computers

MDPI

Article

Effects of OpenCL-Based Parallelization Methods on Explicit Numerical Methods to Solve the Heat Equation

Dániel Koics [1], Endre Kovács [2,*] and Olivér Hornyák [3]

[1] Institute of Automation and Infocommunication, University of Miskolc, 3515 Miskolc, Hungary; daniel.koics@uni-miskolc.hu
[2] Institute of Physics and Electrical Engineering, University of Miskolc, 3515 Miskolc, Hungary
[3] Institute of Applied Informatics, University of Miskolc, 3515 Miskolc, Hungary; oliver.hornyak@uni-miskolc.hu
* Correspondence: endre.kovacs@uni-miskolc.com

Abstract: In recent years, the need for high-performance computing solutions has increased due to the growing complexity of computational tasks. The use of parallel processing techniques has become essential to address this demand. In this study, an Open Computing Language (OpenCL)-based parallelization algorithm is implemented for the Constant Neighbors (CNe) and CNe with Predictor–Corrector (CpC) numerical methods, which are recently developed explicit and stable numerical algorithms to solve the heat conduction equation. The CPU time and error rate performance of these two methods are compared with the sequential implementation and Euler's explicit method. The results demonstrate that the parallel version's CPU time remains nearly constant under the examined circumstances, regardless of the number of spatial mesh points. This leads to a remarkable speed advantage over the sequential version for larger data point counts. Furthermore, the impact of the number of timesteps on the crossover point where the parallel version becomes faster than the sequential one is investigated.

Keywords: heat transfer; partial differential equations; numerical methods; stability of numerical methods; *Euler*'s method; *CNe* method; *CpC* method; parallelization; Open CL

Citation: Koics, D.; Kovács, E.; Hornyák, O. Effects of OpenCL-Based Parallelization Methods on Explicit Numerical Methods to Solve the Heat Equation. *Computers* **2024**, *13*, 250. https://doi.org/10.3390/computers13100250

Academic Editor: Paolo Bellavista

Received: 15 August 2024
Revised: 12 September 2024
Accepted: 20 September 2024
Published: 1 October 2024

1. Introduction

Parallel computing has emerged as a key approach to meet the increasing demand for high-performance computing solutions. By dividing computational tasks into smaller sub-tasks that can be executed simultaneously, parallel computing offers the potential to significantly improve the speed and efficiency of numerical methods [1]. One popular framework for parallel computing is *OpenCL*, which allows the utilization of heterogeneous computing platforms, including CPUs, GPUs, and FPGAs.

Several methods are used and processed for solving the equations related to heat transfer and diffusion, e.g., semi-analytical methods [2], domain-decomposition methods [3], and finite-element solvers [4].

The most frequent approach may be the spatial discretization of the equation by the central difference formula and then the solution of the obtained ODE (ordinary differential equation) system by either an explicit or an implicit finite difference [5] algorithm. The most well-known explicit methods, however, such as the Runge–Kutta family, are only conditionally stable, and the solution is expected to diverge whenever the timestep size is above a threshold [6], which is called the CFL (Courant–Friedrichs–Lewy) limit. The advantage of the explicit schemes is that they can execute a timestep quite quickly and they can be easily coded and parallelized.

Implicit methods have much fewer problems with stability; thus, they are commonly used by many scientists and software [7–9]. However, the unknown values cannot be expressed explicitly; hence, a system of algebraic equations must be solved in each timestep.

This is computationally challenging, especially in multiple space dimensions, where the matrix is enormous and non-tridiagonal. Semi-explicit or semi-implicit algorithms, as combinations of explicit and implicit algorithms, are also developed [10–13].

However, we believe that the trend toward increasing parallelization can be followed using explicit schemes, which can be more efficient even if short timesteps have to be used. For example, Essongue et al. showed that the explicit Euler time integration can be faster than the implicit one for 3D thermal simulations [14]. Moreover, not all explicit methods suffer from severe stability issues. A new suite of robust techniques has been created by our group for addressing the non-stationary heat equation, which is explicit and unconditionally stable at the same time. In our original works [15,16], the new algorithms were theoretically examined, verified, and tested. Most importantly, it has been analytically proven that when using these schemes, the temperature values at the new time level are the convex combination of those at the old time level, which implies the stability of the arbitrary time step size and arbitrary space discretization. The main weakness of these explicit and stable schemes is that their accuracy does not always match the standard methods with the same order of convergence. Even with this, these non-conventional methods can severely outperform the standard ones, see, e.g., [17]. In those papers, we frequently mentioned that they are easily parallelizable. The novelty of this research is that this is the first occasion the implementation is completed, which delves into examining how *OpenCL*-based parallelization affects two of these methods, namely, the Constant Neighbors and the CNe with Predictor–Corrector (which is a two-stage version of the *CNe* similar to second-order Runge–Kutta schemes).

In this study, the methods being investigated are tested with a couple of initial conditions using various temporal and spatial scales. For each test, a reference solution is used to measure the error. The error is collected as a function of the CPU time required to run the algorithm. The error–runtime functions of the parallel versions of *CNe* and *CpC* are compared with the sequential counterparts, as well as with *Euler*'s explicit time integration. Additionally, we aim to investigate the impact of the number of timesteps on the crossover point where the parallel version becomes faster than the sequential one. The crossover point typically occurs due to the overhead associated with parallelization, which can be higher for small problem sizes but becomes advantageous as the problem size increases [18,19]. By analyzing the performance characteristics of these parallel implementations, we can gain insights into the benefits and limitations of utilizing *OpenCL*-based parallelization in numerical methods for solving heat equations.

2. Equation and Methods

2.1. The Numerical Methods Being Investigated

This work focuses on the heat conduction equation. It is mathematically analogous to the diffusion equation. The homogenous version of them is as follows:

$$\frac{\partial f(\boldsymbol{r},t)}{\partial t} = D \cdot \Delta f(\boldsymbol{r},t), \tag{1}$$

which is to be used with an $f(\boldsymbol{r}, T_{\text{init}}) \equiv f_{\text{init}}(\boldsymbol{r})$ initial condition, and where

f is the temperature [°C, K] —or, in the case of diffusion, concentration,
t is the simulated time [s],
$\boldsymbol{r} = \begin{bmatrix} x \\ y \\ z \end{bmatrix}$ is the position vector [m],
D is the diffusion coefficient [m^2/s], and
$\Delta \equiv \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2} + \frac{\partial^2}{\partial z^2}$ is the Laplacian operator.

As for the f concentration term, one should note that the mass concentration [kg/m^3], molar concentration [mol/m^3], and particle count concentration [m^{-3}] are all applicable as long as the same type of concentration is used consistently on both side of the equation.

After expanding the Laplacian, we have the following equation:

$$\frac{\partial f(\boldsymbol{r},t)}{\partial t} = D\cdot\left(\frac{\partial^2 f(\boldsymbol{r},t)}{\partial x^2} + \frac{\partial^2 f(\boldsymbol{r},t)}{\partial y^2} + \frac{\partial^2 f(\boldsymbol{r},t)}{\partial z^2}\right).$$

In this study, the equations are applied to simple, rectangle-like domains and a fixed *Dirichlet*'s condition is used at the boundaries (see Equations (12) and (13) in Section 2.3).

The algorithms mentioned in Section 1 belong to the family of finite difference methods, which makes spatial and temporal discretization necessary. The spatial discretization divides the domain into a grid (with initial points X_{init}, Y_{init}, and Z_{init} and endpoints X_{fin}, Y_{fin}, and Z_{fin}):

$$\boldsymbol{r}_{i,j,k} = \begin{bmatrix} X_{\text{init}} + i\cdot\Delta x \\ Y_{\text{init}} + j\cdot\Delta y \\ Z_{\text{init}} + k\cdot\Delta z \end{bmatrix} = \begin{bmatrix} X_{\text{init}} \\ Y_{\text{init}} \\ Z_{\text{init}} \end{bmatrix} + \begin{bmatrix} \Delta x & \Delta y & \Delta z \end{bmatrix}\cdot\begin{bmatrix} i-1 \\ j-1 \\ k-1 \end{bmatrix}, \tag{2}$$

where the spatial step sizes are

$$\Delta x = (X_{\text{fin}} - X_{\text{init}})/(N_{\text{x}} - 1),$$
$$\Delta y = (Y_{\text{fin}} - Y_{\text{init}})/(N_{\text{y}} - 1),$$
$$\Delta z = (Z_{\text{fin}} - Z_{\text{init}})/(N_{\text{z}} - 1);$$

the index ranges are

$$i \in \{1, 2 \ldots N_{\text{x}}\},$$
$$j \in \{1, 2 \ldots N_{\text{y}}\},$$
$$k \in \{1, 2 \ldots N_{\text{z}}\};$$

and N_{x}, N_{y}, and N_{z} are the number of nodes along the spatial mesh, i.e., the data point counts. It should be mentioned that in Section 3.1, the notation N_{r} is used as the total number of spatial grid-points, which is

$$N_{\text{r}} = N_{\text{x}} \tag{3}$$

for a 1-dimensional measurement and

$$N_{\text{r}} = N_{\text{x}}\cdot N_{\text{y}} \tag{4}$$

for 2-dimensional data.

