Computers and Mathematics with Applications 123 (2022) 184–203

Contents lists available at ScienceDirect

Computers and Mathematics with Applications

www.elsevier.com/locate/camwa

A new GPU-based corrected explicit-implicit domain decomposition scheme for convection-dominated diffusion problems

Ali Foadaddini [a], Seyed Alireza Zolfaghari [a,*], Hossein Mahmoodi Darian [b], Hamid Saadatfar [c]

[a] Department of Mechanical Engineering, University of Birjand, Birjand, Iran
[b] School of Engineering Science, University of Tehran, Tehran, Iran
[c] Department of Computer Engineering, University of Birjand, Birjand, Iran

ARTICLE INFO

Keywords:
High-performance computing
GPU
CUDA
Corrected explicit-implicit domain decomposition
Convection-diffusion equation

ABSTRACT

In this study, a class of GPU-based corrected explicit-implicit domain decomposition schemes (GCEIDD) is proposed to accelerate the solution of convection-dominated diffusion problems on GPUs. All of these methods take advantage of the fractional steps and corrected explicit-implicit domain decomposition techniques. In each sweep, the domain is decomposed into many strip-divided subdomains. The aim is to reduce the size and increase the number of tri-diagonal systems to provide enough parallelism to keep the GPU occupied. In different steps of the algorithms, we use low-complexity schemes with a high degree of parallelism. The generic form of GCEIDD allows different schemes for the prediction, correction, and one-dimensional implicit solution. We have proposed two methods based on modified upwind and characteristics finite difference and a combined method that bypasses the limitations of these two. Also, important aspects of CUDA programming, which are critical to the implementation of the GCEIDD algorithm, are discussed in detail. For solving tri-diagonal systems, a three-stage strategy is proposed, which creates a balance between occupancy, cache hit rate, and shared memory usage and reduces cache contentions. Results show that GCEIDD has good accuracy and stability even when the number of subdomains is very large. The proposed method can accelerate the solution by a factor of up to 3 compared with the GPU implementation of the classic fractional step methods.

1. Introduction

Convection-diffusion equations are widely used as a mathematical model in important applications such as simulating underground water pollution, oil reservoir simulation, wind flow simulation, global weather prediction, etc. In most of these problems, the convection process dominates the diffusion. Dealing with such problems using the methods like modified upwind or characteristics finite difference, demands a significant computational effort; therefore, many researchers have proposed parallel schemes which are mostly based on the multi-core CPUs.

With hundreds of cores and large memory bandwidth, GPUs offer new opportunities to accelerate Partial Differential Equation (PDE) solvers using fine-grained parallelism. Due to their particular architecture (Single Instruction Multiple Threads: SIMT), GPUs are more effective when the problem can be partitioned into many small tasks performing the same instructions on multiple data. Therefore, fully explicit schemes are very suitable methods to solve PDEs on GPU. However, they impose strict restrictions on the time step size, which is a barrier to the development of fast solvers. On the other hand, fully implicit methods do not impose any restriction on the time step size; however, they are hard to parallelize because the solution at each point depends on the solution at all other points in the entire domain. In these methods, the solution procedure requires the inversion of a large sparse matrix which needs a lot of processing and memory space.

Many researchers have proposed fractional steps and ADI techniques to reduce the complexity and increase the parallelism of the implicit solvers for GPU [1–6]. In these methods, the original higher-dimensional Finite Difference Equation (FDE) is replaced by a series of one-dimensional FDEs. The solution requires the inversion of several independent tri-diagonal matrices which can be performed by using serial or parallel algorithms. In [1], [4], [7], and [8] the Thomas algorithm is used to solve tri-diagonal systems on a GPU. In this method, every independent tri-diagonal system is solved by one thread. The Thomas algorithm has a very low computational complexity and is simple to implement but does not allow partitioning the workload. In [2], [3], [6], [9–11] parallel algorithms like CR, PCR and RD are used. These algo-

* Corresponding author.
E-mail address: zolfaghari@birjand.ac.ir (S.A. Zolfaghari).

https://doi.org/10.1016/j.camwa.2022.08.015
Received 10 August 2021; Received in revised form 5 May 2022; Accepted 5 August 2022
Available online 22 August 2022

rithms create many threads for each tri-diagonal system, but they are very complex due to the high number of arithmetic operations and communication. Some researchers have proposed combined methods like PCR-Thomas [12] and SPIKE-Thomas [13]. In these methods, the original large tri-diagonal system is partitioned into smaller independent systems by using few steps of PCR or the SPIKE algorithm. Then, the Thomas algorithm solves these systems efficiently. Combined methods strike a balance between occupancy and complexity, but the partitioning procedure still can undermine the performance.

As discussed earlier, using parallel tri-diagonal matrix algorithms for solving systems of equations is the mainstream methodology for dealing with implicit numerical schemes on GPUs. Another way of partition the workload is the domain decomposition technique which is a common approach in parallel computing based on multi-core central processing units (CPUs) [14]. This method splits the spatial domain of a problem to smaller subdomains to be solved independently by multiple processors. The main difficulty with domain decomposition is the data communication between adjacent subdomains. Performing an Iterative procedure or using overlapping subdomains, which are proposed in many studies [15–20], have a large computational overhead and require repeated data communication between subdomains. Explicit-implicit domain decomposition methods are among the non-overlapping and non-iterative domain decomposition methods [21–34]. In each time step, first, boundary conditions on the interface of the subdomains are predicted using an explicit scheme and then the solution in the subdomains is computed using an implicit method. Defining a proper boundary condition on the interfaces prevents excessive computation and communication overheads. Therefore, these methods are suitable for the big-scale problems on massively parallel processors like GPUs [24].

In the present study, a class of GPU-based corrected explicit-implicit domain decomposition schemes (GCEIDD) is proposed to accelerate two-dimensional convection-dominated convection-diffusion equations on a GPU. The generic form of the GCEIDD algorithm takes the attractive advantages of the CEIDD and fractional steps method for GPU computing. Similar to the fractional steps methods, the present method performs two sweeps in x and y directions. In each sweep, the domain is decomposed into multiple strip-divided subdomains, and the values on the interfaces are predicted using an explicit scheme. Then, the solution in the subdomains is computed by the one-dimensional implicit scheme. Finally, correction is performed to stabilize the solution and increase the accuracy. The aim of this process is to reduce the size and increase the number of the tri-diagonal systems compared with the traditional fractional steps methods so that the method can provide enough parallelism to keep the GPU occupied. The prediction and correction schemes used in the proposed methods have very low complexity and are easy to parallelize so GCEIDD methods have a significant potential to accelerate the computations. The generic form of GCEIDD allows different schemes for the prediction, correction and one-dimensional implicit solution. We have discussed two methods based on the modified upwind [35] and characteristics finite difference [36] and a combined method that bypasses the limitations of these two.

The paper is organized as follows: Section 2 reports recent works on explicit-implicit domain decomposition methods and explains the generic form of the methods proposed in the present study. Section 3 introduces three different GCEIDD algorithms based on the modified and characteristics finite difference methods. In section 4, important aspects of CUDA programming are introduced, and different implementation strategies are presented. The accuracy, stability, and performance of the GCEIDD method are investigated in Section 5. The final conclusions are given in section 6.

2. GPU-based corrected explicit-implicit domain decomposition

Among different methods, explicit-implicit domain decomposition schemes have a very low computation and communication cost. However, the accuracy and stability of them are highly dependent on the explicit scheme used in the prediction step. In fact, these methods have a severe restriction on the time step size due to stability considerations. To relax the stability conditions, the scheme presented in reference [22] factorizes the predictor and uses a large spatial step size for the interface points. Also, high order explicit schemes and multilevel explicit schemes are proposed in [21]. However, all of these methods are conditionally stable. In corrected explicit-implicit domain decompositions (CEIDD) predicted values of the interfaces are corrected using an implicit scheme which may improve the accuracy and stabilize the explicit predictor [23], [24], [37], [38]. In some studies, researchers have developed CEIDD schemes which use a combination of the previous time levels in the prediction step [25], [27], [34].

In CEIDD, correction step is performed in different ways. When a two-dimensional domain is decomposed into strip-divided subdomains using simple straight-lines, the correction needs an elliptic solver [34], which solves a system of equations along the interface line. For the case of block-divided subdomains (decomposition by using crossover interfaces), using implicit Euler scheme for correction step leads to globalization of data transferring [38], [24]. To overcome this challenge, Sheng et al. [25] proposed two-step correction scheme. In the first step, some tri-diagonal systems are solved to obtain interface values; then, the cross points are treated in an explicit manner. For zigzag-line interfaces proposed in [23], the computation of the corrected values can be performed explicitly, which eliminates the need for an elliptic solver and prevents the globalization of data transferring. However, using zigzag-line interfaces may complicate the implementation of the algorithm and its compatibility with existing codes. Also, the method proposed in [38], uses composite interfaces that consist of straight line and zigzag fractions.

In this study, a class of corrected explicit-implicit domain decomposition schemes is presented. Regardless of the schemes used for the prediction, correction or implicit solution, all of these methods take advantage of the fractional steps technique. Accordingly, they have two consecutive one-dimensional implicit solution steps named sweeps. In each sweep, the domain is decomposed into many small subdomains (Fig. 1). The aim is to reduce the size and increase the number of tri-diagonal systems to provide enough parallelism to keep the GPU occupied. It should be noted that when the domain is decomposed into many small subdomains, a large portion of grid points is interface points. Therefore, the complexity, parallelism, and accuracy of the prediction and correction schemes are a matter of great importance.

In GCEIDD the position of the subdomains changes in each sweep. Therefore, the tri-diagonal systems can be decomposed in both sweeps. Changing the position of the subdomain is not efficient in parallel computing with distributed memory, but does not impose any challenges to GPU computing. High-order or multilevel prediction schemes, which are used in the previous works, result in a heavy computational load. In the present research, we use low-complexity schemes with a high degree of parallelism. In the correction step, we use an interpolation scheme which eliminates the need for an elliptic solver or zigzag-line interfaces. This method is easy to parallelize and has a low computational cost.

In the present research, the GCEIDD approach is introduced as an alternative to parallel tri-diagonal matrix algorithms which are the mainstream methodology for dealing with implicit numerical schemes on GPUs. Parallel algorithms like CR, PCR, RD etc. are suitable for GPU computing because they create many threads for each tri-diagonal system, but they are very complex due to the high number of arithmetic operations and communication. In GCEIDD the number of the independent tri-diagonal systems is large enough to provide enough parallelism for GPU. Therefore, the serial Thomas algorithm can be used for solving them. Compared to the parallel algorithms, the Thomas algorithm performs fewer number of arithmetic operations. Therefore, GCEIDD not only provide enough parallelism, but also does less work per step.

The generic GCEIDD scheme is described below:

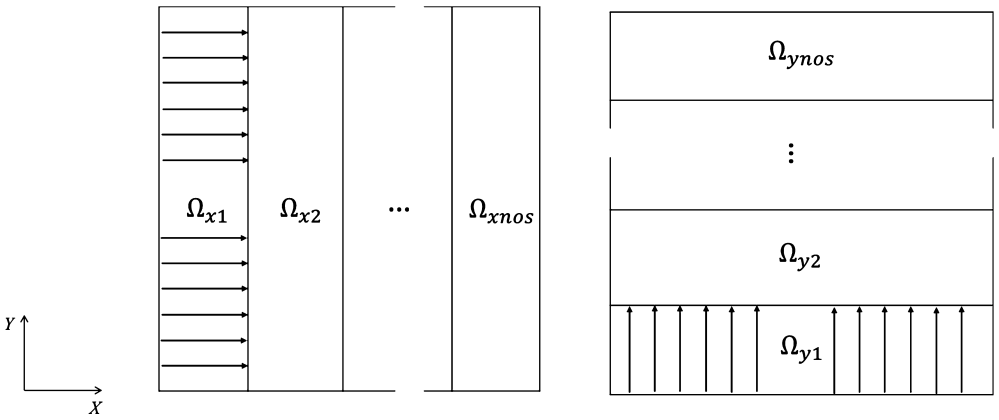

Fig. 1. Decomposing the domain Ω into several equal subdomains in the x-direction (left) and y-direction (right). Arrows represent the tri-diagonal systems.

Consider the convection-diffusion equation:

$$\begin{cases} \frac{\partial u(x,t)}{\partial t} = Lu(x,y,t) + f(x,y,t) & (x,y) \in \Omega \quad t \geq 0 \\ u(x,y,t) = u_b(x,y,t) & (x,y) \in \partial\Omega \ \ t \geq 0 \\ u(x,y,0) = u^0(x,y) & (x,y) \in \Omega \end{cases} \tag{1}$$

The domain is $\Omega \subset R^2$ with boundary $\partial\Omega$. L is the spatial differential operator:

$$L = \frac{\partial}{\partial x}\left(a_1(x,y)\frac{\partial}{\partial x} - b_1(x,y)\right) + \frac{\partial}{\partial y}\left(a_2(x,y)\frac{\partial}{\partial y} - b_2(x,y)\right), \tag{2}$$

where b_1 and b_2 are the components of the velocity field and a_1 and a_2 are the diffusion coefficients.

We choose a discrete spatial grid Ω_h. In the discrete domain, $\partial\Omega_h$ is defined as $\partial\Omega \cap \Omega_h$.

The GCEIDD algorithm advances from time n to n + 1 in two steps named y-sweep and x-sweep:

Step 1- x-sweep: The domain is decomposed into *nos* equal strip-divided subdomains in x direction $\Omega_{x1}, \Omega_{x2}, \Omega_{x3} \ldots, \Omega_{xnos}$ with interface boundaries denoted by Γ_x. The union of the subdomains is denoted by Γ_x^c, so $\Omega = \Gamma_x \cup \Gamma_x^c$. In the discrete domain Γ_{x_h} and $\Gamma_{x_h}^c$ are defined as $\Gamma_x \cap \Omega_h$ and $\Gamma_x^c \cap \Omega_h$.

- **Sub-step 1-1:** Prediction: Compute the predicted values $\tilde{u}^{n+1/2}$ at Γ_{x_h} using an explicit scheme.
- **Sub-step 1-2:** Implicit solution: Compute the $u^{n+1/2}$ at Γ_x^c using a scheme which is implicit in x direction. In the implicit scheme, $u^{n+1/2}$ is replaced by $\tilde{u}^{n+1/2}$ at the interface points.
- **Sub-step 1-3:** Correction: Compute the corrected values $u^{n+1/2}$ at Γ_{x_h} in an explicit manner.

Step 2- y-sweep: The domain is decomposed into *nos* equal strip-divided subdomains in y direction $\Omega_{y1}, \Omega_{y2}, \Omega_{y3} \ldots, \Omega_{ynos}$ with interface boundaries denoted by Γ_y. The union of the subdomains is denoted by Γ_y^c, so $\Omega = \Gamma_y \cup \Gamma_y^c$. In the discrete domain Γ_{y_h} and $\Gamma_{y_h}^c$ are defined as $\Gamma_y \cap \Omega_h$ and $\Gamma_y^c \cap \Omega_h$.

- **Sub-step2-1:** Prediction: Compute the predicted values $\tilde{u}^{n+1}$ at Γ_{y_h} using an explicit scheme.
- **Sub-step2-2:** Implicit solution: Compute the u^{n+1} at Γ_y^c using a scheme which is implicit in y direction. In the implicit scheme, u^{n+1} is replaced by $\tilde{u}^{n+1}$ at the interface points.
- **Sub-step2-3:** Correction: Compute the corrected values u^{n+1} at Γ_{y_h} in an explicit manner.

2.1. MUGCEIDD: modified upwind GCEIDD

Modified upwind is a finite difference scheme which is widely used for numerical solution of the convection-dominated problems [35]. This method prevents the non-physical oscillations by simulating the direction of the information propagation. However, to obtain accurate solutions, this method requires very fine grid and small time step size which leads to a heavy computational load for large scale problems [27]. Therefore, this is very helpful to propose a parallel scheme to accelerate the solution procedure.

The generic form of the GCEIDD algorithm allows different choices for the prediction, correction and implicit solution steps. In the present section the modified upwind GCEIDD is proposed which uses a combination of previous time levels for the prediction, a modified upwind scheme for the implicit solution, and an interpolation scheme for the correction step.

For simplicity, consider a rectangular domain $\Omega = (0,1) \times (0,1)$. The domain is discretized uniformly and the grid points are $x_i = i \times h$, $y_j = j \times h$ where $h = 1/J$ for some integer J. The time step size is $\tau = T/N$ where T is the final time and N is some integer. Also, $t^n = n\tau$ is the discrete time level and $u_{i,j}^n = u(x_i, y_j, t^n)$.

