An Optimizing Code Generator for a Class of Lattice-Boltzmann Computations

IRSHAD PANANILATH, ARAVIND ACHARYA, VINAY VASISTA,
and UDAY BONDHUGULA, Indian Institute of Science

The Lattice-Boltzmann method (LBM), a promising new particle-based simulation technique for complex and multiscale fluid flows, has seen tremendous adoption in recent years in computational fluid dynamics. Even with a state-of-the-art LBM solver such as Palabos, a user has to still manually write the program using library-supplied primitives. We propose an automated code generator for a class of LBM computations with the objective to achieve high performance on modern architectures. Few studies have looked at time tiling for LBM codes. We exploit a key similarity between stencils and LBM to enable polyhedral optimizations and in turn time tiling for LBM. We also characterize the performance of LBM with the Roofline performance model. Experimental results for standard LBM simulations like Lid Driven Cavity, Flow Past Cylinder, and Poiseuille Flow show that our scheme consistently outperforms Palabos—on average by up to $3\times$ while running on 16 cores of an Intel Xeon (Sandybridge). We also obtain an improvement of $2.47\times$ on the SPEC LBM benchmark.

Categories and Subject Descriptors: D.3.4 [Processors]: Compilers, Optimization, Code generation

General Terms: Design, Performance

Additional Key Words and Phrases: Lattice-Boltzmann method, time tiling, polyhedral framework, performance modeling

ACM Reference Format:

Irshad Pananilath, Aravind Acharya, Vinay Vasista, and Uday Bondhugula. 2015. An optimizing code generator for a class of Lattice-Boltzmann computations. *ACM Trans. Architect. Code Optim.* 12, 2, Article 14 (May 2015), 23 pages.

DOI: <http://dx.doi.org/10.1145/2739047>

1. INTRODUCTION

Lattice-based simulations are widely used in computational physics for solving partial differential equations. An important one is the Lattice-Boltzmann method (LBM) for hydrodynamics [Succi 2001]. LBM, an alternative to conventional computational fluid dynamics, is routinely used for high-performance computing of complex flows and fluid–structure interaction problems. LBM refers to an entire domain in itself as opposed to a single application or a set of applications.

There have been several manual efforts at optimizing LBM computations on multi-*many-core* processors. These works reveal the “Ninja gap” [Nadathur et al. 2012] in achieving the best performance for LBM on modern processors. The large programmer effort involved in developing optimized codes on modern multi-/*many-core* processors

New article, not an extension of a conference paper.

Authors’ addresses: I. Pananilath, A. Acharya, V. Vasista, and U. Bondhugula, Department of Computer Science and Automation, Indian Institute of Science, Bangalore, Karnataka 560012, India; emails: {irshad, aravind.acharya, vinay.vasista, uday}@csa.iisc.ernet.in.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

2015 Copyright is held by the owner/author(s). Publication rights licensed to ACM.

ACM 1544-3566/2015/05-ART14 \$15.00

DOI: <http://dx.doi.org/10.1145/2739047>

has spurred research in domain-specific languages and compilers that can bridge this Ninja gap. Domain-Specific Languages (DSLs) expose high-level abstractions that encapsulate domain knowledge and are easy to use for the application developer. The underlying compilation infrastructure deals with the task of generating optimized code for the target machine. The objectives of high productivity, high performance, and portability are thus simultaneously achieved.

Optimizing code generators that deliver good performance with little programmer input is currently not available for LBM computations. Due to the large memory footprint and low data reuse within a single time step of an LBM simulation, blocking or tiling along the time dimension is a key optimization in achieving high compute efficiency on current and future processors [Nguyen et al. 2010]. Scientists are typically interested in weak scaling for LBM simulations; that is, on a single node with shared memory, a dataset of size comparable to its total memory is used. Hence, the data set sizes well exceed the combined last-level cache capacities of all chips of an SMP system. In the absence of any automated machinery, the programmer has to incur the burden of implementing and tuning time tiling optimization in LBM on different platforms.

Although a number of compiler optimization works, including those based on the polyhedral framework, have studied optimizing stencil computations [Song and Li 1999; Wonnacott 2002; Krishnamoorthy et al. 2007; Bondhugula et al. 2008; Bandishti et al. 2012; Henretty et al. 2013; Grosser et al. 2014], all of these mainly used small representative stencil kernels to demonstrate their utility. LBM computations exhibit near-neighbor interactions conceptually similar to those representative stencils and potentially very different arithmetic intensities and underlying data structures. On the other hand, recent works that did study optimizing LBM have been manual efforts [Nguyen et al. 2010; Pohl et al. 2003]. Library-based approaches and LBM solvers like Palabos [Palabos 2009] still require users to manually write LBM simulations using library-supplied primitives. Though this improves productivity, obtaining high performance is elusive as our evaluation later demonstrates.

Our optimization approach exploits the similarity between stencils and LBM to model LBM in a way suitable for reasoning about time tiling and other polyhedral compiler optimizations. Our key contributions are as follows:

- Proposing an abstraction and representation for LBM that lends itself to reasoning and application of cache blocking along its time dimension
- Building an end-to-end tool that includes a domain-specific front end and an optimization chain
- Providing a detailed performance evaluation and modeling to understand bottlenecks

The class of LBM simulations handled by our tool are those that do not involve a moving obstacle in the data grid, although the techniques and optimizations we present are conceptually applicable to all LBM simulations. We demonstrate the effectiveness of our approach on the SPEC LBM benchmark and LBM simulations like Lid Driven Cavity, Flow Past Cylinder, Poiseuille Flow, on an Intel Xeon SMP based on the Sandy-bridge microarchitecture. We show that our code generator consistently outperforms Palabos [2009], a library-based DSL, with a mean speedup of $3\times$. We also report an improvement of $2.5\times$ over the SPEC LBM benchmark. To the best of our knowledge, this is the first approach to automatically perform transformations of the complexity of time tiling for LBM computations.

The roofline model [Ofenbeck et al. 2014; Williams et al. 2009] relates performance of kernels to the architecture's characteristics and can often provide insights into bottlenecks and tuning. In particular, one can determine if an execution is memory bandwidth bound, computation bound, or neither. We present the roofline analysis for a 2D (mrt-d2q9) and a 3D (ldc-d3q27) LBM application.

The rest of this article is organized as follows. Section 2 provides background on LBM along with a discussion of related work. Section 3 describes the representation, optimization process, and design of our code generator. Section 4 presents experimental results and performance modeling. Conclusions are presented in Section 5.

2. BACKGROUND AND PAST APPROACHES

In this section, we provide background on the Lattice-Boltzmann method. For the purposes of optimization, only the nature of dependencies and computation need to be understood by a reader.

2.1. Lattice-Boltzmann Methods

LBM has its origins in the Lattice Gas Automata (LGA) theory and is being increasingly used for simulation of complex fluid flows [Chen and Doolen 1998] in computational fluid dynamics. The crux of LBM is the construction of simple kinetic models that capture the essential physics of microscopic processes such that the averaged macroscopic properties follow the corresponding laws at the macroscopic scale. The Lattice-Boltzmann equation is a discretized form of the Boltzmann equation with the quantities time, space, and momentum being discretized.

