MICADO-MAORY SCAO RTC system prototyping: assessing the real-time capability of GPU

F. Ferreira[a], A. Sevin[a], J. Bernard[a], and D. Gratadour[a]

[a]LESIA, Observatoire de Paris, Université PSL, CNRS, Sorbonne Université, Université de Paris, 5 place Jules Janssen, 92195 Meudon, France

ABSTRACT

MICADO is an ELT first light near-infrared imager. A Single Conjugate Adaptive Optics (SCAO) system will correct the impact of atmospheric turbulence to allow the imager to work at the diffraction limit of the telescope, and so to obtain the best image quality. A critical component of this system is the Real-Time Computer (RTC) which computes the commands to apply on the Deformable Mirror (DM) from the measurements obtain from the Wave Front Sensor (WFS). To reach the expected AO performance of MICADO, the RTC has to compute the commands in 285 μs after the WFS image last pixel receipt. Considering a classical command law based on a Matrix-Vector Multiply (MVM), this requirement leads to around 1,800 GB/s memory bandwidth in single precision for a Pyramid WFS giving around 25,500 slopes and 5,000 DM actuators to command. Graphics Processing Units (GPU) architecture provides high memory bandwidth that makes the use of this hardware the baseline of the hardware RTC for MICADO SCAO module. In this paper, we present OCEAN, an optimized AO RTC environment that allows to configure, execute, validate and benchmark the GPU-based algorithms developed for the RTC. Validation is made by interfacing the RTC with the end-to-end AO simulation platform COMPASS which acts here as a WFS image and a DM correction simulator. Communication between RTC and simulation tool is based on standard multi processes interfaces developed as part of the CACAO framework and semaphores based. We will provide the performance obtained with the RTC core kernels based on standard CUDA implementation. Measurements of the real time capability of this approach is also described in terms of latency and jitter. Finally, the use of half precision floating point format on GPU will be discussed.

Keywords: Adaptive optics, ELT, RTC, GPU, MICADO, COSMIC

1. INTRODUCTION

The near-infrared imager MICADO[1] (Multi-AO Imaging Camera for Deep Observations) is one the first light instrument for the incoming ELT. It will be equipped with an adaptive optics (AO) system to compensate the atmospheric perturbations on the scientific image: a Multi-Conjugate Adaptive Optics (MCAO) system called MAORY.[2] This AO system will also have a simpler Single Conjugate AO (SCAO) mode[3] commanding the ELT M4 actuators from a pyramid wavefront sensor (WFS). The current baseline for this mode is to use a 96x96 pyramid WFS using the so-called "full pixels" algorithm,[4] leading to around 25,500 measurements.

For this SCAO mode, the AO Real-Time Controller (RTC) will then have to handle those 25,500 measurements and to compute command for the 5,316 actuators of the ELT M4 deformable mirror. Considering a classical command law based a Matrix Vector Multiplication (MVM), and the 285 μs time budget allowed to the RTC computation, it leads to a memory bandwidth of 1.8 TB/s in single precision. In this context, Graphics Processing Units (GPU) have been chosen as the hardware core component for the SCAO RTC. Indeed, most recent GPU cards bring up to 900 GB/s of memory bandwidth.

GPU suitability for AO RTC had already been studied during the GreenFlash project.[5] Legacy of the latter leads to the COSMIC platform[6] on which the MICADO MAORY SCAO RTC is based. COSMIC aims to propose an open, standard and modular RTC platform that can serve any AO systems. It leverages the GreenFlash outputs of both FPGA flexibility to adapt to hardware, and GPU acceleration for high performance

Further author information: (Send correspondence to F. Ferreira)
F. Ferreira: E-mail: florian.ferreira at obspm.fr, Telephone: +33 (0)1 45 07 71 78

computing, especially for linear algebra. The work presented is part of the H-RTC core implementation of this platform.

Most recent Nvidia GPU cards come with dedicated half precision computation unit, called tensor cores. Those units promise high throughput performance for half precision linear algebra operations, mainly aiming at deep learning and artificial intelligence applications. As most of the AO systems rely on MVM to compute the DM commands, leveraging half precision possibilities could be interesting, especially for extreme-class telescope.

This paper focus on the developments that led to the OCEAN (Optimized Core Environment for Adaptive optics pipeliNe) which define the core software stack of the COSMIC platform, and so of the MICADO MAORY SCAO RTC. The goal is to provide a modulable and scalable RTC implementation from its core computational kernels on the GPU from the user interface that allows easy control and monitoring of the RTC while it is running. This approach allows us to leverage the COMPASS[7] RTC module which already include most of the GPU computation kernel that will be used in the RTC. This approach also allows common development and validation between the simulation tool and the RTC.

