CUDA-Based Particle Swarm Optimization in Reflectarray Antenna Synthesis

Amedeo Capozzoli, Claudio Curcio, Angelo Liseno
Universitá di Napoli Federico II, Dipartimento di Ingegneria Elettrica e delle Tecnologie dell'Informazione, via Claudio 21, I 80125 Napoli (Italy)
Corresponding author: Amedeo Capozzoli (e-mail: a.capozzoli@unina.it)

ABSTRACT The synthesis of electrically large, highly performing reflectarray antennas can be computationally very demanding both from the analysis and from the optimization points of view. It therefore requires the combined usage of numerical and hardware strategies to control the computational complexity and provide the needed acceleration. Recently, we have set up a multi-stage approach in which the first stage employs global optimization with a rough, computationally convenient modeling of the radiation, while the subsequent stages employ local optimization on gradually refined radiation models.
The purpose of this paper is to show how reflectarray antenna synthesis can take profit from parallel computing on Graphics Processing Units (GPUs) using the CUDA language. In particular, parallel computing is adopted along two lines. First, the presented approach accelerates a Particle Swarm Optimization procedure exploited for the first stage. Second, it accelerates the computation of the field radiated by the reflectarray using a GPU-implemented Non-Uniform FFT routine which is used by all the stages.
The numerical results show how the first stage of the optimization process is crucial to achieve, at an acceptable computational cost, a good starting point.

INDEX TERMS Reflectarray antenna synthesis, CUDA, parallel programming, Graphics Processing Units (GPUs), Particle Swarm Optimization.

I. INTRODUCTION

Microstrip reflectarrays (RAs) have attracted much interest in the last years [1], [2] and several new application fields have been introduced to RAs, as THz frequencies [3], 5G [4] and smart antennas for indoor applications [5].
The design of high performance microstrip reflectarrays requires the ability of exploiting all the degrees of freedom (DoFs) of the antenna as the positions, the orientations, the characteristics of the scattering elements or the shape of the reflecting surface.
Managing a large number of DoFs requires a proper synthesis strategy trading-off effectiveness and efficiency through efficient and effective numerical algorithms guaranteeing reliability, keeping the computational complexity low and exploiting highly performing computing hardware.
To guarantee reliability and accuracy, a multistage approach has been devised [6]–[8]. The idea is to employ a rough modeling of the structure to enable robustness against suboptimal solutions at the early stages and then refining the model from stage to stage. The success of the whole procedure is submitted to the capability of the early stages to mitigate the false solution issue. To this end, the unknowns of the problem are given proper representations enabling a progressive enlargement of the parameters space as well as a modulation of the computational complexity of the approach [6]–[8]. At the beginning of the whole procedure, few properly chosen polynomials are employed for the command phases and the element positions (Zernike polynomials for the phases and Lagrange polynomials for the positions in this paper) and progressively increased in number [6]–[8].
The success of the whole procedure is submitted to the capability of the early stages to mitigate the false solution issue. To this end, a global optimizer should be adopted. However, choosing the global optimization algorithm is not a simple task and a wide discussion is contained in [9]. The No Free Lunch Theorems [10] provide insight in choosing the strategy to get a successful algorithm for the problem at hand. They explore the relationship between an efficient optimization algorithm and the problem that it is asked to solve. These results state that a general purpose algorithm that can be efficiently used in all optimization problems does not exist: on average, the performances of any two optimization algorithms are the same across all the possible optimization problems [10]. Explicitly or implicitly inserting some a priori information on the structure of the problem at hand should improve the performance. It is suggested that an algorithm can be "aligned" to the structure of the problem because of an implicit tuning procedure due to training and/or

to the years of research [10].
On the other side, the high dimensionality of a problem is the main obstacle towards effective global optimization. The amount of computing time required to get a reliable solution becomes inordinate as long as the number of variables increase [11], [12]. In particular, the dependence of the performances of a global optimization algorithm versus the dimension of the searching space can be rigorously established thanks to Nemirovsky and Yudin Theorem [12]. It proves the exponential dependence of the number of minimization steps on the dimensionality of the problem.
Accordingly and once again, the number of unknowns of the problem must be modulated during the design by using a proper representation of the unknowns enabling its progressive enlargement [6]–[8]. Then, global optimization can be exploited as starting stage, while local tools can be employed in the subsequent ones.
In this paper, we use a Particle Swarm Optimization (PSO) approach [13]. PSO, indeed, is a simple, but powerful optimization algorithm which searches for the optimum of a function following rules inspired by the behavior of flocks of bees looking for food. PSO has recently gained more and more popularity due to its robustness, effectiveness, and simplicity. Furthermore, PSO enables a very efficient implementation since it maps into a highly parallelizable computing pattern and is amenable of acceleration using Graphics Processing Units (GPU) computing.
We show how PSO can take profit from this technology. In particular, we provide an implementation of PSO in CUDA language, the NVIDIA Compute Unified Device Architecture that represents the extension of C++ to let a large class of users able to program GPUs for numerical computing applications. CUDA has indeed already shown to be very promising for antenna analysis [14], [15] and synthesis [8] to accelerate computations. We show how computing time can be kept at very convenient duration for the application at hand.
The presented approach is further accelerated by the use of Non-Uniform FFTs [16] for the evaluation of the field radiated by the RA [14]. The Authors have been the first employing NUFFT algorithms for the calculation of the field radiated by aperiodic arrays and RAs [14]. They have presented an improved version of [16] in [17].
The paper is organized as follows. In Section II, the approach is sketched. In particular, the radiative model is described along with its calculation by a NUFFT. Furthermore, the NUFFT algorithm is briefly described along with its GPU implementation. Section III is devoted to discuss the GPU-based PSO implementation. In Section IV, the results are presented. Finally, in Section V, the conclusions are drawn.

