An efficient implementation of the Shack-Hartmann centroids extraction for edge computing

JACOPO MOCCI[1], FEDERICO BUSATO[1], NICOLA BOMBIERI[1], STEFANO BONORA[2], AND RICCARDO MURADORE[1,*]

[1] *Department of Computer Science, University of Verona, Italy*
[2] *CNR-IFN of Padova, Italy*
[*] *Corresponding author: riccardo.muradore@univr.it*

Compiled August 14, 2020

Adaptive optics is an established technique to measure and compensate for optical aberrations. One of its key components is the wavefront sensor (WFS), which is typically a Shack-Hartmann sensor (SH) capturing an image related to the aberrated wavefront. We propose an efficient implementation of the SH-WFS centroids extraction algorithm, tailored for edge computing. In the edge-computing paradigm, the data is elaborated close to the source (i.e., at-the-edge) through low-power embedded architectures, in which CPU computing elements are combined with heterogeneous accelerators (e.g., GPUs, FPGAs). Since the control loop latency must be minimized to compensate for the wavefront aberration temporal dynamics, we propose an optimized algorithm that takes advantage of the unified CPU/GPU memory of recent low-power embedded architectures. Experimental results show that the centroids extraction latency obtained over spot images up to 700×700 pixels wide is smaller than 2ms. Therefore, our approach meets the temporal requirements of small to medium-sized AO systems, which are equipped with deformable mirrors having tens of actuators.

http://dx.doi.org/10.1364/ao.XX.XXXXXX

1. INTRODUCTION

Light beams emanated from an object are distorted by perturbations presented in the optical path between the object and the observer. Such perturbations affect the phase of each beam, leading to an aberrated wavefront which, in turn, deteriorates the image quality of the observed object [1].

Adaptive Optics (AO) is a widely known technique which objective is to obtain diffraction-limited images of the observed object. An AO system is composed of one or more WaveFront Sensors (WFS), one or more Deformable Mirrors (DM) and a controller. By the phase conjugation principle, the controller shapes the DM into the conjugated wavefront measured by the WFS, compensating for the aberrations. Extensive literature in a wide range of fields including astronomy [2], ophthalmology [3], microscopy [4], communication [5] and high power laser [6] has demonstrated the practical value of AO for improving the image quality despite optical and/or atmospheric aberrations.

The most common WFS is the Shack-Hartmann WFS (SH-WFS), which measures the wavefront local gradients by spatially sampling the incoming beam with a lenslet array, with each lenslet focusing the local subaperture into a CCD or CMOS camera pixel array. Several algorithms deal with the centroid extraction from the spots image, the choice of which depends on the dynamic range of the centroids (e.g., spots overlapping neighboring reference spots), image quality (e.g., faint, non-uniformly distributed spots) and software complexity [7–9].

Since the AO control loop frequency should be at least one order of magnitude higher than the cutoff frequency of the wavefront aberrations dynamics, WFS measurement latency must be minimized by either sacrificing wavefront spatial resolution (e.g., smaller beam aperture leads to shorter exposure time because of the higher photon flux and fewer pixels to be transferred), or by choosing a fine-tuned extraction algorithm to be implemented into a suitable architecture. Under normal conditions, each subaperture is completely independent of the others. Hence, it is natural to tackle the SH-WFS centroid extraction problem exploiting the throughput-oriented Single-Instruction Multiple-Data (SIMD) paradigm, so that the centroids can be computed in parallel using the same set of instructions.

Graphic Processing Units (GPU) and Field-Programmable-Gate-Array (FPGA) architectures are examples of architectures well suited to SIMD computation. However, while FPGA excels in raw performance compared to GPU, the latter offers *programming flexibility* also found in Central Processing Units (CPU) latency-oriented architectures. In fact, most of the GPU development tools are extensions of those used for programming CPU code. Therefore, both GPU and CPU solutions offer easier debugging solutions than FPGA and faster deployment for experimental testing.

Wavefronts related to the aberrated eye pupil can be reconstructed from SH-WFS images using a desktop-class GPU, achieving lower latency when compared to its CPU implementation [10]. The wavefront can also be estimated from SH-WFS images by means of a GPU-accelerated neural network, with detection accuracy near that of traditional methods when properly trained [11]. The big amount of data coming from the SH-WFS arrays in the Extremely Large Telescope (ELT) class can be tackled by *GPU clusters* to accelerate the gradients extraction [12, 13]. Aside from the real-time computation, introductory work on the tomographic reconstruction of the atmospheric turbulence through neural networks on GPU has also been done [14] and several GPU-accelerated optics simulation frameworks have been developed [15–17]. Some AO-related algorithms implemented on GPU are demonstrated to perform even better than their FPGA counterparts [18].

However, little attention has been given to the implementation of the SH-WFS algorithms for *edge computing* [19], which is an effective solution for small- to medium-size telescopes [2]. Many solutions for edge computing combine a hardware accelerator to the input sensor. Examples are FPGAs embedded with camera sensors (i.e., the *smart-cameras*) [20], which can process the wavefront with minimal latency [21, 22]. Nonetheless, heterogeneous architectures in which CPU processing elements are combined with GPU accelerators are the preferred alternative to FPGA-based smart cameras when dealing with hard to implement visual computing algorithms [23–25]. In addition, recent heterogeneous low-power architectures (e.g., NVIDIA Jetson TX2 and Nano) allow to reduce the memory transfer overhead by implementing a unified CPU/GPU memory. They are getting pervasive for *edge-computing* thanks to their compact foot-print, power and energy-efficiency, and competitive High-Performance Computing (HPC) capabilities [26].

In this paper we propose a SH-WFS centroids extraction algorithm that is tailored for edge computing on low-power CPU/GPU devices with unified memory. We show that the time needed to complete the centroid extraction from the image acquisition is small enough to satisfy the AO closed-loop latency constraint. Our contribute is the implementation of such an algorithm that:

- Can efficiently run on portable, low-power, energy-efficient devices, enabling *at-the-edge* AO control.
- Is flexible with respect to the WFS hardware configuration;
- Guarantees lower latency than the on-board CPU counterparts;

The paper is organized as follow. Section 2 briefly recalls how a Shack-Hartmann sensor works, whereas Section 3 introduces the moment-based centroid extraction algorithm. Section 4 explains the parallel implementation of the centroid extraction algorithm, whose experimental results are shown in Section 5. In Section 6 some conclusions are drawn together with a future work plan.

2. SH-WFS OPERATION PRINCIPLE

The SH-WFS is a sensor that measures the wavefront distorsions by computing its local gradients. Fig. 1 shows the optical principle that allows the SH-WFS to spatially and temporally sample the incoming wavefront. The l-th lenslet of the lenslet array focuses the local wavefront into a light spot on the *pixel array* of the capture device. Assuming a point-like light source (e.g., a distant star), the *centroid position* $c_l \in \mathbb{R}^2$ of the spot is related to the spatial displacement of the incoming aberrated wavefront with respect to the flat wavefront (no aberrations).

