Design and implementation of the PALM-3000 real-time control system

Tuan N. Truong, Antonin H. Bouchez, Rick S. Burruss, Richard G. Dekany, Stephen R. Guiwits, et al.

Tuan N. Truong, Antonin H. Bouchez, Rick S. Burruss, Richard G. Dekany, Stephen R. Guiwits, Jennifer E. Roberts, Jean C. Shelton, Mitchell Troy, "Design and implementation of the PALM-3000 real-time control system," Proc. SPIE 8447, Adaptive Optics Systems III, 84472F (13 September 2012); doi: 10.1117/12.927867

SPIE.

Event: SPIE Astronomical Telescopes + Instrumentation, 2012, Amsterdam, Netherlands

Design and Implementation of the PALM-3000 Real-Time Control System

Tuan N. Truong[1], Antonin H. Bouchez[2], Rick S. Burruss[1], Richard G. Dekany[3], Stephen R. Guiwits[3], Jennifer E. Roberts[1], Jean C. Shelton[1] , Mitchell Troy[1]

[1]Jet Propulsion Laboratory, California Institute of Technology, Pasadena, CA, USA 91109
[2]Giant Magellan Telescope, Pasadena, CA, USA 91109
[3]California Institute of Technology, Pasadena, CA, USA 91125

ABSTRACT

This paper reflects, from a computational perspective, on the experience gathered in designing and implementing real-time control of the PALM-3000 adaptive optics system currently in operation at the Palomar Observatory. We review the algorithms that serve as functional requirements driving the architecture developed, and describe key design issues and solutions that contributed to the system's low compute-latency. Additionally, we describe an implementation of dense matrix-vector-multiplication for wavefront reconstruction that exceeds 95% of the maximum achievable bandwidth on NVIDIA GeForce 8800GTX GPU.

Keywords: PALM-3000, adaptive optics, wavefront processor, matrix-vector multiplication, GeForce 8800GTX GPU

1. INTRODUCTION

Adaptive optics (AO) by its real-time nature entails fast response to wavefront changes. For an AO system to work well, commands sent to actuators changing the surface of a deformable mirror (DM) to provide the necessary compensations must be computed while the aberrations are still small. The primary goal in designing the PALM-3000 real-time control (RTC) algorithms is to achieve the highest possible control bandwidth, while assuring stability and robustness in a wide range of atmospheric conditions. From the computational perspective, this implies lowest possible compute-latency. The aim of this paper is to reflect on key design and implementation issues encountered and the solutions developed that contributed to meeting the RTC computational requirements.

The remainder of the paper is organized as follows. The RTC algorithms that serve as functional requirements driving the architecture developed are first described in Section 2. From there it will be evident that the matrix-vector multiplication (VMM) operation used for wavefront reconstruction is the computational bottleneck. Section 3 presents the distributed multi-GPU architecture. Section 4 provides the latency of each algorithm used. Section 5 describes a GPU implementation of VMM optimized specifically for the RTC reconstructor size on the NVIDIA Compute-Unified-Device (CUDA) architecture[1] and compares its performance to a leading method. Section 6 concludes the paper.

2. ALGORITHMS

The PALM-3000[2,3,4] adaptive optics system which has been in operation on the 5.1 meter Hale Telescope at the Palomar Observatory since 2011[5], employs a 64x64 subaperture Shack-Hartmann high-order wavefront sensor (HOWFS) for measurement, and three adaptive mirrors for correction. The mirrors include a tip/tilt mirror (TTM), a low-order deformable mirror (LODM) of 241 active actuators, and a high-order deformable mirror (HODM) of 3388 active actuators. The two DMs are operated in a woofer-tweeter configuration, with the LODM providing large stroke at low spatial frequencies (±2.5 μm surface stroke with 17 actuators across the telescope pupil) and the HODM providing fine correction at high spatial frequencies (±0.5 μm surface stroke with 64 actuators across the pupil).

