UCLA
Adaptive Optics for Extremely Large Telescopes 4 – Conference Proceedings

Title
Durham AO Real-time Controller (DARC) running on Graphics Processing Units (GPUs)

Permalink
https://escholarship.org/uc/item/6809n74d

Journal
Adaptive Optics for Extremely Large Telescopes 4 – Conference Proceedings, 1(1)

Authors
Bitenc, Urban
Basden, Alastair
Dipper, Nigel
et al.

Publication Date
2015

DOI
10.20353/K3T4CP1131620

Copyright Information
Copyright 2015 by the author(s). All rights reserved unless otherwise indicated. Contact the author(s) for any necessary permissions. Learn more at https://escholarship.org/terms

Peer reviewed

eScholarship.org Powered by the California Digital Library
University of California

Durham AO Real-time Controller (DARC) running on Graphics Processing Units (GPUs)

Urban Bitenc, Alastair G. Basden, Nigel A. Dipper, Richard M. Myers

Centre for Advanced Instrumentation, Physics Department, Durham University, UK

ABSTRACT

The requirements on the real-time control systems for ELT instruments strongly encourage an investigation of newly emerging hardware and an assessment of its suitability for the job. We have implemented the full AO data processing pipeline on Graphics Processing Units (GPUs), within the framework of Durham AO Real-time Controller (DARC). The pixel data are copied from the CPU memory to the GPU memory. On the GPU, the data are processed and the DM commands are copied back to the CPU. For a system of 80x80 subapertures, the highest rate achieved on a single GPU is 550 frames per second. When running on two or more GPUs, the kernel launching time limits the increase in frame rate. We have also implemented the correlation centroiding algorithm, which - when used - reduces the frame rate by about a factor of two.

Keywords: AO - Adaptive Optics, ELT - Extremely Large Telescope, RTC - Real-Time Control, GPU - Graphics Processing Unit

1. INTRODUCTION

In order to reach their true potential in resolution, the next generation of extremely large telescopes (ELT) will depend on the technique of adaptive optics (AO). Whilst the technique is now well established on 8 and 10 metre class telescopes, its extension to 30 to 40 metre telescopes remains a significant challenge. One major aspect of that challenge is the provision of a suitable low latency real-time control system (RTC) that can provide the large amounts of processing power required for the high order AO systems that are being designed.

We have been investigating the technology and methods that should be used to provide an RTC for the European Extremely Large Telescope (E-ELT). This research program is being implemented in partnership with the European Southern Observatory (ESO) and with other collaborating institutions both in the UK and throughout Europe. Our investigation of RTC techniques is a part of an overall program of demonstrating new AO techniques both in the laboratory and on sky. To this end, we have developed two AO demonstrator systems. CANARY[7] is a fully operational AO experiment that has provided the first demonstration of tomographic multi-object AO (MOAO) on sky[1] on the 4 metre William Herschel Telescope on La Palma. The DRAGON test bench[6] is an experiment designed to demonstrate new techniques in AO in the laboratory. Both of these systems require a low latency RTC and this requirement has led to the development of the Durham AO Real-time Controller, DARC.[2] With a modular design, and an architecture well suited to multi-core and many-core processors, DARC is an ideal software platform for testing new algorithms and hardware acceleration, making it well suited to addressing the ELT-scale AO RTC problem.[3]

CANARY was originally a low order system (7x7 sub-apertures) and was extended to higher order (17x17 sub-apertures) in 2015. DRAGON operates with 31x31 sub-apertures. At these orders, DARC can meet the performance requirements of both systems using multiple multi-core processors in a server class CPU based system. At the higher orders required by ELT scale instruments, it cannot. We have previously demonstrated the feasibility of both the large matrix-vector multiplies (MVM) and large matrix inversions required for high order AO instrumentation.[3,5] In order to demonstrate an RTC at ELT scales in DRAGON, we have ported the DARC software to NVidia GPUs to accelerate all stages of the data pipeline. It is initial results from this full DARC implementation on GPU that are reported here.