II. THE APPROACH

A simplified layout of the multi-stage synthesis approach is illustrated in Fig. 1. We refer to [6]–[8] for a complete description of the approach.
In the initial stages, an approximate radiative model, namely, the Phase-Only (PO) model, is adopted leading to the Phase-Only Synthesis (POS) stage. For a POS, the unknowns to be determined are the element positions as well as the command phases. Then, an accurate model exploiting all the DoFs refines the results from the POS stages and leads to the Accurate Synthesis (AS). For the AS, the unknowns turn to be the internal DoFs of each element. The element positions can be further refined or, for the sake of simplicity, the result of the POS, in terms of element positions, can be kept fixed. In this paper, attention is focused to the POS only.
In order to limit the number of unknowns and make the use of a global optimizer affordable to generate a good initial guess, in stage 1), few polynomials are exploited to represent command phases and element positions. In stage 2), the number of the unknowns is progressively enlarged by increasing the order of the involved polynomials and a local optimizer is considered. Stage 3) exploits an impulsive representation of the unknowns involving all the effective DoFs at disposal for the command phases.
The details common to the three synthesis stages are now in order. The purpose of each stage is to find the unknown parameters, which change from stage to stage, in order to satisfy the coverage requirements. The unknown parameters, say $\underline{\underline{R}}$ for the command phases and $\underline{\underline{P}}$ for the element positions, should be obtained by minimizing a proper objective functional Φ given by:

$$\Phi(\underline{\underline{R}},\underline{\underline{P}}) = \left\| A_{co}(\underline{\underline{R}},\underline{\underline{P}}) - P_{\mathcal{U}_{co}}(A_{co}(\underline{\underline{R}},\underline{\underline{P}})) \right\|^2 + \left\| A_{cr}(\underline{\underline{R}},\underline{\underline{P}}) - P_{\mathcal{U}_{cr}}(A_{cr}(\underline{\underline{R}},\underline{\underline{P}})) \right\|^2 \quad (1)$$

where (A_{co}, A_{cr}) is the relevant radiation operator, namely

$$\begin{cases} A_{co} = |E_{co}|^2 \\ A_{cr} = |E_{cr}|^2 \end{cases}, \quad (2)$$

E_{co} and E_{cr} are the co-polar and cross-polar components of the far-field, $P_{\mathcal{U}_{co}}$ and $P_{\mathcal{U}_{cr}}$ are the metric projectors onto the set $\mathcal{U}_{co}$ and $\mathcal{U}_{cr}$ that contain all the power patterns satisfying the specifications for the co-polar and cross-polar components [18], and $\|\cdot\|$ is a properly chosen norm.
The sets $\mathcal{U}_{co}$ and $\mathcal{U}_{cr}$ are defined by mask functions (m_{co}, M_{co}) and (m_{cr}, M_{cr}), respectively, determining upper and lower bounds for A_{co} and A_{cr}, respectively [18]. The metric projectors $P_{\mathcal{U}_{co}}(A_{co})$ and $P_{\mathcal{U}_{cr}}(A_{cr})$ are defined as [18]

$$P_{\mathcal{U}_{co,cr}}(A_{co,cr}) = \begin{cases} m_{co,cr} & \text{if } A_{co,cr} \le m_{co,cr} \\ A_{co,cr} & \text{if } m_{co,cr} \le A_{co,cr} \le M_{co,cr} \\ M_{co,cr} & \text{if } M_{co,cr} \le A_{co,cr} \end{cases}. \quad (3)$$

For the cases of interest here, Φ will refer only to the copolar term and only the A_{co} operator (which will changed depending on the stage) will be relevant to our purposes.
In this paper, for the sake of brevity, only some implementation details of the POS stages are discussed.

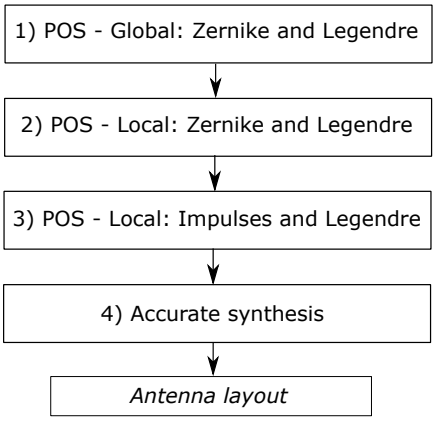

FIGURE 1. Stages of RA synthesis approach.

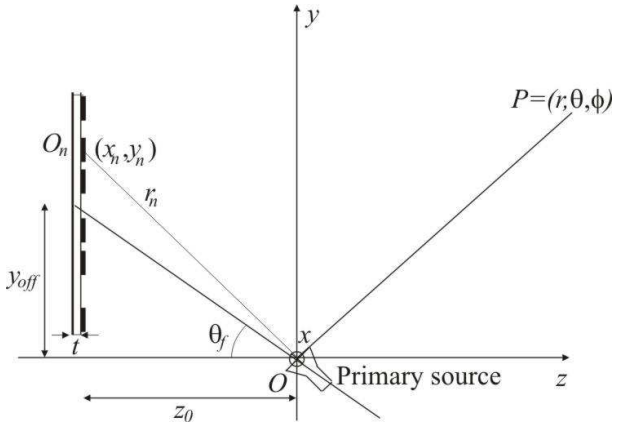

FIGURE 2. RA geometry.

A. THE RADIATIVE MODEL AND THE NUFFT

We refer to a flat aperiodic RA, made up by N elements, illuminated by a feed located at the origin of the $Oxyz$ reference system as shown in Fig. 2. The feed is assumed to have the typical $\cos^{m_f}(\theta_n)$ pattern, where θ_n is the angle under which the feed sees the n-th element having coordinates (x_n, y_n, z_0) and located at a distance r_n from the feed. In this way, the 2D aperiodic array factor F is:

$$F_{kl} = \sum_{n=1}^{N} e^{j\varphi_n} \cos^{m_f}(\theta_n) \frac{e^{-j\beta r_n}}{r_n} e^{j2\pi\left(\frac{k\Delta u}{\lambda} x_n + \frac{l\Delta v}{\lambda} y_n\right)} \quad (4)$$

when evaluated at a regular spectral grid $(u_k, v_l) = (k\Delta u, l\Delta v)$, with $k = -N_1/2, \ldots, N_1/2 - 1$ and $l = -N_2/2, \ldots, N_2/2 - 1$, the φ_n's being the element control phases, λ being the wavelength and $\beta = 2\pi/\lambda$ the wavenumber.

