Accelerating Lattice Boltzmann Fluid Flow Simulations Using Graphics Processors

Peter Bailey[1], Joe Myre[2], Stuart D. C. Walsh[3], David J. Lilja[4], Martin O. Saar[5]
{bail0253, myrex013, sdcwalsh, lilja, saar}@umn.edu

[1,4] Department of Electrical and Computer Engineering
[2] Department of Computer Science
[3,5] Department of Geology & Geophysics
University of Minnesota, Twin Cities

***Abstract*—Lattice Boltzmann Methods (LBM) are used for the computational simulation of Newtonian fluid dynamics. LBM-based simulations are readily parallelizable; they have been implemented on general-purpose processors [1][2][3], field-programmable gate arrays (FPGAs) [4], and graphics processing units (GPUs) [5][6][7]. Of the three methods, the GPU implementations achieved the highest simulation performance per chip. With memory bandwidth of up to 141 GB/s and a theoretical maximum floating point performance of over 600 GFLOPS [8], CUDA-ready GPUs from NVIDIA provide an attractive platform for a wide range of scientific simulations, including LBM. This paper improves upon prior GPU LBM results for the D3Q19 model [7] by increasing GPU multiprocessor occupancy, resulting in an increase in maximum performance by 20%, and by introducing a space-efficient storage method which reduces GPU RAM requirements by 50% at a slight detriment to performance. Both GPU versions are over 28 times faster than a quad-core CPU version utilizing OpenMP.**

I. Introduction

The Lattice-Boltzmann Method (LBM) is a technique for performing computational fluid dynamics simulations. LBM-based computations are particularly well-suited for simulations of fluid flow through detailed (micro-scale) descriptions of pore-spaces, and are capable of simulating many complex fluid flow problems including turbulent flows [9], multicomponent miscible and immiscible fluids[10], [11], and free surface problems[12]. Examples of the diverse applications of LBM include modeling flow in complex pore space geometries[13] and in a static mixer [14]. LBM codes have also been used in problems such as geofluidic flows, *e.g.,* gas, water, oil, or magma flow through porous media [15], macro-scale solute transport [16], the dispersion of airborne contaminants in an urban environment [17], impact effects of tsunamis on near-shore infrastructure [18], melting of solids and resultant fluid flow in ambient air [19], and to determine permeability of materials [20][15].

LBM simulations are modeled within a one, two, or three dimensional lattice, each node in the lattice representing a quantized region of space containing either fluid or solid. The fluid is simulated by fluid packets that propagate through the lattice in discrete time steps, and collide with each other at lattice points. As collisions are restricted to the local lattice nodes, the collision computation only depends on data from neighboring nodes. This spatial locality of data access makes LBM an excellent candidate for parallelization. While easily parallelizable, LBM simulations are not "embarrassingly" parallel in nature since communication between neighbors is required at each timestep.

LBM simulations are particularly suited for parallelization on special-purpose accelerators such as graphics processing units (GPUs) [5][7][6] or the Cell Broadband Engine [21]. As special-purpose accelerators, GPUs are designed to process large graphics data sets quickly, providing real-time interactive visual feedback. Use of NVIDIA's CUDA (Compute Unified Device Architecture) extension of the C language [22] to employ GPUs in LBM simulations has been shown to provide speedups of one to two orders of magnitude over general-purpose CPU versions [6]. This paper improves upon existing GPU LBM implementations by two non-orthogonal methods: 1) providing an increase in performance primarily by increasing GPU multiprocessor occupancy, and 2) introducing a memory access technique to reduce memory space requirements by 50%, allowing efficient simulation of much larger lattices on the GPU.

The remainder of this paper provides an overview of how LBM simulations are computed and our improvements (Sec. II), the results of our improvements (Sec. III), and a comparison of related GPU implementations

(Sec. IV).

II. LBM Computation

This lattice Boltzmann implementation uses the D3Q19 layout (shown in Fig. 2), a regular three-dimensional lattice (D3) with 19 distinct fluid packet velocities (Q19). Time is advanced in the lattice-Boltzmann simulation in discrete timesteps composed of two parts: a collision step, in which momentum is exchanged between the fluid packets at each node; and a streaming step, in which the fluid packets are shifted to the next node along their path.

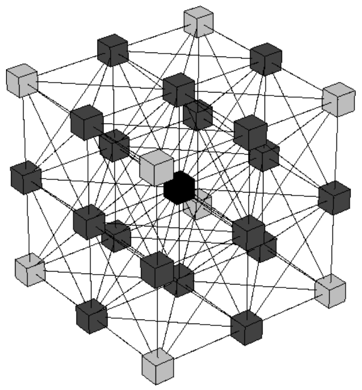


Fig. 1: D3Q19 lattice interconnection pattern

The collision step accepts 19 incoming fluid packets arriving at the node as input, and produces 19 outgoing fluid packets, the densities of which are redistributed according to a local collision rule. The manner in which the fluid packets are redistributed depends on the behavior of the fluid at the node. Three collision rules were implemented in our simulations: 1) the standard single-relaxation D3Q19 collision rule for simulating fluid dynamics; 2) a bounce-back collision rule for simulating solid-fluid boundaries [23]; and 3) a collision rule for simulating pressure-controlled boundary conditions.