Further author information:
U.B.: E-mail: urban.bitenc@durham.ac.uk

Until very recently, telescope AO system RTCs only made use of a MVM for the reconstruction of the wave-front. As system sizes increase, the number of operations required for the other stages of the data pipeline, pixel calibration and Shack-Hartmann (SH) centroiding, scale linearly with the size of the system, $\mathcal{O}(N)$. Here N is the number of subapertures which is proportional to the square of the telescope diameter. The MVM however scales as $\mathcal{O}(N^2)$. Thus, for the high order AO systems proposed for the E-ELT (typically 80x80 sub-apertures), the wave-front reconstruction stage dominates in computational power. This has led to two different approaches to the design of these systems. The first simply entails speeding up the MVM using new massively parallel computer technology. This is the approach described in this paper. The second involves replacing the MVM with a more sophisticated algorithm that scales as $\mathcal{O}(N)$ or $\mathcal{O}(N * \ln N)$. Several such different algorithms have been proposed and tested on-sky, including the Fourier Transform Reconstructor[10] and CuReD algorithm.[9] Whilst this algorithmic approach is valuable and will be required for the highest order AO systems, the MVM approach is much more widely tested. As we will show, most E-ELT requirements can be met using this method and the application of new acceleration technologies such as GPUs. It is the implementation of a full ELT scale RTC on GPU technology that is the subject of this paper.

There are several competing technologies for the acceleration of the wavefront-reconstruction process. Several of these have been applied to existing AO RTC systems. The development of the technology that is required for our application is driven by the requirements of the High Power Computing (HPC) sector. Since the release by NVidia of the CUDA development environment, NVidia GPUs have been a popular technology for the acceleration of AO systems. The PALM 3000 system, on the 5 metre telescope at Mount Palomar, makes use of multiple GPUs on a system that, in its highest order mode, uses 64x64 sub-apertures. It is thus not unreasonable to test this technology at ELT scales. However, we recognize that other existing and emergent technologies may also contribute to the eventual RTC for the E-ELT, which is likely to be a hybrid system. This hybrid approach was successfully used to define the existing RTC for AO systems on the Very Large Telescope (VLT). This system, SPARTA, made use of field programmable gate arrays (FPGA) for pixel calibration and centroiding, whilst digital signal processors were used for wave-front reconstruction. Whilst DSPs have now been largely supplanted by GPUs, it is likely that FPGAs still have a role to play in pixel data handling. Indeed, the CANARY experiment already has an option for pixel processing in FPGA.

The major advantage of GPUs over FPGAs is their ease of programming and the resultant faster development cycle. This is likely to be an even greater advantage in the use of the emerging technology of many integrated core (MIC) systems such as the Xeon Phi, which we are also investigating. These competing technologies emphasise the importance of portability of code. Software developed under CUDA can only operate on NVidia GPU devices. It is highly desirable that code for the E-ELT be written in a language that renders it portable between existing technologies and, more significantly, to new emerging technologies. The results reported here are based on CUDA. We are however investigating the implementation of DARC in OpenCL and its porting to systems using GPUs from other manufacturers, MIC devices and FPGAs.

2. THE DATA PROCESSING PIPELINE

We have implemented the full AO data processing pipeline on GPUs, within the framework of Durham AO Real-time Controller (DARC).

2.1 Durham AO Real-time Controller (DARC)

DARC[2,3] is a flexible, modular CPU-based AO real-time control system, that is primarily CPU-based, but can have optional hardware acceleration modules. It was first developed for use with the CANARY on-sky AO demonstrator instrument,[1] and has since seen use with several other instruments worldwide. DARC aims for computational efficiency using a horizontal processing strategy, where all processing threads perform similar tasks to optimize load balancing, rather than a more conventional strategy where some threads will perform calibration, some slope computation, and some wavefront reconstruction. A horizontal strategy leads to a significant reduction in inter-thread communication requirements. The achieved measured performance of DARC make it suitable for ELT use with appropriate computational hardware. Key flexibility is provided by the modular design, with dynamic loading and unloading of modules allowing development and testing of new algorithms without a system restart. This therefore makes DARC highly suited to operation in laboratory environments, where continued system development is often necessary.

2.2 Key components of the data processing pipeline

To implement the data processing pipeline, we implemented three modules of DARC to run on GPUs and the rest of DARC remained unchanged. These three modules are: pixel calibration, centroiding and reconstruction. Apart from copying the pixel data from CPU to GPU and copying the DM commands from GPU to the CPU, these three modules only launch kernels on each cycle to process data on GPU. (A "kernel" is the GPU equivalent to a "function"; launching a kernel on GPU is similar to calling a function on a CPU.)