Similarly, the temporal discretization divides the time interval into smaller timesteps as follows:

$$t_n = T_{\text{init}} + n\cdot\Delta t, \tag{5}$$

where the step size and index range are

$$\Delta t = (T_{\text{fin}} - T_{\text{init}})/N_{\text{t}},$$
$$n \in \{0, 1, 2 \ldots N_{\text{t}}\}.$$

Euler's method, which serves as a reference in this paper, uses a straightforward conception to produce the data point values of timestep $n + 1$ as an explicit function of the values of timestep n. Approximating the Laplacian operator with the central difference formula, the stepper formula is (for ease of understanding, in 2 dimensions) as follows:

$$f_{i,j}^{(n+1)} \cong f_{i,j}^{(n)} + D\cdot\Delta t\cdot\left(\frac{f_{i+1,j}^{(n)} - 2\cdot f_{i,j}^{(n)} + f_{i-1,j}^{(n)}}{\Delta x^2} + \frac{f_{i,j+1}^{(n)} - 2\cdot f_{i,j}^{(n)} + f_{i,j-1}^{(n)}}{\Delta y^2}\right), \tag{6}$$

where the index ranges are

$$i \in \{1, 2 \ldots N_x\},$$
$$j \in \{1, 2 \ldots N_y\}.$$

This method is susceptible to numerical instability and requires a small timestep size for accuracy. To eliminate stability problems, *CNe* method uses a convex combination of the function values to calculate the values of the next timestep. First, an average temperature value $f_{i,j}^{(\mathrm{neb},n)}$ of the neighboring datapoints is calculated, and then this value is used as an ambient or terminal temperature to apply Newton's law of heat transfer (i.e., to calculate the exponentially converging value after the timestep has elapsed):

$$f_{i,j}^{(n+1)} \cong \exp\left\{-\frac{\Delta t}{\tau}\right\} \cdot f_{i,j}^{(n)} + \left(1 - \exp\left\{-\frac{\Delta t}{\tau}\right\}\right) \cdot f_{i,j}^{(\mathrm{neb},n)}, \tag{7}$$

where

$$\tau = 1 / \left[2D \cdot \left(\frac{1}{\Delta x^2} + \frac{1}{\Delta y^2}\right)\right]$$

is the time constant of the node and

$$f_{i,j}^{(\mathrm{neb},n)} = \frac{1}{2} \cdot \left(\frac{f_{i+1,j}^{(n)} + f_{i-1,j}^{(n)}}{\Delta x^2} + \frac{f_{i,j+1}^{(n)} + f_{i,j-1}^{(n)}}{\Delta y^2}\right) / \left(\frac{1}{\Delta x^2} + \frac{1}{\Delta y^2}\right)$$

is the asymptotic value to which the node temperature tends.

The bottleneck of this method is its relatively poor accuracy. To mitigate this problem, *CpC* method combines it with a predictor–corrector approach, namely the second-order *Runge–Kutta* scheme. In the first stage, an approximating value is calculated at a fractional timestep using the *CNe* formula. In the second stage, a linear combination of the original value and the result of the first stage is used as the terminal temperature for *CNe* to calculate the full-timestep value as follows:

$$f_{i,j}^{(n+p)} = \exp\left\{-\frac{\Delta t}{\tau}\right\} \cdot f_{i,j}^{(n)} + \left(1 - \exp\left\{-\frac{\Delta t}{\tau}\right\}\right) \cdot f_{i,j}^{(n)\infty}, \tag{8}$$

$$f_{i,j}^{(n+1)} = \exp\left\{-\frac{\Delta t}{\tau}\right\} \cdot f_{i,j}^{(n)} + \left(1 - \exp\left\{-\frac{\Delta t}{\tau}\right\}\right) \cdot \left[\left(1 - \frac{1}{2p}\right) \cdot f_{i,j}^{(n)\infty} + \frac{1}{2p} f_{i,j}^{(n+p)\infty}\right]; \tag{9}$$

where $0 < p \leq 1$ is the step fraction.

It is mathematically proven in [15] that the above algorithm with coefficients $1 - 1/2p$ and $1/2p$ indeed yields to a second-order error term. The results of [15] also show that the accuracy does not depend significantly on the value of parameter p; hence, $p = 1/2$ is a good choice to simplify calculations. By doing so, the coefficient of the pre-step value becomes zero, enabling the result of the first stage to be used directly in the second stage without having to calculate the linear combination:

$$f_{i,j}^{(n+1/2)} = \exp\left\{-\frac{\Delta t}{\tau}\right\} \cdot f_{i,j}^{(n)} + \left(1 - \exp\left\{-\frac{\Delta t}{\tau}\right\}\right) \cdot f_{i,j}^{(n)\infty}, \tag{10}$$

$$f_{i,j}^{(n+1)} = \exp\left\{-\frac{\Delta t}{\tau}\right\} \cdot f_{i,j}^{(n)} + \left(1 - \exp\left\{-\frac{\Delta t}{\tau}\right\}\right) \cdot f_{i,j}^{(n+1/2)\infty}, \tag{11}$$

which yields an algorithm analog to the *midpoint* method.

2.2. Parallelization and Applying OpenCL

Both the CNe and CpC methods are parallelizable, and the same applies to the Euler's method. When determining whether an algorithm is parallelizable, several key criteria must be considered. One important criterion is the independence of tasks, often referred to

as data parallelism. This refers to the extent to which tasks or operations can be performed independently of each other. If the algorithm's tasks do not depend on the results of other tasks, they can be executed in parallel. Another crucial factor is task granularity, which refers to the size of the tasks being parallelized. Fine-grained tasks, which involve small and fast operations, can create overhead in managing parallel tasks, whereas coarse-grained tasks, involving larger and more complex operations, may reduce overhead but require careful balancing. Communication overhead is also a key consideration. This involves the amount of data exchange or synchronization required between parallel tasks. Algorithms that require minimal communication between tasks are easier to parallelize. Load balancing, which is the ability to distribute the workload evenly across all available computing units, is another important criterion. For effective parallelization, the workload should be balanced to avoid situations where some processors are idle while others are overloaded. Scalability is another critical factor, referring to how well the algorithm's performance improves with the addition of more processing units. An algorithm is considered scalable if its performance continues to improve as more cores or processors are added. A scalable algorithm might demonstrate near-linear speedup as the number of processors increases, such as in parallel matrix multiplication. Synchronization needs must also be considered. These involve the need for synchronization between tasks during execution. Memory access patterns are also important. The way an algorithm accesses memory can affect parallelization, with algorithms that have predictable and non-conflicting memory access patterns being easier to parallelize. For example, algorithms with regular, sequential memory access patterns, such as matrix operations, are easier to optimize for parallel execution compared to those with irregular access patterns that may cause contention. In this research, a cell-by-cell parallelization is carried out, which means that each kernel invocation calculates one cell value.

In this paper, OpenCL [20] was used. Open Computing Language is a framework for writing programs that execute across heterogeneous platforms, including CPUs, GPUs, DSPs, and FPGAs. Its primary purpose is to facilitate parallel computing by allowing developers to harness the computational power of diverse hardware architectures within a unified programming model. By leveraging parallelization, OpenCL aims to significantly enhance performance in various computational tasks, from scientific simulations to real-time data processing. OpenCL is widely used for high-performance computing [21]. Its cross-platform nature [22] makes it useful in various domains of application. Parallelization was used in [23] for solving nonlinear ordinary differential equations. A sparse matrix-vector multiplication is presented in [24]. Parallel computing in the modeling of fractional-order state–space systems is presented by [25]. The Smith–Waterman implementation is shown in [26].

The kernel function for Euler's method, as well as for the *CNe* method, can be found in Listing 1 (in 2 dimensions, supposing a GPU supports *Khronos*-style 64-bit floating point arithmetic [27]):

Listing 1: Kernel of *CNe* and the 1st stage of *CpC*.

```
#pragma OPENCL EXTENSION cl_khr_fp64 : enable
__kernel void calc_cell_2D(
        __global const double *before, __global double *after,
        double2 coeffs, uint2 size, long4 neighbours) {
    const size_t index = get_global_id(0) + size.x * get_global_id(1);
    __global const double *old_cell = before + index;
    after[index] = (1 − 2 * (coeffs.x + coeffs.y)) * *old_cell
        + coeffs.x *
            (old_cell[neighbours.even.x] +
            old_cell[neighbours.odd.x])
        + coeffs.y *
            (old_cell[neighbours.even.y] +
            old_cell[neighbours.odd.y]);
}
```

where

- before, after are pointers to the array containing data before and after the timestep;
- coeffs is a vector containing numerical coefficients for computation;
- size is a vector containing the $[N_x, N_y]$ number of datapoints;
- neighbours is a vector containing pointer differences between the current datapoint and its neighbours (introduced to simplify handling boundary conditions).

During each timestep, the calculation uses the value of the neighboring cells. To ensure that none of these neighboring values has been updated by parallel cell computation—i.e., the before-step value of all the neighbors is used—two memory buffers are necessary [28]. During odd timestep computations, buffer 1 is read while buffer 2 is written to. During even timestep computations, buffer 2 is read while buffer 1 is written to, as follows (Figure 1):

This is an explicit buffer parallelization technique [29]. Its advantage is that the programmer controls when and how data are moved, optimizing the data transfer to minimize latency and maximize throughput. It allows the use of device-specific features and capabilities, potentially enabling hardware-specific optimizations [30] that are not possible with a more abstracted approach. The first stage of the CpC method can also use the same kernel as shown in Listing 1, but this algorithm also needs a second-stage kernel, as well (Listing 2):

Listing 2: Kernel of the 2nd stage of *CpC*.

```
__kernel void stage2p05_calc_cell_2D(
            __global const double *stage0,
            __global const double *stage1,
            __global double *after,
            double2 coeffs, uint2 size, long4 neighbours) {
        const size_t index = get_global_id(0) + size.x * get_global_id(1);
        __global const double *old_cell = stage0 + index;
        __global const double *midpoint = stage1 + index;
        after[index] = (1 − 2 * (coeffs.x + coeffs.y)) * *old_cell
                + coeffs.x *
                    (midpoint[neighbours.even.x] +
                    midpoint[neighbours.odd.x])
                + coeffs.y *
                    (midpoint[neighbours.even.y] +
                    midpoint[neighbours.odd.y]);
}
```

where

- stage0 is a pointer to the array containing data before stage 1;
- stage1 is a pointer to the array containing the result of stage 1;
- after is a pointer to the array containing data after all the stages;
- coeffs is a vector containing numerical coefficients for computation;
- size is a vector containing the $[N_x, N_y]$ number of datapoints;
- neighbours is a vector containing pointer differences between the current datapoint and its neighbours (introduced to possibly simplify handling boundary conditions).

Method CpC also requires 2 memory buffers, but they are used in a different way. The computation of each step uses both buffers. The first stage reads buffer 1 and writes the result to buffer 2. The second stage reads both buffer 1 (stage0) and buffer 2 (stage1) and stores the result in buffer 1 (after), as shown in Figure 2.

