Commun. Comput. Phys.
doi: 10.4208/cicp.351011.260112s

Vol. **13**, No. 3, pp. 867-879
March 2013

Developing Extensible Lattice-Boltzmann Simulators for General-Purpose Graphics-Processing Units

Stuart D. C. Walsh[1,*] and Martin O. Saar[2]

[1] *Lawrence Livermore National Laboratory, Livermore, California, USA.*[†]
[2] *Department of Earth Sciences, University of Minnesota, Minneapolis, Minnesota, USA.*

Received 31 October 2011; Accepted (in revised version) 26 January 2012

Available online 29 August 2012

Abstract. Lattice-Boltzmann methods are versatile numerical modeling techniques capable of reproducing a wide variety of fluid-mechanical behavior. These methods are well suited to parallel implementation, particularly on the single-instruction multiple data (SIMD) parallel processing environments found in computer graphics processing units (GPUs).

Although recent programming tools dramatically improve the ease with which GPU-based applications can be written, the programming environment still lacks the flexibility available to more traditional CPU programs. In particular, it may be difficult to develop modular and extensible programs that require variable on-device functionality with current GPU architectures.

This paper describes a process of automatic code generation that overcomes these difficulties for lattice-Boltzmann simulations. It details the development of GPU-based modules for an extensible lattice-Boltzmann simulation package – LBHydra. The performance of the automatically generated code is compared to equivalent purpose written codes for both single-phase, multiphase, and multicomponent flows. The flexibility of the new method is demonstrated by simulating a rising, dissolving droplet moving through a porous medium with user generated lattice-Boltzmann models and subroutines.

PACS: 47.11.-j, 07.05.Bx

Key words: Lattice-Boltzmann methods, graphics processing units, computational fluid dynamics.

1 Introduction

Lattice-Boltzmann simulations are a "bottom-up" numerical method capable of modeling a variety of complex fluid mechanical problems (for example, complex boundary condi-

*Corresponding author. *Email addresses:* walsh24@llnl.gov (S. D. C. Walsh), saar@umn.edu (M. O. Saar)
†This work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344.

tions, immiscible fluids, and heat and solute transport) that are difficult or impossible to handle with other modeling methods [1–3][‡]. Their versatility and relative ease of implementation makes lattice-Boltzmann methods particularly attractive for a wide range of applications in both science and engineering [2–4].

In addition, lattice-Boltzmann methods are readily parallelizable and are particularly suited to implementation on single-instruction multiple data (SIMD) parallel processing environments. In recent years, substantial performance increases have been achieved with lattice-Boltzmann methods by exploiting the SIMD environment in modern computer graphics processing units (GPUs) [5–8]. Possibly the first such model proposed by Li *et al*. [9] in the early 2000's achieved an impressive $50\times$ speedup over single core implementations at the time with 9.87 million lattice-node updates per second (MLUPs). Early on, significant drawbacks in the GPU programming model (reduced precision and the requirement that the algorithm be cast in terms of graphics operations), hindered the programmer's ability to develop more complex lattice-Boltzmann models, such as multiphase and multicomponent fluid flow simulations, and presented a significant barrier to widespread use of GPU-based programs [10]. In the years since, however, these barriers have been largely removed with the release of several general purpose GPU-based programming tools, such as BrookGPU [11], the ATI CTM platform [12], the Compute Unified Device Architecture (CUDA) programming model released by NVIDIA [13], and the closely related cross-platform OpenCL standard [14]. In this paper we use NVIDIA's CUDA – a C-like language for general purpose graphics card programming [13]. CUDA provides new functionality that distinguishes it from the early GPU programming models (*e.g.* random access byte-addressable memory and support for coordination and communication among processes through thread synchronization and shared memory), thereby allowing more efficient processing of complex data dependencies. CUDA also supports single and double precision, and IEEE-compliant arithmetic [13]. In addition, higher-level libraries have been created to simplify CUDA code development, such as the Thrust library, a collection of parallel algorithms modeled on the C++ Standard Template Library [15]. These advances have extended the applicability of GPU computation to a much broader range of computational problems in science and engineering [8].

