Multi-GPU acceleration of direct pore-scale modeling of fluid flow in natural porous media

Saeed Ovaysi *, Mohammad Piri

Department of Chemical and Petroleum Engineering, University of Wyoming, Laramie, WY 82071-2000, USA

A R T I C L E   I N F O
Article history:
Received 23 June 2011
Received in revised form 8 March 2012
Accepted 16 April 2012
Available online 25 April 2012

Keywords:
GPU computing
Parallel programming
Moving Particle Semi-implicit
Particle-based methods
Porous media
MMPS

A B S T R A C T
Modified Moving Particle Semi-implicit (MMPS) is a particle-based method used to simulate pore-scale fluid flow through disordered porous media. We present a multi-GPU implementation of MMPS for hybrid CPU–GPU clusters using NVIDIA’s Compute Unified Device Architecture (CUDA). Message Passing Interface (MPI) functions are used to communicate between different nodes of the cluster and hence their respective GPUs. The accuracy and stability of the GPU implementation of MMPS are verified through careful comparison with the results obtained on conventional CPU-only clusters. We then examine the speedup and scalability of the GPU implementation for pore-scale flow simulations in samples with various sizes taken from the same natural porous system. We achieve a $134 \times$ speedup with 60 graphics cards compared to 6 CPU cores while maintaining a linear scalability. Incompressible fluid flow simulation to reach steady-state through a $1 \text{mm} \times 1 \text{mm} \times 8 \text{mm}$ microtomography image of Bentheimer sandstone is also performed in less than 1 h.

© 2012 Elsevier B.V. All rights reserved.

1. Introduction

Fluid flow in porous media is of great importance in many areas of science and technology including petroleum production, hydrology, and environmental remediation. Better understanding of fluid flow in porous media, however, requires examining the physics of fluid flow at the pore level (micron level). Currently, it is very difficult to achieve this goal through only experimental means. Therefore, it is crucial to develop models that are capable of reproducing the true physics of fluid flow through porous media at the pore level.

Recently, Modified Moving Particle Semi-implicit (MMPS) has been developed to directly model fluid flow in disordered porous media at the pore level [1]. MMPS, when applied on high-resolution images obtained using X-ray microtomography, can shed light on transport phenomena in natural porous media [2]. However, the high resolutions required to capture the pore-level complexities of natural porous media make the simulations computationally expensive. We have previously presented a parallel implementation of the method which scales linearly on distributed memory clusters [1]. Nonetheless, several hours are still required to complete a physically useful simulation on more than 200 processing cores. Fortunately, with the advances made in General Purpose computation on Graphics Processing Units (GPGPU), it is possible to reduce the computational cost significantly at a considerably lower price.

In this paper, we use Compute Unified Device Architecture (CUDA) developed by NVIDIA [3] to perform the simulations on Graphics Processing Units (GPUs). CUDA, which is an extension of C, is integrated with Message Passing Interface (MPI) to allow simulations across a multi-GPU platform with distributed memory. The underlying architecture of the code is written in C++. In the following sections, we first briefly introduce the computational algorithm of MMPS. Next, our single-GPU algorithm is discussed. We then integrate the single-GPU code with the domain decomposition technique facilitated by MPI to finalize a multi-GPU code that runs on distributed memory computer clusters. This is then followed by our scalability results obtained using the above-mentioned code. Finally, we present a case study in which fluid flow in a large sample is simulated using the multi-GPU code.

