J. Myre1,∗, †, S. D. C. Walsh2, D. Lilja3 and M. O. Saar2

1Department of Computer Science, University of Minnesota, Twin Cities, Minneapolis, MN, U.S.A.
2Department of Geology and Geophysics, University of Minnesota, Twin Cities, Minneapolis, MN, U.S.A.
3Department of Electrical and Computer Engineering, University of Minnesota, Twin Cities, Minneapolis, MN, U.S.A.

SUMMARY

The lattice-Boltzmann method is well suited for implementation in single-instruction multiple-data (SIMD) environments provided by general purpose graphics processing units (GPGPUs). This paper discusses the integration of these GPGPU programs with OpenMP to create lattice-Boltzmann applications for multi-GPU clusters. In addition to the standard single-phase single-component lattice-Boltzmann method, the performances of more complex multiphase, multicomponent models are also examined. The contributions of various GPU lattice-Boltzmann parameters to the performance are examined and quantified with a statistical model of the performance using Analysis of Variance (ANOVA). By examining single- and multi-GPU lattice-Boltzmann simulations with ANOVA, we show that all the lattice-Boltzmann simulations primarily depend on effects corresponding to simulation geometry and decomposition, and not on the architectural aspects of GPU. Additionally, using ANOVA we confirm that the metrics of Efficiency and Utilization are not suitable for memory-bandwidth-dependent codes. Copyright © 2010 John Wiley & Sons, Ltd.

Received 26 February 2010; Revised 25 June 2010; Accepted 26 June 2010

KEY WORDS: graphics processing unit; CUDA; lattice-Boltzmann method; OpenMP; fluid dynamics; multiphase; multicomponent; performance analysis

1. INTRODUCTION

The lattice-Boltzmann method is a popular technique for modeling fluid mechanics of complex fluids [1]: notably, in the simulation of fluid systems comprising multiple phases (e.g. liquid and gas) and different fluid components (e.g. oil and water) [2, 3], as well as the simulation
of fluids and fluid mixtures through porous media with complex domain geometries [4–9]. The method has a strong spatial locality of data access, and consequently is an excellent candidate for parallelization. The lattice-Boltzmann method belongs to the structured grid computing class [10] and is hence particularly suited for implementation in single-instruction multiple data (SIMD) environments, as provided by general purpose Graphics Processing Units (GPGPUs) [11, 12]. This paper discusses the performance of such GPU programs when used in combination with more traditional parallel-programming tools, i.e. using OpenMP to simulate lattice-Boltzmann methods on multi-GPU clusters. Specifically, we focus on shared memory systems using OpenMP, as such systems are widely available as commercial off-the-shelf products and are most immediately available to research scientists and engineers.

Early lattice-Boltzmann fluid flow applications using graphics cards employed a graphics-pipeline or ‘shader’ approach. Possibly the first such model was created by Li et al. [13], who achieved an impressive 9.87 million lattice-node updates per second (MLUPS), approximately a 50× speedup over single core implementations at that time. Nevertheless, significant drawbacks in the shader programming approach, such as reduced precision (e.g. Li et al. [13] employed fixed 8-bit precision) remained. Furthermore, 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 the widespread adoption of GPU-based programs [14].

However, these problems have largely dissipated as general purpose graphics programming has matured. The barriers to adoption were reduced with the development of shader programming APIs, such as Sh [15, 16], HLSL [17], and Cg [18]. These tools allowed more sophisticated manipulation of the graphics pipeline through access to programmable vertex and fragment processors as well as reducing the translation requirements by providing libraries for certain operations [19]. GPGPU programming was further improved with the next generation of general purpose graphics programming environments, such as BrookGPU [19], the ATI CTM platform [20], the Compute Unified Device Architecture (CUDA) programming model released by NVIDIA [21], and the recently released OpenCL standard [22]. These environments have all but erased the underlying graphics origins of the programming model. For example, CUDA, the GPGPU programming environment discussed throughout this paper, provides extensions to the C programming language that allow the programmer to write kernels—functions executed in parallel on the GPU. CUDA also provides new functionality that distinguishes it from shaders (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. Finally, CUDA also supports single and double precision, and IEEE-compliant arithmetic [21]. These advances have extended the applicability of GPU computations to a much broader range of computational problems in science and engineering [23].

The release of this next generation of GPGPU programming environments has enabled significant advances in GPU lattice-Boltzmann implementations. Tölke [12] reported speeds of up to 568 MLUPS for the two-dimensional nine-vector (D2Q9) lattice-Boltzmann model and 592 MLUPS for the three-dimensional 13-vector (D3Q13) lattice-Boltzmann model using the first generation 8800 Ultra [24]. Bailey et al. [11] achieved speeds of up to 300 MLUPS for the three-dimensional 19-vector (D3Q19) lattice-Boltzmann model using the 8800 GTX. More recently the release of NVIDIA’s Tesla class of GPUs, specifically designed for scientific and engineering computation, has enabled even greater performance gains, a development that should continue with the new Fermi class of GPUs.
However, some limitations remain. For example, the maximum simulated lattice size in these single-GPU implementations is limited by the available GPU memory. In addition, it is unclear to what extent the performance gains seen on single GPUs can be sustained on larger lattices distributed across multiple CPUs hosting several GPUs. In this paper, we report on the performances of multiple GPU clusters that employ OpenMP for inter-GPU communication. In addition to the standard single-phase lattice-Boltzmann models, we also discuss how these multi-GPU systems perform when more complex multiphase and multicomponent lattice-Boltzmann methods are implemented.

Furthermore, although the latest GPGPU programming environments improve programmability by masking much of the complexity of modern GPUs, achieving high-performance code is still difficult. Many of the factors affecting the performance result in nonlinear responses, making program optimization difficult. Because of this nonlinearity, it is a difficult task to relate analytically the number of MLUPS to the important factors of the hardware problem. Moreover, it seems likely that even if such a model was found, it would become obsolete with the introduction of new generations of GPU hardwares and compilers.

Instead, in this paper we use the Analysis of Variance (ANOVA) [25] statistical design of experiments to uncover the factors and interdependencies that are key to the overall system performance. While not as concrete as an analytical performance model, this approach provides the developer with a statistical measure to better direct their optimization efforts. Additionally, we compare the results of our model with the results obtained using the program optimization carving method from Ryoo et al. [26].