During the streaming step, eighteen of the outgoing fluid packets from each node are distributed to the corresponding incoming fluid packet locations for the neighboring nodes. In the following timestep, the nodes use these incoming fluid packets and the one stationary fluid packet in the next collision step. As shown by Pohl et al. [24], the collision and streaming steps can be combined to reduce memory traffic within the system. This technique enhances the utilization of on-chip memory by improving the temporal locality of access to fluid packet data.

A. Memory Access Patterns

Two distinct fluid packet access patterns are tested in this implementation. The first, subsequently referred to as the “A-B” pattern (Sec. II-A1), requires two sets of fluid packets to reside in memory at all times, while the second, subsequently referred to as the “A-A” pattern (Sec. II-A2), requires only one set of fluid packets in RAM.

1) “A-B” Pattern: The A-B memory access pattern, commonly referred to as “ping-pong buffering,” requires two sets of fluid packets to reside in memory throughout the course of the simulation. The source and destination sets are alternated at each timestep (Fig. 2). In Fig. 3, a D2Q9 version of the A-B pattern is shown. Black arrows represent fluid packets involved in the collision and streaming steps of the center node. Note how, from Fig. 3a to Fig. 3b, the arrows travel in the direction of their orientation. This method is used by Tölke and Krafczyk [6] and Habich [7] on the GPU, and Wilke et al. [24], Walsh et al. [16], and others on the CPU. We use this method for our CPU version of the LBM simulation.

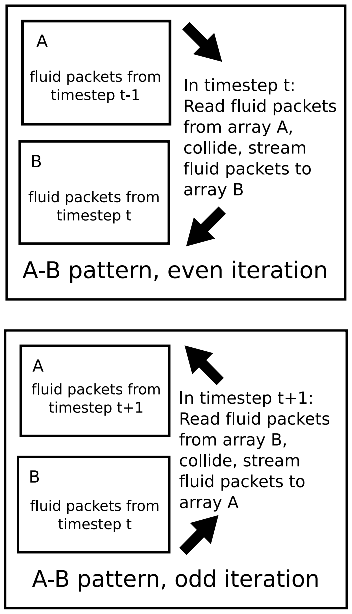
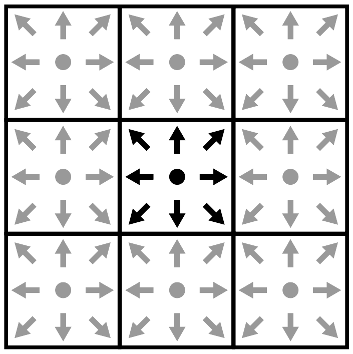

A
fluid packets from timestep t-1
B
fluid packets from timestep t
In timestep t: Read fluid packets from array A, collide, stream fluid packets to array B
A-B pattern, even iteration
A
fluid packets from timestep t+1
B
fluid packets from timestep t
In timestep t+1: Read fluid packets from array B, collide, stream fluid packets to array A
A-B pattern, odd iteration

Fig. 2: Execution of the A-B memory pattern (ping-pong)

2) “A-A” Pattern: The A-A pattern needs only one set of fluid packets in memory (Fig. 4), reducing the memory requirement for a given lattice size by a factor of two. This pattern alternates execution of two routines. The routine for even-iterations (A-A:1) executes a single collision (Fig. 5), and the routine for odd-iterations (A-A:2) executes a combined streaming-collision-streaming step (Fig. 6). Fig. 5a shows fluid packets following the second streaming step in Fig. 6b.

Routines for both even and odd iterations write outgoing fluid packet data to the same locations from which


(a) A-B pattern incoming fluid packet reads

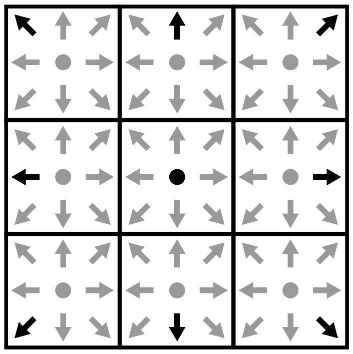

(b) A-B pattern outgoing fluid packet writes (to separate array)

Fig. 3: Simplified D2Q9 version of A-B pattern fluid packet reads (a) and streaming writes after collision (b). The center fluid packet is stationary.

