Parallel lattice Boltzmann flow simulation on emerging multi-core platforms

Liu Peng, Ken-ichi Nomura, Takehiro Oyakawa, Rajiv K. Kalia, Aiichiro Nakano, Priya Vashishta

Collaboratory for Advanced Computing and Simulations, Department of Computer Science,

Department of Physics & Astronomy, Department of Chemical Engineering & Materials Science,

University of Southern California, Los Angeles, CA 90089-0242, USA

{liupeng, knomura, oyakawa, rkalia, anakano, priyav}@usc.edu

Abstract

A parallel Lattice Boltzmann Method (pLBM), which is based on hierarchical spatial decomposition, is designed to perform large-scale flow simulations. The algorithm uses critical section-free, dual representation in order to expose maximal concurrency and data locality. Performances of emerging multi-core platforms—PlayStation3 (Cell Broadband Engine) and Compute Unified Device Architecture (CUDA)—are tested using the pLBM, which is implemented with multi-thread and message-passing programming. The results show that pLBM achieves good performance improvement, 2.43 for Cell over a traditional Xeon cluster and 8.76 for CUDA graphics processing unit (GPU) over a Sempron central processing unit (CPU). The results provide some insights into application design on future many-core platforms.

Keywords: *Lattice Boltzmann Method; Flow simulation; Parallel computing; Hybrid thread + message passing programming; Spatial decomposition; Critical section-free, dual representation; PlayStation3 cluster; Cell Broadband Engine architecture; CUDA.*

1. Introduction

We are witnessing a dramatic change into a multi-core paradigm, and computer industry is facing a historical shift, in which Moore's law due to ever increasing clock speeds has been subsumed by increasing numbers of cores per microchip [1,2]. While Intel is deploying multi-core processors across key product lines as a pivotal piece [1], AMD begins its multi-core strategy by introducing quad-core processors, and IBM, Sony, and Toshiba provide Cell, which is also an ideal application test bed to prepare for coming many-core revolution [3,4]. The many-core revolution will mark the end of the free-ride era (i.e., legacy software will run faster on newer chips), resulting in a dichotomy—subsiding speed-up of conventional software and exponential speed-up of scalable parallel applications [5]. Recent progresses in high-performance technical computing have identified key technologies for parallel computing with portable scalability. An example is an Embedded Divide-and-Conquer (EDC) algorithmic framework to design linear-scaling algorithms for broad scientific and engineering applications based on spatiotemporal locality principles [6]. The EDC framework maximally exposes concurrency and data locality, thereby achieving reusable "design once, scale on new architectures" (or metascalable) applications. It is expected that such metascalable algorithms will continue to scale on future many-core architectures.

As mentioned above, multi-core processor is the trend for future supercomputers, and how to develop metascalable applications on such platforms is in great need. The Lattice Boltzmann Method (LBM) for fluid simulations—which features robustness, complicated geometry, multiphases, and ease of parallelization—is a representative of this kind of applications. For example, Williams et al. studied performance optimization of the LBM on multi-core platforms [7], while Stuermer implemented the LBM on Cell [8,9] and Li et al. tested the LBM on graphics processing units (GPUs) [10]. In this paper, we present our design of a unified parallel implementation of the LBM on several emerging platforms including a cluster of Cell-based PlayStation3

consoles and Compute Unified Device Architecture (CUDA) based implementations on GPUs. The paper is organized as follows. Section 2 describes the parallelization of the LBM algorithm. The test beds, testing results, and performance analysis are presented in Section 3. Conclusions and future directions are contained in Section 4.

2. Parallel Lattice Boltzmann Flow Simulation Algorithm

2.1 Lattice Boltzmann method

The essential quantity in the Lattice Boltzmann Method (LBM) [11] is a density function (DF) $f_i(\vec{x},t)$ on a discrete lattice, $\vec{x} = (j\Delta x, k\Delta y, l\Delta z)$ ($j \in [1, N_x], k \in [1, N_y], l \in [1, N_z]$), with discrete velocity values $\vec{e}_i$ ($i \in [0, N_v - 1]$) at time t. Here, each $\vec{e}_i$ points from a lattice site to one of its N_v near-neighbor sites. N_x, N_y, and N_z are the numbers of lattice sites in the x, y, and z directions, respectively, with Δx, Δy and Δz being the corresponding lattice spacings and N_v (= 18) being the number of discrete velocity values. From the DF, we can calculate various physical quantities such as fluid density $\rho(\vec{x},t)$ and velocity $\vec{u}(\vec{x},t)$:

$$\rho(\vec{x},t) = \sum_i f_i(\vec{x},t), \quad (1)$$

$$\rho(\vec{x},t)\vec{u}(\vec{x},t) = \sum_i \vec{e}_i f_i(\vec{x},t). \quad (2)$$

