International Journal of Applied Mathematics
Volume 25 No. 4 2012, 547-557

FINITE POINTSET METHOD FOR 2D DAM-BREAK PROBLEM WITH GPU-ACCELERATION

M. Panchatcharam[1] §, S. Sundar[2]
[1,2]Department of Mathematics
IIT Madras, Chennai, 600 036, INDIA
[1]e-mail: mpanch13114@gmail.com

Abstract: A Lagrangian particle scheme is applied to the projection method for the incompressible Navier-Stokes equations. The approximation of spatial derivatives is obtained by a computationally expensive Finite Pointset method. GPU computations are applied to improve the computational speed-up. The numerical solutions are obtained for the broken dam problem and are compared with the analytical solutions. We used a single-core NVIDIA Tesla M2050 GPU to compare a single-core CPU (Intel Xeon) and achieved a 60 times speed up from GPU.

AMS Subject Classification: 65Y05, 65Y20, 35Q30, 76D05
Key Words: finite pointset method, CUDA, GPU, dam-break

1. Introduction

Application of Graphics Processing Units (GPUs) in large-scale simulations leads to high performance computing with a peak performance and inexpensive cost. GPU has recognized the application of massively parallel hardwares as one of the ingredients to satisfy future computing requirements. They are a prototype for a general class of many-core processors with a high thread-parallelism which is expected to dominate future computer clusters. Computational fluid dynamics (CFD) is one of the important area to utilize the multi-core GPU.

Received: June 29, 2012

§Correspondence author

Harada et al [5] obtained 28 speedup on GPU for Smoothed Particle Hydrodynamics (SPH) solver using 262,144 particles. Rosinelli et al [8] present a GPU-accelerated solver for simulations of bluff body flows in 2D using a remeshed vortex particle method and 30 speed-up was obtained. In this paper, we have chosen NVIDIA GPU with CUDA platform.

We solve the incompressible Navier-Stokes equation using a Lagrangian method called Finite Pointset Method (FPM) which is a mesh free particle method developed by Tiwari et al [9]. In this method, the computational domain is filled by a finite number of particles (pointset) which move with fluid velocities and they carry the fluid quantities like density, velocity, pressure and so on. The differential operators at a particle position are approximated by its neighbor particles. The distribution of particles can be arbitrary and hence they may scatter, make holes, and accumulate at some place, which in result yields numerical instability. To get the numerical stability, one has to mange the particles. Although FPM solves many real world problems, it is computationally so expensive compare to mesh based method. Therefore, we employ GPU on FPM to decrease the computational cost.

2. Governing Equations and Numerical Technique

Let us consider the incompressible Navier-Stokes equations in the Lagrangian form

$$\frac{D\vec{v}}{Dt} = -\frac{1}{\rho}\nabla p + \nu\Delta\vec{v} + \vec{g}, \tag{1}$$

$$\nabla.\vec{v} = 0, \tag{2}$$

where $\vec{v}$ is the velocity vector, $\vec{g}$ is the body force acceleration vector, ν is the kinematic viscosity and p is the dynamic pressure. Appropriate initial and boundary conditions are supplemented for the equations (1) - (2). In this paper, we consider the solid wall and free surface boundary conditions. For a solid wall, we have used no slip boundary condition. The surface stress boundary condition on the interface or free surfaces is given by

$$[\tau.\vec{n} - p\vec{n}] = \sigma\kappa\vec{n}, \tag{3}$$

where σ is the surface tension of the fluid, which is assumed to be constant, κ is the curvature on the interface, τ is the stress tensor. The symbol [.] denotes the jump on the free surface boundary between two fluids. Suppose the viscosity of

the fluid adjacent to the free surface is negligible and has the pressure p_0, then the normal and tangential components of (3) are given by

$$p - \hat{n}\tau\vec{n} = p_0 + \sigma\kappa, \tag{4}$$

$$\hat{t}\tau\vec{n} = 0. \tag{5}$$

Tiwari et al [9] used Chorin's projection method to solve these equations. It consists of two steps. The first step is to compute the new particle position and intermediate velocity $\vec{v}^*$,

$$\vec{x}^{n+1} = \vec{x}^n + \Delta t\vec{v}^n, \tag{6}$$

$$\vec{v}^* = \vec{v}^n + \Delta t\nu\Delta\vec{v}^n + \Delta t\vec{g}^n. \tag{7}$$

