mathematics

MDPI

Article

An Accelerating Numerical Computation of the Diffusion Term in a Nonlocal Reaction-Diffusion Equation

Mitică CRAUS and Silviu-Dumitru PAVĂL *

Faculty of Automatic Control and Computer Engineering, Technical University "Gh. Asachi", Dimitrie Mangeron, 27, 700050 Iași, Romania; craus@tuiasi.ro
* Correspondence: silviu.paval@tuiasi.ro

Received: 9 November 2020; Accepted: 23 November 2020; Published: 26 November 2020

Abstract: In this paper we propose and compare two methods to optimize the numerical computations for the diffusion term in a nonlocal formulation for a reaction-diffusion equation. The diffusion term is particularly computationally intensive due to the integral formulation, and thus finding a better way of computing its numerical approximation could be of interest, given that the numerical analysis usually takes place on large input domains having more than one dimension. After introducing the general reaction-diffusion model, we discuss a numerical approximation scheme for the diffusion term, based on a finite difference method. In the next sections we propose two algorithms to solve the numerical approximation scheme, focusing on finding a way to improve the time performance. While the first algorithm (sequential) is used as a baseline for performance measurement, the second algorithm (parallel) is implemented using two different memory-sharing parallelization technologies: Open Multi-Processing (OpenMP) and CUDA. All the results were obtained by using the model in image processing applications such as image restoration and segmentation.

Keywords: nonlocal reaction-diffusion equation; parallel processing; image processing

1. Introduction

Image processing tasks such as noise filtering, image deblurring and shape extraction (image segmentation) are often based on partial differential equation (PDE) models; see, e.g., [1–3].

After Perona and Malik introduced an anisotropic diffusion model for image denoising [2], their PDE-based model became a the starting point for numerous other derived models [3] which aim to improve it. The nonlocal diffusion model is one of these derived models which in cases of image restoration tasks aims to eliminate image blurring while preserving the edges and avoiding staircase effect at the same time.

To introduce the nonlocal form of the reaction-diffusion equation, we consider (1) where $v(t,x)$ is the unknown function to be solved.

$$\begin{aligned}\frac{\partial}{\partial t}v(t,x) = \int_{\Omega} K(x-y)\Big[v(t,y)-v(t,x)\Big]dy + \int_{\partial\Omega} K(x-y_s)w(t,y_s)dy_s \\ +p_1\nabla v(t,x) + p_2\left[v(t,x)-v^3(t,x)\right] + f(t,x)\end{aligned} \tag{1}$$

having the initial condition

$$v(0,x) = v_0(x), \tag{2}$$

The mathematical notations used in (1) and (2) have the following meanings:

- $T > 0$ is the upper limit of time and Ω is the spatial domain in $\mathbb{R}^2$, whose boundary $\partial\Omega$ is smooth. The image being analyzed forms the input domain, Ω, and time is a synthetic variable used to describe the iterative process;
- $x, y \in \Omega$, $y_s \in \partial\Omega$ and $t \in (0, T]$;
- $v(t, x)$ is the unknown function used to model the evolution of the image in time (after each processing step). $v_0(x)$ is the initial data which in order to make possible the evolution of the model in time are set to be equal to the pixel intensities of the input image that is analyzed;
- $\frac{\partial}{\partial t} v(t, x)$ is the partial derivative of $v(t, x)$ with respect to t;
- $K : \mathbb{R}^2 \to \mathbb{R}$ is a continuous nonnegative and symmetric real function, compactly supported in the unit ball, and such that $\int\limits_{\mathbb{R}^2} J(z)dz = 1$;
- $w(t, y_s)$ describe the boundary control conditions;
- $f(t, x)$ represents an external force for the system;
- p_1 and p_2 are model parameters, their values being tuned accordingly with the task (image restoration or segmentation).

Equations (1) and (2) were qualitatively and quantitatively analyzed by T. Barbu, A. Miranville and C. Moroşanu in [1,4]; well-posedness and numerical convergence were covered in [4].

Applications to image restoration were well studied in [1], wherein a local form of a newly introduced anisotropic diffusion system was used. This model can also be seen applied to image processing and other domains in [2,3,5,6]. In [7] we described a numerical approximation for a similar model applied to a 3D input domain but having fewer experimental results.

In this paper we are working on providing a numerical approximation scheme for the diffusion term and analyzing several implementation methods to achieve better time performance for the computations required by the numerical approximation.

