Quasi-real-time end-to-end adaptive optics simulations at the E-ELT scale

Damien Gratadour[1a], Arnaud Sevin[1], Eric Gendron[1], and Gerard Rousset[1]

Laboratoire d'Etudes Spatiales et d'Instrumentation en Astrophysique (LESIA), Observatoire de Paris, CNRS, UPMC, Universite Paris Diderot, 5 places Jules Janssen, 92195 Meudon France[c]

Abstract. Our team has started the development of a code dedicated to GPUs for the simulation of adaptive optics systems at the E-ELT scale. It uses the CUDA toolkit and an original binding to Yorick (an open source interpreted language) to provide the user with a comprehensive interface. In this paper we present the first performance analysis of our simulation code, showing its ability to provide Shack-Hartmann (SH) images and measurements at the kHz scale for VLT-sized AO system and in quasi-real-time (up to 70 Hz) for ELT-sized systems on a single top-end GPU. The simulation code includes multiple layers atmospheric turbulence generation, ray tracing through these layers, image formation at the focal plane of every sub-aperture of a SH sensor using either natural or laser guide stars and centroiding on these images using various algorithms. Turbulence is generated on-the-fly giving the ability to simulate hours of observations without the need of loading extremely large phase screens in the global memory. Because of its performance this code additionally provides the unique ability to test real-time controllers for future AO systems under nominal conditions.

1 Introduction

The simulation of an AO system involves multiple physics from atmospheric turbulence models to tomographic reconstruction to control theory. Due to the stochastic nature of the turbulence, Monte Carlo simulations provide the most realistic results, the extremely large diameter of an ELT (up to 39m in the case of the E-ELT) making each iteration a large scale problem. Moreover, the model of an AO system and its environment exhibits several levels of parallelism [2]. First, the model computation involves operations that are known to have a good degree of parallelism such as matrix multiplies, fast Fourier transform or ray-tracing. Additionally some sub-systems such as the WFS have a structure adequate for parallelization. Finally, advanced AO concepts involving several sensors and correctors for which the models can be computed in parallel provide another level of parallelism. Full length E-ELT simulations are thus compute-intensive applications and as such good candidates for considering the use hardware accelerators like manycore processors.

Graphical Processing Units (GPU) are massively parallel computing devices (up 512 collaborating scalar stream processors in the case of NVIDIA GPUs) which can deliver up to a Tflop at an affordable price. The CUDA hardware architecture [3] introduced in 2007 was designed to provide graphics processors equipped with HPC-compatible features (e.g. IEEE754-like floating point unit, memory model) offering a tremendous performance potential at a very low power. With the emergence of GPGPU (General Purpose GPU) computing it is now possible to run general code on GPUs with minimal effort. GPUs hence provide a cheap solution to build a massively parallel cluster for large scale computations.

Substantial efforts have been led by our team during the past year to develop a simulation code running on GPUs and able to target the E-ELT size and codenamed YoGA_AO. This approach is based on a C++ implementation which relies on the CUDA toolkit for core computations and makes an extensive use of the Standard Template Library so that important simulation parameters (such as the number of turbulent atmospheric layers) can be updated dynamically during a simulation run. Additionally, thanks to the use of an original binding to Yorick an open source interpreted language,

[c] Groupement d'Interet Scientifique PHASE (Partenariat Haute resolution Angulaire Sol Espace) between ONERA, Observatoire de Paris, CNRS, Universite Paris Diderot, IPAG and LAM

[a] damien.gratadour@obspm.fr

this code comes with a comprehensive user-friendly interface to easily test complex configurations and fine tune parameters.

The parallel versions of the physical models included in this simulation code have been validated against theory and a sequential version running on CPU. The code has been extensively tested with various system dimensioning to evaluate the performance on a single high-end GPU. It includes a multiple layers turbulence model and a Shack-Hartmann wavefront sensor model for natural and laser guide stars. It is highly configurable and allows to simulate a variety of systems from single conjugate NGS AO to multiple guide stars LGS AO.