Figure 1 depicts end-to-end dataflow of the RTC with all computation steps used to convert pixel values from the HOWFS to commands issued to the DMs. As shown, the raw pixels are read out in frames of 128x128 unsigned 16-bit values, background-subtracted and multiplied by the flat field before inputting to the current active centroiding algorithm

Adaptive Optics Systems III, edited by Brent L. Ellerbroek, Enrico Marchetti, Jean-Pierre Véran, Proc. of SPIE Vol. 8447, 84472F · © 2012 SPIE · CCC code: 0277-786/12/$18 · doi: 10.1117/12.927867

to obtain a vector of 8192 slope measurements or centroids. Background subtraction, which compensates for the camera bias, sky background, and other additive flux, involves simply subtracting a background, or dark, value acquired with no guide-star light from each raw pixel value. Flat-field correction, which compensates for individual pixel gain or QE variations in the wavefront sensor camera, requires only a multiplication operation per pixel.

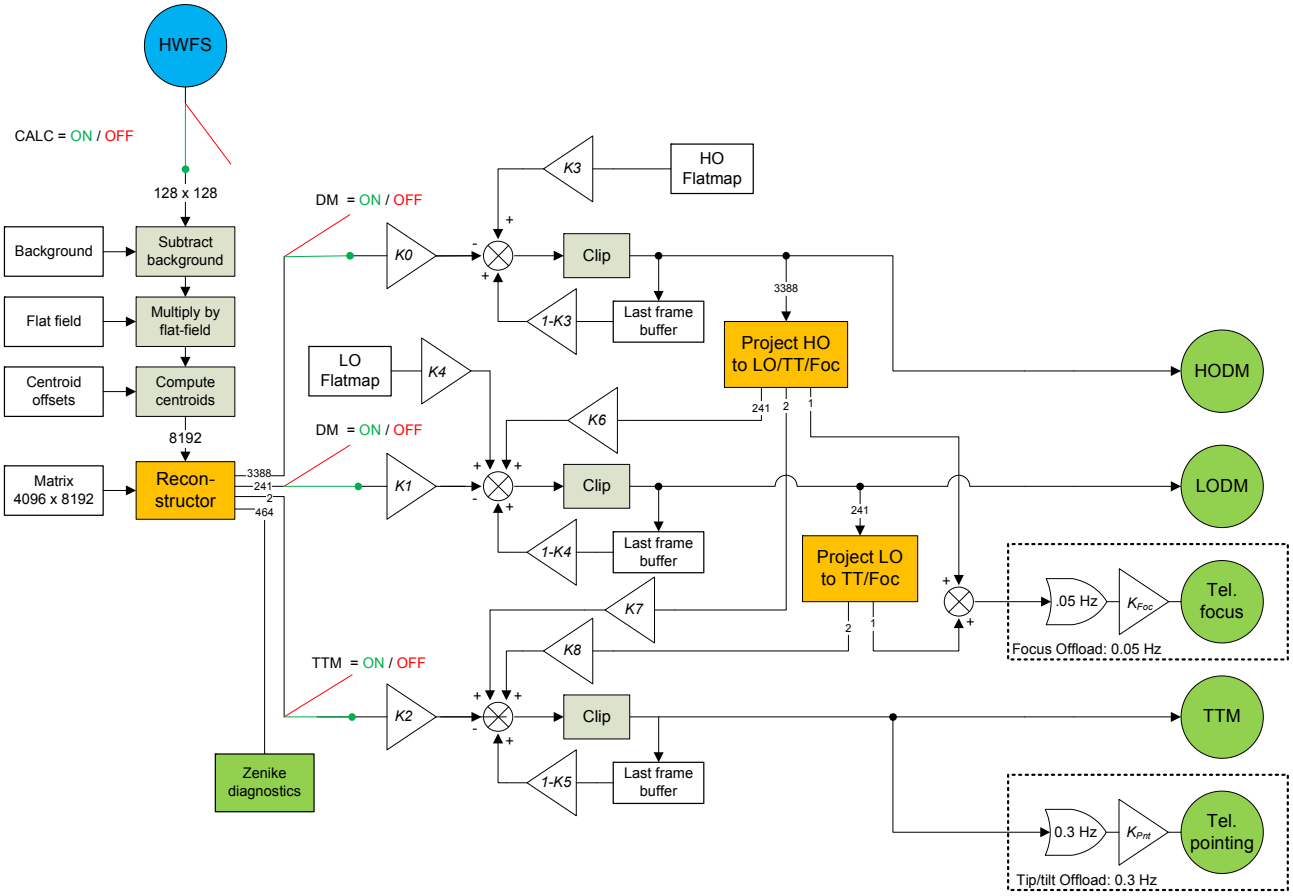

Figure 1: PALM-3000 Servo Control

The centroid vector is computed with a weighted sum over the flat-fielded pixels, divided by the total light in the subaperture. Equations below show the quad-cell centroiding algorithm used in the RTC. Therein c denotes the centroid vector output, d the centroid offset vector, $f_{\min}$ the user-selectable minimum flux, g the weights +1,+1;-1,-1 and +1,-1;+1,-1 for resolving x and y slopes, s the pixel values from flat fielding, i the index of subaperture, j the index of pixels within a subaperture, and N the total number of subapertures.

Note the clamping of the denominator to $f_{\min}$ rather than zero. The effect of this is to retain AO performance for the faintest objects rather than simply quitting. This centroiding technique known as *clamped denominator*[6] along with reconstructor upgrades[7] have shown performance improvements on sky when operating PALM-3000 on faint targets.

$$S_i = \sum_{j=1}^{N} s_{i,j}$$

$$c_i = w_i\left(\sum_{j=1}^{N} g_{i,j} s_{i,j} - d_i S_i\right)$$

$$\text{if } S_i > f_{min} \text{ then } w_i = \frac{1}{S_i} \text{ else } w_i = \frac{1}{f_{min}}$$

Next, the output centroid vector containing 8192 slope measurements (2 per subaperture) is multiplied with the current active reconstructor matrix to generate 4096 residuals that estimate the wavefront errors. Of the 4096 residuals, only 3388 values go to the HODM servo loop, 241 to the LODM, and 2 to the TTM, to generate the corresponding DM and TTM commands, with the remaining values available for diagnostics information.

