FRIEDRICH-ALEXANDER-UNIVERSITÄT ERLANGEN-NÜRNBERG
INSTITUT FÜR INFORMATIK (MATHEMATISCHE MASCHINEN UND DATENVERARBEITUNG)

Lehrstuhl für Informatik 10 (Systemsimulation)

Cache Optimizations for the Lattice Boltzmann Method in 2D

Jens Wilke

Studienarbeit

Cache Optimizations for the Lattice Boltzmann Method in 2D

Jens Wilke
Studienarbeit

Aufgabensteller:	Prof. Dr. Ulrich Rüde
Betreuer:	Dipl.–Phys. T. Pohl, Dipl.–Inf. M. Kowarschik
Bearbeitungszeitraum:	Mai 2002 – Februar 2003

Erklärung:

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Erlangen, den 28. Februar 2003 ....

Contents

Abstract

The lattice Boltzmann Method (LBM) has, in recent years, gained wide acceptance as an efficient alternative to simulate flows in complex geometries as opposed to using algorithms that numerically solve the Navier-Stokes equation. The lattice Boltzmann method divides space into a regular grid, where each grid site determines the local behavior of the fluid. For algorithms working on regular grids, execution speed is often limited by access to the main memory. However, the speed of computer processors is currently increasing much faster than this memory latency decreases. Thus, modern computer architectures employ memory hierarchies based on caches that are 30 to 40 times faster than the main memory, to mitigate this effect. It has been shown that many algorithms working on regular grids can be optimized in terms of spatial and temporal locality, to efficiently use these memory hierarchies. This thesis will investigate the transferability of these techniques to the lattice Boltzmann method using the C++ programming language and it will provide quantitative values for the gain in performance.

Chapter 1

Introduction to Lattice Boltzmann

1.1 Lattice Methods

Conventional methods of computational fluid dynamics compute flow fields, such as velocity or pressure, by solving the time–dependent Navier-Stokes equation In contrast, the lattice methods begin from a particle description of matter instead of computing fields. The lattice is formed by particles that exist on a set of discrete points spaced at regular intervals. Time is also divided into discrete time-steps. During each time-step particles "jump" to the next lattice site and then scatter according to relatively simple kinetic models, the Boltzmann equation in particular, that conserve mass, momentum, and energy. This simple model of molecular dynamics is constructed in such a way as to include the essential properties of real microscopic processes. Thus, the mesoscopic properties of the lattice simulations closely approximate the continuous equations as described by Navier-Stokes. Nevertheless, they retain the advantages of a particle description, such as intuitive physical processes and straightforward implementation of boundary conditions and parallel algorithms [CDE94].

1.2 The Lattice-Boltzmann-Equation

In the LBM, real numbers at each lattice site represent the single-particle distribution function at that site. The distribution function describes the expected number of particles in each of the particle states i. In the simplest model, which is the one used in the course of this thesis, each particle state i is defined by a particle velocity, which is limited to a discrete set of allowed velocities. During each discrete time-step of the simulation, particles are *streamed* to the nearest neighboring cells in their direction of motion, where they *collide* with other particles that arrive at the same site. The outcome of the collision is determined by solving the kinetic Boltzmann equation for the new particle distribution function for the next time-step at that site [CDE94].

The Boltzmann equation deals with the single particle distribution function $f(\mathbf{x}, \xi, t)$, where ξ is the particle velocity, in phase space $(\mathbf{x}, \xi)$ and time t. The kinetic model using the Boltzmann equation with with the single relaxation time approximation is [MSY02]:

$$\frac{\partial f}{\partial t} + \xi \frac{\partial f}{\partial \mathbf{x}} = -\frac{1}{\tau}\left[f - f^{(0)}\right] \tag{1.1}$$

where $f^{(0)}$ is the equilibrium distribution function (the Maxwell–Boltzmann distribution function) used to calculate the new values for the particle distribution function. To solve for f numerically, equation 1.1 is first discretized in the velocity space ξ using a finite set of velocities $\{\xi_\alpha\}$:

$$\frac{\partial f_\alpha}{\partial t} + \mathbf{e}_\alpha \frac{\partial f_\alpha}{\partial \mathbf{x}} = -\frac{1}{\tau}\left[f_\alpha - f_\alpha^{(0)}\right] \tag{1.2}$$

where $f_\alpha(\mathbf{x}, t) = f(\mathbf{x}, \xi, t)$ and $f_\alpha^{(0)}(\mathbf{x}, t) = f^{(0)}(\mathbf{x}, \xi, t)$. For this thesis the nine velocity LBE model on a 2-D rectangular lattice, denoted as the D2Q9 model, was used. For 3-D flows, as well as 2-D flows, there are several models that have been used in the literature, see [MSY02]. The velocities in the D2Q9 model are as follows [WG00]:

$$\mathbf{e}_\alpha = \begin{cases} (0,0), & \alpha = C \\ (0,\pm c), & \alpha = N, S \\ (\pm c, 0), & \alpha = E, W \\ (c, \pm c), & \alpha = NE, SE \\ (-c, \pm c), & \alpha = NW, SW \end{cases} \tag{1.3}$$

where $c = \Delta x/\Delta t$ and Δx and Δt are the lattice constant and the time step, respectively. To facilitate further reading, the directions these velocities point to will be referred to as center (C), north (N), east (E), south (S), west (W), northeast (NE), southeast (SE), southwest (SW), and northwest (NW). Finally, equation 1.2 is discretized in space $\mathbf{x}$ and time t into:

$$f_\alpha\left(\mathbf{x} + \mathbf{e}_\alpha \Delta t, t + \Delta t\right) = f_\alpha(\mathbf{x}, t) - \frac{\Delta t}{\tau}\left[f_\alpha(\mathbf{x}, t) - f_\alpha^{(0)}(\mathbf{x}, t)\right] \tag{1.4}$$

For athermal fluids, the equilibrium distribution function $f_\alpha^{(0)}$ is approximated by a Taylor series expansion [MSY02]:

$$f_\alpha^{(eq)} = \omega_\alpha \rho \left[1 + \frac{3}{c^2}\mathbf{e}_\alpha \cdot \mathbf{u} + \frac{9}{c^4}(\mathbf{e}_\alpha \cdot \mathbf{u})^2 + \frac{3}{2c^2}(\mathbf{u} \cdot \mathbf{u})\right] \tag{1.5}$$

where ω_α is a weighting factor and $\mathbf{e}_\alpha$ is a discrete velocity from equation 1.3. The mass density $\rho(t, \mathbf{x})$ and the momentum density $\rho(t, \mathbf{x})\mathbf{u}(t, \mathbf{x})$ are evaluated by the following quadrature formulas [WG00]:

$$\rho = \sum_\alpha f_\alpha \tag{1.6}$$

$$\rho \mathbf{u} = \sum_\alpha \mathbf{e}_\alpha f_\alpha \tag{1.7}$$

In order to implement the LBM all that is still missing are the initial values for the particle distribution function of a fluid at rest, and the weighting factors ω_α which, for the D2Q9 model, are [WG00]:

$$\omega_\alpha = f_\alpha(\mathbf{x}, 0) = \begin{cases} \frac{4}{9}, & \alpha = C \\ \frac{1}{9}, & \alpha = N, E, S, W \\ \frac{1}{36}, & \alpha = NE, SE, SW, NW \end{cases} \tag{1.8}$$

1.3 Boundary Condition

Depending on what one is simulating, most boundary conditions for a solid boundary used in the LBE method are based upon the *bounce-back* boundary condition. This boundary is also known as the *no-slip* condition, as opposed to the *free-slip* boundary. In the bounce back boundary condition Any particles that hit a boundary simply reverse their velocity so that the average velocity at the boundary is automatically zero, as observed experimentally [CDE94].

1.4 Lid Driven Cavity

In order to test and benchmark the various implementations of the LBM, a well known test case in fluid dynamics known as the lid driven cavity was used. The lid driven cavity consists of a closed box, where the top of the box, the lid, is continually dragged in the same direction across the fluid. After a short while, the fluid in the box will form a flow similar to that in Figure 1.1.

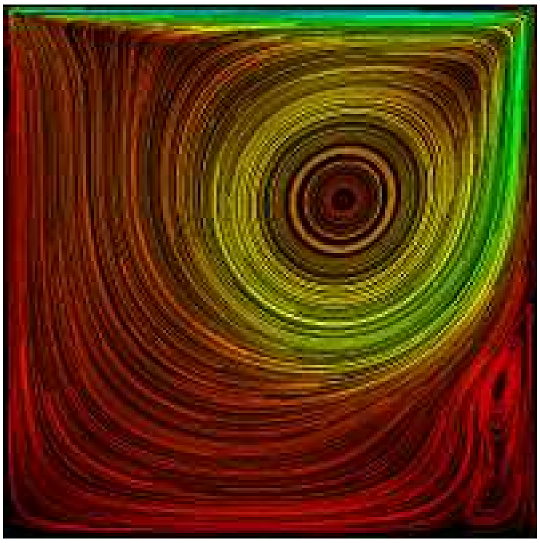


Figure 1.1: An example picture of the lid driven cavity problem. This picture is the 10000th time-step using a 200^2 sized grid.

Chapter 2

Initial Lattice-Boltzman Implementation

This chapter will present the first implementation of the lattice Boltzmann method in C++ using the equations presented in the previous chapter. Preliminary performance results will also be presented, to use as a baseline to gauge the effectiveness of the cache optimizations presented in later chapters.

2.1 Jacobi Method

The purpose of this section is to demonstrate parallels between the Jabobi method and the LBM. Those readers not familiar with the Jacobi method can safely skip this section or see [BF01] for more information. The Jacobi method is one of the simplest numerical methods of solving linear systems of equation. This section will consider a specific variant of the Jacobi method used to solve those linear systems generated when solving boundary values problems (BVP) of elliptic partial differential equations (PDE). Although solving PDEs using the Jacobi method has little physical correlation to simulating fluid dynamics using the LBM, there are nonetheless several important algorithmic similarities. Therefore, throughout the rest of this thesis the Jacobi method will occasionally be used to illustrate the steps performed by the LBM. Algorithm 2.1 shows a simple implementation of the Jacobi algorithm for BVPs.

Algorithm 2.1 Jacobi Algorithm

double $src[n, m]$
double $dst[n, m]$
for $it = 0$ **to** $it = MAXIT$ **by** 1 **do**
    for $i = 1$ **to** $i = n - 1$ **by** 1 **do**
        for $j = 1$ **to** $j = m - 1$ **by** 1 **do**
            $dst[i, j] = \frac{1}{4}(src[i-1, j] + src[i, j-1] + src[i, j+1] + src[i+1, j])$
        end for
    end for
    $swap(src, dst)$
end for

While the calculation performed inside the loop nest is not terribly important in this context, the stencil access to the *src* array required to perform it, as well as the structure of the loop nest bear closer examination. To calculate the new value of each element in the *dst* array, the elements to the south, east, west, and north of the element in the same position in the *src* array must be accessed. This access pattern is called a 4-point stencil. As will be shown later in this chapter, the LBM uses a 9-point stencil to update each element in the lattice.

On examining the loop structure it becomes obvious that the two inner loops are traversing the *src* and *dst* array almost completely, excepting all border elements, whereas the outermost loop determines how often the grid is traversed. The *swap* command denotes that at iteration n, data will flow in the opposite direction than in iteration $n - 1$. In other words, once *dst* has been completely updated in iteration $n-1$, it becomes *src* in the next iteration n, whereas *src* in iteration $n-1$ becomes *dst* in iteration n.

A more efficient algorithm for solving linear systems, the Gauss-Seidel algorithm, uses only a single 2-dimensional array. This is possible because in the Gauss-Seidel algorithm elements being updated in iteration n are allowed use elements already updated in the same iteration n, as well as elements from iteration $n-1$. This kind of transformation cannot be performed on the LBM. In the Jacobi method as well as the Gauss-Seidel method each iteration step is necessary to find a solution to a system, but do not represent any time dependent, physical process. On the other hand, the LBM is constructed specifically to model a dynamic process, and not to converge to a solution. Each individual time-step is a solution of the dynamic behavior of a fluid and depends entirely upon the behavior of the fluid in the previous time-step. Therefore time-steps cannot be mixed when calculating new values as they are in the Gauss-Seidel method.

2.2 Implementing the LBM

To facilitate the implementation of the LBM, the lattice Boltzmann equation 1.4 can be separated into two steps, known as the *collide step* and the *stream step*, shown in equation 2.1 and 2.2.

$$\tilde{f}_\alpha(\mathbf{x}, t+\Delta t) = f_\alpha(\mathbf{x}, t) - \frac{\Delta t}{\tau}\left[f_\alpha(\mathbf{x}, t) - f_\alpha^{(0)}(\mathbf{x}, t)\right] \tag{2.1}$$

$$f_\alpha\left(\mathbf{x} + \mathbf{e}_a \Delta t, t+\Delta t,\right) = \tilde{f}_\alpha(\mathbf{x}, t+\Delta t) \tag{2.2}$$

where $\tilde{f}_\alpha(t, \mathbf{x})$ denotes the post-collision state of the distribution function. Equation 2.1 can be interpreted as follows: the particles that are currently at lattice site $\mathbf{x}$ at time-step t interact with one another, affecting the particle distribution in the next time- step $t+\Delta t$. This interaction is modeled by the right hand side of the equation. Equation 2.2 can be interpreted as the flow of the particles from one lattice site to the next. The particles described by the value of the distribution function at N, $\tilde{f}_N(\mathbf{x}, t+\Delta t)$, will flow to N at the site to the north of this site, described by $f_N\left(\mathbf{x} + \mathbf{e}_N \Delta t, t+\Delta t,\right)$, the value of the distribution function at E will flow to E at the site to the east, and so on for each direction except C, which remain at the current lattice site.

An equivalent and perhaps more intuitive formulation for equations 2.1 and 2.2 switches the order of the collide and stream steps as follows [CDE94]:

$$\tilde{f}_\alpha(\mathbf{x}, t+\Delta t) = f_\alpha\left(\mathbf{x} - \mathbf{e}_a \Delta t, t\right) \tag{2.3}$$

$$f_\alpha(\mathbf{x}, t+\Delta t) = \tilde{f}_\alpha(\mathbf{x}, t+\Delta t) - \frac{\Delta t}{\tau}\left[\tilde{f}_\alpha(\mathbf{x}, t+\Delta t) - f_\alpha^{(0)}(\mathbf{x}, t+\Delta t)\right] \tag{2.4}$$

In this formulation, the stream step (Equation 2.3) gathers the values for the particle distribution function from the surrounding lattice sites, and then the collide step (Equation 2.4) is performed. See Figure 2.1 for an illustration. The first step, the stream step, gathers particle distribution values from the eight sites surrounding the site being updated, the *rest* particle distribution comes from the center site, and writes them to the particle distribution function of the cell being updated. In the second step, the collide step, the particle distribution function values are smoothed towards equilibrium, represented by the circle. This is the method used in this thesis. The similarity to the Jacobi algorithm is quite apparent here; values are gathered from surrounding sites using a stencil, and then used to calculate new values for the next time-step.

