Three-dimensional LBM simulations of buoyancy-driven flow using Graphics processing units

Prasanna R. Redapangu and Kirti Chandra Sahu
Department of Chemical Engineering
Indian Institute of Technology Hyderabad
Andhra Pradesh, India
rprasannarani@gmail.com
ksahu@iith.ac.in

***Abstract*—Three-dimensional simulations of buoyancy-driven flow of two immiscible liquids are performed using lattice Boltzmann method (LBM) implemented on a graphics processing unit (GPU). Graphics processing unit is a new paradigm for computing fluid flows and has become more popular in the recent years. It is a powerful and convenient to use. LBM, which is an excellent alternative technique for fluid flow simulation, when implemented on GPUs gives a very high computational speed-up. Our present GPU based LBM solver gives a speed-up 25 times corresponding CPU based code.**

***Index Terms*— graphics processing unit, lattice Boltzmann method, buoyancy-driven flow.**

I. Introduction

Simulating immiscible fluids is computationally very expensive using the traditional solving methods of fluid flow problems. The solution of pressure-Poisson equation demands large number of iterations and is the most time consuming step. Also, the interfacial dynamics and the phase segregation in multiphase flows need very high domains to track the interface between the fluids efficiently. In this regard, the lattice Boltzmann method has become a promising technique in the past two decades for fluid flow simulation. The method is easy to implement and reduces the computational effort. Our LBM algorithm runs six times faster than the corresponding Navier-Stokes solver. The implementation of LBM algorithm on a graphics processing unit increases the computational speed tremendously. It helps the user to solve many complex computational problems in a more efficient way than on a CPU. GPU has become more popular in the recent years and the implementation of LBM on GPU is easier and effective. With the use of GPU, even a very high domain three-dimensional simulation can be performed miraculously faster. Our present GPU based LBM solver gives 25 times speed up over the corresponding CPU based code. The GPU is designed in such a way that it yields very high computation power. Its high performance computing and low energy consumption have made it more beneficial for use in various applications.

II. GPU Architecture

Graphics processing unit is a powerful tool for performing parallel computations. It is a new paradigm for computing fluid flow problems. It can be thought as a massively parallel computer that is capable of executing instructions, simultaneously on a large number of arithmetic units. The architecture of GPU is quite different to that of a CPU so as to result in significant computational speed-up. It is designed with more transistors that are dedicated to computation and less resource dedicated to data caching and flow control compared to that of a CPU. The parallel processing in GPU is designed using multiple threads that execute instructions in a program. Each thread is unique and will be assigned to a unique element of data to be processed. The instructions for a GPU are written in a “kernel” which is similar to a function in the C programming. When a kernel is executed on a GPU, each thread executes the statements in that kernel and map to different elements of data. The threads are organized into ‘blocks’ and the blocks are organized to a ‘grid’ (see figure 1). The threads within a block share data through some ‘shared memory’ and synchronize their execution to coordinate memory accesses. The number of threads per block and the number of blocks depend on the data size to be processed. CPU executes the main program and the GPU can be utilized by launching the kernels from the main program. Therefore the CPU and GPU act simultaneously.

The memory spaces (RAM) of the CPU and GPU are separate. The GPU has multiple memory spaces; global memory, local memory, shared memory, constant memory and texture memory. Global memory is the largest memory

space on the GPU. Each thread contains local memory to store the variables declared in the kernel. Blocks have the shared memory and the threads within the block have access to this memory. Detailed description of the memory spaces of GPU can be found in [1, 2].

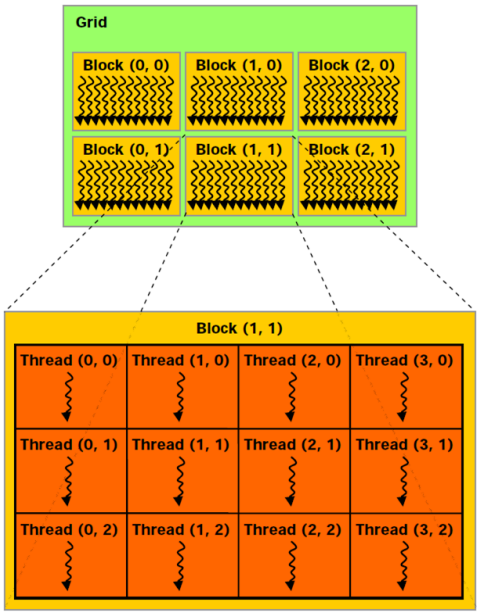

Fig. 1: Thread Hierarchy of the GPU [1]