The time evolution of the DF is governed by the Boltzmann equation in the Bhatnagar-Gross-Krook (BGK) model. The LBM simulation thus consists of a time-stepping iteration, in which collision and streaming operations are performed as time is incremented by Δt at each iteration step:

Collision:

$$f_i(\vec{x},t^+) \leftarrow f_i(\vec{x},t) - \frac{1}{\tau}\left(f_i(\vec{x},t) - f_i^{\mathrm{eq}}(\rho(\vec{x}),\vec{u}(\vec{x}))\right), \quad (3)$$

Streaming:

$$f_i(\vec{x}+\vec{e}_i, t+\Delta t) \leftarrow f_i(\vec{x},t^+). \quad (4)$$

In Eq. (4), the equilibrium DF is defined as

$$f_i^{\mathrm{eq}}(\rho,\vec{u}) = \rho(A + B(\vec{e}_i \cdot \vec{u}) + C(\vec{e}_i \cdot \vec{u})^2 + D\vec{u}^2), \quad (5)$$

where A, B, C and D are constants, and the time constant τ is related to the kinematic viscosity ν through a relation $\nu = (\tau - 1/2)/3$.

It should be noted that the collision step involves a large number of floating-point operations that are strictly local to each lattice site, while the streaming step contains no floating-point operation but solely memory copies between nearest-neighbor lattice sites.

2.2 Parallel LBM Algorithm

Our parallel Lattice Boltzmann Method (pLBM) algorithm consists of three functions: collision, streaming, and communication. The total simulation system Ω is decomposed into several sub-domains Ω_i, where $\Omega = \cup_i \Omega_i$, and each domain is mapped onto a processor (see Fig. 1). The collision and streaming functions update DFs on a single domain, while the communication function is responsible for inter-domain DF migrations. To simplify discussion, Fig. 1 shows a schematic of a 2-dimentional system (the actual implementation is for 3 dimensions). Here, the white squares denote open nodes that have DFs, the black squares denote closed nodes that represent obstacles (and hence no flow), and the gray squares denote buffer nodes that hold buffer DFs for inter-domain communication, which are initialized with the corresponding geometry information (open or closed) in neighbor domains at the beginning of simulation. In the 2-dimensional example, a single domain consists of $N_x \times N_y$ nodes, where N_x and N_y are the numbers of lattice sites in the x and y directions, respectively. Each domain is augmented with a surrounding buffer layer of

one lattice spacing, which is used for inter-domain DF migrations. A boundary condition is imposed on DFs propagating toward the closed nodes: reflecting DFs propagation into the closed nodes toward the opposite direction.

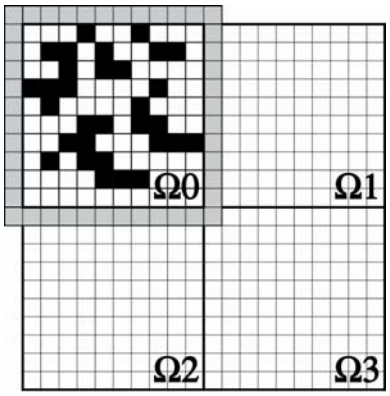

Fig. 1. Schematic of spatial decomposition in 2 dimensions with several (here is 4) domains. White squares are open lattice sites that have the DFs of flow particles. Black squares represent obstacles, where flow does not exist. Gray squares are buffer sites, where some of the DFs move in after streaming function.

2.2.1 Parallel LBM Algorithm on PlayStation3

We use a Linux cluster consisting of PlayStation3 consoles. Within each PlayStation3 console, a main program runs on the Power Processing Element (PPE) and spawns POSIX threads that run on multiple Synergistic Processing Elements (SPEs). Direct Memory Access (DMA) commands are used for data transfer between the main memory of the PPE and the local storage of the SPEs, since there is no access from SPEs to main memory. (Either SPE or PPE can issue DMA commands, which include a get command for retrieving memory contents, and a put command for writing data into memory.) For inter-console message passing, we use the Message Passing Interface (MPI). The hybrid thread + message passing programming thus combines: (1) Inter-console parallelization with spatial decomposition into domains based on message passing; and (2) intra-console parallelization through multi-thread processing of interleaved rows of the lattice within each domain.