Firstly, we present the OCEAN framework and its different components in terms of software architecture. Then, we validate the overall RTC implementation thanks to the COMPASS simulation tool and, finally, we perform benchmark for the MICADO MAORY SCAO module scale and discuss the results obtained.

2. THE OCEAN FRAMEWORK

In this section, we present the OCEAN framework and the architecture of its different components which, once combined together, define the HRTC core architecture. Figure 1 shows this architecture, and the interaction

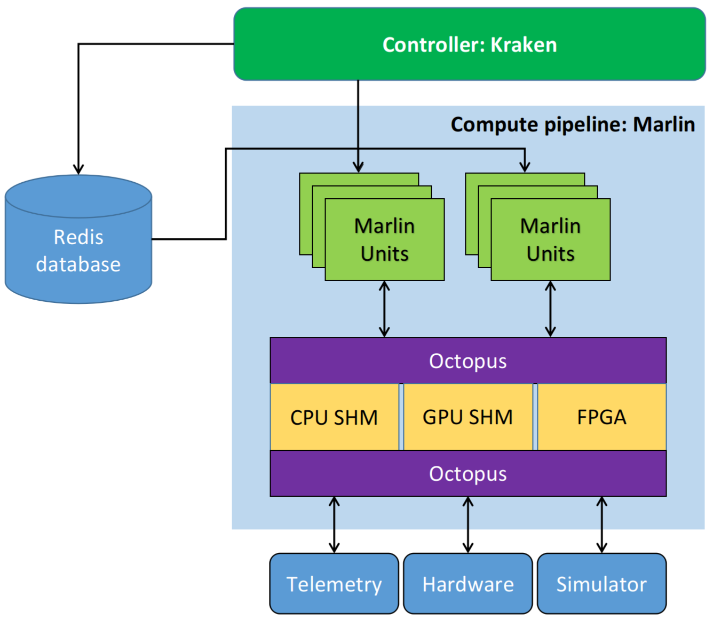


Figure 1. HRTC Core architecture

between its three main components :

- *Marlin*, the core computation pipeline
- *Octopus*, the shared memory interface
- *Kraken*, the pipeline controller

2.1 Marlin: the worker

Marlin is C++ library that defines a container for the core real-time components of the AO RTC pipeline. This container can then be launched as an independent process, called business unit, which is in charge of a specific computation inside the whole RTC pipeline. Then, the full pipeline is composed of several business units that can be executed concurrently or sequentially depending on the way the pipeline is built. Indeed, Marlin includes an abstraction layer that allows the user to define the interfaces of a business unit. Those ones are datas stored in shared memory: access to them and synchronization between the business units are provided through Octopus, which is described in the next section.

The main interfaces that must be defined for a business unit are the inputs, i.e. which datas the business unit has to wait for before starting its computation, and the outputs which are the location in shared memory where the results of the computation have to be written. With this approach, two business units can naturally be executed sequentially if the output of one is the input of the other. Other interfaces can also be defined if the business unit needs auxiliary data for its computation, such as command matrix or reference slopes for instance.

Marlin also provides an abstraction of the concept of business unit itself to allow any user to add a new business unit in the library. To do so, the user/developer only has defined 3 functions in order to add its new feature in the library :

- a function that defines the computation that the business unit has to do, i.e. the computation kernel itself. Note that this code could be either GPU or CPU based.
- a function that defines what the business unit is waiting for before launching its kernel
- a function that defines how the business unit notifies that the computation is over and the result is ready

Then, the Marlin executable remains unchanged and will understand that a new business unit is available.

As already said, the RTC pipeline consists of several Marlin business units, as represented in Fig.1 by the light green boxes. In fact, each of those boxes are composed of two separate processes as shown in Figure 2. One

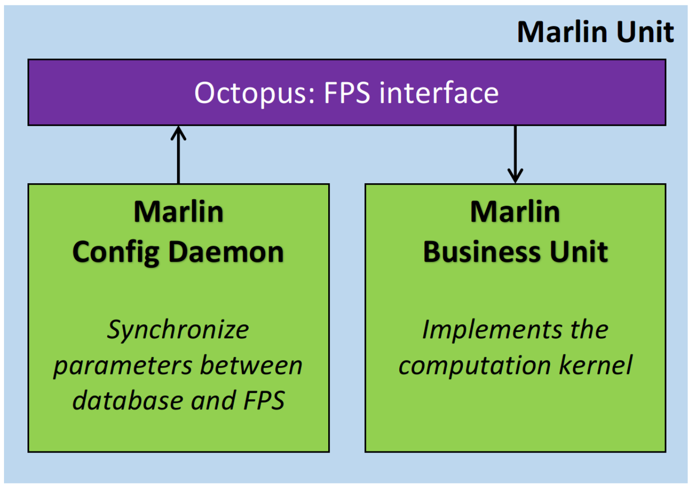

Marlin Unit
Octopus: FPS interface
Marlin
Config Daemon
Synchronize parameters between database and FPS
Marlin
Business Unit
Implements the computation kernel