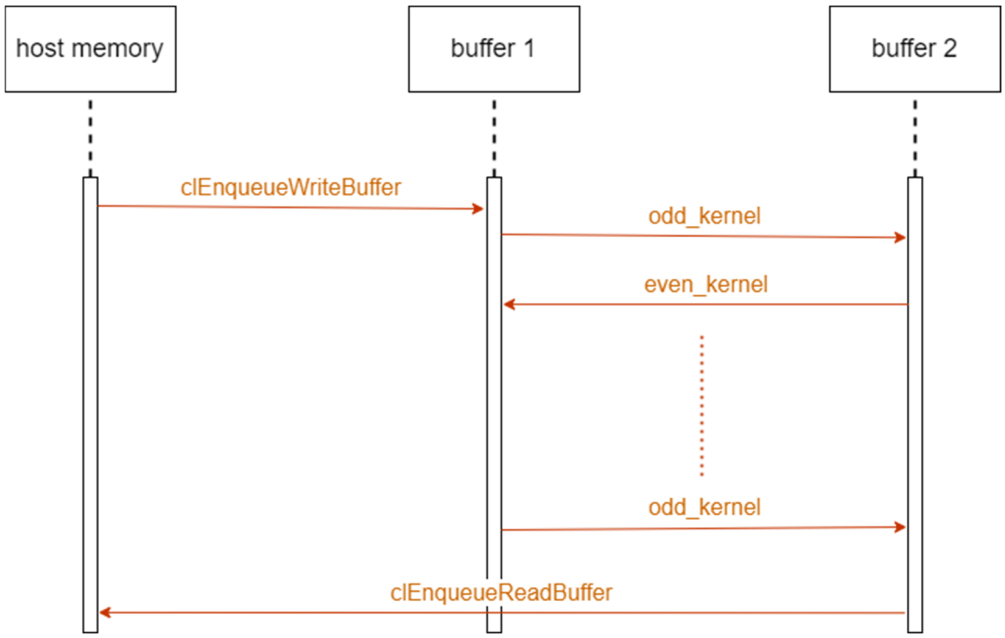

Figure 1. Using memory buffers with *CNe* and the 1st stage of *CpC*.

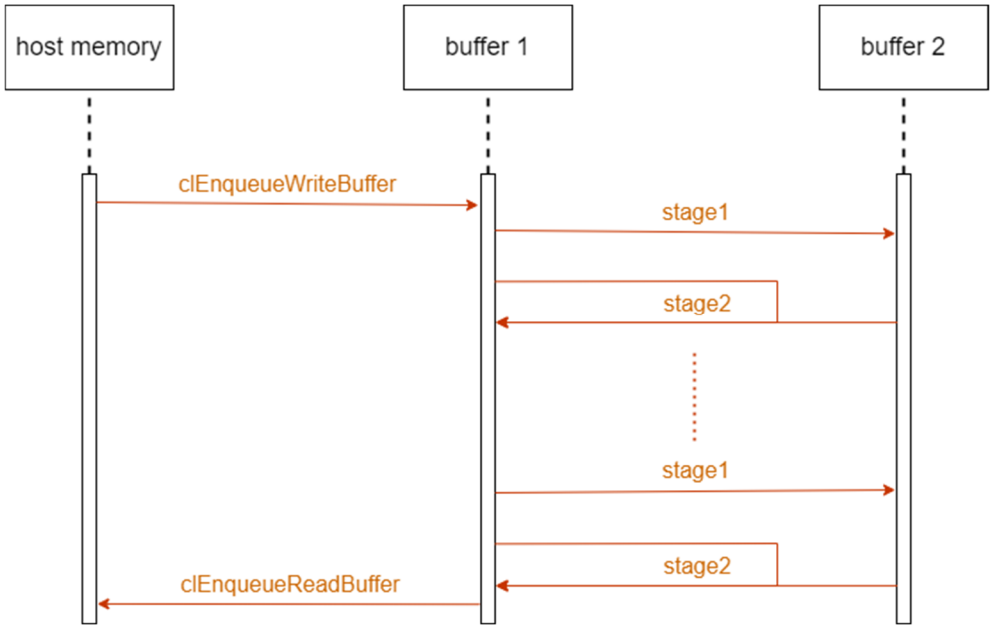

Figure 2. Using memory buffers with *CpC*.

2.3. Initial and Boundary Conditions, Scale Parameters, and the Method of Investigation

In this study, fixed *Dirichlet* boundaries [31] are assumed, meaning that the initial value of the boundaries is kept constant during the calculation, and no calculation is involved on these boundaries, as follows:

$$\begin{aligned} f_1^n &= f_1^{n-1} = \cdots = f_1^0, \\ f_{N_x}^n &= f_{N_x}^{n-1} = \cdots = f_{N_x}^0 \end{aligned} \tag{12}$$

in 1 dimension, and

$$\left\{ f_{1,j}^n = f_{1,j}^{n-1} = \cdots = f_{1,j}^0 \middle| j \in \{1, \ldots N_y\} \right\},$$

$$\left\{ f_{N_x,j}^n = f_{N_x,j}^{n-1} = \cdots = f_{N_x,j}^0 \middle| j \in \{1, \ldots N_y\} \right\},$$

$$\left\{f_{i,1}^{n} = f_{i,1}^{n-1} = \cdots = f_{i,1}^{0} \middle| i \in \{1, \ldots N_{\mathrm{x}}\}\right\},$$
$$\left\{f_{i,N_{\mathrm{x}}}^{n} = f_{i,N_{\mathrm{x}}}^{n-1} = \cdots = f_{i,N_{\mathrm{x}}}^{0} \middle| i \in \{1, \ldots N_{\mathrm{x}}\}\right\} \quad (13)$$

in 2 dimensions.

To deal with the initial values, four different problems are created using two initial condition functions for both 1- and 2-dimensional measurements. In the case of one type of the initial function, we have the analytical solution, which is used as a reference. The other problem does not have an analytical solution; hence, a numerical reference solution is used instead. This reference solution is generated using the built-in ode15s function of *MATLAB* [32]. These initial value functions are the following (assuming $X_{\mathrm{init}} = 0$, $Y_{\mathrm{init}} = 0$ and $T_{\mathrm{init}} = 0$):

- A 1D sine with nodes at the boundaries is

$$f(x, t = 0) = A \cdot \sin\left(\pi \cdot \frac{x}{X_{fin}}\right) + C, \quad (14)$$

where

$$A = 0.8,$$
$$C = 1$$

are arbitrarily chosen parameters and the analytical solution is

$$f(x, t) = A \cdot \exp\left\{-\pi^2 \cdot \frac{D \cdot t}{X_{fin}{}^2}\right\} \cdot \sin\left(\pi \cdot \frac{x}{X_{fin}}\right) + C. \quad (15)$$

- A 1D *Gaussian* function [33] is

$$f(x, t = 0) = \frac{A}{\sqrt{2\pi} \cdot \sigma} \cdot \exp\left\{-\frac{(x - \mu)^2}{2\sigma^2}\right\}, \quad (16)$$

where

$$A = 1,$$
$$\mu = 3.4,$$
$$\sigma = 4.2$$

are arbitrarily chosen parameters. In this case, a numerical reference solution is used.

- In 2D, the product of sinewaves for each dimension with nodes at the boundaries is

$$f(x, y, t = 0) = A \cdot \sin\left(\pi \cdot \frac{x}{X_{fin}}\right) \cdot \sin\left(\pi \cdot \frac{y}{Y_{fin}}\right) + C, \quad (17)$$

where

$$A = 0.8,$$
$$C = 1$$

are arbitrarily chosen parameters and the analytical solution is

$$f(x, y, t) = A \cdot \exp\left\{-\pi^2 D t \cdot \left(\frac{1}{X_{fin}^2} + \frac{1}{Y_{fin}^2}\right)\right\} \cdot \sin\left(\pi \cdot \frac{x}{X_{fin}}\right) \cdot \sin\left(\pi \cdot \frac{y}{Y_{fin}}\right) + C. \quad (18)$$

- A 2D planar wave, propagating at an angle [34] is

$$f(x, y, t = 0) = A \cdot \cos\left(2\pi \cdot \begin{bmatrix} k_x & k_y \end{bmatrix} \cdot \begin{bmatrix} x \\ y \end{bmatrix} - \varphi\right). \tag{19}$$

where

$$A = 0.8,$$
$$\begin{bmatrix} k_x & k_y \end{bmatrix} = \begin{bmatrix} 2.2 & 0.5 \end{bmatrix},$$
$$\varphi = 1$$

are arbitrarily chosen parameters. In this case, a numerical reference solution is used.

The timescales and data grids used together with these functions are listed in Tables 1–4. During the measurement, each row of the timescale tables is combined with each row of the appropriate spatial mesh–grid table (i.e., the rows of Table 1 with the rows of Table 2 and the rows of Table 3 with the rows of Table 4) and one of the 2 appropriate initial functions. For each combination, the measurement is repeated 4 times, and the mean value along with the standard deviation of the CPU time is calculated. The maximal absolute value of the difference between the output data points and the reference solution values (mathematically spoken, the l_∞ distance of the output vector and the reference vector) is considered as the error of the algorithm as follows:

$$Err_{Max} = \max_{i \in \{1,2 \ldots N_x\}} \left| f_i^{N_t,\text{calc}} - f_i^{N_t,\text{ref}} \right| \tag{20}$$

for 1 dimensional calculations and

$$Err_{Max} = \max_{\substack{i \in \{1,2 \ldots N_x\} \\ j \in \{1,2 \ldots N_y\}}} \left| f_{i,j}^{N_t,\text{calc}} - f_{i,j}^{N_t,\text{ref}} \right| \tag{21}$$

for 2 dimensional calculations. (This value is expected to be the same for all the 4 repetitions, since the examined algorithms are deterministic.)

Table 1. The timescales used with 1D data.