We are focusing on studying the nonlocal diffusion term, because in our experiments we noticed it is the one that has the biggest impact on the time complexity of the entire model. We will denote the nonlocal diffusion term as ND, and its expression is given by (3).

$$ND(x, t, v) = \int\limits_{\Omega} K(x - y)\left[v(t, y) - v(t, x)\right]dy + \int\limits_{\partial\Omega} K(x - y_s)w(t, y_s)dy_s \tag{3}$$

From the computational point of view, the ND term will most often lead to intense computational requirements given the two integrals that need to be evaluated and that usually, in practical applications, Ω is a large domain.

2. Numerical Approximation

To numerically evaluate the expression in (3) we are going to use a finite difference method as described in [8]. In our evaluation we will also use the initial conditions set in (2).

The space domain, Ω, will be considered to be a rectangle starting at origin having a length L and width W. Given our specific application to image processing, the space domain represents the dimensions of the image that would be analyzed. The time domain will be set to an interval such as $[0, T]$. Setting the time and space domains to start at 0 is not limiting the theoretical nor experimental results.

We will divide the space domain (Ω) into a grid with step h, transforming it into a discrete domain Ω_h containing the elements $x_{i,j} = (ih, jh)$ where $0 \le i \le I$ and $0 \le j \le J$ with $I = W/h$ and $J = L/h$.

In a similar manner, the time domain ($[0, T]$) will be divided by a time step ϵ, transforming it into a discrete time interval consisting of moments given by: $t_m = m\epsilon$ where $0 \le m \le M$ with $M = T/\epsilon$.

By setting a specific time moment $t_m \in [0, T]$ and a space point $x_{i,j} \in \Omega_h$, (3) becomes (4).

$$ND^{m+1}(x_{i,j}, t_m, v^m) = \int\limits_{\Omega_h} K(x_{i,j} - y)\left[v^m(t_m, y) - v^m(t_m, x_{i,j})\right]dy + \int\limits_{\partial\Omega_h} K(x_{i,j} - y_s)w^m(t_m, y_s)dy_s \tag{4}$$

where v computed at moment t_m is denoted by v^m and v^0 is set by initial condition in (2).

We will also denote $v^m_{i,j}$ by $v(t_m, x_{i,j})$ and consider the first integral in (4) as a double integral over rectangle domain Ω_h. By using Riemann sums to approximate the integrals, we get the approximating expressions for the two integral terms in (4), as seen in (5) and (6).

$$\begin{aligned}
&\int\limits_{\Omega_h} K(x_{i,j} - y)\Big[v^m(t_m, y) - v^m(t, x_{i,j})\Big]dy = \\
&\quad h^2 \Bigg\{ \sum_{d_1=1}^{I-1}\sum_{d_2=1}^{J-1} K(x_{i,j} - y_{d_1,d_2})\Big(v^m_{d_1,d_2} - v^m_{i,j}\Big) \\
&\quad + \tfrac{1}{2}\sum_{d_1=1}^{I-1}\Big[K(x_{i,j} - y_{d_1,0})\Big(v^m_{d_1,0} - v^m_{i,j}\Big) + K(x_{i,j} - y_{d_1,J})\Big(v^m_{d_1,J} - v^m_{i,j}\Big)\Big] \\
&\quad + \tfrac{1}{2}\sum_{d_2=1}^{J-1}\Big[K(x_{i,j} - y_{0,d_2})\Big(v^m_{0,d_2} - v^m_{i,j}\Big) + K(x_{i,j} - y_{I,d_2})\Big(v^m_{I,d_2} - v^m_{i,j}\Big)\Big] \\
&\quad + \tfrac{1}{4}\Big[K(x_{i,j} - y_{0,0})\Big(v^m_{0,0} - v^m_{i,j}\Big) + K(x_{i,j} - y_{I,0})\Big(v^m_{I,0} - v^m_{i,j}\Big) \\
&\qquad + K(x_{i,j} - y_{0,J})\Big(v^m_{0,J} - v^m_{i,j}\Big) + K(x_{i,j} - y_{I,J})\Big(v^m_{I,J} - v^m_{i,j}\Big)\Big]\Bigg\}
\end{aligned} \tag{5}$$

The second integral term is computed along the frontier of Ω_h.

$$\begin{aligned}
&\int\limits_{\partial\Omega_h} K(x_{i,j} - y_s)w(t_m, y_s)dy_s = \\
&h\Bigg\{\sum_{d_1=1}^{I-1}\Big[K(x_{i,j} - y_{d_1,0})w^m_{d_1,0} + K(x_{i,j} - y_{d_1,J})w^m_{d_1,J}\Big] \\
&+ \sum_{d_2=1}^{J-1}\Big[K(x_{i,j} - y_{0,d_2})w^m_{0,d_2} + K(x_{i,j} - y_{I,d_2})w^m_{I,d_2}\Big] \\
&+ K(x_{i,j} - y_{0,0})w^m_{0,0} + K(x_{i,j} - y_{I,0})w^m_{I,0} + K(x_{i,j} - y_{0,J})w^m_{0,J} + K(x_{i,j} - y_{I,J})w^m_{I,J}\Bigg\}
\end{aligned} \tag{6}$$