In this paper, we describe the physical models implemented in this code and report on a preliminary performance analysis.

2 Atmospheric turbulence model

To avoid loading large phase screens in the global memory, turbulence can be generated on the fly by the extrusion of Kolmogorov-type phase screen ribbons. We used the algorithm of Assemat et al [1] further refined by Fried & Clark [4]. In this algorithm, the external scale in taken as infinite which explains why the structure functions in Figure 1 do not saturate for large distances. The core algorithm to update the phase screen mainly involves matrix-vector multiplies and random number generation and run on the GPU, the latter process being dominant in the execution time. Moreover, this algorithm requires the pre-computation of several matrices involving the singular value decomposition of a matrix as large as the phase screen. This is done on the CPU.

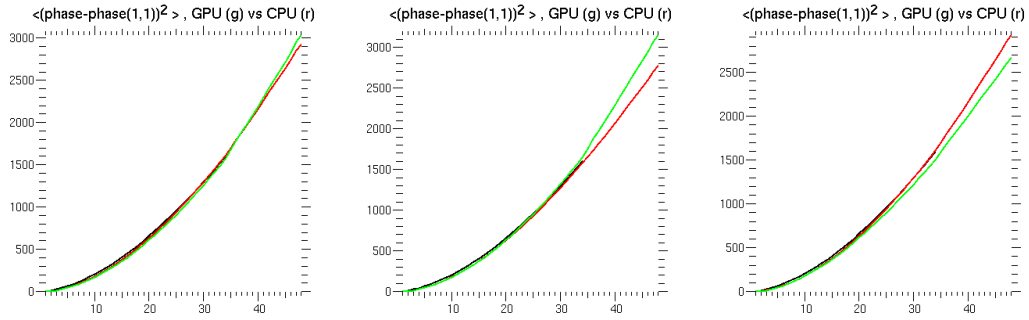

Fig. 1. Validation of the turbulence model against theory. The phase structure function is plotted against the distance in units of r_0. In red, results on the CPU for only one turbulent layer and in green on the GPU for (from left to right) 1, 4 and 12 turbulent layers. In black the theoretical $6.88(r/r_0)^5/3$.

Depending on the simulation parameters, several phase screens can be created at various altitudes, strengths, speeds and directions. The number of lines and columns being extruded per iteration to update the phase screen being determined by the last two parameters and the simulation iteration time. Typical screen speeds (in number of extrusion per iteration) are given in Figure 2 for two different profiles. Remember that the phase screens sizes depend on their altitudes and the directions of the various targets in the simulation. For high altitudes (> 10km) and wide separations (>1'), altitude phase screens can be significantly larger than ground layer phase screens.

In addition a custom raytracing algorithm has been added to compute the phase in various directions in the case of several phase screens. This algorithm takes full advantage of the GPU architecture and is very fast as shown by the central plot of Figure 2, turbulence generation time being dominated by the extrusion process. Two versions of this algorithm have been tested using different memory models available on a GPU, respectively texture memory and shared memory. Both versions lead to similar performance, with the difference that texture memory should be bound to global memory for execution, involving larger overall memory occupancy. Hence the shared memory model have been prefered.