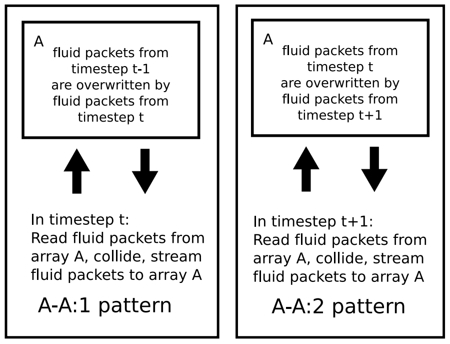
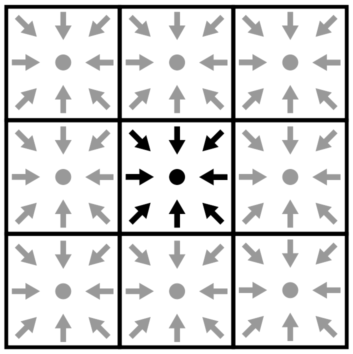

A
fluid packets from timestep t-1 are overwritten by fluid packets from timestep t
In timestep t: Read fluid packets from array A, collide, stream fluid packets to array A
A-A:1 pattern
A
fluid packets from timestep t are overwritten by fluid packets from timestep t+1
In timestep t+1: Read fluid packets from array A, collide, stream fluid packets to array A
A-A:2 pattern

Fig. 4: Execution of the A-A memory pattern


(a) A-A:1 pattern incoming fluid packet reads

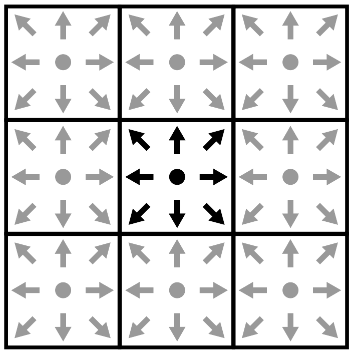

(b) A-A:1 pattern outgoing fluid packet writes (to same array)

Fig. 5: D2Q9 version of A-A:1 pattern fluid packet reads (a) and streaming writes after collision (b).

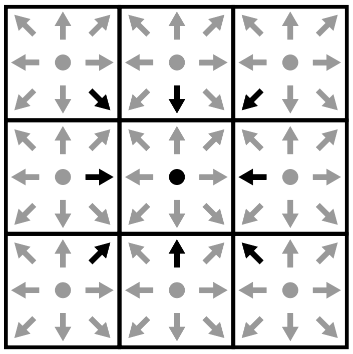
incoming data was read, a characteristic which allows maintenance of only one set of fluid packets in memory, greatly increasing lattice sizes that can be stored and computed entirely on the GPU. This allows the lattice nodes to be computed in any order, which is necessary for the GPU architecture for reasons explained in Sec. II-B. The ability to compute nodes in any order is a key improvement over the "compressed grid" layout [24], which uses roughly the same amount of storage as the A-A pattern, but requires a specific ordering of node computations, thus ruling out efficient implementation on the GPU.

B. Thread Execution

In this implementation, each GPU thread computes one timestep, consisting of the collision and streaming steps, for a single lattice node. Each GPU program, or "kernel," in this implementation is capable of calculating all three collision rules, resulting in warp divergence if a warp contains nodes of multiple collision rules.

Due to data dependencies between threads in distinct blocks, only one LBM timestep per GPU kernel execution is achieved. However, threads within a block can synchronize using a single instruction, and can share data through 16 KB of shared memory per multiprocessor.

Multiprocessor occupancy is defined as the ratio of the number of active warps per multiprocessor to the maximum number of active warps [22]. Occupancy is specific to a single GPU kernel because a multiprocessor can only run one kernel at a time. Running multiple warps allows a multiprocessor to hide global memory access latency (around 400 cycles) by switching out stalled warps for warps ready to execute instructions. Similarly, running multiple blocks per multiprocessor can hide block synchronization latency. In order to


(a) A-A:2 pattern incoming fluid packet reads

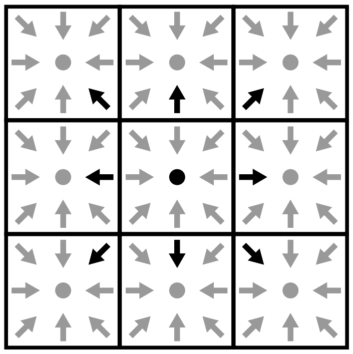

(b) A-A:2 pattern outgoing fluid packet writes (to same array)

Fig. 6: D2Q9 version of A-A:2 pattern fluid packet reads (a) and streaming writes after collision (b).

maximize the memory latency that may be hidden, the number of warps able to be run simultaneously must be maximized. The number of concurrently runnable warps is limited by the number of registers per multiprocessor, the amount of shared memory per multiprocessor, the number of threads per block, and a hardware-based limit.