N_t	Δt
9	1.1111×10^{-1}
15	6.6667×10^{-2}
30	3.3333×10^{-2}
50	2.0000×10^{-2}
90	1.1111×10^{-2}
150	6.6667×10^{-3}
300	3.3333×10^{-3}
500	2.0040×10^{-3}
900	1.1111×10^{-3}
1500	6.6667×10^{-4}
3000	3.3333×10^{-4}
5000	2.0000×10^{-4}
9000	1.1111×10^{-4}
15,000	6.6667×10^{-5}
30,000	3.3333×10^{-5}
50,000	2.0000×10^{-5}
90,000	1.1111×10^{-5}
150,000	6.6667×10^{-6}
300,000	3.3333×10^{-6}

Table 1. *Cont.*

$\mathbf{N_t}$	$\mathbf{\Delta t}$
500,000	2.0000×10^{-6}
900,000	1.1111×10^{-6}
1,500,000	6.6667×10^{-7}
3,000,000	3.3333×10^{-7}
5,000,000	2.0000×10^{-7}
9,000,000	1.1111×10^{-7}

All scales start at $T_{init} = 0$ and finish at $T_{fin} = 1$.

Table 2. The 1D spatial grids.

$\mathbf{N_x}$	$\mathbf{\Delta x}$
50	2.0408×10^{-1}
100	1.0101×10^{-1}
200	5.0251×10^{-2}
400	2.5063×10^{-2}
800	1.2516×10^{-2}
1200	8.3403×10^{-3}
2000	5.0025×10^{-3}
4000	2.5006×10^{-3}
8000	1.2502×10^{-3}
12,000	8.3340×10^{-4}

All grids start at $X_{init} = 0$ and finish at $X_{fin} = 10$.

Table 3. The timescales used with 2D data.

$\boldsymbol{N_t}$	$\boldsymbol{\Delta t}$
9	1.1111×10^{-2}
15	6.6667×10^{-3}
30	3.3333×10^{-3}
50	2.0000×10^{-3}
90	1.1111×10^{-3}
150	6.6667×10^{-4}
300	3.3333×10^{-4}
500	2.0000×10^{-4}
900	1.1111×10^{-4}
1500	6.6667×10^{-5}
3000	3.3333×10^{-5}
5000	2.0000×10^{-5}
9000	1.1111×10^{-5}
15,000	6.6667×10^{-6}
30,000	3.3333×10^{-6}
50,000	2.0000×10^{-6}
90,000	1.1111×10^{-6}
150,000	6.6667×10^{-7}
300,000	3.3333×10^{-7}
500,000	2.0000×10^{-7}
900,000	1.1111×10^{-7}

All scales start at $T_{init} = 0$ and finish at $T_{fin} = 0.1$.

Table 4. The 2D spatial grids.

$\boldsymbol{N_x}$	$\boldsymbol{N_y}$	$\boldsymbol{N_x N_y}$	$\boldsymbol{\Delta x}$	$\boldsymbol{\Delta y}$	$\boldsymbol{\Delta x \Delta y}$
25	25	625	4.1667×10^{-2}	4.1667×10^{-2}	1.7361×10^{-3}
25	50	1250	4.1667×10^{-2}	2.0408×10^{-2}	8.5034×10^{-4}
50	50	2500	2.0408×10^{-2}	2.0408×10^{-2}	4.1649×10^{-4}

Table 4. *Cont.*

N_x	N_y	N_xN_y	Δx	Δy	$\Delta x \Delta y$
50	75	3750	2.0408×10^{-2}	1.3514×10^{-2}	2.7579×10^{-4}
75	75	5625	1.3514×10^{-2}	1.3514×10^{-2}	1.8262×10^{-4}
75	100	7500	1.3514×10^{-2}	1.0101×10^{-2}	1.3650×10^{-4}
100	100	10,000	1.0101×10^{-2}	1.0101×10^{-2}	1.0203×10^{-4}
100	150	15,000	1.0101×10^{-2}	6.7114×10^{-3}	6.7792×10^{-5}
150	150	22,500	6.711×10^{-3}	6.7114×10^{-3}	4.5043×10^{-5}
150	200	30,000	6.7114×10^{-3}	5.0251×10^{-3}	3.3726×10^{-5}
200	200	40,000	5.0251×10^{-3}	5.0251×10^{-3}	2.5252×10^{-5}

All grids start at (X_{init},Y_{init}) = (0,0) and finish at (X_{fin},Y_{fin}) = (1,1).

2.4. The Specifications of the Applied Computer, Further Circuimstances of the Investigation

The aforementioned algorithms can be found at the repository mentioned in Supplementary Materials. These were executed on a personal computer (PC) to validate their performance and functionality. The specifications of the PC used for this purpose are as follows:

- CPU—Gen11 Intel Core i7-11700F, 2.5 GHz;
- RAM—64 GB;
- OS—Windows 10 Professional (22H2), ×64;
- GPU—NVIDIA;
- Open CL—Intel v2020.3.494.

3. Results

The results presented in this section refer to the cases described in Section 2.3.

In this examination, the problem sizes had to be limited to the size that can be practically calculated during the experiments, given the computational resources available, such as the GPU and CPU used. These circumstances include the number of spatial mesh points and the scale of the computational task that the hardware could handle efficiently. In traditional, sequential algorithms, the CPU time typically increases as the number of spatial mesh points grows because more calculations are required. However, in the parallel version of the algorithm studied, the CPU time remained nearly constant across the range of mesh points tested. This means that within the practical limits of the problem sizes tested, the parallel implementation was able to efficiently distribute the workload across multiple processing units (like GPU cores), allowing it to handle larger problem sizes without a corresponding increase in computation time.

3.1. The Dependency of CPU Time on the Number of Data Points

In the following subsections, the CPU time of the given implementation (i.e., *CNe* in Section 3.1.1 and *CpC* in Section 3.1.2) is plotted as the function of the total number of spatial grid points (see Equations (3) and (4) for a definition of N_r) for a fixed number of timesteps, along with the CPU time of Euler's method. For ease of comparison, the 1-dimensional and 2-dimensional data are plotted on the same graph. Each data point in Figures 3–6 contains both the problems with analytical and numerical reference solutions, meaning that the average of eight CPU time values is used instead of four values.

On Figures 3–6, one can observe, that in the parallel case, the CPU time remains unaffected by the data point count (presumably due to no depletion of the GPU capability in these measurements), whereas in the sequential case, it increases with the data point number. For most cases under examination at two-dimensional levels and all one-dimensional instances, the system size consistently falls below a critical cross-over point threshold, and hence finer space grids (i.e., greater datapoint count) are required.

The position of the cross-over point shifts towards lower data point counts as timestep numbers rise. With 500 timesteps, OpenCL demonstrates greater efficiency within a range

of 6000–8000 data points; meanwhile, with 900,000 timesteps, the transition happens at around 3000 data points.

3.1.1. The CNe Method Compared to *Euler*'s Method

See Figures 3 and 4 below.

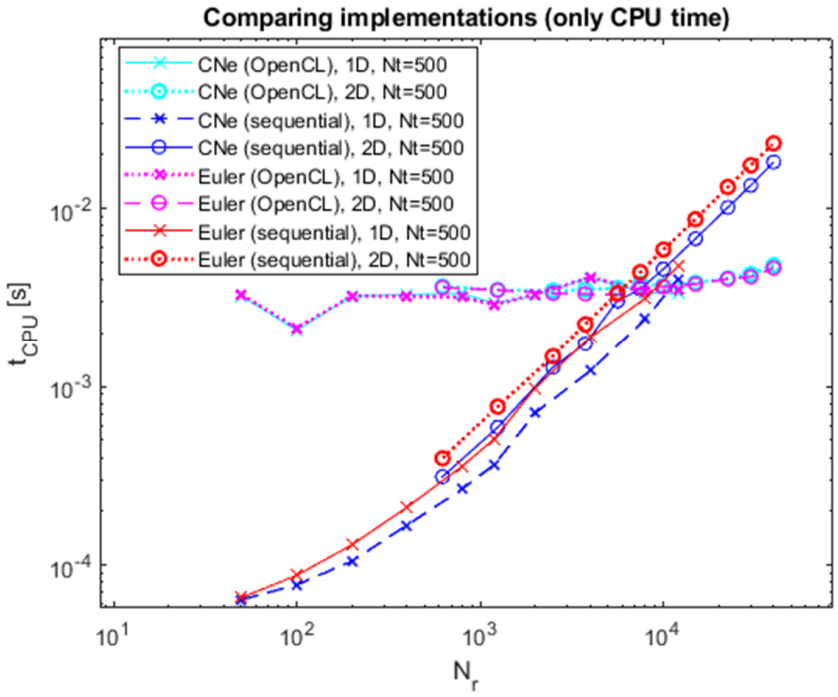

Figure 3. The CPU time of *CNe* and *Euler*'s methods as a function of the number of spatial mesh points for 500 timesteps.

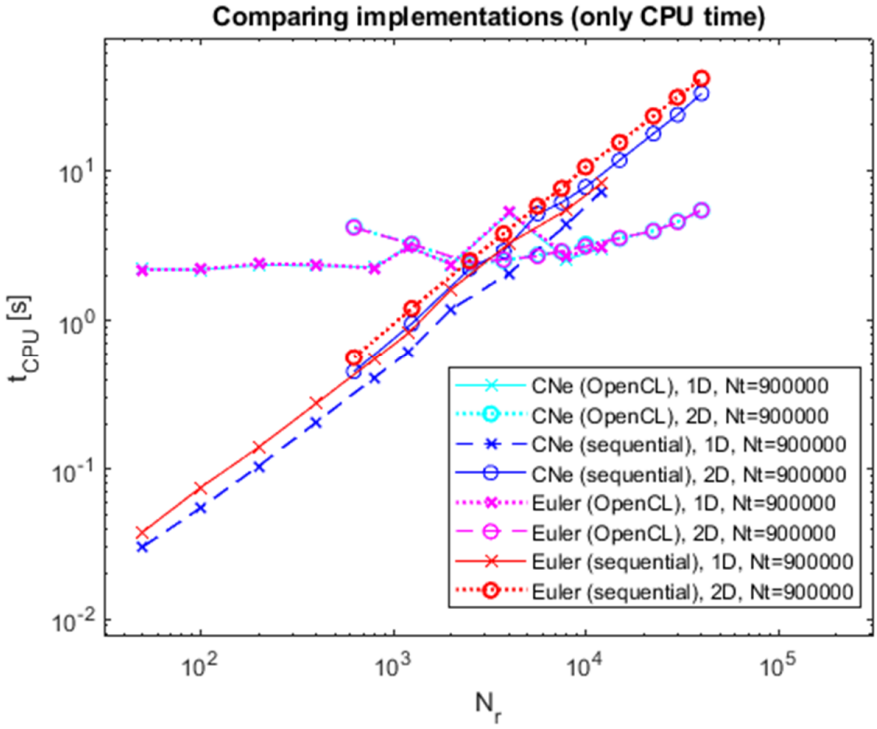

Figure 4. The CPU time of CNe and Euler's methods as a function of the number of spatial mesh points for 900,000 timesteps.