2.2.1 Implementing the Particle Distribution Function

Since the first implementation of the LBM was kept as simple and intuitive as possible, each lattice site is represented by an instance of a simple C++ class. Section 1.2 has shown that for the D2Q9 model, the particle distribution function consists of 9 values, one value for the rest particle (C) and one value for each direction north (N), east (E), south (S), west (W), northeast (NE), southeast (SE), southwest (SW), and northwest (NW). These were implemented as member variables of the C/C++ type *double*. In the implementation the class was named *gridsite* and will be used to represent a generic data structure containing nine double precision (64 bit) floating point values in

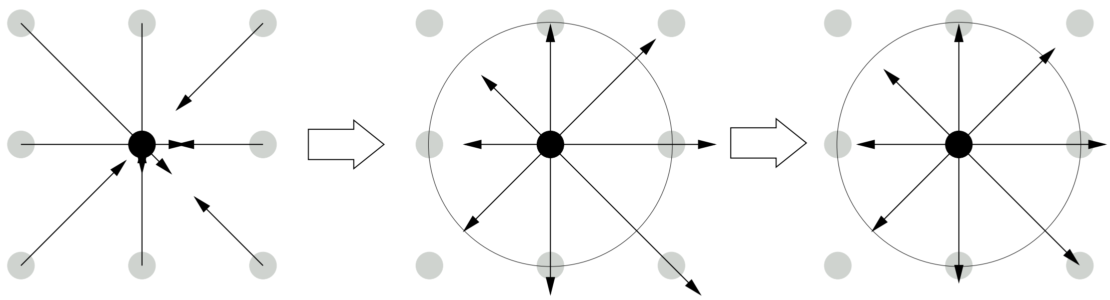


Figure 2.1: The stream and collide steps performed at one lattice site.

the remainder of this chapter. The lattice itself consists of an array of gridsite instances. As is the case in the Jacobi algorithm, two arrays of gridsite instances must be created in order to calculate the new time-step $n+1$ from values from the old time-step n.

2.2.2 Implementing the Stream Step

The stream step, as shown in equation 2.3 is extremely simple to implement. It merely involves copying particle distribution function values from surrounding cells to the cell currently being updated. Using figure 2.1 as a guide, the stream step for a gridsite at row i and column j would be implemented as shown in Algorithm 2.2. To ensure that the stream operation can be performed as it is implemented in Algorithm 2.2 without causing memory access errors, additional sites, dubbed *ghost sites*, are introduced at the top, bottom, left and right of the grid. These sites are never collided, and exist merely to simplify the structure of the algorithm. The ghost nodes are not shown in any figures, yet are always assumed to exist. Due to the use of *boundary sites*, described below, surrounding the entire fluid in the lid driven cavity, information from the ghost sites never reaches the interior fluid cells, and therefore does not influence the fluid dynamics simulation.

Algorithm 2.2 Stream Step

```
dst[i,j].N = src[i-1,j].N
dst[i,j].E = src[i,j-1].E
dst[i,j].S = src[i+1,j].S
dst[i,j].W = src[i,j-1].W
dst[i,j].NE = src[i-1,j-1].NE
dst[i,j].SE = src[i+1,j-1].SE
dst[i,j].SW = src[i+1,j+1].SW
dst[i,j].NW = src[i-1,j+1].NW
```

2.2.3 Implementing the Collide Step

Using a little algebra, the equation describing the collide step (2.4) can be simplified into

$$f_\alpha(\mathbf{x}, t+\Delta t) = (1 - \frac{\Delta t}{\tau})\tilde{f}_\alpha(\mathbf{x}, t+\Delta t) + \frac{\Delta t}{\tau} f_\alpha^{(0)}(\mathbf{x}, t+\Delta t) \tag{2.5}$$

Since $f_\alpha(\mathbf{x}, t+\Delta t)$ and $\tilde{f}_\alpha(\mathbf{x}, t+\Delta t)$ are merely the nine particle distributions, all that remains is the implementation of $f_\alpha^{(0)}(\mathbf{x}, t+\Delta t)$ using equations 1.3, 1.5, 1.6, 1.7, and 1.8. In the D2Q9 model, the momentum density is a two dimensional vector and is therefore, for ease of implementation, separated into two scalars u_x and u_y. Since the values ρ, u_x and u_y required for the calculation of distribution functions as well visualization, their calculations were initially implemented in member functions, similar to those shown in Procedures 2.3, 2.4, and 2.5.

To implement $f_\alpha^{(0)}(\mathbf{x}, t+\Delta t)$, the units are usually chosen such that the distance to the nearest neighbors, Δx and the time step, Δt are unity, so that c=1 and the lattice vector e_α is numerically equal to the velocity of the particles moving in direction α. Equation 1.5 now becomes:

$$[h!] f_\alpha^{(eq)} = \omega_\alpha \rho \left[1 + 3\mathbf{e}_\alpha \cdot \mathbf{u} + 9(\mathbf{e}_\alpha \cdot \mathbf{u})^2 + \frac{3}{2}(\mathbf{u} \cdot \mathbf{u}) \right] \tag{2.6}$$

The complete collide step can now be implemented as shown in Algorithm 2.6.

Procedure 2.3 Calculating ρ

```
Procedure Rho() returns double
double rho;
rho = C + N + E + S + W + NE + SE + SW + NE;
return rho;
```

Procedure 2.4 Calculating u_x

```
Procedure U_x() returns double
double u_x;
u_x = (E + NE + SE - W - NW - SW)/Rho();
return u_x;
```

Procedure 2.5 Calculating u_y

```
Procedure U_y() returns double
double u_y;
u_y = (N + NE + NW - S - SE - SW)/Rho();
return u_y;
```

Since the collide step is completely local to one site, it was implemented as a member function of the gridsite class. This implementation is not quite complete however. To implement a lid driven cavity two additional types of lattice sites must be implemented: a *boundary site* and an *acceleration site*. The boundary site simulates an impassible obstacle, such as a wall, and the acceleration site is responsible for simulating the lid being drawn across the cavity. Both are straightforward to implement. The boundary site simply reverses the direction of each particle distribution value of a site, and the acceleration cells modifies the density and momentum density values. In order to mark a site as a regular fluid, boundary, or acceleration site, an integer type flag is introduced to the gridsite class. The modified collide step, which handles each of the three types of cells is shown in Algorithm 2.7

The completed algorithm now looks like Algorithm 2.8. To initialize the values of the particle distribution function, equation 1.8 is used, no implementation is shown. The parallels to the Jacobi algorithm are now easily seen. The loop structure mirrors that of Jacobi exactly; two inner loops to traverse the lattice, and one outer loop to control iterations, or, in this case, the time-steps performed. In addition, a stencil type access is performed by both algorithms in order to calculate new values. One very obvious difference between the two is the amount of data required to perform the algorithm. A simple implementation of Jacobi requires only one double precision floating point number for each grid site, altogether 8 bytes of data per site. In the case of the LBM, nine double precision values and a single integer value is required for each site. Additionally, the compiler most likely introduces some padding data in order to properly align the double precision values. In any case, at least 72 bytes are required for the double values, and 2 bytes for a 16 bit integer, altogether at least 74 bytes per site.

2.2.4 Performance

The code performance was tested on four different architectures. the details on each can be found in appendix A. In order to perform profiling, PAPI was used on the AMD Athlon machine, and DCPI was used on the Alpha A21164, see [BDG+00] and [ABD+97]. The standard unit for measuring the performance of a LBM implementation are MLSUPS, which stands for Mega Lattice Site Updates Per Second. Another standard unit of performance measurement, generally used to benchmark various numerical code, is the MFLOPS, or Mega FLoating point OPeration per Second.

An important factor to take in account during performance measurements of the LBM code is the influence of the boundary sites. First of all, they are computationally much less intensive than the regular fluid sites, secondly, they do not perform any floating point operations. As a result, very small grids have a relatively high percentage of boundary cells (a 10^2 grid consists of 36%

Algorithm 2.6 Collide Step

double rho = Rho();
double u_x = U_x();
double u_y = U_y();
double u_sqr_trm = $\frac{3.0}{2.0} \cdot$ (u_x $\cdot$ u_x + u_y $\cdot$ u_y);

C = $(1.0 - \frac{1.0}{\tau})$C + $\frac{1.0}{\tau}(\frac{4.0}{9.0}$rho (1.0 - u_sqr_trm));
N = $(1.0 - \frac{1.0}{\tau})$N + $\frac{1.0}{\tau}(\frac{1.0}{9.0}$rho (1.0 + 3.0$\cdot$u_y + $\frac{9.0}{2.0}$(u_y $\cdot$ u_y) - u_sqr_trm));
S = $(1.0 - \frac{1.0}{\tau})$S + $\frac{1.0}{\tau}(\frac{1.0}{9.0}$rho (1.0 - 3.0$\cdot$u_y + $\frac{9.0}{2.0}$(u_y $\cdot$ u_y) - u_sqr_trm));
E = $(1.0 - \frac{1.0}{\tau})$E + $\frac{1.0}{\tau}(\frac{1.0}{9.0}$rho (1.0 + 3.0$\cdot$u_x + $\frac{9.0}{2.0}$(u_x $\cdot$ u_x) - u_sqr_trm));
W = $(1.0 - \frac{1.0}{\tau})$W + $\frac{1.0}{\tau}(\frac{1.0}{9.0}$rho (1.0 - 3.0$\cdot$u_x + $\frac{9.0}{2.0}$(u_x $\cdot$ u_x) - u_sqr_trm));
NE = $(1.0 - \frac{1.0}{\tau})$NE + $\frac{1.0}{\tau}(\frac{1.0}{36.0}$rho (1.0 + 3.0(u_x+u_y) + $\frac{9.0}{2.0}$(u_x+u_y)(u_x+u_y) - u_sqr_trm));
SE = $(1.0 - \frac{1.0}{\tau})$NE + $\frac{1.0}{\tau}(\frac{1.0}{36.0}$rho (1.0 + 3.0(u_x-u_y) + $\frac{9.0}{2.0}$(u_x-u_y)(u_x-u_y) - u_sqr_trm));
SW = $(1.0 - \frac{1.0}{\tau})$NE + $\frac{1.0}{\tau}(\frac{1.0}{36.0}$rho (1.0 + 3.0(-u_x-u_y) + $\frac{9.0}{2.0}$(-u_x-u_y)(-u_x-u_y) - u_sqr_trm));
NW = $(1.0 - \frac{1.0}{\tau})$NE + $\frac{1.0}{\tau}(\frac{1.0}{36.0}$rho (1.0 + 3.0(-u_x+u_y) + $\frac{9.0}{2.0}$(-u_x+u_y)(-u_x+u_y)- u_sqr_trm));

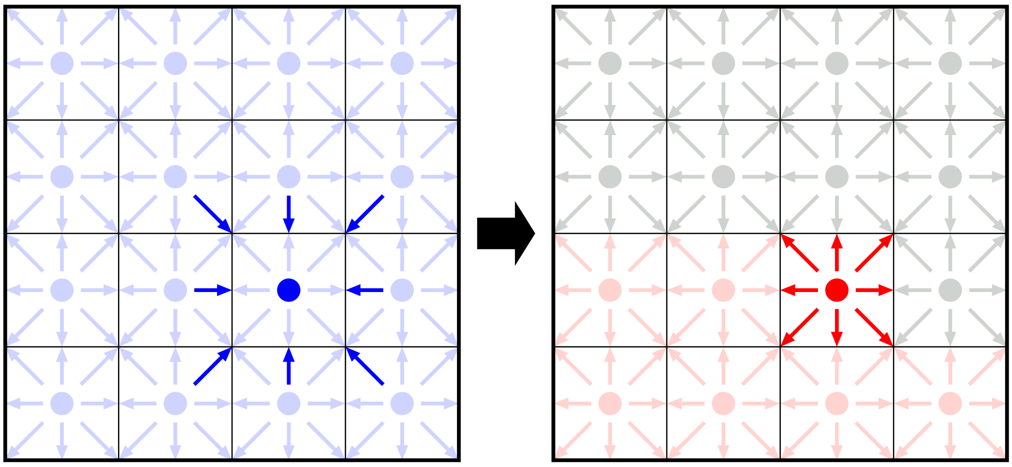


Figure 2.2: Representation of the LBM updating one site. The left grid is the source grid, the darker arrows are the particle distribution function values being read. The right grid is the destination grid, the dark arrows are the particle distribution function values after colliding them.

boundary cells, and 64% fluid cells, at a grids 39^2 or greater have less than 10% boundary sites). This will give the smaller grids unproportionally high performance compared to larger grids. To test the number of MFLOPS required per MLSUPS shown in table 2.1, Algorithm 2.8 was run without boundary sites. All performance measurements in this paper are run with boundary sites, except when otherwise noted. From table 2.1 it can be seen that 1 $MFLOPS \approx 115\ MLSUPS$.

2.3 Non-Cache Optimizations

In light of the poor performance achieved by Algorithm 2.8, some basic optimizations were performed that were not aimed at improving cache usage, but to improve the runtime of the core LBM operations. The optimized resulting algorithm is shown in Algorithm 2.9.

The first optimization stems from the fact that initial profiling using the *gprof* command showed that the compilers were not able to inline the collide function call performed at each lattice site. This causes drops in performance for two reasons. First, there is a certain amount of overhead involved in calling a function. Since the collide step must be called at every grid site for every time-step, this causes a significant amount of time to be spent just on function call overhead. Second, there are many unnecessary loads and stores of data. In the stream step, all nine surrounding particle

size	steps	MLSU	runtime	MLSUPS	MFLOPS
10	300000	30	16.2	1.85	207
50	15000	37.5	26.0	1.40	164
100	2000	20	14.4	1.38	157
300	200	18	23.9	0.75	87

Table 2.1: MLSUPS and MFLOPS as measured on the Alpha A21164 using DCPI.