Figure 2. Processes that composed a Marlin Unit: the configuration daemon synchronize the parameters value set in the database with the one effectively used by the real-time process

of those processes is the real-time business unit itself, as described above. The second one is a non real-time daemon which is in charge of refreshing all the scalar parameters needed by the business unit with respect to the parameters defined by the user in a database using Redis. Basically, this daemon reads all the useful parameters in the database and copy them in a specific shared memory structure on which the business unit can retrieve those parameters value. This approach allows to interact with real-time pipeline while minimizing the additional latency that could be introduced.

Refreshing the auxiliary datas of the business unit is also possible thanks to double buffering provided by Octopus: the new data are written in the buffer and, once ready, the business unit is notified to switch the pointer inside the ring buffer. Then, there is no additional latency due to data transfer.

Currently available business units are based on the developments made for COMPASS, more specifically for its RTC module. It includes:

- A centroider business unit able to perform different flavour of center of gravity algorithm (classical, thresholded, weighted...) and also dedicated pyramid WFS algorithms (full pixels, slopes based). This business unit also handles the WFS raw image calibration (flat, dark...)
- A controller business unit able to computes DM command from WFS measurements using a classical MVM approach coupled to an integrator

As those core kernels have already been highly tested and stressed in the context of COMPASS simulation, this approach limits the development costs and comes with already validated algorithms. Furthermore, any correction of a bug found in the RTC module of COMPASS will be automatically be applied in the business unit.

2.2 Octopus: the interfacer

Octopus is a C++ library based on the CACAO[8] framework that provides an abstraction layer to allocate and expose shared memory on either a CPU, GPU or FPGA device. The library is provided with a Python wrap to make it available from both C++ and Python based codes. The basic features available through Octopus include shared memory allocation, sending and receiving data to and from the shared memory, and synchronization mechanism through semaphores.

Those features are obtained from the CACAO framework, more specifically from the ImageStreamIO library. As just mentioned, it implements a semaphore based synchronization scheme on which Marlin business units could rely in order to execute sequentially. The CACAO framework also implements a parameter structure, called Function Parameter Structure (FPS), that allows to store scalar parameters in shared memory. Octopus then provides an interface with this structure that allows the Marlin configuration daemon to update the parameters from the database, and the business unit to read those parameters from the shared memory.

As shown in Fig.1, Octopus then allows to connect any process to the real-time pipeline through shared memory, preventing then any direct perturbations in terms of latency and jitter. Octopus is then used to retrieve telemetry, connect hardware devices or simulator to the pipeline. This last approach will be described in section 3 for the pipeline validation.

2.3 Kraken: the controller

Kraken is a Python based software that aims to manage the RTC pipeline. It is also the user interface with the RTC. From a set a parameter files given by the user (one file per business unit in the pipeline), Kraken is in charge of:

- Loading the configuration from the file
- Populating the redis database from this configuration
- Allocating and managing the shared memory needed by the pipeline
- Launching each configuration daemon and business unit processes
- Monitoring the pipeline

A business unit should be configured from a dedicated configuration file that specifies the various parameters and shared memory interfaces needed. As an example, it sets the size of the shared memory allocation needed, the device on which it must be allocated (CPU or GPU), or the CPUs on which the business unit will be launched. Each business unit comes with his own configuration file template that indicates the parameters that have to be defined.

From this configuration file, the main application of Kraken, called Kraken Manager, will populate the redis database with a new business unit field that includes all the parameters values. Those values will then be retrieved by the Marlin configuration daemon in order to correctly instantiates the corresponding business unit in the real-time pipeline. A specific attention has been paid to the structure of the database in order to make it invariable whatever the pipeline is. All the business units fields in the database follow the exact same architecture: it is this approach that allows the abstraction layer of Marlin to work, as it always knows where to find the parameters, and this whatever the business unit being created.

Those parameters also define the various shared memory allocation needed. Those allocations are made through Octopus that provide a named interface for each shared memory space needed. All those interfaces are stored by the manager in order to avoid multiple allocation of the same interface, and also multiple writer registration to avoid that two business units writing at the same time on the same location.

Finally, the Kraken Manager launch all the loaded business units as independent processes inside separate tmux sessions. The configuration manager is launched first in order to create the FPS structure in shared memory with updated parameters from the database. Then, the business unit itself is launched.

Kraken also provides methods that allow the user to interact with the RTC and to change its configuration on the fly. Those methods basically wrap the access and modification of the redis database, and it will be pass on the RTC through the configuration daemon while the pipeline is running.