The integral in (5) is computed as a sum of volumes generated around each Ω_h node (see Figure 1) by multiplying the surface assigned to the respective node with integrated function's value at the node (see Figure 2). At the edges of Ω_h the nodes are assigned halves or quarters (for corner nodes) of the area which is regularly assigned to an interior node.

The integral on Ω's boundary in (6) is computed as a sum of areas generated around each Ω_h frontier node by multiplying the segment (of length h) assigned to the respective frontier node with the integrated function's value at the frontier node. All nodes on Ω's frontier will be matched with segments of same length, h.

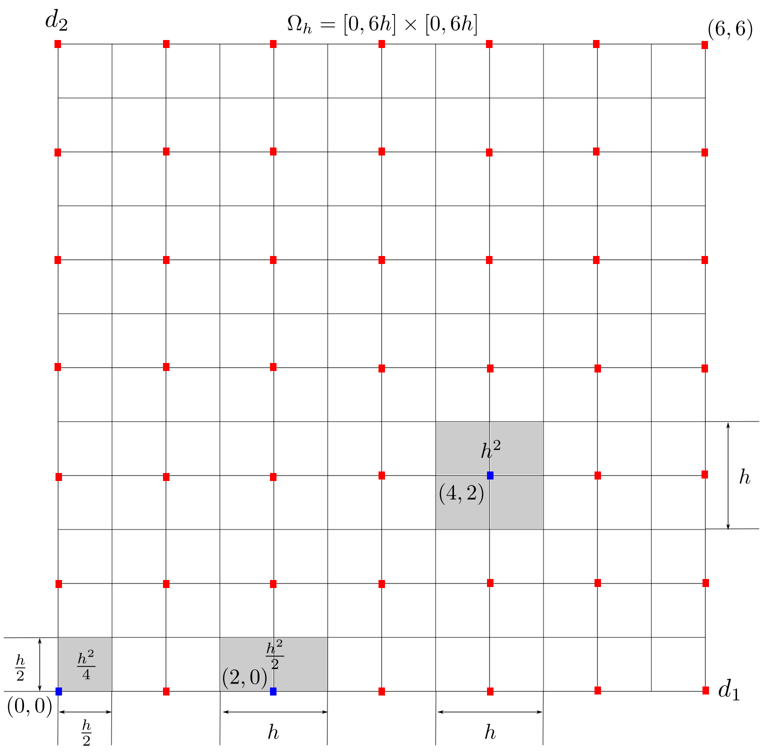

Figure 1. Example of 2D Ω discretization using a 6×6 grid with step h.

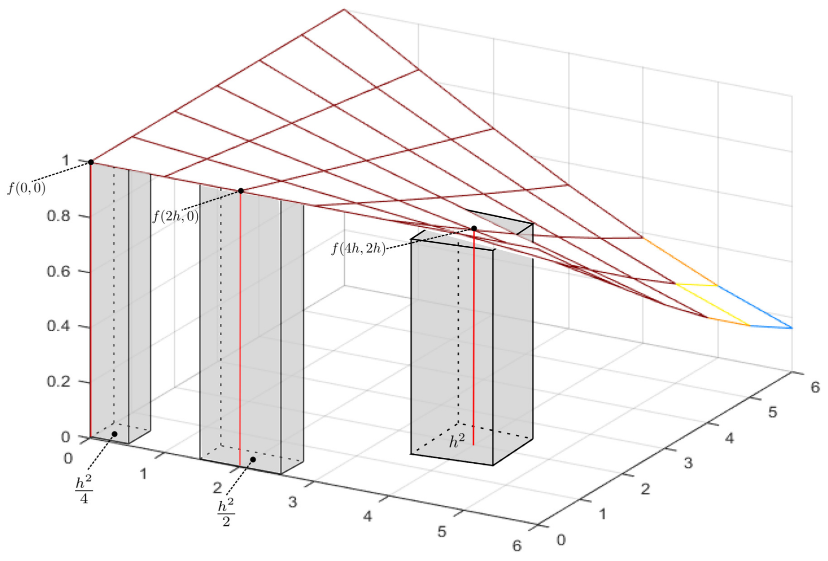

Figure 2. Computing the integral term in (5) as a sum of volumes.