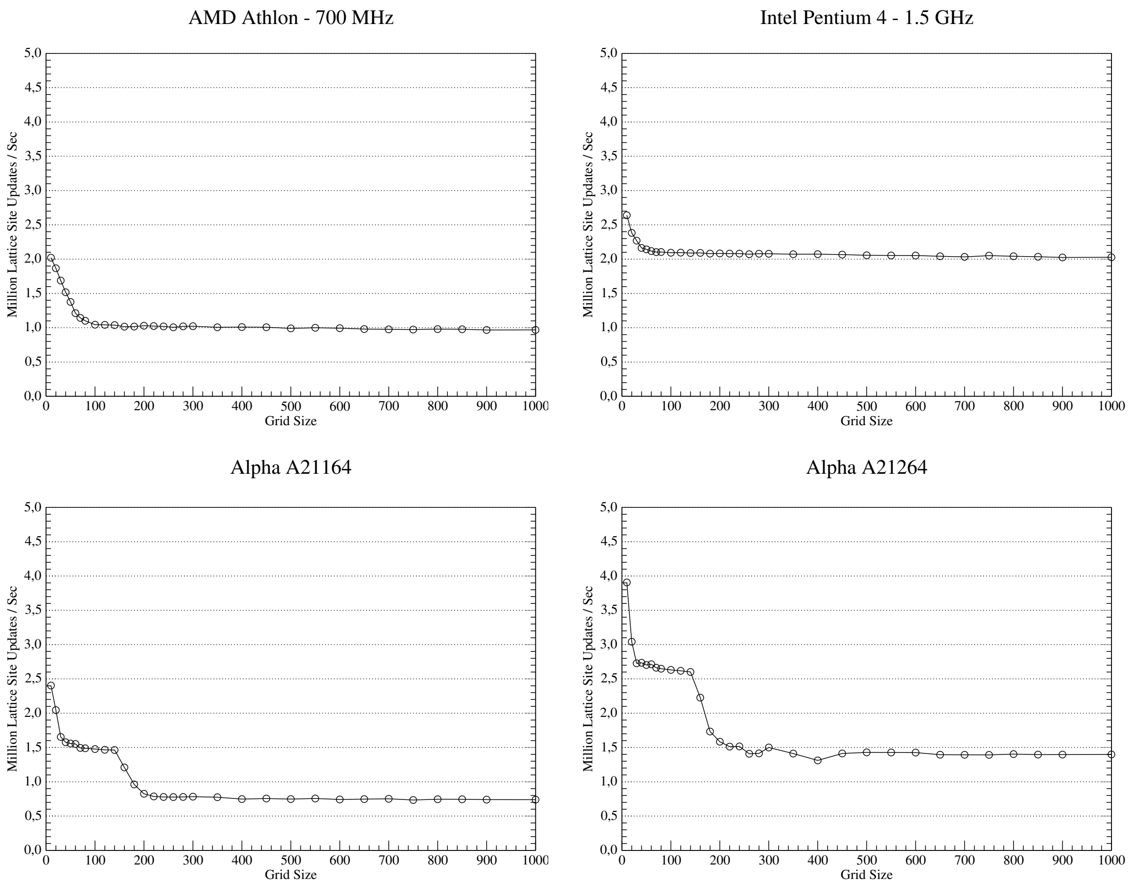


Figure 2.3: The performance of Algorithm 2.8 on four different architectures.

Algorithm 2.7 Collide Step for a Single Site

1: **if** type == "boundary" **then**
2: swap(N, S)
3: swap(E, W)
4: swap(NE, SW)
5: swap(NW, SE)
6: **FINISHED**
7: **end if**
8: double rho;
9: double u_x;
10: double u_y;
11: **if** type == "acceleration" **then**
12: rho = 1.0;
13: u_x = 0.1;
14: u_y = 0.0;
15: **else**
16: rho = Rho();
17: u_x = U_x();
18: u_y = U_y();
19: **end if**
20: double u_sqr_trm = $\frac{3.0}{2}$ · (u_x · u_x + u_y · u_y);
21:
22: C = (1.0 − $\frac{1.0}{\tau}$)C + $\frac{1.0}{\tau}$($\frac{4.0}{9.0}$rho (1.0 - u_sqr_trm));
23: N = (1.0 − $\frac{1.0}{\tau}$)N + $\frac{1.0}{\tau}$($\frac{1.0}{9.0}$rho (1.0 + 3.0·u_y + $\frac{9.0}{2.0}$(u_y · u_y) - u_sqr_trm));
24: S = (1.0 − $\frac{1.0}{\tau}$)S + $\frac{1.0}{\tau}$($\frac{1.0}{9.0}$rho (1.0 - 3.0·u_y + $\frac{9.0}{2.0}$(u_y · u_y) - u_sqr_trm));
25: E = (1.0 − $\frac{1.0}{\tau}$)E + $\frac{1.0}{\tau}$($\frac{1.0}{9.0}$rho (1.0 + 3.0·u_x + $\frac{9.0}{2.0}$(u_x · u_x) - u_sqr_trm));
26: W = (1.0 − $\frac{1.0}{\tau}$)W + $\frac{1.0}{\tau}$($\frac{1.0}{9.0}$rho (1.0 - 3.0·u_x + $\frac{9.0}{2.0}$(u_x · u_x) - u_sqr_trm));
27: NE = (1.0 − $\frac{1.0}{\tau}$)NE + $\frac{1.0}{\tau}$($\frac{1.0}{36.0}$rho (1.0 + 3.0(u_x+u_y) + $\frac{9.0}{2.0}$(u_x+u_y)(u_x+u_y) - u_sqr_trm));
28: SE = (1.0 − $\frac{1.0}{\tau}$)NE + $\frac{1.0}{\tau}$($\frac{1.0}{36.0}$rho (1.0 + 3.0(u_x-u_y) + $\frac{9.0}{2.0}$(u_x-u_y)(u_x-u_y) - u_sqr_trm));
29: SW = (1.0 − $\frac{1.0}{\tau}$)NE + $\frac{1.0}{\tau}$($\frac{1.0}{36.0}$rho (1.0 + 3.0(-u_x-u_y) + $\frac{9.0}{2.0}$(-u_x-u_y)(-u_x-u_y) - u_sqr_trm));
30: NW = (1.0 − $\frac{1.0}{\tau}$)NE + $\frac{1.0}{\tau}$($\frac{1.0}{36.0}$rho (1.0 + 3.0(-u_x+u_y) + $\frac{9.0}{2.0}$(-u_x+u_y)(-u_x+u_y)- u_sqr_trm));
31: **FINISHED**

distribution values are loaded and stored in the current site. In the collide step, the values that were just stored must be loaded again, processed, and stored. By combining the stream and collide step, it is possible to save a function call, and to reduce the number of loads and stores that must be performed. In order to implement this optimization, the majority of the C++ object oriented overhead was discarded. The two grids were implemented as two one-dimensional arrays of double precision values, where ten doubles were allocated for each site, nine for the particle distribution functions, and one to determine the type of the site. Table 2.2 shows the number of loads and stores that are performed by both versions.

The next optimization was the explicit inlining of the U_x(), U_y(), and Rho() function calls in lines 16 to 18 in Algorithm 2.7, in order to avoid performing unnecessary floating point additions and to further reduce function call overhead. This is easily implemented; the resulting code is shown in Algorithm 2.9 on lines 31 to 35.

		Version 1		Version 2	
size	steps	load	store	load	store
10	300000	$9.8 \cdot 10^8$	$6.5 \cdot 10^8$	$4.6 \cdot 10^8$	$2.5 \cdot 10^8$
50	15000	$1.36 \cdot 10^9$	$8.7 \cdot 10^8$	$6.8 \cdot 10^8$	$3.3 \cdot 10^8$
100	2000	$7.3 \cdot 10^8$	$4.7 \cdot 10^8$	$3.7 \cdot 10^8$	$1.7 \cdot 10^8$
300	200	$6.7 \cdot 10^8$	$4.3 \cdot 10^8$	$3.4 \cdot 10^8$	$1.6 \cdot 10^8$

Table 2.2: The number of load and store instructions performed on the A21164. Notable is that after the stream and collide steps were combined, the number of loads and stores went down by more than 50%.

Algorithm 2.8 Initial LBM Implementation

```
gridsite src[n, m]
gridsite dst[n, m]
for i = 1 to i = n − 1 by 1 do
    for j = 1 to j = m − 1 by 1 do
        src.init_values();
    end for
end for
for ts = 0 to ts = MAX_TIMESTEP by 1 do
    for i = 1 to i = n − 1 by 1 do
        for j = 1 to j = m − 1 by 1 do
            dst[i,j].N = src[i-1,j].N;
            dst[i,j].E = src[i,j-1].E;
            dst[i,j].S = src[i+1,j].S;
            dst[i,j].W = src[i,j-1].W;
            dst[i,j].NE = src[i-1,j-1].NE;
            dst[i,j].SE = src[i+1,j-1].SE;
            dst[i,j].SW = src[i+1,j+1].SW;
            dst[i,j].NW = src[i-1,j+1].NW;
            dst[i,j].collide();
        end for
    end for
    swap( src, dst )
end for
```

the third optimization, also aimed at reducing the number of floating point operations, is the pre-multiplication of the weighting factor and the viscosity factor into *rho_*, and the reordering of the particle distribution function calculations in order to avoid floating point divisions.

The terms $\frac{9.0}{2.0}(\pm u\_x \pm u\_y)(\pm u\_x \pm u\_y)$ used to calculate the four diagonal directions can be split into $(3.0(\pm u\_x \pm u\_y)) \cdot (3.0(\pm u\_x \pm u\_y)) \cdot \frac{1.0}{2.0}$. The term $(3.0(u\_x + u\_y))$ could then be pre-calculated and saved in a temporary variable. This was implemented, but did in fact incur a performance penalty. Simply splitting the larger term, but not pre-calculating $(3.0(u\_x + u\_y))$ proved to be the fastest code. The compiler must recognize this term as a common sub-expression and optimize it automatically. Allocating a new variable to perform the pre-calculation simply adds unnecessary instructions. The optimizations can be seen in lines 39 to 42 in Algorithm 2.9.

The performance of this implementation, hereafter referred to as the Non-Cache Optimized (NCO) implementation, is shown in figure 2.4. The performance on all architectures has improved noticeably. The Pentium 4 especially profits from the optimizations described above. Curious is the drop in performance at a grid size of 550^2 however, which was persistant despite repeated testing. Due to the lack of profiling tools available on the P4, no reason for this behavior could be determined.

Looking at the code there are two more obvious operations that can result in a significant performance penalty. One is the **IF** command used twice, the other is the division by *rho_* that occurs twice. To test the performance cost Algorithm 2.9 was run without using boundary cells, and alternately removing all **IF** commands and all division operations. Of course, these changes cause the LBM to no longer produce physically accurate results, and are done for testing purposes only. Figure 2.5 illustrates the performance penalty incurred by these two operations. Although the penalty for the **IF** command is not too severe, the division operations carry a heavy penalty. This is due to the complexity of the division operation at CPU level. It takes a long time to execute, and the execution pipeline stalls while waiting for the result of the division command. This might be avoided by executing the division early enough to avoid stalling the pipeline. An even better solution is a new variant of the LBM, which is implemented without a division operation. For details see [HL97]. Unfortunately this was not known until this thesis was almost concluded, and was therefore not incorporated into the LBM used here. All cache optimizations discussed in the next chapters would be applicable to this “division free” version as well.

Additionally, figure 2.5 demonstrates what grid sizes theoretically fit into the caches of the different architectures. Interesting is the drop in performance seen at each cache boundary on the different processors. This correlation between performance drops and cache sizes is the motivation for carrying out the cache optimizations described in the next chapters. Details on the cache properties of the various architectures can be found in Appendix A.

Algorithm 2.9 Non Cache Optimized LBM Implementation

```
// Allocate memory to process an NxM sized grid. Note the two
// additional rows and columns for ghost nodes, described above.
site src[n + 2, m + 2]
site dst[n + 2, m + 2]
for i = 1 to i = n by 1 do
   for j = 1 to j = m by 1 do
      src.init_values();
   end for
end for
for ts = 0 to ts = MAX_TIMESTEP by 1 do
   for i = 1 to i = n by 1 do
      for j = 1 to j = m by 1 do
         if dst[i,j].type == "boundary" then
            dst[i,j].S = src[i-1,j].N;
            dst[i,j].W = src[i,j-1].E;
            dst[i,j].N = src[i+1,j].S;
            dst[i,j].E = src[i,j+1].W;
            dst[i,j].SW = src[i-1,j-1].NE;
            dst[i,j].NW = src[i+1,j-1].SE;
            dst[i,j].NE = src[i+1,j+1].SW;
            dst[i,j].SE = src[i-1,j+1].NW;
         else
            double rho;
            double u_x;
            double u_y;
            if dst[i,j].type == "acceleration" then
               rho = 1.0;
               u_x = 0.1;
               u_y = 0.0;
            else
               u_x = src[i,j-1].E + src[i-1,j-1].NE + src[i+1,j-1].SE;
               u_y = src[i-1,j].N + src[i-1,j-1].NE + src[i-1,j+1].NW;
               rho = src[i,j-1].W + src[i+1,j].S + src[i+1,j+1].SW + src[i,j].C - src[i-1,j-1].NE + u_x + u_y;
               u_x = ( u_x - src[i,j+1].W - src[i-1,j+1].NW - src[i+1,j+1].SW ) / rho;
               u_y = ( u_y - src[i+1,j].S - src[i+1,j-1].SE - src[i+1,j+1].SW ) / rho;
            end if
            double u_sqr_trm = (3.0/2.0)· ( u_x · u_x + u_y · u_y );
            rho = (1.0/τ) · (1.0/36.0)rho;
            dst[i,j].NE = (1.0 − 1.0/τ)src[i-1,j-1].NE + rho( 1.0 + 3.0(u_x+u_y) + 3.0(u_x+u_y)·3.0(u_x+u_y)·0.5 - u_sqr_trm );
            dst[i,j].SE = (1.0 − 1.0/τ)src[i+1,j-1].SE + rho( 1.0 + 3.0(u_x-u_y) + 3.0(u_x-u_y)·3.0(u_x-u_y)·0.5 - u_sqr_trm );
            dst[i,j].SW = (1.0 − 1.0/τ)src[i+1,j+1].SW + rho( 1.0 + 3.0(-u_x-u_y) + 3.0(-u_x-u_y)·3.0(-u_x-u_y)·0.5 - u_sqr_trm );
            dst[i,j].NW = (1.0 − 1.0/τ)src[i-1,j+1]].NW + rho( 1.0 + 3.0(-u_x+u_y) + 3.0(-u_x+u_y)·3.0(-u_x+u_y)·0.5 - u_sqr_trm );
            rho=4.0·rho;
            dst[i,j].N = (1.0 − 1.0/τ)src[i-1,j].N + rho( 1.0 + 3.0·u_y + (9.0/2.0)(u_y · u_y) - u_sqr_trm );
            dst[i,j].S = (1.0 − 1.0/τ)src[i+1,j].S + rho( 1.0 - 3.0·u_y + (9.0/2.0)(u_y · u_y) - u_sqr_trm );
            dst[i,j].E = (1.0 − 1.0/τ)src[i,j-1].E + rho( 1.0 + 3.0·u_x + (9.0/2.0)(u_x · u_x) - u_sqr_trm );
            dst[i,j].W = (1.0 − 1.0/τ)src[i,j-1].W + rho( 1.0 - 3.0·u_x + (9.0/2.0)(u_x · u_x) - u_sqr_trm );
            dst[i,j].C = (1.0 − 1.0/τ)src[i,j].C + 4.0·rho( 1.0 - u_sqr_trm );
         end if
      end for
   end for
   swap( src, dst )
end for
```