Nevertheless, while these new GPU programming tools dramatically improve on earlier generations, GPU implementations continue to lack some of the flexibility of CPU based programs. In part this lack of flexibility arises from differences in GPU and CPU architectures. Increased GPU performance is achieved by distributing computational tasks across several multiprocessors. Together these multiprocessors are able to achieve substantial processing throughput, however, the individual GPU processing threads lack the performance, independence and resources (*e.g.* registers) found in their CPU counterparts. These hardware differences restrict the complexity of the operations available to

[‡]In contrast with traditional "top-down" numerical methods where the material behavior is modeled directly, lattice-Boltzmann methods may be viewed as "bottom-up" numerical methods in which the behavior emerges from underlying smaller-scale processes – namely the interactions of discrete analogues to the single-body particle distribution functions described by the classical Boltzmann equation.

the more "lightweight" or simpler GPU threads. In addition, because the multiprocessors operate most efficiently when their constituent processing threads act in concert, conditional statements can severely adversely affect code performance. Finally, while CUDA and other programming platforms support object-oriented programming on the CPU host, object-oriented features such as inheritance and function pointers are not supported within GPU kernels (discussed in greater detail in Section 2). A direct consequence of these restrictions is that, under the GPU programming model, it may be difficult to develop modular and extensible programs with variable on-device functionality.

This paper describes a process of automatic code generation to circumvent these difficulties for lattice-Boltzmann methods. It details the development of GPU-based modules for LBHydra (www.lbhydra.umn.edu), an extensible lattice-Boltzmann simulation package capable of modeling a wide array of fluid mechanical behavior. Section 2 discusses lattice-Boltzmann simulations and the automatic code generation process used to add GPU capabilities to LBHydra. In Section 3, we compare the performance of the automatically created code with purpose written codes specifically developed for particular applications as described in previous publications [7, 16]. The flexibility of the method is also demonstrated by employing user generated models to simulate the dissolution of CO_2 droplets in a porous medium. Conclusions are summarized in Section 4.

2 A GPU module for lattice-Boltzmann methods

In this section, we describe how the LBHydra lattice-Boltzmann simulation package has been extended to include GPU based models. LBHydra offers numerous areas for user input and modification, including user-defined material models, lattice-types, and subroutines. It has been employed in simulations of multiple-phase and multiple-component fluids, heat and solute transport, dissolution, dispersion, and detailed pore-scale and macroscopic scale simulations [16–20]. The GPU-based module described in this paper allows the user to retain much of the flexibility of the main LBHydra program, while accelerating the simulations with CUDA-compliant NVIDIA graphics processing units.

Lattice-Boltzmann methods represent fluid mechanical behavior via a collection of fluid packets – discrete analogues to the particle density distribution functions in the classical Boltzmann equation [21]. The fluid packets move about the lattice with fixed velocities, with positions that are updated in discrete time steps. Each fluid packet has an associated density represented by a real number. Through a series of collision and streaming steps, summarized by the following equation, the density distribution is relaxed towards a local equilibrium determined from the macroscopic properties of the fluid at the node:

$$f_i(\mathbf{x}+\mathbf{c}_i\Delta t,t+\Delta t)=(1-\lambda)f_i(\mathbf{x},t)+\lambda f_i^{eq}(\mathbf{x},t)+F_i(x,t), \tag{2.1}$$

where f_i^{eq} is the equilibrium fluid packet density along lattice direction i, λ is the collision frequency, $\mathbf{c}_i$ is the lattice velocity (Table 2), and F_i represents additional forcing

or source/sink terms. The right of Eq. (2.1) is the collision step – the relaxation of the fluid packet densities toward the equilibrium densities, while the left side of the equation represents the streaming step – the propagation of the fluid packets to neighboring nodes in the lattice. The popular D3Q19 (three-dimensional, 19-velocity) lattice is discussed in this paper [22], although the same equation is also applicable to other lattice types. In addition, for the sake of simplicity this paper concentrates on single-relaxation lattice-Boltzmann models, in which the collision frequency λ is a scalar, although the approach presented is equally valid for GPU implementations of multiple relaxation lattice-Boltzmann models (*e.g.* [5,6]).