The following section describes the lattice-Boltzmann computation method and the single- and multi-GPU implementations. The experimental methodology is then presented, followed by an overview and analysis of the results of the simulations. Conclusions and future work are discussed in the final section.

2. LATTICE-BOLTZMANN COMPUTATION

Lattice-Boltzmann methods represent fluid flow via discrete particle distribution functions that are analogous to single-body particle distribution functions described by the classic Boltzmann equation [27]. The particle distribution functions are represented by fluid packets, $f_i$, that move about a regular node lattice in discrete time steps. Interactions between fluid packets are restricted to individual nodes and their lattice neighbors.

The simulations described in this paper are based on the widely employed D3Q19 lattice-Boltzmann model, comprising a three-dimensional lattice (D3) with 19 distinct fluid packet directions per node (Q19) [28]. The fluid packets move about the lattice in a two-stage process, a streaming step in which $f_i$ are copied from one node to the next and a collision step in which $f_i$, arriving at each node, are redistributed according to a local collision rule. For the model employed in this paper, these steps are summarized by

$$f_i(x + c_i \Delta t, t + \Delta t) = (1 - \lambda) f_i(x, t) + \lambda f_i^{eq}(x, t),$$

where $c_i$ is the lattice velocity associated with $f_i$, $\lambda$ is the collision frequency, and $f_i^{eq}$ is the local pseudo-equilibrium particle density distribution function (Equation (2)). This collision step is based on the widely used single-relaxation BGK approximation [28, 29], although it should be noted that multiple-relaxation methods have also been implemented on graphics cards with considerable
success [12, 24]. For the single-component single-phase (SCSP) model, using the D3Q19 lattice, \( f_{eq}^i \) is given by

\[
 f_{eq}^i = \rho w_i [1 + 3 \mathbf{u} \cdot \mathbf{c}_i (1 + 3 \mathbf{u} \cdot \mathbf{c}_i / 2) - 3 \mathbf{u} \cdot \mathbf{u} / 2],
\]

where \( w_i \) are weighting constants specific to the lattice velocities, \( \rho = \sum_i f_i \) is the macroscopic fluid density, and \( \mathbf{u} = (1 / \rho) \sum_i f_i \mathbf{c}_i \) is the fluid velocity.

The multiphase and multicomponent lattice-Boltzmann simulations presented in this paper are based on models by Shan and Chen [5], and He and Doolen [30]. These models introduce an interaction potential to each node that is a function of either the fluid density (in single-component multiphase (SCMP) simulations) or the component concentration (in multiple component simulations). The SCMP models in this paper use the interaction potential given in [5, 30],

\[
 \psi = \psi_0 \exp(-\rho / \rho_0),
\]

where \( \psi_0 \) and \( \rho_0 \) are predefined constants and \( \rho \) is the fluid density. This interaction potential gives rise to a cohesive force, \( \mathbf{F} \),

\[
 \mathbf{F} \Delta t = -G \psi(\mathbf{x}, t) \sum_i w_i \psi(\mathbf{x} + \mathbf{c}_i \Delta t) \mathbf{c}_i,
\]

where \( G \) is an interaction force constant. In multicomponent simulations, we adopt the model of He and Doolen [30], where

\[
 \psi = \rho
\]

and

\[
 \mathbf{F}_{ab} \Delta t = -G \psi_a(\mathbf{x}, t) \sum_i w_i \psi_b(\mathbf{x} + \mathbf{c}_i \Delta t) \mathbf{c}_i
\]

gives the cohesive force, \( \mathbf{F}_{ab} \), acting on component ‘a’ from component ‘b’. In both cases, the forcing term is implemented by adjusting the velocity used to calculate the pseudo-equilibrium density, e.g. in the SCMP model:

\[
 \mathbf{u} = \frac{1}{\rho} \sum_i f_i \mathbf{c}_i + \frac{\mathbf{F} \Delta t}{\lambda \rho}.
\]

Although the interaction potentials given in Equations (3) and (5) are used throughout the remainder of this paper, other interaction potentials may be employed with little or no change in the performance.

2.1. Single-GPU implementation

Under NVIDIA’s CUDA programming environment, individual GPU kernels execute threads in groups known as ‘thread blocks’, which arrange themselves into an array or ‘block grid’. NVIDIA’s CUDA Programming Guide [31] states that thread blocks are additionally segmented on a multiprocessor at execution time. When a GPU multiprocessor is given a thread block to execute, the multiprocessor splits the thread block 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.
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.

One thread is assigned per node along the x-axis, and one block is assigned per (y, z) coordinate.

In this paper, one-dimensional thread blocks are employed in all the lattice-Boltzmann simulations with one thread assigned per node along the x axis and one block per y, z coordinate (Figure 2). This thread-block layout is adopted to satisfy CUDA’s requirements for coalesced memory access. The entirety of the lattice is stored in a configuration that can be visualized as each y, z coordinate block being placed in order and end-to-end to create a one-dimensional
row of sorted thread blocks. As this configuration is stored in the global memory of GPU, it is worth noting that operations using the GPU global device memory are substantially slower than those involving shared memory [21]. However, the high cost of memory access is mitigated using coalesced memory operations, in which threads in a given block read and write to contiguous memory addresses. Owing to the configuration of the lattice in the global memory of GPU, other problem decompositions would lose the ability to use coalesced memory operations resulting in a drop in the performance. Additionally, this implementation does not use any blocking along the x dimension as it would incur communication costs beyond those that are already necessary.

As a result of the requirements for memory coalescence and the thread block geometry, propagation of fluid packets is handled differently depending on whether the fluid packets travel parallel or perpendicular to the x axis. Propagation of $f_i$ parallel to the x-axis is achieved by manipulating the data in shared or local memory. How this is performed depends in turn on the ‘compute capacity’ of GPU—a term introduced by NVIDIA to describe the major and minor revision numbers of the GPU architecture. NVIDIA GPUs of compute capacity 1.1 and lower have stricter requirements for coalesced global memory access than the later GPU versions [21]. These requirements restrict individual threads from reading and writing to specific global memory locations, the implication of which is that the propagation of fluid packets along the length of the block is best achieved by translating the fluid packets into the block’s shared memory, before copying the packets back to global memory (Figure 3(a)). Later-generation devices allow an offset in the location in which data could be written to global memory while preserving coalescence, so that fluid packets can be written directly from local memory without the need for explicit transfer of data in shared memory (Figure 3(b)). While this does not substantially improve the performance for a given block size, the approach simplifies the code and reduces shared memory use substantially thereby enabling implementations with larger block sizes.

