Engineering Analysis with Boundary Elements 40 (2014) 36–49

Contents lists available at ScienceDirect

Engineering Analysis with Boundary Elements

journal homepage: www.elsevier.com/locate/enganabound

Numerical solution of the two-phase incompressible Navier–Stokes equations using a GPU-accelerated meshless method

Jesse M. Kelly [a], Eduardo A. Divo [b], Alain J. Kassab [c,*]

[a] Institute for Computational Engineering and Sciences, The University of Texas at Austin, Austin, TX, USA
[b] Department of Mechanical Engineering, Embry-Riddle Aeronautical University, Daytona Beach, FL, USA
[c] Department of Mechanical and Aerospace Engineering, University of Central Florida, Orlando, FL, USA

ARTICLE INFO

Article history:
Received 31 October 2012
Accepted 22 November 2013
Available online 20 December 2013

Keywords:
Meshless method
Navier–Stokes
GPU
Fluid flow

ABSTRACT

This paper presents the development and implementation of a Meshless two-phase incompressible fluid flow solver and its acceleration using the graphics processing unit (GPU). The solver is formulated as a Localized Radial-Basis Function Collocation Meshless Method and the interface of the two-phase flow is captured using an implementation of the Level-Set method. The Compute Unified Device Architecture (CUDA) language for general-purpose computing on the GPU is used to accelerate the solver. Through the combined use of the LRC Meshless method and GPU acceleration this paper seeks to address the issue of robustness and speed in computational fluid dynamics. Traditional mesh-based methods require extensive and time-consuming user input for the generation and verification of a computational mesh. The LRC meshless method seeks to mitigate this issue through the use of a set of scattered points that need not meet stringent geometric requirements like those required by finite-volume and finite-element methods, such as connectivity and poligonalization. The method is shown to render very accurate and stable solutions and the implementation of the solver on the GPU is shown to accelerate the solution by several orders.

1. Introduction

1.1. Graphics processing unit acceleration

The graphics processing unit, or GPU, is a specialized piece of computer hardware designed to achieve high performance in the execution of parallel computations required for 3-D graphics. The GPU began as a fixed-function device, with programmers having only predefined options for controlling the graphics pipeline. In 1999, NVIDIA introduced the first programmable vertex engine, see [1], marking the beginning of a trend towards flexible graphics hardware. As more pieces of the graphics pipeline became programmable, programmers began implementing general-purpose, i.e. non-graphics related, programs on the GPU. With the recent introduction of the Compute Unified Device Architecture (CUDA) programming language by NVIDIA, programmers can now use a C-like language to achieve complete general-purpose programmability of the GPU. Different memory spaces and execution configurations on the GPU are exposed to the programmer, offering more flexibility and familiarity than programming within the graphics framework.

In the early 2000s, researchers began implementing solutions of the incompressible Navier–Stokes equations (NSE) on the GPU using shader languages. Bolz et al. [2] implemented Conjugate Gradient and Multigrid solvers on the GPU in order to solve the pressure Poisson equation derived from the NSE. They reported a speed increase factor of 1.6 for multigrid implemented on the GPU for an unstructured matrix system, and a 1.8 times speedup for a structured system. The solution of the NSE was carried out on a structured grid following a finite-difference formulation. Another multigrid implementation on the GPU appears in Goodnight et al. [3]. In this case, multigrid on the GPU was used to accelerate the solution of the vorticity-stream function formulation of the NSE; a maximum speed increase factor of about 2.6 was reported. Liu et al. [4] used the GPU to solve the three-dimensional Navier–Stokes equations following the semi-Lagrangian formulation of Stam [5]. While their method was able to handle complex boundaries, the semi-Lagrangian method Stam presents is not physically accurate enough for engineering applications. Liu did not implement a corresponding solver on the CPU, and provides no data for the level of acceleration achieved using the GPU. Li et al. [6] implemented a fluid solver using the Lattice-Boltzmann method (LBM) and assembly language and reported speedup factors as high as 15 for large lattice sizes. In addition, the solver was reported to be second order accurate in space and time.

All of the papers discussed so far have been implemented using shader languages or assembly language for the GPU. Recent

* Corresponding author.
E-mail addresses: jkelly@ices.utexas.edu (J.M. Kelly), divoe@erau.edu (E.A. Divo), alain.kassab@ucf.edu (A.J. Kassab).

0955-7997/$ - see front matter © 2013 Elsevier Ltd. All rights reserved.
http://dx.doi.org/10.1016/j.enganabound.2013.11.015

Nomenclature

CPU	central processing unit
CUDA	Compute Unified Device Architecture
FDM	finite difference method
FVM	finite volume method
GFDM	generalized finite difference method
GPGPU	general-purpose graphics processing unit programming
GPU	graphics processing unit
LBM	lattice-Boltzmann method
LRC	localized radial-basis function collocation
LSM	level-set method
NSE	Navier–Stokes equations
RBF	radial-basis function

σ	coefficient of surface tension
κ	curvature
ρ	density
$\vec{g}$	gravity
φ	Helmholtz potential
I	interface
ν	kinematic viscosity
$\vec{n}$	unit normal vector
p	Pressure
$\vec{V}$	velocity
μ	dynamic viscosity
$\delta(d)$	Dirac delta function
∇	gradient operator $\left(\frac{\partial}{\partial x}\hat{i}+\frac{\partial}{\partial y}\hat{j}\right)$
∇^2	Laplace operator. $\nabla\cdot\nabla$ for scalar; for vector $\vec{a}$, $\nabla^2\vec{a}=\nabla^2 a_x\hat{i}+\nabla^2 a_y\hat{j}$

implementations using CUDA will now be briefly reviewed. Riegel et al. [7] implemented the LBM using CUDA, observing a speedup factor of 9 over a multi-core CPU. Thibault et al. [8] observed a speedup factor of 13 using a single GPU and a speedup factor of 100 using a quad-GPU setup for solution of the NSE using a MAC grid and finite-differencing, both speedups being with respect to a single-core CPU. Cohen and Molemaker [9] reported that GPU-acceleration provided an 8 times speedup with respect to a high-end eight-core CPU for the solution of the NSE using the Boussinesq approximation and a finite-volume method. Brandvik and Pullan [10] implemented a finite-volume solution of the 3-D Euler equations using CUDA and reported a speedup factor of 16 over a single-core CPU.

1.2. Meshless methods

Classical methods in CFD use a mesh or a grid of distributed points that have a certain predefined connectivity. Three of the most popular and widely used classical methods are the Finite Differencing Method, the Finite Volume Method, and the Finite Element Method. Each of these methods requires the user to generate a mesh in order to discretize the solution domain, and the accuracy and convergence of the solution is strongly dependent on mesh quality, see [11]. Idelsohn and Oñate [12] identify the main difficulties in mesh building as the need for a conforming mesh, in which all nodes must lie at element vertices, the need to adhere to boundary contours, and the need to have well-shaped (non-degenerate) elements. Meshless methods eliminate two of these issues: the connectivity between nodes does not need to be conformant, and since no elements are used the presence of degenerate elements is not an issue.

Meshless methods were first introduced in the 1990s in order to avoid the issues prevalent in mesh-based methods. The identifying feature of a Meshless method is that the shape functions used to represent the field variables depend on the nodal distribution alone and are independent of connectivity, see [13]. Examples of existing meshless methods include the Element-Free Galerkin method in [11], Diffuse Element Method in [14], Partition of Unity Method in [15], Local Petrov–Galerkin Methods in [16], and h–p Cloud Methods in [17]. The aforementioned methods are not truly meshless, however, as all require some type of shadow elements or background integration. A family of meshless methods that rely on collocation of Radial-basis interpolation functions over a set of scattered data has been proposed for the solution of fluid flow and elasticity problems as seen in [18] and [19]. The Localized Radial-basis Function Collocation (LRC) Meshless Method formulated in [20,21,22] uses inverse Multiquadric Radial-basis functions, see [23], as interpolating functions. The method only requires a nodal distribution, and has been shown to be robust for irregular nodal distributions, abating the need for user intervention and making the mesh-generation process entirely automated. This eliminates the main bottleneck in CFD analysis. The LRC is based on a local collocation among radial-basis expansion functions over multiple local domains, which does not require a structured grid. This collocation leads to a linear system of equations that can be solved for the coefficients of the expansion functions, leading to interpolation vectors that can be pre-computed and stored for each node. Any linear differential operator can be applied to the expansion functions, thus differential operators are reduced to vectors and the application of such operators becomes a simple vector-vector operation.

A recent development in the LRC family of meshless methods is the use of radial-basis functions to enhance a traditional finite differencing scheme, see [24,25,26,27]. This method combines the simplicity and efficiency of the Finite Differencing Method with the flexibility and ease of use of the LRC method. Essentially, when the local grid distribution is regular enough to allow for a finite-differencing approximation of derivatives, such an approximation is used. Where the grid is irregular or poorly structured for finite-differencing, virtual nodes are introduced in a regular stencil pattern around the nodal position of focus, and these virtual nodes are subsequently used for virtual RBF-enhanced finite differencing.

2. Methodology

An explicit first-order forward Euler time-stepping scheme is employed for the time integration of the Navier–Stokes equations. Approximation of derivatives is carried out using the Generalized Finite Difference Meshless Method (GFDM). The interface between the lighter and denser fluid is captured using the Level-Set Method, the solution of which is also performed using GFDM. Automatic segmentation of the domain, necessary for parallel execution on the GPU, is performed using recursive bisection.