We used the following kernel function for all our experiments: $K_\sigma(x, y) = \frac{1}{\sigma} e^{-\frac{x^2+y^2}{\sigma^2}}$, which is symmetric and even on $\mathbb{R}^2$.

3. Algorithms and Implementations

We propose two methods for accelerating the computations, first by using parallel computation on multiple CPUs, and secondly by using dedicated hardware, in this case nVidia GPUs, to parallelize the computations. We also introduce a variation for all computation methods derived from an observation made about the nature of the kernel function K.

The sequential algorithm which will be used as a basis to develop the parallel versions is detailed in Algorithm 1.

Algorithm 1: Nonlocal diffusion sequential algorithm.

Input:

- $v = \{v_{i,j} |\, 0 \le i \le I,\ 0 \le j \le J\}$ are the values of unknown function v for a specific time moment, for first iteration the values will be set to intensities of the pixels in the image to be analyzed
- $w = \{w_{i,j} |\, 0 \le i \le I,\ 0 \le j \le J\}$ are the values of control function w at a given time step, it will be set only for the pixels belonging to Ω's frontier
- $I + 1$ is the number of pixels on width (d_1 in Figure 1)
- $J + 1$ is the number of pixels on height (d_2 in Figure 1)
- h is the grid step

Output:

- nd is a $(I + 1) \times (J + 1)$ matrix containing ND values computed for a specific time step (4)

```
1  function NDSequential($v, w, I, J, h$)
2      for $i = 0$ to $I$ do
3          for $j = 0$ to $J$ do
4              for $d_1 = 0$ to $I$ do
5                  for $d_2 = 0$ to $J$ do
6                      $kval \leftarrow K((i - d_1) * h, (j - d_2) * h)$
7                      $vdiff \leftarrow v[d_1][d_2] - v[i][j]$
8                      if $(d_1, d_2)$ is internal pixel then
9                          $nd[i][j] \leftarrow nd[i][j] + h^2 * kval * vdiff$
10                     else if $(d_1, d_2)$ is edge pixel then
11                         $nd[i][j] \leftarrow nd[i][j] + \frac{h^2}{2} * kval * (vdiff + w[d_1][d_2] * \frac{2}{h})$
12                     else if $(d_1, d_2)$ is corner pixel then
13                         $nd[i][j] \leftarrow nd[i][j] + \frac{h^2}{4} * kval * (vdiff + w[d_1][d_2] * \frac{4}{h})$
14     return $nd$
```

In the first two loops (lines 2 and 3), the algorithm will go over each pixel (i, j) in Ω_h and compute the sum of volumes over entire input space. As (i, j) changes, the function which is integrated changes as well; this is due to the $x_{i,j}$ parameter under the integrals. The volumes which will be summed over Ω_h will have different values depending on the current (i, j) pair.

The sum of volumes itself is a result of the two inner loops (lines 4 and 5). We need to set apart the volumes for all internal pixels from the volumes of pixels found on the image margins, which are computed differently—take into the account the smaller area assigned to them and the conditions at the frontier represented by the control function $w(t, x)$.

The sequential algorithm has a time complexity order of $\mathcal{O}(I^2 J^2)$; if we consider the image which is analyzed to be of a square form (where $I = J = N$), the time complexity order would be $\mathcal{O}(N^4)$.

The parallel algorithm is obtained by scattering the iterations on lines 2 and 3 in Algorithm 1 between all the processors (see Algorithm 2).

Algorithm 2: Nonlocal diffusion parallel algorithm.

Input:

- v, w, I, J, h: see Algorithm 1

Output:

- nd: see Algorithm 1

```
function NDParallel($v, w, I, J, h$)
    for each pixel $(i, j)$ in $v$ parallel do
        for $d_1 = 0$ to $I$ do
            for $d_2 = 0$ to $J$ do
                $kval \leftarrow K((i - d_1) * h, (j - d_2) * h)$
                $vdiff \leftarrow v[d_1][d_2] - v[i][j]$
                if $(d_1, d_2)$ is internal pixel then
                    $nd[i][j] \leftarrow nd[i][j] + h^2 * kval * vdiff$
                else if $(d_1, d_2)$ is edge pixel then
                    $nd[i][j] \leftarrow nd[i][j] + \frac{h^2}{2} * kval * (vdiff + w[d_1][d_2] * \frac{2}{h})$
                else if $(d_1, d_2)$ is corner pixel then
                    $nd[i][j] \leftarrow nd[i][j] + \frac{h^2}{4} * kval * (vdiff + w[d_1][d_2] * \frac{4}{h})$
    return $nd$
```

In Algorithm 2, on line 2 (parallel do), the computations done for each pixels are being sent to all available processors to be evaluated in parallel. We are able to parallelize the computations in this way due to the independent nature of the computations done in inner loops (lines 3 to 12). In order to evaluate $nd[i][j]$, the computations do not depend on any other $nd[k][l]$ values, where $i \neq k$ and $j \neq l$.