3.1.2. The CpC Method Compared to *Euler*'s Method

See Figures 5 and 6 below.

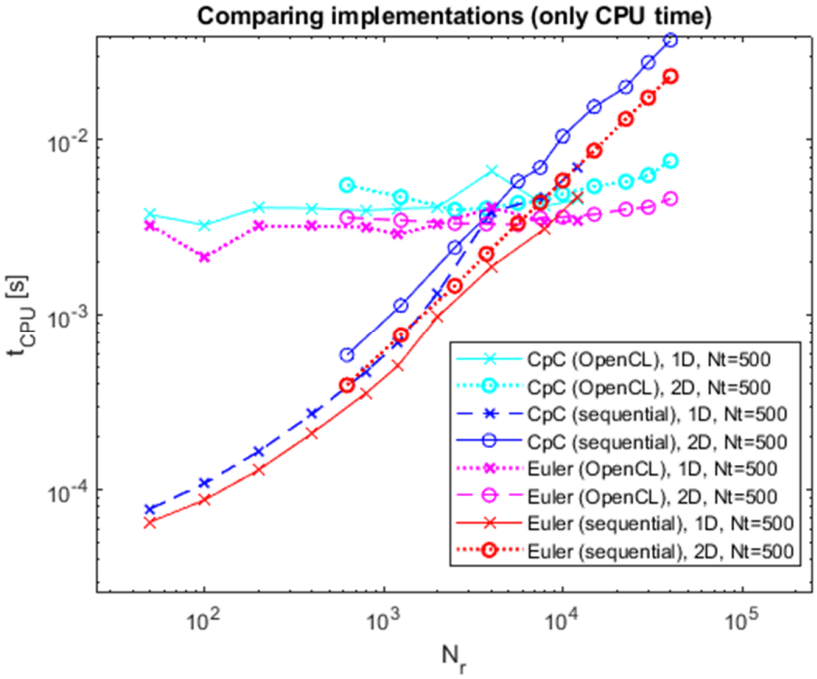

Figure 5. The CPU time of CpC and Euler's methods as a function of the number of datapoints for 500 timesteps.

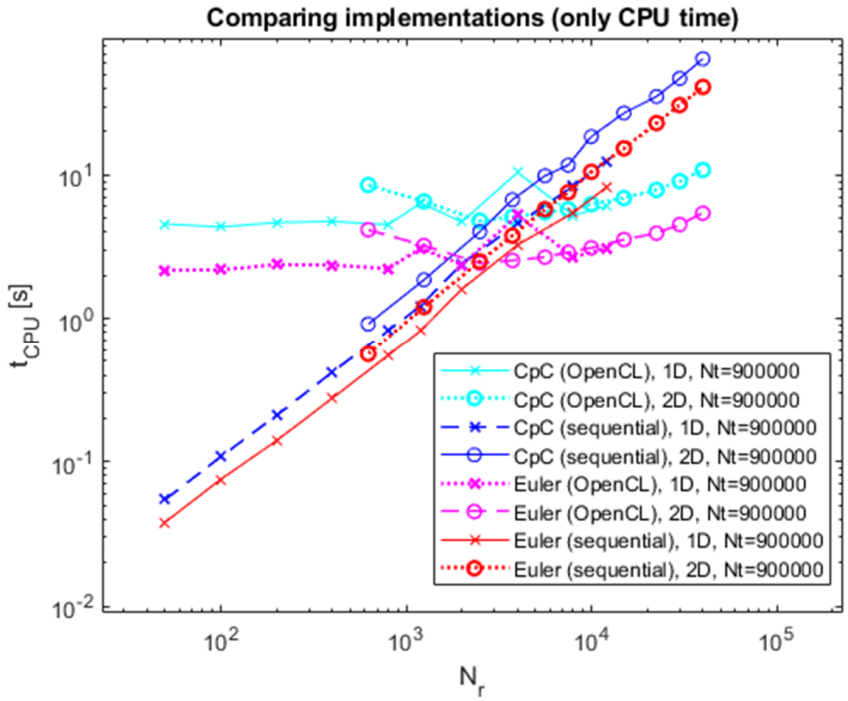

Figure 6. The CPU time of CpC and Euler's methods as a function of the number of data points for 900,000 timesteps.

3.2. The Error as a Function of the Timestep Size

In this section, the error is plotted as a function of the CPU time necessary to execute the algorithm. (For the definition of Err_{Max}, see Equations (20) and (21).) This technique

enables a sophisticated comparison of the examined methods (CNe in Section 3.2.1 and CpC in Section 3.2.2) with Euler's method.

As it can be expected based on Section 3.1, on Figures 7–14, it is noticeable that better performance with *OpenCL* is only achieved in two dimensions for large data grids. It can also be seen that the errors of both the parallel and the sequential algorithms are similar. This confirms that the *OpenCL*-based and sequential implementations conduct identical mathematical operations, and they differ at most in scheduling.

3.2.1. The CNe Method Compared to Euler's Method

See Figures 7–10 below.

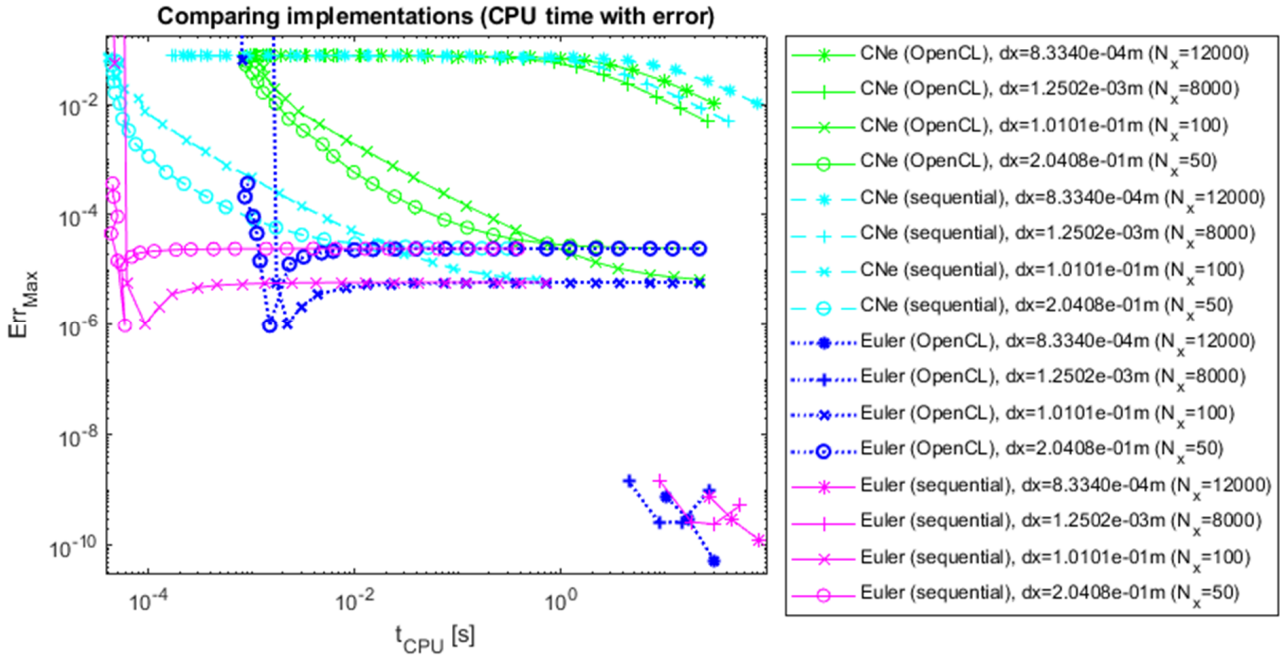

Figure 7. The errors of CNe and Euler's methods as a function of CPU time, 1D with analytical reference.