In order to construct a fractional steps scheme, the following differential operators are introduced:

$$L_x = \frac{\partial}{\partial x}\left(a_1(x,y)\frac{\partial u}{\partial x} - b_1(x,y)\right), L_y = \frac{\partial}{\partial y}\left(a_2(x,y)\frac{\partial u}{\partial y} - b_2(x,y)\right). \tag{3}$$

Also, for the source term we have:

$$f_1(x,y,t) = f_2(x,y,t) = \frac{1}{2}f(x,y,t). \tag{4}$$

Using the fractional steps scheme, the time semi-discretization solution $\hat{u}^{n+1}$ for equation (1) is defined as follows:

$$(I - \Delta t L_x)\hat{u}^{n+\frac{1}{2}} = \hat{u}^n + \Delta t f_1(x,y,t) \tag{5}$$

and

$$(I - \Delta t L_y)\hat{u}^{n+1} = \hat{u}^{n+\frac{1}{2}} + \Delta t f_2(x,y,t) \tag{6}$$

For any grid function ϕ, $\phi_{i,j}^n = \phi(x_i, y_j, t^n), \phi_{i+\frac{1}{2},j}^n = \phi\left(\frac{x_i + x_{i+1}}{2}, y_j, t^n\right)$ and $\phi_{i,j+\frac{1}{2}}^n = \phi\left(x_i, \frac{y_j + y_{j+1}}{2}, t^n\right)$. Let $U_{i,j}^n$ be the numerical approximation to the exact solution at grid point (x_i, y_j) and time t^n. To define the discrete operators $L_{y,h}$ and $L_{x,h}$ which corresponds to the discretization of the operators L_y and L_x, firstly, we introduce the following notations:

$$\delta_x U_{i-\frac{1}{2},j}^n = \frac{1}{h}\left(U_{i,j}^n - U_{i-1,j}^n\right) \quad \delta_y U_{i,j-\frac{1}{2}}^n = \frac{1}{h}\left(U_{i,j}^n - U_{i,j-1}^n\right) \tag{7}$$

Also, for the modified upwind method, let:

$$a^*_{1i-\frac{1}{2},j} = \frac{2a^2_{1i-\frac{1}{2},j}}{2a^2_{1i-\frac{1}{2},j} + h|\mathrm{b}_{1i-\frac{1}{2},j}|} \quad a^*_{2i,j-\frac{1}{2}} = \frac{2a^2_{2i,j-\frac{1}{2}}}{2a^2_{2i,j-\frac{1}{2}} + h|\mathrm{b}_{2i,j-\frac{1}{2}}|} \tag{8}$$

For the diffusive terms, we have:

$$\delta^2_{x,a_1^*}U^n_{i,j} = \frac{1}{h}\left(a^*_{1i+\frac{1}{2},j}\delta_x U^n_{i+\frac{1}{2},j} - a^*_{1i-\frac{1}{2},j}\delta_x U^n_{i-\frac{1}{2},j}\right)$$
$$\delta^2_{y,a_2^*}U^n_{i,j} = \frac{1}{h}\left(a^*_{2i,j+\frac{1}{2}}\delta_y U^n_{i,j+\frac{1}{2}} - a^*_{2i,j-\frac{1}{2}}\delta_y U^n_{i,j-\frac{1}{2}}\right) \tag{9}$$

And the convective terms are discretized as follows:

$$\delta_x\left(\mathrm{b}_1 U^{n,ux}\right)_{i,j} = \frac{1}{h}\left(\mathrm{b}_{1i+\frac{1}{2},j}U^{n,ux}_{i+\frac{1}{2},j} - \mathrm{b}_{1i-\frac{1}{2},j}U^{n,ux}_{i-\frac{1}{2},j}\right)$$
$$\delta_y\left(\mathrm{b}_2 U^{n,ux}\right)_{i,j} = \frac{1}{h}\left(\mathrm{b}_{2i,j+\frac{1}{2}}U^{n,ux}_{i,j+\frac{1}{2}} - \mathrm{b}_{2i,j-\frac{1}{2}}U^{n,ux}_{i,j-\frac{1}{2}}\right) \tag{10}$$

Where

$$U^{n,ux}_{i+\frac{1}{2},j} = \sigma(\mathrm{b}_{1i+\frac{1}{2},j})U^n_{i,j} + \left(1-\sigma(\mathrm{b}_{1i+\frac{1}{2},j})\right)U^n_{i+1,j}$$
$$U^{n,ux}_{i,j+\frac{1}{2}} = \sigma(\mathrm{b}_{2i,j+\frac{1}{2}})U^n_{i,j} + \left(1-\sigma(\mathrm{b}_{2i,j+\frac{1}{2}})\right)U^n_{i,j+1} \tag{11}$$

The piecewise function σ is defined by:

$$\sigma(x) = \begin{cases} 1 & x \geq 0 \\ 0 & x < 0 \end{cases}$$

Then, the discretization of the operators L_y and L_x are defined as follows:
for each y_i:

$$L_{x,h}U^n_{i,j} = \delta^2_{x,a_1^*}U^n_{i,j} - \delta_x\left(b_1 U^{n,ux}\right)_{i,j} \tag{12}$$

for each x_i:

$$L_{y,h}U^n_{i,j} = \delta^2_{y,a_2^*}U^n_{i,j} - \delta_y\left(\mathrm{b}_2 U^{n,ux}\right)_{i,j} \tag{13}$$

Using these notations, we can now present the modified upwind GCEIDD (MUGCEIDD). Numerical schemes used for different parts of the algorithm are as follows:
Step 1- x-sweep:

- **Sub-step 1-1:** The values on the interface points are predicted using data from the previous x-sweep step:

$$\tilde{U}^{n+1/2} = U^n_{i,j} + \left(U^{n-1/2}_{i,j} - U^{n-1}_{i,j}\right) \tag{14}$$

- **Sub-step 1-2:** The solution at the inner points of the subdomains is calculated using the x-directional modified upwind implicit scheme:

$$\frac{U^{n+1/2}_{i,j} - U^n_{i,j}}{\tau} = \delta^2_{x,a_1^*}U^{n+1/2}_{i,j} - \delta_x\left(\mathrm{b}_1 U^{n+1/2,ux}\right)_{i,j} + f^{n+1}_{1i,j} \tag{15}$$

- **Sub-step 1-3:** Corrected values at the interface points are computed using an interpolation scheme:

$$U^{n+1/2}_{i,j} = \frac{2}{3}\left(U^{n+1/2}_{i+1,j} + U^{n+1/2}_{i-1,j}\right) - \frac{1}{6}\left(U^{n+1/2}_{i+2,j} + U^{n+1/2}_{i-2,j}\right) \tag{16}$$

Step 2- y-sweep:

- **Sub-step 2-1:** The values on the interface points are predicted using data from the previous y-sweep step:

$$\tilde{U}^{n+1} = U^{n+1/2}_{i,j} + \left(U^n_{i,j} - U^{n-1/2}_{i,j}\right) \tag{17}$$

- **Sub-step 2-2:** The solution at the inner points of the subdomains is calculated using the y-directional modified upwind implicit scheme:

$$\frac{U^{n+1}_{i,j} - U^{n+1/2}_{i,j}}{\tau} = \delta^2_{y,a_2^*}U^{n+1}_{i,j} - \delta_y\left(\mathrm{b}_2 U^{n+1,ux}\right)_{i,j} + f^{n+1}_{2i,j} \tag{18}$$

- **Sub-step 2-3:** Corrected values at the interface points are computed using an interpolation scheme:

$$U^{n+1}_{i,j} = \frac{2}{3}\left(U^{n+1}_{i,j+1} + U^{n+1}_{i,j-1}\right) - \frac{1}{6}\left(U^{n+1}_{i,j+2} + U^{n+1}_{i,j-2}\right) \tag{19}$$

2.2. CFDGCEIDD: characteristic finite difference GCEIDD

Many studies have proposed characteristic finite difference (CFD) methods to overcome the numerical dissipations and non-physical oscillations which arise in the convection-dominated problems [36], [39–44]. In the CFD methods, traditional finite difference is used for diffusion terms and the convection terms are treated by the method of characteristics. This method reduces the time truncation error and allows larger time step sizes compared with the traditional finite difference methods [36].

Now, we construct a fractional step scheme for the convection-diffusion equation which employs the method of characteristics for the convection procedure. To solve the problem (1), we solve two subproblems sequentially in subinterval $[t^n, t^{n+1}]$:

$$\frac{\partial u^*}{\partial t} = -b_1\frac{\partial u^*}{\partial x} + \frac{\partial}{\partial x}\left(a_1\frac{\partial u^*}{\partial x}\right) + f_1 \quad u^*(x,y,t^n) = u(x,y,t^n), \quad t \in \left[t^n, t^{n+1}\right] \tag{20}$$

and

$$\frac{\partial u}{\partial t} = -b_2\frac{\partial u}{\partial y} + \frac{\partial}{\partial y}\left(a_2\frac{\partial u}{\partial y}\right) + f_2 \quad u(x,y,t^n) = u^*(x,y,t^{n+1}) \quad t \in \left[t^n, t^{n+1}\right] \tag{21}$$

To employ the CFD method let $\psi_1 = \sqrt{1+b_1^2}$, and $\tau_1 = \tau_1(x)$ be a unit vector along $(1, b_1)$ which is the characteristic direction of the hyperbolic equation $u_t + b_1 u_x = 0$. The derivative along the τ can be written as:

$$\frac{d}{d\tau_1} = \frac{1}{\psi_1}\frac{\partial}{\partial t} + \frac{b_1}{\psi_1}\frac{\partial}{\partial x} \tag{22}$$

Then the equation (20) can be written in the equivalent form:

$$\psi_1\frac{du^*}{d\tau_1} = \frac{\partial}{\partial x}\left(a_1\frac{\partial u^*}{\partial x}\right) + f_1 \tag{23}$$

Similarly, let $\psi_2 = \sqrt{1+b_2^2}$, and $\tau_2 = \tau_2(y)$ be a unit vector along $(1, b_2)$. Then the equation (21) can be written in the equivalent form:

$$\psi_2\frac{du}{d\tau_2} = \frac{\partial}{\partial y}\left(a_2\frac{\partial u}{\partial y}\right) + f_2 \tag{24}$$

The characteristic derivatives on the left hand sides of equation (23) can be approximated as follows:

$$\begin{aligned}\psi_1\frac{du^*(x,y,t)}{d\tau_1} &= \psi_1\frac{u^*(x,y,t) - u^*(x-b_1\Delta t, y, t-\Delta t)}{\sqrt{(x-\overline{x})^2 + (\Delta t)^2}} + O(\Delta t) \\ &= \frac{u^*(x,y,t) - u^*(x-b_1\Delta t, y, t-\Delta t)}{\Delta t} + O(\Delta t)\end{aligned} \tag{25}$$

Then, we have:

$$\psi_1\frac{d}{d\tau_1}\left(U^{*n+1}_{i,j}\right) = \frac{U^{*n+1}_{i,j} - \overline{U^*}^n_{i,j}}{\Delta t} + O(\Delta t), \tag{26}$$

and similarly for equation (24), we have:

$$\psi_2\frac{d}{d\tau_2}\left(U^{n+1}_{i,j}\right) = \frac{U^{n+1}_{i,j} - \overline{U}^n_{i,j}}{\Delta t} + O(\Delta t), \tag{27}$$

where $\overline{U^*}^n_{i,j} = U^*(x_i - b_{1i,j}\Delta t, y_j, t^{n-1})$ and $\overline{U}^n_{i,j} = U(x_i, y_j - b_{2i,j}\Delta t, t^{n-1})$. If the locations $x_i - b_{1i,j}\Delta t$ and $y_j - b_{2i,j}\Delta t$ does not coincide with any point in the grid, $\overline{U^*}^n_{i,j}$ and $\overline{U}^n_{i,j}$ can be approximated using a linear interpolation.

For the diffusive term on the right hand side of the equations (20) and (21) the following notations are introduced:

$$\delta^2_{x,a_1}U^{*n}_{i,j} = \frac{1}{h}\left(a_{1i+\frac{1}{2},j}\delta_x U^n_{i+\frac{1}{2},j} - a_{1i-\frac{1}{2},j}\delta_x U^n_{i-\frac{1}{2},j}\right),$$
$$\delta^2_{y,a_2}U^n_{i,j} = \frac{1}{h}\left(a_{2i,j+\frac{1}{2}}\delta_x U^n_{i,j+\frac{1}{2}} - a_{2i,j-\frac{1}{2}}\delta_x U^n_{i,j-\frac{1}{2}}\right). \tag{28}$$

With these notations, the equations (20) and (21) can be presented in the following discrete form:
for each y_i:

$$\frac{U^{*n+1}_{i,j} - \overline{U^{*}}^{n}_{i,j}}{\Delta t} = \delta^2_{x,a_1} U^{*n+1}_{i,j} + f_{1,i,j}, \quad U^{*n}_{i,j} = U^{n}_{i,j} \tag{29}$$

for each x_j:

$$\frac{U^{n+1}_{i,j} - \overline{U}^{n}_{i,j}}{\Delta t} = \delta^2_{y,a_2} U^{n+1}_{i,j} + f_{2,i,j}, \quad U^{n}_{i,j} = U^{*n+1}_{i,j} \tag{30}$$

we rearrange the equation (29) and (30) and use the intermediate grid function $U^{n+1/2}_{i,j}$ to match the notation with the generic GCEIDD algorithm:
for each y_i:

$$\frac{U^{n+1/2}_{i,j} - \overline{U}^{n}_{i,j}}{\Delta t} = \delta^2_{x,a_1} U^{n+1/2}_{i,j} + f_{1,i,j} \tag{31}$$

for each x_j:

$$\frac{U^{n+1}_{i,j} - \overline{U}^{n+1/2}_{i,j}}{\Delta t} = \delta^2_{y,a_2} U^{n+1}_{i,j} + f_{2,i,j} \tag{32}$$

Having the definitions above, now we can introduce the characteristics finite difference GCEIDD scheme (CFDGCEIDD):
Step 1- x-sweep:

- **Sub-step 1-1:** The values on the interface points are predicted using the following explicit scheme:

$$\frac{\tilde{U}^{n+1/2}_{i,j} - \overline{U}^{n}_{i,j}}{\Delta t} = \delta^2_{x,a_1} U^{n}_{i,j} + f_{1,i,j} \tag{33}$$

- **Sub-step 1-2:** The solution at the inner points of the subdomains is calculated using the x-directional implicit scheme:

$$\frac{U^{n+1/2}_{i,j} - \overline{U}^{n}_{i,j}}{\Delta t} = \delta^2_{x,a_1} U^{n+1/2}_{i,j} + f_{1,i,j} \tag{34}$$

- **Sub-step 1-3:** Corrected values at the interface points are computed using an interpolation scheme:

$$U^{n+1/2}_{i,j} = \frac{2}{3}\left(U^{n+1/2}_{i+1,j} + U^{n+1/2}_{i-1,j}\right) - \frac{1}{6}\left(U^{n+1/2}_{i+2,j} + U^{n+1/2}_{i-2,j}\right) \tag{35}$$

Step 2- y-sweep:

- **Sub-step 2-1:** The values on the interface points are predicted using the following explicit scheme:

$$\frac{\tilde{U}^{n+1}_{i,j} - \overline{U}^{n+\frac{1}{2}}_{i,j}}{\Delta t} = \delta^2_{y,a_2} U^{n+\frac{1}{2}}_{i,j} + f_{2,i,j} \tag{36}$$

- **Sub-step 2-2:** The solution at the inner points of the subdomains is calculated using the y-directional implicit scheme:

$$\frac{U^{n+1}_{i,j} - \overline{U}^{n+\frac{1}{2}}_{i,j}}{\Delta t} = \delta^2_{y,a_2} U^{n+1}_{i,j} + f_{2,i,j} \tag{37}$$

- **Sub-step 2-3:** Corrected values at the interface points are computed using an interpolation scheme:

$$U^{n+1}_{i,j} = \frac{2}{3}\left(U^{n+1}_{i,j+1} + U^{n+1}_{i,j-1}\right) - \frac{1}{6}\left(U^{n+1}_{i,j+2} + U^{n+1}_{i,j-2}\right) \tag{38}$$

2.3. CPGCEIDD: characteristics prediction GCEIDD

Although modified upwind and characteristics finite difference reduce the errors in convection-dominated problems, both have deficiencies in terms of computational complexity. The modified upwind method overcomes the non-physical oscillations, however, it still needs very fine grid and small time step size to obtain accurate solutions [27]. On the other hand, in the characteristics finite difference method, for near boundary points, characteristics lines may go through the boundaries and thus the points $x_i - b_{1i,j}\Delta t$ and $y_j - b_{2i,j}\Delta t$ lie outside the domain Ω. In these cases, we need some assumptions to extrapolate the fields beyond the boundaries [39], [43], [45]. Obviously, this complicates the calculations and causes errors. Therefore, in many studies the time step size is restricted to ensure that the characteristics lines do not go through the boundaries [41], [45], [46].

In the present section a combined method is proposed which takes advantage of both the modified upwind and characteristics finite difference methods. In this method, the prediction of interface values is performed using the characteristics finite difference method and the inner subdomains are solved using the modified upwind scheme. As interface points are far from the boundaries, in the present method, it is less likely that characteristics lines go through the boundaries. Therefore, we can choose larger time steps without adding any complexity to the computation.