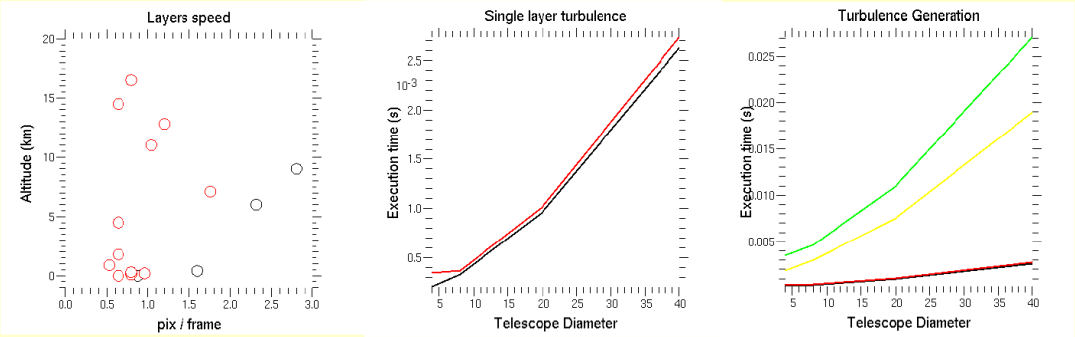

Fig. 2. Layers speeds for two profiles (4 and 12 layers) in number of extrusion per iteration and turbulence generation speeds for various profiles and various telescope diameters. Black : 1 layer, 1 target ; red : 1 layer, 5 targets ; yellow 4 layers, 5 targets ; green 12 layers, 5 targets (GPU used : Tesla C2070).

3 Wave front sensor model

The code also includes a Shack-Hartmann wavefront sensor model. The phase screen obtained for a particular direction is divided in various portions for every sub-aperture. Each portion is used to generate a small image for the considered sub-aperture. The model also includes a realistic approach for the generation of LGS images (as described in Gratadour et al. [5]). In this model, we assume that the upward effect of the atmosphere and beam aberrations and divergence on the beacon can be modeled using a Gaussian distribution of light at the Sodium layer. Then a one-dimensional version of the Sodium density profile (that can be obtained from measured profile or using a simple Gaussian distribution), properly stretched to match the off-axis perspective effect was convolved by this Gaussian beam shape to form an elongated beacon image. It is quite straightforward to demonstrate that the convolution of a one-dimensional profile by a two-dimensional circularly symmetric Gaussian can be first done in one dimension and then extended to the second dimension. This algorithm allowed us to implement a fast and reliable routine, that avoids any discretization or under-sampling issue, to build a high resolution image for any sub-aperture. The overall model has been parallelized and the LGS spot images are computed at each iteration so as to minimize to memory occupancy in the case of extremely elongated LGS. Indeed a full E-ELT simulation with LGS launched at the edge of the pupil can be run on a laptop GPU.

The rest of the model mainly involves fast Fourier transforms (FFT), random numbers generation and image binning that all benefit from a parallel implementation. While the individual FFTs are small, the code benefits greatly from an optimal dispatch of the workload using native batched FFTs in CUDA. Figure 3 outlines the main steps of the Shack-Hartmann model for various configurations.

An important parameter driving the WFS model dimensioning is the pixel size requested by the user for the final sub-apertures images. It is approximated following a rather robust approach to cope for any kind of dimensioning. We used an empirical coefficient to set the simulated sub-apertures field of view (FoV) to 6 times the ratio of the observing wavelength over r_0 at this wavelength. This provides sufficient FoV to include most of the turbulent speckles. The same empirical coefficient is used to define de number of phase points per subaps as 6 times the ratio of the subaps diameter over r_0. This ensures a proper sampling of r_0. From this number of phase points we compute the size of the support in the Fourier domain. The ”quantum pixel size” is then deduced from the ratio of the wavelength over r_0 over the size of the Fourier support. Then the pixel size actually simulated is obtained using the product of an integer number by this quantum pixel size as close as possible to the requested pixel size. If the requested FoV is larger than the default simulated FoV or in the case of extremely elongated LGS spot, the PSF images computed from the phase screen portions are immersed in a larger support before binning.

