An efficient GPU Parallelization for arbitrary collocated polyhedral finite volume grids and its application to incompressible fluid flows

Shashank Jaiswal, Rajesh Reddy, Raja Banerjee
Dept. of Mechanical and Aerospace Engineering
Indian Institute of Technology Hyderabad
Hyderabad, India
{*me12b1033, me10p006, rajabanerjee*}*@iith.ac.in*

Shingo Sato, Daisuke Komagata, Makoto Ando, Jun Okada
Numerical Simulation Research Dept.
Steel Research Laboratory
JFE Steel Corporation
Kawasaki, Japan
{*shing-sato, d-komagata, m-ando, j-okada*}*@jfe-steel.co.jp*

***Abstract*—This paper presents GPU parallelization for a computational fluid dynamics solver which works on a mesh consisting of polyhedral cells, where each cell has an arbitrary number of faces and each face has an arbitrary number of vertices. The parallelization is achieved using NVIDIAs compute unified device architecture (CUDA). The developed code specifically targets performance improvement on NVIDIA-Tesla accelerator GPUs. The implementation has been carried out in a general purpose open-source CFD framework namely OpenFOAM which is capable of solving arbitrary flow problems involving complex geometries with polyhedral unstructured grids. The present work considers incompressible flow simulations, where solving pressure Poisson equation is the most computationally expensive step. The Poisson equation is solved using conjugate gradient method preconditioned by algebraic multigrid method. This part of the solver is outsourced by OpenFOAM to GPU. The GPU pressure Poisson solver acceleration is determined with respect to OpenFOAM serial version (single cpu core) and MPI parallelized version (8 cpu cores). The GPU solver acceleration was tested by simulating a standard benchmark test case called lid driven cavity flow for different grid sizes. The current GPU based solver has shown a speedup of approximately 16× when compared to single cpu core and 3.3 × when compared to OpenFOAM MPI version using 8 cpu cores, for a grid size of 5 million cells.**

***Keywords*-GPU computing; Computational Fluid Dynamics; preconditioned conjugate gradient methods; OpenFOAM framework; CUDA;**

I. Introduction

Graphics Processing Units (GPU) have emerged as programmable devices instead of just hardwired graphics accelerators. Due to higher computing to cost (C/C) ratio, GPUs have now become cost-effective scalable co-processors and hence an integral part of high performance computing (HPC). The massive number of GPU threads and high memory bandwidth is readily employed to accelerate data computations in numerous fields such as Computational Fluid Dynamics (CFD) [1], [2], Linear Algebra [3], [4], Computer Vision, Computational Biology, etc.

However, use of GPUs for computational tasks is by no means a new idea. On early GPUs having low *C/C* ratio, particle based fluid flow computation methods such as Lattice-Boltzmann, have obtained significant speedups [5]. Various other methods of solving PDEs such as Finite-Difference Finite-Time (FDFT) [6], Finite Element (FEM) [7], Finite Volume (FVM) [8]–[10], Discrete Galerkin (DG) [11], etc have been extensively studied for tackling numerous *continuum mechanics* problems on GPUs.

Computational Fluid Dynamics, a branch of continuum mechanics, emphasizes on simulating fluid flows, including compressible, incompressible, and multiphase, as well as flows involving further physics such as chemical reactions. On complex geometries, flow computations demand very high computational resources and hence the simulations are usually carried on HPC systems. Within the continuum region (see [12]–[14]), Navier-Stokes-Fourier (NSF) equations govern the fluid flows.

In case of incompressible fluids, NSF equations are used to solve the velocity-pressure field. A three step process (see Section III) i.e *Predictor*, *Pressure Poisson*, and *Corrector*, is used to compute the flow solution [12]. Solving pressure Poisson equations consumes most of the computing time in Navier-Stokes simulations [15]. More than an order of magnitude of speedup in computation time can be achieved by accelerating solvers for this step.

A. The Collocated Polyhedral grid problem

In CFD, the spatial domain is divided into a number of contiguous control volumes or cells. On meshes of polyhedral cells, each with an arbitrary number of faces, each face has an arbitrary number of vertices. In general, the number of neighboring cells can vary from cell to cell. The cell connectivity is such that a cell face is either internal and intersects two cells only, or comprises part of an external boundary and belongs to a single cell only [16]–[20]. The mesh is not necessarily aligned with the co-ordinate system as well.

In such a collocated grid system, the values of a physical property (Example: pressure), is defined at each cell-center p and its value is dependent on the values of its neighborhood $N(p)$. Value at p is given by the sum of the values at cell-centers in $N(p)$ scaled by predefined constants. The constants depend on the distance between the cell-centers as shown in Fig 1 (adapted from [18], [19]). The value $v(p)$ at p is given by

$$v(p) = \sum_{q \epsilon N(p)} a_{pq} v(p), \qquad \sum_{q \epsilon N(p)} a_{pq} < 1 \tag{1}$$

where a_{pq} is the scaling factor.