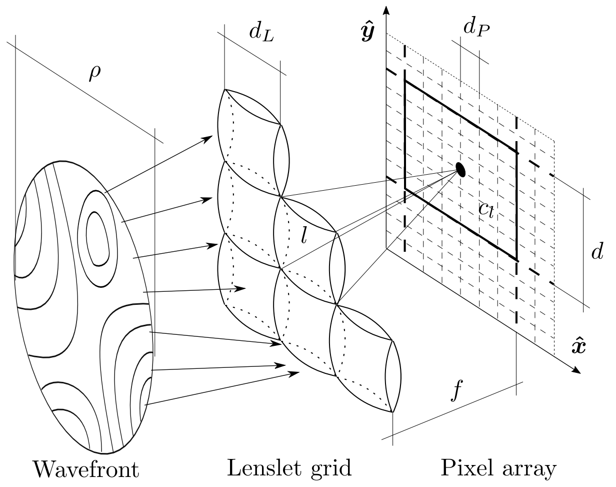

Fig. 1. Operation principle of the Shack-Hartmann WFS.

Let $\rho \in \mathbb{R}$ be the diameter of the telescope circular aperture. The grid-shaped lenslet array L inscribes the wavefront of the incoming light beam, with the number of lenslets in each row given by

$$w_L = \left\lfloor \frac{\rho}{d_L} \right\rfloor, \tag{1}$$

where $d_L \in \mathbb{R}$ is the size of each lenslet as shown in Fig. 1. The lower bracket $\lfloor z \rfloor$ is the floor rounding operator that returns the greatest integer smaller than or equal to $z \in \mathbb{R}$.

The lenslets focal length $f \in \mathbb{R}$ determines the size of the pixel region in which the l-th spot is focused into. A large focal length favors centroids sensitivity, whereas a small focal length increases the centroids dynamic range. Spots might be imaged anywhere in the pixel array, resulting in arbitrarily large and overlapping pixel regions. However, if f is sufficiently small in relation to the expected incoming wavefront spatial variance, then the pixel regions are disjoint. This is a valid assumption when measuring the wavefront compensated by an AO control system.

Each pixel region is a square since the lenslet is arranged in a grid. The number of pixels contained in a row of a pixel region is

$$d = \frac{d_L}{d_P}, \tag{2}$$

where $d_P \in \mathbb{R}$ is the pixel width (assuming no *dead zone* among regions).

Let $p = (x_p, y_p)$ be a pixel position, with $x_p, y_p \in \mathbb{N}$ being the Cartesian coordinates of the pixel in the pixel array. Each lenslet l focuses the spot into the pixels in the l-th pixel region

$$P_l = \{p \mid \lfloor x_l \rfloor \leq x_p < \lfloor x_l + d \rfloor, \lfloor y_l \rfloor \leq y_p < \lfloor y_l + d \rfloor\}, \tag{3}$$

where $(x_l, y_l) \in \mathbb{R}^2$ is the pixel coordinate of the bottom-left corner of the pixel region. The mapping from the lenslet array into the pixel array is calibrated by identifying the pixel position (x_0, y_0) located at the bottom-left pixel of the bottom-left lenslet in the lenslet grid [1]. Then, the bottom-left pixel coordinate of the l-th pixel region is calculated as

$$(x_l, y_l) = \left(\left\lfloor x_0 + d \left\lfloor \frac{l}{w_L} \right\rfloor \right\rfloor, \lfloor y_0 + d\,(l \mod w_L) \rfloor \right). \quad (4)$$

Merging all pixel regions yields to the pixel Region of Interest (RoI) P, which is a square region having width [2]

$$w_P = dw_L, \quad (5)$$

as illustrated in Fig. 2.

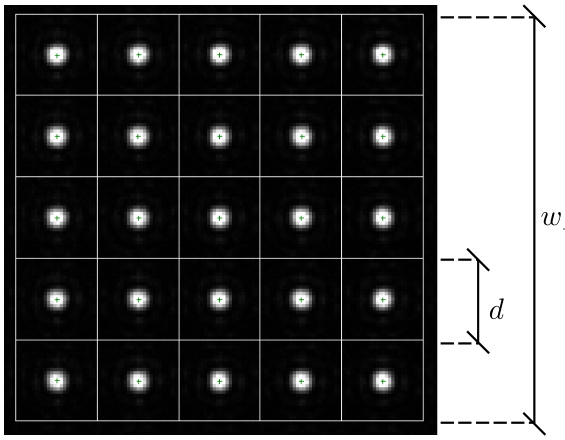

Fig. 2. SH-WFS spot image. The pixel regions P_L are adjacent each-other, and together form the pixel RoI P.

Comparing the measured centroid position c_l with its *reference centroid*, i.e. the centroid measured when the wavefront is flat, yields to the incoming wavefront phase gradients (slopes) from which the wavefront phase can be reconstructed by zonal or modal techniques [1]. The centroids can be extracted from the WFS spot image by a moment-based method [7], which is used to calculate the Center of Gravity (CoG) of each spot on the pixel array.

3. MOMENT-BASED CENTROID EXTRACTION

Let $I(p) \in \mathbb{R}$ be the measured intensity value at a RoI pixel $p \in P$. The image moments for the l-th spot image are defined as

$$m_l^{ij} = \sum_{p \in P_l} x_p^i y_p^j I(p) \quad (6)$$

where $i, j \in \mathbb{N}$ are the moments order over the Cartesian directions. The l-th centroid position c_l is the CoG of the l-th spot calculated using the image moments as

$$c_l = \left(\frac{m_l^{10}}{m_l^{00}}, \frac{m_l^{01}}{m_l^{00}} \right). \quad (7)$$

According to the pixel region defined in Eq. (3), all the centroids c_l of the spot image can be extracted by implementing the Alg. 1 [27].

The lower bound of the computational cost for the centroids extraction algorithm is given by the analysis of the two nested for-loops. Since the pixel regions P_l covering the lenslet array L are adjacent, disjoint and completely contained in the pixel RoI P, the inner loop iterates over each pixel $p \in P$. The pixel RoI P is square and hence its size is $\lfloor dw_L \rfloor^2$. A total of 5 operations are computed for each pixel p to update the moments. The outer loop iterates over each lenslet $l \in L$, where the lenslet grid L is a square of size w_L^2. Each centroid c_l requires 2 instructions to be calculated from the moments as shown in Eq. (7).

Algorithm 1. Lenslet-wise Centroids Extraction

```
1: for all $l \in L$ do
2:     $m_l^{00}, m_l^{10}, m_l^{01} \leftarrow 0$
3:     for all $p \in P_l$ do
4:         $m_l^{00} \leftarrow m_l^{00} + I(p)$
5:         $m_l^{10} \leftarrow m_l^{10} + x_p I(p)$
6:         $m_l^{01} \leftarrow m_l^{01} + y_p I(p)$
7:     $c_l \leftarrow \left( \frac{m_l^{10}}{m_l^{00}}, \frac{m_l^{01}}{m_l^{00}} \right)$
```

Therefore, the lower bound complexity depends on the lenslet grid width w_L and the pixel region width d:

$$\Omega(w_L, d) = 2w_L^2 + 5 \lfloor dw_L \rfloor^2. \quad (8)$$

Since the *moment partials* of Eq. (6) (i.e. the addends of the sum) are mutually independent, they can be computed concurrently at once and then summed up to yield the lenslet moment.

