Computational and Applied Mathematics (2020) 39:219
https://doi.org/10.1007/s40314-020-01244-1

Parallel performance analysis of coupled heat and fluid flow in parallel plate channel using CUDA

Asif Afzal[1] · Zahid Ansari[2] · M. K. Ramis[1]

Received: 4 December 2019 / Revised: 18 June 2020 / Accepted: 2 July 2020 / Published online: 15 July 2020

Abstract
The heat transfer analysis coupled with fluid flow is important in many real-world application areas varying from micro-channels to spacecraft's. Numerical prediction of thermal and fluid flow situation has become very common method using any computational fluid dynamics software or by developing in-house codes. One of the major issues pertinent to numerical analysis lies with immense computational time required for repeated analysis. In this article, technique applied for parallelization of in-house developed generic code using CUDA and OpenMP paradigm is discussed. The parallelized finite-volume method (FVM)-based code for analysis of various problems is analyzed for different boundary conditions. Two GPUs (graphical processing units) are used for parallel execution. Out of four functions in the code (U, V, P, and T), only P function is parallelized using CUDA as it consumes 91% of computational time and the rest functions are parallelized using OpenMP. Parallel performance analysis is carried out for 400, 625, and 900 threads launched from host for parallel execution. Improvement in speedup using CUDA compared with speedup using complete OpenMP parallelization on different computing machines is also provided. Parallel efficiency of the FVM code for different grid size, Reynolds number, internal flow, and external flow is also carried out. It is found that the GPU provides immense speedup and outperforms OpenMP largely. Parallel execution on GPU gives results in a quite acceptable amount of time. The parallel efficiency is found to be close to 90% in internal flow and 10% for external flow.

Keywords Parallelization · Conjugate heat transfer · CUDA · OpenMP · Speedup · Parallel efficiency

Mathematics Subject Classification 65K05 · 65Y05

Communicated by Jorge X. Velasco.

✉ Asif Afzal
asif.afzal86@gmail.com

✉ M. K. Ramis
ramismk@pace.edu.in

[1] Department of Mechanical Engineering, P. A. College of Engineering (Affiliated to Visvesvaraya Technological University Belagavi), Mangaluru, India

[2] Department of Computer Science and Engineering, P. A. College of Engineering (Affiliated to Visvesvaraya Technological University Belagavi), Mangaluru, India

1 Introduction

Conjugate heat transfer is a phenomenon in which conduction mode of heat transfer in solid is combined with convection mode of heat transfer in fluid. Conduction usually dominates in solids and convection in fluids (Ate et al. 2010). There is wide area of applications in which conjugate heat transfer phenomenon is observed. For example, the optimal design of heat exchanger involves the combination of heat transfer by conduction, in the walls of heat exchanger, and by convection in the flowing fluid. In most of the electronic devices for regulating optimal temperature level of heat sink, in design and development of heating furnace used in metallurgical process, in turbo jet engine flow of hot gases, etc., the conjugate analysis is important. For safe, reliable, and efficient performance of electrochemical energy conversion system and many more, the conjugate condition is most appropriate (Cukurel et al. 2012; He et al. 2011; Guan et al. 2017). Conjugate study can be performed experimentally or numerical method. Numerical methods nowadays are used to simulate fluid flow situations for problems ranging from molecular level to global level. Numerical methods have advanced from the analysis of flow over two-dimensional configurations to three-dimensional configurations. Numerical methods are extensively used as a design and optimizing tool in industries. With the use of computational analysis, multi-phase flow modeling has become easy, adopting huge fine mesh resolution. Generally, the numerical simulation is performed by running the codes/software on modern-day computing machines (Bhatti et al. 2020; Khan et al. 2019, 2020; Ullah et al. 2019).

The thermal and fluid flow behavior study related to parallel plate channel has been reported in the abundant available literature. Different application aspects were covered under various operating conditions. The areas of application mainly include electronic components, equipment and devices, nuclear fuel elements, electric vehicle battery system, heat exchangers, solar flat plate collector, mini and micro-channels, etc. (Meng et al. 2016; Lindstedt and Karvinen 2017; Poddar et al. 2015; Adelaja et al. 2014). The arrangement of the channel formed by the set of plates in these studies is horizontal, vertical, or inclined. Among the research available in the subject of parallel plate channel, some of the investigations are purely based on the uncoupled mode of heat transfer, while some other investigations are based on coupled mode of heat transfer (Arici and Aydin 2009; Bilir 2002; Harman and Cole 2001). For numerical analysis of conjugate heat transfer and fluid flow analysis, finite-difference method (FDM) or finite-volume method (FVM) is most commonly adopted. SIMPLE algorithm to solve the Navier–Stokes (NS) equation for fluid flow analysis is a very popular method used by researcher in these areas. However, this algorithm is compute intensive and slow in convergence and, hence, demands parallelization for reducing the computational cost achieving faster convergence. To the best of our knowledge, parallelization of this algorithm in this field is not reported in the literature. The following literature review is pertinent to parallelization of various codes developed for thermal and fluid flow analysis.

Gropp et al. (2001) demonstrated parallelization of FUN3D code developed at NASA. This code uses an FVM-based discretization, such that for convective flux approximation, variable order Roe scheme is used, and for viscous terms, Galerkin discretization is used. The subdomain parallelization of FUN3D code was successfully demonstrated using MPI (message-passing interface) tool. The flux calculation phase needed 60% of computational time, which was parallelized using OpenMP (open multi-processing) tool. Transient NS equation for incompressible flows in three dimension for analysis of shear flows was solved by Passoni et al. (2001) developing a code. The parallelization of the developed computational code was achieved using MPI applying their own schemes (scheme A/B/C). The parallel performance was analyzed using two computers with different processors and for various grid points. The results show that the parallel performance is better for problem with medium grid size than larger size. 91% of parallel efficiency on doubling the processors and on eightfold increase in processors 60% efficiency was obtained. Schulz et al. (2002) used lattice Boltzmann method (LBM) for fluid flow analysis and established data structures to reduce the memory requirements. MPI parallelization for grid size of 2×10^7 was used and achieved about 90% of parallel efficiency. MPI was used for domain-decomposition-based parallelization of reproducing kernel particle method to analyze three-dimensional flow over a cylinder, flow past a building. 70 processors were used to obtain speedup of 35 (Zhang et al. 2002). Peigin and Epstein (2004) used NES code for aerodynamic design optimization which is computationally very expensive. Hence, MPI was used for multilevel parallelization of code NES for optimization. A cluster of 144 processors was used for implementing parallel execution which provided around 95% parallel efficiency.

Eyheramendy (2003) used FEM analysis of lid-driven cavity problem solving for 50182 degrees of freedom (dofs) on a four-processor Compaq machine. By increasing the number of threads for different dofs, the parallel efficiency reduced. Jia and Sunden (2004) used in-house developed CALC-MP which uses FVM and collocated mesh arrangement. To interpolate velocity values at faces this code uses Rhie and Chow method and SIMPLEC (SIMPLE Consistent) for coupling pressure and velocity. Out of three schemes proposed, scheme 1 provided the best parallel performance due to overlap of computational and communication phase. Lehmkuhl et al. (2007) used CFD code ThermoFluids for solving accurately and to have reliable results for industrial problems of turbulent flows. The parallelization of ThermoFLuids was carried out using METIS software on a cluster of ten processors. Oktay et al. (2011) used CFD code FAPeda which applies unstructured FVM-based cell-centered tetrahedral formulation. As the problem is complex, it requires heavy computations, hence, demanding the necessity of parallelization. Using MUMPS library centered on multi-frontal approach, parallelization of FAPeda was achieved. The CFD code GenIDLEST used for simulation of real-world problems was parallelized using OpenMP tool by Amritkar et al. (2012). GenIDLEST solves transient NS equation in a body fitted multi-block coordinate system using cell-centered FVM formulation. OpenMP parallelism by fine-grained tuning on 256 cores, the performance was shown to match with the MPI. In another study, Amritkar et al. (2014) provided parallelization strategies using OpenMP for simulation of dense particulate system by discrete element method (DEM) with the same code GenIDLEST. Rotary kiln heat transfer and fluidized bed problems were selected for demonstration of DEM-CFD coupled problem. OpenMP speedup was twice than the speedup of MPI on 25 cores. Steijl and Barakos (2018) applied Quantum Fourier transform to solve Poisson equation to a vortex-in-cell method. MPI was used for parallelization of Poisson solver required for simulation of quantum circuits. Gorobets et al. (2018) described parallelization of compressible NS equation for viscous turbulent fluid flow. OpenMP + MPI + OpenCL (open computing language) were used for parallelization. Grid size of 29 million was chosen and about 250 GPUs (graphical processing unit) and 2744 cores were employed for scalability and parallel performance analysis. Compute unified device architecture (CUDA) was used by Lai et al. (2018) to parallelize compressible NS equation on NVIDIAs GTX 1070 GPU. When the block size was varied from 50 to 750, the parallel efficiency varied significantly showing a maximum efficiency for a block size of 256.

CFD code ultraFluidX based on LBM was parallelized using CUDA by Niedermeier et al. (2018). 95% efficiency for empty wind tunnel problem and 80% efficiency for wind tunnel with car problem were obtained on eight GPUs. OpenMP was used to parallelize the Green function-based code which requires immense computational time applying 16 number of threads (Shan et al. 2018). For different discretization methods, the speedup up to 12 was achieved using 16 threads (Shan et al. 2018). For two model simulation analysis of bacterial biofilm model and solute simulation, FENICS software was used based on FEM. In modeling of bacteria growth dynamics, huge computational time is spent as reported by Sheraton and Sloot (2018). The computations were performed on 3.20 GHz Intel® Core™ i7-6900 K CPU running on Ubuntu Linux 14.04 using METIS library. The parallelization analysis revealed that during the initial growth of bacteria, the number of grids required is less, so the necessity of parallel execution at this stage is not required (Sheraton and Sloot 2018). Wang et al. (2018) developed in-house CFD code to solve NS equation for compressible viscous flow in three dimensions. CPU (central processing unit) + Intel Xeon-phi co-processors were used for heterogeneous parallel computing employing MPI, OpenMP, and offload programming model. CUDA, OpenMP, and hybrid OpenMP + CUDA-based parallelization for in-house CFD codes is also reported in the literature (Simmendinger and Kügeler 2010; Kafui et al. 2011; Xu et al. 2014; Jacobsen and Senocak 2011). Review articles by Afzal et al. (2017) and Pinto et al. (2016, 2017) provide a detailed insight into parallel computing strategies for different CFD applications.