Computing the DM actuator command requires the previous timestep actuator command vector, the flatmap voltages to which we want the DM to relax in absence of an error signal, the current residual vector from the reconstructor, and two servo gain coefficients. Computing a LODM actuator command requires an additional servo gain coefficient and the previous timestep LODM offload vector, which is the projection of the HODM onto the LODM calculated as the product of the HODM-to-LODM projection matrix and the HODM actuator command.

Computing the TTM command requires four servo coefficients, the current residual value, the previous timestep actuator command, the previous timestep LODM offload vector, and the previous TTM offload vector which represents the projection of the LODM onto the TTM which is calculated by multiplying the LODM-to-TTM projection matrix with a LODM actuator command.

The HODM-to-LODM and LODM-to-TTM projections are combined into one VMM operation of size 256x4096 with the estimated wavefront error vector. Note since this operation is done after the actuator commands have been issued, its execution time is excluded from the total compute-latency of the system.

Camera mode	Pupil Sampling	Available subapertures N_{sub}	Illuminated subapertures (>50%)	
			No obscuration	Central obscuration
0	64	4096	3145	2800
1-3, 9	32	1024	792	704
4-6, 10-11	16	256	200	176
7-8, 12	8	64	52	48

Table 1: Number of subapertures for various observing modes

There are optimization techniques[8] that can be performed independent of architecture. First, to minimize idle time, we overlap communication with computation, and ensure that the work done is evenly distributed across the system. Specifically, transfers of pixels are done via DMA in blocks of half-frames. This permits calculations to start as soon as a block has been received and progress while the second half of pixels making up the frame is read out. Section 4 confirms the latency improvement expected of half-frame blocking. This technique requires one buffer for calculations and at least another for transfers. We allocate a ring of buffers for transfer to tolerate occasional jitters due to kernel preemption on standard (non-real-time) Linux. Second, we exploit non-blocking communication. That is, we issue multiple asynchronous sends and receives to allow simultaneous transmission and reception of data from multiple sources to multiple destinations. To ensure load balanced, equal but different portions of data are calculated and transferred using the same number of instructions. Toward this end, we divide and coalesce memory fragments, one

from each data product, so that every processing node makes the same minimum number of equal size DMA transfers. This data shuffling in memory is justified since memory bandwidth is orders of magnitude faster than communication bandwidth.