Collision

It is a challenging task to design a parallel algorithm due to Cell hardware restrictions. Six SPE programs can be simultaneously performed using POSIX threads on PlayStation3 (only 6 SPEs out of 8 are available for user programming). As mentioned in previous researches, the partitioning of work among the SPEs for load balancing is crucial to high performance [8]. For optimal load balancing, we parallelize by first dividing the simulation problem into a large number (N_x) of chunks, where chunk ID j ($j \in [0, N_x - 1]$) processes lattice sites $\vec{x} = ((j+1)\Delta x, k\Delta y, l\Delta z)$ ($k \in [1, N_y]$, $l \in [1, N_z]$). Here, N_x, N_y and N_z denote the numbers of lattice sites *per domain* in the x, y and z directions, respectively. We then interleavingly assign chunks to threads, i.e., chunk ID j is assigned to SPE with thread ID j mod N_{thread}, $j \in [0, N_{thread}-1]$. In our case, the number of threads N_{thread} is 6, so chunk 0 and chunk 6 are assigned to SPE 0, while chunk 1 and chunk 7 are assigned to SPE 1. In Fig. 2(a), the area enclosed by the dotted lines shows the computational task assigned to the first thread with thread ID 0.

One problem in the interleaved thread parallelization is that multiple threads may update a common lattice site. To avoid such a critical section, we have designed a double-layered DF consisting of two floating-point arrays *DF0* and *DF1*, shown in Fig. 2(b). (In Eqs. (3) and (4), $f_i(\vec{x}, t)$ and $f_i(\vec{x}, t^+)$ denote *DF0* and *DF1*, respectively.) In each LBM loop, the collision subroutine transfers DFs from the array *DF0* to local store on SPE, updates the DFs, and subsequently copies it back to the array *DF1*. The pseudo-code of collision subroutine is given in Table 1, where *fetchAddrData* is the address for a DMA get operation from *DF0* to local store of SPE, *fetchAddrFlag* is the address for DMA get

from main memory to local storage of SPE, and *putAddrData* is the address for DMA put from *DF1* to main memory. In the table, *geom*(*i*,*j*) denotes the flags (open or closed) of the *j*-th cell in chunk *i*. We have not used single instruction multiple data (SIMD) programming in the current implementation.

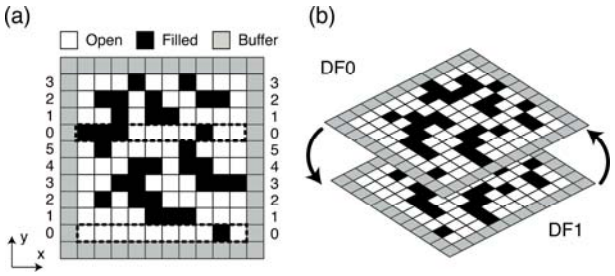

Fig. 2. (a) Schematic of a 2-dimensional system setup for each domain in spatial decomposition. White squares are open lattice sites that have the DF's of flow particles. Black squares represent obstacles in the system, where flow does not exist. Gray squares are buffer sites, where some of the DFs move in after streaming. The simulation system is divided into N_y computational chunks, each of which consists of N_yN_z lattice sites, and the chunks are interleavingly assigned to SPEs. The numerals show thread ID responsible for each chunk. (b) Schematic of a double-layered DF calculation comprising of two floating point arrays *DF0* and *DF1*. The collision function reads DF's from the array *DF0* to do updates, and then store the updated information in the array *DF1*. Subsequently, the streaming function propagates DF's from the array *DF1* to the array *DF0*.

Table 1. Collision calculation algorithm within SPE.

Input:
N_x, N_y, N_z
{number of LBM lattice sites in the *x*, *y* and *z* directions}
N_{thread} {number of threads}
tID {thread ID}
array *DF0* in PPE of size *N*
{array of density functions, where $N = N_xN_yN_z$}
array *geom* {array of geometry flags}

Output:
array *DF1* in PPE of size *N* {array of density functions}

Steps:
1 *chunkID* ← *tID*
2 *chunksize* ← N/N_x
3 **while** *chunkID* < N/N_x **do**
4 *fetchAddrData* ← address of *DF0* + *chunkID*×*chunksize*
5 *fetchAddrFlag* ← address of *geom* + *chunkID*×*chunksize*
6 *putAddrData* ← address of *DF1* + *chunkID*×*chunksize*
7 initiate DMA transfers to get data
8 fetch data from *DF0* and *geom*
9 wait for the data
10 **for** *j* ← 0 to *chunksize*−1
11 $\rho(\vec{x},t) \leftarrow \sum_i f_i(\vec{x},t)$ {see Eq. (1)}
12 $\vec{u}(\vec{x},t) \leftarrow \rho^{-1}(\vec{x},t)\sum_i \vec{e}_i f_i(\vec{x},t)$ {see Eq. (2)}
13 $f_i(\vec{x},t^+) \leftarrow f_i(\vec{x},t) - [f_i(\vec{x},t) - f_i^{eq}(\rho(\vec{x}),\vec{u}(\vec{x}))]/\tau$ {see Eq. (3)}
14 **if** *geom*(*chunkID*, *j*) is open **then** update density functions
15 initiate DMA put for the computed results
16 *chunckID* ← *chunkID* + N_{thread}
17 synchronize using inter-SPE communication