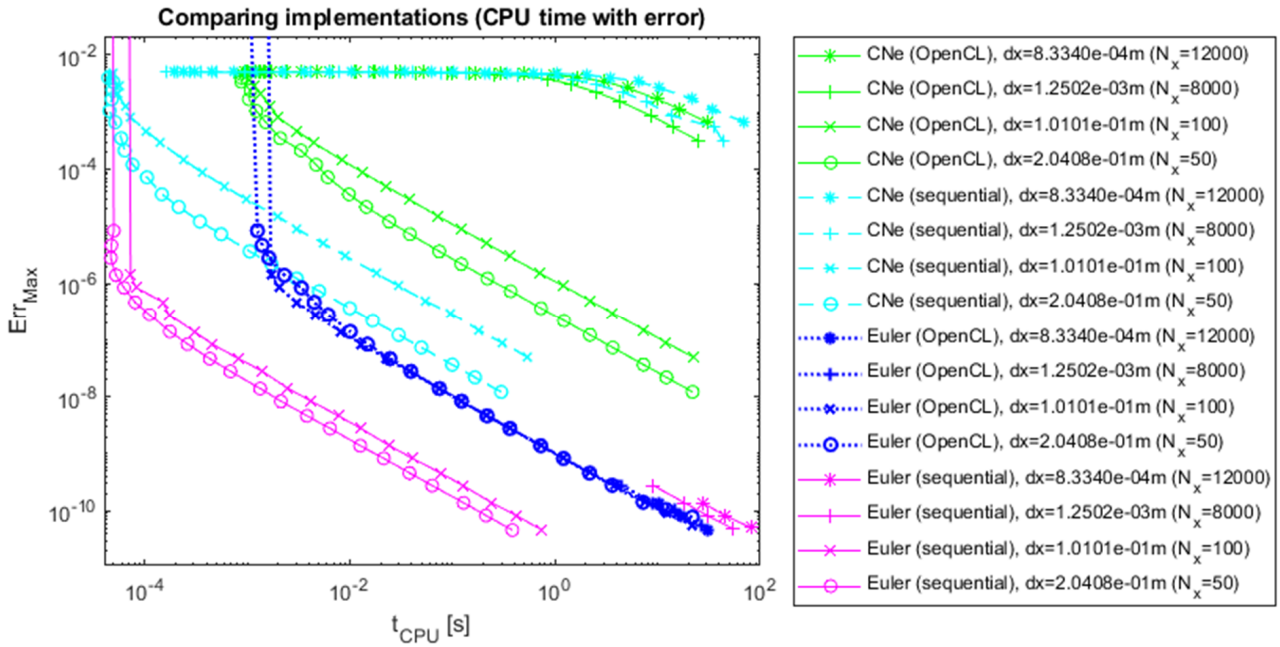

Figure 8. The errors of CNe and Euler's methods as a function of CPU time, 1D with numerical reference.

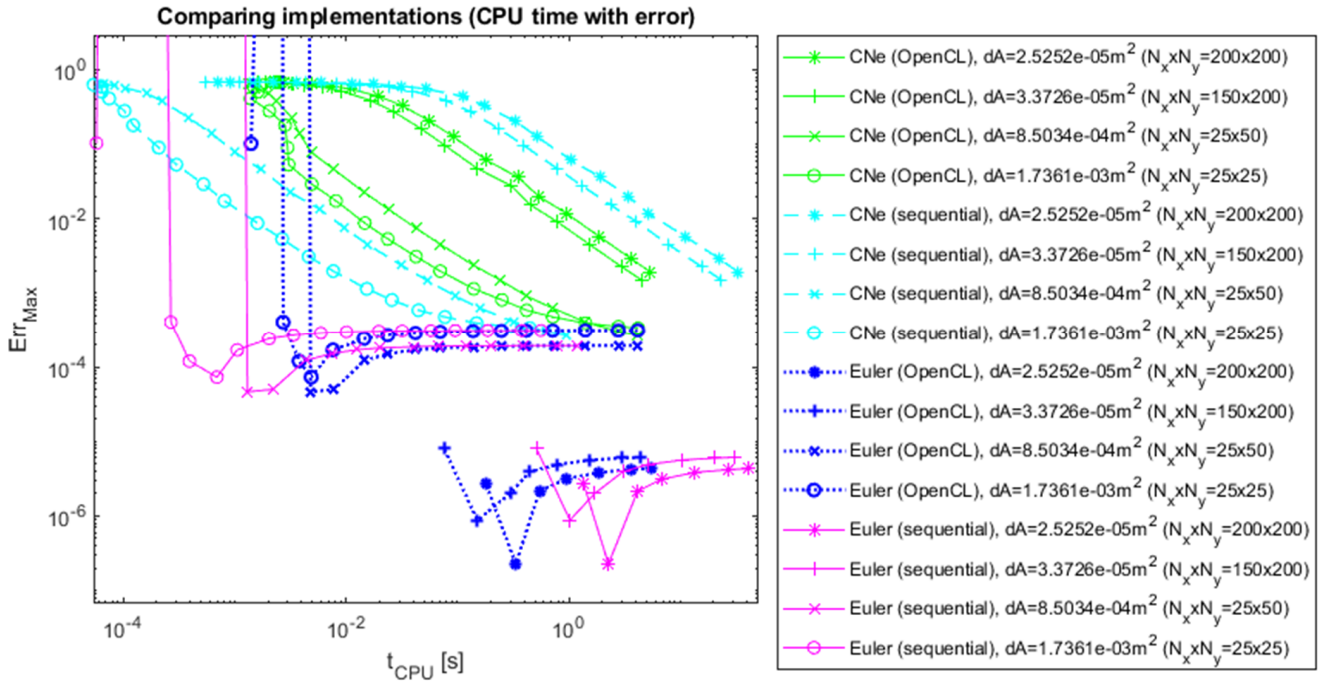

Figure 9. The errors of CNe and Euler's methods as a function of CPU time, 2D with analytical reference.

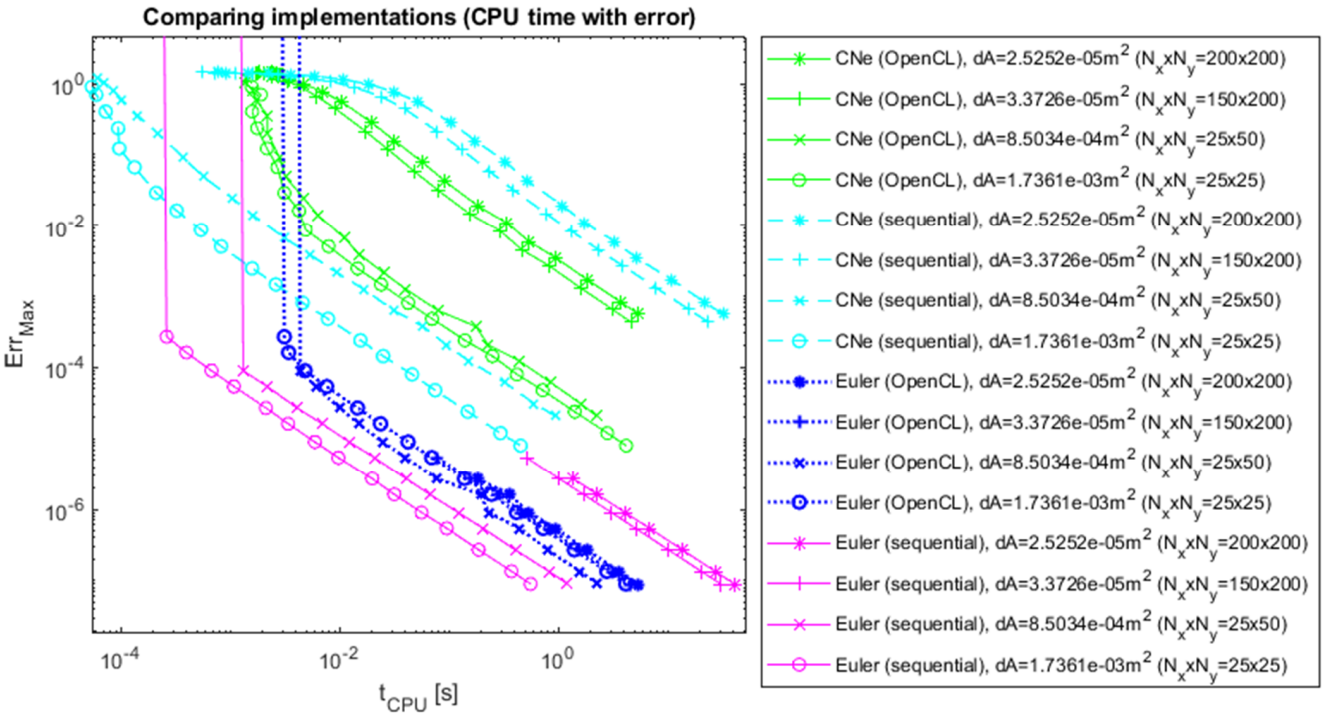

Figure 10. The errors of CNe and Euler's methods as a function of CPU time, 2D with numerical reference.

3.2.2. The CpC Method Compared to Euler's Method

See Figures 11–14 below.

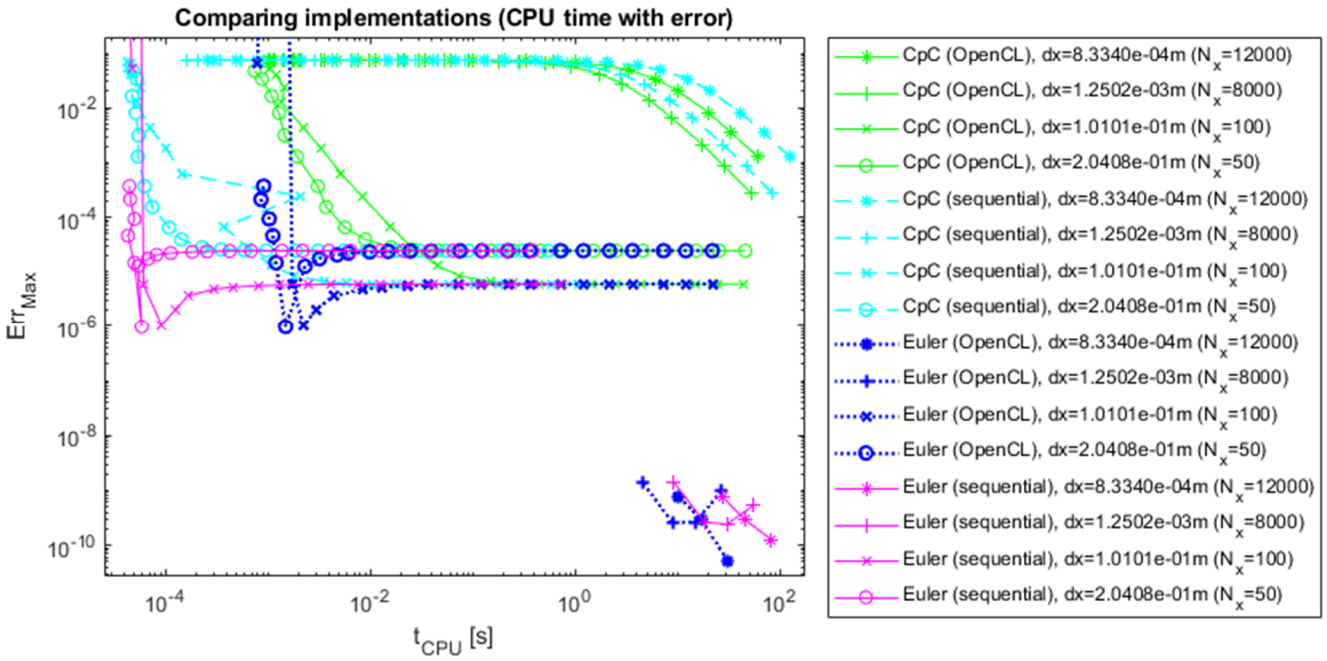

Figure 11. The errors of CpC and Euler's methods as a function of CPU time, 1D with analytical reference.