For the majority of our tests, the pixel data were read in from a file when starting the application and then the same pixel data were used on every iteration. However, we also performed a test with real pixel data from a 10G ethernet camera, EVT-20007, and the behavior was very similar.

Table 1 lists the key steps performed on every cycle.

Table 1. The key steps of the data processing pipeline. The CPU prepares pixel data, launches the data copy commands and GPU kernels, and finalizes the DM commands. The GPU copies the data and executes the kernels.

	CPU each thread	CPU one thread only	GPU each stream	GPU default stream
1	copy pixels used to a buffer			
	launch:			
2	- copy pixels to GPU			
3	- pixel calibration			
4	- centroiding			
5	- MVM			
		launch:		
6		- sum up DM commands		
7		- copy DM com. to CPU		
8		- event "copied OK"		
9			copy pixels to GPU	
10			perform calibration	
11			perform centroiding	
12			perform MVM	
13				sum up DM commands
14				copy DM com. to CPU
15				record "copied OK"
16		add partial DM com. to final DM com.		

Step 1: Copy the pixels from the illuminated subapertures to a separate buffer (and convert them to float32 at the same time). The memory for this buffer was allocated using cudaMallocHost(), which results in the so called "pinned" memory being used. This enables one to start processing a part of the pixel data on CPU, while other pixel data are still being copied from CPU to GPU. This is done by dividing the whole pixel frame into several blocks of subapertures; each block is controlled by one CPU thread and processed by one cuda stream. As soon as the data of the first block is copied to the GPU memory, the GPU starts processing it, while the data of other blocks are still being copied. The larger the number of subaperture blocks, the more the data copying is shadowed by processing the data in parallel.

Steps 2-5: each CPU thread launches the copy command and kernel calls into the corresponding cuda stream.

Steps 6-8: one of the CPU threads launches the "sum-up" kernel and the copy-back-to-CPU command to the default cuda stream.

Steps 9-12: each cuda stream copies its data to GPU and processes it.

Steps 13-15: When the last cuda stream has finished processing its data, the default stream sums up all the partial DM commands on this GPU and copies them to the CPU.

Step 16: When the CPU gets a notification that the (partial) DM commands from a GPU have been copied to CPU, it adds these commands to the final DM command array. (When using one GPU, the DM commands from the GPU are already the final ones.)

Copying pixel data to GPU in step 9 runs to a large extent in parallel to pixel processing on GPU (steps 10-12). This is implemented by using cuda streams. Each stream processes a block of subapertures, from copying the pixel data to GPU (step 9) to performing its part of the MVM (step 12). After all the data of the first stream has been copied to GPU, the first stream starts processing its data while other streams are copying their data to GPU.

For centroiding, we implemented a center-of-gravity algorithm on GPU.

3. BENCHMARKING

At different stages during the course of this project, this software was benchmarked on 5 different GPU servers and used a number of different GPUs. Most of the results were obtained using GeForce GTX 580 and Tesla C2070 GPUs, but we also used Quadro 600, GTX 780 Ti, Tesla K20Xm and Tesla K40.

3.1 Simulated system

The main properties of the simulated system used for benchmarking are given in Table 2.

Table 2. The size of the simulated system used for benchmarking.

subaperture grid	80 x 80
number of illuminated subapertures	4624
subaperture size (in pixels)	16 x 16
number of controlled actuators	4816
MVM size	9248 x 4828

3.2 Optimizing the parameters

We identified several parameters that can be tuned to achieve the optimal average frame rate:

- The number of CPU threads (which is equal to the number of subaperture blocks and CUDA streams).
- The number of CUDA threads per block when executing the MVM kernel. Generally this should be a multiple of 64, but the optimal value depends on the number of GPUs used and the number of CPU threads.
- Include or exclude a call to "cudaStreamSynchronize()" before starting launching the kernels for calibration (in each CPU thread). Although it should have no effect, on Fermi GPUs this speeds up the computation (perhaps due to a bug or un-understood feature in cuda?), while on Keplers it has no effect or even slows down the computation, as expected.
- Which CPU threads run on which CPU cores and which GPUs are connected to those cores.
- The MVM kernel: the extent of unrolling the loop
- The MVM kernel: copy the array of slopes to the GPU shared memory or use them from global memory?
- linux kernel used: *generic*, *lowlatency*
- linux log-on method: as *root* or as a non-priviliged user

The task that takes longest is MVM. The parameters that most influence the running time are the number of CPU threads and the number or cuda threads per block in the MVM kernel. For each case we optimized these two numbers and we report the highest frame rate achieved.