Propagation of fluid packets perpendicular to the thread block axis is accomplished by manipulating the location of block reads and writes to global memory. Here again multiple choices for memory access patterns exist. Perhaps the simplest approach is a ‘Ping-Pong’ buffer pattern in which fluid packets are read from one matrix, the collision step is performed, and the results are copied to a second matrix. The pattern is then reversed in the following time step. This approach has the advantage that no additional synchronization steps are required, however, it has the disadvantage that memory must be made available for the two fluid packet matrices. We also discuss two alternate memory access patterns that require approximately half the memory of the ‘Ping-Pong’ buffer. We refer to them as the ‘Lagrangian’ pattern and the ‘Flip-Flop’ pattern. Under the ‘Lagrangian’ pattern rather than associate each global memory location with a fixed point in

---

**Figure 3.** Data propagation along the x-axis: (a) for compute capacity 1.1 and lower the data must be propagated in shared memory to preserve coalescence and (b) for compute capacity 1.2 and above the copy back to global memory is made without the transfer in shared memory.

Copyright © 2010 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe
LATTICE-BOLTZMANN SIMULATION PERFORMANCE ON GPU CLUSTERS

Figure 4. Three memory access models for data propagation along y and z axes: (a) ping-pong; (b) flip-flop; and (c) lagrangian. Fluid packet position in the diagram designates the same global memory location (although not stored in this geometry), arrows indicate fluid packet directions, dark arrows show fluid packets associated with the center node. In (a) and (b) the steps shown in this figure are repeated, while in (c) the ring of fluid packets arriving at the center node expands outwards on subsequent time steps (see Appendix A for more details).

space, we instead assign fixed global memory locations to individual fluid packets. The threads, in contrast, remain associated with a fixed set of points in space. Thus, the fluid packets are stored in global memory in the same location throughout the simulation, and selectively loaded into each thread’s local memory according to which fluid packets would arrive at the thread’s location in the current time step. The collision step is then performed and the fluid packets are copied back to their original locations. Under the ‘Flip-Flop’ memory access pattern two time steps are required to complete a memory access cycle. In the first time step fluid packets are read from, and written back, to the same buffer in global memory, but with the directions reversed. During the subsequent time step, the fluid packets are read from the reversed configuration, and then written back to the same buffer with the original direction orientation restored. In order to propagate the fluid packets correctly, in this second time step, the reads and writes occur on adjacent nodes. Illustrations of all three memory access patterns are presented in Figure 4, and more detailed descriptions are given in Appendix A.

Investigation of all three memory access patterns reveals only slight differences in the performance. The performance of the Ping-Pong memory access pattern is not considered sufficient to warrant the approximately twofold increase in global memory use. Lagrangian and Flip-Flop memory access patterns show similar performance characteristics, with the Lagrangian pattern outperforming the Ping-Pong and Flip-Flop patterns in larger lattices. A brief comparison between the three methods is given in Section 4, however, the Lagrangian access pattern is adopted in the remaining simulations considered in this paper.

The multiphase and multicomponent lattice-Boltzmann models both introduce non-local interaction forces and, as a result, the collision step is no longer solely a function of information at each node. For these models, two changes are made to the basic lattice-Boltzmann simulation: first, interaction potentials are calculated at each node and stored in global memory prior to the collision and streaming steps. Second, at the start of the collision step, the interaction potentials

---

of the thread block and the eight neighboring rows are read into shared memory and used to calculate the cohesive force. Out-of-plane interaction potentials are shifted along the rows in the thread-block’s shared memory, thereby reducing the number of global memory operations used in this step from 19 to 9 per node.

2.2. Multi-GPU implementation

The CUDA programming environment allows for asynchronous operations in which calculations take place simultaneously on both the GPU device and the CPU host. Use of asynchronous operations is key to the improved performance on multiple devices to mask the cost of data transfer between the host and device. To maximize the relative amount of time spent on GPU computation and reduce data transfer, we adopt a decelerator model [32], 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.

Our multi-GPU implementation divides the simulation into cuboid regions of equal size, which are assigned to the available devices. During each time step, calculations are first carried out on the outer $y$ and $z$ faces of each region. While the contribution from the interior nodes are calculated, the data from the outer faces are simultaneously transferred between GPUs—a three-step process of copying data from the device to the CPU host, then from host to host (for multiple CPUs), then back from the host to the new device (Figure 5). OpenMP threads control communication between devices on the same CPU host, whereas dedicated data pipelines, via persistent MPI communication operations, are employed for data transfer between hosts. However, this paper is focused on shared memory GPU systems and thus the host-to-host MPI communication is not discussed further. Under the standard lattice-Boltzmann model, only one round of inter-GPU communication is required for the single-phase, single-component lattice-Boltzmann model to transmit the values of the fluid packets on boundary nodes. For the multiphase and multicomponent models an additional round of data transfer is needed to transmit the interaction potentials. Thus in each region, (1) the outer shell of interaction potentials is calculated and (2) the outer shell is transferred between GPUs.

Figure 5. Flowchart of the multi-GPU implementation. The steps shown are repeated for the interaction potentials in the multicomponent and multiphase versions and given in greater detail in Appendix B.
while simultaneously the remaining inner core of interaction potentials is calculated. After the interaction potentials are transmitted, the same pattern is repeated for the fluid packet collision and streaming steps (Appendix B).

3. EXPERIMENTAL METHODOLOGY

Parameters affecting the GPU performance include the number of threads per thread block, overall lattice size, and memory usage at multiple tiers within the GPU memory hierarchy (e.g. registers, shared, and global). As varying these factors induces a nonlinear performance response [33], analysis is necessary to adequately determine the parameter values that will reliably predict the performance of lattice-Boltzmann codes. Furthermore, it is useful to quantify the contributions of each parameter to improve future optimization efficiencies.

