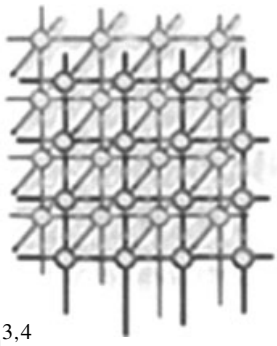
CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2010; **22**:1–14
Published online 4 September 2009 in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/cpe.1466



A flexible high-performance Lattice Boltzmann GPU code for the simulations of fluid flows in complex geometries

Massimo Bernaschi[1,*,†], Massimiliano Fatica[2], Simone Melchionna[3,4], Sauro Succi[1,5] and Efthimios Kaxiras[4]

[1]*Istituto Applicazioni Calcolo, CNR, Viale Manzoni, 30-00185 Rome, Italy*
[2]*Nvidia Corporation, 2701 San Tomas Expressway, Santa Clara, CA 95050, U.S.A.*
[3]*SOFT, Istituto Nazionale Fisica della Materia, CNR, P.le A. Moro, 2-00185 Rome, Italy*
[4]*Department of Physics and School of Engineering and Applied Sciences, Harvard University, Cambridge, MA 02138, U.S.A.*
[5]*Initiative in Innovative Computing, Harvard University, Cambridge, MA 02138, U.S.A.*

SUMMARY

We describe the porting of the Lattice Boltzmann component of MUPHY, a multi-physics/scale simulation software, to multiple graphics processing units using the Compute Unified Device Architecture. The novelty of this work is the development of *ad hoc* techniques for optimizing the indirect addressing that MUPHY uses for efficient simulations of irregular domains. Copyright © 2009 John Wiley & Sons, Ltd.

Received 28 November 2008; Revised 4 March 2009; Accepted 11 May 2009

KEY WORDS: Lattice Boltzmann; multi-physics; GPU; MPI

1. INTRODUCTION

Recently, there has been renewed interest in floating point *accelerators* and *co-processors*, which can be defined as devices that carry out arithmetic operations concurrently with or in place of the CPU. Two solutions received special attention from the high-performance computing community: (i) the *Cell* processor, designed and developed jointly by IBM, Sony and Toshiba and (ii) Graphics

*Correspondence to: Massimo Bernaschi, Istituto Applicazioni Calcolo, CNR, Viale Manzoni, 30-00185 Rome, Italy.
†E-mail: m.bernaschi@iac.cnr.it

Contract/grant sponsor: Innovative Computing of Harvard University

Processing Units (GPU), originally developed for video cards and graphics, that are now used for very demanding computational tasks. On both platforms, astonishing results have been reported for a number of applications covering, among others, atomistic simulations, fluid–dynamic solvers, and option pricing.

In this paper we describe the porting to NVIDIA GPUs of the fluid–dynamics component of MUPHY, our MUlti/scale PHYsics software for the simulation of complex bio-physical phenomena. The final result is a very flexible code that thanks to the MPI support for using multiple GPUs and the indirect addressing scheme can simulate very large arbitrary geometries that simply do not fit in memory using other existing GPU implementations.

2. THE MUPHY CODE

In this section, we briefly introduce MUPHY, the code we developed for multi-physics, multi-scale simulations of particles embedded and interacting with fluids (for further details see [1–3]). Our application is targeted toward the simulation of bio-fluidic systems, large molecules in fluid flows, and hemodynamics. Typically, these problems require massive computational resources to tackle extended computational domains. In contrast to the vast majority of fluid–dynamic solvers, our work is based on a *mesoscopic/hydrokinetic* representation of the fluid, via the so-called Lattice Boltzmann (LB) method [4–6].

2.1. The LB method

The LB equation is a minimal form of the Boltzmann kinetic equation in which all details of molecular motion are removed except those that are strictly needed to recover hydrodynamic behavior at the macroscopic scale (mass–momentum and energy conservation). The result is an elegant equation for the discrete distribution functions $f_i(\mathbf{x}, t)$ describing the probability of finding a particle at Lattice site $\mathbf{x}$ at time t with speed $\mathbf{v}=\mathbf{c}_i$. More specifically, since we are dealing with nanoscopic flows, in this work we shall consider the fluctuating LB equation that takes the following form:

$$f_i(\mathbf{x}+\mathbf{c}_i\Delta t, t+\Delta t) = f_i(\mathbf{x}, t) - \omega\Delta t\,(f_i - f_i^{eq})(\mathbf{x}, t) + F_i\Delta t + S_i\Delta t \tag{1}$$

where $f_i(\mathbf{x}, t)$ represents the probability of finding a fluid particle at spatial location $\mathbf{x}$ and time t with discrete speed $\mathbf{c}_i$. The particles can move only along the links of a regular Lattice defined by the discrete speeds, so that the synchronous particle displacements $\Delta\mathbf{x}_i = \mathbf{c}_i\Delta t$ never take the fluid particles away from the Lattice. For the present study, the standard three-dimensional 19-speed Lattice is used [4–6] (see Figure 1). The right-hand side of Equation 1 represents the effect of intermolecular solvent–solvent collisions through a relaxation toward local equilibrium, f_i^{eq}, typically a second-order (low Mach) expansion in the fluid velocity of a local Maxwellian with speed $\mathbf{u}$

$$f_i^{eq} = w_i\rho\left\{1 + \beta\mathbf{u}\cdot\mathbf{c}_i + \frac{\beta^2}{2}[\mathbf{uu}:(\mathbf{c}_i\mathbf{c}_i - \beta^{-1}\overleftrightarrow{I})]\right\} \tag{2}$$

where $\beta = m_f/k_BT_f$ is the inverse fluid temperature (with k_B the Boltzmann constant), w_i is a set of weights normalized to unity, and $\mathbf{I}$ is the unit tensor in configuration space. The relaxation