Apart from an unessential conjugation, F has the same expression of a 2D Non-Uniform Discrete Fourier Transform (NUDFT) of NED (Non-Equispaced-Data) type whose expression is

$$\hat{z}_{kl} = \sum_{i=1}^{N} z_i e^{-j2\pi\widetilde{x}_i \frac{k}{N_1}} e^{-j2\pi\widetilde{y}_i \frac{l}{N_2}}. \quad (5)$$

In eq. (5), the z_i's represent the sequence to be transformed, sampled at the non-uniform spatial coordinates $(\widetilde{x}_i, \widetilde{y}_i)$, and the $\hat{z}_{kl}$'s represent the transformed one. Accordingly, the array factor (4) can be effectively and efficiently evaluated using a 2D NED-NUFFT algorithm [14], [16], [17].

B. THE 2D NED-NUFFT ALGORITHM

To guarantee a fast and accurate processing, the "non-uniformly sampled" exponentials $\exp(-j2\pi\widetilde{x}_i k/N_1)$ and $\exp(-j2\pi\widetilde{y}_i l/N_2)$ appearing in eq. (5) need to be properly interpolated. The idea behind NED NUFFTs is to use "uniformly sampled" exponentials $\exp(-jm\xi)$.

The exponential representation is made possible by the use of the Poisson summation formula [17], [19]. Indeed, under general hypotheses, given a function f, then $\sum_{m\in\mathbb{Z}} f(\xi + 2m\pi)$ converges absolutely almost everywhere to the 2π-periodic locally integrable function simply expressed through a Fourier series

$$\sum_{m\in\mathbb{Z}} f(\xi + 2m\pi) \sim \sqrt{2\pi} \sum_{m\in\mathbb{Z}} \hat{f}(m) e^{jm\xi}, \quad (6)$$

where $\hat{f}(m) = \mathcal{F}[f; m]$ is the Fourier transform of f calculated at m.

Let us then consider a window function $\phi(\xi)$ having compact support in $(-\xi_M, \xi_M)$. The Poisson formula can be then applied to the function $\phi(\xi)\exp(-jx\xi)$:

$$e^{-j\xi x} = \sqrt{2\pi} \frac{\sum_{m\in\mathbb{Z}} \mathcal{F}\left[\phi(\xi) e^{-j\xi x}; m\right]}{\sum_{m\in\mathbb{Z}} \phi(\xi + 2m\pi) e^{-j2m\pi x}}. \quad (7)$$

In order to obtain a computationally convenient expression of $\exp(-jx\xi)$, the denominator must be factored in the variables x and ξ. A straightforward way to achieve factorization is to avoid that the replicas $\phi(\xi + 2m\pi)\exp(-j2k\pi\xi)$ overlap, which is related to the replication period 2π of the Poisson summation formula in eq. (7) and to a proper choice of the support of $\phi(\xi)$. In particular, we have:

$$e^{-j2\pi\widetilde{x}_i \frac{k}{N_1}} \simeq \frac{\sqrt{2\pi}}{\phi\left(\frac{2\pi k}{cN_1}\right)} \sum_{m=\mu_i - K}^{\mu_i + K} \hat{\phi}(cx_i - m) e^{-j2\pi m \frac{k}{cN_1}}, \quad (8)$$

where ϕ is the window function with support in $(-\pi/c, \pi/c)$ having Fourier transform $\hat{\phi}$ with support in $(-K, K)$, c is an oversampling factor, and $\mu_i = Int[c\widetilde{x}_i]$, $Int[\cdot]$ denoting he integer part. A similar expression can be obtained for the exponential with $\widetilde{y}_i$, by introducing $\nu_i = Int[c\widetilde{y}_i]$.

By denoting with

$$\begin{cases} \phi_k^1 = \phi\left(2\pi\frac{k}{cN_1}\right) & \phi_k^2 = \phi\left(2\pi\frac{l}{cN_2}\right) \\ \hat{\phi}_{it} = \frac{\hat{\phi}(cx_i-(\mu_i+t))}{\sqrt{2\pi}} & \hat{\psi}_{it} = \frac{\hat{\phi}(cy_i-(\nu_i+t))}{\sqrt{2\pi}} \end{cases}, \quad (9)$$

by assuming that both $\hat{\phi}_{it}$ and $\hat{\psi}_{it}$ vanish outside $0 \leq i \leq M$ and $-K \leq t \leq K$ and by exploiting the fact that $\exp(-j2\pi(m + cN_1)k/(cN_1)) = \exp(-j2\pi mk/(cN_1))$ and $\exp(-j2\pi(n + cN_2)l/(cN_2)) = \exp(-j2\pi nl/(cN_2))$, eq. (5) can be written as

$$\hat{z}_{kl} = \frac{1}{\phi_k^1\phi_l^2} \sum_{m=-cN_1/2}^{cN_1/2-1} \sum_{n=-cN_2/2}^{cN_2/2-1} u_{mn} e^{-j2\pi m\frac{k}{cN_1}} e^{-j2\pi n\frac{k}{cN_2}}, \quad (10)$$

where

$$u_{mn} = \sum_{i=-\infty}^{+\infty} \sum_{p=-\infty}^{+\infty} \sum_{q=-\infty}^{+\infty} z_i \hat{\phi}_{i\ m+cpN_1-\mu_i} \hat{\psi}_{i\ n+cqN_2-\nu_i}. \quad (11)$$

Therefore, eq. (10) is essentially given by the three steps [17]:

- interpolation to obtain u_{mn} (eq. (11));
- standard two-dimensional FFT on $cN_1 \times cN_2$ points;
- scaling and decimation.