Two classes of performance parameters are examined in this paper. The first set (Table I) consists of fundamental factors associated with the GPU architecture and lattice-Boltzmann model. These parameters include the number of threads per thread block, the number of nodes in the lattice, the total number of GPUs used in computation (hereafter denoted as NP), and the number of Parallel Thread eXecution (PTX) instructions in the GPU kernel. PTX is an intermediate code representation that NVIDIA uses as a virtual GPU instruction set architecture [34]. The assembly code executed by GPUs is created by translating and optimizing the PTX representation into a supported GPU architecture. The number of PTX instructions gives an estimate of the number of actual instructions executed on the GPU. Other factors associated with the GPU architecture have been withheld from this class of performance parameters for the analysis given here. Shared memory and register usage, although fundamental, are withheld as they remain constant across lattice configurations. Occupancy, while important, is not considered fundamental here as it is a derived metric. However, due to the importance of occupancy, it is included in the discussion of the results in Section 4.

The second set of performance parameters comprises two analytically derived metrics given in Ryoo et al. [26]: Efficiency and Utilization (defined below in Equations (8) and (9)). Efficiency is a measure of the instruction efficiency of the kernel to be run on the GPU. Efficiency is obtained by counting the number of instructions executed to complete a task of a given size. All others being equal, a more efficient program should execute faster than a less efficient one. However, Efficiency is based on the gross number of instructions executed across all threads. As such, it does not account for secondary effects on the performance such as wait time as a result of blocking instructions and the availability of work for independent thread groups. These quantities are instead accounted for via the Utilization metric, which gives an approximate measure of how much a kernel makes use of the GPU’s available compute resources.
Efficiency is a function of the number of instructions that will be executed per thread ($I_T$) as well as the number of threads that will be executed for a given kernel ($T_K$). Utilization depends on the number of instructions executed per thread ($I_T$), the number of regions of the kernel delimited by synchronization or high latency instructions ($R_K$), the number of warps in a thread block ($W_{TB}$), and the maximum number of thread blocks assigned to each streaming multiprocessor ($B_{SM}$):

$$\text{Efficiency} = \frac{1}{I_T \times T_K}, \quad (8)$$

$$\text{Utilization} = \frac{I_T}{R_K} \left[ \frac{W_{TB} - 1}{2} + (B_{SM} - 1)(W_{TB}) \right]. \quad (9)$$

We normalize Efficiency by the problem size (overall lattice size) for our analysis by multiplying Ryoo et al. [26] Efficiency by the number of nodes in a given lattice. As the number of nodes in a lattice is equivalent to $T_K$, our normalized Node Efficiency is found to be $I_T^{-1}$. This Node Efficiency will be referred to as Efficiency in the remainder of this paper.

Both metrics are application specific, therefore there is no single Efficiency and Utilization value to indicate good performance, rather implementations with both higher Efficiency and Utilization perform better than equivalent programs with lower Efficiency and Utilization. Hence, the two metrics can be used to define Pareto optimal curves, which were employed by Ryoo et al. [26] in an approach known as ‘Tradeoff carving’ to rapidly find optimal application implementations. However, these metrics fail to account for memory bandwidth bottlenecks on the performance. Although Ryoo et al. [26] explicitly state that the use of Efficiency and Utilization are not applicable to bandwidth-bound programs (e.g. lattice-Boltzmann fluid flow simulations), we nonetheless include them in our analysis with the purpose of quantifying their applicability in comparison to the more universally used first set of performance parameters.

3.1. Procedure

The set of codes tested are composed of the complete combination of the number of nodes in each dimension of the lattice, the number of GPUs involved, and the type of simulation being performed. It should be noted that the number of nodes in the $x$-dimension of the lattice is equivalent to the number of threads per block and that the product of the number of nodes in each dimension is the lattice size. Furthermore, the simulation type defines the type of fluid flow simulation, as well as some measure of the relative difficulty of computation. Ordered from the simplest to the most complex, simulation types are SCSP, SCMP, and multicomponent multiphase (MCMP). Of these, the SCSP simulation type is the only major type with implementations of all the memory access patterns: Lagrangian, Ping-Pong, and Flip-Flop (described in Section 2.1). We implement the SCMP and MCMP codes using only the Lagrangian memory access pattern for two reasons. First, the Lagrangian method outperforms the Ping-Pong method by 11% and the Flip-Flop method by 18% on average. Second, the Lagrangian method requires only half the global memory of the Ping-Pong access pattern. This is particularly important for the SCMP and MCMP simulations, which have increased memory requirements as a result of the interaction potentials and additional fluid components.

Each of these simulation types are complemented by a corresponding multi-GPU implementation using the Lagrangian memory access pattern and OpenMP for communication. Single-GPU OpenMP implementations are also considered to determine the overhead of the communication method.
Table II. GPU specifications.

<table>
<thead>
<tr>
<th>Device</th>
<th>Tesla C1060</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clock</td>
<td>1.296 GHz</td>
</tr>
<tr>
<td>Global memory</td>
<td>4096 MiB</td>
</tr>
<tr>
<td>Mem. clock</td>
<td>1600 MHz</td>
</tr>
<tr>
<td>Bus width</td>
<td>512 bits</td>
</tr>
<tr>
<td>Processing elements</td>
<td>240</td>
</tr>
</tbody>
</table>

The resultant performances are then analyzed via an ANOVA approach [25] to quantify the impact of each set of performance factors described in Section 3. The ANOVA analysis determines whether there is a statistically significant difference among the means of each test. It accomplishes this by separating the total observed measurement variation into the variation within a system (assumed due purely to measurement error) and variation between systems (due both to actual differences between systems and to measurement error). Statistically significant differences between system means are determined using an $F$-test that compares variances across the systems.

As this experiment is analyzed with two different sets of multiple factors (Section 3), an $m$-factor ANOVA analysis is required. When two or more factors are present in an ANOVA analysis, the interactions between factors must also be taken into account. Each of these unique factors, and interactions thereof, are called effects. Having obtained the results of the ANOVA analysis, it is then trivial to compute the impact percentage of each effect by taking the ratio of the total variation in a measurement due to each effect with the sum of total variation of all effects and measurement errors. The $m$-factor ANOVA analyses given here are calculated with the statistics package R.