2. MMPS

MMPS is a Lagrangian particle-based method used to solve the incompressible Navier–Stokes equations in disordered porous media. The voxel image of the porous medium renders itself to a particle-based representation where the rock (void) space is mapped into solid
(fluid) particles. To solve the incompressible Navier–Stokes equations, MMPS uses a dual-level approach known as the pressure projection method to calculate the velocity of fluid particles. First, an explicit velocity is calculated using

$$\mathbf{v}_E = \mathbf{v} + \left( -\frac{1}{\rho} \nabla P_s + \frac{\mu}{\rho} \nabla^2 \mathbf{v} + \mathbf{g} \right) \Delta t$$

(1)

where \( \mathbf{v} \) is velocity, \( \rho \) is density, \( \mu \) is viscosity and \( \mathbf{g} \) is gravity. Subscript \( E \) stands for explicit and \( \frac{\partial}{\partial t} \) denotes the substantial derivative with respect to time, \( t \). \( P_s \), static pressure, is calculated using

$$\nabla^2 P_s = 0.$$  

(2)

The explicit velocity is then used to calculate dynamic pressure, \( P_d \), using

$$\nabla^2 P_d = \frac{\rho}{\Delta t} \left( \nabla \cdot \mathbf{v}_E + \frac{1}{\Delta t} \sum_{k=1}^{N} \nabla \cdot \mathbf{v}^k \Delta t^k \right)$$

(3)

which is then used in the following equation to calculate the implicit velocity:

$$\mathbf{v}_I = -\frac{\Delta t}{\rho} \nabla P_d.$$  

(4)

It must be noted that the second term in the right hand side of Eq. (3) takes into account any deviation from the incompressibility criterion in the previous time steps, see [1] for more details. The above equations are linearized using the particle-based summations proposed by Koshizuka [4], which define the gradient of the scalar field \( A \) for particle \( i \) as

$$\nabla_i A = \frac{d}{\lambda_i} \sum_{j}^{N_i} \left( \frac{A_j - A_i}{r_{ij}^2} (r_j - r_i) \right) W_{ij}$$  

(5)

where \( d \) is the number of spatial dimensions, \( r \) denotes the coordinates, \( r_{ij} = |r_i - r_j| \) is the distance between particles \( i \) and \( j \), \( N_i = \sum_{j}^{N} W_{ij} \) is the number density, and \( W \) is a kernel that gives higher weights to the particles close to \( i \) than the particles at a distance. To reduce the computational and memory costs, this summation is performed only over a predefined radius of neighborhood, kernel size, from \( i \) which is denoted by \( h \) in Eq. (6). The number of neighboring particles residing in this neighborhood is denoted by \( N \). In this work, we use the following polynomial kernel:

$$W_{ij} = \begin{cases} 
\left( \frac{2r_{ij}}{h} \right)^2 & 0 \leq r_{ij} < h \\
\left( \frac{2r_{ij}}{h} - 2 \right)^2 & h \leq r_{ij} < h/2 \\
0 & r_{ij} \geq h.
\end{cases}$$

(6)

Similar principles are used to compute the divergence operator of the vector field \( \mathbf{v} \) and the Laplacian operator of the scalar field \( A \) for particle \( i \):

$$\nabla_i \cdot \mathbf{v} = \frac{d}{\lambda_i} \sum_{j}^{N_i} \frac{\mathbf{v}_j - \mathbf{v}_i \cdot (r_j - r_i)}{r_{ij}^2} W_{ij}$$

(7)

$$\nabla_i^2 A = \frac{2d}{\lambda_i N_i} \sum_{j}^{N_i} (A_j - A_i) W_{ij}$$

(8)

where \( \lambda = \int \frac{W(x)}{x^2} dx \). The Bi-Conjugate Gradient Stabilized (BiCGStab) solver is used to solve the systems of linear equations encountered in steps 3 and 5. These systems of linear equations
Fig. 1. Computational times required to perform the main numerical operations in one time step of MMPS. The computational times are averaged over 1000 time steps for a system composed of 315,000 particles on one Intel Xeon X5670 core.

\[ \hat{\mathbf{N}}_i^s - \sum_j P_j W_{ij} = 0 \]  

where \( \mathcal{M}_i \) represents the subset of particles in the neighborhood of \( i \) that do not influence its \( P_s \), i.e., solids and disconnected particles.