The above presented past parallelization research works using different tools for several applications are reported for parallelization of in-house developed codes using OpenMP, CUDA, and MPI. These codes mostly belong to fluid flow application for analysis of aero foils, fluidized beds, heat exchangers, etc. Like Amritkar et al.'s (2012) parallelized GenIDLEST code, Niedermeier et al. (2018) parallelized ultraFluidX, Oktay et al. (2011) used CFD code FAPeda, Sheraton and Sloot (2018) parallelized FENICS software, and many more. However, none of the work says about parallelization of CFD codes using CUDA meant for conjugate heat transfer analysis, in which the conduction in solid and convection in fluids is dominated. The parallelization of numerical method for coupled heat and fluid flow problem solved using SIMPLE algorithm using CUDA and OpenMP tool is still a gap. The motivation of this work is to parallelize an in-house built generic CFD code for analysis of various applications using combined OpenMP and CUDA programming model. The prime objectives of this work are listed below:

1. To find the effect of using different GPUs, various thread blocks, and wide range of fluid flow conditions on parallel performance of the parallelized FVM-based CFD code using CUDA.
2. To provide in-depth understanding of speedups and parallel efficiency of the in-house code for different boundary conditions applied for heat-generating battery cells.
3. Comparison between the speedup achieved using OpenMP parallelization on different computing machines with CUDA parallelization on GPUs.

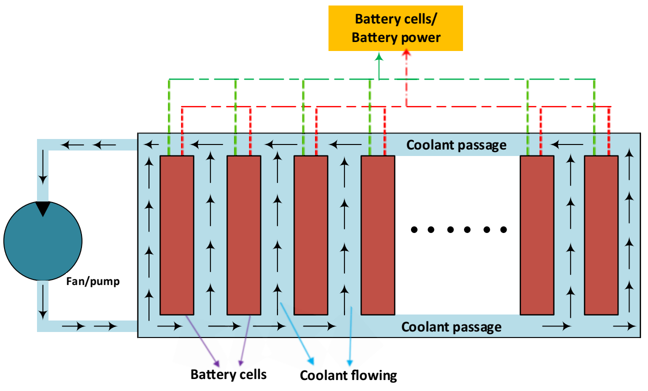
In this work, the incompressible two-dimensional NS equation solved using staggered grid and SIMPLE algorithm for fluid flow analysis can be used for analysis in the area of plates, parallel plates, heat-generating plates, etc. like battery cells, nuclear fuel plate, fins, and many more. As a demonstration, the heat-generating Li-ion battery system is considered in which the maximum temperature has to be kept within safe limits for effective thermal performance of battery system. To avoid the maximum temperature reaching its limit, air as coolant is forced to flow over the battery cells. Hence, this simulation analysis provided by the in-house developed code helps in understanding the thermal behavior of coupled heat and fluid flow scenario, but at the cost of computational expenses. Parallelization of FVM code using CUDA helps in achieving exhaustive numerical analysis in possible time duration. Organization of the rest of the paper is as follows. Section 2 provides the details of the numerical methodology. Section 3 provides the parallelization strategy and results obtained are discussed in Sect. 4. Finally, in Sect. 5, conclusions are drawn.

Nomenclature

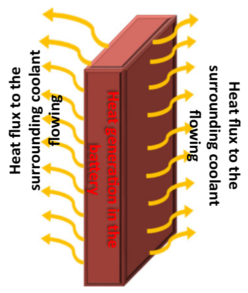
Ar	Aspect ratio of battery cell	V	Non-dimensional velocity along the transverse direction
L	Length of battery cell	w	Half width
k	Thermal conductivity	$\bar{W}$	Non-dimensional width
l_o	Length of extra outlet fluid domain	x	Axial direction
l_i	Length of extra fluid domain	X	Non-dimensional axial direction
L_o	Dimensionless length of extra outlet fluid domain	y	Transverse direction
L_i	Dimensionless length of extra inlet fluid domain	Y	Non-dimensional transverse direction
q'''	Volumetric heat generation	Greek symbols	
$\bar{q}$	Non-dimensional heat flux	α	Thermal diffusivity of fluid
$\bar{S}_q$	Dimensionless volumetric heat generation	ν	Kinematic viscosity of fluid
Pr	Prandtl number	ρ	Density of fluid
Re	Reynolds number	ζ_{cc}	Conduction–convection parameter
T	Temperature	μ	Dynamic viscosity
T_o	Maximum allowable temperature of battery cell	Subscripts	
$\bar{T}$	Non-dimensional temperature	c	center
u	Velocity along the axial direction	f	Fluid domain
U	Non-dimensional velocity along the axial direction	s	Solid domain (battery cell)
u_∞	Free stream velocity	∞	Free stream
v	Velocity along the transverse direction	m	mean
Q_r	Heat removed from surface (non-dimensional)		

2 Numerical methodology

A battery module usually consists of battery cells that are densely packed to obtain higher power densities. For ease of operation and better thermal uniformity, the number of battery cells in each module is less. In this paper, a computationally efficient thermal model used for simulating the thermal behavior of modern electric vehicle battery cells generating uniform heat during charging and discharging operations at a steady state is simulated. A parallel channel with liquid coolant flow is employed to cool the battery cells during operation. The developed thermal model is then used to analyze the thermal behavior of the battery cell for various parameters, as shown in Fig. 1. Alongside, the computational domain is symmetric along the vertical axis; therefore, to reduce the computational cost, only half of the domain through a flow passage, configuration is modeled. Figure 2 shows the simulated domain which consists of two sub-domains, which are the Li-ion battery cells, a vertical parallel flow path channel and the coolant. The fluid flow inside the channel is commonly in laminar



(a) Compact battery module with stack of batteries and coolant passages


Heat flux to the surrounding coolant flowing
Heat generation in the battery
Heat flux to the surrounding coolant flowing

(b) Heat flux from the battery surface

Fig. 1 Schematic view of arrangement of battery cells with air circulation fan and heat generation in batteries

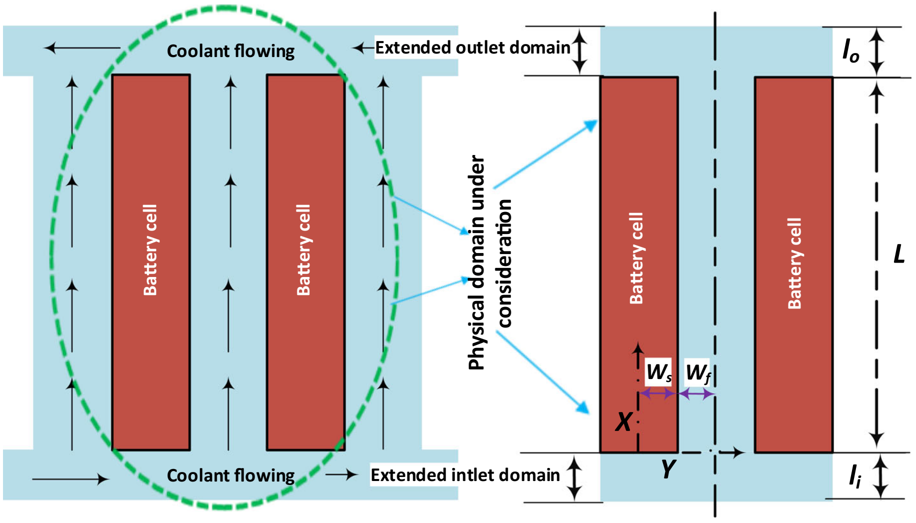


Fig. 2 The symmetric battery (prismatic cell) and coolant flow domain considered for computational analysis

regime owing to the low velocity of the flow inside the channel (Karimi and Li 2012; Xu and He 2013).

The governing equations describing the heat transfer process when discharging/charging the Li-ion battery cell are given by:

$$k_s \nabla^2 T_s + q''' = 0, \tag{1}$$

where q''' is volumetric heat generation term.

The governing equations for two-dimensional, steady, incompressible, laminar, forced-convection flow in the fluid domain are continuity equation, and x and y momentum equations and equation of energy, which are as follows:

$$\nabla u = 0 \tag{2}$$

$$(u\nabla u) = -\frac{1}{\rho}\nabla p + \mu\nabla^2 u \tag{3}$$

$$u\nabla T = \alpha\nabla^2 T_f.. \tag{4}$$

In Eqs. (1)–(4), the term T_s represents the temperature of solid battery, T_f temperature of fluid body, u is velocity, p is pressure, and α and μ are thermal diffusivity and viscosity of the fluid. Equation (2) is mass conversation equation, Eq. (3) is momentum equation, and Eq. (4) is energy equation representing the temperature in fluid domain.

The above equations are non-dimensionalized using the following set of normalizing parameters mentioned in Eq. (5):

$$\begin{aligned}
&\bar{S}_q = \frac{q''' w_s^2}{k_s(T_o - T_\infty)}, \quad C = 4\mathrm{Ar}^2 \quad \bar{T} = \frac{T - T_\infty}{T_0 - T_\infty}, \quad L_i = \frac{l_i}{L}, \quad L_o = \frac{l_o}{L}\\
&X = \frac{x}{L}, \quad U = \frac{u}{u_\infty}, \quad V = \frac{v}{u_\infty}, \quad P = \frac{p}{\rho u_\infty^2}, A_r = \frac{L}{2w_s}\\
&Y_s = \frac{y_s}{w_s}, \quad Y_f = 1 + \frac{y_f}{L}, \quad \bar{W}_f = \frac{w_f}{L}, \quad \zeta_{cc} = \frac{k_f}{k_s}\left[\frac{w_s}{L}\right]\\
&\mathrm{Re} = \frac{u_\infty L}{\nu}, \quad \mathrm{Pr} = \frac{\nu}{\alpha}.
\end{aligned} \tag{5}$$

The above non-dimensionalized parameters are selected based on previous research work that are close to conjugate study of parallel plate channel (Kaladgi et al. 2019; Abdul Razak et al. 2019; Mohamme Samee et al. 2019; Samee et al. 2018).