The expression for the equilibrium fluid packet densities, f_i^{eq}, depends on the constitutive behavior of the fluid being modeled and the type of lattice employed. For example, to simulate the Navier Stokes equations using the D3Q19 lattice, the equilibrium packet densities, f_i^{eq}, are a function of the net fluid-packet density, $\rho=\sum_i f_i$, and the net velocity of the fluid packets, $\mathbf{u}=\sum_i f_i\mathbf{c}_i/\rho$:

$$f_i^{eq}=\rho\omega_i\left[1+3\mathbf{u}\cdot\mathbf{c}_i\left(1+3\mathbf{u}\cdot\mathbf{c}_i/2\right)-3\mathbf{u}\cdot\mathbf{u}/2\right], \tag{2.2}$$

where ω_i are lattice weights (Table 2). Different boundary conditions and constitutive behaviors are invoked by modifying the collision step. Typically, several different collision rules are required in a simulation, with the simulation geometry dictating how rules are distributed across nodes. If the definition for f_i^{eq} in Eq. (2.2) is adopted, the terms on the right of Eq. (2.1) are local to individual nodes. However, in more complex lattice-Boltzmann methods (*e.g.* multiple-phase and multiple-component methods [24,25]), the equilibrium density distribution f_i^{eq}, the forcing terms F_i, and even the collision frequency λ, may be functions of additional state (or internal) variables – properties of the node and its neighbors.

Nevertheless, the relative simplicity of the fluid-packet interactions and the strong locality of the method, make lattice-Boltzmann models excellent candidates for parallel implementation on the SIMD environments found in GPUs, as they permit large numbers of lightweight processing threads to act simultaneously.

NVIDIA's CUDA programming environment provides a set of minimal extensions to the C programming language that enables programs to direct the large-numbers of GPU threads efficiently. The CUDA programming environment organizes individual GPU kernels into groups of threads or "thread blocks", themselves arranged into an array or "block grid" [13]. These thread blocks are additionally segmented on a multiprocessor at execution time into contiguous 32-thread groups known as "warps." Each kernel call specifies the number and arrangement of thread blocks (*i.e.* regular arrays of one or two dimensions) to be executed, as well as the number and arrangement of threads within each block (in one, two, or three dimensional arrays). While there are relatively few restrictions on the dimensions of the block grid, the dimensions of the individual thread blocks and the number of blocks that can be executed simultaneously are strongly influenced by underlying hardware constraints (Table 1). Instructions exist to synchronize execution of threads within each block, and inter-thread communication is supported

Table 1: Maximum thread block and block grid dimensions for the Tesla C1060 GPU.

Thread Block		
	Maximum number of Threads per Block	512
	Maximum X and Y Dimension	512
	Maximum Z Dimension	64
Block Grid		
	Maximum X and Y Dimension	65535
	Maximum Z Dimension	1

Table 2: Lattice velocities, $\mathbf{c}_i$, and respective weights, ω_i, for D3Q19 lattices. Curly braces, $\{\}$, indicate lattice velocities comprising the distinct permutations terms within the braces.

$\mathbf{c}_i\ (\Delta x/\Delta t)$	ω_i
(0,0,0)	$1/3$
$\{0,0,\pm1\}$	$1/18$
$\{0,\pm1,\pm1\}$	$1/36$

within blocks through shared memory (Fig. 1). Communication between separate blocks is more difficult as the order in which individual blocks are executed is not predetermined.

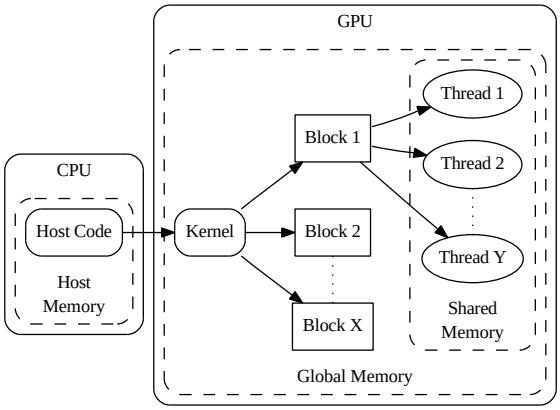

Figure 1: GPU kernel execution is based on groups of threads known as thread blocks. Inter-thread communication is made possible through shared memory, accessible to all threads within the same block. In addition, threads have access to the GPU's global memory, a larger data storage space which is available on the device, distinct from the CPU host's memory.

The different types of memory dictate the basic pattern of implementation of lattice-Boltzmann methods on the GPU. To maximize the relative amount of time spent on GPU computation and reduce data transfer, we adopt a decelerator model [23], in which calculation is confined to the GPU devices and the CPU host is reserved for subsidiary tasks such as data initialization, data transfer between devices, and data output. In addition,

because CPU and GPU tasks are separated under the decelerator model, multiple-GPU versions of the code (either with single or multiple CPUs) can be implemented with relative ease, and communication time can be reduced by using asynchronous operations on the CPU and GPU.

