Computers and Fluids 167 (2018) 285–291

Contents lists available at ScienceDirect

Computers and Fluids

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

Benchmark solutions

Development of a finite element flow solver for solving three-dimensional incompressible Navier–Stokes solutions on multiple GPU cards

Neo Shih-Chao Kao[a], Tony Wen-Hann Sheu[a,b,c,*]

[a] *Department of Engineering Science and Ocean Engineering, National Taiwan University, Taipei, Taiwan*
[b] *Center of Advanced Study in Theoretical Sciences (CASTS), National Taiwan University, Taipei, Taiwan*
[c] *Institute of Applied Mathematical Sciences, National Taiwan University, Taipei, Taiwan*

ARTICLE INFO

Article history:
Received 9 February 2017
Revised 3 March 2018
Accepted 9 March 2018
Available online 12 March 2018

Keywords:
CUDA
GPU
Finite element
Petrov–Galerkin
Streamline upwind

ABSTRACT

In this paper a multi-GPU-based finite element flow solver is developed to solve the three-dimensional incompressible Navier–Stokes equations at steady-state. To circumvent the convective instability problem at high Reynolds numbers, the proposed streamline upwinding finite element model minimizes the wavenumber error for the convection terms. Mixed finite element formulation is adopted and the resulting nearly ill-conditioned finite element equations are solved iteratively. To avoid the Lanczos or pivoting breakdown, the finite element equations are first normalized. The computationally efficient preconditioned conjugate gradient (PCG) solver can then be applied to get the unconditionally convergent solution. The developed finite element code implemented on multi-GPU cards will be verified and validated by solving the problem amenable to analytical solution and the benchmark lid-driven cavity problem, respectively.

1. Introduction

The finite element method (FEM) has long been regarded as an important tool to predict the incompressible flow equations because of its appealing advantages in handling complex geometry and easy treatment of the natural boundary condition. FEM is also mathematically rich in providing analysis of convergence proof [1]. However, the cost of solving the resulting finite element equations for three-dimensional problem is very often prohibitive. For the sake of efficiency, the parallel computing technique is often used to accelerate the calculation [2].

Recently, graphic processing unit (GPU), originally designed for rendering high resolution graphics, now has drawn more attention for scientific computing application due to its tremendous floating-points peak performance, multicore design and high memory bandwidth compared to CPU [3,4]. Moreover, the introduction of the compute unified device architecture (CUDA) programming model proposed by Nvidia in 2007 [5] makes the GPU become a high performance accelerator for data-parallel and compute-intensive tasks.

Based on the above facts, the objective of this study is to develop a fast and robust finite element flow solver to solve the three-dimensional incompressible Navier-Stokes equations. The finite element equations will be solved iteratively on a multi-GPU architecture.

This paper is organized as follows: In Section 2, the incompressible Navier-Stokes equation cast in primitive variables is introduced. The developed Petrov–Galerkin finite element model and the elementary matrix modification procedure are described in Section 3. In Section 4, the multi-GPU-based iterative solver is introduced. In Section 5, the developed GPU finite element code is verified and validated by solving the problem amenable to the analytical solution and investigating the benchmark lid-driven cavity flow problem. The speedup performance is also investigated. Finally, some conclusions will be drawn in Section 7.

2. Governing equations

Let $\Omega \subseteq \mathbb{R}^3$ be an open and bounded domain and Γ denote its boundary. The steady, viscous and incompressible Navier–Stokes equations, which describe the fluid flow driven by a pressure gradient ∇p and a force term $\underset{\sim}{f}$, are expressed in terms of the

* Corresponding author at: Department of Engineering Science and Ocean Engineering, National Taiwan University, Taipei, Taiwan.
E-mail addresses: d97525011@ntu.edu.tw (N.S.-C. Kao), twhsheu@ntu.edu.tw (T.W.-H. Sheu).

https://doi.org/10.1016/j.compfluid.2018.03.033
0045-7930/© 2018 Elsevier Ltd. All rights reserved.

primitive-variable form ($\underline{u}$, p)

$$-\frac{1}{Re}\nabla^2\underline{u} + (\underline{u}\cdot\nabla)\cdot\underline{u} = -\nabla p + \underline{f} \tag{1}$$

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

In Eq. (1), the dimensionless Reynolds number Re comes out as the direct result of the normalization of equations. The solution of the above elliptic system of partial differential equations for the velocity vector $\underline{u} = \{u, v, w\}$ is sought in a region that is enclosed by the boundary. The condition applied at Γ with an outward normal vector $\underline{n} = \{n_x, n_y, n_z\}$ must satisfy the integral equation given by $\int_\Gamma \underline{u}\cdot\underline{n}d\Gamma = 0$

The mixed finite element formulation is employed to solve Eqs. (1) and (2) because the divergence-free condition is unconditionally satisfied. The resulting finite element equations are, in general, asymmetric, indefinite and are not diagonally dominant. Application of some well known iterative solvers (e.g. BICGSTAB, GMRES [6]) may not yield convergent solutions for high Reynolds number case. To get the unconditionally convergent solution, we will employ the normalization strategy to resolve the convergent problem [7].

3. Streamline upwind finite element model

We denote by $\mathcal{L}_0^2(\Omega)$ the constraint space for the pressure, which consists of square integrable function having zero mean over Ω. In addition, we introduce the space $\mathcal{H}_0^1(\Omega)$, which consists of function and its derivative is square integrable over Ω and vanishes on the boundary. Given the above functional spaces, the solutions for $(\underline{u}, p) \in (\mathcal{H}^1(\Omega)\times\mathcal{H}^1(\Omega))\times\mathcal{L}^2(\Omega)$ are sought from the following variational formulation for (1).

$$\int_\Omega (\underline{u}\cdot\nabla\underline{u})\cdot\underline{w}\,d\Omega + \frac{1}{Re}\int_\Omega \nabla\underline{u} : \nabla\underline{w}\,d\Omega - \int_\Omega p\,\nabla\cdot\underline{w}\,d\Omega = \int_\Omega \underline{f}\cdot\underline{w}\,d\Omega, \tag{3}$$

$$\int_\Omega (\nabla\cdot\underline{u})\,q\,d\Omega = 0 \tag{4}$$

The Eqs. (3) and (4) hold for all weighting functions $\underline{w}\ (\equiv N_i + B_i) \in \mathcal{H}_0^1(\Omega)\times\mathcal{H}_0^1(\Omega)$ and $q \in \mathcal{L}_0^2(\Omega)$.