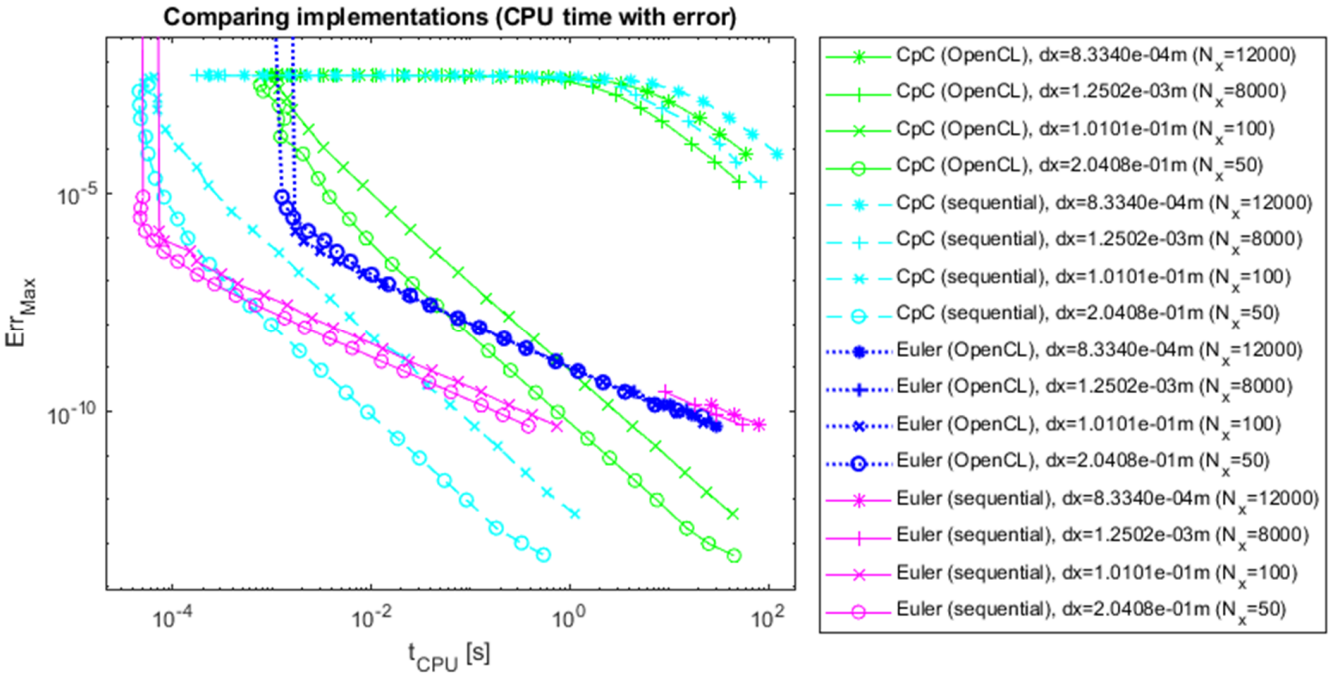

Figure 12. The errors of CpC and Euler's methods as a function of CPU time, 1D with numerical reference.

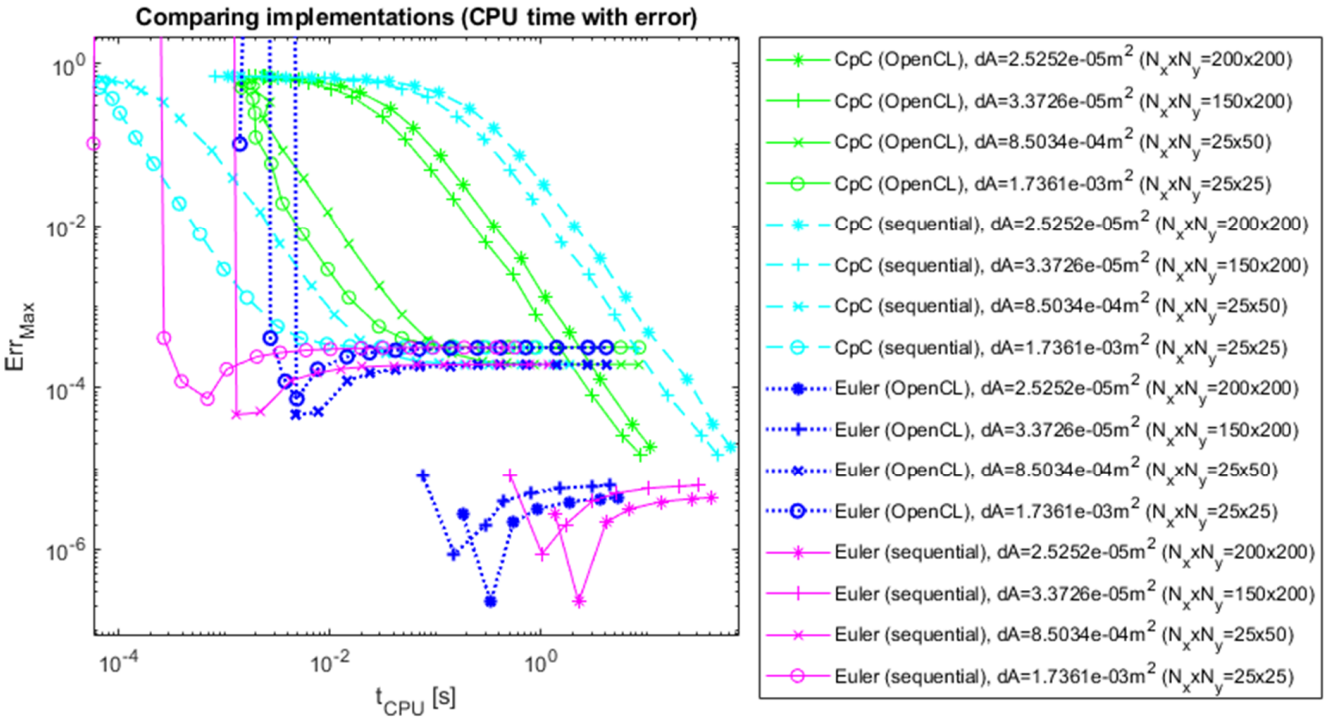

Figure 13. The errors of CpC and Euler's methods as a function of CPU time, 2D with analytical reference.

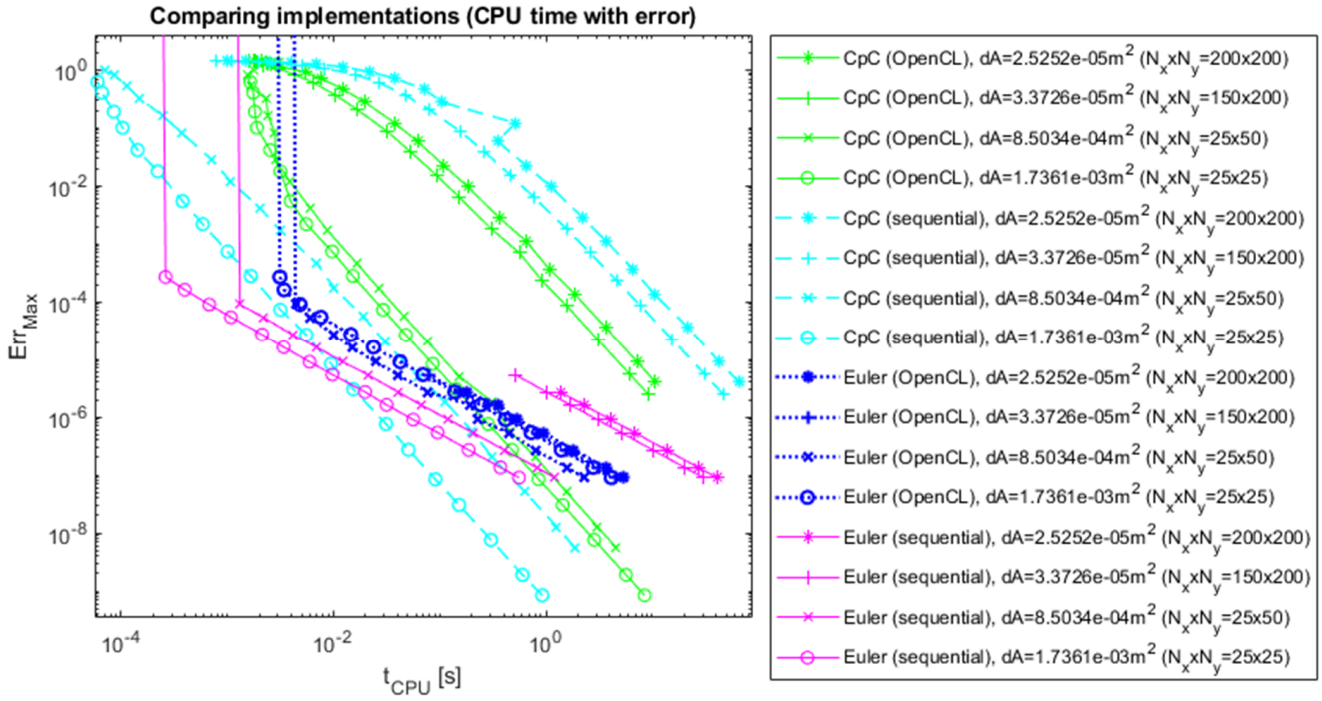

Figure 14. The errors of CpC and Euler's methods as a function of CPU time, 2D with numerical reference.