The final set of non-dimensionalized governing equations are provided in Eqs. (6)–(9):

$$\frac{\partial^2 \bar{T}_s}{\partial X^2} + C\frac{\partial^2 \bar{T}_s}{\partial Y_s^2} + C\bar{S}_q = 0 \tag{6}$$

$$\nabla U = 0 \tag{7}$$

$$U\nabla U = -\nabla P + \frac{1}{\mathrm{Re}}\nabla^2 U \tag{8}$$

$$U\nabla\overline{T_f} = \frac{1}{\mathrm{RePr}}\nabla^2\overline{T_f}. \tag{9}$$

The boundary conditions applied for the above energy, conduction, and momentum equations for the conjugate numerical computations of upward flow of air collecting the heat from the lateral surface of battery cell are provided in detail in Eq. (10):

$$\begin{aligned}
&X = 0; \quad 0 \le Y_s \le 1, \quad U = 0, \quad V = 0, \quad \bar{T}_s = 0\\
&Y_s = 0; \quad 0 \le X \le 1, \quad \frac{\partial \bar{T}_s}{\partial X} = 0\\
&Y_s = 1; \quad 0 \le X \le 1, \quad \frac{\partial \bar{T}_s}{\partial X} = 0\\
&Y_s = 1; \quad 0 \le X \le 1, \quad \bar{T}_s = \bar{T}_f\\
&X = 1; \quad 0 \le Y_s \le 1, \quad \frac{\partial \bar{T}_s}{\partial Y_s} = 0,\\
&Y_f = 1; \quad -L_i \le X \le 0 \ \text{ and } \ L \le X \le L_o \quad \frac{\partial \bar{T}_f}{\partial Y_f} = 0, \quad \frac{\partial U}{\partial Y_f} = 0, \quad V = 0\\
&Y_f = 1; \quad 0 \le X \le L, \quad \frac{\partial T_f}{\partial Y_f} = \frac{1}{\zeta_{cc}}\frac{\partial T_s}{\partial Y_s}, \quad U = 0, \quad V = 0
\end{aligned}$$

$$
\begin{aligned}
&Y_{\mathrm{f}} = 1 + \bar{W}_{\mathrm{f}}; \quad -L_{\mathrm{i}} \le X \le (L + L_{\mathrm{o}}), \quad \frac{\partial \bar{T}_{\mathrm{f}}}{\partial Y_{\mathrm{f}}} = 0, \quad V = 0, \quad \frac{\partial U}{\partial Y_{\mathrm{f}}} = 0 \\
&X = -L_{\mathrm{i}}; \quad 1 \le Y_{\mathrm{f}} \le (1 + \bar{W}_{\mathrm{f}}), \quad \bar{T}_{\mathrm{s}} = 0, \quad U = 1, \quad V = 0 \\
&X = L + L_{\mathrm{o}}; \quad 1 \le Y_{\mathrm{f}} \le (1 + \bar{W}_{\mathrm{f}}), \quad \frac{\partial \bar{T}_{\mathrm{s}}}{\partial X} = 0, \quad \frac{\partial U}{\partial Y_{\mathrm{f}}} = 0, \quad V = 0.
\end{aligned} \tag{10}
$$

2.1 Solution strategy

The numerical solution of the conjugate problem consisting of energy and momentum equations is obtained by employing staggered grid method of finite-volume method (FVM). SIMPLE algorithm is used to solve the coupled momentum and continuity equation to obtain velocity and pressure components. The SIMPLE algorithm steps involved in solving the continuity and momentum equations are as follows:

Step 1: guess and initialize the variables $U*$, $V*$, and $P*$.

Step 2: solve the discretized equations of $U*$ and $V*$ using the guessed pressure field:

$$
\begin{aligned}
(a_{\mathrm{p}}) U^{*}_{i,j} &= -\left(P^{*}_{i+1,j} - P^{*}_{i,j}\right) A_{i,j} + \sum a_{nb} U^{*}_{nb} + b_{i,j} \\
(a_{\mathrm{p}}) V^{*}_{I,Jj} &= -\left(P^{*}_{I,J+1} - P^{*}_{i,j}\right) A_{I,J} + \sum a_{nb} U^{*}_{nb} + b_{I,J}.
\end{aligned}
$$

Step 3: solve the pressure correction equation P' using the previously calculated U' and V':

$$
(a_{\mathrm{p}}) P'_{i,j} = \sum a_{nb} P'_{nb} + F^{*}_{u} + F^{*}_{v}
$$

Step 4: correct the pressure and velocity equations:

$$
P = P^{*} + P' \quad U = U^{*} + U' \quad \text{and} \quad V = V^{*} + V'.
$$

Step 5: update the $U*$, $V*$, and $P*$ values with just computed *values and go back to step 1 till error is within desired limit.

Here, the terms a_{p} and a_{nb}, $b_{i,j}$, and $F^{*}_{u} + F^{*}_{v}$ are sum of neighboring coefficients, source term, and velocity of neighboring points, respectively. Discretization of the governing equations was done by central differencing scheme for continuity and momentum equations. The diffusion equation of cell domain and energy equation of fluid domain are coupled as the conjugate condition at the cell–fluid interface should be satisfied. The $U*$ and $V*$ velocity components arrived in the pressure correction procedure are solved by line-by-line Gauss–Seidel iteration method and Thomas algorithm considering the boundary conditions mentioned earlier and guessed pressure $P*$. The pressure correction P' equation obtained employing the continuity equation is solved by successive over-relaxation (SOR) method using the calculated $U*$ and $V*$ velocity from the previous computations. The corrected pressure P' calculated to satisfy the continuity equation is further used to correct the guessed $U*$, $V*$, and $P*$. At this stage, the temperature $\bar{T}_{\mathrm{s}}$ from the 2-D conduction equation with source term $\bar{S}_q$ and energy equation for $\bar{T}_{\mathrm{f}}$ are solved simultaneously using the obtained corrected U, V, and P values from the previous computations. Line-by-line Gauss–Seidel iteration method and Thomas algorithm are used for solving the $\bar{T}$ of solid and fluid domains. The U, V, and P values are used as a guessed value for the next iteration with some under relaxation and so on until the error in the continuity equation and $\bar{T}$ is $<$ 10e$-$6.

To evaluate the accuracy of the numerical results, grid-independence tests to check the dependence of the obtained results on the grid resolution was conducted. The considered mesh systems have grid size of 42 $\times$ 82, 62 $\times$ 122, and 82 $\times$ 162 in fluid domain and grid size of 82 $\times$ 82, 82 $\times$ 122, and 122 $\times$ 122 in the solid domain, respectively. For typical cases calculated in the present work, the numerical results obtained for non-dimensional temperature for different mesh system were less than 5%. However, to reduce the computational time without affecting the accuracy, a grid size of 62 $\times$ 122 for solid domain and fluid domain is used. The detailed numerical method, boundary conditions, solution strategy, and validation are discussed in Afzal et al. (2019, 2020a, b), which are avoided here for the sake of briefness.

3 Parallelization strategy

Parallel architectures are in significant attention to offer immense computational power by utilizing multiple processing units. The progress in the growth of parallel processing is due to the stagnation of central processing units (CPUs) clock speed. To benefit out of the present multicore/processor and GPUs, the programs have to be developed for parallel execution (Mudigere et al. 2015; Couder-Castaneda et al. 2015; Xu et al. 2015). In this research work, an effort is made to parallelize the developed FVM code for the present conjugate heat transfer problem. Parallelization of the in-house developed indigenous code written in C language is carried out using NVIDIAs GPU GTX 980. Parallel computing paradigm CUDA is employed for the parallelization of the FVM code. Parallelization of the FVM code is achieved using Red and Black Successive over Relaxation (RBSOR) scheme.

For fluid flow conditions like internal flow, external flow, internal flow with outlet domain extended, and internal flow with inlet and outlet domain extended, computational speedup obtained is investigated in detail. Grid size of 42 $\times$ 82, 52 $\times$ 102, 62 $\times$ 122, and 72 $\times$ 142 for internal flow and grid size of 62 $\times$ 24 for inlet and outlet extended domain are adopted for parallel performance analysis. For external flow, the grid sizes chosen are 122 $\times$ 122, 162 $\times$ 162, 202 $\times$ 202, and 242 $\times$ 242 to understand the parallel speedup achieved. In case of internal flow, the spacing between the parallel battery cells $\bar{W}_\mathrm{f} = 0.1$ is kept constant. For both internal and external flow, Re= 250, 750, 1250, and 1750 is considered. The other parameters are fixed to their base values for complete parallelization analysis. Parallel efficiency of the parallelized code is also investigated to understand the fraction of time for useful processor utilization.

3.1 The RBSOR method

The computational time taken by the developed FVM code depending upon on parameters varies from approximately 30 min to 24 h. From the profile analysis of different functions used in the code, it is found that up to 91% of computational time is spent/used on pressure correction function. The remaining time is used by U and V velocity, temperature, and printing output results function. Hence, the major focus is made on parallelizing the pressure correction function using RBSOR scheme and on CMs with different configuration. SOR method is employed for solving the pressure correction equation obtained using the SIMPLE algorithm technique. For the remaining functions, TDMA is used for formulating the corresponding discretized equations. One of the commonly known schemes for parallelization of SOR is red and black scheme. However, the use of wavefront scheme or their combinations is

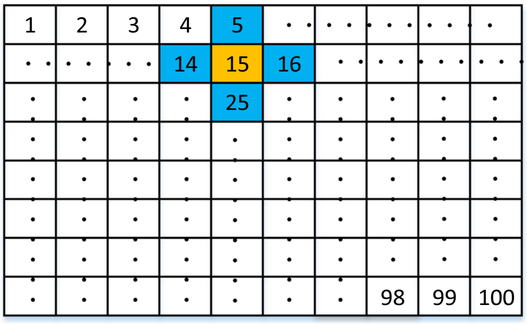


Fig. 3 Serial execution of grid points depending upon the four neighboring cells

not reported in the literature. In the following section, a detailed description of working of RBSOR scheme and wavefront scheme is provided.

The SOR is an important iterative method to solve the system of linear equations. SOR is an expansion/improvement of Gauss–Seidel method that speed ups convergence. It over-relaxes and combines the old values and current values by a factor greater than unity (Niemeyer and Sung 2014). In this work, SOR is used to solve the pressure correction equation with over-relaxation factor equal to 1.8. This pressure correction function consumes maximum computational time as mentioned earlier, due to inner iterations required for correcting the pressure.

