PETER FRANK PERRONI

POINT SPREAD FUNCTION ESTIMATION OF SOLAR SURFACE IMAGES WITH A COOPERATIVE PARTICLE SWARM OPTIMIZATION ON GPUS

Dissertação apresentada como requisito parcial à obtenção do grau de Mestre. Programa de Pós-Graduação em Informática, Setor de Ciências Exatas, Universidade Federal do Paraná.
Orientador: Prof. Dr. Daniel Weingaertner

CURITIBA

2013

PETER FRANK PERRONI

POINT SPREAD FUNCTION ESTIMATION OF SOLAR SURFACE IMAGES WITH A COOPERATIVE PARTICLE SWARM OPTIMIZATION ON GPUS

Dissertação apresentada como requisito parcial à obtenção do grau de Mestre. Programa de Pós-Graduação em Informática, Setor de Ciências Exatas, Universidade Federal do Paraná.
Orientador: Prof. Dr. Daniel Weingaertner

CURITIBA

2013

CONTENTS

LIST OF FIGURES

LIST OF TABLES

INDEX

RESUMO

Apresentamos um método para a estimativa da função de espalhamento pontual (PSF) de imagens de superfície solar obtidas por telescópios terrestres e corrompidas pela atmosfera. A estimativa é feita obtendo-se a fase da frente de onda usando um conjunto de imagens de curta exposição, a reconstrução de granulado óptico do objeto observado e um modelo PSF parametrizado por polinômios de Zernikes. Estimativas da fase da frente de onda e do PSF são computados através da minimização de uma função de erro com um método de otimização cooperativa por nuvens de partículas (CPSO), implementados em OpenCL para tirar vantagem do ambiente altamente paralelo Um método de calibração é apresentado para ajustar os parâmetros do que as unidade de processamento gráfico (GPU) provém. algoritmo para resultados de baixo custo, resultando em sólidas estimativas tanto para imagens de baixa frequência quanto para imagens de alta frequência. Os resultados mostram que o método apresentado possui rápida convergência e é robusto a degradação causada por ruídos. Experimentos executados em uma placa NVidia Tesla C2050 computaram 100 PSFs com 50 polinômios de Zernike em $\approx$ 36 minutos. Ao aumentar-se o número de coeficientes de Zernike dez vezes, de 50 para 500, o tempo de execução aumentou somente 17%, o que demonstra que o algoritmo proposto é pouco afetado pelo número de Zernikes utilizado.

ABSTRACT

We present a method for estimating the point spread function (PSF) of solar surface images acquired from ground telescopes and degraded by atmosphere. The estimation is done by retrieving the wavefront phase using a set of short exposures, the speckle reconstruction of the observed object and a PSF model parametrized by Zernike polynomials. Estimates of the wavefront phase and the PSF are computed by minimizing an error function with a cooperative particle swarm optimization method (CPSO), implemented in OpenCL to take advantage of highly parallel graphical processing units (GPUs). A calibration method is presented to adjust the algorithm parameters for low cost results, providing solid estimations for both low frequency and high frequency images. Results show that the method has a fast convergence and is robust to noise degradation. Experiments run on an NVidia Tesla C2050 were able to compute 100 PSFs with 50 Zernike polynomials in $\approx$ 36 minutes. The increase on the number of Zernike coefficients tenfold, from 50 to 500, caused the increase of 17% on the execution time, showing that the proposed algorithm is only slightly affected by the number of Zernikes used.

CHAPTER 1

INTRODUCTION

As the technology improves and the hardware becomes faster and cheaper, compute intensive techniques that previously were impractical are now feasible to be implemented. On images taken from telescopes where the object is found somewhere in the other side of the atmosphere, a post correction process is a mandatory step that could be very expensive depending on the desired result. Therefore, there is a demand for robust and reliable techniques that require as few input data as possible, and that could provide results quickly enough to be used as a daily tool.

The method proposed here is based on the work developed at [53], where a new method was created for estimating the point spread function (PSF) of solar surface images acquired from ground telescopes and degraded by atmosphere. The estimation is done by retrieving the wavefront phase using a set of short exposures, the speckle reconstruction of the observed object and a PSF model parametrized by Zernike polynomials. The PSF is estimated by minimizing an error function that relies on the quality of the Zernike coefficients used to compute the wavefront phase. The original work implements the Simulated Annealing (SA) to optimize the PSF that is used to recover the corrupted image.

This work is motivated by the fact that the daily flow of images generated by an observatory is very high (see section 5.4 for details), then the reduction of the post processing time is of special interest for these institutions. The currently available solution [53] might takes days to process the images obtained in one day of observation, even when increasing the number of machines to assist on the processing. On current method, one set of 100 images takes about 3 hours to process on a cluster of 23 dual core servers Dell Poweredge 1950, what means that the post processing of a full set of images generated in one day (1000 images) would take 30 hours to complete. Furthermore, as the number of Zernikes is increased on the estimations, the processing time is linearly raised, what means that the implementation of the atmosphere simulation while estimating the PSF is currently unfeasible, since this kind of simulation usually requires a high number of Zernikes.

Given the computation of the Fourier transform is a CPU intensive task, the main problem with this PSF estimation method is the number of Fourier transforms required to estimate each PSF. Considering the PSF parameters are unknown, a demanding optimization method is also required to estimate such parameters. As consequence, the entire estimation process takes too long for a real life application when processing on CPUs. To reduce the estimation time, as well as keeping good final results, a more suitable optimization method must be chosen and cheaper parallel hardwares must be used, so that this method can be considered as a post processing tool on observatories' daily work.

The objective of this work is:

- Develop a parallel solution to reduce the time required to estimate PSFs of solar surface images, using Zernike polynomials to compute the PSFs;
- Use the Cooperative Particle Swarm Optimization (CPSO) to estimate the Zernike coefficients, since this optimization method is well known for its quick convergence, noise robustness and good solutions on high dimensional problems;
- Implement the entire solution on OpenCL language, to take advantage of the low-cost easy-acquisition graphical processing unit (GPU) hardwares available from multiple vendors.

We present a new method to quickly generate good Zernike coefficients for the estimation of PSFs of solar surface images. The optimization method CPSO was used to minimize the error function. The calculations required to compute the PSF, the CPSO and the error function were implemented in OpenCL to take advantage of highly parallel environments provided by heterogeneous GPU devices. A calibration method is described to adjust the CPSO algorithm parameters for low cost results, given a well adjusted optimization not only requires less processing time but also obtains better estimations. The complete implementation of the method presented here can be found at `http://web.inf.ufpr.br/vri/alumni/peter-frank-perroni-msc-2013`.

The results obtained by the proposed PSF estimation method with CPSO show that the post processing time of the images obtained in one day of observation takes about 5h53m running on one Intel i7-975 CPU with 1 GPU NVIDIA Tesla C2050. Good PSFs were estimated even under high presence of noise, and stable results were obtained for both the best and the worst cases. Both low-frequency and high-frequency images obtained similar good resulting estimations, meaning that this method is not restricted to the estimation of PSFs for solar images. The results also show that a tenfold increase in the number of Zernkes (from 50 to 500) increased only 17% the running time, what means that the number of Zernikes used for the PSF estimation caused only a slight impact on the overall processing time.

In practical terms, this means that the method developed here can reduce in many ways the costs associated with the post processing time for the image correction (eg., hardware, energy and waiting time), besides providing additional options for optimizing complex and high dimensionality problems through a calibrated CPSO.

Notice that the physics involved on this method is not the focus of the work developed here, therefore only a general description of the fundamental math is presented and no further discussion is taken in that sense (introduction on chapter 2 and section 2.1).

As previously stated, the baseline for the PSF estimation method presented here is described on [53]. The Kiepenheuer-Institut für Sonnenphysik (Kiepenheuer Institute for Solar Physics - KIS[1]) provided an implementation of the Simulated Annealing as the optimization method for the PSF estimations, besides images for the progress of this work. This SA implementation was studied in details and its strengths and weakness were considered while choosing the CPSO as optimization method for this work.

This work is organized on the following sequence: the next chapter (2) presents the theoretical basis for the development of this work; the chapter 3 describes metaheuristics, its parallel implementations and provides an overview of PSF estimation methods; the chapter 4 shows the results obtained for this PSF estimation method with Simulated Annealing and presents the proposed Cooperative PSO for OpenCL devices; the PSF

[1]http://www.kis.uni-freiburg.de/

Estimation results with CPSO are shown on chapter 5, where a CPSO calibration method is proposed and the results obtained for every experiment are shown and discussed. The conclusions and final considerations are discussed on chapter 6.

CHAPTER 2

THEORETICAL BASIS

In ground-based solar observation systems, turbulence caused by atmosphere events, like solar heating during daylight hours, is a well known problem that causes inhomogeneities in atmospheric temperature and density that, in turn, affect its diffraction index. As consequence of this effect, astronomic images captured by ground telescopes suffer from severe distortions of the original image, a phenomenon that is called “image speckle” [45].

Regardless of their purpose, all images are produced to record information. Since the image formation and the recording processes are always imperfect, a recorded image represents invariably a degraded version of the original scene. The two principal degradation types, blur and noise, act in an image formation process:

- Blur can be originated from events like the relative motion between the optical system and scene, from an out-of-focus optical system or from atmospheric turbulence in aerial images [20].
- Noise is an ever-present effect caused by many factors, like the imaging equipment itself and optical defects from instrumentation [54].

The spatial impulse response function, commonly referred to as the Point Spread Function (PSF), is used to represent such distortions in the recorded image (Figure 2.1).

Adaptive optics (AO) is a technique commonly used to overcome these distortions, where the atmospheric effect is corrected in real time by a deformable mirror whose shape

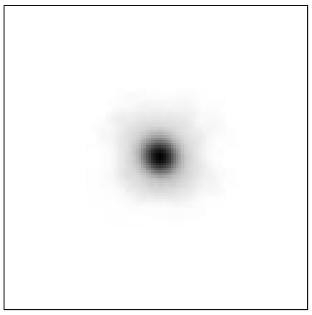


Figure 2.1: Example of a Point Spread Function.

is continuously updated to match the current state of the atmospheric turbulence. The correction, however, is never perfect and the long exposure images acquired with an AO system are still affected by a residual blur which reduces the contrast of the fine details. This residual blur is of course much less severe than the aberrations in the uncorrected images and may be further reduced by means of post data processing [52].

Most techniques used for post-processing images are Speckle Interferometry [58], Phase Diversity [32] and Multi-Object Multi-Frame Blind Deconvolution [51]. However, when the instantaneous PSF of a short exposed image is known, non blind deconvolution techniques can be used to obtain superior image quality, besides providing better understanding of the solar AO system performance [53].

Having estimated a set of instantaneous PSFs using short exposures images, a long exposure PSF is computed as being the mean of these instantaneous PSFs. The long exposure PSF so constructed can be used to deconvolve long exposures for real images. The problem in using this technique is that estimating instantaneous or long exposure PSFs is a demanding task, because the information about the image degradation is always incomplete [35] (e.g., when using AO system, only a limited number of Zernike modes can be measured).

The objective of an image restoration process is to reduce the image blur occurred during the imaging process, but because the restoration will enlarge the noises, denoising is an interesting step to obtain a better visual effect [60]. Given that the images recorded with a telescope are always corrupted by noise and that the estimation of an instantaneous PSF corresponds to a division in the Fourier domain, the presence and (pre/post) treatment of such noise is a serious issue. However, the technique used in this work does not require additional pre-processing over the corrupted image for direct removal of the noise before estimating the PSF.

In the framework of Speckle-Interferomety, the Fourier-components of the observed object are estimated using a set of short exposed images. Suitable methods are employed in order to separately estimate the Fourier-amplitudes (based on Labeyries-Method [31] and the extension of Wöger [[57] and references therein]) and the Fourier-phases (based

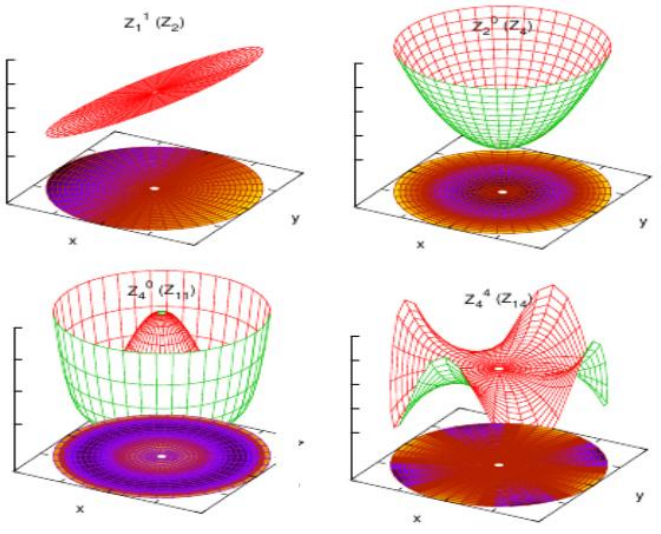


Figure 2.2: Example of Zernikes terms.
This figure shows examples of the Zernike terms Z2, Z4, Z11 and Z14.

on polyspectra of the data, e.g., Triple-correlation methods [33, 55] or Extended Knox-Thomson methods [30, 8] (the reference [36] describes the implementation of extended Knox-Thomson method in detail)).

For the purpose of this technique, the Zernike polynomials are used to estimate how distorted is the image compared to the original object. The Zernike polynomials are a set of 2D orthonormal basis functions that can be used to represent aberrations in the image formation process [54]. They represent the statistical eigenfunctions of optical distortions that quantitatively classify each aberration by using a set of polynomials. The linear combination of K aberrations results in an approximation of the perturbation K. Each term contains the appropriate amount of each lower order term to make it orthogonal to each lower order term [59]. Every term results in a specific aberration of the PSF, e.g., Z_0 is a constant called of "piston" term, Z_1 and Z_2 are tilt terms, Z_3 represents focus and Z_4 is astigmatism plus defocus (Figure 2.2). When $K = \infty$, it is an exact representation of the aberrations.

The next section will explain how this PSF estimation method calculates the PSF based on the Zernike terms, and how the fitness value (calculated by the error function) is obtained. The subsequent sections will explain the original optimization algorithm used for the estimations (Simulated Annealing, section 2.2) and the algorithm proposed to accelerate the results (Cooperative Particle Swarm Optimization, sections 2.3 and 2.4). Finally, the section 2.5 will give an overview of OpenCL, which is the language used to develop the GPU code of this work.

2.1 PSF Estimation Method

The image degradation process can be modeled as a degradation function $P(x)$, together with an additive noise $N(x)$, operating on an input object $O(x)$ to produce a degraded image $I(x)$. As a result of the degradation process and noise inter-fusion, the original image becomes a degraded image, representing image blur in different degrees [23]:

$$I(x) = O(x) \otimes P(x) + N(x) \tag{2.1}$$

where $\otimes$ represents the convolution of the object $O(x)$ with the degradation function $P(x)$ and x is a coordinate in the focal plane of the telescope.

According to the convolution theorem illustrated in the Figure 2.3, the convolution of two spatial functions is denoted by the product of their Fourier transforms in the frequency domain:

$$i(s) = o(s) \cdot p(s) + n(s) \tag{2.2}$$

where $i(s)$, $o(s)$, $p(s)$ and $n(s)$ are the Fourier transform of the image, of the object, of the PSF and of the noise, respectively.

The instantaneous PSF $P(x)$ is given by the modulus of the Fourier transform of the optical field $[A(u) = A_0(u)exp(i\phi(u))]$ at the telescope aperture:

$$P(x) = |F(A_0(u)exp(i\phi(u)))|^2 \tag{2.3}$$

where u is a coordinate in the telescope aperture and F denotes the Fourier transformation. We assume the transmission $A_0(u)$ of the telescope aperture to be constant, i.e., we neglect scintillation effects and obstructions. Furthermore, $A_0(u)$ is tuned such that the PSF conserves energy.

The phase $\phi(u)$ of the optical field is parametrized by using n Zernike terms:

$$\phi(u) = \sum_{i=1}^{n} \gamma_i Z_i(u) \tag{2.4}$$

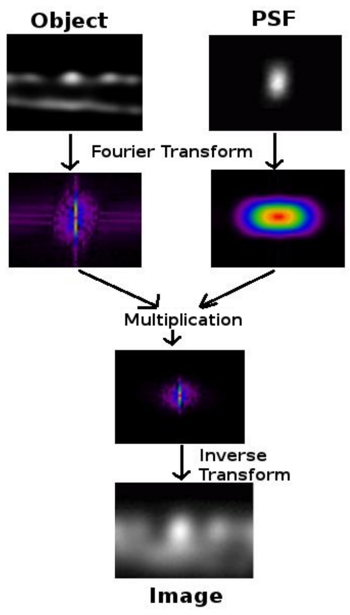


Figure 2.3: Convolution process.
The Fourier Transform of both the Object and the PSF are multiplied together, then the Inverse transform is applied to the result, obtaining the Convolved Object.

where every γ_i is a coefficient to be applied to the respective Zernike term as its weight. The summation in the equation (2.4) can be graphically represented as in the Figure 2.4 (for $n = 8$), where each summed circle can be thought of as a Zernike term multiplied by its weight, giving as result (top left circle) a phase that would represent each kind of aberration present in the image [49].

Given the parametrization of the wavefront phase (Equation 2.4), the Equation 2.3 as a model for the instantaneous PSF, and the imaging model (Equation 2.2), we estimate an instantaneous PSF by estimating a set of n Zernike expansion coefficients γ that minimizes the error function in the Equation 2.5 [18]:

$$E(\gamma) = \sum_{|s| \leq sc} |i(s) - \overline{o}(s)\overline{p}(s)|^2 \tag{2.5}$$

where sc denotes the cut-off frequency of the telescope, and $\overline{o}(s)$ and $\overline{p}(s)$ denote the Estimates of the Fourier Transform of the object and of the PSF, respectively. Since $i(s)$ is directly calculated from the input image and $\overline{o}(s)$ is estimated through speckle reconstruction, $\overline{p}(s)$ is the only value that ultimately has to be estimated.

At this PSF estimation method, no filtering of the image $I(x)$ is used since the algo-

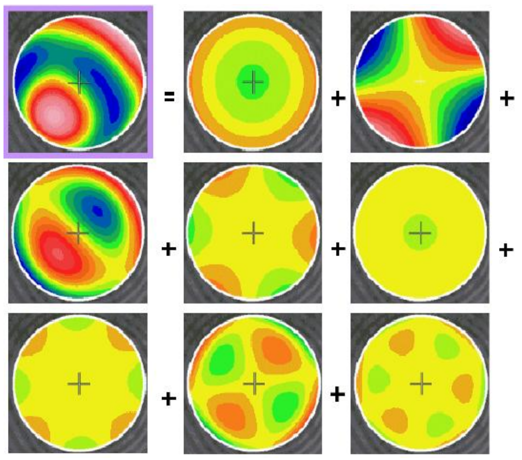


Figure 2.4: Graphical representation of the phase calculation.
The upper left circle represents the sum of all 8 weighted Zernikes, where each Zernike Z_i is multiplied by its respective coefficient γ_i (as described on equation 2.4).

rithm inherently deals with any noise $N(x)$ that is present in the image and its Fourier transform.

To improve the ability of a Fourier to extract spectral data, a Windowing Function can be used. When a windowing function is applied over the data before processing its Fast Fourier Transform (FFT), an effect called Leakage is reduced. Leakage occurs when a FFT is applied over the image, causing the spectral information show up at the wrong frequencies [16]. In this work, the Hanning function [25] is used to create a window for FFT filtering. The Hanning function can be written as described by Equation 2.6, where N is the number of samples in the signal and n is the position within the signal.

$$w(n) = 0.5 + 0.5cos\left(\frac{2\pi n}{N}\right), 0 \leq n \leq N-1 \tag{2.6}$$

A common way to compare the contrast of two different images is by calculating their Root Mean Square (RMS) contrast, since it does not depend on the image's spatial frequency or the image's spatial distribution of contrast [39]. The RMS contrast is defined by Equation 2.7:

$$RMS = \sqrt{\frac{1}{n-1}\sum_{i=1}^{n}(x_i - \overline{x})^2} \tag{2.7}$$

$$\overline{x} = \frac{1}{n}\sum_{i=1}^{n} x_i \tag{2.8}$$

where x_i is a normalized gray-level value such as $0 \leq x_i \leq 1$ and $\overline{x}$ is the mean normalized gray-level (Equation 2.8). To validate numerically how near the restored image was from the original object, the RMS contrast was used in this work to compare the contrasts of the original object O, the corrupted image I and the deconvolved image.

2.2 Simulated Annealing

The global minimization of the error function (Equation 2.5) for this PSF estimation method was originally obtained by an implementation of the Simulated Annealing Algorithm [53]. Simulated Annealing (SA) is an algorithm that simulates a collection of atoms in equilibrium at a given temperature [29]. At every iteration of this algorithm, one atom is randomly displaced and the resulting change in the energy of the system, ΔE, is computed. If $\Delta E \leq 0$, the displacement is accepted and the configuration with the displaced atom is used as the starting point for the next iteration. When $\Delta E > 0$, the probability that the configuration is accepted is described by the Equation 2.9.

$$P(\Delta E) = \exp\left(\frac{-\Delta E}{T}\right) \tag{2.9}$$

where T is the heat bath temperature at a given time.

Random numbers uniformly distributed in the interval [0..1] are a convenient means of implementing the random part of the algorithm. One such number is selected and then compared with $P(\Delta E)$:

- if it is less than $P(\Delta E)$, the new configuration is retained and becomes the new original configuration;
- if not, then the previous original configuration is used to start the next iteration.

By repeating this basic steps many times, one simulates the thermal motion of atoms in thermal contact with a heat bath at temperature T.

For the purpose of the PSF estimation, the ΔE is computed as the difference from the previous calculated error function $E(\gamma)$ (Equation [18]) to the current calculated one. Every error function is computed based on a set of n random coefficients $\gamma_i (i = 1..n)$.

The Algorithm 1 describes how the PSF estimation technique was implemented with SA. The original source code was provided by the Kiepenheuer-Institut für Sonnenphysik (Kiepenheuer Institute for Solar Physics - KIS) and was used as base for the development of this work. Because the SA algorithm enforces only small displacements, for this specific optimization method only one of the coefficients γ_i (Equation 2.4) is changed at every iteration (Algorithm 1, line 7), what means it requires n iterations to have all the coefficients effectively changed and evaluated.

Algorithm 1 PSF Estimation by Simulated Annealing

```
1: Read Input: $\{I(x), O(x)$, number of Zernikes, number of evaluations$\}$
2: Calculate $i(s)$
3: Calculate $o(s)$
4: Calculate the cooling schedule based on number of evaluations {cooling cycle and
   cooling step tables}
5: for every cooling cycle do
6:   for every cooling step do
7:     for every $\gamma_i$ {One single Zernike term is evaluated at every iteration} do
8:       Calculate the new phase $\phi$(u)
9:       Calculate the new psf $P(x)$ and its Fourier $\overline{p}(s)$
10:      Calculate the error function $E(\gamma)$
11:      if accept new configuration based on $\Delta$E {Equation 2.9} then
12:        Save new parameter set $\gamma$ and the PSF $P(x)$
13:      end if
14:    end for
15:  end for
16: end for
17: return The estimated parameter set $\gamma$ and the PSF $P(x)$
```

Given the SA algorithm requires a very slow cooling (lines 5 and 6 of Algorithm 1), good results are usually obtained only after a long processing time, what makes the SA a non feasible solution for a quick minimization of the error function (Equation 2.5).

The section 4.1 describes the initial results obtained by using the SA to estimate the PSFs.

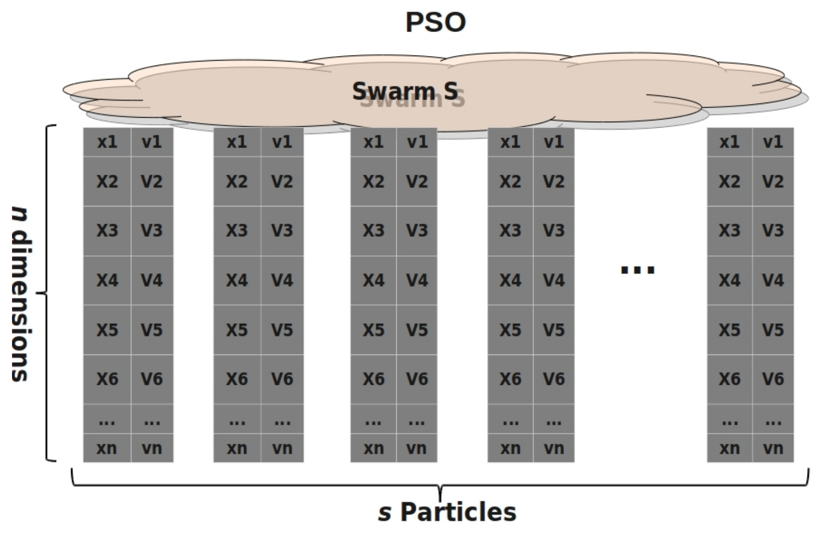


Figure 2.5: Particle Swarm Optimization.
The PSO is composed of particles and each particle is composed of two vectors: position x and velocity v. Every dimension inside a particle represents a variable of the problem being optimized.

2.3 Particle Swarm Optimization

Created in 1995 by Kennedy and Eberhart, the Particle Swarm Optimization (PSO) algorithm is a stochastic (nondeterministic) optimization technique based on the behavior and dynamic of a bird flock. It is used in optimization problems like neural networks and minimization functions, where the objective is to find good regions of the search space instead of the best possible result. This algorithm has the characteristic of quickly converging to a local minima [11].