The programs are tested on NVIDIA Tesla C1060 GPUs of compute capacity 1.3 (complete GPU specification found in Table II). 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 3.0, V0.2.1221 with the compiler flag ‘-use_fast_math’. Lattices are tested with side lengths ranging from 32 nodes to 192 nodes in 32 node increments. In an effort to reduce simulation time, a single test is run when overlap occurs between the $y$ and $z$ dimensions (e.g. to the GPU, there is no difference between lattice configurations of $128 \times 32 \times 64$ and $128 \times 64 \times 32$). The simulation performed for the SCSP tests is Poiseuille flow through a rectangular conduit. The multiphase and multicomponent tests simulate initially homogeneous systems with a random perturbation ($<1\%$) to trigger phase separation. The performance of lattice-Boltzmann simulations is measured in MLUPS, indicating the number of lattice site collisions and streaming steps performed in 1 s.

4. PERFORMANCE ANALYSIS

4.1. Simulation overview

By simply using updated hardware, we find an improvement in the performance for single-GPU SCSP tests over those found by Bailey et al. [11]. With the Lagrangian memory access pattern we obtain a maximum performance of 444 MLUPS representing a speedup of 145\% over Bailey et al. The Flip-Flop and Ping-Pong access patterns achieved peak performances of 396 MLUPS and 418 MLUPS, respectively.
Interestingly, all the memory access patterns in the single-GPU tests exhibit a drop in performance when the number of threads per thread block is 128. Other GPU codes, such as the lattice-Boltzmann code from Bailey et al. [11] and Xin Li’s GPU-based hash accelerator [35], display sawtooth patterns in their performance curves. The NVIDIA CUDA Programming Guide [31] hints at an explanation by stating that best results should be achieved when the number of threads per thread block is a multiple of 64. With this in mind, if the number of threads per thread block is incremented by 32 during testing, a sawtooth performance pattern is not unexpected. However, the single-GPU lattice-Boltzmann codes in this paper have an irregular drop in performance when there are 128 threads per thread block, a multiple of 64. This leads to the conclusion of shared memory bank conflicts as a reasonable explanation for the drop in performance. More specifically, that 128 threads per thread block may be an especially bad configuration for shared memory bank conflicts, resulting in a more impactful loss in performance than other configurations.

All multi-GPU simulations in Figures 6 and 7 show increased performance compared to their single-GPU counterparts. Furthermore, the scaling behavior exhibited by all multi-GPU simulations can be seen by plotting how performance changes as the number of GPUs involved in computation increases (Figures 8 and 9). These figures show that all the simulations are operating in a near-linear scaling region. By performing a linear fit to the scaling curves of each simulation type, it is shown that the arithmetic mean of all coefficients of determination ($R^2$) is 0.9884 with the worst fit resulting in an $R^2$ value of 0.9483. As the number of GPUs available in shared memory systems increases with future systems, the extent of the near-linear scaling region can be determined.

![Figure 6. Average speeds in millions of lattice-node updates per second (MLUPS) of single-component single-phase (SCSP) single- and multi-GPU implementations as a function of the number of threads per thread block. SCSP_Pattern refers to the single-GPU version of the code with the corresponding memory access pattern (Pattern), whereas SCSP_OpenMP_NP_X refers to the multi-GPU version of the code run with X GPUs. In SCSP_OpenMP_NP_1, boundary data is collected and copied back to the CPU each time step before being returned back to the GPU (mimicking GPU to GPU memory transfer in larger multi-GPU simulations), whereas in SCSP_Pattern boundary conditions are enforced entirely on the GPU.](image)

Figure 6. Average speeds in millions of lattice-node updates per second (MLUPS) of single-component single-phase (SCSP) single- and multi-GPU implementations as a function of the number of threads per thread block. SCSP_Pattern refers to the single-GPU version of the code with the corresponding memory access pattern (Pattern), whereas SCSP_OpenMP_NP_X refers to the multi-GPU version of the code run with X GPUs. In SCSP_OpenMP_NP_1, boundary data is collected and copied back to the CPU each time step before being returned back to the GPU (mimicking GPU to GPU memory transfer in larger multi-GPU simulations), whereas in SCSP_Pattern boundary conditions are enforced entirely on the GPU.
4.2. Lattice-Boltzmann GPU bottlenecks

As it is likely the case that any given researcher would be using either a single- or a multi-GPU code exclusively, it is prudent to determine the impact of each input factor to single- and multi-GPU implementations exclusively. The impact percentage that each input factor has
Figure 9. Curves illustrating the scaling behavior of the (a) single-component multiphase and (b) multicomponent multiphase OpenMP implementation by the number of threads per thread block. The coefficients of determination ($R^2$) for linear fits to each curve have been included.

Table III. Single GPU—Per cent impact of primary effects on total variation in performance.

<table>
<thead>
<tr>
<th></th>
<th>SCSP</th>
<th>SCSP (Lagrangian)</th>
<th>SCMP</th>
<th>MCMP</th>
</tr>
</thead>
<tbody>
<tr>
<td>kSize</td>
<td>48.0</td>
<td>kSize</td>
<td>70.2</td>
<td>kSize</td>
</tr>
<tr>
<td>Memory access pattern</td>
<td>38.1</td>
<td>nThreads</td>
<td>25.3</td>
<td>nThreads</td>
</tr>
<tr>
<td>nThreads</td>
<td>8.5</td>
<td>nThreads:kSize</td>
<td>4.4</td>
<td>nThreads:kSize</td>
</tr>
<tr>
<td>Memory access pattern:kSize</td>
<td>3.7</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>98.3</td>
<td>Total</td>
<td>99.9</td>
<td>Total</td>
</tr>
<tr>
<td>Total</td>
<td>98.6</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

on the overall code performance is determined using the results of the ANOVA technique described in Section 3.1. Only statistically significant effects with a large contribution to the total variation in the performance are discussed in detail. Single-factor effects are reported as the name of the factor, with interactions of factors reported as the names of the interacting factors separated by a colon (:). Withheld effects are statistically insignificant, low-end contributors, or both. These types of effects are typically higher order interactions and measurement error (system noise). System noise never accounted for more than 0.35% of the total variation in measurements. Consequently, error bars and other indicators of measurement variation are not shown.