Since the CoG method to extract the centroids is sensitive to the WFS Signal-to-Noise Ratio (SNR), standard image-processing techniques could be implemented to enhance the SNR such as (i) *thresholding* pixels' intensities below the noise level, (ii) correcting the gamma function of the intensity by applying a power-law transformation, and (iii) excluding pixels near the lenslet edges (i.e. *windowing*) [28].

4. CUDA IMPLEMENTATION

CUDA and OpenCL are the two dominant frameworks for parallel programming. They offer the same capabilities, e.g., exploiting unified memory, albeit with different hardware terminology and code syntax. OpenCL is open-source and compatible with a wide range of GPU and multi-core CPU architectures, whereas CUDA is proprietary and only compatible with NVIDIA architectures. On the other hand, since the CUDA framework is specific to NVIDIA architectures, it guarantees tighter implementation than OpenCL and deeper development tools integration.

We rely on the CUDA parallel framework to fully exploit the Jetson architecture [29]. In the CUDA programming language, a *device* (GPU) utilizes its *threads* and *memory* to execute *kernels* (i.e., functions) called by the *host* (CPU). Threads are organized in *blocks*, and blocks are contained in a *grid*. The scheduler maps blocks into multiple *streaming multiprocessors*. Each thread in a block is mapped to a *core*. Up to 32 threads (i.e., one *warp* of threads) can be scheduled to concurrently execute the same instructions (i.e. SIMD).

Each thread can efficiently access *register*, *local* or *shared* memory. While the first two are limited to the thread scope, the latter is available to all threads in a block and, hence, can be used for efficient communication among threads. The register memory is faster than local memory, but it is also the least available. Register, local and shared memories are device-side resources meant to store temporary results. The host must access the device slower *global* memory to store the dataset to be processed by the kernel and read the results.

[1]Throughout the paper, 1-dimension lenslet and pixel arrays are indexed as row-major arrays starting from 0.

[2]The pixel RoI width $w_P \in \mathbb{N}$ is expressed in pixel units, whereas the lenslet RoI width $w_L \in \mathbb{N}$ is expressed in lenslet units.

A. Data Levels

Since the spot image is stored into memory as a *row-major* linear array, the pixel expressed in Cartesian coordinate $p = (x_p, y_p)$ is mapped into the array index

$$\phi = x_p + w_P y_p. \tag{9}$$

By accessing contiguous elements along the rows, neighboring data chunks are cached in fast memory leading to smaller transfer latency time. To leverage the memory cache, the spot image is partitioned into data levels.

The topmost level represents the entire *spot image* in which the pixels RoI P, i.e., the lenslet grid, is immersed. The lenslet grid is divided into rows that are spanned by *lenslet groups*. Each lenslet group $\bar{l}$ has the same power-of-two size, up to 16 lenslets wide, to satisfy the condition for the reduction operation on its elements. Multiple lenslet groups are concatenated to cover all lenslets in a row. Eventual lenslets remaining in the last lenslet group are padded with zeros. A lenslet l is partitioned in a power-of-two number of stacked *row groups*. The size of each row group $\phi^l_{\bar{r}}$ is given by the rounded up ratio between the number of pixel rows in a lenslet and the number of row groups to be assigned for a lenslet. The remaining rows of the last row group are outside the lenslet and hence are ignored from the computation. Each *row* $\phi_r^{l,\bar{r}}$ in a row group is composed of *pixels*. Pixels in a row are covered by *pixel groups*, with each pixel group $\phi_{\bar{p}}^{l,\bar{r},r}$ containing 4 pixels $\phi_p^{l,\bar{r},r,\bar{p}}$ to optimize memory transfer. Since pixel groups are aligned to the beginning of the spot image array, leading and trailing pixel groups may contain pixels outside the scope of the actual lenslet. The intensities of such pixels are read as zero so that they do not contribute to the final result.

Let the *pixel partial* be the set of partial moments m^{00}, m^{10} and m^{01} calculated at the pixel position $\phi_p^{l,\bar{r},r,\bar{p}}$:

$$\mu_p^{l,\bar{r},r,\bar{p}} = \{m^{00}(\phi_p^{l,\bar{r},r,\bar{p}}); m^{10}(\phi_p^{l,\bar{r},r,\bar{p}}); m^{01}(\phi_p^{l,\bar{r},r,\bar{p}})\}. \tag{10}$$

The sum between two pixel partials $\mu_i^{l,\bar{r},r,\bar{p}} + \mu_j^{l,\bar{r},r,\bar{p}}$ is the set defined as the piece-wise sum of their respective partial moments. Summing together all the pixel partials associated to a pixel group yields to the *pixel group partials* $\mu_{\bar{p}}^{l,\bar{r},r}$. The *row partial* $\mu_r^{l,\bar{r}}$ of a row group is obtained by adding the pixel group partials calculated on its underlying pixel groups. The *row group partial* $\mu^l_{\bar{r}}$ of a lenslet is given by summing up all the row partials in a row group. Finally, the *lenslet moment* μ_l of the lenslet grid is the sum of the row group partials contained in the l-th lenslet.

Fig. 3 describes how a single pixel p of the spot image is indexed through the data levels. The lenslet grid is 7 lenslets wide and the lenslet size is 10×10 pixels. With the lenslet groups size fixed to 4 lenslets, it takes 2 lenslet groups to cover a row of lenslets, with one remaining lenslet. The 5-th lenslet group accesses lenslets 18, 19 and 20, zero padding the others. Each lenslet is partitioned into 2 row groups and hence each row group spans 5 rows, with no rows left out. The lenslet pitch is 10 pixels and can be covered by up to 4 pixel groups, depending on the memory alignment. In the figure, the 4-th pixel row only needs 2 pixel groups. The remaining pixel intensities of the 2-nd pixel group are read as zeros (and no further pixel groups are to be drawn). The selected pixel p is indexed as $\phi_1^{20,1,4,2}$ and produces the partial pixel moment $\mu_1^{20,1,4,2}$. Adding together the partial moments from the bottom to the top of the data hierarchy levels yields to the lenslet moment μ_{20}.