At the second step, we correct $\vec{v}^*$ by solving the equation

$$\vec{v}^{n+1} = \vec{v}^* - \Delta t\nabla p^{n+1} \tag{8}$$

with the constraint

$$\nabla.\vec{v}^{n+1} = 0. \tag{9}$$

From (8) and (9), we get

$$\Delta p^{n+1} = \frac{\nabla.\vec{v}^*}{\Delta t}. \tag{10}$$

Neumann boundary condition for p is obtained by projecting the equation (8) on the outward unit normal vector $\vec{n}$ to the boundary Γ with the assumption $\vec{v}.\vec{n} = 0$ on Γ, i.e.

$$\left(\frac{\partial p}{\partial \vec{n}}\right)^{n+1} = 0 \tag{11}$$

on Γ. Moreover, the Dirichlet boundary condition

$$p = p_0 + \sigma\kappa + \hat{n}\tau\vec{n} \tag{12}$$

applies for free surface as well as for outflow particles in the context of the pressure Poisson equation.

We approximate the spatial derivatives appearing in (7), (8) and the pressure Poisson equation (10) by FPM as explained in Tiwari et al [9].

3. GPU Implementation

Our aim is to solve the above system of equations in GPU. Before implementing them in GPU, let us have an outlook of GPU and CUDA.

3.1. GPU and CUDA Architecture

Graphics processing is characterized by doing the same operation on massive amounts of data. To accommodate this way of processing, graphical processing units (GPUs) consist of a large number of relatively simple processors, fast but limited local memory, and fast internal buses to transport the operands and results. An advantage of GPU is that they are relatively cheap because of the enormous numbers sold for graphical use in virtually every PC. But, one drawback is the 32-bit precision of the typical GPU and, in some cases more importantly, there is no error correction available. GPU vendors have been quick to focus on the HPC community and they renamed their graphics cards GPGPUs, i.e., general-purpose GPUs. It is no longer necessary to reformulate a computational problem into a piece of graphics code. There are C-like languages and runtime environments available that make the life of a developer of GPUs much easier: for NVIDIA this is CUDA (Compute Unified Device Architecture), which has become quite popular with users of these systems.

NVIDIA is the big player in the GPU field with regard to HPC. The GigaThread Engine is able to schedule different tasks in the streaming multiprocessors (SMs) (now called CUDA cores by NVIDIA) in parallel. This improves the occupation rate of the SMs. Tesla M2050 is an assembly of 14 multiprocessors (MP), with 32 cores in each. Each possesses its own shared memory (48 KB) commonly used by all the 32 cores in it. Communications between the MPs are performed through the device memory (global memory; 3 GB), which is accessible to all the cores of the MPs. But, when the ECC (Error Correction Code) is on, a portion of dedicated memory is used for ECC bits, so the available user memory is reduced by 12.5%

The CUDA programming model is a group of threads running in parallel. A group of threads is called a block which runs on MP. An assembly of all blocks in a single execution is called a grid. The number of threads is limited by the amount of shared memory and registers, so it is application-dependent.

In this paper, we have written the CPU code on *Fortran* and whenever CUDA is required, we assist C to transfer the values from CPU to GPU.

We will end this section by explaining how a CUDA program is compiled. Compiling is done in several levels. In the first level, the code dedicated to CPU is extracted from the file and passed to the standard compiler. In the next level, the code dedicated to the GPU is converted into an intermediate language PTX which is like an assembler. Finally, the last level translates this intermediate language into commands that are specific to the GPU and encapsulates them in binary form in the executable.

3.2. GPU on FPM

Algorithm 1 shows the computational procedure of the FPM method which is used in this paper. This computational procedure uses GPU efficiently. At first, it initializes the particle in CPU and then search of neighboring particles is done in GPU. The implicit calculation calculates the pressure term. Pressure values are obtained by solving the Poisson equation (10), which is discretized into simultaneous linear equation using FPM. As the solver of this equation, the BICGSTAB method is implemented in the original FPM method. In addition, GPU is equipped with particle management to avoid the numerical instability. Drum et al [3] explained the details of particles management. At each step, searches of neighboring particles are executed in GPU.

In this paper, the BICGSTAB method is applied for pressure calculation in both GPU-based and CPU-based code.