DOI: 10.1002/cpe

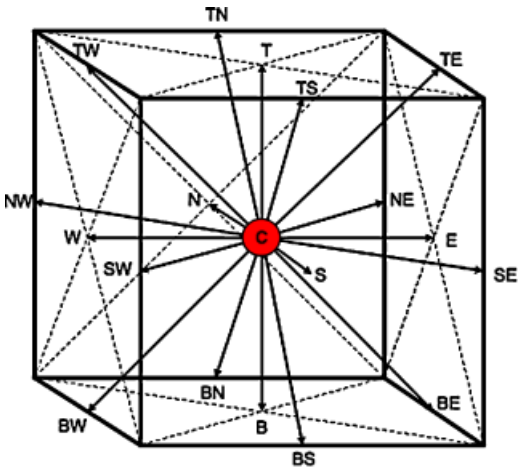


Figure 1. The directions of the fluid populations according to the D3Q19 scheme. The arrows represent a set of 18 discrete velocities (plus one null vector corresponding to particles at rest).

frequency ω controls the fluid kinematic viscosity v through the relation

$$v = c_s^2(1/\omega - \Delta t/2) \tag{3}$$

where c_s is the sound speed in the solvent [7]. Knowledge of the discrete distributions f_i allows the calculation of the local density ρ, flow speed $\rho\mathbf{u}$, and momentum–flux tensor $\overleftrightarrow{P}$, by a direct summation upon all discrete distributions:

$$\rho(\mathbf{x}, t) = \sum_i f_i(\mathbf{x}, t) \tag{4}$$

$$\rho\mathbf{u}(\mathbf{x}, t) = \sum_i f_i(\mathbf{x}, t)\mathbf{c}_i \tag{5}$$

$$\overleftrightarrow{P}(\mathbf{x}, t) = \sum_i f_i(\mathbf{x}, t)\mathbf{c}_i\mathbf{c}_i \tag{6}$$

The diagonal component of the momentum–flux tensor gives the fluid pressure, while the off-diagonal terms give the shear–stress. Unlike in hydrodynamics, both quantities are available locally and at any point in the simulation.

Thermal fluctuations are included through the source term F_i, which reads as follows (index notation):

$$F_i = w_i\rho\{F^{(2)}_{ab}(c_{ia}c_{ib} - \beta^{-1}\delta_{ab}) + F^{(3)}_{abc}g_{iabc}\} \tag{7}$$

where $F^{(2)}$ is the fluctuating stress tensor (a 3×3 stochastic matrix). Consistency with the fluctuation–dissipation theorem at *all* scales requires the following conditions:

$$\langle F^{(2)}_{ab}(\mathbf{x}, t)F^{(2)}_{cd}(\mathbf{x}', t')\rangle = \frac{\gamma k_B T}{m}\Delta_{abcd}\delta(\mathbf{x} - \mathbf{x}')\delta(t - t') \tag{8}$$

DOI: 10.1002/cpe

where Δ_{abcd} is the fourth-order Kronecker symbol [8]. $F^{(3)}$ is related to the fluctuating heat flux and g_{iabc} is the corresponding basis in kinetic space, essentially a third-order Hermite polynomial (full details are given in [9]).

The particle–fluid back reaction is described through the source term S_i, which represents the momentum input per unit time due to the reaction of the particle on the fluid population f_i.

The hydrodynamic equations are then obtained by following the usual Chapman–Enskog adiabatic procedure. First, we Taylor-expand the streaming operator to second order in the time step, thereby generating a set of first-order partial differential equations for the discrete populations f_i. These equations are then projected on the first two kinetic moments, by summing upon $\sum_i \ldots$ and $\sum_i \mathbf{c}_i \ldots$, respectively. This generates a continuity equation for the fluid density ρ and an evolution equation for the fluid current $\mathbf{J} = \rho\mathbf{u}$. This latter equation involves the divergence of the momentum–flux tensor $\overleftrightarrow{P}$, which is still an unknown quantity at this stage. This equation is closed by assuming that the momentum–flux tensor is adiabatically enslaved to its equilibrium value, which is entirely specified in terms of the fluid density and current. As a result, one is left with a self-consistent set of partial differential equations for ρ and $\mathbf{u}$, which takes the familiar form of a continuity plus Navier–Stokes equations

$$\partial_t \rho + \nabla \cdot \rho\mathbf{u} = 0 \tag{9}$$

$$\partial_t \rho\mathbf{u} + \nabla \cdot \rho\mathbf{u}\mathbf{u} = -\nabla P + \nabla \cdot \nabla(\mu\mathbf{u}) + \rho\mathbf{F} \tag{10}$$

where $P = \rho c_s^2$ is the fluid pressure, $\mu = \rho\nu$ is the dynamic viscosity, and $\mathbf{F}$ is the volume force acting on the fluid.

