Cache Performance Optimizations for Parallel Lattice Boltzmann Codes

Jens Wilke, Thomas Pohl, Markus Kowarschik, and Ulrich Rüde

Lehrstuhl für Systemsimulation (Informatik 10)
Institut für Informatik
Friedrich–Alexander–Universität Erlangen–Nürnberg, Germany
{*Jens.Wilke,Thomas.Pohl,Markus.Kowarschik,Ulrich.Ruede*}*@cs.fau.de*

Abstract. When designing and implementing highly efficient scientific applications for parallel computers such as clusters of workstations, it is inevitable to consider and to optimize the single–CPU performance of the codes. For this purpose, it is particularly important that the codes respect the hierarchical memory designs that computer architects employ in order to hide the effects of the growing gap between CPU performance and main memory speed. In this paper, we present techniques to enhance the single–CPU efficiency of lattice Boltzmann methods which are commonly used in computational fluid dynamics. We show various performance results to emphasize the effectiveness of our optimization techniques.

1 Introduction

In order to enhance the performance of any parallel scientific application, it is important to focus on two related optimization issues. Firstly, it is necessary to minimize the parallelization overhead itself. These efforts commonly target the choice of appropriate load balancing strategies as well as the minimization of communication overhead by hiding network latency and bandwidth. Secondly, it is necessary to exploit the individual parallel resources as efficiently as possible; e.g., by achieving as much performance as possible on each CPU in the parallel environment. This is especially true for distributed memory systems found in computer clusters based on off–the–shelf workstations communicating via fast networks. Our current research focuses on this second optimization issue.

In order to mitigate the effects of the growing gap between theoretically available processor speed and main memory performance, today's computer architectures are typically based on *hierarchical memory designs*, involving CPU registers, several levels of cache memories (caches), and main memory [10]. Remote main memory and external memory (e.g., hard disk drives) can be considered as the slowest components in any memory hierarchy. Fig. 1 illustrates the memory architecture of a current high performance workstation [12].

Efficient execution in terms of work units per second can only be obtained if the codes exploit the underlying memory design. This is particularly true for numerically intensive codes. Unfortunately, current compilers cannot perform

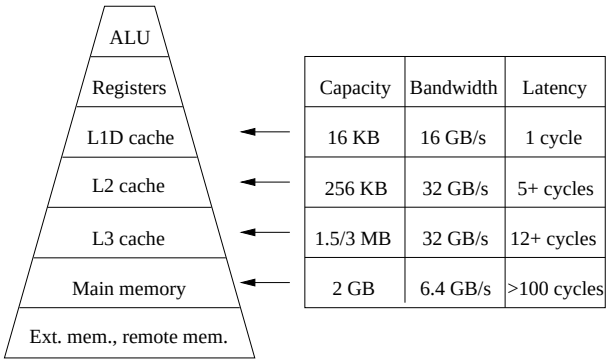

Fig. 1. Memory architecture of a workstation based on an Intel Itanium2 CPU with three levels of on–chip cache [12].

highly sophisticated code transformations automatically. Much of this optimization effort is therefore left to the programmer [8, 14].

Generally speaking, efficient parallelization and cache performance tuning can both be interpreted as *data locality optimizations*. The underlying idea is to keep the data to be processed as close as possible to the corresponding ALU. From this viewpoint, cache optimizations form an extension of classical parallelization efforts.

Research has shown that the cache utilization of iterative algorithms for the numerical solution of linear systems can be improved significantly by applying suitable combinations of *data layout optimizations* and *data access optimizations* [3, 6]. The idea behind these techniques is to enhance the *spatial locality* as well as the *temporal locality* of the code [11]. Similar work focuses on other algorithms of numerical linear algebra [16] and hardware–oriented FFT implementations [7]. An overview of cache optimization techniques for numerical algorithms can be found in [13]. Our current work concentrates on improving the cache utilization of a parallel implementation of the *lattice Boltzmann method (LBM)*, which represents a particle–based approach towards the numerical simulation of problems in computational fluid dynamics (CFD) [5, 17].

This paper is structured as follows. Section 2 contains a brief introduction to the LBM. Section 3 presents code transformation techniques to enhance the single–CPU performance of the LBM and introduces a *compressed grid storage* technique, which almost halves its data set size. The performance results of LBM implementations on various platforms are shown in Section 4. We conclude in Section 5.