Each multiprocessor has a limited number of registers. These registers are used by all concurrently running threads. Therefore, using fewer registers per thread allows more thread blocks to be run concurrently. Similarly, shared memory is used by the same threads and must be divided accordingly.

C. *Memory Constraints*

The design of the memory architecture on the GPU results in maximum memory bandwidth when threads align their memory accesses correctly. If the pattern of memory access prevents this coalescence, then bandwidth is reduced by up to an order of magnitude. For proper alignment, consecutive threads in a warp must access consecutive locations in memory, starting from a globally aligned base memory location. For specific requirements, see Sec. 5.1.2.1 in [22].

Two previous GPU LBM implementations by Habich [7] and Tölke et al. [6] have shown that a structure-of-arrays, or collision-optimized layout [25] of fluid packets in memory fits the GPU hardware well. This layout indexes fluid packets in global memory by the following variables, in the following order: X, Y, Z, fluid packet direction, and timestep. The layout is optimized for simulation performance on processors with cache memory, but it is ideal on the GPU because it allows for easy coalescence of global memory reads and writes. Given such a layout, consecutive threads within a thread block compute outgoing data for consecutive nodes in the X-dimension. This allows one block of GPU threads to compute an nX-by-1-by-1 section of the lattice while maintaining coalesced global memory access, where nX is equal to the X-dimension of the lattice.

This LBM implementation utilizes shared memory to maintain coalesced global memory access.

When two separate arrays are kept in memory for incoming and outgoing fluid packets (A-B pattern, Sec. II-A1, Fig. 2), all global memory reads are automatically aligned. This is due to the collide-stream pattern of execution (Fig. 3). Writes from each node to neighboring nodes along only the Y- and Z- dimensions are automatically aligned. However, writes along directions with an X component are complicated in that they require propagation in shared memory, followed by an aligned write to global memory as described in [6].

When a single fluid packet array is present (A-A pattern, Sec. II-A2, Fig. 4), global memory reads and writes along only the Y- and Z- dimensions are aligned, but reads and writes along directions with an X component require an exchange in shared memory before and after the collision step. Neither access pattern results in shared memory bank conflicts.

III. Experimental Methodology and Results

A. *Methodology*

We perform test runs on three different NVIDIA GPUs, an 8800 GTX, an 8800 Ultra, and a 9800 GX2. It should be noted that the 9800 GX2 is composed of two GPUs, each individually addressable by CUDA programs. The two GPUs communicate through the PCI Express 1.0 x16 bus, just as any other two GPUs would do in one system. In this paper, tests on the 9800 GX2 use only one of the two GPUs. The specifications for each of these GPUs are found in Table I.

For additional comparison we perform test runs on the Intel quad-core CPU in the system housing the GPU, the specifications of which can be found in Table II.

We use the G++ compiler version 4.1.2 for CPU code and NVCC release 2.0, V0.2.1221 for GPU code. We use nvcc compiler flags “-O3” and “-maxrregcount ”, followed by “32” or “64”, depending on the GPU kernel being compiled. For G++, we use the compiler flags “-funroll-loops -fno-trapping-math -O3”.

Tested lattice sizes include cubic lattices with side lengths ranging from 32 nodes to 160 nodes in 32-node increments. Non-cubic lattices of the following dimensions are also tested: 256x128x128, 128x128x64, 64x250x250, 192x156x156, and 96x192x192. The simulation performed on each lattice is Poiseuille flow through a box.

Performance of lattice Boltzmann simulations is measured in Lattice node Updates Per Second (LUPS), which indicates the number of lattice site collision and streaming steps performed in one second. For recent implementations, a more convenient unit of measurement is Million Lattice node Updates Per Second (MLUPS). It should be noted that although the same metric is used for different lattice layouts (*e.g.,* D3Q13, D2Q9) and precisions (single vs. double), one lattice node update in one layout may require more memory transfers and computation than a lattice node update in another layout. For example, a single-precision D3Q19 lattice node update reads and writes at least $19 * 4 = 76$ bytes, while a single-precision D3Q13 lattice node update reads and writes $13 * 4 = 52$ bytes.

B. Results

Two key improvements are presented in this paper. First is an increase in maximum simulation speed by 20% over published results, and second is a reduction in memory requirements by 50% over published GPU-based lattice Boltzmann implementations. It should be noted that these improvements are not orthogonal; when storage reduction is utilized, the performance gain is less, and when maximum simulation speed is desired, storage reduction is not available. Both the 20% performance increase and the disparity between the two implementations are correlated with changes in occupancy. Habich [7] uses 40 registers per thread, resulting in a maximum occupancy of 25% and maximum performance of 250 MLUPS. When two arrays are present in our implementation (A-B pattern), the GPU kernel uses 32 registers per thread, allowing a maximum of 33% occupancy. The major difference between the A-B pattern and the A-A pattern is the lower average occupancy of the GPU kernels used. In the A-A pattern, two kernels are employed. One (A-A:1) uses 32 registers per thread, while the other (A-A:2) uses 64. On the 8800 GTX, kernels using 64 registers per thread can achieve a maximum occupancy of 17%.