When employing the mixed finite element formulation to solve Eqs. (1) and (2), the $\mathcal{LBB}$ (or inf-sup) condition is our guideline for choosing the basis space for ($\underline{u}$, p) [8,9] to avoid the even-odd pressure oscillations. The $\mathcal{LBB}$-satisfied element schematic in Fig. 1 is employed in the present study. The primitive velocity vector and pressure are, therefore, approximated by the following tri-quadratic N_j ($j = 1 \sim 27$) and tri-linear M_l ($l = 1 \sim 8$) basis functions, respectively.

$$N_j = \left(\frac{3}{2}\overline{\xi}^2 + \frac{1}{2}\overline{\xi} + 1 + \xi^2 - \xi_j^2\right)\left(\frac{3}{2}\overline{\eta}^2 + \frac{1}{2}\overline{\eta} + 1 + \eta^2 - \eta_j^2\right) \times \left(\frac{3}{2}\overline{\zeta}^2 + \frac{1}{2}\overline{\zeta} + 1 + \zeta^2 - \zeta_j^2\right)$$

$$M_l = \frac{1}{8}(1+\overline{\xi})(1+\overline{\eta})(1+\overline{\zeta})$$

In the above equations, ξ_j, η_j and ζ_j denote the normalized coordinates for j-node in an element and $\overline{\xi} = \xi\xi_j$, $\overline{\eta} = \eta\eta_j$, $\overline{\zeta} = \zeta\zeta_j$.

By substituting the finite element approximation $\underline{u} = \sum_{j=1}^{27}\underline{u}_j N_j$ and $p = \sum_{l=1}^{8} p_l M_l$ into Eqs. (3) and (4), the resulting sparse, asymmetric and indefinite finite element matrix equation $\underline{\underline{A}}\,\underline{x} = \underline{b}$ for the solution vector $\underline{x} = \{u_j, v_j, w_j, p_l\}^{\mathrm{T}}$ is given by

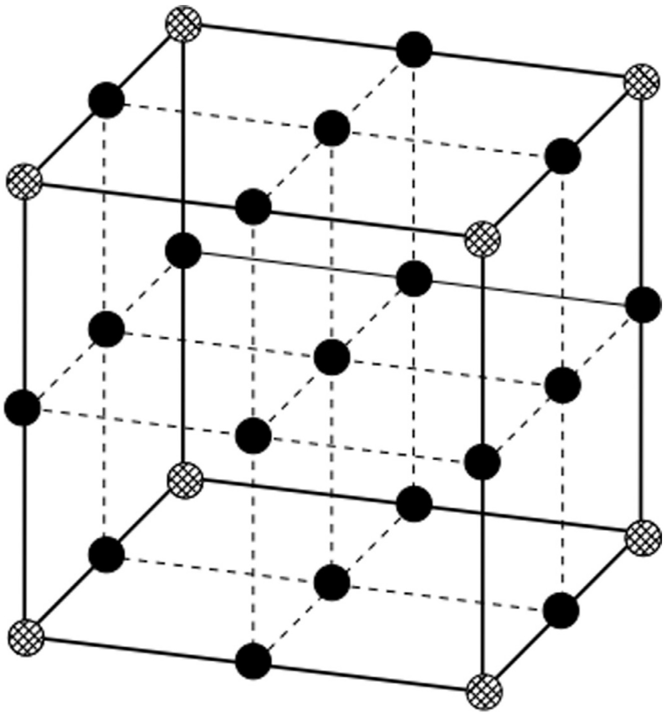

Fig. 1. Schematic of the primitive variable storage in a tri-quadratic element.

$$\underline{\underline{A}} = \int_\Omega \begin{pmatrix} C_{ij} & 0 & 0 & -M_l\dfrac{\partial N_i}{\partial x} \\ 0 & C_{ij} & 0 & -M_l\dfrac{\partial N_i}{\partial y} \\ 0 & 0 & C_{ij} & -M_l\dfrac{\partial N_i}{\partial z} \\ M_l\dfrac{\partial N_j}{\partial x} & M_l\dfrac{\partial N_j}{\partial y} & M_l\dfrac{\partial N_j}{\partial z} & 0 \end{pmatrix} d\Omega,$$

$$\underline{b} = -\int_{\Gamma_{out}} N_i \begin{pmatrix} pn_x - \dfrac{1}{\mathrm{Re}}\dfrac{\partial u_j}{\partial n} \\ pn_y - \dfrac{1}{\mathrm{Re}}\dfrac{\partial v_j}{\partial n} \\ pn_z - \dfrac{1}{\mathrm{Re}}\dfrac{\partial w_j}{\partial n} \\ 0 \end{pmatrix} d\Gamma$$

Finite element simulation of the incompressible Navier–Stokes equations normally encounters non-physical pressure and velocity oscillations. These oscillations can pollute the flow field and deteriorate the solution accuracy. In this study, the problem of pressure oscillations has been circumvented owing to the employed $\mathcal{LBB}$-based basis functions.

Velocity oscillations are due to an incorrect discretization of the nonlinear convection terms which dominated the diffusion terms. To resolve this difficulty, the Petrov–Galerkin finite element model, in which the trial and weighting functions are chosen from different function spaces, is our recommended strategy. The finite element model with minimal wavenumber error will be applied [7].