The parallel implementation of SOR technique is not easy as it uses the values of neighboring cells/grid points of the current iteration, as shown in Fig. 3. The gird point/cell shown in yellow color (number 15) requires the values of upper, lower, left, and right side cells (number 5, 25, 14, and 16, respectively) shown in blue color. Each grid point is serially executed one after the other taking the newly calculated values of neighboring points. This is mentioned serially from 1 to 100 in Fig. 3. Hence, this brings in sequential dependency and may lead to different results in parallelly executed SOR. To overcome this sequential dependency and parallelize the SOR algorithm, graph coloring methods are used (Abdi and Bitsuamlak 2015). Using this coloring method, the single sweep of SOR can be broken into multiple sweeps which are suitable for parallel processing. The RBSOR scheme can be thought as a compromise among Gauss–Seidel and Jacobi iteration. As shown in Fig. 4, the RBSOR solves by coloring in the checkerboard with alternative red/black grids. At first, all the red cells are computed simultaneously considering the neighboring black points. Then, black cells are computed using the updated red cells parallelly. The RBSOR scheme implementation is mentioned briefly in Algorithm 1. In Algorithm 1, $i_{\min}$ refers to start point of grid number along x-direction and $j_{\min}$ along y-direction. m and n are the maximum numbers of grid points along x- and y-direction given by the user. w is the over-relaxation term which is set to 1.8 in this algorithm.

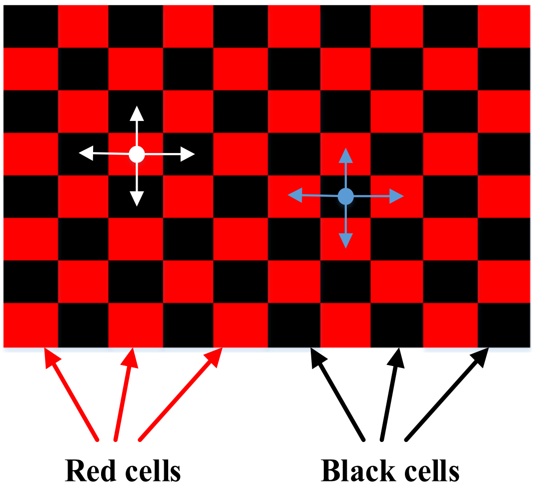


Fig. 4 RBSOR and wavefront scheme for parallel implement of SOR

Algorithm 1: Pseudo code of RBSOR method

```
1. for $i_{min}$ = 1 to $m$ do                                   // iterations for red cells
2.     for $j_{min}$ = 2 - ( $i$ % 2 ) to $n$ do                 // inner iterations
3.         $P^{(k+1)}_{i,j} = (1 - w) * P^{(k)}_{i,j} + N^{(k+1)}_{i,j}$    // $N^{(k+1)}_{i,j}$ is sum of neighbouring points
4.         $j = j + 2$
5.     end for
6. end for
7. for $i_{min}$ = 1 to $m$ do                                   // iterations for black cells
8.     for $j_{min}$ = 1 + ( $i$ % 2 ) to $n$ do                 // inner iterations
9.         $P^{(k+1)}_{i,j} = (1 - w) * P^{(k)}_{i,j} + N^{(k+1)}_{i,j}$    // $N^{(k+1)}_{i,j}$ is sum of neighbouring points
10.        $j = j + 2$
11.    end for
12. end for
```

3.2 Parallelization using CUDA

Existing graphics processing units (GPUs) offer an excessive computational power in the form of graphics hardware. A GPU is an enormously parallelly coprocessing architecture to the CPU and is a set of SIMD (Single Instruction Multiple Data). GPUs and the present CPUs have similar memory hierarchy (Poddar et al. 2015). GPUs are nowadays commonly used as a programmable engine with the use of parallel programming tools like CUDA, OpenAcc (open accelerator), and OpenCL (open computing language). CUDA is a well-known parallel computing tool provided by NVIDIA to create immense parallel applications. Porting of applications on CPU is made easy by CUDA as it avoids the previous graphics pipeline concepts. CUDA establishes high abstraction levels by providing a new programming model for high-performance computing architecture. To enable heterogeneous programming, it consists of set of extensions and to manage devices or memory, it has straightforward APIs (application program interface) (Adelaja et al. 2014; Arici and Aydin 2009; Bilir 2002).

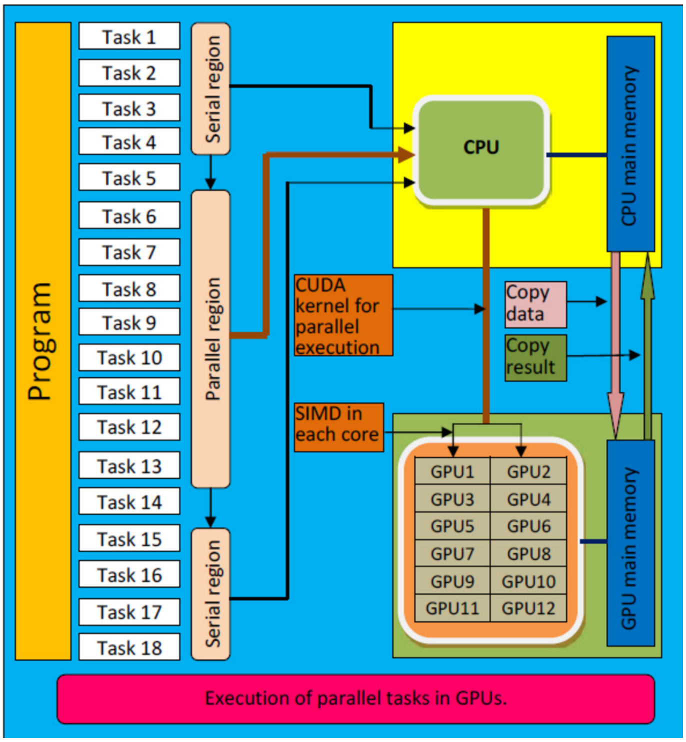


Fig. 5 Overview of memory copy and kernel launching in CUDA paradigm

In CUDA paradigm, the computational core is known as kernel, CPU with its memory is known as host, and GPU with its memory is known as device. The CUDA paradigm overview with its data/result copying and kernel launching is illustrated in Fig. 5. Figure 5 shows the passing/launching of kernel on GPU which parallelly executes using data streams by array (scalable) of multithreaded Streaming Multiprocessors (SMs).

The kernel launched from the host CPU is mapped to GPUs thread grid which has several blocks of threads along with different directions. These threads share the memory within a block which synchronizes. The SMs work together for massive computations managed by employing the architecture of single instruction multiple thread (SIMT). The SIMT architecture describes the characteristics of SMs that are same as all devices. The detailed description of SIMT and multithreading architecture can be found in NVIDIAs GPU programming guidelines in detail in Harman and Cole (2001), Gropp et al. (2001) and Passoni et al. (2001).

In this work, the FVM code parallelization using CUDA is adopted only for pressure correction function as it requires around 91% of the total runtime of the code. The remaining U and V velocity and temperature functions are parallelized using OpenMP for inner for() lop of the TDMA. The steps involved in parallelization using CUDA for the present developed code in mentioned in Algorithm 2. The specifier __global__ is added in the code to be identified by CUDA C++ compiler to be run on GPU and is called by the host program. The other declarations like cudaMalloc, cudaMemcpy, and cudaFree are used for the management of device memory. The kernel for RBSOR has grid and block dimension declared by dim3 command. The grid size is always two-dimensional, whereas the block dimension can be of one-, two-, or three-dimensional. The grid size specifier is for defining number of blocks per

Table 1 Specifications of CMs on which GPUs are installed

CM number	Processor	Frequency (GHz)	RAM (GB)	Number of logical cores	Hard disk storage (TB)
CM1	Intel ® core ™ i7-4970	3.6	16	8	1.3
CM2	Intel ® core ™ i7-4970	3.6	8	8	0.7
CM3	Intel ® core ™ i5-3470 s	2.9	8	4	0.5
CM4	Intel ® core ™ i3-3240	3.4	4	4	0.5

Table 2 Specifications of NVIDIAs' GPU used for parallelization

GPU number	Installed on CM	Engine	CUDA cores	Memory (GB)	Base clock (MHz)	Memory clock (Gbps)
GPU1	CM1	GTX 980	2048	12	1127	7.0
GPU2	CM2	GTX 960	1024	8	1126	7.0

grid and the block specifier is for declaring number of threads per block. __synchthreads() is used to synchronize threads inside a kernel from the same block. In CUDA model global synchronization is difficult, and hence, to enforce it, the kernel is exited before a new kernel is launched. The details of GPUs installed on two different CMs (specifications in Table 1) used for parallel execution are mentioned in Table 2. Four different CMs are employed for comparison of relative increase in speedup with GPU1. The specifications of these four CMs are mentioned in Table 1.

Algorithm 2: Algorithm of CUDA parallelized FVM code

```
1. __global__ void redcells()
2. __global__ void blackcells()
3. Input                      // parametric values and options
4. cudaMalloc()               // memory allocation on device
5. Initialization             // initializing the variables
6. for $It$ = 1 to $It_{max}$ do
7.     update boundary grid points
8.     compute $U$ velocity          // solved using TDMA
9.         omp parallel construct for inner for loop
10.    compute $V$ velocity          // solved using TDMA
11.        omp parallel construct for inner for loop
12.    cudaMemcpy()          // copy data from host to device
13.    compute $P_{prime}$ if RBSOR is opted        // solved using SOR
14.        redcells<<<grid,block>>>() // kernel launching
15.        blackcells<<<grid,block>>>()// kernel launching
16.    cudaMemcpy()          // copy data from device to host
17.    compute $T$ temperature of battery cell  // TDMA
18.        omp parallel construct for inner for loop
19.    compute $T$ temperature of fluid domain // TDMA
20.        omp parallel construct for inner for loop
21. end for
22. cudaFree()          // cleanup alloted memory on device
23. print $U$, $V$, $P$ and $T$ values
```

3.3 Speedup and parallel efficiency

The use of multiple processors to work together simultaneously on a common task is commonly known as parallel computing. The performance of a parallel algorithm implemented on a parallel architecture for parallel computations is measured by speedup and parallel efficiency. The ratio of time taken to execute the sequential algorithm on a single processor to the time taken by parallel algorithm to execute on multiple-processor is known as speedup. Parallel efficiency is defined as the ratio of parallel speedup achieved the number of processers. Parallel efficiency gives the measure of the fraction of computational time at which a processor is used efficiently (Darmana et al. 2006; Walther and Sbalzarini 2009; Wang et al. 2005; Mininni et al. 2011).