The simulation has been tested with various system dimensioning to evaluate the performance. For a given telescope size we consider two system dimensionings: classical AO (or single conjugate AO, i.e. SCAO) for which the sub-apertures have a diameter of 50cm (case of the NaCO instrument

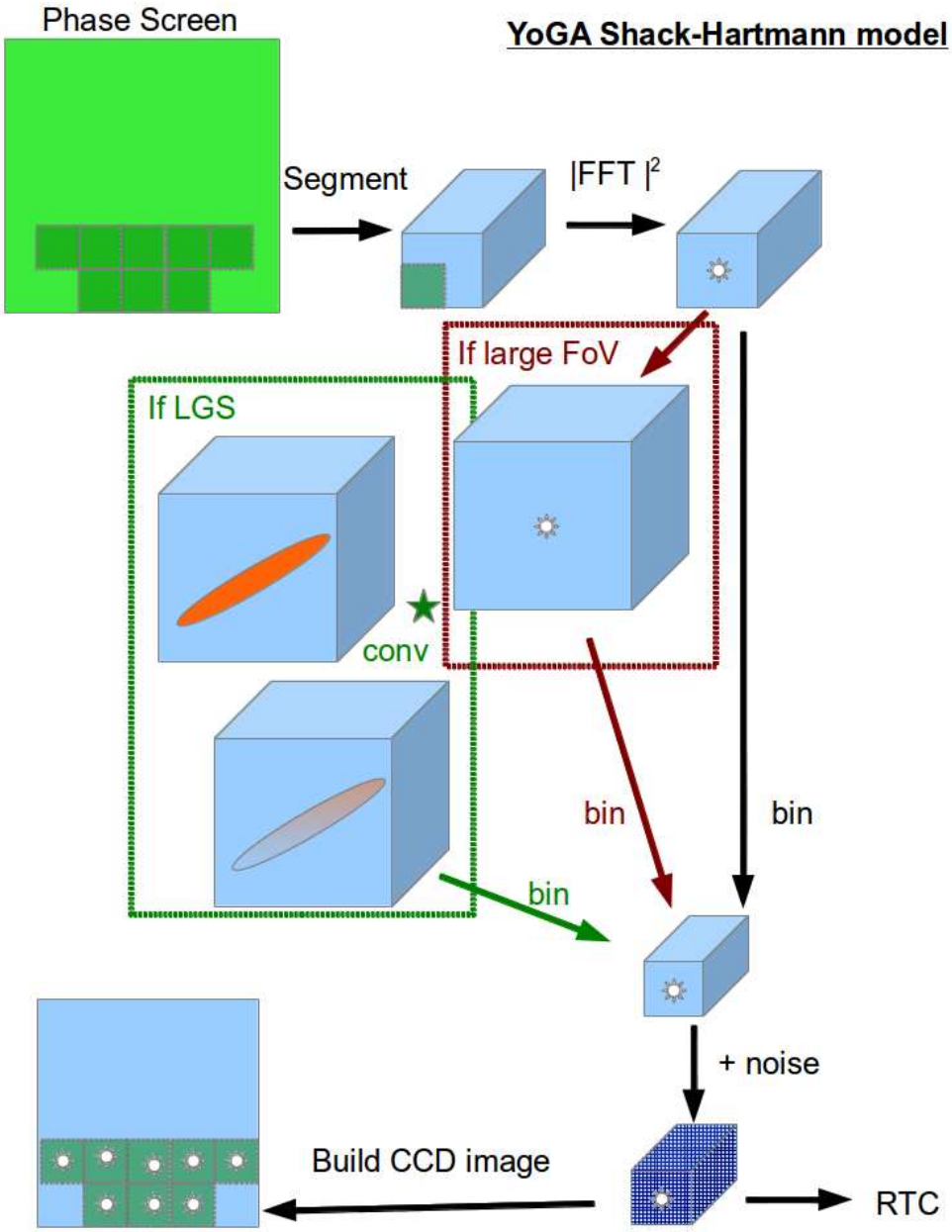

Fig. 3. Schematic representation of the main steps of the Shack-Hartmann model implementation including large FoV and extended LGS cases.