3. REAL-TIME PIPELINE SIMULATION VALIDATION

In this section, we present validation process of the OCEAN framework. It is based on end-to-end simulations provided by COMPASS which acts here as a simulator of turbulence, WFS and DM. For this section and the rest of the paper, we will consider a monolithic version the HRTC pipeline where the all the computations are handled by a single business unit. The goal is to bring a simple proof concept of the whole software stack for the HRTC.

3.1 Validation setup

Even if all the current computation kernels available in Marlin had already been strongly validated through COMPASS simulation, the overall approach of the proposed HRTC architecture still has to be validated to ensure that the communication and synchronization between business units through Octopus interfaces do not introduce errors.

Figure 3 shows the simulation setup that is provided by Kraken and COMPASS. On the left part of the figure, the block diagram represents the steps of each iteration of a COMPASS AO loop simulation, from atmosphere generation to deformable mirror shape modeling. The principle of our simulation setup with the RTC is to plug a COMPASS simulation on Octopus interfaces: the generated WFS image are written on shared memory to make it available for the RTC, and the latter writes the DM commands it has computed on shared memory for COMPASS. This way, the RTC pipeline simply replaces the RTC module of the COMPASS simulation to compute the DM commands from a WFS image.

A monolithic RTC pipeline is composed of a single business unit in charge of computing the DM commands from the WFS frames. This pipeline includes 2 different shared memory interfaces to store the WFS image and the DM commands. It implies the same number of synchronization points between the simulation processus and the business unit.

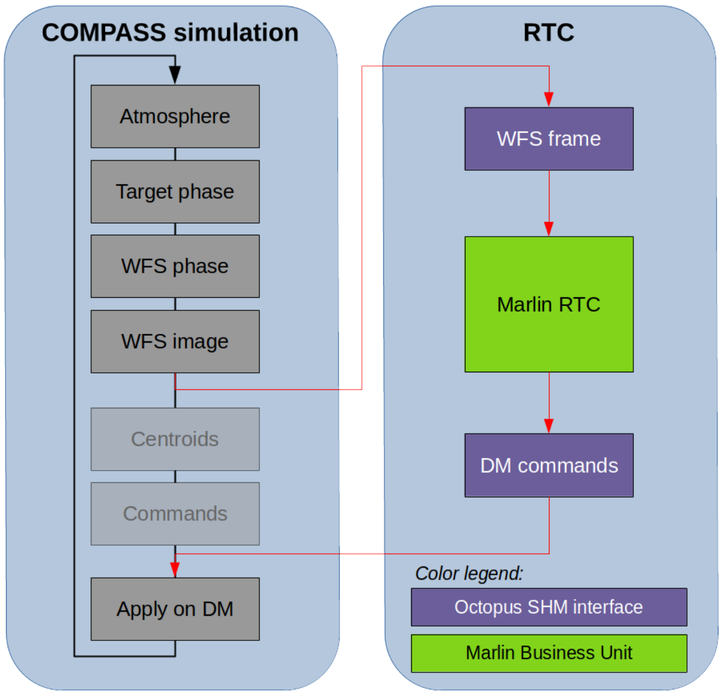


Figure 3. Simulation process used with COMPASS to validate the RTC pipeline given by the OCEAN framework

3.2 Results

As the business unit kernels relies on the ones developed in COMPASS, we obviously expect to have the exact same simulation result between a full COMPASS simulation and this setup. Then, we ran 100,000 iterations of a simulation on a 10-meters class telescope, and we compare the final Point Spread Functions (PSF) which are shown in Fig. 4. It led to a maximum difference of 2.10^{-7} for a PSF maximum of 0.70 which is comparable with the numerical precision for floating point operation.

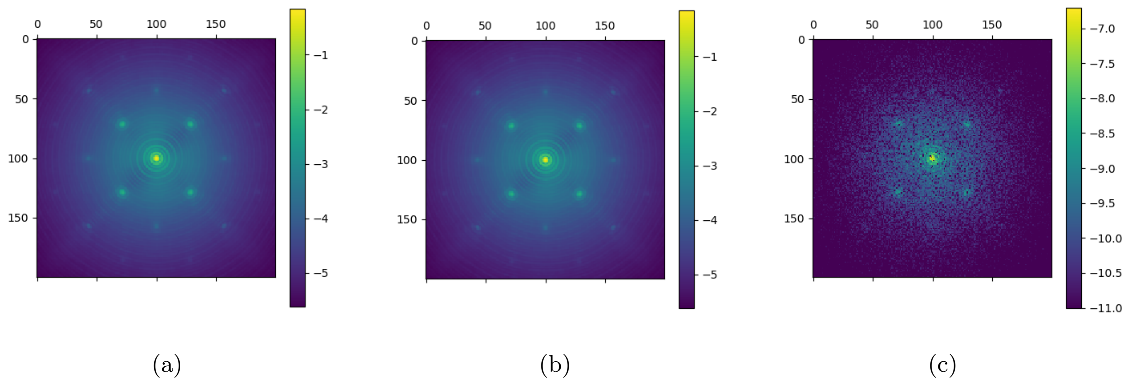


Figure 4. (a) PSF obtained from a full COMPASS simulation, (b) PSF obtained with the RTC pipeline plugged to COMPASS (c) Difference between the two PSFs. All images are represented in a logarithmic scale