The PSO population is represented by a swarm composed of particles, where every single particle contains a candidate solution for the optimization problem (Figure 2.5). At every iteration, the particle moves towards its best personal solution found so far, at same time as it moves towards the best global solution found so far by the entire swarm. This behavior causes the good information to be shared among the members of the population, contributing to the cognitive knowledge of the particles through the social knowledge of the swarm [27].

The PSO consists of one best global position $\hat{y}$ ($gbest$) and a population P of s particles, where every particle has n dimensions and each dimension is composed of the attributes: the current position in the search space $x_{i,j}$ ($i = 1..s$, $j = 1..n$), the current velocity $v_{i,j}$ and the best personal position in the search space $y_{i,j}$ ($pbest_{i,j}$) [50].

$$\begin{aligned} v_{i,j}(t+1) = wv_{i,j}(t) \quad &+ c_1 r_{1,i}(t)[y_{i,j}(t) - x_{i,j}(t)] \\ &+ c_2 r_{2,i}(t)[\hat{y}_j(t) - x_{i,j}(t)] \end{aligned} \tag{2.10}$$

$$x_i(t+1) = x_i(t) + v_i(t+1) \tag{2.11}$$

The attributes $x_{i,j}$ and $v_{i,j}$ are usually randomly initialized and $y_{i,j}$ is calculated before the first iteration begins. The parameters c_1 and c_2 are the acceleration coefficients that control the particle direction, the parameter w is the amount of inertia preserved from the previous iteration, and r_1 and r_2 are two random values in the interval [0..1] that are generated at each iteration for every particle. At every iteration t, all dimensions $j \in [1..n]$ of the particles must be updated by the Equations (2.10) and (2.11). The Equations (2.12) and (2.13) show the *pbest* and *gbest* calculation (respectively) for a minimization function f.

$$y_i(t+1) = \begin{cases} y_i(t), & \text{if } f(x_i(t+1)) \geq f(y_i(t)), \\ x_i(t+1), & \text{if } f(x_i(t+1)) < f(y_i(t)). \end{cases} \tag{2.12}$$

$$\hat{y}(t+1) = arg \underset{y_i}{min} \ f(y_i(t+1)), \quad 1 \leq i \leq s \tag{2.13}$$

To limit the movement of the particles and avoid that they escape of the search space, the velocity limit $[-v_{max}, v_{max}]$ and the search area limit $[x_{min}, x_{max}]$ are defined.

A Linear Decreasing Weight (LDW) is a strategy used to control the inertia of the particles, obtaining a better performance from the search [61], what is described by the Equation (2.14):

$$w = (w_{ini} - w_{end}) \left(\frac{T_{max} - t}{T_{max}} \right) + w_{end} \tag{2.14}$$

where t is the current iteration time, T_{max} is the largest iteration time, w_{ini} is the initial inertia weight and w_{end} is the inertia weight when T_{max} is achieved. Studies have shown that the use of a linearly decreasing inertia weight coefficient (w) gives better results than a fixed w [26].

A mechanism to increase the particle activity while exploring the search space is the Re-initialization of Inactive Particles [24] (reset particle). This mechanism keeps a higher level of activity and variety of the particles such that better results can be obtained. This is done by calculating, for each particle, the euclidean distance from the previous position to the current position after the move. If the distance is less than a specified parameter (e.g., 15% of the search space size), then one of the following actions are randomly selected to ensure the convergence of the strategy:

- The particle is reset to have new values for all its variables (x, v, w, c_1 and c_2);
- Only the parameters are reset (w, c_1 and c_2) and the particle attributes (x and v) are not changed.

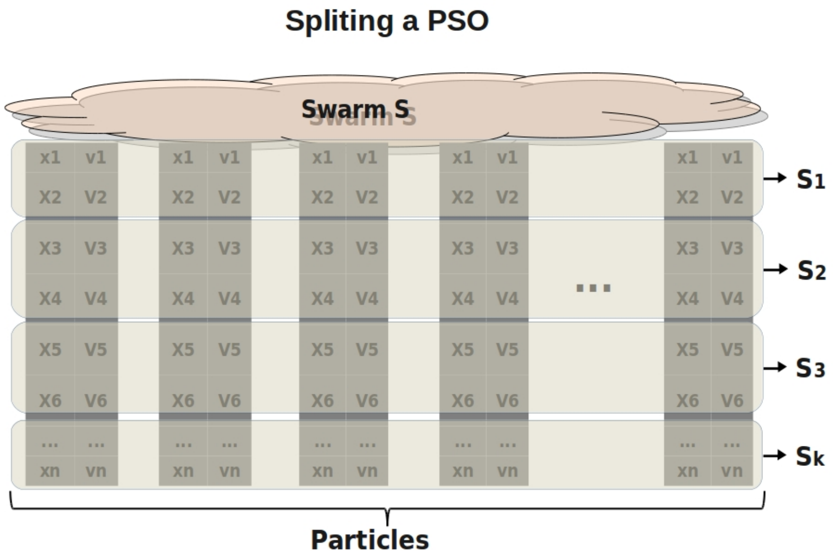


Figure 2.6: Converting a PSO into a CPSO.
The original PSO has its dimensions subdivided into k smaller swarms.

2.4 Cooperative PSO

For problems with high dimensional search space, just like many other optimization techniques, the PSO has its performance reduced by the *Curse of Dimensionality* , which simply put, implies that the performance deteriorates as the dimensionality of the search space increases. One direct solution for this problem is to divide the dimensions into K partitions (Figure 2.6) [50]. Each partition is managed by a different swarm and every swarm cooperates with each other, creating a Cooperative Particle Swarm Optimization (CPSO) (Figure 2.7).

The main difference to the classical PSO is the presence of a context function $\boldsymbol{b}$ (Algorithm 2, line 1), where $(P_1.\hat{y}, \cdots P_k.\hat{y})$ represents the *pBest* for swarms $[1..k]$, and $\mathbf{z}$ represents the vector x for the current particle being evaluated. This function will join the n/K dimensions managed by the swarms, forming a complete candidate solution for evaluation by the fitness function $\boldsymbol{f}$.

Since there is no rule to split the swarms, the dimensions can be divided as needed by using whatever metric that might be convenient, like for example the correlation between the neighbor dimensions. Also, the number of dimensions per swarm does not need to be the same for all swarms. In the case non-neighbor dimensions have much higher correlation, they can be moved between the swarms as long as the function $\boldsymbol{b}$ is adjusted to map them back to their original places at fitness evaluation time. For the purpose of this work, since there is no direct correlation among the Zernike terms (dimensions), the dimensions will be split equally among the swarms.

The algorithm 2 describes the CPSO for a minimization function.

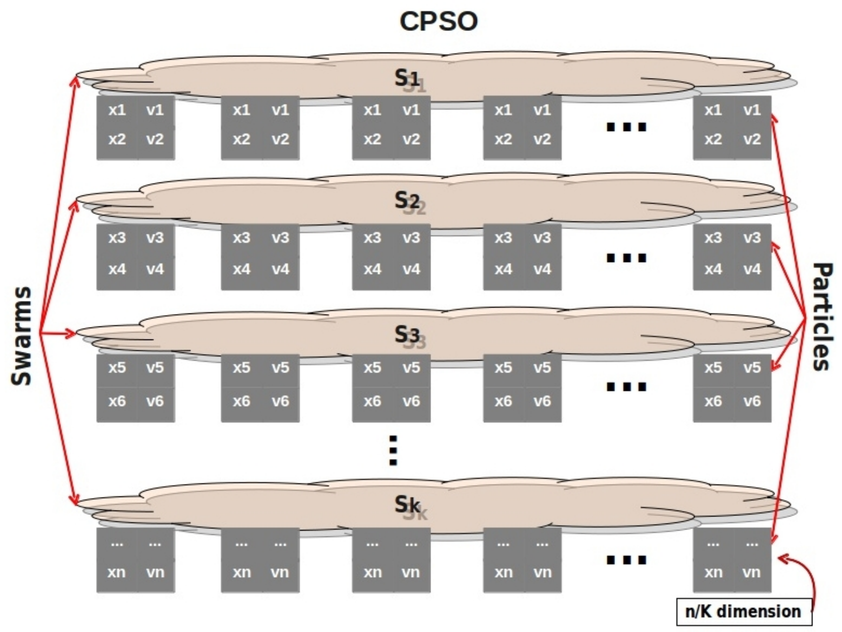


Figure 2.7: Cooperative PSO.
The smaller swarms now collaborate with each other to optimize the variables, reducing the dependency between the dimensions and improving the ability of obtaining good results more quickly.

Algorithm 2 CPSO for a minimization function [50]

```
1: $b(j,\mathbf{z}) = (P_1.\hat{y}, \cdots, P_{j-1}.\hat{y}, \mathbf{z}, P_{j+1}.\hat{y}, \cdots P_k.\hat{y})$
2: Initialize $k$ swarms, with the $n$ dimensions
   distributed as needed per each swarm
3: repeat
4:   for each swarm $j \in [1..k]$ do
5:     for each particle $i \in [1..s]$ do
6:       if f(b($j$, $P_j.x_i$)) < f(b($j$, $P_j.y_i$)) then
7:         $P_j.y_i = P_j.x_i$
8:       end if
9:       if f(b($j$, $P_j.y_i$)) < f(b($j$, $P_j.\hat{y}$)) then
10:        $P_j.\hat{y} = P_j.y_i$
11:      end if
12:    end for
13:    Perform PSO update on partition $P_j$ using eqs. (2.10-2.11)
14:  end for
15: until stopping condition is true
```

2.5 OpenCL

OpenCL (Open Computing Language) is an open, royalty-free standard for cross-platform, parallel programming of modern processors found in personal computers, servers and handheld/embedded devices that greatly improves speed and responsiveness for a wide spectrum of applications [5]. Many hardware vendors, like Intel, AMD, Apple, IBM and NVIDIA, implement OpenCL on their devices, what allows for a great code reuse regardless the device providers. Another great advantage of OpenCL language is the fact that any set of supported hardware can be jointly used on same program without any special adaptation of the code.

Before processing any task on OpenCL, the respective work must be decomposed into smaller pieces called Work Items. These work items must be organized into processing batches called Work Groups. The task executed by a work group is computed through an execution instance called Kernel [2]. Every kernel must be running inside one specific OpenCL device (for example, one GPU card).

Every work item is given a unique global ID within the entire task, besides one unique ID within its work group. Work groups are also assigned a unique group ID within the group set. The combination of these IDs represent a specific point in an Index Space where every work item should be working on the data. The index space supported on OpenCL is called NDRange, which is a N-dimensional space where N can be one, two or three and every dimension can have its specific size (Figure 2.8).

The Memory Address Space of the OpenCL architecture (Figure 2.9) is composed of:

- Global Memory: a global slow access space, where all global data is stored.
- Local Memory: a much quicker and smaller memory than Global Memory, it can be used to share data among work items in a workgroup and to reduce access to global memory.
- Private Memory: every work item has one fast private memory, which is similar to CPU's registers.
- Constant Memory: a global memory space, this is a read-only memory which can have different advantages depending on the hardware. This is usually used for broadcast operations.
- Host Memory: the memory on CPU side (computer's RAM memory).

OpenCL uses an explicit memory management, meaning the data must be moved by the code itself from Host memory to Global memory, then from Global memory to Local memory, then from Local memory to Private memory, then all the way back. No automatic data move is done by the devices nor by the OpenCL.

The OpenCL Framework is context based, where every context can group several devices and every device can enqueue kernels to run on it (Figure 2.10). The code is stored inside a .cl file, which is compiled at runtime (provided to the context for compilation).

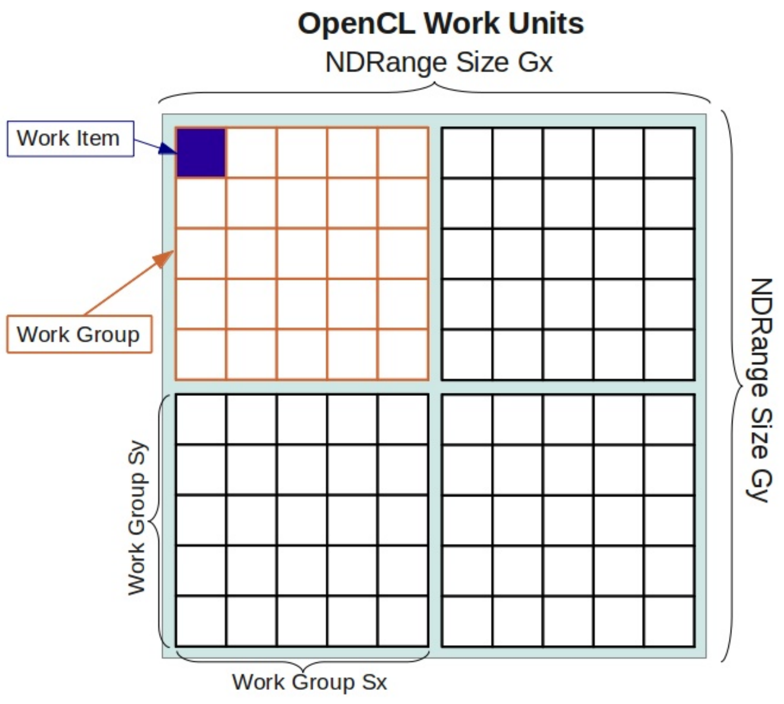


Figure 2.8: OpenCL Work Units.
The Index Space on OpenCL is composed by a combination of global ID, local ID, work group ID and dimension. The work on OpenCL device is divided in NDRange size (G_x, G_y, G_z) and Work Group size (S_x, S_y, S_z), where (G_x, G_y, G_z) must be multiple of (S_x, S_y, S_z).

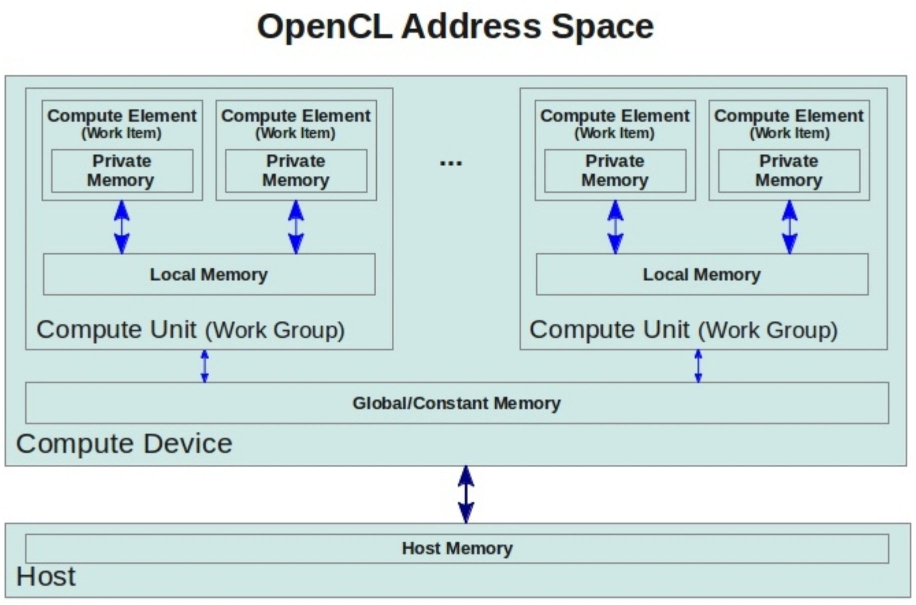


Figure 2.9: OpenCL Address Space.
There are two main memory spaces: the Host memory and the (OpenCL) Compute Device memory. The Compute Device memory is divided into Global/Constant memory (slow speed), Local memory (quick speed) and Private memory (register speed).

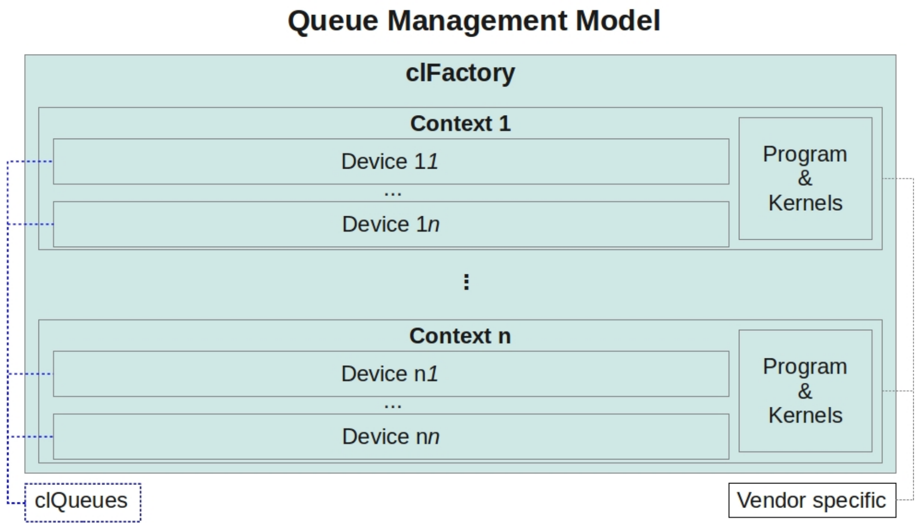


Figure 2.10: OpenCL Queue Management Model.
This figures shows how the solution developed in this work deals with all aspects of a complete OpenCL environment. The classes clFactory and clQueue are described in details on section 4.2.3.

Once the program is compiled, the kernels can be instantiated at context level for later execution.

Before executing a kernel, a Command Queue must be instantiated for every device where the tasks will run. The Command Queue is used to enqueue the kernels that will run on the devices, where such queuing can be set to execute in-order (at the order the kernels were submitted) or out-of-order (at the order the OpenCL decides internally). When the queue is set to out-of-order, there is no guarantee that the kernels will be executed at the same order as submitted, therefore if a specific sequence is necessary, events can be used to force (for example) a kernel B to be executed only after a kernel A has finished. The advantage of the out-of-order queue is not forcing the OpenCL to follow any specific sequence, thus allowing it to decide when it is more advantageous to submit every kernel. For that reason, this work uses out-of-order Command Queues.

At the time of calling the kernel, both the kernel instance and its parameters must be submitted to a Command Queue. The command used to submit the kernel is non blocking, therefore multiple kernels can be enqueued on same Command Queue without having to wait for the previous kernels to finish. Therefore, it is OpenCL's responsibility to organize its work internally and make sure the kernel calls are all configured and submitted correctly.

2.5.1 Random Numbers for CPSO on GPU Cards

One mandatory requirement of the CPSO algorithm is the generation of a large amount of random numbers. This makes CPSO sensitive to the quality of the numbers generated, therefore, the use of a good Random Number Generator (RNG) is critical for the quality of the optimization. One solution for the GPU-based CPSO is pre-generate on CPU all the required random numbers before they are actually used. Due to memory limitation on GPU cards, these pre-generated random numbers must be transferred from CPU to GPU periodically, what impacts the running time given this kind of transfer is too slow compared to the GPU clock cycle. Besides, every CPSO iteration requires a large amount of random numbers, what in turn would require extra memory from the GPU cards to store such values, thus reducing the capacity of processing PSF estimations in parallel.

To avoid the impact caused by this transfer, the random numbers should be generated on GPU side. The work developed on [9] shows that it is possible to have a good GPU-based RNG (e.g., a Xorshift implementation) that provides sufficient quality to be used by a CPSO algorithm. The RNG used on this work is described on Algorithm 3.

Algorithm 3 Xorshift

```
1: Input: $\{seed \ \ \{the\ random\ seed\} \ \}$
2: $tmp \leftarrow seed[0]$ ^ $(seed[0] << 11)$
3: $seed[0] \leftarrow seed[1]$
4: $seed[1] \leftarrow seed[2]$
5: $seed[2] \leftarrow seed[3]$
6: $seed[3] \leftarrow seed[3]$ ^ $(seed[3] >> 19)$ ^ $(tmp$ ^ $(tmp >> 8))$
7: return $seed[3]$
```

Given that a single seed can bring a limit to the number of calls allowed for the RNG, to minimize the risk of reducing the quality of such random numbers, every particle in this solution contains its own seed and such seeds are refreshed with new values from CPU at every CPSO iteration.

The chapter 4.2 describes in details how the CPSO was developed to take advantage of the massive parallel environment provided by a GPU device.

2.6 Considerations

The purpose of this work is to develop a parallel solution to run on OpenCL enabled devices, so that the PSF Estimation of solar surface image could run on feasible times with good results under cheap hardwares. The Cooperative PSO is proposed for the optimization of the PSF parameters and a calibration method is presented (section 5.1) to adjust the CPSO for obtaining good results under low processing times.

The next chapter discuss the optimization methods available, presenting results obtained by other researches. Variations of PSF estimation methods are also discussed.

CHAPTER 3

BIBLIOGRAPHIC REVIEW

The following section describes metaheuristics and parallel approaches for their implementation. Then, the section 3.2 provides an overview of PSF estimation methods and describes examples of works developed for different types of PSF estimations.

3.1 Parallel Approaches for Metaheuristics

Metaheuristics is the term used to describe any stochastic optimization method used to find good solutions for problems where it is not known how to search for good candidate solutions [34]. Stochastic optimization is the general class of algorithms and techniques which employ some degree of randomness to find good solutions to hard problems.

The metaheuristics can be classified into two main categories [13]:

1. Trajectory-based metaheuristics: starts with a single initial solution and at each step of the search, the current solution is replaced by another solution found in its neighborhood (often the best solution). This metaheuristics allows for a locally optimal solution to be found quickly, promoting intensification in the search space. Examples of this metaheuristics includes Simulated Annealing (SA), Tabu Search (TS) and Local Search (LS).
2. Population-based metaheuristics: makes use of a population of solutions. The initial population is randomly generated and then enhanced through an iterative process. These techniques promote the diversification in the search space. Examples of this metaheuristics includes Evolutionary Algorithms (EAs), Ant Colony Optimization (ACO) and Particle Swarm Optimization (PSO).

The main difference between intensification and diversification is that during an intensification, the search focuses on examining neighbors of elite solutions. The diversification, on the other hand, encourages the examination of unvisited regions, generating solutions that differ in various significant ways from those seen before [21]. Considering that Zernike terms have no correlation between each other and the estimation of their coefficients require the examination of the entire search space, only the population-based metaheuristics were considered for the improvement of this PSF estimations method.

Parallel implementations of EAs, ACO, and PSO (described below) have been the preferred choices for efficiently solving optimization problems related to real-world applications [7], while parallel trajectory-based metaheuristics (SA, LS) have been also used as viable second options.

The Ant Colony Optimization (ACO) is a metaheuristics based on the social behavior of ants. It has a constructive population-based approach, where an ants population gradually walk towards a common goal by using heuristics and pheromone information. As the ants find their paths, they updates the pheromone values on these paths, what guides the remaining ants to potentially better paths. After a pre-determined number of iterations, the best path (the one with more pheromone) is chosen as the best solution for the problem. Even though the ACO model facilitates the development of CPU-based parallel optimizations, the work developed in [19] shows that when developing the ACO to GPU architectures some important challenges comes up, mainly due to pheromone management and to the size of the data structures that have to be maintained. Even though good improvements are shown on that work, complex algorithms were required and memory limitations become evident.

Given the ACO success strongly depends on the nature of the particular problem and the underlying hardware available, the work developed in [17] tries to overcome the limitations of ACO by a specific thread scheduling on GPU and by creating strategies to access the memory. As results, even more complex algorithms were required to be able to improve the optimization.

Basically, the ACO is meant for problems where all possible values are known beforehand, as the Traveling Salesman Problem (where all vertices are known previously), what makes the ACO not feasible for the PSF estimation problem.

The term Genetic Algorithm (GA) was first used by John Holland in 1975 to describe a probabilistic search procedures designed to work on large spaces involving states that can be represented by strings of values [22]. These methods are inherently parallel, using a distributed set of samples from the space (a population of strings) to generate a new set of samples. The common thread in these ideas was the use of mutation and selection, which are the concepts at the core of the neo-Darwinian theory of evolution. Several strategies to keep the good results are used [43], like roulette wheel selection, tournament selection, ranking, crossover, mutation and hybridization. Currently, many variations of the GA are being experimented, with different improvements obtained in every variant.

The work developed in [38] uses a Quantum-Inspired Genetic Algorithm (QIGA) for the optimization. QIGA is a hybrid heuristic algorithm, inspired on the biological evolution and the unitary evolution of quantum systems. Concepts such as qubits, observations and superposition of states are involved in different stages of the algorithm. In QIGA, genes are modeled upon the concept of qubits (a bi-dimensional matrix of 0's and 1's), which brings an additional element of randomness and a "new dimension" into the algorithm. Given the complexity of the algorithm to run on CPUs and the parallel nature of this algorithm, the implementation of QIGA developed on that work obtained large speed ups on GPU device when compared to the CPU version.

Compared to others swarm based algorithms such as AG and ACO, the PSO has the advantage of easy implementation, while maintaining strong abilities of convergence and global search. In recent years, PSO has been used increasingly as an effective technique for solving complex and difficult optimization problems in practice [62]. Nevertheless, PSO still needs a long time to find solutions for large scale or high complexity problems, such as problems with large dimensions and problems which need a large swarm population for searching in the solution space. The main reason is that the optimization process of PSO requires a large number of fitness evaluations, which is usually done in a sequential way on CPU.

However, the PSO is well known as a metaheuristics with high capacity for parallelization. The work developed in [46] implemented a GPU-PSO algorithm by treating each particle with its own individual thread. That work showed that the highly CPU intensive tasks required by bioinspired metaheuristics for complex problems can be greatly benefited by the massive parallel environment provided by a GPU device.