Algorithm 1 CPU- GPU Algorithm

1. Initialize particles in CPU

For time step n=1,2, $\cdots$

2. Transfer Data from CPU to GPU

3. Particle management & sorting kernel in GPU and copy sorted list to CPU

4. Use Chorin's projection

5. Compute pressure Poisson equation (involves matrix inversion in GPU)

6. Generate matrix and launch BICGSTAB in CPU
(with the help of GPU kernel (for matrix, vector operations))

7. Update coordinates of particles.

In principle, one thread processes one particle. When a thousand particles are calculated, a thousand threads are launched. Since GPU uses SIMT (Single Instruction Multiple Thread) technique, thousands of calculation are done by parallel process. The performance improves if one uses the shared memory which can be accessed much faster than the device memory. But, as mentioned

above, the shared memory is limited to blocks. In this paper, we handled shared memory in an efficient way. For example, to find the matrix-vector product, each row accesses the vector many times. In this case, we instruct the vector to load from device memory to shared memory and then the shared memory is accessed during the calculation. The global memory accesses are performed on segments. In order to optimize the global memory transactions, memory accesses should be coalesced whenever possible. The threads of a block are executed in groups of 32 threads called warps. A warp executes a single instruction at a time to all threads. To avoid warp-divergence, we have implemented synchronization of all threads.

We employed GPU for neighbor searching and particles management, since it is a time consuming part. Texture memory is quite useful for the neighbor search kernel. In this kernel, warp-divergences are occurring and each thread for one particle visits neighboring particles. Since each particle contains different number of neighboring particles, every thread in one warp must wait until every thread of the warp finishes accessing their neighboring particles. A particle sometimes possesses huge number of neighbors than near particles do. Therefore, a thread accessing the particles which has more neighbors keeps any other threads in the same warp waiting, even if the other threads access only the particles possessing a few neighbors. This would result in a non-optimized function of threads. Consequently, in this kernel, it takes the longest computation time in the whole time of the search for neighbors. Therefore, the speed of the calculation of the search for neighbors can be increased, if that of this kernel is improved by using the texture memory.

3.3. BICGSTAB

Since the iterative solver includes a sparse matrix-vector (SpMV) multiplication, it monopolizes the computation time in one iterative operation on a CPU as well as on a GPU. The calculation of a parallel process by many threads on GPU would not get much faster than that by a sequential process of CPU, if one translated the code natively. Nathan et al [7] and Gret et al [2] explained the implementation of BICGSTAB in GPU. Memory latency affects performance of SpMV multiplication. The coefficient matrix is stored into global memory, using CSR format. Therefore, the coalesced global memory access can be achieved on loading elements of the coefficient matrix. But, in this paper, again we used texture memory to accelerate the computation in order to reduce the effect of the irregular and random access.

4. GPU Benchmarking

In this section, the results of simulations by using both the GPU code and CPU code are discussed. Since GPU cannot work alone, we need the assistance of CPU as well. Hence, here GPU code means it is indirectly saying GPU+CPU code. 2D-simulation of dam-break problem is considered for the validation of GPU code. Breaking dam problem is a very popular and simple test case to validate many numerical schemes for the simulations of the free surface flows. It consists of simple configurations and simple initial and boundary conditions. Martin et al [6] reported the experimental results and Hansbo [4] reported his numerical results.

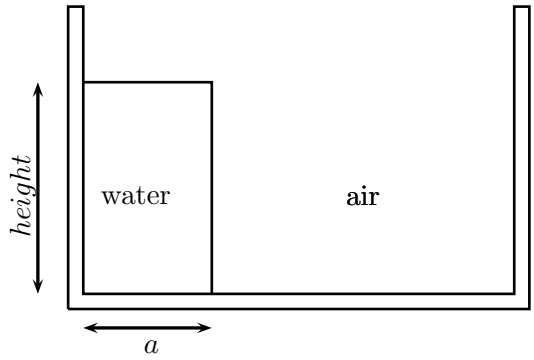

Figure 1: Breaking Dam problem initial setup

Consider the rectangular domain of water with a width $a = 0.1m$ and height $0.2m$. The lines $x = 0, y = 0$and $x = 0.6$ consist of the solid wall. In the simulation, the upper and the right boundary of the water columns are considered as the free surface boundary. The gravity $g = 9.81m/s^2$ acts downwards. The initial velocity and pressure is set to zero. The air pressure is assumed to be zero and surface tensions are neglected. No slip boundary condition is used on the solid walls. Figure 1 shows the initial setup of the problem, where as Figures 2 & 3 show the particle positions successive in time.