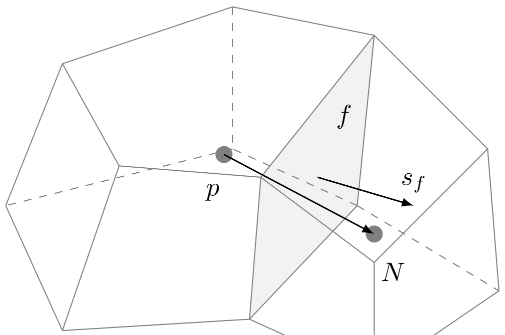


Figure 1: Finite Volume Discretization for collocated mesh with polyhedral cells, each with arbitrary number of faces, and each face with arbitrary number of vertices.

The objective is to calculate $v(p)$ for all cell-centers (p) in the computational domain. The problem reduces to solving large set of sparse linear equations of the form

$$[A]_{mxn} \{x\}_{nx1} = \{b\}_{mx1} \tag{2}$$

where $[A]_{mxn}$ is the matrix of dimensions (m x n) which has to be inverted and multiplied with the known source vector $\{b\}_{mx1}$ to obtain the solution $\{x\}_{nx1}$. The typical size of the matrix considered in the current work is in the order of millions which makes it a computational intensive and time consuming task.

B. CUDA

Compute Unified Device Architecture (CUDA) presents an application programming interface that gives direct access to the GPU's virtual instruction set and parallel computational elements, for the execution of compute kernels. An efficient sparse linear iterative solver for system of equations such as Eq. 2, can be created using CUDA kernels. The pressure Poisson equation, as stated previously, is the most time consuming part of the entire solver. Solution of the pressure Poisson equation requires solving linear system of equations. The involved sparse matrix computations are performed on GPU by implementing CUDA kernels. Here the sparse matrix $[A]$ is generated for a initial value flow problem for a considered computational domain using OpenFOAM.

C. Open-Source CFD Framework: OpenFOAM

The OpenFOAM® [16] (Open Field Operation and Manipulation) CFD Toolbox is an open source CFD library which provides an interface for solving complex fluid flow problems on arbitrary *collocated polyhedral* grids. Primarily written in C++ [21], it makes use of C++ templates, function and operator overloading [16], so that the top-level syntax of the code as close as possible to conventional mathematical notation for tensors and partial differential equations. Numerous discretization, interpolation, and reconstruction schemes along with iterative solvers such as Gauss-Seidel are integrated together in the form of OpenFOAM framework.

OpenFOAM supports MPI parallelization on host using the well known method of domain decomposition. However, there is no inbuilt support for *dihybrid (GPU/CPU) solvers* which can leverage the capabilities of both host (CPU) and the device (GPU). The present work focuses on integrating GPU computing with OpenFOAM.

D. Related Work

The efficient iterative solvers on an arbitrary polyhedral grid using GPUs for OpenFOAM framework have been less studied in the past. Although there are several works in which GPUs are used to solve problems on structured/unstructured grids [8], [22]–[26], the usability of these solvers is limited to problems with simple geometry.

The multigrid methods, also known to be convergence accelerators, have received moderate attention in the past, but mostly on structured grid [8], [27] and few on unstructured grids [1], [24], [25]. Considerable work has been done on parallelizing Algebraic Multi-grid (AMG) solvers [9], [10], [26] which includes implementing various parallel coarsening and smoothing techniques on parallel computers. But, again the usability of these solvers is limited to problems with simple geometry.

The present work was done as a part of the vision of integrating a family of GPU parallelized iterative solvers with a family of Open-Source CFD solvers (provided as a part of OpenFOAM). The developed solver is intended to solve arbitrary fluid flow problems (incompressible, compressible, or multiphase) on arbitrary 2D/3D complex geometries represented by polyhedral unstructured grids. Our implementation works for arbitrary 2D/3D structured/unstructured grids. Since, the CUDA and OpenFOAM framework are freely available and widely used, the implementations are of interest to large number of CFD/OpenFOAM users.

In this work, the contribution is three folds: (a) efficient integration of GPU based parallel iterative solvers on arbitrary collocated polyhedral meshes, (b) GPU aware algorithm for improved performance, and (c) Benchmarking.

E. Paper Organization

A brief overview of GPU architecture and CUDA is given in section II. In section III, the OpenFOAM computation model is discussed. Section IV describes the GPU implementation details and proposed improvements. Benchmark test cases & results are discussed in section V. Concluding remarks are presented in section VI.

II. GPU Architecture and CUDA

In this section a brief overview of GPU architecture and NVIDIA's CUDA programming model are presented with an emphasis on some of the important performance optimization aspects for efficient GPU implementation.

A GPU consists of a set of streaming multiprocessors (SM) and each SM contains a number of simple processor cores called CUDA cores. In CUDA, computational grid is divided into number of blocks, and each block consists of several threads. A thread is the basic set of instructions that is to be executed in a computer code. An SM is designed to execute several hundred of these threads concurrently, using their Single Instruction Multiple Threads (SIMT) architecture i.e each thread executes same set of instructions (called kernels [28]) but on different data.

