*Parallel Computing: Technology Trends*
*I. Foster et al. (Eds.)*


*doi:10.3233/APC200056*

# A GPU-CUDA Framework for Solving a Two-Dimensional Inverse Anomalous Diffusion Problem

P. De Luca [a], A. Galletti [b], H. R. Ghehsareh [c], L. Marcellino [b] and M. Raei [c]
[a] *University of Salerno, Department of Computer Science, Fisciano, Italy*
[b] *University of Naples Parthenope, Department of Science and Technology, Italy*
[c] *Malek Ashtar University of Technology, Department of Mathematics, Isfahan, Iran*

**Abstract.** This paper deals with the solution of an inverse time fractional diffusion equation described by a Caputo fractional derivative. Numerical simulations, involving large domains, give rise to a huge practical problem. Hence, by starting from an accurate meshless localized collocation method using radial basis functions (RBFs), here we propose a fast algorithm which exploits the GPU-CUDA capabilities. More in detail, we first developed a C code which uses the well-known numerical library LAPACK to perform basic linear algebra operations in order to implement an efficient sequential algorithm. Then we propose a GPU software based on ad hoc parallel CUDA-kernels and efficient usage of parallel numerical libraries available for GPUs. Performance analysis will show the reliability and the efficiency of the proposed parallel implementation.

**Keywords.** Fractional calculus; GPU computing; parallel algorithms; CUDA

## 1. Introduction

In recent decades, fractional calculus theory received high interest due to its application in several fields in science and engineering. Indeed, fractional models are beneficial and powerful mathematical tools to describe the inherent properties of processes in mechanics, chemistry, physics, and other sciences [1,2,3]. Main methods to solve fractional models include meshless and radial basis functions based methods which represent reliable and efficient techniques, specially suitable for high-dimensional and irregular computational domains [4,5]. This choice is because they allow to avoid the expensive task related to the mesh construction. However, as is well-known, the use of global RBFs in such cases gives rise to very ill-conditioned discrete problems. Moreover, practical problems with large domains increase the computational cost dramatically. Therefore, the use of fast algorithms and parallel computational kernels becomes unavoidable and necessary. So, in this work, by starting from an accurate meshless localized collocation method using local radial point interpolation (LRPI), we propose a fast algorithm which exploits the GPU-CUDA capabilities. More in detail, we first developed a C code based

on the numerical library LAPACK to perform all basic linear algebra operations. Then, we moved on a GPU-parallel software based on ad hoc parallel CUDA-kernels and efficient usage of parallel numerical libraries available on CUDA (Compute Unified Device Architecture) environment. To be specific, to improve performances, we introduce a parallel approach which implements parallel modules by using the CUDA computing platform for GPUs [6] (a massively parallel architecture, also know as Graphic Cards, for general purpose computing). Moreover, this GPU-parallel software involves the use of the cuSOLVER library [7], which in turn provides all the functions of LAPACK for the GPU environment, and it includes a easy-to-use interface making possible to vary both the main parameters of the problem and the blocks and threads configuration which are required by the parallel execution.

The rest of the paper is organized as follows: section 2 briefly describes the problem we deal with and the numerical procedure to discretize the related inverse time fractional diffusion equation; in section 3 the description of both sequential and parallel implementation details of the algorithms is given; in section 4 the experimental results highlight the performance gain, in terms of execution times and speed-up, compared to the sequential version; finally, conclusions close the paper in section 5.