In LBM, fluid flows are modeled as hypothetical particles moving in a discretized lattice domain (space) with different lattice velocities (discretized momentum) over different time steps (discretized time). Equation (1) shows the discretized form of the LB equation, where f_i is the particle distribution function (PDF) along the i th direction, x is the lattice point at which the PDFs are computed, c_i is the velocity vector along the i th direction, Δt is the time increment, Ω_i is the collision operator, and n is the number of PDFs based on the lattice used:

$$f_i(\mathbf{x} + \mathbf{c}_i \Delta t, t + \Delta t) = f_i(\mathbf{x}, t) + \Omega_i(f_i(\mathbf{x}, t)), \quad i = 0, \dots, n. \quad (1)$$

This equation is usually solved in two steps, the *collision* step (Equation (2)) and the *advection* step (Equation (3)):

$$f_i^*(\mathbf{x}, t + \Delta t) = f_i(\mathbf{x}, t) + \Omega_i(f_i(\mathbf{x}, t)) \quad (2)$$

$$f_i(\mathbf{x} + \mathbf{c}_i \Delta t, t + \Delta t) = f_i^*(\mathbf{x}, t + \Delta t). \quad (3)$$

2.1.1. Collision Models. Several collision operators have been proposed in the literature to implement the collision step in Equation (1). The simplest formulation is the Bhatnagar–Gross–Krook (BGK) [Bhatnagar et al. 1954] collision operator, also referred to as Single Relaxation Time (SRT). It utilizes the same relaxation parameter, τ , for all the PDFs. f_i^{eq} is an approximation of the Maxwellian equilibrium distribution function. This is shown in Equation (4):

$$\Omega_i = -\frac{1}{\tau} (f_i - f_i^{eq}). \quad (4)$$

The Multiple Relaxation Time (MRT) collision operator given in Equation (5), proposed by d’Humières [2002], uses a collision matrix to relax the PDFs toward their local equilibria. Substituting $\mathbf{S}$ with $\frac{1}{\tau} \mathbf{I}$, where $\mathbf{I}$ is an identity matrix, recovers the BGK model from MRT:

$$\vec{\Omega} = -\mathbf{S}(\vec{f} - \vec{f}^{eq}). \quad (5)$$

2.1.2. Common Lattice Arrangements. Different lattice arrangements can be utilized for LBM and are represented as $DmQn$, where m is the space dimensionality of the lattice and n is the number of PDFs involved. Typical examples are $D2Q9$, $D3Q19$, and $D3Q27$. Figure 1 shows the arrangement of two of these.

Fig. 1. D2Q9 (left) and D3Q19 (right) lattice arrangements.

Fig. 2. LBM: Push model for D2Q9 lattice [Wittmann et al. 2013].

2.1.3. Boundary Conditions. Boundary conditions are an integral part of any LBM simulation specification. They are used to model solid boundaries, moving walls, flow interactions with obstacles, and no-slip conditions. They essentially specify a way to determine the unknown particle distribution functions in the previously listed situations. Midgrid Bounce Back [Succi 2001] and Zou-He [Zou and He 1996] are the two commonly used boundary conditions. One of the attractive features of LBM over other simulation methods is that the use of a simple bounce-back boundary condition for complex obstacle scenarios still provides second-order accuracy.

2.1.4. Implementation Strategies. A majority of the implementations directly translate the two-stage nature of the LBM formulation into code. This results in a *collision* phase and an *advection* phase for each time step of the computation kernel. The *collision* phase computes new values for the PDFs and the *advection* phase propagates this information to the neighboring nodes for use in the next time step.

Based on the interleaving of these two phases, LBM implementations can follow either a *push* model or a *pull* model. In the *push* model (Figure 2), the collision step happens first followed by the *advection* step, resulting in the postcollision values to be propagated to the neighboring nodes.

In contrast, the *pull* model (Figure 3) implements the *advection* step first to pull in the postcollision values from the previous time step and then performs the collision on these to produce the new PDFs.

Both the *pull* and *push* models can be implemented using a single data grid or two data grids to store the PDFs. This choice is normally driven by two factors: the flexibility available for parallelization and the memory footprint. Single-grid approaches like *Compressed Grid* by Pohl et al. [2003] and *Swap algorithm* by Mattila et al. [2007]

Fig. 3. LBM: Pull model for D2Q9 lattice [Wittmann et al. 2013].

reduce the memory footprint of the program but impose an order on the processing of the nodes, thereby restricting the parallelism that can be exploited.

The *Compressed Grid* technique, for example, puts the two lattices previously required, shifted by one cell in each direction, on top of each other. Thus, only one lattice has to be stored, plus a layer of cells in each dimension. Although the storage required is reduced by a factor close to two, the nodes have to be processed in a specific order to remove the data dependency between the nodes. In even time steps, the iteration proceeds from the top right corner of the grid to the bottom left. In odd time steps, it happens from bottom left to top right. In the *Swap algorithm*, the space required is exactly half of the naive scheme (two grids), but the iteration order is even more strict. A comparison of different strategies is presented in Wittmann et al. [2013]. They note that although advanced propagation strategies like *A-A Pattern* and *Esoteric Twist* provide significantly higher performance, the programming effort required is also equally large.

Prior works have also explored the effect of different data layouts on performance. In the *SoA* (structure of arrays) layout, a particular PDF for all the nodes in the domain is stored consecutively in an array and one such array for each of the PDFs is found to suit the *advection* phase. The *AoS* (array of structures) layout, where all the PDFs of a node are stored together in memory and an array of such structures holds the information for the whole domain, is a collision-optimized layout. Hybrid layouts (*SoAoS* [Shet et al. 2013]) have also been looked at in the past to combine the benefits of the two.

2.2. Related Work on Optimizing LBM

Several past works [Massaioli and Amati 2002; Williams et al. 2009, 2011] have focused on optimizing LBM computations for both shared memory and distributed memory architectures. Williams et al. [2009] presents an autotuning approach to optimize performance for LBM with a detailed performance modeling; it does not develop an automatic approach to apply optimizations and for code generation.

Time blocking (tiling) [Wonnacott 2000, 2002; Song and Li 1999] is an optimization that can greatly improve performance of stencil computations. The earliest work to perform time blocking for LBM was Wilke et al. [2003]. They performed both 1D blocking and 2D blocking on the LBM code with a *Compressed Grid*. One-dimensional blocking was not found to provide any performance benefit. With 2D blocking, using a blocking factor of 16x16, they were able to improve performance. However, it imposed an order on the processing of tiles, diagonally downward and toward the left. They concluded that manually generating cache-optimized codes for LBM was tedious and error prone.

Nitsure et al. [2006] proposed a cache-oblivious algorithm for LBM, *COLBA*. It uses virtual domain decomposition to generate tasks, and parallelization is done using Intel's Workqueuing Model for OpenMP. The algorithm is recursive and utilizes a time-blocking factor (TB) when generating tasks. They report that they had found it difficult to work with the usual OpenMP *parallel* and *parallel for* pragmas due to the recursive

nature of the algorithm and also resulted in “orphan” directives that never spawned threads. The execution is pipelined and suffers from pipeline fill and drain phases.

The 3.5D blocking technique by Nguyen et al. [2010] is a recent effort toward time tiling LBM computations. It utilizes 2.5D blocking in the space dimensions and additional blocking in the time dimension to achieve good performance. The parallel 3.5D *blocking* algorithm involves some amount of overlapped computation and a sequential section in the prologue of each XY block processed by a thread.