To further improve the performance, in both sequential and parallel cases, one can notice that if kernel function, K, is an even function, meaning $K(-x, -y) = K(x, y)$ for any $x, y \in \mathbb{R}$, we need to compute $K((i - d1) * h, (j - d2) * h)$ where $(i - d1)$ and $(j - d2)$ vary between 0 and I and 0 and J respectively. This fact allows us to precompute kernel function values and store them in an $(I + 1) \times (J + 1)$ matrix.

To precompute the kernel function values we use Algorithm 3. By introducing the precomputed K values into sequential Algorithm 1 and parallel Algorithm 2, we can rewrite them to receive a new input parameter k and use them to replace the kernel function computations on lines 6 and 5 respectively. The updated algorithms are listed in Algorithms 4 and 5 respectively.

For experiments, we implemented the parallel algorithms using two different technologies, namely: OpenMP and CUDA. The first technology uses multiple CPUs and threads to implement parallelism, whereas the second one needs special hardware (GPU) to run tasks in parallel.

Algorithm 3: Precomputed K function values.

Input:

- I, J, h: see Algorithm 1

Output:

- k is a $(I+1) \times (J+1)$ matrix containing precomputed K function values

```
function precomputeK($I, J, h$)
    for $i = 0$ to $I$ do
        for $j = 0$ to $J$ do
            $k[i][j] \leftarrow K(i * h, j * h)$;
    return $k$
```

Algorithm 4: Nonlocal diffusion sequential algorithm using precomputed K.

Input:

- v, w, I, J, h: see algorithm 1
- k, the precomputed K function values

Output:

- nd: see Algorithm 1

```
function NDSequentialPK($v, w, k, I, J, h$)
    for $i = 0$ to $I$ do
        for $j = 0$ to $J$ do
            for $d_1 = 0$ to $I$ do
                for $d_2 = 0$ to $J$ do
                    $kval \leftarrow k[abs(i - d_1), abs(j - d_2)]$
                    $vdiff \leftarrow v[d_1][d_2] - v[i][j]$
                    if $(d_1, d_2)$ is internal pixel then
                        $nd[i][j] \leftarrow nd[i][j] + h^2 * kval * vdiff$
                    else if $(d_1, d_2)$ is edge pixel then
                        $nd[i][j] \leftarrow nd[i][j] + \frac{h^2}{2} * kval * (vdiff + w[d_1][d_2] * \frac{2}{h})$
                    else if $(d_1, d_2)$ is corner pixel then
                        $nd[i][j] \leftarrow nd[i][j] + \frac{h^2}{4} * kval * (vdiff + w[d_1][d_2] * \frac{4}{h})$
    return $nd$
```

Algorithm 5: Nonlocal diffusion parallel algorithm using precomputed K.

Input:

- v, w, I, J, h: see Algorithm 1
- k, the precomputed K function values

Output:

- nd: see Algorithm 1

```
function NDParallelPK($v, w, k, I, J, h$)
    for each pixel $(i, j)$ in $v$ parallel do
        for $d_1 = 0$ to $I$ do
            for $d_2 = 0$ to $J$ do
                $kval \leftarrow k[abs(i - d_1), abs(j - d_2)]$
                $vdiff \leftarrow v[d_1][d_2] - v[i][j]$
                if $(d_1, d_2)$ is internal pixel then
                    $nd[i][j] \leftarrow nd[i][j] + h^2 * kval * vdiff$
                else if $(d_1, d_2)$ is edge pixel then
                    $nd[i][j] \leftarrow nd[i][j] + \frac{h^2}{2} * kval * (vdiff + w[d_1][d_2] * \frac{2}{h})$
                else if $(d_1, d_2)$ is corner pixel then
                    $nd[i][j] \leftarrow nd[i][j] + \frac{h^2}{4} * kval * (vdiff + w[d_1][d_2] * \frac{4}{h})$
    return $nd$
```

3.1. Details on OpenMP Implementation

OpenMP (Open Multi-Processing) is an application program interface implemented by most of the modern C/C++ compilers which allows for fast development of portable multi-threaded applications based on shared memory (see [9,10] for more details).