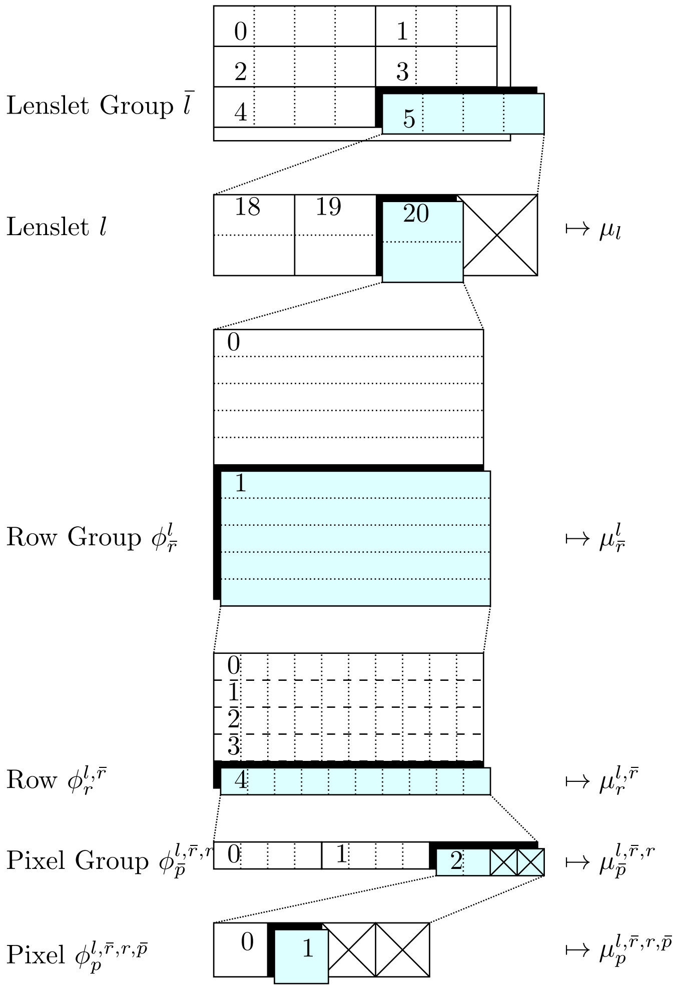
Lenslet Group $\bar{l}$
0
1
2
3
4
5
Lenslet l
18
19
20
$\mapsto \mu_l$
Row Group $\phi^l_{\bar{r}}$
0
1
$\mapsto \mu^l_{\bar{r}}$
Row $\phi_r^{l,\bar{r}}$
0
1
2
3
4
$\mapsto \mu_r^{l,\bar{r}}$
Pixel Group $\phi_{\bar{p}}^{l,\bar{r},r}$
0
1
2
$\mapsto \mu_{\bar{p}}^{l,\bar{r},r}$
Pixel $\phi_p^{l,\bar{r},r,\bar{p}}$
0
1
$\mapsto \mu_p^{l,\bar{r},r,\bar{p}}$

Fig. 3. Diagram of the data hierarchy levels as seen from the CUDA extraction algorithm. On the right: partials obtained by reducing the current level data. With the exception of the lenslet moment, numerical indexes are relative to the level.

B. Optimized GPU Data Transfer through Coalesced Memory Accesses

In most architectures, the global memory and the CPU memory are physically decoupled. This means that data has to be transferred between memories, with each transfer increasing the temporal overhead over the execution time. To overcome such overheads, latency-hiding techniques can be exploited. However, the kernel execution time must be comparable to the transfer time to take advantage of those techniques. Since host and device on a Jetson architecture physically share the same global memory, allocated memory can be addressed both by host and device by *pinning* it (page-locked mapped memory, also called Zero-Copy), hence avoiding any transfer overhead.

Remark 1 *Page-locked memory on the Jetson TX2 GPU is not cached when accessed by the CPU. As a consequence, host-side memory reading is not optimized. However, the CPU accesses the memory only to write the intensity values acquired from the sensor and read back the computed commands to be sent to the actuators. Therefore, there is no performance penalty using page-locked memory instead of other addressing options.*

Up to 128 Bytes of data in global memory can be accessed in one transaction. To fully exploit the cache, threads in a warp should ideally access consecutive single precision words (4 Bytes) starting from a 128 Bytes-aligned address to realize a *coalesced* memory transfer. Since the spot image is encoded into a row-major 8-bit array with stride divisible by 4, each thread of a warp reads adjacent words of 4 Bytes, hence accessing 4 pixels (a full pixel group) at once.

The highest bandwidth throughput is achieved when a warp reads 32 pixel groups, i.e. 128 Bytes. To maximize caching performance, the warp operates on the contiguous rows of a lenslet group instead of a single lenslet row. This way, a warp services multiple lenslets. The lenslet pitch d determines the maximum number of pixel groups needed to cover a single lenslet row. Since pixel groups are read from contiguous memory, their memory alignment must be accounted for. The worst case scenario is that the first pixel of the row addresses the last memory location of a packet, hence loading the full packet while only requiring 1 pixel out of 4. Similarly, the logical address of the last pixel of the row can point to the first memory location of the packet, again ignoring the remaining 3 pixels. The maximum number of packets is therefore calculated by considering those extra 6 offset pixels:

$$n_{packets} = \left\lfloor \frac{d+3+3}{4} \right\rfloor . \quad \textbf{(11)}$$

The number of packets determines the optimal lenslet group size:

$$d_{\bar{L}} = \exp_2 \left\lfloor \log_2 \left\lfloor \frac{32}{n_{packets}} \right\rfloor \right\rfloor . \quad \textbf{(12)}$$

Since the lenslet groups are consecutively stacked to cover a row of lenslets of the lenslet grid starting from the leftmost lenslet, the lenslets of the last rightmost lenslet group outside the lenslet grid are ignored.

C. Data Reduction

Parallel data reduction summarizes (by using a commutative binary operator) all the homogeneous data of a dataset by exploiting communication and synchronization functions among concurrent execution units (i.e. CUDA threads). Given 2^n elements in a dataset mapped to a pool of 2^n execution units, $n \in \mathbb{N}$, the elements of each disjoint pair of execution units in the pool are reduced concurrently into intermediate results, which are half the size of the original dataset. This process is iterated over such intermediate results, with each iteration halving their size. The reduction result is the intermediate result at the last iteration, reached when there are no pairs left. Since operations are done concurrently in a *logarithmic-tree* fashion, the computational cost to reduce 2^n elements is $O(n)$. If the number of elements is not power-of-two, then the dataset is padded with elements which value is neutral with respect to the binary operator considered.

In the CUDA programming model, the parallel reduction can be efficiently implemented at warp level through *shuffle* primitives, which are low-level CUDA communication and synchronization functions [30]. The shuffle instructions let threads access the registers of other threads scheduled in the same warp, despite registers being local to the threads. By exploiting such primitives, the parallel reduction operates over registers, which are the fastest type of memory in the CUDA architecture.

Since a warp consists of 32 concurrent threads, the dataset to be reduced must be 32 elements large to achieve the peak efficiency. However, several smaller datasets can be operated at once by combining them into a 32-elements dataset. To do so, the datasets must have the same power-of-two size, eventually padding remainder elements with neutral elements. Then, they are concatenated into the full dataset that must be also padded with neutral elements if its size is not power-of-two.

The full dataset is processed via the *XOR* scheme (butterfly accessing pattern [30]) to reduce the sub-datasets concurrently. Each thread participating in the shuffle stores the sum of its value with the one of the thread addressed by the bitwise XOR between the caller thread index and the mask value. The reduction can then be performed on all datasets without cross-talking by specifying the mask value at each iteration of the algorithm. Assuming 2^m datasets (i.e. the lenslet group size), the reduction algorithm returns the results after $\log_2(32) - m + 1$ iterations.