III. GPU PROGRAMMING

The structure of a program for a GPU is different to that of a CPU program. In a CPU code the instructions given by a program are executed by a single CPU thread whereas in a GPU code the instructions are executed in parallel by a batch of threads. The GPU code eliminates the 'for' loops written in a CPU code and allows the code to be executed in parallel. Other than the usual programming languages like C/C++ or FORTRAN, GPU algorithms are built on special programming languages that can provide parallel computing platform. We use CUDA (Compute Unified Device Architecture), a parallel programming model developed by Nvidia Corporation to implement on GPUs. OpenCL, HLCL, Cg etc. are some other GPU programming languages. CUDA is available as CUDA C/C++ and CUDA Fortran depending on the compilers the programmers wish to use. The high level languages are used with CUDA extensions to express parallelism, data locality and thread cooperation. CUDA's programming model assumes that the CUDA threads execute on a physically separate 'device' (GPU) that operates as a coprocessor to the 'host' (CPU). That means when the kernels execute on a GPU rest of the program executes on CPU. The model assumes that both the 'host' and the 'device' maintain their own RAM, referred to as 'host memory' and 'device memory', respectively. Therefore, the program manages the global, constant and texture memory spaces visible to kernels through calls to the CUDA runtime. This includes device memory allocation and deallocation, as well as data transfer between the host and the device memory. The algorithm written in CUDA and implemented on GPU should be optimized carefully so as to get maximum performance. Minimum memory copies from host-to-device and device-to-host and the optimal block size and grid size would give higher throughput.

IV. PREVIOUS WORKS

Several researchers have implemented LBM on GPUs. Li et al. [3] implemented single phase LBM on GPU with Cg and OpenGL as programming model on NVIDIA GeForce FX 5900 Ultra GPU. They achieved 8-15 times speed up than the CPU counterpart. Tolke [4] implemented two-phase LBM on GPU using CUDA for simulation of flow through porous media. They achieved 10 times speed up over the corresponding CPU program. Peng et al. [5] implemented three-dimensional lattice Boltzmann method on a GPU using CUDA and observed 8.67 times speed-up. These speed-up differences are caused due to various properties like the CPU compiler, hardware and also the type of CPU core. We have implemented LBM on GPU to simulate various flow problems using Nvidia Tesla 1060 and Nvidia Tesla 2070. Sahu and Vanka[6] implemented two-phase lattice Boltzmann method on GPU to simulate the buoyancy-driven flow of two immiscible liquids. Redapangu et al. [7] studied the pressure-driven displacement flow of two liquids using LBM on GPU. Our GPU based LBM code is 25 times faster than its CPU counterpart.

In the present work, we implemented two-phase lattice Boltzmann method on graphics processing units to perform the three-dimensional simulations of buoyancy-driven flow of two immiscible liquids. We use the new Nvidia Tesla Kepler K10, which is the fastest and most efficient high performance processor released by Nvidia most recently. The Kepler compute architecture gives three times higher performance than the Fermi compute architecture. The Tesla Kepler GPU computing accelerators make hybrid computing dramatically easier and applicable to a broader set of computing applications. Tesla K10 accelerator board features two Kepler GPUs each of 8GB onboard memory. Thus two GPU based programs can be run at a time. All our GPU simulations are based on the single precision calculations. However, please note that as all the steps in LBM are of explicit in nature, thus the error does not accumulate like in case of implicit code. We compared our results with those of double-precision calculations and found that the difference in the residual occurs only in the 4^{th} decimal place, but the speed-up of the code decreases to 8 times.