In the past few years, there have been a particularly large number of time tiling techniques [Krishnamoorthy et al. 2007; Bondhugula et al. 2008; Bandishti et al. 2012; Strzodka et al. 2011; Grosser et al. 2014] demonstrated on small representative stencils. However, their utility for LBM has not yet been studied in spite of more than two decades of research into stencil tiling. A comprehensive framework that can address the shortcomings of previous works and automatically generate code for LBM that achieves high performance on different architectures is thus highly desirable.

3. LBM REPRESENTATION FOR TIME TILING

In this section, we present *msLBM*, a system to automatically generate LBM implementations for shared-memory architectures in a turnkey manner.

3.1. Design Decisions

Unlike general implementations, which employ two passes over the data grid per time step, once for performing collision and a second time for advection, we chose to work with a fused version of the LBM kernel. The data required for the collision at a domain node is retrieved on demand from its neighbors, and new PDFs at the current node are calculated and updated. This results in better cache utilization and necessitates only a single pass over the grid. Since we no longer have an explicit *advection* phase, a *collision*-optimized AoS data layout can be used.

Since the entire data grid has to be accessed once per time step, the bandwidth requirement of LBM kernels is very high and they tend to be bottlenecked by the total memory bandwidth available in the system. This situation worsens, as discussed in Nguyen et al. [2010], with the use of higher-order lattices (D3Q19, D3Q27, etc.) where 19 or 27 separate memory locations have to be accessed for each node in the grid. Time tiling is thus a very important optimization as it improves temporal reuse and prevents computation from becoming memory bandwidth bound by blocking along the time dimension.

Although using a single grid to store data for the nodes in the domain reduces the memory footprint of the implementation, it severely restricts time tiling as it induces memory-based dependencies. Figure 4 illustrates this with a simple 1D LBM case. The collision phase for node k in time step t reads the required PDFs from the neighbors in time step $t - 1$, computes new PDFs, and updates the values in place. Processing the next node, $k + 1$, for time step t requires a value of node k from time step $t - 1$, and this would be overwritten before its last use. Techniques like *Compressed Grid* work around this problem by specifying an order and direction for the processing and updates. However, this creates complications when dealing with advanced boundary conditions [Körner et al. 2006] and hinders further optimizations.

Our framework therefore adopts a two-grid, fused, *pull* scheme with an AoS data layout for the LBM implementations. With this configuration of LBM kernels, it is easy to notice the similarity to normal stencils as illustrated in Figure 5. The only difference is that stencils store a single value at each node in the domain, whereas LBM stores multiple PDFs at each node depending on the lattice used. Notice that if all three points in the rectangular box in Figure 4 were collapsed to one point, the dependence structure becomes identical to a 1D stencil. We exploit this inference to

Fig. 4. Illustrative 1D LBM with single grid.

enable polyhedral time tiling and subsequent optimizations on LBM implementations generated. By employing two grids, both the *push* and *pull* schemes become amenable to time tiling.

Per-tile storage buffers could be used to perform storage compaction and reduce the memory footprint of the LBM implementation. This is described further in Section 3.5.

Overall LBM Optimization System. Our LBM optimization framework, msLBM, is illustrated in Figure 6. It starts with a specification of the target LBM simulation in the form of a JSON [JSON 2001] configuration file. A serial code for the LBM implementation is composed from the JSON specification. We utilize the publicly available polyhedral source-to-source optimizer, Pluto [pluto 2008], for performing locality optimizations on the serial code. Further complementary optimizations (source to source) are implemented in Python with the generated serial code as the input. The post-processed code is further tuned, taking into consideration the target architecture. The JSON notation provides an intuitive and compact way of capturing the LBM simulation parameters. It also simplifies the parsing of the input configuration.

3.2. Polyhedral Optimizations

The polyhedral model for compiler optimization [Bastoul 2008] is well suited to model transformations for sequences of loop nests with regular access patterns. Stencil computations often fall into this class. It is often very difficult or infeasible to develop code for stencils that is tiled in a way that is optimized for both locality and parallelism, especially on 2D or 3D data grids. The polyhedral model provides a linear algebraic view of the iteration space, dependencies, and transformations. Code generation is again

Fig. 5. Illustrative example: Pull scheme on a 1D LBM.

Fig. 6. Overview of msLBM.

automatic, relying on machinery primarily from linear algebra and integer linear programming.

3.2.1. Polyhedral Representation for LBM in a DSL. The polyhedral framework normally requires computation to have regular (affine) data accesses and affine bounds; that is, the access functions, loop bounds, and any conditionals need to be affine functions of outer loop variables and program parameters (typically problem sizes). However, in the context of our DSL code generation and optimization system, we need not care about many of these restrictions—the dependence structure is already well understood and the entire loop body in its atomicity, irrespective of whether it has affine accesses or accesses that appear irregular at compile time, can be treated as a single monolithic polyhedral statement. Its execution domain or index set is obtained from the loops surrounding it. Each loop is abstracted as a dimension in the polyhedron representing the index set and each integer point in the index set represents an execution instance of the loop body (in its entirety). The set of loop body instances, which correspond to the collision computations, are thus transformed atomically. As a result of the view taken in Section 3.1, the code becomes amenable to time tiling, and any of the existing techniques can be used.

3.2.2. A DSL-Based Approach Versus Extending Pluto. The choice of pursuing a DSL-based approach as opposed to one of extending Pluto to directly deal with an input C implementation was primarily driven by two factors. As mentioned in Section 3.1, it was necessary to abstract away from the user the choice of selecting the right implementation strategy, for example, that of choosing between a two-grid fused pull scheme and one of a separate advection and collision stage with a single grid. The latter as we described prevents a compiler from inferring time tiling. Also, the ability to model the entire LBM kernel atomically as a single polyhedral statement (as described earlier in this subsection) is a key abstraction. An LBM implementation in C or C++ is often not in such a suitable form. This in turn would make a polyhedral front end extract a large number of statements (several tens), including potentially a large number of temporary variables and statements computing temporary results. Affine dependencies typical of representative stencils may be deduced only after additional analysis, and this may not necessarily be done by the time a polyhedral representation has been extracted. Dealing with all of these aspects only increases the amount of time spent in various phases of polyhedral optimization. A DSL approach provides the right abstractions to the polyhedral optimization phase by keeping the representation simple while utilizing all its capabilities—it exposes only the necessary affine dependencies on the computation grid and treats all local action at a grid point as a black-box polyhedral statement.

3.3. Time Tiling

Blocking along the time dimension leads to extraction of temporal reuse along the time loop; that is, it increases the number of operations that can be performed for the same amount of data that is brought on to the chip. Since the number of operations per byte of data moved in is increased, a higher compute rate can be sustained for the same amount of memory bandwidth utilized. This is even more useful when simultaneously running on multiple cores of a single chip or multiple chips. Moreover, the benefits of the compute power of a single chip via SIMDization are also seen when the data comes from the cache. A simple calculation shows that the amount of bandwidth needed to keep the SIMD units is prohibitively high when there is no cache reuse. The space loops of LBM, that is, loops that iterate over the data grid, have limited (a constant factor) reuse. In particular, consider D2Q9, where there may be about 70 to 80 operations performed for every nine elements of the grid. This ratio of operations to off-chip load/stores for the straightforward original execution order is not sufficient to keep all compute units busy. This will also be supported by experimental results presented later.