Contrary to our stated goal of minimizing latency, the current version of the software receives and processes in half-frames of 64x128 pixels each, regardless of the HOWFS camera mode currently operating. It also multiplies the reconstructor with all subapertures whether illuminated or not. Thus the vector of measured slopes *output* by RTC centroiding algorithms always contain 8192 elements, in spite of various observing modes with different number of subapertures (see Table 1). Having said that, the number of elements actually *computed* does vary according to observing modes. VMM is also currently done on dense 4096x8192 matrices, despite the total number of controlled actuators, not including those slaved electrically is only 3631 (3388 for HODM, 241 for LODM and 2 for TTM). Future versions will correct these issues.

3. ARCHITECTURE

This section presents the architecture of the PALM-3000 computer system and describes how the RTC algorithms are mapped and parallelized on the hardware adopted. The RTC architecture is shown in Figure 2 along with external entities that use it. The system consists of five major functional subsystems, RTC, AODD, AOCA, AODB, and GUI, that spread over three separate locations, the cass cage, the computer room, and the users/observing area, that in turn interconnect via 1Gbit Ethernet. Non-critical or low speed data use this network, while real-time high-speed data go on the dedicated fiber link with direct point-to-point connections to the cass cage or via the Quadric switch.

The cass cage holds the optics, DMs, TTM, HOWFS and all hardware that must absolutely be in proximity for proper operation such as motors, fiber optic receivers/transmitters, acquisition camera and a Linux PC that hosts the necessary device drivers. The observing area, on the other end, represents the logical location of the users, which may be either operators and scientists that control the system via the graphical user interface software provided or external systems that interface with PALM-3000 such as TCS, Pharo or P1640.

Shown in the center of the figure is the computer room that houses ten AO system PCs. Nine computers named *pc0* to *pc8* run the RTC algorithms described, while the 10th named *telem* hosts the database (AODB) and command/automation (AOCA) subsystems. All ten PCs are HP xw9400 running the standard RedHat Linux 2.6.18 kernel with two dual-core processors AMD Opteron 2220 2.8GHz, 128KB L1 cache and 1MB L2 cache *per core*, 4GB memory DDR2 667MHz, and two NVIDIA GeForce 8800GX GPUs of full PCIe x16 bandwidth. Exceptions are *pc0*, which has only one GPU, and *telem,* which has 16GB of memory but no GPUs. All PCs are linked via an ultra-low latency high bandwidth Quadrics switch that delivers over 900 MB/s of user-space-to-user-space bandwidth each direction for a total of 14.4 GB/s of bisectional bandwidth and latency of only 1.8μs.

Pixel values from the HOWFS camera are replicated by the optical splitter and sent to 8 processing nodes (*pc1* to *pc8*) as they are read out. Once one half of the pixels have been received, each GPU computes the same centroids and multiplies by a different portion of the reconstruction matrix to obtain a "half-computed" wavefront error (residual) vector for one-sixteenth of the output actuators. When this process repeated on the second half of the pixels making up the frame completes, the two half-computed residual vectors are summed and then transmitted to the central node (*pc0*). Once all residuals have been received by *pc0,* the servo loop begins. The servo algorithm computes the output actuator command, sends it off to the respective mirror, and then calculates the offload projections before repeating the loop on the next residual vector.

pc0 additionally synchronizes and manages all traffic in and out of the eight slaves. All messages are tagged with a 32-bit control word containing a session number and a sequence number that uniquely identifies a message in that session. Sequence number 0 is reserved for messages containing commands and responses. Each slave increments its own sequence number every time it sends out a new message containing data. Pc0 uses the sequence numbers to identify and coalesce all residuals belonging to the same frame, provided that the data messages have the same current session number. Otherwise, they are dropped which could happen when one slave detects a frame error and notifies *pc0* quicker than others. The session number, on the other hand, is incremented by *pc0* and broadcasted to the slaves each time all synchronize in a rendezvous. Slaves synchronize in a rendezvous only when commanded by *pc0*. Otherwise they run

freely, even right after they have detected a hardware fault. All user commands from AOCA go through *pc0* which parses them out to the slaves, waits for the results from each, and replies on their behalf before synchronizing in a rendezvous with all of them.