V. LBM SIMULATION OF BUOYANCY-DRIVEN FLOW

Buoyancy-driven flow is the lock-exchange problem that refers to the interpenetration of two immiscible liquids which

are initially separated by a partition and are suddenly allowed to mix under the action of gravity. This phenomenon plays an important role in the design of chemical and petroleum engineering processes and various atmospheric sciences. The two-phase lattice Boltzmann method proposed by He and co-workers [8] is used to solve the problem. Two-dimensional simulations of the problem were previously carried out by Sahu and Vanka [6] to study the effects of density differentials and Redapangu et al. [9] to study the effects of viscosity differentials in an inclined channel. The present study reports the three-dimensional simulations of buoyancy-driven flow of two immiscible liquids. With the use of GPUs, we could simulate the flow using very high grid that can are necessary for the three-dimensional instabilities to appear.

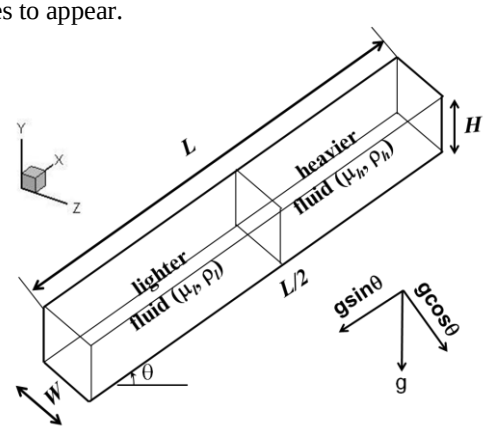

Fig. 2: Initial configuration of the problem.

We consider a three-dimensional rectangular coordinate system (*x, y, z*) to model the flow dynamics. Initially, a lighter fluid of density ρ_l and viscosity μ_l occupies the bottom part and the heavier fluid of density ρ_h and viscosity μ_h occupies the upper part of the channel as shown in figure 2. The inlet and the outlet of the channel are located at x = 0 and L, respectively. The rigid and impermeable walls of the channel are located at y = 0, H and z = 0, W. Here we considered W = H and the aspect ratio of the channel L/H is 32. θ is the angle of inclination measured with the horizontal. The three components of gravity g_x(= gsinθ), g_y(= gcosθ) and g_z(= 0) act in the negative axial, negative transverse and the azhimuthal directions, respectively. The grid size considered here is (32×64)×64×64.

The flow is a balance between inertial and the gravity forces, we define characteristic velocity as $V = \sqrt{gH}$. Various dimensionless parameters describing the flow are the viscosity ratio ($m = \mu_h / \mu_l$), which characterizes the viscosity ratio; the density differential is characterized by Atwood number ($At = (\rho_h - \rho_l)/(\rho_h + \rho_l)$); Reynolds number ($\mathrm{Re} = H\sqrt{gH}/\upsilon_l$), magnitude of surface tension (κ). The gravity is chosen such that $\sqrt{gH} = 0.08$.

We use the three-dimensional-fifteen-velocity (D3Q15) lattice model to simulate the flow. LBM uses index distribution function (*f*) and pressure distribution function (*g*), the evolution equations of which are given below:

$$f_\alpha(\mathbf{x}+\mathbf{e}_\alpha\delta t, t+\delta t) - f_\alpha(\mathbf{x},t) = -\frac{f_\alpha(\mathbf{x},t) - f_\alpha^{eq}(\mathbf{x},t)}{\tau} - \frac{2\tau-1}{2\tau}\frac{(\mathbf{e}_\alpha - \mathbf{u}).\nabla\psi(\phi)}{c_s^2}\Gamma_\alpha(\mathbf{u})\delta t$$

..... [1]

$$g_\alpha(\mathbf{x}+\mathbf{e}_\alpha\delta t, t+\delta t) - g_\alpha(\mathbf{x},t) = -\frac{g_\alpha(\mathbf{x},t) - g_\alpha^{eq}(\mathbf{x},t)}{\tau} + \frac{2\tau-1}{2\tau}(\mathbf{e}_\alpha - \mathbf{u}).\left[\Gamma_\alpha(\mathbf{u})(\mathbf{F_s}+\mathbf{G}) - (\Gamma_\alpha(\mathbf{u}) - \Gamma_\alpha(0))\nabla\psi(\rho)\right]\delta t$$

..... [2]

Here **u** = (*u*, *v*, *w*) represents the three-dimensional velocity field; *u*,*v* and *w* denote the velocity components in the axial, transverse and azhimuthal directions, respectively; δt is the time step; τ is the single relaxation time. The kinematic viscosity (υ) is related to the relaxation time as $\upsilon = (\tau - 1/2)\delta t c_s^2$. f_α^{eq} and g_α^{eq} are equilibrium distribution functions, Γ_α(**u**), which is a function of macroscopic velocity. $\mathbf{F_s}$ and **G** are the surface tension and the gravity forces respectively. The readers are referred to Sahu and Vanka [6] for detailed formulation.