As seen in Fig. 7, GPU LBM performance for the A-B pattern positively correlates with GPU occupancy. Additionally, the maximum performance for occupancies less than 33% is 236 MLUPS, and no simulation with an occupancy of 33% runs at less than 250 MLUPS.

TABLE I: GPU Specifications

Device	Clock	RAM	Mem. Clock	Bus Width	Processing Elements
8800 GTX	1.35 GHz	768 MiB	900 MHz	384 bits	128
8800 Ultra	1.51 GHz	768 MiB	1080 MHz	384 bits	128
9800 GX2	1.50 GHz	2x512 MiB	1 GHz MHz	256 bits	2x128

TABLE II: CPU Specifications

	Clock	RAM	Mem. Clk	Bus Width	Cores
Intel Q6600	2.4 GHz	4 GiB	400 MHz	64-bit	4

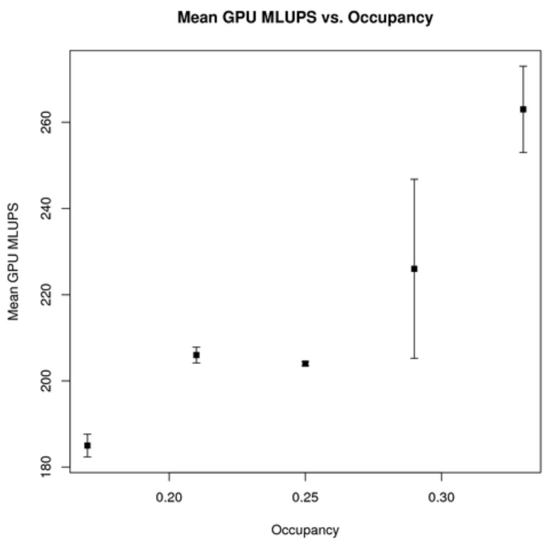


Fig. 7: Mean GPU pattern A-B performance vs. GPU occupancy with 95% confidence interval

Due to differences in occupancy and kernel complexity, the two memory access patterns perform at different levels. For the test cases outlined in Sec. III-A, the average performance of the A-A pattern implementation is 209 MLUPS, with a maximum measured performance of 260 MLUPS. The A-B pattern implementation achieves an average performance of 246 MLUPS on the same test cases, with a maximum measured performance of 300 MLUPS.

The 8800 GTX test platform has a theoretical maximum computation-to-memory bandwidth ratio of 345.6

GFLOPS / 86.4 GBps = 4 FLOPS / byte. Thus, the observed FLOPS / byte ratio is 50 GFLOPS / 46.8 GBps = 1.08 FLOPS / byte, or about 25% of the theoretical maximum.

C. Comparison of CPU and GPU versions

The CPU version of the lattice Boltzmann simulation is a single-precision implementation by Walsh et al. [15]. The collision and streaming steps are separate. Two sets of fluid packets are maintained (A-B pattern), and each set uses the following layout: fluid packet direction, X, Y, Z. Parallelization of the CPU implementation is achieved by using OpenMP to subdivide loops into sections that can run on multiple processor cores. The parallelized loops calculate the collision and streaming steps separately for each node.

In the test cases described in Sec. III-A, the CPU implementation, using a single core, achieved an average performance of 6.18 MLUPS with a maximum measured performance of 7.53 MLUPS. When tested across 4 cores, the CPU implementation achieved an average performance of 8.99 MLUPS with a maximum measured performance of 11.1 MLUPS. When compared to the high-performance GPU A-B memory access pattern implementation results in Sec. III-B, it is seen that the A-B pattern achieves average speedups of 40.3 and 27.7 over the single- and quad-core results, respectively, of the CPU implementation.

D. Important Optimizations

To achieve high levels of performance on the GPU, we utilize a number of strategies. Among these are coalesced global memory access, use of shared memory, restricting the number of registers per thread, careful use of conditional statements, and a memory access pattern that allows for the simulation of larger lattices. For the A-B pattern, these strategies result in a 20% increase in maximum MLUPS over the results in [7]. For the A-A pattern, they allow for a 50% reduction in memory requirements for a given lattice size.

To achieve maximum bandwidth to global memory, the GPU's memory controller combines global memory reads and writes from a warp of threads into a single operation if certain conditions are met concerning address alignment and ordering between threads. In the LBM simulation, changing only the layout of fluid packets in global memory is observed to cause performance penalties of up to 70% due to lack of coalescence. 100% coalescence is achieved by propagating, in shared memory, directions that would have caused non-coalescence if written directly to global memory. Each multiprocessor has a limited number of registers, which are divided between concurrently executing thread blocks. The lower the number of registers required per thread, the higher the number of thread blocks that can be run concurrently. The compiler can be coerced into limiting the number of registers per thread (with the –maxrregcount switch), but this practice can lead to substitution of (relatively slow) global GPU memory for registers. Still, trading off higher occupancy for a few extra global memory accesses occasionally improves performance. The kernels used in this LBM implementation require between 32 and 64 registers each, giving a maximum warp occupancy of 33%.