The use of an LB solver for the simulation of a fluid solvent is particularly well suited because of the following reasons:

(i) Free-streaming proceeds along straight trajectories. This is in stark contrast with hydrodynamics, in which the fluid momentum is transported by its own space–time-varying velocity field. Besides securing exact conservation of mass and momentum of the numerical scheme, this also greatly facilitates the imposition of geometrically complex boundary conditions.
(ii) The pressure field is available locally, with no need to solve any (computationally expensive) Poisson problem for the pressure, as in standard hydrodynamics.
(iii) Unlike hydrodynamics, diffusivity is not represented by a second-order differential operator, but it emerges instead from the first-order LB relaxation–propagation dynamics. The result is that the kinetic scheme can march in time steps that scale linearly, rather than quadratically, with the mesh resolution. This facilitates high-resolution down-coupling to atomistic scales.
(iv) Solute–solute interactions preserve their local nature, since they are explicitly mediated by the solvent molecules through direct solvent–solute interactions. As a result, the computational cost of hydrodynamic interactions scales only linearly with the number of particles (no long-range interactions).
(v) Since *all* interactions are local, the LB scheme is ideally suited to parallel computing.

Here, we do not describe the details of the molecular dynamics part of MUPHY, since the focus of the paper is on the porting of the LB part to the GPU, but we highlight that the LB/MD coupling in our approach is much tighter than for current Navier–Stokes/MD hybrid schemes [2].

DOI: 10.1002/cpe

3. GPU AND CUDA

The features of the NVIDIA graphics hardware and the related programming technology named CUDA are thoroughly described in the NVIDIA documentation [10]. Here, we report just the key aspects of the hardware and software we used. Most of the development has been carried out using as GPU the video card of a Mac Book Pro (an NVIDIA GeForce 8600 M GT with a clock rate of 750 MHz). For performance tests we had access to a NVIDIA Tesla C870 equipped with 16 multiprocessors with 8 processors each, for a total of 128 computational cores that can execute at a clock rate of 1.3 GHz. The processors operate integer types and a 32-bit floating point types (the latter are compliant with the IEEE 754 single-precision standard). Each multiprocessor has a memory of 16 KB size shared by the processors within the multiprocessor. Access to data stored in the shared memory has a latency of only two clock cycles allowing for fast non-local operations. Each multiprocessor is also equipped with 8192 32-bit registers.

The total on-board *global* memory on the Tesla C870 amounts to 1.5 GByte with a 384-bit memory interface to the GPU that delivers 76.8 GB/s memory bandwidth. The latency for the access to this *global* memory is approximately 200 cycles (two order of magnitude slower than access to shared memory) with any location of the global memory visible by any thread, whereas shared memory variables are *local* to the threads running within a single multiprocessor.

For the GPU programming, we employed the CUDA Software Development Toolkit, which offers an *extended* C compiler and is available for all major platforms (Windows, Linux, Mac OS). The extensions to the C language supported by the compiler allow starting computational kernels on the GPU, copying data back and forth from the CPU memory to the GPU memory and explicitly managing the different types of memory available on the GPU. The programming model is a single instruction multiple data type. Each multiprocessor is able to perform the same operation on different data 32 times in two clock cycles, so the basic computing unit (called *warp*) consists of 32 threads. To ease the mapping of data to threads, the threads identifiers may be multidimensional and, since a very high number of threads run in parallel, CUDA groups threads in *blocks* and *grids*. Since for our purposes a one-dimensional mapping is sufficient, as detailed in the next section, we do not describe these mechanisms (the interested reader may refer to [11] for additional information).

One of the crucial requirements to achieve a good performance on the NVIDIA GPU is that *global* memory accesses (both read and write) should be coalesced. This means that a memory access needs to be *aligned* and coordinated within a group of threads. The basic rule is that the thread with id n ($\in 0, \ldots, N-1$) should access element n at byte address $StartingAddress + sizeof(type) * n$, where $sizeof(type)$ is equal to either 4, 8, or 16 and `StartingAddress` is a multiple of $16 * sizeof(type)$. Figure 2 illustrates an example of *coalesced* memory access, whereas Figure 3 illustrates a few examples of possible *uncoalesced* memory access. The overhead due to an *uncoalesced* memory access can have dramatic effects on the global performance of the GPU since the effective memory bandwidth can degrade up to one order of magnitude. Some of our tests were performed on a new generation of GPUs (GT200). While the new hardware has better coalescing capability, the performance difference between fully coalesced memory accesses and uncoalesced accesses is still remarkable and it is good programming practice to maximize the memory bandwidth.

Functions running on a GPU with CUDA have some limitations: they cannot be recursive; they do not support *static* variables; they do not support variable number of arguments; function pointers are meaningless. Nevertheless, CUDA makes GPU programming much more simple as compared

DOI: 10.1002/cpe

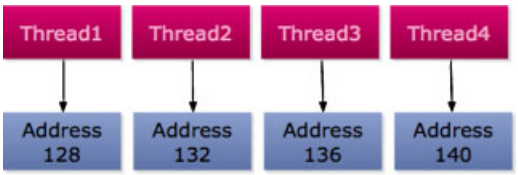


Figure 2. An example of *coalesced* memory access (by Johan Seland).