According to the definition of speedup and parallel efficiency, they are calculated as given by Eq. (11) (Walther and Sbalzarini 2009; Darmana et al. 2006):

$$S_{(\mathrm{par})} = \frac{T_{(\mathrm{seq})}}{T_{(\mathrm{par})}} \quad E_{(\mathrm{par})} = \frac{S_{(\mathrm{par})}}{N_{(\mathrm{par})}}, \tag{11}$$

where $S_{(\mathrm{par})}$ is the parallel speedup achieved, $T_{(\mathrm{seq})}$ is the elapsed (wall) time taken by the sequential program, $T_{(\mathrm{par})}$ is the wall time taken by the parallel program for execution, $E_{(\mathrm{par})}$ is the parallel efficiency, and $N_{(\mathrm{par})}$ is the number of processors employed for parallel execution. The efficiency of loss occurred due to communication of data, computational task partitioning, processors scheduling, management of data, etc. during parallel implementation is accounted by parallel efficiency. The elapsed time for computations in parallel on N processors can be written as composed of (Shang 2009):

$$T_{(\mathrm{par})} = T_{(\mathrm{seq})N} + T_{(N)} + T_{(\mathrm{comm})N} + T_{(\mathrm{misc})N}. \tag{12}$$

Equation (12) shows the total amount of time taken by a code that is capable of running parallelly on multiple processors. In another way, it can be said that the code which is parallelized (or some part of the program which is parallelized) takes some time for parallel execution on multiple cores for a certain amount of time. This parallel execution time can be faster or slower depending upon the technique adopted for parallelization of the code. This is due to the amount of time taken by the master processor to schedule the parallel tasks, divide the work, divide the data, parallel execution, synchronization, etc., and this entire amount of time taken by the code is accountable under $T_{(\mathrm{par})}$.

Where $T_{(\mathrm{seq})N}$ is the time taken by CPU for computations of the sequential part of the program. $T_{(N)}$ is the time taken by CPU for computations of parallel part of the program on N processors, $T_{(\mathrm{comm})N}$ is the time taken by CPU for communication with N processors, and $T_{(\mathrm{misc})N}$ is the idle time or extra time spent induced due to parallelization of program.

4 Results and discussion of speedup and parallel efficiency using CUDA

Using CUDA parallel computing paradigm, the parallelization of FVM code developed in-house is analyzed in the form of parallel speedup and efficiency. Two GPUs, namely GPU1 and GPU2, are adopted for parallel execution. 400, 625, and 900 threads are launched from the host to device and their speedup is also checked. In this section, speedup and parallel efficiency of the parallelized FVM code are provided in detail. Using RBSOR scheme, various grid sizes, and Res for internal and external flow, the computational time analysis is carried out. In this entire parallel performance analysis, the operating heat and fluid flow parameters are kept fixed at $\bar{S}_q$= 0.5, ζ_{cc}= 0.06, and Ar= 10, for both the flow conditions. For computational

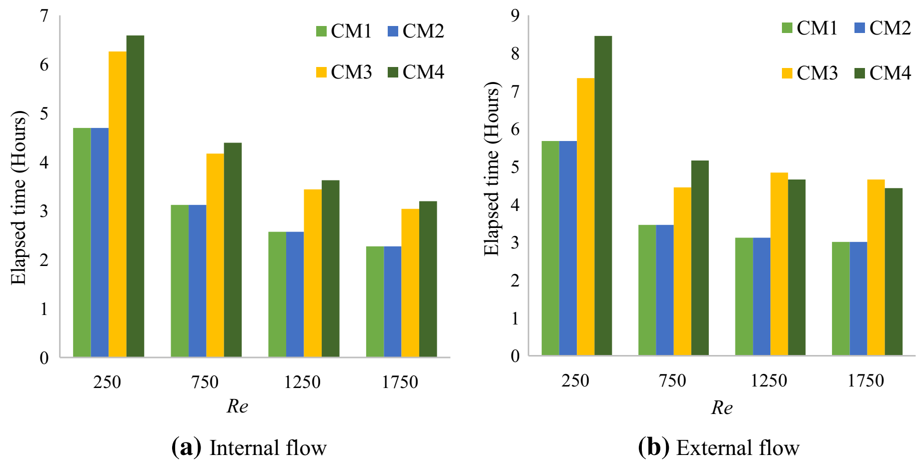


Fig. 6 Elapsed time on different machines for different flow conditions

time analysis, the grid sizes chosen are 42 × 82, 52 × 102, 6 × 122, and 72 × 142 for internal flow. The grid size considered for outlet and inlet domains extended is 24 × 122. In case of external flow 122 × 122, 162 × 162, 202 × 202, and 242 × 242 grid sizes are chosen, while Re is varied from 250 to 1750.

4.1 Elapsed time

The elapsed time of the FVM code for internal and external flow on all the four machines is noted at first, to get an idea of computational cost for different conditions. In Fig. 6 the elapsed time on CM1, CM2, CM3, and CM4 is shown for internal flow and external flow. The flow Re= 250 to 1750 and grid size of 62 × 122 and 202 × 202 for internal and external flow, respectively, is fixed to note the elapsed time. Based on clock speed and RAM of the machines, the elapsed time is minimum for CM1 and maximum for CM4 at Re= 250 for both the flow conditions. Time for CM1 and CM2 are almost the same for all flow Re. This elapsed time is for outlet and inlet domains not extended during internal flow analysis. Surely, if higher grid size and extra flow domains are considered, the computational time increases up to 24 h and above.

4.2 Speedup achieved

The speedup of parallel FVM code achieved for internal and external flow on GPU1 using CUDA paradigm and RBSOR scheme is shown in Fig. 7. The speedup using CUDA is calculated considering the serial performance of the FVM code on CM1. GPU1 is selected by default and 400 threads were launched on the device during each iteration from the host CPU for this entire parallel computations. For different Re and grid size, the speedup achieved on GPU1 shown in Fig. 7 indicates massive parallel performance of the CUDA parallelized FVM code. For internal flow, the speedups are higher than speedups achieved for external flow. The SMs of the GPU perform very fast computations as they are very light threads and do not require any kind of forking/joining mechanism. In internal flow, the inner iterations required for pressure correction function are higher compared to external flow. And

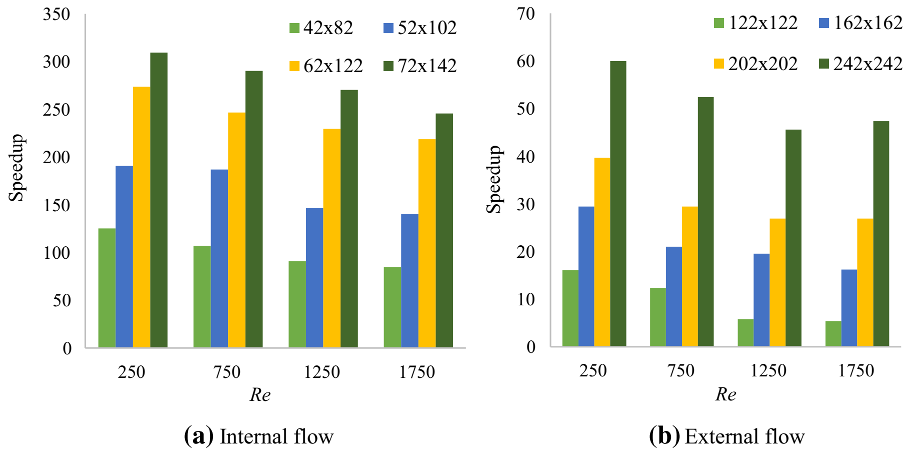


Fig. 7 Speedup obtained for different Re and grid sizes

on another hand, the total computational time for serial execution of FVM code for higher Re is less in case of both internal and external flow as shown previously in Fig. 6. Thus, due to the reduction in serial computational time for higher Re and massive parallel performance capability of GPU, the speedup reduces. Whereas for lower Re due to more utilization of GPU for parallel computation, the speedup increases. Similarly, due to reduced inner iterations in external flow, the speedup is also reduced compared to internal flow.

To analyze the computational performance of GPU when serial computations are made, only one thread on SM during each iteration is launched. To get an idea of elapsed time for serial computations on GPU, only internal flow with 62 $\times$ 122 grid size is selected. The elapsed time calculated is for serial execution of the FVM code in which only the pressure correction functions are made on GPU and the rest computations are made on CPU. The pressure correction function is chosen as it consumes 91% of total time, mentioned earlier. During each iteration due to the copying of memory back and forth to GPU/CPU and launching the kernel, extra computational effort is required. Hence, the computational time required for serial computations on GPU is huge, as shown in Fig. 8a.

When two GPUs with different configuration mentioned in Table 2 are used for parallel computation, the respective speedup obtained is shown in Figure 8b that the speedup on both the GPUs is very close to each other. The default analysis of speedup using CUDA paradigm is performed by launching 400 threads for different conditions explained previously. In Fig. 9, speedup obtained by launching 400, 625, and 900 threads on GPU1 is presented for different Re. The flow chosen is internal with 62 $\times$ 122 grid size without any extended domain. As explained earlier, for higher Re, the speedup using CUDA is comparatively less. It is also seen that by launching higher number threads reduces the speedup further. The reason can be attributed to synchronization time required after parallel computations. As more threads are launched, more synchronization time is required, hence increasing the parallel execution time marginally.