2 The Lattice Boltzmann Method

It is important to mention up front that we only give a brief description of the LBM in this section, since the actual physics behind this approach are not essential for the application of our optimization techniques.

The usual approach towards solving CFD problems is based on the numerical solution of the governing partial differential equations, particularly the Navier–Stokes equations. The idea behind this approach is to discretize the computational domain using finite differences, finite elements or finite volumes, to derive algebraic systems of equations and to solve these systems numerically.

In contrast, the LBM is a particle–oriented technique, which is based on a microscopic model of the moving fluid particles. Using $\boldsymbol{x}$ to denote the position in space, $\boldsymbol{u}$ to denote the particle velocity, and t to denote the time parameter, the so–called *particle distribution function* $f(\boldsymbol{x}, \boldsymbol{u}, t)$ is discretized in space, velocity, and time. This results in the computational domain being regularly divided into cells (so–called *lattice sites*), where the current state of each lattice site is defined by an array of floating–point numbers that represent the distribution functions w.r.t. to the discrete directions of velocity.

The LBM then works as follows. In each time step, the entire grid (lattice) is traversed, and the distribution function values at each site are updated according to the states of its neighboring sites in the grid at the previous discrete point in time. This update step consists of a *stream* operation, where the corresponding data from the neighboring sites are retrieved, and a *collide* operation, where the new distribution function values at the current site are computed according to a suitable model of the microscopic behavior of the fluid particles, preserving hydrodynamic quantities such as mass density, momentum density, and energy. Usually, this model is derived from the *Boltzmann equation*

$$\frac{\partial f}{\partial t} + \langle \boldsymbol{u}, \nabla f \rangle = \frac{1}{\lambda} \left(f - f^{(0)} \right) \quad ,$$

which describes the time–dependent behavior of the particle distribution function f. In this equation, $f^{(0)}$ denotes the equilibrium distribution function, λ is the relaxation time, $\langle ., . \rangle$ denotes the standard inner product, and ∇f denotes the gradient of f w.r.t. the spatial dimensions [17].

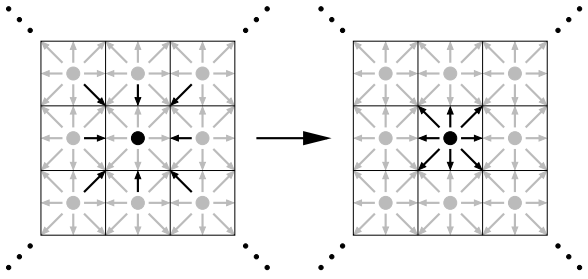

Fig. 2. Representation of the LBM updating a single lattice site by a stream operation (left) and a subsequent collide operation (right).

Fig. 2 shows the update of an individual lattice site in 2D. The grid on the left illustrates the source grid corresponding to the previous point in time, the

darker arrows represent the particle distribution function values being read. The grid on the right illustrates the destination grid corresponding to the current point in time, the dark arrows represent the new particle distribution function values; i.e., after the collide operation.

Note that, in every time step, the lattice may be traversed in any order since the LBM only accesses data corresponding to the previous point in time in order to compute new distribution function values. From an abstract point of view, the structure of the LBM thus parallels the structure of an implementation of Jacobi's method for the iterative solution of linear systems on structured meshes. The time loop in the LBM corresponds to the iteration loop in Jacobi's method.

In contrast to the lattice gas approach [17] where hexagonal grids are more common, we focus on orthogonal grids, since they are almost exclusively used for the LBM.

3 Optimization Techniques for Lattice Boltzmann Codes

3.1 Fusing the stream and the collide operation

A naive implementation of the LBM would perform two entire sweeps over the whole data set in every time step: one sweep for the stream operation, copying the distribution function values from each lattice site into its neighboring sites, and a subsequent sweep for the collide operation, calculating the new distribution function values at each site.

A first step to improve performance is to combine the streaming and the collision step. The idea behind this so–called *loop fusion* technique is to enhance the temporal locality of the code and thus the utilization of the cache [1]. Instead of passing through the data set twice per time step, the fused version retrieves the required data from the neighboring cells and immediately calculates the new distribution function values at the current site, see again Fig. 2. Since this tuning step is both common and necessary for all subsequent transformations, we consider this fused version as our starting point for further cache optimizations.