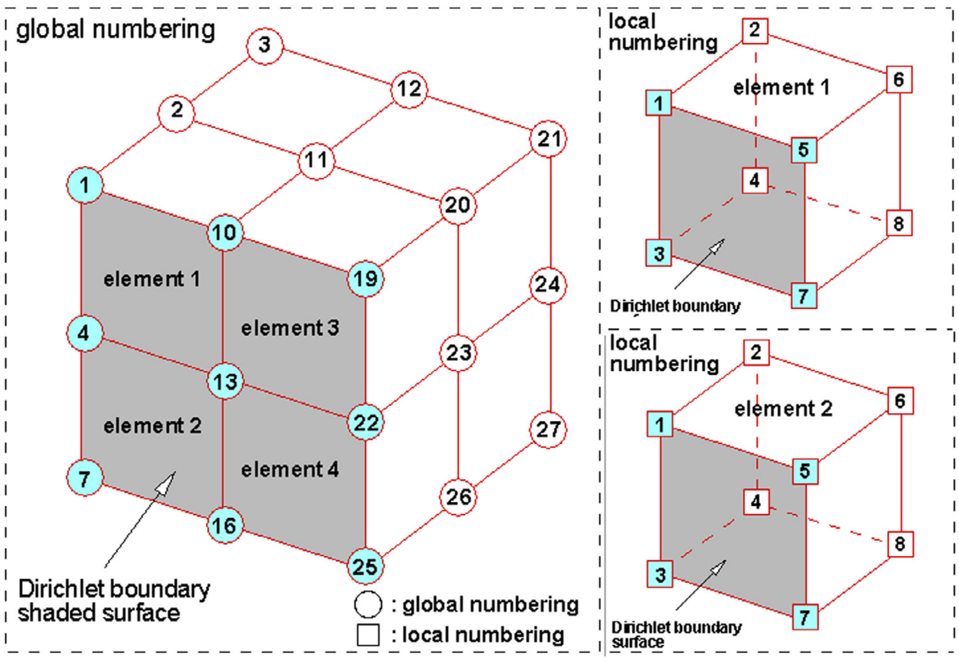

Fig. 2. Illustration of the Dirichlet boundary condition implementation in an elementary matrix.

The component C_{ij} shown in $\underline{\underline{A}}$ is expressed as follows :

$$C_{ij} = (N_i + B_i)(N_m \tilde{u}_m)\frac{\partial N_j}{\partial x_k} + \frac{1}{Re}\frac{\partial N_i}{\partial x_k}\frac{\partial N_j}{\partial x_k}, \tag{5}$$

where $B_i = \tau(N_m\tilde{u}_m)\frac{\partial N_i}{\partial x}$ denotes the biased part. In practice, it is customary to set the value of $\tilde{u}_m$ as a constant in order to linearize the momentum equations. The stabilized parameter τ in three-dimensional problem is expressed as follows:

$$\tau = \frac{\delta_\xi\, u_\xi\, h_\xi + \delta_\eta\, u_\eta\, h_\eta + \delta_\zeta\, u_\zeta\, h_\zeta}{2|u_i\, u_j|}, \tag{6}$$

where $u_{Y_i} = \hat{e}_{Y_i} \cdot \underline{u}$ and $(Y_1, Y_2, Y_3) = (\xi, \eta, \zeta)$. In the above expression, the upwinding coefficients $(\delta_\xi,\ \delta_\eta,\ \delta_\zeta)$ are chosen depending on the nodal classification in the chosen tri-quadratic element [7].

In the classical finite element analysis, all the elementary matrices are calculated and assembled to form a global matrix. This global matrix is then modified by including the Dirichlet boundary condition and is then solved by some suitable solvers. Since the global matrix is sparse, only the non-zero entries can be stored by some specific sparse matrix formats in order to save the computer memory. A discussion of these formats can be found in [6]. However, employment of these formats still suffers from large memory requirements in three-dimensional problem because the significantly increased non-zero entries.

In this finite element flow solver, all the elementary matrices can be only stored in an element level. There is no need to assemble the global matrix and a large amount of computer memory can be reduced. To this end, the elementary matrix associated with element containing the Dirichlet boundary condition must be modified to include the Dirichlet boundary condition.

We denote by $\underline{\underline{A}}_e^{(b)}$ the eth element matrix and the superscript 'b' means that this element contains the Dirichlet boundary node. We also define the variable $S_k^{(e)}$ which represents the times of shared Dirichlet boundary node k in element e. For the sake of simplicity, we take the tri-linear element as an example. In Fig. 2, the shaded region represents the Dirichlet boundary condition. The number shown in the circle and the square symbol stands for the *global* and *local* numbering, respectively. The global corner boundary nodes 1,7,19,25 share only one element $(S_1^{(1)} = S_3^{(2)} = S_5^{(3)} = S_7^{(4)} = 1)$. The middle edge nodes 4,10,16,22 share with two elements $(S_3^{(1)} = S_5^{(1)} = S_1^{(2)} = S_7^{(2)} = S_1^{(3)} = S_7^{(3)} = S_3^{(4)} = S_5^{(4)} = 2)$. The node 13 is shared by four elements $(S_7^{(1)} = S_5^{(2)} = S_3^{(3)} = S_1^{(4)} = 4)$. We denote by ϕ_{bn} the Dirichlet boundary value at the boundary node 'bn'. The modification procedure is stated in Algorithm 1.

Algorithm 1: The modification procedure for $\underline{\underline{A}}_e^{(b)}$.

```
Input :
  $Nel^{(b)}$ ← The number of elements containing the Dirichlet boundary nodes;
  $n_L$ ← The degree of freedom of elementary matrix;
  $e$ ←The ID of element containing the Dirichlet boundary node;
  $S_k^{(e)}$ ← The number of shared boundary node $k$ in the element $e$;
  $\phi_{bn}$ ← The Dirichlet boundary value at the boundary node 'bn';
  $\underline{\underline{A}}_e^{(b)}$ ← The boundary element matrix;
  $\underline{b}$ ← The global right hand side vector;
Output :
  $\underline{\underline{A}}_e^{(b)}$ ← The modified boundary element matrix;
for $e = 1 \rightarrow Nel^{(b)}$ do
    Find the local row $r$ and global row index $i$ for boundary node $k$ in element $e$;
    Set $\underline{\underline{A}}_e^{(b)}|_{rr} = 1.0/S_k^{(e)}$;
    for $j = 1 \rightarrow n_L,\ j \neq r$ do
        $\underline{\underline{A}}_e^{(b)}|_{rj} = 0$
    end
    $\underline{b}(i) = \phi_{bn}$;
end
```

This procedure can be easily extended to tri-quadratic element when solving the steady-state incompressible Navier–Stokes equations using the mixed finite element formulation.