3.3 Result

The highest average frame rate achieved was 550 fps, using GeForce GTX 580. All other GPUs used were slower by at least 10%, even though some of them are a lot newer and a lot more expensive.

As for jitter, examples of distributions of execution times per cycle are shown in Fig.1. The distribution of times taken by each cycle at the maximum frame rate, has a root-mean square value of 100-200 μs, but exhibits very few or no high-value outliers. Jitter differs between different servers and is sometimes difficult to understand. Generally we got best results using *lowlatency* linux kernel. On one of the servers this gave us a distribution where there was not a single outlier in 9.000.000 cycles (5 hours running at 500 fps), but on another server there was a small number of outliers, using the same linux kernel.

The study of jitter is not finalized yet. We need to understand the main reason for the width of the distributions in Fig.1 and try to reduce it. We think that at least a part of the width comes from the fact that the subaperture blocks are processed in a random order, rather than in a deterministic order.

4. LESSONS LEARNED

The main conclusions of our work are the following.

4.1 Copying pixel data to GPU

The main concern with the use of GPUs (or other accelerators) is the additional step of copying the data from the CPU memory to the GPU memory. With the copying rate of about 5 GB/s, copying the pixel data to GPU takes about 1 ms, which would cause a significant increase in latency. However, by using cuda streams and processing the data on GPU in parallel to copying data from CPU to GPU, we manage to reduce the overhead of copying the data by 80% - 90%.

To demonstrate this, we performed the following test. We do not copy any data from CPU to GPU and make no other changes to the code; the GPUs are then processing pixel data which are initialized with 0. This is similar to the scenario where the pixel data would be brought from the camera into to GPU memory directly, without involving the CPU memory. The average frame rate increased from 550 fps to 660 fps for GTX 580 (a 20% increase), and from 420 fps to 480 fps for Tesla C2070 (a 15% increase).

We conclude that using one GPU, copying the data from the CPU memory to the GPU memory may not a big issue. However, before making a final claim on that, we need to understand better the jitter.

4.2 Parameter tuning

There are several parameters that can be tuned in order to achieve highest frame rate. The optimal values depend on the GPU type, on the number of GPUs used and on the architecture of the server hosting the GPUs.

4.3 Using more than one GPU

When using more than one GPU, the frame rate gains are considerably lower than anticipated. Using two GPUs instead of one, the maximum possible gain is 100%; in reality one would expect an 85% - 90% increase. In contrast to this expectation, the frame rate increases by 50% only. Using three GPUs, the frame rate increases by 70% compared to the frame rate obtained with one GPU, whereas one would realistically expect a gain of 170% (and in an ideal case, a 200% gain).

We investigated two possible reasons for this: synchronization at the end of each cycle and the impact of the kernel launching time.