To present the characteristics prediction GCEIDD (CPGCEIDD) algorithm, we use the notations already introduced in sections 2.1 and 2.2:
Step 1- x-sweep:

- **Sub-step 1-1:** The values on the interface points are predicted using the following explicit scheme:

$$\frac{\tilde{U}^{n+1/2}_{i,j} - \overline{U}^{n}_{i,j}}{\Delta t} = \delta^2_{x,a_1} U^{n}_{i,j} + f_{1,i,j} \tag{39}$$

- **Sub-step 1-2:** The solution at the inner points of the subdomains is calculated using the x-directional implicit scheme:

$$\frac{U^{n+1/2}_{i,j} - U^{n}_{i,j}}{\tau} = \delta^2_{x,a^*_1} U^{n+1/2}_{i,j} - \delta_y\left(\mathrm{b}_1 U^{n+1/2,ux}\right)_{i,j} + f^{n+1}_{1i,j} \tag{40}$$

- **Sub-step 1-3:** Corrected values at the interface points are computed using an interpolation scheme:

$$U^{n+1/2}_{i,j} = \frac{2}{3}\left(U^{n+1/2}_{i+1,j} + U^{n+1/2}_{i-1,j}\right) - \frac{1}{6}\left(U^{n+1/2}_{i+2,j} + U^{n+1/2}_{i-2,j}\right) \tag{41}$$

Step 2- y-sweep:

- **Sub-step 2-1:** The values on the interface points are predicted using the following explicit scheme:

$$\frac{\tilde{U}^{n+1}_{i,j} - \overline{U}^{n+\frac{1}{2}}_{i,j}}{\Delta t} = \delta^2_{y,a_2} U^{n+\frac{1}{2}}_{i,j} + f_{2,i,j} \tag{42}$$

- **Sub-step 2-2:** The solution at the inner points of the subdomains is calculated using the y-directional implicit scheme:

$$\frac{U^{n+1}_{i,j} - U^{n+1/2}_{i,j}}{\tau} = \delta^2_{y,a^*_2} U^{n+1}_{i,j} - \delta_y\left(\mathrm{b}_2 U^{n+1,ux}\right)_{i,j} + f^{n+1}_{2i,j} \tag{43}$$

- **Sub-step 2-3:** Corrected values at the interface points are computed using an interpolation scheme:

$$U^{n+1}_{i,j} = \frac{2}{3}\left(U^{n+1}_{i,j+1} + U^{n+1}_{i,j-1}\right) - \frac{1}{6}\left(U^{n+1}_{i,j+2} + U^{n+1}_{i,j-2}\right) \tag{44}$$

3. GPU architecture

GPUs are many-core processors with a high peak computational performance and memory bandwidth. These processors are optimized for

problems with high level of data parallelism and simple control flow. A GPU cannot work as a standalone processor and must be hosted by a CPU-based system. GPU and its memory are called the device and CPU and its memory is called the host. In the present study, we use a Quadro K2200 hosted by a system with a core i7 CPU and a 16 GB DDR3 RAM. Quadro K2200 uses Maxwell architecture and has a compute capability of 5.0. It has 5 streaming multiprocessors (SMs) and a 4 GB DRAM (device memory). Each SM contains 128 single precision CUDA cores, 64 double-precision units and 32 special function units. Also, there is a 64 KB shared memory, 64K 32-bit registers, and 64 KB L1 cache per SM. The L2 cache capacity is 2 MB which is shared by all the SMs. The local, constant, texture and global memory reside on the device DRAM.

The Programming platform used in the present study is CUDA. The heterogeneous CUDA C++ program consists of serial and parallel components which are executed on CPU and GPU, respectively. The parallel functions written for GPUs are called kernel. When a kernel is called from the host, a large number of threads are generated which execute the same instructions on different elements of the arrays involved. Threads are organized in the same-sized groups called block. A whole block resides in one SM and the compute resources of the SMs including the registers and shared memory are partitioned between resident blocks and threads. Threads of a block are divided into groups of 32 threads called warps. If the size of the block is not an even multiple of 32, some threads in the last warp would be inactive. All of the threads within a warp execute the same instruction. They all start together and then each thread has its own execution path.

Another important feature of the CUDA programming model is the exposed memory hierarchy. There are different types of programmable memory with different sizes, speeds, and scopes. “Global memory” is the largest and the slowest memory on a GPU. This memory is visible to all the threads, and all of the input data is copied from host to this memory at the start of the computation. A faster memory on a GPU is “shared memory”, which is accessible to threads within a block. Shared memory is used when several threads within a block read/write a variable several times during kernel execution. “Register memory” is the fastest memory on a GPU and is private to a thread. Generally automatic variables in a kernel are stored in register memory. Furthermore, there are several types of memories including L1 and L2 caches which are non-programmable. There is no explicit control on these memories but with a good memory access pattern they can be efficiently utilized to improve the performance.

Deep understanding of CUDA execution and memory model can help us to design more efficient kernels. In the following, some important aspects of CUDA programming are introduced which are related to implementing the GCEIDD algorithm:

Occupancy

A GPU uses the thread-level parallelism to enhance the utilization of its resources. A large number of threads can help the GPU to maximize throughput and hide memory latencies. The numerical method proposed in this study provides a high degree of data-parallelism. This means, from a logical point of view, there are many threads that can run in parallel in most parts of the computation. But, not all of these threads reside on the SMs at the same time. In compute capability of 5.0, the maximum number of resident threads per SM is 2048 (64 warps). However, the number of active threads can be further limited by execution configuration and resource usage. The ratio of upper limit for number of active warps on an SM to the maximum number of resident warps supported by SM (64 warps) is called theoretical occupancy.

The factors that affect the theoretical occupancy are as follows: 1) In compute capability of 5.0 the maximum number of blocks per SM is 32. Therefore, if the number of threads per block is set to 2048/32 = 64 or bigger, the occupancy can reach to 1. For block sizes less than 64 the occupancy will necessarily be less than 1. 2) The maximum number of 32-bit registers per SM is 64K. In order to achieve the occupancy of 1, the number of registers per thread must be limited to 64*1024/2048 = 32 or less. 3) The maximum amount of shared memory per SM is 64 KB. For instance, if a kernel is executed with blocks each of them needing 2 KB of shared memory, then 64 KB/2 KB=32 blocks can reside in each multiprocessor.

The theoretical occupancy of a kernel may be limited by one of the above factors and it can be improved by adjusting relevant configuration entries and implementation strategies. It should be noted that theoretical occupancy shows the upper limit for the number of active threads which may not be achieved if the total number of launched threads is less than this number. Therefore, in many cases, like the ones presented in this study, the achieved occupancy is far less than the theoretical occupancy. Another important point is that although in many applications increasing the number of active threads is a way of improving the performance, some kernels perform better with fewer number of active warps. This is the case for kernels that exhibit intra-thread locality like the one design for solving tri-diagonal systems in the present study.

Coalesced memory access

Most GPU applications are bandwidth bound and optimizing the memory bandwidth utilization has a significant effect on their performance. In CUDA, memory operations are issued per warp. Therefore, when executing a memory instruction, all the memory addresses provided by threads within a warp are accessed by a single request. In QUADRO K2200 all global memory requests go through the L2 cache. When the data is not found in L2, the requests are serviced by DRAM with 32-byte transactions. Only the aligned 32-byte segments can be accessed by the memory transactions. The number of transactions is equal to the number of 32-byte segments necessary to service all of the warp threads. In the example illustrated in Fig. 2a, memory accesses have inter-thread locality. This means adjacent threads in a warp, access locations in the memory which are contiguous to each other. In such cases, the number of memory transactions is minimized which leads to an efficient bandwidth utilization. This efficient pattern for reading/writing the memory is called coalesced memory access. In the example illustrated in Fig. 2b, the memory locations accessed by adjacent threads are 16*4 bytes apart. Therefore, memory accesses can not be coalesced and a large number of transactions are needed to access the required data. In this case, only a small amount of the transferred data is used and the bandwidth is wasted. In many applications, access pattern can be improved by memory rearrangement and choosing more suitable strategies for mapping threads to data.

Cache contention reduction

As explained above, in the example illustrated in Fig. 2b, a whole 32-byte segment is brought into the cache for each thread. Although all the bytes read into L2 by a certain access are not used in that access, they may remain in the cache and can be used (and reused) by the same thread in the subsequent computations. This is intra-thread locality. Exploiting the intra-thread localities is very important as it prevents the costly accesses to the slow DRAM especially in the case of non-coalesced access pattern.

To exploit localities, cache contention must be avoided. The cache contention occurs when a large number of threads with large working sets reside on SM. In this situation, due to lack of space, cache lines allocated for one thread may get evicted by other threads before any additional reuse. In the present study, throttling technique is used to reduce cache contention. This method reduces the cache contention by limiting the number of active thread groups. In the present study, we use software thread throttling [47], which reduces the number of resident threads by modifying the source code.

4. Implementation

The generic form of GCEIDD algorithm provides a high degree of data-parallelism in different steps of domain decomposition procedure.

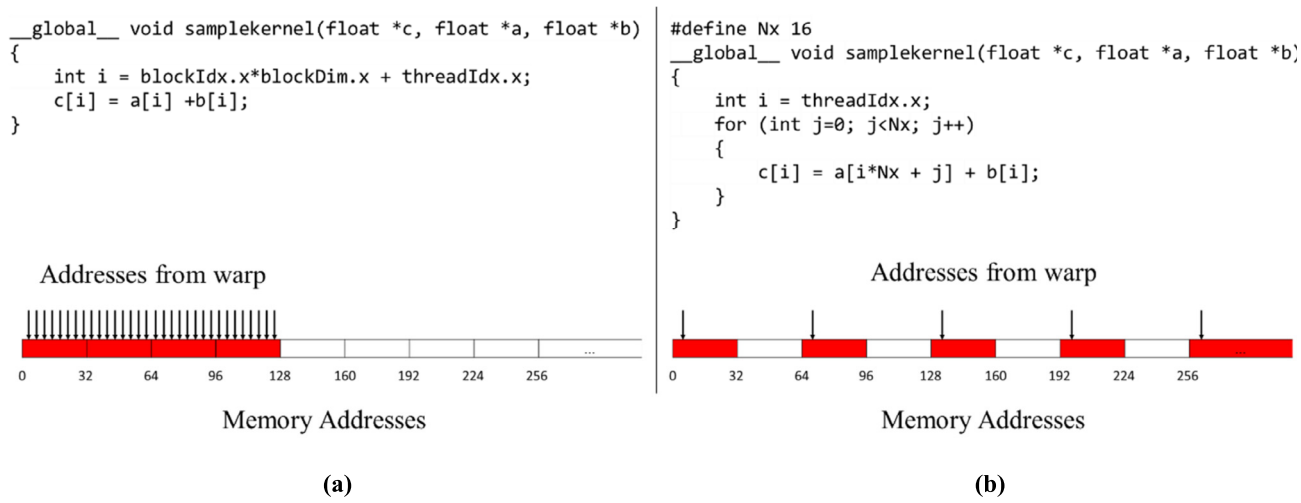

Fig. 2. Memory transactions when threads within warp access contiguous locations in memory (a) and when threads within warp access non-contiguous locations in memory (b).

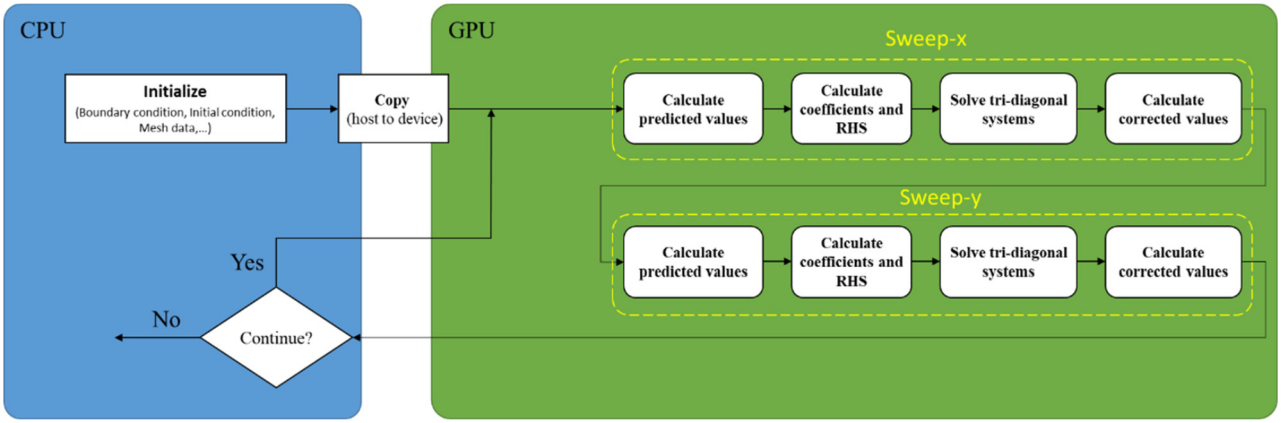

Fig. 3. Graphical view of the GCEIDD code.

The workload related to the explicit prediction, correction and the solution of the independent tri-diagonal systems can be partitioned among many threads with minimal programming efforts. However, to achieve the best performance, different implementation strategies and parameter settings are analyzed in the present research.

The structure of the code developed in this study is summarized in Fig. 3. Initializing the fields and calculating the geometry and mesh variables are assigned to the CPU. Then, all the necessary data is transferred to device memory and the time advancement procedure is performed on the GPU. Until the final time step, there is no data transfer from device to host.

On device memory, all the multi-dimensional data, including fields data (dev_U), matrix diagonals (dev_a, dev_b, and dev_c), and right-hand side (dev_d), are mapped to one-dimensional arrays. Rows of the two-dimensional arrays are sequentially aligned in one-dimensional arrays. So, for a grid size of $N_x \times N_y$, the array $A_{i,j} = dev\_A[k]$, which $k = i + j \times N_x$.

As mentioned before, time advancement procedure consists of two sweep steps, each containing three sub-steps: prediction, implicit solution and correction. The structure of the code presented here contains four kernels. Two kernels for the prediction and correction and two for the implicit solution inside the subdomains. For the implicit solution, one kernel is designed to calculate coefficients and right hand side of the tri-diagonal systems and another one is designed to solve the systems. The way of implementing these kernels is presented in the following.

Prediction

In different variants of GCEIDD, the predicted values for interface points are calculated using an explicit scheme. Therefore, the computation related to each interface point is assigned to one thread and the results are stored in global memory. As different GCEIDD methods use different prediction schemes, the workload of this part of the computation varies in different methods. It is worth mentioning that the MUGCEIDD methods require to store the values of one previous time level. Therefore, the workload of this operation is also considered in the overall workload of the prediction step.

Calculating the coefficients and the right hand side

For calculating the coefficients and right-hand side (RHS) of the tri-diagonal systems, each thread is mapped to one point inside the subdomains and calculates the coefficients and RHS of the equation corresponding to that point. In this kernel, each thread uses a large amount of registers, thus limiting the number of resident threads. Maximum number of registers per thread can be manually controlled by the `maxrregcount` compiler option. Although restricting the number of registers improve the occupancy, if there are not enough registers to hold all the variables in use, some variables may be moved to slow local memory. To strike a balance between occupancy and memory access speed, optimal value for maximum number of registers can be decided by a profile-driven optimization approach using the CUDA profiler. As for the prediction part, the workload of calculating the coefficients and the right hand sides varies in different GCEIDD methods.

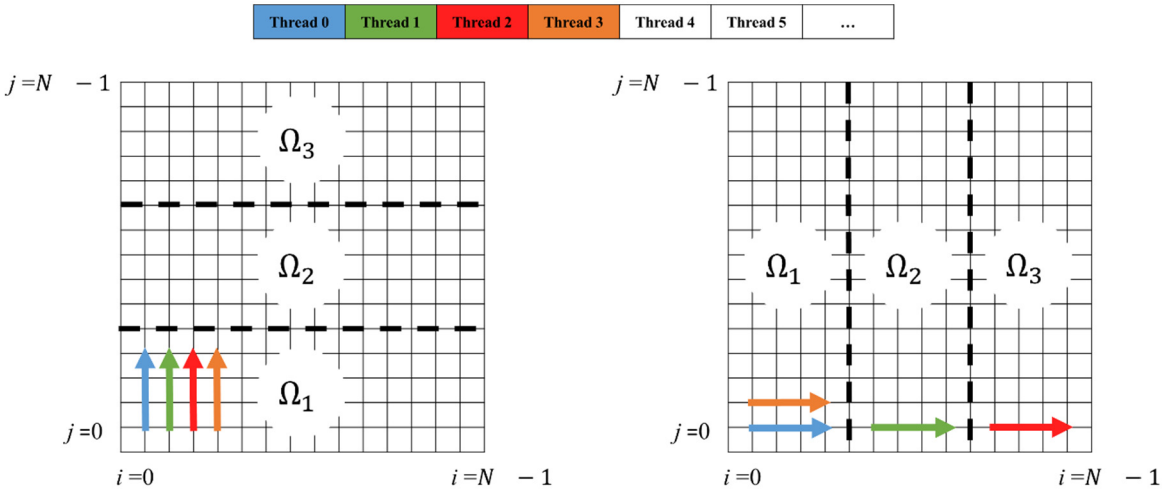

Fig. 4. The correspondence between threads and fields data for sweep in y-direction (left) and sweep in x-direction (right).