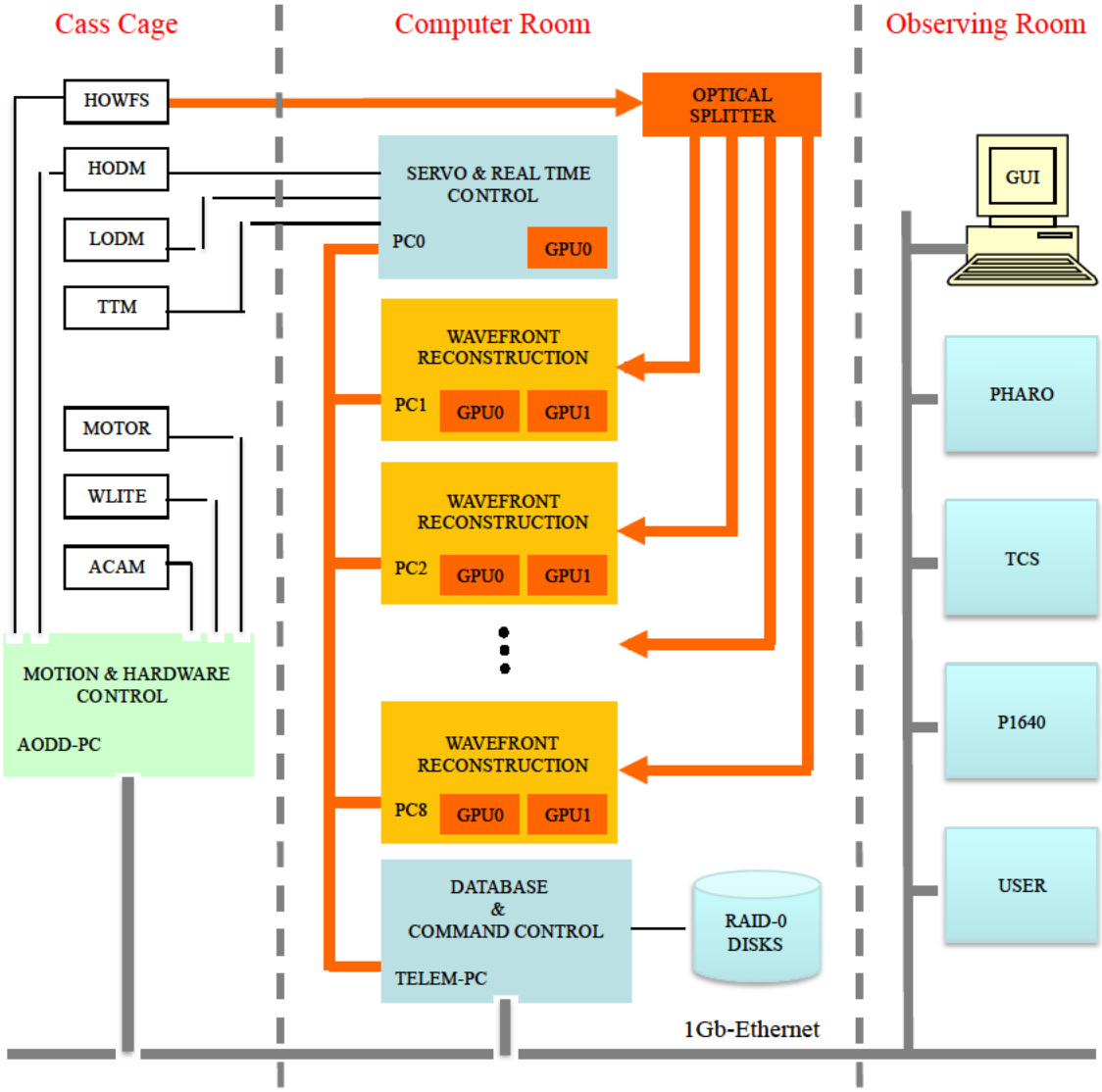

Figure 2: PALM-3000 hardware architecture. HOWFS denotes the high-order wavefront sensor, HODM the high-order deformable mirror, LODM the low-order deformable mirror, TTM the tip/tilt mirror, MOTOR all motors controlled, WLITE the white light source, ACAM the acquisition camera, AODD the device drivers, AOCA the command and automation software, TELEM the PC hosting the database, AOCA and RAID-0 file system, GUI the graphical user interface provided to control the PALM-3000 system, PHARO the Palomar high angular resolution observer, a near-infrared camera built by Cornel University, TCS the telescope control system, P1640 the advanced coronagraph and high-spectral imaging system built by American Museum of Natural History, USER any process/system subscribing to PALM-3000 data.

All nine RTC computers log data directly to the RAID-0 system located on *telem* via AODB, a publish-subscribe software layer built in-house from the ground up on top of Berkeley DB[9] and used by all PALM-3000 software components for communication and database services. To enable seamless distributed processing across machines, we use bidirectional proxy components that are interposed between user components and remote provider components.

The RTC software implements the nesC[10] programming model extended with threads support. In this model, lengthy operations and system services are performed in split-phase, i.e., using two functions, one called *command* which starts the operation and returns immediately, and the other, a callback, called *event* which indicates when the operation completes. Since events represent typically hardware interrupts, they are assumed to execute quickly and can preempt one another. In contrast, tasks run to completion and without strict timing requirements. They do not preempt one another, but events can always preempt them. As a deferred computation mechanism, tasks are posted (scheduled) for later execution more often from asynchronous events than from other scheduled tasks. To drive multiple GPUs and to avail of the parallelism afforded by multiple CPU cores, we extended nesC with threads support. Specifically, we introduced a mechanism for associating a nesC task with a thread of execution, which may be an existing thread or one newly created by the task scheduler. We use threads like we use external hardware. In this manner, threads signaling events are akin to hardware issuing interrupts. Thus, events signaled and commands invoked by tasks associated with threads are declared asynchronous (i.e., *async*). Additionally, we enhanced the language to allow more than one task to be posted in the same call for later execution, where any number of the tasks could be defined outside the current local scope (i.e., in other components, modules or processes). We