on the VLT) and extreme AO (XAO) for which the sub-apertures have a diameter of 20cm (case of the SPHERE instrument on the VLT). Because there are 4 to 5 times more sub-apertures in the case of XAO, a sequential simulation code will give poorer performance for this case. However, because as previously said the models for each sub-aperture of the WFS can be computed independantly, a

parallel code should perform better because the number of phase points per sub-apertures is reduced in the case of XAO, the physical size of the sub-apertures being reduced (50cm -¿ 20cm). This leads to better properties in terms of GPU internal memory management. Because of the reduced number of phase pixels per sub-apertures, the spot computation can be done entirely in the local memory of a block of threads which minimize the access to global memory hence the latency in the computation. Moreover this gain is multiplied by the number of subaps which explains why the speedup is increasing with telescope size. This is what is shown in figure 4, where the execution time of a SCAO simulation is plotted against other AO concepts: XAO, LGS-AO and MOAO.

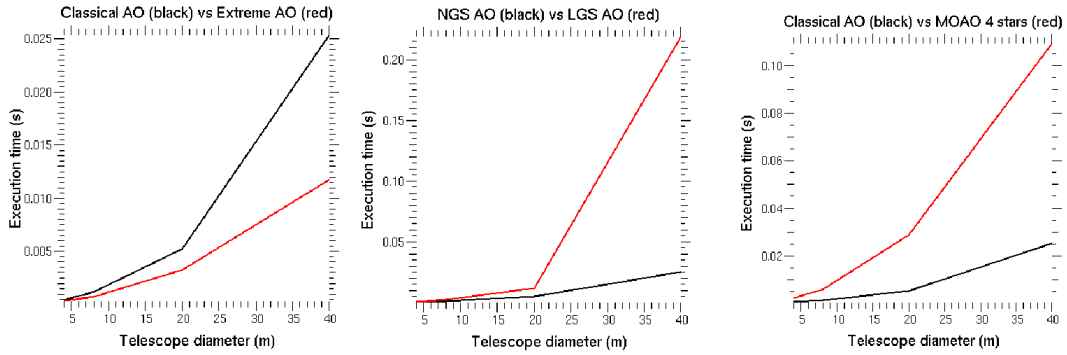

Fig. 4. Performance comparison between various AO concepts: classical AO (subaps = 50cm, 1 atmospheric layer), extreme AO (left, subaps = 20cm, 1 atmospheric layer), LGS AO (center, subaps = 50cm, 1 atmospheric layer) and MOAO (right, subaps = 50cm 4 WFS, 4 atmospheric layers) for various telescope sizes.

In the case of LGS-AO however, the sub-apertures FoV is larger and increases with the laser spot elongation hence the telescope diameter. In this case, we are unable to fit the whole computation for one sub-aperture in the GPU shared memory which explains why no significant gain is obtained as compared to SCAO. Finally, in the case of MOAO, because there are several WFS treated sequentially in the actual code, there is no gain as compared to the SCAO run.

4 Centroiding

For the computation of the WFS model, the individual sub-aperture images are arranged in a 3D structure to optimize the batched-FFT computation. As shown in figure 3, after the noise has been added, images can be either re-arranged into a 2D frame as it would be if obtained on a real system or stay in a 3D structure to be sent to the RTC module of YoGA_AO keeping an optimized arrangement of the data for better performance. The RTC module itself has been designed to be compatible with both arrangements to provide a generic real-time core for AO.

For now, the real-time core only performs centroiding on the images. Several centroiding algorithms, considered as the baseline for the E-ELT AO modules design, have been implemented: classical center of gravity (COG), weighted COG, brightest pixels COG, thresholded COG and correlation. As in the case of raytracing, these algorithms take the full advantage of the GPU architecture and can auto-tune for a specific GPU, taking into account the number of stream processors and the caracteristics of memory. Their contribution to the overall wavefront sensing time is minimal as shown in Figure 5.