There are several time tiling techniques for stencils in the literature. It is well known that straightforward rectangular tiling of stencils is not valid [Wolfe 1995; Xue 2000]. It is necessary to skew the space loops typically by a factor of one with respect to the time loop. Hence, the most natural tiling technique has been to use hyper-parallellograms [Wolfe 1995; Xue 2000; Song and Li 1999; Wonnacott 2002]. This is equivalent to skewing the space loops with respect to the time loop and performing rectangular tiling. Parallellogram tiling suffers from pipelined startup and drain besides limiting the amount of parallelism on the wavefront depending on the number of time steps and time-tile sizes. Another approach is to use overlapping trapezoids [Krishnamoorthy et al. 2007]. Such an overlap not only leads to redundant computation but also makes code generation very hard to automate for shared memory—there is no automatic generator for it yet. Another approach is to use a combination of trapezoids and inverted trapezoids as is done by Pochoir [Tang et al. 2011] with a recursive cache oblivious technique. The recently proposed diamond tiling technique [Bandishti et al. 2012] allows concurrent start in the tile space while not adding any additional complexity to code generation or any redundant computation. Hexagonal tiling with hybrid shapes [Grosser et al. 2013,

Fig. 7. Tile shape for a 2D LBM (D2Q9 for example); arrows represent hyperplanes.

2014] and jagged polygon shapes [Shrestha et al. 2015] have recently been proposed and evaluated for GPUs and Xeon Phi, respectively. Among benefits these techniques provide is their suitability for exploiting parallelism at multiple levels; their benefits over diamond tiling for general-purpose multicores are yet to be evaluated.

3.4. Diamond Tiling

Diamond tiling [Bandishti et al. 2012] is a time tiling technique for stencils that allows concurrent start of tiles along one or more iteration space boundaries. It does not have a pipelined start and drain phase and thus provides asymptotically more parallelism than parallelogram tiling [Wonnacott and Strout 2013]. The number of tiles on the wavefront is limited only by data grid extents and not by the number of time iterations. Figure 7 shows a 3D tile (for a 2D data grid) that would allow full concurrent start along the 2D plane composed of the i and j dimensions.

A hyperplane is represented by a vector normal to it. Three linearly independent hyperplanes in this case define a tile shape, which is hyperparallelepiped. The entire iteration space is uniformly sliced into such hyperparallelepipeds. The diamond tiling hyperplanes have the property that the sum of the hyperplanes is a vector in the same direction as the time dimension—this is also a direction along which there exists pointwise concurrent start in the iteration space. The ij plane in Figure 7, for example, allows point-wise concurrent start. In practice, concurrent start along only a subset of space dimensions is sufficient to keep all cores busy—we call this partial concurrent start. It results in simpler code while providing the desired benefits [Bandishti et al. 2012].

With diamond tiling having already been demonstrated as the simpler and better-performing one on modern multicores [Bandishti et al. 2012], we choose diamond tiling as our time tiling scheme for LBM.

3.5. Complementary Optimizations

We now describe each of the optimizations utilized in the msLBM framework, complementary to time tiling, to refine the code and extract maximum performance from the target architecture. The set of optimizations applied by the framework depends on the nature of the LBM simulation being processed and the target architecture.

<pre> z4 = o5 + n2 + e2 + l4 + o0 + z2 + i7 + f2 + z3; i6 = (i7 + e2 + o0) - (f2 + l4 + z2); i6 = i6 / z4; a1 = (z2 + o5 + o0) - (f2 + n2 + i7); a1 = a1 / z4; q4 = 1.5 * (i6 * i6 + a1 * a1); *m2 = (1.0 - i9) * z3 + (i9 * (j9 * z4 * (1.0 - q4))); *g9 = (1.0 - i9) * o5 + (i9 * (e0 * z4 * (1.0 + 3.0 * a1 + 4.5 * a1 * a1 - q4))); *a3 = (1.0 - i9) * n2 + (i9 * (e0 * z4 * (1.0 - 3.0 * a1 + 4.5 * a1 * a1 - q4))); *r6 = (1.0 - i9) * e2 + (i9 * (e0 * z4 * (1.0 + 3.0 * i6 + 4.5 * i6 * i6 - q4))); *d0 = (1.0 - i9) * l4 + (i9 * (e0 * z4 * (1.0 - 3.0 * i6 + 4.5 * i6 * i6 - q4))); *i3 = (1.0 - i9) * o0 + (i9 * (r3 * z4 * (1.0 + 3.0 * (i6 + a1) + 4.5 * (i6 + a1) * (i6 + a1) - q4))); *f9 = (1.0 - i9) * z2 + (i9 * (r3 * z4 * (1.0 + 3.0 * (-i6 + a1) + 4.5 * (-i6 + a1) * (-i6 + a1) - q4))); *g8 = (1.0 - i9) * i7 + (i9 * (r3 * z4 * (1.0 + 3.0 * (i6 - a1) + 4.5 * (i6 - a1) * (i6 - a1) - q4))); *g2 = (1.0 - i9) * f2 + (i9 * (r3 * z4 * (1.0 + 3.0 * (-i6 - a1) + 4.5 * (-i6 - a1) * (-i6 - a1) - q4))); </pre>	<pre> p9 = z3 + l7 + l6; f1 = j4 + b4 + z4; y7 = j4 + x8 + z3; w0 = z4 + k0 + l6; d4 = p9 + (x8 + p8 + k0) + f1; h5 = 1.0 / d4; w4 = h5 * (p9 - f1); h1 = h5 * (y7 - w0); t1 = w4 * w4; t0 = h1 * h1; z2 = 1.5 * (t1 + t0); q3 = 1.0 - z2; s2 = o9 * w1 * d4; m7 = o9 * w2 * d4; z0 = o9 * w3 * d4; w8 = 3.0 * h1; c9 = 4.5 * t0; k3 = 3.0 * w4; r6 = 4.5 * t1; s1 = w4 + h1; g3 = w4 - h1; s8 = 3.0 * s1; t9 = 3.0 * g3; p5 = 4.5 * s1 * s1; l9 = 4.5 * g3 * g3; c7 = n8 * p8 + s2 * (q3); w9 = n8 * x8 + m7 * (q3 + w8 + c9); t5 = n8 * k0 + m7 * (q3 - w8 + c9); x3 = n8 * l7 + m7 * (q3 + k3 + r6); c3 = n8 * b4 + m7 * (q3 - k3 + r6); k1 = n8 * z3 + z0 * (q3 + s8 + p5); j3 = n8 * j4 + z0 * (q3 - t9 + l9); v5 = n8 * l6 + z0 * (q3 + t9 + l9); l5 = n8 * z4 + z0 * (q3 - s8 + p5); </pre>
--	---

(a) Original code with 145 ops
(b) SymPy simplified code with 83 ops

Fig. 8. CSE with SymPy for a D2Q9 kernel.

Simplification of the LBM kernel. We use SymPy [Joyner et al. 2012], a Python library for symbolic mathematics, to simplify the LBM kernel. SymPy enables us to do common subexpression elimination (CSE) across multiple statements, thereby reducing the number of floating-point operations (FP-ops) required for the LBM kernel. In some cases, we observed that the compiler might not be able to achieve this even with a relaxed floating-point model. Figure 8 shows a simple *D2Q9* kernel (variables renamed for brevity) where SymPy helps reduce the number of FP-ops by 42% from 145 to 83. We couldn't get the compiler to do this automatically. With higher-dimensional kernels, this contributes to efficiency of msLBM-generated code.