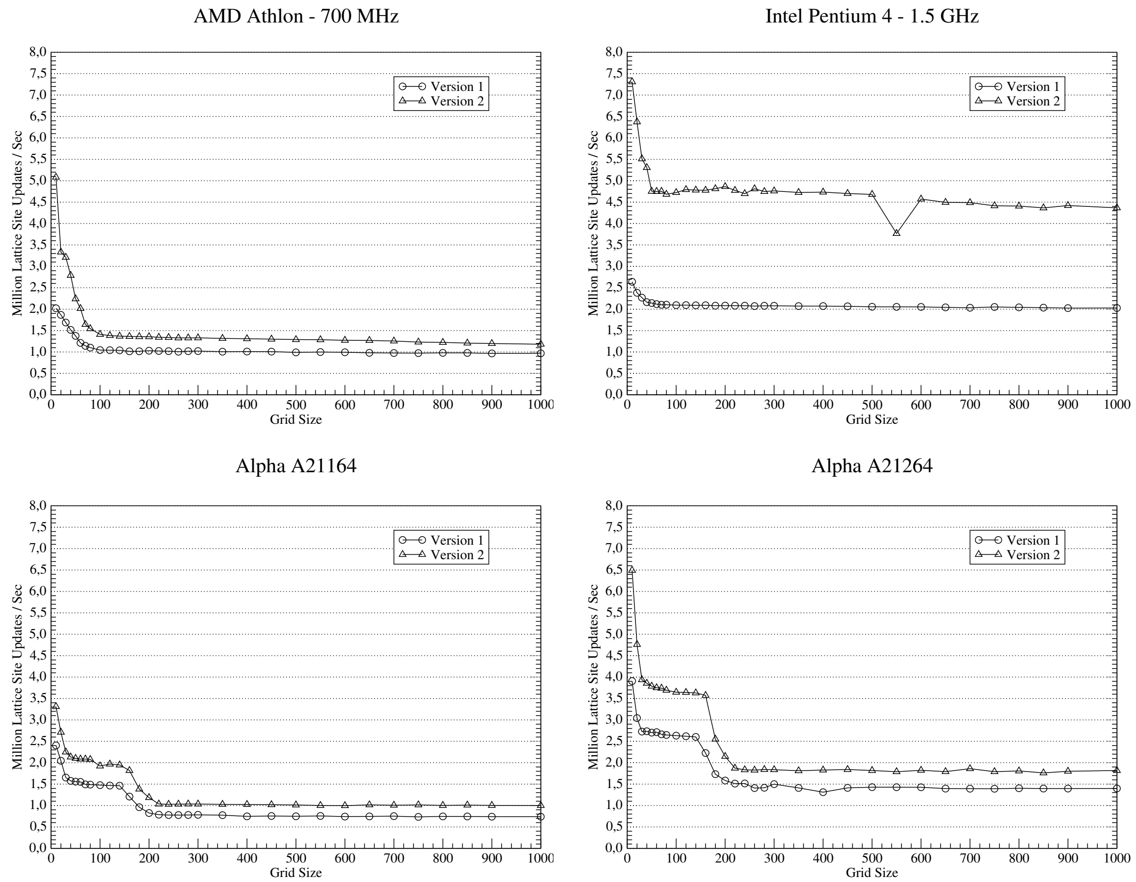


Figure 2.4: The performance of Algorithm 2.9 as opposed to Algorithm 2.8 on four different architectures.

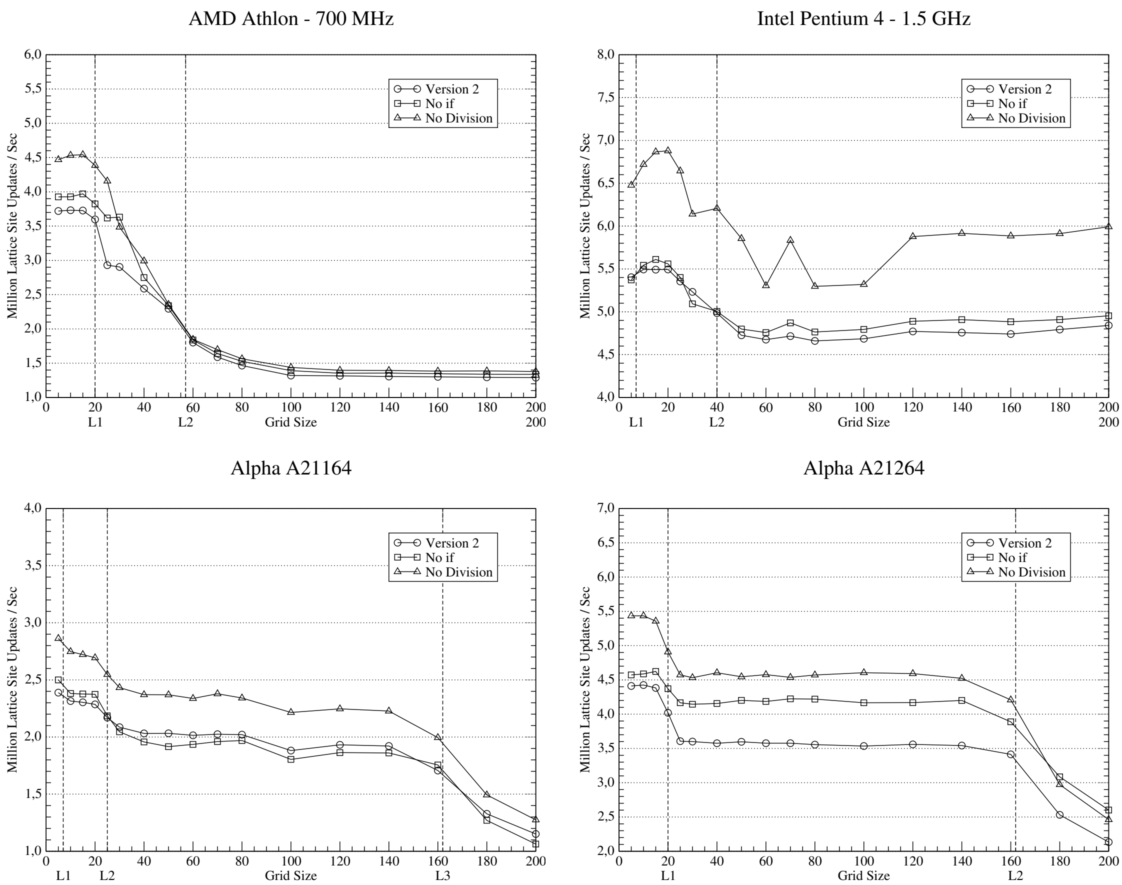


Figure 2.5: The performance penalty for performing different kinds of operations. The dashed vertical lines represent cache boundaries.

Chapter 3

Cache Optimization Overview

This chapter will provide a brief overview of the purpose and properties of cache, and how it affects CPU performance. In addition, several code optimizations that were used to improve the cache usage of the LBM will be presented. It should be noted that there are additional optimizations for cache usage beyond those mentioned here, a more complete and detailed presentation of cache techniques can be found in [Sto02], [AK02] and [GH01] for example.

3.1 The Purpose of Cache

In the last two decades innovations in chip process design and technology such as multi-scalar architectures and pipelining have greatly increased the theoretical peak processing speed of modern CPUs. At the same time, the memory subsystem has not advanced fast enough to be able to provide the CPU with data at a rate allowing it to run at full capacity, forcing the CPU to much lower performance levels.

The purpose of cache is to provide a small, extremely fast data buffer between the CPU and the memory subsystem in order to quickly provide the CPU with data as often and quickly as possible. When requested data is loaded from cache, known as a *cache hit*, the CPU spends fewer cycles waiting for data than when a *cache miss* occurs and data must be loaded from much slower memory. The cache is located directly on the processor die and must be kept small to allow it to run at a high clock speed and to keep production costs low. Oftentimes, this cache, known as level 1 cache (L1), is augmented by a larger, albeit slower and therefore less expensive, L2 cache, usually located off-chip. Occasionally, hardware manufacturers also include an even larger L3 cache to further reduce memory accesses. This system of caches, the CPU registers, and the main memory are altogether referred to as the memory hierarchy, shown in Figure 3.1. A request for data propagates down through each level in the hierarchy until it is found. Obviously, data found in higher levels will require much less time to arrive at the CPU than data from lower levels.

3.2 Cache Properties

In principle, caches are able to provide requested data by exploiting reference locality, of which there are the following two types: spatial locality and temporal locality. Spatial locality assumes that when a byte of data is requested by the CPU, subsequent requests will access neighboring bytes as well. Temporal locality assumes that when one byte is accessed, it will be accessed again in the near future. Different techniques have been developed to exploit both spatial and temporal locality for different algorithms.

To best exploit spatial locality it would be ideal to load a block of contiguous data the size of the cache itself into the cache whenever a cache miss occurs. Subsequent accesses to neighboring bytes would then result in cache hits. This technique exploits temporal locality poorly however. If the CPU repeatedly accesses data not found in the same contiguous block, each access would incur a cache miss. To best exploit temporal locality it would be best to load each byte separately, thereby minimizing the chance that data will be overwritten. This, however, disregards spatial locality. As

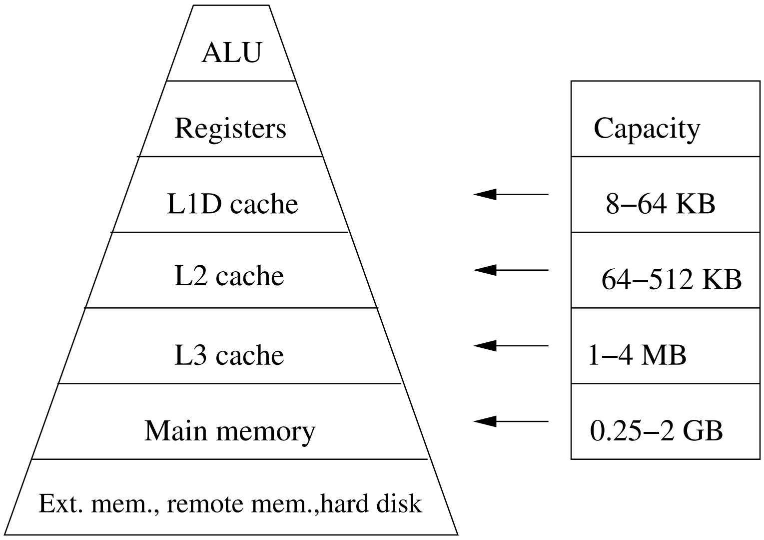


Figure 3.1: Cache hierarchy on modern architectures.

a compromise, data is loaded into cache in small blocks known as *cache lines*, each of which being several bytes in length. Precise cache line lengths vary from architecture to architecture.

The manner in which memory is mapped to cache also varies greatly. the simplest strategy, from a conceptual and implementational standpoint, results in a *direct mapped cache*, shown in Figure 3.2. In this type of cache, each memory location can be mapped to only *single* cache line. This often results in rather poor performance however, since data accessed in multiples of cache size will be mapped to the same cache line, causing a situation known as *cache thrashing*. Another conceptually simple type of cache which would avoid this problem is known as *fully associative cache*. In this type of cache, a memory location can be mapped to any of the available cache lines, as shown in Figure 3.3. Making this type of cache perform well is difficult from an implementational standpoint however. When the CPU requests data, the hardware responsible for managing the cache must quickly determine whether or not the requested data is in cache or not. This is easy to determine in a direct mapped cache, since only a simple modulo operation must be performed. In a fully associative cache however, each cache line must be examined whether it contains the requested data or not. This search would incur a high latency penalty.

To combine the short search times of direct mapped cache with the flexibility of fully associative cache, the cache is divided into n sets, resulting in a *n-way associative cache*. In a 4-way associative cache for example, the cache would be divided into sets of 4 cache lines. Each memory location would be mapped to only a single set but to any of the cache lines within that set. Now, a modulo operation is required to locate which set a byte of data might be located in, followed by a search of a very small number of cache lines. It should be noted, that all associative caches use some sort of algorithm, usually a type of least recently used (LRU) algorithm, to determine which cache line will be replaced or *evicted* by new data. Figure 3.4 demonstrates a 4-way set associative cache. See A for information on the line size and associativity of the architectures tested in the course of this thesis. See [AK02] and [GH01] for a more in depth look at the technical details of modern architectures and optimization techniques.

3.3 Cache Optimizations

Cache optimizations typically include transformations of the algorithm or memory layout in order to reduce the number of cache misses occurring during runtime. Cache misses are categorized into the following three types [Wei01]:

- **Compulsory misses** are unavoidable misses which occur when requesting data for the very first time.

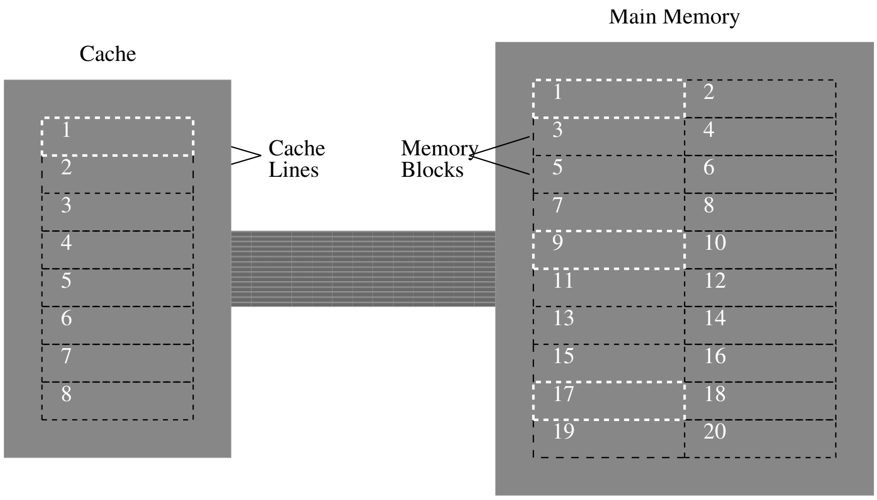


Figure 3.2: Direct mapped cache. The white memory blocks are the only memory locations that can be mapped to the white cache line.

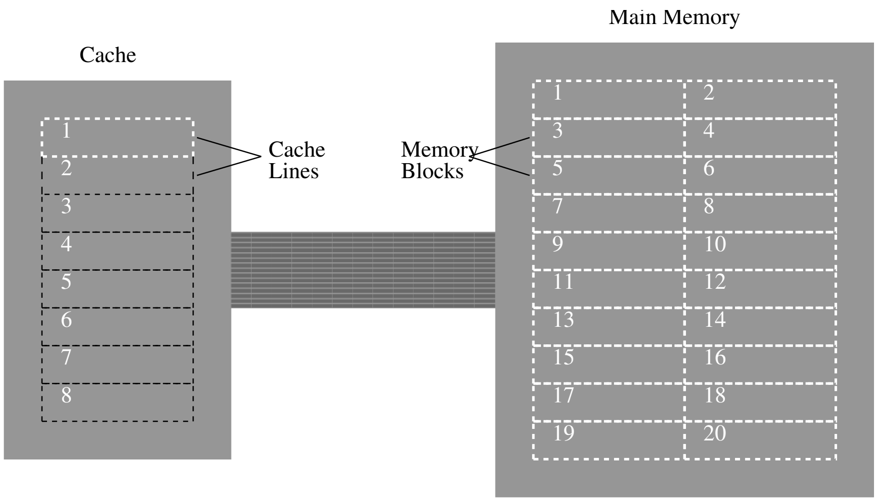


Figure 3.3: Fully associative cache. Any memory block can be mapped to a cache line.