The GPU memory is divided into three layers: (1) global memory, largest and with highest latency and accessible by CPU and all threads across all blocks, (2) shared memory or on-chip memory shared by all threads in a thread block and (3) register memory with lowest latency, and is unique to each thread. Each thread has access to all three layers of memory space.

Through the *global memory*, the matrix $[A]$ (and the associated data) to be inverted (Eq. 2) is transferred from CPU to GPU. Access to *global memory* data requires hundreds of clock cycles, and the unnecessary data transfer to and from global memory should be avoided. For data transfer from host to device's global memory wherever necessary, a library called Nvidia-Thrust is used. Nvidia-Thrust is a collection of C++ GPU-accelerated algorithms (for example: sort, scan, transform, reductions) and data-structures.

In order to leverage the capabilities of underlying GPU, optimization becomes crucial in addition to effective parallelization of the code. The optimizations include maximizing SM utilization, memory and instruction throughput [28]. Increasing occupancy, and coalesced memory access significantly improves the performance of the applications. Scattered memory accesses by threads in a warp (group of 32 threads) leads to transfer of unused data between global memory and the cache. Further details on CUDA optimization can be obtained in [28].

III. OpenFOAM Computing Model

Various fluid flows such as compressible, incompressible, and multiphase, as well as flows involving further physics such as chemical reactions, can be described by systems of linked partial differential equations of the form [16]

$$\frac{\partial \rho \mathbf{Q}}{\partial t} + \nabla.(\rho U \otimes Q) - \nabla.(\rho\phi(\nabla \mathbf{Q})) = S \tag{3}$$

where U is the fluid velocity, ρ its density, and $\mathbf{Q}$ is any tensor-valued property of the flow, such as temperature. These equations involve time derivatives ($\partial \rho \mathbf{Q}/\partial t$), convective terms ($\nabla.(\rho U \otimes \mathbf{Q})$), diffusive terms ($\nabla.(\rho\phi(\nabla \mathbf{Q}))$), and source terms S.

A. The Incompressible Navier-Stokes

Let $\mathbf{Q}$ be a first order tensor-valued property: fluid-velocity ($\mathbf{Q} = \{1, U\}$). With the assumption of constant fluid density, Eqn. 3 can then be simplified to

$$\frac{\partial U}{\partial t} + \nabla.(U \otimes U) - \nabla.(2\nu \mathbf{D})) = -\frac{1}{\rho}\nabla p \tag{4}$$

where

$$D = \frac{1}{2}(\nabla U + \nabla U^T) \tag{5}$$

with continuity (mass conservation) equation as

$$\nabla.U = 0 \tag{6}$$

B. Representation of PDEs

In OpenFOAM, the top-level syntax of the code is as close as possible to conventional mathematical notation for tensors and partial differential equations. Equation 4 can be represented as

```
    fvMatrixVector Ueqn
    (
        fvm::ddt(U)
       +fvm::div(phi, U)
        -fvm::laplacian(nu, U)
    );
```

$$\text{solve(Ueqn == } -\frac{1}{rho}*\text{fvc::grad(p));} \tag{7}$$

The top-level syntax of the code as close as possible to governing differential equation. This is one of the major advantages of OpenFOAM, and a key motivation for developing an efficient GPU parallelized version of the solvers. Foam enables users to develop reliable and efficient computational fluid dynamics codes for tackling different fluid flow situations with ease. The reader is referred to [16], [18], [29] for details of some of the user implemented solvers in OpenFOAM.

C. Solution to Incompressible Navier-Stokes

The governing equations have been discretized using finite volume method. In FVM, the computational domain M is divided into a set of discrete volumes $\triangle V_i$ such that $\sum \triangle V_i = M$ and $\triangle V_i \cap \triangle V_j = \varnothing$ for i≠j. The fluid-flow equations are then volume integrated over each individual finite volume $\triangle V_i$. Generally, a three step predictor-pressure poisson-corrector algorithm (For example: PISO [30], SIMPLE [12]) is used for solving incompressible Navier-Stokes Eqns.4–6. In the PISO algorithm, the pressure-velocity methods are decoupled and solved iteratively as follows [16], [17]

1) *Predictor:* An initial guess for the pressure field is made (or the previous time step pressure solution) and the momentum equation is solved to a predefined tolerance to give an approximate velocity field (Eq. 7).
2) *Pressure Poisson:* The pressure Poisson equation is then formulated with the divergence of the partial velocity flux as a source term and solved to give a new estimate of the pressure field (Eq. 8).
3) *Corrector:* The corrected pressure field is used in an explicit correction to the velocity field.

The second step i.e pressure Poisson equation can be mathematically expressed as [12]

$$\nabla^2 p = \frac{\rho}{\triangle t}\nabla . U \tag{8}$$

where $\triangle t$ is the time step, while other symbols have their standard meanings as defined previously. Finite volume discretization of the equation is given by

$$\sum_f \nabla p_f . s_f = \frac{\rho}{\triangle t}\sum_f F_f \tag{9}$$

where ∇p_f is gradient of pressure at each face of the cell, s_f is the surface area of respective face of the cell and F_f is the flux through each face of the cell. Using finite-difference approximations, a set of difference equations can be obtained that can be described in matrix form as specified by Eq. 2.