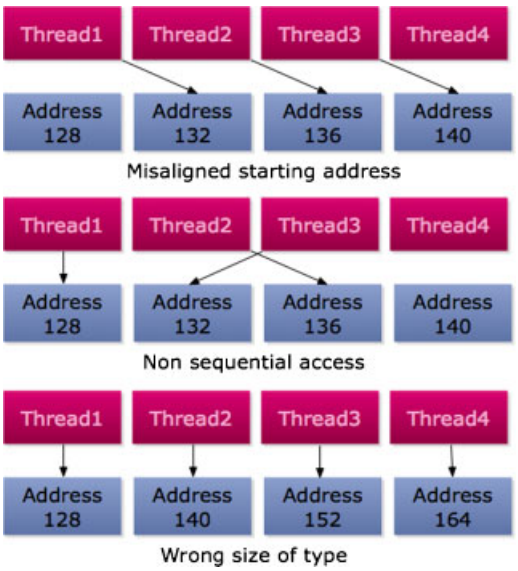


Figure 3. Examples of *uncoalesced* memory access (by Johan Seland).

with the approach described in [12], where the LB Method was implemented by using directly the graphics operations provided by the hardware.

4. PORTING THE LB COMPONENT OF MUPHY TO THE GPU

MUPHY was originally developed in Fortran 90 and the original parallelization was done with MPI, offering high portability among different platforms and a good performance due to the availability of highly tuned implementations.

The parallelization of the LB method and molecular dynamics algorithms, as two distinct and *separate* components, has been extensively studied for a number of years [13–19]. However, the coupling of these techniques raises new issues that need to be addressed in order to achieve scalability and efficiency for large-scale simulations. We approached the task by starting from a serial version of the combined code, instead of trying to combine two existing parallel versions.

DOI: 10.1002/cpe

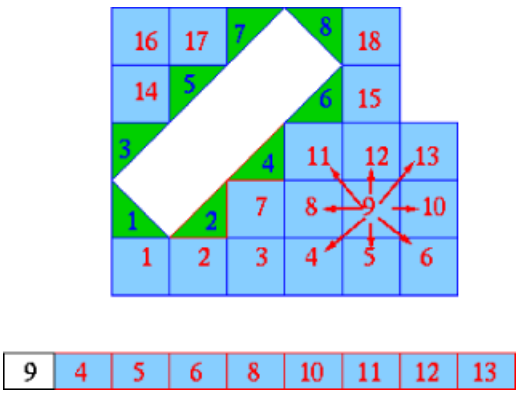


Figure 4. Entries in the connectivity matrix for fluid node 9 in a simplified two-dimension case with an irregular domain. Fluid nodes are in light grey (light blue for the online version) and wall nodes are in dark grey (green for the online version). Solid nodes (not numbered since they are not stored) are in white.

The original LB code was initially optimized to take advantage of: (i) removal of redundant operations; (ii) buffering of multiply used operations; and (iii) 'fusion' of the collision and streaming steps in a single loop. This latter technique, already in use with other high-performance LB codes, significantly reduces the data traffic between memory and processor, since there is only one read and one store of all LB populations at each time step.

In variance with other LB kernels recently ported to the GPU [11,20], in MUPHY nodes are not stored in *full matrix* mode (i.e. rigid geometrical order), but according to a linearized indirect addressing scheme [21,22]. Each node of the LB lattice is labeled with a *tag* that identifies it as belonging to a specific subregion of the computational domain, (i.e. fluid, wall, inlet, outlet, solid), as read from an input file. It is possible to define a *default* value for the tag so that nodes that are not explicitly tagged are assumed to have that value for the tag (this assumption reduces significantly the size of the input file that describes the computational domain). Nodes are grouped according to their features in several one-dimensional arrays, so that there is an array of fluid nodes, an array of *wall* nodes, an array of *inlet* nodes, etc. with the exception of *solid* nodes that are not stored at all since they refer to inactive regions of the domain.

As a consequence, *homogeneous* nodes (i.e. all fluids nodes, all wall nodes, etc.) are contiguous in memory regardless of their geometrical distance. This type of data organization requires, for each node, an additional data structure (*connectivity matrix*) that contains the list of all positions, within the above mentioned one-dimensional arrays, of its neighboring nodes (see Figure 4 for a simple two-dimension case).

At a first glance such mechanism may appear a waste in terms of efficiency and memory storage. However, since only the nodes playing an active role in the LB dynamics need to be accessed and stored in memory, it provides huge savings in storage requirements and simulation times [23] for most (non-trivial) geometries. An indirect addressing scheme allows to support very flexible domain decomposition strategies, a fundamental requirement for a good load balancing among computational tasks. As a matter of fact, MUPHY supports all possible Cartesian decompositions

DOI: 10.1002/cpe

(along X, Y, Z, XY, XZ, YZ, XYZ) and *custom* decompositions, e.g. those produced by tools like METIS [24], which are necessary for irregular domains.

For the molecular dynamics section, a parallelization strategy suitable for the particle–fluid coupling has been developed, taking into account that for multiscale MD applications the spatial distribution of particles is typically highly inhomogeneous. A straightforward approach to achieve a good load balancing is to resort to a domain decomposition such that each task has (approximately) the same number of particles. In this way, the size of the spatial subdomains assigned to each task may vary significantly. In a stand-alone molecular dynamics simulation, this is acceptable, but in our case the LB component would be hindered, since the computational load is proportional to the size of the spatial domain assigned to each task. One might opt for two separate domain decompositions for the LB and the MD engines. However, the exchange of momentum among particles and the surrounding fluid would become a non-local operation which, specially on a distributed memory platform, might represent a serious communication bottleneck. In the end, we decided to resort to a domain decomposition strategy where the parallel subdomains coincide with those of the LB scheme. Each computational task performs both the LB and MD calculations and the interactions of the particles with the fluid are completely localized (there is no need to exchange data among tasks during this stage). To compensate the resulting unbalance of the MD computational load, we resort to a *local dynamic* load balancing algorithm described in [3].