To reduce memory requirements for lattice storage, the A-A pattern of execution maintains a single array of fluid packets, rather than two arrays as in previous GPU implementations [6] [7]. This reduction is achieved by alternating two kernels that read fluid packets from, and write fluid packets to, the same memory locations, thus avoiding the necessity that lattice node computations be executed in a particular order. The use of two kernels for the A-A pattern, one requiring 64 registers per thread, adds a significant amount of execution overhead.

IV. Related Work

Many parallel implementations of lattice Boltzmann simulations have been produced for CPU clusters, as well as single and clustered GPUs. Unless otherwise noted, GPU architectures use single-precision floating point, and other architectures use double-precision. Li et al. [26] used a single GPU to run a D3Q19 lattice Boltzmann simulation, yielding a speedup of 15.3 over a CPU implementation. The maximum processing rate of the single GPU was 3.8 MLUPS. Fan et al. [5] used GPUs to accelerate lattice Boltzmann simulations, resulting in a speedup of 21.4 over a cluster of an equal number of CPUs. The maximum processing rate of the GPU cluster was 49.2 MLUPS. Körner et al. [1] ran D3Q19 LBM simulations on three clusters, each with 64 processors. On an Itanuim2 cluster, 64 nodes achieved 180 MLUPS.

Implementations utilizing CUDA tools have fared better; Tölke and Krafczyk [6] ported a D3Q13 LBM simulation to an NVIDIA GeForce 8800 Ultra GPU using CUDA tools, reaching 592 MLUPS. Habich [7], also using CUDA, produced a D3Q19 LBM simulation that achieved 250 MLUPS on a single GeForce 8800 GTX GPU.

The D3Q13 model [6] is simpler than the D3Q19 used in this study model in that it tracks only 13 velocity vectors at each node, instead of 19. Also, due to the interconnection pattern of nodes in a D3Q13 lattice, a grid of nodes can be decomposed into two independent sub-grids, thus reducing computational requirements by 50% if only one sub-grid is simulated. However, even if both sub-grids are simulated, the effective resolution of fluid flow is less than that of the D3Q19 layout in which all nodes are interconnected. Consequently, D3Q19 LBM implementations are frequently preferred since they produce higher quality scientific results, particularly when highly-resolved flow vector fields in complex pore spaces are desired [15].

Compared to the D3Q13 implementation described in [6], our implementation uses a similar percentage of theoretical GPU memory bandwidth (54.3% vs. 61%, respectively). When "register spillage" is taken into account, the percentage utilization of theoretical bandwidth is the same as in [6].

This difference in memory bandwidth utilization is attributable to three main factors: 1) the addition of six additional fluid packets per lattice node for the D3Q19 model and their associated computation and bandwidth requirements, 2) the difference in memory clock speeds (1080 MHz vs. 900 MHz) between the NVIDIA 8800 Ultra used by Tölke et al. [6] and the NVIDIA 8800 GTX used in this implementation, and 3) "register spillage," a phenomenon that occurs when the CUDA compiler successfully implements a program in a user-specified number of registers, but uses some global memory as a substitute for additional registers. Register spillage has the potential to increase multiprocessor occupancy, thus increasing parallelism and latency-hiding. Unfortunately, there is a trade-off involved; global memory access is much slower than register access (∼400 clock cycles vs. ∼1 clock cycle). In the "A-B" kernel of this implementation, 32 registers are used for each kernel, but global memory replaces the equivalent of four registers per thread, reducing the effective bandwidth for fluid packet transfers by $4/(32+4) = 11\%$. The "register spillage" witnessed in this application is due to the increased complexity of the D3Q19 model. If 36, rather than 32, registers were used per thread, multiprocessor occupancy would suffer at some block sizes (block size = lattice X dimension).

Assuming that global memory bandwidth, rather than floating point calculation, is the limiting factor, the estimated performance, when converting from the D3Q13 implementation in [6] to a D3Q19 implementation, is 592 MLUPS $*13/19*(900\ MHz)/(1080\ MHz)*32/(32+4) = 300$ MLUPS. Thus, our implementation, resulting in 300 MLUPS, achieves 100% of the estimated peak performance, when limited by memory bandwidth.