Simulation speeds of few 1000 iterations/s are reached for classical AO systems on a 8m telescope: this is faster than real-time control alone. Additionally about a 100 iterations/s are reached for classical AO systems at the E-ELT scale: this is realistic enough to start full scale design studies. Moreover, the performance reached for more advanced AO concepts show that the code takes full advantage of the GPU architecture for core computations.

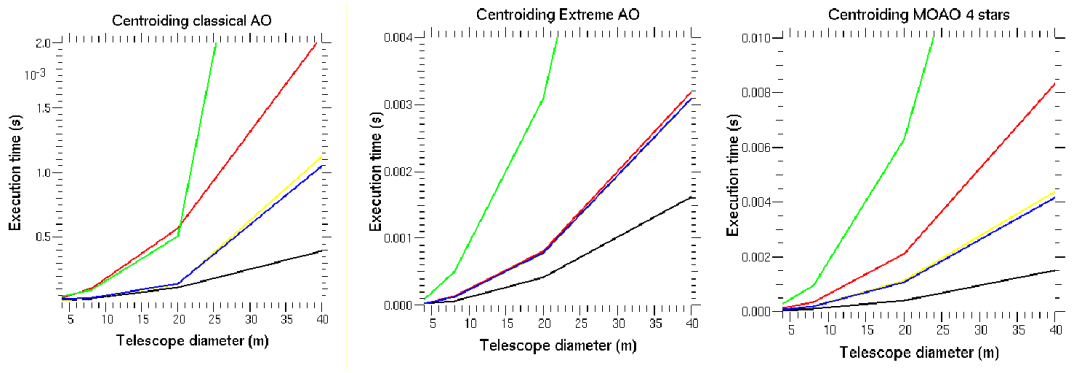

Fig. 5. Performance comparison between various centroiding methods (phase difference square pupil (black) phase difference real pupil (red), center of gravity (yellow), brightest pixels COG (green) and thresholded COG (blue) for various AO concepts classical AO (left), extreme AO (center) and MOAO (right) for various telescope sizes. The brightest pixels (in this case 10 pix) algorithm is the most expensive in each case due to its complexity (requires several levels of parallel reduction). Thesholded COG has similar performance than standard COG.

5 User-friendly interface

Complex systems simulations always benefit from the use of an interpreted language that provides a comprehensive interface to design and use the code with adequat tools for diagnosis and debug. Yorick ([9]) is an interpreted programming language for scientific simulations or computations. It is written in ANSI-C and runs on most OS. It has a compact C-like syntax with array operators and extensive graphics possibilities. It is easily expandable through dynamic linking of C libraries. Many plugins / extensions dedicated to adaptive optics and image processing are available (see for instance [10]). Our team has developped an original binding between CUDA and Yorick (codename YoGA). This binding allows to manipulate persistent objects on the GPU from within a Yorick session and thus optimally launch intensive computations on these objects through an interpreted environment. YoGA ([6]) is an open source project and has a BSD licence. It relies on a double-sided implementation with a Yorick specific API and a high level C++ API containing default classes aimed at residing on the GPU memory. YoGA_AO relies on the YoGA binding for the interaction with the user and the specific YoGA_AO classes are built from the generic YoGA objects so that important aspects such as memory management are dealt with transparently to the user. A screenshot of the YoGA_AO graphical user interface is provided in figure 6.

6 Conclusion and future work

The simulation code now lacks control algorithms as well as a deformable mirror model to get a full end-to-end tool. However the performance achieved up to now are very encouraging. They demonstrate that core algorithms (for instance centroiding) can be implemented very efficiently with a favorable scale factor when seeking for ELT conditions. Moreover, depending on the system dimensioning, the implementation can benefit greatly from massive parallelism as demonstrated by the XAO performance.