2.1. Solution of the Navier–Stokes equations

The incompressible Navier–Stokes equations, given by Eq. (1), are fully coupled nonlinear partial differential equations. Here, the only external force we consider is gravity $\vec{g}$:

$$\nabla\cdot\vec{V}=0$$

$$\frac{\partial \vec{V}}{\partial t} = -(\vec{V}\cdot\nabla)\vec{V} - \frac{1}{\rho}\nabla p + \frac{\mu}{\rho}\nabla^2\vec{V} + \vec{g} \tag{1}$$

In two-phase flow a moving boundary between the two phases exists and the flow will be governed by two equations, one for each phase of the flow (along with the continuity equation, which holds over the entire fluid domain):

$$\rho_h\left[\frac{\partial \vec{V}_h}{\partial t} + (\vec{V}_h\cdot\nabla)\vec{V}_h\right] = \nabla p_h + \mu_h\nabla^2\vec{V}_h + \vec{g}, \quad \vec{x}\in \text{denserfluid}$$

$$\rho_l\left[\frac{\partial \vec{V}_l}{\partial t} + (\vec{V}_l\cdot\nabla)\vec{V}_l\right] = \nabla p_l + \mu_l\nabla^2\vec{V}_l + \vec{g}, \vec{x}\in \text{lighterfluid} \tag{2}$$

In Eq. (2), the subscript h indicates the denser fluid, and the subscript l indicates the lighter fluid. An additional boundary condition is needed at the fluid interface, and is given by Eq. (3), where I denotes the fluid interface, $\vec{n}$ denotes a unit vector normal to the interface, σ is the coefficient of surface tension and κ is the curvature of the interface, see Batchelor [28] for details:

$$\begin{aligned} &(\mu_h\nabla^2\vec{V} - \mu_l\nabla^2\vec{V})\cdot\vec{n} = (\rho_h - \rho_l + \sigma\kappa)\vec{n}, && \vec{x}\in I \\ &\vec{V}_h = \vec{V}, && \vec{x}\in I \end{aligned} \tag{3}$$

We can then define

$$\vec{V} = \begin{cases} \vec{V}_h, & \vec{x}\in \text{denserfluid} \\ \vec{V}_l, & \vec{x}\in \text{lighterfluid} \end{cases}$$

$$\mu = \begin{cases} \mu_h, & \vec{x}\in \text{denserfluid} \\ \mu_l, & \vec{x}\in \text{lighterfluid} \end{cases}$$

$$\rho = \begin{cases} \rho_h, & \vec{x}\in \text{denserfluid} \\ \rho_l, & \vec{x}\in \text{lighterfluid} \end{cases} \tag{4}$$

Combining Eqs. (2) and (3)to form an equation which is valid over the entire domain, see Chang et al. [29], we can arrive at

$$\frac{\partial \vec{V}}{\partial t} = -(\vec{V}\cdot\nabla)\vec{V} - \frac{1}{\rho}\nabla p + \frac{\mu}{\rho}\nabla^2\vec{V} + \vec{g} - \sigma\kappa\delta(d)\vec{n} \tag{5}$$

In Eq. (5), $\delta(d)$ is the Dirac delta function and d is the signed distance from the fluid interface. We define the signed distance as the positive distance from the interface when $\vec{x}$ is in the area of the domain occupied by the lighter fluid, and as the negative distance from the interface when $\vec{x}$ is in the area of the domain occupied by the heavier fluid.

Eq. (5) is solved using a fractional step method similar to the method presented by Chorin [30]. Integration in time is carried out using a first-order Euler time stepping scheme. We begin with an initial velocity field that satisfies the continuity equation:

$$\nabla\cdot\vec{V}^0 = 0 \tag{6}$$

We define three time levels, k, $k+1$, and $*$ such that $t^{k+1} = t^k + \Delta t$. An intermediate velocity field $\vec{V}^*$ can be calculated using:

$$\vec{V}^* = \vec{V}^k + \Delta t\left[-(\vec{V}^k\cdot\nabla)\vec{V}^k - \frac{1}{\rho}\nabla p^k + \nu\nabla^2\vec{V}^k + \vec{g}^k - ST^k\right] \tag{7}$$

where ST^k represents the surface tension effects:

$$ST^k = \sigma\kappa^k\delta(d^k)\vec{n}^k \tag{8}$$

Since $\vec{V}^*$ has been calculated without considering the continuity equation, it will not be divergence-free. The field $\vec{V}^*$ can be decomposed into a divergence-free and an irrotational component as a consequence of the Helmholtz-Hodge decomposition theorem. We can define the divergence-free component of $\vec{V}^*$ as $\vec{V}^{k+1}$, resulting in

$$\vec{V}^* = \vec{V}^{k+1} + \nabla\phi \tag{9}$$

where ϕ is a scalar field called the Helmholtz potential. Taking the divergence of Eq. (9) and using the fact that $\nabla\cdot\vec{V}^{k+1} = 0$ results in

$$\nabla^2\phi = \nabla\cdot\vec{V}^* \tag{10}$$

Eq. (10) is a Poisson equation for the Helmholtz potential ϕ, and is solved using the following boundary conditions:

$$\begin{aligned} &\nabla\phi\cdot\vec{n} = 0 && \text{at walls and inlets} \\ &\phi = 0 && \text{at outlets} \end{aligned} \tag{11}$$

Once the Helmholtz potential has been obtained, the velocity field can be updated by rearranging Eq. (9) as

$$\vec{V}^{k+1} = \vec{V}^* - \nabla\phi \tag{12}$$

Boundary conditions for the velocity are applied at the end of each time step before pressure is computed, and are given by the following equations, where $\hat{V}$ is a prescribed velocity:

$$\begin{aligned} &\vec{V} = \hat{V} && \text{at inlets} \\ &\left(\frac{\partial \vec{V}}{\partial t}\right)\cdot\vec{n} = \left[-(\vec{V}\cdot\nabla)\vec{V} - \tfrac{1}{\rho}\nabla p + \nu\nabla^2\vec{V} + \vec{g} - ST\right]\cdot\vec{n} && \text{at outlets} \\ &\vec{V} = 0 && \text{at no slip walls} \end{aligned} \tag{13}$$

We then consider the momentum equation at time level $k+1$ and perform the divergence operation resulting in

$$\nabla\cdot\left(\frac{1}{\rho}\nabla p^{k+1}\right) = -\nabla\cdot(\vec{V}^{k+1}\cdot\nabla)\vec{V}^{k+1} + \nabla\cdot(\nu\nabla^2\vec{V}^{k+1}) - \nabla\cdot ST^{k+1} \tag{14}$$

Eq. (14) is an advection-diffusion equation for the pressure at $k+1$. This is solved using the boundary conditions:

$$\begin{aligned} &\nabla p\cdot\vec{n} = -\rho\left[\frac{\partial \vec{V}}{\partial t} + (\vec{V}\cdot\nabla)\vec{V} - (\nu\nabla^2\vec{V}) - \vec{g}n_y\right]\cdot\vec{n} && \text{at walls and outlets} \\ &p = \hat{p} && \text{at inlets} \end{aligned} \tag{15}$$

which are derived directly from the Navier–Stokes equations and $\hat{p}$ as a prescribed pressure value. Once the pressure field has been obtained, the iteration count is advanced to $k+1$ and the solution proceeds from Eq. (7). Using this solution method, an initial pressure field must be obtained to start the solution. As long as the initial velocity of the fluid is zero throughout the entire domain, the pressure can be initialized to the hydrostatic pressure by solving Eq. (16) with its corresponding boundary conditions as

$$\begin{aligned} &\nabla^2 p^{k+1} - \left(\frac{\nabla\rho}{\rho}\right)\nabla p^{k+1} = 0 \\ &\nabla p\cdot\vec{n} = -\rho g n_y \quad \text{at boundaries} \\ &p = 0 \quad \text{at } y = h_{ref} \end{aligned} \tag{16}$$

Here h_{ref} denotes a prescribed reference height.

2.2. Numerical treatment of the fluid interface

The Level-Set Method (LSM), introduced by Osher and Sethian [31], is used to track the fluid interface. The LSM embeds an interface in a given dimension as a function in the next highest dimension. So a two-dimensional interface, such as the fluid–fluid interface treated in this paper, is embedded as the zero Level-Set of a three-dimensional function. For example, a circle can be represented by the LSM using a cone. The region inside the circle is defined where the Level-Set is negative, the region outside the circle is defined where the Level-Set is positive, and the edge of

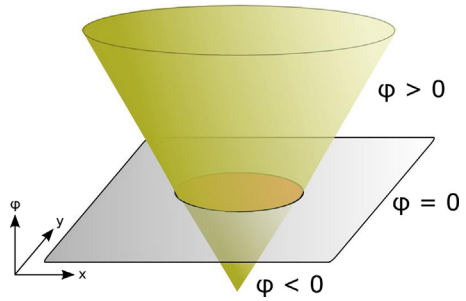

Fig. 1. Level-set representation of a circle.

the circle is defined where the Level-Set is zero, as shown in Fig. 1. The Level-Set function is treated as a scalar property of the fluid, and is advected using the following equation:

$$\frac{\partial\varphi}{\partial t}+(\vec{V}\cdot\nabla)\varphi=0 \tag{17}$$

Eq. (17) is advanced in time using the same first-order Euler method used in the solution of the Navier–Stokes equations, and so the LSM is straightforward to implement in conjunction with the fluid solver. Following the formulation by Sussman et al. [32], we can define the density and viscosity at any point in the domain as

$$\begin{aligned}\rho(\varphi)&=H(\varphi)(\rho_l-\rho_h)+\rho_h\\ \mu(\varphi)&=H(\varphi)(\mu_l-\mu_h)+\mu_h\end{aligned} \tag{18}$$

where $H(\varphi)$ is the Heaviside function. The use of the Heaviside function leads to a sharp discontinuity at the fluid interface that is difficult to resolve numerically. In order to provide an easier numerical treatment of the interface, the Heaviside function is replaced by the smoothed, or mollified, Heaviside function that smears the interface over a predefined interface thickness:

$$H_s(\varphi)=\begin{cases}1, & \varphi>\varepsilon\\ \frac{1}{2}\left[1+\frac{\varphi}{\varepsilon}+\frac{1}{\pi}\sin\left(\frac{\pi\varphi}{\varepsilon}\right)\right], & |\varphi|\le\varepsilon\\ 0, & \varphi<-\varepsilon\end{cases} \tag{19}$$

The Heaviside function H in Eq. (18) is replaced with Hs, giving the interface an approximate thickness i, which is controlled using the parameter ε as

$$i=\frac{\varepsilon}{|\nabla\varphi|} \tag{20}$$

Additionally, the Dirac delta function in Eq. (5) is replaced with the mollified delta function, which is defined as the derivative of the mollified Heaviside function with respect to φ:

$$\delta_s(\varphi)=\begin{cases}\frac{1}{2\varepsilon}\cos\left(\frac{\pi\varphi}{\varepsilon}\right) & |\varphi|\le\varepsilon\\ 0 & \text{otherwise}\end{cases} \tag{21}$$

The curvature and unit normal vector of the interface are also required in order to solve Eq. (5). If the Level-Set function φ is a signed distance function, several useful properties arise which ease the computation of the interface curvature and unit normal vector, which can be found using Eq. (22), see Osher and Fedkiw [33]:

$$\begin{aligned}\kappa&=\nabla\cdot\frac{\nabla\varphi}{|\nabla\varphi|}\\ \vec{n}&=\frac{\nabla\varphi}{|\nabla\varphi|}\end{aligned} \tag{22}$$

In order for the curvature and normal equations to remain valid, the Level-Set function must be periodically reinitialized to a signed distance function. This is accomplished using the reinitialization relation, see [34], given by the following equation:

$$\frac{\partial\varphi}{\partial t}=sign(\varphi)(1-|\nabla\varphi|) \tag{23}$$

The reinitialization equation is solved to steady state. Generally, only a few iterations using a first-order Euler's method are necessary as φ is already close to a signed distance function.

