Computers and Fluids 184 (2019) 99–106

Contents lists available at ScienceDirect

Computers and Fluids

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

Benchmark solutions

An improved mixed Lagrangian–Eulerian (IMLE) method for modelling incompressible Navier–Stokes flows with CUDA programming on multi-GPUs

Rex Kuan-Shuo Liu[a,b], Cheng-Tao Wu[a], Neo Shih-Chao Kao[a], Tony Wen-Hann Sheu[a,c,d,*]

[a] Department of Engineering Science and Ocean engineering, National Taiwan University, No. 1, Sec. 4, Roosevelt Road, Taipei, Taiwan
[b] Research Department, CR Classification Society, 8F., No. 103, Sec. 3, Nanjing E. Road, Taipei, Taiwan
[c] Institute of Applied Mathematical Sciences, National Taiwan University, Taiwan
[d] Center for Advanced Study in Theoretical Sciences, National Taiwan University, Taiwan

ARTICLE INFO

Article history:
Received 30 October 2018
Revised 21 February 2019
Accepted 24 March 2019
Available online 28 March 2019

Keywords:
Incompressible Navier–Stokes equations
Moving least squares (MLS) interpolation
Conjugate gradient (CG) method
CUDA programming
OpenMP programming

ABSTRACT

In this study, a GPU-accelerated improved mixed Lagrangian–Eulerian (IMLE) method is proposed to solve the three-dimensional incompressible Navier–Stokes equations. To improve the prediction accuracy, the proposed IMLE method approximates the total derivative term in Lagragian sense, and the spatial derivative terms are approximated on Eulerian coordinates. Transfer of data from Lagrangian particles to data on Eulerian grids is accurately carried out by adopting moving least squares (MLS) interpolation method. The velocity-pressure decoupling issue is overcome by adopting pressure-free projection method in which the pressure field is calculated by solving a pressure Poisson equation (PPE). It is noted that the MLS interpolation is time consuming since this procedure belongs to a pointwise scheme in which a local matrix equation shall be solved on each grid point. In addition, the discretized PPE forms a large sparse matrix and it is computationally intensive to solve by using the conjugate gradient (CG) method. Therefore, we are aimed to resort to CUDA- and OpenMP-programming means to accelerate the computation. In this study, the performance of the multiple GPUs code can reach up to 27 times faster with respect to multi-threads CPU performance.

1. Introduction

It has been well known that there are two major classes of numerical methods to solve the incompressible Navier–Stokes equations: density and pressure based methods [1]. For the density-based method, the mass conservation equation is solved for the density field and the pressure field is calculated by the equation of state for ideal fluid. This class of methods is usually used for compressible flows, namely high Mach (*Ma*) number flows. On the other hand, for incompressible (low *Ma* number) fluid flows, a pressure-based method is adopted such as the SIMPLE-family [2–4], PISO [5], or projection [6] algorithms. In these segregated solution algorithms, elliptic pressure Poisson equation (PPE) for pressure or pressure correction of different sorts shall be invoked. It is quite time consuming and becomes sometimes a bottle-neck to solve the discretized PPE, thereby posing a great computational challenge. Thanks to the ever-improving hardware technology, parallel programming technique nowadays plays an important role in solving the incompressible Navier–Stokes equations efficiently in a large physical domain for many industrial flow simulations.

In this study, a GPU-based improved mixed Lagrangian–Eulerian (IMLE) method [7] is proposed. The IMLE method is the refined version of the previously proposed MLE method [8]. In the MLE [8] and IMLE [7] methods, the total and spatial derivative terms in the governing equations are discretized within the Lagrangian and the Eulerian frameworks, respectively. The key contribution of the IMLE method is that a particle reinitialization procedure is adopted in each time step such that the spatial accuracy order can be improved from second (MLE method [8]) to fourth (IMLE method [7]). By adopting the IMLE method, one can then theoretically get rid of the problematic convective instability and retain higher accuracy order at the same time and can effectively transfer data from Lagrangian particles to Eulerian grids by employing the third order accurate moving least squares (MLS) interpolation method [7]. The velocity and pressure variables are separately solved by adopting pressure-free projection method [6] in which a PPE is solved for the primitive pressure variable. It is noted that the MLS

* Corresponding author.
E-mail addresses: ksliu@crclass.org (R.K.-S. Liu), r06525062@ntu.edu.tw (C.-T. Wu), d97525011@ntu.edu.tw (N.S.-C. Kao), twhsheu@ntu.edu.tw (T.W.-H. Sheu).

https://doi.org/10.1016/j.compfluid.2019.03.024
0045-7930/© 2019 Elsevier Ltd. All rights reserved.

interpolation procedure involves solving a large number of smaller local linear systems. The number of local linear systems is equal to the number of mesh cells and the size of local linear system is 5×5 and 10×10 for two- and three-dimensional cases, respectively. On the other hand, the procedure of discretizing the PPE leads to a sparse symmetric and positive definite (SPD) matrix. Even though the use of the conjugate gradient (CG) iterative method can guarantee us to get a convergent solution of a SPD matrix, it is still quite computationally intensive, because the CG method consists of a series of matrix-vector multiplications and inner product operations. The fact that the MLS interpolation and CG methods take over 90% of the computation time (cross-reference to Tables 2, 7 and 8 listed in [7]) motivates us to adopt parallel programming technique to improve the computational performance.

To parallelize the computer code and execute it on multiple GPUs, the data will be firstly sliced into several blocks along the *z*-direction and then these data block are properly distributed onto GPUs. Once the data are evenly scattered onto multiple GPUs, execution of the parallel code is similar to the execution on a single GPU subject to some data synchronization barriers. This study invokes three major numerical methods, namely, the cell-center combined compact difference scheme (CC-CCD), the MLS interpolation method, and the CG iterative method, which consume all together roughly 99% of the execution time. How to properly make the aforementioned three methods parallelizable is indeed an important issue. To solve the solution from the CC-CCD matrix equation, the block-tridiagonal LU factorization with multiple right-hand side method [9] will be utilized. For solving the discretized PPE, the CG method will be applied on multiple GPUs [10–12] to solve the linear system. With the use of multiple GPUs, the speedup ratio can reach up to 27x in our four GPUs platform based on the fine tuned multi-threads CPU results.