<pre> for(...;...;...) { if(v5<v6) { v1 += v3; } } </pre> <p>(a) Nest with conditional</p>	<pre> for(...;...;...) { vcmppi_lt k7 , v5, v6 vaddpi v1{k7}, v1, v3 } </pre> <p>(b) Predicated SIMD code</p>
---	---

Fig. 9. Predicated SIMD example.

Eliminating modulus from the inner kernel. Each of the two storage grids is to be chosen as a target for write operations alternately while moving along the time dimension. This results in modulo (%) operations on the array index corresponding to the time dimension. This in turn severely affects performance of the diamond tiled LBM kernel and cannot be eliminated altogether. To reduce the effect of modulo operations, we hoist out this check to the outermost possible loop and then version the loop body based on its output, that is, the result of the $(var \% 2)$ computation.

Boundary tile separation. LBM requires preferential treatment for the boundary nodes of the domain. This results in either *IF* conditions inside the computation kernel to identify and handle boundary nodes separately or different versions of the LBM kernel for the boundary nodes and in-domain nodes. Such checks can result in overheads for the processing of the in-domain nodes and hence degrade performance. We identify and separate out boundary and inner tiles in the diamond tiled code so that a clean and uniform LBM kernel is available for processing the inner tiles. This is done in a way similar to existing work on full and partial tile separation from tiled iteration spaces [Renganarayanan et al. 2007].

Data alignment. Storage allocated, either statically as global/local variables or dynamically, can be aligned to specific byte boundaries. Loads/stores on aligned data are more efficient than unaligned ones. Vectorization efficiency of code heavily depends on data alignment. In the absence of aligned data, gather/scatter operations have to be generated to bring in data. We use the compiler directive `__attribute__((aligned(X)))` for static allocations and the allocator `_mm_aligned(...)` for dynamic allocations to align data. Data movement is optimal when the starting address of the data is aligned on a 32-byte boundary on Intel Sandybridge.

Inlining and unrolling. For most of the transformations, we treat the LBM kernel as a black box. However, this might prohibit the compiler from performing certain optimizations. Inlining the LBM kernel eliminates overheads associated with function invocations. Unrolling by a suitable factor provides the compiler opportunities to efficiently schedule instructions to achieve a high sustained compute performance. Combined with boundary tile separation, the LBM kernel for the inner tiles can be both inlined and unrolled. We rely on the native compiler's (icc) automatic vectorization to perform the rest.

Predicated SIMDization. In the presence of obstacles in the domain, which can be represented using either a flag for each node or an analytic expression that is evaluated at each node, vectorizing the code becomes difficult. Predicated SIMD instructions, which allow per-lane conditional flow, can help vectorize such cases. A simple example is shown in Figure 9. We structure our code in a way that makes it easy for the compiler to generate predicated SIMD code.

Memory optimization. Recall that we chose to use a two-grid model since the resulting data dependencies allowed application of time tiling transformations. Domain scientists are often interested in weak scaling for LBM. Using a two-grid approach

Table I. Architecture Details

Intel Xeon E5-2680	
Clock	2.7GHz
Cores/socket	8
Total cores	16
L1, L2 cache/core	32KB, 512KB
L3 cache/socket	20MB
Main memory	64GB non-ECC DDR3-1600 RAM
Peak GFLOPs	172.8
Compiler	Intel C compiler (icc) 14.0.1
Compiler flags	-O3 -xHost -fno-alias -fno-finalias -ipo -restrict -fp-model precise -fast-transcendentals
Linux kernel	3.8.0-38

thus allows them to only run half the data set they could have, given an upper limit on the available main memory. It is, however, possible to reduce the memory footprint of the two-grid optimized code posttransformation to the single grid one on a per-tile basis, that is, for sequential execution of a single tile on a core. This is achieved with a modulo mapping, which emulates a rotating buffer. Consider a 1D LBM with N data points in all with time loop t and space loop i —this would consume $2 * N$ lattice elements of storage with two grids and N with a single grid. If a grid or a portion of the grid (a tile) is executed sequentially, note that the write at location (t, i) can reuse the location of iteration $(t - 1, i - 1)$ since the last read for data generated at $(t - 1, i - 1)$ has already happened at (t, i) . Hence, only a rotating buffer of size $N + 1$ lattice elements is needed, as opposed to $2 * N$. Analogously, for 2D LBMs, the storage can be reduced from $2 * N^2$ elements to $N^2 + N + 1$ elements, and for a 3D stencil, from $2 * N^3$ elements to $N^3 + N^2 + N + 1$. Hence, using a two-grid approach and then reducing the storage requirement is a cleaner way for automatic optimization purposes. Techniques for analyzing and achieving such mappings exist [Strout et al. 1998; Bhaskaracharya and Bondhugula 2014]. Although such a memory reduction may not necessarily improve performance over a tiled code already optimized for locality, it is crucial for weak scaling—in enabling a larger problem size given a system with a fixed amount of main memory.

Tile sizes. The determination of the right tile sizes is crucial for single-thread performance and scaling to a larger number of processing elements. In a domain-specific context where the computation and transformations applied on the computation are expected and completely understood, determining tile sizes is an easier problem. For the benchmarks evaluated, we empirically determined the tile sizes that provided the best results with msLBM. We found that using a tile size of 8 for each of the space dimensions and the time dimension yielded the best performance with our system. We do not present a model for selection of tile sizes here, but we believe that one based on certain rough thumb rules can be easily incorporated.

4. PERFORMANCE MODELING AND EVALUATION

In this section, we evaluate our code generator by presenting experimental results on a range of LBM simulations and benchmarks and by using a performance model.

4.1. Experimental Setup

We empirically evaluate performance on a dual-socket server with each socket seating an Intel Xeon E5 2680 processor. The processors are based on the Intel Sandybridge microarchitecture, and some details of the microarchitecture are presented in Table I.

Fig. 10. Setting for a 2D LDC simulation.

The Sandybridge microarchitecture includes the 256-bit Advanced Vector Extensions (AVX). All computations used herein employ double-precision floating-point arithmetic. Information on the compiler and flags used to compile all code is also shown in Table I.

Code generated by our framework is labeled msLBM. msLBM-novec is code we generate without the postprocessing transformations that we perform to enable vectorization. The difference between the two thus captures the efficiency and benefit of vectorization. seq refers to the straightforward sequential version, that is, serial code generated without application of any of our transformations or optimizations. icc-auto-par refers to the icc autoparallelized version, that is, seq compiled with the “-parallel” flag in addition.

Although weak scaling is important for LBM, since our experiments are on a single node that is a shared-memory multicore system, only strong scaling is meaningful in our context. Hence, all our results are with strong scaling; that is, the same problem size is used while executing it on an increasing number of cores.

4.2. Benchmarks

For evaluation purposes, we choose a set of benchmarks that represents a wide diversity of LBM computations in terms of dimensionality, arithmetic-to-memory operation ratios, participating neighbors, and boundary conditions. Each of the benchmarks used is described briefly as follows.