Multiple studies cite cache effects and relatively low memory bandwidth as reasons for sub-optimal LBM performance on general-purpose processors [25][1]. Vector machines avoid such problems by design. Wellein et al. [25] and Pohl et al. [1] used an LBM code to compare the performance of various CPU-based systems with that of vector machines. In these studies it was discovered that vector processors are particularly suited for LBM computations, and that general-purpose processors are adversely affected by cache behavior. With no cache, a single NEC SX6 vector processor outperformed a single Itanium2 processor by a factor of 8.

The graphics processing unit (GPU) architecture (Single Instruction, Multiple Thread) is similar to that of a vector machine (Single Instruction, Multiple Data). GPUs are designed for high pixel throughput, which requires high memory bandwidth for large data sets. By maintaining high bandwidth to main memory, GPUs avoid some limitations imposed by cache size and cache effects typical of general-purpose processors. For example, an application may perform poorly on a CPU due to cache capacity misses or cache trashing. The same application may perform better on a GPU, because the GPU has no cache that fills the same role.

V. Conclusions and Future Work

We show that a GPU-based D3Q19 lattice Boltzmann simulation of fluid flow can achieve 300 MLUPS using one pattern of execution, and efficient utilization of GPU RAM using another pattern of execution. Both patterns result in speedups of more than 28x over a quad-core CPU implementation using OpenMP. The increase in performance over a previous D3Q19 GPU implementation [7] is due to increased GPU multiprocessor occupancy. Based on memory bandwidth utilization, our D3Q19 simulation is as efficient as a D3Q13 GPU implementation [6], and models fluid flow at a higher level of detail.

Future work will include extension of the GPU-based lattice Boltzmann simulation to multiple GPUs and GPU clusters. Because the maximum lattice size that can be simulated efficiently is determined by GPU RAM, using multiple GPUs will allow for the simulation of larger domains.

Programming in NVIDIA's CUDA environment is a significant departure from traditional parallel environments, such as MPI or OpenMP. Failure to adhere to a few key strategies, such as global memory coalescence, avoiding divergent warps, and maximizing occupancy, can result in lackluster performance. However, memory coalescence requirements have been reduced in NVIDIA's more recent GPUs.

Despite the additional difficulty, GPUs offer substantial performance benefits to a wide variety of applications, *e.g.,* [18] [26] [27] [28] [15], and CUDA allows more precise utilization of graphics hardware than other alternatives, including OpenGL, Rapidmind [29], and BrookGPU [30].

VI. Acknowledgments

This work was supported in part by the National Science Foundation under Grants No. EAR-0510723 and DMS-0724560. We would also like to thank the George and Orpha Gibson Endowment for its generous support of the Hydrogeology and Geofluids Research Group, as well as Erich Frahm, the Laboratory for Computational Science & Engineering, and NVIDIA for their advice and support.

References

[1] T. Pohl, F. Deserno, N. Thurey, U. Rude, P. Lammers, G. Wellein, and T. Zeiser, "Performance evaluation of parallel large-scale lattice boltzmann applications on three supercomputing architectures," in *SC '04: Proceedings of the 2004 ACM/IEEE conference on Supercomputing*. Washington, DC, USA: IEEE Computer Society, 2004, p. 21.

[2] G. Amati, S. Succi, and R. Piva, "Massively parallel lattice-boltzmann simulation of turbulent channel flow," *International Journal of Modern Physics C*, vol. 8, pp. 869–877, 1997.

[3] X. Wu, V. Taylor, C. Lively, and S. Sharkawi, "Performance analysis and optimization of parallel scientific applications on cmp cluster systems," *Parallel Processing Workshops, International Conference on*, vol. 0, pp. 188–195, 2008.

[4] K. Sano, O. Pell, W. Luk, and S. Yamamoto, "Fpga-based streaming computation for lattice boltzmann method," *Field-Programmable Technology, 2007. ICFPT 2007. International Conference on*, pp. 233–236, Dec. 2007.

[5] Z. Fan, F. Qiu, A. Kaufman, and S. Yoakum-Stover, "Gpu cluster for high performance computing," in *SC '04: Proceedings of the 2004 ACM/IEEE conference on Supercomputing*. Washington, DC, USA: IEEE Computer Society, 2004, p. 47.

[6] J. Tölke and M. Krafczyk, "Teraflop computing on a desktop pc with gpus for 3d cfd," *International Journal of Computational Fluid Dynamics*, vol. 22, pp. 443–456, Aug. 2008.

[7] J. Habich, "Performance evaluation of numeric compute kernels on nvidia gpus," Master's thesis, University of Erlangen-Nürnberg, June 2008.

[8] NVIDIA, "Geforce gtx 280 specification," 2008, [Online]. Available: http://www.nvidia.com/object/product_geforce_gtx_280_us.html

[9] S. Succi, R. Benzi, and F. Higuera, "The lattice Boltzmann equation: A new tool for computational fluid-dynamics," *Physica D: Nonlinear Phenomena*, vol. 47, no. 1-2, pp. 219–230, 1991.