## 2. Mathematical model and numerical procedure details

In the current work we deal with the solution of a two-dimensional inverse time fractional diffusion equation [8,9,10], defined as follows:

$$ {}_0^cD_t^\alpha v(\mathbf{x},t) = \kappa \Delta v(\mathbf{x},t) + f(\mathbf{x},t), \qquad \mathbf{x} = (x,y) \in \Omega \subseteq R^2, \quad t \in ]0,T], \tag{1} $$

with the initial and Dirichlet boundary conditions:

$$ \begin{aligned} &v(\mathbf{x},0) = \varphi(\mathbf{x}), && \mathbf{x} \in \Omega, \\ &v(\mathbf{x},t) = \psi_1(\mathbf{x},t), && \mathbf{x} \in \Gamma_1, \quad t \in ]0,T], \\ &v(\mathbf{x},t) = \psi_2(\mathbf{x})\rho(t), && \mathbf{x} \in \Gamma_2, \quad t \in ]0,T], \end{aligned} \tag{2} $$

and the non-local boundary condition:

$$ \iint_\Omega v(\mathbf{x},t)d\mathbf{x} = h(t), \qquad t \in ]0,T], \tag{3} $$

where $v(\mathbf{x},t)$ and $\rho(t)$ are unknown functions and ${}_0^cD_t^\alpha = \frac{\partial^\alpha}{\partial t^\alpha}$ denotes the Caputo fractional derivative of order $\alpha \in ]0,1]$. The numerical approach to discretize the problem (1) is summarized as follows [8,9].

### 2.1 The time discretization approximation

Let us choose a time step $\tau > 0$ and set $t^n = n\tau$, for $n = 0,\ldots,T/\tau$ (assume $T/\tau$ be an integer). By substituting $t = t^{n+1}$ in the equation (1), the following relation is obtained:

$$ {}_0^cD_t^\alpha v(\mathbf{x},t^{n+1}) = \kappa \Delta v(\mathbf{x},t^{n+1}) + f(\mathbf{x},t^{n+1}), \quad (\mathbf{x},t^{n+1}) \in \Omega \times (0,T]. \tag{4} $$

Therefore, we exploit the following second-order time discretization for the Caputo derivative of $v(\mathbf{x},t)$ at point $t=t^{n+1}$ [11,12]:

$$\left[{}_0^cD_t^\alpha v(\mathbf{x},t)\right]_{t=t^{n+1}} = \sum_{j=0}^{n+1}\frac{\omega^\alpha(j)}{\tau^\alpha}v(\mathbf{x},t^{n+1-j}) - \frac{t^{-\alpha}}{\Gamma(1-\alpha)}v(\mathbf{x},0)+O(\tau^2), \tag{5}$$

where:

$$\omega^\alpha(j)=\begin{cases}\dfrac{\alpha+2}{2}p_0^\alpha, & j=0,\\ \dfrac{\alpha+2}{2}p_j^\alpha-\dfrac{\alpha}{2}p_{j-1}^\alpha, & j>0,\end{cases} \quad \text{and} \quad p_j^\alpha=\begin{cases}1, & j=0,\\ \left(1-\dfrac{\alpha+1}{j}\right)p_{j-1}^\alpha, & j\geq 1.\end{cases}$$

By substituting the equation (5) in (4) the following relation is obtained:

$$\frac{\omega^\alpha(0)}{\tau^\alpha}v^{n+1}-\kappa\Delta v^{n+1} = -\sum_{j=1}^{n}\frac{\omega^\alpha(j)}{\tau^\alpha}v^{n+1-j}+\frac{t^{-\alpha}}{\Gamma(1-\alpha)}v^0+f^{n+1}, \tag{6}$$

with $f^{n+1}=f(\mathbf{x},t^{n+1})$ and $v^{n+1-j}=v(\mathbf{x},t^{n+1-j})$ $(j=0,\ldots,n+1)$.