use this feature to synchronize and notify components of system-wide events. One such event is application shutdown, which entails closing of all opened Berkeley DB handles. Lastly, we augmented the scheduler with prioritized task dispatching. The same mechanism for associating a task with a thread of execution is used to set any task's priority, including a pure nesC task. We chose this approach to multithread support in order to retain the benefits of the static race-condition analysis feature of the language. A problematic alternative would be to have applications interfaced directly to a thread library.

The PALM-3000 software consists of programs that embody many of the TinyOS design patterns described in [11]. Each of the programs is an assembly of components connected ("wired") via named *interfaces*. To ease usability, we engineered interfaces to contain only the minimum functionality required for most usage scenarios. Configuration and customization functions were purposely excluded. They are, instead, enabled only when needed at runtime via component-specific command lines switches, environment variables, and configuration files. For example, on our system, components using the interface to subscribe to data deal only with one function, namely the callback, to receive the data. There are no calls to make to initialize the providing component that implements the interface, change the subscribed data rate, or specify the location of the data server. The interface simply does not require them. This design approach has resulted in not only a system that is more extensible but also components that are simpler to develop.

4. PERFORMANCE

Figure 3 lists the execution times in microseconds, averaged over 1 million invocations ignoring the first, of each algorithm computed by the RTC as described previously. It shows clearly the benefit of half-frame blocking over the obvious full-frame. Measuring from the time the last pixel was received by the application up to the instant actual commands are ready to be issued to the mirrors, the total elapsed time is 205μs for half-frame blocking versus 281μs for full-frame. Not included in these totals nor shown in the figure are the latencies of the inter-actuator check, the DM driver and the offload calculation that follows in parallel. The inter-actuator check currently requires 35μs with room for improvement. The DM driver's latency at 125μs is expected to drop down to about 25μs later this year. The offload latency of 80μs, which included 13μs for the launch overhead, is excluded from the total compute-latency because the calculation is started after the commands have been issued. From the transfer times shown, it is clear that getting data in and out of GPUs is not of an issue, provided that the input and output buffers are allocated from non-pageable memory.