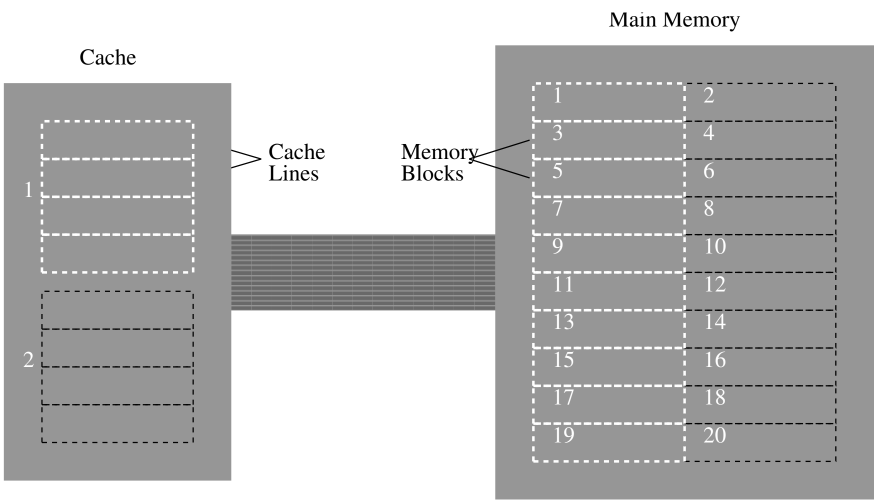


Figure 3.4: Set associative cache, the white memory blocks can be mapped to any of the cache lines in the white set.

- **Capacity misses** are the number of total cache misses minus the number of compulsory misses that would occur in a fully associative cache with the same size, cache line length, and replacement strategy as the cache being examined. These types of misses occur when the data set being worked on is too large to fit into cache.

- **Conflict misses** occur when too much data is mapped to the same cache line. The number of conflict misses equals the number of total misses minus the number of compulsory and capacity misses. Conflict misses lead to cache thrashing as mentioned above.

The optimizations used in the development of a cache optimized LBM are described below.

3.3.1 Array Merging

Merging is a relatively simple data layout optimization that can be used to improve spatial locality between elements in different data structures. In Lattice Boltzmann or Jacobi for example, two separate arrays are used to calculate the new time step or iteration step respectively. When merging these two arrays, the data from each array is interleaved to form one single array as shown in Figure 3.5.

3.3.2 Loop Blocking

Loop blocking is a code transformation designed to break up a large working data set into several smaller sets, thereby greatly reducing the number of capacity misses as well as improving temporal locality [Wei01]. The transformation increases the depth of a loop nest of depth n by splitting each loop into two loops. The outer loop traverses the original iteration space in steps equal to the increment traversed by the inner loop. The resulting loop nest has a depth $(n+1)$ to $(2 \cdot n)$. Blocking n loops in a loop nest will be referred to as n-way blocking in this paper. It should be noted that a dependency analysis must be performed prior to blocking a loop nest, to determine whether blocking will change the semantics of the algorithm. The LBM as well as the Jacobi algorithm have relatively simple dependencies; the calculation of new value or values for a single cell depends on the cells immediately surrounding it. See Figure 3.1 for a code example of 3-way blocking and Figure 3.6 for an illustration. Blocking of the time loop is not visible in the figure. Blocking in time would result in each of the blocks being processed several times before processing the next block. In the optimization of the LBM 1-way and 3-way blocking was used and will be detailed in Chapter 4 and Chapter 5.

3.3.3 Padding

Although blocking can drastically reduce the number of capacity misses, it does have its drawbacks. If the row of a 2D array is a multiple of the cache size, the stencil accesses can incur severe amounts of conflict misses, since the elements accessed in each row will map to the same cache line [RT98]. Figure 3.7 illustrates this problem further. As will be shown in later chapters, even higher amounts of set associativity will not alleviate this problem when performing blocking. In order to eliminate the inflict misses and regain spatial reuse, *padding* can be used to change the data layout in memory

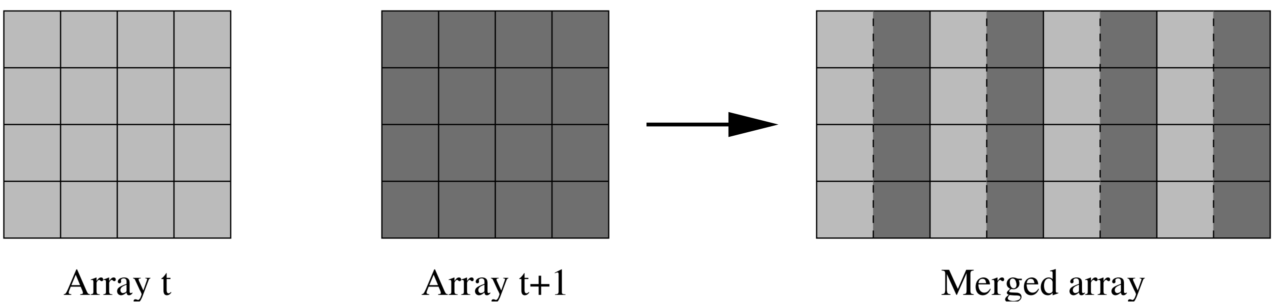


Figure 3.5: Merging two separate arrays, which contain data for time-steps t and $t+1$ into a single array containing both time-steps.

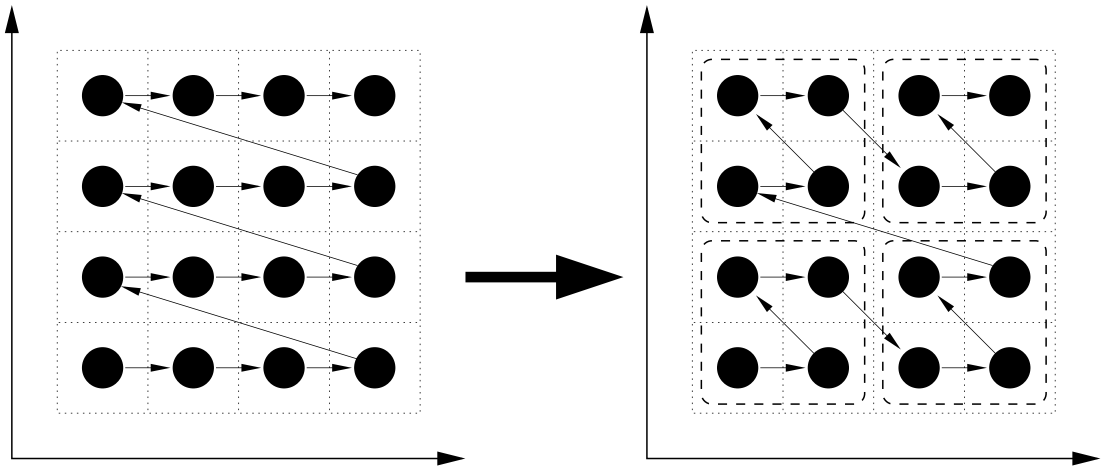


Figure 3.6: Example of 3-way blocking. The grid on the left has been blocked in both the x and y direction, blocking in time is not visible.

Algorithm 3.1 Loop blocking

```
// Original code:
for t = 0 to MAX do
    for i = 1 to m do
        for j = 1 to n do
            // Calculation
            // involving a[i,j]...
        end for
    end for
end for
```

```
// Loop blocked code:
for tt = 1 to MAX by T do
    for ii = 1 to m by B do
        for jj = 1 to n by B do
            for t = 1 to T do
                for i = ii to min(ii + B − 1, m) do
                    for j = jj to min(jj + B − 1, n) do
                        // Calculation involving a[i,j]...
                    end for
                end for
            end for
        end for
    end for
end for
```

so that elements in different rows will no longer map to the same cache line. Introducing a pad at the end of each row in Figure 3.7 would cause the data to be shifted so that stencil accesses would no longer cause conflict misses. See [WL91] and [LRW91] for more information on blocking and padding.

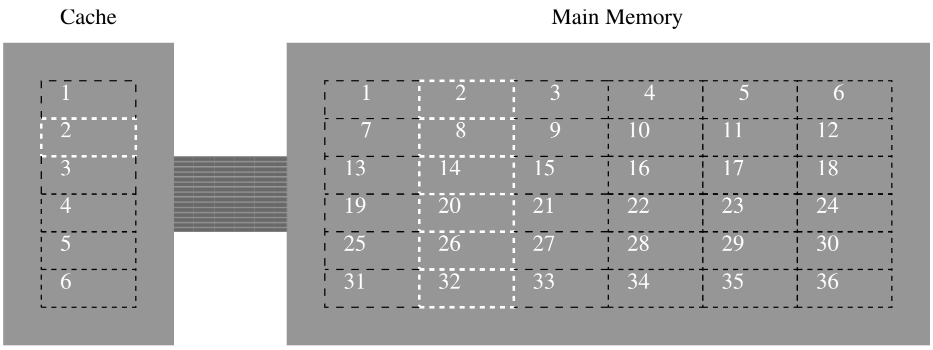


Figure 3.7: Cache thrashing using a direct mapped cache. One row in memory is the size of the cache, therefore elements in each column are mapped to the same cache line. A stencil access involving elements in several rows would suffer from many conflict misses.

Chapter 4

Cache Optimized Lattice-Boltzmann

This chapter will optimize algorithm 2.9 using the different types of cache optimizations presented in chapter 3.

4.1 Merging

As explained in section 3.3.1, the merging technique is generally used to improve spatial locality by loading elements from previously separate grids into one cache line, thereby avoiding the memory latency incurred when reading each element out of non-contiguous memory locations. However, since the data size of a single grid site exceeds the cache line size of all the architectures tested, no improvement in performance is expected. The merging technique is still necessary however, in order to obtain a single contiguous block of memory on which to perform the blocking techniques described in the next sections. Figure 4.1 shows the data layout of the LBM after merging.

Implementing the merged implementation of the LBM is relatively simple. First, instead of allocating two separate arrays of doubles, a single array large enough to store the data contained in the two separate arrays can be allocated. Some memory can be saved by noting that for each pair of cells at location (x, y) representing time-steps n and $n-1$, only a single double must be allocated for the *rest* particle value, and for the type flag. The total storage space required for each lattice site "pair" is 18 double precision values. Second, C++ pointers are used to access different elements in the single array.

Figure 4.2 shows the performance of the merged algorithm as opposed to the non-merged algorithm on the four different architectures. As expected, performance levels did not increase, in fact,

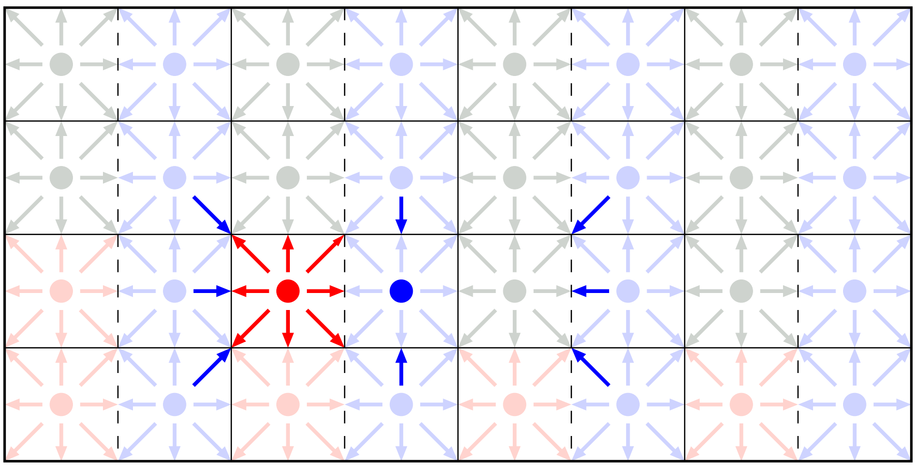


Figure 4.1: Each lattice site is now paired with an previous time-step version of itself.

they decreased on all except the AMD Athlon. The rise or drop in performance is, in general, not very significant on all architectures except on the Intel Pentium 4, which demonstrates a very large drop in performance. Due to the lack of profiling tools installed at the time, no cause could be found. A possibility is that, due to the dual channel Rambus memory on that machine, requests for data in memory locations very distant from one another can be handled by both channels, whereas requests for data lying close together can only be handled by one channel. In effect, it seems that there is some sort of interleaving occurring in the Rambus memory system, which performed well when the source and destination sites were distant in memory, but performed poorly when the sites were very close together, as they are in the merged algorithm.

Figure 4.3 illustrates the number of cache misses incurred per lattice site update, and its effect on the performance of the algorithm. On the Athlon, the correlation is not very distinct, in general it is clear that with an increasing number of cache misses, the performance drops significantly. Obvious is also, that because of the different capacities of the caches, the number of L1 misses increase at a smaller grid size than the L2 cache. A similar graph for the A21164 demonstrates a clear correlation between the number and type of cache miss, and the performance. The L1 and L2 cache overflow at quite small grid sizes, which cause a sharp performance drop right in the beginning. At a grid size of 160^2, the large L3 cache starts to overflow, which coincides exactly with another sharp drop in performance.

4.2 1-Way Blocking

To implement 1-way blocking of the LBM, one of the three loops building the loop nest is split into two separate loops, preferably in such a way as to reuse data that is still residing in cache. In Chapter 2 it was shown that to calculate the particle distribution function for a site at (x, y) for time-step n, the particle distribution functions from time-step $n-1$ of the eight surrounding sites are required. In Figure 4.4, the two bottom rows have been updated to time-step n, whereas the rest of the grid is still in time-step $n-1$. It is obvious that the very bottom row can be updated again, to time-step $n+1$, since the data dependencies required by each site are fulfilled. One can now advance through the entire grid, updating row y to time-step n, then updating row $y-1$ to time-step $n+1$, then advancing in y to update rows $y+1$ and y to time-steps n and $n+1$, respectively, and so on, through the entire grid. After this process completes, the entire grid will be at time-step $n+1$. Figure 4.4 demonstrates this process more closely. In this case, it is the time loop which has been blocked. Algorithm 4.1 displays a code example of 1-way blocked LBM code of arbitrary block size.

4.3 Performance using 1–Way Blocking

Figure 4.5 presents the performance results for the 1-way blocked LBM. There are several interesting results involved. Note that for very small grid sizes, the computationally trivial boundary cells contribute to an unproportionally large increase in performance. Details on all cache properties mentioned below can also be found in Appendix A.

4.3.1 AMD Athlon

The AMD Athlon benefits from 1-way blocking predominately for smaller grid sizes. Initially, the blocked algorithms are much faster since they effectively reuse data still in cache. The 8–row blocked algorithm reuses data the most often, therefore it is initially the fastest of the blocked algorithms. At a grid size of approximately 350^2 and 600^2 the working set of the 8–row blocked and 4–row blocked algorithm, respectively, start to become too large for the 512K level 2 cache. The working set of the 2–row blocked algorithm actually stays in cache up to a grid size of approximately 900^2, but since it does not reuse data in cache often, its performance is overall not very high. Eventually, the performance of all three blocked algorithms drops to that of the unblocked algorithm. Figure 4.6 plots the number of cache misses that occurred with the 4–row blocked algorithm for different grid sizes. Clearly, an increase in cache misses is shifted towards larger grid sizes than it was in the non–blocked algorithm as demonstrated in Figure 4.3. However, at larger grid sizes there are, once again, a very high number of cache misses.