The index function (ϕ), pressure (p) and the velocity filed (**u**) are calculated using:

$$\phi = \sum f_\alpha$$

$$p = \sum g_\alpha - \frac{1}{2}\mathbf{u}.\nabla(p - c_s^2\rho)\delta t$$

$$\rho\mathbf{u}c_s^2 = \sum \mathbf{e}_\alpha g_\alpha + \frac{c_s^2}{2}(\mathbf{F_s}+\mathbf{G})\delta t$$

..... [3]

The fluid density and kinematic viscosity are calculated from the index function as:

$$\rho(\phi) = \rho_l + \frac{\phi - \phi_l}{\phi_h - \phi_l}(\rho_h - \rho_l)$$

$$\upsilon(\phi) = \upsilon_l + \frac{\phi - \phi_l}{\phi_h - \phi_l}(\upsilon_h - \upsilon_l)$$

..... [4]

Here, υ_l and υ_h are the kinematic viscosities of lighter and the heavier fluids, respectively. ϕ_l and ϕ_h are the minimum and the maximum values of the index function. In the present study, ϕ_l and ϕ_h are given values of 0.02381 and 0.2508, respectively.

VI. Programming structure

The programming structure of LBM on GPU cuda and on CPU differs significantly. The subroutine in fortran/C become kernel in cuda. When these kernels are called, they are executed N times in parallel by N different cuda threads. But in regular C functions, the execution is only once. We show below a simple algorithm of a calculation done using CPU fortran and that in CUDA. Below we show the structure of code in fortran and in cuda for the calculation of the fluid density in eqn.(4).

This subroutine in fortran can be written as:

```
do i =1,nx
   do j =1,ny
      do k = 1, nz
         rho(i, j, k) = rhol + ((phi(i, j, k) - phil)/(phih - phil)) *
                              (rhoh - rhol)
      end do
   end do
end do
```

When the subroutine is called, the density is estimated for all the grid points. In GPU code, we use one-dimensional array by defining ijk = i + (j-1) × nx + (k-1) × nx × ny, where nx, ny and nz are the number of grids in the x, y and z-directions, respectively. Thus the total number of grids, 'ntotal' becomes nx × ny× nz. The above subroutine in cuda kernel can be now written as follows:

```
__global__ void
findvar_kernel (float *rho_d, float rhol, float rhoh, float
*phi_d, float phil, float phih)

{
  int ijk;
  float phit;
  int tx  = threadIdx.x + blockIdx.x * blockDim.x;
  ijk = tx + 1;

  phit  = phi_d[ijk];

 rho_d[ijk] = rhol + ((phit - phil) /(phih - phil)) * (rhoh - rhol);

  __syncthreads();

}
```

Kernel is defined using global declaration specifier. The variables in the kernel are declared using the syntax kernelname( ). Here the variable stated as rho_d correspond to rho on the device i.e., GPU. Each variable is allocated memory on host (CPU) and device in the main program before calling the kernel. This is implemented as:

```
rho      = (float *)malloc(memsize_all);
```

```
float *rho_d;  cudaMalloc((void**) &rho_d,     memsize_all);
```

where 'memsize_all' is the dynamic memory allocated on the host. Enough memory is allocated to reach the largest index. 'cudaMalloc' allocates memory on the device.

Each thread is given a unique thread ID which can be accessible within the kernel through threadIDx. ThreadIDx is a three-component vector so that either a one-dimensional or two-dimensional or three-dimensional thread block can be used. Although we use one-dimensional thread block, in general, one could have three-dimensional thread block. The total number of threads is equal to the number of threads per block times the number of blocks. For more details, the reader is referred to the cuda manual [1].