In Fig. 10, speedup obtained with outlet domain extended and with both inlet and outlet domain extended in case of internal is represented. The grid size for the inlet/outlet domain extended is 62 $\times$ 24 for each. The grid size is fixed at 62 $\times$ 122 for internal flow, while Re is changed from 250 to 1750. It is observed that the speedup for both the cases remains nearly the same at all Re. It can also be seen that when the speedup is compared with Fig. 7a, for extended

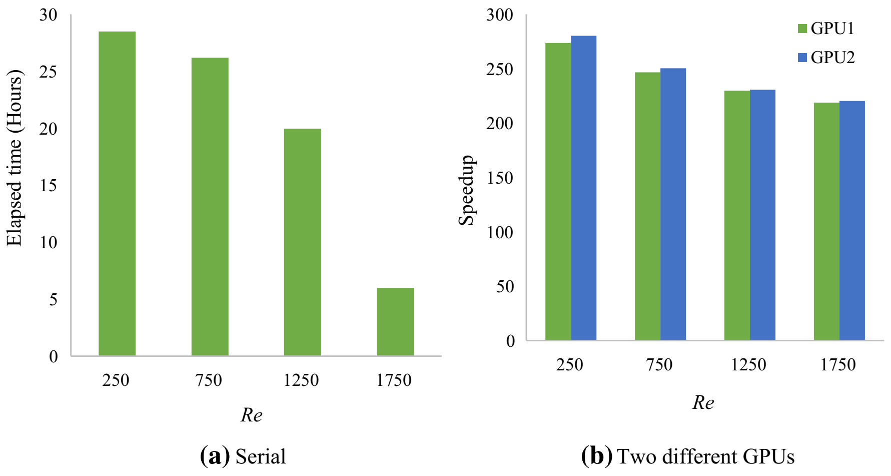


Fig. 8 **a** Time consumption of the FVM code for serial execution on GPU1 and **b** speedup obtained on GPU1 and GPU2

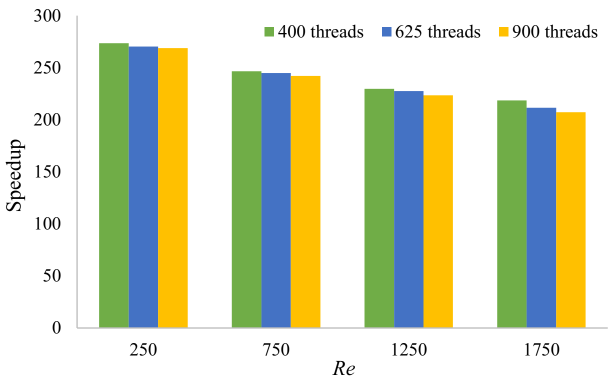


Fig. 9 Speedup obtained by launching kernel with different number of threads on GPU1

domains, the speedup is reduced in internal flow. With the extension of flow domains, the parallel execution time has also increased, which shows that the pressure correction iterations required have drastically increased, reducing the speedup. This analysis provides an insight of behavior of parallel performance of FVM code using SIMPLE algorithm to solve NS equation. One thing that can be deduced is that, without any extended domain, the speedup for internal flow is best, and extension of fluid body before the leading edge and after the trailing edge causes immense computational cost.

Out of curiosity to know how the FVM code behaves computationally, if the extensions of domain are applied for external flow, parallel performance analysis was carried out. In Fig. 11, the speedup obtained for only outlet domain and both the domains extended in case of external flow is depicted. Grid size of 202×202 is fixed for the flow over plate, whereas for outlet/inlet extended domain, 40×202 is fixed for each. Compared to results presented in Fig. 7b, the speedup has increased significantly when extended domains are chosen. However, the speedup for either only outlet domain or both domains extended remains more or less the same for all Re. One interesting aspect noticed is that the fluctuation in speedup with

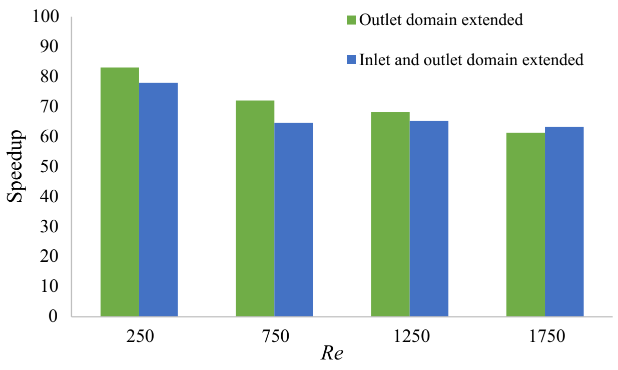


Fig. 10 Speedup in internal flow considering only outlet domain and both the domains extended

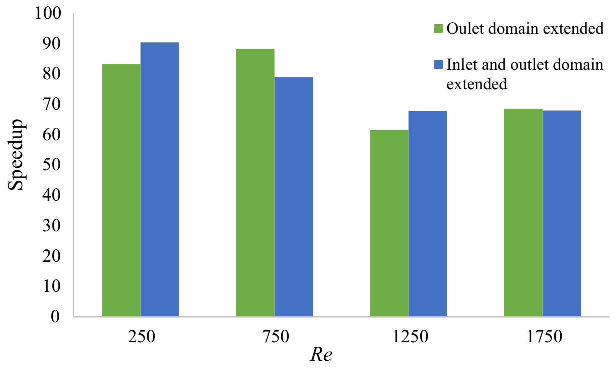


Fig. 11 Speedup in external flow considering only outlet domain and both the domains extended

domains extended is in line with the speedup fluctuations observed from Fig. 7b. The increase in speedup for external flow with extended domain may be attributed to the physical behavior of fluid over/between the plates. According to the boundary layer theory, in case of internal flow, the boundary layers of parallelly placed plates mix and get fully develop near to the leading edge if the spacing between the plates is less. Hence, due to the mixing of boundary layers, the pressure correction required to satisfy the continuity equation for initially guessed velocity and pressure field are very immense. This causes increased computational time of the FVM code. If the spacing between the plates is further reduced, the computational time will inversely increase, and if the spacing is increased, the pressure correction required is less and the related computational time is also less. This fluid behavior is in agreement with reduced speedup obtained for internal flow with extended domain. In contrast to this, in case of external flow, the boundary layers never mix and the flow is parallel to the plate. Therefore, the pressure correction required is very less which leads to reduced computational time, and hence, addition of extended domain or grids causes more efficient computations on SMs of GPU. The parallel efficiency may, however, be more for internal flow with extended domain than that of external flow with extended domain due to better utilization of parallel processors.

The speedup obtained on GPU1 compared to the sequential time of different CMs for external and internal flow is presented in Fig. 12. The previous results of speedup on GPU1 were considering sequential time of CM1. The flow domain considered is without any exten-

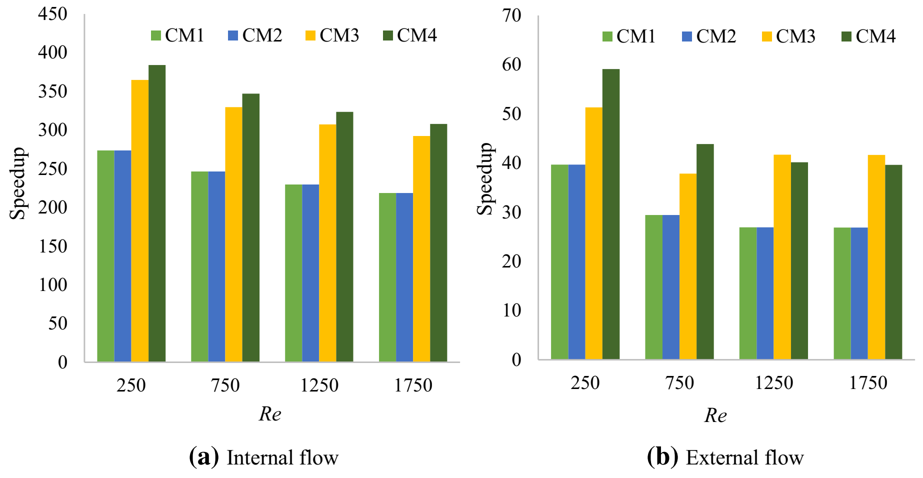


Fig. 12 Speedup obtained using CUDA considering serial time on different computing machines

sion of fluid body in both the cases. The grid size is fixed at 62 × 122 and 202 × 202 for internal and external flow, respectively. As per the preceding discussions, the speedup on CM1 for different Re is the same and similar is for CM2 due to very close configurations of both the CMs. Yet, another similar trend of speedup on GPU1 for increasing Re in internal and external flow when serial time of CM3 and CM4 is considered is obtained. It can be pointed out that the speedup on GPU1 considering serial time of CM1/CM2 is less than the speedup obtained considering serial time of CM3/CM4. The speedup on GPU1 is highest if serial time of CM4 is considered in case of internal and external flow, as well. These speedups obtained are corresponding to the computational capability of the CMs having different configurations. These speedups obtained give an idea of massive parallel computational capability of SMs of GPU. However, the speedups are significantly more for internal flow relative to external flow.

As mentioned earlier, the three functions (U, V, and T) are parallelized using OpenMP tool and the P function is parallelized using CUDA tool. In another attempt, the P function is also parallelized using OpenMP making the entire FVM code parallelized using OpenMP paradigm. Four CMs, namely CM1, CM2, CM3, and CM4, were selected for analysis of speedup using OpenMP parallelized code. The %improvement in speedup obtained using GPU1 compared to speedup of these four CMs is depicted in Fig. 13. In this analysis, internal flow without any extended domain with grid size 62 × 122 was selected for demonstration. It can be easily seen that the %improvement in speedup obtained using GPU1 compared to OpenMP speedup on different CMs is very close to each other. The prime reason behind this same %improvement in speedup is due to very low or unnoticeable speedup obtained on CMs using OpenMP when compared with speedup using CUDA. Hence, from this analysis, it is crystal clear that the compute capability of GPUs is incomparable with that of CPUs. It is also clear that the OpenMP parallelization is 100 times slower than the CUDA-based parallelization.

In Fig. 14, the speedup achieved during each iteration on GPU1 for internal flow considering Re= 750 and different grid sizes is depicted. During the initial iterations, the speedup is very high and as the iterations continue, the speedup reduces, and finally, they reach close to speed up around 75 for the last iteration. The speedups look unrealistic as such speedups are normally not achievable. In this work, the inner iterations of pressure correction equa-

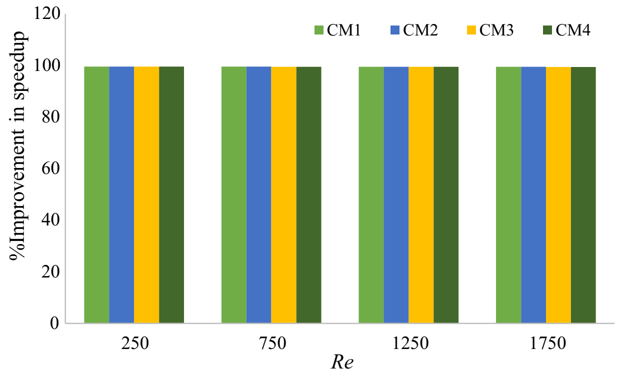


Fig. 13 %Improvement in speedup using GPU1 with respect to speedup obtained on four CMs using OpenMP

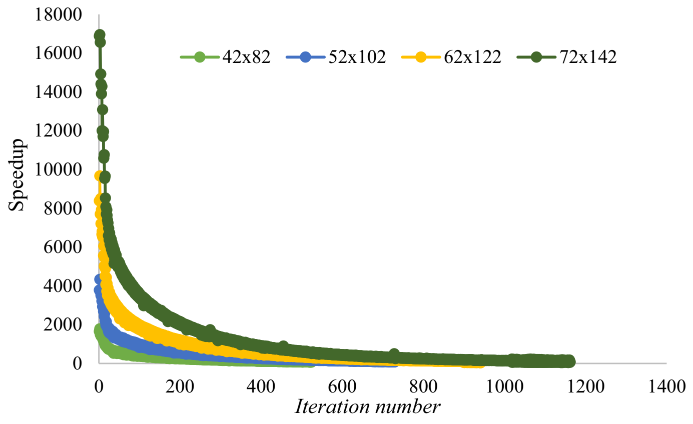


Fig. 14 Speedup achieved during each iteration on GPU1 for different grid size

tion parallelized using red–black scheme are completely computed on GPUs. 91% of time is used by pressure function as the inner iterations are high in number. Using an outer for loop to launch the kernels for fixed number of times on GPU, this speedup is possible. For higher grid size, the speedup at initial iterations is very high compared to lower grid size during the initial iterations. The speedup obtained during the initial iterations is very high due to immense inner pressure corrections required as the velocity and pressure are in guessed state. As the iterations proceed, the velocity and pressure field values are computed and, hence, the pressure correction required also reduces. The speedup obtained and change in it are directly related with the inner pressure corrections computed. If pressure corrections are highly required, then the serial execution takes immense time for that iteration. Hence, during that iteration, if parallel execution of pressure corrections is performed on GPU, it gives enormous speedup.