*2.2 The meshless localized collocation method*

This section describes the meshless localized collocation approach. This choice is because, in last decades, meshless methods have been employed successfully in several fields of science and engineering [4] and allow to avoid the expensive task related to the mesh construction. In the meshless localized collocation method, the global domain $\Omega$ is partitioned into local sub-domains $\Omega_i$ $(i=1,\ldots,N)$ corresponding to every point. These sub-domains ordinarily are circles or squares and cover the entire global domain $\Omega$. Then the radial point interpolation shape functions, $\phi_i$, are constructed locally over each $\Omega_i$ by combining radial basis functions and the monomial basis function [10] corresponding to each local field point $\mathbf{x}_i$. In the current work, it is used one of the most popular RBFs, i.e., the generalized multiquadric radial basis function (GMQ-RBF) $\phi(r)=(r^2+c^2)^q$ (q=2.5) where $c$ is the shape parameter. The local radial point interpolation shape function generates the $N\times N$ sparse matrix $\Phi$. Therefore $v$ can be approximated by:

$$v(\mathbf{x})=\sum_{i=1}^{N}\phi_i(\mathbf{x})v_i \tag{7}$$

where $\phi_i(\mathbf{x})=\phi(\|\mathbf{x}-\mathbf{x}_i\|_2)$ (the norm $\|\mathbf{x}-\mathbf{x}_i\|_2$ denotes the Euclidean distance between $\mathbf{x}$ and field point $\mathbf{x}_i$). Substituting formula (7) in equations (6), (2) and (3) yields:

$$\sum_{i=1}^{N}\left[\frac{\omega^\alpha(0)}{\tau^\alpha}\phi_i(\mathbf{x}_j)-\kappa\left[\frac{\partial^2\phi_i}{\partial x^2}+\frac{\partial^2\phi_i}{\partial y^2}\right](\mathbf{x}_j)\right]v_i^{n+1} = -\sum_{j=1}^{n}\frac{\omega^\alpha(j)}{\tau^\alpha}\sum_{i=1}^{N}\phi_i(\mathbf{x}_j)v_i^{n+1-j}$$

$$+\frac{t^{-\alpha}}{\Gamma(1-\alpha)}\sum_{i=1}^{N}\phi_i(\mathbf{x}_j)v^0+f^{n+1}, \qquad j=1,\ldots,N_\Omega \tag{8}$$

$$\sum_{i=1}^{N}\phi_i(\mathbf{x}_j)v_i^{n+1}=\psi_1^{n+1}(\mathbf{x}_j), \qquad j=N_\Omega+1,\ldots,N_\Omega+N_{\Gamma_1}, \tag{9}$$

$$\sum_{i=1}^{N} \phi_i(\mathbf{x}_j) v_i^{n+1} = \psi_2^{n+1}(\mathbf{x}_j)\rho^{n+1}, \quad j = N_\Omega + N_{\Gamma_1} + 1, \ldots, N_\Omega + N_{\Gamma_1} + N_{\Gamma_2}, \tag{10}$$

$$\sum_{i=1}^{N} \left( \int_\Omega \phi_i(\mathbf{x}) d\Omega \right) v_i^{n+1} = h^{n+1}. \tag{11}$$

The collocation equations (8) are referred to the $N_\Omega$ interior points in $\Omega$, while the $N_{\Gamma_1}$ equations (9) and the $N_{\Gamma_2}$ equations (10) (involving also the unknown $\rho^{n+1} = \rho(t^{n+1})$) arise from the initial and Dirichlet boundary conditions. Finally, a further equation is obtained by means of 2D Gaussian-Legendre quadrature rules of order 15. Therefore, the time discretization approximation and the local collocation strategy construct a linear system of $N+1$ linear equations with $N+1$ unknown coefficients ($N = N_\Omega + N_{\Gamma_1} + N_{\Gamma_2}$). The unknown coefficients $\mathbf{v}^{(n+1)} = (v_1^{n+1}, \ldots, v_N^{n+1}, \rho^{n+1})$ are obtained by solving the sparse linear system:

$$\mathscr{A}\mathbf{v}^{(n+1)} = \mathscr{B}^{(n+1)}, \tag{12}$$

where $\mathscr{A}$ is a $(N+1)\times(N+1)$ coefficient matrix and $\mathscr{B}^{(n+1)}$ is a $(N+1)$ vector. Let us notice that, unlike $\mathscr{B}^{(n+1)}$, the coefficient matrix $\mathscr{A}$ does not change its entries along time steps. Moreover, due to the local approach the coefficient matrix $\mathscr{A}$ is sparse.
Previous discussion can be summarized through the following scheme, Algorithm 1, which describes the main steps of the numerical procedure.