2.3. Localized radial-basis function interpolation

In order to discretize Eq. (5), a generalized finite-differencing scheme using radial-basis function (RBF) interpolation of field variables at virtual nodes is used, following the method presented by Gerace et al. [24]. A radial-basis function is a scalar function whose value at a point depends on the distance from the point to a predefined data center. The RBF used in the solver that is presented here follows the form of the Hardy Multiquadrics as

$$\chi(\vec{x})=[r_j^2(\vec{x})+c^2]^{n-(3/2)} \tag{24}$$

where n is a positive integer, c is a predefined shape parameter, and r_j is the Euclidean distance between point $\vec{x}$ and some predefined point $\vec{x}_j$. In this paper we choose $n=1$, leading to the so-called inverse-Multiquadrics RBF:

$$\chi(\vec{x})=\frac{1}{\sqrt{r_j^2(\vec{x})+c^2}} \tag{25}$$

The value of the shape parameter c can be estimated to optimize the interpolation as shown in [24]. The RBF can be used as a local interpolating function given the values of a field variable at several locations. Consider a general field variable given by $f(\vec{x})$, of which several values f_i are known at points $\vec{x}_i$. Let χ_i be an inverse-Multiquadrics RBF whose data center is the point $\vec{x}_i$. The value of the function can be approximated at any point $\vec{x}$ using the known values of the function by performing a local RBF expansion according to

$$f(\vec{x})=\sum_{i=1}^{NF}\alpha_i\chi_i(\vec{x}) \tag{26}$$

where the coefficients α_i are scalar expansion coefficients while NF is the number of influence points. Consider the nodal distribution shown in Fig. 2. The influence topology for each node is chosen by selecting the neighboring influence nodes that fit into an expanding circle with center at the node using the following criteria: the topology circle has to include a minimum of 8 ($NF=9$ including the center node) influence points and points in every direction must be included as long as a boundary point is not detected in the topology, in which case the boundary acts as a limit in that particular direction. The expansion coefficients for the data center may be determined by collocation of the RBF at each point $\vec{x}_i$ in the influence topology. Applying Eq. (26) to all points $\vec{x}_i$ in the influence topology a system of linear equations for the expansion

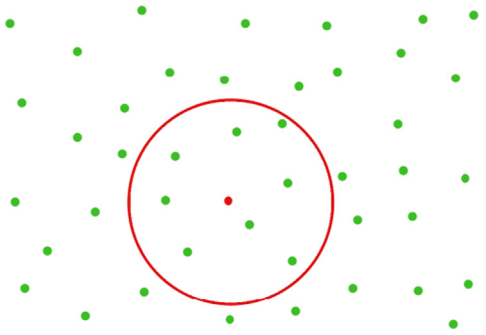

Fig. 2. Local RBF influence topology.

coefficients α_i is generated as shown in the following equation:

$$\begin{bmatrix} \chi_1(\vec{x}_1) & \chi_2(\vec{x}_1) & \cdots & \chi_N(\vec{x}_1) \\ \chi_1(\vec{x}_2) & \chi_2(\vec{x}_2) & \cdots & \chi_N(\vec{x}_2) \\ \vdots & \vdots & \ddots & \vdots \\ \chi_1(\vec{x}_N) & \chi_2(\vec{x}_N) & \cdots & \chi_N(\vec{x}_N) \end{bmatrix} \begin{Bmatrix} \alpha_1 \\ \alpha_2 \\ \vdots \\ \alpha_N \end{Bmatrix} = \begin{Bmatrix} f_1 \\ f_2 \\ \vdots \\ f_N \end{Bmatrix} \tag{27}$$

or

$$[\chi]\{\alpha\} = \{f\} \tag{28}$$

This system is solved for the expansion coefficients as

$$\{\alpha\} = [\chi]^{-1}\{f\} \tag{29}$$

Eq. (26) can be recast in vector form for the interpolation of the function $f(\vec{x})$ at any point within the influence topology such as the data center $\vec{x}_k$ leading to f_k as

$$f_k = \{\chi_k\}^T\{\alpha\} \tag{30}$$

where

$$\{\chi_k\} = \begin{Bmatrix} \chi_1(\vec{x}_k) \\ \chi_2(\vec{x}_k) \\ \vdots \\ \chi_N(\vec{x}_k) \end{Bmatrix}, \quad \{\alpha\} = \begin{Bmatrix} \alpha_1 \\ \alpha_2 \\ \vdots \\ \alpha_N \end{Bmatrix} \tag{31}$$

Substituting Eq. (29) into Eq. (30) yields

$$f_k = \{\chi_k\}^T[\chi]^{-1}\{f\} \tag{32}$$

or

$$f_k = \{\zeta_k\}^T\{f\} \tag{33}$$

where the interpolation vector $\{\zeta_k\}$ can be pre-computed as

$$\{\zeta_k\}^T = \{\chi_k\}^T[\chi]^{-1} \tag{34}$$

Notice that the interpolation of the function value at the data center f_k is accomplished through the simple vector–vector inner product in Eq. (33) where the interpolation vectors $\{\zeta_k\}$ can be pre-computed and stored for every data center prior to the solution process as they are only dependent on geometry and the parameters of the RBF as shown in Eq. (34).

2.4. Generalized finite differencing

Consider the nodal distribution shown in Fig. 3(a). A field variable $f(\vec{x})$ is known at each of the nodes and the derivative of $f(\vec{x})$ with respect to the x- and y-coordinate directions at the data center $\vec{x}_k$ is desired. Let us introduce several virtual nodes, denoted by *'s in Fig. 3(b). The virtual nodes are spaced Δx distance apart in the x-direction, and Δy distance apart in the y-direction about the data center $\vec{x}_k$. Applying standard second-order central finite-differencing to approximate the derivatives of $f(\vec{x})$ with respect to x and y using the virtual nodes, we have

$$\frac{\partial f}{\partial x} = \frac{f(x_{k+1}) - f(x_{k-1})}{2\Delta x}, \quad \frac{\partial f}{\partial y} = \frac{f(y_{k+1}) - f(y_{k-1})}{2\Delta y} \tag{35}$$

Any finite-difference stencil, such as forward or backward differencing, can be used. Let the values of the field variable at any general virtual points separated by any spacing vector $\vec{d}$ be

$$\begin{aligned} f_{k-2} &= f(\vec{x} - 2\vec{d}), f_{k-1} = f(\vec{x} - \vec{d}), \ldots f_{k+1} \\ &= f(\vec{x} + \vec{d}), f_{k+2} = f(\vec{x} + 2\vec{d}) \end{aligned} \tag{36}$$

We can apply an RBF expansion at each of the virtual nodes using Eq. (33), selecting other points in the vicinity of the data center to act as influence points to aid in the interpolation of the field variable at the virtual nodes. Note that all virtual nodes for a given data center use the same influence topology. Performing collocation with all of the influence points we can obtain interpolation vectors for each of the virtual nodes. The value of the field variable at each of the virtual nodes is then given by

$$f_v = \{\zeta_v\}^T\{f\}, \quad v = k-2, \quad k-1, \ldots, k+2 \tag{37}$$

where $\{f\}$ is a vector composed of all the known values of the field variable within the influence topology. Substituting Eq. (37) into the standard second-order finite-difference formula, we obtain an expression for the derivative of f with respect to any direction $s = \vec{d}/\Delta s$, where $\Delta s = |\vec{d}|$ as

$$\begin{aligned} \left.\frac{\partial f}{\partial s}\right|_k &= \frac{f_{k+1} - f_{k-1}}{2\Delta s} \\ \left.\frac{\partial f}{\partial s}\right|_k &= \frac{\{\zeta_{k+1}\}^T\{f\} - \{\zeta_{k-1}\}^T\{f\}}{2\Delta s} \\ \left.\frac{\partial f}{\partial s}\right|_k &= \left[\frac{\{\zeta_{k+1}\} - \{\zeta_{k-1}\}}{2\Delta s}\right]^T\{f\} \\ \left.\frac{\partial f}{\partial s}\right|_k &= \{\delta s_k\}^T\{f\} \end{aligned} \tag{38}$$

where $\{\delta s_k\}$ is a vector representing the first derivative operator with respect to the direction s, and can be pre-computed as

$$\{\delta s_k\} = \frac{\{\zeta_{k+1}\} - \{\zeta_{k-1}\}}{2\Delta s} \tag{39}$$

Vectors similar to that in Eq. (39) can be derived, pre-computed, and stored for any finite-difference stencil of any order in any direction.