The PSO developed for GPU is not limited to a single algorithm and every different problem might require a different PSO implementation. The work developed in [41], for example, implemented an automated Gamming PSO and used a separate thread to treat each dimension of every particle in each swarm. The results showed good speed up, but the tests were not complex enough to reach the maximum capacity of the proposed algorithm.

The work develop in [62] is another case of one thread per particle, where the thread was implemented to iterate over all the particle's dimensions. The results show that problems with more complex arithmetics and problems with a large number of dimensions have much higher speed improvements when compared with the CPU version. No important improvement was obtained while increasing the population sizes, so smaller populations can obtain as good results as large populations by just increasing the number of iterations, what is an opportunity to fine tune the PSO for the physical GPU characteristics.

Similarly, on [14] it was found that the number of particles does not represent any direct improvement on the results obtained. In general, the configuration of a PSO is a time consuming task that requires manual adjusts of several parameters, which invariably has no correlation with the problem being optimized [56]. The PSO calibration process, in short, is essentially a trial and error process and a bad parameter set can lead to poor optimization results. Therefore, regarding the parameter set, it is known that the PSO lacks an easily measurable calibration processes.

The PSO algorithm also has been used to estimate PSFs while recovering corrupted images. On [47], both the GA and PSO are used to seek the unknown PSF of a blured image, obtaining similar results on both optimization methods. Since that specific problem had only one dimension, the conclusion of that work stated that for multi-dimensional estimation problem, the PSO should outperform the GA.

The work developed in [12] also uses both GA and PSO to estimate unknown parameters for a specific multi-dimensional problem (biochemical systems). As result, the PSO outperformed the GA on the experiments and the work concluded that the PSO is the best method for that specific optimization problem. A successful usage of the PSO to optimize unknown parameters for PSF estimation is found on [40], where the coefficients for a filter mask are searched by a PSO to restore a corrupted image without any knowledge of the PSF.

3.2 PSF Estimation Methods

There are several methods available to calculate the PSF, each one developed to address peculiarities of a specific problem. Usually, one of three ways is used to calculate the PSF:

1. Using well-defined (on shape and size) targets for comparison with the corrupted image.
2. Applying a filter over a non-corrupted image and then comparing the results with the corrupted image.
3. Using the system design specifications and the system analytic model.

On [10], the second method is implemented by obtaining a high resolution terrestrial image from one satellite sensor and a second terrestrial image from a low resolution sensor of another satellite, then applying a low-pass filtering over the high resolution image and comparing with the low resolution image. The optimization process is implemented by adjusting the filter at every iteration, while minimizing the image differences.

The PSF methods are also divided into parametric and non-parametric:

- The parameterized models for PSF estimation requires estimation of parameters (eg., functions or filters parameters).
- The non-parametrized models use the additional characteristics available to estimate the PSF.

The work developed in [15] used a non-parametric model to estimate the PSF based on a digital template registered on the camera before the image was taken, then comparing the template with the actual image taken, what allowed for a successful calculation of the distortion and a much better image recovering. As example of a parametric PSF model, on [44] a single-parametric PSF was used to adjust the focus on generic blurred images.

The PSF estimation is a challenge when the type of blur or motion in the image is unknown. Even when the PSF is known, deblurring results in amplified noise and the high frequencies are usually lost. Besides, the deconvolution process used for restoration is limited by the lost of spatial frequencies while restoring the corrupted image. Therefore, the quality of the estimated PSF is critical for a good image restoration process.

The work developed in [42] used the intervention of the camera user to assist on the blur path estimation instead of calculating the PSF directly, what proved to increase the quality of the restoration. On [37], the PSF was estimated by adding a secondary low-resolution sensor in the camera to capture the real motion, what was used to calculate a more exact PSF, then deconvolving the primary high-resolution image with the PSF so calculated, resulting in superior images.

The model used as base for the work developed in this paper is presented on [53], where a parametric PSF model is proposed to estimate the wavefront phase and the PSF from the data obtained by a single imaging channel from a telescope. A series of sequential solar images is taken and combined, generating a clear reference image that is used for comparison with an artificial image. The artificial image is generated by convolving the clear reference image with a random parametric PSF, where the resulting artificial image is compared with a real distorted solar image. The difference calculated between the artificially generated image and the real image is used for minimizing an error function, which is optimized by a Cooperative PSO. The quality of the PSF obtained is the main focus of the CPSO proposed here.

3.3 Considerations

Despite the fact that there are many good optimization methods available, not all of them are of simple implementation. When developing metaheuristics methods for GPU cards, several additional limitation can arise, what in turn could require more complex code structures to overcome these problems. The PSO has proven to be a simple optimization method that can be easily implemented on GPU cards, besides not suffering from any critical limitation imposed by the GPU hardware. A good advantage of the Cooperative PSO when compared to the other methods is the strong quality of results obtained when the number of dimensions is very high.

Regarding calibration methods for PSO, no standard process exists for its configuration, meaning that it is up to the experienced PSO user to set up its parameters heuristically.

There are many methods for PSF estimation, each one developed to solve specific problems under well defined environments. The PSF estimation method used on this work relies on a single imaging channel from a telescope to obtain all the data necessary to recover the corrupted solar image. A Cooperative PSO is used to optimize the PSF parameters and a calibration method for the CPSO is proposed for this specific PSF estimation problem.

On next chapter, previous results obtained for this PSF estimation method with Simulated Annealing are presented, and the proposed Cooperative PSO for OpenCL devices is detailed at implementation level.

CHAPTER 4

POINT SPREAD FUNCTION COMPUTATION

During the development of this work, several techniques were used to improve both the speed of the PSF estimation and the quality of the results obtained, in such a way that a good balancing could be obtained between a reasonable estimation and the time taken to calculate it.

The original method used as base for this work uses the Simulated Annealing as optimization method, which generates only small changes from iteration to iteration and requires long cooling cycles, meaning that good results need long processing times. To better understand the PSF estimation problem, the existent SA code (provided by the Kiepenheuer Institute) was refactored then ported to multicore. To investigate how the PSF estimation can be processed on the massive parallel environment provided by GPUs, the SA was also reimplemented for graphical cards. No exhaustive tests were executed for the SA, since it was used only as reference to develop an improved PSF estimation method with CPSO.

The optimization method proposed in this work, the PSO, is a well known fast convergence method that is noise robust and built to simultaneously tackle multiple dimensions (or Zernike terms, on this work) of the problem being optimized, instead of analyzing only one dimension at a time. A cooperative version of the PSO (CPSO) was used to better control the aggressive convergence behavior of the method and to take advantage of the vector processing model of a GPU based code.

Given the GPU devices provide a SIMD environment (Single Instruction, Multiple Data), where the same instruction is executed over multiple data points simultaneously, additional parallelism occurs while accessing most of the data required for this method, like for example the images, the phases, the PSFs, the Zernikes and the masks (details provided on section 4.2).

The next section (4.1) shows directly the results obtained by the original method proposed to estimate the PSFs (by using Simulated Annealing for optimization). The proposal of this work is presented on section 4.2, where the estimation method and its implementation are explained in details.

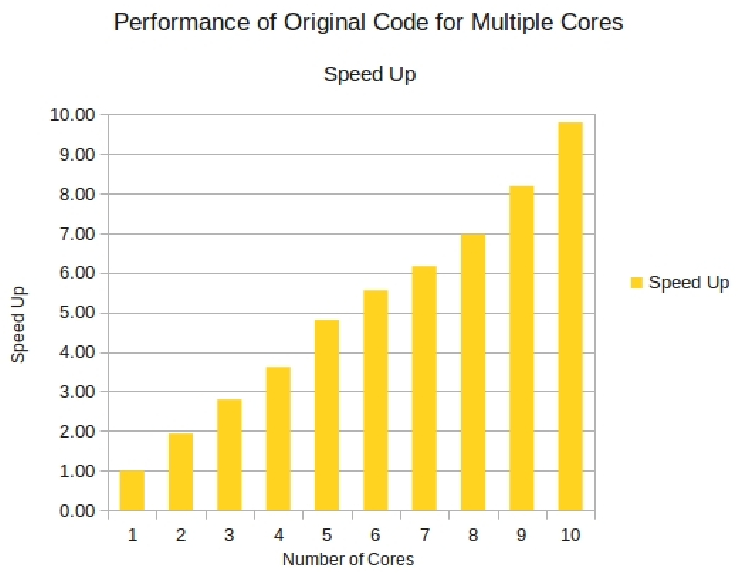


Figure 4.1: Performance of the Original code per CPU core.
The original Simulated Annealing shows a good scaling behavior.

4.1 Initial Results with Simulated Annealing

The original version of this PSF estimation technique is a computationally expensive one that requires long processing periods in a single core. For multiple cores (developed with OpenMP - Open Multi-Processing [4]), even though we see a good scaling behavior (Figure 4.1), it still requires a large amount of resources to process the estimations. Given the SA requires long cooling cycles, for the purpose of this work every result presented here was obtained after having evaluated the fitness 1635000 times.

All experiments on CPU for SA were executed on a machine with the following configuration:

- 4 x AMD Opteron(tm) Processor 6136 (8 cores each processor)
- 120GiB RAM

By looking carefully at the original SA algorithm used for the PSF Estimation (Figure 4.2), one can see that the most expensive part of this technique is the computation of the phase $\phi(u)$, which consumes nearly 90% of the CPU time. In order to have the phase calculated, it is necessary to iterate over each one of the n Zernike terms Z_i, where every Z_i is composed by a square matrix of size $(\phi_{width})^2$ (for the purpose of these experiments, $width = 128$). Therefore, to calculate a generic $\phi(u)$ it is necessary to execute $n(\phi_{width})^2$ multiplications plus $n(\phi_{width})^2$ sums.

To minimize this problem, the code was refactored and the most computationally expensive codes were replaced by their quicker versions, with exact same results. As part of the refactoring, the problem of the expensive phase calculation was solved by storing separately partial phases $\phi(u)_i$, each one containing only the $\gamma_i Z_i(u)$ of the Equation

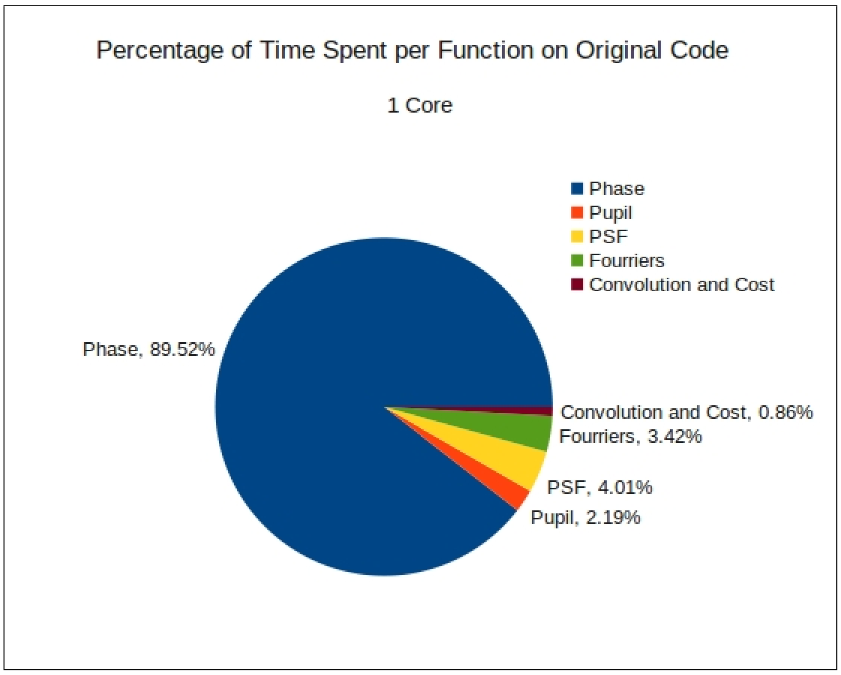


Figure 4.2: Time spent per function on Original code, in %.
The original Simulated Annealing shows critical problems with the phase calculation.

(2.4). This solution requires additional memory of $n(\phi_{width})^2$ double values to store partial results, but since the required additional memory is not a problem for these experiments, it worth the cost. This solution is also possible due to the fact that the simulated annealing algorithm requires only minor changes from iteration to iteration, what is achieved in this work by updating only one γ_i at a time.

After refactoring the code to a quicker version, the running time was improved sufficiently (Figure 4.3) to be used as a good comparison measure with its parallel versions (given the speed up metric requires the best known sequential version).

By comparing the results obtained by the experiments on CPU, we see that the refactoring alone was responsible for a speed up of 26 on the calculation of the phase, what made the refactoring a real important step while improving the performance of this PSF estimation technique. For multiple CPU cores (8 cores with OpenMP), the refactored code still shows (on Figure 4.4) a good scaling in the PSF calculation in comparison with the refactored version for 1 core (speed up of 6.54), but no improvement on the Fourier calculation. This happened because the code was using an open library for Fourier calculation which did not process in multiple cores at the time this experiment was executed (FFTW [1]). Thus, this processing time could be yet reduced by using a Fourier library that support multiple cores. The Figure 4.5 shows the comparison of the speed up obtained by the refactored code in both 1 and 8 cores, when compared to the original code.

A second parallel version was developed on CUDA language (NVIDIA Compute Unified Device Architecture) for machines providing GPGPU graphical cards, and this was

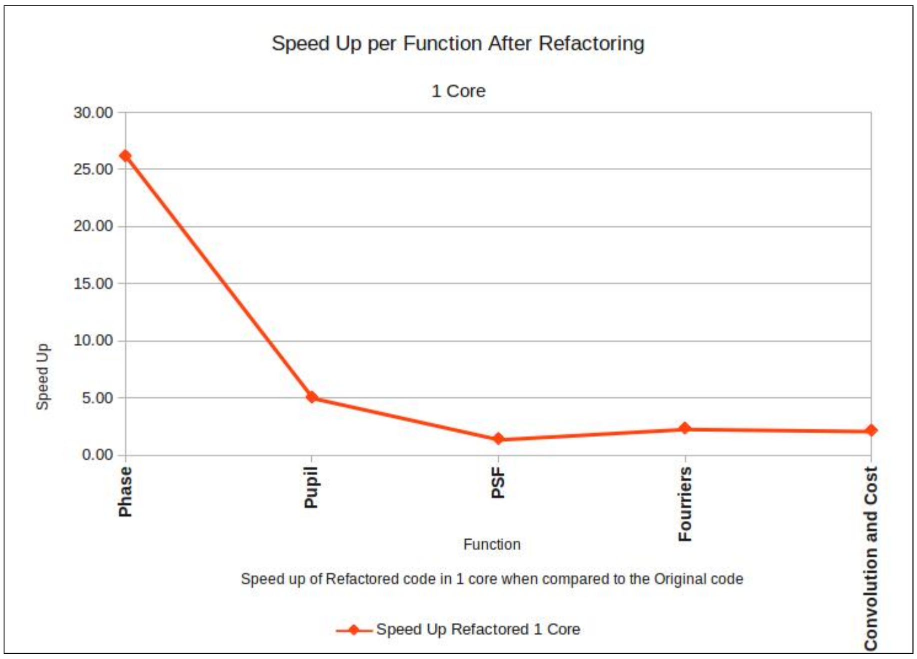


Figure 4.3: Speed up per function after the Refactoring.
Speed up of the refactored versus the original code for the main functions.

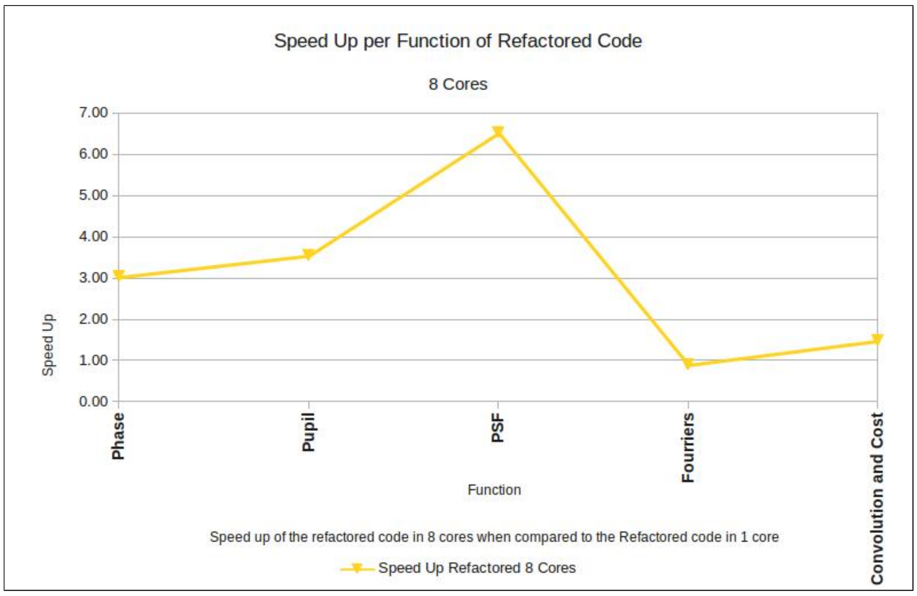


Figure 4.4: Speed up of the refactored in 8 cores versus in 1 core, for the main functions.
This speed up is not considering the original code. The multiple cores were enabled by using OpenMP library.

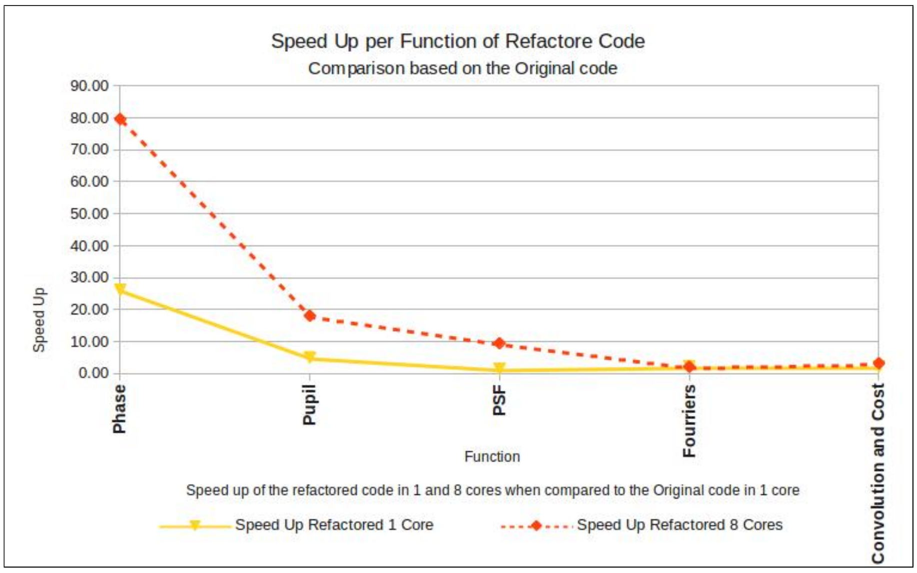


Figure 4.5: Speed up per function (1 vs. 8 cores) compared to the Original Code. This figure is comparing the original code with the refactored versions for both 1 and 8 cores.

the code that presented the best performance for SA. The experiments on GPU device were executed on a machine with the following configuration:

- AMD Opteron(tm) Processor 8387 (4 cores each processor)
- 10GiB RAM
- GPU card model NVIDIA Tesla C1060 (30 Multiprocessors, Warp size 32, 240 Cores, max. Block size 512, 1.30 GHz, 4GB GDDR3, CUDA 3.2)

Every kernel was executed with several different block sizes configurations, then the configurations that presented the best results were used to optimize these kernels. The Fourier was calculated by CUFFT library (NVIDIA CUDA Fast Fourier Transform [3]), thus its behavior is not controlled by the user and all its results are constant among the different configurations executed.

The Figure 4.6 presents the speed up on the total running time, obtained by the GPU optimized version and all CPU versions, when compared to the original version. Every result presented in the figure shows the improvement obtained in each phase of the investigation of this PSF estimation method.

We can see that the best performance for CPU (refactored version for 8 cores) obtained a speed up of ≈28, whereas the GPU version obtained a speed up of ≈147 (with 1 CPU core and 1 GPU device). Given that both versions used CPU hardwares of approximate processing power per core, the use of a GPU device showed an important increase on the capacity of estimating PSFs, what justified the development of a new PSF Estimation method to run on GPUs. Still, the GPU solution did not achieve its best possible results because the process submitting these kernels was using 100% of the CPU time, meaning that the CPU was acting as a bottleneck and the speedup could be even higher for the fine tuned GPU version.

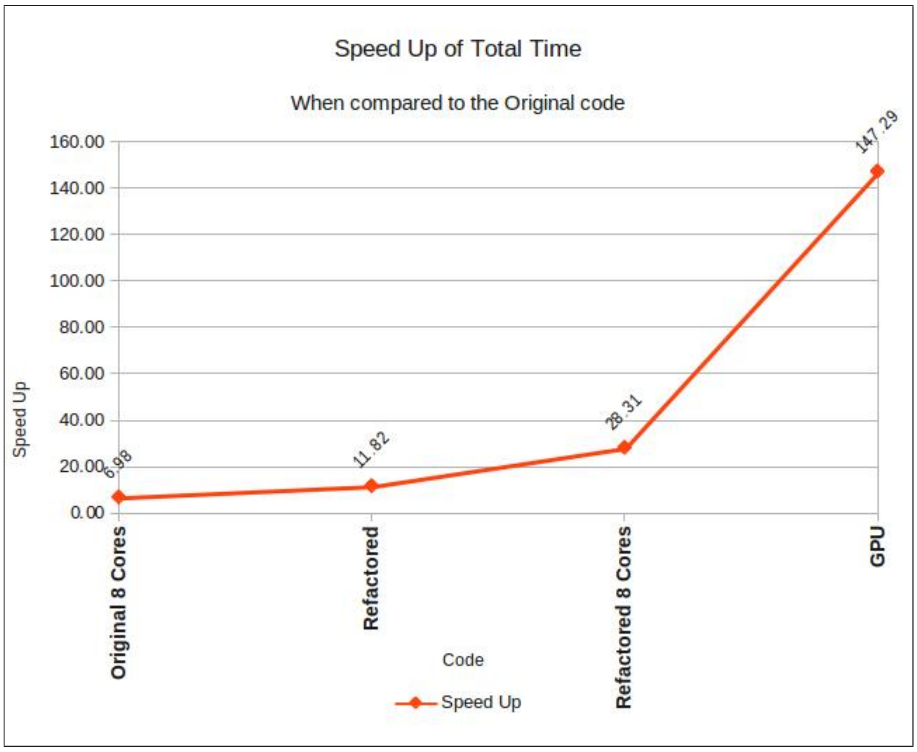


Figure 4.6: Improvements obtained during the SA investigation.
The figure shows the Speed up obtained on the Total Time, for the Original version on multi-core, the Refactored version, the Refactored multi-core version and the GPU optimized version, when compared to the original code. The results clearly justify the development of a new PSF estimation method to run on GPU devices.

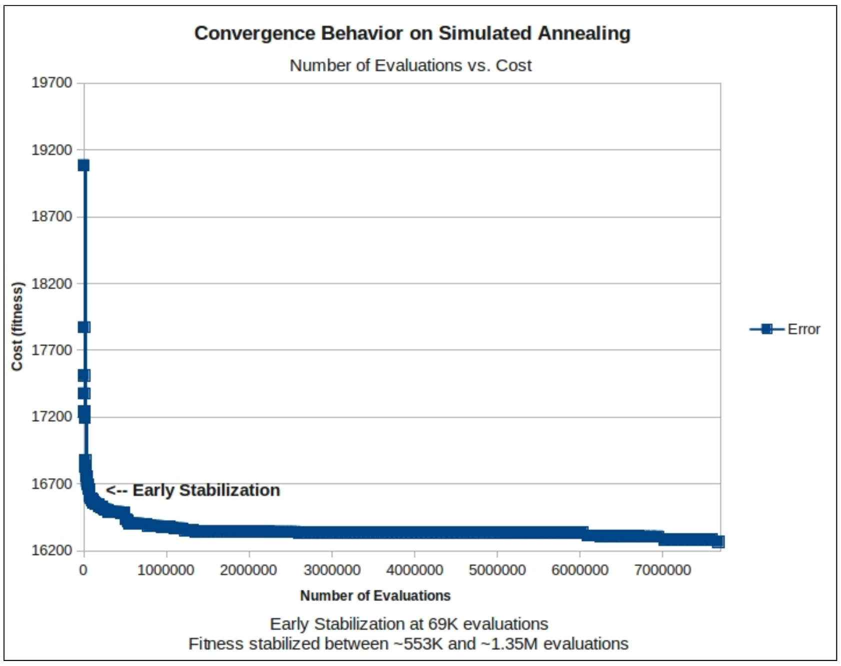


Figure 4.7: Convergence behavior with Simulated Annealing.
Convergence behavior shows strong decrease in the fitness value on first evaluations only, followed by a stuck fitness on subsequent evaluations (i.e., almost zero improvements from 1.35M to 6M), which is caused mainly by the poor management of the Zernike dimensions on SA.

About the convergence behavior of Simulated Annealing as the chosen optimization method for these experiments, we can see improvements occurring at every evaluation until $\approx 69K$ evaluations (Figure 4.7), but the first real fitness stabilization only occurred after $\approx 1.35M$ evaluations, returning to decrease the fitness values only after $\approx 6M$ evaluations (meaning no significant improvement occurred between $1.35M$ and $6M$ evaluations).

