**MSc in Photonics**

PHOTONICS**BCN**

Universitat Politècnica de Catalunya (UPC)
Universitat Autònoma de Barcelona (UAB)
Universitat de Barcelona (UB)
Institut de Ciències Fotòniques (ICFO)

http://www.photonicsbcn.eu

*Master in Photonics*

## MASTER THESIS WORK

# Just in time calculations for partially coherent imaging

**Marcos Pérez Aviñoa**

**Supervised by Dr. Artur Carnicer, UB**

Presented on date 23rd July 2019

Registered at

Escola Tècnica Superior d'Enginyeria de Telecomunicació de Barcelona

# Just in time calculations for partially coherent imaging

**Marcos Pérez Aviñoa**

Facultat de Física, Departament de Física Aplicada, Universitat de Barcelona (UB), Martí i Franqués 1, 08028, Spain

E-mail: mperezav7@alumnes.ub.edu

**Abstract.**

Algorithms to calculate partially coherent image formation are centered around the classic formula developed by Hopkins. Algebraic methods have been developed to make feasible the integration process, although with strong limitations in the complex degrees of coherence used. In this work we revisit the formula laid by Hopkins and implement it to take full advantage of the parallelization given by Graphics Processing Units. Execution times are compared and the simulations compared with experimental results.

## 1. Introduction

The most comprehensive description for the propagation of classical electromagnetic waves is the theory of partial coherence [1, 2, 3]. The electromagnetic field is treated as a stochastic process and the fundamental quantities of the theory become the correlation functions or coherence functions. Unsurprisingly, since the very beginning of the diffraction theory of optical imaging partially coherent optical fields have been thoroghly discussed [1, 4] . Specifically, Hopkins [1] gave a method to calculate the intensity in the image plane of an optical system as a sum of Fourier transforms, one for each point in the plane. This method was computationally expensive and not fit to perform simulations for real applications. Moreover, the two extreme cases of coherence, complete coherence and complete incoherence, were easily described by the theory of linear systems and analogous in their application [3, 5, 6]. Their results were good enough for most applications and partially coherent imaging was left as a curiosity.

However, with the aparition of optical lithography the approaches of complete coherence or incoherence were not able to reproduce most of the results observed. The coherence of the field was also found to be a key quantity in the resolution and shape of the final image intensity, which determined the fidelity of the circuit to be printed [7]. Due to this, Hopkins method has been revisited several times and brought a new algebraic description of it, which reduced the number of Fourier Transforms needed. The method [8, 9, 10] has been rediscovered several times, with different theoretical backgrounds but being the same method after all.

Nowadays, computers have increased exponentially in computing power and parallelization has emerged as the new norm of computation. Specifically, Graphics Processing Units (GPUs) have brough parallelization to a level such that simple operations on a whole arrays may be performed in constant time [11]. This has profoundly changed the paradigm of scientific computing, as many of the state of the art problems can be improved by parallelization [12, 13].

Despite this, to the best of our knowledge there has not been improvements of the partially coherent image formation algorithms in terms of parallelization. Therefore, in this work we revisit the direct Hopkins method under the new perspective of parallel computing with GPUs. First, we introduce the basic theory behind the problem of partially coherent image formation. Then, we develop in detail the direct Hopkins algorithm implemented both in CPU and GPU, in such a way as to occupy the least amount of memory possible. The algebraic Hopkins method is also described and implemented. Simulation results are presented with a comparison with some experimental results. Finally, we end with some concluding remarks.

## 2. Statement of the problem

In the most general case, an electromagnetic field is a stochastic process. Up to second order, it can be characterized by a correlation function, named the mutual coherence function [2]. In the scalar approximation of the electric field it is written as

$$\Gamma(\vec{r}_1, \vec{r}_2, t_1, t_2) = \langle E^\star(\vec{r}_1, t_1) E(\vec{r}_2, t_2) \rangle \tag{1}$$