Lid Driven Cavity: Lid Driven Cavity (LDC) [Mohamad 2011] is a very common test problem among LBM simulations and is used to estimate the accuracy and efficiency of LBM implementation techniques. Flow in a cavity filled with an incompressible fluid is driven by a moving wall at the top as shown in Figure 10. A simple LBGK collision operator is used here. We consider both 2D and 3D versions of LDC and for multiple lattice types. This results in three cases labeled as ldc-d2q9, ldc-d3q19, and ldc-d3q27.

SPEC LBM: The LBM benchmark *470.lbm* from the SPEC CPU2006 benchmark suite can simulate either LDC or flow through a porous medium. The SPEC suite uses only the LDC case in its benchmarking process. *470.lbm* simulates LDC for a three-dimensional cavity using a *D3Q19* lattice, but with an ellipsoid obstacle inside the cavity. The obstacle is specified using an ASCII obstacle file with a ‘.’ representing a point in the domain and a ‘#’ denoting an obstacle point. This is one way of specifying

Fig. 11. 2D Poiseuille Flow setting.

Table II. Problem Sizes Used for Benchmarks

Benchmark	Grid size	Time steps
ldc-d2q9	$1,024 \times 1,024$	50,000
ldc-d3q19	$200 \times 200 \times 200$	1,000
ldc-d3q27	$200 \times 200 \times 200$	1,000
fpc-d2q9	$1,024 \times 256$	40,000
poiseuille	$1,024 \times 256$	40,000
mrt-d2q9	$1,024 \times 1,024$	20,000
spec-lbm	$100 \times 100 \times 130$	3,000

obstacles. The other way is to have an analytical equation describing the obstacle, but we adhere to the configuration used by SPEC here. This is referred to in our results as `spec-lbm`.

Poiseuille Flow: Poiseuille Flow, labeled as `poiseuille` in the graphs, simulates the flow of an incompressible fluid between two flat, parallel plates. Our benchmark simulates the 2D case. Zou-He [Zou and He 1996] pressure boundary conditions are used to drive the flow.

Flow Past Cylinder: Two-dimensional simulation of a circular cylindrical obstacle in an incompressible viscous channel flow, labeled as `fpc-d2q9` in our experiments, is used to study the vortex structures generated and their interaction in the direction of the flow. A *D2Q9* lattice is used to store the PDFs.

MRT – GLBM: As explained in Section 2, the LBGK model is a simple collision model that uses a single relaxation parameter to reach the local equilibrium. The MRT Lattice-Boltzmann method, also known as the Generalized Lattice-Boltzmann method, utilizes a relaxation matrix for the quantities involved. This results in a significantly large number of FP-ops for the LBM kernel. We use MRT LBM to model the 2D LDC simulation and evaluate how efficiently *msLBM* handles the increased compute density of the LBM kernel. This is referred to in our results as `mrt-d2q9`.

Problem sizes used for the benchmarks are listed in Table II. The grid sizes used are sufficiently large so that the total data size is larger than the system memory. The time steps chosen are such that the corresponding benchmarks reach a steady state and the results can be reliably reproduced.

4.3. Performance Metrics

LBM implementations are generally memory bandwidth bound rather than being limited by the available compute resources. In addition, the compute parts involve a large number of operations that could benefit from subexpression reuse and related optimizations. This makes it hard to use the number of floating-point operations per second

(FLOPS) as a yardstick of performance as it is often not indicative of the true performance and bottlenecks. Thus, performance is, like Wellein et al. [2006] and Nguyen et al. [2010], measured in *Mega Lattice Site Updates per Second* (MLUPS) and is calculated as follows:

$$MLUPS = \frac{N_x * N_y * N_z * N_{ts}}{10^6 * T}, \quad (6)$$

where N_x , N_y , and N_z are the domain sizes in the x, y, and z dimensions; N_{ts} is the number of time steps used; and T is the total time taken for the simulation. In case of 2D simulations, N_z is not included in the previous equation.

4.4. Analysis of Performance

Figure 12 presents results for all benchmarks reporting MLUPS on the y-axis. We observe that msLBM provides significantly better single-thread performance due to better locality optimization and allows good or better scaling than Palabos. For spec-lbm, poiseuille, and fpc-d2q9 (Figures 13, 12(e), 12(f)), we were unable to find a way to express them with Palabos; hence, no results with Palabos are presented in those cases. For msLBM, the speedups when running on multiple cores over running on one core are provided separately in Figure 14 to make the scaling efficiencies clearer.

As is evident from the graphs, the results we see for icc-auto-par on multiple cores are just its sequential performance. We observed that icc was unable to parallelize the core loop nest due to conservatively assumed dependencies, likely arising from imprecise alias information. We thus manually parallelized the code in a straightforward way using OpenMP, and this is reported in the graphs as (icc-omp-par). The outermost space loop was marked parallel. The lower single-thread performance for icc-omp-par and poorer scaling confirms earlier remarks that the original execution order was memory bandwidth bound. The difference between msLBM-novec and msLBM quantifies the improvement achieved with vectorization. We consistently observe a lower vectorization efficiency for the 3D cases than that in the 2D ones. Note that the trip counts for the loops in the 3D case are significantly lower, leading to a larger fraction of partial tiles. This results in a suboptimal scalar-code-to-vector-code ratio in the autovectorized program. Improving vectorization efficiency for the 3D cases is one of the subjects of our future work.

We observe good scaling with msLBM, with a close-to-ideal scaling for d2q9 configurations (Figure 14). The spec-lbm data grid, as mentioned earlier, involves an obstacle that is of size comparable to the data grid itself. Our code generator currently supports extraction of only a single degree of parallelism, and as a result we run out of useful work here. This is purely a limitation of the current state of implementation, and the scaling is expected to be better when multiple degrees of parallelism are exploited in such cases. The anomalies in performance and scaling for msLBM in some cases, when close to the maximum number of cores, is due to (1) the number of tiles not being perfectly divisible by the number of cores and (2) difficulty in reproducing performance for the maximum number of cores due to interference from certain internal OS kernel processes.

Figure 15 shows the performance of msLBM-generated code for different benchmarks in a single graph in a way that allows comparison of the performance achieved with different models. Using the MLUPS measure for the comparison of different LBM implementations having distinct lattice arrangements is not considered fair. To this end, *Million Elementary Updates Per Second* (MEUPS), as proposed in Shet et al. [2013], is obtained by multiplying the MLUPS value with $n - 1$, where n is the number of PDFs present in the lattice used. The y-axis in MEUPS thus makes the comparison meaningful across different configurations.

Fig. 12. Performance of various LBM simulations on Intel Xeon E5-2680 (Sandybridge).

The LBM simulations presented earlier do not involve periodic boundary conditions. Tiling in the presence of such conditions requires additional complementary techniques of folding [Osheim et al. 2008] or an Index Set splitting [Bondhugula et al. 2014]. Performance for these LBM simulations in the exclusive presence of periodic boundary

Fig. 13. Performance on LBM from SPEC2006FP (470.lbm).

Fig. 14. msLBM scaling: speedup over corresponding single-core execution for msLBM for each of the benchmarks.

conditions has been presented in Acharya and Bondhugula [2015] as part of a larger objective to demonstrate the utility of negative coefficients in affine transformations.