The rest of the paper is organized as following. Section 2 shows the governing equations considered in this study, followed by a review of the IMLE method. Section 3 details the implementation of CUDA programming. Section 4 verifies the proposed GPU-accelerated IMLE method by conducting the three-dimensional lid-driven cavity flow simulation and a comparison study with the CPU version of the IMLE method will be made. Section 5 draws some concluding remarks.

2. Numerical methodology

2.1. Governing equations

In this study, the equations for viscous flow motion, namely, the mass and momentum conservation equations for incompressible fluid flow are considered in a three-dimensional space Ω with a boundary $\partial\Omega$.

$$\frac{d\mathbf{r}}{dt} = \mathbf{u} \tag{1}$$

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

$$\frac{d\mathbf{u}}{dt} = -\frac{1}{\rho}\nabla p + \nu\nabla^2\mathbf{u} \tag{3}$$

In the above equations, $\mathbf{r}$ is the particle coordinate, $\mathbf{u}$ the particle velocity, ρ the fluid density, p the pressure and ν the kinematic viscosity of the fluid under investigation.

2.2. The improved mixed Lagrangian–Eulerian (IMLE) method

The most outstanding advantage of solving the momentum equation in Lagrangian sense is that there is no need to discretize the convection terms. As a result, there is no numerical dispersion [13,14] and false diffusion [15,16] errors generated owing to the adoption of upwinding schemes. However, it is not easy to implement high order schemes to approximate the pressure gradient and the velocity Laplacian (physical diffusion) terms on randomly distributed particles [8]. In the IMLE method, the above mentioned spatial derivative terms are discretized by the following sixth order accurate cell-centered combined compact difference (CC-CCD) scheme on the Eulerian grids.

$$\begin{cases} \phi_1' + \frac{8}{9}\phi_2' - \frac{h}{6}\phi_2'' = \frac{1}{h}\left(-\frac{16}{27}\phi_L - \phi_1 + \frac{43}{27}\phi_2\right) \\ \phi_1'' + \frac{2}{3h}\phi_2' = \frac{1}{h^2}\left(\frac{32}{9}\phi_L - 6\phi_1 + \frac{22}{9}\phi_2\right) \end{cases} \tag{4}$$

$$\begin{cases} \phi_1' + \frac{31}{27}\phi_2' - \frac{2h}{9}\phi_2'' = \frac{4}{27}\phi_L' - \frac{2}{h}(\phi_1 - \phi_2) \\ \phi_1'' - \frac{8}{9h}\phi_2' + \frac{1}{3}\phi_2'' = -\frac{8}{9h}\phi_L' \end{cases} \tag{5}$$

$$\begin{cases} \frac{7}{16}\phi_{i-1}' + \phi_i' + \frac{7}{16}\phi_{i+1}' + \frac{h}{16}\left(\phi_{i-1}'' - \phi_{i+1}''\right) \\ = \frac{15}{16h}(-\phi_{i-1} + \phi_{i+1}) \\ -\frac{9}{8h}\left(\phi_{i-1}' - \phi_{i+1}'\right) - \frac{1}{8}\phi_{i-1}'' + \phi_i'' - \frac{1}{8}\phi_{i+1}'' \\ = \frac{3}{h^2}(\phi_{i-1} - 2\phi_i + \phi_{i+1}) \end{cases}, \quad 1 < i < nc \tag{6}$$

In the above equations, ϕ can be one of the velocity components or pressure in order to calculate the physical diffusion or pressure gradient term. Eq. (4) is adopted to calculate the first and the second derivative terms at the first mesh cell corresponding to the Dirichlet boundary condition ϕ_L while Eq. (5) is for the Neumann boundary condition. For interior mesh cells, in Eq. (6), nc denotes the number of cells along each direction. The adopted cell-centered collocated grid system and boundary point arrangement are schematically shown in Fig. 1. It is noted that in an uniform Cartesian mesh, the CC-CCD matrix is the same for each grid line. Therefore, a CC-CCD linear system with $9nc^2$ right hand sides can be solved in parallel to get the first and the second derivative values for every function with respect to every direction at all mesh cells simultaneously. In other words, the values of $(u_x)_{ijk}$, $(u_{xx})_{ijk}$, $\cdots$, $(w_z)_{ijk}$ and $(w_{zz})_{ijk}$, where $i,\ j,\ k = 1 \sim nc$, can be obtained simultaneously.

In order to solve the variables $\mathbf{u}$ and p in a decoupled way, the pressure-free projection method in [6] is adopted. In the first step, the intermediate particle velocity is calculated from Eq. (3) without

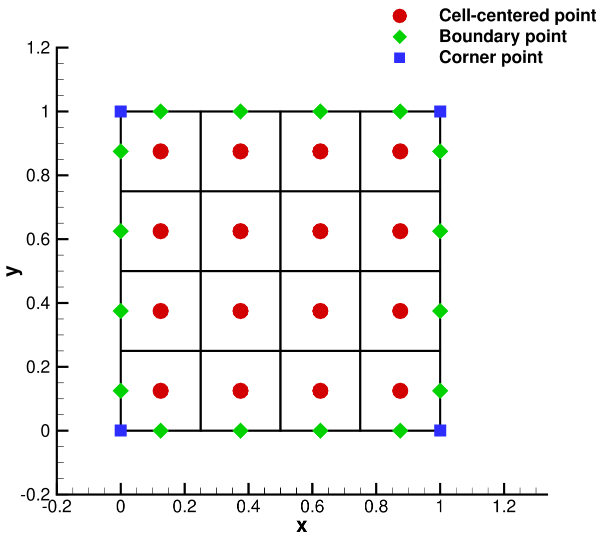

Fig. 1. Cell-centered collocated grids in a two-dimensional case. (Cell-centered point stores variables $\mathbf{u}$ and p; Boundary point stores boundary values of $\mathbf{u}$ and p; Corner point stores boundary values of $\mathbf{u}$.).