4. Iterative solver on multi-GPUs

4.1. Introduction of CPU/GPU platform

CUDA is a heterogeneous computing model developed to exploit the computing power of Nvidia's GPU based on a scalable programming model on an instruction set architecture. With the advent of CUDA, the parallel computing on GPUs for non-graphic applications becomes available. The CUDA programming model is an extension of the programming language (C/C++ or Fortran). In CUDA model, GPU (device) is a co-processor to the CPU (host). The host solely controls function calls for device memory allocation and memory transfer. The device is responsible for the most time-consuming computation tasks. Since the data in host must be copied to device, the investigated problem size is limited by the memory size in GPU. In order to investigate a larger sized problem and get a better speedup performance, one can utilize the multiple GPU cards. In this case, the communication between different GPU cards may be necessary.

4.2. Multi-GPU-based PCG solver

In the employed mixed finite element formulation, asymmetry and indefiniteness pose a grand challenge [7]. Employment of a Gaussian-elimination-based direct solver with scaling pivoting technique has long been considered as a means of solving this kind of finite element equations. However, the memory demand of direct solver in three-dimensional problem is prohibitive even using the state-of-the-art storage technology. Thus, we have no choice but turn to employing an iterative solver. Among them, a class of Krylov subspace iterative solvers was regarded to be effective to solve the three-dimensional finite element equations. Moreover, it is suitably implemented in parallel on GPU.

To get unconditionally convergent solution, our strategy is to transform the asymmetric and indefinite matrix equations into an equivalent symmetric and positive definite (SPD) counterpart by multiplying its transpose $\underline{\underline{A}}^{\mathrm{T}}$ on the matrix equation $\underline{\underline{A}}$ under investigation, leading to the normal matrix equations $\underline{\underline{\widetilde{A}}}\,\underline{x} = \underline{\widetilde{b}}$ where $\underline{\underline{\widetilde{A}}} = \underline{\underline{A}}^{\mathrm{T}}\underline{\underline{A}}$, $\underline{\widetilde{b}} = \underline{\underline{A}}^{\mathrm{T}}\,\underline{b}$

Since the normal matrix equations become SPD, the computationally effective conjugate-gradient (CG) iterative solver [6] can be applied to get the unconditionally convergent solution. Use of this approach, however, increases the condition number and makes the convergence of CG very slow. A suitable preconditioner can be chosen to accelerate the convergence.

4.3. GPU kernel functions

4.3.1. Vector, precondition operation kernel

For the given vector size *N* and the number of threads (denote by *Nthread*) in each block, we can decide the number of blocks (denote by *Nblock*). Perfectly, *Nthread* should be the multiple of the warp size in order to meet alignment constraint of global memory coalescing. The *Nblock* can be calculated by the formula $Nblock = \frac{N-(Nthread-1)}{Nthread}$ so that all the components of vector will be processed. *N* is, in general, less than the product of *Nblock* and *Nthread*. Note that in order to alleviate the discrepancy of calculation of each block, the last block can be complemented with zeros.

The vector operations consist of the vector addition/substraction and the scalar multiplication of vector. In this study, the Jacobi preconditioner is employed due to its easy parallel implementation. The preconditioned equation $\underline{\underline{M}}\,\underline{z}_j = \underline{r}_j$ is, therefore, very easy to be solved and parallelized. To get a better performance, we can combine all the four-vector operation kernel functions into a single kernel function in Algorithm 2.

Algorithm 2: multi-GPU-based PCG solver for $\underline{\underline{\widetilde{A}}}\,\underline{x} = \underline{\widetilde{b}}$.

```
Input :
$\underline{\underline{A}}$ ← All the elementary matrices ; $\underline{\underline{A}}^{\mathrm{T}}$ ← All the transposed elementary matrices;
$\underline{b}$ ← The global right-hand side vector ; $\underline{\underline{M}}$ ← The Jacobi preconditioner;
Output :
$\underline{x}$ ← The solution vector;
Starting from an initial guess solution $\underline{x}_0$;
Compute the normalized right hand side $\underline{\widetilde{b}} = \underline{\underline{A}}^{\mathrm{T}}\,\underline{b}$;
Compute $\underline{\underline{\widetilde{A}}}\,\underline{x}_0$ // matrix-vector multiplication operation;
Compute the initial residual $\underline{r}_0 = \underline{\widetilde{b}} - \underline{\underline{\widetilde{A}}}\,\underline{x}_0$ // vector operation;
Solve $\underline{\underline{M}}\,\underline{z}_0 = \underline{r}_0$ // preconditioner equation solving;
Set $\underline{p}_0 = \underline{z}_0$;
Compute $\underline{q}_0 = \underline{\underline{\widetilde{A}}}\,\underline{p}_0, \underline{s}_0 = \underline{\underline{\widetilde{A}}}\,\underline{z}_0$;
$\alpha = (\underline{r}_0, \underline{z}_0)/(\underline{\underline{\widetilde{A}}}\,\underline{z}_0, \underline{z}_0), \beta = 0$;
for $j$ = 1, 2, ..., do
    $\underline{p}'_j = \underline{z}'_j + \beta\,\underline{p}'_{j-1}$ // vector operation;
    $\underline{q}'_j = \underline{s}'_{j-1} + \beta\,\underline{q}'_{j-1}$ // vector operation;
    $\underline{x}'_j = \underline{x}'_{j-1} + \alpha\,\underline{p}'_j$ // vector operation;
    $\underline{r}'_j = \underline{r}'_{j-1} - \beta\,\underline{q}'_j$ // vector operation;
    check the convergence;
    Solve $\underline{\underline{M}}\,\underline{z}_j = \underline{r}_j$ // precondition operation;
    Compute $\underline{s}_j = \underline{\underline{\widetilde{A}}}\,\underline{z}_j$ // matrix-vector multiplication operation;
    $\beta = (\underline{z}_j, \underline{r}_j)/(\underline{z}_{j-1}, \underline{r}_{j-1})$ // inner product operation;
    $\alpha = (\underline{z}_j, \underline{r}_j)/\left[(\underline{s}_j, \underline{z}_j) - (\frac{\beta}{\alpha})(\underline{s}_j, \underline{r}_j)\right]$ // inner product operation
end
```