3.3 Half-precision impact on AO performance

As stated in introduction of this paper, using half-precision floating format in the RTC pipeline could be something to explore for extremely large telescopes. However, the current cuBLAS library only provide matrix-matrix multiplication algorithm. Furthermore, tensor cores are optimized for specific dimensions of those matrices: the number of rows should be a multiple of 4 while the number of columns should be a multiple of 8. As the command matrix does not necessarily respect those conditions, it has to be zero-padded to right dimensions to extract maximum performance from tensor cores. Then, we use the matrix-matrix multiplication algorithm by specifying that the second matrix has only 1 column to emulate a matrix-vector multiply.

In order to verify that half precision could be used to perform the MVM operation in the RTC without loss of AO performance, we compare the instantaneous Strehl ratio (SR) obtained from two exact same simulations, i.e. with the same turbulent screen at each iteration, except that one of them performs the MVM in half-precision while the other is running in single precision.

Figure 5 shows the results obtained by plotting the SR obtained in single precision versus the one obtained in half precision, and also by plotting the difference between those SR frame by frame.

4. PERFORMANCE

In this section, a benchmark of the simple RTC pipeline exposed in the previous section is performed at the MICADO MAORY SCAO scale. We first present the method used to time the pipeline operation, the benchmark environment, and we finally expose the results obtained in terms of latency and jitter.

4.1 Timing method & Environment

As the core computation kernels run on the GPU, the pipeline is timed using cudaEvents that allow to properly time GPU execution. Indeed, classical CPU-clock based method could not bring correct timings as the CPU and GPU execution threads are asynchronous. CudaEvents are proposed in the CUDA toolkit to perform this kind of operation.

For this benchmark, cudaEvents are used to get the timing between the reception of last pixel of the WFS frame, i.e. when the corresponding interface is notified, and the end of the DM commands computation.

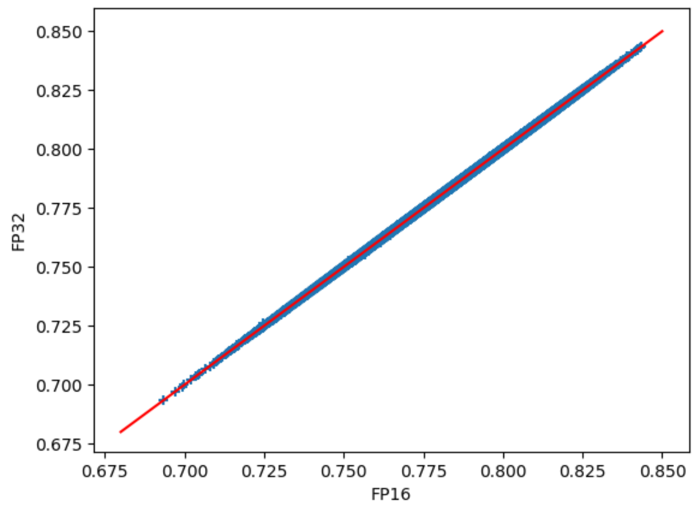
It was done on a Nvidia DGX-1 server, with Dual 20-core Intel Xeon E5-2698 v4 @ 2.2 GHz and 8 Tesla V100-SXM2 GPU. Only one of those GPU is used by the current RTC pipeline.

The latter is dimensioned as the MICADO MAORY SCAO RTC baseline. It includes:

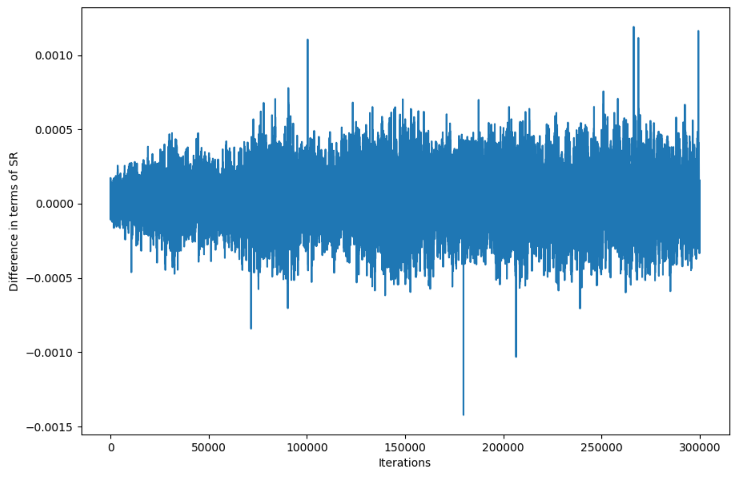
- calibration in terms of dark and flat frames of the 240 by 240 pixels WFS frames
- "full-pixel" pyramid algorithm retrieving and normalizing 25,520 pixels from the four pyramid pupils
- 25,520 by 5,316 MVM computation
- Addition of perturbation voltage, clipping and conversion to correct format for the DM

The benchmark was performed over 10 millions iterations at the maximum speed of the pipeline: when an image is sent, the resulting DM commands are awaited before sending a new WFS frame. It is equivalent to almost 2 hours of running time. The same benchmark has been performed with both single and half-precision format. For the latter case, only the MVM is performed in half-precision.

In order to avoid a maximum of system interrupts on the pipeline, it is launched on isolated CPU with RT priority.



(a)

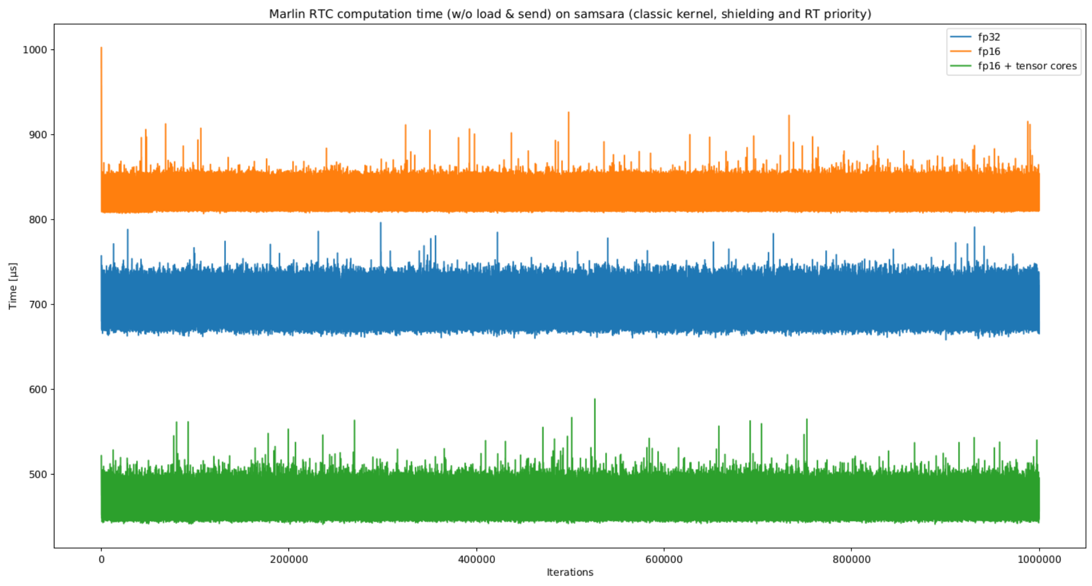


(b)

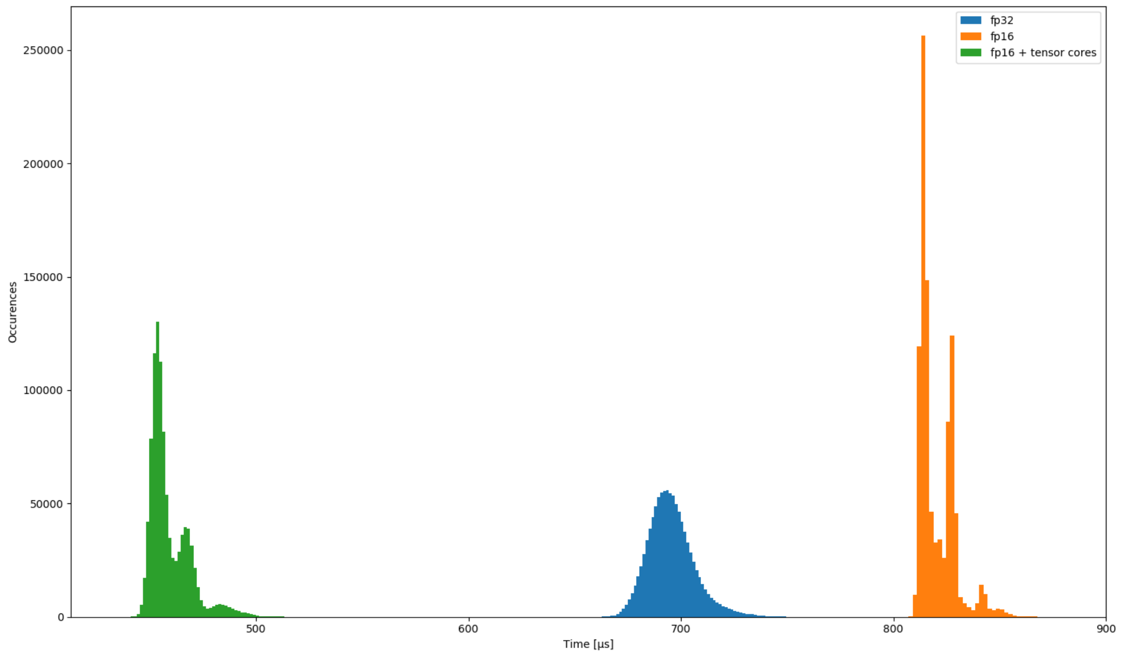
Figure 5. (a) Instantaneous Strehl ratio obtained from a simulation performing the MVM in single precision versus the same simulation in single precision. The red curve is y=x (b) Difference between the Strehl ratios obtained in single precision and half precision