2.5. Upwinding

Accurate and stable evaluation of the convective derivatives in the momentum equation, the advection–diffusion equation for pressure, and the Level-Set advection equation requires the use of

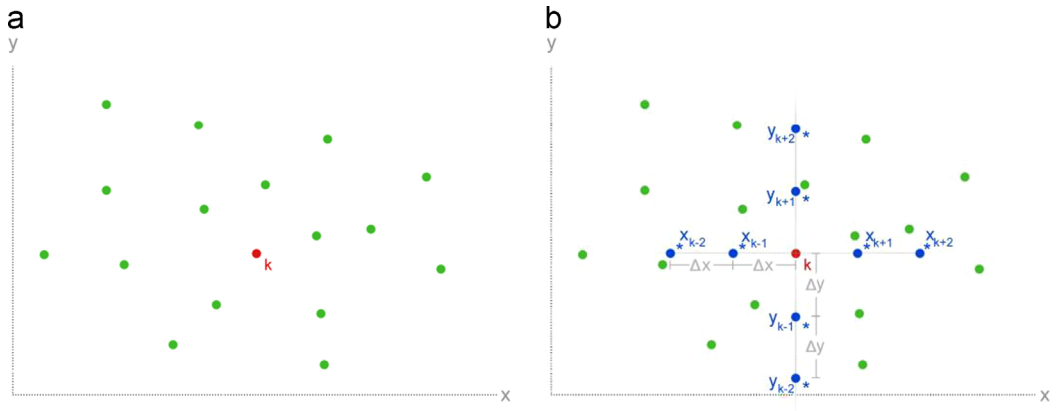

Fig. 3. (a) Irregular point distribution and (b) virtual node distribution.

an upwinding scheme. For a general field variable $f(\vec{x})$ being advected by a general velocity field $\vec{V}$, the derivative operators used to calculate the spatial derivatives of $f(\vec{x})$ in each of the coordinate directions can be upwinded using the components of $\vec{V}$ in those coordinate directions. In the solver presented here, several spatial derivative operators in each coordinate direction are pre-computed and stored for each data center using forward, backward, and central differencing. For example, for a data center $\vec{x}_k$, three x-derivative operators are pre-computed using the virtual finite-differencing method:

$$\left\{\delta x^{for}{}_k\right\} = \frac{\{\zeta_{k+1}\} - \{\zeta_k\}}{\Delta x}$$
$$\left\{\delta x^{cen}{}_k\right\} = \frac{\{\zeta_{k+1}\} - \{\zeta_{k-1}\}}{2\Delta x}$$
$$\left\{\delta x^{bac}{}_k\right\} = \frac{\{\zeta_k\} - \{\zeta_{k-1}\}}{\Delta x} \tag{40}$$

Three y-derivative operators are computed in a similar fashion. Although the operators shown here are first-order, any order of finite-differencing can be used. When evaluating the derivative of

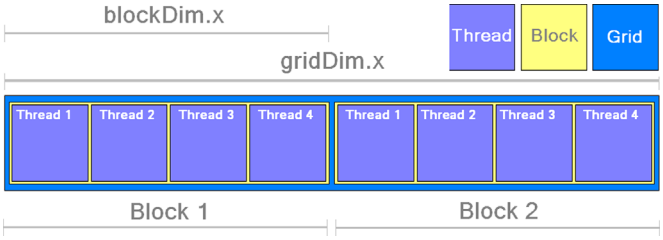

Fig. 4. Layout of CUDA threads, blocks and grid.

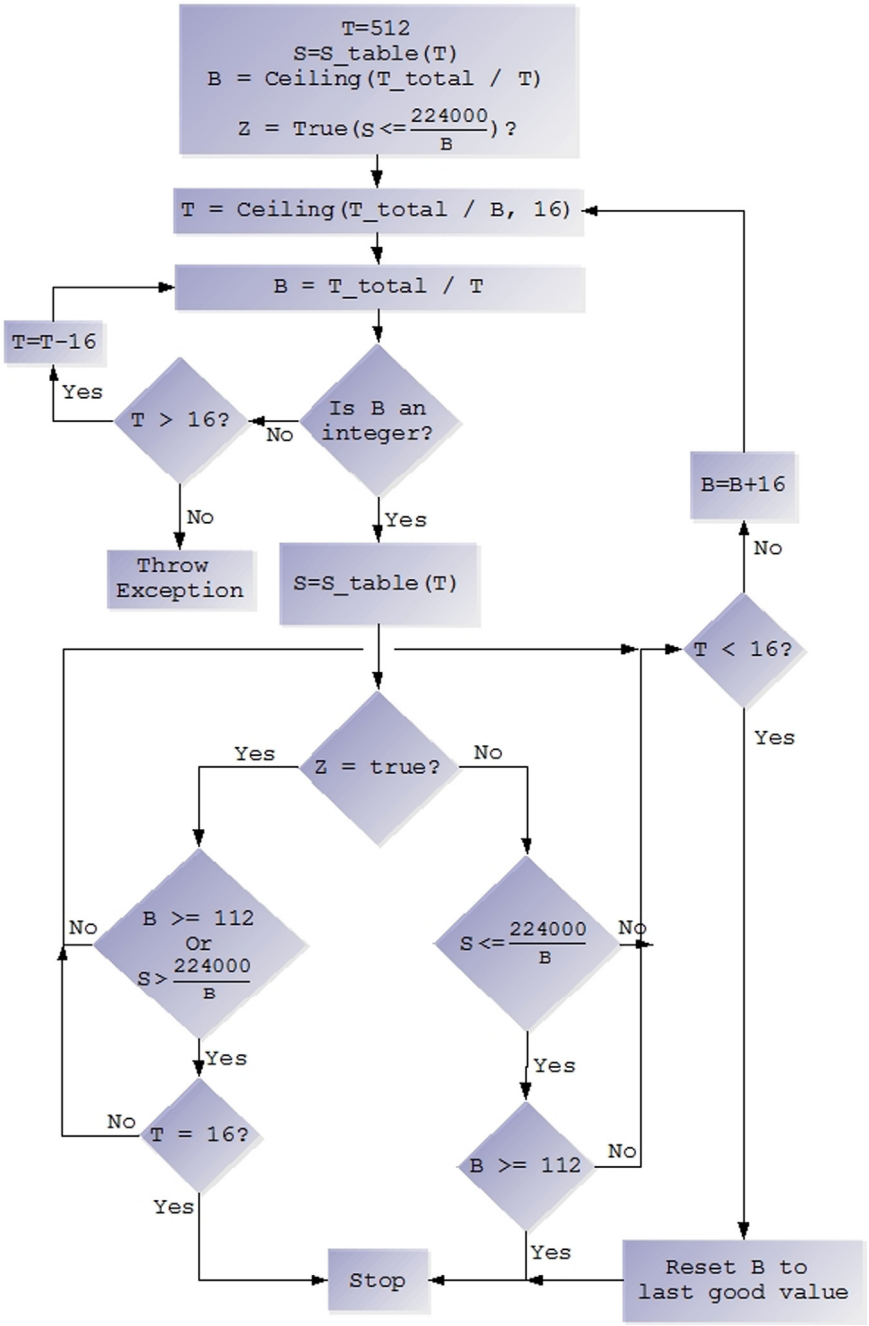

Fig. 5. Flow chart for execution configuration optimization algorithm.

a field variable being transported by some velocity $\vec{V}$, the derivative operator used is determined using the Courant–Friedrichs–Lewy condition as follows:

$$\{\delta x_k\} = \begin{cases} \{\delta x^{for}{}_k\} & \text{if } V_x \le -\frac{\Delta x}{\Delta t} \\ \{\delta x^{cen}{}_k\} & \text{if } -\frac{\Delta x}{\Delta t} < V_x < \frac{\Delta x}{\Delta t} \\ \{\delta x^{bac}{}_k\} & \text{if } V_x \ge \frac{\Delta x}{\Delta t} \end{cases} \tag{41}$$

Upwinding in the y-direction is performed analogously. Upwinding is also used in the iterative solution of the advection–diffusion equations for pressure, where the advection velocity used is given by $(\nabla\rho/\rho)$.

2.6. Parallelization

Parallelization of the solution process for implementation on the GPU requires the segmentation of the domain into groups of nodes, or segments. The segmentation of the nodes is not a trivial process. In order for the solution to operate efficiently in parallel, each segment should contain nodes that are within close physical proximity to one another, ensuring memory and communication requirements are kept to a minimum for each segment. Additionally, efficient load balancing will occur when each segment contains approximately the same number of nodes. Communication between segments occurs along the boundaries of the segments, and so the ratio of boundary to interior points should be minimized for each segment. The GPU is a Single-Instruction Multiple-Data (SIMD) device, meaning it executes the same or near-same code in each thread, but on different pieces of data. CUDA organizes thread execution on three levels: threads, blocks, and grids. A block is a collection of threads, and threads within a block may be indexed with up to 3 dimensions. A grid is a collection of blocks, and may also be indexed with up to 3 dimensions. The graphics hardware organizes threads executing in a block into "warps". Each warp is made up of 16 threads. The memory access architecture on the GPU is designed to give optimal memory bandwidth for sequential memory reads of 32-bit words by each thread in a warp. The GPU will be most efficient if all warps are fully populated, meaning that each block size is a multiple of 16. The GPU is generally expected to show a better performance gain over the CPU when all of its multiprocessors are occupied, so a larger percent increase in speed should be expected for larger computational grid sizes.

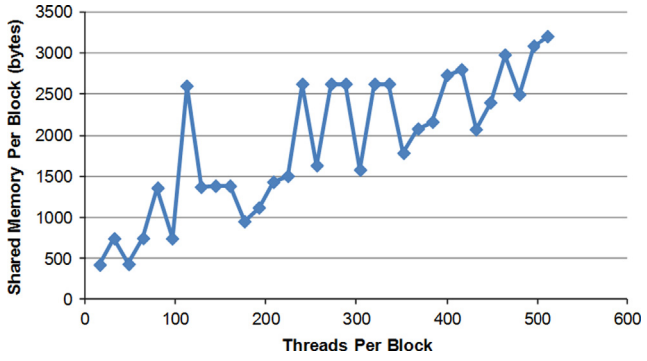

Fig. 6. Shared memory requirements.

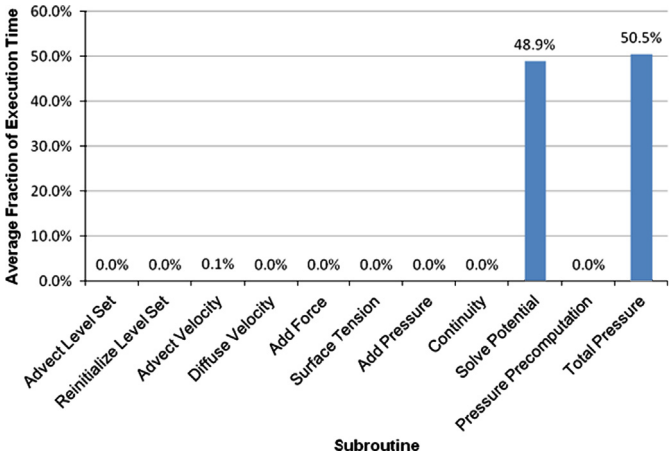

Fig. 7. Average fraction of execution time for each subroutine.