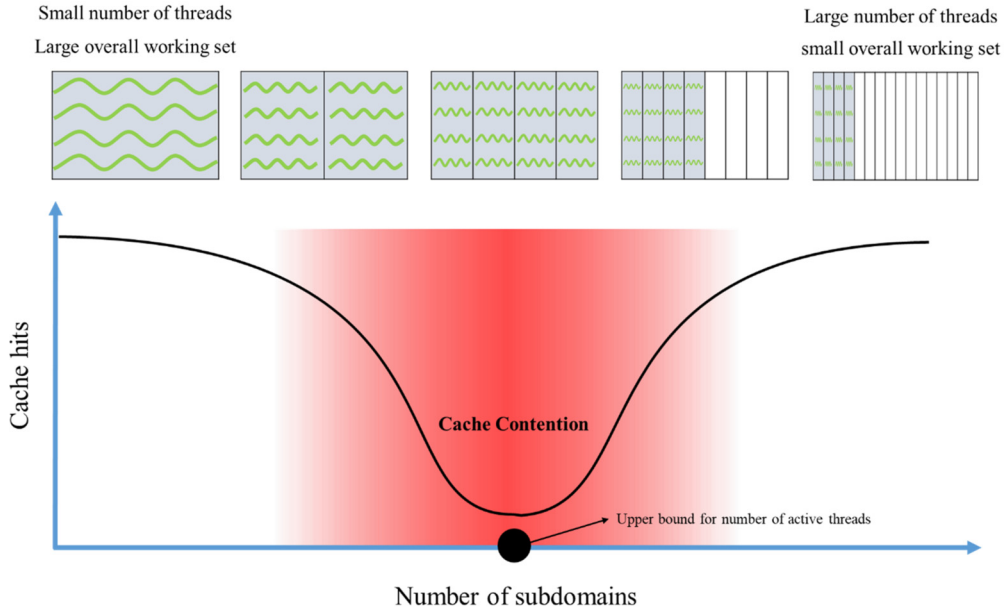

Fig. 5. Schematic diagram of the relation between L2 cache hit rate and the number of subdomains.

Solving tri-diagonal systems

Tri-diagonal systems are solved by using Thomas algorithm. In this kernel, each system of equation is mapped to one thread. Fig. 4 shows the correspondence between threads and systems of equations. As can be seen, for sweep in the y-direction the threads indices are mapped to the first index of the two-dimensional data. Therefore, considering the memory arrangement, sequential threads access contiguous locations in memory, and thus the memory access is coalesced.

For sweep in the x-direction the `x-aligned` strategy proposed in [48] is adopted. As shown in Fig. 4, in this strategy, the threads indices are mapped to the subdomains numbers (Thread 0, Threads 1, Thread 3, Thread 4, … are mapped to subdomain 1, subdomain 2, subdomain 3, subdomain 4, …). Consequently, sequential threads access locations in memory which are not contiguous and their distance is dependent on the size of the subdomains. As explained in section 3, in this situation access to global memory cannot be coalesced. However, we can take advantage of intra-thread localities if avoid cache contentions.

In the present study, we monitor the cache hit rate to analyze the effect of the different parameters value on the cache contention. The cache hit rate is the number of cache hits divided by the total number of memory requests over a given time interval. A cache hit occurs when the requested data can be found in cache. Fig. 5 presents a schematic diagram of the relation between L2 cache hit rate and the number of subdomains in the `x-aligned` strategy. As shown, when the *nos* (and thus the number of resident threads that share the L2 cache) is small, cache hit rate is high. This is because there is enough space on cache to allocate the useful cache lines for all of the active threads. However, by increasing the *nos* (and thus the number of active threads) threads start to compete for cache space which causes a sharp drop in cache hit rate. The minimum cache hit rate occurs when the occupancy reaches to its maximum determined by resource usage and execution configuration. After this point, increasing the number of subdomain does not affect the number of resident threads but interestingly the size of the data that each thread needs during its computations (working set) decreases. Therefore, the data can fit in the cache and this increases the cache hit rate. The occurrence of cache contention and its influence on performance depends on GPU architecture and especially the cache size. In our experiments the introduced trend occurs when the mesh size is large (larger than 256×256) and the peak of the curve shifts to smaller number of subdomains for larger mesh sizes.

According to the analysis above, to avoid the cache contention the number of subdomains must be small or very large. But, a small num-

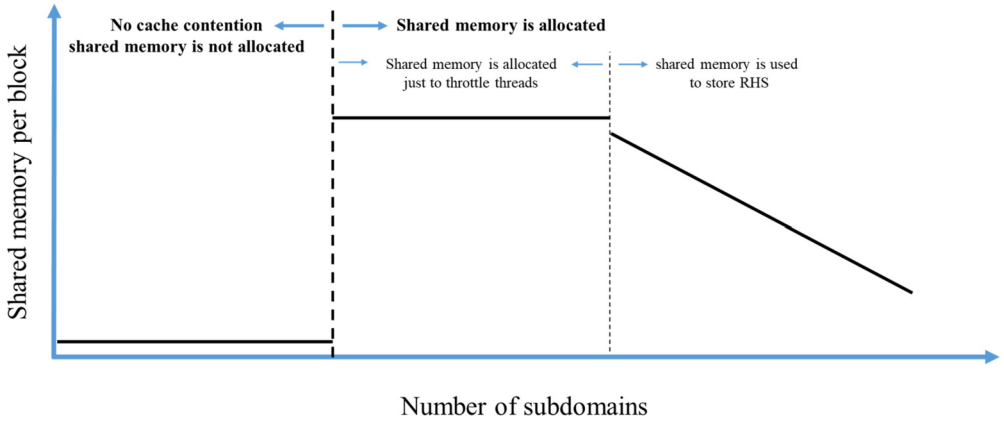

Fig. 6. A three-stage strategy (3S) for allocating shared memory to improve the memory access speed in solving tri-diagonal systems in x-direction.

ber of subdomains reduces the number of concurrent threads and a large number of subdomains may cause low accuracy in some cases. In the present study, we use software thread throttling to prevent cache contention in middle *nos* range.

In software thread throttling, the source code must be modified to reduce the number of active threads in middle *nos* range. Here, we allocate shared memory space to reduce the theoretical occupancy. This shared memory can be also used as a container for the right hand sides of the tri-diagonal systems which is accessed several times during the calculations. In this manner, we can propose an effective approach for using both 2 MB off-chip L2 and 64 KB shared memory.

To achieve the best performance, appropriate strategy must be adopted to create a balance between occupancy, cache hit rate and shared memory usage in different *nos* ranges. In the present study, a three-stage strategy named 3S is proposed which is illustrated in Fig. 6. As shown, when the number of subdomains is small, there is no need for especial treatments as there is no cache contention. In the middle *nos* range we allocate a constant amount of shared memory per block just to reduce the active blocks. In this range of *nos* the size of tri-diagonal systems is large and using shared memory for storing the RHS is either impossible or leads to a very low occupancy and low performance. However, when *nos* large enough, the RHS can fit in memory and provide enough parallelism at the same time. In this range of *nos*, the amount of shared memory per block is exactly the amount needed to store RHS for calculations of one block and therefore decreases linearly by increasing the *nos*. To simplify the analysis, in the present research for all grid sizes, we use shared memory for storing RHS just when the size of the tri-diagonal systems is less than 32. However, the starting point for cache throttling and the size of the allocated shared memory is different for each grid size and is decided by profiling the application. In the results section, the performance of the 3S strategy is analyzed and compared with the naive strategy that simply uses global memory to store data and does not use thread throttling.

Correction

For calculating the corrected values, the computation related to each interface point is assigned to one thread. The implementing of this part of the computation is straight forward and no special technique is employed.

5. Results and discussion

5.1. Accuracy and stability

In this study, three GCEIDD algorithms are proposed which use different schemes in the prediction and implicit solution steps. In this section the accuracy and stability of these methods are tested using four examples. The discrete error-norms are defined as follows:

Table 1
Spatial error for different GCEIDD algorithms for example 1 ($D = 0.001, T = 1.0$, $\Delta t = 1/1024$).

Method	Error	Grid size			
		32×32	64×64	128×128	256×256
MUGCEIDD	L_∞	4.55E-03	2.52E-03	1.39E-03	7.95E-04
	Ratio		0.85	0.86	0.80
	L_2	9.57E-04	5.32E-04	2.93E-04	1.82E-04
	Ratio		0.85	0.86	0.69
CFDGCEIDD	L_∞	4.46E-03	2.36E-03	1.17E-03	5.30E-04
	Ratio		0.92	1.01	1.14
	L_2	9.41E-04	5.05E-04	2.56E-04	1.22E-04
	Ratio		0.90	0.98	1.06
CPGCEIDD	L_∞	4.65E-03	2.59E-03	1.43E-03	8.06E-04
	Ratio		0.84	0.86	0.83
	L_2	9.76E-04	5.46E-04	3.00E-04	1.69E-04
	Ratio		0.84	0.86	0.83

$$L_2^{n+1} = \left\{ \sum_{i,j} \Delta x \Delta y \left(U_e\left(x_i, y_j, t^{n+1}\right) - U_{i,j}^{n+1} \right)^2 \right\}^{1/2},$$
$$L_\infty^{n+1} = \max_{i,j} \left\{ \left| U_e\left(x_i, y_j, t^{n+1}\right) - U_{i,j}^{n+1} \right| \right\}, \tag{45}$$

in which U_e is the exact solution. Also, the courant number (CFL) is defined as:

$$\mathrm{CFL} = \max(b_1, b_2) \frac{\Delta t}{h}. \tag{46}$$

Example 1

For this example, the velocity field is given by $b_1 = b_2 = 1$, the diffusion coefficients are $a_1 = a_2 = D$, where D is a positive constant. The computational domain is taken as $\Omega = [0, 1] \times [0, 1]$, and the exact solution is given by:

$$u(x, y, t) = \exp(-t) x^2 \left(1 - x^2\right) y^2 \left(1 - y^2\right). \tag{47}$$

The right-hand side function and the boundary conditions are set by the exact solution.

Fig. 7 presents the errors obtained by the three proposed GCEIDD algorithms for different number of subdomains at final time $T = 1.0$. The numerical experiments are performed for different courant numbers, $\mathrm{CFL} = 0.5, 1.0$ and 2.0, $h = 1/256$, and $D = 0.001$. As seen, the most accurate results are obtained by CFDGCEIDD. Although the error obtained by CPGCEIDD decreases continuously by increasing *nos*, the error of MUGCEIDD generally has a peak at middle range *nos* and then decrease.

To see the effect of the time step size on the accuracy, Fig. 8 illustrates the errors obtained by GCEIDD algorithms for different courant numbers at $T = 1.0$. Numerical experiments are performed for $h = 1/256$ and $nos = 32$, and $D = 0.001$. As seen, for $\mathrm{CFL} \leq 2$, CFDGCEIDD is the

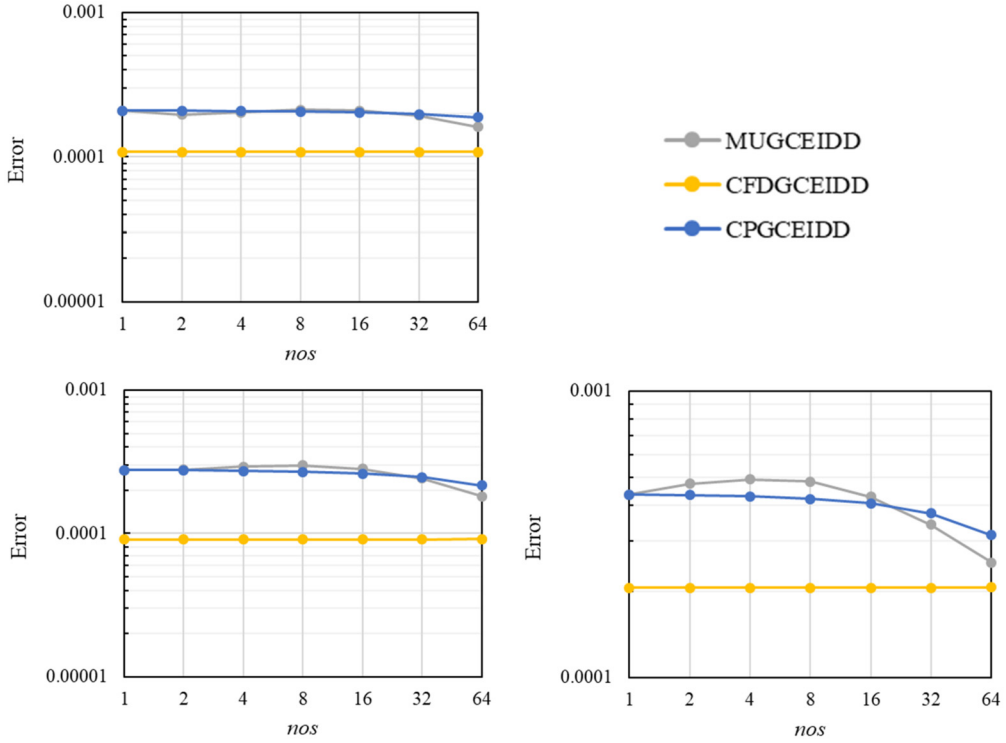

Fig. 7. The errors in $L_2 - norm$ by different GCEIDD algorithms for example 1 ($T = 1.0$ s, $h = 1/256$): CFL = 0.5 (top-left), CFL = 1.0 (bottom-left) and CFL = 2.0 (bottom-right).

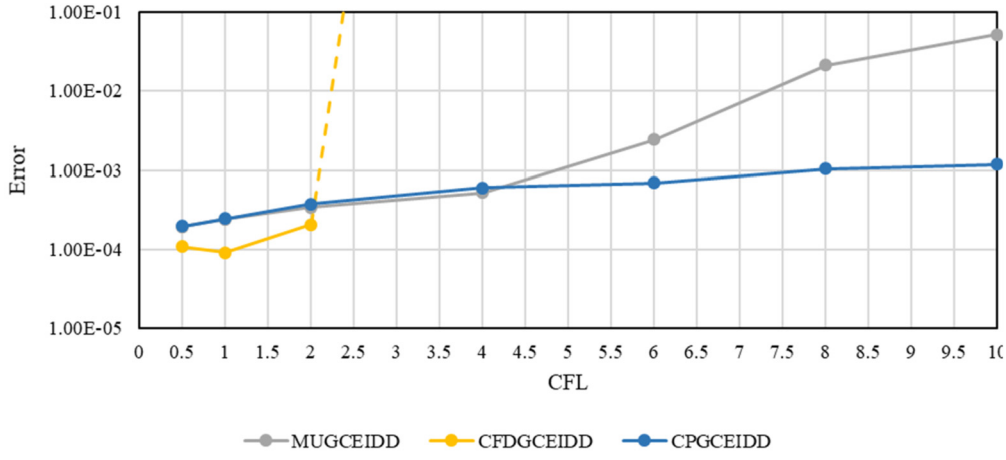

Fig. 8. The errors in $L_2 - norm$ by different GCEIDD algorithms for example 1 ($T = 1.0$, $h = 1/256$).

most accurate algorithm, however, for CFL > 2, the algorithm is not stable. For MUGCEIDD, error increases sharply with increasing the time step size. What is striking in Fig. 8, is that CPGCEIDD exhibits good stability and accuracy at large time step sizes.

To find the convergence rate of GCEIDD algorithms in space, numerical experiments are performed for $\Delta t = 1/1024$, different spatial step sizes ($h = \frac{1}{32}, \frac{1}{64}, \frac{1}{128}$ and $\frac{1}{256}$), $nos = 8$ and final time $T = 1.0$. Table 1 and Table 2 show the results for $D = 0.001$ and 0.05, respectively. The numerical results show that the present method obtains more accurate results for the larger diffusion numbers.

Example 2

In this example, the convection-diffusion problem of a rotating Gaussian pulse is considered. The velocity field is given by $b_1 = -4y$ and $b_2 = 4x$ and the diffusion coefficients are $a_1 = a_2 = D$ where D is a positive constant. The computational domain is taken as $\Omega = [0, 1] \times [0, 1]$, and the exact solution is as follows:

$$u(x, y, t) = \frac{\sigma^2}{\sigma^2 + 4Dt} \exp\left(-\frac{(\overline{x} - x_0)^2 + (\overline{y} - y_0)^2}{\sigma^2 + 4Dt}\right), \tag{48}$$

where $\overline{x} = x\cos(4t) + y\sin(4t)$, $\overline{y} = -x\sin(4t) + y\cos(4t)$, $x_0 = 0.5$, $y_0 = 0.75$, $\sigma^2 = 0.002$. The initial and boundary conditions are set by the above exact solution.