\[ \mathcal{N}_i P_{di} - \sum_j P_j W_{ij} = -\frac{\rho \lambda}{2 \Delta t} \left( \sum_j \left( \mathbf{v}_{E,j} - \mathbf{v}_{E,i} \right) \cdot \left( \mathbf{r}_j - \mathbf{r}_i \right) / r_{ij}^2 \right) W_{ij} + \sum_j \frac{\left( \mathbf{v}_{N,j} - \mathbf{v}_{N,i} \right) \cdot \left( \mathbf{r}_j - \mathbf{r}_i \right) / r_{ij}^2}{\rho \lambda} W_{ij} - \frac{\rho \lambda N_i}{2d \Delta t^2} \sum_{k=1}^{n-1} \nabla \cdot \mathbf{v}_k \Delta t^k \]  

where superscripts \( n \) and \( k \) denote the previous time step and the time steps earlier than \( n \), respectively.

Fig. 1 presents the contribution of each major computation in one time step of MMPS. The computational times are averaged over 1000 time steps for a system comprised of 315,000 particles. Knowing that the particles move only a fraction of their size at every time step, it is not necessary to perform neighbor search at every time step (for instance, once every 10 time steps here). This reveals the solution of the system of linear equations at step 5 as the bottleneck of the simulations with a degree of freedom equal to the total number of particles. Depending on the resolution at which the porous medium is imaged, the total number of particles for a \((1 \text{ mm})^3\) sample is of the order of tens of millions. Also, considering the movement of particles from one time step to another and the resulting change in the velocity of the particles, the system of linear equations at step 5 presents a non-symmetric system that requires several iterations of the BiCGStab solver to converge. In the next section, we discuss a single-GPU approach to overcome the computational cost associated with these computations.

3. Single-GPU acceleration

A GPU consists of hundreds of stream processors which are grouped in stream multi-processors, see Table 1 for the specifications of two NVIDIA graphics cards. Fig. 2 illustrates the memory hierarchies that are explicitly accessible to the stream processors through the CUDA API functions. To access the GPU (device) processors, the CPU (host) must launch a kernel function. This moves the execution to the device where \( \text{Number of blocks per grid} \times \text{Number of threads per block} \) threads execute the same instruction in the kernel function. Launching a kernel with a greater number of threads than the actual number of processors can potentially overlap computation with high latency memory access to the GPU’s global memory. Utilizing the shared and constant memories as well as coalescing are other efficient ways to speed up GPU memory access, see [5] for the details. In this section, we briefly explain the techniques used to speed up the main MMPS operations outlined in Section 2.
Table 1

<table>
<thead>
<tr>
<th>Specifications of two NVIDIA graphics cards.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Graphics card</td>
</tr>
<tr>
<td>Number of multiprocessors</td>
</tr>
<tr>
<td>Number of processors</td>
</tr>
<tr>
<td>Clock rate (GHz)</td>
</tr>
<tr>
<td>Global memory (Mbytes)</td>
</tr>
<tr>
<td>Constant memory (bytes)</td>
</tr>
<tr>
<td>Shared memory per multiprocessor (bytes)</td>
</tr>
<tr>
<td>Registers per multiprocessor</td>
</tr>
<tr>
<td>Memory bandwidth (Gbytes/s)</td>
</tr>
<tr>
<td>Max number of blocks per grid</td>
</tr>
<tr>
<td>Max number of threads per block</td>
</tr>
</tbody>
</table>

3.1. Neighbor search

Before using the linked-list algorithm, the entire domain must be divided into subdomains (cubes in three dimensional systems) as large as the kernel size, $h$. As shown in Fig. 3, to find the neighbors of any particle $i$, only the subdomains in the immediate neighborhood of $i$ are searched. Below is a pseudo-code for the CUDA kernel, performing neighbor search using the linked-list algorithm. $np$ is the number of particles. Storing the coordinates of the particles in the shared memory at the beginning reduces the cost of future memory accesses to these data.

```plaintext
__global__ void neighbor_search(...) {
    id = blockIdx.x * blockDim.x + threadIdx.x;
    if(id<np) {
        store coordinates of particle[id] in the shared memory
        for(over the x coordinate of the closest subdomains)
            for(over the y coordinate of the closest subdomains)
                for(over the z coordinate of the closest subdomains)
                    for(j= all the particles in the subdomain)
                        r = distance between particle[id] and particle[j]
                        if(r<h)
                            add index of particle[j] to array of neighbors for particle[id]
    }
}
```