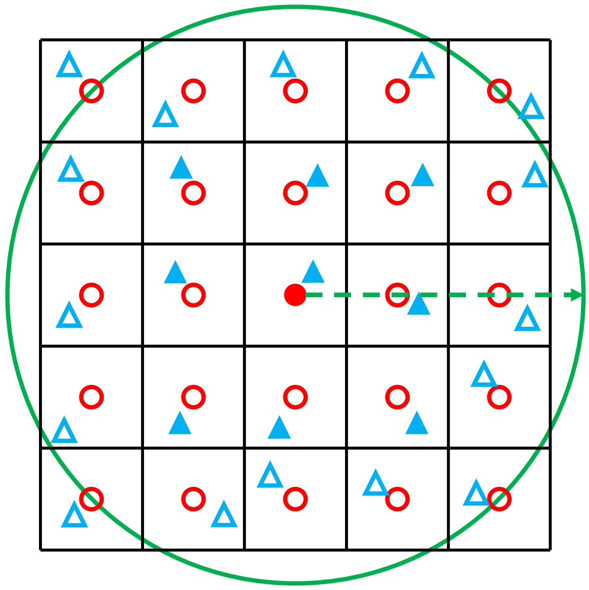

Fig. 2. Schematic of the MLS interpolation process. (Circles denote Eulerian grids; Triangles denote Lagrangian particles; Solid symbols denote the points participating in interpolation process; Hollow symbols denote the points not participating in interpolation process).

considering the pressure gradient term, followed by the calculation of the intermediate particle coordinate.

$$\mathbf{u}^* = \mathbf{u}^n + \Delta t \nu \nabla^2 \mathbf{u}^n \tag{7}$$

$$\mathbf{r}^* = \mathbf{r}^n + \Delta t \mathbf{u}^* \tag{8}$$

In the above equations, the superscripts n and $*$ denote the time step level and Δt is the time step size. After performing the particle movement step (Eq. (8)), the particles are no longer uniformly located on the Cartesian grids but they are distributed slightly non-uniform. Therefore, a MLS interpolation step shall be taken to interpolate the intermediate velocities from the Lagrangian particles to the Eulerian grids, as shown in Fig. 2. To perform MLS interpolation, a local polynomial, taking the 2D case for an example, is constructed for each grid point as following.

$$f(x,y) = a_0 + a_1 x + a_2 y + a_3 x^2 + a_4 xy + a_5 y^2 + O\left(x^3, y^3\right) \tag{9}$$

The above six introduced coefficients can be solved from the following derived equation underlying the moving least squares method.

Since the intermediate velocity field is not divergence-free, the following modified step is adopted.

$$\mathbf{u}^{n+1} = \mathbf{u}^*_{interpolated} - \frac{\Delta t}{\rho} \nabla p^{n+1} \tag{11}$$

It is noted that the value of p^{n+1} is needed to be calculated prior to solving Eq. (11). The PPE can be therefore derived by taking a divergence operation on Eq. (11) and, at the same time, setting $\nabla \cdot \mathbf{u}^{n+1} = 0$.

$$\nabla^2 p^{n+1} = \frac{\rho}{\Delta t} \nabla \cdot \mathbf{u}^*_{interpolated} \tag{12}$$

The above PPE can be discretized by the conventional second order central difference scheme and the resulting matrix equation can be solved by the CG solution solver. Since the resulting sparse matrix is large in a three dimensional context, we are motivated to conduct parallel programming calculation to increase computational performance. The numerical procedure of the IMLE method is summarized in Algorithm 1.

Algorithm 1 Algorithm of the IMLE method.

```
1: Read input file
2: Calculate the computational parameters
3: for $time < MaxTime$ do
4:     Calculate the diffusion terms on Eulerian grids.
5:     Calculate the intermediate velocities on Eulerian grids.
6:     Calculate the intermediate Lagrangian particle locations according to the intermediate velocities.
7:     Interpolate the intermediate velocities from Lagrangian particles to Eulerian grids.
8:     Solve the PPE.
9:     Update the intermediate velocities on Eulerian grids.
10: end for
11: Output the results for post-processing
```

3. CUDA implmentation

In our computing environment, there are four Nvidia Titan V GPUs and two Intel Xeon E5 CPUs. The GPU cards are connected with CPU by using PCIe Gen3 $\times$ 16 slots. The theoretical bandwidth for each PCIe bus is 15.8 GB/s. In comparsion with the GPU theoretical computational performance, which is 7.8 TFLOPS for double precision [17], the PCIe bandwidth is extremely small. Therefore, how to properly distribute data to GPUs will affect the number of data movement between CPU and GPUs, which can significantly affect the parallel performance. After the data distribution, the multiple GPUs computation is similar to the computation executed on

$$\begin{bmatrix}
\sum\omega_i^2 & \sum\omega_i^2 x_i & \sum\omega_i^2 y_i & \sum\omega_i^2 x_i^2 & \sum\omega_i^2 x_i y_i & \sum\omega_i^2 y_i^2 \\
\sum\omega_i^2 x_i & \sum\omega_i^2 x_i^2 & \sum\omega_i^2 x_i y_i & \sum\omega_i^2 x_i^3 & \sum\omega_i^2 x_i^2 y_i & \sum\omega_i^2 x_i y_i^2 \\
\sum\omega_i^2 y_i & \sum\omega_i^2 x_i y_i & \sum\omega_i^2 y_i^2 & \sum\omega_i^2 x_i^2 y_i & \sum\omega_i^2 x_i y_i^2 & \sum\omega_i^2 y_i^3 \\
\sum\omega_i^2 x_i^2 & \sum\omega_i^2 x_i^3 & \sum\omega_i^2 x_i^2 y_i & \sum\omega_i^2 x_i^4 & \sum\omega_i^2 x_i^3 y_i & \sum\omega_i^2 x_i^2 y_i^2 \\
\sum\omega_i^2 x_i y_i & \sum\omega_i^2 x_i^2 y_i & \sum\omega_i^2 x_i y_i^2 & \sum\omega_i^2 x_i^3 y_i & \sum\omega_i^2 x_i^2 y_i^2 & \sum\omega_i^2 x_i y_i^3 \\
\sum\omega_i^2 y_i^2 & \sum\omega_i^2 x_i y_i^2 & \sum\omega_i^2 y_i^3 & \sum\omega_i^2 x_i^2 y_i^2 & \sum\omega_i^2 x_i y_i^3 & \sum\omega_i^2 y_i^4
\end{bmatrix}
\begin{Bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ a_4 \\ a_5 \end{Bmatrix}
=
\begin{Bmatrix} \sum\omega_i^2 f_i \\ \sum\omega_i^2 f_i x_i \\ \sum\omega_i^2 f_i y_i \\ \sum\omega_i^2 f_i x_i^2 \\ \sum\omega_i^2 f_i x_i y_i \\ \sum\omega_i^2 f_i y_i^2 \end{Bmatrix} \tag{10}$$

In the above equation, subscript i is the particle index, f_i the interpolated function (u^*, v^*) on particle i, ω_i the weighting function which is shown in [8]. It is noted that at each cell center, a local matrix equation needs to be solved to get the intermediate velocities. Therefore, a total number of nc^{dim} (nc is the number of mesh cells along each direction and dim is the dimension of the problem) local matrix equations should be solved and it is a computationally intensive step.

a single GPU. The only difference lies in the requirements of additional synchronous barriers and data exchange.