Table 1. Benchmark statistical results in terms of mean latency, standard deviation (jitter RMS), and jitter peak-to-valley. Results are given in microseconds

	FP32	**FP16**	**FP16 + tensor cores**
Avg. latency	695	820	459
Jitter rms	11	8	9
Jitter P2V	134	196	147



(a)



(b)

Figure 6. (a) Execution time of the RTC at each iteration (b) Histograms of the execution times measured. In each figure, the blue curve tends for single-precision MVM, the orange one for half-precision MVM without using the tensor cores, and the green one for half-precision MVM with tensor cores

4.2 Results

Figure 6 shows the results from the benchmark described previously. Table 1 summarizes those results from a statistical point of view. Results are characterized in terms of average latency, jitter rms and jitter peak-to-valley (P2V). Jitter rms is defined as the standard deviation of the latency, and jitter P2V as the difference between the maximum and minimum values of the latency.

In single-precision, we note a latency of around 700 μs, which is the expected performance for the V100 GPU card measured with 750 GB/s of memory bandwidth as MVM operation is memory bound. A more surprising result is that half-precision led to higher latency than single precision if the tensor cores are not enabled. That could be explained by the fact we are using a matrix-matrix multiplication algorithm to perform the MVM, which is not optimal. Enabling tensor cores finally leads to 34 % latency reduction compared to single precision, without a significant impact on AO performance as shown in Section 3 and with equivalent jitter.

We can also notice from Fig. 6.(b) that the usage of half-precision seems to induce some secondary peaks in the histograms. This unexpected behavior still has to be investigated.

In terms of jitter, we can observe some peaks that seems to be regularly spaced in Fig. 6.(a). This could be induced by CPU interrupts: even if all the computation are made on the GPU, the CPU still has to launch the CUDA kernels to start the computation. A CPU interrupt that occurs between two kernel launches could lead to some jitter with an amplitude depending on the interrupt nature. Some additional tests have been made using a Linux real-time kernel without significant impact on the observed jitters.

Finally, even with the best performance obtained with the tensor cores, the performance still not fulfill the current MICADO MAORY SCAO RTC requirement of 285 μs. However, this statement should be tempered as the computation only started when the last pixel of the image was received. The requirement should be reached by overlapping computation with the pixels reception, which is feasible especially using a full-pixel algorithm.

5. CONCLUSION & PERSPECTIVES

We presented the software architecture of the core HRTC which is prototyped for MICADO MAORY SCAO RTC from the COSMIC platform. It is based on the OCEAN framework which includes libraries that allows core computation on hardware accelerator, inter-processes communication through shared memory, and management and control of the HRTC. The proposed architecture is modular, configurable and scalable to fit to many AO systems.

A proof of concept has been established by plugging the RTC to an end-to-end simulation and showing the expected AO correction. The impact of using half-precision floating point format has been also demonstrated to have not a significant impact on the AO performance if used to perform the matrix-vector multiplication in a classical MVM and integrator command law.

Finally, performance in terms of latency and jitter has been measured for a MICADO SCAO scale system, leading to an average latency of 459 μs using half-precision coupled to tensor cores. Even if this performance does not match the requirement of 285 μs, overlapping computation with the pixel reception should close the remaining gap.

Future work will explore the impact of having multiple business units running simultaneously on the performance, especially on the jitter. The overlap with pixel reception will also be tested. The prototyping phase will also lead to integration on bench with both a Shack-Hartmann and a pyramid WFS and a DM with around 3,200 actuators.