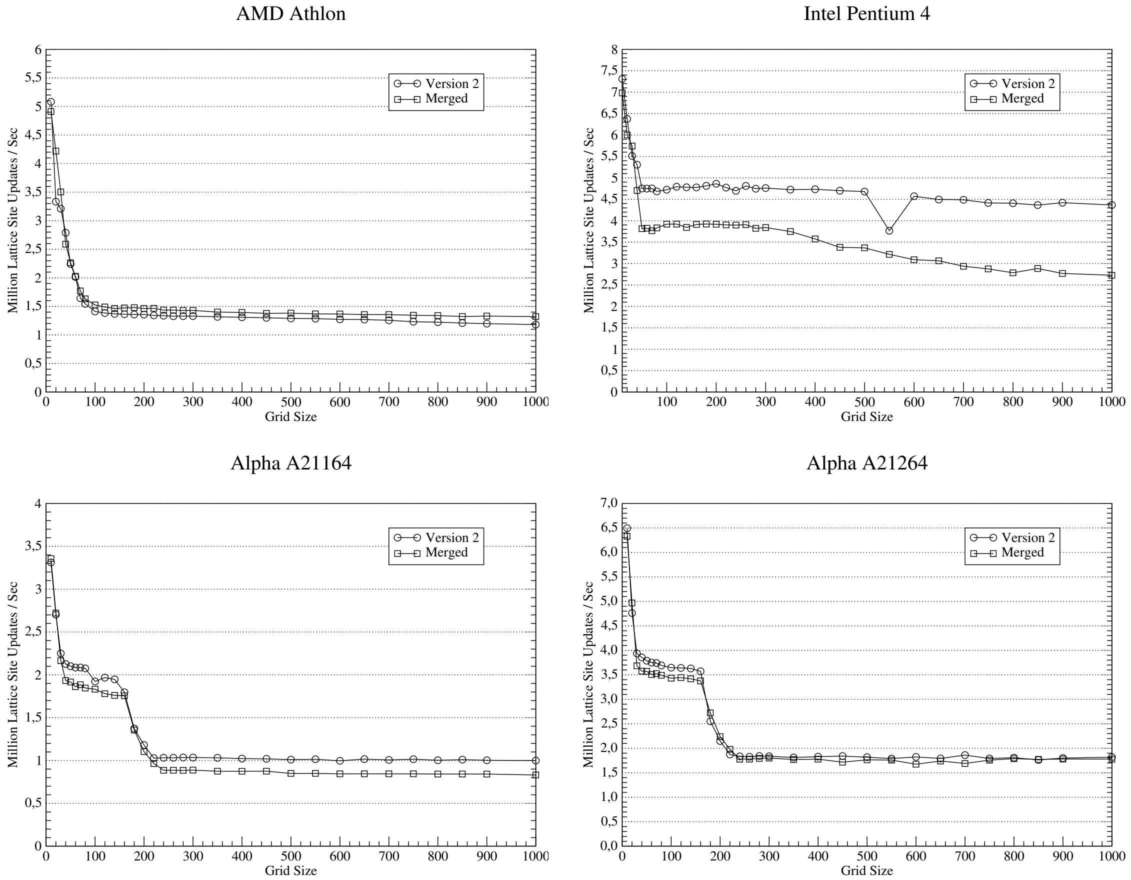


Figure 4.2: Performance of the Merged implementation of the LBM as opposed to the non-merged algorithm.

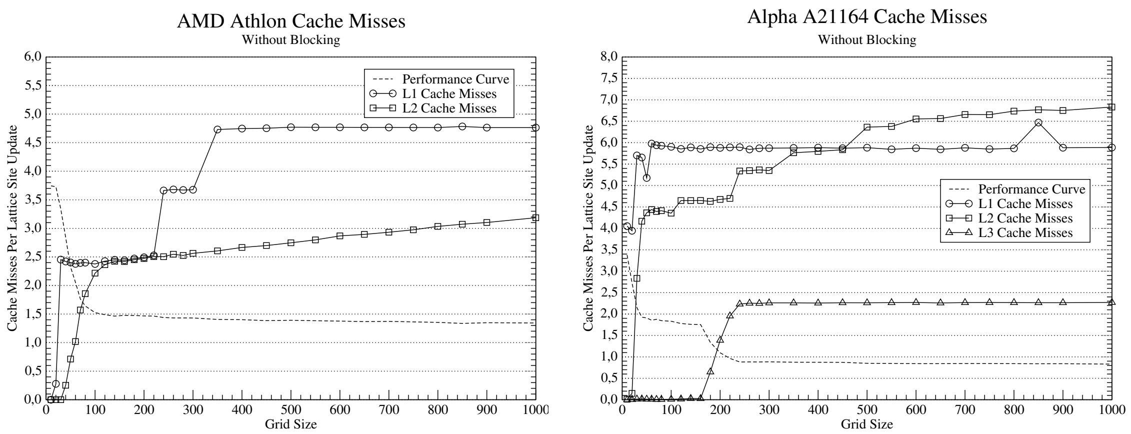


Figure 4.3: Number of cache misses that occur per lattice site update as measured with PAPI on the Athlon and DCPI on the A21164.

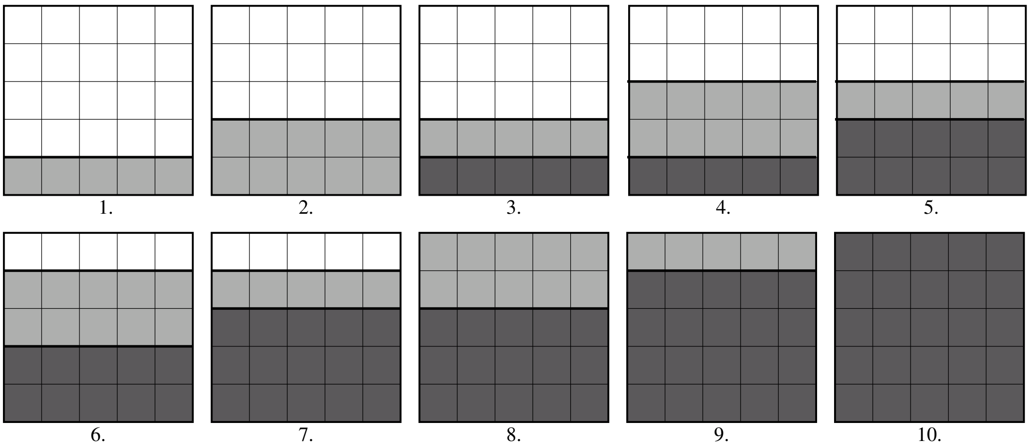


Figure 4.4: Illustration of 1-way blocking of a grid with size (n, m) using a block size of two.

Algorithm 4.1 1-Way Blocked LBM

```
// Original code:
for ts = 1 to MAX_TIME by 1 do
   for i = 1 to m by 1 do
      for j = 1 to n by 1 do
         // Stream_and_Collide(i,j)
      end for
   end for
end for
```

```
// 1-Way blocked code:
for tt = 1 to MAX_TIME by BS do
   // Special handling for the bottom rows
   for i = 1 to BS − 1 by 1 do
      for t = 0 to i − 1 by 1 do
         for j = 1 to n by 1 do
            // Stream_and_Collide(i-t,j)
         end for
      end for
   end for
   // Blocking
   for i = BS to m by 1 do
      for t = 0 to BS − 1 by 1 do
         for j = 1 to n by 1 do
            // Stream_and_Collide(i-t,j)
         end for
      end for
   end for
   // Special handling for the top rows
   for i = 1 to BS − 1 by 1 do
      for t = 0 to BS − i − 1 by 1 do
         for j = 1 to n by 1 do
            // Stream_and_Collide(m-t,j)
         end for
      end for
   end for
end for
```

4.3.2 Intel Pentium 4

The Pentium also sees some performance benefit when using small grid sizes. The 8–row blocked algorithm as well as the 4–row blocked algorithm see a performance benefit over the merged algorithm until the working sets start to become too large for the 256K L2 cache. This can be seen very clearly at a size of about 140^2, where the working set of the 8-row blocked algorithm requires approximately 204.5K, and 220^2, where the working set of the 4–row blocked algorithm requires 191.8K, as the suddenly drops in performance. The 2–row blocked algorithm exhibits very odd behavior, which, without proper profiling tools cannot be analyzed in more detail. All blocked codes eventually become slower than the original code. Overall though, the non–merged algorithm is still the best performing algorithm on the P4.

4.3.3 Alpha A21164

The slower of the two Alpha architectures clearly benefits from the blocking optimizations. Grids smaller than 170^2 fit completely into the four megabyte large L3 cache, and, whereas the non–blocked code immediately drops to a performance of about 2.0 MLSUPS, the 4–way and 8–way blocked code manage to reuse data in the L2 cache and achieve higher performance levels. Especially interesting however is the sustained performance gain for even very large grids. This again is due to the large L3 cache. The working set of an 8–way blocked, 1000^2 sized grid is only 1.44M large, therefore fits completely into the L3 cache, and reduces slow memory accesses. As can be seen though, the L3 cache is too slow to provide data as fast as the processor requires it. A blocking strategy aimed at making better use of the higher cache levels would provide much better performance. Figure 4.6 plots the number of cache misses per lattice site update which occurred on the A21164. Although the L1 cache overflows almost immediately, the increase in L2 misses occurs somewhat later than it did in the non–blocked algorithm (see Figure 4.3). Also, the maximum number of L3 misses is much lower than in the non–blocked implementation, which accounts for the significant performance increase of the 1–way blocked algorithm.

4.3.4 Alpha A21264

The faster Alpha shows similar results as the slower Alpha. Some interesting points are the fact that especially the 8–way blocked algorithm drops in performance much faster than it did on the slower Alpha. The L1 cache on the A21264 has a size of 64K, so the 8–way and 4–way blocked algorithms overflow into the 4M large L2 cache much sooner than it did on the A21164. Additionally, the 2–way blocked algorithm exhibits a relatively higher performance increase on the A21264 than on the A21164. Again, the large grid sizes continue to perform better than the non–blocked implementation due to the large 4M cache, but cache strategies targeting higher cache levels would exhibit higher performance.

4.4 3-Way Blocking

As is the case with 1–way blocking, the goal of 3–way blocking is to increase reference locality. The difference between between 1–way and 3–way blocking is that, on a 2D grid, 3–way blocking attempts to reuse data in higher levels of the memory hierarchy than 1–way blocking does. Since the higher cache levels are able to respond to data requests much faster than the lower cache levels, this should provide the LBM with higher performance levels than 1–way blocking did.

The basic idea in the 3–way blocking of a 2D grid is that the large grid is divided into several smaller blocks, each fitting completely into the highest possible cache level, and then processing that block as often as data dependencies allow. To implement 3–way blocking of the LBM, the three loops used in the loop nest in the unblocked algorithm are all split into two loops, resulting in a six loop deep loop nest. As in 1–way blocking, special care must be taken to correctly process the top and bottom rows of the grid, but the beginning and end of each row of blocks must be specially treated as well. Figures 4.7 to 4.11 illustrate each step that must be taken to correctly 3–way block the LBM using a block size of 4 in x and y, as well as a time "block" of 4. Section B.1 in Appendix B demonstrates the implementation of a 3–way blocked LBM. Notable is the depth of the central loop nest, which now involves six loops.

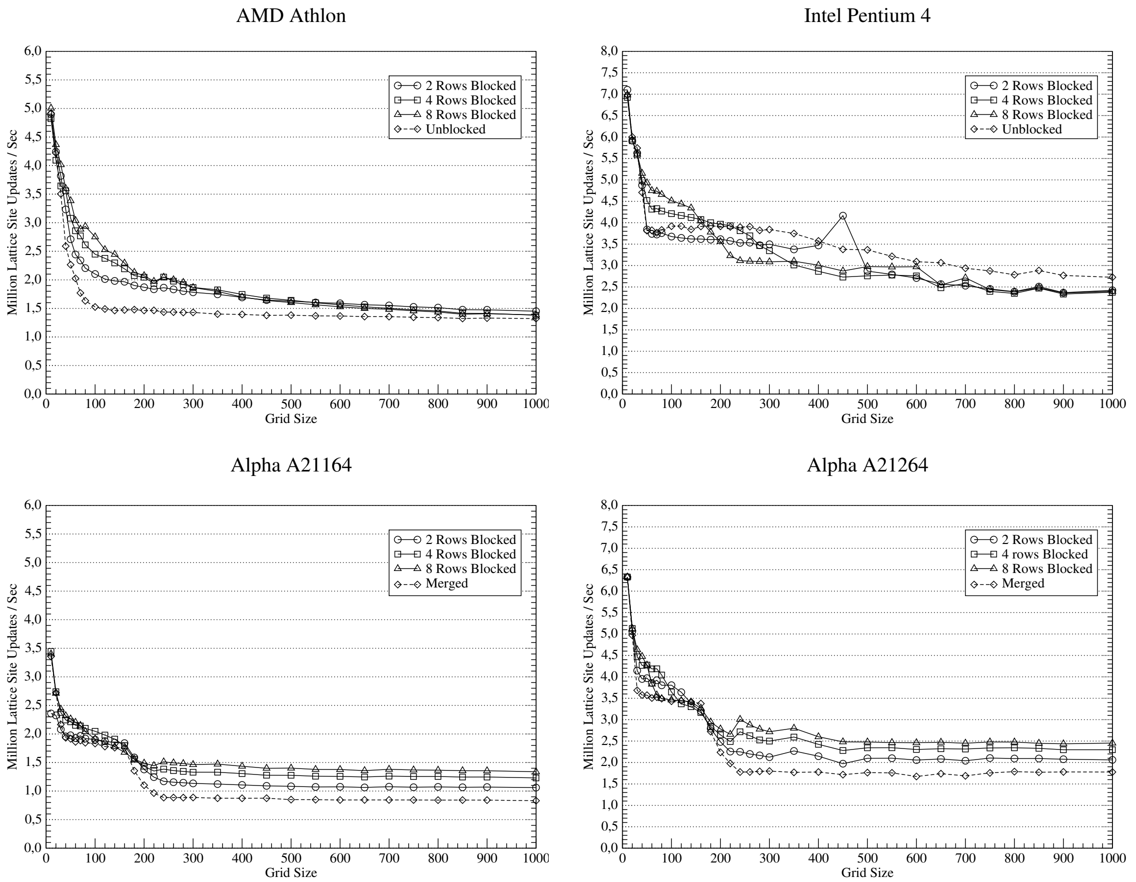


Figure 4.5: Performance of the 1-way blocked LBM implementation as opposed to the merged implementation.

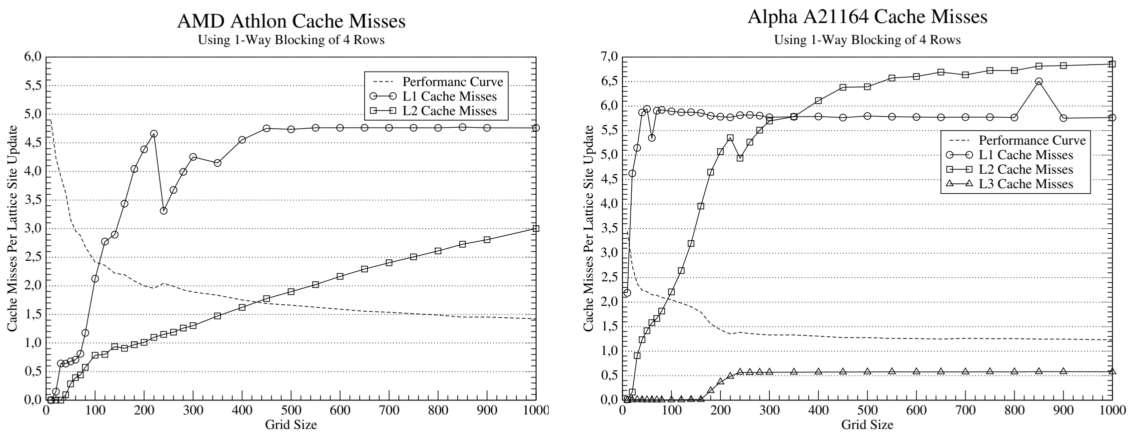


Figure 4.6: Number of cache misses that occur per lattice site update as measured with PAPI on the Athlon and DCPI on the A21164.