[10] S. P. Dawson, S. Chen, and G. D. Doolen, "Lattice Boltzmann computations for reaction-diffusion equations," *The Journal of Chemical Physics*, vol. 98, no. 2, pp. 1514–1523, 1993.

[11] X. Shan and H. Chen, "Lattice boltzmann model for simulating flows with multiple phases and components," *Phys. Rev. E*, vol. 47, no. 3, pp. 1815–1819, Mar 1993.

[12] N. Thurey and U. Rude, "Free surface lattice-boltzmann fluid simulations with and without level sets," in *Vision, Modeling, and Visualization 2004*. IOS Press, 2004, pp. 199–208.

[13] D. Raabe, "Overview of the lattice boltzmann method for nano- and microscale fluid dynamics in materials science and engineering," *Modelling and Simulation in Materials Science and Engineering*, vol. 12, pp. R13–R46(1), November 2004. Available: http://www.ingentaconnect.com/content/iop/msmse/2004/00000012/00000006/art00r01

[14] D. Kandhai, D. J.-E. Vidal, A. G. Hoekstra, H. Hoefsloot, P. Iedema, and P. M. A. Sloot, "Lattice-Boltzmann and Finite Element Simulations of Fluid Flow in a SMRX Static Mixer Reactor," *International Journal for Numerical Methods in Fluids*, vol. 31, pp. 1019–1033, Nov. 1999.

[15] S. D. C. Walsh, M. O. Saar, P. Bailey, and D. J. Lilja, "Accelerating geo-science and engineering system simulations on graphics hardware," *In Review*, 2008.

[16] S. D. C. Walsh and M. O. Saar, "Macroscale lattice-boltzmann models for solute and heat transport in heterogeneous porous media." *In Preparation*, 2009.

[17] F. Qui, Z. Fan, Y. Zhao, H. Lorenz, J. Zhou, and A. Kaufman, "Gpu-based visual simulation of dispersion in urban environments," 2006.

[18] M. Krafczyk and J. Tölke, "Towards real-time prediction of tsunami impact effects on nearshore infrastructure," 2007.

[19] Y. Zhao, L. Wang, F. Qiu, A. Kaufman, and K. Mueller, "Melting and flowing in multiphase environment," *Computers and Graphics*, vol. 30, no. 4, pp. 519 – 528, 2006. Available: http://www.sciencedirect.com/science/article/B6TYG-4JXY3T2-2/2/188091218f2d278899e2b42d6ab3f11c

[20] W. J. Bosl, J. Dvorkin, and A. Nur, "A Study of Porosity and Permeability Using a Lattice Boltzmann Simulation," *Geophysical Research Letters*, vol. 25, pp. 1475–1478, 1998.

[21] S. Williams, J. Carter, L. Oliker, J. Shalf, and K. Yelick, "Lattice boltzmann simulation optimization on leading multicore platforms," *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pp. 1–14, April 2008.

[22] *CUDA Programming Guide*, 1st ed., NVIDIA, 2008.

[23] S. Succi, *The Lattice Boltzmann Equation for Fluid Dynamics and Beyond*. Oxford University Press, 2001.

[24] T. Pohl, M. Kowarschik, J. Wilke, K. Igleberger, and U. Rude, "Optimization and profiling of the cache performance of parallel lattice boltzmann codes," *Parallel Processing Letters*, vol. 13, pp. 549–560, 2003.

[25] G. Wellein, T. Zeiser, G. Hager, and S. Donath, "On the single processor performance of simple lattice boltzmann kernels," *Computers and Fluids*, vol. 35, no. 8-9, pp. 910 – 919, 2006, proceedings of the First International Conference for Mesoscopic Methods in Engineering and Science. Available: http://www.sciencedirect.com/science/article/B6V26-4HVDYJJ-4/2/21fc67683bc889045d128a80434bc31d

[26] W. Li, Z. Fan, X. Wei, and A. Kaufman, *GPU Gems 2*. Addison-Wesley, 2005, ch. Flow Simulation with Complex Boundaries, pp. 747–764.

[27] L. Nyland, M. Harris, and J. Prins, "Fast n-body simulation with cuda," in *GPU Gems 3*, H. Nguyen, Ed. Addison Wesley Professional, August 2007, ch. 31.

[28] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. E. Lefohn, and T. J. Purcell, "A survey of general-purpose computation on graphics hardware," *Computer Graphics Forum*, vol. 26, no. 1, pp. 80–113, 2007.

[29] Rapidmind, "Rapidmind datasheet," 2007, [Online]. Available: http://www.rapidmind.net/pdfs/RapidmindDatasheet.pdf

[30] I. Buck, T. Foley, D. Horn, J. Sugerman, and P. Hanrahan, "Brook for gpus," 2003, [Online]. Available: http://graphics.stanford.edu/projects/brookgpu/AF-Brook.ppt