Fig. 4 shows how the pixels of the spot image are reduced to lenslet moments μ_l. As described in Subsection B, a warp covers the adjacent pixel rows of the lenslets contained in the lenslet group $\bar{l}$, one row per lenslet. Each thread in the warp sequentially reduces the pixel partials $\mu_p^{l,\bar{r},r,\bar{p}}$ into the pixel group partial $\mu_{\bar{p}}^{l,\bar{r},r}$. Then, the warp performs a parallel shuffle reduction over the dataset built on the pixel group partials, yielding to the row partials $\mu_r^{l,\bar{r}}$.

Row partials within a row group $\phi_{\bar{r}}^l$ are operated one after the other by a single warp. To increase *occupancy*, multiple warps can be associated to a lenslet group, one for each row group. To do so, threads in a block are partitioned into warps by using *cooperative groups*, a CUDA implementation feature that lets warps to be synchronized independently. All row group partials $\mu_{\bar{r}}^l$ are then stored in shared memory and reduced in parallel into lenslet partials μ_l.

In the example in Fig. 4, the lenslet moments of the lenslets 0 and 1, which are inside the same lenslet group, are extracted at the same time. Since the lenslet pitch is $d = 10$, the maximum number of packets for each row is $n_{packets} = 4$ and the optimal lenslet group size is $d_{\bar{L}} = 2$ according to Eq. (12) (assuming 8 threads per warp for the sake of space). The pixel groups memory location needed for the 2 lenslet rows, $\bar{p}_0$ to $\bar{p}_5$, are loaded from the global memory into each thread. The memory location from $\bar{p}_0$ to $\bar{p}_3$ contains the pixel groups $\phi_0^{0,0,r}$ to $\phi_3^{0,0,r}$, whereas the location from $\bar{p}_3$ to $\bar{p}_5$ contains the pixel groups $\phi_0^{1,0,r}$ to $\phi_2^{1,0,r}$. Since pixel groups are contiguous, $\bar{p}_3$ is used by both lenslets and is therefore cached. Each thread demuxes and sums the pixels into group partials. The last pixel group partial does not make part of the currently considered lenslets and is therefore imposed to zero. The shuffle reduction yields to the row partials of both lenslets, which are then added to the other row partials (the accumulators are denoted by σ_0^0 and σ_0^1). The sum of all row partials gives the 0-th row group partials of both lenslets and is stored into shared memory. Since the lenslet is partitioned into 2 row groups, the number of threads scheduled for a block is set to 16 so that two cooperative groups of 8 threads can be formed. Hence, the 0-th and 1-st row group partials are calculated concurrently. Reducing the row group partials stored in shared memory yields to the two lenslet moments.

D. Kernel Algorithm

Alg. 2 illustrates the steps of each concurrent thread to extract the centroids of a spot image. Lines 6-11 calculate the pixel group partials. To obtain the row partial, the butterfly shuffle reduction is carried out in Line 12. It is worth remarking that also rows from other lenslets in the lenslet group are reduced at the same time. The rows in the row group are scanned in Lines 3-13, accumulating the row partials into the row group partial at

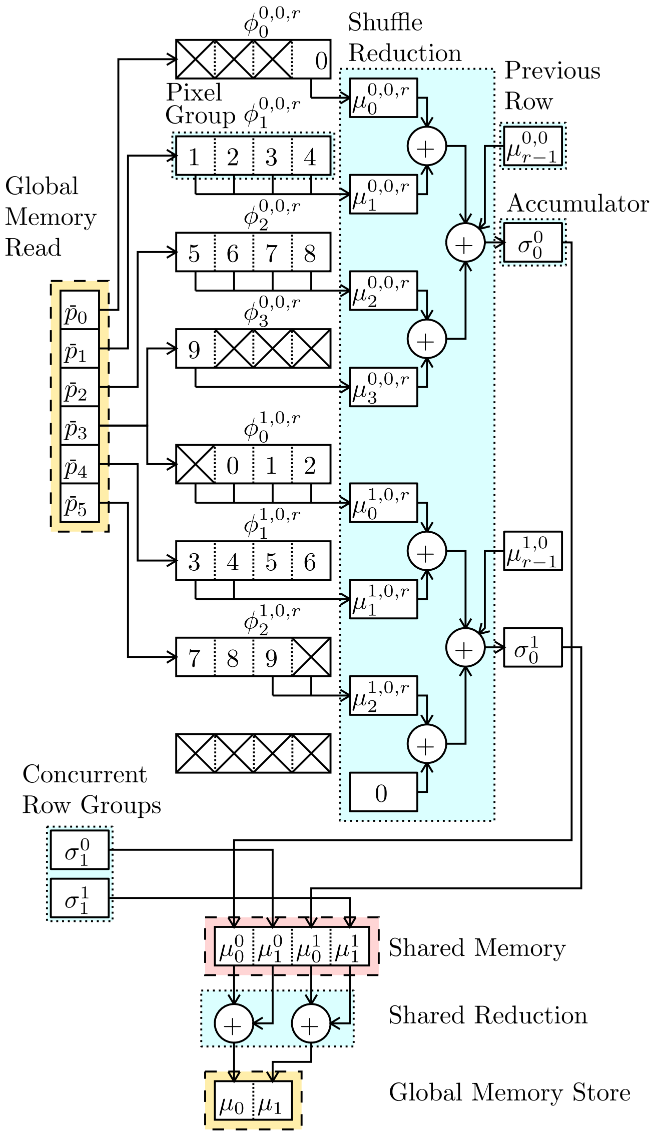

Fig. 4. Diagram of the data reduction. In this example, since the pixel region width (i.e. lenslet pitch) is $d = 10$, it is required to read 6 pixel groups from global memory ($\bar{p}_0$ to $\bar{p}_5$) in order to cover the two lenslet rows. The rows are partitioned into 2 row groups of 5 rows each which are processed independently. Each thread reduces the pixel group into a partial moment. Assuming 8 threads per warp, the pixel groups partials of 2 lenslet rows are reduced at once using the shuffle reduction. The resulting row partial moments are accumulated into the row group partial moments. When all row groups are processed, their results are reduced in shared memory yielding to the 2 lenslet moments and then copied into global memory.

each iteration. The shared reduction over the row group partials performed in Line 16 yields to the lenslet moments.