4.3.2. Inner-product operation kernel

The inner-product of the vector in CUDA is a special operation since the reduction operation is invoked. The proposed inner-product kernel function is stated in Algorithm 3. Since the reduction operation will be invoked, we declare a buffer of *SM*-sized shared memory in each block in order to get a better performance. The kernel-1 is implemented in two steps. Firstly, we compute the temporal products of *vec*1 and *vec*2 and store the products in the shared memory. The tree reduction algorithm is then performed in each block to calculate the partial sum of each block. Finally, all the *Nblock* partial sums are accumulated to get the required inner-product.

4.3.3. Element-by-element matrix-vector product kernel

In Algorithm 2, the matrix-vector product represents the most expensive operation. As a result, a good acceleration of this operation will significantly improve the speed performance.

Since the global matrix in this study is never assembled, we decompose the matrix-vector product into a sum of element-level matrix-vector product via the element-by-element (EBE) concept [10]. For the given global matrix $\underline{\underline{A}}$ and the vector $\underline{v}$, the matrix-vector product can be reformulated in element-wise fashion as

$$\underline{\underline{A}}\,\underline{v} = \sum_{e=1}^{Nel} (\underline{\underline{C}}^e)^{\mathrm{T}}\,\underline{\underline{A}}^e\,\underline{\underline{C}}^e\,\underline{v} = \sum_{e=1}^{Nel} (\underline{\underline{C}}^{\,e})^{\mathrm{T}} (\underline{\underline{A}}^e\,\underline{v}^e) \tag{7}$$

where *Nel* is the number of elements. $\underline{\underline{C}}^e$ denotes the transition matrix which represents the mapping between the *local* and *global* node numbering. In Eq. (7), the product of an assembled global matrix and a vector is equivalent to the assembled vector of the elementary matrix-vector product. Kiss et al. [11] also presented the similar concept, but the detailed algorithm is not given therein.

Algorithm 3: Inner-product kernel function (kernel-1).

```
Input :
$Vec1, Vec2, N \leftarrow$ Given vectors and their sizes;
$Nthread \leftarrow$ The number of threads in each block;
$Nblock \leftarrow$ The number of blocks in grid
$(Nblock = (N + (Nthread - 1))/Nthread)$;
Output :
$ps(Nblock) \leftarrow$ partial sum vector;
1: Allocate shared memory vector $V_{SM}(512)$ with double-precision type;
2: $tid \leftarrow$ local thread index in each block;
3: $bid \leftarrow$ global block index in grid;
4: $gid \leftarrow$ global thread index in grid;
5: $V_{SM}(tid) = vec1(gid) * vec2(gid)$; call syncthreads();
6: if $(tid \le 256)\, V_{SM}(tid) = V_{SM}(tid) + V_{SM}(tid + 256)$; call syncthreads();
7: if $(tid \le 128)\, V_{SM}(tid) = V_{SM}(tid) + V_{SM}(tid + 128)$; call syncthreads();
8: if $(tid \le 64)\, V_{SM}(tid) = V_{SM}(tid) + V_{SM}(tid + 64)$; call syncthreads();
9: if $(tid \le 32)\, V_{SM}(tid) = V_{SM}(tid) + V_{SM}(tid + 32)$; call syncthreads();
10: if $(tid \le 16)\, V_{SM}(tid) = V_{SM}(tid) + V_{SM}(tid + 16)$; call syncthreads();
11: if $(tid \le 8)\, V_{SM}(tid) = V_{SM}(tid) + V_{SM}(tid + 8)$; call syncthreads();
12: if $(tid \le 4)\, V_{SM}(tid) = V_{SM}(tid) + V_{SM}(tid + 4)$; call syncthreads();
13: if $(tid \le 2)\, V_{SM}(tid) = V_{SM}(tid) + V_{SM}(tid + 2)$; call syncthreads();
14: if $(tid = 1)\, ps(bid) = V_{SM}(tid) + V_{SM}(tid + 1)$; call syncthreads();
```

Note that the implementation of Eq. (7) on GPU may fail because of the race condition problem. The mesh coloring technique is used to circumvent this problem. All the elements $\mathcal{E}$ are divided into the number of n disjoint subsets such that any two elements in a given subset are not allowed to share the same node. The proposed kernel function of the EBE matrix-vector product for a given subset is stated in Algorithm 4. Some notes are given below:

- All the elementary matrices $\underline{\underline{A}}_e, e \in \mathcal{E}$ are stored as the one-dimensional vector $\underline{A}$ in global memory of GPU. If the elementary matrix is obtained from the element containing the Dirichlet boundary node, the elementary matrix must be modified via Algorithm 1.
- The local degree of freedom n_L of $\underline{\underline{A}}_e, e \in \mathcal{E}$ is equal to 89 due to the chosen $\mathcal{LBB}$ basis functions. In order to meet alignment constraint global memory coalescing, we enlarge the size of n_L from 89 to few multiple warp size (e.g. 96). For n_L numbering from 90 to 96, they are set to have the values of zero.