In the first experiment, the effect of the number of subdomains on the accuracy is investigated. Fig. 9 shows the results obtained by GCEIDD at final time $T = \frac{\pi}{2}$. Numerical experiments are performed for $h = 1/256$ and CFL = 0.5, 1.0, and 2.0. As seen, the error of the MUGCEIDD is larger than the error that obtained by the other methods and it increases continuously by increasing nos at all courant numbers. The experimental results show that by increasing the number of subdomains the error of the MUGCEIDD algorithm is increased by factors of up to 2.2, 2.7, and 2.9 for courant numbers 0.5, 1.0, and 2.0 respectively. The CFDGCEIDD algorithm maintains the accuracy even when nos is very large. As it is expected, the accuracy of CPGCEIDD lies

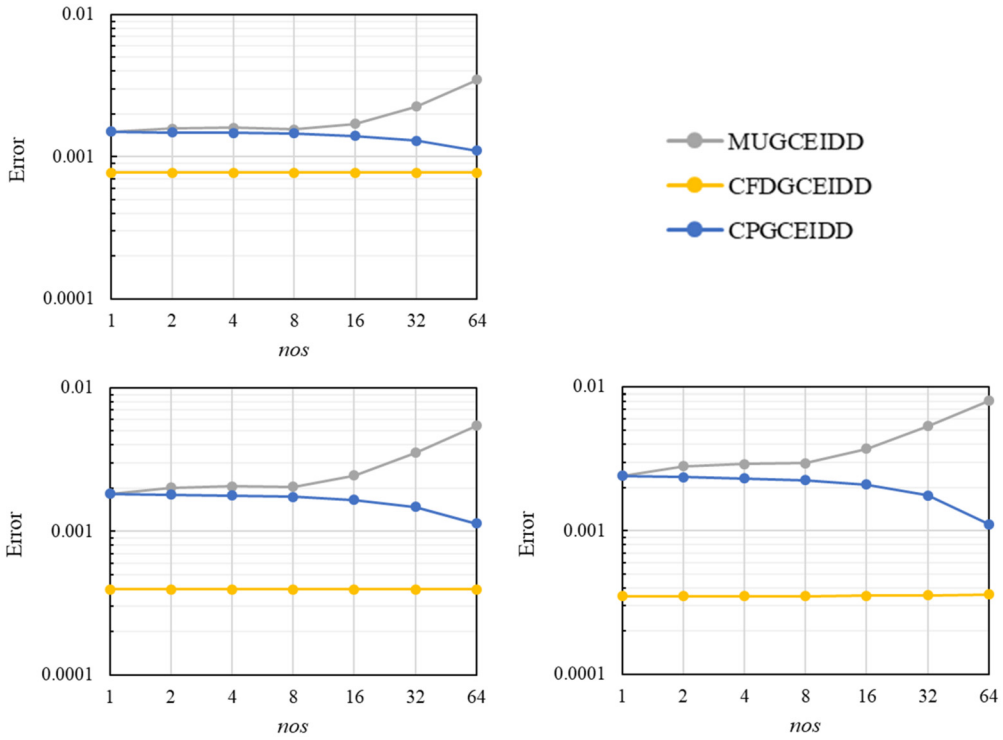

Fig. 9. The errors in $L_2 - norm$ by different GCEIDD algorithms for example 2 ($T = \frac{\pi}{2}$, $h = 1/256$): CFL = 0.5 (top-left), CFL = 1.0 (bottom-left) and CFL = 2.0 (bottom-right).

Table 2
Spatial error for different GCEIDD algorithms for example 1 ($D = 0.05, T = 1.0$, $\Delta t = 1/1024$).

Method	Error	Grid size			
		32 × 32	64 × 64	128 × 128	256 × 256
MUGCEIDD	L_∞	3.02E-03	1.54E-03	7.66E-04	4.73E-04
	Ratio		0.97	1.00	0.70
	L_2	7.72E-04	3.62E-04	2.16E-04	1.25E-04
	Ratio		1.09	0.75	0.79
CFDGCEIDD	L_∞	2.93E-03	1.60E-03	8.21E-04	4.05E-04
	Ratio		0.88	0.96	1.02
	L_2	7.54E-04	4.11E-04	2.15E-04	1.12E-04
	Ratio		0.88	0.93	0.95
CPGCEIDD	L_∞	3.02E-03	1.69E-03	9.21E-04	5.04E-04
	Ratio		0.84	0.88	0.87
	L_2	7.72E-04	4.32E-04	2.38E-04	1.34E-04
	Ratio		0.84	0.86	0.83

Table 3
Spatial error for different GCEIDD algorithms for example 2 ($D = 0.005, T = \frac{\pi}{2}$, $\Delta t = 1/1024$).

Method	Error	Grid size			
		32 × 32	64 × 64	128 × 128	256 × 256
MUGCEIDD	L_∞	3.07E-02	2.19E-02	1.53E-02	1.14E-02
	Ratio		0.49	0.52	0.43
	L_2	5.60E-03	3.88E-03	2.66E-03	1.99E-03
	Ratio		0.53	0.54	0.42
CFDGCEIDD	L_∞	2.85E-02	1.79E-02	8.82E-03	2.66E-03
	Ratio		0.67	1.02	1.73
	L_2	5.10E-03	3.06E-03	1.48E-03	4.82E-04
	Ratio		0.74	1.05	1.61
CPGCEIDD	L_∞	3.08E-02	2.19E-02	1.49E-02	1.05E-02
	Ratio		0.49	0.55	0.51
	L_2	5.61E-03	3.84E-03	2.55E-03	1.75E-03
	Ratio		0.55	0.59	0.54

between the accuracy of MUGCEIDD and CFDGCEIDD and its error decreases by increasing the number of subdomains.

To show the effect of time step size on the accuracy, Fig. 10 presents the errors of GCEIDD algorithms at final time $T = \frac{\pi}{2}$ for different courant numbers. Numerical experiments are performed for $h = 1/256$ and $nos = 32$. As indicated by the experimental results, for courant number equal or less than 4, the CFDGCEIDD is very accurate. However, for larger courant numbers the method is not stable. The CPGCEIDD method maintains the stability even for large courant numbers and it is more accurate compared with MUGCEIDD.

Fig. 11 provides a visual comparison of the solutions obtained using different GCEIDD algorithms at final time $T = \frac{\pi}{2}$. The spatial step size is $1/256$ and $nos = 32$. Results show deformation and overshoot in the solutions obtained using MUGCEIDD. For instance, at CFL = 8.0, there are severe oscillations and distortion at locations with large gradients. The CFDGCEIDD solutions are in perfect agreement with the exact solution for courant number equal to 1.0 and 4.0. In solution obtained using CPGCEIDD, the peak of the Gaussian pulse is smeared, however, there is no oscillation and distortion like there is in MUGCEIDD. What is interesting in Fig. 11, is the relatively accurate solution obtained using CPGCEIDD at CFL = 8.0.

The spatial error of GCEIDD algorithms is presented in Table 3 and Table 4 for $D = 0.005$ and 0.01, respectively. Numerical experiments are performed for $\Delta t = 1/1024$, $nos = 8$ and the final time $T = \frac{\pi}{2}$. As it is observed, the GCEIDD algorithms exhibit better accuracy for the larger diffusion numbers.

Example 3

In this example, the moving sharp problem is considered. The velocity field is given by $b_1 = 1$ and $b_2 = 1$ and the diffusion coefficients are $a_1 = a_2 = 0.001$. The computational domain is taken as $\Omega = [0, 1] \times [0, 1]$, and the initial conditions are given by:

$$u(x, y, 0) = \begin{cases} 1, & 0 \le x, y \le 0.2 \\ 0, & \text{Otherwise}, \end{cases} \tag{49}$$

and the boundary conditions are as follows:

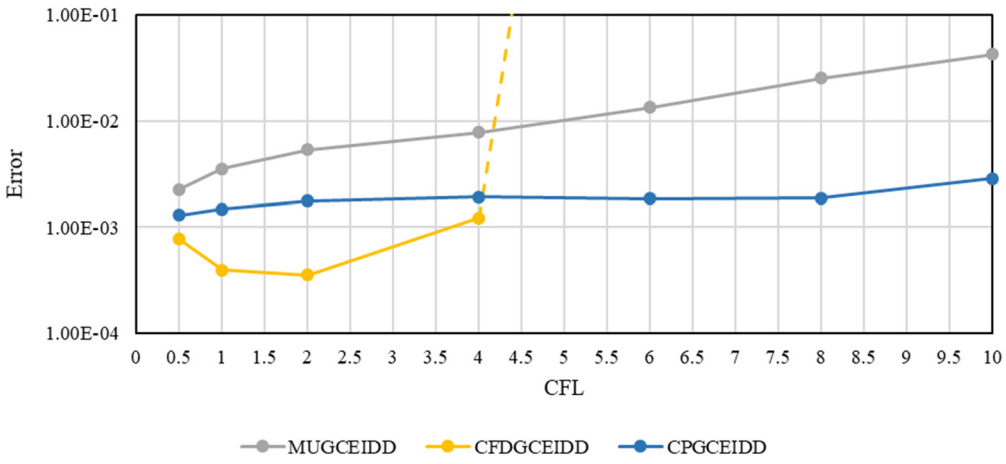

Fig. 10. The errors in $L_2 - norm$ by different GCEIDD algorithms for example 1 ($T = \frac{\pi}{2}$, $h = 1/256$).

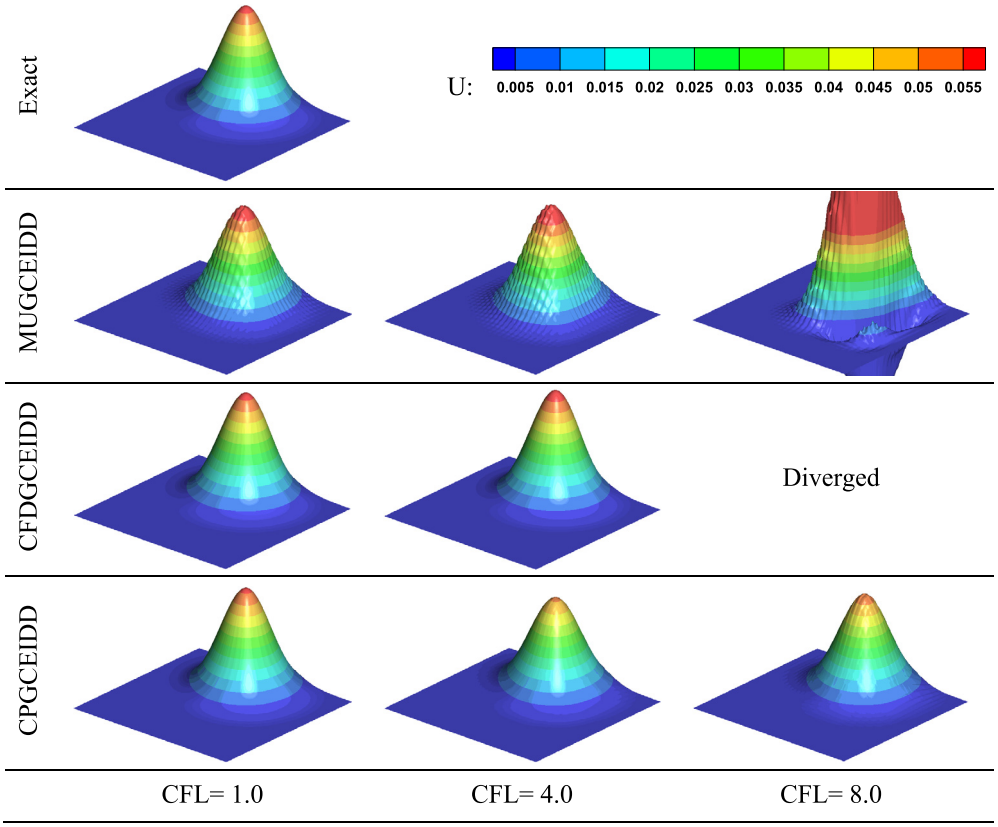

Fig. 11. Visual comparison of the solutions obtained using different GCEIDD algorithms for example 2 ($T = \frac{\pi}{2}$).

$$u(0,y,t)=1, y\in[0,1], \quad u(x,0,t)=1, x\in[0,1], t\in(0,T],$$
$$\frac{\partial u(1,y,t)}{\partial x}=0, y\in[0,1], \quad \frac{\partial u(x,1,t)}{\partial y}=0, x\in[0,1], t\in(0,T]. \tag{50}$$

Due to the small value of the diffusion coefficients and discontinuous initial conditions, the numerical simulation of this example is challenging. Fig. 12 presents the solutions obtained by the three proposed GCEIDD algorithms at the final time $T = 0.2$ on a sectional plane $x = 0.5$. The spatial step size is $1/256$ and $nos = 32$. The results are compared against a reference solution obtained by the characteristics finite difference scheme with a very fine mesh and without domain decomposition ($h = 1/1024$ and CFL = 1). As seen, the most accurate results are obtained using CFDGCEIDD for CFL = 1.0 and 2.0. However, the solution is not converged for CFL = 4.0. The CPGCEIDD method maintains the stability even for large courant numbers.

Fig. 13 presents a visual comparison of the solutions obtained using different GCEIDD algorithms. The spatial step size is $1/256$, CFL = 2.0 and $nos = 32$. As seen, there is severe numerical oscillations and overshoots in the solution obtained using the MUGCEIDD method. Also, compared to the CFDGCEIDD method, the results of CPGCEIDD exhibit more dissipations.

Example 4

This example, corresponds to the problem of a forced convection heat transfer from two flat heating elements in a two-dimensional channel (Fig. 14). Therefore, u corresponds to the temperature and the diffusion coefficient corresponds to the thermal diffusivity. In this example, the velocity field is given by $b_1 = -V\,y(0.05 - y)$, $b_2 = 0$ where V is a positive constant, and the diffusion coefficients are taken as $a_1 = a_2 =$

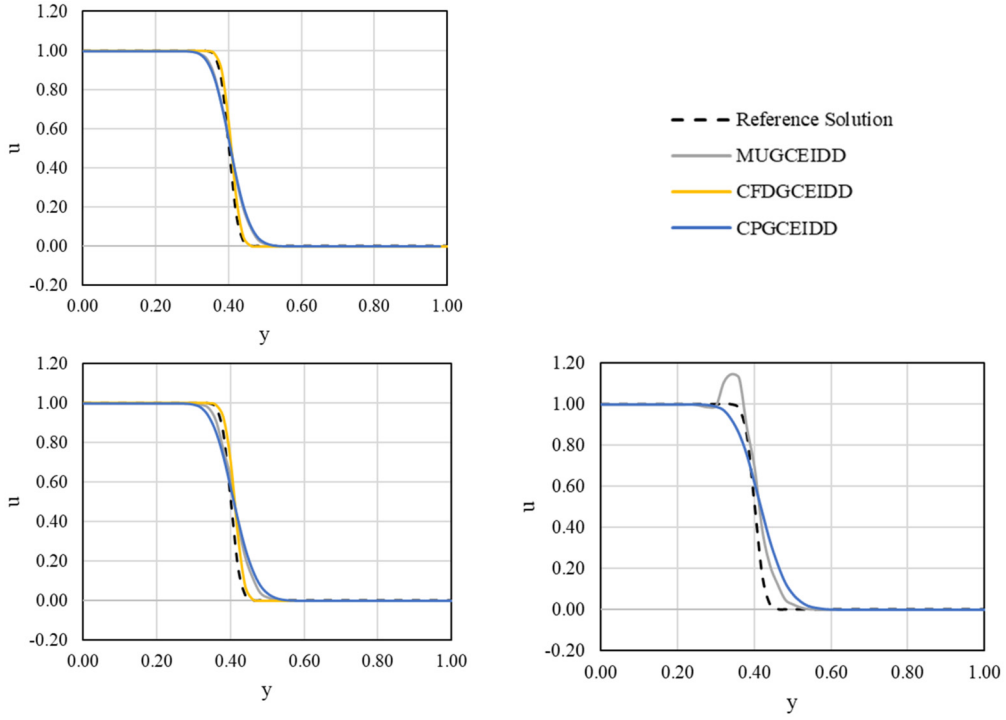

Fig. 12. The solution obtained by different GCEIDD algorithms for example 3 ($T = 0.2$, $h = 1/256$): CFL = 1.0 (top-left), CFL = 2.0 (bottom-left) and CFL = 4.0 (bottom-right). For CFL = 4.0 CFDGCEIDD is not converged.

Table 4
Spatial error for different GCEIDD algorithms for example 2 ($D = 0.01, T = \frac{\pi}{2}$, $\Delta t = 1/1024$).