Nguyen et al. [2010]’s approach performs 3.5D blocking and other optimizations including vectorization manually. Their implementation is not publicly available for a

Fig. 15. Comparison of MEUPS across benchmarks with msLBM.

direct comparison. The performance reported therein was for SPEC LBM on a four-core Nehalem core i7 machine running at 3.2GHz. Given that all our results are on a much more recent microarchitecture with different memory bandwidth and compute peaks, an accurate comparison or normalization is not feasible.

4.5. Performance Modeling Using the Roofline Model

The Roofline model [Williams et al. 2009; Ofenbeck et al. 2014] relates performance of kernels to the architecture's characteristics and can often provide insights into bottlenecks and tuning. In particular, one can determine if an execution is memory bandwidth bound, computation bound, or neither.

Figure 16 presents the roofline plot for a 2D (mrt-d2q9) and a 3D (ldc-d3q27) LBM kernel. Note that all points plotted, except seq and those explicitly mentioned as being for eight cores, correspond to parallel execution on all 16 cores. The objective of time tiling is to increase the number of operations performed for an element of data brought into a faster level of the memory hierarchy. It would thus move a point in the plot to the right; however, this in turn allows a better utilization of the compute units on the CPU and a higher GFlops/s. Hence, we move to the right and upward simultaneously with time tiling. For mrt-d2q9, Palabos provides performance close to the ceiling possible for an operational intensity of 1.0. Our framework improves the operational intensity of the kernel to 2.26 flops/byte and is able to obtain performance close to the maximum possible for that operational intensity. Unlike our approach, Palabos does not perform any kind of time tiling and only employs boundary tile separation and space-blocking optimizations. With ldc-d3q27, Palabos improves the operational intensity, but msLBM obtains further improvement over that in both operational intensity and sustained performance.

In conclusion, the performance of msLBM-optimized code, when running on all 16 cores of our system, is still memory bandwidth bound. Note that there is an upper bound on the sustainable flops/byte ratio for an algorithm given the cache size and possibly

Fig. 16. Roofline model with ceiling for LBM (logarithmic scales are used on both axes). All plotted points, except seq and those explicitly mentioned as being for eight cores, correspond to parallel execution on all 16 cores.

other parameters. To the best of our knowledge, such theoretical upper bounds on the I/O complexity of LBM computations have not been studied. If such bounds are known, one could determine (1) whether the optimal performance will be bounded by available memory bandwidth or compute power and (2) how far our achieved performance is from that optimal performance.

4.6. Summary and Mean Speedups

In summary, msLBM achieves a mean (geometric) speedup of $3\times$ over Palabos and a mean speedup of $2.6\times$ over a native production compiler (icc-omp-par) while running on 16 cores. In addition, we obtain a speedup of $2.47\times$ over the native production compiler on SPEC LBM. Our primary objectives have been to improve over the state of the art with respect to performance while improving programmer productivity, and to present LBM as a class to demonstrate domain-specific compilation and optimization. The ultimate goal is to achieve good performance with a high degree of scaling, which requires at least very good single-node performance based on techniques that could in principle be scaled. msLBM, as evident from the results presented earlier, improves the single-node performance of LBM codes significantly over the state of the art. The problem of bridging the gap between the achievable peak and the sustained performance further is left for future work.

5. CONCLUSIONS

We proposed an automated code generator for a class of LBM computations with the objective to achieve high performance with no user effort involved beyond providing a high-level specification of the problem. Few studies in the past had looked at time

tiling for LBM codes, and had done so manually. We exploited an important similarity between typical representative stencil kernels and LBM to enable polyhedral loop transformations and in turn time tiling for LBM. We also characterized the performance of LBM with the Roofline performance model and concluded that msLBM obtains further improvement over Palabos in operational intensity, sustained performance, and peak achievable performance. Experimental results for standard LBM simulations like Lid-Driven Cavity, Flow Past Cylinder, and Poiseuille Flow showed that our scheme consistently outperformed Palabos—on average by $3\times$ while running on 16 cores of an Intel Xeon Sandybridge system. We also obtain a very significant improvement of $2.47\times$ over the native production compiler on the SPEC LBM benchmark. Besides demonstrating the effectiveness of our system, these results show the merit of automatic time tiling techniques for a very important application domain in computational science.

ACKNOWLEDGMENTS

We are grateful to the reviewers of ACM TACO for their numerous insightful comments. We would also like to thank Anand Deshpande, Bharat Kaul, Aniruddha Shet, and Sunil Sherlekar from Intel Labs, Bangalore, India, for technical discussions on Lattice-Boltzmann Method computations as well as for a research gift that supported this work in part. We also thank them for donation of equipment and software that was used for the experimental evaluation presented.

REFERENCES

- Aravind Acharya and Uday Bondhugula. 2015. PLUTO+: Near-complete modeling of affine transformations for parallelism and locality. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'15)*. 54–64.
- Vinayaka Bandishti, Irshad Pananilath, and Uday Bondhugula. 2012. Tiling stencil computations to maximize parallelism. *Supercomputing (SC)*. Article 40, 40:1–40:11.
- Cédric Bastoul. 2008. Clan: The Chunky Loop Analyzer. (2008). *The Clan User Guide*.
- Somashekar Bhaskaracharya and Uday Bondhugula. 2014. *Automatic Intra-Array Storage Optimization*. Technical Report IISc-CSA-TR-2014-3. Indian Institute of Science, Bangalore, India.
- P. L. Bhatnagar, E. P. Gross, and M. Krook. 1954. A model for collision processes in gases. I. Small amplitude processes in charged and neutral one-component systems. *Physics Review* 94, 3 (May 1954), 511–525. DOI: <http://dx.doi.org/10.1103/PhysRev.94.511>
- Uday Bondhugula, Vinayaka Bandishti, Albert Cohen, Guillaín Potron, and Nicolas Vasilache. 2014. Tiling and optimizing time-iterated computations on periodic domains. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 39–50.
- Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. 2008. A practical automatic polyhedral parallelizer and locality optimizer. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'08)*. 101–113.
- Shiyi Chen and Gary D. Doolen. 1998. Lattice Boltzmann method for fluid flows. *Annual Review of Fluid Mechanics* 30, 1 (1998), 329–364.
- Dominique d'Humières. 2002. Multiple-relaxation-time lattice Boltzmann models in three dimensions. *Philosophical Transactions of the Royal Society of London, Series A: Mathematical, Physical and Engineering Sciences* 360, 1792 (2002), 437–451.
- Tobias Grosser, Albert Cohen, Justin Holewinski, P. Sadayappan, and Sven Verdoolaege. 2014. Hybrid Hexagonal/Classical Tiling for GPUs. In *International Symposium on Code Generation and Optimization (CGO)*. 66–75.
- Tobias Grosser, Albert Cohen, Paul H. J. Kelly, J. Ramanujam, P. Sadayappan, and Sven Verdoolaege. 2013. Split tiling for GPUs: Automatic parallelization using trapezoidal tiles. In *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units (GPGPU-6'13)*. 24–31.
- Tom Henretty, Richard Veras, Franz Franchetti, Louis-Noël Pouchet, J. Ramanujam, and P. Sadayappan. 2013. A stencil compiler for short-vector SIMD architectures. In *ACM International Conference on Supercomputing (ICS)*. 13–24.
- David Joyner, Ondřej Čertík, Aaron Meurer, and Brian E. Granger. 2012. Open source computer algebra systems: SymPy. *ACM Communications in Computer Algebra* 45, 3/4 (2012), 225–234.