3.1. Data decomposition

In order to properly distribute data to GPUs and reduce the number of data transferring, the original particles and grids will

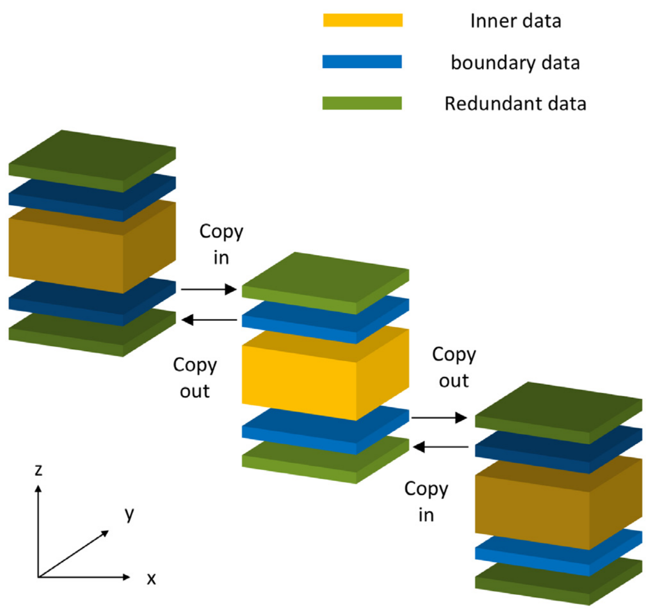

Fig. 3. The original data will be sliced into several pieces. In each piece, there are inner data, boundary data and redundant data blocks.

be evenly divided into several blocks along the z-axis. Besides, additional data blocks will be added into each GPU, which are the boundary data blocks and the redundant data blocks. As shown in Fig. 3, these additional data blocks will be used to exchange the data with their nearby GPUs. In our computational environment, NVLink is not supported. Therefore, the boundary data of GPU will be first copied to the main memory and, then, copied to the nearby GPUs' redundant data blocks. With the use of these redundant data blocks, it is convenient for data accessing and is possible to reduce the total amount of communication. It is worth to note that with the use of redundant data blocks, CUDA streams can be utilized to asynchronously copy the boundary data to the main memory. One can also use another stream for computing the inner data. Therefore, the communication cost can be hidden by another computation.

3.2. *Cell-centered combined compact difference scheme*

On uniform Cartesian mesh, the first and the second derivative terms can be computed by using the sixth-order accurate combined compact difference scheme described in Section 2.2. By virtue of Eqs. (4) and (6) or Eqs. (5) and (6), one can get the CC-CCD matrix and the right hand side vector for the case with Dirichlet or Neumann boundary condition, respectively. It is worthy to note that the CC-CCD matrices are the same in the same direction. As a result, the linear system can be solved with multiple right-hand side (MRHS) vectors.

By using one of the Eqs. (5) and (4) and Eq. (6), the CC-CCD matrix can be constructed. The size of the CC-CCD matrix is $2nc \times 2nc$, where nc denotes the number of mesh cells along one direction. In addition, the CC-CCD matrix is a block-tridiagonal matrix. The size of the submatrix is 2×2. As a result, the CC-CCD matrix can be compactly stored as a $6 \times 2nc$ matrix. Besides, all the coefficients in the CC-CCD matrix are constants. Thus, the CC-CCD matrix can be firstly factorized by the block-tridiagonal LU decomposition method and the matrices **L** and **U** can be reused in the subsequent computation. Since the CC-CCD matrix is banded and will be computed only once in our algorithm, it is convenient to use the CPU to factorize the CC-CCD matrix and copy the resulting **L** and **U** matrices to GPUs.

After the LU-factorized CC-CCD matrices are computed and passed to GPUs, the MRHS vectors will be computed on GPU through Eqs. (4) and (6) or Eqs. (5) and (6) for the cases subject to Dirichlet or Neumann boundary condition, respectively. The size of each RHS column is $2nc$, and there are nc^2 independent RHS vectors in one coordinate direction. To construct and then solve the linear system, nc^2 CUDA threads will be launched and executed in parallel. For three-dimensional problems, the CC-CCD scheme will be applied to all three directions separately.

3.3. *Moving-Least-Squares interpolation method*

After the calculations of the intermediate velocities and coordinates of the Lagrangian particles, the MLS interpolation method will be used to interpolate the three-dimensional velocities from Lagrangian particles to Eulerian grids. In the MLS interpolation method, a 10 by 10 matrix **A** (the three-dimensional version of the matrix shown in (10)) and a 10 by 3 RHS matrix will be constructed for each grid to interpolate the intermediate velocities to Eulerian coordinates. To solve this 10 by 10 local matrix, Gaussian elimination direct solver will be applied. In CPU computation, the three nested loops will be utilized to compute the solution for the three-dimensional problem. Also, since the constructions of the matrix **A** and the RHS matrix are highly data independent, the code can be easily parallelized for multi-threads CPU. Only the OpenMP parallel do directive (`$omp parallel do`) needs to be added to the outermost loop and no synchronization barrier shall be encountered.