Remark 2 *Algorithm 2 does not implement the image-processing techniques described in Section 3 for enhancing the SNR ratio. However, since such techniques operate on each pixel individually, they can be introduced by transforming the intensities right after reading them from memory, keeping the same level of concurrency. The added computational cost is therefore negligible. Alternatively, per-pixel processing can be implemented by a look-up table intensity transformation (e.g., fine-tuning the camera intensity mapping).*

Algorithm 2. CUDA Centroids Extraction

```
1: Pixel group $\phi_{\bar{p}}^{l,\bar{r},r}$ ← index inferred from thread context
2: Row group partial $\mu_{\bar{r}}^{l} \leftarrow 0$.
3: for all Rows $\phi_r^{l,\bar{r}} \in$ row group $\phi_{\bar{r}}^{l}$ do
4:     Pixel group $\phi_{\bar{p}}^{l,\bar{r},r}$ ← global memory
5:     Pixel group partial $\mu_{\bar{p}}^{l,\bar{r},r} \leftarrow 0$
6:     for all Pixels $\phi_p^{l,\bar{r},r,\bar{p}} \in$ pixel group $\phi_{\bar{p}}^{l,\bar{r},r}$ do
7:         if Pixel $\phi_p^{l,\bar{r},r,\bar{p}}$ outside $l$ then
8:             Pixel partial $\mu_p^{l,\bar{r},r,\bar{p}} \leftarrow 0$
9:         else
10:            Pixel partial $\mu_p^{l,\bar{r},r,\bar{p}}$ ← moments calculated from intensity value $I(\phi_p^{l,\bar{r},r,\bar{p}})$
11:        Pixel group partial $\mu_{\bar{p}}^{l,\bar{r},r} \leftarrow \mu_{\bar{p}}^{l,\bar{r},r} + \mu_p^{l,\bar{r},r,\bar{p}}$
12:    Row partial $\mu_r^{l,\bar{r}}$ ← butterfly shuffle sum reduction over pixel group partials $\mu_{\bar{p}}^{l,\bar{r},r}$
13:    Row group moment accumulator $\sigma_{\bar{r}}^{l} \leftarrow \sigma_{\bar{r}}^{l} + \mu_r^{l,\bar{r}}$
14: if Pixel group index $\bar{p} \neq 0$ then return
15: Row group moment accumulator $\sigma_{\bar{r}}^{l} \rightarrow$ shared memory $\mu_{\bar{r}}^{l}$
16: Lenslet moments $\mu_l$ ← shared memory sum reduction over row group partials $\mu_{\bar{r}}^{l}$
17: if Row group index $\bar{r} \neq 0$ then return
18: Centroid $c_l \leftarrow \left(\frac{m_l^{10}}{m_l^{00}}, \frac{m_l^{01}}{m_l^{00}}\right)$
19: Centroid $c_l$, lenslet moments $\mu_l$ → global memory
```

5. EXPERIMENTS

In these experiments the centroids extraction is performed on images stored in memory. This choice is motivated by the fact that the exposure and transfer of the image are performed before processing, and hence they have no impact on the execution time of the centroids extraction implementation.

The moment-based centroids extraction routine assumes that the captured image contains one intensity spot per lenslet, as shown in Fig. 2. However, since all pixels must be accessed and processed (with no additional data-dependent conditions), the information contained in the spot image does not impact the execution time. A pool of 8-bit white images ($I = 255$ for all pixels) is fed to both the proposed implementation of Alg. 2 and the sequential implementation of Alg. 1 to test the correctness of our approach. However, the spot image used in the benchmark experiments are synthesized as 8-bit images with each pixel having random intensity ($I \in [0, 255]$) to prevent memory caching optimizations.

The target platform is the NVIDIA Jetson TX2, which integrates a 256-cores Pascal GPU, a dual-core NVIDIA Denver 2 CPU and a quad-core ARM Cortex-A57 CPU. The test-bench runs on the stock Linux distribution that comes with the NVIDIA Jetpack 4.2.1 firmware. The GPU frequency is locked at 1.3 GHz and each CPU core frequency is locked at 2 GHz.

A test run measures the time elapsed from the issue of the extraction command to the transfer of all extracted centroids, averaged over 50 executions on the same spot image.

Different WFS optical aperture diameters ρ, lenslet sizes d_L and pixel sizes d_P are taken into account with the pixel region

Table 1. Execution times of CPU and GPU (t_{CPU}, t_{GPU}) and relative speed-up for the configurations of RoI width w_P and pixel region width d, along with the number of extracted centroids w_L^2.

$w_P[px]$	$d[px]$	w_L^2	$t_{CPU}[\mu s]$	$t_{GPU}[\mu s]$	**Speed-up**
100	3.8	676	**54**	104	0.5192
100	11	81	**44**	59	0.7458
100	20	25	**10**	57	0.1754
100	29	9	**1**	57	0.0175
200	3.8	2704	308	**151**	2.0397
200	11	324	186	**57**	3.2632
200	20	100	158	**57**	2.7719
200	29	36	102	**57**	1.7895
500	3.8	17161	2095	**615**	3.4065
500	11	2025	1268	**208**	6.0962
500	20	625	1220	**165**	7.3939
500	29	289	1106	**107**	10.3364
700	3.8	33856	4194	**1090**	3.8477
700	11	3969	2475	**331**	7.4773
700	20	1225	2405	**265**	9.0755
700	29	576	2228	**162**	13.7531
1000	3.8	69169	8922	**2307**	3.8674
1000	11	8100	5158	**631**	8.1743
1000	20	2500	4266	**481**	8.8690
1000	29	1156	3210	**317**	10.1262

width d and pixels RoI width w_P parameters (as shown in Fig. 2). The RoI ranges from 100×100 to 1000×1000 pixels, while the pixel region resolution ranges from 3×3 to 28×28 pixels. The result of each test run is presented in Tab. 1, where the CPU and GPU execution time t_{CPU}, t_{GPU} for a given combination of d and w_P are compared to calculate the speed-up:

$$\text{Speed-up} = \frac{t_{CPU}}{t_{GPU}}. \quad \textbf{(13)}$$

For every configuration except for the 100×100 RoI size, the GPU implementation results in a speed-up over the CPU implementation from 2 up to 13. Figs. 5, 6, 7, 8 and 9 show the execution time for all parameters combinations. The algorithm execution time takes less than 1 ms for pixel region widths d larger than 5 pixels. Small pixel regions mean that more lenslets fit the same RoI and this leads to a large number of centroids, increasing the execution time. In the case of small RoI size (Fig. 5) the overhead latency when issuing a CUDA kernel launch (experimentally measured to be $50\mu s$ on average with an empty kernel) completely dominates the GPU execution time. The relatively high execution time variance for the CPU implementation is due to the underlying OS task scheduler behavior.

6. CONCLUSION

The NVIDIA Jetson platform is a CPU/GPU hybrid platform which, while compact and power-efficient, is powerful enough to justify its use in edge-computing and HPC. Due to its unified memory architecture, the latency introduced by copying data is avoided. Hence, images are processed as soon as they are transferred from the camera. The Jetson platform is positioned as an alternative to FPGA-based smart cameras, with the advantage of being easier to program and more flexible.