After the fluid packet data is transferred from the CPU to the GPU, the thread blocks iterate over the lattice with each thread acting on distinct nodes. The threads copy the fluid packet data associated with the node from the global memory of the GPU to the thread registers, where the collision step is performed, after which the updated fluid packet densities are written back to the global memory. The streaming step is either conducted explicitly in this process (by moving the fluid packet data to different locations in the GPU's global memory) or implicitly (by storing the fluid packets in fixed locations and altering the manner in which fluid packets are read from local to global memory) [7, 16].

A particular feature of the GPU architecture is the high cost of data transfer between global memory and thread registers – a result of the slow rate of transfer and the overhead required to access global memory. The overhead is reduced in part with so-called "coalesced" reads and writes from global memory, in which the threads act on contiguous blocks of memory. Nevertheless, the slower memory access means that such memory operations should be kept to a minimum to achieve good performance.

Rather than invoke separate GPU kernels for each collision rule, the fluid packets at each node are copied into the thread registers and different node types are distinguished by a conditional statement within a single kernel to reduce the number of global memory operations. The presence of the conditional statement results in so-called "divergent warps". Divergent warps occur when threads within a warp (a group of 32 threads executed simultaneously by the GPU) follow separate execution paths. This reduces performance as the conditional branches in these divergent warps are executed in series. However, this is not a significant concern for lattice-Boltzmann simulations: it is uncommon that more than two different collision rules are applied within a warp, and often only one rule will be encountered. More importantly, increasing the number of collision rules within a kernel increases register pressure, as the most complex collision rule dictates the number of registers required, and hence the number of warps that can be executed simultaneously. Thus for good performance it is important the number of collision rules within the kernel be kept to a minimum.

A drawback of this approach is that it does not readily lend itself to extensible applications that permit user-defined lattice-Boltzmann models. Although CUDA fully supports the C++ programming language for host (CPU) code, only a subset of C++ is supported on the GPU [13]. Specifically, CUDA provides GPU support for polymorphism, default parameters, operator overloading, namespaces, function templates and classes. However, function pointers and virtual class functions are not currently supported for the GPU. This prevents, for example, inheritance and upcasting within GPU kernels. These restrictions also apply to higher-level programming tools based on CUDA, such as the Thrust library.

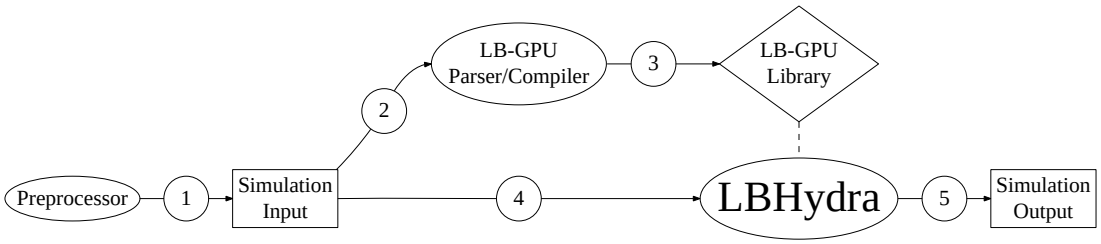

Figure 2: LBHydra GPU simulation pipeline. The preprocessor generates an input file (1) which is first read (2) by the Parser/Compiler to generate (3) a dynamic library linked to the main LBHydra program. The input file is then read (4) by LBHydra which uses the newly-created library to run the GPU simulation (5).

To overcome these limitations, the GPU module for LBHydra introduces a parser/compiler step into the simulation pipeline (Fig. 2). The parser is an object-oriented Python code containing objects representing the collision rules and state variables. Collision rule objects record the physical properties of the component (*e.g.* fluid viscosity) and the state variables the method requires, and contain routines to generate the CUDA code for the collision rule. The parser reads the simulation's input files and determines the minimal set of collision rules and state variables. This ensures that state variables are not duplicated and reduces register pressure by eliminating unnecessary conditionals. The parser then generates CUDA kernels to calculate both the state variables and collision rules, and control communication between CPU and GPU. From this information, the compiler creates a customized library, dynamically linked to the LBHydra application, that enables the simulation to run on the GPU. Pseudo-code outlining the code generated by the LBHydra GPU module is given in Appendix A.