Table III lists the contribution percentages of the factors from Table I that most impact the total variation in performance for each of the single-GPU implementations. In the case of the SCSP implementation, differing memory access patterns account for 38% of the variation in performance. However, as only the highest-performing memory access pattern (Lagrangian) is implemented universally, as discussed in Section 3.1, the results of an additional ANOVA analysis focusing only on the Lagrangian memory access pattern are also shown. By removing the memory access patterns as a factor, we find that the remaining single-factor bottlenecks of lattice size and the number of threads per block account for over 98% of the total variation in performance. Based
on this information, it is reasonable for users of GPU lattice-Boltzmann codes looking to improve performance to maximize the lattice size as well as orient the lattice such that the $x$ dimension is the largest dimension.

The validity of these results are confirmed by using the NVIDIA occupancy calculator. Bailey et al. found occupancy to be the major indicator of lattice-Boltzmann GPU codes [36]. When examining these codes with the NVIDIA occupancy calculator, the only factor that changes in the occupancy calculation is the number of threads per block. However, the occupancy calculator indicates that reducing the number of registers per thread block would result in a greater improvement in occupancy. Because the number of registers per block remains constant across tested codes, registers per block would not be a contributing factor to the performance within the set of tested codes. The number of registers per block remains as an avenue to increased performance if a method to decrease register usage can be found.

The analysis of the SCMP and MCMP implementation reveals that the same bottlenecks to the performance in the SCSP case still apply. In contrast, the importance of lattice size and the number of threads per thread block are flipped in the SCMP and MCMP cases. This is not to say that the importance of either factor is significantly diminished in either case. Instead, all these results seem to imply that these single-GPU lattice-Boltzmann fluid flow implementations fall into an area in the GPU parameter space where the problem size and the number of threads per thread block are the primary factors to be examined when optimizing for performance. Within the parameter space, the SCMP and MCMP implementations seem to lie across some equilibrium point from the SCSP implementation.

When considering the results of the ANOVA analysis of the multi-GPU implementations, shown in Table IV, in each case four effects are the source of over 98% of the total variation in performance. These effects are the same as those that primarily impact the single-GPU case, with the inclusion of the number of GPUs involved in computation. Additionally, the impacts of each effect are practically uniform across every case. The increase in the performance corresponding to these effects is seen in Figures 6 and 7.

Comparing the results from single- and multi-GPU implementations yields an interesting realization regarding the architectural considerations that enter GPU computing. In the single-GPU case, the primary effects that bottleneck the performance are the number of threads per block (nThreads) and the size of the lattice being solved (kSize). Interestingly, these effects are associated with the geometry of the simulation. The inclusion of the number of GPUs involved in computation in the multi-GPU implementations is no exception as each additional GPU contributes another lattice of ‘kSize’ in some manner defined by the decomposition.

The shared primary effects between the single- and multi-GPU implementations that bottleneck the performance have little room for improvement due to architectural limitations of the GPU; the

<table>
<thead>
<tr>
<th></th>
<th>SCSP</th>
<th>SCMP</th>
<th>MCMP</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP</td>
<td>47.7</td>
<td>49.1</td>
<td>NP</td>
</tr>
<tr>
<td>nThreads</td>
<td>36.9</td>
<td>38.4</td>
<td>nThreads</td>
</tr>
<tr>
<td>kSize</td>
<td>9.1</td>
<td>8.0</td>
<td>kSize</td>
</tr>
<tr>
<td>nThreads:NP</td>
<td>5.1</td>
<td>3.3</td>
<td>nThreads:NP</td>
</tr>
<tr>
<td>Total</td>
<td>98.8</td>
<td>98.8</td>
<td>Total</td>
</tr>
</tbody>
</table>

Table IV. Multi-GPU—Per cent impact of primary effects on total variation in performance.
available shared memory on the GPU inhibits the number of threads per block, and the available
global memory limits the size of the lattice the GPU is capable of storing. The remaining effects
provide limited room for improvement as combined, they account for slightly more than 1% of
the total variation in performance.

The use of multiple GPUs somewhat circumvents the architectural limitations in single-GPU
simulations. The number of GPUs involved in computation is found to dominate the total variation
in performance, alongside the number of threads per thread block. This means that computa-
tional scientists unfamiliar with GPU computing would be able to apply their existing parallel
programming knowledge with only superficial architectural details regarding GPUs to implement an
effective multi-GPU code, at least for memory-bandwidth-bound codes, such as lattice-Boltzmann
methods. Specific insights into GPU architecture and its effects on computation simply do not
contribute enough to the total variation in performance to make them an impediment. There-
fore, resources and efforts can be focused on simply increasing the number of GPUs rather than
optimizing the performance of a smaller number of GPUs.

4.3. Memory-bound efficiency and utilization

Ryoo et al. [33] use the derived metrics of Efficiency and Utilization (Equations (8) and (9)) to
show that it is possible to find a subset of high performing GPU kernels from the set of available
kernels. Ryoo et al. [33] find the high performing subset of kernels using optimization carving
consisting of threshold carving and tradeoff carving. Threshold carving removes kernels that violate
some aspect of the performance, requiring too much memory bandwidth for example. Given the
fact that lattice-Boltzmann codes are strongly dependent on memory bandwidth, threshold carving
is disregarded entirely here. This leaves the option of using tradeoff carving with a Pareto-optimal
search. An ANOVA test is performed with the purpose of determining if this procedure is truly
incompatible with our purposes.

In the kernels Ryoo et al. [33] considered, Efficiency and Utilization were both important.
By normalizing and plotting the Utilization and Efficiency values of varying kernel configurations
against each other, one should find high performing kernels among the right- or top-most regions
of the plot. Restated, the highest performing kernels should lie on the upper-right hull of the plot
(e.g. the Pareto-optimal curve). However, this is not always the case. Although infrequent, it is
possible for the highest performing kernel to lie some distance away from the Pareto-optimal curve.
Unfortunately, all the high-performing lattice-Boltzmann kernels from every implementation lie
some distance from the Pareto-optimal curve, thereby reducing the applicability of the tradeoff
carving method.