Segmentation is accomplished using a recursive bisection algorithm, see [35]. Each segment produced is directly mapped to a block on the GPU, with each thread on the GPU handling one node in the solution domain, so each segment should contain some number of points that is a multiple of 16. The algorithm begins by dividing the domain into two segments using a heuristic to determine the optimal layout of the segments. Each of these segments is then recursively split using the same heuristic until the desired number of segments has been generated.

In order to execute a kernel on the GPU, an execution configuration is required at run-time. An execution configuration is an abstraction of the way the execution of a kernel is mapped to the actual graphics hardware. The execution configuration includes the block dimensions and the grid dimensions, which together specify the total number of threads executed. The execution configuration is limited by the number of registers and shared memory per multiprocessor on the GPU. This paper used the NVIDIA GeForce 9800 GT, which has 8192 registers and 16 KB of shared memory per multiprocessor. For a given kernel, denote the required number of registers per block as R and the required amount of shared memory as S for a given execution configuration using B blocks on M multiprocessors (Fig. 4). A necessary condition for execution of the kernel is then given by

$$\left(\frac{S\cdot B}{M} \le S_{avail}\right) \wedge \left(\frac{R\cdot B}{M} \le R_{avail}\right) \tag{42}$$

Table 1
Summary of GPU data transfer and storage modes.

Data	GPU data structure	Transfer
Right-hand side vector	1-D texture	Every time step (write)
Boundary conditions	1-D texture	Every time step (write)
Advection-diffusion operators	2-D texture	Every time step (write)
Normal derivative operators	2-D texture	Once (write)
Laplacian operators	2-D texture	Once (write)
Map of local to global indices	2-D texture	Once (write)
Local influence topologies	2-D texture	Once (write)
Node sharing ID	2-D texture	Once (write)
Boundary node type ID	1-D texture	Once (write)
Unknown vector	Array in global memory	Every time step (write/read)

The GeForce 9800 GT has 14 multiprocessors, so $M=14$, yielding

$$\left(S \le \frac{224}{B}\mathrm{KB}\right) \wedge \left(R \le \frac{114688}{B}\right) \tag{43}$$

Each multiprocessor on the GPU hides memory latency by rotating the blocks that are being actively executed, running blocks whose memory accesses have been served while those that have incomplete memory operations wait to become active. Thus, it is good to have at least twice as many blocks as there are multiprocessors available on the device in order to mask memory access overhead. The maximum number of threads per block is 512 and the maximum number of active blocks per multiprocessor is 8, and so if full occupancy of the GPU is desired the optimal execution configuration is 112 blocks per grid and 96 threads per block. Typically, however, the number of threads per grid will be prescribed and the optimal execution configuration can only be approximately satisfied in terms of either the number of blocks B or the number of threads per block T. Consider the case where the total number of threads is specified as T_{total} and the number of blocks is allowed to vary. T should be a multiple of 16 for optimal memory performance on the GPU, so B should also be a multiple of 16. Additionally, the amount of shared memory allocated per block must satisfy Eq. (42). An algorithm for selecting the optimal execution configuration for a given number of threads per grid was devised, and is shown in Fig. 5.

The amount of shared memory per block S depends largely on the global data center distribution. In order to provide an estimate for S as a function of T, S was measured for several different selections of T for a regular data center distribution of 130×130 equally spaced data centers. The results are shown in Fig. 6. The number of threads per block was varied from 16 to 512 in multiples of 16.

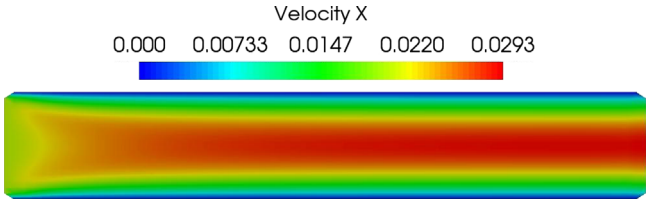

Fig. 10. Velocity contours of flow between parallel plates.

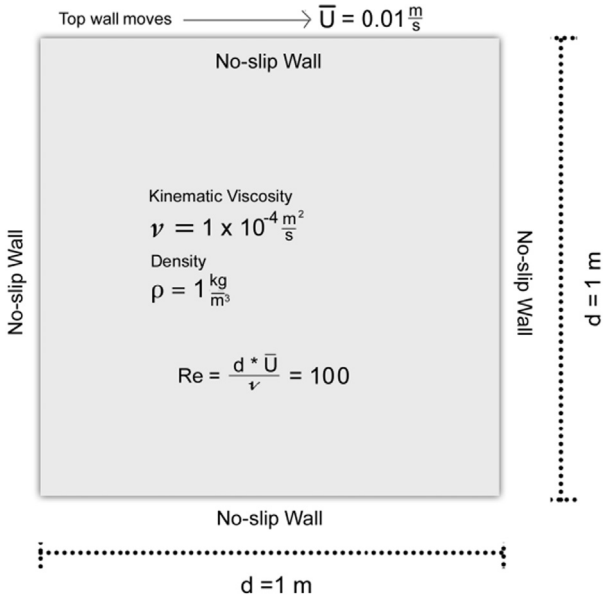

Fig. 11. Setup of lid-driven cavity flow problem.

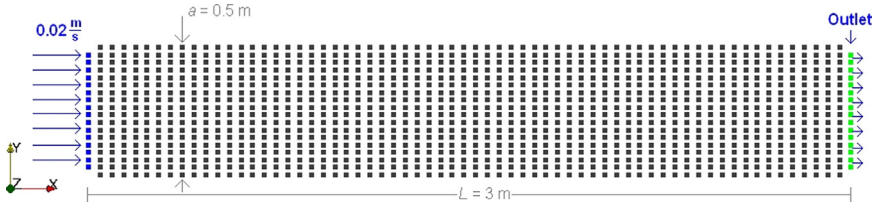

Fig. 8. Point distribution for flow between two infinite parallel plates.

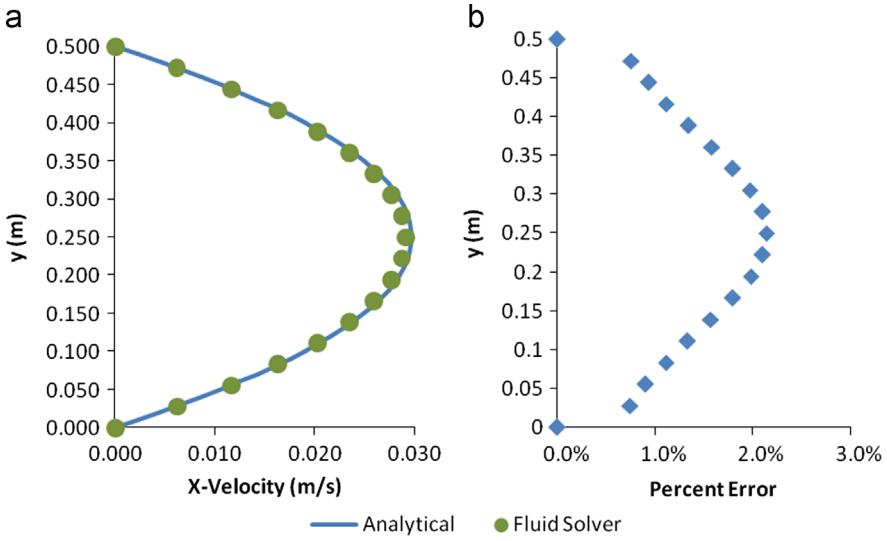

Fig. 9. Velocity profile of fully developed flow between parallel plates: (a) Analytical vs. Meshless solutions and (b) Percent error.

2.7. Graphics processing unit acceleration

Two versions of the fluid solver were programmed, one that was entirely serial and executed solely on the CPU, and one that was GPU-accelerated that had portions of the code ported to the GPU. In order to determine which routines in the solver should be ported to the GPU, the serial code was analyzed for bottlenecks. To identify bottlenecks in the serial code, two problems were run on four grid sizes and the execution times of each of the primary subroutines in the solver were recorded. Each solution was run for 100 time steps, and the execution times were recorded every 10 time steps. The average execution times across all of the problems run were then calculated for each of the primary subroutines. These average times were normalized according to the average time step for all of the problems, and the results are shown in Fig. 7. The solution of the Poisson equation for the Helmholtz potential and the solution of the advection–diffusion equation for the pressure were found to take up 99% of the total execution time of each time step of the solver, and so these subroutines were targeted for GPU-acceleration.

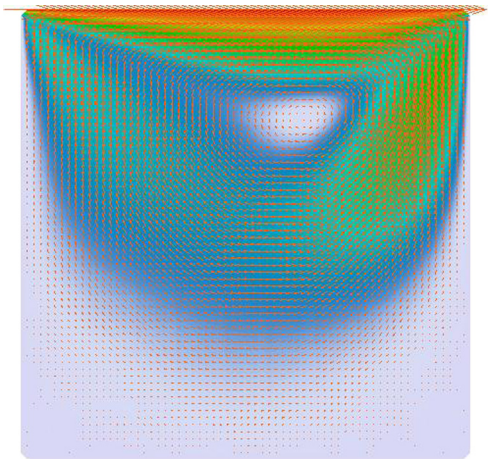

Fig. 12. Velocity contours and vectors for lid-driven cavity flow problem.