It is clearly that the kernel-2 can be launched on different GPU cards for different subsets. To launch the kernel-2 on multi-GPU, we copy the elementary matrices of different subsets and global vectors to the corresponding GPU device. As shown in Fig. 3, the kernel-2 is then performed in the all respective GPUs. The partial elementary matrix-vector product stored in the GPU-2 to GPU-n is then copied back to GPU-1 to sum up the required global matrix-vector product.

5. Code verification/validation

The present finite element code is written in CUDA Fortran and compiled with the PGI (The Portland Group, Inc., Lake Oswego, OR, USA) accelerator. Some features of the employed CPU and GPU architectures are listed in Table 1. All the calculations are performed using double-precision arithmetic.

Algorithm 4: Element-by-element matrix-vector product kernel function (kernel-2).

```
Input :
$Nthread \leftarrow$ The number of threads in block;
$Nblock \leftarrow$ The number of blocks in grid
$(Nblock = (N + (Nthread - 1))/Nthread)$;
$\mathcal{E}^{k} \leftarrow$ The $k$th subset of element containing the same color;
$Nel^{(k)} \leftarrow$ The number of elements in $k$th subset $\mathcal{E}^{k}$;
$n_L \leftarrow$ The local degree of freedom; $n_L^E \leftarrow$ The enlarged local degree of freedom;
$n_G \leftarrow$ The global degree of freedom;
$N \leftarrow n_L^E * n_L * Nel^{(k)}$;
$\underline{A}_{N\times 1} \leftarrow$ All the elementary matrices in subset $\mathcal{E}^{k}$;
$\underline{v}_{n_G\times 1} \leftarrow$ The global vector;
$\underline{\underline{C}}^{(e)} \leftarrow$ The $e$th transition matrix, $e \in \mathcal{E}^{k}$;
output :
$\underline{Av}(n_G) \leftarrow$ The elementary matrix-vector product;
1: Allocate the shared memory vector $V_{SM}(n_L^E)$ with double-precision type;
2: $tid \leftarrow$ local thread index in block;
3: $bid \leftarrow$ global block index in grid;
4: $gid \leftarrow$ global thread index in grid;
5: $msize \leftarrow n_L^E * n_L$;
6: if $(gid \le msize * Nel^{(k)})$ then
7:     $icr \leftarrow mod(gid, msize) + int((msize\text{-}mod(gid, msize))/msize) * msize$;
8:     $e \leftarrow int(gid + (msize - 1)/msize)$;
9:     $col \leftarrow int((icr + (n_L^E - 1))/n_L^E)$;
10:    $row \leftarrow mod(gid, n_L^E) + int((n_L^E - mod(gid, n_L^E))/n_L^E)*n_L^E$;
11:    $ptr \leftarrow (e - 1) * msize + (col - 1) * n_L^E + row$;
12:    $V_{SM}(tid) = \underline{A}(ptr) * \underline{v}((e - 1) * n_L^E + row)$; call syncthreads();
13:    if $(tid \le 48)\, V_{SM}(tid) = V_{SM}(tid) + V_{SM}(tid + 48)$; call syncthreads();
14:    if $(tid \le 24)\, V_{SM}(tid) = V_{SM}(tid) + V_{SM}(tid + 24)$; call syncthreads();
15:    if $(tid \le 12)\, V_{SM}(tid) = V_{SM}(tid) + V_{SM}(tid + 12)$; call syncthreads();
16:    if $(tid \le 6)\, V_{SM}(tid) = V_{SM}(tid) + V_{SM}(tid + 6)$; call syncthreads();
17:    if $(tid \le 3)\, V_{SM}(tid) = V_{SM}(tid) + V_{SM}(tid + 3)$; call syncthreads();
18:    if $(tid=1)\underline{Av}(\underline{\underline{C}}^{(e)}(bid)) += V_{SM}(tid) + V_{SM}(tid+1) + V_{SM}(tid+2)$; call syncthreads();
19: end if
```

5.1. Analytic verification

In an unit cube, all the nodal boundary velocities are analytically prescribed by $u = \frac{1}{2}(y^2 + z^2)$, $v = -z$ and $w = y$. The corresponding exact pressure solution can be derived as $p = \frac{1}{2}(y^2 + z^2) + \frac{2}{Re}x$. In Table 2, we tabulate the predicted L_2-error norms and the corresponding spatial rates of convergence based on the solutions obtained at four different number of elements. Good agreement between the exact and the predicted solution is demonstrated. The predicted spatial rate of convergence is also very close to its theoretical rate of convergence.