**Algorithm 1** Pseudo-code for problem (1)

`Input:` $\kappa, \alpha, T, \tau, \varphi, \Psi_1, \Psi_2, h,$
$\{\mathbf{x}_i\}_{i=1}^{N_\Omega}$, % interior points
$\{\mathbf{x}_{i+N_\Omega}\}_{i=1}^{N_{\Gamma_1}}$ % $\Gamma_1$ boundary points
$\{\mathbf{x}_{i+N_\Omega+N_{\Gamma_1}}\}_{i=1}^{N_{\Gamma_2}}$ % $\Gamma_2$ boundary points

`Output:` $\left\{\{v_i^{n+1}\}_{i=1}^{N}\right\}_{n=0}^{T/\tau-1}$, $\{\rho^{n+1}\}_{n=0}^{T/\tau-1}$

1: `build` $\mathbf{A}$ *% by following (8,9,10,11)*
2: `for` $n = 0, 1, \ldots, T/\tau - 1$ *% loop on time slices*
3: `build` $\mathbf{B}^{(\mathbf{n+1})}$ *% by following (8,9,10,11)*
4: `compute` $\mathbf{v}^{(n+1)}$ : *% solution of* $\mathscr{A}\mathbf{v}^{(n+1)} = \mathscr{B}^{(n+1)}$ *in (12)*
5: `endfor`

## 3. Sequential and parallel implementation

In order to implement the Algorithm 1, we need to define: a 2D regular grid, called `CenterPoints` and the sub-sets `center_I`, `center_b1` and `center_b2`, the interior points and the boundary points of the `CenterPoints`, respectively. Therefore, we find for each fixed interior point ($i = 1, \ldots,$size(`center_I`)) its local neighbors and, by eval-

uating the Laplacian of the local RBF interpolating function, we build the $i$-th row of `A_total` (i.e. $\mathbf{A}$). We highlight that this step requires to solve multiple linear systems of small size (number of neighbors) for each point in `center_I`. Thus we build next size(`center_b1`) + size(`center_b2`) rows of `A_total` (by using (9) and (10)) and we build last row of `A_total` by evaluating the integral in (3) by means of 2D Gaussian-Legendre quadrature rules of order 15. Finally, after, a discrete 1D time interval `tt`$= [0 : \tau : T]$, with step $\tau$, generation, for each time in `tt`, we build the right-hand side vector `B` (i.e. $\mathbf{B}^{(\mathbf{n+1})}$) and we solve the sparse linear system `A_total` $\cdot$ `sol = B`, where `sol` is the computed value of $\mathbf{v}^{(\mathbf{n+1})}$.
Algorithm 2 illustrates the necessary steps in detail. Observe that, our sequential implementation provides the multiple linear systems solution, at lines 9, 10 and 11 by using the routine `dgesv` of the LAPACK library based on the LU factorization method, while to solve the sparse linear system, at line 16, a specific routine of the CSPARSE library is employed [14], i.e. the `cs_lusol` routine, typical for linear systems characterized by sparse coefficient matrices.