Since each thread block on the GPU references threads according to a local index, mapping is required to map thread indices within each block to the global indices of the data centers. Additionally, the influence topologies that are transferred to the GPU are mapped to local block indices so each block can make efficient use of shared memory rather than using global indices to reference global memory. Once segments have been constructed, the mapping process detailed below is applied to each segment. First, the influence topologies of each of the nodes in a segment are compared. A unique list of indices that the nodes within the segment will reference is constructed. For example, if multiple nodes in a segment both reference node *n*, only one entry in the unique list exists for node n. In pseudo-code, this process looks like:

```
AssembleUniqueList(Segment s)
{       s.UniqueList=empty
        FOR EACH node n IN s
                FOR EACH node f IN n.InfluencePoints
                        IF s.UniqueList.DoesNotContain(f) THEN s.
  UniqueList.Add(f)
                END FOR
    END FOR       }
```

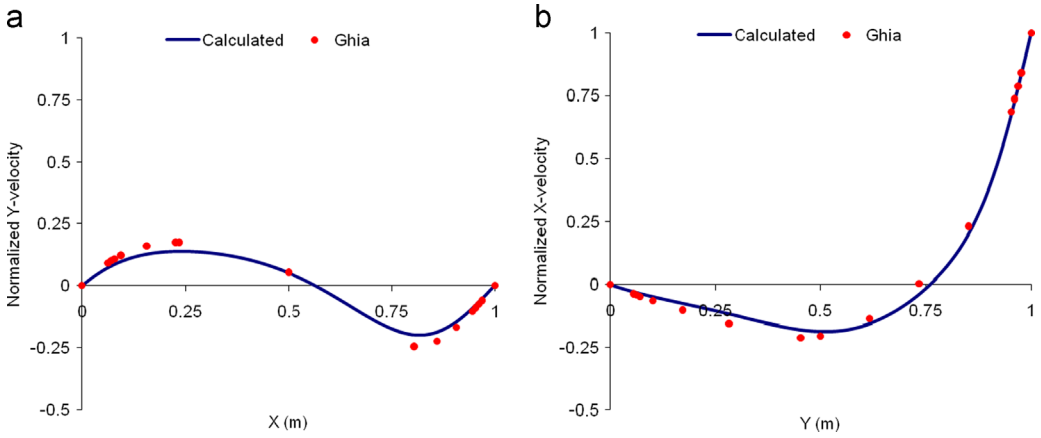

Fig. 13. Lid-driven cavity flow velocity profiles: (a) along horizontal center line and (b) along vertical center line.

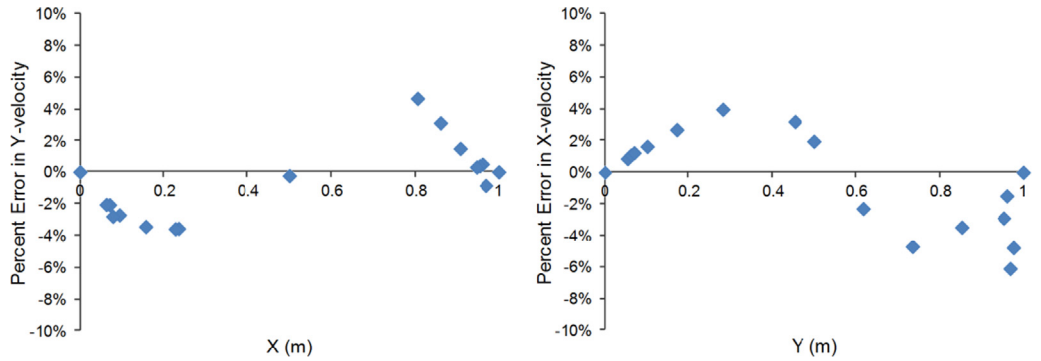

Fig. 14. Percent difference of computed solution relative to Ghia et al. [37], normalized by velocity span.

Once the unique list has been assembled for each segment, a new influence topology is generated for each point within the segment that maps the point's global influence topology to local segment indices. In pseudo-code, this process looks like:

```
AssembleLocalTopologies(Segment s)
{       FOR EACH node n IN s n.LocalInfluencePoints=empty
            FOR f=0 TO n.InfluencePoints
                G_idx=n.InfluencePoints(f)
                FOR u=0 TO s.UniqueList.Size
                    IF G_idx == s.UniqueList(u) THEN
  n.LocalInfluencePoints(f)=u
                END FOR
        END FOR
    END FOR       }
```

On the GPU, each thread uses its local influence point array to perform differential operations using the shared memory of the block. For the solution of the Poisson and advection–diffusion equations, all data such as differential operators and boundary node identifiers are transferred to the GPU before iteration begins. At each time step, the right-hand sides of the equations and the boundary conditions are transferred to texture memory on the GPU, and the global unknown vectors are transferred to and from global memory on the GPU. Each thread block loads the nodal values it requires from global to shared memory, performs operations using the shared data, and then writes all values required by other thread blocks to global memory. Table 1 provides a summary of GPU data transfer and storage modes.

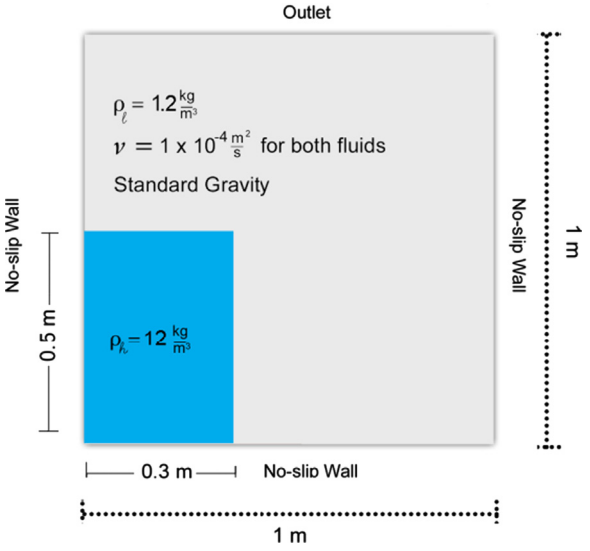

Fig. 15. Dam break problem setup.

3. Results

This section will present the results obtained using the fluid solver. First, the accuracy of the solver will be verified, and then the execution times of the GPU-accelerated solver will be compared to that of the serial solver.

3.1. Verification of accuracy

The first verification problem that will be presented is the solution of flow between infinite parallel plates, for which an analytical solution is available against which the numerical results may be compared. The problem setup is shown in Fig. 8, and consists of two plates, each $L=3$ m long, spaced $a=0.5$ m apart. A Meshless nodal distribution of 18×66 nodes was used in the solution of the flow. An inlet was placed at the left side of the domain that had a prescribed pressure of zero and a prescribed velocity of 0.02 m/s. The no-slip condition was applied at the walls. The density of the fluid was 1.0 kg/m^3, and its viscosity was 10^{-4} N s/m^2. An outlet was placed at the rightmost side of the domain.

Using the spacing between the plates as the characteristic length, and using the mean velocity near the inlet as the characteristic velocity, the Reynolds number of the flow is given by

$$\mathrm{Re} = \frac{\rho \overline{V} L}{\mu} = 100 \tag{44}$$

An analytical expression for the x-velocity of the fluid in the fully developed region as a function of the y-coordinate is given below:

$$u(y) = -4u_{\max}\left[\left(\frac{y}{a}\right)^2 - \frac{y}{a}\right] \tag{45}$$

The x-position at which the flow may be considered fully developed is given in Eq. (46), see Fox et al. [36]:

$$x_{fd} = 0.05a \cdot \mathrm{Re} = 2.475 \tag{46}$$

The problem was evolved in time until steady-state was reached. The x-velocity was sampled along a vertical line at a point after which the flow was fully developed, and the computed values are shown along with the exact result in Fig. 9(a). The percent error for the calculated solution is shown as a function of y in Fig. 9(b), and illustrates that the solver was able to represent the exact solution to within an average of 1.3% accuracy. Fig. 10 shows the computed velocity contours.

A lid-driven cavity flow problem was also used to verify the accuracy of the solver. The problem setup is shown in Fig. 11.

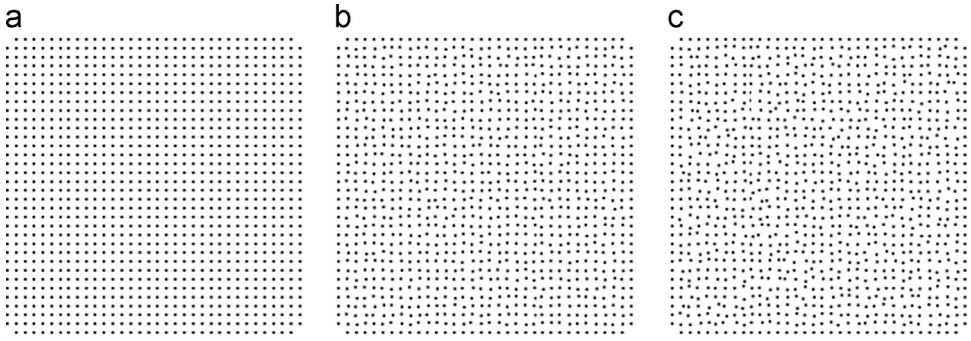

Fig. 16. Point distribution: (a) evenly spaced, (b) once perturbed, and (c) twice perturbed.