Streaming

The streaming function propagates the DF according to their flow directions, see Eq. (4). Here the DFs are copied from main memory to main memory, between array *DF1* and array *DF0* in Fig. 2(b). Before propagating DF's, a boundary condition such as reflection rule must be considered according to the simulation geometry. In the case of a static geometry, where the relation between source and destination lattice sites does not change, we avoid repeated computing of boundary condition by defining another array to keep the indices of destination lattice sites for each DF, which significantly speeds up the streaming function. Furthermore, we find that the hardware-supported threads on PPE improve the performance of the complicated memory copy. We use two POSIX threads, each of which is responsible for half of the data transfer. This improves the performance of the streaming computation by 20-30%.

Communication

After the streaming function, some of the DF's move out of their domains. In the communication function, DFs in the buffer lattice sites migrate to proper destination domains. Figure 1 shows a schematic of the domain decomposition consisting of four sub-domains Ω0-Ω3. We employ a 6-way dead-lock free

communication scheme, in which data transfer is completed in 6 steps. The inter-domain communication is implemented with MPI.

2.2.2 Parallel LBM Algorithm on CUDA

Modern GPUs contain hundreds of arithmetic units to provide tremendous acceleration for numerically intensive scientific applications. The high-level GPU programming language, CUDA, has made this computational power more accessible to computational scientists. We have implemented the pLBM algorithm on GPUs using CUDA.

Specifically, we test NVIDIA's GeForce 8800 GTS that contains 96 stream processors. A block of threads (of which the number of threads per block is user-defined) is processed by each stream processor, and while each block allows its threads to access fast shared memory and communicate with each other, blocks do not have any method of synchronizing with other blocks. Fortunately, the LBM allows each lattice site to be solved independently of all the others during the collision phase, and it only requires knowledge of other subdomains during the streaming phase.

We test the performance of CUDA by implementing both the streaming and collision calculations independent of the CPU, that is, having each time step run entirely on the GPU.

The GPU-based pLBM algorithm consists of two parts as shown in Table 2. First, data is initialized and transferred to the GPU. Then, the collision and streaming computation kernels are executed until the simulation ends. The kernels are designed so that each thread solves one lattice site at a time. Subdomain size (i.e., the number of blocks used) is taken to be the size of the system divided by the block size. Thread management and synchronization, often a potential for major problems in large simulations, is facilitated through CUDA's _syncthreads() function. Because of CUDA's C-like syntax and behavior as well as allowing the GPU to perform the collision and streaming functions entirely, the resulting code strongly resembles that of uniprocessor-based code. Moreover, since the data and programs are small, we put all of them to the memory of GPU.

Table 2. GPU-based pLBM algorithm.

Steps:	
1	initialize data to send to GPU
2	define block and thread parameters initialized
3	**for** each time step
4	execute collision kernel
5	execute streaming kernel

3. Experiments

3.1. Experimental Test Bed

PlayStation3 Cluster

The Sony Toshiba IBM (STI) Cell processor is the heart of the Sony PlayStation3 (PS3) video game console, whose design is intended to meet the demanding computational requirements of video games. Cell adopts a heterogeneous approach to multi-core, with one conventional processor core (Power Processing Element, PPE) to handle OS and control functions, combined with up to eight simpler SIMD cores (Synergistic Processing Elements, SPEs) for the computationally intensive work [11, 12]. The SPEs differ considerably from conventional core architectures due to their use of a disjoint software-controlled local memory instead of the conventional hardware-managed cache hierarchy employed by the PPE. Rather than using prefetch to hide latency, the SPEs have efficient software-controlled DMA engines that asynchronously fetch data from DRAM into the 256KB local store. This approach allows more efficient use of available memory bandwidth than is possible

with standard prefetch schemes on conventional cache hierarchies, but also makes the programming model more complex. In particular, the hardware provides enough concurrency to satisfy Little's Law [2] and conflict misses, while potentially eliminating write fills, however capacity misses must be handled in software.