The multiple blocks are organized into grids. The dimension of the grid is specified in the main program. The grid and the block dimension for the kernel execution are specified when calling the kernel using the <<<…, …>>> syntax. This part in the main is shown below:

```
dim3 grid(ntotal/bx);
 findvar_kernel<<<grid, block>>> (rho_d, rhol, rhoh, phi_d,
                                  phil, phih);

cutilCheckMsg("Kernel execution failed");
```

Here bx is the block size. The execution of each kernel is checked using 'cutilCheckMsg'.

Once the calculation is done on the device, the results are copied from device to host in the main. This is done as:

```
cudaMemcpy(rho,  rho_d,  memsize_all,
cudaMemcpyDeviceToHost);
```

After performing the simulation the allocated memory is deallocated which can be done using the following syntax:

```
free(rho);
```

The simulation time in cuda can be recorded using the syntax given below.

```
CUDA_SAFE_CALL( cudaThreadSynchronize() );
CUT_SAFE_CALL(cutStopTimer(timerApp));
printf("GPU Kernel Time: %f (seconds)\n",
      cutGetTimerValue(timerApp) / 1000.0);
CUT_SAFE_CALL(cutDeleteTimer(timerApp));
```

This gives the simulation time using GPU. We have compared this GPU time with that of the time taken to run the corresponding C/fortran LBM CPU code for same number of iterations (say 100 iterations). The comparison of the time taken by both codes reveals that GPU code is 25 times faster than our CPU based LBM code.

VII. RESULTS

The spatio-temporal evolution of the iso-surface of ϕ at the interface is shown in figure 3 at different times. The parameters that are used to generate the contours are Re = 500, m =1, At = 0.05, κ = 0 and the angle of inclination, θ = 60° measured from the horizontal. As the time progresses, the heavier fluid occupying the upper part of the channel move into the region of the lighter fluid at the bottom part of the channel. This downward motion is caused by the axial component of gravity, $g\sin\theta$. The transverse component of gravity, $g\cos\theta$ induces segregating effect. We see that the interface between the fluids become unstable and the instabilities develop in the form of small structure.

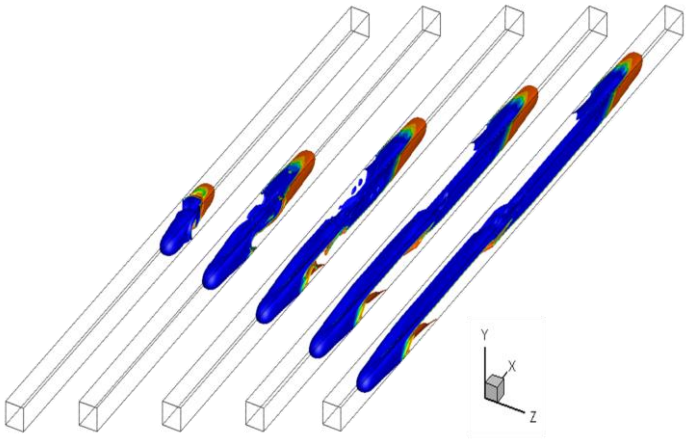

Fig. 3: The evolution of iso-surface of ϕ at the interface at times t = 20, 40, 60, 80 and 100 (from left to right). The parameters are Re = 100, m = 1, At = 0.05, κ = 0 and θ = 60°.

The X-Y cross-section of the three-dimensional channel is taken and the contours obtained in this plane are shown in figure 4. For the same parameters, two-dimensional simulation is carried out and the result obtained in shown in fig. 5. Therefore, comparing fig. 4 and 5, we see that the two-dimensional simulation gives the contours with more small scale structures which are not seen in 3D. We can say that the structure of the instabilities is different in 3D and a more stable mixing is more is seen for this set of parameters. The instabilities appear more at the central portion of the channel and smooth fingers are seen towards the both ends. We also observe that the three-dimensional instabilities are more coherent than that of the two-dimensional counterparts as found by Hallez and Magnaudet [10] previously. Longer finger lengths are observed in 3D than that in 2D at the same instant of time. This is in agreement with the findings of Oliveria and Meiburg [11].