The high convergence time on SA is due to the change of only one single coefficient at each iteration, requiring n iterations just to experiment one different value for every Zernike coefficient. Furthermore, given this is a non guided optimization method, there is no direct correlation between one good value kept in one iteration and the next good value found, what makes the fitness improvements too random to obtain a better convergence curve.

The results obtained in these initial experiments show that the SA is not a satisfactory choice for quickly finding good results for this PSF Estimation method, since even though good results were obtained by this method, a high processing time was required to obtain such results and no real improvement can be obtained after a certain number of iterations, given its weak convergence behavior.

4.2 Cooperative PSO Proposed for OpenCL Devices

The Pooling pattern describes how expensive acquisition and release of resources can be avoided by recycling the resources no longer needed [28]. A pool of objects holds unused instances of same class. Once an instance is required, instead of having to create a new one, it will be obtained directly from the pool. If no instance is available and if the pool has not reached its maximum size, a new instance is created and assigned to the pool. In order to create a highly parallel solution for PSF estimation based on CPSO optimization method, it was created a set of pools and classes to maximize the object reuse during the estimation of multiple PSFs.

The Algorithm 4 gives the succinct explanation of how the processing happens. There are two objects' pools that benefit from the instances already prepared for the use: one for the CPSO instances (line 11) and another for the clQueue instances (line 14).

The PsfEstimator class is responsible for managing the CPSO instances, while the clFactory is the class responsible for managing the clQueue instances. Every time the clFactory class creates one clQueue instance, it selects the next device available in the system (in $round - robin$ sequence) so that the load can be balanced among the devices. Every object is created on demand (lazy load) and it holds all resources until the program terminates.

The windowing function (see section 2.1) *Hanning* is applied over both the original Object O and the distorted Image I_n before they are copied to the device, such that the FFT's (Fast Fourier Transformations) "leakage" effect is reduced and the quality of the results is improved (Algorithm 4, line 18).

Before every execution of a CPSO instance (line 20), the object and the image are copied to the device (line 19) on the previously allocated resources (line 16). After the CPSO instance has been used and the PSF was estimated, the CPSO object is released back to the pool (line 22).

Notice that the parameter $nParalellism$ controls the number of images I_n that will be processed in parallel (line 8), limited by the number of OpenCL devices available (one parallel estimation per OpenCL device). The control of CPSO instances and Command Queues through managed pools allows for a transparent parallel estimation of multiple PSFs.

Algorithm 4 PSF Estimation with CPSO

```
1: Input:
2:     $O(x), I(x)[], nSwarms, nParticles, nCycles,$
3:     $nZernikes, w, c1, c2, resetFactor, nParallelism$
4:
5: $nImages = arraySize(I)$  {count how many images the array I contains}
6: Initialize PsfEstimator with:
7:     $nSwarms, nParticles, nZernikes, w, c1, c2, resetFactor$
8: for $n = 1$ to $nImages$  {run $nParallelism$ threads}  do
9:    if no CPSO instance available fom PsfEstimator's pool then
10:      Create a new CPSO instance and assign to cpso
11:      Add the new CPSO instance to PsfEstimator's pool as in use
12:      if no clQueue instance available from clFactory's pool then
13:         Create a new clQueue instance for next device (in $round - robin$) and set it on cpso
14:         Add the new clQueue to clFactory's pool as in use
15:      end if
16:      Prepare the device for cpso  {allocate resource, send static data, ..}
17:   end if
18:   Apply Hanning windowing function over $O$ and $I_n$
19:   Prepare the device for $O$ and $I_n$ under cpso's area
20:   Run cpso instance up to $nCycles$ cycles
21:   Save all results to disk
22:   Release the CPSO instance back to PsfEstimator's pool as not used
23: end for
24: Free up all local and OpenCL device resources
```

The Figure 4.8 gives a visual explanation of the Algorithm 4.

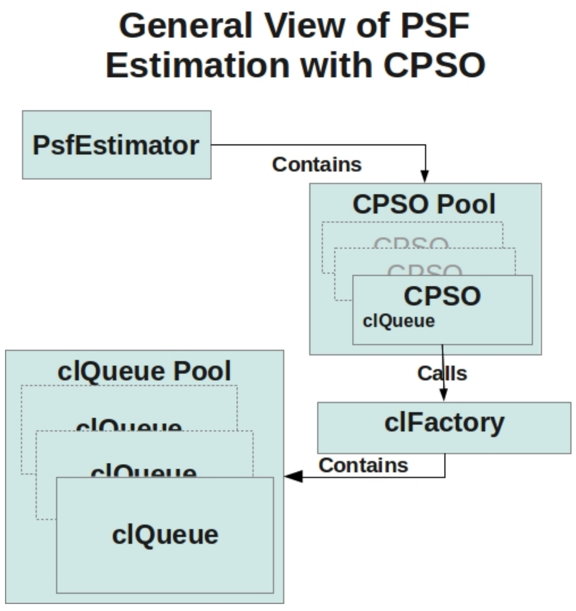


Figure 4.8: Solution for PSF Estimation with CPSO.
PsfEstimator is the main class, that manages a pool of CPSO instances, which in turn contains a clQueue object that is obtained from the clFactory class' pool.

The following section describes how the kernels are submitted to the OpenCL devices. The section 4.2.2 explains how the CPSO data was adapted to fit the OpenCL device memory. The detailed explanations of the classes PsfEstimator and clFactory are described on section 4.2.3. The CPSO class is described on section 4.2.4 and the PSF estimation flow is presented on section 4.2.5.

4.2.1 OpenCL Kernels

The PSF Estimation is processed on OpenCL device by using two different approaches:

1. At the places where the parallelism is possible (Figure 4.14, box *All Particles at Once*), every work group is configured to process an entire image or an entire phase (depending on what is being processed), and each work item will be responsible for either one row or one column. This means that one single kernel call will process all images or phases at once. The following configuration was used:
    - Number of Dimensions[1] = 1
    - NDRange Size[2] = number of sub-particles $\times$ (image width or phase width)
    - Work Group size = image width or phase width
    - A single kernel call processes all images or all phases.
2. The places where no extra parallelism is possible (Figure 4.14, box *One Particle at a Time*), it was used the classical parallelism on OpenCL device, where every work item is responsible for one single point in the image or phase area only (and not an entire row or column). To help reduce the overhead of calling multiple kernels and assist the OpenCL device in managing the kernel code, all calls for the same kernel are submitted sequentially and the synchronization point occurs only after the kernel was called for all images or phases (eg.: Algorithm 7, line 7).
    - Number of Dimensions = 1
    - NDRange Size = image area or phase area
    - Work Group size = image width or phase width
    - A single kernel call processes 1 image or 1 phase.

For the CPSO algorithm (Figure 4.13), every work group is configured to process an entire swarm and the work item will process all aspects of a sub-particle. Given the work item is responsible for iterating through all dimensions of its sub-particle, large dimensions are allowed to be processed with only minor additional effort for the OpenCL device (thus, the solution supports a large number of Zernike terms). After a single CPSO kernel call, a complete iteration is processed at once for all sub-particles in all swarms.

- Number of Dimensions = 1

[1]N-dimensional space is defined on section 2.5.
[2]NDRange is defined on section 2.5 and Sub-particle is defined on section 4.2.2.

- NDRange Size = number of particles $\times$ number of swarms
- Work Group size = number of particles
- A single kernel call processes all sub-particles in all swarms.

At the cases when it is necessary to work with phases bigger than the image sizes (eg., images of 128x128 vs. phases of 256x256), the PSF becomes larger than the own image. On those cases, a smaller PSF must be extracted from the calculated one (Algorithm 7, line 27), so that the resulting PSF has the same size as the image. The extracted values can be obtained by either reading from the corners or by extracting the central portion of the larger PSF (depending on the desired range of frequency). If the PSF has the same size as the image size, the PSF is just taken as-is for the subsequent calculations.

All Fourier calculations were executed by ViennaCL 1.2.1 library [6], which does not support the calculations of multiple FFT's in a single kernel call. Therefore, all parts of the code directly related to the FFT calculation is computing every PSF at a separate kernel call (i.e., one sub-particle at a time). Three different parts of the code require the FFT calculation: the Focus (Algorithm 7, line 9), the scaled Focus (line 21) and the FFT of the scaled extracted PSF (line 36).

4.2.2 Memory Mapping for CPSO on OpenCL Device

The CPSO developed on this work was designed to run on OpenCL devices and follows the same basic steps as any generic CPSO algorithm made to run on Host (CPU). However, to take advantage of the local (shared) memory assigned to a workgroup, reduce the communication with the global memory and allow the entire CPSO iteration to be calculated at one single kernel call, several additional adjusts were necessary, resulting in a good flexibility on the number of swarms, particles and dimensions supported.

Since a particle represents an entire possible solution for the problem, every single particle must be evaluated to obtain the best solution found by the CPSO. In this work, a particle is always linked to an exclusive PSF because each particle contains its own Zernike coefficients that are used to calculated a PSF. Also, given that the complete (actual) particle containing the entire solution is split among all swarms, every subset under every swarm of this major particle was called here *sub-particle*.

One important consideration when dealing with OpenCL devices is the fact that every work item is responsible for one specific region of the memory, thus breaking this rule would result in wrong behavior of the code. Therefore, the Private Area shown in the Figure 4.9 is the memory location where only its respective sub-particle will be updating, and the Shared Area is the memory region where all sub-particle in all swarms collaborate together to have the values populated (i.e., they are not related to the device's Private or Local Memory). The figure shows a CPSO containing k swarms, p particles and n dimensions.

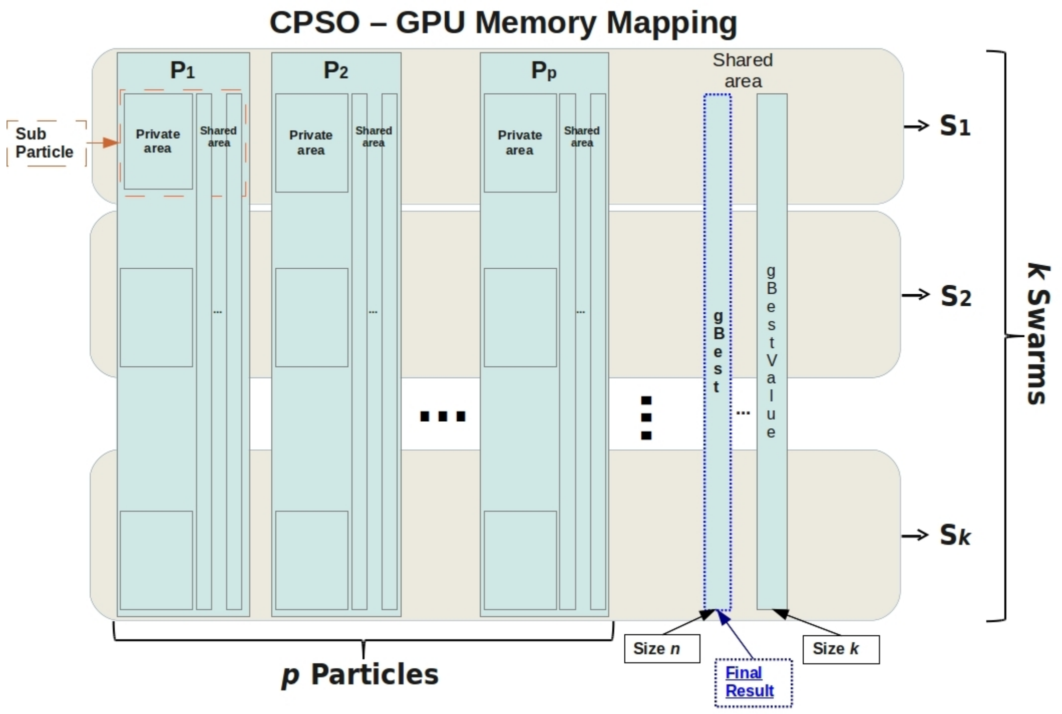


Figure 4.9: CPSO - OpenCL Memory Mapping.
This figure shows how the CPSO algorithm was mapped to the OpenCL device memory. There are three main distinct categories for the data: the swarm's shared area, the sub-particle's private area and the sub-particle's shared area. The final result of a CPSO execution is the value stored under the swarm's shared area, $gBest$.

The private area contains data that is used exclusively by the sub-particle, meaning that no other sub-particle in any other swarm will be touching this data.

There are two types of shared area on Figure 4.9:

1. One for every joined (non split) particle p, containing data used by all sub-particles at same swarms' position p;
2. One for the swarms, containing data used by all sub-particles in all swarms. The swarms' shared area stores two values: the fitness values found by every swarm ($gBestValue$) and the final result of the CPSO optimization ($gBest$).

Notice that the variable $gBest$ contains the final Zernike coefficients that were ultimately optimized, whereas the $gBestValue$ contains fitness values calculated by every swarm through the error function (Equation 2.5). Similarly, the $pBestValue$ (Figure 4.10) contains the actual fitness value for the Zernike coefficients of a sub-particle's $pBest$.

Given that the complete particle is divided among the swarms, any fitness evaluation requires the joining of all others sub-particles at same position p (as defined on Algorithm 2, line 1). These groupings cause some undesired overheads on OpenCL device, thus to minimize the impact of joining these dimensions, the PSF coefficients stored inside the sub-particle's private area are composed by the following data (Figure 4.10), requiring just a single read to obtain all the information necessary to calculate the fitness:

- The subset of the Zernike coefficients $x_K (K = 1..k)$, related to its swarm's position K;
- A copy of the swarm's shared area $gBest$ from all others swarms.

The sub-particle's $pBest$ is stored within this private area and its contents is copied "as-is" from its own PSF coefficients at every iteration.

The sub-particle's shared area stores partial values and every sub-particle is responsible for updating its section K of the area. The following data is stored inside this shared area:

- The sub-vector for velocity v;
- The parameters w, c_1 and c_2;
- The random seed used by the RNG (random number generator) on OpenCL device;
- The $pBestValue$ (actual fitness value of $pBest$).

As the final result of the CPSO execution, the $gBest$ is stored inside the swarm's shared area and it can be easily obtained by a single read.

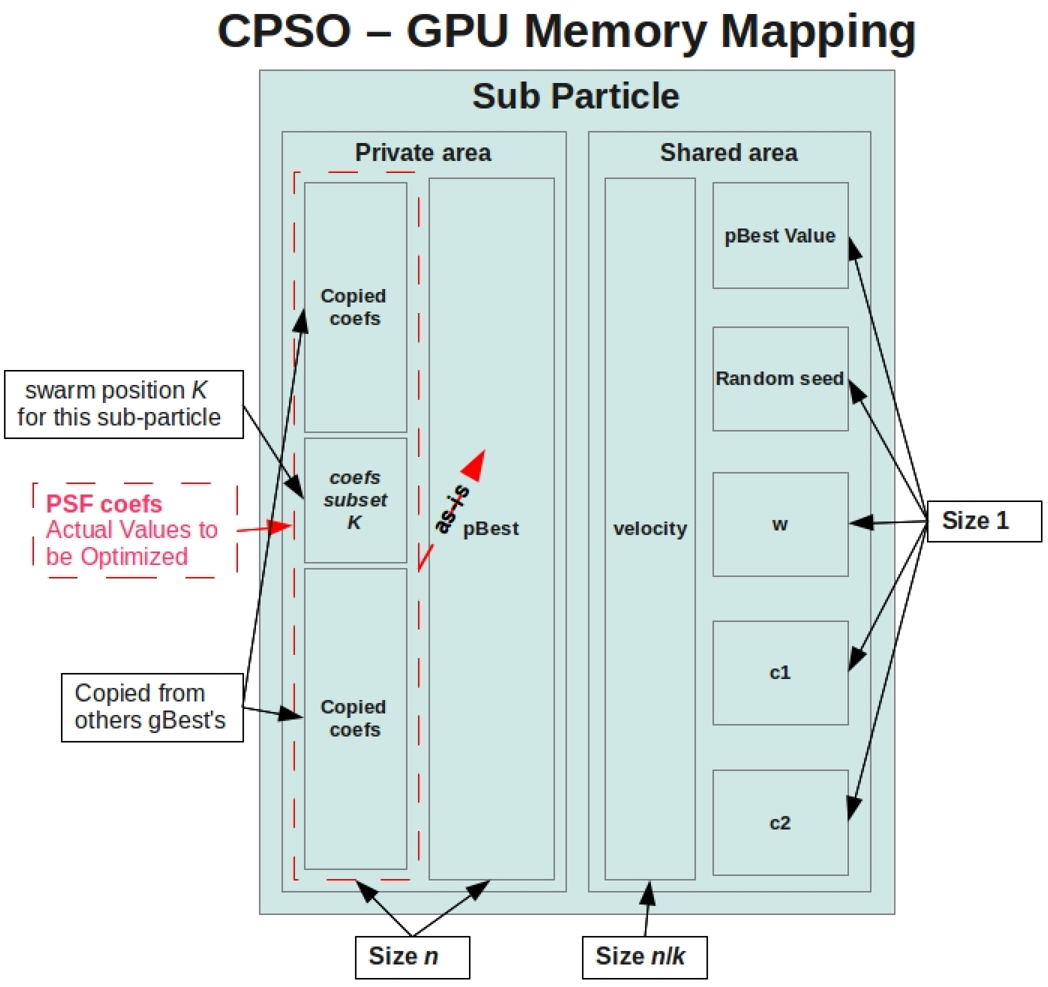


Figure 4.10: CPSO - OpenCL Memory Mapping for Sub-particle.
The sub-particle's memory area is internally divided into Private and Shared areas. In the private area, the PSF coefficients are built by joining the particle's subset x_K (under current swarms' responsibility) together with all others swarms' *gBest*'s. Then, the *pBest* is a direct copy of this joined vector. The shared area stores the velocity v_K, that is part of a larger vector involving all swarms, where the sub particle updates only its subsection K of this larger vector. Notice that: n = number of dimensions, k = number of swarms, $K = 1..k$ (swarm's position).

4.2.3 PsfEstimator and clFactory classes

The PsfEstimator class is responsible for acquiring the CPSO instances. Every instance of the CPSO class (section 4.2.4) is responsible for managing all the steps required to complete one single PSF estimation, including calling all the kernels, managing the kernel parameters, and allocating the memory on Host and on OpenCL Devices. As soon as the CPSO instance is not required anymore, it is released back to the pool so that the same resources can be reused by another PSF estimation, reducing the overhead of releasing resources and then allocating them again for the next estimation.

The clFactory is the class responsible for interfacing with the OpenCL enabled devices, including all startup and shutdown procedures, the *.cl* code compilation, the kernel instantiations and the management of the command queues. Once a request to acquire a new command queue is made, the clFatory verifies if there is any unused queue on the pool and if there is not, it creates a new one. The clFactory uses all OpenCL enabled devices to create command queues, so every new command queue instance is created for the next device available in the system. All information required to deal with the newly created command queue is wrapped inside a clQueue instance, which can also be reused after it is released.

4.2.4 CPSO class

The basic behavior of the CPSO class can be seen on the Algorithm 5. Before the PSF estimation can occur, the CPSO instance must be initialized to process the new data, like setting the Image, the Object and the particle's random values (Figure 4.11, box (1)).

Every time the CPSO instance is called to estimate a new PSF (Figure 4.11,box *Run procedure*), the code does a Startup for the particles based on the previously initialized values. The iteration that calculates the CPSO optimization (Figure 4.11, box (3)) and its error function (Figure 4.11, box (6)) is composed of:

- The execution of 1 iteration of the CPSO optimization algorithm (described previously on section 2.3) on OpenCL device (Algorithm 5, line 8);
- The renew of the Random seeds from host to OpenCL device, because the CPSO algorithm consumes a large amount of random numbers (Algorithm 5, line 9);
- The execution of a PSF Estimation algorithm on OpenCL device, except on first iteration (Algorithm 5, line 6).

Once the CPSO class finishes all iterations, it collects the Best values found during the optimization and then makes it available for the external use. The best values are collected by the CPSO algorithm while evaluating the *pBest* and *gBest* (Algorithm 5, line 8).

Algorithm 5 Run CPSO

```
1: Input: {$nCycles$}
2: Run PSF Estimator for 1 Swarm on OpenCL device (2.1) [1]
3: Copy the Results for All others Swarms on OpenCL device (2.2)
4: for $i = 1$ to $nCycles$  {on Host (5)}  do
5:    if $i != 1$ then
6:       Run PSF Estimation on OpenCL device (6)
7:    end if
8:    Run 1 Iteration of CPSO on OpenCL device (3)  {best values are collected here}
9:    Renew Random Seeds on OpenCL device (4)
10: end for
11: Collect the Best Values from OpenCL device (7)
```

[1]The numbers at the end of the lines on Algorithm 5 (2.1, 2.2, 3, ..) can be found on Figures 4.11 and 4.12, for better reference.

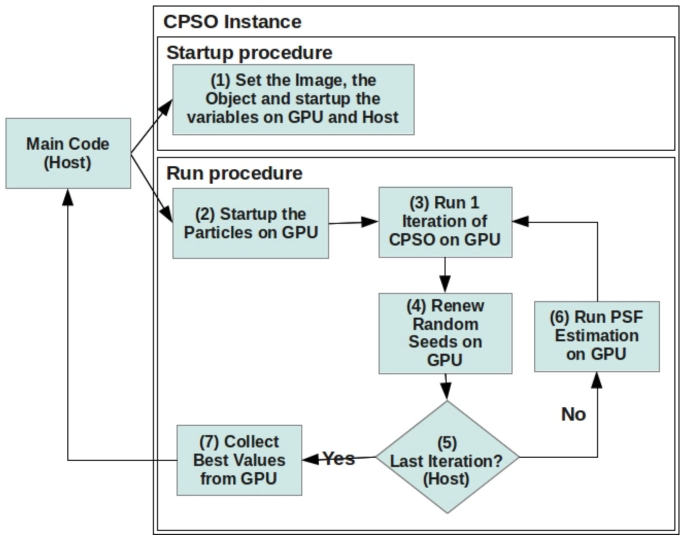

CPSO Instance
Startup procedure
(1) Set the Image, the Object and startup the variables on GPU and Host
Main Code (Host)
Run procedure
(2) Startup the Particles on GPU
(3) Run 1 Iteration of CPSO on GPU
(4) Renew Random Seeds on GPU
(6) Run PSF Estimation on GPU
No
(7) Collect Best Values from GPU
Yes
(5) Last Iteration? (Host)

Figure 4.11: CPSO workflow.
Two main procedures are executed by the CPSO instance: the *Startup* and the *Run*.

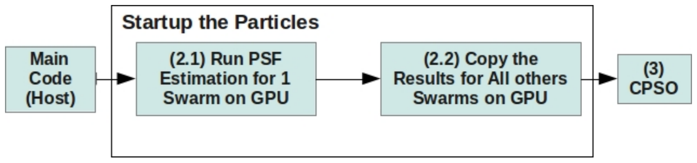

Startup the Particles
Main Code (Host)
(2.1) Run PSF Estimation for 1 Swarm on GPU
(2.2) Copy the Results for All others Swarms on GPU
(3) CPSO

Figure 4.12: Startup CPSO Particles.
The box (2.1) is the same procedure as presented on Figure 4.11 box(6)

```
12: return best values found
```

The Startup of the particles consists of one execution of the PSF Estimation for 1 swarm only (regardless of how many swarms the CPSO configuration actually contains - Figure 4.12, box (2.1)), followed by the copy of its results to all remaining swarms (box (2.2)). Since this is just a startup process for a CPSO algorithm, any value can be used for its initialization as long as they are valid values, therefore the copy of the results from a single swarm to all others swarms reduces the time required for the CPSO initialization. At the first iteration of the algorithm (Figure 4.11, boxes [3-6]), all values will be recalculated and new random positions will be assigned, then the copy of the results at Figure 4.12 (box (2.2)) is not a problem.

During an iteration of the CPSO algorithm (Algorithm 6), all particles update their *pBests* and then the first particle of each swarm updates the swarm's *gBest* (lines 2 and 3, respectively). Then, as a strategy to keep part of the swarm near to the best results found so far, the first particle of each swarm chooses randomly if it will have its own position replaced or not:

- If it decides by the replacement, it replaces the particle's position with the swarm's *gBest* (line 15);
- If not, it will do the same as all the other particles (i.e., normal particle update).

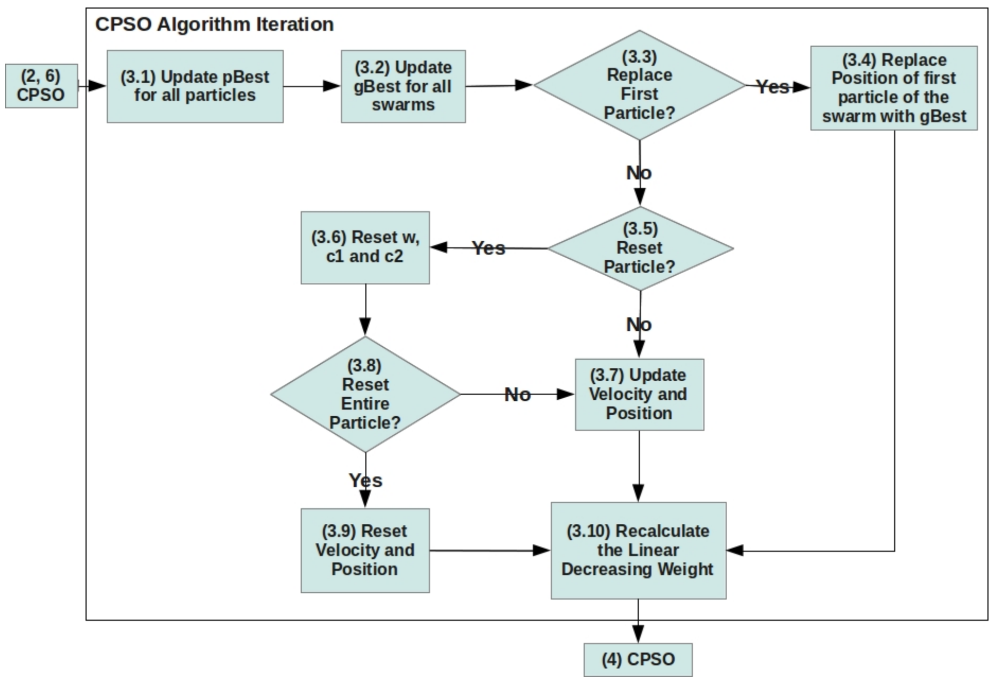

CPSO Algorithm Iteration
(2, 6) CPSO
(3.1) Update pBest for all particles
(3.2) Update gBest for all swarms
(3.3) Replace First Particle?
Yes
(3.4) Replace Position of first particle of the swarm with gBest
No
(3.5) Reset Particle?
Yes
(3.6) Reset w, c1 and c2
No
(3.8) Reset Entire Particle?
No
(3.7) Update Velocity and Position
Yes
(3.9) Reset Velocity and Position
(3.10) Recalculate the Linear Decreasing Weight
(4) CPSO

Figure 4.13: CPSO Algorithm Iteration.
This flowchart describes the box(3) presented on Figure 4.11. At this part of the implementation, the *gBest* for each swarm is updated, and the *pBest*, the velocity and the position for every particle is recalculated.