For GPU, it is, however, difficult to launch a thread block to construct multiple 10 by 10 matrices and 10 by 3 RHS matrices and, then, store these matrices in shared memory. The sizes of the registers and the shared memory on GPU are limited. Also, the Gaussian elimination solver is highly sequential, it's difficult to solve these nc^3 linear systems in parallel. Instead of trying to store all the data within shared memory and solve the linear system separately, all the matrices will be first computed and stored in the global memory. The size of the memory is (nc, nc, 10, 10, nc) for matrices **A** and the size is (nc, nc, 10, 3, nc) for RHS matrices. After the matrix being obtained, the Gaussian elimination method will be performed on these small matrices. To get the matrices and compute the solution, one thread will take responsibility for one grid computation. In sum, (nc, nc, nc) threads will be launched in each iteration.

As the problem becomes larger, the number of matrices **A**, which is nc^3, will increase dramatically. Once the size of matrices **A** exceeds 2 GB, it is impossible to index the memory by using the 4-byte signed integer. For three-dimensional problems, the issue of the indexing limitation will occur if more than 138 particles are used ($nc^3 = \frac{2^{31}\text{bytes}}{(8\text{bytes}\times 10^2)} \Rightarrow nc \approx 138$). By dividing these particles into several batches along the z direction with the formula $\frac{nc\times(100\times 8\times nc^2)}{2^{31}} + 1$, the memory consumption will not exceed 2 GB. Therefore, no matter how large the problem is, the MLS interpolation method will never suffer from memory limitation issue.

3.4. *Pressure Poisson equation*

The PPE is discretized by the conventional second order central difference scheme expressed as

$$p_{i+1,j,k} + p_{i-1,j,k} + p_{i,j+1,k} + p_{i,j-1,k} + p_{i,j,k+1} + p_{i,j,k-1} - 6p_{i,j,k}$$
$$= \frac{\rho h}{2\Delta t}\left(u^*_{i+1,j,k} - u^*_{i-1,j,k} + v^*_{i,j+1,k} - v^*_{i,j-1,k} + w^*_{i,j,k+1} - w^*_{i,j,k-1}\right) \quad (13)$$

In the above equation, u^*, v^* and w^* are the interpolated intermediate velocities and they are used to construct the RHS vector **b**.

To solve the pressure value p, the linear system will be solved by using the CG iterative method since the matrix $\mathbf{A}$ is symmetric and positive definite.

The detailed CG method is shown in Algorithm 2. For each iteration (lines 4–14), there are two dot products (lines 7 and 10), three vector operations (lines 8, 9, and 12) and one sparse matrix-vector multiplication (line 6) operation. For the dot product operation, it is convenient to call the Nvidia's cuBLAS library.

Algorithm 2 Algorithm of the conjugate gradient solver on multiple GPUs.

```
1:  $\mathbf{r} = \mathbf{b} - \mathbf{A}\mathbf{x}_0$
2:  $\mathbf{v} = \mathbf{r}$
3:  $rs_{old} = \mathbf{r} \cdot \mathbf{r}$
4:  for $Iter < MaxIter$ and $\sqrt{rs_{old}/\mathrm{size}(\mathbf{A})} > MaxErr$ do
5:      Synchronize boundary value of $\mathbf{v}$ with the nearby GPUs
6:      $\mathbf{v}_2 = \mathbf{A}\mathbf{v}$
7:      $\alpha = \mathbf{v} \cdot \mathbf{v}_2 / rs_{old}$
8:      $\mathbf{x} = \mathbf{x} + \alpha\mathbf{v}$
9:      $\mathbf{r} = \mathbf{r} - \alpha\mathbf{v}_2$
10:     $rs_{new} = \mathbf{r} \cdot \mathbf{r}$
11:     $\beta = rs_{new}/rs_{old}$
12:     $\mathbf{v} = \mathbf{r} + \beta\mathbf{v}$
13:     $rs_{old} = rs_{new}$
14: end for
```

For the sparse matrix-vector multiplication operation (line 6), since the matrix coefficients are not stored in the CSR format but in grids, it is impossible to use the SPMV library which was developed by Nvidia. Instead, the in-house kernel is utilized. With the use of this in-house kernel, the memory consumption can be significantly reduced because no additional memory is required to store the index array of matrix $\mathbf{A}$. The size of the index array is roughly equal to $6 \times nc^3 + nc^3$, where nc denotes the number of mesh cells along three directions. For the largest case conducted in this study, the number of mesh cells is 200 along three directions and the total amount of memory being used is roughly 448 MB (in double precision). In CUDA programming, memory consumption is a crucial issue since Nvidia Titan V GPU only has 12 GB memory or even less in the previous version of GPU. The more memory one can save, the larger problem one can solve.

To execute the code on multiple GPUs, additional synchronization barrier is encountered since all the data are sliced into several blocks along z-direction and so do the residual vector $\mathbf{r}$ and the conjugate vector $\mathbf{v}$. To synchronize the data of conjugate vector $\mathbf{v}$, line 5 will be added to Algorithm 2 for exchanging the boundary data $\mathbf{v}$ with the nearby GPUs. In our computing environment, NVLink is not supported. As a result, to exchange these data, all boundary data need to be copied to main memory first and then copied back. This is time consuming in each iteration of employing CG method, since for 200 mesh cells, 312.5 KB ($200^2 \times 8$ bytes) data need to be transferred back and forth between the main memory and GPUs. It is possible to use CUDA streams to hide the data transferring process by computing the non-overlapping data. With the use of the overlapping technique, the overall performance becomes slightly better than those without the use of the overlapping technique [10,11].

4. Validation and speedup

4.1. Lid-driven cavity flow

To validate the developed GPU code, the three-dimensional lid-driven cavity problem in a unit cube domain will be considered. The simulation setting for the characteristic velocity, length and

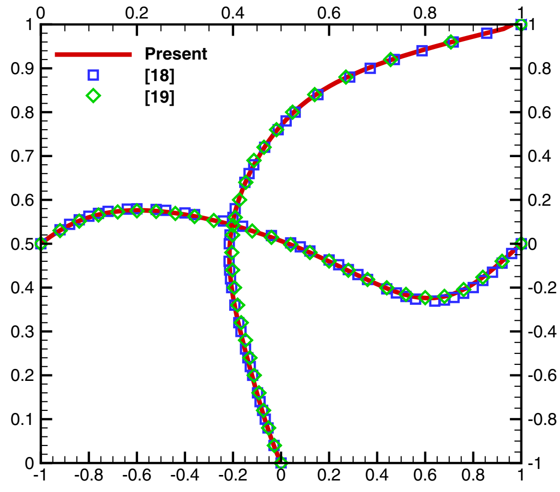

Fig. 4. Velocity profiles for the case of $Re = 100$ on the vertical and horizontal centrelines.