The porting of the LB component of MUPHY to the GPU entailed some additions to the original code, activated by simple pre-processor compiler directives. First of all, the routines in charge of the LB update have been ported from Fortran 90 to CUDA. In the resulting code there is an additional initialization routine that starts the GPU and copy all necessary data from the main memory to the GPU *global* memory. The initial fluid populations and the connectivity matrix required by the indirect addressing scheme constitute most of these data. To follow the *best practices* of CUDA programming with respect to alignment and memory access by threads, it is necessary to modify the order of these variables in the copy from the main memory to the GPU global memory. As a consequence, the fluid populations of a lattice site are not contiguous in the GPU global memory. Besides the fluid populations and the connectivity matrix, it is necessary to transfer to the GPU some *read-only* variables (like the external force, if any, acting on the fluid) and pointers to the main memory locations where hydrodynamic variables need to be stored. This represents a crucial point and deserves additional explanations. The fluid populations once uploaded on the GPU memory do not need to be copied back to the main memory unless a dump of the whole configuration is required (MUPHY offers this functionality to support check-pointing of long simulations). However, hydrodynamic variables have to be written back to the main memory much more frequently since they define the interaction of the fluid with the particles (and vice versa). Fortunately, the number of hydrodynamic variables per lattice site is small compared with the number of fluid populations (in our case there are four hydrodynamic variables versus 19 fluid populations) so the run-time overhead of the copy from the GPU-memory to the CPU-memory is small compared with the initialization overhead.

The *read-only* variables and all values required during the *collision* phase of the LB update that can be precomputed (e.g. the quadrupole tensor $\mathbf{c}_i\mathbf{c}_i$ in Equation 2) are stored in the so-called *constant* memory of the GPU. This is as fast as reading from a register as long as all threads read the same address (as it is in our case). After the initialization, the GPU carries out the collision and streaming phases of the LB update in a single primitive that splits the computational load in

DOI: 10.1002/cpe

a number of block and threads defined via an external input file. Each thread loads all 19 fluid populations of a lattice site from the global memory of the GPU to local variables. Besides the populations, also the entries corresponding to a given lattice site in the connectivity matrix are loaded from the GPU global memory to local variables. All these load operations are aligned and *coalesced*, that is they have an optimal layout according to the GPU memory model. All operations required by the collision phase of the LB update are performed using the local copy of the fluid populations and constant memory locations. When the local copy of all fluid populations of a lattice site is updated, they are copied back to the GPU global memory using the 'double buffer' solution to maintain the current and the new configuration [25]. However, these store operations are not necessarily aligned and *coalesced* since their target location is defined by the connectivity matrix of the lattice site (these scattered write operations implement the *streaming* phase of the LB update). The problem is intrinsic to the LB scheme and not related to the indirect addressing scheme, as reported also in [11], where a simple direct addressing scheme based on the geometrical order is used. However, in case of direct addressing, it is possible to alleviate the problem by leveraging the GPU shared memory as a sort of temporary buffer for the store operations that would cause uncoalesced memory accesses [11]. This approach cannot be applied to our case since the access pattern depends on the specific geometry of the fluid domain, that is a simple knowledge of the dimensions of the *bounding box* domain is not sufficient to determine where the memory locations of the neighbors of a lattice site are located. We describe in Section 5 the impact of the *uncoalesced* store operations on the global performance of the LB scheme. The function that computes the value of the hydrodynamic variables is the last component of the GPU porting of the LB scheme. In this case all memory operations are 'local' meaning that only the fluid populations of a lattice site are required and that the resulting hydrodynamic variables belong to the same lattice site. As a consequence, there are no *uncoalesced* memory accesses.

5. RESULTS

In this section we present performance results obtained for the LB component of MUPHY ported on the GPU. We carried out three tests: the first set aimed at measuring the performance of our LB implementation with respect to the peak performance of the GPU; the second one aimed at comparing the performance of the LB on the GPU with two standard CPUs; the last set tested the performance of a small *cluster* of GPUs. The results of the first set of tests, carried out on a Tesla C870 board, are shown in Table I.

As a *baseline* benchmark, we measured the time required for a simple load/store of the 19 fluid populations with no floating point operations and with the store operations executed on the same lattice site (that is without using the information about the real target nodes available in the connectivity matrix). We define these as 'local' store operations to distinguish them from 'scattered' store operations that require the connectivity matrix. To measure the actual number of load and store operations, we resorted to the CUDA Visual Profiler, a tool recently released by NVIDIA, that classifies each load/store operation as Coherent, if it determines a *coalesced* access (see Section 3) or Incoherent, if, for any of the reasons described in Section 3, the access to the global memory is *uncoalesced*. Given the data organization of MUPHY, all load operations are coherent and that is the reason why we do not report the number of incoherent load operations.

DOI: 10.1002/cpe

Table I. Information provided by the CUDA profiler about the execution of 1000 iterations on a lattice with 55 800 fluid nodes ($62 \times 30 \times 30$).