In small-scale AO systems, the SH-WFS is usually implemented on CPU or FPGA architectures. However, despite being

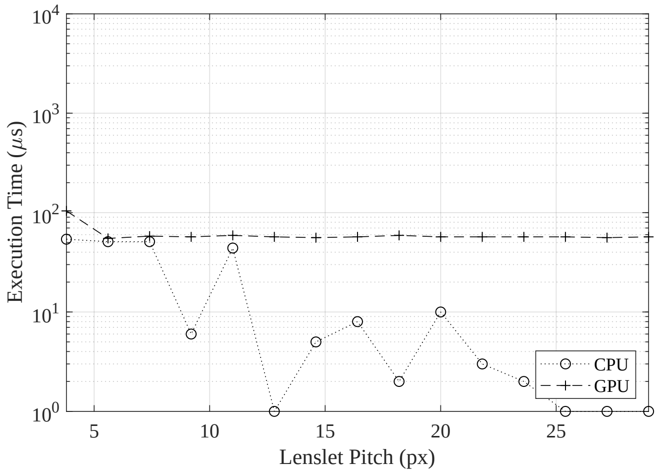

Fig. 5. Execution times of the centroid extraction for a 100×100 RoI of the spot image.

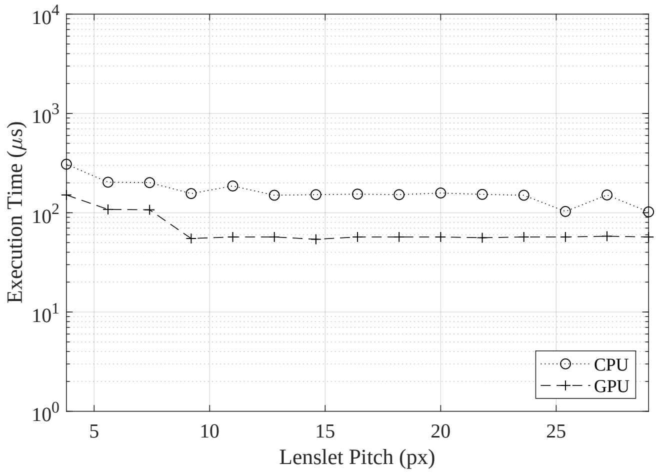

Fig. 6. Execution times of the centroid extraction for a 200×200 RoI of the spot image.

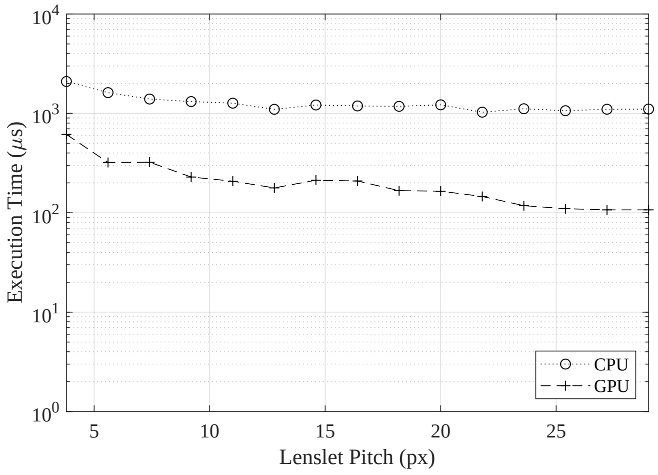

Fig. 7. Execution times of the centroid extraction for a 500×500 RoI of the spot image.

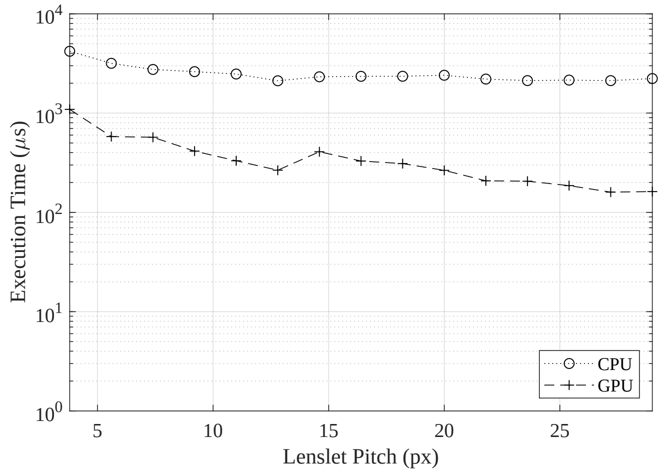

Fig. 8. Execution times of the centroid extraction for a 700 × 700 RoI of the spot image.

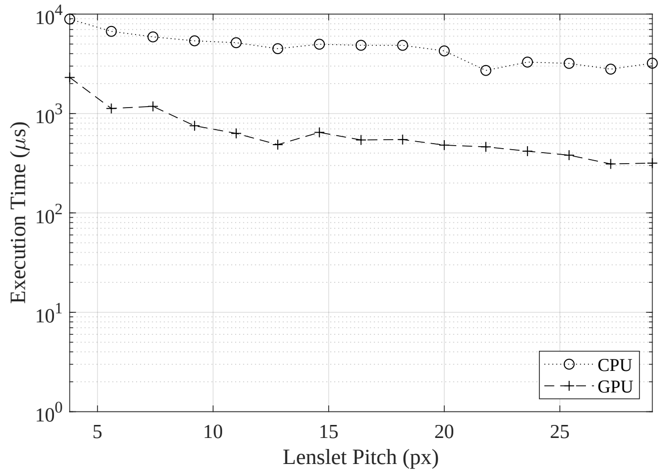

Fig. 9. Execution times of the centroid extraction for a 1000 × 1000 RoI of the spot image.

low-latency, CPU solutions are not portable and FPGA designs lead to long development time. The experimental results carried on the Jetson CPU/GPU platform show that the time required for the centroid extraction is less than 1 ms given a pixel region width larger than 5 pixels, and hence compatible with the AO closed-loop latency constraint. Our approach suggests that an embedded GPU architecture is a valid alternative to FPGA-based SH-WFS solutions. The parallel capabilities of the device can be leveraged to develop advanced wavefront reconstruction schemes, e.g., extended source and high dynamics sensing.

As future work, it is worth highlighting that the full AO control loop can be implemented into the Jetson, leveraging on its GPU computational power to produce the commands for the deformable mirror of small to medium-size AO systems. Since deformable mirrors in such systems have tens to a few hundreds actuators, the resolution of the Shack-Hartmann grid needed to image the deformable mirror into the measured wavefront ranges from hundreds to thousands lenslets. Depending on the AO setup requirements, the proposed centroids extraction algorithm can be adapted to extract the same number of centroids from a small image with small lenslet pitch (e.g. low light condition) or a larger image with larger pitch (i.e. to improve detection accuracy). Taking into account the transfer delay of a USB3 camera and the actuators interface, the wavefront-to-command latency of a standard Proportional-Integral array (e.g., one PI regulator per wavefront mode) can be quantifiable to be smaller than 2ms. More effective control techniques like modal control and Predictive/Optimal Control can take advantage of the GPU parallelism to be computationally feasible. Furthermore, the spatio-temporal dynamics of atmospheric aberrations can be learned and predicted by on-board machine learning algorithms, routinely updating the controller parameters.

DISCLOSURES

The authors declare no conflicts of interest.