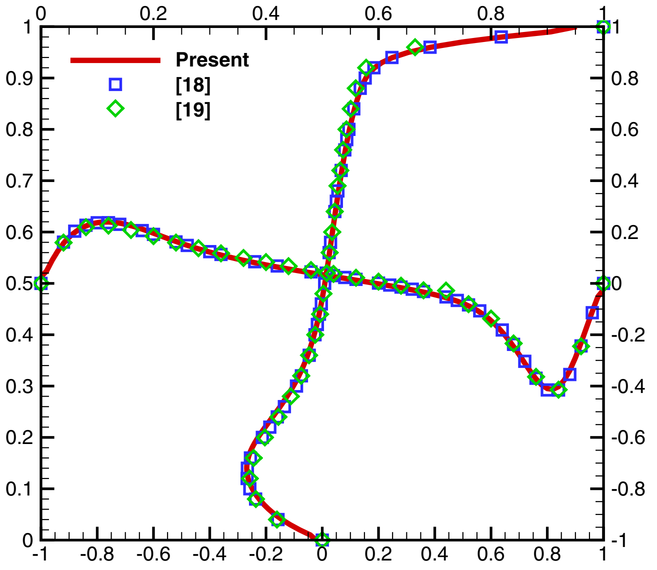

Fig. 5. Velocity profiles for the case of $Re = 1000$ on the vertical and horizontal centrelines.

density are 1. Besides, the Reynolds numbers under consideration are 100 and 1000 while the simulation times are 50.0 and 200.0, respectively. The u velocity data are obtained along the line which is intersected by planes $x = 0.5$ and $y = 0.5$, while the w velocity data are collected along the line which is intersected by the planes $y = 0.5$ and $z = 0.5$. To properly calculate the diffusion and the convective effects, fine grids/particles are utilized. Thus, our validation studies are based on the results obtained from 100^3 grids/particles. In Figs. 4 and 5, the GPU simulated results are compared with the reference data collected from [18,19]. For the cases with two Reynolds numbers 100 and 1,000, our results agree quite well with the reference data. According to Figs. 4 and 5, we claim that our multiple GPU code is correct.

4.2. Speedup

For the computational environment, the CPU code is executed on a CPU server which is featured with two Intel Xeon E5-2630 v2 CPUs. The performance of CPU code will then be assessed with that obtained from the multiple GPUs code which is executed on

Table 1
Specifications of CPU and GPU computing enviroments.

Characteristics	Dual Intel Xeon E5-2630 v2	Nvidia Titan V
Number of cores	12 cores	5120 CUDA cores
DP theoretical performance	297.8 GFlops	6900 GFlops
Cache	30 MB for L3 cache	4.5 MB for L2 Cache
Memory	128 GB	12 GB on chip memory
Max memory bandwidth	102.4 GB/s	652.8 GB/s

four Nvidia Titan V GPUs. The specifications for both the CPU and GPU are summarized in Table 1.

In the following performance assessment, since it is time consuming to simulate the lid-driven cavity flow problem in 101^3 grids on a single CPU core, it is better to adopt multi-threads CPU code for calculating the speedup of GPU calculation. Therefore, the fine tuned multi-threads CPU code will be first juxtaposed with the single thread version. By doing so, it is possible to justify whether the multi-threads CPU code has been properly parallelized. Later, the simulation in 101^3 grids will be chosen to show the speedup of twelve threads CPU and multiple GPUs results where the speedup is calculated based on the single thread CPU result. Finally, the multiple GPUs speedup will be evaluated based on the twelve threads CPU result. For the sake of comparison, the grid points of 81^3, 101^3, 151^3, 201^3 are adopted.

4.2.1. Performance evaluation of multi-threads and single thread CPU code

To validate whether the multi-threads CPU code is properly parallelized or not, the speedup of each subroutine is shown in Fig. 6. It is evident that the MLS interpolation can easily approach the theoretical speedup while the CC-CCD and CG methods can not. This is because the MLS interpolation method has a higher data independency and data locality in comparison with the CC-CCD and CG methods. Owing to higher data independency and locality, the performance will not be limited by the memory bandwidth or the size of L1 to L3 caches.

On the one hand, poor speedup of the CC-CCD method is owing to the calculation of y- and z-directional first and second derivative terms. Since the data of u, v, w velocities and pressure p are stored in three dimensional array, it is impossible to access the y and z directional data in alignment. In order to access these data and compute the first and second derivative terms in parallel, the computation speedup will be limited by the size of memory bandwidth. In our computing environment, the memory bandwidth is 102.4 GB/s for two CPUs. In the twelve threads computational setting, each thread can only access 8.5 GB data per second. For the single thread setting, however, one thread can access 51.2 GB data in each second. Relatively small memory bandwidth can not completely satisfy the memory requirement for the multi-threads CPU execution. Therefore, it is evident that the speedup of CC-CCD method is limited by memory bandwidth.

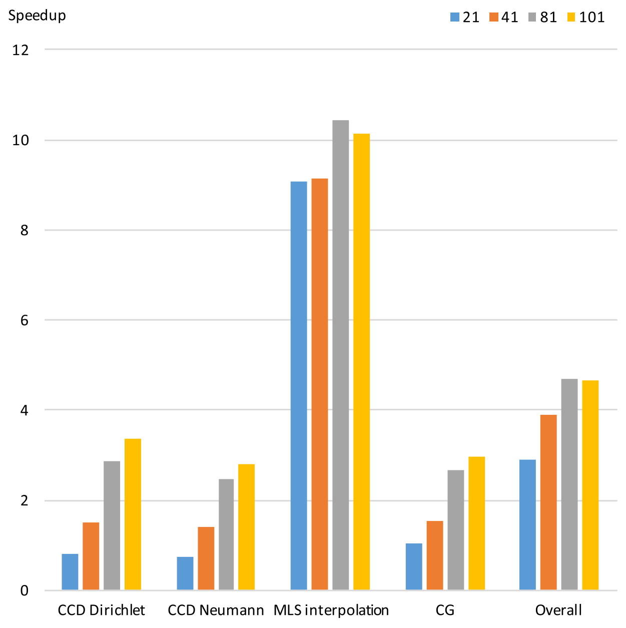

Fig. 6. Speedup of the twelve threads setting compared to the single thread setting for calculations carried out in different grids.