Kernel operations	GPU time	Global load coherent	Global store coherent	Global store incoherent
load/local store	0.15	8 283 088	33 132 352	0
load/scattered store	0.9	16 131 112	1 743 904	251 093 664
load/floatingpoint/localstore	0.33	8 285 634	33 142 536	0
load/floatingpoint/scatt. store	1.0	16 132 222	1 744 024	251 110 368

Here, 'Coherent' means that the operation determines a *coalesced* memory access. 'Incoherent' corresponds to any of the situations described in Section 3 that determines an *uncoalesced* memory access.

For the simple load/store kernel the execution time for 1000 iterations on a small lattice with 55 800 fluid nodes is 0.15 s, corresponding to an effective bandwidth of $\sim$57 GB/s, that is $\sim$75% of the peak bandwidth. This result is obtained with 128 blocks of 64 threads. As shown in [11], by tuning the number of blocks and threads, the bandwidth may change quite significantly. However, what matters for our purpose is to compare this result with those obtained by adding the other functionalities of the LB update. The next step is to modify the local store operations with the scattered ones. To this purpose it is necessary to load the 18 values of the connectivity matrix for each node (the population at rest does not require an entry in the connectivity matrix since it remains always on the same node). As expected, there is now a large number of *uncoalesced* store operations and the total execution time for 1000 iterations jumps to 0.9 s ($\sim$6 times higher than before), corresponding to an effective bandwidth of $\sim$14 GB/s, since there are now 18 more load operations per lattice site. This means $\sim$20% of the peak memory bandwidth of the GPU. Next, we added the floating point operations required by the collision phase of the LB update, but we stored the results as in the first test case, that is without using the information in the connectivity matrix. The resulting time for 1000 iterations is 0.33 s with no *uncoalesced* store operations. On this computational kernel it is possible to measure the performance in terms of Millions of Fluid Lattice UPdates per Seconds (MFLUPS, a standard performance measure for LB codes). For this kernel the number of MFLUPS is $\sim$170. This is an impressive result since, on high-end standard processors, the number of MFLUPS, even for highly tuned LB implementations, is in the range of 10 MFLUPS [25] for codes based on the full matrix representation. Obviously, this computational kernel cannot be used for real work since the store operations do not take into account the connectivity matrix. Once the direct store operations are replaced with the scattered ones that take into account the actual connectivity, we recover the complete LB computational kernel. The time for 1000 iterations is now 1.0 s, that is about three times the time required by the computational kernel with direct store operations and corresponds to $\sim$56 MFLUPS. Two considerations are in order. First of all, the second kernel is six times slower than the first kernel, whereas the fourth kernel (which implements the full LB method) is only three times slower than the third kernel. Since the difference between the first and the second kernel is the same as the difference between the third and the fourth kernel (that is the first and the third kernel execute direct store operations, whereas the second and the fourth kernel execute scattered store operations that cause *uncoalesced* memory accesses), it is clear that the floating point operations of the fourth kernel hide most of the latency of the store operations and reduce significantly the penalty due to *uncoalesced* memory accesses that remain, however, quite high. We discuss in Section 6 some of the possible alternatives for either reducing the number of

DOI: 10.1002/cpe

uncoalesced memory accesses or raising the overlap between computation and memory operations in order to further improve the performance of the LB update. The second observation is that these results are obtained on a small lattice with no special alignment characteristics ($62 \times 30 \times 30$). On the other hand, most of the results presented in previous implementations [11] were obtained on domains whose linear dimensions are the multiple of 16 or 32 to facilitate the mapping of the domain with the lowest number of threads required for optimal usage (that is 32 threads, the *warp*, in CUDA jargon, mentioned in Section 3). It is unclear what is the performance of those implementations for domains whose dimensions are not multiples of 16. For our implementation, if the linear dimensions of the domain are a multiple of 16, there is an improvement of $\sim$40% of the performance which reaches, for a $64 \times 32 \times 32$ Lattice, the value of $\sim$80 MFLUPS.

The second set of tests was carried out to compare the performance of the LB update on two different GPUs with our highly tuned version of LB update for standard multi-core CPUs. The first GPU was the GeForce 8600 M GT, which equips as video card a Mac Book Pro having an Intel Core 2 Duo at 2.4 GHz. The main features of this GeForce 8600M are: clock rate equal to 750 MHz; total global memory 256 MBytes; total number of registers per block 8192. We ran two test cases: (i) a small regular domain $64 \times 32 \times 32$ and (ii) an irregular geometry (shown in Figure 5) representing a human coronary artery having a bounding box $89 \times 234 \times 136$. It is interesting to note

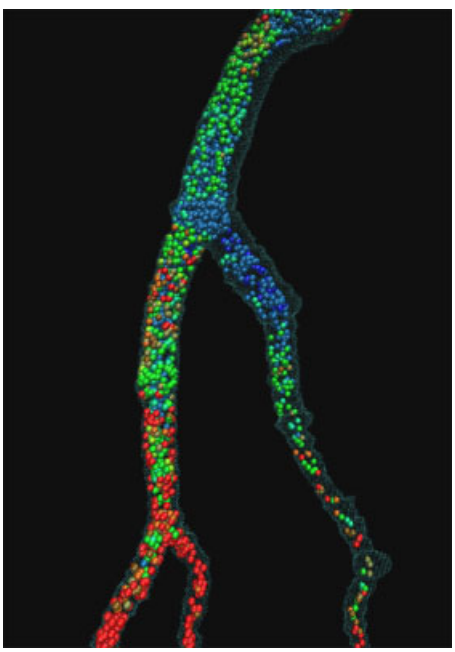


Figure 5. Blood flow pattern in a human coronary artery. Spheres represent transported white cells, whose size has been magnified as compared to realistic values, to highlight their distribution inside the arterial region. The grey tone (color for the online version) of the spheres represent the Shear Stress (SS) experienced locally during blood motion: dark grey corresponds to low SS, light grey to high SS (for the online version, blue: low SS; green: intermediate SS; red: high SS).