Assuming that the field is stationary at least in the wide sense, $\Gamma(\cdot)$ depends only on the difference of the two points of time:

$$\Gamma(\vec{r}_1, \vec{r}_2, \tau) = \langle E^\star(\vec{r}_1, t) E(\vec{r}_2, t+\tau) \rangle \tag{2}$$

where $\tau = t_2 - t_1$ is the time difference between between the two realizations of the electric field. Further simplifications can be made under the assumption of quasimonochromatic fields. These fields contain a narrow range of frequencies centered around a given frequency $\nu_0$, e.g. a field generated by a laser. Assuming that the electric field can be written as a Fourier Transform pair,

$$E(\vec{r}, t) = \int_{-\infty}^{+\infty} U(\vec{r}, \nu) \exp(-2\pi i \nu t) d\nu \tag{3}$$

$$= A(\vec{r}, t) \exp(-2\pi i \nu_0 t) \tag{4}$$

where now $A(\vec{r}, t)$ is a slowly varying function in time. Further assuming that the region of observation of the intensity is small, the time difference $\tau$ will also be a small quantity. Therefore we may write

$$\begin{aligned} \Gamma(\vec{r}_1, \vec{r}_2, \tau) &= \exp(-2\pi i \nu_0 \tau) \langle A^\star(\vec{r}_1, t) A(\vec{r}_2, t+\tau) \rangle \approx \\ &\approx \exp(-2\pi i \nu_0 \tau) \langle A^\star(\vec{r}_1, t) A(\vec{r}_2, t) \rangle = \exp(-2\pi i \nu_0 \tau) J(\vec{r}_1, \vec{r}_2) \end{aligned} \tag{5}$$

Where $t$ is constant and $J(\vec{r}_1, \vec{r}_2)$is called the mutual intensity of the field. The complex exponential has not been replaced by unity as $\nu_0$ is a big quantity and small changes of $\tau$ produce appreciable changes in the exponential. However, in our approximation this time difference will not play a significant role and we may work just with the mutual intensity.

We define the process of image formation as the caltulation of the image intensity given the mutual coherence at the object plane of the system. Under these conditions, the process is described by a four dimensional convolution [3]

$$I(\vec{r}) = \iint_{-\infty}^{+\infty} J(\vec{r_1}, \vec{r_2}) h^\star(\vec{r} - \vec{r}_1) h(\vec{r} - \vec{r}_2) d^2 r_1 d^2 r_2 \tag{6}$$

where $h(\vec{r})$ is the point spread function (PSF), which describes the imaging of a single point through the optical system. One would be tempted to perform this integral with the convolution

theorem of the Fourier Transforms. However, memory constraints make this problem impossible even for moderate image sizes. For the mutual intensity in this form, it is required to define a four dimensional array. In the most general case, this array is composed of complex numbers. Assuming single precision complex numbers, of 64 bits each, an image of 256x256 pixels would need 32 GB of RAM just for the allocation of the array. Further simplifications are needed to deal with a reasonable problem. Assuming now that the mutual intensity is separable in the following form

$$J(\vec{r}_1, \vec{r}_2) = u^\star(\vec{r}_1)u(\vec{r}_2)j(\vec{r}_2 - \vec{r}_1) \tag{7}$$

where $u(\vec{r})$ contains the complex transmittance of the object and the amplitude of the illumination, while $j(\vec{r}_2 - \vec{r}_1)$ is the so-called complex degree of coherence and represents a measure of the coherence of the field at two points separated a given distance. With this, the four dimensional convolution can be expressed as

$$\begin{aligned} I(\vec{r}) &= \iint_{-\infty}^{+\infty} u^\star(\vec{r}_1)u(\vec{r}_2)j(\vec{r}_2 - \vec{r}_1)h^\star(\vec{r} - \vec{r}_1)h(\vec{r} - \vec{r}_2)d^2r_1 d^2r_2 \\ &= \iint_{-\infty}^{+\infty} j(-\vec{\rho})u^\star(\vec{\rho} - \vec{r}_2)u(\vec{r_2})h^\star(\vec{r} - \vec{\rho} - \vec{r}_2)h(\vec{r} - \vec{r}_2)d^2r_2 d^2\rho \end{aligned} \tag{8}$$

where $\vec{\rho} = \vec{r}_1 - \vec{r}_2$. Taking the observation point $\vec{r}$ as constant, the rightmost integral is recognized as a convolution,

$$I(\vec{r}) = \int_{-\infty}^{+\infty} j(-\vec{\rho})d^2\rho \int_{-\infty}^{+\infty} u^\star(\vec{\rho} + \vec{r}_2)u(\vec{r_2})h^\star(-\vec{\rho} - \vec{r}_1; \vec{r})h(-\vec{r}_2; \vec{r})d^2r_2 \tag{9}$$