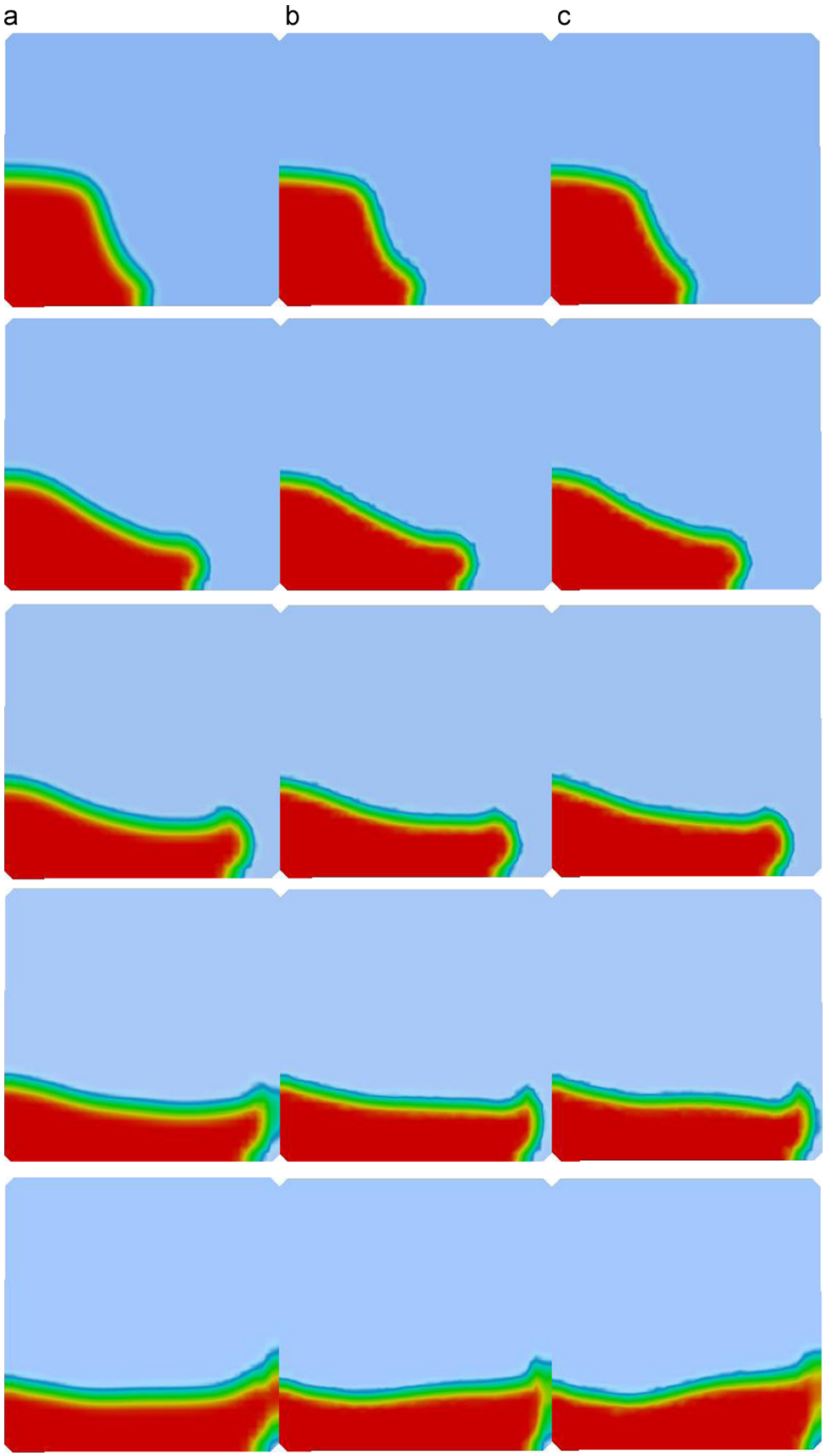

Fig. 17. Dam break problem evolution after $t=0.1783$ s, 0.2960 s, 0.4047 s, 0.4859 s, and 0.5821 s: (a) evenly spaced points, (b) once perturbed, and (c) twice perturbed.

The cavity flow is a single-phase steady-state problem in which all boundaries on a square domain are no-slip walls and the topmost wall is moving to the right with a constant velocity. The density of the fluid was set to 1 kg/m^3 and its kinematic viscosity was set to 10^{-4} m^2/s. The Reynolds number for the problem based on the length of one of the sides of the domain was 100. Fig. 12 shows the resulting steady-state velocity contours and vectors, revealing the large-scale circulatory behavior of the fluid. The normalized x-velocity along the vertical geometric centerline of the domain and the normalized y-velocity along the horizontal geometric centerline of the domain showed good agreement with the results of Ghia et al. [37], which are considered canonical for the lid-driven cavity problem. The velocities were normalized to the velocity of the top wall. Fig. 13(a) and (b) plots the normalized velocities along the horizontal and vertical centerlines, respectively. Fig. 14 (a) and (b) shows the percent difference of the computed velocity along the horizontal and vertical centerlines, respectively, normalized by velocity span, relative to the results presented in Ghia et al. [37]. These results show relatively accurate comparisons limited to a single value of Reynolds number (Re=100) and a single point distribution. The central mission of this paper is to present a framework for the extension of the LRC meshless method to two-phase flows using the Level-Set method and implementing it in massively parallel platforms such as the GPGPU. Extensive verification of the accuracy of the LRC Meshless method using classical benchmark problems such as the lid-driven cavity at different Reynolds numbers and point resolutions can be found in the literature, see [20,25,26].

Table 2
Integrated density at multiple time levels for Dam break problem.

Time (s)	Regular grid	1st level perturbed	2nd level perturbed
0.0000	3.12097	3.11035	3.11344
0.1783	3.20369	3.17817	3.22279
0.2960	3.32571	3.29973	3.34034
0.4047	3.40764	3.39243	3.43980
0.4859	3.43341	3.42580	3.47261
0.5821	3.48225	3.47872	3.59681

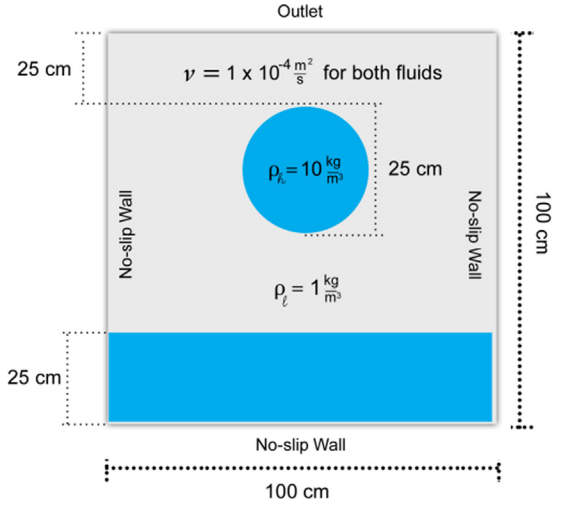

Fig. 18. Droplet problem setup.

Comparison of two-phase results to existing numerical or experimental results was difficult. Numerical solution of two-phase flow is still a problem under much development in CFD, and so canonical data is difficult to acquire. Additionally, experimental data for problems such as the dam break or droplet problems are lacking. For this reason, experimental verification of two-phase accuracy is omitted here, although future work will include efforts to demonstrate agreement of two-phase results with established data.

To demonstrate the robustness of the solver in two-phase flow a dam break problem setup as shown in Fig. 15 was run using three nodal distributions: Evenly spaced, perturbed, and perturbed twice as seen in Fig. 16. At $t=0$, the fluid is allowed to collapse under gravity.

Fig. 17 shows the evolution of the dam break flow on these three nodal distributions through time. They show that solutions to the dam break problem on successively disordered nodal distributions agree very well, demonstrating that the solver is robust regarding both the discontinuity between the interfaces as well as with the ability to handle irregular nodal distributions. In order to determine how well mass was conserved in each of the

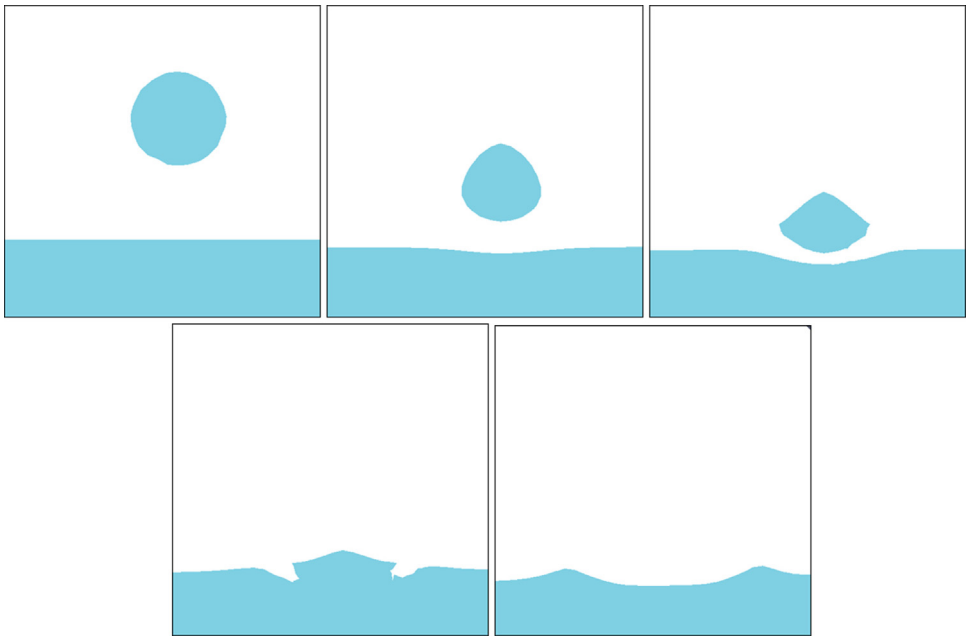

Fig. 19. Evolution of the droplet problem.

three cases, the integral of density was performed numerically at each time step shown. The results are shown in Table 2, and demonstrate good agreement between the regular and perturbed grids at each time step.

A droplet problem in which a droplet of a dense fluid was allowed to fall under the force of gravity through a litter fluid into a pool of the dense fluid was run using a 34 × 34 node Meshless point distribution. The problem setup is shown in Fig. 18. Fig. 19 shows the evolution of the droplet flow. While the droplet remains relatively symmetric, there are instabilities present as it nears the surface of the pool of the denser fluid.

3.2. Benchmarking of GPU-accelerated routines

Benchmarking of the two GPU-accelerated routines was performed using four different grid sizes for two different problems, one single-phase and one two-phase. The four grid sizes used were 18×18, 34×34, 66×66, and 130×130. Fig. 20 shows the speedup factors for each of these grid sizes and problems. As can be seen, the acceleration obtained is independent of the problem being solved, and strongly depends on grid size. It was observed that the acceleration factor increases considerably as the grid size increases. Fig. 21 shows the execution time per time step for the serial code and the GPU-accelerated code as a function of grid size. It is interesting to note that the execution time for the GPU-accelerated code scales fairly linearly with grid size, while execution time for the serial code appears to scale quadratically with grid size. This is a result of the architecture of the GPU. As more threads are executed on the device, its efficiency increases due to the higher occupancy per multiprocessor than for lower numbers of threads.

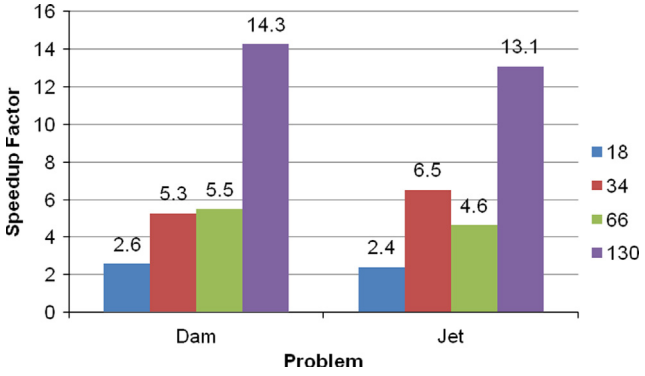

Fig. 20. Speedup factors for one and two-phase problems on several grid sizes.