In a shared-memory model all threads have access to the globally shared memory and may have private memory allocated as well. While shared data are accessible by all threads, the private data are only visible locally to the threads. Synchronizing threads' access to shared memory needs to be taken into account at coding time.

In our implementation we used an OpenMP directive called "parallel for" to parallelize the loop on line 2 in Algorithm 2. At run time this will result in having multiple threads running different iterations at the same time. The number of threads used is configurable and in our case was set to eight to match the number of logical processors we had running for our experiments.

3.2. Details on CUDA Implementation

A graphical processing unit (GPU) provides more processing power than a similarly priced CPU. We used an nVidia GPU and CUDA programming framework which nVidia describes as being "a general purpose parallel computing platform and programming model that leverages the parallel compute engine in nVidia GPUs to solve many complex computational problems in a more efficient way than on a CPU" (see [11]).

A program written in CUDA, referred as a kernel, is executed by multiple threads on specialized hardware. A main thread, running on the host CPU, will prepare the input data and load it together with the kernel onto the GPU. After the loading takes place, the main thread will trigger the massive parallel execution of the kernel on the GPU. Massive parallel execution means thousands of threads running the kernel at the same time.

While OpenMP has a flat thread model, in CUDA there is a logical hierarchy for threads organization in blocks of threads and grids. The blocks of threads are logically organized to have up to three dimensions. For our specific implementation, because the model is defined on a 2D domain and we were analyzing images, we chose to define 2D thread blocks of size 32×32 to use the maximum size of 1024 threads per block.

Just like in the OpenMP case, CUDA implements a shared memory model too. In CUDA there are more levels of memory that threads can use: private memory to every thread (local to thread), shared memory between threads (local to a block) and global shared memory accessible by all threads in all blocks. The access time decreases going from private to global shared memory.

Physically, GPU memory is separate from the host memory, used by the CPU, and it is called device memory. Before starting the parallel execution of code on the GPU, the input data are copied from host memory to device memory. Computations' results are also copied from device memory to host memory in order to be accessible by the main host thread. This is a performance overhead which is included in our experimental results in next section.

According to Algorithm 2, the diffusion implementation in CUDA will have a thread assigned to each input image pixel. A specific thread will do all the necessary computations to complete a parallel block (see Algorithm 2, lines 3 to 12).

4. Experiments

For experimenting we implemented the sequential algorithm in C++ using the GCC compiler (version 9.3.0) and the parallel algorithms using both OpenMP on the same GCC compiler and CUDA using NVCC compiler (version V10.1.243).

For testing we used grayscale images of sizes: 128×128, 256×256, 512×512 and 1024×1024 pixels.

The sequential experiments were run on an Intel i7-7700 CPU at 3.60 GHz, using only one logical processor. Parallel CPU experiments (implemented in OpenMP) were run on the same CPU, only we used eight logical processors for the task. The third type of experiment consisted of using an nVidia GPU: Tesla P100-PCIE-16GB. In this case we implemented the parallel algorithm in CUDA using single precision (see [12,13] for image processing using CUDA).

The calculations to precompute kernel function were done on a CPU for all testing scenarios. We experienced short computational times for this task (under 1 second) and did not consider it to need any improvement by implementing it on a GPU.

Simulation results are presented in Tables 1–4. In the first table for relatively small images (128×128 pixels) the parallel GPU implementation results are comparable to parallel CPU ones, while the sequential implementation results are already a few orders of magnitude worse than parallel ones.

In Table 2 the performance gap starts to be bigger as we increase the input domain size (256×256 pixels). This trend is even more accentuated if we look at the results in Tables 3 and 4 where the input domain size increased to 512×512 and 1024×1024 respectively.

We can see a plot of the data in Tables 1–4 in Figure 3. As expected, the GPU implementation performed very well, obtaining better results by a factor of up to 10^3 (see Figure 4).

Precomputing kernel function values led to some performance improvements—although not decisive ones—especially for bigger input domain sizes.

Looking at the performance improvement factors in Figure 4 we see that the parallel GPU implementation is clearly superior to both sequential and parallel CPU implementations. The improvement that parallel CPU implementation brings over sequential implementation is negligible compared to the improvement that parallel GPU implementation brings over any of the other two implementations.

Table 1. Time comparison for the three diffusion algorithm types running on a 128 × 128 image.

	NDSequential	NDParallel	NDParallelGPU
Regular implementation running duration (in sec)	5.29	1.16	0.03
Using precomputed K values running duration (in sec)	2.23	0.52	0.02

Table 2. Time comparison for the three diffusion algorithm types running on a 256 × 256 image.