Now, the theorem involving Fourier Transforms [14]

$$\int_{-\infty}^{+\infty} \hat{f}(u)\hat{g}(u)\hat{h}(u)du = \frac{1}{\sqrt{2\pi}} \iint_{-\infty}^{+\infty} f(-x_1 - x_2)g(x_1)h(x_2)dx_1dx_2 \tag{10}$$

where the hats represent Fourier Transofrms, can be modified trivially to two dimensions,

$$\iint_{-\infty}^{+\infty} \hat{f}(u_1, u_2)\hat{g}(u_1, u_2)\hat{h}(u_1, u_2)du = \tag{11}$$

$$\frac{1}{2\pi} \iiiint_{-\infty}^{+\infty} f(-x_1 - x_2, -y_1 - y_2)g(x_1, y_1)h(x_2, y_2)dx_1dx_2dy_1dy_2 \tag{12}$$

Recognizing from (12) $f(u_1, u_2) \rightarrow j(\vec{\rho})$, $g(u_1, u_2) \rightarrow u^\star(\vec{\rho} + \vec{r}_2)h^\star(-\vec{r_1} - \vec{\rho}; \vec{r})$ $h(u_1, u_2) \rightarrow u(\vec{r}_2)h^\star(\vec{r}_2)$ this convolution can be furhter simplified to

$$\begin{aligned} I(\vec{r}) &= 2\pi \int_{-\infty}^{+\infty} \mathcal{F}[j(\vec{\rho})](\vec{q})|\mathcal{F}[u(\vec{r}_2)h(-\vec{r}_2\vec{r}))](\vec{q})|^2 d^2q \\ &= 2\pi \int_{-\infty}^{+\infty} \hat{j}(\vec{q})|\phi(\vec{q}; \vec{r})|^2 d^2q \end{aligned} \tag{13}$$

where $\mathcal{F}[\cdot]$ denotes a Fourier Transform and $\hat{j}(\cdot), \phi(\cdot)$ represent the Fourier Transforms in the preceding brackets. The variable $\vec{q}$ is the spatial frequency. This form is much simpler than (6), as all the four dimensional operations have been reduced to two dimensions. We have also avoided the need to define a four dimensional array completely, relying on the decomposition of the mutual intensity.

## 3. Numerical implementation

To study the numerical implementation of the Hopkins algorithms, each quantity in (13) is discretized. The PSF, the complex transmittance of the object and the complex degree of coherence are represented by arrays. If the complex transmittance is sampled into an array of $N_x \times N_y$ points, the complex degree of coherence must also have the same number of points and sampling, as $j(\vec{u})$ depends on the coordinate difference at the object plane. Idem with the PSF, expressed in the coordinates of the object coordinates. In this way, the integral (13) is written in the more useful way

$$I[i,j] = \sum_{k=0}^{N_x} \sum_{l=0}^{N_y} \hat{j}[k,l] |\phi_{ij}[k,l]|^2 \tag{14}$$

In this section three implementations of this equations are studied. First, we begin with a direct CPU implementation, using parallelism with the multiple cores providede by modern CPUs. Then, the integral is performed using the massive parallelization provided by GPUs. Finally, we implement an algebraic method in CPU, which optimizes the number of Fourier Transforms needed to calculate the final intensity.

### *3.1. CPU implementation*

In this section we exploit the parallelization provided by the multiple cores of CPUs to compute the intensity of each pixel in the image in parallel. Proper parallelization demands that each loop be independent of the results of each other. The input parameters of the algorithm we design must not be modified in any way, as to not interfere with each separate process. As inputs, we just need the three arrays $u, h, j$ which represent the complex transmittance of the object, the PSF of the system and the complex degree of coherence of the illumination. We describe here an algorithm capable of computing the intensity of each pixel without relying on sharing the state of the input variables or intermediate results between processes.

Direct implementation implies the calculation of $N_x \times N_y$ Fourier Transforms, as evidenced by (14). However, the Fourier Transform of the complex degree of coherence is constant, independent of the pixel whose intensity value we are calculating. We then are able to calculate it just once and pass the final array by reference when the summation is performed.

Determination of $\phi$ implies that for each pixel we must perform the following calculation:

$$\phi_{ij}[k,l] = h[i-k, j-l] u[k,l] \tag{15}$$

To perform such a product, we assume periodic boundary conditions on the array $h$, representing the PSF of the system. Then, the complete array may be calculated in the following way, in Algorithm 1. This way, $\phi$ is generated without modification of the input arrays $u, h$.