4.3 Parallel efficiency of the FVM code

The parallel efficiency of FVM code using CUDA and applying RBSOR scheme and 400 threads on GPU1 for internal and external flow is shown in Fig. 15. It can be easily seen from

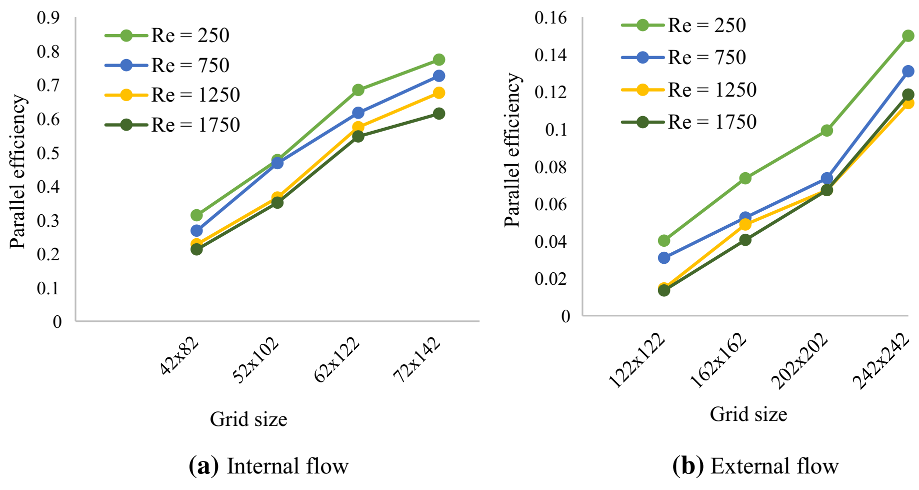


Fig. 15 Parallel efficiency obtained for different grid size and Re

Fig. 15a, b for internal and external flow that for lower grid size, the parallel efficiency is lowest, and for higher grid size, it is highest irrespective of any Re. In internal flow problem, the parallel efficiency is higher compared to external flow at all Re due to increased speedup obtained on GPU1, as shown in Fig. 7. As the speedups are highest at 72 × 122 grid size, the parallel efficiency can be seen to be continuously improving. The parallel efficiency indicates the efficiency of utilization of multi-processors for parallel execution. Basically, if the time taken by the code for serial execution is more, the associated parallel execution time will be less. This indicates that for the most of the time, the code is run parallelly. If the number of cores is more and the amount of computations is less than the parallel efficiency reduces. Therefore, for efficient utilization of multiple cores, the code must be parallelized with a proper strategy. For lower Re, the serial time is high due to more number of inner iterations involved in pressure correction of large boundary layer. For high Re, the boundary layer is closer to solid part and, hence, the inner iterations are less. As the iterations are more for low Re, the serial time is also more which, in turn, improves the ability of code to utilize more efficiently the multiple cores. Therefore, for lower Re, the parallel efficiency is higher. An increase in grid size also causes further improvement in parallel efficiency due to better utilization of SMs of GPU for parallel computations. The parallel efficiency for all Re in external flow is found to be very close to each other, but increases with grid size. However, in the case of external flow, very high grid size can be used to improve the parallel efficiency. However, the use of higher grid size for the present problem is not required. Nevertheless, to increase the parallel efficiency, the scalable problem in external flow is necessary.

5 Conclusions

Parallel performance analysis of the parallelized FVM code developed in-house is analyzed in the form of parallel speedup and parallel efficiency. CUDA parallel computing paradigm is used for parallelization applying the RBSOR scheme. Four computing machines CM1, CM2, CM3, and CM4 for OpenMP parallelization are employed. For computational time analysis, the grid sizes chosen are 42 × 82, 52 × 102, 6 × 122, and 72 × 142 for internal flow. The grid size considered for outlet and inlet domain extended is 24 × 122. In case of external flow 122 × 122, 162 × 162, 202 × 202, and 242 × 242 grid sizes are chosen, while Re is varied from 250 to 1750.

From the complete speedup and parallel efficiency analysis of the parallelized FVM code using different methods, the following important conclusions are drawn.

1. Parallelization using CUDA paradigm gives massive speedup for the present FVM code. In case of internal flow, the speedup is very high compared to speedup achieved in case of external flow.
2. The use of two different GPUs, mainly GPU1 and GPU2, have provided very similar speedup. If different block size of threads is launched, that is 400, 625, and 900 threads, the speedup is again nearly the same, but is the highest for 400 threads.
3. For internal flow considering inlet and outlet domain extended reduces the speedup significantly compared to internal flow without any extended domain. Whereas the speedup increased for external flow when extended domains are considered.
4. Speedup on GPU considering the serial time of CM1/CM2 is the same. If serial time of CM3 and CM4 is considered, the speedup on GPU significantly improves due to the low frequency of CM3 and CM4.
5. The %improvement in speedup using CUDA compared to speedup obtained using OpenMP parallelization is very immense.
6. The parallel efficiency in case of internal flow improves with increasing grid size for all Re considered. And in external flow at all Re, the parallel efficiency remains nearly the same with slight improvement for the increase in grid size.

References

Abdi DS, Bitsuamlak GT (2015) Asynchronous parallelization of a CFD solver. J Comput Eng. https://doi.org/10.1155/2015/295393

Abdul Razak RK, Afzal A, Mohammed Samee AD, Ramis MK (2019) Effect of cladding on thermal behavior of nuclear fuel element with non-uniform heat generation. Prog Nucl Energy 111:1–14. https://doi.org/10.1016/j.pnucene.2018.10.013

Adelaja AO, Dirker J, Meyer JP (2014) Effects of the thick walled pipes with convective boundaries on laminar flow heat transfer. Appl Energy 130:838–845. https://doi.org/10.1016/j.apenergy.2014.01.072

Afzal A, Ansari Z, Faizabadi A, Ramis M (2017) Parallelization strategies for computational fluid dynamics software: state of the art review. Arch Comput Methods Eng 24:337–363. https://doi.org/10.1007/s11831-016-9165-4

Afzal A, Mohammed Samee AD, Abdul Razak RK, Ramis MK (2019) Effect of spacing on thermal performance characteristics of Li-ion battery cells. J Therm Anal Calorim 135:1797–1811. https://doi.org/10.1007/s10973-018-7664-2

Afzal A, Samee ADM, Razak RKA, Ramis MK (2020a) Thermal management of modern electric vehicle battery systems (MEVBS). J Therm Anal Calorim. https://doi.org/10.1007/s10973-020-09606-x

Afzal A, Ansari Z, Ramis MK (2020b) Parallelization of numerical conjugate heat transfer analysis in parallel plate channel using OpenMP. Arab J Sci Eng. https://doi.org/10.1007/s13369-020-04640-1

Amritkar A, Tafti D, Liu R, Kufrin R, Chapman B (2012) OpenMP parallelism for fluid and fluid-particulate systems. Parallel Comput 38:501–517. https://doi.org/10.1016/j.parco.2012.05.005

Amritkar A, Deb S, Tafti D (2014) Efficient parallel CFD-DEM simulations using OpenMP. J Comput Phys 256:501–519. https://doi.org/10.1016/j.jcp.2013.09.007

Arici ME, Aydin O (2009) Conjugate heat transfer in thermally developing laminar flow with viscous dissipation effects. Heat Mass Transf 45:1199–1203. https://doi.org/10.1007/s00231-009-0494-9

Ate A, Darici S, Bilir E (2010) Unsteady conjugated heat transfer in thick walled pipes involving two-dimensional wall and axial fluid conduction with uniform heat flux boundary condition. Int J Heat Mass Transf 53:5058–5064. https://doi.org/10.1016/j.ijheatmasstransfer.2010.07.059

Bhatti MM, Ullah Khan S, Anwar Bég O, Kadir A (2020) Differential transform solution for Hall and ion-slip effects on radiative-convective Casson flow from a stretching sheet with convective heating. Heat Transf Res 49:872–888

Bilir Ş (2002) Transient conjugated heat transfer in pipes involving two-dimensional wall and axial fluid conduction. Int J Heat Mass Transf 45:1781–1788. https://doi.org/10.1016/S0017-9310(01)00270-8

Couder-Castaneda C, Barrios-Pina H, Gitler I, Arroyo M (2015) Performance of a code migration for the simulation of supersonic ejector flow to SMP, MIC, and GPU using OpenMP, OpenMP + LEO, and OpenACC directives. Sci Program. https://doi.org/10.1155/2015/739107

Cukurel B, Arts T, Selcan C (2012) Conjugate heat transfer characterization in cooling channels. J Therm Sci 21:286–294. https://doi.org/10.1007/s11630-012-0546-1

Darmana D, Deen NG, Kuipers JAM (2006) Parallelization of an Euler-Lagrange model using mixed domain decomposition and a mirror domain technique: application to dispersed gas-liquid two-phase flow. J Comput Phys 220:216–248. https://doi.org/10.1016/j.jcp.2006.05.011

Eyheramendy D (2003) Object-Oriented parallel CFD with JAVA. Parallel Comput Fluid Dyn Adv Numer Methods Softw Appl 2004:409–416. https://doi.org/10.1016/B978-044451612-1/50052-4