Figure 4.7 shows the first step that must be when 3–way blocking the LBM. This step must be performed once for each sweep through the lattice. Note that each site in the figures represent one "old" and one "new" version of the same cell, as shown in Figure 4.1. The shading represents the time–step of the "new" version.

Figure 4.8 shows the preparation step that must be taken when starting a row of blocks. This is essentially the same preparation as explained in the previous paragraph, except performed column–wise instead of row–wise. This step must be performed for each row of blocks that is processed in the lattice.

Figure 4.9 shows the actual processing of the 4x4 blocks. Individual sites within the block are processed from left to right, and bottom to top. Each block of sites is updated in such a way as to fulfill all data dependencies. The northern–most and eastern–most sites of a block cannot be updated to time–step $n + 1$ since they are neighbored by sites from time–step $n - 1$. However, an additional row to the south and an additional column to the west can be updated, since their dependencies are fulfilled. Hence, the 4x4 blocks travel down and left between time steps. The sites that overlap from time–block to time–block are those that will remain in cache and can be fetched very quickly when processing the next time–step.

Figure 4.10 demonstrates the completion of a row of blocks. All sites which could not be updated within a block are processed here. This step must also be performed at the end of each row of blocks.

Finally, Figure 4.11 shows the completion of the sweep through the grid. Here, all remaining rows, which could not be updated in a row of blocks, are updated to the latest time–step.

4.5 Performance using 3–Way Blocking

Figure 4.12 presents the performance results for the 3-way blocked LBM. An obvious change when compared to the non–blocked as well as the 1–way blocked implementations is the fact that the performance does not drop sharply as the grid size is increased, even for very large grids. Very small grids, of course, profit from the relatively high percentage of boundary cells. There are several small decreases in performance to be seen though, these will be more closely examined in Section 4.6.

4.5.1 AMD Athlon

The Athlon clearly benefits from the blocking technique. Though performance still drops somewhat, it levels off much higher than the unblocked algorithm. At block sizes greater than 16×16, the block becomes too large for the L1 cache and performance starts to drop again. The 16×16 block requires a total of 45.6KB. Figure 4.13 shows the relationship between the performance of the algorithm and the cache behavior. In general, the number of L1 misses have been drastically reduced when compared to the non–blocked and 1–way blocked algorithm. There are several sharp spikes however, coinciding exactly with performance drops. The will be more closely examined in the next section.

4.5.2 Intel P4

The Pentium 4 also gains in performance when compared to the non–blocked algorithm. Overall however, the performance is roughly that of the non–merged algorithm presented in chapter 2. Oddly enough, even the 4×4 blocked algorithm, in which a block should theoretically fit into the small 8KB L1 cache, performs more poorly than algorithms using larger blocks. Predominant is also the drop in performance at at grid size of 450^2, which is exhibited by all architectures.

4.5.3 Alpha A21164

The older Alpha also benefits from the application of loop blocking, the blocked algorithm is more than twice as fast as the non–blocked algorithm. In chapter 2 it was shown that the 2.0 MLSUPS exhibited by the A21164 is equal to about 230 MFLOPS, which is approximately one quarter of the theoretical peak of 1.0 GFLOPS. Clearly, if the actual calculation of new particle distributions performed better, the benefits of cache optimization would be much more apparent.

Similar to the P4, the algorithm using a small block size of 4×4, which should fit into the 8KB L1 cache, does not out perform the algorithm using a larger block size, which do not fit into the L1 cache. Figures 4.13 and 4.14 show the cache behavior when using a 16×16 and 4×4 block,

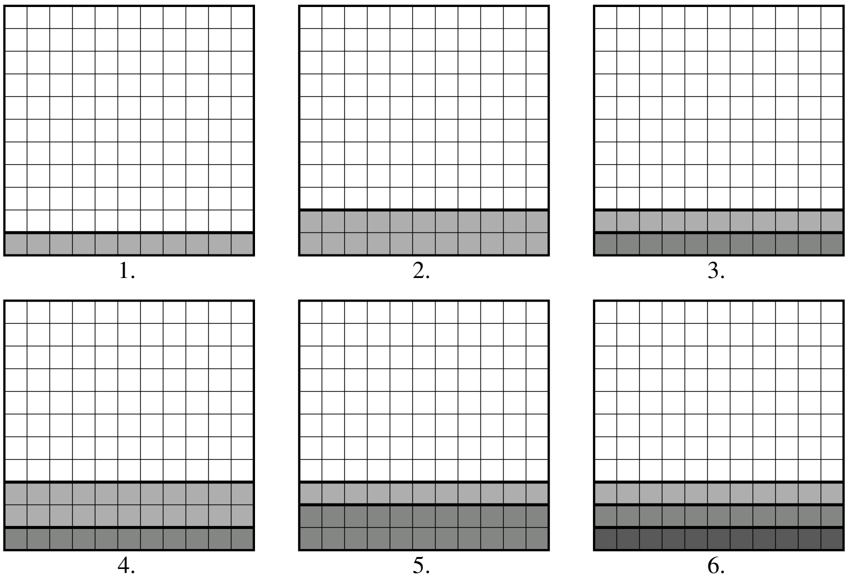


Figure 4.7: Initialization of the bottom rows is the same as in 1–way blocking.

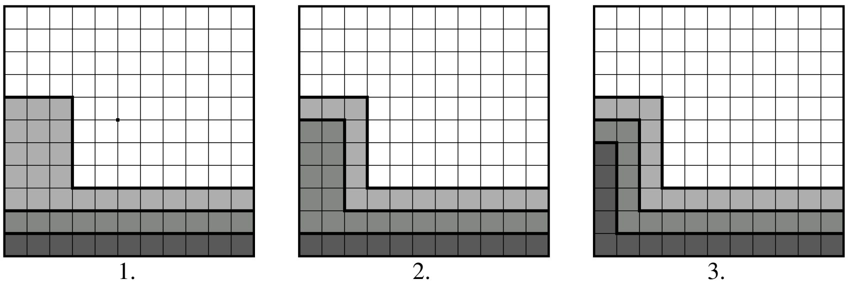


Figure 4.8: Initializing a row of blocks.

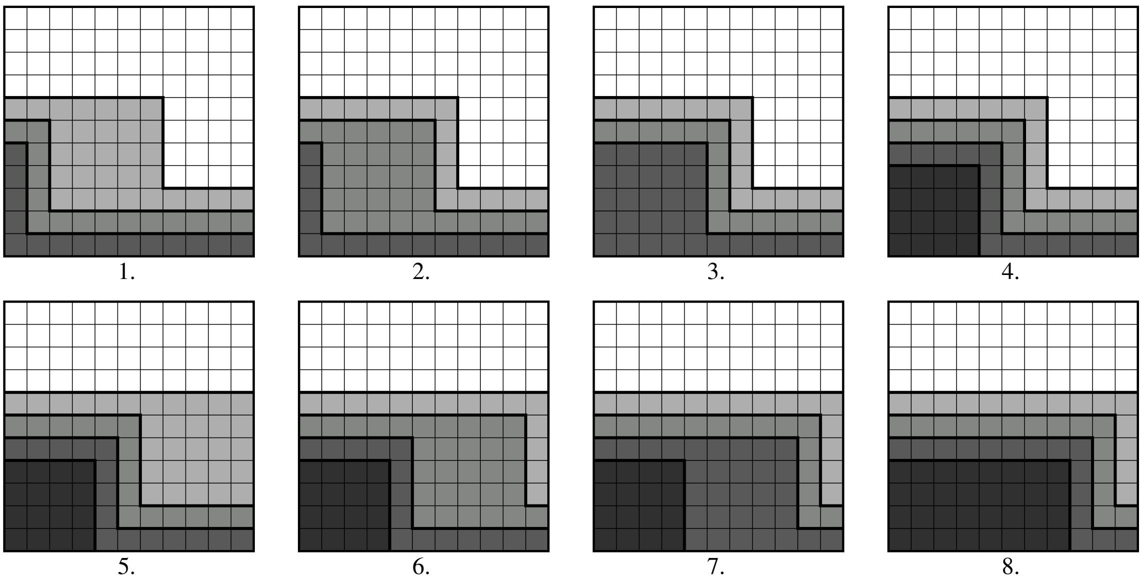


Figure 4.9: As many 4x4 blocks as possible are processed in one row of blocks.

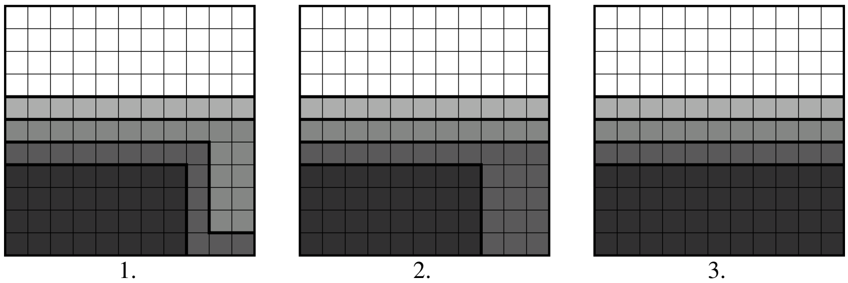


Figure 4.10: All remaining sites that were not updated in a block must be brought to the newest possible time step.

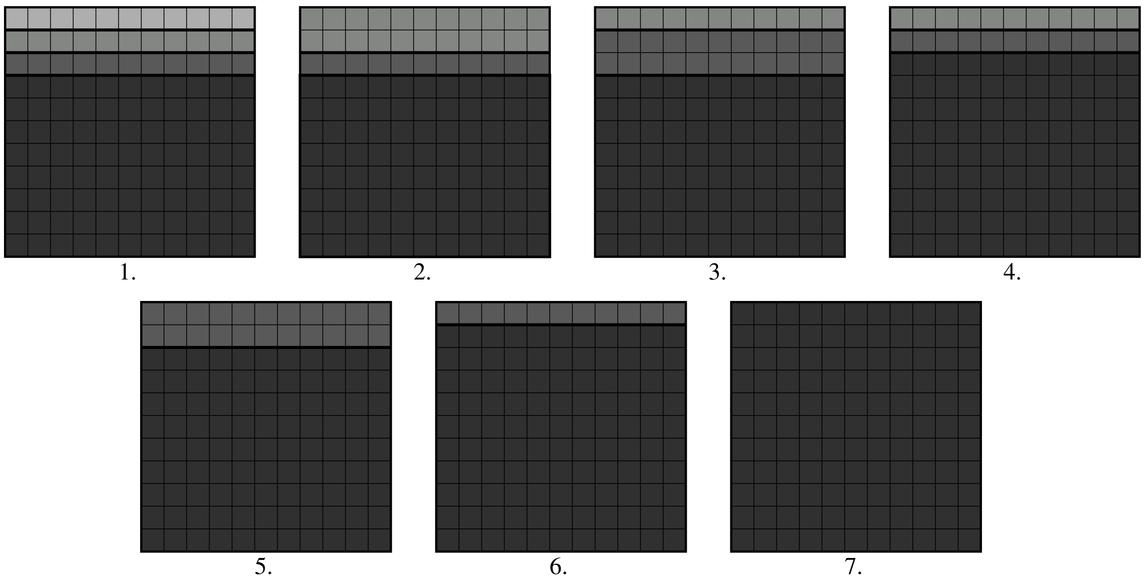


Figure 4.11: After all possible block rows have been processed, all the rows not in the latest time step must be completed.

respectively. Although the 4×4 block does in fact result in fewer L1 misses, it suffers from twice as many L3 misses. This is attributable to the fact that in one pass through the grid, the 16×16 blocked algorithm performs 16 time steps. The 4×4 blocked implementation, on the other hand, performs only 4 time steps, which forces the algorithm to perform more passes through the entire grid than the 16×16 blocked implementation. In general, the fewer passes one has to make over the entire grid, the better, since the size of a large grid prevents any data reuse from previous passes.

Overall, the amount of data required by the LBM to update a single site is too large to effectively use the small L1 caches provided by the A21164 and the P4. Blocking sizes targeted at using the much larger L2 cache are much more effective. In fact, due to the 96KB L2 cache on the A21164, larger block sizes up to about 22×22 perform slightly better than using a block size of 16×16.

4.5.4 Alpha A21264

Finally, the Alpha A21264 also shows a performance increase. Since the L1 cache is the same size as that of the Athlon, a block size of 16×16 exhibits the best performance.

4.6 Padding

One notable characteristic shared by all architectures however, is an occasional, very local, drop in performance. This occurs for example at a grid size of 450^2 on every architecture that was tested. Figure 4.15 shows a more detailed plot on the performance and cache behavior occurring at this grid size. Both the Athlon and the Alpha show a sharp increase in high level cache misses, peaking at 454^2 on the Alpha and 456^2 on the Athlon. After some simple calculation, it turns out that at a grid size of 455^2, a single row of the grid requires 65808 bytes of memory. This is almost exactly the size of the L1 cache on the Athlon and is a multiple of the L1 cache on the A21164. Apparently, the sudden increase in cache misses is due to conflict misses caused by large amounts of data being mapped to the same cache line. As was shown in the previous chapter, conflict misses can be avoided by introducing additional data at the end of each row, a *pad*, in order to cause data to be mapped to different cache lines.

Figure 4.16 demonstrates the performance of the same algorithm as shown in Figure 4.15, this time using a pad of 160 bytes (the size of 20 double precision values) at the end of each row in the lattice. As can be seen, the pad has not eliminated the performance drop due to conflict misses, it has, however, shifted them to a smaller grid size, by exactly the number of double values used as a pad. Therefore, padding cannot eliminate conflict misses, if the grid size that one is calculating is suffering from a high number of conflict misses, padding can be introduced in order to shift the conflict misses to a different grid size.