**Algorithm 1** Calculation of $\phi_{i,j}$

```
function MultShift(u, h, i, j)
    φ ← empty array of shape N_x × N_y
    i ← shift in the x direction
    j ← shift in the y direction
    for k in 0 : N_x do
        for l in 0 : N_y do
            φ[k,l] = h[(i − k) mod N_x, (j − l) mod N_y] u[k,l]
    return φ
```

Now, to parallelize the computation we must create a function to be applied at each iteration of the loop, which we call the *kernel* of the computation. In our case, this kernel has the form present in Algorithm 2a.

**Algorithm 2** Kernel of the parallel CPU computation

1: **function** KERNEL(u, h, $\hat{\text{j}}$, i, j)
2: $\phi \leftarrow$MULTSHIFT(u, h, i, j)
3: $f \leftarrow$ Fourier Transform of $\phi$
4: **return** Sum of $\hat{j} \cdot f \cdot f^{\star}$

The algorithms used to perform the discrete Fourier Transforms belong to the library Fastest Fourier Transform in the West (FFTW3), which are extremely well optimized to perform transformations on array sizes multiples of 2, 3, 5, 7 and combinations.

*3.2. GPU implementation*

GPUs consist of an array of small processing elements which perform calculations in parallel. Each processing element has associated a small amount of memory, albeit extremely fast, to store intermediate results of computations. Processing elements associate into working groups, sharing a certain amount of shared memory. This memory is slower, but allows for communication of results between processing elements and synchronization. The set of working groups share the global memory, a large pool of slow memory where the final calculations of the processing elements are stored. This global memory connects with the main memory of the computer, but with an extremely slow speed. Therefore, communication between main memory and GPU memory must be reduced as much as possible during comptutations.

The key point in GPU programming is the implementation of kernels, functions to be executed by each processing element in parallel. There exist two languages made for this purpose: CUDA and OpenCL. The former is a commercial implementation made by NVIDIA to support solely their own line of products. It is a very mature language, aimed specifically to scientists and engineers, with excellent documentation. The latter is an open standard developed by the Khronos Group, whose range of applications is not limited to just GPUs. The present work uses a third party wrapper called Reikna, which unifies both NVIDIA and OpenCL languages under the same syntax, letting the programmer select at compilation time which API to use.

The functions we are implementing are `MultShift` and `MultSum`. The first one is the multiplication with shift exposed in Algorithm 1, while the latter corresponds to line 4 of Algorithm 2. The FFT needed is already implemented by NVIDIA and other third parties which correctly exploit the massive parallelization of GPUs, although working optimally just with array sizes corresponding to powers of two.

Firstly, we present the implementation of `MultShift` in Algorithm 1.

**Algorithm 3** GPU kernel of MultShift

1: **function** KERNELMULTSHIFT(n, $\phi$, $h$, $u$, k, l)
2: $i \leftarrow x$ position of the processing element in the array
3: $j \leftarrow y$ position of the processing element in the array
4: $i_2 \leftarrow k - i$ ▷ shifted x position in the array
5: $j_2 \leftarrow l - j$ ▷ shifted y position in the array
6: **if** $i_2 < 0$ **then**
7: $i_2 \leftarrow i_2 + n$ ▷ Circular shift in the x dimension, being $n$ the number of points
8: **if** $j_2 < 0$ **then**
9: $j_2 \leftarrow j_2 + n$ ▷ Circular shift in the y dimension, being $n$ the number of points
10: $\phi[i, j] \leftarrow h[i_2, j_2]u[i, j]$

The most interesting factor of this Algorithm is the disappearence of all the loops in the

element-wise multiplication. Each processing element has assigned a number corresponding to the element of the array it acts upon. The iterations are performed in parallel throughout the entire GPU. If enough cores are available, this algorithm is performed in constant time.

The implementation of `MultSum` is presentedn in Algoritm 1.