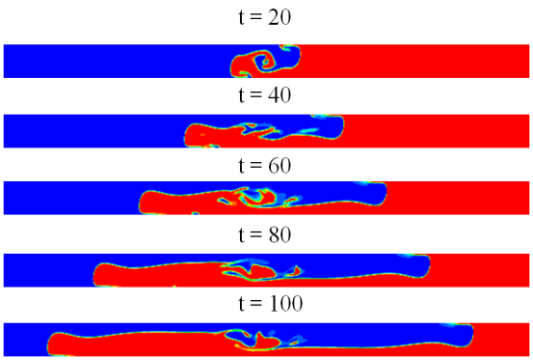

Fig. 4: The contours of the index function ϕ at different times in the X-Y cross-section. The parameters are the same as that of fig. 3.

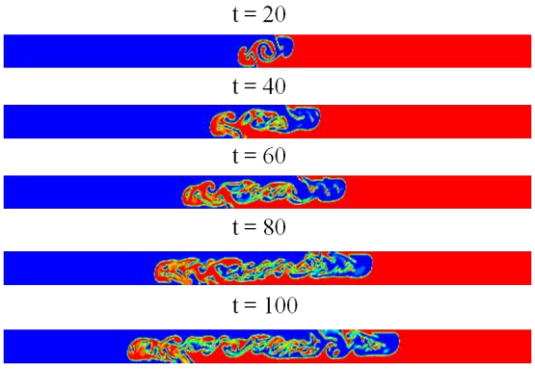

Fig. 5: The contours of the index function ϕ obtained in the two-dimensional simulation for the parameters considered same as that of fig. 3.

To summarize, we have implemented the lattice Boltzmann method on a graphics processing unit to perform three-dimensional simulations of buoyancy-driven flow of two immiscible liquids. Implementation of LBM on GPU is miraculously faster than that of CPU. With the use of GPU,

larger domains could be used that are necessary for 3D simulations. The contours obtained in 3D simulation and the comparisons with the corresponding two-dimensional case are shown. More findings will be presented at the conference.

ACKNOWLEDGMENT

The support from Indian Institute of Technology Hyderabad, India and the Department of Science of Technology, India are gratefully acknowledged.

REFERENCES

[1] NVIDIA. NVIDIA CUDA C Programing Guide. Version 3.2, 2012.

[2] S. P. Vanka, A. F. Shinn and K. C. Sahu, "Computational fluid dynamics using Graphics processing units: Challenges and opportunities", Proceedings of the ASME IMECE 2011.

[3] W. Li, X. Wei, and A. Kaufman, "Implementing lattice Boltzmann computation on graphics hardware", Vis. Comput. 19 (7 - 8), 2003, pp. 444-456.

[4] J. Tolke, "Implementation of a lattice Boltzmann kernel using the compute unified device architecture developed by NVIDIA, Comput. Vis. Sci. 13, 2010, pp. 29-39.

[5] L. Peng, K. Nomura, T. Oyakawa, R. Kalia, A. Nakano and P. Vashishta, Parallel lattice Boltzmann flow simulation on emerging multi-core platforms, in: Euro-Par2008--Parallel Processing, in: Lecture Notes in Computer Science, vol. 5168, 2008, pp. 763-777.

[6] K. C. Sahu and S. P. Vanka, "A multiphase Lattice Boltzmann Study of Buoyancy–Induced Mixing in a Tilted Channel," Computers and Fluids 50, 199 (2011).

[7] P. R. Redapangu, K. C. Sahu and S. P. Vanka, "A study of pressure-driven displacement flow of two immiscible liquids using a multiphase lattice Boltzmann approach", Phys. Fluids. 24, 2012, pp. 102110;

[8] X. He, S. Chen, and R. Zhang, "A lattice Boltzmann scheme for incompressible multiphase flow and its application in simulation of Rayleigh-Taylor instability," J. Comput. Phys 152, 642 (1999).

[9] P. R. Redapangu and K. C. Sahu, "Multiphase lattice Boltzmann simulations of buoyancy-induced flow of two immiscible fluids with different viscosities," European Journal of Mechanics B/Fluids 34, 105 (2012).

[10] Y. Hallez and J. Magnaudet, "Effects of channel geometry on buoyancy-driven mixing", Phys. Fluids 20, 2008, 053 306.

[11] R. M. Oliveira and E. Meiburg, "Miscible displacements in hele-shaw cells: three-dimensional navier-stokes simulations", J. Fluid Mech. 687, 2011, 431–460.