The generally used numerical methods to solve such a system of linear equation are direct methods such as Gaussian elimination, LU factorization etc., and iterative methods such as Jacobi method, Gauss-Seidel method and Conjugate gradient method [12]. For large systems the direct methods are inefficient whereas iterative methods are more efficient. The iterative methods obtain the solution $\{x\}$ without actually inverting the matrix. These methods are efficient in terms of both memory and time.

In this work, OpenFOAM uses a highly efficient method called as algebraic preconditioned conjugate gradient method for solving the pressure Poisson equation. An efficient parallelized version of this method is implemented here using CUDA. The implemented code can be called by OpenFOAM. The rest of the steps in the solution procedure are taken care by OpenFOAM. The current implementation is highly modular & robust, and both GPU and Serial version of code can be run side-by-side without any recompilation. The end user can still easily switch between various iterative methods (For example: Gauss-Seidel, PCG, PBiCG, etc) as usually done in OpenFOAM i.e using *fvSolution* dict via *runTimeSelection* mechanism [16].

IV. Implementation

In a distributed computing model, only non-zero matrix entries & their corresponding matrix position are stored in the memory. There are multiple ways in which the matrix entries and the corresponding positions can be retrieved either directly or indirectly from memory. Various storage formats have their own advantages and disadvantages. For some of them, the binary operation cost can be costlier than others. OpenFOAM implements its own sparse matrix storage format which they call as *lduMatrix*. Consider the following example of a simple matrix.

$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{bmatrix} \tag{10}$$

In *lduMatrix* format, the given matrix is stored in a 5-array form, which can be expressed in zero-based indexing format as:

$$d : \{1, 6, 11, 16\} \tag{11}$$
$$l : \{5, 9, 10, 13, 14, 15\} \tag{12}$$
$$u : \{2, 3, 4, 7, 8, 12\} \tag{13}$$
$$lA : \{0, 0, 0, 1, 1, 2\} \tag{14}$$
$$uA : \{1, 2, 2, 3, 3, 3\} \tag{15}$$

where d corresponds to diagonal entries, u corresponds to upper-triangular entries, l corresponds to lower-triangular entries of matrix (10). For entries in u, lA corresponds to row index and uA corresponds to column index. And for entries in l, uA corresponds to row index and lA corresponds to column index. In CFD, while d corresponds to cell values, l & u corresponds to face-values. Size of d is equivalent to number of cells i.e *nCells*, and size of l is equal to number of faces i.e *nFaces*.

A. CUDA Implementation

The Conjugate Gradient (CG) method is a very promising iterative method for solving sparse systems of linear equations. Preconditioning is used for improving the condition number of a matrix. Suppose that M is a symmetric, positive-definite matrix that approximates A, but is easier to invert. In linear algebra, a symmetric $m \times m$ real matrix M is said to be positive definite if the scalar $z^T M z$ is positive for every non zero column vector z of m real numbers. The system $\mathbf{Ax} = \mathbf{b}$ can be solved indirectly by

$$M^{-1}Ax = M^{-1}b \tag{16}$$

If the condition number of $M^{-1}A$ is less than A then the number of iterations required for solving Eq. 16 will be lesser than the original problem. However, the computational overhead of applying the preconditioner must not cancel out the benefit of fewer iterations. To be a valid preconditioner for CG method, the matrix M^{-1} should be symmetric and positive definite. For a matrix of size n, conjugate gradient method converges in atmost n steps. Usually preconditioning is necessary to ensure fast convergence. A general preconditioned conjugate gradient algorithm can be obtained at [31], [32].

Current implementation of PCG algorithm uses *LDU* matrix storage format for computation. The following GPU kernels are used for implementing different steps of PCG method. It is to be noted that before kernel execution, all the matrix entries and other data are to be copied from host memory to device memory. After the completion of GPU execution, the solution array is to be copied back from device memory to host memory inorder to execute the rest of the solution algorithm on CPU.

1) Amul $([A]_{n\times n}\{x\}_{n\times 1})$*:* Two kernels are used for implementing the sparse matrix-vector product. One of the kernels creates as many threads as number of cells in the computation domain and calculates the product for diagonal entries. The other kernel creates as many threads as number of off diagonal entries in upper triangular half of matrix $[A]$, and calculates the product of off-diagonal entries with the corresponding column-vector entries.

2) Dot $([x]_{1\times n}\{y\}_{n\times 1})$*:* The kernel takes the starting index of each vector and length, creates as many thread as the number of entries in the vector and calculates the value at each array index.