**Algorithm 4** GPU kernel of MultSum

```
1: function KernelMultSum(n, I, f, ĵ, k, l)
2:     i ← xposition of the processing element in the array
3:     j ← y position of the processing element in the array
4:     p ← x position of the processing element in the work group
5:     q ← x position of the processing element in the work group
6:     t[p, q] ← f · [i, j]f*[i, j] · ĵ[i, j]  ▷ Array to save the products performed by each element
       in the work group
7:     Synchronize here the threads: all processing elements in the workgroup stop here.
8:     if p = 0 and q = 0 then
9:         S = 0                                   ▷ initialize a value to hold the sums
10:        for w in 0 : N_wg do                    ▷ N_wg lateral dimension of workgroup
11:            for z in 0 : N_wg do
12:                S ← S + t[w, z]
13:        I[k, l] ← atomic addition of S
```

In this algorithm a sum of a multiplication of arrays is performed and the result stored into a single memory position of the array $I[i, j]$. As all processing elements act in parallel, some of the writes to this single memory location do not get through. Therefore, coordination between the processing elements is needed. As they can only be coordinated inside their respective work groups, a temporary array is created to store the values of the multiplication at each point. This array lives in the shared memory of the work group and its dimensions are severly limited by the size of this memory. Then, all threads are locked at the same point in the execution. Then, the first thread is made to perform the summation of all the elements of this temporary array. Once done, we perform an atomic addition: this directive tells the GPU to coordinate the memory writes from each workgroup into a single memory location without corrupting the resutl. With this, one single pixel of the image has been calculated.

*3.3. Algebraic method*

This method exploits the possibility of developeing the integrand of (13) in a finite series of functions. Proceeding as in [9], we write (13) as

$$I(x, y) = \iint \hat{j}(f', g')|\mathcal{F}[\hat{u}(f - f', g - g')]|^2 df' dg' \tag{16}$$

where $\hat{u}(f, g)$ and $\hat{j}(f, g)$ are the Fourier Transforms of $u(\vec{r})$ and $j(\vec{r})$ respectively. As stated in [9], this can be rewritten in matrix form transforming the integration into a summation in the following matrix form

$$I(x, y) = \mathbf{\Phi}^\dagger \mathbf{B}^\dagger \mathbf{B} \mathbf{\Phi} \tag{17}$$

Where $\mathbf{\Phi}$ is a row matrix and $\mathbf{B}$ is an $N^2 \times m$ matrix, where $N$ is the lateral number of points of the object array and $m$ is the number of nonzero point in the Fourier Transform of the complex degree of coherence, $\hat{j}(f, g)$. Performing singular value decomposition (SVD) onto this matrix

allows us to rewrite (17) in the form

$$I(x,y) = \sum_{i=1}^{m} |\lambda_i \mathcal{F}[\psi_i(f,g)]|^2 \tag{18}$$

where $\psi_i(f,g)$ is one of the eigenfunctions of $\mathbf{B}$ with its corresponding eigenvalue $\lambda_i$. Algorithm 1 implements the calculation of this method in a practical way.

**Algorithm 5** Function implementing an algebraic method for partially coherent image formation

**function** AlgebraicPartial(u, $\hat{h}$, $\hat{j}$)
  $N \leftarrow$ Lateral number of points of the object array
  $a \leftarrow \mathcal{F}[u]$ ▷ Fourier transform of the amplitude
  $m \leftarrow$ Number of nonzero points of $\hat{j}$
  $B \leftarrow$ Empty complex array of shape $(m, N^2)$
  $AP \leftarrow a \cdot \hat{h}$
  **for** $i$ `in` 0:m **do**
    $B[i,:] \leftarrow$ Flattened array AP shifted in the x and y directions by the position of the $i$th nonzero element of $\hat{j}$.
  $U, s, V_h \leftarrow \text{svd}(B)$ ▷ Singular value decomposition
  $I \leftarrow$ Empty float array to store the final intensity.
  **for** $i$ `in` 0:m **do**
    $\psi_i \leftarrow V_h^\star$ reshaped into $(N, N)$
    $I \leftarrow I + |\mathcal{F}[u \cdot \psi_i \cdot s[i,:]]|^2$
  **return** $I$

## 4. Results

The algorithms have been implemented in Python, using numerical computation libraries and an API that unifies CUDA and OpenCL interfaces under the same language. CPU parallelization is performed with the library multiprocessing of the standard Python library. The specifications of the system used for the calculations are presented in Table 1.