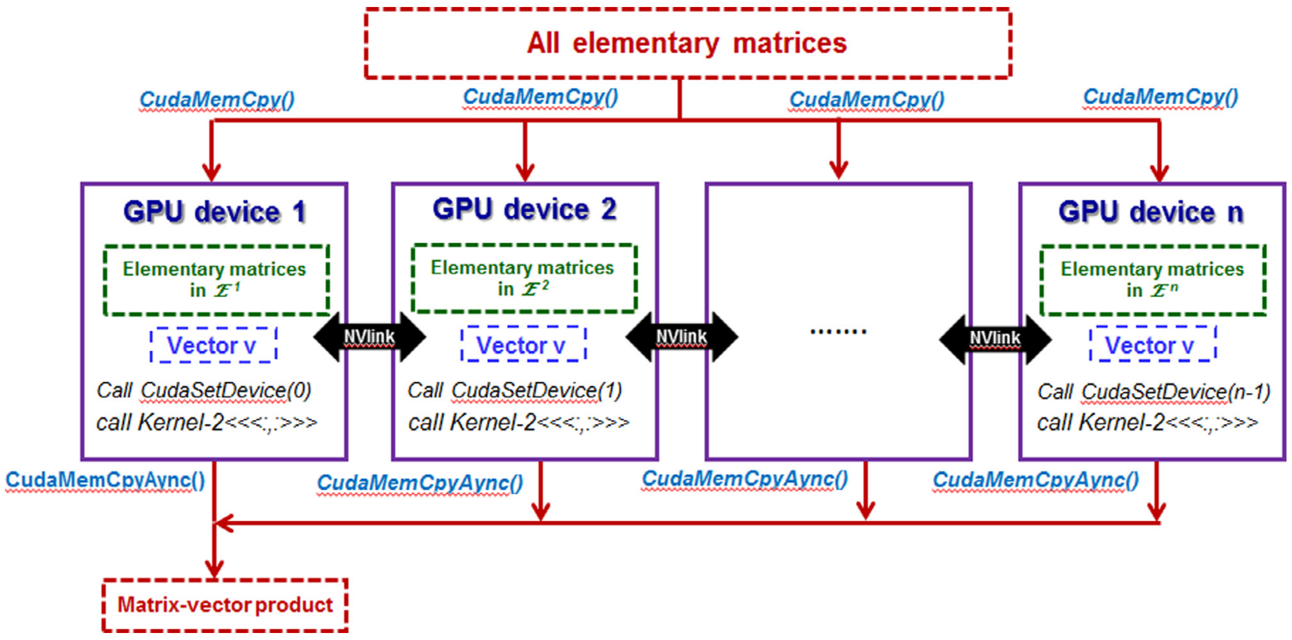

Fig. 3. Implementation of SpMV kernel function on multi-GPUs.

Table 1
Some key specifications of the employed CPU and the GPU architectures.

	CPU	GPU
Processor	Intel E5-2690 V4	Nvidia Pascal P100
Number of cores	14	3584 (SP)
		1792 (DP)
Off-chip memory	1.54 TB	16 GB
Peak flops	37.1 GB/s (DP)	10.6 TFlops/s (SP)
		5.3 TFlops/s (DP)
Memory bandwidth	76.8 GFlops/s	732 GB/s (Nvlink)
IEEE 754 single/double	Yes / yes	Yes / yes

SP : Single precision.
DP : Double precision.

5.2. Three-dimensional lid-driven cavity flow problem

The benchmark three-dimensional lid-driven cavity flow problem schematic in Fig. 4(a) is investigated due to its simple geometry, embedded rich flow physics and easy boundary condition implementation. The predicted and reference [12] mid-plane velocity profiles $u(x, 0.5, 0.5)$ and $w(0.5, y, 0.5)$ are plotted in Fig. 4(b) and (c). The simulated good agreement demonstrates that the developed finite element GPU code can accurately predict the incompressible flow behavior.

6. Speedup performance

In this section, a performance comparison between the developed CPU and GPU codes is discussed with the goal of demonstrating the benefit of running the code on GPUs. The lid-driven cavity problem with different element sizes and Reynolds numbers are performed to assess the performance. The total computing time T_{run}^{GPU}, including the data preparation, elementary matrix calculation, preconditioner calculation and matrix equation solver, is measured by running the GPU code on 1-, 2- and 4-GPU cards. The T_{run}^{CPU} is also measured by running the equivalent OpenMP CPU code with 14-cores (28-threads).

For the sake of comparison, the unit time $T_{unit}(\equiv\frac{T_{run}\times 10^6}{N_{iter}\times N_{elem}})$ defined in [13] is calculated. The N_{elem} and N_{iter} represent the number of elements and the times of nonlinear iteration, respectively. The unit time T_{unit} at all studied cases are tabulated in Tables 3 and 4 and the speedup ratios are obtained by comparing the unit time of GPU code against those obtained with the OpenMP CPU code. It is clearly seen from Tables 3 and 4 that some study cases are not available because its memory requirements exceed the built-in memory size in GPU. This is why we proposed the flow solver on multiple GPU cards. Our solver can solve the problem whose total degree of freedom is more than ten millions. Moreover, the predicted speedup ratio is increased with respect to the element size. The results shown in Tables 3 and 4 justify the choice of the proposed finite element flow solver for the solution of the three-dimensional incompressible Navier–Stokes equations.

7. Concluding remarks

In this study, the $\mathcal{LBB}$-based Petrov–Galerkin finite element model was used to solve the steady-state incompressible Navier–Stokes equations. This finite element model minimizes the error of numerical wavenumber for the convection term so as to ef-

Table 2
The predicted L_2 error norms for the analytical verification problem in Section 5.1.

N_{elem}	*u*-velocity		*v*-velocity		*w*-velocity		*p*-pressure	
	L_2-norms	R.O.C.	L_2-norms	R.O.C.	L_2-norms	R.O.C.	L_2-norms	R.O.C.
10^3	6.80E-4	–	3.59E-3	–	3.65E-3	–	2.75E-3	–
20^3	1.08E-4	2.65	4.99E-4	2.84	5.71E-4	2.67	7.04E-4	1.96
30^3	4.37E-5	2.17	1.39E-4	3.07	1.53E-4	3.16	3.18E-4	1.91
40^3	2.42E-5	2.12	5.56E-5	3.30	5.85E-5	3.46	1.94E-4	1.78