It should be noticed that, in eq. (11), the u_{mn}'s receive contribution only from a very limited number of terms. Indeed, the $\hat{\phi}$ and $\hat{\psi}$ in eq. (11) are different from zero only for

$$|m+cpN_1-\mu_i| \leq K \ll cN_1 \quad |n+cqN_2-\nu_i| \leq K \ll cN_2, \quad (12)$$

where typical values for K are 3 and 6.

Finally, let us highlight that the obtained overall computational complexity is $\mathcal{O}((cN)^2 \log(cN))$ for $N \sim N \sim N_2$, $N_1, N_2, M \gg K$.

C. ACCELERATION OF THE PATTERN EVALUATION

Once described the 2D NED-NUFFT algorithm, let us briefly sketch the GPU acceleration.

The most difficult step to be implemented on GPU is the interpolation stage [8], [14]. Accordingly, only for this stage, we present some details of the strategy exploited to profit of the available massive parallelism.

If we denote the input index with i and output indices with m and n, according to eq. (12), a given input index contributes only to a small number of output terms. The adopted solution assigns each GPU thread to an input index. Unfortunately, with such a choice, race conditions occur since different inputs can contribute to the same output. The problem is solved by profiting of a special feature of GPU: the atomic operations. They make the access of different threads to the same output location (in the GPU global memory) serial.

Dynamic Parallelism has been exploited to perform the two nested "for loops" with indices p and q occurring in eq. (11), each one spanning only $2K + 1$ cycles. Indeed, the CUDA Dynamic Parallelism allows each CUDA function (kernel) to give work to itself, thus making the easy implementation of recursive algorithms possible, exploiting different levels of parallelism.

We present now some codes to let the Reader take a closer look at the details of the adopted coding strategy. In the Listing 1, each parent thread is assigned to an input index i. The parent thread (first level of parallelism) generates $(2K + 1) \times (2K + 1)$ children (second level of parallelism), by calling the "series terms" kernel shown in Listing 2. Each child thread takes into account for a single loop cycle (double summation).

The implemented NUDFT NED provides a significant speedup of the array factor calculation, of one order of magnitude with respect to a CPU code, for a RA made of 100×100 elements

```
__global__ void InterpolationNFFT2_2DGPUKernel(_)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    double cc_points1=cc*x[i];
    // It is the mu
    double r_cc_points1=rint(cc_points1);
    const double cc_diff1 =
          cc_points1-r_cc_points1;

    double cc_points2=cc*y[i];
    // It is the nu
    double r_cc_points2=rint(cc_points2);
    const double cc_diff2 =
          cc_points2-r_cc_points2;

    double2 temp_data = data[i];
    dim3 dimBlock(13,13); dim3 dimGrid(1,1);
    series_terms<<<dimGrid,dimBlock>>>(_); }
```

Listing 1. CUDA interpolation Kernel using the dynamic parallelism.

III. PSO

PSO [13] is a global optimization algorithm working according to the behavior of flocks of bees in search of food. It shows appealing convergence properties that can be obtained by tuning its parameters in order to align the algorithm to the problem at hand. PSO shows a highly parallelizable computing pattern, making its implementation amenable of acceleration using GPUs [20]. This mitigates the need of large number of iterations and of particle updates and fitness evaluations [13].

The search space dimension in PSO equals the overall number of unknowns N_{unkn}, i.e., the number of polynomial coefficients used to represent the command phase and the element coordinates. The particles' position and the so-called "velocity" $\underline{X}(t)$ and $\underline{V}(t)$, respectively, at the iteration step fictitiously represented by the time variable t, are N_{unkn}-dimensional vectors whose update rule is given by

```
__global__ void series_terms(_) {
    int m = threadIdx.x;
    int n = threadIdx.y;

    double tempd, phi_cap;

    double P = K*K-(cc_diff1 -(m-K))*
               (cc_diff1 -(m-K));

    if(P<0.) {
       tempd=rsqrt(-P);
       phi_cap = (1./pi_double)*
             ((sin(alpha/tempd))*tempd);   }
    else if(P>0.) { tempd=rsqrt(P);
       phi_cap = (1./pi_double)*
             ((sinh(alpha/tempd))*tempd); }
    else phi_cap = alpha/pi_double;

    P = K*K-(cc_diff2 -(n-K))*
        (cc_diff2 -(n-K));

    if(P<0.) {tempd=rsqrt(-P);
    phi_cap = phi_cap*(1./pi_double)*
           ((sin(alpha/tempd))*tempd);   }
    else if(P>0.) {tempd=rsqrt(P);
    phi_cap = phi_cap*(1./pi_double)*
           ((sinh(alpha/tempd))*tempd); }
    else phi_cap = phi_cap*alpha/pi_double;

    int PP1 = modulo((r_cc_points1+
              (m-K)+N1*cc/2),(cc*N1));
    int PP2 = modulo((r_cc_points2+
              (n-K)+N2*cc/2),(cc*N2));
    atomicAdd(&result[IDX2R(PP1,PP2,cc*N2)].x,
          temp_data.x*phi_cap);
    atomicAdd(&result[IDX2R(PP1,PP2,cc*N2)].y,
          temp_data.y*phi_cap); }
```

Listing 2. CUDA summation kernel invoked by the interpolation one.

$$\begin{cases} \underline{V}(t) = & wV(t-1)+ \\ & C_1R_1\left[\underline{X}_{best}(t-1)-\underline{X}(t-1)\right]+ \\ & C_2R_2\left[\underline{X}_{gbest}(t-1)-\underline{X}(t-1)\right] \\ \underline{X}(t) = & \underline{X}(t-1)+\underline{V}(t) \end{cases} \tag{13}$$

In eqs. (13), the velocity is actually dealt with as a "displacement" vector. Furthermore, in eqs. (13), C_1 and C_2 are proper positive constants, R_1 and R_2 are two random numbers uniformly distributed in $[0,1]$, $\underline{X}_{best}(t-1)$ is the best-fitness position (local best) reached by an individual particle at the time $t-1$ and $\underline{X}_{gbest}(t-1)$ is the best-fitness position (global best) ever found by the whole swarm up to $t-1$. Moreover, the term $\underline{X}_{best}(t-1)-\underline{X}(t-1)$ represents the cognitive contribution while $\underline{X}_{gbest}(t-1)-\underline{X}(t-1)$ is the social contribution. Finally, w is the so-called inertia weight [21] balancing global and local search.

