground truth.

For functional verification we execute the target implementation and store the results, which are subsequently converted to IEEE-754 double binary format. Finally, the target binary output is compared to the reference binary output and select the maximum difference value, which is the reference error.

We selected the binary representation for the input, output and comparison, because the precision of a decimal representation of an IEEE-754-encoded value has not always enough significant digits. For example, two floating point values printed with 15 digits after the decimal point cannot be represented unambiguously. For example, both values (in IEE754 format) 0x327d7168acfae2c8 and 0x327d7168acfae2cc are displayed as 1.747362713315761e-65. For this reason, we have written a MATLAB/GNU Octave program which performs the comparison of the benchmark output against the golder output and displays the residual error. As an indication, the residual error reported by our procedure for the matrix multiplication benchmark using the reference input on several NVIDIA platforms is negligible, 7.4351e-15 which is acceptable, considering that much of on-board processing is currently performed in fixed point and with much less precision bits. Since both our implementations and the CUDA library provides identical results, this difference from our sequential reference CPU version comes from the parallel execution of floating point instructions which slightly affects its result [17].

VI. Ease of Development and Optimisation

Our benchmark suite design doesn't only focus on the comparison of the hardware platforms that it is executed on, but also their software stack, including their libraries and programming models. In order to support the code portability in all embedded GPUs, we designed our benchmarks to allow multiple implementations from different programming models, which is described in detail in the next Section. However, even within the same programming model, we provide 3 different implementations: a naive implementation, an optimised version and a version using the vendor provided library if available. The naive implementation is a straightforward GPU parallelisation which may be suboptimal, but it is portable across different GPUs. The optimised version is parametric and includes common optimisations like thread coarsening, tiling/blocking (use of shared memory in order to reduce bandwidth from the main memory and allow for reuse across threads in the same block/workgroup) and loop unrolling. This optimised version requires tuning on the specific GPU target, in order to find the optimal configuration of the optimisation parameters. Finally, the vendor provided library version is known to be the most optimised version available for a GPU.

All implementations of a given benchmark are undertaken by the same expert GPU developer and the development time of each version is recorded. This allows three different types of evaluation. First, by comparing the performance of the naive and optimised implementations against the vendor optimised library, we can get an indication of how close a handwritten implementation can get to the true performance capabilities of the GPU. If the naive implementation is good enough, there is no need for further optimisation of the code. Otherwise, the optimised version can provide an indication of how close to the optimal performance can be achieved with reasonable effort.

TABLE II
TIME SPENT IN THE DEVELOPMENT OF EACH BENCHMARK

Algorithm	Total Development Time per Benchmark in hours	Time to Develop in hours						
		CPU	CUDA	OpenCL	CUDA Opt	OpenCL Opt	CUDA Lib	OpenCL Lib
Matrix Multiplication	8.5	1	1	0.5	1	1	2	2
FFT	33	8	14	2	5	1	2	1
FFT window (ADS-B)	12	2	3	2	1	1	1	2
FFT 2D	11	5	-	-	-	-	3	3
Convolution 2D	11	5	1	1	2	1	1	-
Relu	4	0.5	0.5	0.5	1	0.5	1	-
Max Pooling	6	1	1	0.5	2	0.5	1	-
Softmax	10	1	1	2	3	2	1	-
Local Response Normalization	3.5	0.5	0.5	0.5	0.5	0.5	1	-
Finite Impulse Response	4	1	2	1	-	-	-	-
Neural Network for CIFAR-10	40	-	10	10	8	8	4	-
Neural Network for CIFAR-10 Multiple	15	-	2	2	4	5	2	-
Total	158	25	36	22	27.5	20.5	19	8

Second, the recording of the development time can provide an indication of the development effort compared with the additional performance gained. For example, if the optimised version comes with a 20% increase in the development time and provides a bigger performance increase e.g. 50%, then the optimisation can be deemed worthwhile. Third, the development time of the algorithm provides useful information about how easy and productive is to use a certain programming model. If the naive implementation development takes too long – or much longer than another programming model – it might influence the decision of selecting a certain GPU architecture.

Finally, it is worth noting that the handwritten implementations are also important for another reason. In critical systems subject to certification/qualification (which is not the case of all space systems though), the availability of source code of all software is critical. However, this is not the case of vendor provided libraries, which are provided in a black box form and only their interfaces are available.

Table II provides the development time of each of the building blocks of GPU4S. Despite OpenCL is a lower-level language than CUDA and therefore is more challenging, this is not reflected in the development time of the benchmarks. The reason for this is that we first implemented the 3 CUDA versions before moving to the OpenCL one and as such we greatly benefited from basing our OpenCL implementation on the CUDA versions. Moreover, we notice that the time it took us to implement the reference CPU implementation compared to the first CUDA implementation (naive) is almost equivalent, which confirms that the ease of development.