The intention with this random replacement is to reduce the impact caused by the particle reset strategy. To avoid any tendency on the particle's movements and to reduce conflicting updates between particle's replacement versus particle's reset, only one particle per swarm has its position replaced by the *gBest*, and the replaced position (regardless how good or bad it is) is not verified before replaced.

Algorithm 6 CPSO Algorithm Iteration

```
1: Input: $\{resetFactor, range, nIt, itMax\}$
2: Update $pBest$ for all particles (3.1) [1] {*based on Equation 2.12*}
3: First particle of every swarm updates the swarm's $gBest$ (3.2) {*based on Equation 2.13*}
4: $replace = random\ true/false$
5: if $nIt == itMax$ then
6:     $nIt = 0$
7:     $reset = true$
8:     $resetAll = random\ true/false$
9: else
10:    $reset = false$
11:    $resetAll = false$
12: end if
13: $x_{old} = x$
14: if (this is first particle on swarm) and ($replace$) (3.3) then
15:    Replace $x$ with $gBest$ (3.4)
16: else
17:    if $reset$ (3.5) then
18:        Create random values for $w$, $c_1$ and $c_2$ (3.6)
19:        if $resetAll$ (3.8) then
20:            Create random values for $v$ and $x$ (3.9)
21:        end if
22:    end if
23:    if $not\ resetAll$ then
24:        Update $v$ and $x$ constrained to the search space $[-range, +range]$ (3.7)
25:    end if
26: end if
27: Recalculate the Linear Decreasing Weight (LDW) for $w$ (Equation 2.14) (3.10)
28: if (Euclidean distance between $x_{old}$ and $x$) $<$ ($resetFactor * range$) then
29:    $nIt = nIt + 1$
30: else
31:    $nIt = 0$
32: end if
33: return $nIt$
```

Every particle that did not have its position replaced decides if it will be reset or not, based on the difference between the last particle's position x_{old} and the current calculated position x (Euclidean distance on line 28), being reset only if after a certain numbers of iterations $itMax$ this difference is still within a certain small percentage of the search space $resetFactor$ (line 5). Both $itMax$ and $resetFactor$ must be defined heuristically.

If a reset is required, a random decision is made to select between a simple or a full reset:

- If it is a Simple reset, it will create random values for its w, c_1 and c_2 (line 18);

[1] The numbers at the end of the lines on Algorithm 6 (3.1, 3.2,..) can be found on Figure 4.13, for better reference.

- If it is a Full reset, it will create random values for its w, c_1, c_2, v (velocity) and x (position) (lines 18 and 20).

The intention with the reset strategy on this work is to keep the CPSO as aggressive as possible (since it must obtain good results with a minimum number of iterations), but without compromising the quality of the estimations.

The particles that were not fully reset, now calculate new values for their v and x variables (line 24) according to the CPSO algorithm calculation for velocity and position (equations 2.10 and 2.11, respectively). Constraints for both v and x are added ($[-range, +range]$) to avoid them getting out of the search space. As a final step of this algorithm, a Linear Decreasing Weight (LDW) is applied over its w (line 27) to gradually reduce the convergence of the optimization.

4.2.5 Algorithm for Psf Estimation

The PSF Estimation algorithm is responsible for evaluating if the values found by the CPSO iteration is in fact a good result or not, acting as the Fitness function for this CPSO solution (Figure 4.14). This algorithm executes many sequential calculations that cannot be executed in parallel, thus explaining the difficulty of improving the fitness evaluation speed of this PSF estimation method. Besides, most of these calculations depends entirely on FFT's to obtain the values required by the next internal step of the algorithm, what causes an additional performance problem, given that no Fourier library available (at the time this work was developed) could calculate more than one image per kernel call. This means that the speed of this PSF estimation method could be further increased by using a library that can process multiple independent FFT's in parallel at a single kernel call on OpenCL device.

To reduce the impact caused by the FFT's serialization, at some parts of this solution the images calculated by all particles were grouped together, so that all of them could be processed at one single call of a device kernel, being split again when the FFT calculation was required. It was considered the balancing between the effort for the device to split/regroup the data vs. the benefit of calculating all the particle's data in parallel at once, making it parallel only at those places where it clearly would bring much more speed improvements than additional overhead.

The Algorithm 7 shows the places where these images are processed in parallel and the places where they are executed serially. See that most of the algorithm is sequential, computing one PSF at a time (Algorithm 7, lines [6 -39]), having parallel PSFs being evaluated only at lines 3, 41, 43 and 45.

Notice that all commands shown on Algorithm 7 (*phase_pupil*, *copy*, *FFT*, *calc_psf*, *reduce*, *scale*, *extract*, *convolution*, *cost* and *sync*) are actually being executed on OpenCL device (thus, they are actual kernel calls), but the control of every kernel is managed by

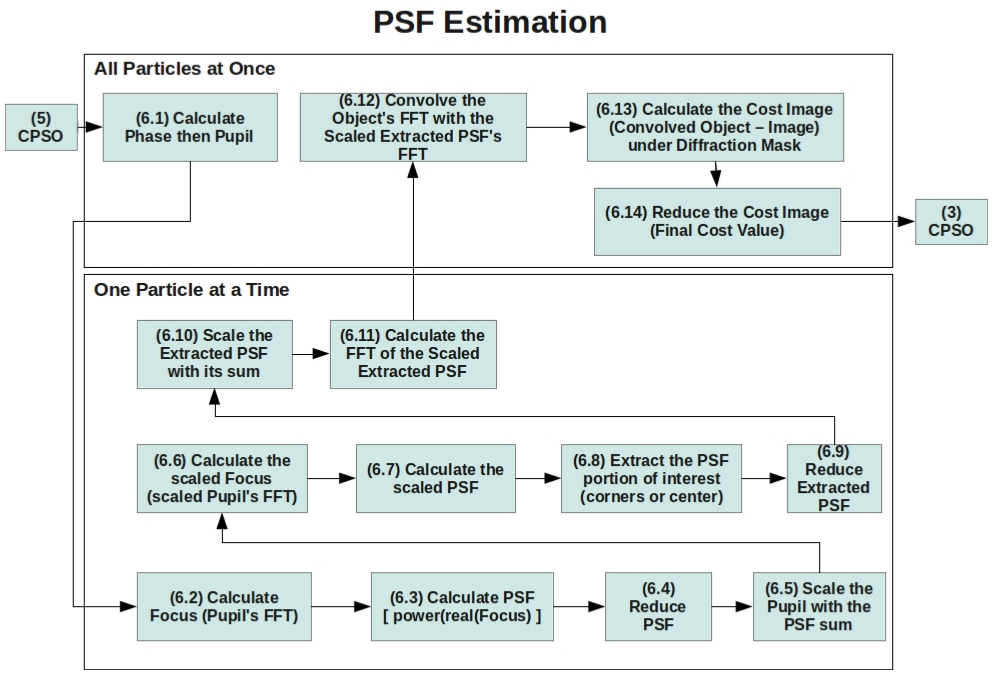

PSF Estimation
All Particles at Once
(5) CPSO
(6.1) Calculate Phase then Pupil
(6.12) Convolve the Object's FFT with the Scaled Extracted PSF's FFT
(6.13) Calculate the Cost Image (Convolved Object – Image) under Diffraction Mask
(6.14) Reduce the Cost Image (Final Cost Value)
(3) CPSO
One Particle at a Time
(6.10) Scale the Extracted PSF with its sum
(6.11) Calculate the FFT of the Scaled Extracted PSF
(6.6) Calculate the scaled Focus (scaled Pupil's FFT)
(6.7) Calculate the scaled PSF
(6.8) Extract the PSF portion of interest (corners or center)
(6.9) Reduce Extracted PSF
(6.2) Calculate Focus (Pupil's FFT)
(6.3) Calculate PSF [power(real(Focus)]
(6.4) Reduce PSF
(6.5) Scale the Pupil with the PSF sum

Figure 4.14: PSF Estimation workflow.
The PSF Estimation process has two main distinctions: the codes where only one image or phase is calculated at every kernel call, and the codes where they are calculated all together in a single kernel.

the Host (CPU). Also, the submission of a kernel is non blocking and, for that reason, it must be synchronized at some point. Therefore, the command *sync* represents the synchronization of the Command Queue and it is called after every group of kernels has been submitted to OpenCL. At the places where the data is grouped, the command *sync* is called immediately after the kernel call (eg., line 4). However, on those places where the data is split, the command *sync* is called only once after the entire group of kernels was enqueued (eg., line 7).

The input parameter $g\_coefs[]$ is a contiguous array containing the Zernike coefficients for all particles in all swarms, i.e., $g\_coefs[n]$ will access the Zernike coefficients for the particle n, and $nPSFs$ (line 2) will contain the total number of particles within all swarms (or sub-particles).

The term *Reduce* used by this algorithm refers to the technique of summing up all the values inside a vector by using parallel cores to access multiple contents at exact same time, what increases the speed of a sum. The reduction of the cost image (Algorithm 7, line 45) is the actual fitness result for each estimated PSF. Also, the term *pupil* (present on Figure 4.14 and Algorithm 7) is simply the result of the FFT applied over the phase (see phase calculation on Equation 2.4).

It is important to emphasize that the PSF Estimation algorithm deals with two types of images: the original image (external parameter) and the estimated image (calculated by the PSF Estimation algorithm). The original image is copied from external source to the OpenCL device's global memory only once and no duplication is made from this data. The estimated image is computed by Algorithm 7 and every sub-particle has its own version of such image. Therefore, every time the term *particle's image* is used in this work, it is referring to the images estimated by every sub-particle (random image) and not to the original image (external image).

Algorithm 7 Sequence of Kernel calls for PSF Estimation

```
1: Input: $\{g\_obj\_fft, g\_img\_fft, g\_coefs[], center\_corner, g\_diffraction\_mask\}$
2: $nPSFs = size(g\_coefs[])$
3: $g\_pupil = phase\_pupil(g\_coefs)$ $\{Calculate\ all\ phase's\ and\ pupil's\}$ (6.1) [1]
4: sync
5: for $n = 1$ to $nPSFs$ do:
6:     Copy from $g\_pupil[n]$ to $g\_pupil_n$ $\{Split\ to\ separate\ memory\ space\}$
7: sync
8: for $n = 1$ to $nPSFs$ do:
9:     $g\_focus_n = FFT(g\_pupil_n)$ $\{Calculate\ the\ focus\}$ (6.2)
10: sync
11: for $n = 1$ to $nPSFs$ do:
12:     $g\_psf_n = calc\_psf(g\_focus_n)$ $\{Calculate\ the\ PSF\}$ (6.3)
13: sync
14: for $n = 1$ to $nPSFs$ do:
15:     $summation_n = reduce(g\_psf_n)$ $\{Reduction\ of\ PSF\}$ (6.4)
16: sync
17: for $n = 1$ to $nPSFs$ do:
18:     $g\_pupil_n = g\_pupil_n / summation_n$ $\{Scale\ the\ pupil\}$ (6.5)
19: sync
20: for $n = 1$ to $nPSFs$ do:
21:     $g\_focus_n = FFT(g\_pupil_n)$ $\{Calculate\ the\ scaled\ focus\}$ (6.6)
22: sync
23: for $n = 1$ to $nPSFs$ do:
24:     $g\_psf_n = calc\_psf(g\_focus_n)$ $\{Calculate\ the\ scaled\ PSF\}$ (6.7)
25: sync
26: for $n = 1$ to $nPSFs$ do:
27:     $g\_psfe_n = extract(g\_psf_n, center\_corner)$ $\{Extract\ the\ portion\ of\ interest\ from\ the\ PSF\}$ (6.8)
28: sync
29: for $n = 1$ to $nPSFs$ do:
30:     $summation_n = reduce(g\_psfe_n)$ $\{Reduce\ the\ Extracted\ PSF\}$ (6.9)
31: sync
32: for $n = 1$ to $nPSFs$ do:
33:     $g\_psfe_n = g\_psfe_n / summation_n$ $\{Scale\ the\ Extracted\ PSF\}$ (6.10)
34: sync
35: for $n = 1$ to $nPSFs$ do:
```

[1]The numbers at the end of the lines on Algorithm 7 (6.1, 6.2, ..) can be found on Figure 4.14, for better reference.

```
36:     $g\_psfe\_fft_n = FFT(g\_psfe_n)$ {*Calculate the FFT of the Scaled Extracted PSF*} (6.11)
37: sync
38: for $n = 1$ to $nPSFs$ do:
39:     Copy $g\_psfe\_fft_n$ to $g\_psf\_fft[n]$ {*Regroup the Scaled Extracted PSF's to sequential memory*}
40: sync
41: $g\_conobj = convolution(g\_obj\_fft, g\_psf\_fft)$ {*Convolve the Object with all final PSFs*} (6.12)
42: sync
43: $g\_cost = cost(g\_conobj, g\_img\_fft, g\_diffraction\_mask)$ {*Calculate all Cost Images (Convolved Object - Image) under Diffraction Mask*} (6.13)
44: sync
45: $final\_costs = reduce(g\_cost)$ {*Reduce all Cost Images*} (6.14)
46: sync
47: return $final\_costs$ {*Vector of Final Cost Values*}
```

4.3 Considerations

The Simulated Annealing, as an optimization method that is built for search intensification, has proven to be a weak method for finding good PSF parameters quickly. The CPSO, on the other hand, is built to provide diversification, what is more appropriate for estimation of multiple non-correlated continuous variables as the ones required to estimate PSFs.

An extensive description of the proposed PSF estimation method with CPSO optimization for OpenCL enabled devices was presented on this chapter, as long as implementation details for the execution on OpenCL devices.

The next chapter presents the calibration method proposed to adjust the CPSO for low-cost good results, besides presenting how the experiments were conducted. The results obtained for every experiment are also shown and discussed.

CHAPTER 5

EXPERIMENTAL RESULTS

In order to validate the results obtained by this method, a few distinct experiments were executed:

1. To calibrate the CPSO parameters (section 5.1):
   A measurable CPSO calibration method is proposed to quickly determine a low-cost configuration set that provides good and stable results. Different artificially distorted images (Algorithm 10) are generated for every PSF estimation.
2. To validate the quality of the results obtained by the CPSO developed in this work (section 5.2):
   New sets of artificial images are generated to validate the quality of the estimation. Different degrees of noise are added to the images (no noise, moderate noise and high noise) to verify noise robustness. A validation procedure is described in details, then the best and the worst results are presented and analyzed. Since the solar images are of low-frequency, the method is also validated by estimating PSFs of high-frequency artificial images (generated from the Lena image).
3. To verify how robust is the solution regarding the number of Zernikes and images sizes (section 5.3):
   Different number of Zernikes were used while estimating PSFs, then the processing times were measured and compared to determine the impact on the running time as the number of dimensions increases. Also, a calibrated CPSO was executed against a similar image of different size, to verify if the calibrated parameters can be reused.

The section 5.4 discusses processing times, comparing this current work with the previous work developed in [53].

All experiments were executed on a machine with the following configuration:

- Intel Core i7-975 CPU (4 cores, 8 threads, 3.33 GHz, cache L2 1MB and L3 8MB)
- 6GiB RAM
- GPU card model NVIDIA Tesla C2050 (14 Multiprocessors, Warp size 32, 448 cores, max. Block size 1024, 1.15 GHz, 3GB GDDR5, CUDA 3.2 OpenCL 1.0)
- Operating system Linux version 3.2.0-3-amd64 (Debian 3.2.23-1) (gcc version 4.6.3 (Debian 4.6.3-8))

5.1 CPSO Calibration

Because the CPSO contains parameters that are very relevant for the quality of the solution and for the speed of the convergence, the CPSO parameters must be calibrated for every different problem before running the experiments. Various factors like the processing time, the convergence behavior and the memory usage must be considered.

Given the possible combination of parameters are infinite, to determine a unique set of parameters for all the experiments, the parameters that have no direct impact on the processing time for the CPSO optimization method proposed on this work were fixed heuristically, based on previous experiments:

- $w = 1.25$
- $c_1 = 1$
- $c_2 = 0.5$
- $resetFactor = 0.25$
- $nZernikes = 50$

The range of the Zernike coefficients was set to [-2, 2], representing the size of search space. For the other parameters (*nSwarms*, *nParticles*, *nCycles*), 5 possible combinations were chosen considering both the hardware limitations and previous experiences, in such a way that these configurations would involve some good representation of the possible combinations:

- 1 swarm, 256 particles (standard PSO)
- 3 swarms, 32 particles
- 3 swarms, 128 particles
- 5 swarms, 64 particles
- 10 swarms, 32 particles

When dealing with any PSO algorithm, one important configuration item that must be carefully considered is the number of cycles (or iterations). In general, the larger the number of iterations, the better the result obtained. However, on this specific PSF estimation problem, one iteration means one fitness evaluation for every sub-particle within each swarm inside the CPSO. Since the fitness evaluation implies calculating the error function defined on Equation 2.5, which in turn requires calling the PSF Estimation code (Algorithm 7), every iteration of the CPSO algorithm requires $[nSwarms \times nParticles]$ PSF Estimations. This means that large CPSO configurations requires large processing time per iteration. For that reason, the chosen *nSwarms* and *nParticles* configurations specified previously are considering both the hardware limitations on GPU devices (including memory limitation) and the time required to complete one iteration.

The are a few ways of defining the number of swarms and how they will be divided, for example they can be defined according to any correlation that the particle dimensions could have, or following some previous knowledge over the problem, or arbitrarily chosen for testing purposes. Since the Zernikes dimensions are uncorrelated, for the purpose of this work the dimensions are divided in equal sizes and the number of swarms are chosen arbitrarily, respecting the hardware limitations.

To determine the number of cycles to be used, a good convergence point must be chosen considering both obtaining good results and running as few fitness evaluations as possible.

Given a reasonable number of fitness evaluations, and considering that it is guaranteed the fitness values will always be decreasing for a minimization problem, the criteria defined on Equation 5.1 was used in this work to find a minimum number of evaluations E_e (earliest evaluation) required to reach a low cost fitness stabilization point, where C_e (earliest cycle) is the respective number of cycles (or iterations).

$$E_e(f) = C_e(f) \times nSwarms \times nParticles \tag{5.1}$$

For every PSF estimation, a fitness cut point (F_c) was calculated by the Equation 5.2, where n is the number of cycles (or iterations) the CPSO was executed, $f(i)$ is the best fitness value obtained at CPSO iteration i, and $\overline{f}$ is the mean of all values in f. The number of cycles required to reach the fitness cut point was calculated by the Equation 5.3, by searching for the iteration i whose fitness value $f(i)$ is on the upper edge of F_c.

$$F_c(f) = (\overline{f} + \sqrt{\tfrac{1}{n-1}\textstyle\sum_{i=1}^{n}(f(i) - \overline{f})^2}), 1 \leq i \leq n \tag{5.2}$$

$$C_e(f) = i \mid \begin{cases} f(i) & >= & F_c(f) \\ f(i+1) & <= & F_c(f) \end{cases} \tag{5.3}$$

On CPSO optimizations, a large number of execution cycles tends to produce very similar (but decreasing) fitness values for most of the results after the convergence point was reached. Since the intention is to discover an earliest low-cost convergence point, these similar values must be excluded before the calibration process begins. Therefore, to determine a better convergence point for the chosen configurations, they all were run until the mark of 2 million fitness evaluations for one artificial Solar image, then the worst resulting convergence point from all configurations was chosen as the cut point $cutEvals$ to find the earliest general convergence point $nCycles$ (Algorithm 8, where $O(x)$ represents the original object and cfg_i represents every chosen configuration).

Algorithm 8 Calculate the Cut point to run the Calibration

```
1: Input: ${O(x), cfg(nSwarms, nParticles)[]}$
2: for $i = 1$ to $size(cfg)$ do
3:     Generate one artificial image $I_a(x)$ from $O(x)$ [1]
4:     Run $CPSO(O(x),\ I_a(x),\ cfg_i \rightarrow nSwarms, cfg_i \rightarrow nParticles)$ up to $2M$ evaluations
5:     Calculate $E_e$ based on equations 5.1, 5.2 and 5.3
6:     if $E_e$ is the worst result {higher value} then
7:         $cutEvals = E_e$
8:     end if
9: end for
10: return $cutEvals$
```

Once the *cutEvals* was found, a set of 30 artificial Solar images of size 128x128 were processed up to *cutEvals* fitness evaluations for the 5 configurations, then their times and earliest convergence cycles C_e were measured (Algorithm 9). The mean of these measurements (line 10) was used as criteria to manually define a calibrated set of CPSO parameters that satisfies the needs.

Algorithm 9 Calibrate the CPSO parameters *nSwarms*, *nParticles* and *nCycles*

```
1: Input: ${O(x), cfg(nSwarms, nParticles)[], cutEvals, nImages}$
2: $K = size(cfg)$
3: for $i = 1$ to $K$ do
4:     Generate $nImages$ artificial images $I_a(x)$ from $O(x)$
5:     for $j = 1$ to $nImages$ do
6:         Run $CPSO(O(x),\ I_{a,j}(x),\ cfg_i \rightarrow nSwarms, cfg_i \rightarrow nParticles)$ up to $cutEvals$ evaluations
7:         Calculate $C_{e,j}$ based on equations 5.2 and 5.3
8:         Store $C_{e,j}$
9:     end for
10:    $meanC_i = mean(C_e)$ {mean of the nImages previous estimations}
11:    $stddevC_i = stddev(C_e)$ {standard deviation of the nImages previous estimations}
12:    $vSwarms_i = cfg_i \rightarrow nSwarms$
13:    $vParticles_i = cfg_i \rightarrow nParticles$
14: end for
15: $results = [meanC,\ stddevC,\ vSwarms,\ vParticles]$
16: Present K $results$ and wait for user select visually the best result $K_{best}$
17: $nSwarms = results \rightarrow vSwarms[K_{best}]$
18: $nParticles = results \rightarrow vParticles[K_{best}]$
19: $nCycles = results \rightarrow meanC[K_{best}] + results \rightarrow stddevC[K_{best}]$
20: return $nSwarms,\ nParticles,\ nCycles$
```

As the final calibration result, the swarm and particle configuration from the matrix *results* (Algorithm 9, line 15) that presented the more suitable behavior (K_{best}) is visually chosen by the user as the calibrated CPSO parameters:

[1]The Algorithm 10 describes how the artificial image $I_a(x)$ is generated.

- $nSwarms$ (line 17);
- $nParticles$ (line 18);
- $nCycles$ (line 19): The earliest convergence cycle $nCycles$ was calculated as the mean plus standard deviation of the convergence points from the chosen configuration K_{best}.

All artificial images were calculated based on the Algorithm 10. Every image is artificially generated (I_a) by creating one random PSF (PSF_r, line 4) and convolving it with the original object O_o (line 5).

Algorithm 10 Generate Random PSF (PSF_r) and Artificial Image (I_a)

```
1: Input: ${O_o(x, s), nZernikes, range}$
2: Generate $\gamma$ as: $nZernikes$ random numbers from $[-range, +range]$ and normal distribution ($mean = 0, standard\ deviation = 0.5$)
3: Generate the phase $\phi$ based on equation 2.4 and $\gamma$
4: Calculate the $PSF_r$ based on equation 2.3 and $\phi$
5: $I_a = O_o \circ PSF_r$   {generate one artificial image (convolution)}
6: return $PSF_r$, $I_a$
```

It is important to mention that every CPSO parameter is meaningful and can make significant difference for the overall results of the experiments, therefore the convergence behavior still have many points of possible improvements which are subject to further experiments on future works.