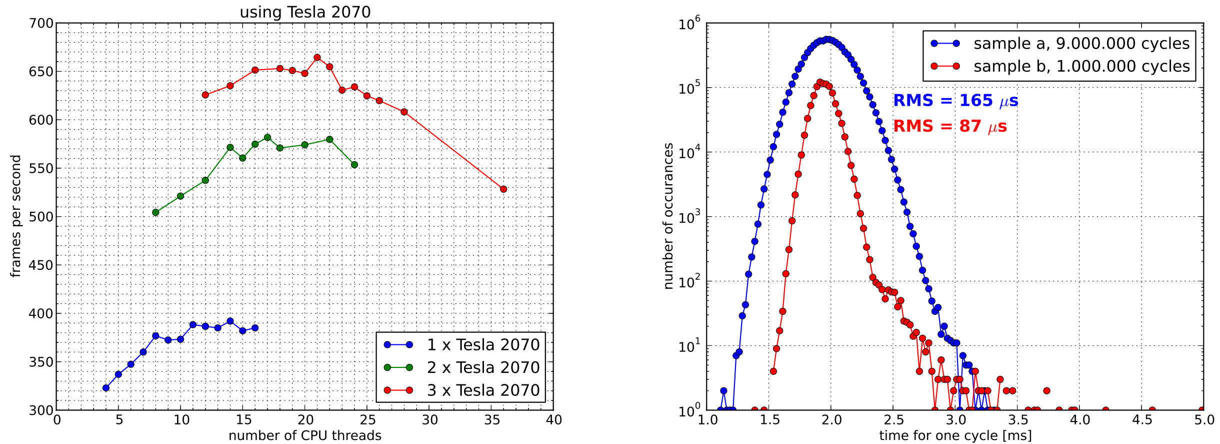

Figure 1. **Left:** Illustration of parameter tuning: frame rate against the number of CPU threads for one, two and three Tesla 2070 GPUs. One can clearly see a lower-than-expected gain in frame rate, when using two or three GPUs, compared to one. The values shown were improved later by about 5%-10%, but the full parameter tuning has not been repeated yet. **Right:** Jitter obtained with the linux low latency kernel, but on different servers and using different GPUs. The points above 3.5 μs all belong to sample b.

4.3.1 Synchronization at the end of cycle

When two or more GPUs are used, it is very important that:

(a) the computational load is spread evenly among all the GPUs used,

(b) after the last GPU has completed, the CPU finalizes the computation of the DM commands with a minimal delay.

While point (a) was trivial to satisfy, point (b) required a lot more attention. The challenge is that after step 8 is completed, the CPU waits for each of the GPUs to complete, before it can finalize the computation of DM commands (step 16). When using a single GPU, the final synchronization with GPU is automatic if using the function "cudaMemcpy()" to copy the DM commands from GPU to CPU, but if using several GPUs this would lead to unnecessary delays.

In the case of several GPUs it is advantageous to copy partial DM commands from GPU to CPU using function "cudaMemcpyAsync()", but in this case the developer must take care of the synchronization himself. We have investigated four options:

(a) "cudaDeviceSynchronize()"

(b) "cudaStreamSynchronize()"

(c) "cudaEventSynchronize()"

(d) "cudaEventQuery()"

The last two are the fastest. The only difference between (c) and (d) is due to the fact that a single CPU thread is doing the finalization for several GPUs, therefore a for-loop is needed. If GPU2 finalizes the computation before GPU1, the CPU can not proceed with the finalization for GPU2, because it is in a for-loop and is waiting for GPU1 to finalize first. After it has done the finalization for GPU1, it can proceed to the finalization of GPU2, leading to an additional delay of about 10 μs (per each additional GPU), and additional jitter, if the order is not deterministic.

Method (d) polls all GPUs in a for-loop, checking if any of the GPUs are finished. The polling typically last 2 μs per GPU. As soon as one GPU is found finished, its DM commands will be finalized, independently of the order in which the GPUs finish their work. We checked that the polling does not decrease the frame rate.

However, even with this carefully optimized synchronization the speed-up when using multiple GPUs was not significantly improved.

4.3.2 Kernel launching time

An investigation with NVidia Visual Profiler (nvvp) reveals that the usual kernel launching time on CPU is between 10 μs and 35 μs. In total, the steps 2-5 usually take between 100 μs and 150 μs. However, when running in several threads in parallel, the nvvp revealed that the kernel launching time in some cases increases significantly and takes hundreds of μs. We therefore introduced a mutex enforcing that, while all threads are performing step 1 in parallel, only one thread at a time is performing steps 2-5. This increased the frame rate by a few percent.

We performed a test in which we in the configuration file excluded half of the illuminated subapertures. The resulting frame rate was about 1.7 times higher, rather than 2.0. Excluding two thirds of the illuminated subapertures, the achieved frame rate was 2.3 time higher, rather than 3.0 times. This supports the understanding, that the missing improvement is due to the time spent for launching the kernels.

Next we made use of "dynamic parallelism". This feature was introduced in cuda version 5.0 and enables kernels running on a GPU to launch new kernels. We launched one kernel from CPU and that kernel then launched the calibration, cog and MVM kernels (steps 3-5) on the GPU. The rationale was that if CPU spends too long launching three kernels, launching only one kernel would be an improvement. However, the nvvp revealed that in this case, launching the one kernel took about the same amount of time as launching the three kernels separately. Also measuring the frame rate, we did not find any improvement.