	NDSequential	NDParallel	NDParallelGPU
Regular implementation running duration (in sec)	89.62	20.19	0.21
Using precomputed K values running duration (in sec)	36.60	8.31	0.09

Table 3. Time comparison for the three diffusion algorithm types running on a 512 × 512 image.

	NDSequential	NDParallel	NDParallelGPU
Regular implementation running duration (in sec)	1683.06	354.09	1.77
Using precomputed K values running duration (in sec)	771.79	169.53	1.22

Table 4. Time comparison for the three diffusion algorithm types running on a 1024 × 1024 image.

	NDSequential	NDParallel	NDParallelGPU
Regular implementation running duration (in sec)	26917.50	5954.42	25.54
Using precomputed K values running duration (in sec)	14812.90	3148.71	19.11

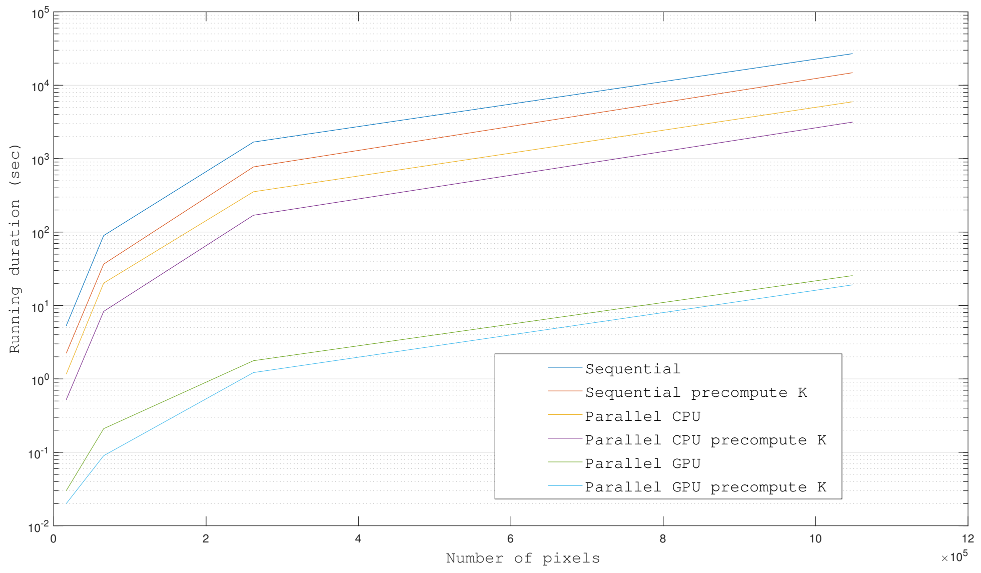

Figure 3. Running durations for all algorithms.

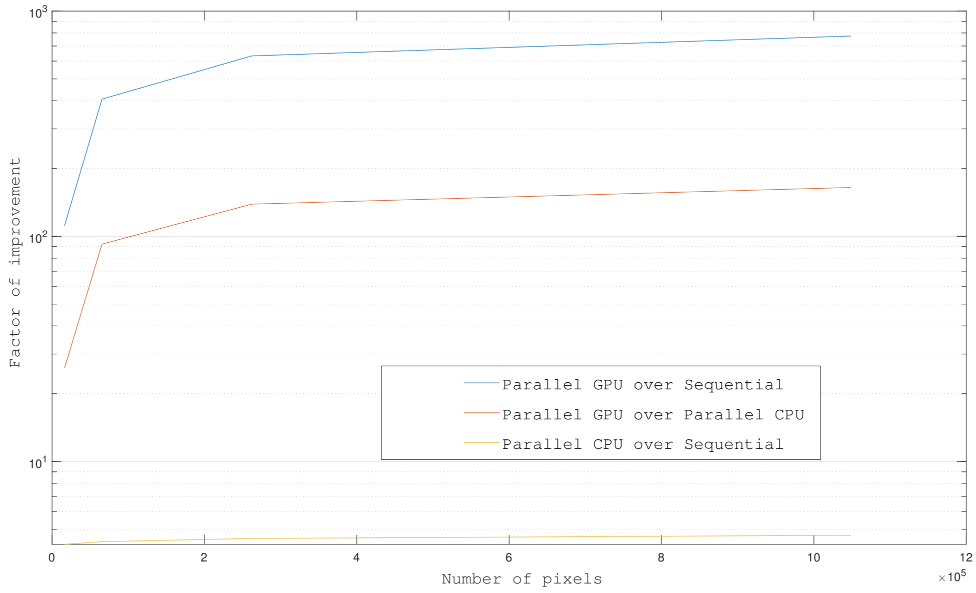

Figure 4. Improvement factors for the three algorithm types.