**Algorithm 2** Sequential algorithm

```
1: STEP 0: input phase
2: generate CenterPoints
3: find Center_I        % interior points
4: find Center_b1       % boundary points
5: find Center_b2       % boundary points
6: STEP 1: construction of the coefficient matrix
7: for each point of CenterPoints
8:     find its local neighbors
9:     solve multiple linear systems     % one for each point of center_I
10:    solve multiple linear systems     % one for each point of center_b1
11:    solve multiple linear systems     % one for each point of center_b2
12: endfor
13: STEP 2: loop on time
14: for n = 0; n < T/τ; step = 1 do
15:    build B      % (by using results of lines 8,9,10,11)
16:    solve A_total · sol = B
17:    set sol_M[n+1]:=sol
18: end for
19: STEP 3: output phase and condition number evaluation
20: reshape matrix sol_M
```

Starting from some preliminary and interesting results obtained in [13], where a multicore strategy was used, here we propose a different parallel approach that exploits the powerful of modern GPU architectures. This new implementation comes from the idea of increasing the threads number (using all cores available on the GPU architecture) in order to observe the optimal gain obtained by using last generation parallel machines.

Firstly, to achieve a satisfying execution of our code (in terms of execution time but, above all, to ensure that the software reaches the highest performance) we car-

ried out an ad-hoc configuration of the CUDA environment, because for large input a high number of bytes are needed to be allocated with respect to the number of operations. So, for each thread, the local stack and heap size have increased by using the `cudaThreadSetLimit(op, size)` routine and by setting parameters:

– `op:=cudaLimitMallocHeapSize`
– *size:=1024*1024*1024*.

This routine has to be called before the CUDA environment starts. In this way, we tried to overcome, as possible, the well-known problem of the limited memory inherent to the GPU architectures. Moreover, with this arrangement it is possible to allocate memory dynamically on a GPU and, as explained later, to reduce the transfer of host-device data. About the decomposition approach, we combine the classical *domain* decomposition with a more sophisticated *functional* decomposition, following this schema: each sub-domain of work is demanded to a set of threads, using one-dimensional blocks and grid, if the chosen number of threads is less or equal than 1024; otherwise the blocks and the grid are set as two-dimensional structures. Therefore, following the domain decomposition approach, each thread is linked at a single input point and it performs all the operations needed to build the final sparse matrix; while in accordance with the *functional* decomposition, a pool of threads, based on a fixed configuration, works on different tasks of the overall algorithm in a parallel way.
More in detail, we describe our GPU-parallel approach STEP by STEP referring to the serial version showed in Algorithm 2.

The input phase, in STEP 0, uses a domain decomposition-based parallelization strategy. The local neighbors are defined by considering a sub-set of the `CenterPoint` set. In particular, for each point the thread associated with it builds the sub-domains corresponding to the *inner* points and the *boundary* points. In this way, the construction of local structures becomes very simple and faster.
In STEP 1 the build of the sparse matrix coefficient is designed following the domain decomposition criterion and the functional decomposition approach: for each sub-domain of the `CenterPoint` set the local neighbors are found, as in the STEP 0, therefore, each thread (linked at one input point) builds the local corresponding multiple linear systems whose solutions will be computed by a pull of threads, asynchronously, and then collected in the global matrix `A_total`.

In order to avoid the copy overhead, a suitable workload distribution is performed by using Algorithm 3.

**Algorithm 3** Compute global matrix for each thread

```
1: index = threadIdx.x + (blockDim.x * blockIdx.x) % thread index definition
2: g_Mats[], g_Terms[] % allocation on the host
3: FOR ALL thread
4:     g_Mats[index] % local matrix building
5:     g_Terms[index] % local note terms computation
6: ENDFOR ALL
```

Observe that, at lines 4, 5, 6 the values are computed and stored in the local memory of each thread in a one-dimensional array following the *row-major* order method.
Now, to solve the local multiple linear systems a specific technique has been implemented, because CUDA are not able to directly call the routines of the cuSOLVER library from the device. More precisely, firstly the local matrices are transferred on the host and collected in a global array whose the rows size is equal to the thread number. In this array each portion, related to a single thread, contains the local matrix. A similar approach is used to build and store the right-hand side vectors of each linear system. Therefore, from the host each local portion of the global matrices `g_Mats` and `g_Terms` are used as parameters for the routine `cusolverDnDgetrs` of the cuSOLVER-Dense library, called to solve each linear system, as shown in Algorithm 4.