3.2 Data Layout Optimizations

Accessing main memory is very costly compared to even the lowest cache level. Therefore, it is essential to choose a memory layout for the implementation of the LBM which allows the code to exploit the benefits of the hierarchical memory architecture.

Since we need to maintain the grid data for any two successive points in time, our initial storage scheme is based on two arrays of records. Each of these records stores the distribution function values of an individual lattice site. Clustering the distribution function values is a reasonable approach since the smallest unit of data to be moved between main memory and cache is a *cache block* which typically contains several data items that are located adjacent in memory.

In the following, we present two data layout transformations that aim at further enhancing the spatial locality of the LBM implementation; *grid merging* and *grid compression*.

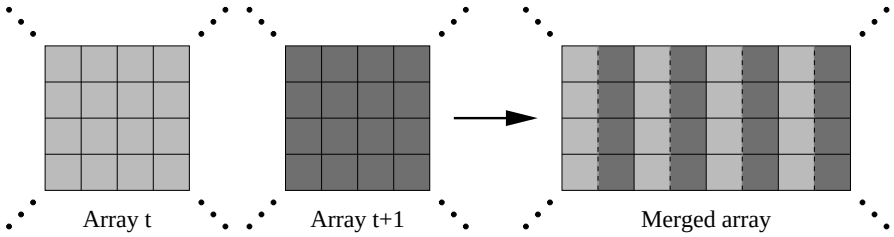

Fig. 3. The two arrays which store the grid data at time t and time $t+1$, respectively, are merged into a single array.

Grid merging. During a fused stream–and–collide step it is necessary to load data from the grid (array) corresponding to time t, to calculate new distribution function values, and to store these results into the other grid corresponding to time $t+1$.

Due to our initial data layout (see above) the first two steps access data which are tightly clustered in memory. Storing the results, however, involves the access of memory locations that can be arbitrarily far away. In order to improve spatial locality, we introduce an interleaved data layout where the distribution function values of each individual site for two successive points in time are kept next to each other in memory. This transformation is commonly called *array merging* [11, 13]. Fig. 3 illustrates the application of this technique for the 2D case.

Grid compression. The idea behind this data layout is both to save memory and to increase spatial locality. We concentrate on the 2D case in this paper, while the extension of this technique to the 3D case is currently being implemented.

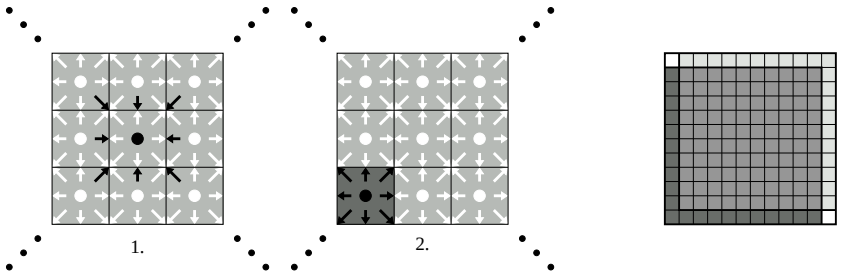

Fig. 4. Fused stream–and–collide step for the cell in the middle (left, 1 and 2), layout of the two overlayed grids after applying the grid compression technique (right).

In an implementation of the LBM in 2D, nearly half of the memory can be saved by exploiting the fact that only the data from the eight neighboring cells are required to calculate the new distribution function values at any regular site

of the grid[1]. It is therefore possible to overlay the two grids for both points in time, introducing a diagonal shift of one row and one column of cells into the stream–and–collide operation. The direction of the shift then determines the update sequence in each time step since we may not yet overwrite those distribution function values that are still required henceforth.

In Fig. 4, the light gray area contains the values for the current time t. The two pictures on the left illustrate the stream–and–collide operation for a single cell: the values for time $t+1$ will be stored with a diagonal shift to the lower left. Therefore, the stream–and–collide sweep must also start with the lower left cell. Consequently, after one complete sweep, the new values (time $t+1$) are shifted compared to the previous ones (time t). For the subsequent time step, the sweep must start in the upper right corner. The data for time $t+2$ must then be stored with a shift to the upper right. After two successive sweeps, the memory locations of the distribution functions are the same as before. This alternating scheme is shown in the right picture of Fig. 4.