The low applicability of the tradeoff carving method is verified by the results of an ANOVA
analysis. For ANOVA results to agree with the Pareto-optimal search method, both Efficiency and
Utilization should be statistically significant contributing factors to the performance. However,
we find that only Utilization is a notable effect in every single- and multi-GPU simulation in
Tables V and VI. Across all simulations Utilization is typically found to be responsible for
between 26.8 and 36.2% of the total variation. Although, in the outstanding case of the single-
GPU SCMP implementation, Utilization is the primary bottleneck constituting 44.5% of the total
variation in performance. Unlike Utilization, Efficiency is not represented well across all simulation
types. In four of the six implementations, Efficiency is shown to represent less than half the
total variation of Utilization. It is only in the single-GPU SCMP and MCMP simulations where
Table V. Single GPU—Per cent impacts of primary effects impacting total variation in performance.

<table>
<thead>
<tr>
<th></th>
<th>SCSP</th>
<th>SCSP (Lagrangian)</th>
<th>SCMP</th>
<th>MCMP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory access pattern</td>
<td>39.2</td>
<td>kSize</td>
<td>50.4</td>
<td>Utilization</td>
</tr>
<tr>
<td>kSize</td>
<td>38.0</td>
<td>Utilization</td>
<td>36.2</td>
<td>Efficiency 27.2</td>
</tr>
<tr>
<td>Utilization</td>
<td>11.6</td>
<td>Efficiency</td>
<td>10.1</td>
<td>kSize</td>
</tr>
<tr>
<td>Memory access pattern:kSize</td>
<td>6.5</td>
<td>Utilization:kSize</td>
<td>2.4</td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>95.3</td>
<td>Total</td>
<td>99.1</td>
<td>Total</td>
</tr>
</tbody>
</table>

Table VI. Multi-GPU—Per cent impacts of primary effects impacting total variation in performance.

<table>
<thead>
<tr>
<th></th>
<th>SCSP</th>
<th>SCMP</th>
<th>MCMP</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP</td>
<td>47.4</td>
<td>Utilization</td>
<td>NP</td>
</tr>
<tr>
<td>Utilization</td>
<td>26.8</td>
<td>NP</td>
<td>32.5</td>
</tr>
<tr>
<td>kSize</td>
<td>11.4</td>
<td>kSize</td>
<td>23.5</td>
</tr>
<tr>
<td>Efficiency</td>
<td>10.8</td>
<td>Efficiency</td>
<td>7.5</td>
</tr>
<tr>
<td>Total</td>
<td>96.4</td>
<td>Total</td>
<td>96.2</td>
</tr>
</tbody>
</table>

Efficiency results in an effect that is more noteworthy, at 27.2 and 32.1% of the total variation, respectively.

Importantly, the results in Tables V and VI indicating that Utilization plays a more important role than Efficiency, are in agreement with what one should expect from memory-bandwidth-bound kernels. With these kernels it is less important to execute more instructions across all executing threads than to make use of the available GPU compute resources to mask memory transfers. In conjunction with the finding that there is minimal interaction between the effects of Utilization and Efficiency, it can be concluded that the statement by Ryoo et al., that program optimization carving is not valid for memory-bandwidth-bound kernels, is accurate.

Finally, this suggests that developers considering these metrics should focus on increasing Utilization over Efficiency to achieve better performance when dealing with memory-bandwidth-bound kernels. This can be achieved by reducing the number of synchronization regions, by increasing the block size, or by reducing register use to increase the number of blocks executed per multiprocessor.

5. CONCLUSIONS AND FUTURE WORK

GPGPUs provide a SIMD environment well suited to the lattice-Boltzmann method. Here we have demonstrated that the performance gains seen in single GPGPU implementations scale favorably when applied to multiple GPUs employing OpenMP on a shared memory system. Three lattice-Boltzmann models were considered: the standard SCSP model, the multiphase model of Shan and Chen [5], and the multicomponent model of He and Doolen [30].

The following properties of the GPU architecture were investigated with ANOVA: the number of PTX instructions, the number of threads per thread block, the overall lattice size,
and the number of processors. For memory-bound GPU kernels, such as lattice-Boltzmann simulations, the ANOVA analysis showed that effects corresponding with easily understandable aspects of GPU programming (number of threads per thread block and overall problem size) account for the largest percentage of variation in performance (>97%). This result holds across all lattice-Boltzmann simulation types tested (SCSP, SCMP, and MCMP). We, therefore, conclude that the necessity of understanding GPU architectures is reduced through multi-GPU implementations. In such multi-GPU cases, more fundamental parallel programming concepts (e.g. problem size and the degree of multiprogramming at the thread and processor level) dominate.

This study also considered two derived performance metrics described by Ryoo et al. [26, 33]: Efficiency and Utilization. We show that neither threshold nor tradeoff carving is applicable to lattice-Boltzmann codes as none of our high-performing kernels lie near the Pareto-optimal curve. Our analysis also shows that only the Utilization metric results in a meaningful effect across all lattice-Boltzmann implementations. Finally, the finding that there is no statistically significant interaction between Efficiency and Utilization to provide any meaningful variation in performance agrees with the statement of Ryoo et al. [26] that their program optimization carving method is not applicable to bandwidth-bound GPU kernels.

Ultimately, the ANOVA results presented here can be approached by adopting an Amdahl’s Law [37] perspective. The effects that provide the largest percentage of total variation in performance are those that will provide the greatest returns when improved. To that end, focusing on lattice size and orientation (in relation to the overall lattice size and the number of threads per thread block) will yield the greatest improvements in performance for single-GPU lattice-Boltzmann implementations. We show that an even greater performance improvement is available with the introduction of multiple GPUs in a shared memory context. Additional investigations would indicate to what degree multiple GPUs would be useful on large-scale shared memory systems. Further work should also explore the performance of lattice-Boltzmann codes on large-scale distributed GPU clusters.