Method	Error	Grid size			
		32×32	64×64	128×128	256×256
MUGCEIDD	L_∞	1.08E-02	6.93E-03	4.35E-03	3.26E-03
	Ratio		0.64	0.67	0.42
	L_2	2.54E-03	1.59E-03	9.82E-04	7.43E-04
	Ratio		0.68	0.70	0.40
CFDGCEIDD	L_∞	9.65E-03	5.48E-03	2.50E-03	7.61E-04
	Ratio		0.82	1.13	1.72
	L_2	2.24E-03	1.24E-03	5.64E-04	1.98E-04
	Ratio		0.86	1.13	1.51
CPGCEIDD	L_∞	1.07E-02	6.87E-03	4.14E-03	2.51E-03
	Ratio		0.64	0.73	0.72
	L_2	2.54E-03	1.58E-03	9.19E-04	5.39E-04
	Ratio		0.68	0.78	0.77

2×10^{-5}. The computational domain is taken as $\Omega = [0, 0.25] \times [0, 0.05]$, and the initial conditions are given by:

$$u(x, y, 0) = 25, x \in [0, 0.25], y \in [0, 0.05], \tag{51}$$

and the boundary conditions are taken as:

$$\begin{aligned}
&\frac{\partial u(0, y, t)}{\partial x} = 0, y \in [0, 0.05], \qquad u(0.25, y, t) = 25, y \in [0, 0.05], \\
&\frac{\partial u(x, 0, t)}{\partial y} = 0, x \in [0, 0.15] \cup [0.2, 0.25], t \in (0, T], \\
&\frac{\partial u(x, 1, t)}{\partial y} = 0, x \in [0, 0.15] \cup [0.2, 0.25], t \in (0, T], \\
&\frac{\partial u(x, 0, t)}{\partial y} = -5 \times 10^4, x \in [0.15, 0.2], t \in (0, T], \\
&\frac{\partial u(x, 1, t)}{\partial y} = 5 \times 10^4, x \in [0.15, 0.2], t \in (0, T].
\end{aligned} \tag{52}$$

To solve this problem, the domain is uniformly divided by $x_i = i \times h_x$ and $y_j = j \times h_y$, where h_x and h_y are the spatial step sizes in x- and y-direction, respectively. Here, we have $\text{CFL} = \max(\frac{b_1}{h_x}, \frac{b_2}{h_y})\Delta t$. Also, the number of subdomains in x-sweep and y-sweep steps are *nosx* and *nosy*, respectively. The GCEIDD schemes can be easily extended to handle any two-dimensional problem with a rectangular domain.

First, the performance of GCEIDD algorithms is investigated for different values of V. The results, presented in Fig. 15, are obtained at the final time $T = 3.0$, for $h_x = \frac{1}{512}, h_y = \frac{1}{128}, nosx = 64$ and $nosy = 16$. The reference solution is obtained by the characteristics finite difference scheme with a very fine mesh and without domain decomposition ($h_x = 1/1024$, $h_y = 1/512$ and CFL = 1). As seen, due to the dominance of the convection term, the heat transfer mainly occurs along the direction of the velocity. As it is expected, increasing the V from 250 to 1000 reduces the average temperature in the channel and on the surface of the heating elements. Overall, the results obtained by the present domain decomposition schemes have a good agreement with the reference solution. To have a more precise investigation on the accuracy of the present domain decomposition schemes, the next numerical experiment is provided.

Fig. 16 shows the temporal changes of temperature at the center of the heating element (point A) obtained by different GCEIDD algorithms. Numerical experiments are performed for $h_x = \frac{1}{512}, h_y = \frac{1}{128}, nosx = 64$ and $nosy = 16$. As can be seen, the solution obtained by CFDGCEIDD and CPGCEIDD is almost identical to the reference solution at the second half of the simulation time; however, there is a slight deviation at the first half. Although, the results of these methods are in agreement with the reference solution for both CFL = 1 and CFL = 2, the errors of the MUGCEIDD method are considerable.

5.2. Efficiency

In the present section, the performance of the GCEIDD method is evaluated in terms of computation time. The main objective of proposing GCEIDD algorithm is to accelerate solving tri-diagonal systems in the fractional steps method. Therefore, in the following, first, the effect of different parameters on the computation time of solving tri-diagonal systems is analyzed. Then, the computation time of the prediction and

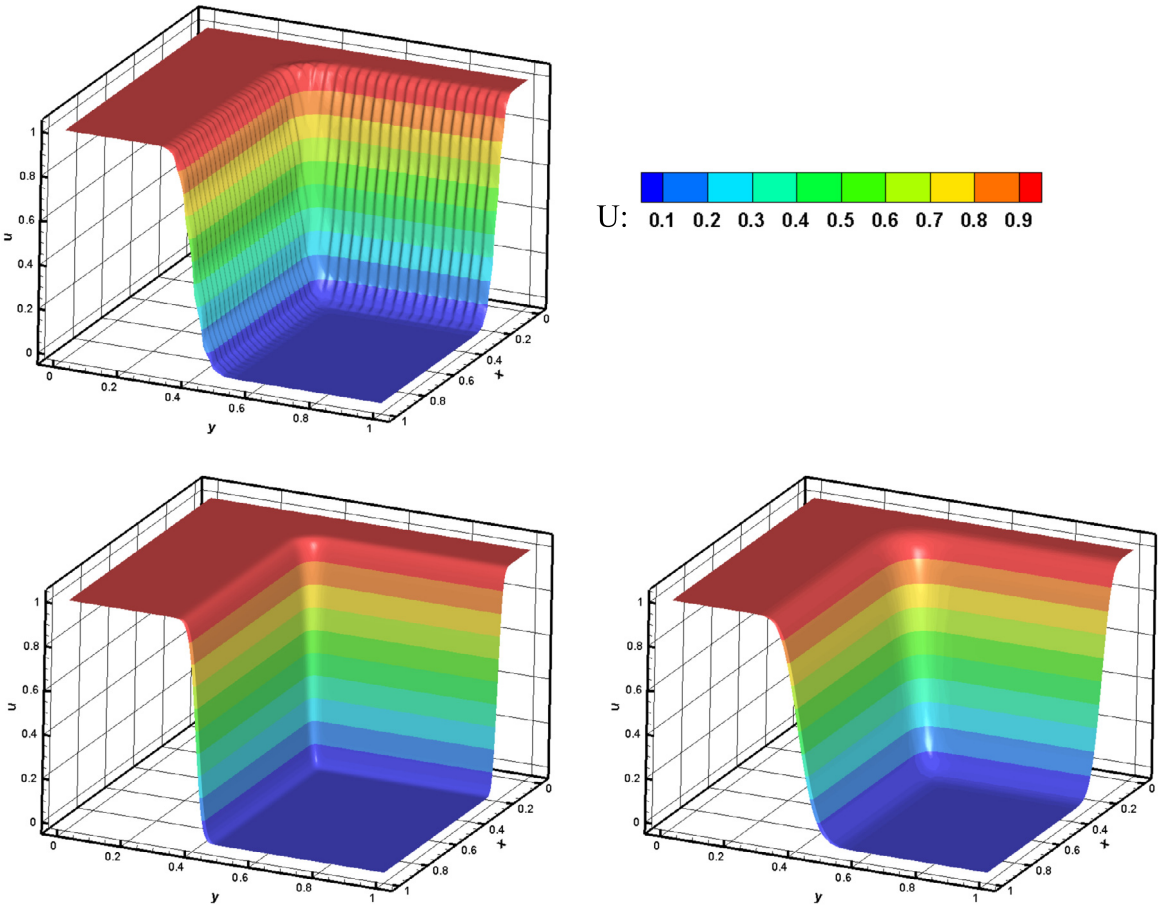

Fig. 13. Visual comparison of the solutions obtained using different GCEIDD algorithms for example 3 ($h = \frac{1}{256}$, CFL = 2.0, $T = 0.2$): MUGCEIDD (top-left), CFDG-CEIDD (bottom-left) and CPGCEIDD (bottom-right).

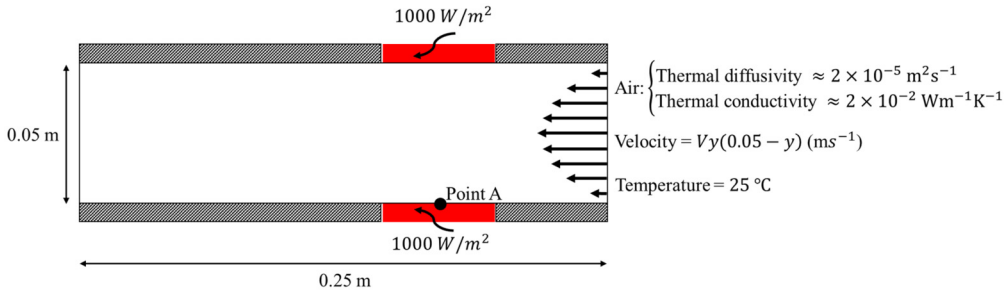

Fig. 14. Forced convection heat transfer from two flat heating elements in a two-dimensional channel (example 4).

correction step is investigated for different GCEIDD algorithms. Also, the performance of the GCEIDD method is compared with the GPU implementation of the classic fractional step methods which do not use domain decomposition ($nos = 1$). Finally, the performance of the present approach is compared with the CRPCR method which uses parallel tri-diagonal matrix algorithm to solve systems of equations in the fractional step method.

The examples presented in the previous section have no significant difference in terms of the implementation and programming. Therefore, all of the measurements in this section are performed only on example 2. However, the results are applicable for any other problem which is solved using the GCEIDD algorithms.

Solving tri-diagonal systems: cache contention and optimization

In the present section, the effect of *nos* on the computation time of solving tri-diagonal system is analyzed. Fig. 17 presents the computation time of solving tri-diagonal systems in y-direction for different number of subdomains and different grids. As it was expected, for all grid sizes the computation time decreases by increasing the *nos*. Results show that by using the GCEIDD method the solution of the tri-diagonal systems is accelerated by factors of up to 16.4, 10.6, 5.8 and 3.5 for 128×128, 256×256, 512×512 and 1024×1024 grids, respectively. This is because a large number of subdomains improves the achieved occupancy and leads to better exploitation of GPU resources. Fig. 18 shows the achieved occupancy for different grid sizes and number of subdomains. As seen, by increasing the *nos*, the achieved occupancy gets closer to the theoretical value which is equal to 62.5%. The effect of domain decomposition on the achieved occupancy is more intense at small grid sizes. For large grid sizes (512×512 and 1024×1024), the occupancy gets almost saturated at $nos = 32$, however, further domain decomposition still helps to improve memory bandwidth.

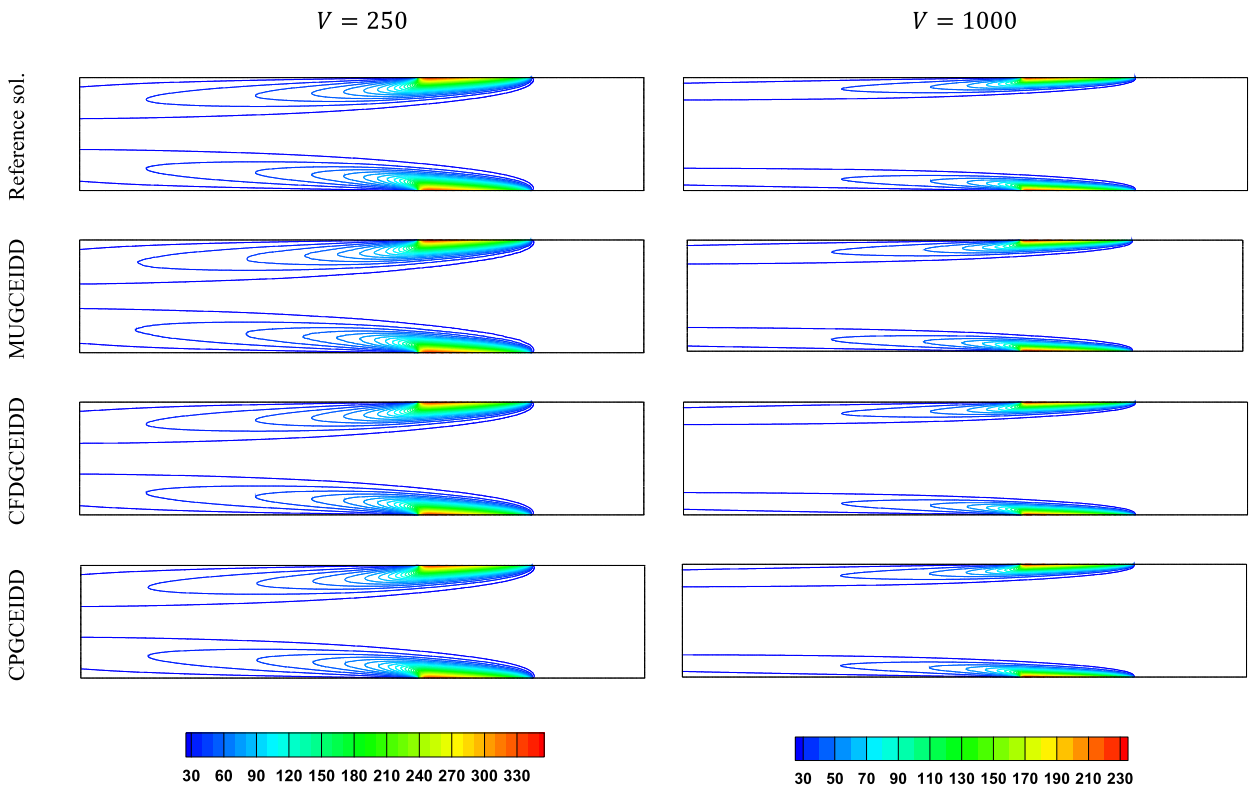

Fig. 15. Temperature contours obtained by different GCEIDD algorithms.

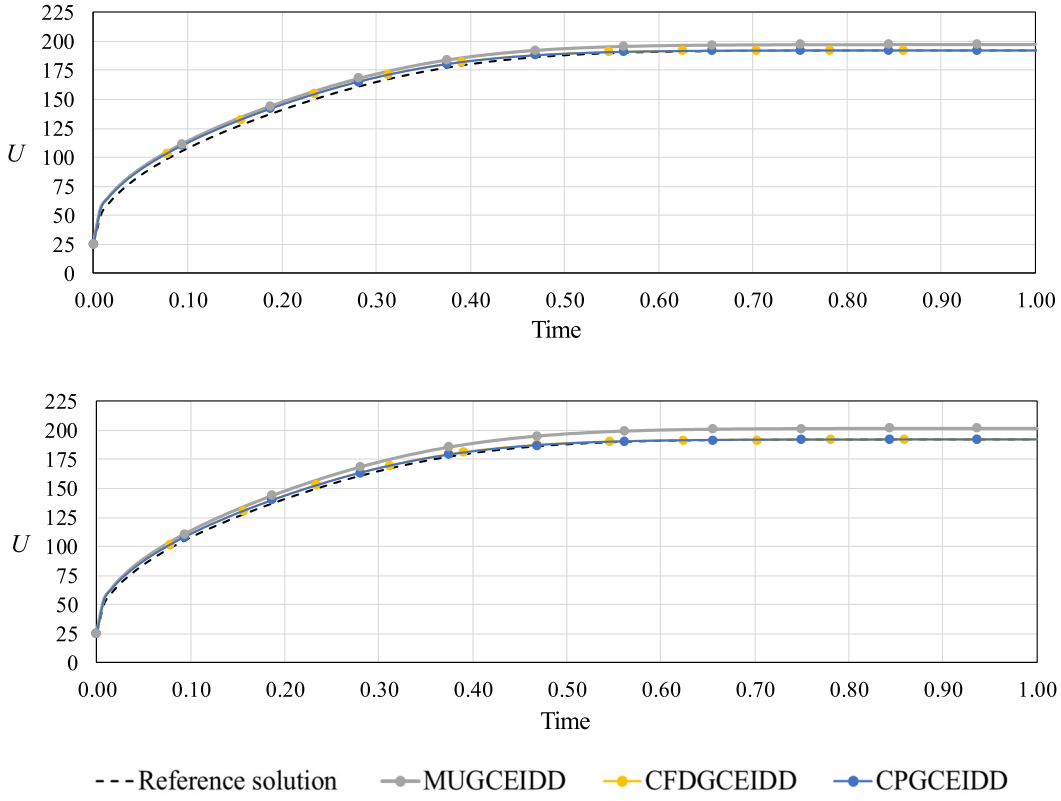

Fig. 16. Temperature at the center of the heating elements versus time obtained by different GCEIDD algorithms: CFL = 1 (top), CFL = 2 (bottom).