Gorobets A, Soukov S, Bogdanov P (2018) Multilevel parallelization for simulating compressible turbulent flows on most kinds of hybrid supercomputers. Comput Fluids 173:171–177. https://doi.org/10.1016/j.compfluid.2018.03.011

Gropp WD, Kaushik DK, Keyes DE, Smith BF (2001) High-performance parallel implicit CFD. Parallel Comput 27:337–362. https://doi.org/10.1016/S0167-8191(00)00075-2

Guan T, Zhang J, Shan Y, Hang J (2017) Conjugate heat transfer on leading edge of a conical wall subjected to external cold flow and internal hot jet impingement from chevron nozzle—part 1: experimental analysis. Int J Heat Mass Transf 106:329–338. https://doi.org/10.1016/j.ijheatmasstransfer.2016.06.101

Harman SA, Cole KD (2001) Conjugate heat transfer from a two-layer substrate model of a convectively cooled circuit board. J Electron Package 123:156–158. https://doi.org/10.1115/1.1348011

He J, Liu L, Jacobi AM (2011) Conjugate thermal analysis of air-cooled discrete flush-mounted heat sources in a horizontal channel. J Electron Package 133:041001. https://doi.org/10.1115/1.4005299

Jacobsen D, Senocak I (2011) Scalability of incompressible flow computations on multi-GPU clusters using dual-level and tri-level parallelism. In: 49th AIAA aerospace sciences meeting including the new horizons forum and aerospace exposition, p 947

Jia R, Sundén B (2004) Parallelization of a multi-blocked CFD code via three strategies for fluid flow and heat transfer analysis. Comput Fluids 33:57–80. https://doi.org/10.1016/S0045-7930(03)00029-X

Kafui DK, Johnson S, Thornton C, Seville JPK (2011) Parallelization of a Lagrangian–Eulerian DEM/CFD code for application to fluidized beds. Powder Technol 207:270–278

Kaladgi AR, Afzal A, AD MS, MK R (2019) Investigation of dimensionless parameters and geometry effects on heat transfer characteristics of liquid sodium flowing over a vertical flat plate. Heat Transf Asian Res 48:62–79. https://doi.org/10.1002/htj.21368

Karimi G, Li X (2012) Thermal management of lithium-ion batteries for electric vehicles. Int J Energy Res 37:13–24. https://doi.org/10.1002/er.1956

Khan SU, Shehzad SA, Nasir S (2019) Unsteady flow of chemically reactive Oldroyd-B fluid over oscillatory moving surface with thermo-diffusion and heat absorption/generation effects. J Braz Soc Mech Sci Eng 41:72

Khan N, Nabwey HA, Hashmi MS, Khan SU, Tlili I (2020) A theoretical analysis for mixed convection flow of maxwell fluid between two infinite isothermal stretching disks with heat source/sink. Symmetry (Basel) 12:62

Lai J, Li H, Tian Z (2018) CPU/GPU heterogeneous parallel CFD solver and optimizations. In: Proceedings of 2018 international conference service robotics technology. ACM, Chengdu, China, pp 88–92

Lehmkuhl O, Borrell R, Soria M, Oliva A (2007) TermoFluids: a new parallel unstructured CFD code for the simulation of turbulent industrial problems on low cost PC cluster. Parallel Comput Fluid Dyn 2009:275–282. https://doi.org/10.1007/978-3-540-92744-0

Lindstedt M, Karvinen R (2017) Conjugated heat transfer from a uniformly heated plate and a plate fin with uniform base heat flux. Int J Heat Mass Transf 107:89–95. https://doi.org/10.1016/j.ijheatmasstransfer.2016.10.079

Meng F, Dong S, Wang J, Guo D (2016) A new algorithm of global tightly-coupled transient heat transfer based on quasi-steady flow to the conjugate heat transfer problem. Theor Appl Mech Lett 6:233–235. https://doi.org/10.1016/j.taml.2016.08.005

Mininni PD, Rosenberg D, Reddy R, Pouquet A (2011) A hybrid MPI-OpenMP scheme for scalable parallel pseudospectral computations for fluid turbulence. Parallel Comput 37:316–326

Mohamme Samee AD, Afzal A, Ramis MK, Abdul Razak RK (2019) Optimal spacing in heat generating parallel plate channel: a conjugate approach. Int J Therm Sci 136:267–277. https://doi.org/10.1016/j.ijthermalsci.2018.10.025

Mudigere D, Sridharan S, Deshpande A, Park J, Heinecke A, Smelyanskiy M, Kaul B, Dubey P, Kaushik D, Keyes D (2015) Exploring shared-memory optimizations for an unstructured mesh CFD application on modern parallel systems. In: 2015 IEEE international parallel and distributed processing symposium, May 25. IEEE, pp 723–732

Niedermeier CA, Janßen CF, Indinger T (2018) Massively-parallel multi-GPU simulations for fast and accurate automotive aerodynamics. In: 7th European conference on computational fluid dynamics

Niemeyer KE, Sung C-J (2014) Recent progress and challenges in exploiting graphics processors in computational fluid dynamics. J Supercomput 67:528–564. https://doi.org/10.1007/s11227-013-1015-7

Oktay E, Akay HU, Merttopcuoglu O (2011) Parallelized structural topology optimization and CFD coupling for design of aircraft wing structures. Comput Fluids 49:141–145. https://doi.org/10.1016/j.compfluid.2011.05.005

Passoni G, Cremonesi P, Alfonsi G (2001) Analysis and implementation of a parallelization strategy on a Navier–Stokes solver for shear flow simulations. Parallel Comput 27:1665–1685. https://doi.org/10.1016/S0167-8191(01)00114-4

Peigin S, Epstein B (2004) Embedded parallelization approach for optimization in aerodynamic design. J Supercomput 29:243–263. https://doi.org/10.1023/B:SUPE.0000032780.68664.1b

Pinto RN, Afzal A, Navaneeth IM, Ramis MK (2016) Computational analysis of flow in turbines. In: IEEE invention computer technology (ICICT). International Conference Coimbatore, India: (3), pp 1–5. https://doi.org/10.1109/INVENTIVE.2016.7830174

Pinto R, Afzal A, D'Souza L, Ansari Z, Mohammed Samee AD (2017) Computational fluid dynamics in turbomachinery: a review of state of the art. Arch Comput Methods Eng 24:467–479. https://doi.org/10.1007/s1183

Poddar A, Chatterjee R, Chakravarty A, Ghosh K, Mukhopadhyay A, Sen S (2015) Thermodynamic analysis of a solid nuclear fuel element surrounded by flow of coolant through a concentric annular channel. Prog Nucl Energy 85:178–191. https://doi.org/10.1016/j.pnucene.2015.06.018

Samee AdM, Afzal A, Razak Rk A, Mk R (2018) Effect of Prandtl number on the average exit temperature of coolant in a heat-generating vertical parallel plate channel: a conjugate analysis. Heat Transf Asian Res. https://doi.org/10.1002/htj.21330

Schulz M, Krafczyk M, Tölke J, Rank E (2002) Parallelization strategies and efficiency of CFD computations in complex geometries using lattice Boltzmann methods on high-performance computers. High Perform Sci Eng Comput 21:115–122. https://doi.org/10.1007/978-3-642-55919-8

Shan P, Zhu R, Wang F, Wu J (2018) Efficient approximation of free-surface Green function and OpenMP parallelization in frequency-domain wave–body interactions. J Mar Sci Technol. https://doi.org/10.1007/s00773-018-0568-9

Shang Y (2009) A distributed memory parallel Gauss–Seidel algorithm for linear algebraic systems. Comput Math Appl 57:1369–1376. https://doi.org/10.1016/j.camwa.2009.01.034

Sheraton MV, Sloot PMA (2018) Parallel performance analysis of bacterial biofilm simulation models. Computer Science—ICCS. Lecture Notes Computer Science, vol 10862. Springer International Publishing, Berlin, pp 496–505. https://doi.org/10.1007/978-3-319-93713-7

Simmendinger C, Kügeler E (2010) Hybrid parallelization of a turbomachinery CFD code: performance enhancements on multicore architectures. In: Proceedings of the V European conference on computational fluid dynamics ECCOMAS CFD

Steijl R, Barakos GN (2018) Parallel evaluation of quantum algorithms for computational fluid dynamics. Comput Fluids 173:22–28. https://doi.org/10.1016/j.compfluid.2018.03.080

Ullah KS, Ali N, Hayat T, Abbas Z (2019) Heat transfer analysis based on Cattaneo–Christov heat flux model and convective boundary conditions for flow over an oscillatory stretching surface. Therm Sci 23:443–455

Walther JH, Sbalzarini IF (2009) Large-scale parallel discrete element simulations of granular flow. Eng Comput 26:688–697. https://doi.org/10.1108/02644400910975478

Wang X, Guo L, Ge W, Tang D, Ma J, Yang Z et al (2005) Parallel implementation of macro-scale pseudo-particle simulation for particle-fluid systems. Comput Chem Eng 29:1543–1553. https://doi.org/10.1016/j.compchemeng.2004.12.006

Wang YX, Zhang LL, Liu W, Cheng XH, Zhuang Y, Chronopoulos AT (2018) Performance optimizations for scalable CFD applications on hybrid CPU + MIC heterogeneous computing system with millions of cores. Comput Fluids 173:226–236. https://doi.org/10.1016/j.compfluid.2018.03.005

Xu XM, He R (2013) Research on the heat dissipation performance of battery pack based on forced air cooling. J Power Sour 240:33–41. https://doi.org/10.1016/j.jpowsour.2013.03.004

Xu C, Deng X, Zhang L, Fang J, Wang G, Jiang Y et al (2014) Collaborating CPU and GPU for large-scale high-order CFD simulations with complex grids on the TianHe-1A supercomputer. J Comput Phys 278:275–297. https://doi.org/10.1016/j.jcp.2014.08.024

Xu Z, Zhao H, Zheng C (2015) Accelerating population balance-Monte Carlo simulation for coagulation dynamics from the Markov jump model, stochastic algorithm and GPU parallel computing. J Comput Phys 281:844–863. https://doi.org/10.1016/j.jcp.2014.10.055

Zhang LT, Wagner GJ, Liu WK (2002) A parallelized meshfree method with boundary enrichment for large-scale CFD. J Comput Phys 176:483–506. https://doi.org/10.1006/jcph.2002.6999