Although we cannot provide full performance results for all the platforms we have evaluated so far, since the project is still on-going, we provide some indicative results on two embedded platforms based on NVIDIA GPUs, which are the ones with more available vendor provided libraries, the NVIDIA TX2 and NVIDIA Xavier. For matrix multiplication for floating point our hand-written versions provide very low performance around 5 and 10% of the provided library. However, for double precision on the Xavier we outperform the NVIDIA library, probably because it is not optimised yet for this platform. Surprisingly, we also outperform both cuFFT and cuDNN basic blocks for our tested sizes. The reason in both cases is that both libraries have a large initialisation cost, which is not amortised in relative small sizes (64K elements) required in on-board processing. Moreover, cuDNN is more efficient for training of neural networks as opposed to inference which we are interested in. However, if the initialisation cost of cuFFT is excluded and it is executed repeatedly 30K times, our handwritten versions achieve only 10-15% of its performance.

VII. BENCHMARK STRUCTURE

Our benchmark suite is designed to use multiple programming models, similar to other general purpose GPU suites. However, our main difference is that in order to guarantee fair comparison between them, we don't provide separate benchmark implementations which may structurally differ for each programming model like e.g. Rodinia [14], but each benchmark is divided in target-agnostic and target specific portions. Although in the current version of the GPU4S suite we only support CUDA and OpenCL, our design can support other programming models which we plan to support in the future such as OpenGL ES 2, OpenGL SC 2, Brook Auto [24] and Vulkan in order to cover the entire embedded GPU hardware space. Our implementation allows us to share as much as possible code between the different programming model versions. The structure of each benchmark is as follows:

1) Platform initialization: this step performs the necessary actions to select the compute device etc. It is mostly required for OpenCL since the user needs to explicitly initialize the compute environment.
2) Read reference input files: in this step the application reads the reference input files in the format explained in Section V, and puts them in the host memory of the accelerator. This is a platform independent step.
3) Copy the reference input data to the GPU memory: the benchmark initiates the transfer towards the accelerator. This step uses different calls in CUDA and OpenCL, but it provides a common function interface with different implementation per programming language.
4) Kernel Invocation: in this step the computation is offloaded to the GPU. Its implementation depends on

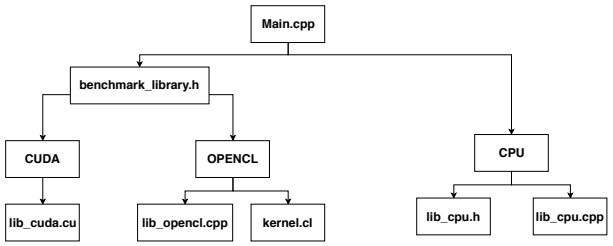

Fig. 1. Generic benchmark structure

CUDA or OpenCL but it is done similar to the previous step. As explained in Section VI there are three different versions which are selected: naive, hand-optimised and vendor-provided library implementation.

5) Copy the reference output back in the host memory: this is the same as step 3 with the opposite direction of the transfer. The steps from 3 to 5, or only the 4 can be executed multiple times in a loop, in order to increase the execution time of the benchmark so that the measurement is not subject to measurement precision errors. Moreover, allows to simulate real use scenarios of the GPU like repetitive transfers to the GPU between computations, or a single transfer of data followed by several kernel invocations between transfers.
6) Write reference output file: in this step, the benchmark output is saved in a file, using the binary format of its implementation data type. This functionality is platform independent. If the benchmark output type does not match the reference output type, an offline tool written in Matlab/GNU Octave, performs the conversion and the comparison.

The main benchmark body is platform independent, written in C for maximum portability and it only contains calls to the different steps. The platform independent steps are implemented in a common library, while the platform dependent steps are implemented in separate libraries with same contracts. Preprocessor directives and different makefile rules ensure that each version uses the appropriate components. The CUDA and OpenCL kernels are implemented in separate files, but they support the same interface. A visual representation of the benchmark structure is provided in Figure 1. The main data types involved in the computations for both the main benchmark body as well as the kernel implementations are written in such a way that can be changed so that different benchmark versions are implemented for the data types of interest. This is achieved either with preprocessor directives or typedefs depending on the particular needs of each benchmark.

VIII. Conclusions

In this paper we described the design and implementation of the GPU4S benchmark suite, which targets GPU on board-processing for space applications. We have explained our design principles and design decisions, as well as our methodology. Finally, we have presented some indicative results, showing that the benchmark suite covers all our needs.

References

[1] K. McManamon et al., "ExoMars Rover Vehicle Perception System Architecture and Test Results," *12th Symposium on Advanced Space Technologies in Robotics and Automation (ASTRA)*, 2017.