3) Sum $(\sum [x]_{n\times 1})$*:* The kernel creates p blocks, each block with q threads such that $p * q <= n$. Each block computes the sum of elements assigned to it and stores it in an output vector of length p. This output vector becomes the input for the next sum kernel call, and the procedure is recursively followed till the length of output vector equals unity.

4) sumA $(\sum_{j=0}^{n} [A]_{ij})$*:* It creates as many threads as number of cells, and calculates the sum of matrix entries present in a given row.

5) precondition $([M]^{-1}_{n\times n}\{r\}_{n\times 1})$*:* Preconditions the residual vector using matrix M.

B. Algebraic Multi-grid Preconditioning

The standard *V-cycle* algebraic multi-grid (AMG) algorithm has been used for preconditioning the residual vector in our present study. The reader is referred to [26], [33], [34] for details of AMG method. Multigrid methods employ two interdependent processes namely *Smoothing* and *Coarse-grid correction*. The former involves the application of a simple iterative method like Jacobi which reduces/smooths high frequency errors. The latter involves transfer of information to a coarser grid through *restriction* followed by solving a coarse-grid system of equations and then transferring information back to the fine grid through *interpolation*. The V-cycle algorithm combines these Smoothing and Coarse-grid correction steps. While the first half of the V-cycle recursively proceeds from finer to coarser grids, the latter half recurses from coarser to finer grids. Current AMG preconditioner uses two iterations of V-cycle algorithm. Jacobi iterative method has been used for smoothening step. The hierarchy of grids are created using the standard coarsening algorithm [33], which partitions the points into two disjoint sets. The strength of connection matrix is computed using the standard symmetric measure i.e an off-diagonal connection A_{ij} is strong iff $|A_{ij}| >= \theta * \sqrt{|A_{ii}|.|A_{jj}|}$ and $\theta \in (0, 5]$.

V. Benchmark Tests and Results

Transient incompressible lid-driven cavity flow problem is solved using currently developed solver. The MPI-parallelized CPU version of solver is termed as *CPU*, and GPU-parallelized solver using custom Cuda kernels (employing PCG method) is called *GPU/Cuda*. The given lid-driven cavity flow problem studies the transfer of momentum from a medium at rest to a wall moving at a constant velocity. It is assumed that the temperature within the system remains constant i.e the energy equation is not solved.

A. Hardware Configuration

Two sets of hardwares are used for evaluating the performance of current GPU/Cuda solver. The first set tests improvements on lower end GPU architecture Quadro, while the second tests higher end GPU architecture Tesla-K20m. All the simulations are done with double precision floating point values.

1) Configuration A: Serial implementations are run on Intel Xeon CPU E5-2620 v3 2.40 GHz. The operating system used is 64-bit Ubuntu 12.04 LTS. Parallel implementations of solver are run on the same machine with NVIDIA Quadro K600 GPU with CUDA driver 7.0 and CUDA runtime 6.5. The GPU has 192 CUDA cores, 1GB device memory. The solver has been written in C++ and is compiled using g++ 4.8 and nvcc 6.5.12 compiler with third level optimization flag.

2) Configuration B: Serial implementations are run on Intel Xeon CPU E5-2650 v2 2.60 GHz. The operating system used is 64-bit Centos 6.0. Parallel implementations of solver are run on the same machine with NVIDIA Tesla

K20m GPU with CUDA driver 7.0 and CUDA runtime 6.5. The GPU has 2496 CUDA cores, 5GB device memory. The solver has been written in C++ and is compiled using g++ 4.8 and nvcc 6.5.12 compiler with third level optimization flag.

B. *Transient Lid-Driven Cavity Flow Problem*

Standard benchmark test case of 2D incompressible Lid-driven cavity flow is considered for checking the efficiency of GPU-parallelized OpenFOAM code. The unsteady problem is solved on a square computational domain as shown in Fig 2. Sample mesh is shown in Fig. 3. The PISO algorithm (Pressure Implicit with Splitting of Operator) proposed by Issa [30] is used here for solving the flow governing equations. The parameters used in the simulation are presented in Table I. Dirichlet boundary condition for velocities has been used at all the surfaces (u = 1; v = 0 for lid and u = 0; v = 0 for all other surfaces) and Neumann boundary condition for pressure at all surfaces ($\frac{\partial p}{\partial n}$ = 0). Time step used in the simulations is obtained based on Courant-Friedrichs-Lewy (CFL) condition. For an explicit time integration scheme it can be defined as

$$\frac{u\Delta t}{\Delta x} + \frac{v\Delta t}{\Delta y} \leq 1$$

where u and v are x-component and y-component of velocity in a computational cell. For a case with grid size 2250 $\times$ 2250 (approximately 5 million computational cells), $\Delta x = \Delta y = 4.5 \times 10^{-5}$. With initial conditions u=1 and v=0, maximum limit for Δt can be roughly estimated as 4.5×10^{-5}. However for an implicit scheme like PISO this time step constraint can be relaxed. Nevertheless a large time step size may result in numerical inaccuracy and instability (depending on flow conditions). Hence here time step size $\Delta t = 5 \times 10^{-8}$ is chosen.