**Table 1.** Specifications of the computer used to run the algorithms

| | |
|---|---|
| **CPU** | Intel® Core™i5-4460 @ 3.20 GHz |
| **GPU** | NVIDIA® GeForce® GTX 750 |
| **RAM** | 8 GB |

### *4.1. Execution time of the algorithms*

Four different array sizes are used in this section: $64\times64$, $128\times128$, $256\times256$ and $512\times512$. The comparison is performed under two different situations. In the first one, the Fourier Transform of the complex degree of coherence has a constant number of nonzero pixels across all sizes. For

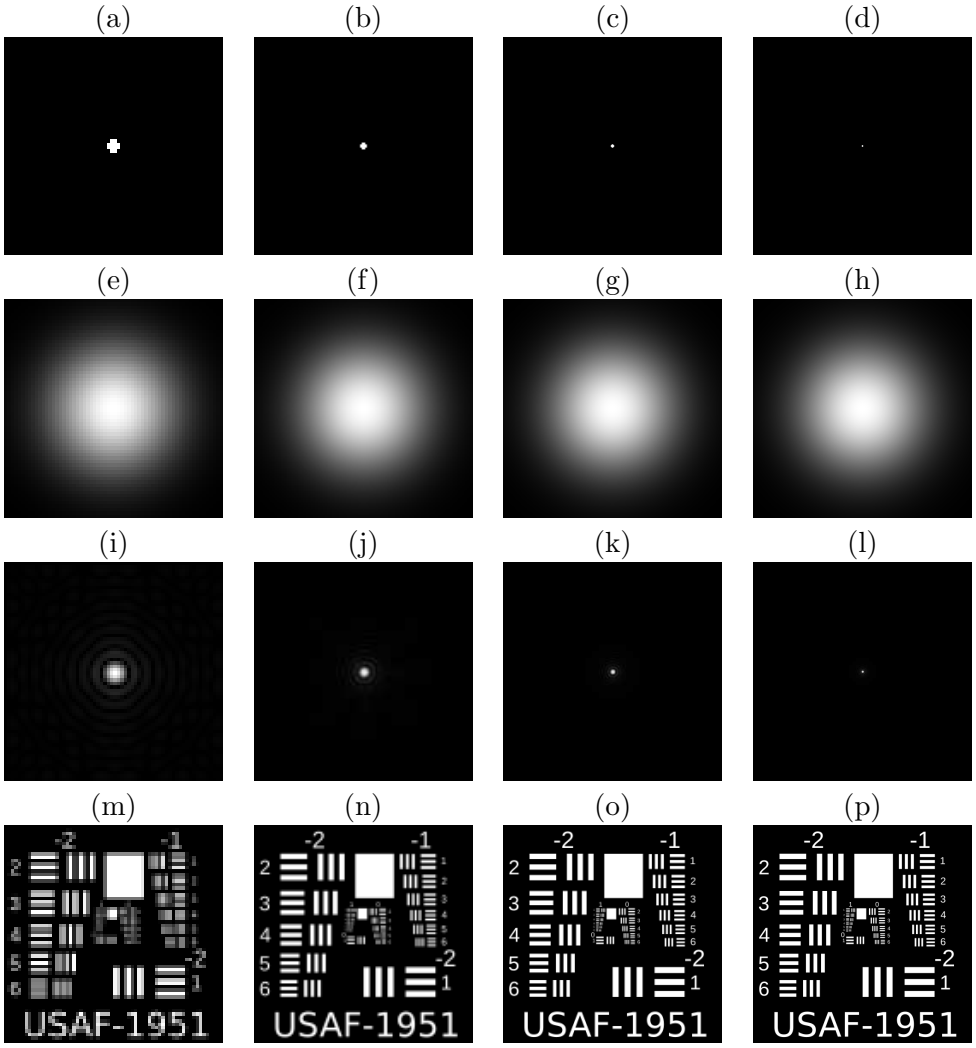


**Figure 1.** Input data for the algorithms in the two situations described. (a)-(d) are the Fourier Transforms of the complex degree of coherence for the first case, (e)-(h) are the complex degrees of coherence for the second case, (i)-(l) are the PSFs for both simulations and (m)-(p) are the object amplitudes.

simplicity, we arrange the nonzero points in a circle as in Figure 1 (a)-(d). The second situation is characterized by a constant complex degree of coherence. We impose its form to be

$$j(\vec{r}) = \exp\left[-\frac{r^2}{2\mu_0}\right] \tag{19}$$

where $\mu_0$ is the so-called coherence length, constant for all images in this arrangement. The complex degrees of coherence used in this case are presented in Figure 1 (e)-(h). Finally, the PSFs under this situation are shown in Figure 1 (i)-(l).