The preliminary performance study led with YoGA_AO and MAOS [11] have shown that a single currently available high end GPU could provide sufficient computing power to drive a classical AO system on the E-ELT. This is still an order of magnitude away from the power needed to perform tomographic reconstruction required for both E-ELT AO modules in real-time but general trends in GPU technology indicate that this kind of throughput should be achieved by the end of this decade (see for instance NVIDIA's Echelon project which targets a factor of 10 improvement in computing power and memory bandwidth [8]). GPUs are thus extremely appealing candidates for both simulating AO systems and accelerating the next generation AO real-time controllers.

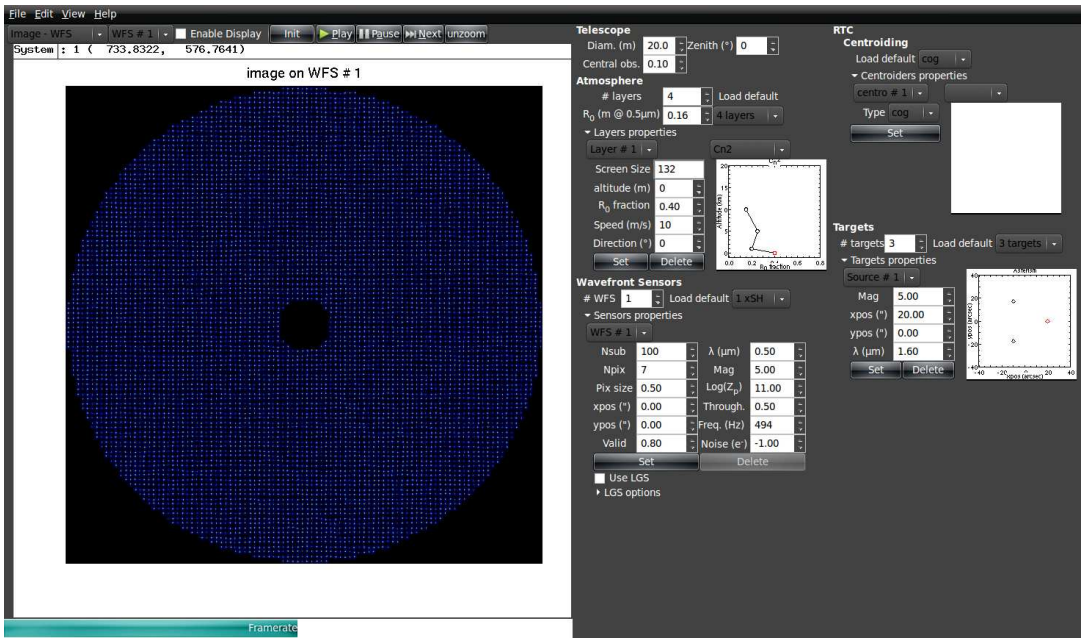

Fig. 6. A screenshot of the YoGA_AO user interface, runing an XAO system on a 20m telescope.

References

1. Francois Assemat et al., Optics Express, **Vol. 14**, (2007), pp. 988-999
2. A. Basden et al., Applied Optics, *Vol. 46*, *Issue 7* (2007) pp. 1089-1098
3. Jason Sanders & Edward Kandrot, *CUDA by example: An introduction to general purpose GPU programming* (Addison-Wesley Professional, Edition 1 2010)
4. David L. Fried & Tim Clark, JOSA A, **Vol. 25, Issue 2**, (2008), pp.463-468
5. Damien Gratadour et al., JOSA A, **Vol. 27, Issue 11**, (2010), pp.A171-A181
6. Damien Gratadour & Arnaud Sevin, https://github.com/yorick-yoga/yorick-yoga/wiki
7. Damien Gratadour, http://github.com/dgratadour/yoga_ao/wiki
8. S. Keckler et al., Micro IEEE, **Vol. 31, Issue 5**, (2011), pp.7-17
9. Dave Munroe, http://yorick.sourceforge.net
10. Francois Rigaut, http://www.maumae.net/yao/index.html
11. Lianqi Wang, **Analysis of the Improvement in Sky coverage for TMT NFIRAOS**, this conference