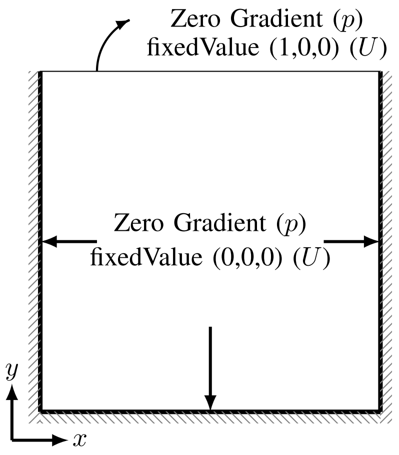


Figure 2: Numerical setup for 2D lid-driven cavity flow.

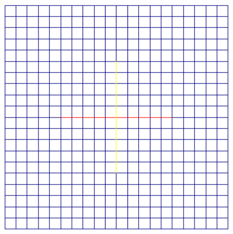


Figure 3: Sample mesh for 2D lid driven cavity flow.

Domain (m)	Time step (s)	Kinematic Viscosity (m^{2}s^{-1})
0.1 x 0.1	5 x 10^{-8}	0.01

Table I: Initial configuration parameters of lid-driven cavity flow test case.

1) *Validation:* Before proceeding to solver performance, it is required to compare the solution obtained from *GPU-parallelized code* against the standard *Serial code* solution. We compare the normalized pressure and velocity distribution along the centerline of computational domain i.e Y = 0.05. Fig. 4 shows the comparison between the solutions obtained using GPU and serial version of incompressible laminar transient solver *icoFoam*. An excellent agreement is obtained between the two solutions as evident from the figure.

2) *Comparison of Solver Performance:* The simulations are carried out for the mentioned case up to t = 5×10^{-4}s on five different grid sizes 5×10^4, 1×10^5, 5×10^5, 1×10^6, and 5×10^6. Speed up obtained with GPU based solver is presented in Table II. As can be seen from the table, the acceleration due to GPU parallelization increases with increase in the size of computational grid. This is because with larger grid sets, the time required by the GPU to perform the computations becomes larger than the time required for data transfer between the host and the device. Hence, the non-computational overhead decreases which results in increase of the computational efficiency of the GPU. However, as number of cells increase beyond a certain limit i.e 1×10^6, almost all of SMs present get occupied and the parallel computation overhead becomes larger. So, the increase in cell count does not increase the computation speedup. On a mesh consisting of 5 million cells, the current implementation achieves a speed up of ∼16x on Tesla-K20m, when compared with the serial version of code. OpenFOAM solvers can be run on CPU using MPI communication. Present solver achieves a speedup of ∼4x on Tesla-K20m, when compared with MPI-enabled OpenFOAM code running on 8 processors. Even on a low end easily accessible GPU processor like Nvidia Quadro-K600, the current implementation has achieved a speed up

Number of Cells	**Total Solve Time (s)**			**Overall Speed up**	
(Approximate)	CPU/1P	CPU/8P	GPU/Cuda	1P vs GPU	8P vs GPU
5 x 10^4	64^*	-	87^*	0.73	-
1 x 10^5	187^*	-	155^*	1.2	-
5 x 10^5	2450^*	403^*	653^*	3.75	0.61
$^+$1 x 10^6	5131^+	830^+	318^+	16.13	2.61
$^+$5 x 10^6	26075^+	5561^+	1672^+	15.59	3.33

*Intel Xeon(R) CPU E5-2620 v3 @ 2.40GHz Quadro K600
$^+$Intel Xeon(R) CPU E5-2650 v3 @ 2.60GHz Tesla K20m

Table II: Performance of GPU based solver for 2D lid-driven cavity flow using PISO algorithm. The Pressure Poisson step is only run on GPU in the above set of simulations. Work units represent the total simulation time for first 100 steps.

Number of Cells	**Total Time (s)**	**Pressure Poisson Time (s)**		**Current Speed up**
(Approximate)		CPU/1P	GPU/Cuda	1P vs GPU
1 x 10^6	5131	5026^+	163^+	30.83
5 x 10^6	26075	25305^+	842^+	30.05

$^+$Intel Xeon(R) CPU E5-2650 v3 @ 2.60GHz Tesla K20m

Table III: Pressure Poisson solve time comparison for 2D lid-driven cavity flow using double correction PISO algorithm (Tested by running the case for first 100 steps).

of ∼4x, when tested with half million computational cells and compared against Serial Code.

The real speedup obtained with GPU can be witnessed by comparing the time taken for pressure Poisson step alone. This is due to the fact that the algorithm steps other than pressure Poisson solver are carried out on CPU in serial fashion in the current work. Therefore exclusive pressure Poisson solver time comparison has been presented in Table III. The pressure Poisson algorithm running on GPU (Tesla-K20) is ∼30 times faster than the serial version of Pressure Poisson.