We conclude that when using more than one GPU, the gain is significantly lower than expected due to the kernel launching time. This time becomes comparable with the kernel execution time, because the data has been split to such small chunks. If running on a single GPU, the optimal number of CPU threads is 8-12. It takes about 1 ms to launch the GPU kernels for 10 threads. With two or three GPUs we get in a situation, where the CPU does not manage to launch the kernels fast enough to keep the GPU occupancy high. The reason for launching 8-12 threads per GPU is that in this way the data copying time gets used for data processing. If the data was brought to the GPU memory directly (not through the CPU memory), a lower number of threads would perhaps be sufficient. This needs further investigation.

4.4 Advanced algorithms

We implemented correlation wavefront sensing, which is advantageous for use with laser guide stars due to the spot elongation. For this we insert the calculation of the correlation pattern between the pixel calibration and the (center-of-gravity) centroiding algorithm. The calculation of the correlation pattern consists of the following steps: (a) zero padding subapertures (from 16x16 to e.g. 32x32), (b) Fourier transform of the subapertures, (c) complex multiplication with the reference, (d) inverse Fourier transform, (e) clipping zeros (to speed-up the center-of-gravity calculation). For steps (b) and (d), we used the library "cufft", whereas for (a), (c) and (e) we wrote the cuda kernels required.

Using the correlation, the average frame rate was about two times lower than without correlation. This is similar to the slow down of the CPU version of DARC.

5. CONCLUSIONS

We have implemented the entire real-time control data pipeline on Graphics Processing Units, using DARC. For a simulated system of 80x80 subapertures, with dummy pixel data from a file, the maximum average frame rate achieved is 550 frames per second, using GTX 580. With real pixel data from a camera we achieved a similar frame rate. The overhead of copying the pixel data from GPU to CPU was largely shadowed by processing the data in parallel to copying. If using a single GPU, the benefit of copying the pixel data to the GPU memory directly from the camera (by-passing the CPU memory) would be of the order of 15% - 20%.

The distribution of times taken by each cycle at the maximum frame rate, has a root-mean square value of 100-200 μs, but exhibits very few or no high-value outliers.

When running on more than one GPU, the gain in frame rate is considerably lower than expected. We believe the reason for this is that the kernel launching time that becomes comparable to the kernel execution time, as the data is divided into smaller and smaller parts. Hence if one needs to run the AO correction at rates higher than about 500 frames per second, it would perhaps be beneficial to copy the pixel data from the camera directly to the GPU memory, bypassing the CPU memory.

We have also implemented the correlation wavefront sensing and observe a slow-down of about a factor of two, compared to the standard centroiding.

Acknowledgments

In UK this research project is supported by Science and Technology Facilities Council. We appreciate useful discussions with Sofia Dimoudi.

REFERENCES

[1] E. Gendron *et al.*: *MOAO first on-sky demonstration with CANARY,* Astronomy & Astrophysics **529**, L2 (Mar 2011)

[2] A. Basden, D. Geng, R. Myers and E. Younger: *Durham adaptive optics real-time controller,* Applied Optics **49**(32), 6354-6363 (Nov 2010)

[3] A.G. Basden and R. Myers: *The Durham adaptive optics real-time controller: capability and Extremely Large Telescope suitability,* MNRAS **424**, 1483-1494 (Aug 2012)

[4] M. Rosensteiner, *Wavefront reconstruction for extremely large telescopes via CuRe with domain decomposition*, J. Opt. Soc. Am. A **29**(11), 2328-2336 (Nov 2012).

[5] N. A. Dipper, A. G. Basden, D. Geng, E. J. Younger, R. M. Myers, *AO Real-time Control Systems for the ELT Era,* AO4ELT2 Conference (2012).

[6] S. Rolt et al.: *DRAGON, a flexible, visible-light AO testbed,* AO4ELT2 Conference (2012).

[7] R. Myers *et al.*: *CANARY: the on-sky NGS/LGS MOAO demonstrator for EAGLE,* Proc. SPIE 7015, 70150E (July 2008)

[8] L. Wang and B. Ellerbroek *Computer simulations and real-time control of ELT AO systems using graphics processing units,* Proc. SPIE **8447**, 844723-1 (2012).

[9] U. Bitenc *et al.*: *On-sky tests of the CuReD and HWR fast wavefront reconstruction algorithms with CANARY*, MNRAS, **448** (2), 1199-1205 (2015).

[10] Poyneer L. A. *et al.*: *On-sky performance during verifiation and commissioning of the Gemini Planet Imager's adaptive optics system*, Proc. of SPIE 9148, 91480K-1 (2014).