5.1.1 Calibration Results

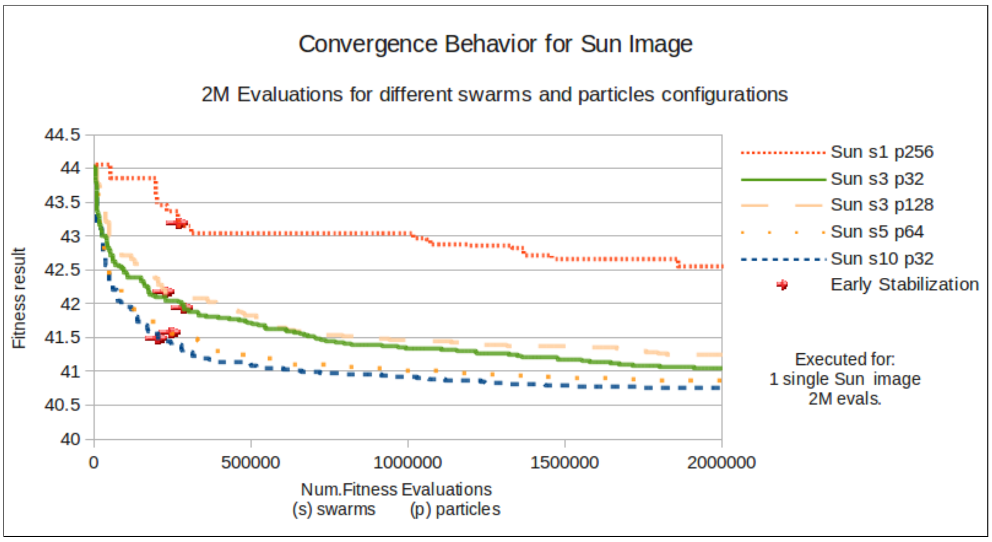


Figure 5.1: Convergence behavior for Sun image.

To determine a low cost convergence point, one Solar image was processed for the 5 chosen configurations of swarms(s) and particles(p) ($[s = 1, p = 256]$; $[s = 3, p = 32]$; $[s = 3, p =$

128]; [$s = 5, p = 64$]; [$s = 10, p = 32$]) up to 2 millions fitness evaluations each (Figure 5.1), then the slowest initial convergence point $cutEvals$ found was taken as the upper limit to initiate the calibration.

We can see on Figure 5.1 the standard PSO ([$s = 1, p = 256$]) behaving irregularly on its convergence, whereas the CPSO with the larger number of partitions ([$s = 10, p = 32$]) presented the most stable behavior while running a large number of evaluations, what demonstrates the power of the cooperation between multiple swarms while controlling the convergence of the optimization.

The configurations with the same number of partitions ([$s = 3, p = 32$] and [$s = 3, p = 128$]) presented very similar behaviors, with the smaller number of particles ($p = 32$) obtaining slightly better stabilization than with more particles ($p = 128$). This shows that for a large number of evaluations, more partitions helped to stabilize the convergence, but more particles did not improve the convergence.

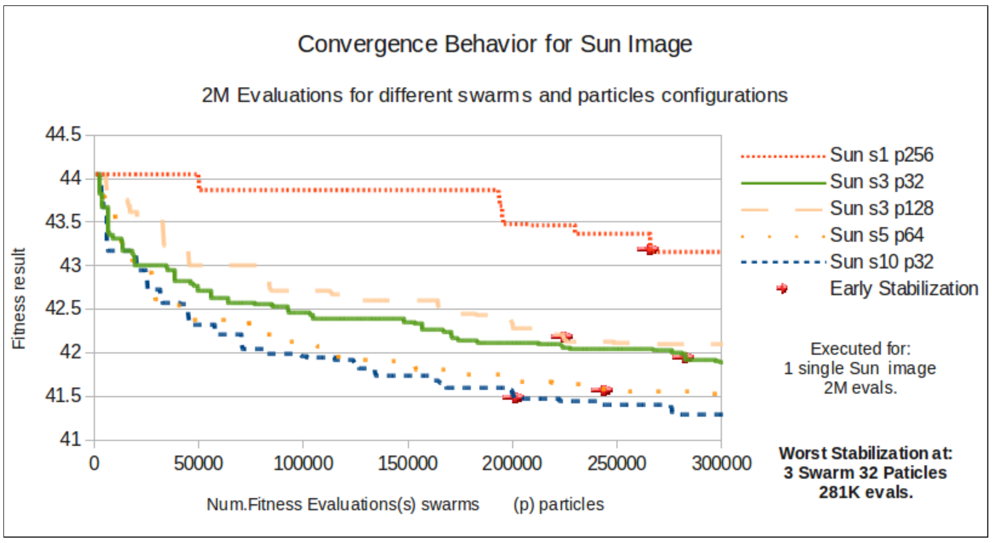


Figure 5.2: Slowest Convergence for Sun image.
The slowest convergence for the solar image was obtained at [$s = 3, p = 32$] (by using the algorithm 8), requiring $cutEvals = 281K$ evaluations to begin converge.

The Figure 5.2 shows more clearly how the number of particles affected the configurations ([$s = 3, p = 32$]) and ([$s = 3, p = 128$]). While the smaller number of particles obtained a more stable behavior, the larger number of particles resulted in an early stabilization of lower cost, but both with very similar fitness values.

Even though the configuration ([$s = 5, p = 64$]) obtained the third place in the ranking of early stabilization, its resulting fitness value obtained the second place and the configuration ([$s = 10, p = 32$]) obtained the first place, what initially suggest that ([$s = 10, p = 32$]) would the best option for a high cost optimization.

The early stabilization of the configuration ([$s = 3, p = 32$]) was selected to set a value for the parameter $cutEvals$ ($cutEvals = 281K$).

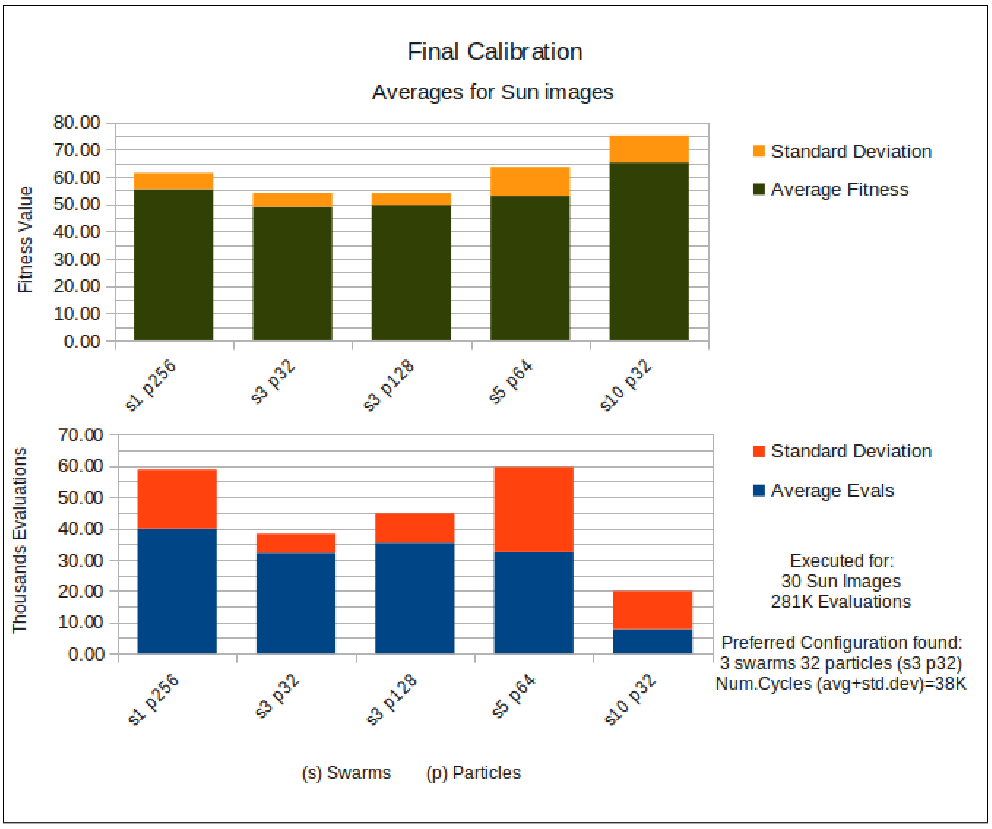


Figure 5.3: Calibration Result for Sun Image.

To calibrate the swarms(s), the particles(p) and the number of evaluation cycles(c) for a low-cost optimization, the PSF estimation was executed for 30 artificial Solar images (generated from the same original object) up to 281K fitness evaluations ($cutEvals$ is described on Figure 5.2), then their averages and standard deviations were collected for both the earliest stabilization and the fitness values (as described by the algorithm 9).

The Figure 5.3 shows that for a low cost optimization, the configurations ($[s = 1, p = 256]$) and ($[s = 5, p = 64]$) presented bad standard deviations on their earliest stabilizations while evaluating the fitness 281K times. Their averages plus standard deviations obtained the worst ranking on running time, but on the other hand they obtained the third and forth places in the fitness value ranking.

The configuration ($[s = 10, p = 32]$) presented a very quick stabilization, but when looking at its poor fitness value result, we can see that it happened because the average fitness values did not converge easily after a short number of evaluations.

The configurations ($[s = 3, p = 32]$) and ($[s = 3, p = 128]$) presented very similar results again on the fitness values, but ($[s = 3, p = 32]$) obtained the same fitness results more quickly and with better standard deviations. We can see here an opposite behavior as for the high cost optimization, because a smaller number of particles stabilized more quickly than with a large number of particles. Also, a larger number of partitions ($[s = 5, p = 64]$ and $[s = 10, p = 32]$) presented unstable behaviors for a low cost optimization and therefore cannot be trusted for the experiments on following sections.

After evaluating the calibration results, the winner configuration K_{best} is the $([s = 3, p = 32])$, where the average number of fitness evaluations required to reach the low cost fitness stabilization area was 32K and its standard deviation was 6K ($nCycles = (32K\ avg. + 6K\ stddev.)/(3\ swarms \cdot 32\ particles) \approxeq 396\ cycles$). Therefore the final calibration result was:

- $s = 3$
- $p = 32$
- $c = 396$

Notice the apparent correlation between the warp size (32) and the number of particles found by the calibration (also 32), what might suggest that the algorithms that use one thread per particle could benefit from a configuration with the number of particles equals to the warp size.

Given the time required to evaluate each image was very similar among all executions (2080±24 seconds for 2M evaluations and 87±0.7 seconds for 281K evaluations), the time was omitted from the figures for simplification reasons.

5.2 CPSO Validation

Once the parameters were calibrated, the CPSO was validated by processing another set of 100 artificial Solar images of size 128x128, (generated by the Algorithm 10). Notice that these artificial images are different from the images used during the calibration process, what allows for a more accurate validation of how stable this method actually is. The parameters $nSwarms$, $nParticles$ and $nCycles$ found during the calibration phase were used here, as well as the fixed parameters w, c_1, c_2, $resetFactor$ and $nZernikes$.

As we can see in the Algorithm 11, every output result is composed of the random PSF (PSF_r), the estimated PSF (PSF_e), the artificial image (I_a), the estimated image (I_e), and the deconvolved image (I_d, line 16, where $\bar{\circ}$ represents a deconvolution).

Algorithm 11 Validate the CPSO

```
1: Input: $\{O_o(x), nImages, nSwarms, nParticles, nCycles, nKeep, N_p\}$
2: **for** $i = 1$ *to nImages* **do**
3:    Generate $PSF_r$ and $I_a$ based on Algorithm 10
4:    **if** $N_p > 0$ **then**
5:       $I_a = I_a + N_p$  {*add Poisson noise*}
6:    **end if**
7:    Run $CPSO(O_o(x),\ I_a(x),\ nSwarms,\ nParticles,\ nCycles)$
8:    **if** Fitness result is among the $nKeep$ best results **then**
9:       Keep $PSF_r, PSF_e, I_a, I_e$
10:   **end if**
11:   **if** Fitness result is among the $nKeep$ worst results **then**
12:      Keep $PSF_r,\ PSF_e,\ I_a,\ I_e$
13:   **end if**
14: **end for**
15: **for** every kept result  {*best and worst results*}  **do**
16:    $I_d = I_a \bar{\circ} PSF_e$  {*deconvolve artificial image with estimated PSF*}
17:    Keep $I_d$
18: **end for**
19: **return**  best and worst vectors $PSF_r,\ PSF_e,\ I_a,\ I_e,\ I_d$
```

This same procedure was also executed for artificial images corrupted by Poisson noise (N_p, line 5):

- 100 new artificial images corrupted by moderate Poisson noise;
- Another set of 100 artificial images severely corrupted by Poisson noise.

5.2.1 Images Obtained by the Calibrated CPSO

The images presented on Table 5.1 through 5.12 show the visual results obtained by the CPSO, calibrated with the following configuration:

- $w = 1.25$
- $c_1 = 1$

- $c_2 = 0.5$
- $resetFactor = 0.25$
- $nZernikes = 50$
- $s = 3$
- $p = 32$
- $c = 396$

The algorithm 11 was used to return the images presented in each table below (with $nKeep = 4$). The resulting images were obtained after running the calibrated CPSO for 100 artificial images of size 128x128 (generated from the same original object), then collecting the best and the worst results (based on fitness value calculated by the Algorithm 7) and presenting on separated tables. Separate executions over different artificial images were submitted for:

- Artificial Images (tables 5.1, 5.2, 5.3 and 5.4).
- Artificial Images corrupted by moderated noise (tables 5.5, 5.6, 5.7 and 5.8).
- Artificial Images severely corrupted by noises (tables 5.9, 5.10, 5.11 and 5.12).

The image deconvolution (I_d) was calculated by the BiaQIm image processing suite, an external program that is described on [48].

A separated calibration was executed for the Lena image (size 128x128) then 100 artificial images were processed. The best and the worst results were collected and presented in Tables 5.13 and 5.14.

The Tables 5.1 and 5.3 show the best and the worst results (respectively) for the Solar image, where we can see a clear distinction between the images ($O_o - I_a$) [a clear solar image] and ($O_o - I_d$) [a grayed image], what demonstrates that the PSF estimation obtained a very approximate result (PSF_e) from the real one (PSF_r), even for the worst results. The RMS contrast[1] showed how different the best artificial image ($rms \approxeq 0.158$) was from the original object ($rms \approxeq 0.206$), where the deconvolved image nearly restored the original contrast ($rms \approxeq 0.200$). Even on the worst result, the RMS contrast was recovered from a bad contrast $\approxeq 0.129$ to a good approximation from the original $\approxeq 0.193$.

Notice that the worst result was not caused by any unstable behavior of the solution, but instead it was caused by the fact that the artificial image (I_a) was much more blurred on the worst result than the artificial image on the best result. Also, the dark border on the estimated image (I_e) is generated by the windowing function used to reduce the leakage effect caused by the Fourier calculations.

The Tables 5.2 and 5.4 show how near every PSF estimation have achieved from the random PSF by presenting the subtraction ($PSF_r - PSF_e$). We can see that almost empty PSF images were obtained on these subtractions, both on the best and on the worst results, with the most important values matching each other.

[1]RMS contrast is defined on section 2.1.

The Tables 5.5 and 5.7 show the best and the worst results (respectively) for the Solar image with addition of moderate Poison noise. We can see by the resulting images that this solution deals inherently with the presence of noise, with the final subtraction image $(O_o - I_d)$ still presenting a grayed image. The PSF subtractions presented on the Tables 5.6 and 5.8 show a matching level slightly below the matching obtained by the PSFs without addition of noise, but still they are close results from the correct PSFs.

The images severely corrupted by noise presented on Tables 5.9 and 5.11 demonstrate how robust is this solution for noised blurred images. Even under strong presence of noise, the resulting PSFs (on Tables 5.10 and 5.12) remain stable and near from the actual PSF_r.

The time required to process these 100 Solar images with the configuration specified in the beginning of this section was 39.44±0.18 seconds per image, summing up a total of 3950 seconds for the 100 images.

Notice that the purpose of the CPSO to estimate the PSF is not find the perfect match of the correct PSF, but instead it is built to find the closest PSF it can, given the number of iterations, the number of swarms (or partitions) and the number of particles. Therefore, the calibration process is meant to discover the cheapest satisfactory result for an acceptable processing time. The results obtained by the Solar images demonstrate that both the PSF Estimation and the Calibration process achieved their purposes, since good resulting final images and approximate PSFs were obtained in all cases.

To validate how robust is the solution for images of different frequencies, the Lena image was calibrated separately (with results: $s = 3$, $p = 32$, $z = 50$ and $c = 512$ ($49K$ evals.)), then 100 artificial images were generated (from the same original Lena object) and the PSF estimation was executed for these artificial images. The results were presented on Tables 5.13 and 5.14, where we can see close results again for both the PSFs generated and for the estimated images. Both the best and worst results for $(O_o - I_d)$ were very similar, what shows a stable behavior while processing high frequency images. However, the highest frequencies appear as residual values on these same subtractions (with the deconvolved image I_d), possibly caused by the deconvolution process that was used to deconvolve the artificial images.

For all the images used for validation, the worst results (generated from the most severely corrupted images for each set of 100 images) obtained a good the contrast recovering on the deconvolved image, when compared to the respective best results.

As a final validation, the PSF_r and the PSF_e were compared at every iteration and the difference was used to compare with the fitness value. It was found that the PSFs' differences oscillate up and down whereas the fitness value is decreasing, what is likely caused by the fact that many different PSFs can generate approximate images. Therefore, the PSF difference does not help while estimating the PSF on this method.

Table 5.1: Best Results for Sun Images.
RMS contrasts: $O_o = 0.205623$, $I_a = 0.158230$, $I_d = 0.199694$

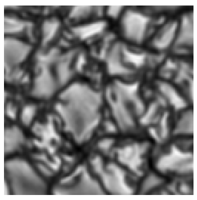
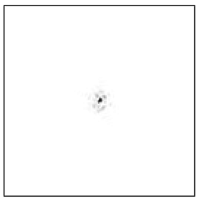
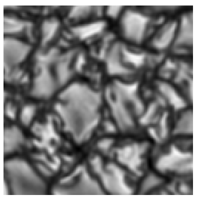
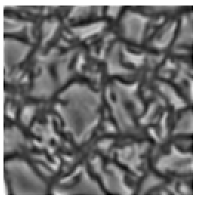
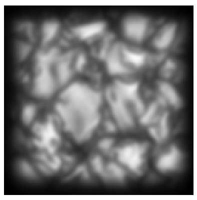
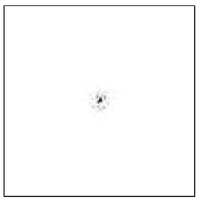
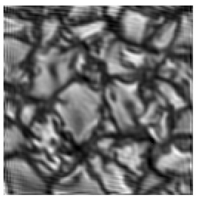
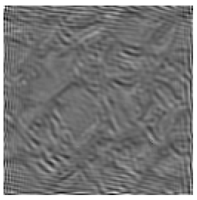
Original Object (O_o)	**Random PSF** (PSF_r)	**Artificial Image** (I_a)	**Difference** ($O_o - I_a$)
			
Estimated Image (I_e)	**Estimated PSF** (PSF_e)	**Deconvolved Image** (I_d)	**Difference** ($O_o - I_d$)
			

Table 5.2: Best Results for Sun Images - PSFs Comparisons.
Configuration used: {[100 Artificial Sun Images, kept the best and the worst], [CPSO: s=3, p=32, z=50, c=396 (38K evals.)], [Images: 128x128]}.

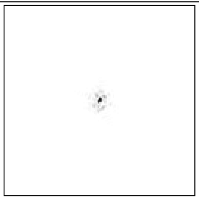
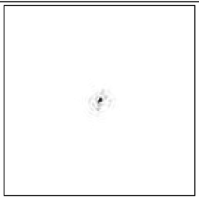
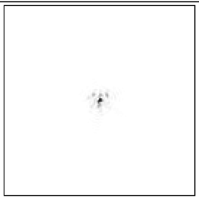
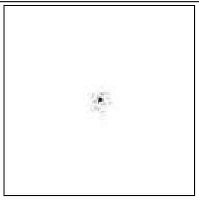
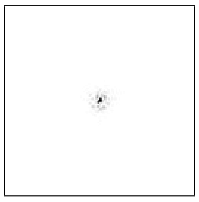
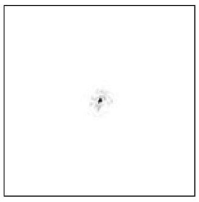
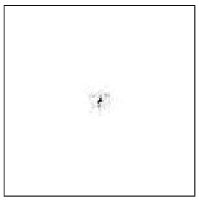
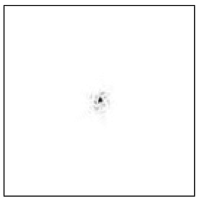
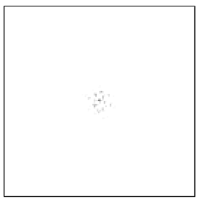
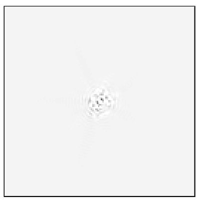
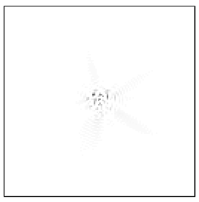
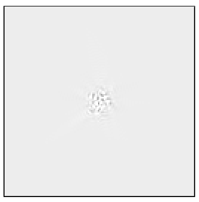
Rank	#1	#2	#3	#4
Random PSF (PSF_r)				
Estimated PSF (PSF_e)				
Difference ($PSF_r - PSF_e$)				

Table 5.3: Worst Results for Sun Images.
RMS contrasts: $O_o = 0.205623$, $I_a = 0.129243$, $I_d = 0.192615$

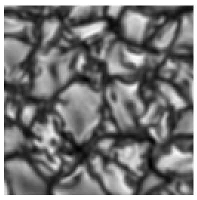
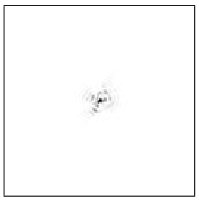
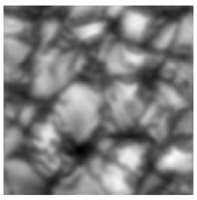
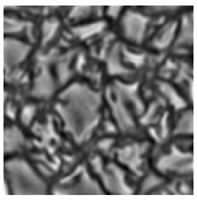
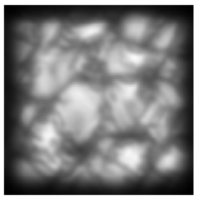
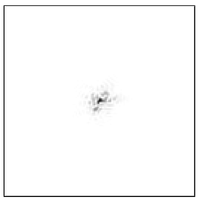
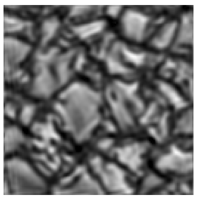
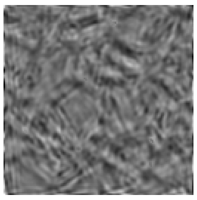
Original Object (O_o)	**Random PSF** (PSF_r)	**Artificial Image** (I_a)	**Difference** ($O_o - I_a$)
			
Estimated Image (I_e)	**Estimated PSF** (PSF_e)	**Deconvolved Image** (I_d)	**Difference** ($O_o - I_d$)
			

Table 5.4: Worst Results for Sun Images - PSFs Comparisons.
Configuration used: {[100 Artificial Sun Images, kept the worst and the worst], [CPSO: s=3, p=32, z=50, c=396 (38K evals.)], [Images: 128x128]}.

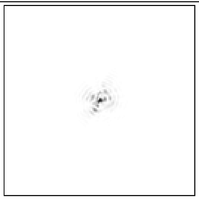
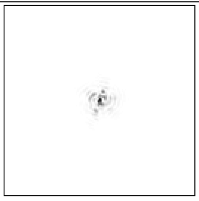
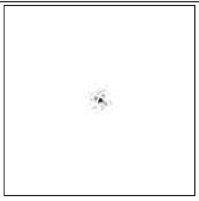
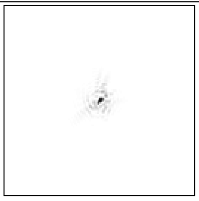
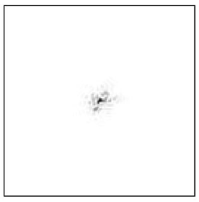
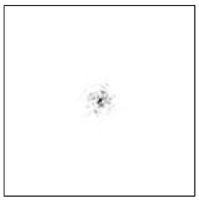
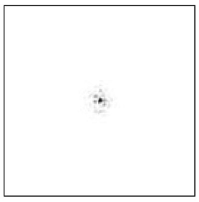
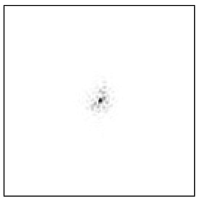
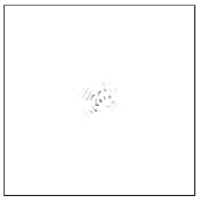
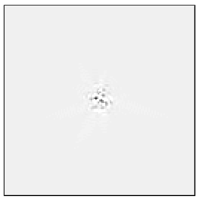
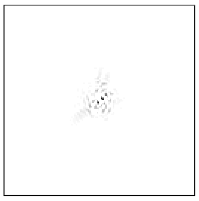
Rank	#1	#2	#3	#4
Random PSF (PSF_r)				
Estimated PSF (PSF_e)				
Difference ($PSF_r - PSF_e$)				

Table 5.5: Best Results for Sun Images with moderate Poison noise.
RMS contrasts: $O_o = 0.205623$, $I_a = 0.150366$, $I_d = 0.187255$

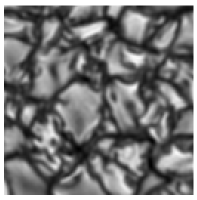
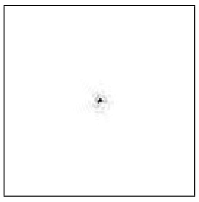
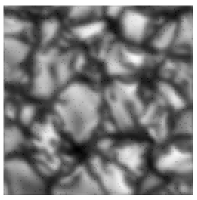
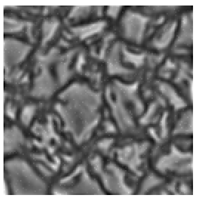
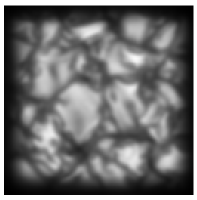
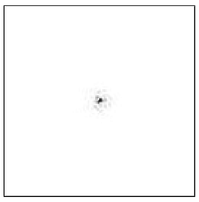
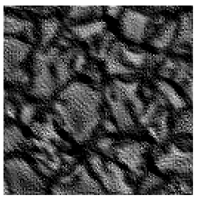
Original Object (O_o)	**Random PSF** (PSF_r)	**Artificial Image** (I_a)	**Difference** ($O_o - I_a$)
			
Estimated Image (I_e)	**Estimated PSF** (PSF_e)	**Deconvolved Image** (I_d)	**Difference** ($O_o - I_d$)
			

Table 5.6: Best Results for Sun Images with moderate Poison noise - PSFs Comparisons.
Configuration used: {[100 Artificial Sun Images, kept the best and the worst], [CPSO: s=3, p=32, z=50, c=396 (38K evals.)], [Images: 128x128]}.