**Algorithm 4** Solving local Linear Systems - Host code

1: start cuSOLVER & cuBLAS environments
2: `index = threadIdx.x + (blockDim.x * blockIdx.x)` *% thread index definition*
3: FOR ALL thread-pull
4: call cuSOLVER routine with `g_Mats[index]` and `g_Terms[index]`
5: `__syncthreads()`
6: copy result in the `A_total` sparse matrix
7: ENDFOR ALL

The solutions, returned by the cuSOLVER routine are inserted in the `A_total` sparse matrix. During this phase we perform a synchronization by using the `__syncthreads()` routine, in order to avoid any memory contention. The final STEP 2 provides the sparse linear system solution. It is executed in a similar way to what described in the STEP 1, i.e. following the scheme showed in Algorithm 3 to manage the host-device data transfer. For this last STEP only the domain decomposition approach has been used. To be specific, the *loop for*, related to time discretization, runs in parallel on `T` threads (corresponding to the time interval size). However, before to compute this final STEP, a copy of the `A_total` sparse matrix is stored in the local memory of each thread using the CSR format, [16]. Everything described is shown in detail in Algorithm 5. More in details, each thread builds the local `B` vector by using the `cublasDgemv()` routine of the cuBLAS library [15], for solving a matrix-vector multiplication, at line 6. Hence, the main sparse linear system is solved by the host with the `cusolverSpDcsrlsvlu()` routine of the cuSOLVER library, at each time step.

## 4. Numerical tests

The GPU-parallel algorithm, described in the previous section, has been implemented on a computer machine with the following technical specifications:

- two CPU Intel Xeon with 6 cores, E5-2609v3, 1.9 Ghz, 32GB of RAM, 4 channels 51Gb/s memory bandwidth
- two NVIDIA GeForce GTX TITAN X, 3072 CUDA cores, 1 Ghz Core clock for core, 12 GB DDR5, 336 GBs as bandwidth

**Algorithm 5** Solving the sparse linear system

```
start a CUDA pool of T threads
on the device:
index = threadIdx.x + (blockDim.x * blockIdx.x) % compute index thread
allocation g_B[] % contains i-th B vector
FOR ALL thread
    build B
    g_B[i_th] = B
ENDFOR ALL
return to host:
for n = 0; n < T; step = 1/τ do
      solve A_total · sol = g_B[n]
      set sol_M[n+1]:=sol
end for
```

In the following we show some numerical tests in order to highlight the performance gain in terms of execution times and memory occupancy. In Table 1 some executions of our software are shown, by varying both the input size and the threads configuration. Time values are obtained by averaging ten test runs for all each considered case.

**Table 1.** Execution times in seconds (s) achieved, by varying the CUDA configuration: block $\times$ threads and the problem input size.

| input size | serial times | $1 \times 256$ | $1 \times 512$ | $1 \times 1024$ | $2 \times 256$ | $2 \times 512$ | $2 \times 1024$ |
|---|---|---|---|---|---|---|---|
| $3.6 \times 10^3$ | $1.6 \times 10^3$ | 5.48 | 1.40 | 2.30 | 3.66 | 4.32 | 7.50 |
| $8.1 \times 10^3$ | $1.2 \times 10^4$ | 7.60 | 1.86 | 3.45 | 5.42 | 6.72 | 13.22 |
| $1.0 \times 10^4$ | $3.19 \times 10^4$ | 13.30 | 3.60 | 4.56 | 7.88 | 8.24 | 18.77 |
| $1.69 \times 10^4$ | $1.66 \times 10^5$ | 17.25 | 5.20 | 6.28 | 10.62 | 12.32 | 23.00 |
| $1.98 \times 10^4$ | $2.46 \times 10^5$ | 19.89 | 8.12 | 9.10 | 13.4 | 16.20 | 27.82 |
| $2.25 \times 10^4$ | $3.9 \times 10^5$ | 26.80 | 8.12 | 14.22 | 17.65 | 21.20 | 32.10 |