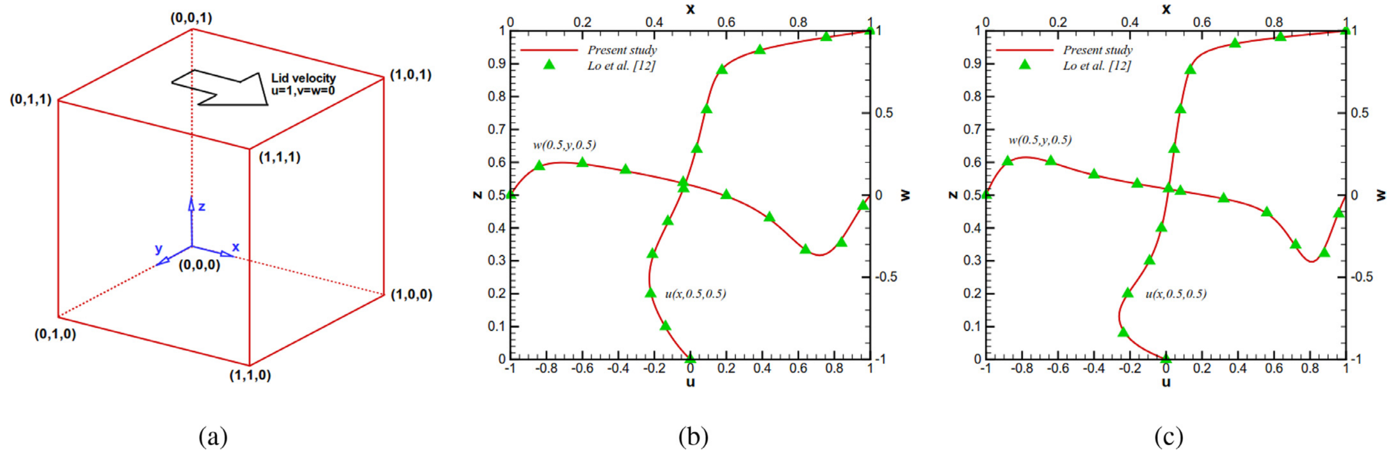

Fig. 4. Schematic and comparison of the predicted velocity profiles of the three-dimensional lid-driven cavity flow problem. (a) Problem schematic; (b) Velocity profiles at Re = 400; (c) velocity profiles at Re = 1000.

Table 3
Comparison of the computing time (microseconds) and speedup ratios for the three-dimensional lid-driven cavity flow problem investigated at $Re = 400$. N_{DOF} represents the number of total degree of freedom. The symbol "–" indicates that this calculation is not available.

N_{elem}	N_{DOF}	T_{unit} (microseconds)				Speedup ratios		
		1-CPU(A)	1-GPU(B)	2-GPUs(C)	4-GPUs(D)	(A)/(B)	(A)/(C)	(A)/(D)
40^3	1,663,244	124627.0	2206.0	1877.7	1662.7	56.4 ×	66.3 ×	74.9 ×
50^3	3,223,554	154976.9	2424.7	2030.0	1790.2	63.9 ×	76.3 ×	86.5 ×
64^3	6,714,692	178111.8	–	2339.9	2093.0	–	76.1 ×	85.0 ×
75^3	10,767,829	259914.0	–	–	2256.9	–	–	115.1 ×

Table 4
Comparison of the computing time (microseconds) and speedup ratios for the three-dimensional lid-driven cavity flow problem investigated at $Re = 1000$. N_{DOF} represents the number of total degree of freedom. The symbol "–" indicates that this calculation is not available.

N_{elem}	N_{DOF}	T_{unit} (microseconds)				Speedup ratios		
		1-CPU(A)	1-GPU(B)	2-GPUs(C)	4-GPUs(D)	(A)/(B)	(A)/(C)	(A)/(D)
40^3	1,663,244	265879.9	3924.4	3215.1	2956.2	67.7 ×	82.6 ×	89.9 ×
50^3	3,223,554	389859.5	4581.8	3727.7	3421.7	85.0 ×	104.5 ×	113.9 ×
64^3	6,714,692	526862.8	–	4500.1	4158.7	–	117.0 ×	126.6 ×
75^3	10,767,829	636523.9	–	–	4734.0	–	–	134.4 ×

fectively suppress the non-physical velocity oscillations at high Reynolds number. To get unconditionally convergent solutions from the nearly ill-conditioned finite element equations, we solved the equivalent normalization equations and applied the preconditioner to accelerate the convergence [7]. To implement the PCG solver on multiple GPU cards, the mesh coloring and EBE techniques were adopted. From the numerical results, it was observed that this solver is accurate, reliable and can be used to solve very large-sized problems. Moreover, a considerable gain in speedup ratios is obtained when compared to the CPU calculation using 4 cores. The results from this study demonstrate that one may solve the three-dimensional flow equations on GPUs, reducing the computing time and thus allowing the proposed solver to be applied to investigate practical flow problems.

Acknowledgments

This work is supported by MOST (Ministry of Science and Technology) under the grants MOST-105-2811-E-002-005- and by the CASTS (Center for Advanced Study in theoretical Sciences) at National Taiwan University. Thanks are also given to Ingrasys Technology Inc for making the current GPU computation possible.

References

[1] Brenner SC, Seott LR. The mathematical theory of the finite element method. Springer Verlag New York, 1994.
[2] Tezduyar TE, Sameh A. Parallel finite element computations in fluid mechanics. Comput Methods Appl Mech Eng 2006;195:1872–81.
[3] Kuo FA, Smith MR, Hsieh CW, Chou CY, Wu JS. GPU acceleration for general conservation equations and its application to several engineering problems. Comput Fluids 2011;47:147–54.
[4] Posey S. Consider for GPU acceleration of parallel CFD. Procedia Eng 2013;61:388–91.
[5] Nvidia. CUDA C Programming Guide 8.0; 2016.
[6] Sadd Y. Iterative methods for sparse linear system. SIAM Press 2003.
[7] Kao NeoSC, Sheu TonyWH, Tsai SF. On a wavenumber optimized streamline upwinding method for solving steady incompressible navier-stokes equations. Numer Heat Transf Part B 2015;67:1–25.
[8] Babuška I. Error bounds for finite element methods. Numer Math 1971;16:322–33.
[9] Brezzi F, Douglas J. Stabilized mixed methods for the stokes problems. Numer Math 1988;53:225–35.
[10] Hughes TJR, Levit I, Wingt J. Element-by-element implicit algorithm for heat conducting. J Eng Mech 1983;109(2):576–89.
[11] Kiss I, Gyimóthy S, Basics Z, Pávó J. Parallel realization of the element-by-element FEM technique by CUDA. IEEE Trans Magn 2012;48(2):507–10.
[12] 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.
[13] Xia Y, Lou J, Luo H, Edwards J, Mueller F. Openacc acceleration of an unstructured CFD solver based on a reconstructed discontinous galerkin method for compressible flow. Int J Numer Methods Fluids 2015;78:123–39.