According to eq. (13), the processing consists of five main steps: a) Initialization; b) Positions update; c) Fitness evaluation function; d) Local best update kernel; e) Global best update kernel. A code snippet of the PSO is illustrated in Listing 3.

Note that, in the set up implementation, the PSO optimization can be stopped according to two criteria: a) a fixed number of iterations; b) the global optimum does not change for a certain number (e.g., 3) of consecutive iterations. In other words, the termination condition accounts for either a prefixed computation time, which is related to the overall number of iterations, or a persistent stagnation connected to a significant reduction of the functional with respect to the initial value. Obviously, a prefixed number of iterations does not ensure the algorithm convergence, but it indicates the depletion of an assigned resource, namely, computation time. We have anyway experienced that, for the synthesis cases of our interest, a number of about 100 iterations is sufficient to achieve satisfactory results, so that criterion a) will be henceforth used. The implementation in Listing 3 corresponds to this choice.

The five steps are detailed in the following Subsections.

```
void h_PSO_Optimize(void) {

   ...

   particles_initialization <<<...>>>(...);

   CostFunctionalCalculation(...);

   local_best_update <<<...>>>(...);

   for(gen = 1; gen < numGen; ++gen) {

      g_positionsUpdate <<<...>>>(...);

      CostFunctionalCalculation(...);

      local_best_update <<<...>>>(...);

      h_findGlobalBest(...); }

   h_findGlobalBest(...);}
```

Listing 3. PSO function.

A. INITIALIZATIONS

Outside of the loop, the particles are initialized by using routines from the cuRAND library. Furthermore, the fitness for each particle and the local particle bests are initialized. The latter will be illustrated in Subsection 3.4. The former is sketched in Listing 4.

```
__global__ void particles_initialization(...) {

   const unsigned int tid = threadIdx.x
      + blockIdx.x * blockDim.x;

   float R = curand_uniform(&devStates[tid]);
   float pos = c_minValues[threadIdx.x] +
      R * c_deltaValues[threadIdx.x];
   d_positions[tid] = pos;
   d_best_personal_positions[tid] = pos;

  R = curand_uniform(&devStates[tid]);
   float vel = c_minValues[threadIdx.x] +
      R * c_deltaValues[threadIdx.x];
   d_velocities[tid] = (vel - pos) / 2.0; }
```

Listing 4. Particles initialization function.

B. POSITIONS UPDATE

Concerning the positions update, eq. (13) shows that, once the information related to the global best has been achieved, there is no need for any further communication among the particles. Accordingly, the update can occur in parallel both across the particles and across the particle dimensions. The kernel function performing the position update is schematically reported in Listing 4. For the execution of such a kernel, different blocks are assigned to handle different particles, while their corresponding block threads manage the particle dimensions. Care has been taken to store as much as possible local (to the thread) temporary data within the registers while avoiding register spilling. It is noted that the kernel function in Listing 4 also performs the update of the best personal particle position. However, to save global memory storage time, such update is performed only if the neighboring particles have better personal best positions than the particle at hand. The particle neighborhood is evaluated by storage of the relevant arrays in the texture memory. Further global memory loads are saved by storing the Boolean update information in the shared memory.

The positions update kernel is sketched in Listing 5.

```
__global__ void g_positionsUpdate(...){

   unsigned int tid = blockIdx.x * blockDim.x
                    + threadIdx.x;

   __shared__ unsigned int s_update;
   __shared__ unsigned int s_bestID;

   if(threadIdx.x == 0){
      s_update = d_to_be_updated[blockIdx.x];
      s_bestID = d_localBestIDs[blockIdx.x]
               * blockDim.x;
   }
   __syncthreads();

   float pos = d_positions[tid];
   float bestPos = d_best_personal_positions[tid];
   float vel = d_velocities[tid];

   float R1, R2;
   R1 = curand_uniform(&devStates[tid]);
   R2 = curand_uniform(&devStates[tid]);

   if (s_update){
      bestPos = pos;
      d_best_personal_positions[tid] = bestPos; }

   __threadfence();

   vel *= W;
   vel += C1 * R1 * (bestPos - pos);
   vel += C2 * R2 *
   (d_best_personal_positions[s_bestID
       + threadIdx.x] - pos);

   d_velocities[tid] = vel;
   pos += vel;
   pos = min(pos, c_maxValues[threadIdx.x]);
   pos = max(pos, c_minValues[threadIdx.x]);

   d_positions[tid] = pos; }
```

Listing 5. Positions update kernel function.

C. FITNESS EVALUATION FUNCTION

Regarding the evaluation of the fitness function in eq. (1), the far field pattern related to the unknowns represented by the current particle is calculated first by using the parallel kernels implementing the 2D NED NUFFT in Listings 1 and 2. Then, the fitness function for each particle is computed by a reduction operation implemented by using the Thrust library and stored in an array. Since the local best update kernel requires finding the best functional around each particle position as it will be shortly seen, in order to improve the caching mechanisms, such an array has been bound to a texture. The fitness evaluation function is reported in Listing 6.

```
float raFunctionalCalculation(...) {

   ...

   absKernel<<<...,...>>>(d_far_field_abs,
      d_far_field, ...);

   thrust::device_ptr<float>
      d_far_field_abs_device_pointer =
         thrust::device_pointer_cast(
            d_far_field_abs);
      float scale_factor =
         thrust::reduce(
            d_far_field_abs_device_pointer,
            d_far_field_abs_device_pointer +
            (2 * Nu) * (2 * Nv),
            static_cast<float>(0),
            thrust::maximum<float>());

   vectorMulConstant(d_far_field_abs,
      static_cast<float>(1) /
         scale_factor,
            (2 * Nu) * (2 * Nv));

   ...

   evaluateProjection<<<...,...>>>(
      d_far_field_abs_projected,
         d_far_field_abs,
            d_Internal_Coverage_f,
               d_External_Coverage_f,
                  ...);

      evaluateSquaredDifference<<<...,...>>>
         (d_far_field_abs_projected,
            d_far_field_abs, ...));

      ...

      float Numerator = thrust::reduce(
         d_far_field_abs_device_pointer,
         d_far_field_abs_device_pointer +
         (2 * Nu) * (2 * Nv));
      float Denominator =
         thrust::transform_reduce(
          d_far_field_abs_proj_device_ptr,
            d_far_field_abs_proj_device_ptr +
            (2 * Nu) * (2 * Nv),
            AbsTwo<float>(),
            static_cast<float>(0),
            thrust::plus<float>());
      Denominator = Denominator * Denominator;

   ... }
```