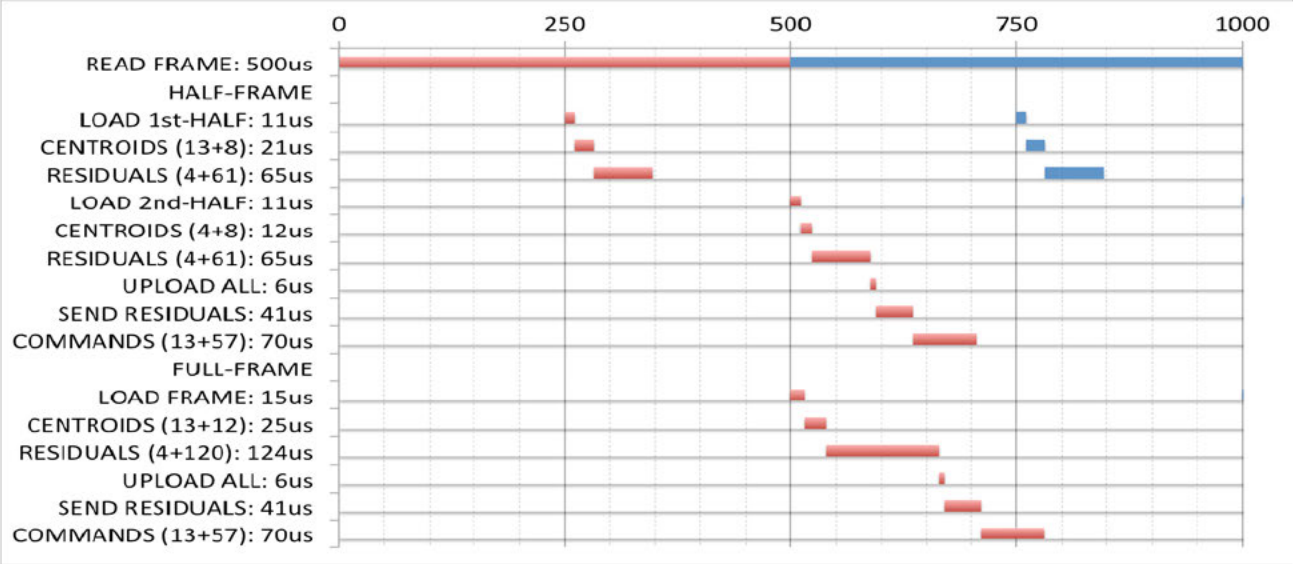

Figure 3: RTC latencies in microseconds. Shown is the processing timeline of the RTC servo algorithm for two frame periods at 2 KHz frame rate. Measuring from the last pixel received by the application up to the point actuator commands are ready to be issued to the mirror, the total latency is 205μs for half-frame blocking and 281μs for full-frame. Not included in these totals nor shown in the figure are the latencies of inter-actuator check and DM driver. "Upload All" denotes uploading (GPU-to-host) of all data, which include pixels, flux, centroids, and residuals. The transfer size equals the total divided by the number of GPUs in the system.

5. MATRIX VECTOR MULPLICATION

This section presents an improved VMM algorithm used by the RTC. Figure 4 shows the core loop of the code on the NVIDIA CUDA architecture. Table 2 compares its performance with the leading method of Fujimoto[12]. The times shown were averaged over 1 million kernel invocations ignoring the first, for several matrix sizes of interest to PALM-3000, excluding data transfers between CPU and GPU and kernel launch overhead. The thread blocks used for the comparison are 16x4 for the PALM-3000 method and 16x16 as suggested for [12].

Matrix Size	Fujimoto	RTC	Speedup
256 x 4096	76	61	1.24
256 x 2048	40	32	1.25
512 x 4096	150	129	1.16
512 x 2048	80	68	1.17

Table 2: Matrix-vector multiplication performance comparison. Speedup is the ratio of two times (μs).