APPENDIX A: MEMORY ACCESS PATTERN DESCRIPTIONS

We have employed three different memory access patterns to control the propagation of fluid packets perpendicular to the thread-block axis, referred to in the text as ‘Ping-Pong’, ‘Flip-Flop’, and ‘Lagrangian’ patterns. In this appendix, we provide a detailed description of the differences between the different memory access patterns. To better distinguish the different memory access patterns, the following notation is introduced to describe the global memory location occupied by the fluid packets:

\[ f_i(y, t, j), \]  

where \( f_i \) denotes a fluid packet moving in the direction \( c_i \) and the terms inside the brackets following \( f_i \) denote an array location holding the fluid packet variable at time \( t \), i.e. \( f_i(y, t, j) \) represents the global memory location at time \( t \) associated with global memory indices \( y \) and \( j \). The first index, \( y \), is a three-part vector representing a set of coordinates (similar to lattice coordinates) and the second, \( j \), is a scalar index between 1 and 19 (similar to the lattice direction).
In the Ping-Pong memory access pattern, the global memory coordinates \( y \) equal the lattice-coordinates \( x \) at which the fluid packet is located and index \( j \) is equal to the fluid packet direction \( i \). Thus the collision rule given in Equation (1) is

\[
f_i^b(x + c_i \Delta t, t + \Delta t, i) = (1 - \lambda) f_i^a(x, t, i) + \lambda f_i^{eq}(x),
\]

(A2)

where \( f_i^{eq}(x) \) represents the equilibrium fluid packet density at position \( x \) and \( f_i^a \) and \( f_i^b \) represent two separate global memory arrays. The collision rule for the alternate time step is

\[
f_i^a(x + 2c_i \Delta t, t + 2\Delta t, i) = (1 - \lambda) f_i^b(x + c_i \Delta t, t + \Delta t, i) + \lambda f_i^{eq}(x + c_i \Delta t),
\]

(A3)

and so on. A single global memory array cannot be used for this rule, as fluid packets at time \( t + \Delta t \) would overwrite those at time \( t \). In another context, this might be avoided by temporarily storing the fluid packets in a local buffer, however, this solution does not work with the GPU programming environment where the order of thread block execution is not guaranteed.

Instead, we adopt memory access patterns that overwrite the same location in global memory: namely, the Flip-Flop pattern and the Lagrangian pattern.

The Lagrangian memory access pattern is perhaps the simplest of the two, given by

\[
f_i(x_o, t + \Delta t, i) = (1 - \lambda) f_i(x_o, t, i) + \lambda f_i^{eq}(x_o + c_i t),
\]

(A4)

where \( x_o \) is the fluid packet location at time \( t = 0 \). In this pattern, fluid packets occupy the same positions in global memory while all collision and streaming steps are calculated. Before the result of the simulation is returned to the CPU, the fluid packets are rearranged back to an Eulerian configuration (i.e. where the position in global memory, \( y \), equals the simulated position of the fluid packet \( x \)). The cost of the rearrangement is less than a single streaming and collision step.

The Flip-Flop memory access pattern exploits the fact that the global memory index \( j \) need not match the fluid packet direction \( i \). As the name suggests, the pattern has two steps. The first step swaps the fluid packets \( f_i \) and \( f_{\tilde{i}} \) (where \( \tilde{i} \) indicates the reverse direction, i.e. \( c_{\tilde{i}} = -c_i \)), but otherwise keeps the packets at the same global memory location,

\[
f_i(x, t + \Delta t, \tilde{i}) = (1 - \lambda) f_i(x, t, i) + \lambda f_i^{eq}(x).
\]

(A5)

The second step restores the fluid packet directions and positions to the Eulerian configuration,

\[
f_i(x + 2c_i \Delta t, t + 2\Delta t, i) = (1 - \lambda) f_i(x, t + \Delta t, \tilde{i}) + \lambda f_i^{eq}(x + c_i \Delta t).
\]

(A6)

If an odd number of time steps is performed, an additional step rearranges the fluid packets into the correct configuration before they are returned to the CPU.

**APPENDIX B: MULTIPLE-GPU PSEUDO CODE**

This appendix outlines key steps in the multi-GPU implementations.
1. Allocate memory on CPU for GPU-GPU transfer, initialize CPU-CPU pipelines (MPI only).

2. Create one thread per GPU: #pragma omp parallel num_threads(numGPUs)

3. Within the #pragma omp parallel num_threads(numGPUs) directive:
   3.1. Get the thread id: int th_id = omp_get_thread_num();
   3.2. Assign thread to GPU: cudaSetDevice(th_id);
   3.3. Copy initial conditions and problem geometry from CPU to GPU
       e.g. float* f_D; int sizeF = 19*GPUNodes*sizeof(float);
           cudaMalloc((void**)&f_D, sizeF);
           cudaMemcpy(f_D, f[th_id], sizeF, cudaMemcpyHostToDevice);
   3.4. for(timestep = 0; timestep < maxTimesteps; timestep += outputFrequency):
       3.4.1. for(t = 0; t < outputFrequency; t++):
           Multiple phase/multiple component models only:
           i. Calculate \( \phi \) for outer shell of nodes
           ii. Copy outer shell \( \phi \) from GPU to CPU
               Copy outer shell \( \phi \) from CPU to GPU (MPI only)
               Copy neighboring \( \phi \) from CPU to GPU
           iii. Calculate inner \( \phi \) values (simultaneous with ii) on the GPU.
           All models:
           iv. Perform \( f_i \) streaming/collision step for outer node shell
           v. Copy outgoing outer shell \( f_i \) from GPU to CPU
               Copy outgoing outer shell \( f_i \) from CPU to GPU (MPI only)
               Copy incoming outer shell \( f_i \) from CPU to GPU
           vi. Perform inner \( f_i \) streaming/collision (simultaneous with v) on the GPU.
   3.4.3. For the Lagrangian memory access pattern\(^\ast\):
           Rearrange \( f_i \) to Eulerian configuration (location in memory = spatial position).
   3.4.4. Copy \( f_i \) data from GPU to CPU and save result to file.
           e.g. cudaMemcpy(f[th_id], f_D, sizeF, cudaMemcpyDeviceToHost);
   3.5. Free GPU memory:
           e.g. cudaFree(f_D);


ACKNOWLEDGEMENTS

This work was supported in part by National Science Foundation (NSF) Grants No. DMS-0724560, EAR-0510723, and EAR-0941666. 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. MOS also thanks the George and Orpha Gibson endowment for its generous support of the Hydrogeology and Geo Fluids research group. We are also grateful to Peter Bailey, Erich Frahm, Victor Barocas, Paul Woodward, and NVIDIA for their advice and support, and thank our Reviewers for their excellent comments.

\(^\ast\)or Flip-Flop pattern if output frequency is odd.
LATTICE-BOLTZMANN SIMULATION PERFORMANCE ON GPU CLUSTERS

REFERENCES