4. Discussion and Conclusions

According to this research, it can be argued that employing OpenCL has the potential to achieve improved performance; however, whether such enhanced performance is realized depends on certain conditions. The size of the data grids examined in this study is not enough to deplete the parallel computing capacity of the computer used for the calculation. As a result, the CPU time required by OpenCL-based algorithms is determined mostly by the number of timesteps and only to a much smaller extent by the number of data points.

Consequently, OpenCL becomes more efficient than its sequential counterpart only when a sufficiently high number of data points is employed (which appears to decrease as the number of timesteps increases). In our investigation, only certain 2-dimensional scenarios meet this criterion; in other cases, it has been demonstrated that sequential calculation outperforms parallel processing.

The constancy in CPU time is likely due to the parallel nature of the algorithm, where the computational load is spread out across multiple processors. However, this effect is observed only up to the point where the problem size is within the capabilities of the hardware. Beyond this point, or with extremely large mesh sizes, the CPU time might increase as the computational resources become a limiting factor. The key takeaway is that for the problem sizes tested, which the hardware could manage effectively, the parallel implementation showed a significant advantage in maintaining a consistent CPU time, making it particularly suitable for larger-scale problems.

Future experiments should utilize finer data grids to illustrate the effectiveness of *OpenCL* compared to sequential computing. In one dimension, a minimum of 300 to 400 data points is required, while in two dimensions, a grid size of at least 150×150 would be suitable for the upper limit of test cases.

Supplementary Materials: The repository of the implementation of the examined algorithms can be found at https://www.bitbucket.org/koicsd/diffusion2, accessed on 19 September 2024. The snapshots of the code used in this investigation can be found by the following tag-names: Sequential Euler's method, *bison04Apr2024_seqEuler*; OpenCL-based Euler's method, *bison04Apr2024_clEuler*; sequential CNe method, *bison04Apr2024_seqCNe*; OpenCL-based CNe method, *bison05Apr2024_clCNe*; sequential CpC method, *bison02Apr2024_seqCpC*; and OpenCL-based CpC method, *bison03Apr2024_clCpC*. The test framework used is hosted at https://www.bitbucket.org/koicsd/test-tool, accessed on 19 September 2024.

Author Contributions: Conceptualization, methodology, supervision, project administration, and resources, E.K. and O.H.; software, validation, investigation, visualization, and writing—original draft preparation, D.K.; writing—review and editing, D.K., E.K., and O.H. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Data Availability Statement: The data are available from the first author on request.

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

References

1. Robert, E.T.; Larson, E. Parallel Processing Algorithms for the Optimal Control of Nonlinear Dynamic Systems. *IEEE Trans. Comput.* **1973**, *100*, 777–786.
2. Patel, Y.F.; Dhodiya, J.M. Efficient algorithm to study the class of Burger's Fisher equation. *Int. J. Appl. Nonlinear Sci.* **2022**, *3*, 179–266. [CrossRef]
3. Xue, G.; Gao, Y. A Samarskii domain decomposition method for two-dimensional convection-diffusion equations. *Comput. Appl. Math.* **2022**, *41*, 283. [CrossRef]
4. Kumar, V.; Chandan, K.; Nagaraja, K.V.; Reddy, M.V. Heat conduction with Krylov subspace method using FEniCSx. *Energies* **2022**, *15*, 8077. [CrossRef]
5. Köroğlu, C.; Aydin, A. Exact and nonstandard finite difference schemes for the Burgers equation B(2,2). *Turk. J. Math.* **2021**, *45*, 3. [CrossRef]
6. Ruan, C.; Dong, C.; Zhang, Z.; Chen, B.; Liu, Z. Finite difference-peridynamic differential operator. *Comput. Model. Eng. Sci.* **2024**, *140*, 2707–2728.
7. Yang, J.; Kim, J. Consistently and unconditionally energy-stable linear method for the diffuse-interface model of narrow volume reconstruction. *Eng. Comput.* **2024**, *40*, 2617–2627. [CrossRef]
8. Mbroh, N.A.; Munyakazi, J.B. A robust numerical for singularly perturbed parabolic reaction-diffusion problems via the method of lines. *Comput. Math.* **2021**, *99*, 1139–1158. [CrossRef]
9. Aydin, A.; Koroglu, C. A nonstandard numerical method for the modified KdV equa-tion. *Pramana—J. Phys.* **2017**, *89*, 72. [CrossRef]
10. Beuken, L.; Cheffert, O.; Tutueva, A.; Butusov, D.; Legat, V. Numerical stability and performance of semi-explicit and semi-implicit predictor-corrector methods. *Mathematics* **2022**, *10*, 2015. [CrossRef]

11. Ji, Y.; Xing, Y. Highly accurate and efficient time integration methods with unconditional stability and flexible numerical dissipation. *Mathematics* **2023**, *11*, 593. [CrossRef]
12. Dou, N.; Dlamini, P.; Jacobs, B.A. Enhanced unconditionally positive finite difference method for advection-diffusion-reaction equations. *Mathematics* **2022**, *10*, 2639. [CrossRef]
13. Jaglan, J.; Singh, A.; Maurya, V.; Yadav, V.S.; Rajpoot, M.K. Strong stability preserving multiderivative time marching methods for stiff reaction-diffusion systems. *Math. Comput. Simul.* **2024**, *225*, 267–282. [CrossRef]
14. Essongue, S.; Ledoux, Y.; Ballu, A. Speeding up mesoscale thermal simulations of powder bed additive manufacturings thanks to the forward Euler time integration scheme: A critical assesment. *Finite Elem. Anal. Des.* **2022**, *211*, 103825. [CrossRef]
15. Kovács, E.; Nagy, Á.; Saleh, M. A set of new, stabl, explicit, second-order schemes for the nonsationary heat conduction equation. *Mathematics* **2021**, *9*, 2284. [CrossRef]
16. Kovács, E. A Class of New Stable, Explicit Methods to Solve the Non-Stationary Heat Equation. *Numer. Methods Partial Differ. Equ.* **2020**, *37*, 2469–2489. [CrossRef]
17. Askar, A.H.; Nagy, Á.; Barna, I.F.; Kovács, E. Analytical and numerical results for the diffusion-reaction equation when the reaction coefficient depends on simultaneously the space and time coordinates. *Computation* **2023**, *11*, 127. [CrossRef]
18. Midkiff, S.P. *Automatic Parallelization: An Overview of Fundamental Compiler Techniques*; Springer Nature: Cham, Switzerland, 2022.
19. Parhami, B. *Introduction to Parallel Processing: Algorithms and Architectures*; Springer Science & Business Media: Cham, Switzerland, 2006.
20. Munshi, A. The OpenCL specification. In *2009 IEEE Hot Chips 21 Symposium (HCS)*; IEEE: Stanford, CA, USA, 2009; pp. 1–314. [CrossRef]
21. Kang, P. Programming for high-performance computing on edge accelerators. *Mathematics* **2023**, *11*, 1055. [CrossRef]
22. Takáč, M.; Petráš, I. Cross-Platform GPU-Based Implementation of Lattice Boltzmann Method Solver Using ArrayFire Library. *Mathematics* **2021**, *9*, 1793. [CrossRef]
23. Tavakkoli, V.; Mohsenzadegan, K.; Chedjou, J.C.; Kyamakya, K. Contribution to Speeding-Up the Solving of Nonlinear Ordinary Differential Equations on Parallel/Multi-Core Platforms for Sensing Systems. *Sensors* **2020**, *20*, 6130. [CrossRef]
24. Baskaran, M.M.; Bordawekar, R. Optimizing Sparse Matrix-Vector Multiplication on GPUs. IBM Research Report RC24704, (W0812–047) 2009. Available online: https://dominoweb.draco.res.ibm.com/reports/rc24704.pdf (accessed on 19 September 2024).
25. Stanisławski, R.; Kozioł, K. Parallel Implementation of Modeling of Fractional-Order State-Space Systems Using the Fixed-Step Euler Method. *Entropy* **2019**, *21*, 931. [CrossRef]
26. Di Tucci, L.; O'Brien, K.; Blott, M.; Santambrogio, M.D. Architectural optimizations for high performance and energy efficient Smith-Waterman implementation on FPGAs using OpenCL. In Proceedings of the Design, Automation & Test in Europe Conference & Exhibition (DATE), Lausanne, Switzerland, 27–31 March 2017; pp. 716–721.
27. Khronos Group. OpenCL 3.0 Specification. 2020. Available online: https://www.khronos.org/registry/OpenCL/specs/3.0-unified/html/OpenCL_API.html (accessed on 19 September 2024).
28. Jo, G.; Jung, J.; Park, J.; Lee, J. Memory-Access-Pattern Analysis Techniques for OpenCL Kernels. In *International Workshop on Languages and Compilers for Parallel Computing*; Springer International Publishing: Cham, Switzerland, 2017; pp. 109–126.
29. Jääskeläinen, P.; de La Lama, C.S.; Schnetter, E.; Raiskila, K.; Takala, J.; Berg, H. pocl: A performance-portable OpenCL implementation. *Int. J. Parallel Program.* **2015**, *43*, 752–785. [CrossRef]
30. Wang, Z.; He, B.; Zhang, W.; Jiang, S. A performance analysis framework for optimizing OpenCL applications on FPGAs. In Proceedings of the 2016 IEEE International Symposium on High Performance Computer Architecture (HPCA), Barcelona, Spain, 12–16 March 2016; pp. 114–125.
31. Smith, G.D. *Numerical Solution of Partial Differential Equations: Finite Difference Methods*; Oxford University Press: Oxford, UK, 1985.
32. MathWorks. MATLAB—ODE15S. 2023. Available online: https://www.mathworks.com/help/matlab/ref/ode15s.html (accessed on 19 September 2024).
33. Weisstein, E.W. (n.d.). Gaussian Function. In MathWorld—A Wolfram Web Resource. Available online: https://mathworld.wolfram.com/GaussianFunction.html (accessed on 19 September 2024).
34. Jackson, J.D. *Classical Electrodynamics*, 3rd ed.; Wiley: Hoboken, NJ, USA, 1998.