Results for the first case are presented in Figure 2 (a). The algebraic method is, at worst, one order of magnitude faster than the rest and, at best, two orders of magnitude faster. The number of operations needed to obtain the final intensity in this case is radically smaller than in the other two cases. However, if the number of points in the Fourier Transform of $\hat{j}$ increases, the execution time and memory consumption aslo increase. In fact, the memory increases in

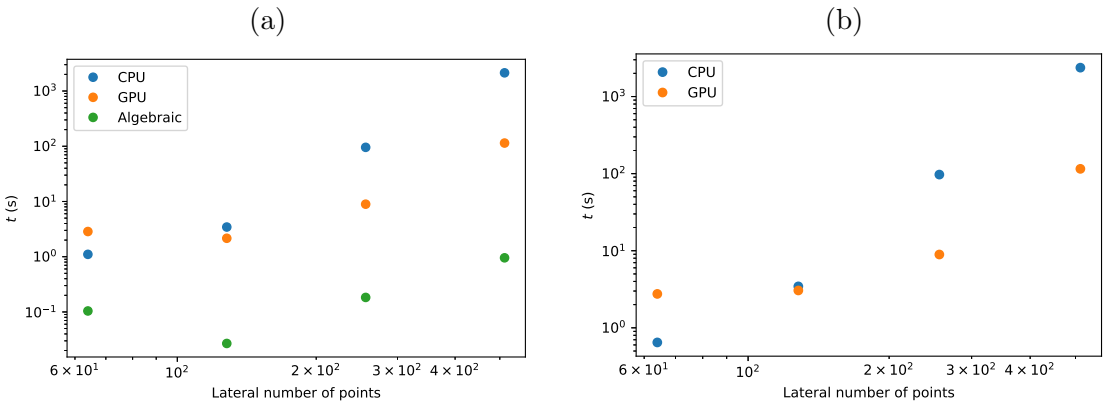


**Figure 2.** Execution times of the CPU, GPU and algebraic implementations. (a) correspond to the simulations with constant Fourier Transform of the complex degree of coherence and (b) are those with constant complex degree of coherence. (b) does not show the algebraic implementation as it was not able to finish due to memory constraints.

such a way that our computer is unable to perform the calculations in the algebraic method. For this reason, the results of the second situation in Figure 2 (b) show only the GPU and CPU algorithms. Interestingly, the execution times in these two situations for both algorithms remains constant. These implementations of the integral (13) are independent of the exact form of the complex degree of coherence, PSF and complex amplitude involved. Moreover, the memory consuption remains constant throughout.

*4.2. Comparison with experimental results*

Finally, we compare the GPU algorithm with experimental results. We reproduce the results found in [15], with the same experimental setup. As seen in Figure 3, the same edge effects can be observed in the simulated (upper row) and experimental (lower row) images. The accuracy of our algorithm is thus verified.