VI. Conclusion and Future Work

A GPU based solver is developed to solve the pressure Poisson equation, which is the most time consuming part of the solution algorithm of incompressible Navier-Stokes equations. The developed code effectively parallelizes the algebraic multigrid preconditioned conjugate gradient method. The implemented solver works for any arbitrary geometry with polyhedral cells and can be called by OpenFOAM CFD tool framework instead of its default pressure Poisson solvers. The current methodology is well suited for Tesla and Quadro GPU architectures and it is believed that it can give similar performance improvement for other GPUs as well. The GPU based solver performance was tested by simulating standard benchmark test case called lid driven cavity flow for different grid sizes. The efficiency of GPU parallelization increased with increase in computational grid size. The current GPU based solver has shown a speedup of approximately 16 when compared to single cpu core and 3.3 when compared to OpenFOAM MPI version using 8 cpu cores, for a grid size of 5 million cells. Even on a low end easily accessible GPU processor like Nvidia Quadro-K600, the current implementation has achieved a speed up of ∼4x against serial code, when tested with half million computational cells.

Future perspective of improvements include additional GPU parallelization of predictor and corrector steps in the Navier-Stokes solution algorithm. Also extending the implementation to multi-GPU multi-CPU architectures is another interesting direction for future work.

Acknowledgment

The research leading to these results has received funding from JFE-Steels grant ME/2015-16/028/JFESC/RB/0190. The authors would also like to thank Dr. Nishanth Dongari, Kumar Mayank, and Sarang Holay for their invaluable inputs.

References

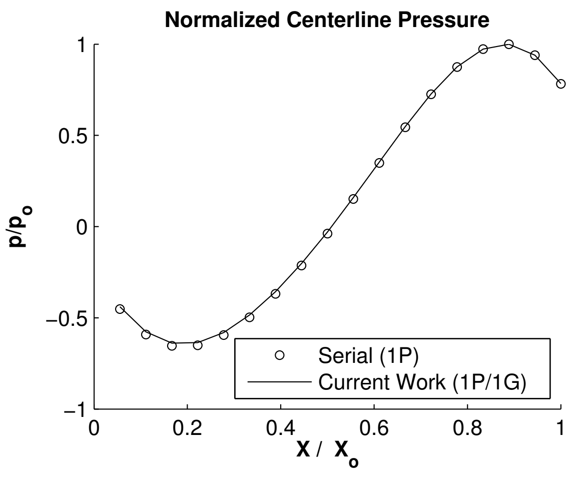
[1] M. Adams and J. Demmel, "Parallel multigrid solver for 3d unstructured finite element problems," in *Supercomputing, ACM/IEEE 1999 Conference*. IEEE, 1999, pp. 27–27.

[2] A. Vose, B. Mitchell, and J. Levesque, "Tri-hybrid computational fluid dynamics on DOE's Cray XK7, Titan." Online, May 2014, Cray User Group.

[3] *AmgX Reference Manual*, Nvidia Corporation, 2014, Release API Ver. 2.

[4] J. Kruger and R. Westermann, "Linear algebra operators for gpu implementation of numerical algorithms," *ACM Transactions on Graphics (TOG)*, vol. 22, no. 3, pp. 908–916, 2003.

[5] W. Li, X. Wei, and A. Kaufman, "Implementing lattice boltzmann computation on graphics hardware," *Visual Computing*, vol. 19, pp. 444–456, 2003.



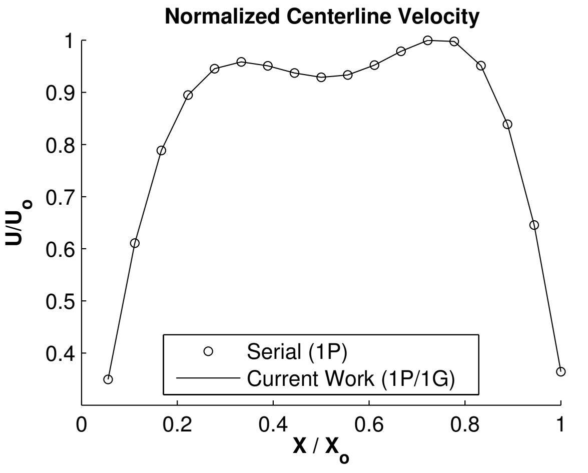


Figure 4: Comparison of normalized centerline pressure and velocity distributions obtained using GPU/Code and Serial-openFOAM code for lid driven cavity flow problem.

[6] S. Krakiwsky, L. Turner, and M. Okoniewski, "Acceleration of finite-difference time-domain (FDTD) using graphics processor units (gpu)," in *2004 IEEE MTT-S International Microwave Symposium Digest*, vol. 2, 2004, pp. 1033–1036.

[7] D. Göddeke, R. Strzodka, and S. Turek, "Accelerating double precision FEM simulations with gpus," in *Proceedings of ASIM*, 2005.

[8] J. Bolz, I. Farmer, E. Grinspun, and P. Schröoder, "Sparse matrix solvers on the gpu: conjugate gradients and multigrid," *ACM Transactions on Graphics (TOG)*, vol. 22, no. 3, pp. 917–924, 2003.