3.3 Data Access Optimizations

Data access optimizations change the order in which the data are referenced in the course of the computation, while respecting all data dependencies. Our access transformations for implementations of the LBM are based on the *loop blocking (loop tiling)* technique. The idea behind this general approach is to divide the iteration space of a loop or a loop nest into blocks and to perform as much computational work as possible on each individual block before moving on to the next block. If the size of the blocks is chosen according to the cache size, loop blocking can significantly enhance cache utilization and, as a consequence, yield significant performance speedups [1, 8, 13].

In the following, we present two blocking approaches in order to increase the temporal locality of the 2D LBM implementation. Both blocking techniques take advantage of the stencil memory access exhibited by the LBM. Because of this local operation, a grid site can be updated to time $t+1$ as soon as the sites it depends on have been updated to time t.

It is important to point out that each of these access transformations can be combined with either of the two layout transformations which we have introduced in Section 3.2.

1D blocking. Fig. 5 illustrates an example, where two successive time steps are blocked into a single pass through the grid. White cells have been updated to time t, light gray cells to time $t+1$, and dark gray cells to time $t+2$. It can be seen that in Grid 2, all data dependencies of the bottom row are fulfilled, and can therefore be updated to time $t+2$, shown in Grid 3. This is performed repeatedly until the entire grid has been updated to time $t+2$, see Grids 4 to 10. The downside to this method is that, even two rows may contain too much

[1] For the sake of simplicity, we focus on regular sites and omit the description of how to treat boundary sites and obstacle sites separately.

data to fit into cache if the grid is too large. In this case, no performance gain will be observed.

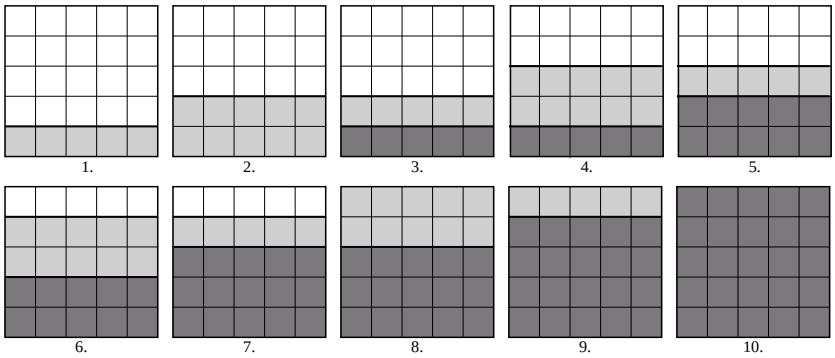

Fig. 5. 1D blocking technique to enhance temporal locality.

2D blocking. Fig. 6 illustrates an example, in which a 4×4 block of cells is employed. This means that four successive time steps are performed during a single pass through the grid. Since the data contained in the 2D block is independent of the grid size, it will always (if the size is chosen appropriately) fit into the highest possible cache level, regardless of the grid size. In Fig. 6, Grids 1 to 4 demonstrate the handling of one 4×4 block, and Grids 5 to 8 a second 4×4 block. The block which can be processed moves diagonally down and left in order to avoid violating data dependencies. Obviously, special handling is required for those sites near grid edges which cannot form a complete 4×4 block.

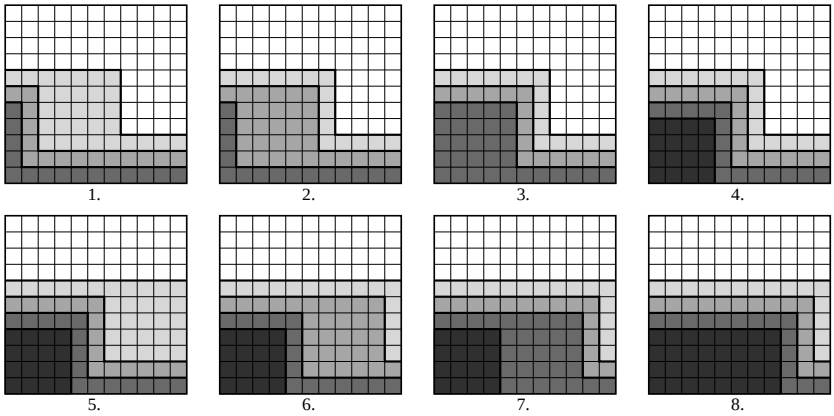

Fig. 6. 2D blocking technique to enhance temporal locality.