[2] M. Mammarella et al., "The Lunar Space Tug: A sustainable bridge between low Earth orbits and the Cislunar Habitat," *Acta Astronautica*, vol. 138, pp. 102 – 117, 2017.

[3] S. Kawamoto et al., "Current Status of Research and Development on Active Debris Removal at JAXA," *7th European Conference on Space Debris (SDC7)*, 2017.

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

[5] M. Gschwind et al., "Synergistic Processing in Cell's Multicore Architecture," *IEEE Micro*, vol. 26, no. 2, pp. 10–24, March 2006.

[6] A. Arevalo et al., *Programming the Cell Broadband Engine Architecture: Examples and Best Practices*. IBM Red Books, 2008.

[7] J. Gaisler, "Benchmarking of 32-bit processors for space applications," ESA/ESTEC, Tech. Rep. WDI/JG/2105/NL Issue 4, 20-11-1995, 1995. [Online]. Available: http://microelectronics.esa.int/erc32/misc/ERC32-ADA-Benchmarking-Gaisler-1995-11-20.pdf

[8] 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.

[9] F. J. Cazorla et al., "Multicore OS Benchmarks," Final Report, ESA-ESTEC, Tech. Rep. RFQ- 3-13153/10/NL/JK, 2012. [Online]. Available: http://microelectronics.esa.int/gr740/MulticoreOSBenchmark-FinalReport_v7.pdf

[10] G. Beltrame et al., "Benchmarks for the GINA platform," Final Report, ESA-ESTEC, Tech. Rep., 2008. [Online]. Available: http://microelectronics.esa.int/gr740/GINABench.pdf

[11] D. Hellström and F. Cros, "RTEMS SMP Final Report: Development Environment for Future Leon Multi-core," Final Report, ESA-ESTEC, Tech. Rep. RTEMSSMP-FR-00, 2015. [Online]. Available: http://microelectronics.esa.int/gr740/RTEMS-SMP-FinalReport-CGAislerASD-OAR.pdf

[12] J. Franklin, "NGDSP Benchmarking & SDE Evaluation. Final Report. European DSP Trade-off and Definition Study," ESA-ESTEC, Tech. Rep. 22645/09/NL/LvH, 2012. [Online]. Available: http://spacewire.esa.int/edp-page/events/DSP_Day_Presentation_Astrium_UK_NGDSP-tradeoff-study.PDF

[13] ESA, "On-Board Data Processing - Benchmarks," 2012, https://www.esa.int/Enabling_Support/Space_Engineering_Technology/Onboard_Data_Processing/General_Benchmarking_and_Specific_Algorithms.

[14] S. Che et al., "Rodinia: A Benchmark Suite for Heterogeneous Computing," in *2009 IEEE International Symposium on Workload Characterization (IISWC)*, Oct 2009, pp. 44–54.

[15] J. Stratton et al., "Parboil: A Revised Benchmark Suite for Scientific and Commercial Throughput Computing," University of Illinois at Urbana-Champaign, Urbana, Tech. Rep. IMPACT-12-01, Mar. 2012.

[16] EEMBC. (2019) The ADASMark Benchmark. [Online]. Available: https://www.eembc.org/adasmark/

[17] N. Whitehead et al., "Precision & Performance: Floating Point and IEEE 754 Compliance for NVIDIA GPUs," NVIDIA, Tech. Rep., 2011.

[18] R. Brochard et al., "Scientific Image Rendering for Space Scenes with the SurRender Software," in *69th International Astronautical Congress (IAC)*, 2018.

[19] M. Strohmeier, M. Schfer, V. Lenders, and I. Martinovic, "Realities and Challenges of Nextgen Air Traffic Management: The Case of ADS-B," *IEEE Communications Magazine*, vol. 52, no. 5, pp. 111–118, 2014.

[20] The Consultative Committee for Space Data Systems, *Image Data Compression Recommended Standard CCSDS 122.0-B-2*, 2017.

[21] E. Maliet et al., "Geostationary Observation Space Surveillance System (GO3S) Real Time Video From Space," in *65th International Astronautical Congress (IAC)*, 2014.

[22] CNES, "CERES: Three satellites to boost Frances intelligence capabilities," 2019, https://ceres.cnes.fr/en/ceres-2.

[23] A. Krizhevsky, "Learning Multiple Layers of Features from Tiny Images, , 2009." University of Toronto, Tech. Rep., 2009.

[24] M. M. Trompouki and L. Kosmidis, "Brook Auto: High-Level Certification-Friendly Programming for GPU-powered Automotive Systems," in *55th Annual Design Automation Conference (DAC)*, 2018.