3.2. System of linear equations

The most time-consuming operation in the BiCGStab solver is matrix–vector multiplication. To perform this efficiently, we allocate each row of the sparse matrix to a block. We then use the shared memory to calculate the summation of multiplications for each row. This algorithm proves to be four times faster than the case where each row is allocated to one thread. Below, we present a pseudo-code to multiply a sparse matrix characterized by its elements, i.e., $mult$, and element ids, i.e., $mult_id$, by vector $B$. The results are stored in vector $C$. It should be noted that, in calculating the global summation, especial care must be taken to minimize thread divergence and hence improve the achieved speedup. To do that, we have employed the global summation technique presented in [5].

```plaintext
__global__ void multiply(...) {
    id = blockIdx.x;
    tx = threadIdx.x;
    __shared__ float s_C[Dimension of the block];
    if(id<np) {
        s_C[tx] = mult[tx]* B[mult_id[tx]];
        __syncthreads();
        for(s=blockDim.x>>2; s>0; s>>=1) {
            if(tx<s)
                s_C[tx] = s_C[tx+s];
            __syncthreads();
        }
        if(!tx)
            C[id] = s_C[0];
    }
}
```
4. Multi-GPU acceleration

Flow simulation in large systems requires a considerable amount of memory that is beyond what a single GPU can provide. For this reason, and also to achieve higher speedups, it is necessary to develop a multi-GPU implementation. Here, we first discuss domain decomposition. Each subdomain is then allocated to one MPI process which in turn hands over the simulation to one GPU. Message passing and synchronization of the GPUs are done by MPI functions. An optimized domain decomposition must lead to a balanced distribution of computational load and also minimum communication between the processes [6].

To achieve load balance, we decompose the domain into subdomains with equal volumes. Since we model the flow of incompressible fluids, subdomains with equal volumes imply handing over an equal number of particles to each MPI process. Also, to optimize the neighbor search step for each process, domain decomposition must be done along the subdomains of the linked-list algorithm discussed in Section 3. Fig. 4 illustrates a case where the entire domain is decomposed between 4 processes. At the initialization phase, the grey linked-list subdomains (small squares in the figure) are marked and the neighbor search is initiated at each time step by passing the coordinates of the particles in the grey subdomains to other processes. It should be noted that, in practice, only a fraction of the particles in the grey subdomains are the actual neighbors of the particles in the neighboring processes. Since each iteration of the BiCGStab solver
Fig. 5. The actual neighboring particles of process $P_n$ are shown with filled particles. This constitutes only 68% of the particles in the grey area.

Fig. 6. GPU-computed velocity distribution in a 0.5 mm $\times$ 0.5 mm $\times$ 1 mm Berea sandstone X-ray image with 5.345 µm resolution and a total of 2,519,424 particles. The sample is under 1 Pa pressure difference along the $z$ direction (from left to right) and the figure illustrates the results after 0.01 s of the real time.

requires twice updating the information in the other processes, it is wise to first mark the actual neighboring particles of each process and then for the rest of the communications only pass the actual neighboring particles. In Fig. 5, the actual neighboring particles constitute only 68% of the particles in the grey zone. Considering that several iterations are required to solve the system of linear equations arising from Eqs. (2) and (3), this 32% reduction in the communication cost goes a long way in further optimizing the code. More optimization is achieved by overlapping the computations with communications using non-blocking MPI calls. For instance, while performing neighbor search, a non-blocking MPI call can pass the coordinates of the particles residing in the grey subdomains to the neighboring process at the beginning with almost no cost. The receiving process can start by calling a non-blocking receive operation and immediately begin searching the inner subdomains that do not need information from the neighboring processes and then wait for the non-blocking receive operation to complete, which then allows it to proceed with searching the peripheral subdomains. Also, further overlapping is achieved by performing the communications that are required to mark the actual neighbors in another thread of the same MPI process while the master thread performs computations.