4 Performance Results

In order to test and benchmark the various implementations of the LBM, a well known problem in fluid dynamics known as the *lid–driven cavity* has been used. It consists of a closed box, where the top of the box, the lid, is continually dragged across the fluid in the same direction. The fluid eventually forms a circular flow around the center of the box [9].

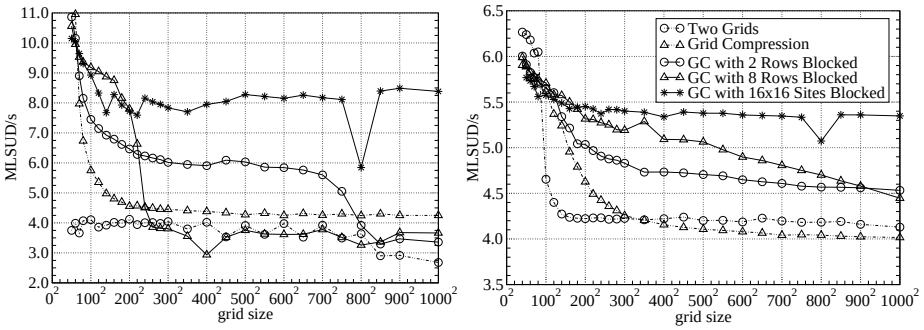

Fig. 7. Performance in millions of lattice site updates per second (MLSUD/s) on machines based on an AMD Athlon XP 2400+ (left) and on an Intel Itanium2 (right), respectively, for various grid sizes.

Performance. Fig. 7 demonstrates the performance of five different implementations of the LBM in ANSI C++ on two different machines[2] for various grid sizes. Note that problem size n means that the grid contains n^2 cells. The simple implementation involving a source and destination grid is the slowest, the implementation using the layout based on grid compression is slightly faster. Two implementations of 1D blocking are shown, one with two time steps blocked, and one with eight. Initially, they show a significant increase in performance. For large grids, however, performance worsens since the required data cannot fit into even the lowest level of cache. This effect is more dramatic on the AMD processor since it has only 256 kB of L2 and no L3 cache, whereas the Intel CPU even has 1.5 MB of L3 cache on–chip. Finally, the 2D blocked implementation shows a significant increase in performance for all grid sizes. It should be noted that each implementation based on the grid merging layout performs worse than its counterpart based on grid compression. Therefore, no performance results for the grid merging technique are shown.

We have obtained similar performance gains on several further platforms. For example, our results include speedup factors of 2–3 on machines based on DEC

[2] We use an AMD Athlon XP 2400+ based PC (2 GHz) [2], Linux, gcc 3.2.1, as well as an Intel Itanium2 based HP zx6000 [12], Linux, Intel ecc V7.0. Aggressive compiler optimizations have been enabled in all experiments.

Alpha 21164 and DEC Alpha 21264 CPUs. Both of them particularly benefit from large off–chip caches of 4 MB.

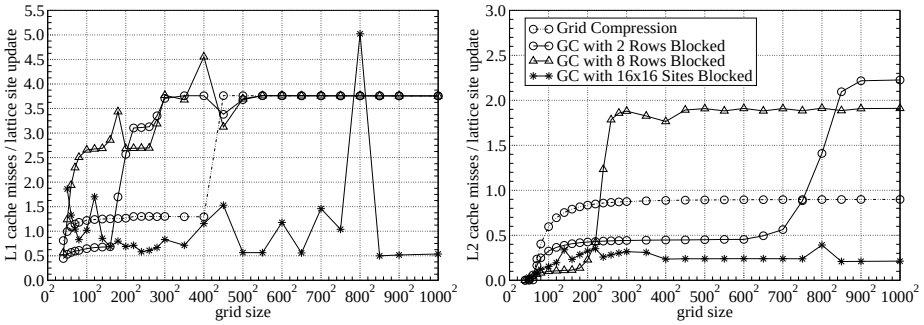

Fig. 8. Cache behavior of the AMD Athlon XP measured with PAPI.