We connect nine PlayStation3 consoles via a Gigabit Ethernet switch, where each PlayStation3 contains: (1) a 3.2 GHz 64-bit RISC PowerPC processor (PPE) with 32KB L1 and 512KB L2 caches and 256MB main memory; and (2) eight 3.2GHz 32-bit SPEs with 256KB of local store (LS) and Memory Flow Controller (MFC). The PPE, SPEs, and main memory are interconnected by a fast internal bus called the Elemental Interface Bus (EIB), with the peak bandwidth of 2,048GB/s, while the memory and I/O interface controller (MIC) supports a peak bandwidth of 25GB/s inbound and 35GB/s outbound. Each PlayStation3 has a Gigabit Ethernet port.

We have installed a Fedora Core 6 Linux OS distribution with libraries and infrastructure to support the IBM Cell Software Development Kit (SDK) version 2.1. The SDK offers an IBM compiler and the GNU compiler collection for the Cell processor. Message Passing Interface (MPI) is installed as in a standard Linux cluster. We use the Cell SDK for instruction-level profiling and performance analysis of the code. The code is compiled using GNU C compiler (gcc) with optimization option '-O3' and MPI version 1.2.6.

NVIDIA CUDA-GPU platform

The GeForce 8800 GTS contains 96 stream processors running at 1.35 GHz. While the memory clock is 800MHz and the memory size is 640MB, the memory interface is 320bit and the memory bandwidth is 64GB/sec.

The system used to run the simulation consists of an AMD Sempron 3500+ CPU with 1 GB of RAM and an NVIDIA GeForce 8800 GTS. The operating system used is Fedora Core 7, kernel version 2.6.23.1-21.fc7. NVIDIA's CUDA is used to develop the application, and nvcc version 1.0 is used to compile the software.

3.2 Performance Test Results

3.2.1 Performance of PlayStation3 Cluster

We first test the intra-processor scalability of pLBM based on multithreading on a single Playstation3 console. Figure 3(a) shows the running time for the collision function as a function of the number of SPEs, S, from 1 to 6 for a simulation system with 64^3 lattice sites. Figure 3(b) shows the corresponding strong-scaling speed-up, i.e., the running time on a single SPE divided by that on S SPEs. The algorithm scales nearly linearly with the number of SPEs. On 6 SPEs, the speed-up is 5.29, and the parallel efficiency (defined as the speed-up divided by S) is 0.882.

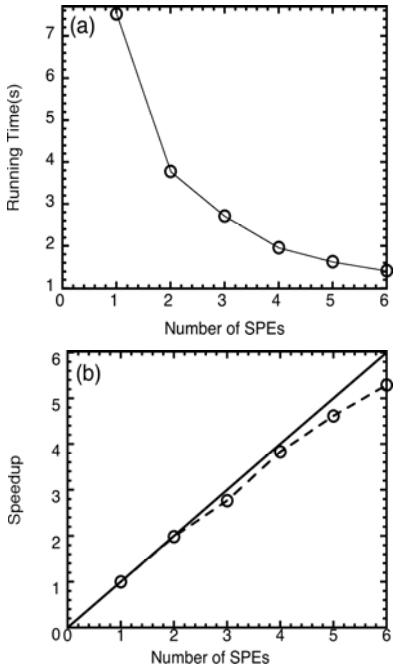

Fig. 3. (a) Running time for the pLBM flow simulation involving 64^3 lattice sites on a single PlayStation3 console as a function of the number of SPEs. (b) Strong-scaling speed-up of the pLBM algorithm (circles) on a single PlayStation3 console as a function of the number of SPEs. The solid line shows the ideal speed-up.

Then we implemented our pLBM both on the PlayStation3 cluster as well as on a Xeon cluster to assess the comparative performance of the PlayStation3 cluster over a conventional Linux cluster. The latter is composed of 256 computation nodes, with each node having two processors of 2.8GHz and 512KB L1 cache, where the nodes are connected via 4 Gbit/s Myrinet. (We make comparison using one node.) In Fig. 4(a), the running time of PlayStation3 is compared with that of Xeon for various system sizes from 8^3 to 64^3 lattice points. Figure 4(b) shows the speed-up of PlayStation3 over Xeon.