The parser and compiler steps reintroduce some of the missing C++ functionality, because they are implemented on the CPU. This includes the ability to extend the parser and compiler with dynamic libraries to introduce user-defined collision rules and state variables. The parser also allows additional user-defined subroutines and CUDA kernels to control data initialization and functions to query and modify the simulation within each time step. Separating the GPU-specific functions from the lattice-Boltzmann simulation in this way enhances code reuse, and lets the user concentrate on the lattice-Boltzmann simulation, rather than the details of the underlying CUDA implementation. This latter point is particularly important given the rate at which GPU architectures have been developing and the impact that subsequent changes to coalesced memory operations, the relative amounts of GPU memory, and other hardware changes have on the overall simulation performance.

3 Results and discussion

The performance of LBHydra's automatically generated code is compared to that of hand-written single-purpose codes from our previous publications [7,16] in Table 4 for a

Table 3: GPU Specifications.

Device	Tesla C1060
Clock	1.296 GHz
Global Memory	4096 MiB
Mem. Clock	1600 MHz
Bus Width	512 bits
Processing Elements	240

Table 4: Comparison between peak performances obtained with LBHydra's automatically generated code and equivalent purpose written codes on a single GPU. Speeds are measured in Millions of Lattice-node Updates Per second (MLUPs). Lattice dimensions: $160\times160\times160$.

Model	GPU codes		Difference
	Single-purpose	Generated by LBHydra	
Single phase	444 MLUPs	436 MLUPs	-1.8%
Multiphase	218 MLUPs	219 MLUPs	+0.4%
Multicomponent	218 MLUPs	198 MLUPs	-10.1%

Table 5: Performances of single-phase LBHydra simulations for multiple GPUs on a single CPU. Problem size is scaled with the number of processors. Speeds are measured in Millions of Lattice-node Updates Per second (MLUPs).

GPUs	Lattice-Size	GPU codes		Difference
		Single-purpose	Generated by LBHydra	
1	160x160x160	444 MLUPs	436 MLUPs	-1.8%
2	160x160x320	494 MLUPs	556 MLUPs	+11.1%
3	160x160x480	714 MLUPs	827 MLUPs	+13.7%
4	160x160x640	938 MLUPs	1067 MLUPs	+12.1%

single GPU and in Table 5 for multiple GPUs. The programs are tested on NVIDIA Tesla C1060 GPUs of compute capacity 1.3 (complete GPU specification found in Table 3). The host code is compiled using the GNU g++ compiler 4.3.3 with the compiler flag "-O3" for compiler optimizations and the CUDA kernel code using NVIDIA compiler NVCC release 2.3, V0.2.1221 with the compiler flag "–use_fast_math."

The overhead due to the parser and compiler is minimal (less than a minute) compared to the time required to run a typical simulation (which may be several hours or more depending on the application). In addition, the parser and compiler step is not required every time – the same dynamic library can be used in multiple simulations that employ the same combination of lattice-Boltzmann methods. This latter feature is particularly useful, for example, when conducting parameter space analysis and optimization studies.

However, there is some loss of performance in the automatically generated code for certain applications. To demonstrate this, three different simulations are considered: a

single-phase pressure driven flow; a multiphase simulation of phase separation; and a multicomponent simulation of two immiscible fluids. The multiphase and multicomponent lattice-Boltzmann simulations are based on well-known models by Shan and Chen [24], and He and Doolen [25]. These models introduce an interaction potential to each node that is a function of either the fluid density (in single-component multiphase simulations) or the component concentration (in multicomponent simulations). The automatically generated code's speed is roughly equivalent to our previous implementations for both the single-phase and multiple-phase lattice-Boltzmann methods. However, the multiple-component simulation is slower than the equivalent purpose written code. In the later case, the automatically generated code makes less efficient use of shared memory and global memory operations, as we have chosen to isolate state variable calculations per-component. This choice was adopted to reduce register pressure in simulations with many state variables per node, rather than a few state variables calculated across multiple components. Nevertheless, the loss of performance is not very large – approximately 10% in this example. Due to changes in the GPU-CPU communication subroutine, the automatically generated code outperforms our previous purpose written code [16] when implemented on multiple GPUs (Table 5). However, the increase in performance is similar in both cases. Due to the additional communication overhead at each timestep, the codes' performances show only marginal improvement when increasing from one to two GPUs, but then scale approximately linearly as the number of GPUs is increased.