- JSON 2001. Introducing JSON. (2001). <http://www.json.org/>.
- Carolin Körner, Thomas Pohl, Ulrich Rüde, Nils Thürey, and Thomas Zeiser. 2006. Parallel lattice Boltzmann methods for CFD applications. In *Numerical Solution of Partial Differential Equations on Parallel Computers*. Springer, 439–466.
- Sriram Krishnamoorthy, Muthu Baskaran, Uday Bondhugula, J. Ramanujam, Atanas Rountev, and P. Sadayappan. 2007. Effective automatic parallelization of stencil computations. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 235–244.
- Federico Massaioli and Giorgio Amati. 2002. Achieving high performance in a LBM code using OpenMP. In *4th European Workshop on OpenMP*.
- Keijo Mattila, Jari Hyväluoma, Tuomo Rossi, Mats Aspö, and Jan Westerholm. 2007. An efficient swap algorithm for the lattice Boltzmann method. *Computer Physics Communications* 176, 3 (2007), 200–210.
- Abdulmajeed A. Mohamad. 2011. *Lattice Boltzmann Method: Fundamentals and Engineering Applications with Computer Codes*. Springer.
- Satish Nadathur, Changkyu Kim, Jatin Chhugani, Hideki Saito, Rakesh Krishnaier, Mikhail Smelyanskiy, Milind Girkar, and Pradeep Dubey. 2012. Can traditional programming bridge the ninja performance gap for parallel computing applications? In *Proceedings of the 39th Annual International Symposium on Computer Architecture (ISCA)*. 440–451.
- Anthony D. Nguyen, Nadathur Satish, Jatin Chhugani, Changkyu Kim, and Pradeep Dubey. 2010. 3.5-D blocking optimization for stencil computations on modern CPUs and GPUs. In *Supercomputing (SC)*. 1–13.
- Aditya Nitsure, Klaus Iglberger, Ulrich Rüde, Christian Feichtinger, Gerhard Wellein, and Georg Hager. 2006. Optimization of cache oblivious lattice Boltzmann method in 2D and 3D. In *Frontiers in Simulation: Simulationstechnique—19th Symposium in Hannover*. 265–270.
- Georg Ofenbeck, Ruedi Steinmann, Victoria Caparros, Daniele G. Spampinato, and Markus Püschel. 2014. Applying the roofline model. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'14)*. 76–85.
- Nissa Osheim, Michelle Mills Strout, Dave Rostron, and Sanjay Rajopadhye. 2008. Smashing: Folding space to tile through time. In *Workshop on Languages and Compilers for Parallel Computing (LCPC)*. Springer-Verlag, 80–93.
- Palabos 2009. Palabos. (2009). <http://www.palabos.org/>.
- Pluto. 2008. PLUTO: A polyhedral automatic parallelizer and locality optimizer for multicores. (2008). <http://pluto-compiler.sourceforge.net>.
- Thomas Pohl, Markus Kowarschik, Jens Wilke, Klaus Iglberger, and Ulrich Rüde. 2003. Optimization and profiling of the cache performance of parallel lattice Boltzmann codes. *Parallel Processing Letters* 13, 4 (2003), 549–560.
- L. Renganarayanan, DaeGon Kim, Sanjay Rajopadhye, and Michelle Mills Strout. 2007. Parameterized tiled loops for free. In *ACM SIGPLAN Conference on Programming Languages Design and Implementation (PLDI)*. 405–414.
- Aniruddha G. Shet, Shahajhan H. Sorathiya, Siddharth Krithivasan, Anand M. Deshpande, Bharat Kaul, Sunil D. Sherlekar, and Santosh Ansumali. 2013. Data structure and movement for lattice-based simulations. *Physics Review E* 88 (July 2013), 013314.
- Sunil Shrestha, Guang R. Gao, Joseph Manzano, Andres Marquez, and John Feo. 2015. Locality aware concurrent start for stencil applications. In *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO'15)*. 157–166.
- Yonghong Song and Zhiyuan Li. 1999. New tiling techniques to improve cache temporal locality. In *ACM SIGPLAN Conference on Programming Languages Design and Implementation (PLDI'99)*. 215–228.
- M. Strout, Larry Carter, Jeanne Ferrante, and Beth Simon. 1998. Schedule-independent storage mapping for loops. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 24–33.
- Robert Strzodka, Mohammed Shaheen, Dawid Pajak, and H.-P. Seidel. 2011. Cache accurate time skewing in iterative stencil computations. In *International Conference on Parallel Processing (ICPP'11)*. 571–581.
- Sauro Succi. 2001. *The Lattice Boltzmann Equation: For Fluid Dynamics and Beyond*. Oxford University Press.
- Yuan Tang, Rezaul Alam Chowdhury, Bradley C. Kuszmaul, Chi-Keung Luk, and Charles E. Leiserson. 2011. The pochoir stencil compiler. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA'11)*. 117–128.

- Gerhard Wellein, Thomas Zeiser, Georg Hager, and Stefan Donath. 2006. On the single processor performance of simple lattice Boltzmann kernels. *Computers & Fluids* 35, 8 (2006), 910–919.
- Jens Wilke, Thomas Pohl, Markus Kowarschik, and Ulrich Rde. 2003. Cache performance optimizations for parallel lattice Boltzmann codes. In *Euro-Par 2003 Parallel Processing*. Springer, 441–450.
- Samuel Williams, Jonathan Carter, Leonid Oliker, John Shalf, and Katherine Yelick. 2009. Optimization of a Lattice Boltzmann computation on state-of-the-art multicore platforms. *Journal of Parallel and Distributed Computing* 69, 9 (Sept. 2009), 762–777.
- Samuel Williams, Leonid Oliker, Jonathan Carter, and John Shalf. 2011. Extracting ultra-scale lattice Boltzmann performance via hierarchical and distributed auto-tuning. In *Supercomputing (SC)*. IEEE, 1–12.
- Samuel Williams, Andrew Waterman, and David Patterson. 2009. Roofline: An insightful visual performance model for multicore architectures. *Communications of the ACM* 52, 4 (2009), 65–76.
- Markus Wittmann, Thomas Zeiser, Georg Hager, and Gerhard Wellein. 2013. Comparison of different propagation steps for lattice Boltzmann methods. *Computers & Mathematics with Applications* 65, 6 (2013), 924–935.
- Michael Wolfe. 1995. *High Performance Compilers for Parallel Computing*, Carter Shanklin and Leda Ortega (Eds.). Addison-Wesley Longman Publishing Co., Inc.
- D. Wonnacott. 2000. Using time skewing to eliminate idle time due to memory bandwidth and network limitations. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 171–180.
- David Wonnacott. 2002. Achieving scalable locality with time skewing. *International Journal of Parallel Programming* 30, 3 (2002), 181–221.
- David Wonnacott and Michelle Strout. 2013. On the scalability of loop tiling techniques. In *International Workshop on Polyhedral Compilation Techniques (IMPACT)*.
- Jingling Xue. 2000. *Loop Tiling for Parallelism*. Kluwer Academic Publishers.
- Qisu Zou and Xiaoyi He. 1996. On pressure and velocity flow boundary conditions and bounceback for the lattice Boltzmann BGK model. *arXiv preprint comp-gas/9611001* (1996).

Received January 2015; accepted February 2015