The limited speedup of the CG method is not due to memory bandwidth but is resulted from some dependent operations. For the CG method, although most of the operations can be done in parallel, the dot product operation is highly sequential. To conduct the dot product operation, each element of a vector needs to be multiplied by its corresponding element of another vector. Then, a reduction algorithm will be applied to sum up the multiplied data together. Later, the synchronization barrier will be inserted to collect the results. This synchronization barrier is, however, the bottleneck of achieving theoretical speedup. With the synchronization barrier, all the threads need to wait for the slowest thread to finish its computation. In CG method, this synchronization barrier will be launched multiple times during the calculation of linear system. After all, the performance will be slowed down and can not reach the theoretical speedup.

As to the MLS interpolation method, unlike the CC-CCD and CG methods, this method has an extremely high data independency and does not require large memory bandwidth. It is because the MLS interpolation method involves mainly the calculation of 10 by 10 local matrix **A** and 10 by 3 right hand side matrix **b**. The matrices **A** and **b** take 960 bytes (in double precision) for each thread, which is relatively small in comparison with 30 MB L3 cache. Thanks to the nature of higher memory locality, the execution of MLS interpolation will not be limited by memory bandwidth. Therefore, the actual speedup for the MLS interpolation can approach almost the theoretical speedup.

4.2.2. Performance evaluation of multiple GPUs, multi-threads and single thread codes

Since the speedup of twelve threads result becomes saturated for the case involving 101^3 grid points, it is better to consider this number of grid points to verify the speedup of GPU result. In Fig. 7, the single thread performance is used as the baseline, and the speedup of the twelve CPU threads result and the multiple GPUs result are juxtaposed.

In the single GPU version of the CC-CCD method, tremendous speedup performance can be easily achieved. In contrast, in multiple GPUs' calculation, no apparent speedup can be gained due to the additional data exchange needed between different GPUs. In single GPU, all the data are stored inside GPU's on-chip memory, with wider memory bandwidth and more execution cores. As a result, it can easily approach larger speedup. For multiple GPUs

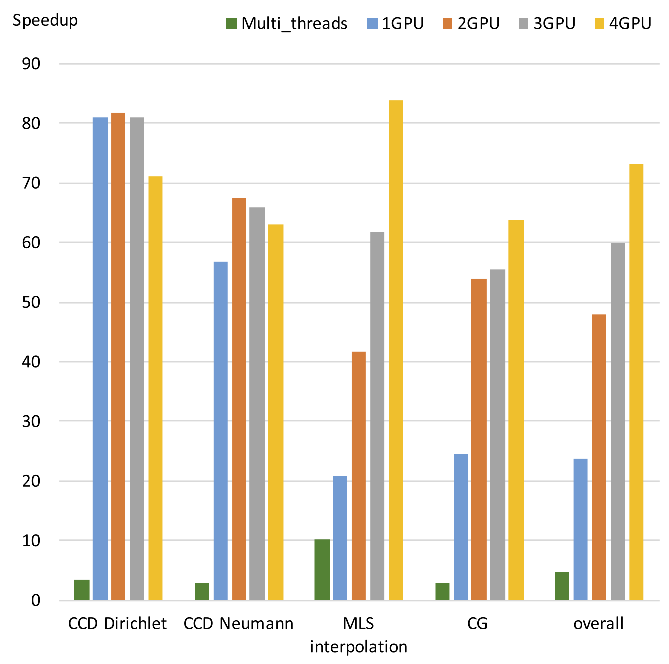

Fig. 7. The speedup of twelve CPU threads and multiple GPUs based on single thread performance.

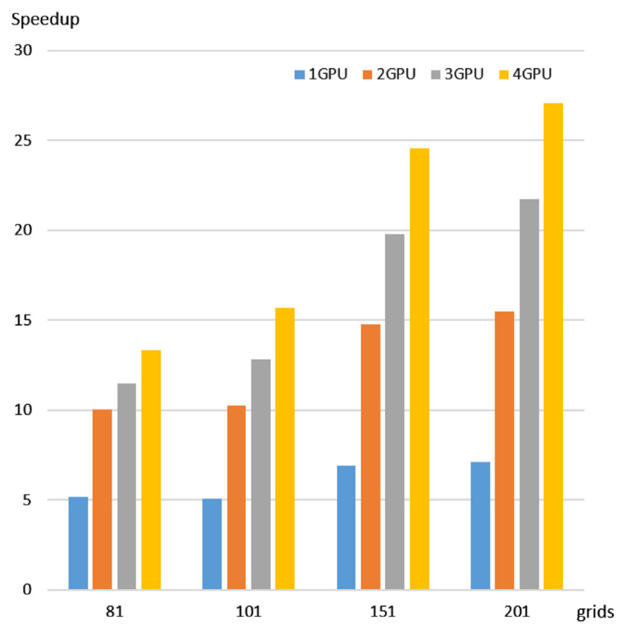

Fig. 8. Speedup of the multiple GPUs results in comparison with the twelve threads CPU results obtained in different grid points.

code, however, the data of velocities u, v, w and pressure p have been sliced into pieces along the z-direction and these data are scattered to different GPUs. To compute the z-directional first and second derivative terms, all the data need to be patched together. In our computing environment, without the support of NVLink, all the data need to be copied to the main memory and then copied back to the GPUs' memory. Although the execution overlapping technique with CUDA stream can be utilized, the speedup is still limited due to a large amount of data needed to be transferred. Taking 101^3 grid points as an example, the amount of data that needs to be transferred between four GPUs is $8 \times 100^2 \times 25$ bytes (≈ 2 MB) for a variable. Therefore, no furthermore speedup can be gained any longer in multiple GPUs code.