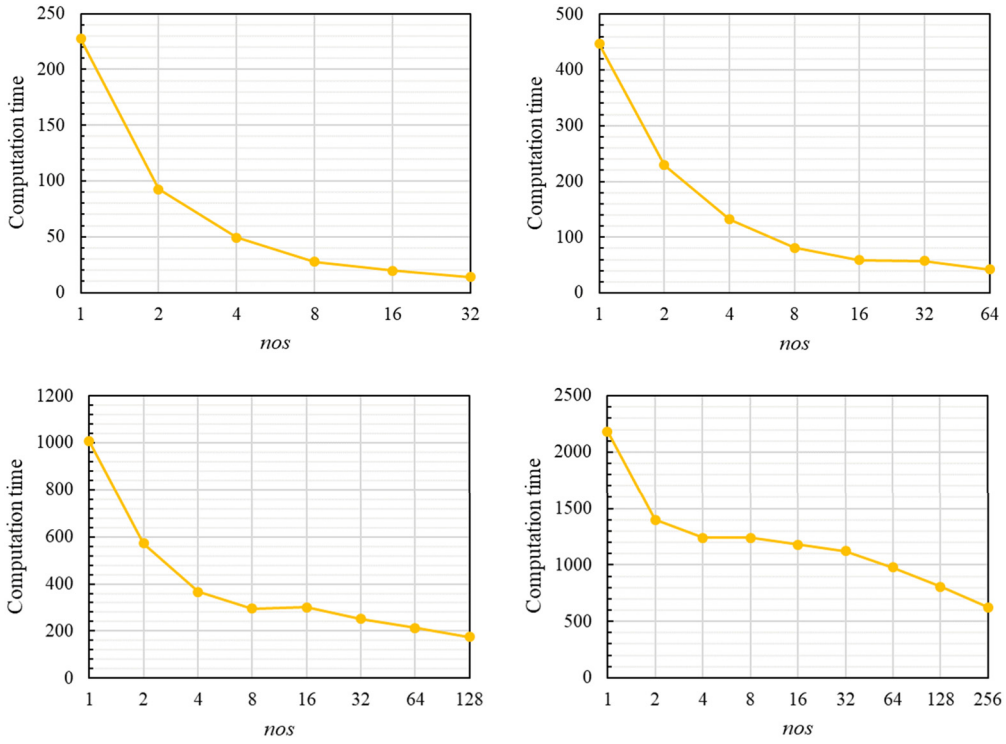

Fig. 17. Computation time of solving tri-diagonal systems in y-direction for: 128×128 (top-left), 256×256 (top-right), 512×512 (bottom-left) and 1024×1024 (bottom-right) grids.

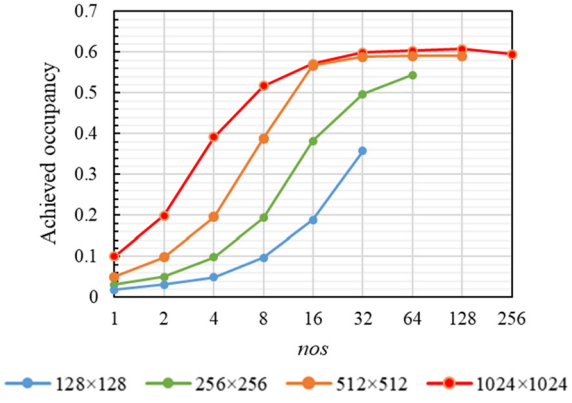

Fig. 18. Achieved occupancy for solving tri-diagonal systems in y-direction.

Next, we analyze the performance of GCEIDD in solving tri-diagonal systems in x-direction. Fig. 19 shows the computation time spent on this part of the computation for both the `3S` and `naive` strategies. As seen, for small and large number of subdomains the computation times for the two strategies are almost equal. However, in middle range *nos* and large grid sizes, there is a significant difference between them. As explained before, this is due to the fact that the efficiency of solving tri-diagonal systems in x-direction is highly affected by cache contentions in middle range *nos*. Fig. 20, shows the L2 cache hit rate for different grid sizes and number of subdomains. As seen, in the `naive` strategy cache hit rate decreases sharply after $nos = 2$ and $nos = 1$ for 512×512 and 1024×1024 grids, respectively. But, interestingly, employing the `3S` strategy makes a great improvement in cache hits. For example, using the `3S` strategy for the 1024×1024 grid increases the cache hit rate by 43% at $nos = 8$ which results in accelerating the computation by a factor of 3.7 compared with the `naive` strategy.

Overall, the GCEIDD method accelerates the computation of solving the tri-diagonal systems in x-direction by factors of up to 10.9, 6.7, 4.9 and 4.7 for 128×128, 256×256, 512×512 and 1024×1024 grids, respectively.

Next, the computational load of the prediction and correction sub-steps is investigated. In GCEIDD, the domain is decomposed into many subdomains, which leads to creating a large number of boundary points. Therefore, the computational load of the prediction and correction sub-steps must be considered in efficiency analysis. Breakdowns of the computation time of different GCEIDD algorithms are shown in Fig. 21. The results are presented for $nos = 2$ and $nos = 128$ and grid size of 512×512. As seen, at $nos = 2$ in all of the GCEIDD algorithms, the prediction and correction sub-steps take 2 to 3 percent of the total computation time, however at $nos = 128$, 15 to 20 percent of the computation time is spent on these parts of the computation.

Now, we investigate the overall performance of the GCEIDD method. Since different GCEIDD methods have almost the same computation time, here we only investigate the CPGCEIDD method. Fig. 22 shows the computation time of the GCEIDD method for different grids and number of subdomains. Also, the results are compared with the computation time of the GPU-implemented CFD method without domain decomposition ($nos = 1$). As seen, GCEIDD accelerates the solution by factors of up to 3, 2.5, 1.8 and 1.5 for 128×128, 256×256, 512×512 and 1024×1024 grids respectively. The reduction of total computation time slows down at large *nos* due to emergence of a balance between the decrease of the computation time occurred in solving tri-diagonal systems and the increase of the computation time occurred in the prediction and correction sub-steps. Although, as mentioned before, these measurements are performed on example 2, the numerical experiments show that the difference between these results and the results obtained by the other examples is less than 5%.

As mentioned before, using parallel tri-diagonal matrix algorithm for solving systems of equations is the mainstream methodology for dealing with implicit numerical schemes on GPUs. Therefore, here, the performance of the present approach is compared against parallel tri-diagonal matrix algorithm. Here CPGCEIDD represents the GCEIDD method. On the opposite side the parallel CRPCR algorithm is used to solve tri-diagonal system in the modified upwind fractional steps scheme. CRPCR

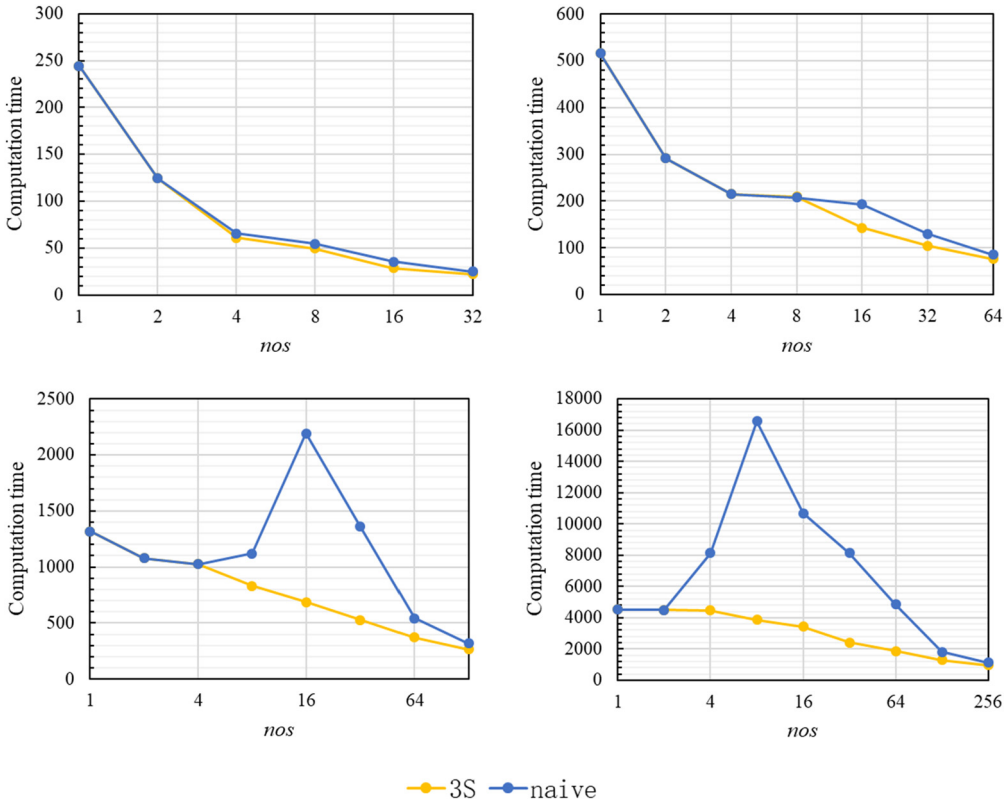

Fig. 19. Computation time of solving tri-diagonal systems in x-direction for: 128×128 (top-left), 256×256 (top-right), 512×512 (bottom-left) and 1024×1024 (bottom-right) grids.

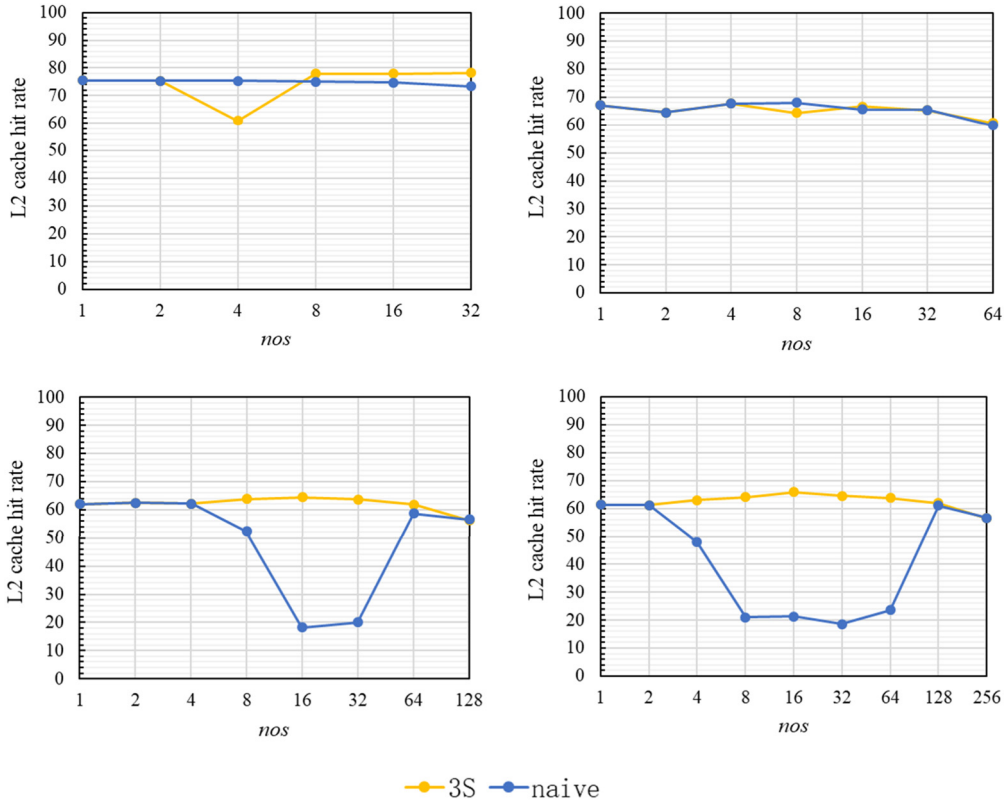

Fig. 20. L2 cache hit rate in the `3S` and `naive` strategies for: 128×128 (top-left), 256×256 (top-right), 512×512 (bottom-left) and 1024×1024 (bottom-right) grids.

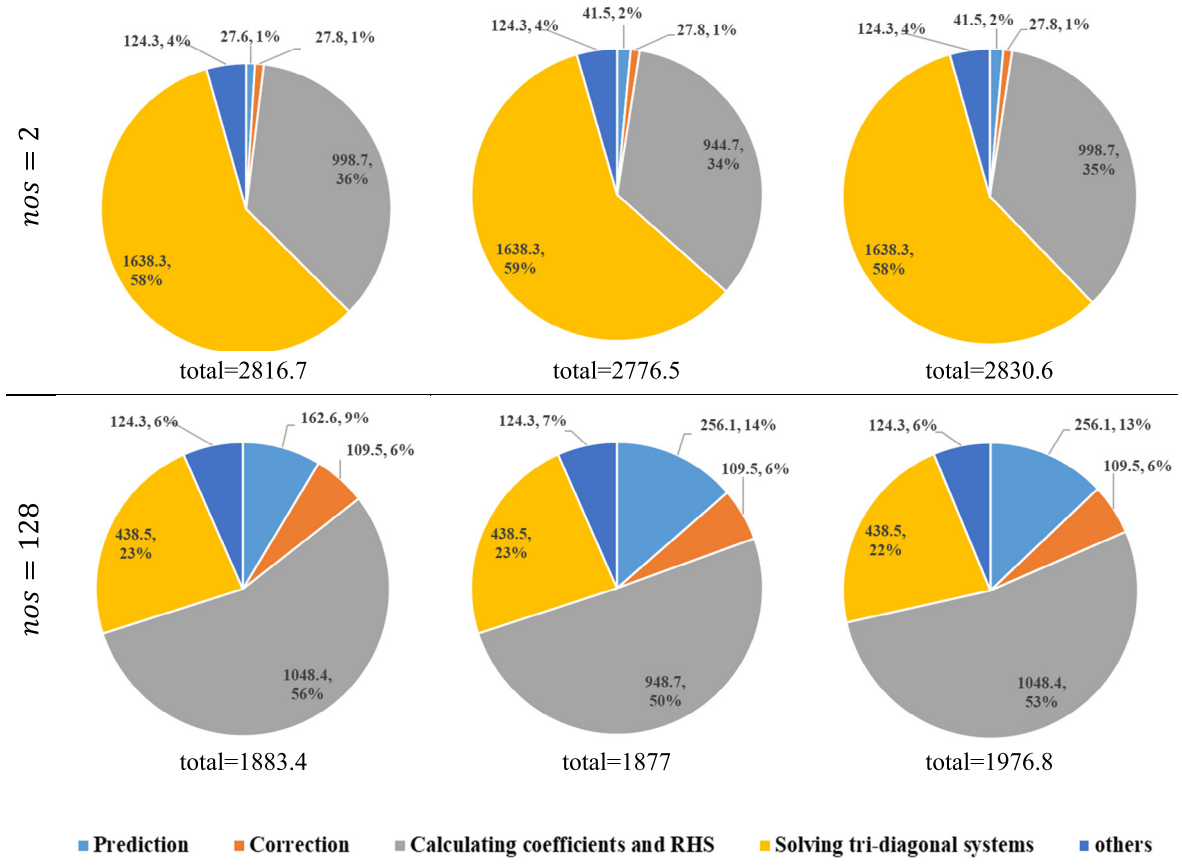

Fig. 21. Time spent on different parts of the computation for: MUGCEIDD (left), CFDGCEIDD (middle) and CPGCEIDD (right). (Grid size = 512 × 512.)

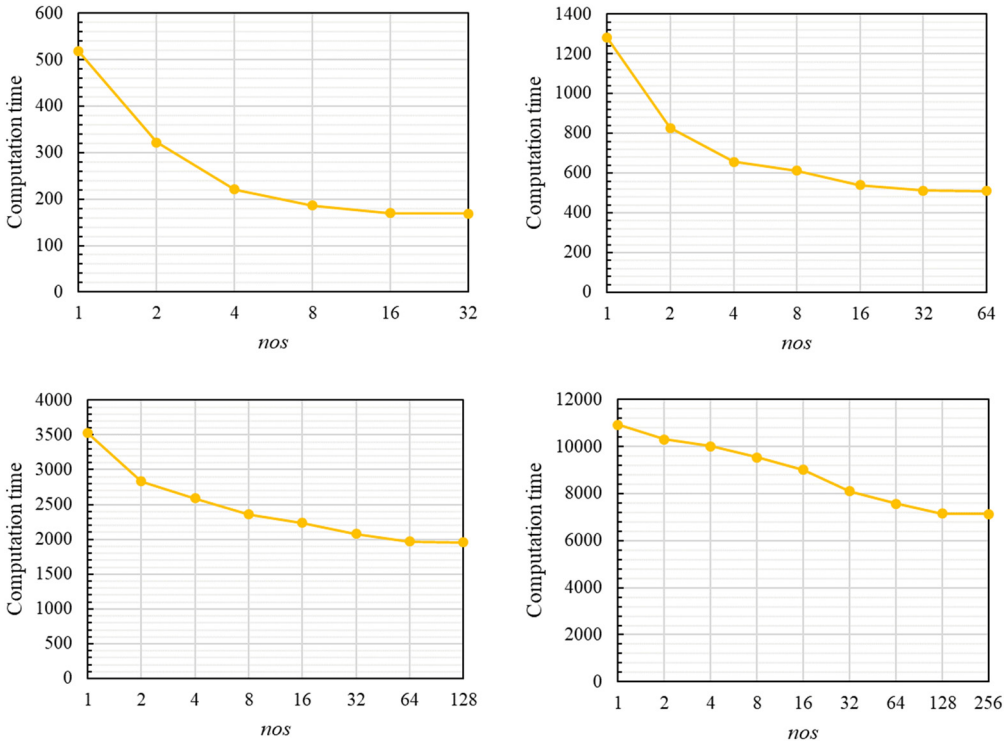

Fig. 22. Total computation time of the CPGCEIDD algorithm versus number of subdomains for: 128 × 128 (top-left), 256 × 256 (top-right), 512 × 512 (bottom-left) and 1024 × 1024 (bottom-right) grids.