4.1. GPU Performance

Many researchers on GPU computing discussed performance by comparing the run-times on one GPU and one CPU core. In this paper, we compare similar

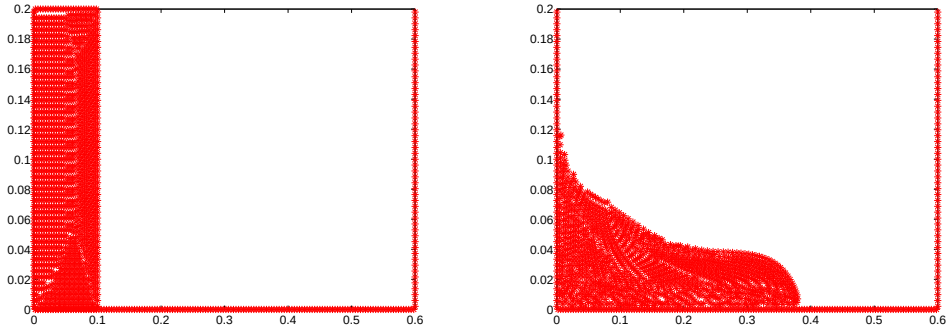

Figure 2: Particle positions at t=0.0 and t=0.24

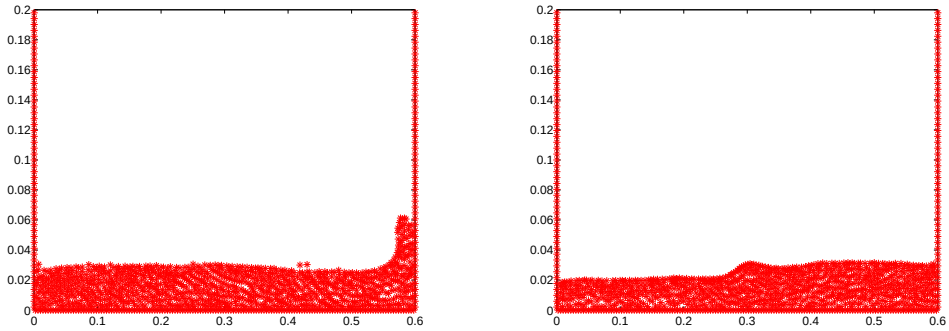

Figure 3: Particle positions at t=0.56 and t=1.2

priced hardware. One always has to buy a CPU if one wants to use a GPU. One can buy a less expensive CPU to control the GPU. Since for highly specialized GPU systems, the controlling CPUs in general cost less than 10% of the GPU price. The benchmark setup for this study is given in table 1.

In most situation, a GBytes size Device memory is enough to contain millions of particles which can support a normal 2D or 3D problem. Thus, we can carry out the computation entirely on GPU without data transfer between Host and Device for most problems. As for the situation with much larger data size in which the Device memory is not enough (if there is only one GPU), our method can hold such an acceleration ratio with using of stream. The data is divided into smaller parts and processed by asynchronous copy. In our present study, the most important problem is to reduce the time spent on data transporting between GPUs. Setting a relatively large buffer memory is an effective way.

	Nvidia Tesla M2050
Number of streaming Processors	448
Size of Memory	3 GB (2.625 GB if ECC is on)
Clock rate	1.15 GHz
Double precision peak value	515 GFlops
Single precision peak value	1030 GFlops
	Dual Intel Xeon 5660
Number of cores	6 (but only one core is used in this study)
Clock rate	2.8 GHz
RAM	32 GB

Table 1: Benchmark Setup

The actual peak values are calculated by clock cycles in the kernel function and are given in the Table 2 together with efficiencies, where efficiency is the ratio between actual (GFLOPS) and theory (GFLOPS).

CPU/GPU	Actual (GFLOPS)	Ratio	Theory (GFLOPS)	Ratio
Intel Xeon 5660 (6 cores)	~ 48		64	
NVIDIA Tesla M2050	~ 148	3.1 $\times$	515	8.1 $\times$
Efficiency	28%			

Table 2: CPUs and GPU have different performance characteristics. The ratios reflect the speed-ups between the grouped CPUs and GPU with respect to double-precision floating-point and memory bandwidth.