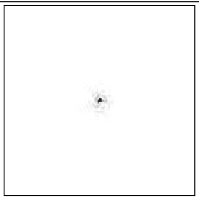
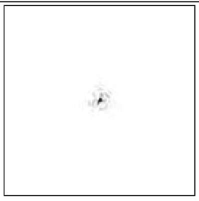
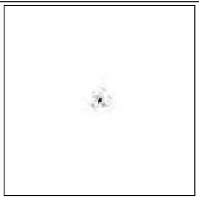
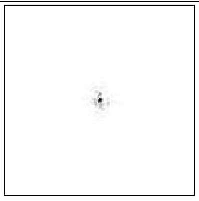
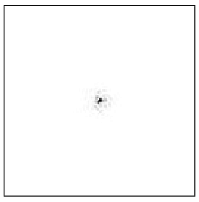
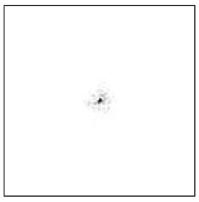
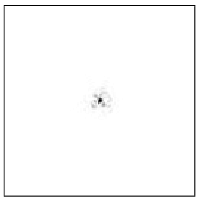
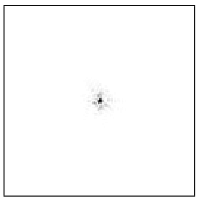
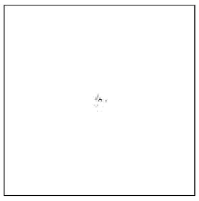
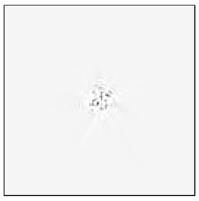
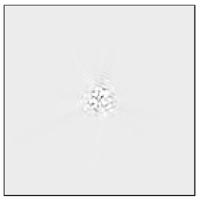
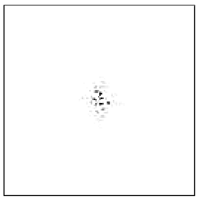
Rank	#1	#2	#3	#4
Random PSF (PSF_r)				
Estimated PSF (PSF_e)				
Difference ($PSF_r - PSF_e$)				

Table 5.7: Worst Results for Sun Images with moderate Poison noise.
RMS contrasts: $O_o = 0.205623$, $I_a = 0.146568$, $I_d = 0.190736$

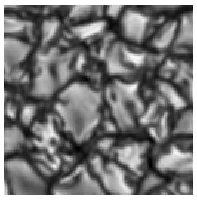
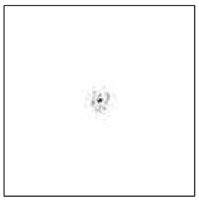
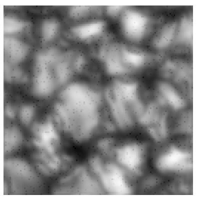
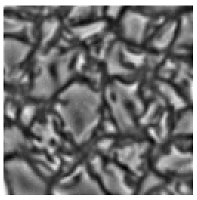
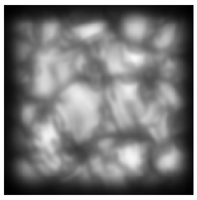
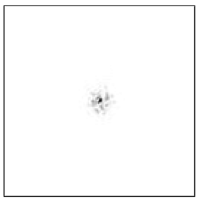
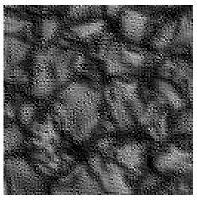
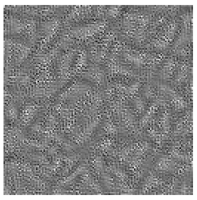
Original Object (O_o)	**Random PSF** (PSF_r)	**Artificial Image** (I_a)	**Difference** ($O_o - I_a$)
			
Estimated Image (I_e)	**Estimated PSF** (PSF_e)	**Deconvolved Image** (I_d)	**Difference** ($O_o - I_d$)
			

Table 5.8: Worst Results for Sun Images with moderate Poison noise - PSFs Comparisons.
Configuration used: {[100 Artificial Sun Images, kept the worst and the worst], [CPSO: s=3, p=32, z=50, c=396 (38K evals.)], [Images: 128x128]}.

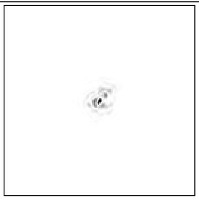
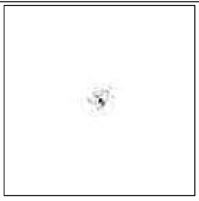
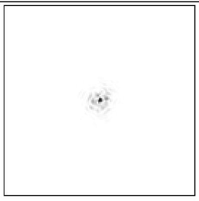
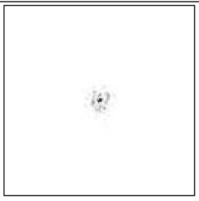
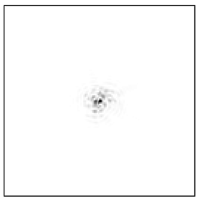
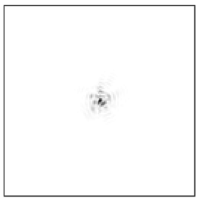
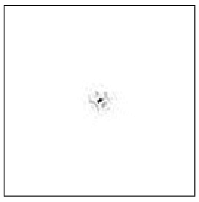
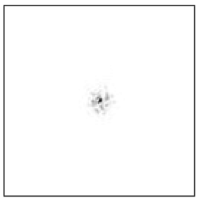
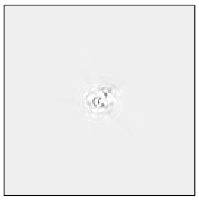
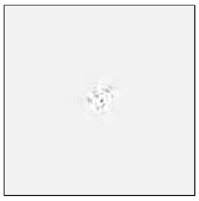
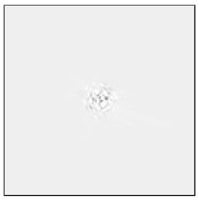
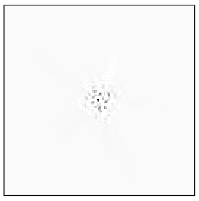
Rank	#1	#2	#3	#4
Random PSF (PSF_r)				
Estimated PSF (PSF_e)				
Difference ($PSF_r - PSF_e$)				

Table 5.9: Best Results for Sun Images severely corrupted by Poison noise.
RMS contrasts: $O_o = 0.205623$, $I_a = 0.170690$, $I_d = 0.194117$

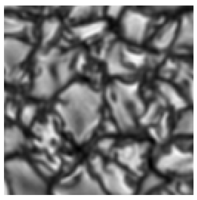
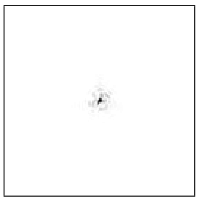
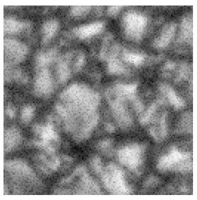
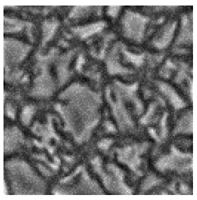
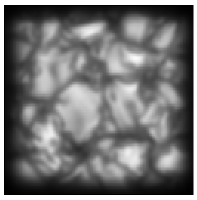
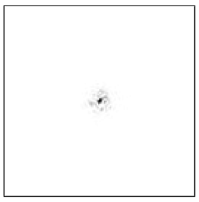
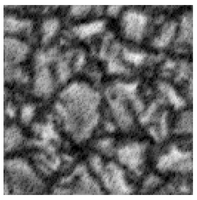
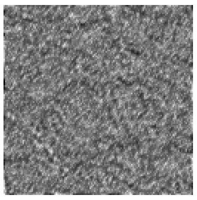
Original Object (O_o)	**Random PSF** (PSF_r)	**Artificial Image** (I_a)	**Difference** ($O_o - I_a$)
			
Estimated Image (I_e)	**Estimated PSF** (PSF_e)	**Deconvolved Image** (I_d)	**Difference** ($O_o - I_d$)
			

Table 5.10: Best Results for Sun Images severely corrupted by Poison noise - PSFs Comparisons.
Configuration used: {[100 Artificial Sun Images, kept the best and the worst], [CPSO: s=3, p=32, z=50, c=396 (38K evals.)], [Images: 128x128]}.

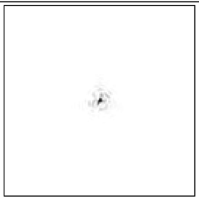
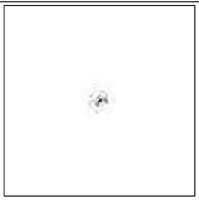
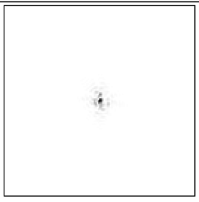
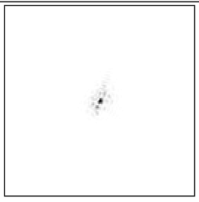
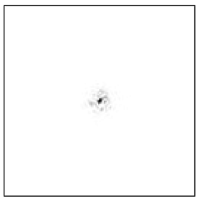
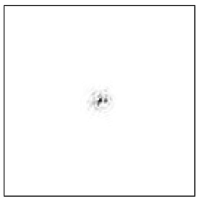
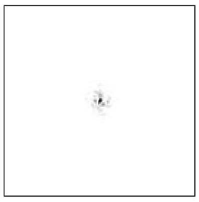
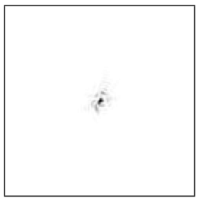
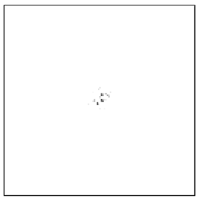
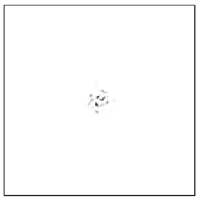
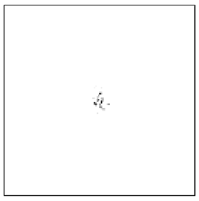
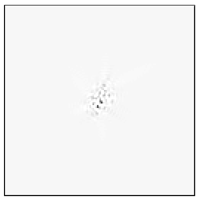
Rank	#1	#2	#3	#4
Random PSF (PSF_r)				
Estimated PSF (PSF_e)				
Difference ($PSF_r - PSF_e$)				

Table 5.11: Worst Results for Sun Images severely corrupted by Poison noise.
RMS contrasts: $O_o = 0.205623$, $I_a = 0.147227$, $I_d = 0.178575$

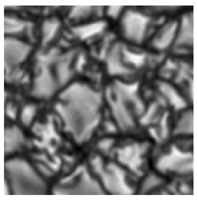
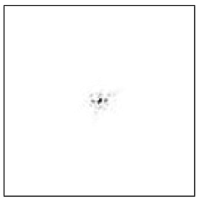
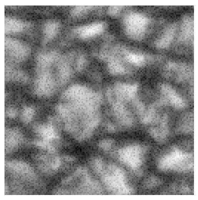
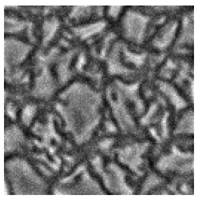
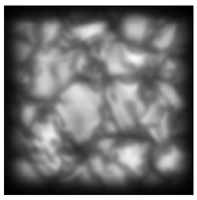
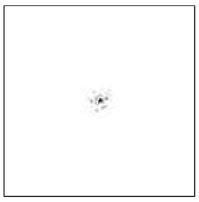
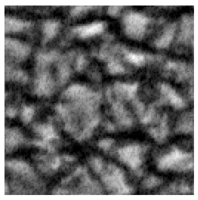
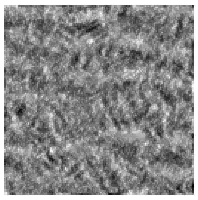
Original Object (O_o)	**Random PSF** (PSF_r)	**Artificial Image** (I_a)	**Difference** ($O_o - I_a$)
			
Estimated Image (I_e)	**Estimated PSF** (PSF_e)	**Deconvolved Image** (I_d)	**Difference** ($O_o - I_d$)
			

Table 5.12: Worst Results for Sun Images severely corrupted by Poison noise - PSFs Comparisons.
Configuration used: {[100 Artificial Sun Images, kept the worst and the worst], [CPSO: s=3, p=32, z=50, c=396 (38K evals.)], [Images: 128x128]}.

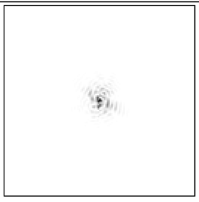
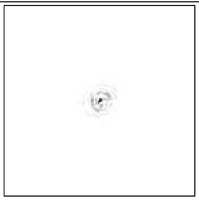
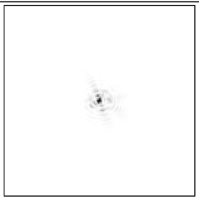
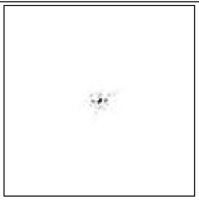
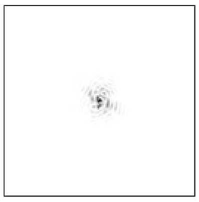
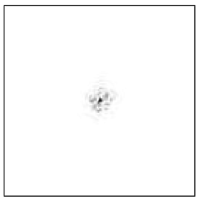
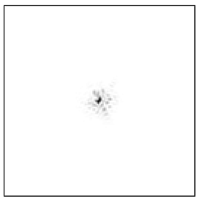
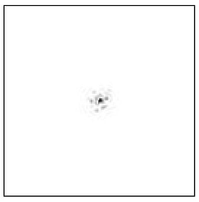
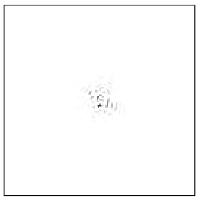
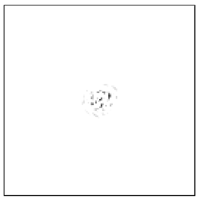
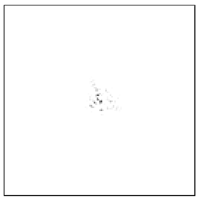
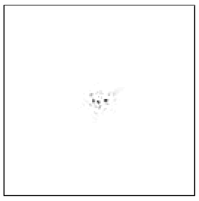
Rank	#1	#2	#3	#4
Random PSF (PSF_r)				
Estimated PSF (PSF_e)				
Difference ($PSF_r - PSF_e$)				

Table 5.13: Best Results for Lena Images.
RMS contrasts: $O_o = 0.192776$, $I_a = 0.164514$, $I_d = 0.172227$

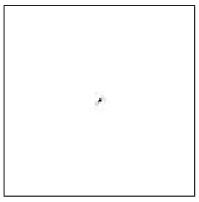
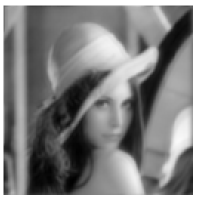
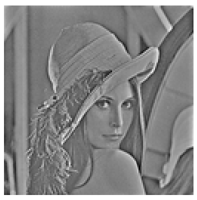
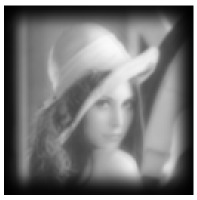
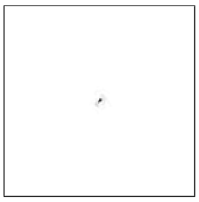
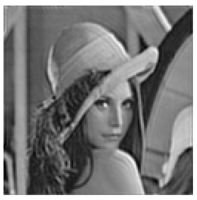
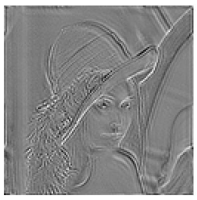
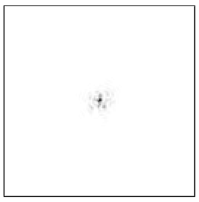
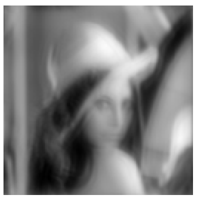
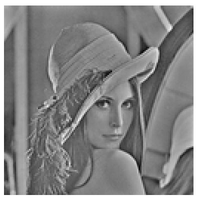
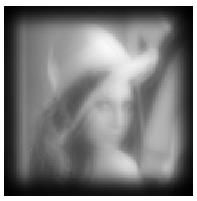
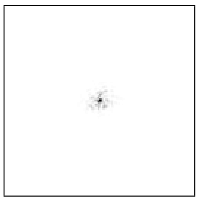
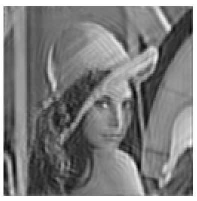
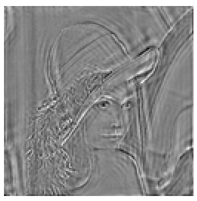
Original Object (O_o)	**Random PSF** (PSF_r)	**Artificial Image** (I_a)	**Difference** ($O_o - I_a$)
			
Estimated Image (I_e)	**Estimated PSF** (PSF_e)	**Deconvolved Image** (I_d)	**Difference** ($O_o - I_d$)
			

Table 5.14: Worst Results for Lena Images.
RMS contrasts: $O_o = 0.192776$, $I_a = 0.143206$, $I_d = 0.164518$
Configuration used: {[100 Artificial Lena Images, kept the best and the worst], [CPSO calibrated for Lena: s=3, p=32, z=50, c=512 (49K evals.)], [Images: 128x128]}.

Original Object (O_o)	**Random PSF** (PSF_r)	**Artificial Image** (I_a)	**Difference** ($O_o - I_a$)
			
Estimated Image (I_e)	**Estimated PSF** (PSF_e)	**Deconvolved Image** (I_d)	**Difference** ($O_o - I_d$)
			

5.3 Robustness and Scalability

To verify how robust is the solution as the number of Zernikes increases, 50 Solar images were processed for different number of zernikes ($nZernikes = 25, 50, 150, 250, 350, 500, 600$) then their times were measured, calculating the mean time for every $nZernikes$ configuration. Given the error function (fitness) is not comparable between different number of Zernikes, the fitness results were not compared on this experiment. Furthermore, the larger the Zernike terms the more it requires device memory, therefore the object and images size used for this Zernike experiment was smaller (64x64) than the size used during the previous experiments (128x128).

To validate if the calibrated solution behaves similarly for different image sizes, the CPSO parameters for one previously calibrated image of size 128x128 were used to process another similar image of size 64x64, then both convergence behaviors were measured and compared (64x64 vs. 128x128).

Both experiments above were executed based on the calibrated parameters set found during the section 5.1.

5.3.1 Dependence on the Number of Zernikes

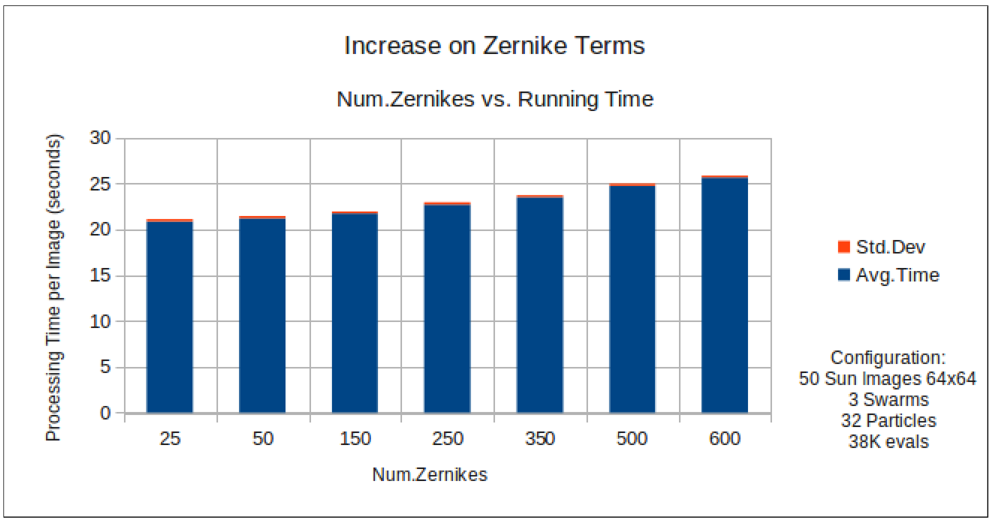


Figure 5.4: Scaling while Increasing the Zernike terms.

To evaluate how robust is the solution as the number of Zernike terms increases, 50 Solar images of size 64x64 were processed for several number of Zernikes ($nZernikes = 25, 50, 150, 250, 350, 500, 600$), then their averages and standard deviations of processing times were collected.

The similar standard deviations presented in Figure 5.4 shows a stable behavior among the number of Zernike terms used. We can see very small increases in the processing time occurring, having an increase of only 22.42% in the processing time from 25 to 600 Zernike terms (17% when increasing tenfold, from 50 to 500).

The 50 images were processed in the following times (in seconds):

Table 5.15: Scaling behavior for the number of Zernike terms.

Zernike Terms	**Per Image (seconds)**	**Total (seconds)**
25	20.87± 0.25	1045
50	21.22±0.24	1063
150	21.75±0.19	1090
250	22.70±0.25	1138
350	23.51±0.20	1180
500	24.74±0.24	1244
600	25.65±0.20	1291

Given that the processing time of the PSF estimation method developed in this work depends entirely on the number of Zernikes and on the image size, based on the Table 5.15 and on Figure 5.4 we can say that the number of Zernikes is not a critical factor while reducing the processing time.

5.3.2 Calibration Reuse for Different Image Sizes

To verify if the parameter values calibrated for one object can be reused for the same (or very similar) object of different size, a solar image of size 128x128 was firstly calibrated and then a similar image of smaller size (64x64) was processed against the same CPSO parameter set.

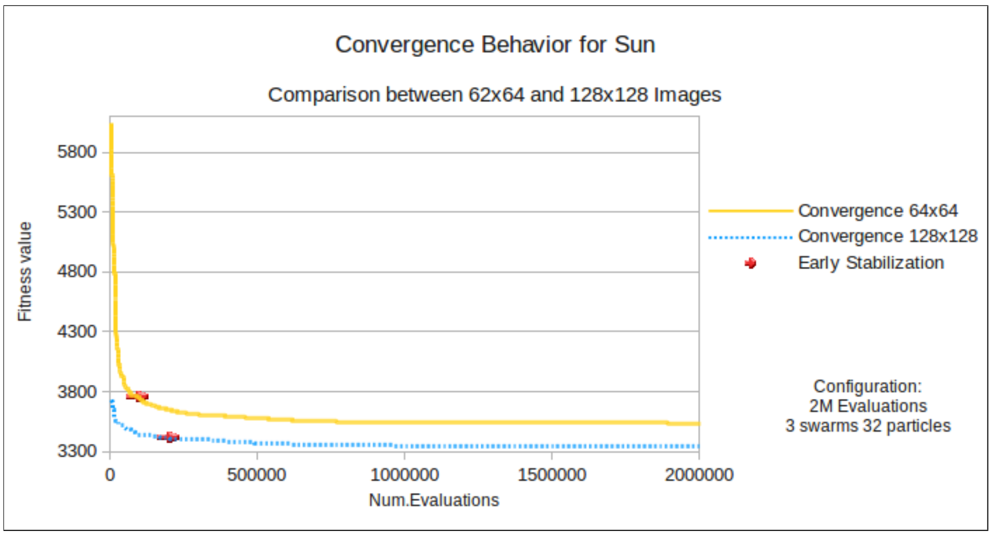


Figure 5.5: Convergence Behavior for Solar images of different sizes.
The figure shows the convergence behavior for 2 artificial Solar images. These images were generated artificially from similar Sun Objects of different sizes (64x64 and 128x128), then they were processed up to 2 million fitness evaluations and their convergences were compared.

The results shown in Figure 5.5 present similar convergence behavior for both image sizes, with the image 128x128 requiring additional evaluation cycles to converge, given its area is 4 times bigger than 64x64. This suggests that different image sizes requires a separated calibration, but only to adjust the parameter c(cycles), because both parameters s(swarms) and p(particles) from image 64x64 have benefited from the same convergence behavior as for the image 128x128.

5.4 Processing Times for PSF Estimations

The original SA work used as base for the development of this method [53] used a cluster of 23 servers Dell Poweredge 1950 (2 x Intel Xeon Processor 5160 per server) to process the images, obtaining 1 PSF result at every 108 seconds (100 PSFs on 3 hours) for Solar images of size 64x64.