REFERENCES

[1] Davies, R., "Micado sci-ops: shaping its adaptive optics systems," in [*6th AO4ELT conference-Adaptive Optics for Extremely Large Telescopes*], (2019).

[2] Diolaiti, E., Ciliegi, P., Abicca, R., Agapito, G., Arcidiacono, C., Baruffolo, A., Bellazzini, M., Biliotti, V., Bonaglia, M., Bregoli, G., Briguglio, R., Brissaud, O., Busoni, L., Carbonaro, L., Carlotti, A., Cascone, E., Correia, J.-J., Cortecchia, F., Cosentino, G., De Caprio, V., de Pascale, M., De Rosa, A., Del Vecchio, C., Delboulbé, A., Di Rico, G., Esposito, S., Fantinel, D., Feautrier, P., Felini, C., Ferruzzi, D., Fini, L., Fiorentino, G., Foppiani, I., Ghigo, M., Giordano, C., Giro, E., Gluck, L., Hénault, F., Jocou, L., Kerber, F., La Penna, P., Lafrasse, S., Lauria, M., le Coarer, E., Le Louarn, M., Lombini, M., Magnard, Y., Maiorano, E., Mannucci, F., Mapelli, M., Marchetti, E., Maurel, D., Michaud, L., Morgante, G., Moulin, T., Oberti, S., Pareschi, G., Patti, M., Puglisi, A., Rabou, P., Ragazzoni, R., Ramsay, S., Riccardi, A., Ricciardi, S., Riva, M., Rochat, S., Roussel, F., Roux, A., Salasnich, B., Saracco, P., Schreiber, L., Spavone, M., Stadler, E., Sztefek, M.-H., Ventura, N., Vérinaud, C., Xompero, M., Fontana, A., and Zerbi, F. M., "MAORY: adaptive optics module for the E-ELT," in [*Adaptive Optics Systems V*], *Proc. SPIE* **9909**, 99092D (July 2016).

[3] Clénet, Y., Buey, T., Gendron, E., Hubert, Z., Vidal, F., Cohen, M., Chapron, F., Sevin, A., Fédou, P., Barbary, G., Borgo, B., and Davies, R., "MICADO MAORY SCAO: preliminary design, development plan and calibration strategies," in [*6th AO4ELT conference-Adaptive Optics for Extremely Large Telescopes*], (2019).

[4] Deo, V., Vidal, F., Gendron, E., Buey, T., Gratadour, D., Hubert, Z., Cohen, M., and Rousset, G., "Enlarging the control space of the pyramid wavefront sensor: Numerical simulations and bench validation," in [*5th AO4ELT conference-Adaptive Optics for Extremely Large Telescopes*], (2017).

[5] Gratadour, D., Dipper, N., Biasi, R., Deneux, H., Bernard, J., Brule, J., Dembet, R., Doucet, N., Ferreira, F., Gendron, E., Laine, M., Perret, D., Rousset, G., Sevin, A., Bitenc, U., Geng, D., Younger, E., Andrighettoni, M., Angerer, G., Patauner, C., Pescoller, D., Porta, F., Dufourcq, G., Flaischer, A., Leclere, J.-B., Nai, A., Palazzari, P., Pretet, D., and Rouaud, C., "Green FLASH: energy efficient real-time control for AO," in [*Adaptive Optics Systems V*], *Proc. SPIE* **9909**, 99094I (July 2016).

[6] Gratadour, D., Rigaut, F., Biasi, R., Jameson, A., Sevin, A., Ferreira, F., Perret, D., Guyon, O., Patauner, C., Pescoller, D., Andrighettoni, M., Smith, M., Glazebrook, K., and Deller, A., "The cosmic real-time control platform," in [*6th AO4ELT conference-Adaptive Optics for Extremely Large Telescopes*], (2019).

[7] Ferreira, F., Gratadour, D., Sevin, A., Doucet, N., Vidal, F., Deo, V., and Gendron, E., "Real-time end-to-end AO simulations at ELT scale on multiple GPUs with the COMPASS platform," in [*Proc. SPIE*], *Society of Photo-Optical Instrumentation Engineers (SPIE) Conference Series* **10703**, 1070347 (Jul 2018).

[8] Guyon, O., Sevin, A., Gratadour, D., Bernard, J., Ltaief, H., Sukkari, D., Cetre, S., Skaf, N., Lozi, J., Martinache, F., Clergeon, C., Norris, B., Wong, A., and Males, J., "The compute and control for adaptive optics (CACAO) real-time control software package," in [*Proc. SPIE*], *Society of Photo-Optical Instrumentation Engineers (SPIE) Conference Series* **10703**, 107031E (Jul 2018).