Listing 6. Fitness evaluation function.

D. LOCAL BEST UPDATE KERNEL

The local best update kernel is meant to update the personal best fitness value. In this case, a different thread manages a different particle. Again, to save global memory storage time, the update is executed only provided that the neighboring particles have better personal fitness. The necessary minimum value calculations are obtained by in-kernel Thrust calls. This is illustrated in the Listing 7.

```
__global__ void local_best_update(...) {

   int tid = blockIdx.x * blockDim.x
       + threadIdx.x;

   float *local_fun =
       (float *)malloc((2 * RADIUS + 1)
          * sizeof(float));

   for (int i = 0; i < 2 * RADIUS + 1; i++)
      local_fun[i] =
            tex1D(functional_texture,
               (float)(tid + 0.5 + (i - RADIUS))
                  / (float)numPart);

   thrust::device_ptr<float> dev_ptr
      = thrust::device_pointer_cast(local_fun);
   thrust::device_ptr<float> min_ptr
      = thrust::min_element(thrust::seq,
        dev_ptr, dev_ptr + 2 * RADIUS + 1);

   d_localBestIDs[tid] =
      (tid + (&min_ptr[0] - &dev_ptr[0])
         - RADIUS + numPart) % numPart;

   if (gen > 0) {
      unsigned int update = local_fun[RADIUS]
         < d_personal_best_functional[tid];
      d_to_be_updated[tid] = update;
      if (update)
         d_personal_best_functional[tid] =
            local_fun[RADIUS]; }
          else d_personal_best_functional[tid]
             = local_fun[RADIUS]; }}
```

Listing 7. Local best update kernel function.

E. GLOBAL BEST UPDATE KERNEL

The best-fitness position $\underline{X}_{gbest}(t-1)$ is determined by a direct CUDA Thrust call as illustrated in Listing 8.

```
void h_findGlobalBest(...){

   thrust::device_ptr<float> dp =
      thrust::device_pointer_cast(
         d_personal_best_functional);
   thrust::device_ptr<float> pos =
      thrust::min_element(
         dp, dp + tid);

        *h_globalBestID =
            thrust::distance(dp, pos);

        cudaMemcpy(h_globalBestFitness,
           &d_personal_best_functional
           [*h_globalBestID], sizeof(float),
           cudaMemcpyDeviceToHost); }
```

Listing 8. Global best update kernel function.

IV. RESULTS

The computational performance of the GPU implementation of the 2D NED-NUFFT algorithm has been compared with that of an analogous CPU implementation for 2D RAs having randomly located elements as well as random command phases. The codes have been run on an Intel Core i7-6700K, 4GHz, 4 cores (8 logical processors), equipped with an NVIDIA GTX 960 card, compute capability 5.2. Fig. 3 shows the GPU vs. CPU speedup when RAs having N elements are considered. The achieved speedup is larger than 10 even though atomic operations are employed: atomic operations on modern GPU architectures are indeed very fast. The speedup also saturates for values of N approaching millions of elements due to the saturation of the GPU resources.

Let us now turn to the results of the RA synthesis. The antenna here considered is an aperiodic RA working at $14.25GHz$, with a circular footprint, $z_0 = 58.3cm$, θ_f equal to about $11.2°$, $y_{off} = 11.6cm$, feed shape factor $m_f = 12$. The POS synthesized according to stages 1-3 of Fig. 1 provides the number of elements, their coordinates, and the corresponding command phase. Indeed, the optimization of the spatial distribution of the elements moves them freely, but only those elements falling within the assigned circular antenna footprint are effectively considered as scatterers. However, constraints on the minimum and the maximum spacing, equal to 0.49λ and 0.7λ, respectively, are enforced in all the stages, by discarding solutions not satisfying the requirements.

A target coverage of South America has been adopted for the test case, and enforced by two mask functions having a flat top behavior on the coverage, and a nominal side-lobe level lower than $40dB$. PSO has been executed with a swarm of 100 particles, by using 56 unknowns, $w = 0.721$, $C_1 = C_2 = 1.193$. The optimization on 100 iterations takes about $2hours$ on a PC equipped with a NVIDIA GTX 960. Accordingly, thanks to a proper parallelization of the global optimization and of direct radiation, the code has run in a convenient time notwithstanding the computational burden of the problem.

A total number of 2032 elements has been considered for the synthesized RA.

Fig. 4 reports the directivity pattern synthesized by the first stage (see Fig. 1), together with the mask functions (black lines). In other words, the result in Fig. 4 is the one obtained by the global optimization (PSO) stage which is appointed to operate on a rough model of the radiation in order to reduce the overall number of unknowns and to drive the solution as close as possible to the global optimum of the problem. Two observations can be made. First, the radiated pattern has a coverage corresponding to the prescribed one which suggests that the stage has been successful to bring the radiated solution close to the desired one. Second, as a result of the rough modeling, the pattern does not yet satisfactorily shape the target coverage.