It should also be noted that, the problem of optimizing GPU codes is highly nonlinear, and subtle changes in the calculation may adversely impact the code's performance [26]. No effort is currently made to optimize the code's performance for specific models or GPU architectures in the parser/compiler step. However, it may be possible in the future to combine the present approach with an optimization algorithm (*e.g.* [16, 26–28]).

In Fig. 3, the GPU module for LBHydra is used to simulate the dissolution of a rising droplet of carbon dioxide into ambient water within a porous medium. The simulation combines models representing the two immiscible fluids with a third component representing the dissolved concentration of carbon-dioxide. A detailed description of the lattice-Boltzmann methods used in this simulation is provided in Walsh and Saar [19]. Briefly stated, in this model, particular care is taken to ensure that the correct boundary conditions are applied at the moving surface of the droplet. To achieve this, a more complex two component model [29] is employed in place of the multiphase models described earlier to reduce "parasitic" or "spurious" interface velocities [30, 31]. The first-order boundary condition at the droplet surface is imposed at lattice-edges rather than at the nodes themselves [19]. This boundary condition is enforced in a separate calculation step, after the collision step has been applied to each node. To demonstrate the flexibility of the method, the simulation is implemented with user-defined lattice-Boltzmann methods, rather than the native methods supplied with the code, and employs user-generated functions to control the boundary condition.

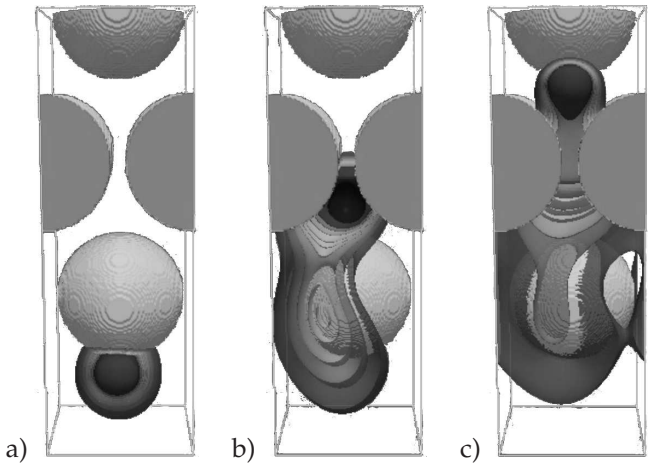

Figure 3: LBHydra's GPU module models a buoyant droplet of CO_2 dissolving in ambient water within a regular array of spheres. Contour surfaces indicate concentrations of dissolved CO_2 between 5% and 95% of the saturation concentration. The innermost contour denotes the surface of the pure CO_2 droplet. Lattice dimensions: $100\times100\times300$.

4 Conclusion

Lattice-Boltzmann methods are versatile fluid mechanical modeling techniques, well suited to parallel implementation on Graphics Processing Units (GPUs). While recent advances in GPU programming environments and architectures have improved the process of developing GPU programs, these tools currently lack the full functionality of their CPU equivalents. In particular, the absence of function pointers and true class inheritance on the GPU, can make writing flexible and extensible codes difficult while simultaneously maintaining high performance.

This paper discussed a means for lattice-Boltzmann simulations to avoid these limitations by automatically generating the code required for specific simulations. This process overcomes the lack of function pointers and inheritance on the GPU by generating a minimal library of collision rules based on the simulation's input files. In certain cases, the current approach does not make as efficient use of shared memory as codes that have been purpose written for specific applications. Nevertheless, the resultant reduction in performance is not extreme (approximately 10% or less for the codes considered here), and may be further mitigated in the future with the introduction of automatic code optimization routines. The automatically generated code is particularly flexible, allowing the addition of user defined models and subroutines. This feature was demonstrated by using the method to implement a complex simulation of dissolution from a rising droplet in a porous medium. By separating the lattice-Boltzmann methods from the CUDA implementation, the approach presented here allows the user to concentrate on the details of the lattice-Boltzmann simulation rather than the vagaries of the GPU programming environment.

Acknowledgments

MOS thanks the George and Orpha Gibson endowment for its generous support of the Hydrogeology and Geofluids research group. In addition, this material is based upon support by the National Science Foundation (NSF) under Grant No. DMS-0724560, Grant No. EAR-0838541, and Grant No. EAR-0941666, and the Department of Energy (DOE) under Grant No. DE-EE0002764. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF or the DOE. We would also like to thank both anonymous reviewers for their helpful comments.