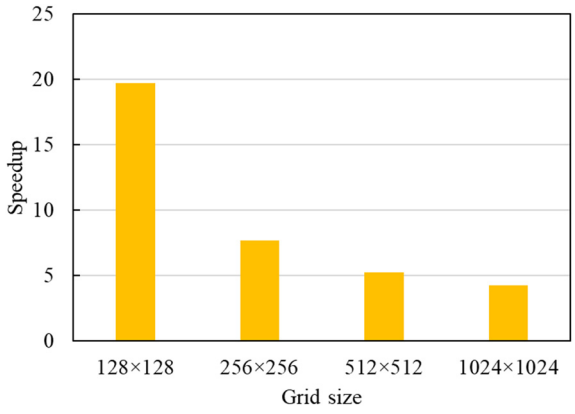

Fig. 23. Speedup of GCEIDD versus the CRPCR method for different grid sizes. In GCEIDD for 128×128 grid $nos = 32$, for 256×256 grid $nos = 64$, for 512×512 grid $nos = 128$ and for 1024×1024 grid $nos = 256$.

was firstly introduced in [49] and it is frequently used in many studies as an efficient solver. In the present study, we use the version provided by the cuSPARSE library. Fig. 23 shows the speedup of the GCEIDD algorithm versus CRPCR for different grid sizes. As seen, the GCEIDD is 19.7 to 4.2 times faster than CRPCR. The speedup of GCEIDD is more significant on small grid sizes than large ones.

6. Conclusions

In this study, a class of GPU-based corrected explicit-implicit domain decomposition schemes (GCEIDD) is proposed to accelerate the solution of convection-dominated convection-diffusion problems on GPUs. All of these methods take advantage of the fractional steps and corrected explicit-implicit domain decomposition technique. In each sweep, the domain is decomposed into many strip-divided subdomains. The aim is to reduce the size and increase the number of tri-diagonal systems to provide enough parallelism to keep GPU occupied. In different steps of the algorithms, we use low-complexity schemes with a high degree of parallelism.

The generic form of the GCEIDD algorithm allows different choices for the prediction, correction, and implicit solution steps. Three different variants of GCEIDD are proposed in the present study. MUGCEIDD uses a combination of previous time levels for the prediction, a modified upwind scheme for the implicit solution, and an interpolation scheme for the correction step. CFDGCEIDD uses the characteristic finite difference in the prediction and implicit solution steps. CPGCEIDD takes advantage of both modified upwind and characteristics finite difference. In this method, the prediction of interface values is performed using the characteristics finite difference method, and the inner subdomains are solved using the modified upwind scheme.

Based on the results, the GCEIDD method can obtain results with good accuracy and stability even when the number of subdomains is very large. CFDGCEIDD is the most accurate method in small courant numbers. For large courant numbers, the CFDGCEIDD diverges and MUGCEIDD obtains results with overshoots and deformations; however, the CPGCEIDD method produces accurate solutions.

This study shows that for efficient calculation of the coefficients and right-hand side, we must create a balance between occupancy and memory access speed by restricting the number of registers per thread. When solving the tri-diagonal systems in y-direction, the memory access is coalesced; therefore, there are no challenges related to the implementation and memory management. When solving the tri-diagonal in x-direction, access to global memory cannot be coalesced. However, we can take advantage of intra-thread localities if avoid cache contentions. To achieve the best performance, a three-stage strategy is adopted: when the number of subdomains is small, there is no need for especial treatments as there is no cache contention. For medium number of subdomains, cache contention occurs as threads compete for limited cache space. Therefore, thread throttling is performed by allocating a constant amount of shared memory per block. For a large number of subdomains, the size of the tri-diagonal systems is small, so shared memory can be used to store right-hand sides.

Based on the efficiency results, GCEIDD can provide a significant performance benefit for solving convection-dominated convection-diffusion problems on GPUs. This method can accelerate the solution of tri-diagonal systems by increasing the achieved occupancy. Also, employing our three-stage strategy for solving tri-diagonal systems in x-direction makes a great improvement in cache hits and accelerates this computation by a factor of up to 3.7. Overall, GCEIDD can accelerate the solution by a factor of up to 3 compared with the GPU implementation of the classic fractional step methods, which do not use domain decomposition.

One direction for future studies is to extend the GCEIDD method for more complicated problems like non-linear convection-diffusion equations.

References

[1] N. Sakharnykh, Tridiagonal solvers on the GPU and applications to fluid simulation, in: NVIDIA GPU Technology Conference, 2009, pp. 17–19.
[2] V. Esfahanian, B. Baghapour, M. Torabzadeh, H. Chizari, An efficient GPU implementation of cyclic reduction solver for high-order compressible viscous flow simulations, Comput. Fluids 92 (2014) 160–171, https://doi.org/10.1016/j.compfluid.2013.12.011.
[3] Z. Wei, B. Jang, Y. Jia, A fast and interactive heat conduction simulator on GPUs, J. Comput. Appl. Math. 270 (2014) 496–505, https://doi.org/10.1016/j.cam.2013.11.030.
[4] Y. Wang, et al., Solving 3D incompressible Navier-Stokes equations on hybrid CPU/GPU systems, in: Proceedings of the High Performance Computing Symposium, 2014, p. 12.
[5] L. Deng, H. Bai, F. Wang, Q. Xu, CPU/GPU computing for an implicit multi-block compressible Navier-Stokes solver on heterogeneous platform, Int. J. Mod. Phys. Conf. Ser. 42 (2016) 1660163, https://doi.org/10.1142/S2010194516601630.
[6] S. Ha, J. Park, D. You, A GPU-accelerated semi-implicit fractional-step method for numerical solutions of incompressible Navier–Stokes equations, J. Comput. Phys. 352 (September) (2018) 246–264, https://doi.org/10.1016/j.jcp.2017.09.055.
[7] H.S. Kim, S. Wu, L.W. Chang, W.M.W. Hwu, A scalable tridiagonal solver for GPUs, in: Proceedings of the International Conference on Parallel Processing, 2011, pp. 444–453.
[8] G. Alfonsi, S.A. Ciliberti, M. Mancini, L. Primavera, GPGPU implementation of mixed spectral-finite difference computational code for the numerical integration of the three-dimensional time-dependent incompressible Navier-Stokes equations, Comput. Fluids 102 (2014) 237–249, https://doi.org/10.1016/j.compfluid.2014.07.005.
[9] D. Göddeke, R. Strzodka, Cyclic reduction tridiagonal solvers on GPUs applied to mixed-precision multigrid, IEEE Trans. Parallel Distrib. Syst. 23 (1) (2011) 22–32, https://doi.org/10.1109/TPDS.2010.61.
[10] A. Davidson, J.D. Owens, Register packing for cyclic reduction: a case study, in: Proc. Fourth Work. Gen. Purp. Process. Graph. Process. Units, 2011, pp. 0–5.
[11] Z. Wei, B. Jang, Y. Zhang, Y. Jia, Parallelizing alternating direction implicit solver on GPUs, Proc. Comput. Sci. 18 (2013) 389–398, https://doi.org/10.1016/j.procs.2013.05.202.
[12] A. Davidson, Y. Zhang, J.D. Owens, An auto-tuned method for solving large tridiagonal systems on the GPU, in: Proc. - 25th IEEE Int. Parallel Distrib. Process. Symp. IPDPS 2011, 2011, pp. 956–965.
[13] L.W. Chang, J.A. Stratton, H.S. Kim, W.M.W. Hwu, A scalable, numerically stable, high-performance tridiagonal solver using GPUs, in: Int. Conf. High Perform. Comput. Networking, Storage Anal. SC, 2012, pp. 1–11.
[14] A. Toselli, O.B. Widlund, Domain Decomposition Methods — Algorithms and Theory, Springer Science & Business Media, Berlin, 2005.
[15] X.C. Cai, O.B. Widlund, Multiplicative Schwarz algorithms for some nonsymmetric and indefinite problems, SIAM J. Numer. Anal. 30 (4) (1993) 936–952, https://doi.org/10.1137/0730049.
[16] Y.A. Kuznetsov, New algorithms for approximate realization of implicit difference schemes, Russ. J. Numer. Anal. Math. Model. 3 (2) (1988) 99–114, https://doi.org/10.1515/rnam.1988.3.2.99.
[17] T.P. Mathew, P.L. Polyakov, G. Russo, J. Wang, Domain decomposition operator splittings for the solution of parabolic equations, Soc. Ind. Appl. Math. 19 (3) (1998) 912–932.
[18] J. Yang, D. Yang, Additive Schwarz methods for parabolic problems, Appl. Math. Comput. 163 (1) (2005) 17–28, https://doi.org/10.1016/j.amc.2004.03.025.
[19] J. Qiang, Additive Schwarz algorithms for parabolic problems, Appl. Math. Comput. 208 (2) (2009) 531–541, https://doi.org/10.1016/j.amc.2008.12.045.

[20] S. Li, R. Chen, X. Shao, Parallel two-level space–time hybrid Schwarz method for solving linear parabolic equations, Appl. Numer. Math. 139 (2019) 120–135, https://doi.org/10.1016/j.apnum.2019.01.016.
[21] Q. Du, M. Mu, Z.N. Wu, Efficient parallel algorithms for parabolic problems, SIAM J. Numer. Anal. 39 (5) (2002) 1469–1487, https://doi.org/10.1137/S0036142900381710.
[22] D.J. Magee, K.E. Niemeyer, Accelerating solutions of one-dimensional unsteady PDEs with GPU-based swept time–space decomposition, J. Comput. Phys. 357 (2018) 338–352, https://doi.org/10.1016/j.jcp.2017.12.028.
[23] H. Shi, H. Liao, Unconditional stability of corrected explicit-implicit domain decomposition algorithms for parallel approximation of heat equations, SIAM J. Sci. Comput. 44 (4) (2006) 1584–1611.
[24] X. Sun, Y. Zhuang, Stabilized explicit-implicit domain decomposition methods for the numerical solution of parabolic equations, SIAM J. Sci. Comput. 24 (1) (2002) 335–358.
[25] Z. Sheng, G. Yuan, X. Hang, Unconditional stability of parallel difference schemes with second order accuracy for parabolic equation, Appl. Math. Comput. 184 (2) (2007) 1015–1031, https://doi.org/10.1016/j.amc.2006.07.003.
[26] L. Zhu, G. Yuan, Q. Du, An explicit-implicit predictor-corrector domain decomposition method for time dependent multi-dimensional convection diffusion equations, Numer. Math. Theory Methods Appl. 2 (3) (2009) 301–325.
[27] Y. Akhavan, D. Liang, M. Chen, Second order in time and space corrected explicit–implicit domain decomposition scheme for convection–diffusion equations, J. Comput. Appl. Math. 357 (2019) 38–55, https://doi.org/10.1016/j.cam.2019.02.017.
[28] C. Du, D. Liang, An efficient S-DDM iterative approach for compressible contamination fluid flows in porous media, J. Comput. Phys. 229 (12) (Jun. 2010) 4501–4521, https://doi.org/10.1016/j.jcp.2010.02.019.
[29] D. Liang, C. Du, The efficient S-DDM scheme and its analysis for solving parabolic equations, J. Comput. Phys. 272 (2014) 46–69, https://doi.org/10.1016/j.jcp.2014.04.015.
[30] Z. Zhou, D. Liang, Y. Wong, The new mass-conserving S-DDM scheme for two-dimensional parabolic equations with variable coefficients, Appl. Math. Comput. 338 (2018) 882–902, https://doi.org/10.1016/j.amc.2018.06.021.
[31] D. Liang, Z. Zhou, The conservative splitting domain decomposition method for multicomponent contamination flows in porous media, J. Comput. Phys. 400 (2020) 108974, https://doi.org/10.1016/j.jcp.2019.108974.
[32] Z. Zhou, D. Liang, The mass-preserving and modified-upwind splitting DDM scheme for time-dependent convection–diffusion equations, J. Comput. Appl. Math. 317 (2017) 247–273, https://doi.org/10.1016/j.cam.2016.10.031.
[33] J. Chen, D. Yang, Explicit / implicit and Crank – Nicolson domain decomposition methods for parabolic partial differential equations, Comput. Math. Appl. 77 (7) (2019) 1841–1863, https://doi.org/10.1016/j.camwa.2018.11.020.
[34] L. Zhu, G. Yuan, Q. Du, An explicit-implicit predictor-corrector domain decomposition method for time dependent multi-dimensional convection diffusion equations, Numer. Math. Theory Methods Appl. 2 (3) (2009) 301–325.
[35] O. Axelsson, I. Gustafsson, A modified upwind scheme for convective transport equations and the use of a conjugate gradient method for the solution of non-symmetric systems of equations, IMA J. Appl. Math. 23 (3) (1979) 321–337, https://doi.org/10.1093/imamat/23.3.321.
[36] J. Douglas, T.F. Russel, Numerical methods for convection-dominated diffusion problems based on combining the method of characteristics with finite element or finite difference procedures, Soc. Ind. Appl. Math. 19 (5) (1982) 871–885.
[37] Y. Jun, T.Z. Mai, IPIC domain decomposition algorithm for parabolic problems, Appl. Math. Comput. 177 (1) (2006) 352–364, https://doi.org/10.1016/j.amc.2005.11.017.
[38] H.L. Liao, H.S. Shi, Z.Z. Sun, Corrected explicit-implicit domain decomposition algorithms for two-dimensional semilinear parabolic equations, Sci. China Ser. A 52 (11) (Nov. 2009) 2362–2388, https://doi.org/10.1007/s11425-009-0040-8.
[39] J. Glass, W. Rodi, A high order numerical scheme for scalar transport, Comput. Methods Mech. Eng. 31 (1982) 337–358.
[40] H.H. Liu, J.H. Dane, An interpolation-corrected modified method of characteristics to solve advection-dispersion equations, Adv. Water Resour. 19 (6) (1996) 359–368, https://doi.org/10.1016/0309-1708(96)00014-0.
[41] T.L. Tsai, J.C. Yang, L.H. Huang, Characteristics method using cubic-spline interpolation for advection-diffusion equation, J. Hydraul. Eng. 130 (6) (2004) 580–585, https://doi.org/10.1061/(ASCE)0733-9429(2004)130:6(580).
[42] L. Su, W. Wang, H. Wang, A characteristic difference method for the transient fractional convection-diffusion equations, Appl. Numer. Math. 61 (8) (2011) 946–960, https://doi.org/10.1016/j.apnum.2011.02.007.
[43] K. Fu, D. Liang, The conservative characteristic FD methods for atmospheric aerosol transport problems, J. Comput. Phys. 305 (2016) 494–520, https://doi.org/10.1016/j.jcp.2015.10.049.
[44] Z. Zhou, X. Sun, H. Pan, Y. Wang, An efficient characteristic finite difference S-DDM scheme for convection–diffusion equations, Comput. Math. Appl. 80 (12) (2020) 3044–3065, https://doi.org/10.1016/j.camwa.2020.10.023.
[45] Y. Yuan, The upwind finite difference fractional steps methods for two-phase compressible flow in porous media, Numer. Methods Partial Differ. Equ. 19 (1) (2003) 67–88, https://doi.org/10.1002/num.10036.
[46] C. Li, Y. Yuan, Nonoverlapping domain decomposition characteristic finite differences for three-dimensional convection-diffusion equations, Numer. Methods Partial Differ. Equ. 28 (1) (2012) 17–37, https://doi.org/10.1002/num.20605.
[47] H. Kim, S. Hong, H. Lee, E. Seo, H. Han, Compiler-assisted GPU thread throttling for reduced cache contention, in: ICPP 2019, 2019, pp. 1–10.
[48] A. Foadaddini, S.A. Zolfaghari, H. Mahmoodi Darian, H. Saadatfar, An efficient GPU-based fractional-step domain decomposition scheme for the reaction–diffusion equation, Comput. Appl. Math. 39 (4) (2020) 1–35, https://doi.org/10.1007/s40314-020-01357-7.
[49] Y. Zhang, J. Cohen, J.D. Owens, Fast tridiagonal solvers on the GPU, ACM SIGPLAN Not. 45 (5) (2010) 127, https://doi.org/10.1145/1837853.1693472.