As we can see, a significant gain with respect to the serial version is achieved. This is essentially due to a suitable choice for the memory setup and to the adaptive combination of the different parallel strategies considered. As illustrated in the previous section, these options allow us a good workload balance. More in details, the execution time of the sequential algorithm grows considerably as the input point number increases. However a noticeable speed up is observed in the GPU-parallel version. To be specific, the better execution is reached by using $1 \times 512$ threads. A probable simple reason which explains this result should be that for our graphic card, according the rules for a correct configuration of the CUDA environment (that depend on the relationship between the maximum size of the allocable constant memory and the number of CUDA cores per multiprocessor) the optimal number of threads per block is 512. Moreover, we highlight that, for all last executions where the input is very large, the execution time increases, because of the high bandwidth required during the computation. In fact, using a large number of threads execution times grow for the spurious threads given but not used for the computation. We just observe this phenomenon looking at the different ways of increasing the times by comparing the second last column and the last one.

In Table 2 the execution times, for each single kernel by varying the input size and fixing as CUDA configuration: $1 \times 512$, are shown. As expected the most expensive kernel, in terms of execution time, is the `cusolverDnDgetrs( )` routine, which is the kernel used to solve the multiple linear system needed to build the sparse matrix `A_total`. This task, despite the parallel decomposition/functional approach combination, remains the most expensive in the computation.

**Table 2.** Time execution analysis for each CUDA Kernel: the first lines corresponding to the execution times of computational kernels are expressed in milliseconds, while the execution times of transferred data (the last two lines) are reported in microseconds.

| **cudaKernel** | $3.6 \times 10^3$ | $8.1 \times 10^3$ | $1.0 \times 10^4$ | $1.69 \times 10^4$ | $1.98 \times 10^4$ | $2.25 \times 10^4$ |
|---|---|---|---|---|---|---|
| firstKernel() | 394.12 | 938.03 | 1095.78 | 1860.17 | 2167.66 | 2634.25 |
| computeMatrix() | 9.47 | 22.40 | 27.34 | 44.42 | 52.09 | 59.18 |
| computeCenterb1b2 | 4.22 | 9.45 | 11.72 | 21.81 | 24.85 | 28.37 |
| computeCenterI | 3.04 | 6.85 | 9.53 | 14.77 | 17.82 | 19.02 |
| cusolverDnDgetrs | 962.32 | 1132.71 | 2306.38 | 2910.57 | 3729.67 | 5305.31 |
| cublasDgemv() | 1.89 | 4.95 | 5.25 | 8.87 | 11.42 | 13.81 |
| cusolverSpDcsrlsvlu | 205.81 | 464.50 | 571.69 | 966.14 | 1132.28 | 1328.66 |
| CUDA memcpy HtoD | 808.33 | 2117.74 | 2245.36 | 3814.86 | 4142.87 | 5082.12 |
| CUDA memcpy DtoH | 640.22 | 1667.41 | 1887.38 | 2809.85 | 3151.21 | 4002.37 |

Time values in previous table positively confirm the efficiency of the proposed software by showing the low weight of host-device-host communications with respect to the computational workload. Conclusive considerations on the performance of our algorithm can be made by analyzing Table 3.

**Table 3.** Memory usage in MBytes (MB) - configuration blocks/threads = $1 \times 512$.

| **N** | **MB** |
|---|---|
| $3.6 \times 10^3$ | 1165 |
| $8.1 \times 10^3$ | 1183 |
| $1.0 \times 10^4$ | 1215 |

| **N** | **MB** |
|---|---|
| $1.69 \times 10^4$ | 1251 |
| $1.98 \times 10^4$ | 1332 |
| $2.25 \times 10^4$ | 1491 |

From this table, we deduce that the GPU version allows us to obtain a very good memory occupation, typical of the use of CUDA architectures. In fact, the low use of global CUDA memory (for our graphic card maximum 12213 MBytes) is due to the high utilization of local threads memory, which in addition to reducing the time of device-host-device copy limits the use of the global memory, stemming all the work in the local thread work-space.