Cache Behavior. Fig. 8 demonstrates the behavior of the L1 and L2 cache of the AMD Athlon XP for the different implementations of the LBM. These results have been obtained by using the profiling tool *PAPI* [4]. When compared to Fig. 7 it can be seen that there is a strong correlation between the number of cache misses and the performance of the code. The correlation between the performance drop of the two 1D blocked implementations and their respective rise in L2 cache misses is especially dramatic. Additionally, Fig. 8 reveals the cause of the severe performance drop exhibited by the 2D blocked implementation at a grid size of 800^2. The high number of cache misses are *conflict misses* which are caused when large amounts data in memory are mapped to only a small number of cache lines [15].

5 Conclusions and Future Work

Due to the still widening gap between CPU and memory speed, hierarchical memory architectures will continue to be a promising optimization target. The CPU manufacturers have already announced new generations of CPUs with several megabytes of on–chip cache in order to hide the slow access to main memory.

We have demonstrated the importance of considering the single–CPU performance before using parallel computing methods in the framework of a CFD code based on the LBM. By exploiting the benefits of hierarchical memory architectures of current CPUs, we were have obtained factors of 2–3 in performance on various machines. Unfortunately, the parallelization of cache–optimized codes is commonly tedious and error–prone due to their implementation complexity.

We are currently extending our techniques to the 3D case. From our experience in cache performance optimization of iterative linear solvers, we expect that appropriate data layout transformations and data access optimizations, in particular loop blocking, can be applied to improve the performance.

Acknowledgments

We wish to thank the members of the High Performance Computing Group at the Computing Center in Erlangen *(RRZE)* for their support.

References

1. R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures*. Morgan Kaufmann Publishers, San Francisco, CA, USA, 2001.
2. AMD Corporation. *AMD Athlon XP Processor 8 Data Sheet*, 2002. Publication #25175 Rev. F.
3. F. Bassetti, K. Davis, and D. Quinlan. Temporal Locality Optimizations for Stencil Operations within Parallel Object–Oriented Scientific Frameworks on Cache–Based Architectures. In *Proc. of the Int. Conference on Parallel and Distributed Computing and Systems*, pages 145–153, Las Vegas, NV, USA, 1998.
4. S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci. A Portable Programming Interface for Performance Evaluation on Modern Processors. *Int. Journal of High Performance Computing Applications*, 14(3):189–204, 2000.
5. S. Chen and G.D. Doolen. Lattice Boltzmann Method for Fluid Flow. *Annual Reviews of Fluid Mechanics*, 30:329–364, 1998.
6. C.C. Douglas, J. Hu, M. Kowarschik, U. Rüde, and C. Weiß. Cache Optimization for Structured and Unstructured Grid Multigrid. *Electronic Transactions on Numerical Analysis*, 10:21–40, 2000.
7. M. Frigo and S.G. Johnson. FFTW: An Adaptive Software Architecture for the FFT. In *Proc. of the Int. Conference on Acoustics, Speech, and Signal Processing*, volume 3, pages 1381–1384, Seattle, WA, USA, 1998.
8. S. Goedecker and A. Hoisie. *Performance Optimization of Numerically Intensive Codes*. SIAM, 2001.
9. M. Griebel, T. Dornseifer, and T. Neunhoeffer. *Numerical Simulation in Fluid Dynamics*. SIAM, 1998.
10. J. Handy. *The Cache Memory Book*. Academic Press, second edition, 1998.
11. J.L. Hennessy and D.A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publisher, Inc., San Francisco, CA, USA, second edition, 1996.
12. Intel Corporation. *Intel Itanium2 Processor Reference Manual*, 2002. Document Number: 251110–001.
13. M. Kowarschik and C. Weiß. An Overview of Cache Optimization Techniques and Cache–Aware Numerical Algorithms. In *Algorithms for Memory Hierarchies*, volume 2625 of *LNCS*. Springer, 2003.
14. D. Loshin. *Efficient Memory Programming*. McGraw–Hill, 1998.
15. G. Rivera and C.-W. Tseng. Data Transformations for Eliminating Conflict Misses. In *Proc. of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, Montreal, Canada, 1998.
16. R.C. Whaley and J. Dongarra. Automatically Tuned Linear Algebra Software. In *Proc. of the ACM/IEEE Supercomputing Conference*, Orlando, FL, USA, 1998.
17. D.A. Wolf-Gladrow. *Lattice–Gas Cellular Automata and Lattice Boltzmann Models*. Springer, 2000.