5. Conclusions

We introduced a discretization method for a nonlocal reaction-diffusion model applied to a 2D input domain. We also studied three ways of improving the performance for numerical approximation computation compared to a basic sequential implementation.

The experiments show that better performance was obtained using massive parallelization given by the CUDA implementation. While this was in fact what we expected to see, the factor of 10^3 times better results compared to the best result we could obtain with a CPU sequential implementation, demonstrates that GPU implementations in numerical simulations in general, and for computing integrals in particular, are highly recommended. At the same time, our results show that in cases in which GPU implementations cannot be used, for different reasons, CPU parallel implementations can bring decisive performance improvements as well.

We choose to use shared-memory parallelization standards (OpenMP and CUDA) due to the fine granularity of the calculations involved in our discretization model. In the parallel algorithms we compute numerical values attached to the pixels in a given input image. Should we choose to use a message passing standard (like MPI) to parallelize the computations on distributed multi-core systems, in this case the data transfer between nodes would be an important penalty to performance. This is because the computed values for each pixel depend on values computed for all other pixels in the image at the preceding time step, so all computing nodes should communicate data to all other computing nodes in the cluster at each time step.

Further improvements to our implementation, which could be discussed in a future article, can be achieved by implementing better memory management in cases of parallel algorithms on CUDA.

Author Contributions: The authors have contributed equally to this manuscript. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Barbu, T.; Miranville, A.; Sanu, C.M. A qualitative analysis and numerical simulations of a nonlinear second-order anisotropic diffusion problem with non-homogeneous Cauchy-Neumann boundary conditions. *Appl. Math. Comput.* **2019**, *350*, 170–180. [CrossRef]
2. Perona, P.; Malik, J. Scale-space and edge detection using anisotropic diffusion. *Proc. IEEE Comput. Soc. Workshop Comput. Vis.* **1987**, *12*, 16–22. [CrossRef]
3. Weickert, J. Anisotropic Diffusion in Image Processing. In *European Consortium for Mathematics in Industry*; B. G. Teubner: Stuttgart, Germany, 1998.
4. Miranville, A.; Moroşanu, C. Qualitative and Quantitative Analysis for the Mathematical Models of Phase Separation and Transition. Applications AIMS. 2020. Available online: https://www.aimsciences.org/fileAIMS/cms/news/info/28df2b3d-ffac-4598-a89b-9494392d1394.pdf (accessed on 5 October 2020).
5. Gómez, C.A.; Rossi, J.D. A nonlocal diffusion problem that approximates the heat equation with Neumann boundary conditions. *J. King Saud Univ. Sci.* **2020**, *32*, 17–20. Available online: http://www.sciencedirect.com/science/article/pii/S1018364717307887 (accessed on 5 October 2020).
6. Pavăl, S. Numerical approximation for a nonlocal reaction-diffusion equation supplied with non-homogeneous Neumann boundary conditions. Case 1D. *ROMAI J.* **2019**, *15*, 83–93. Available online: https://rj.romai.ro/arhiva/2019/1/Paval.pdf (accessed on 5 October 2020).
7. Pavăl, S.; Craus, M. Parallel Algorithm for a Nonlocal Diffusion Model Applied to a 3D Input Domain. In Proceedings of the 24th International Conference on System Theory, Control and Computing (ICSTCC), Sinaia, Romania, 8–10 October 2020.
8. Mitchell, A.R.; Griffiths, D.F. *The Finite Difference Method in Partial Differential Equations*; John Wiley & Sons: Hoboken, NJ, USA, 1980.
9. OpenMP. Available online: https://www.openmp.org (accessed on 5 October 2020).
10. Van der Pas, R. An Overview of OpenMP. Available online: https://www.openmp.org/wp-content/uploads/ntu-vanderpas.pdf (accessed on 5 October 2020).
11. NVIDIA Corporation. CUDA Programming Guide 10.0. Available online: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html (accessed on 5 October 2020).
12. Yang, Z.; Zhu, Y.; Pu, Y. Parallel Image Processing Based on CUDA. In Proceedings of the 2008 International Conference on Computer Science and Software Engineering, Hubei, China, 15 June 2008; pp. 198–201. [CrossRef]
13. Fredj, H.B.; Ltaif, M.; Ammar, A.; Souani, C. Parallel implementation of Sobel filter using CUDA. In Proceedings of the Control Automation and Diagnosis (ICCAD) 2017 International Conference, Hammamet, Tunisia, 21 January 2017; pp. 209–212.