5. Scalability and results

To evaluate the scalability of our GPU implementation of MMPS and the accuracy and stability of the results, we studied a 0.5 mm $\times$ 0.5 mm $\times$ 1 mm sample from the Berea sandstone image in [7]. Fig. 6 visualizes the GPU-computed velocity distribution in this sample with the same accuracy as the results obtained using only CPUs. The accuracy of the CPU-only results has previously been verified in [1,2] against analytical/numerical/experimental data available in the literature.

The reported results in this section are obtained using a computer cluster with 30 nodes. Each node is equipped with 2 AMD Opteron 6128 CPUs and 2 NVIDIA GTX 580 GPUs and 32 GB of memory. In addition to that, the head node of this cluster is equipped with 2 Intel Xeon X5670 CPUs and 8 GPUs (4 NVIDIA Tesla C2050s and 4 NVIDIA GeForce GTX 580 GPUs). Depending on whether a parallel CPU-based code has been developed or not, the scalability results can be presented in two ways (see the next paragraph). Since, we have already developed a parallel CPU-based code [1], the scalability results in this work are presented in both ways. Furthermore, all the simulations, i.e., both CPU based and CPU–GPU based, are performed using single precision floating point operations.
In the first approach, we are interested to know how much performance could be gained if we were to perform the MMPS computations on a GPU rather than using a sequential code that utilizes only one core of a CPU. Fig. 7 summarizes the speedup achieved for the major numerical operations in an average time step of MMPS on a single Tesla C2050 GPU for a system composed of 315,000 particles. Compared to Fig. 1, the three main contributors to the computational time, namely neighbor search, computation of $P_s$, and computation of $P_d$, have gained, on average, more than 26x speedup. However, this speedup proved to be dependent on the system size. The same simulation on a system with twice the size of the base system, i.e., with 630,000 particles, achieved more than 28x speedup. Interestingly, the less expensive GTX 580 outperformed Tesla C2050 and achieved a 34x speedup on the base system which translates to a 36% better performance.

Having developed a CPU-based parallel code, in the second approach we are interested in examining how fast the code runs on multiple GPUs when compared to the performance gained using all the cores of a CPU. To do that, we first compute the computational time on all the 6 cores of the Intel Xeon X5670 CPU on the head node. Then, we obtain the computational time for the same simulation on the computing nodes. Fig. 8 presents the scalability results for different system sizes relative to the base system that is composed of 315,000 particles. As shown in this figure, the multi-GPU code performs better when larger systems are used. This observation, which is consistent with the results presented in [8], underscores the cost of expensive memory access in GPUs. Since each particle is assigned to one CUDA thread, by increasing the system size, and hence the number of particles, one increases the number of threads that operate at a given time. This allows a better utilization of the processing power as the idle threads can start processing while other threads wait to access the GPU’s global memory. The actual computational times are also reported in Table 2. These numbers reveal a 3.3x better performance for one NVIDIA GeForce GTX 580 over all the 6 cores of an Intel X5670 CPU. It should be stressed that, depending on the orientation of the processes in the x, y, and z directions, domain decomposition can slightly increase/decrease the number of iterations required to obtain the same level of precision.

6. Case study

In this section we study pore-scale fluid flow in Bentheimer sandstone. The entire microtomography image, shown in Fig. 9, is a 4 mm × 4 mm × 9 mm cylinder with 6.007 µm resolution. We cut a 1 mm × 1 mm × 8 mm sample from the central part of this image which amounts to 43,740,000 particles and apply a 25 Pa pressure difference across its length, i.e., from left to right as shown in Fig. 10.
Fig. 9. Isosurface of a 4 mm × 4 mm × 9 mm Bentheimer sandstone sample with 6.007 µm resolution.

Fig. 10. Distribution of static pressure in a 1 mm × 1 mm × 8 mm sample from a Bentheimer sandstone microtomography image with 6.007 µm resolution. Only the fluid particles are visualized.