REFERENCES

1. R. Tyson, *Adaptive Optics Engineering Handbook*, vol. 10 (CRC Press, 1999).
2. M. Quintavalla, J. Mocci, R. Muradore, A. Corso, and S. Bonora, "Adaptive optics on small astronomical telescope with multi-actuator adaptive lens," in *Free-Space Laser Communication and Atmospheric Propagation XXX*, vol. 10524 (International Society for Optics and Photonics, 2018), p. 1052414.
3. P. Zhang, J. Mocci, D. J. Wahl, R. K. Meleppat, S. K. Manna, M. Quintavalla, R. Muradore, M. V. Sarunic, S. Bonora, E. N. Pugh Jr *et al.*, "Effect of a contact lens on mouse retinal in vivo imaging: Effective focal length changes and monochromatic aberrations," Exp. eye research **172**, 86–93 (2018).
4. M. Quintavalla, P. Pozzi, M. Verhaegen, H. Bijlsma, H. Verstraete, and S. Bonora, "Adaptive optics plug-and-play setup for high-resolution microscopes with multi-actuator adaptive lens," in *Multiphoton Microscopy in the Biomedical Sciences XVIII,* vol. 10498 (International Society for Optics and Photonics, 2018), p. 104981X.
5. S. A. Moosavi, M. Quintavalla, J. Mocci, R. Muradore, H. Saghafifar, and S. Bonora, "Improvement of coupling efficiency in free-space optical communication with a multi-actuator adaptive lens," Opt. letters **44**, 606–609 (2019).
6. M. Negro, M. Quintavalla, J. Mocci, A. G. Ciriolo, M. Devetta, R. Muradore, S. Stagira, C. Vozzi, and S. Bonora, "Fast stabilization of a high-energy ultrafast OPA with adaptive lenses," Sci. reports **8** (2018).
7. S. Thomas, T. Fusco, A. Tokovinin, M. Nicolle, V. Michau, and G. Rousset, "Comparison of centroid computation algorithms in a Shack-Hartmann sensor," Mon. Notices Royal Astron. Soc. **371**, 323–336 (2006).
8. F. Kong, M. C. Polo, and A. Lambert, "Centroid estimation for a Shack-Hartmann wavefront sensor based on stream processing," Appl. Opt. **56**, 6466 (2017).
9. A. Vyas, M. B. Roopashree, and B. R. Prasad, "Optimization of existing centroiding algorithms for Shack-Hartmann sensor," Arxiv preprint arXiv:0908.4328 p. 6 (2009).
10. J. Mompeán, J. L. Aragón, P. M. Prieto, and P. Artal, "Gpu-based processing of hartmann–shack images for accurate and high-speed ocular wavefront sensing," Futur. Gener. Comput. Syst. **91**, 177–190 (2019).
11. L. Hu, S. Hu, W. Gong, and K. Si, "Learning-based shack-hartmann wavefront sensor for high-order aberration detection," Opt. Express **27**, 33504–33517 (2019).
12. D. Perret, M. Lainé, J. Bernard, D. Gratadour, and A. Sevin, "Bridging FPGA and GPU technologies for AO real-time control," Adapt. Opt. Syst. V **9909**, 99094M (2016).
13. M. Lainée, A. Sevin, D. Gratadour, J. Bernard, and D. Perret, "A GPU based RTC for E-ELT adaptive optics: Real time controller prototype," AO4ELT5 (2018).
14. R. Á. F. Díaz, J. L. C. Rolle, N. R. Gutiérrez, and F. J. de Cos Juez, "Using GPUs to speed up a tomographic reconstructor based on machine learning," in *International Joint Conference SOCO'16-CISIS'16-*

ICEUTE'16: San Sebastián, Spain, October 19th-21st, 2016 Proceedings, vol. 527 (Springer, 2016), p. 279.
15. F. Ferreira, D. Gratadour, A. Sevin, and N. Doucet, "COMPASS: An efficient GPU-based simulation software for adaptive optics systems," Proc. - 2018 Int. Conf. on High Perform. Comput. Simulation, HPCS 2018 pp. 180–187 (2018).
16. J. Beck and J. P. Bos, "Open source acceleration of wave optics simulations on energy efficient high-performance computing platforms," Long-Range Imaging II **10204**, 102040F (2017).
17. F. L. Rosa, J. G. Marichal-Hernandez, and J. M. Rodriguez-Ramos, "Wavefront phase recovery using graphic processing units (GPUs)," Opt. Atmospheric Propag. Adapt. Syst. VII **5572**, 262 (2004).
18. V. Venugopalan, "Evaluating latency and throughput bound acceleration of FPGAs and GPUs for adaptive optics algorithms," 2014 IEEE High Perform. Extrem. Comput. Conf. HPEC 2014 pp. 1–6 (2014).
19. W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu, "Edge computing: Vision and challenges," IEEE Internet Things J. **3**, 637–646 (2016).
20. F. Dias, F. Berry, J. Sérot, and F. Marmoiton, "Hardware, design and implementation issues on a FPGA-based smart camera," in *2007 First ACM/IEEE International Conference on Distributed Smart Cameras,* (IEEE, 2007), pp. 20–26.
21. R. Ragazzoni, C. Arcidiacono, E. Diolaiti, J. Farinato, A. M. Moore, and R. Soci, "A smart fast camera," in *Ground-based Instrumentation for Astronomy,* vol. 5492 (International Society for Optics and Photonics, 2004), pp. 121–127.
22. M. Thier, R. Paris, T. Thurner, and G. Schitter, "Low-latency Shack-Hartmann wavefront sensor based on an industrial smart camera," IEEE transactions on instrumentation measurement **62**, 1241–1249 (2012).
23. S.-H. Lee and C.-S. Yang, "A real time object recognition and counting system for smart industrial camera sensor," IEEE Sensors J. **17**, 2516–2523 (2017).
24. M. Carraro, M. Munaro, and E. Menegatti, "Cost-efficient RGB-D smart camera for people detection and tracking," J. Electron. Imaging **25**, 041007 (2016).
25. Z. Zhao, Z. Jiang, N. Ling, X. Shuai, and G. Xing, "ECRT: An edge computing system for real-time image-based object tracking," in *Proceedings of the 16th ACM Conference on Embedded Networked Sensor Systems,* (ACM, 2018), pp. 394–395.
26. Y. Ukidave, D. Kaeli, U. Gupta, and K. Keville, "Performance of the NVIDIA Jetson TK1 in HPC," in *2015 IEEE International Conference on Cluster Computing,* (IEEE, 2015), pp. 533–534.
27. J. Mocci, M. Quintavalla, C. Trestino, S. Bonora, and R. Muradore, "A multiplatform CPU-based architecture for cost-effective adaptive optics systems," IEEE Transactions on Ind. Informatics **14**, 4431–4439 (2018).
28. A. M. Nightingale and S. V. Gordeyev, "Shack-hartmann wavefront sensor image analysis: a comparison of centroiding methods and image-processing techniques," Opt. Eng. **52**, 071413 (2013).
29. J. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable parallel programming with cuda," Queue **6**, 40–53 (2008).
30. NVIDIA Corporation, "NVIDIA CUDA C programming guide," (2020). Version 11.0.