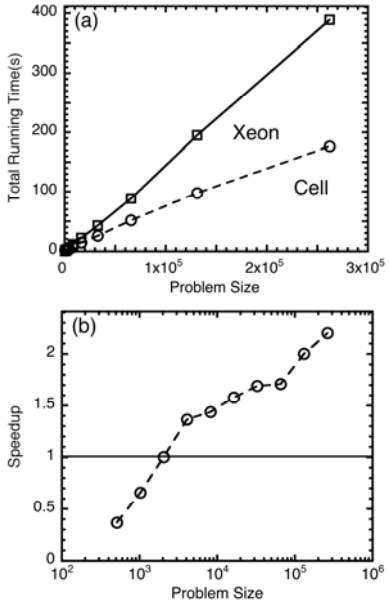

Fig. 4. (a) Total running time of the pLBM flow simulation on Xeon and Cell platforms with different problem sizes. (b) Speed-up of the pLBM flow simulation of Cell over Xeon with different problem sizes.

To study the performance of individual functions of the pLBM, Figs. 5(a), 6(a) and 7(a) show the running time of the collision function, the streaming function and the communication part on the PlayStation3 cluster and the Xeon cluster for various problem sizes. In addition, Figs 5(b), 6(b) and (b) show the corresponding speed-ups of the PlayStation3 cluster over the Xeon cluster.

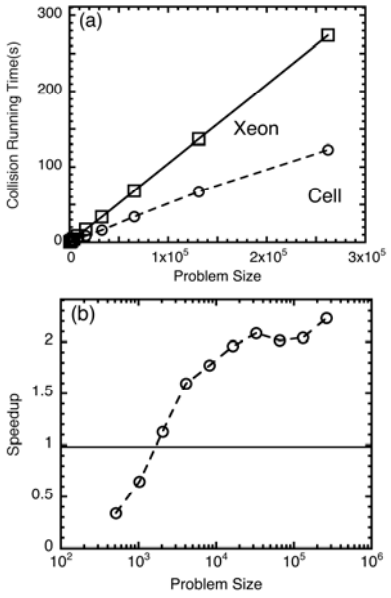

Fig. 5. (a) The running time of the collision function for pLBM flow simulation on Xeon and Cell platforms with different problem sizes. (b) Speed-up of the collision function for pLBM flow simulation of Cell over Xeon with different problem sizes.

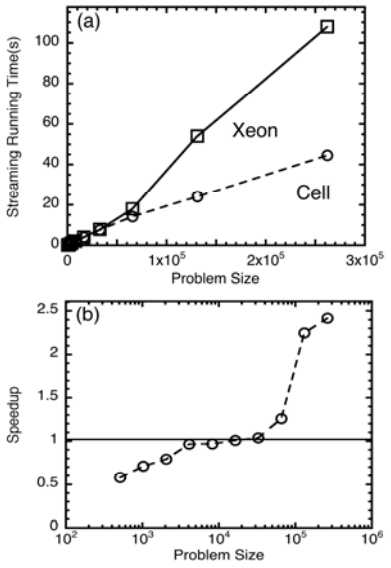

Fig. 6. (a) The running time of the streaming function for pLBM flow simulation on Xeon and Cell platforms with different problem sizes. (b) Speed-up of the streaming function for pLBM flow simulation of Cell over Xeon with different problem sizes.

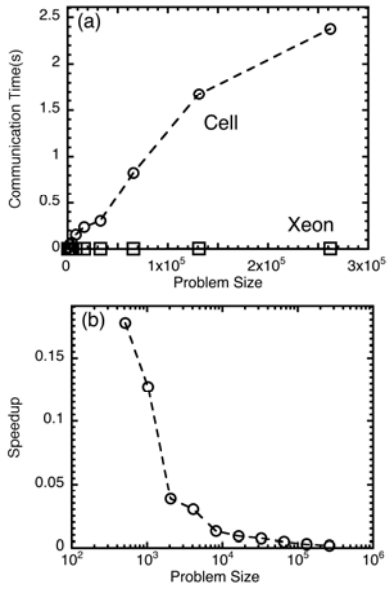

Fig. 7. (a) The running time of the communication part for pLBM flow simulation on Xeon and Cell platforms with different problem sizes. (b) Speed-up of the communication part for pLBM flow simulation of Cell over Xeon with different problem sizes.

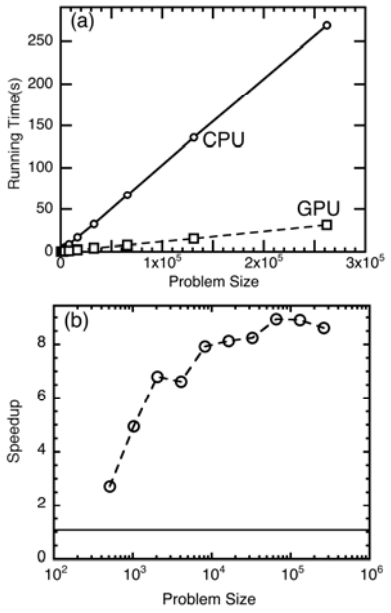

Fig. 8. (a) Total running time of pLBM flow simulation on CPU and GPU with different problem sizes. (b) Speed-up of the total pLBM flow simulation of GPU over CPU with different problem sizes.