In order to refine the synthesis, the outcome in Fig. 4 has been used as input to stage 2) and the outcome of stage 2) as

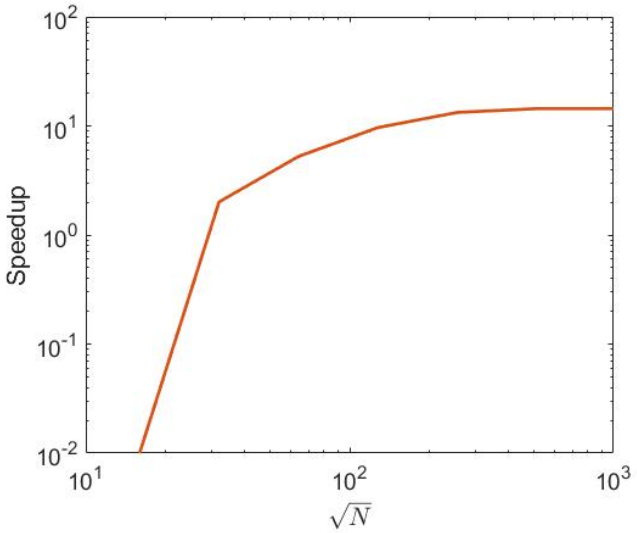

FIGURE 3. Speedup in log-scale of the 2D NED NUFFT implementation.

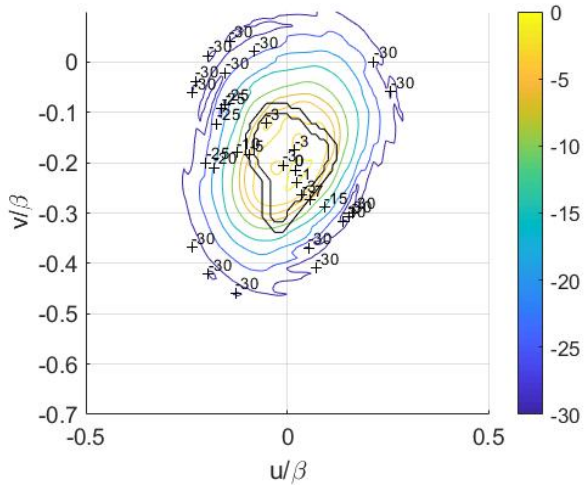

FIGURE 4. Directivity pattern synthesized by the PSO.

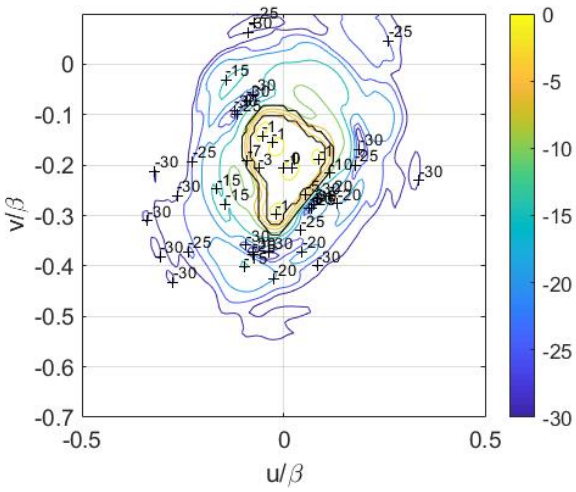

FIGURE 5. Directivity pattern synthesized after stage 3).

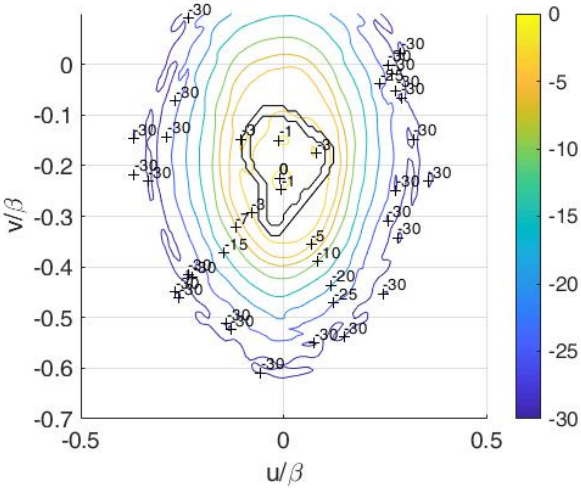

FIGURE 6. Directivity pattern synthesized by a local optimizer using a phase representation based on Zernike polynomials and skipping PSO.

input to stage 3). The final synthesized pattern at the end of stage 3) is shown in Fig. 5. Thanks to the improved radiation model, the pattern shaping is now satisfactory.

We want to finally underline how the use of an efficient global optimization algorithm as PSO is crucial to achieve a satisfactory synthesis matching the design specifications. To this end, we have considered a test case whose first stage has been skipped and we have proceeded directly with a local algorithm using a phase representation based on Zernike polynomials first and on impulses afterwards. The results are shown in Figs. 6 and 7, respectively. As it can be seen, the final result in Fig. 7 is significantly worse than that in Fig. 5.

V. CONCLUSIONS

In this paper, the use of GPU computing in reflectarray antenna synthesis has been discussed.

The described approach uses a multi-stage synthesis structure, using global and local optimizers. Global optimization is afforded by a GPU-based implementation of Particle Swarm Optimization. To further reduce the computational burden, the evaluation of the aperiodic array factor is performed by a GPU-implemented Non Uniform FFT routine.