As shown, the performance of the RTC method on small matrices is unmatched and due to the superior caching design of the GPU texture memory. For a bandwidth-bounded algorithm like VMM, the execution time can be predicted by the equation.

$$\text{Time} = \text{Kernel launch overhead} + \frac{\text{Bandwidth required}}{72\,\text{GB/s}}$$

Using our hardware/software setup, the maximum sustained achievable bandwidth reported by *bandwidth* (an application provided in the NVIDIA software development toolkit) is 72GB/s, and the launch overhead 4μs if the kernel is launched right after another and 13μs otherwise. For the matrix size of 256x4096 operated on by each GPU, the RTC algorithm attains over 95% of the maximum achievable memory bandwidth of the GeForce 8800GTX GPU, providing speedup of 1.24x. However, for large matrices such as 2048x4096 and 4096x4096 that are not of interest, the RTC method is outrun by 10% to 15% respectively.

```
// Y = AX
// A : matrix in texture cache via texA
// X : vector in texture cache via texX
//
// Number of threads:  R x C per block
// Number of blocks:   heightA / R
// Thread id:  tx, ty
// Block id:   bx
//
__shared__ float y[R][C];

row = bx*R + ty;
y[ty][tx] = 0;

for (i = 0; i < widthA/4; i+= C)
  {
    float4 a = tex2D( texA, tx+i, row );
    float4 x = tex1Dfetch( texX, tx+i );
    y[ty][tx] += a.x*x.x + a.y*x.y + a.z*x.z + a.w*x.w;
  }
__syncthreads();

// Y[row] = sum(y[ty][*]) using binary reduction
```

Figure 4: Partial kernel codes for matrix-vector multiplication in RTC

6. CONCLUSIONS

We described the algorithms used for real time control of the PALM-3000 adaptive optics system and presented a scalable architecture that satisfies its computational requirements. The key components we pointed out that contributed to the system's low latency were the optical splitter, the GPUs, and the Quadrics switch. With larger interconnects and additional if not faster GPUs, we expect the architecture could easily support much larger AO systems at higher rates and lower VMM latency. We also presented an improved algorithm for matrix-vector multiplication on the NVIDIA CUDA architecture that reaches within 5% of maximum achievable bandwidth and delivers speedup ranging from 1.16x to 1.25x over a leading method.

ACKNOWLEDGEMENTS

This research was carried out at the Jet Propulsion Laboratory, California Institute of Technology, and was sponsored by the California Institute of Technology and the National Aeronautics and Space Administration..

REFERENCES

[1] NVIDIA, “CUDA Programming Guide,” Version 1.0 (2007).

[2] Dekany, R.G., et al., “PALM-3000: visible light AO on the 5.1 meter telescope,” Proc. SPIE 6272, 62720G (2006).

[3] Bouchez, A., et al., “The PALM-3000 high-order adaptive optics system for Palomar Observatory,” Proc. SPIE 7015, 70150Z (2008).

[4] Truong, T.N., et al., "Real-time wavefront control for the PALM-3000 high-order adaptive optics system," Proc. SPIE 7015, 70153I (2008).

[5] Roberts, J.E., et al., "Results from the PALM-3000 high-order adaptive optics system," Proc. SPIE 8447, 8447-34 (2012).

[6] Shelton, C., “Denominator-free control algorithms for adaptive optics,” Proc. SPIE 3126, 455-459 (1997).

[7] Shelton, C., Troy, M., Bouchez, A., Roberts, J., Trinh, T., Truong, T., “NGS and LGS adaptive optics – Improving faint light performance,” Center For Adaptive Optics Fall Retreat (2009).

[8] Becker, A., Venkataraman, R., Kale, L., “Patterns for overlapping communication and computation,” Dept. of Computer Science, Univ. of Illinois at Urbana-Champaign, Urbana, IL.

[9] Olson, M.A., et al., “Berkeley DB,” Proc. of the FREENIX Track: USENIX Annual Technical Conference (1999).

[10] Gay, D., et al., “The nesC language: A holistic approach to networked embedded systems,” Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation, 1-11 (2003).

[11] Levis, P., “TinyOS Programming,” http://www.tinyos.net (2006).

[12] Fujimoto, N, “Dense matrix-vector multiplication on the CUDA architecture,” Parallel Processing Letters 2008, Vol. 18, No. 4, 511-530 (2008).