DOI: 10.1002/cpe

Table II. Timing of 10 000 iterations on an irregular domain $1057 \times 692 \times 1446$ with $\sim$4 000 000 fluid nodes.

System	Total elapsed time (in s)	MFLUPS
1 C870 GPU	760	53
2 GT200 GPUs	159	252
8 GT200 GPUs	41.9	955

that a code using the full matrix representation could not run such test case with only 256 MBytes of memory whereas our implementation, using the indirect addressing scheme that stores only fluid, wall and inlet/outlet nodes (which all together are a small fraction of the entire domain) fits easily in 256 Mbytes (which, actually, are less than that because a part is reserved for video support). For both test cases the GeForce 8600 GPU outperforms a multi-threaded implementation running on the Intel Core 2 by a factor 4 and achieves more than 10 MFLUPS, an excellent result for the video card of a laptop. The second GPU we tested was the Tesla C870 that we compared with an Intel Xeon E5405 2.0 GHz 12 MB L2 Cache Quad-Core Processor. The GPU outperforms our multi-threaded LB implementation (that exploits all four cores of the processor) by a factor 10 for the first test case (small regular domain) and $\sim$8 for the second test case (irregular geometry).

For the third and last test, we ran our combined GPU+MPI parallel code on a *cluster* of GPUs available internally at NVIDIA composed as follows: 4 Quad core Xeon 2.8 GHz connected by Infiniband and equipped each one with two pre-production S1070 GPU systems (for a total of 8 GT200 GPUs). In the first test, we ran 1000 iterations on a regular $256 \times 128 \times 128$ domain.

The cluster of GPUs completed in 7.5 s (total elapsed time), whereas 512 nodes of an IBM BlueGene/L required 21 s to complete the same test. In the second test we used an irregular domain with a large bounding box ($1057 \times 692 \times 1446$) and a total number of fluid nodes $\simeq$4 000 000. The results are reported in Table II. It is worth to highlight that going from one C870 GPU to 8 GT200 GPUs performance improves by a factor 18. From the viewpoint of the programmer some care is required in the communication among the GPUs since there is an intermediate passage through the host CPUs. In other words, when GPU s needs to send data to GPU t, the following steps take place: (i) GPU s copies data to CPU s memory; (ii) CPU s sends data, through MPI primitives, to CPU t; and (iii) CPU t copies data from its memory to the memory of its GPU. To minimize the number of communications, a buffering mechanism has been implemented to allow sending data that have the same source and target GPUs by means of a single MPI primitive.

6. FLEXIBILITY AND FUTURE PERSPECTIVES

To the best of our knowledge, the work presented in this paper represents the first porting of an LB code using indirect addressing to a GPU using CUDA. The resulting LB computational kernel is seamlessly integrated in MUPHY, one of the first examples of high-performance parallel code for the simulation of multi-physics/scale bio-fluidic phenomena [3]. Since MUPHY is a parallel code based on MPI, it is possible to run this version on multiple GPUs provided that their host systems support MPI. Thanks to indirect addressing, MUPHY seamlessly handles real-life geometrical set-ups,

DOI: 10.1002/cpe

such as blood flows in human arteries in presence of white cells (as Figure 5 illustrates), lipids, drug-delivery carriers, and other suspended bodies of biological interest [26,27].

For the future, we plan to explore several directions for further improvements over the already excellent performance presented here. First of all, we plan to apply re-ordering techniques to the connectivity matrix to reduce as much as possible *uncoalesced* memory accesses that represent the main bottleneck. In this respect, it appears that the new generation of NVIDIA GPU (GT200) reduces significantly, but does not eliminate, the penalty due to *uncoalesced* memory accesses. As a matter of fact, our preliminary tests on the latest-generation GPUs (Tesla S1070) show that performance improves by a factor 2 with full binary compatibility. Finally, the overlap between computation and memory access will be improved by a new scheme that we are going to implement to exploit the very fast context switch capabilities of the GPUs.

ACKNOWLEDGEMENTS

MB, SS, and SM wish to thank the Initiative for Innovative Computing of Harvard University for financial support and the Harvard Physics Department and School of Engineering and Applied Sciences for their hospitality.

We thank NVIDIA Corporation for providing us a GPU for development and testing of our code and for the access to their cluster of GPUs.