The efficiency of the parallel approach for the analysis and synthesis of reflectarray antennas and of the parallel PSO algorithm in particular have been pointed out. More in detail, the computation time for the direct, parallel evaluation of the radiated field has shown a large speedup with respect to the sequential case. Furthermore, we have shown how the use of PSO in the first algorithm stage is crucial to achieve a satisfactory synthesis matching the design specifications. We have also highlighted how, thanks to a proper parallelization of the computationally most demanding stage of the synthesis

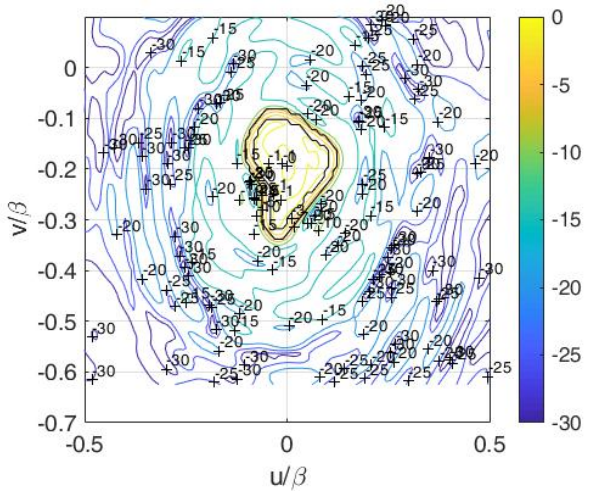

FIGURE 7. Directivity pattern synthesized by a local optimizer using an impulse-based phase representation and skipping PSO.

process, i.e., global optimization, as well as of direct radiation, the code has run in a convenient time notwithstanding the computational burden of the problem.
Finally, several implementation details have been provided to make the approach reproducible by any interested Reader.

REFERENCES

[1] J. Huang, J.A. Encinar, Reflectarray Antennas, J. Wiley & Sons, Hoboken, NJ, 2008.
[2] J.A. Encinar, M. Arrebola, and G. Toso, A parabolic reflectarray for a bandwidth improved contoured beam coverages, Proc. of the Europ. Conf. on Antennas Prop., Edinburgh, UK, Nov. 11–16, 2007, pp. 1–5.
[3] X. You, R.T. Ako, W.S.L. Lee, M.X. Low, M. Bhaskaran, S. Sriram, C. Fumeaux, W. Withayachumnankul, Terahertz reflectarray with enhanced bandwidth, Adv. Opt. Mat., vol. 7, n. 20, pp. 1–8, Oct. 2019.
[4] M.H. Dahri, M.H. Jamaluddin, M.I. Abbasi, M.R. Kamarudin, A review of wideband reflectarray antennas for 5G communication systems, IEEE Access, vol. 5, pp. 17803–17815, Aug. 2017.
[5] X. Tan, Z. Sun, J.M. Jornet, D. Pados, Increasing indoor spectrum sharing capacity using smart reflect-array, Proc. of the IEEE Int. Conf. on Commun., Kuala Lumpur, Malaysia, May 22-27, 2016, pp. 1-6.
[6] A. Capozzoli, C. Curcio, G. D'Elia, A. Liseno, P. Vinetti, G. Toso, Aperiodic and non-planar array of electromagnetic scatterers, and reflectarray antenna comprising the same, World Patent Nr. WO/2011/033388.
[7] A. Capozzoli, C. Curcio, A. Liseno, G. Toso, Fast, Phase-only synthesis of flat aperiodic reflectarrays, Progr. Electromagn. Res. vol. 133, pp. 53–89, 2013.
[8] A. Capozzoli, C. Curcio, A. Liseno, G. Toso, Fast, Phase-Only synthesis of aperiodic reflectarrays using NUFFTs and CUDA, Progr. Electromagn. Res. vol. 156, pp. 83–103, 2016.
[9] A. Capozzoli, G. D'Elia, Global optimization and antennas synthesis and diagnosis, part one: concepts, tools, strategies and performances, Progr. Electromagn. Res. vol. 56, pp. 195-–232, 2006.
[10] D.H. Wolpert, W.G. Macready, No free lunch theorems for optimization, IEEE Trans. Evolutionary Comput., vol. 1, n. 1, pp. 67-82, Apr. 1997.
[11] S.A. Vavasis, Nonlinear Optimization: Complexity Issues, Oxford Science Publication, New York, 1991.
[12] A. S. Nemirovsky, D.B. Yudin, Problem Convexity and Method Efficiency in Optimization, John Wiley & Sons, New York, 1983.
[13] J. Kennedy, R. Eberhart, Particle swarm optimization, Proc. of the IEEE Int. Conf. on Neural Networks, Perth, WA, Nov. 27-Dec. 01, 1995, pp. 1942-1948.
[14] A. Capozzoli, C. Curcio, G. D'Elia, A. Liseno, P. Vinetti, Fast CPU/GPU pattern evaluation of irregular arrays, ACES J., vol. 25, n. 4, pp. 355-372, Apr. 2010.
[15] R.C.M. Pimenta, M.V. Africano, R. Adriano, Ú. C. Resende, 3D CUDA FDTD based method for analysis of microstrip antennas, Proc. of the SBMO/IEEE MTT-S Int. Microw. Optoelectr. Conf., Aguas de Lindoia, Brazil, Aug. 27–30, 2017, pp. 1–5.
[16] K. Fourmont, Non-Equispaced Fast Fourier Transforms with applications to tomography, J. Fourier Anal. Appl., vol. 9, n. 5, pp. 431-450, 2003.
[17] A. Capozzoli, C. Curcio, A. Liseno, Optimized nonuniform FFTs and their application to array factor computation, IEEE Trans. Antennas Prop., vol. 67, n. 6, pp. 3924—3938, Jun. 2019.
[18] O.M. Bucci, G. D'Elia, G. Mazzarella, G. Panariello, Antenna pattern synthesis: A new general approach, Proc. of the IEEE, vol. 82, n. 3, 358—371, Mar. 1994.
[19] R.M. Tribug, E.S. Belinsky, Fourier Analysis and Approximation of Functions, Springer Science+Business Media, Dordrecht, NL, 2004.
[20] L. Mussi, F. Daolio, S. Cagnoni, Evaluation of particle swarm optimization algorithms within the CUDA architecture, Information Sciences 181: 4642–4657, 2011.
[21] Y. Shi, R. Eberhart, A modified particle swarm optimizer, Proc. of the IEEE Int. Conf. on Evolutionary Comput., Anchorage, Ak, May 4-9, 1998, pp. 69–73.