## 5. Conclusions

In this paper, we proposed a GPU-parallel algorithm to solve a two-dimensional inverse time fractional diffusion equation. The algorithm implements a numerical procedure based on the discretization of the Caputo fractional derivative and on the use of a meshless localized collocation method exploiting the radial basis functions properties.

The parallel approach, based our CUDA-kernels efficient implementation and on a reliable use of ad-hoc parallel numerical libraries available on CUDA, provides a significant performance gain in terms of execution times and memory occupancy.

## Acknowledgement

This paper has been supported by project *Algoritmi innovativi per interpolazione, approssimazione e quadratura* (AIIAQ) and project *Algoritmi numerici e software per il trattamento di dati su larga scala in ambienti HPC* (LSDAHPC).

## References

[1] Mohebbi, Akbar, Mostafa Abbaszadeh, and Mehdi Dehghan. "The use of a meshless technique based on collocation and radial basis functions for solving the time fractional nonlinear Schrödinger equation arising in quantum mechanics." Engineering Analysis with Boundary Elements 37.2 (2013): 475-485.

[2] Piret, CéCile, and Emmanuel Hanert. "A radial basis functions method for fractional diffusion equations." Journal of Computational Physics 238 (2013): 71-81.

[3] Aslefallah, Mohammad, and Elyas Shivanian. "Nonlinear fractional integro-differential reaction-diffusion equation via radial basis functions." The European Physical Journal Plus 130.3 (2015): 47.

[4] Cuomo, S., Galletti, A., Giunta, G., Marcellino, L., Reconstruction of implicit curves and surfaces via RBF interpolation (2017) Applied Numerical Mathematics, 116, pp. 157-171. DOI: 10.1016/j.apnum.2016.10.016

[5] Fasshauer, Gregory E. Meshfree approximation methods with MATLAB. Vol. 6. World Scientific, 2007.

[6] https://developer.nvidia.com/cuda-gpus

[7] https://docs.nvidia.com/cuda/cusolver/index.html

[8] L. Yan, F. Yang, The method of approximate particular solutions for the time-fractional diffusion equation with a non-local boundary condition, Computers and Mathematics with Applications, 70 (2015) 254-264.

[9] S. Abbasbandy, H. R. Ghehsareh, M. S. Alhuthali, H. H. Alsulami, Comparison of meshless local weak and strong forms based on particular solutions for a non-classical 2-D diffusion model, Engineering Analysis with Boundary Elements, 39 (2014) 121-128.

[10] Q. Liu, Y. T. Gu, P. Zhuang, F. Liu, Y. F. Nie, An implicit RBF meshless approach for time fractional diffusion equations, Comput Mech, 48 (2011) 1-12.

[11] W.Y. Tian, H. Zhou, W.H. Deng, A class of second order difference approximations for solving space fractional diffusion equations, Mathematics of Computation, 84 (2015) 1703-1727.

[12] A. Kilbas, M.H. Srivastava, J.J. Trujillo, Theory and Application of Fractional Differential Equations, North Holland Mathematics Studies, 204 (2006).

[13] De Luca, P., Galletti, A., Giunta G., Marcellino, L., Raei, M. Performance analysis of a multicore implementation for solving a two-dimensional inverse anomalous diffusion problem (2019) Lecture Notes in Computer Science - Proceedings of NUMTA2019, THE 3RD INTERNATIONAL CONFERENCE AND SUMMER SCHOOL

[14] http://faculty.cse.tamu.edu/davis/suitesparse.html

[15] https://docs.nvidia.com/cuda/cublas/index.html

[16] https://docs.nvidia.com/cuda/cusparse/index.html#csr-format