Considering an observatory producing about 500-1000 daily Solar images, taken by a full spectro-polarimetric scan with a panoramic spectropolarimeter, the PSF estimation method with SA takes about 15-30 hours to post-process these images. Given the processing time on the original method is linearly dependent on the number of Zernikes, a tenfold increase on the dimensions (500 Zernikes) would take $\approx$300 hours for these same images, under the same SA cooling schedule.

For the same image size, the PSF Estimation method for CPSO developed on this work takes $\approx$21.22 seconds to obtain 1 PSF result (100 PSFs on $\approx$36 minutes) on one CPU core and one GPU device (see hardware details on introduction of chapter 5). For the daily 500-1000 solar images generated by the observatory, this CPSO method would estimate the corresponding PSFs in $\approx$ 2h56min - 5h53min (respectively) on the same machine (under the same configuration as the one described on section 5.2). By increasing the number of Zernikes tenfold (500 Zernikes), the processing time would be slightly increased to $\approx$ 3h51m - 6h52m for the same images.

Given only one machine with one GPU card was used on these experiments, this method has large room for improvements. Additional machines with multiple GPU cards could be used to estimate PSFs in parallel, having a central point of control submitting the requests and collecting the responses. Also, fine tuned kernels was not the focus of this work, therefore additional adjusts for coalesced memory access could be implemented to reduce the memory latency. Finally, as the PSF estimation method used as base for this work heavily relies on Fourier calculations, an OpenCL FFT kernel would need to be implemented to allow the calculation of multiple Fourier's at one single kernel call, what should radically reduce the overall processing time.

5.5 Considerations

When dealing with CPSO as the optimization method for high dimensional problems, it is well know how difficult it can be to find a good set of parameters that leads to good results quickly. Instead of having to use the trial and error approach, experimenting dozens of different configurations before deciding for the best known parameters (what not always means that stable results will be obtained among different executions), this chapter has presented a quick measurable method to choose for a good configuration set that ensures at least acceptable results on convenient processing times.

For validation purpose, multiple sets of different artificial solar images were processed by the CPSO calibrated for low-cost results. The experiment was run for different blurred images, blurred images with addition of noise and blurred images severely corrupted by noise, then both the bests and the worsts results where collected for every different degree of corruption. The results has shown that very approximate PSFs were obtained in all cases, with good image recovering even on the most severe distortions. Non solar images with high frequencies were also validated, where the results obtained were as good as the results with the low frequency solar images.

The method was experimented for an increasing number of Zernikes and the results show that the proposed algorithm is only slightly affected by the number of Zernikes used, what facilitates the implementation of a simulated atmosphere while estimating the PSFs. Also, a final experiment has suggested that similar images of different sizes can benefit from the same CPSO parameter set by only adjusting the number of iterations.

In general, the method presented in this work has shown quick good results, strong ability of dealing with noises and robust behavior for both low-frequency and high-frequency images (not only for solar images). Considering that a single machine and only one OpenCL device was used to run these experiments, much better results can be obtained by increasing the number of iterations and combining multiple OpenCL devices from different machines, besides the possibility of using more than one OpenCL device per machine. Given the complexity of this real life application, many others improvements can be implemented on this work, like adjusting the kernels for coalesced memory access and creating a new kernel for parallel FFT calculation on OpenCL.

CHAPTER 6

CONCLUSIONS

Considering the large number of images generated by an observatory, taken daily from objects standing outside the atmosphere, the reduction of the time required for the post processing of these images is critical for a good work performance.

The work presented here has proposed the creation of a parallel PSF Estimation solution to run on OpenCL enabled devices, so that good results could be obtained on feasible times under cheap hardwares.

The Simulated Annealing has shown good PSF estimations, but in turn it required unfeasible processing times to obtain such images, besides presenting a poor convergence behavior, proving to be a weak method for finding good PSF parameters quickly.

Many good optimization methods are available but not all of them can be easily implemented on GPU devices. Given the PSO is a simple optimization method that can be easily implemented on GPU cards, with a strong robustness for high dimensional problems, its Cooperative version, CPSO, was used as the optimization method for this work. The Cooperative PSO, calibrated for low-cost good results, not only has shown competitive results but also obtained them more quickly and with more stable behavior.

The CPSO applied for high dimensional problems can be of very hard configuration. Since there is no standard measurable process for its configuration, its parameters rely entirely on the user experience, what frequently leads to poor optimization results. Instead of having to use the trial and error approach, this work proposed a quick measurable method to choose good parameter values to specifically reduce the processing time and ensure at least acceptable results under convenient time frames. This proposed calibration process for CPSO has shown strong robustness, obtaining approximate PSFs even under moderate and high noise presence. Both low-frequency and high-frequency images obtained similar good results, meaning that different types of images (not only solar images) can benefit from this method. Similar images of different sizes can benefit from the same calibrated CPSO parameter set, requiring only the adjust on the number of cycles to conform with each image size.

Its demonstrated good scaling for large numbers of Zernike terms has shown that this work can be easily adapted for atmosphere simulation algorithms, since it is not heavily affected by a high number of dimensions.

Given the good convergence behavior presented by the CPSO, even better results can be obtained by this solution by just increasing the number of cycles, what means that it depends only on what is an acceptable amount of time to wait for better results.

As application, this work brings the following contributions:

- Reduction of the time required to obtain good PSF estimations.
- Parallel PSF estimations on machines containing multiple GPU cards.
- High number of Zernikes supported.
- PSF estimations also possible for non-solar images.

From the computational standpoint, these are the contributions:

- A PSF estimation method developed to run on OpenCL devices.
- A Cooperative PSO developed for OpenCL devices, with mechanisms to increase its aggressiveness while controlling its convergence.
- A measurable calibration procedure for CPSO to optimize complex and high dimensional problems, ensuring at least acceptable results while specifically reducing the general processing time.

Given the inherent complexity of this kind of real life application, many improvements can still be implemented on this work, like for example:

- The creation of a measurable process to also calibrate the parameters that were heuristically fixed on this work $(w, c_1, c_2, resetFactor, nZernikes)$.
- The implementation of a distributed solution (eg., a master-slave solution) for submitting multiple parallel estimation on multiple machines with multiple OpenCL devices.
- The fine tuning of the OpenCL code implemented on this work, to avoid any uncoalesced memory accesses and reduce the memory latency.
- The implementation of a FFT kernel on OpenCL language to calculate multiple Fourier's in parallel at a single kernel call.

In summary, the proposed PSF estimation method with CPSO optimization proved that the costs associated with the post-processing time for the image correction process can be reduced in many ways (like reducing hardware costs, energy usage and human waiting time). This work has shown that the CPSO can obtain good PSFs in short time, requiring only a low-cost easy-acquisition GPU hardware and a calibration procedure.

CHAPTER 7

APPENDIX

7.1 Kernel Codes

The Xorshift is implemented by Listing 7.1 (on OpenCL language) and uses a random seed on GPU side to generate the random numbers directly on device.

Listing 7.1: Xorshift Implementation for OpenCL

```
uint xorshift(uint4 *seed) {
    uint t = seed->x ^ (seed->x << 11);
    *seed = seed->yzww;
    seed->w = seed->w ^ (seed->w >> 19) ^ (t ^ (t >> 8));
    return seed->w;
}
```

The Listing 7.2 shows a kernel that calculates the Phase (line 25) and Pupil (lines 31 and 32), where every work item iterates through the phase column (line 19) and every work group calculates one entire Phase. Notice that this listing represents the OpenCL code of the line 3 from the Algorithm 7. This is a code snippet and some lines were omitted for simplicity reasons.

Listing 7.2: Snippet - Calculate Phase and Pupil

```
 1  /**
 2   * Input: g_coefs, g_zernikes, n_zernikes, phase_height.
 3   * Output: g_pupil.
 4   */
 5  // Every work item calculates a column.
 6  int lid = get_local_id(0);
 7  // Every work group calculates a PSF.
 8  int gid = get_group_id(0);
 9
10  //Bring this PSF's Zernike Coefficients to local memory.
11  for(i=lid; i < n_zernikes; i+=get_local_size(0)){
12      l_coefs[i] = g_coefs[gid * n_zernikes + i];
13  }
14
15  float p_phase;
16  float2 p_pupil;
17  int z_pos, phase_area = phase_height * phase_height;
18  // For every column.
19  for(i=0; i < phase_height; i++){
20      p_phase = 0;
21      z_pos = i * phase_height + lid;
22      // Iterate through all Zernikes.
23      for (int j=0; j < n_zernikes; j++) {
        // Calculate the phase.
        p_phase += l_coefs[j] *
                   g_zernikes[j * phase_area + z_pos];
    }

    // Calculate the pupil on private memory:
    //    cexp(p * I) = cos(p) + i * sin(p)
    p_pupil.x = cos(phase);
    p_pupil.y = sin(phase);
    // Move PSF's pupil to global memory.
    g_pupil[gid * phase_area + z_pos] = p_pupil;
}
```

The listing 7.3 shows how the PSF is calculated from the Focus (representing the OpenCL code of the line 12 from the Algorithm 7). It reads a complex number from the global memory (Listing 7.3, line 8), converts it to real and then calculates its power, returning the result directly to the global memory (line 13). Every work item on this kernel calculates a single point in the PSF.

Listing 7.3: Snippet - Calculate PSF from Focus

```
 1 /**
 2  * Input:  g_focus.
 3  * Output: g_psf.
 4  */
 5 // Every work item calculates a single point.
 6 int idx = get_global_id(0);
 7 // Bring the complex value to private memory.
 8 float2 p_focus = g_focus[idx];
 9 // Convert from complex to real.
10 float real = sqrt((p_focus.x * p_focus.x) +
11                   (p_focus.y * p_focus.y));
12 // Calculate the power and return to global memory.
13 g_psf[idx] = real * real;
```

The convolution is calculated by multiplying the FFT of the extracted scaled PSF with the FFT of the original Object (Listing 7.4, line 21). Every work item in this kernel iterates through a column while the work group calculates an entire convolution. Notice that the Convolved Object calculated by this kernel is in fact the FFT of the final Estimated Imaged, calculated through the Estimated PSF, which in turn was generated from the x vector of a CPSO sub-particle.

Listing 7.4: Snippet - Convolution

```
 1 /**
 2  * Input:  g_obj_fft, g_psf_fft, img_height.
 3  * Output: g_conobj.
 4  */
 5 // Every work item calculates a column.
 6 int lid = get_local_id(0);
 7 // Every work group calculates a PSF.
 8 int gid = get_group_id(0);
```

```

int obj_pos, psf_pos;
float2 p_psf, p_obj, p_conv;
// For every column.
for(int i=0; i < img_height; i++){
    obj_pos = i * img_height + lid;
    psf_pos = gid * img_area + obj_pos;
    // Bring the PSF's FFT to the private memory.
    p_psf = g_psf_fft[psf_pos];
    // Bring the Object's FFT to the private memory.
    p_obj = g_obj_fft[obj_pos];
    // Do the convolution.
    p_conv = {
        (p_psf.x * p_obj.x) - (p_psf.y * p_obj.y),
        (p_psf.x * p_obj.y) + (p_psf.y * p_obj.x)
    };
    // Sends the result to global memory.
    g_conobj[psf_pos] = p_conv;
}
```

The cost is calculated as the difference between the original Image's FFT and the Convolved object, as shown in the Listing 7.5. The result is converted to real, scaled and then filtered by the Diffraction mask. Every work group of this kernel calculates the cost of an entire convolved image, meaning that a single kernel call will calculate all costs related to all CPSO particles.

Listing 7.5: Snippet - Cost Calculation

```
/**
 * Input: g_conobj, g_img_fft, g_diffraction_mask,
 *        img_height.
 * Output: g_cost.
 */
// Every work item calculates a column.
int lid = get_local_id(0);
// Every work group calculates a PSF.
int gid = get_group_id(0);

int img_pos, obj_pos;
float x, y;
float2 p_img, p_obj;
// For every column.
for(int row=0; row < img_height; row++){
    img_pos = row * img_height + lid;
    obj_pos = gid * img_area + img_pos;
    // Bring the Image's FFT to the private memory.
    p_img = g_img_fft[img_pos];
    // Bring the Convolved Object to the private memory.
    p_obj = g_conobj[obj_pos];
    // Calculate the cost.
    x = (p_img.x - p_obj.x);
    y = (p_img.y - p_obj.y);
    // Calculate a real scaled cost
    // under the diffraction mask,
    // then sends the result to global memory.
    g_cost[obj_pos] = (g_diffraction_mask[img_pos]) *
                        (sqrt(x * x + y * y) / img_area);
}
```

BIBLIOGRAPHY

[1] Fastest fourier transform in the west. http://www.fftw.org/. Accessed in: 01/16/2013.

[2] Khronos opencl api registry. http://www.khronos.org/registry/cl/. Accessed in: 10/29/2012.

[3] Nvidia cuda fast fourier transform. http://developer.nvidia.com/cufft/. Accessed in: 01/19/2013.

[4] Open multi-processing. http://www.openmp.org/. Accessed in: 01/19/2013.

[5] Opencl - the open standard for parallel programming of heterogeneous systems. http://www.khronos.org/opencl/. Accessed in: 10/28/2012.

[6] Viennacl. http://viennacl.sourceforge.net/. Accessed in: 12/16/2012.

[7] Enrique Alba, Gabriel Luque, e Sergio Nesmachnow. Parallel metaheuristics: recent advances and new trends. *International Transactions in Operational Research*, 20(1):1–48, 2013.

[8] G. R. Ayers, M. J. Northcott, e J. C. Dainty. Knox-thompson and triple-correlation imaging through atmospheric turbulence. *Journal of the Optical Society of America A*, 5:963–985, julho de 1988.

[9] Cja Bastos-Filho, Mac Oliveira, Dno Nascimento, e Ad Ramos. *Impact of the Random Number generator quality on particle swarm optimization algorithm running on graphic processor units*, páginas 85–90. IEEE, 2010.

[10] Kamel Bensebaa, Gerald Jean Francis Banon, Leila Maria Garcia Fonseca, José Carlos Neves Epiphanio, e Leila Maria Garcia Fonseca. On-orbit spatial resolution estimation of cbers-1 ccd imaging system using higher resolution images. *Simpósio Brasileiro de Sensoriamento Remoto (SBSR)*, 12(2005):827–834, 2005.

[11] F Van Den Bergh. *An Analysis of Particle Swarm Optimizers.* Tese de Doutorado, PhD thesis, Faculty of Natural and Agricultural Science, University of Pretoria, 2001.

[12] D Besozzi, P Cazzaniga, G Mauri, D Pescini, e L Vanneschi. A comparison of genetic algorithms and particle swarm optimization for parameter estimation in stochastic biochemical systems. *Evolutionary Computation, Machine Learning and Data Mining in Bioinformatics: 7th European Conference, EvoBIO 2009 Tübingen, Germany, April 15-17, 2009 Proceedings*, volume 5483, páginas 116. Springer, 2009.

[13] Christian Blum e Andrea Roli. Metaheuristics in combinatorial optimization: Overview and conceptual comparison. *ACM Computing Surveys (CSUR)*, 35(3):268–308, 2003.

[14] Daniel Bratton e James Kennedy. Defining a standard for particle swarm optimization. *Swarm Intelligence Symposium, 2007. SIS 2007. IEEE*, páginas 120–127. IEEE, 2007.

[15] Johannes Brauers, Claude Seiler, e Til Aach. Direct psf estimation using a random noise target. *IS&T/SPIE Electronic Imaging: Digital Photography VI, p. to appear. SPIE-IST*, 7537, 2010.

[16] J. W. Brault e O. R. White. The analysis and restoration of astronomical data via the fast fourier transform. *Astronomy and Astrophysics*, 13:169, julho de 1971.

[17] José M. Cecilia, José M. García, Andy Nisbet, Martyn Amos, e Manuel Ujaldón. Enhancing data parallelism for ant colony optimization on gpus. *Journal of Parallel and Distributed Computing*, 73(1):42–51, 2013.

[18] J. C. Christou, S. M. Jefferies, e E. K. Hege. Object-independent point spread function and wavefront phase estimation. R. K. Tyson & R. Q. Fugate, editor, *Society of Photo-Optical Instrumentation Engineers (SPIE) Conference Series*, volume 3762 of *Society of Photo-Optical Instrumentation Engineers (SPIE) Conference Series*, páginas 201–210, setembro de 1999.

[19] Audrey Delévacq, Pierre Delisle, Marc Gravel, e Michaël Krajecki. Parallel ant colony optimization on graphics processing units. *Journal of Parallel and Distributed Computing*, 73(1):52–61, 2013.

[20] Vital C Ferreira e Nelson DA Mascarenhas. Analysis of the robustness of iterative restoration methods with respect to variations of the point spread function. *Image Processing, 2000. Proceedings. 2000 International Conference on*, volume 3, páginas 789–792. IEEE, 2000.

[21] Fred Glover e Manuel Laguna. *Tabu Search*, volume 1. Kluwer Academic Pub, 1998.

[22] David E Goldberg e John H Holland. Genetic algorithms and machine learning. *Machine Learning*, 3(2):95–99, 1988.

[23] Er.Neha Gulati e Er.Ajay Kaushik. Remote sensing image restoration using various techniques: A review. *International Journal of Scientific & Engineering Research*, 3(1), 2012.

[24] Jie Guo e Sheng jing Tang. An improved particle swarm optimization with reinitialization mechanism. *Intelligent Human-Machine Systems and Cybernetics, International Conference on*, 1:437–441, 2009.

[25] F.J. Harris. On the use of windows for harmonic analysis with the discrete fourier transform. *Proceedings of the IEEE*, 66(1):51–83, janeiro de 1978.

[26] Wei Hongkai, Wang Pingbo, Cai Zhiming, Chen Baozhu, e Yao Wanjun. Application of particle swarm optimization method in fractional fourier transform. *Image Analysis and Signal Processing (IASP), 2010 International Conference on*, páginas 442–445. IEEE, 2010.

[27] James Kennedy. Swarm intelligence. Albert Zomaya, editor, *Handbook of Nature-Inspired and Innovative Computing*, páginas 187–219. Springer US, 2006.

[28] M. Kircher e P. Jain. Pooling pattern. 2002.

[29] S. Kirkpatrick, C. D. Gelatt, e M. P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, 1983.

[30] K. T. Knox e B. J. Thompson. Recovery of images from atmospherically degraded short-exposure photographs. *The astrophysical journal*, 193:L45–L48, outubro de 1974.

[31] A. Labeyrie. Attainment of diffraction limited resolution in large telescopes by fourier analysing speckle patterns in star images. *Astron. Astrophys*, 6(1):85, maio de 1970.

[32] M. G. Löfdahl e G. B. Scharmer. Phase-diversity restoration of solar images. volume 13, páginas 89–104, dezembro de 1993.

[33] A. W. Lohmann, G. Weigelt, e B. Wirnitzer. Speckle masking in astronomy - triple correlation theory and applications. 22:4028–4037, dezembro de 1983.

[34] Sean Luke. Essentials of metaheuristics. *Lecture notes, George Mason University*, 2009. `http://cs.gmu.edu/sean/book/metaheuristics`. Accessed in: 03/10/2013.

[35] J. Marino, T. R. Rimmele, e J. C. Christou. Long-exposure point spread function estimation from adaptive optics loop data. & R. Ragazzoni D. Bonaccini Calia, B. L. Ellerbroek, editor, *Society of Photo-Optical Instrumentation Engineers (SPIE) Conference Series*, volume 5490 of *Society of Photo-Optical Instrumentation Engineers (SPIE) Conference Series*, páginas 184–194, outubro de 2004.

[36] K. Mikurda e O. von der Lühe. High resolution solar speckle imaging with the extended knox-thompson algorithm. *Solar Physics*, 235(1):31–53, maio de 2006.

[37] S.K. Nayar e M. Ben-Ezra. Motion-based motion deblurring. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 26(6):689–698, 2004.

[38] Robert Nowotniak e Jacek Kucharski. Gpu-based massively parallel implementation of metaheuristic algorithms. *Automatyka*, 15(3):595–611, 2011.

[39] Eli Peli. Contrast in complex images. *Journal of the Optical Society of America A: Optics, Image Science, and Vision, Issue 10*, 7:2032–2040, outubro de 1990.

[40] M Pourmahmood, AM Shotorbani, RM Shotorbani, e O Ismail. Estimation of image corruption inverse function and image restoration using a pso-based algorithm. *International Journal of Video & Image Processing and Network Security IJVIPNSIJENS*, 10(06), 2010.

[41] M. Rabinovich, P. Kainga, D. Johnson, B. Shafer, J.J. Lee, e R. Eberhart. Particle swarm optimization on a gpu. *Electro/Information Technology (EIT), 2012 IEEE International Conference on*, páginas 1–6. IEEE, 2012.

[42] Ramesh Raskar, Amit Agrawal, e Jack Tumblin. Coded exposure photography: motion deblurring using fluttered shutter. *ACM Transactions on Graphics*, 25(3):795, 2006.

[43] Colin Reeves. Genetic algorithms. *Handbook of metaheuristics*, páginas 55–82, 2003.

[44] Filip Rooms, Wilfried R Philips, e Javier Portilla. Parametric psf estimation via sparseness maximization in the wavelet domain. *Society of Photo-Optical Instrumentation Engineers (SPIE) Conference Series*, volume 5607, páginas 26–33. International Society for Optics and Photonics, 2004.

[45] D. Sheppard, B.R. Hunt, e M.W. Marcellin. Super-resolution of imagery acquired through turbulent atmosphere. *Signals, Systems and Computers, 1996. 1996 Conference Record of the Thirtieth Asilomar Conference on*, volume 1, páginas 81–85, novembro de 1996.

[46] Daniel Leal Souza, Glauber Duarte Monteiro, Tiago Carvalho Martins, Victor Alexandrovich Dmitriev, e Otávio Noura Teixeira. Pso-gpu: accelerating particle swarm optimization in cuda-based graphics processing units. *Proceedings of the 13th annual conference companion on Genetic and evolutionary computation*, GECCO '11, páginas 837–838. ACM, 2011.

[47] Tsung-Ying Sun, Sin-Jhe Ciou, Chan-Cheng Liu, e Chih-Li Huo. Out-of-focus blur estimation for blind image deconvolution: using particle swarm optimization. *Proceedings of the 2009 IEEE international conference on Systems, Man and Cybernetics*, SMC '09, páginas 1627–1632. IEEE Press, 2009.

[48] Dr P.J. Tadrous. Deconvolution software. `http://www.deconvolve.net/`. Accessed in: 12/12/2012.

[49] Nanci Tripoli. The zernike polynomials. `http://www.optikon.com/en/articles/keratron_023/media/TheAberrometers_2003_Tripoli%20(Zernike%20Polynomials).pdf`, 2003. Accessed in: 02/13/2012.

[50] F Van den Bergh e A P Engelbrecht. A cooperative approach to particle swarm optimization. *IEEE Transactions on Evolutionary Computation*, 8(3):225–239, 2004.

[51] Michiel Van Noort, Luc Der Voort, e Mats Löfdahl. Solar image restoration by use of multi-frame blind de-convolution with multiple objects and phase diversity. *Solar Physics*, 228:191–215, 2005.

[52] J P Véran, F Rigaut, H Maître, e D Rouan. Estimation of the adaptive optics long–exposure point–spread function using control loop data. *Journal of the Optical Society of America A: Optic*, 14(11):3057–3069, 1997.

[53] T A Waldmann e O Lühe. Point spread function estimation using speckle reconstructions of solar surface images. *Solar Physics*, 267(1):217–231, 2010.

[54] S J Weddell e R Y Webb. The restoration of extended astronomical images using the spatially-variant point spread function. *2008 23rd International Conference Image and Vision Computing New Zealand*, páginas 1–6, 2008.

[55] G. P. Weigelt. Modified astronomical speckle interferometry 'speckle masking'. *Optics Communications*, 21:55–59, abril de 1977.

[56] K. Winner, D. Miner, e M. desJardins. Controlling particle swarm optimization with learned parameters. *Self-Adaptive and Self-Organizing Systems, 2009. SASO '09. Third IEEE International Conference on*, páginas 288–290. IEEE, 2009.

[57] F. Wöger. *High-resolution observations of the solar photosphere and chromosphere.* Tese de Doutorado, Tesis, Kiepenheuer-Institut für Sonnenphysik Albert-Ludwigs-University, Freiburg, Germany, março de 2007.

[58] F. Wöger e O. von der Lühe. Kisip: a software package for speckle interferometry of adaptive optics corrected solar data. *Society of Photo-Optical Instrumentation Engineers (SPIE) Conference Series*, volume 7019 of *Society of Photo-Optical Instrumentation Engineers (SPIE) Conference Series*, páginas 70191E–70191E, agosto de 2008.

[59] James C. Wyant e Katherine Creath. Applied optics and optical engineering, vol. xl. `http://www.optics.arizona.edu/jcwyant/zernikes/Zernikes.pdf`, 1992. Accessed in: 02/13/2012.

[60] Lihong Yang e Jianyue Ren. Remote sensing image restoration using estimated point spread function. *Information Networking and Automation (ICINA), 2010 International Conference on*, volume 1, páginas V1–48–V1–52, outubro de 2010.

[61] Yinxue Zhang, Zhenhong Jia, Haijun Jiang, e Zijian Liu. Image restoration based on robust error function and particle swarm optimization-bp neural network. *2008 Fourth International Conference on Natural Computation*, 7:640–644, 2008.

[62] You Zhou e Ying Tan. Gpu-based parallel particle swarm optimization. *Evolutionary Computation, 2009. CEC '09. IEEE Congress on*, páginas 1493–1500. IEEE, 2009.