3.2.2 Performance of CUDA-GPU platform

We have compared the comparative performance of the CUDA-GPU (NVIDIA 8800 GTS) over a conventional CPU (Sempron 3500+) for pLBM. Here, the running time of each function as well as the speed-up of GPU over CPU are measured for various problem sizes from 8^3 to 64^3 lattice points.

Figures 8(a), 9(a), and 10(a) show the running time of the entire program, the collision function, and the streaming function, respectively, on GPU and CPU. In addition, Figures 8(b), 9(b), and 10(b) show the corresponding speedup of GPU over CPU for the entire program, the collision function, and the streaming function, respectively. Figure 11 shows the running time of the preparation part of pLBM on the two platforms.

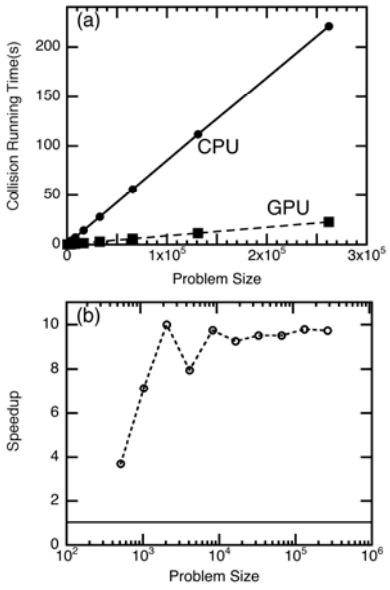

Fig. 9. (a) The running time of the collision function for pLBM flow simulation on CPU and GPU with different problem sizes. (b) Speed-up of the collision function for pLBM flow simulation of GPU over CPU with different problem sizes.

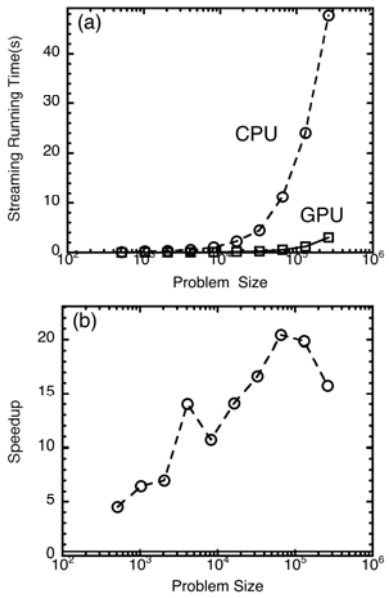

Fig. 10. (a) The running time of the streaming function for pLBM flow simulation on CPU and GPU with different problem sizes. (b) Speed-up of the Streaming function in pLBM flow simulation of GPU over CPU with different problem sizes.

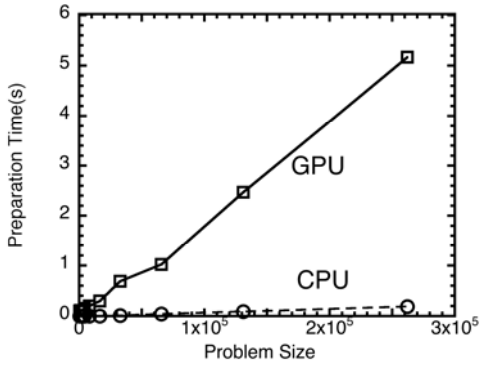

Fig. 11. Preparation time for pLBM flow simulation on CPU and GPU with different problem sizes.

4. Conclusions

From the performance test results discussed above, we can draw several conclusions. For the compute-intensive collision part, PlayStation3 outperforms the traditional Xeon cluster when the problem size is larger than 1024, and the larger the problem size, the better the performance of PlayStation3 over that of Xeon cluster. For the steaming part, which mainly deals with memory access, the performance of PlayStation3 is also better than Xeon when the problem size is larger than 2048, and the speed-up also increases with the problem size. However, for the communication part, the PlayStation3 cluster is not as good as Xeon cluster due to the limited bandwidth of the low-price Ethernet switch.

In general, the PlayStation3 cluster outperforms the Xeon cluster when the problem size is large enough in compute and memory-access intensive applications, and the performance enhancement is an increasing function of the problem size. This indicates that the DMA efficiency increases with the data size. For the largest problem size, the performance enhancement of PlayStation3 over Xeon for pLBM is 2.43.