[9] Z. Li, Y. Saad, and M. Sosonkina, "parms: a parallel version of the algebraic recursive multilevel solver," *Numerical linear algebra with applications*, vol. 10, no. 5-6, pp. 485–509, 2003.

[10] U. M. Yang, *Parallel algebraic multigrid methods with high performance preconditioner*. Springer, 2006, available: http://books.google.co.in/books?id=HkFq69DCFIAC.

[11] A. Klöckner, T. Warburton, J. Bridge, and J. Hesthaven, "Nodal discontinuous galerkin methods on graphics processors," *Journal of Computational Physics*, vol. 228, pp. 7863–7882, 2009.

[12] S. V. Patankar, *Numerical heat transfer and fluid flow*. Taylor & Francis, 1980.

[13] R. H. Pletcher, J. C. Tannehill, and D. Anderson, *Computational fluid mechanics and heat transfer*. CRC Press, 2012, Available: http://books.google.co.in/books?id=Cv4IERczJ4oC.

[14] G. A. Bird, *Molecular gas dynamics and the direct simulation of gas flows*. Clarendon, 1994.

[15] G. P. Williams, "Numerical integration of the three-dimensional navierstokes equations for incompressible flow," *Journal of Fluid Mechanics*, vol. 37, no. 4, pp. 727–750, 1969.

[16] H. G. Weller, G. Tabor, H. Jasak, and C. Fureby, "A tensorial approach to computational continuum mechanics using object-oriented techniques," *Computers in physics*, vol. 12, no. 6, pp. 620–631, 1998.

[17] H. Jasak, "Error analysis and estimation for the finite volume method with applications to fluid flows," Ph.D. dissertation, Imperial College of Science, Technology and Medicine, London, June 1996.

[18] C. J. Greenshields, H. G. Weller, L. Gasparini, and J. M. Reese, "Implementation of semi-discrete, non-staggered central schemes in a colocated, polyhedral, finite volume framework, for high-speed viscous flows," *International journal for numerical methods in fluids*, vol. 63, no. 1, pp. 1–21, 2010.

[19] *OpenFOAM Programmer's Guide*, OpenFOAM Foundation, 2014, Release Version 2.3.1.

[20] *OpenFOAM User Guide*, OpenFOAM Foundation, 2014, Release Version 2.3.1.

[21] B. Stroustrup, *The C++ programming language*. Pearson Education India, 1986.

[22] A. Corrigan, F. F. Camelli, R. Löhner, and J. Wallin, "Running unstructured grid-based cfd solvers on modern graphics hardware," *International Journal for Numerical Methods in Fluids*, vol. 66, no. 2, pp. 221–229, 2011.

[23] J. Waltz, "Performance of a three-dimensional unstructured mesh compressible flow solver on nvidia fermi-class graphics processing unit hardware," *International Journal for Numerical Methods in Fluids*, 2012.

[24] P. R. Brune, M. G. Knepley, and L. R. Scott, "Unstructured geometric multigrid in two and three dimensions on complex and graded meshes," in *CoRR*, 2011, vol. abs/1104.0261.

[25] J. Sebastian, N. Sivadasan, and R. Banerjee, "Gpu accelerated three dimensional unstructured geometric multigrid solver," in *IEEE HPCS*. IEEE, 2014.

[26] K. Ravitej, N. Sivadasan, V. Sharma, and R. Banerjee, “Parallel AMG solver for three dimensional unstructured grids using gpu,” in *IEEE HiPC*. IEEE, 2014.

[27] C. Feng, S. Shu, J. Xu, and C.-S. Zhang, “Numerical study of geometric multigrid methods on cpugpu heterogeneous computers,” *arXiv preprint arXiv: 1208.4247*, 2012.

[28] *CUDA Programming Guide*, Nvidia Corporation, 2012, Release API Ver. 6.

[29] S. Jaiswal and N. Dongari, “Implementation of knudsen layer effects in open source cfd solver for effective modeling of microscale gas flows,” in *Proceedings of 1st International ISHMT-ASTFE and 23rd National Heat and Mass Transfer conference*. Kerala, India: ISHMT-ASTFE, Dec 2015.

[30] R. Isaa, “Solution of the implicitly discretized fluid-flow equations by operator splitting,” *Journal of Computational Physics*, vol. 62, pp. 40–65, 1986.

[31] J. R. Shewchuk, “An introduction to the conjugate gradient method without the agonizing pain,” 1994.

[32] C. Pflaum, “A multigrid conjugate gradient method,” *Applied Numerical Mathematics*, vol. 58, no. 12, pp. 1803–1817, 2008.

[33] R. D. Falgout, “An introduction to algebraic multigrid,” *Comput Sci Eng*, vol. 8, p. 24, 2006.

[34] G. Haase, M. Liebmann, C. C. Douglas, and G. Plank, “A parallel algebraic multigrid solver on graphics processing units,” in *High performance computing and applications*. Springer, 2010, pp. 38–47.