Fig. 11. Distribution of velocity in a 1 mm × 1 mm × 8 mm sample from a Bentheimer sandstone microtomography image with 6.007 µm resolution. Only the fluid particles faster than 3 × 10^{-4} m/s are shown.

Table 2

Average computational times, in seconds, for one time step of flow simulations. CPU results are obtained using all the 6 cores of an Intel Xeon X5670 CPU and the GPU results are based on NVIDIA GeForce GTX 580 GPUs.

<table>
<thead>
<tr>
<th>System size</th>
<th>1×</th>
<th>2×</th>
<th>4×</th>
<th>8×</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 CPU</td>
<td>2.328</td>
<td>4.825</td>
<td>11.15</td>
<td>22.24</td>
</tr>
<tr>
<td>2 GPUs</td>
<td>0.7168</td>
<td>1.424</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>4 GPUs</td>
<td>0.4157</td>
<td>0.7369</td>
<td>1.583</td>
<td>NA</td>
</tr>
<tr>
<td>8 GPUs</td>
<td>0.2273</td>
<td>0.4130</td>
<td>0.8883</td>
<td>0.9724</td>
</tr>
<tr>
<td>12 GPUs</td>
<td>0.09641</td>
<td>0.1629</td>
<td>0.3279</td>
<td>0.6057</td>
</tr>
<tr>
<td>16 GPUs</td>
<td>0.07355</td>
<td>0.1236</td>
<td>0.2729</td>
<td>0.4818</td>
</tr>
<tr>
<td>18 GPUs</td>
<td>0.07200</td>
<td>0.1209</td>
<td>0.2417</td>
<td>0.4297</td>
</tr>
<tr>
<td>24 GPUs</td>
<td>0.05504</td>
<td>0.09203</td>
<td>0.1931</td>
<td>0.3288</td>
</tr>
<tr>
<td>27 GPUs</td>
<td>0.05328</td>
<td>0.09266</td>
<td>0.1669</td>
<td>0.3169</td>
</tr>
<tr>
<td>36 GPUs</td>
<td>0.04337</td>
<td>0.07401</td>
<td>0.1357</td>
<td>0.2473</td>
</tr>
<tr>
<td>48 GPUs</td>
<td>0.03383</td>
<td>0.05440</td>
<td>0.1089</td>
<td>0.1894</td>
</tr>
<tr>
<td>54 GPUs</td>
<td>0.02850</td>
<td>0.04550</td>
<td>0.09436</td>
<td>0.16676</td>
</tr>
</tbody>
</table>

Fig. 11 shows the velocity distribution in the most active fluid channels, i.e., faster than 3 × 10^{-4} m/s, at steady-state, i.e., t = 0.006 s. For this simulation which completed in less than 1 h we used a 3 × 10^{-6} s time step and let the simulation run for 2000 time steps. To cross-check the accuracy of the results, exactly the same simulation was run on the entire 480 CPU cores of the cluster. However, the CPU-based simulation, which was as accurate as its hybrid counterpart, completed in 4 h.

7. Conclusions

We presented an MPI–CUDA implementation of MMPS to simulate fluid flow through naturally-occurring porous media on hybrid CPU–GPU clusters. This implementation has proved to be very efficient with linear scalability and has enabled us to perform physically meaningful simulations in a practical amount of time. Although the largest system size is still limited by the total global memory of all the GPU cards, this implementation has enabled us to look at significantly larger samples of natural porous media with higher resolutions. The scalability results presented in this paper lead us to conclude that hybrid CPU–GPU clusters can deliver the same computational power at
a much lower price than conventional CPU clusters. Considering the current market prices of both the NVIDIA GeForce GTX 580 graphics card and the Intel X5670 CPU and the results presented in Section 5, the same simulation would be $9.6 \times$ less expensive on an NVIDIA GeForce GTX 580 graphics card. This holds true if one were to develop a parallel code that would utilize all the available 6 CPU cores on the Intel X5670 CPU. We believe that this opens up new doors to applications demanding high computational power at lower cost.

References