This manuscript was approved for release by LLNL with release number LLNL-CONF-521460. This document was prepared as an account of work sponsored by an agency of the United States government. Neither the United States government nor Lawrence Livermore National Security, LLC, nor any of their employees makes any warranty, expressed or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States government or Lawrence Livermore National Security, LLC. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States government or Lawrence Livermore National Security, LLC, and shall not be used for advertising or product endorsement purposes.

A Pseudo-code for the LBHydra GPU module

The following pseudo-code outlines the principle steps in an automatically generated LBHydra program. We include directives generated for the OpenMP version of the code, which is designed for systems with multiple GPUs on a single CPU. The MPI version for clusters of multiple CPUs and GPUs contains additional subroutines to copy data between CPUs in Steps (3.3.1.ii) and (3.3.1.v).

1. Allocate memory on CPU for GPU-GPU transfer.
2. Create one thread per GPU: `#pragma omp parallel num_threads(numGPUs)`
3. Within the `#pragma omp parallel num_threads(numGPUs)` directive:
   - 3.1. Assign thread to GPU: `cudaSetDevice(th_id);`
   - 3.2. Copy initial conditions and problem geometry from CPU to GPU
   - 3.3. `for(ts = 0; ts < maxTimesteps; ts += outputFrequency)`:
     - 3.3.1. `for(t = 0; t < outputFrequency; t++)`:
       State variables:

i. Calculate state variables for outer nodes
ii. Copy outer boundary state variables from GPU to CPU
Copy neighboring state variables from CPU to GPU
iii. Calculate state variable values (simultaneous with ii) on the GPU.

Collision and streaming:

iv. Perform f_i streaming/collision steps for outer nodes
v. Copy outgoing outer boundary f_i from GPU to CPU
Copy incoming outer boundary f_i from CPU to GPU
vi. Perform inner f_i streaming/collision (simultaneous with v) on the GPU.

3.3.2. Rearrange f_i so location in memory is the spatial position.

3.3.3. Copy f_i data from GPU to CPU and save result to file.

3.4. Free GPU memory

4. Free CPU memory. End.

References

[1] S. Chen, G. Doolen, Lattice Boltzmann method for fluid flows, Annu. Rev. Fluid Mech 30 (1) (1998) 329–364.
[2] S. Succi, The Lattice Boltzmann Equation for Fluid Dynamics and Beyond, Oxford Univ. Press, Oxford, 2001.
[3] C. K. Aidun, J. R. Clausen, Lattice-Boltzmann method for complex flows, Annual Review of Fluid Mechanics 42 (1) (2010) 439–472.
[4] M. Sukop, D. Thorne, Lattice Boltzmann Modeling: An introduction for geoscientists and engineers, Springer, Heidelberg, Berlin, New York, 2006.
[5] J. Tölke, M. Krafczyk, Teraflop computing on a desktop PC with GPUs for 3D CFD, International Journal of Computational Fluid Dynamics 22 (2008) 443–456.
[6] J. Tölke, Implementation of a Lattice Boltzmann kernel using the Compute Unified Device Architecture developed by nVIDIA, Computing and Visualisation in Science (2008) 11 pages.
[7] P. Bailey, J. Myre, S. D. C. Walsh, M. O. Saar, D. J. Lilja, Accelerating lattice Boltzmann fluid flow simulations using graphics processors, International Conference on Parallel Processing: Vienna, Austria (ICPP 2009).
[8] S. D. C. Walsh, M. O. Saar, P. Bailey, D. J. Lilja, Accelerating Geoscience and Engineering System Simulations on Graphics Hardware, Computers and Geosciences 35 (12) (2009) 2353–2364.
[9] W. Li, X. Wei, A. Kaufman, Implementing lattice Boltzmann computation on graphics hardware. The Visual Computer 2003; **19**:444–456.
[10] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Kr uger, A. E. Lefohn, T. J. Purcell. A survey of general-purpose computation on graphics hardware. Computer Graphics Forum 2007; **26**(1):p80 – 113.
[11] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, P. Hanrahan, Brook for GPUs: stream computing on graphics hardware. SIGGRAPH '04: ACM SIGGRAPH 2004 Papers, ACM: New York, NY, USA, 2004; 777–786.
[12] AMD. ATI CTM Guide: Technical Reference Manual. AMD, 1.01 edn. 2006.
[13] NVIDIA CUDA C Programming Guide 3.3.1, nVIDIA, (2010).