The speedup of the MLS interpolation method is, however, limited in single GPU but the speedup grows linearly as more GPUs are available. For MLS interpolation method implemented in GPUs, instead of constructing and solving the linear system with a 10 by 10 matrix **A** and 10 by 3 matrix **b**, matrices of dimensions (nc, nc, 10, 10, nc) and (nc, nc, 10, 3, nc) are constructed and used to solve for the interpolation intermediate velocities. Since all the CUDA cores will access these data in poor alignment, the computation requires a wider memory bandwidth. The limited speedup is arisen from an extra need of memory bandwidth. However, the computation on each GPU is independent, a relatively linear speedup with multiple GPUs can be achieved.

The CG method implemented in a single GPU encounters the similar speedup restriction as multi-threads CPU code that the dot product operation will slow down the parallel computing. For the application of CG method on multiple GPUs, additional synchronous barrier is needed to exchange data as shown in line 5 of Algorithm 2. As discussed in [10,11], this additional synchronization is used to exchange data between different GPUs and will reduce the performance of multi-GPUs algorithm. As more GPUs are added, the speedup shall be grown in a linear fashion. It is worth to note that the Titan V in our computational environment is much better than the GPUs used in [10,11]. The speedup of our CG method is also much higher than those reported in [10,11].

4.2.3. Speedup of multiple GPUs code based on multi-threads code

Finally, the multi-threads performance will be used as the baseline to evaluate the speedup of multiple GPUs performance for the case with larger number of grid points. The simulations involving 81^3, 101^3, 151^3 and 201^3 grid points are performed.

As shown in Fig. 8, for the case of 201^3 grid points, the overall speedup of single GPU simulation can reach up to 7x, as compared to our multi-threads CPU results. Besides, as more GPUs are used, the speedup for two, three and four GPUs can reach roughly to 15x, 21x, and 27x, respectively. In light of the fact that the communication cost becomes higher when more GPUs are used, the overall performance in terms of speedup is quite satisfactory. In other words, the data decomposition method used together with the execution-communication overlapping technique can indeed accelerate computation and improve the speedup while more GPUs are utilized.

5. Concluding remarks

The newly developed IMLE particle method involves three major numerical schemes, which are the cell-center combined compact difference scheme (CC-CCD), the moving least squares (MLS) interpolation scheme and the conjugate gradient (CG) iterative solver. The block-tridiagonal LU decomposition method with multiple right hand side vectors is used to solve the CC-CCD matrix equation. For the MLS interpolation scheme, Gaussian elimination method is used to solve the multiple local small matrices. Finally, the classical CG method is used to solve the PPE.

To parallelize the code and execute it on multiple GPUs, the computational domain is decomposed along the z-axis. For rendering a convenient data accessing, redundant data blocks are utilized for data exchange. By virtue of the CUDA stream, the data exchange process can be overlapped by the computation on inner data. Through the GPU acceleration on the three time-consuming numerical schemes, the speedup for one to four GPUs can reach 7x, 14x, 21x and 27x, respectively, with respect to the multi-threads CPU simulation carried out in 201^3 grid points.

Acknowledgements

This work was supported by the Ministry of Science and Technology (MOST) of Republic of China (R.O.C.) under the grant MOST 106-2221-E-002-107-MY2.

References

[1] Miettinen A, Siikonen T. Application of pressure- and density-based methods for different flow speeds. Int J Numer MethodsFluids 2015;79:243–67.
[2] Patankar SV, Spalding DB. A calculation procedure for heat, mass and momentum transfer in three-dimenional parabolic flows. Int J Heat Mass Transf 1972;15:1787–806.
[3] Patankar SV. A calculation procedure for two-dimensional elliptic situations. Numer Heat Transf 1981;4:409–25.
[4] Doormaal JP, Raithby GD. Enhancement of the SIMPLE method for predicting incompressible fluid flows. Numer Heat Transf 1984;7:147–63.
[5] Issa R. Solution of the implicitly discretised fluid flow equations by operator-splitting. J Comput Phys 1986;62(1):40–65.
[6] Chorin AJ. Numerical solution of the Navier-Stokes equations. Math Comput 1968;22:745–62.
[7] Liu RKS, Ng KC, Sheu TWH. A new high order particle method for solving high Reynolds number incompressible flows. Computational Particle Mechanics 2018. doi:10.1007/s40571-018-00217-w.
[8] Liu KS, Sheu TWH, Hwang YH, Ng KC. High-order particle method for solving incompressible Navier-Stokes equations within a mixed Lagrangian-Eulerian framework. Comput Methods Appl MechEng 2017;325:77–101.
[9] Jia Y, Luszczek P, Dongarra J. Multi-GPU implementation of LU factorization. Proc Int ConfComput Sci 2012;9:106–15.
[10] Cevahir A, Nukada A, Matsuoka S. Fast conjugate gradients with multiple GPUs. In: Computational Science – ICCS 2009. Springer Berlin Heidelberg; 2009. p. 893–903.
[11] Oyarzuna G, Borrellb R, Gorobetsac A, Olivaa A. MPI-CUDA sparse matrix-vector multiplication for the conjugate gradient method with an approximate inverse preconditioner. Comput Fluids 2014;92:244–52.
[12] Georgescu S, Okuda H. Conjugate gradients on multiple GPUs. Int J Numer Methods Fluids 2010;64:1254–73.
[13] Chiu PH, Sheu TWH. On the development of a dispersion-relation-preserving dual-compact upwind scheme for convection-diffusion equation. J Comput Phys 2009;228:3640–55.
[14] Bhumkar Y, Sheu TWH, Sengupta TK. A dispersion relation preserving optimized upwind compact difference scheme for high accuracy flow simulations. J Comput Phys 2014;278:378–99.
[15] Carey C, Scanlon TJ, Fraser SM. SUCCA - an alternative scheme to reduce the effects of multidimensional false diffusion. Appl Math Model 1993;17:263–270.
[16] Bailey RT. Managing false diffusion during second-order upwind simulations of liquid micromixing. Int J Numer Methods Fluids 2017;83:940–59.
[17] NVIDIA. Nvidia Tesla V100 GPU accelerator; 2018.
[18] Lo DC, Murugesan K, Young DL. Numerical solution of three-dimensional velocity-vorticity Navier-Stokes equations by finite difference method. Int J Numer Methods Fluids 2005;47:1469–87.
[19] Shu C, Wang L, Chew YT. Numerical computation of three-dimensional incompressible Navier-Stokes equations in primitive variable form by DQ method. Int J Numer Methods Fluids 2003;43:345–68.