In terms of GPU, despite the large preparation time of CUDA-GPU, the total performance of CUDA-GPU is much better than that of CPU in all problem sizes we have tested. The best speed-up we obtain is 8.76.

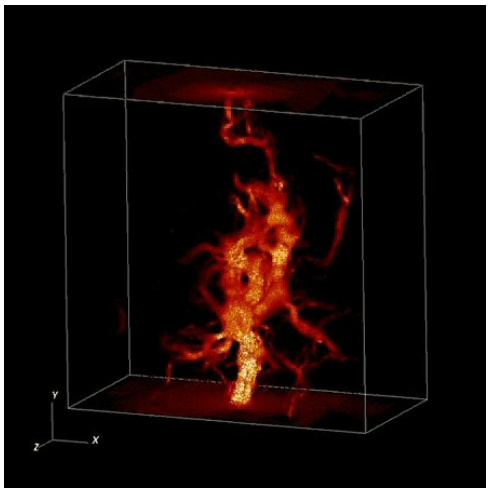

Fig. 12. Visualization of pLBM simulation of fluid flow in fractured silica on the PlayStation3 cluster, where the magnitude of the fluid velocity is color-coded.

The pLBM code has been applied to simulate fluid flow in fractured glass. Figure 12 visualizes our pLBM simulation of fluid flow through fractured silica glass, where the fractured surface is prepared through voxelation of atomistic simulation data [13]. Such flow

simulation in a complex geometry is important in many areas, e.g., for maximizing oil recovery in petroleum industry.

Acknowledgements

This work was partially supported by Chevron-CiSoft, ARO-MURI, DOE-SciDAC/BES, DTRA, and NSF-ITR/PetaApps/CSR. Numerical tests were performed using a Playstation3 cluster at the Collaboratory for Advanced Computing and Simulations and a CPU-GPU system at the Information Sciences Institute at the University of Southern California. We thank Prof. Robert Lucas and Mr. John Tran for providing access to their CPU-GPU system.

Reference

[1] Asanovic K, Bodik R, Catanzaro BC, Gebis JJ, Husbands P, Keutzer K, Patterson DA, Pishker WL, Shalf J, Williams SW, Yelick KA. The landscape of parallel computing research: a view from Berkeley. Berkeley: University of California, 2006.

[2] Shalf J. The new landscape of parallel computer architecture. J Phys: Conf Series 2007;78:012066.

[3] Buttari A, Luszczek P, Kurzak J, Dongarra J, Bosilca G. SCOP3: A Rough Guide to Scientific Computing On the PlayStation 3. Knoxville: University of Tennessee, 2007.

[4] Johns CR, Brokenshire DA. Introduction to the cell broadband engine architecture. IBM Journal of Research and Development 2007;51:503.

[5] Dongarra J, Gannon D, Fox G, Kennedy K. The impact of multicore on computational science software. CTWatch Quarterly 2007;3:11.

[6] Nakano A, Kalia RK, Nomura K, Sharma A, Vashishta P, Shimojo F, van Duin ACT, Goddard WA, Biswas R, Srivastava D, Yang LH. De novo ultrascale atomistic simulations on high-end parallel supercomputers. International Journal of High Performance Computing Applications 2008;22:113.

[7] Williams S, Carter J, Oliker L, Shalf J, Yelick K, Lattice Boltzmann simulation optimization on leading multicore platforms, International Parallel & Distributed Processing Symposium (IPDPS) (to appear), 2008.

[8] Stuermer M, Fluid simulation using the Lattice Boltzmann Method on the Cell Processor, Vortrag: Einladung, Zentralinstitut fur Angewandte Mathematik des Forschungszentrum Juelich, 11.04.2007.

[9] Stuermer M, Optimizing fluid simulation and other scientific applications on the Cell, Vortrag: Einladung vom SFB 716 der Universität Stuttgart, SFB 716, Stuttgart, 14.06.2007

[10] Li W, Wei X, Kaufman A, Implementing Lattice Boltzmann computation on graphics hardware, The Visual Computer, to appear.

[11] Ladd AJC, Verberg R. Lattice-Boltzmann simulations of particle-fluid suspensions. Journal of Statistical Physics 2001;104:1191.

[12] Bader DA, Agarwal V. FFTC: fastest Fourier transform for the IBM Cell Broadband Engine. Proceedings of the International Conference on High Performance Computing (HiPC) IEEE, 2007.

[13] Chen YC, Lu Z, Nomura K, Wang W, Kalia RK, Nakano A, Vashishta P. Interaction of voids and nanoductility in silica glass. Physical Review Letters 2007;99:155506.