[14] Khronos OpenCL Working Group. The OpenCL specification version 1.0. Technical Report 2009.
[15] N. Bell, J. Hoberok, Thrust: A Productivity-Oriented Library for CUDA, in GPU Computing Gems, Jade Edition, Edited by Wen-mei W. Hwu, 2011: pp. 359–371.
[16] J. Myre, S. D. C. Walsh, D. Lilja, M. O. Saar, Performance analysis of single-phase, multiphase, and multicomponent lattice-Boltzmann fluid flow simulations on GPU clusters, Concurrency Computat.: Pract. Exper., 23 (4) (2011) 332–350.
[17] S. D. C. Walsh, M. O. Saar, Macroscale lattice-Boltzmann methods for low-Peclet-number solute and heat transport in heterogeneous porous media., Water Resour. Res. 46 (2010) W07517.
[18] M. A. Davis, S. D. C. Walsh, M. O. Saar, Statistically reconstructing continuous isotropic and anisotropic two-phase media while preserving macroscopic material properties, Phys. Rev. E 83 (2011) 026706.
[19] S. D. C. Walsh, M. O. Saar, Interpolated lattice boltzmann boundary conditions for surface reaction kinetics, Phys. Rev. E 82 (6) (2010) 066703.
[20] S. D. C. Walsh, H. Burwinkle, M. O. Saar, A new partial-bounceback lattice-Boltzmann method for fluid flow through heterogeneous media, Computers and Geoscience 35 (6) (2009) 1186–1193.
[21] L. Boltzmann, Weitere Studien über das Wärmegleichgewicht unter Gasmolekülen [Further studies on the heat equilibrium of gas molecules], Wiener Berichte 66 (1872) 275370.
[22] Y. H. Qian, D. D'Humières, P. Lallemand, Lattice BGK models for Navier-Stokes equation, Europhys. Lett. 17 (6) (1992) 479–484.
[23] K. J. Barker, K. Davis, A. Hoisie, D. J. Kerbyson, M. Lang, S. Pakin, J. C. Sancho, Entering the petaflop era: the architecture and performance of roadrunner. Proceedings of the 2008 ACM/IEEE conference on Supercomputing, IEEE Press: Piscataway, NJ, USA, 2008; 1–11.
[24] X. Shan, H. Chen, Lattice Boltzmann model for simulating flows with multiple phases and components, Phys. Rev. E 47 (3) (1993) 1815–1819.
[25] X. He, G. D. Doolen, Thermodynamic foundations of kinetic theory and lattice Boltzmann models for multiphase flows, Journal of Statistical Physics 107 (1) (2002) 309–328.
[26] S. Ryoo, C. I. Rodrigues, S. S. Stone, S.-Z. U. Sara S. Baghsorkhi, J. A. Stratton, W. mei W. Hwu, Program optimization space pruning for a multithreaded GPU, in: Proceedings of the 2008 ACM/IEEE conference on Supercomputing, IEEE Press, Piscataway, NJ, USA, 2008, pp. 1–11.
[27] B. Jang, S. Do, H. Pien, D. Kaeli, Architecture-aware optimization targeting multithreaded stream computing, in: GPGPU-2: Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units, ACM, New York, NY, USA, 2009, pp. 62–70.
[28] E. Z. Zhang, Y. Jiang, Z. Guo, X. Shen, Streamlining gpu applications on the fly: thread divergence elimination through runtime thread-data remapping, in: Proceedings of the 24th ACM International Conference on Supercomputing, ICS '10, ACM, New York, NY, USA, 2010, pp. 115–126.
[29] L. Wu, M. Tsutahara, L. Kim, M. Ha, Three-dimensional lattice boltzmann simulations of droplet formation in a cross-junction microchannel, Int. J. Multiphas. Flow 34 (9) (2008) 852 – 864.
[30] R. Scardovelli, S. Zaleski, Direct numerical simulation of free-surface and interfacial flow, Annu. Rev. Fluid Mech 31 (1) (1999) 567–603.
[31] A. J. Wagner, The Origin of Spurious Velocities in Lattice Boltzmann, Int. J. Modern Phys. B 17 (2003) 193–196.