REFERENCES

1. Fyta MG, Melchionna S, Kaxiras E, Succi S. Multiscale coupling of molecular dynamics and hydrodynamics: Application to DNA translocation through a nanopore. *Multiscale Modeling and Simulation* 2006; **5**:1156–1173.
2. Fyta M, Sircar J, Kaxiras E, Melchionna S, Bernaschi M, Succi S. Parallel multiscale modeling of biopolymer dynamics with hydrodynamic correlations. *International Journal for Multiscale Computational Engineering* 2008; **6**:25.
3. Bernaschi M, Fyta M, Kaxiras E, Melchionna S, Sircar J, Succi S. MUPHY: A parallel multi physics/scale code for high performance bio-fluidic simulations. *Computer Physics Communications*; DOI: 10.1016/j.cpc.2009.04.001.
4. Benzi R, Succi S, Vergassola M. The lattice Boltzmann equation: Theory and applications. *Physics Reports* 1992; **222**:145–197.
5. Wolf-Gladrow DA. *Lattice Gas Cellular Automata and Lattice Boltzmann Models*. Springer: New York, 2000.
6. Succi S. *The Lattice Boltzmann Equation*. Oxford University Press: Oxford, 2001.
7. Succi S, Filippova O, Smith G, Kaxiras E. Applying the Lattice Boltzmann equation to multiscale fluid problems. *Computing Science and Engineering* 2001; **3**:26–37.
8. Ladd AJC, Verberg R. Lattice–Boltzmann simulations of particle–fluid suspensions. *Journal of Statistical Physics* 2001; **104**:1191–1251.
9. Adhikari R, Stratford K, Cates ME, Wagner AJ. Fluctuating Lattice–Boltzmann. *Europhysics Letters* 2005; **71**:473–477.
10. Available at: http://www.nvidia.com/cuda [2009].
11. Tölke J, Krafczyk M. Towards three-dimensional teraop CFD computing on a desktop pc using graphics hardware. *Proceedings of the International Conference for Mesoscopic Methods in Engineering and Science* (*ICMMES07*), Munich, 2007.
12. Li W, Wei X, Arie E, Kaufman AE. Implementing Lattice Boltzmann computation on graphics hardware. *The Visual Computer* 2003; **19**:7–8.
13. Amati G, Piva R, Succi S. Massively parallel Lattice–Boltzmann simulation of turbulent channel flow. *International Journal of Modern Physics* 1997; **4**:869–877.
14. Boyer LL, Pawley GS. Molecular dynamics of clusters of particles interacting with pairwise forces using a massively parallel computer. *Journal of Computational Physics* 1988; **78**:405–423.
15. Heller H, Grubmuller H, Schulten K. Molecular dynamics simulation on a parallel computer. *Molecular Simulation* 1990; **5**:133–165.
16. Rapaport D. Multi-million particle molecular dynamics II. Design considerations for distributed processing. *Computer Physics Communications* 1991; **62**:198–216.

17. Brown D, Clarke JHR, Okuda M, Yamazaki T. A domain decomposition parallelization strategy for molecular dynamics simulations on distributed memory machines. *Computer Physics Communications* 1993; **74**:67–80.
18. Esselink K, Smit B, Hilbers PAJ. Efficient parallel implementation of molecular dynamics on a toroidal network: I. Parallelizing strategy. *Journal of Computational Physics* 1993; **106**:101–107.
19. Plimpton S. Fast parallel algorithms for short-range molecular dynamics. *Journal of Computational Physics* 1995; **117**:1–19.
20. Geist R, Jones Z, Steele J. Parallel processing flow models on desktop hardware. *Proceedings of the 46th ACM Southeast Conference*, Auburn, Alabama, March 2008; 417–422.
21. Dupuis A. Chopard B. Lattice gas: An efficient and reusable parallel library based on a graph partitioning technique, Sloot P, Bubak M, Hoekstra A, Hertzberger B (eds.), *HPCN*, Europe, 1999. Springer: Berlin, 1999.
22. Schulz M. Krafczyk M. Toelke J. Rank E. Parallelization strategies and efficiency of CFD computations in complex geometries using the Lattice Boltzmann methods on high-performance computers. *High Performance Scientific and Engineering Computing* (*Lecture Notes in Computational Science and Engineering*, vol. 21), Breuer M, Durst F, Zenger C (eds.). Springer: Berlin, 2002. *Proceedings of the 3rd International Fortwihr Conference on HPESC*, Erlangen, 12–14 March 2001.
23. Axner L, Bernsdorf JM, Zeiser T, Lammers P, Linxweiler J, Hoekstra AG. Performance evaluation of a parallel sparse Lattice Boltzmann solver. *Journal of Computational Physics* 2008; **227**:10.
24. Available at: http://glaros.dtc.umn.edu/gkhome/views/metis/ [2008].
25. Wellein G, Zeiser T, Donath S, Hager G. On the single processor performance of simple Lattice Boltzmann kernels. *Computers and Fluids* 2006; **35**:910–919.
26. Bernaschi M, Rybicki FJ, Melchionna S, Mitsouras D, Coskun AU, Whitmore AG. Steigner M, Nallamshetty L, Welt FG, Borkin M, Sircar J, Kaxiras E, Succi J, Stone PH, Feldman CL. Prediction of coronary artery plaque progression and potential rupture from 320-detector row prospectively ECG-gated single heart beat CT angiography: Lattice Boltzmann evaluation of endothelial shear stress. *International Journal of Cardiovascular Imaging*; DOI: 10.1007/s10554-008-9418-x.
27. Ahlrichs P, Duenweg B, Lattice–Boltzmann simulation of polymer-solvent systems. *International Journal of Modern Physics C* 1999; **9**:1429–1438. Simulation of a single polymer chain in solution by combining Lattice Boltzmann and molecular dynamics. *Journal of Chemical Physics* 1999; **111**:8225–8239.

DOI: 10.1002/cpe