4.2. Comparison

Figure 4 shows a single-GPU speed-ups over a single-CPU for the breaking dam problem for different average distances. Since the number of particles will vary in each time step, we have given the approximate number of particles. We get a better performance for large number of particles. This is a typical behavior for GPU, since the cost-intensive global memory data requests can be better overlapped for computations with higher work loads per GPU.

From the figure, one can observe the efficiency of the GPU which is about 60 times faster than the CPU for more than 100 thousand particles. If the number

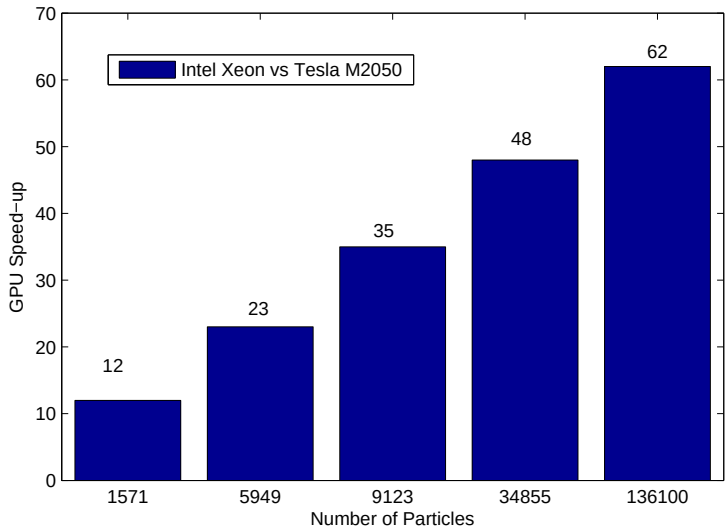

Figure 4: Speed-ups (γ) of single GPU relative to single CPU for Breaking Dam problem

of particles get increases, we can save the computation time enormously by employing GPU.

5. Conclusion

In this paper, we could give very promising benchmarking results which included a fully realistic 60 fold speed-up in comparison to equally priced CPU hardware. We implemented mainly GPU computation for a matrix solver namely BiCGSTAB to reduce the computational domain. In addition, we assisted GPU for neighbor searching and matrix inversion. These are the main processes which consumes most of the computation time in CPU. By the help of GPU, we computed the process which saves computation time. Overall, these results emphasize that our finite pointset method is capable to get good performance on GPU computing.

References

[1] A. Chorin, Numerical solution of the Navier-Stokes equations, *J. Math. Comp.*, **22** (1968), 745-762.

[2] G. Ruetsch, M. Fatica, A CUDA Fortran Implementation of BWAVES (September, 2010) http://www.pgroup.com/lit/articles/insider/v2n3a2.htm.

[3] C. Drum, S. Tiwari, J. Kuhnert, H.J. Bart, Finite pointset method for simulation of the liquid-liquid flow field in an extractor, *Computers and Chemical Engineering*, **32** (2008), 2946-2957.

[4] P. Hansbo, The characteristic streamline diffusion method for the time dependent incompressible Navier-Stokes equations, *Comp. Mech. Appl. Mech. Eng.*, **99** (1992), 171-186.

[5] T. Harada, S. Koshizuka, Y. Kawaguchi, Smoothed particle hydrodynamics on GPUs, *Proceeding of the Spring Conference on Computer Graphics* (2007), 235-241.

[6] J.C. Martin, M.J. Moyce, An experimental study of liquid columns on a liquid horizontal plate, *Philos. Trans. Roy. Soc. London Ser. A*, **244** (1952), 312.

[7] B. Nathan, G. Michael, Efficient sparse matrix-vector multiplication on CUDA, *NVIDIA Technical Report*, NVR-2008-004 (December 2008).

[8] D. Rossinelli, M. Bergdorf, G. Cottet, P. Koumoutsakos, GPU accelerated simulations of bluff body flows using vortex particle methods, *Journal of Computational Physics*, **229** (2010), 3316-3333.

[9] S. Tiwari, J. Kuhnert, Finite pointset method based on the projection method for simulations of the incompressible Navier-Stokes equations, Meshfree methods for partial differential equations (Bonn, 2001), *Lect. Notes on Comput. Sci. Eng.*, Springer, Berlin, **26** (2003), 373-387.