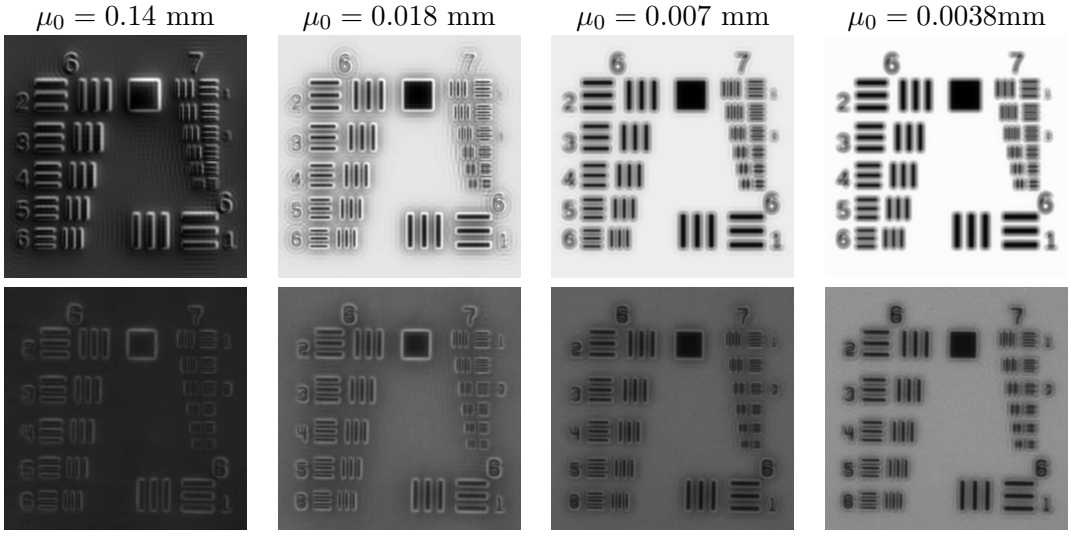


**Figure 3.** Upper row: simulations using the GPU algorithm for different degrees of coherence. Lower row: experimental results for the same degrees of coherence.

## 5. Conclusions

Three different implementations of the same integral for partially coherent image formation have been implemented. One of them treats algebraically the problem: the integrand is effectively divided into a decomposition of orthonormal, two dimensional functions and image instensity is determined by just the sum of the squared modulus of their Fourier Transforms. This results in an extremely fast calculation of the image formation process if the number of required orthonormal functions is small enough. Since the number of functions is exactly the number of nonzero points in the Fourier Transform of the complex degree of coherence, this severly limits the range of simulations we are able to perform with this method.

The other two methods are a direct implementation of (14). The first algorithm uses CPU parallelization to calculate in parallel each pixel on the image. This method is inefficient, as the time required to calculate an image of $512 \times 512$ is around an hour. In comparison, GPU parallelization of the Fourier Transform brings a tenfold improvement of the computation time. As this algorithm does not depend on the shape of the complex degree of coherence, it represents a viable method to calculate image intensities under arbitrary shapes of the aforementioned quantity.

## 6. References


[1] H. H. Hopkins, "On the diffraction theory of optical images," *Proceedings Mathematical Physical and Engineering Sciences*, vol. 217, 1953.

[2] L. Mandel and E. Wolf, *Optical Coherence and Quantum Optics*. Cambridge: Cambridge University Press, 1995.

[3] J. W. Goodman, "Statistical optics," *New York, Wiley-Interscience, 1985, 567 p.*, vol. 1, 1985.

[4] F. Zernike, "The concept of degree of coherence and its application to optical problems," *Physica*, vol. 5, 1938.

[5] M. Born and E. Wolf, *Principles of Optics: Electromagnetic Theory of Propagation, Interference and Diffraction of Light (7th Edition)*. Cambridge University Press, 7th ed., 1999.

[6] J. W. Goodman, "Introduction to fourier optics," *Introduction to Fourier optics, 3rd ed., by JW Goodman. Englewood, CO: Roberts & Co. Publishers, 2005*, vol. 1, 2005.

[7] A. Kwok-Kit Wong, *Resolution Enhancement Techniques in Optical Lithography*. SPIE Digital Library, 1st ed., 2001.

[8] N. Cobb, "Sum of coherent systems decomposition by svd," 1995.

[9] K. Yamazoe, "Two matrix approaches for aerial image formation obtained by extending and modifying the transmission cross coefficients," *J. Opt. Soc. Am. A*, vol. 27, pp. 1311–1321, Jun 2010.

[10] P. Gong, S. Liu, W. Lv, and X. Zhou, "Fast aerial image simulations for partially coherent systems by transmission cross coefficient decomposition with analytical kernels," *Journal of Vacuum Science & Technology B Microelectronics and Nanometer Structures*, vol. 30, 2012.

[11] J. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable parallel programming with cuda," *Queue*, vol. 6, pp. 40–53, Mar. 2008.

[12] K.-S. O. K. Jung, "Gpu implementation of neural networks," *Pattern Recognition*, vol. 37, 2004.

[13] K. Moreland and E. Angel, "The fft on a gpu," in *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pp. 112–119, Eurographics Association, 2003.

[14] E. Titchmarsh, *Introduction to the theory of Fourier integrals*. Clarendon Press, 1948.

[15] M. Prez-Avioa, R. Martnez-Herrero, S. Vallmitjana, P. Latorre-Carmona, I. Juvells, and A. Carnicer, "Partially-coherent spirally-polarized gradual-edge imaging," *Optics and Lasers in Engineering*, vol. 112, 01 2019.