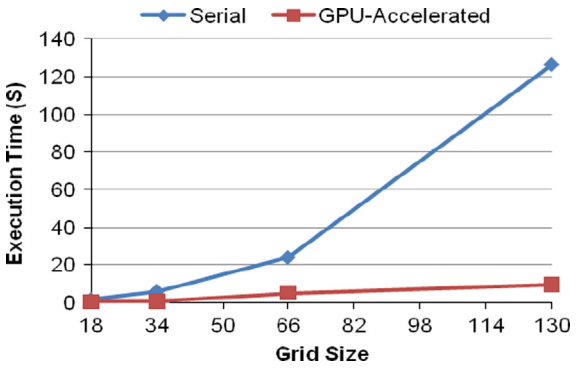

Fig. 21. Average execution times per time step.

Concerning accuracy and robustness, the GPU-accelerated code performs identically to the serial code. Many of the examples presented in the previous sections concerning accuracy and robustness were computed with the GPU-accelerated code. Since all of the data is identical for the two codes, it would be trivial to display any comparative results, as they may already be found throughout the solutions previously discussed.

The data discussed above concerns the acceleration of the entire time step. Looking at individual subroutines rather than the time step as a whole, the largest speedup factor observed was for the solution of the Helmholtz potential in a dam break problem on a 130×130 grid, as shown in Fig. 22(a) and (b). The speedup factor in this case was found to be 16.07. Table 3 compares the maximum speedup factor observed in this paper to the other contemporary GPGPU projects in CFD that were discussed in Section 1.1. The speedup factor achieved in this paper is fairly consistent with other contemporary speedup factors.

4. Conclusions

This paper presents the development and implementation of a meshless two-phase incompressible fluid flow solver and its acceleration using the graphics processing unit (GPU). The solver is formulated as a Localized Radial-basis Function Collocation (LRC) meshless method and the interface of the two-phase flow is captured using an implementation of the Level-Set method. The Compute Unified Device Architecture (CUDA) language for general-purpose computing on the GPU is used to accelerate the solver. Through the combined use of the LRC meshless method and GPU acceleration this paper seeks to address the issue of robustness and speed in computational fluid dynamics. Traditional

Table 3
Speedup factor comparison.

Author	Method	Speedup factor
Brandvik and Pullan [10]	FVM	16
Li et al. [6]	Lattice-Boltzmann	15
This paper	Meshless	14.3
Thibault et al. [8]	FDM	13
Riegel et al. [7]	Lattice-Boltzmann	9

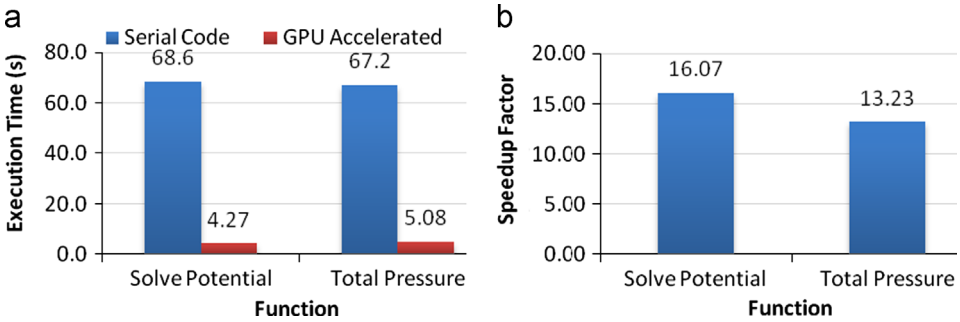

Fig. 22. Dam break problem with 130×130 points: (a) execution times and (b) speedup factors.

mesh-based methods require extensive and time-consuming user input for the generation and verification of a computational mesh. The LRC meshless method seeks to mitigate this issue through the use of a set of scattered points that need not meet stringent geometric requirements like those required by finite-volume and finite-element methods, such as connectivity and poligonalization. The method is shown to render very accurate and stable solutions and the implementation of the solver on the GPU is shown to accelerate the solution by several orders.

References

[1] Lindholm E, Kligard MJ, Moreton H. A user-programmable vertex engine. In: SIGGRAPH '01: Proceedings of the 28th annual conference on computer graphics and interactive techniques. s.l.: ACM Press; 2001. p. 149–58.

[2] Bolz J, Farmer I, Grinspun E, Schroder P. Sparse matrix solvers on the GPU: conjugate gradients and multigrid. In: SIGGRAPH '03. s.l.: ACM Press; 2003. p. 917–24.

[3] Goodnight N, Woolley C, Lewin G, Luebke D, Humphreys G. A multigrid solver for boundary value problems using programmable graphics hardware. In: HWWS '03: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on graphics hardware. s.l.: ACM Press; 2003. p. 102–111.

[4] Liu Y, Liu X, Wu E. Real-time 3D fluid simulation on GPU with complex obstacles. In: 12th Pacific conference on computer graphics and applications; 2004. p. 247–56.

[5] Stam J. Stable fluids. In: SIGGRAPH '99: Proceedings of the 26th annual conference on computer graphics and interactive techniques. s.l.: ACM Press/ Addison-Wesley Publishing Co; 1999. p. 121–28.

[6] Li W, Fan Z, Wei X, Kaufman A. GPU-Based flow simulation with complex boundaries. Addison-Wesley; 2003 (GPU Gems 2. s.l.).

[7] Riegel E, Indinger T, Adams N. Implementation of a Lattice-Boltzmann method for numerical fluid mechanics using the NVIDIA CUDA technology. Comput Sci – Res Dev 2009;23:241–7.

[8] Thibault JC, Senocak I. CUDA implementation of a Navier–Stokes solver on multi-GPU desktop platforms for incompressible flows. In: 47th AIAA aerospace sciences meeting including the new horizons forum and aerospace exposition; 2009.

[9] Cohen JM, Molemaker MJ. A fast double precision CFD code using CUDA. s.l.: NVIDIA Corporation; 2009.

[10] Brandvik T, Pullan, G. Acceleration of a 3D Euler solver using commodity graphics hardware. In: 46th AIAA aerospace sciences meeting and exhibit; 2008.

[11] Belytschko T, Lu YY, Gu L. Element-free Galerkin methods. Int J Numer Methods Eng 1994;37(2):229–56.

[12] Idelsohn S, Oñate E. To mesh or not to mesh. that is the question. Comput Methods Appl Mech Eng 2006;195(37–40):4681–96.

[13] Idelsohn SR, Oñate E, Calvo N, Del Pin F. The meshless finite element method. Int J Numer Methods Eng 2003;58(6):893–912.

[14] Nayroles B, Touzot G, Villon P. Generalizing the finite element method: diffuse approximation and diffuse elements. Comput Mech 1992;10:307–18.

[15] Melenk J. The partition of unity finite element method: basic theory and applications. Comput Methods Appl Mech Eng 1996;139:289–314.

[16] Atluri SN, Zhu T. A new Meshless local Petrov-Galerkin (MLPG) approach in computational mechanics. Comput Mech 1998;22:117–27.

[17] Duarte CA, Oden JT. H-p clouds: an h-p MESHLESS method. Numer Methods Partial Differential Equ 1996;12(6):673–705.

[18] Luo H, Baum J, Lohner R. A hybrid Cartesian grid and gridless method for compressible flows. J Comput Phys 2006;214:618–32.

[19] Tolstykh AI, Shirobokov DA. On using radial basis functions in a finite difference mode with applications to elasticity problems. Comput Mech 2003;33:68–79.

[20] Divo E, Kassab AJ. Localized meshless modeling of natural convective flows. Numer Heat Transf Part B: Fundam 2008;53:487–509.

[21] Divo E, Kassab AJ. An efficient localized radial basis function meshless method for fluid flow and conjugate heat transfer. J Heat Transf 2007;129(2):124–36.

[22] Sarler B, Kosec G. Local RBF collocation method for Darcy flow. Comput Model Eng Sci 2008;25(3):197–207.

[23] Li J. A comparison of efficiency and error convergence of multiquadric collocation method and finite element method. Eng Anal Bound Elements 2003;27:251–7.

[24] Gerace S, Erhart K, Divo E, Kassab AJ. Generalized finite difference Meshless method in computational mechanics and thermofluids. (2009). ECCOMAS Coupled Prob 2009.

[25] Erhart K, Gerace S, Divo E, Kassab AJ. An RBF interpolated generalized finite difference meshless method for compressible turbulent flows. 2009. ASME IMECE; 2009. IMECE2009-11452.

[26] Gerace S, Erhart K, Divo E, Kassab A. Adaptively refined hybrid FDM/Meshless scheme with applications to laminar and turbulent flows. Comput Model Eng Sci 2011;81(1):35–68.

[27] Gerace S, Erhart K, Divo E, Kassab AJ. Local and virtual RBF meshless method for high speed flows, 31. New Forest, UK: WIT Press; 2009; 83–94 (BEM/ MRM).

[28] Batchelor GK. An introduction to fluid dynamics. Caimbridge: Caimbridge University Press; 2000.

[29] Chang YC, Hou TY, Merriman B, Osher S. A level set formulation of eulerian interface capturing methods for incompressible fluid flows. J Comput Phys 1996;124:449–64.

[30] Chorin AJ. Numerical solution of the Navier–Stokes equations. Math Comput 1968;22(104):745–62.

[31] Osher S, Sethian J. Fronts propagating with curvature-dependent speed: algorithms based on Hamilton–Jacobi formulations. J Comput Phys 1988;79 (1):12–49.

[32] Sussman M, Smereka P, Osher S. A level set approach for computing solutions to incompressible two-phase flow. J Comput Phys 1994;114:146–59.

[33] Osher S, Fedkiw R. Level set methods and dynamic implicit surfaces. New York: Springer-Verlag; 2003.

[34] Rouy E, Tourin A. A viscosity solutions approach to shape-from-shading. SIAM J Numer Anal 1992;29(3):867–84.

[35] Fox GC. A review of automatic load balancing and decomposition methods for the hypercube. . Institute for Mathematics and its Applications; 1988.

[36] Fox RW, Pritchard PJ, McDonald AT. Introduction to fluid mechanics. 7th ed.. Hoboken: Wiley; 2009.

[37] Ghia U, Ghia KN, Shin CT. High-RE solutions for incompressible-flow using the Navier–Stokes equations and a multigrid method. J Comput Phys 1982;48 (3):387–411.