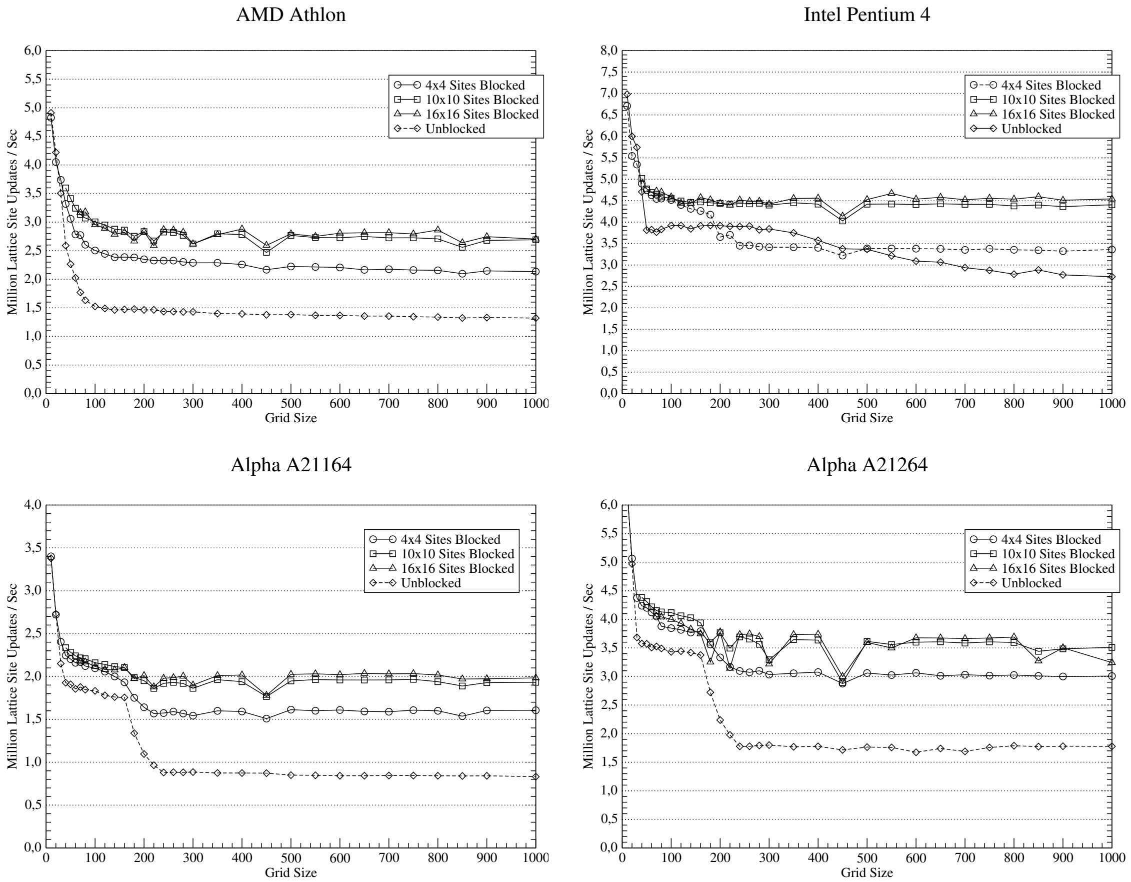


Figure 4.12: Performance of the 3-way Blocked of the LBM as opposed to the non–blocked implementation.

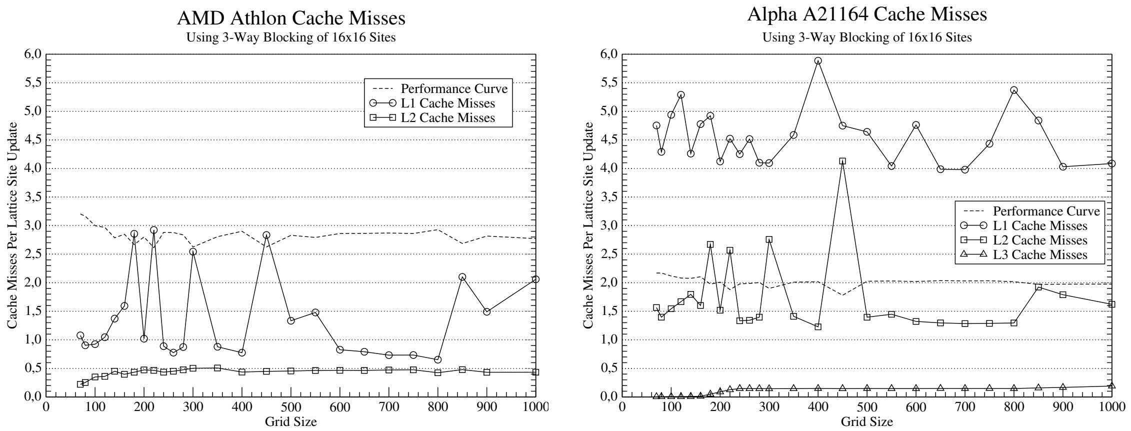


Figure 4.13: Number of cache misses that occur per lattice site update as measured with PAPI on the Athlon and DCPI on the A21164.

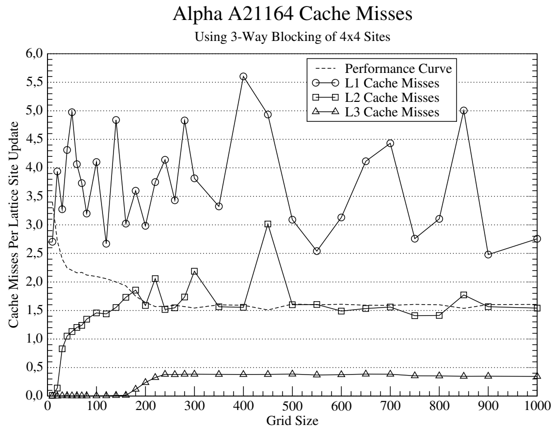


Figure 4.14: Number of cache misses that occur per lattice site using a 4x4 block.

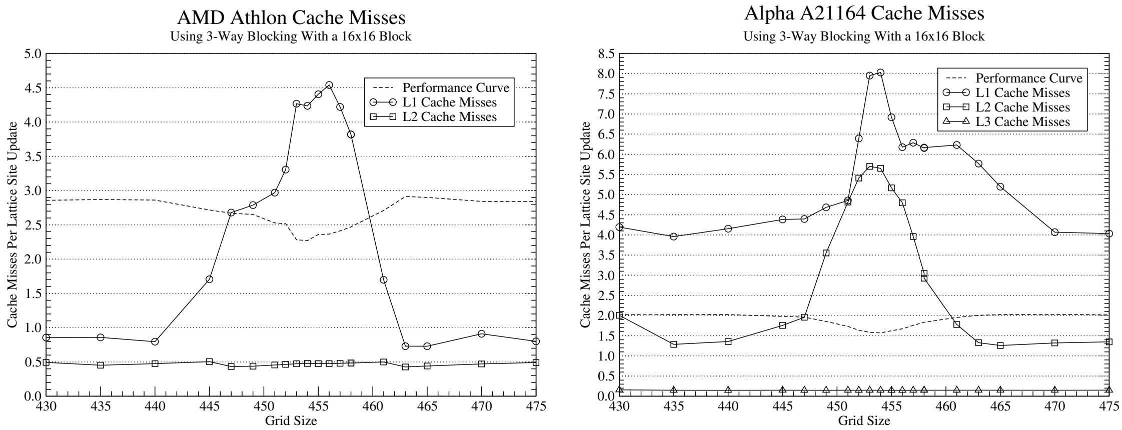


Figure 4.15: Closer examination of the performance drop exhibited on all architectures at a grid size around 450^2 using PAPI and DCPI.

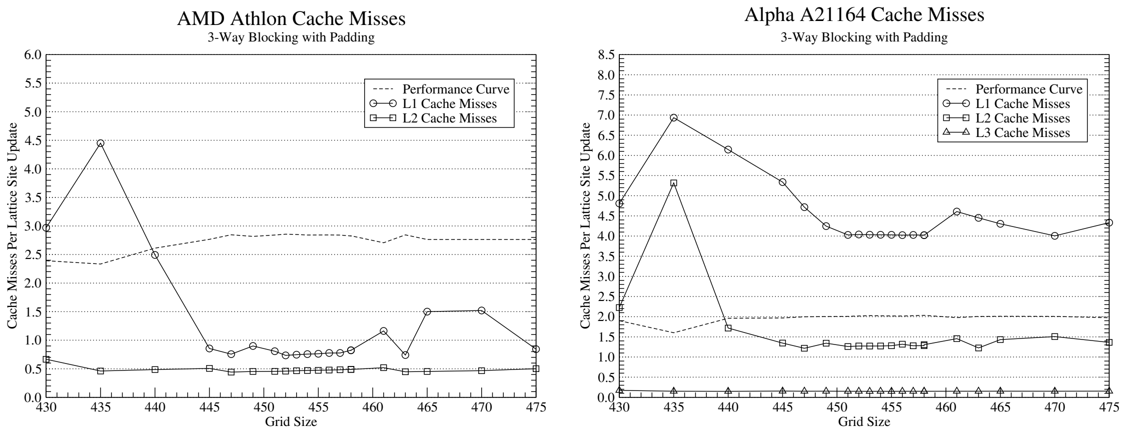


Figure 4.16: The effect of padding on the performance and cache behavior. The performance drop as been shifted left to a smaller grid size.

Chapter 5

Compressed Grid LBM

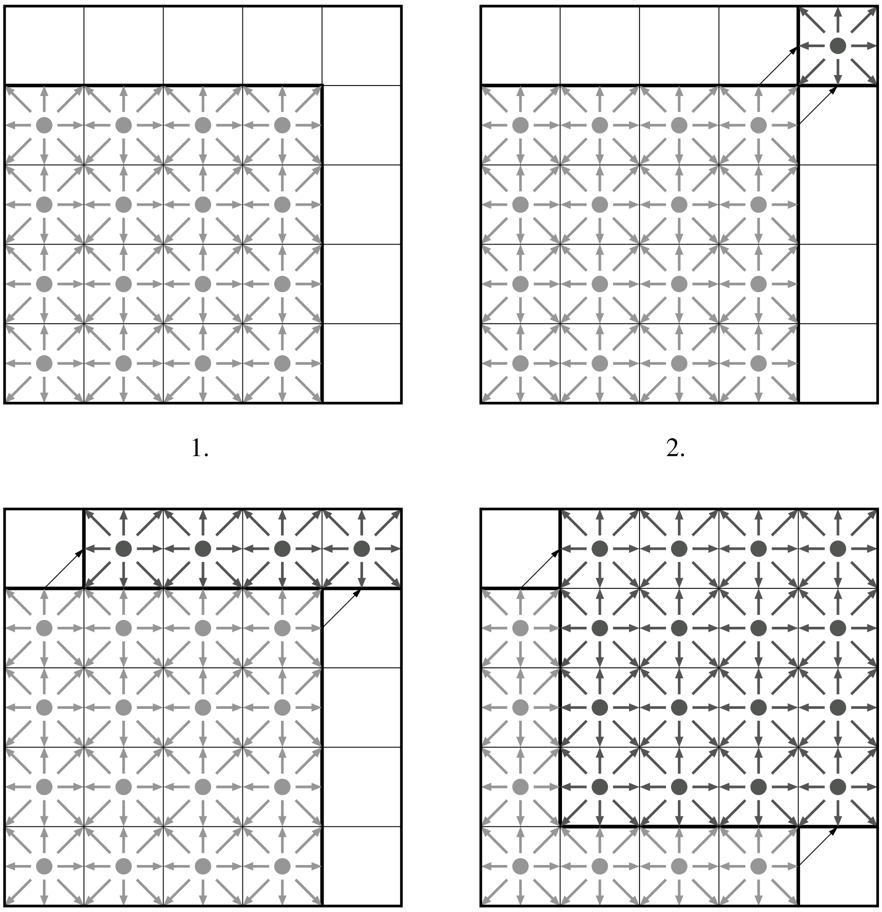
This chapter will discuss an alternative method of combining the two grids of the LBM into a single grid than the merging method presented in the previous chapter in order to reduce the amount of memory required and therefore the memory traffic as well. A similar approach to reducing the memory requirements this type of algorithm (e.g. LBM, Jacobi) using a transitory array was demonstrated in [BDQ98]. Cache optimizations will also be applied to this method and analyzed.

5.1 Compressed Grid Method

One drawback apparent in the LBM is the amount of data required to implement a single grid site. This quickly leads to very large grids, which in turn results in many *capacity misses* in cache. On closer examining the data dependencies when updating a grid site, a strategy to combine the two lattices as well as reducing the amount of required memory without introducing additional computational overhead will be presented.

This method generally works in two phases, figure 5.1 illustrates the first phase. The first grid depicts a single grid, with initialized lattice sites in the lower left 4x4 square. This 4x4 square represents a typical lattice Boltzmann grid of size 4x4 in time step 0. The topmost row and rightmost row have been added to the 4x4 grid and remain uninitialized. Beginning with the top right initialized $(4, 4)$ lattice site, the nine distribution function values are collided, and then streamed to the site diagonally up and right from itself, as is depicted in the second grid in figure 5.1. If one continues to collide and stream the top row, proceeding right to left, the grid would be in the state depicted in the third grid. The uppermost row now contains sites in time step 1. It is important to note, that all the data from the row just updated is still preserved in the grid. This is important since the next row cannot be updated correctly without this information. By continuing in this manner, from right to left and top to bottom, the remaining rows can be collided and streamed, leaving the grid in the state depicted in the fourth grid. A $4{\times}4$ lattice Boltzmann grid, completely in time step 1, is now located in the upper, right $4{\times}4$ square. This completes the first phase.

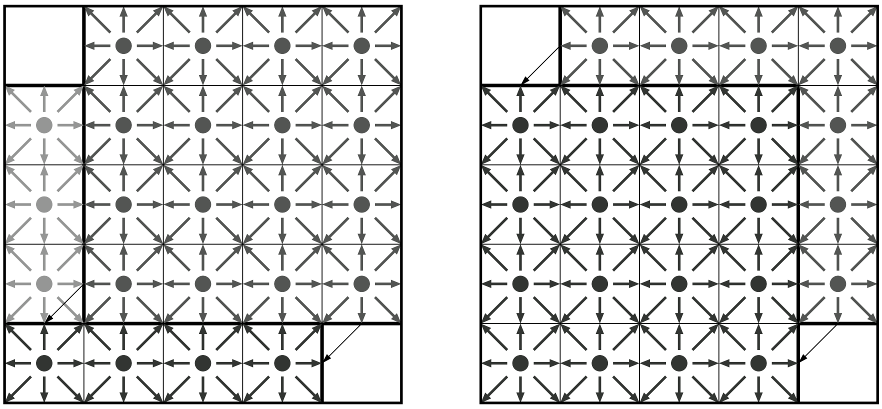
The second phase works exactly the same way as the first phase, except in the opposite direction. The final grid in figure 5.1 reveals a similar picture as the first grid. The difference being that the additional row is on the bottom instead of the top, and the additional column is on the left side instead of the right, and that this row and column contain data from the previous time step. Since this information is no longer needed it can be overwritten. To begin the second phase, the first site to be collided and streamed is the lowest and leftmost site of the $4{\times}4$ lattice $(2, 2)$. Now, the collided distribution function is streamed to the site located down and left of the current site. The first grid in Figure 5.2 illustrates the grid state after processing the bottom row. By traversing the grid from left to right, bottom to top, time step 2 is completed with the complete grid back in the lower, left $4{\times}4$ square, as shown in the second grid in figure 5.2. Phase 1 can now be applied again to yield time step 3, and phase 2 to yield time step 4. These two phases are repeated as often as necessary to complete the required number of time steps. The algorithm is shown in algorithm B.2. It is important to note that the two phases are very similar to the normal version of the LBM presented in Algorithm 2.9. The only differences are the declaration of a single array of double



1. 2.

3. 4.

Figure 5.1: The first phase of the compressed grid method, information flows diagonally up and right.



1. 2.

Figure 5.2: The second phase of the compressed grid method, information flows diagonally down and left.

values, with an extra column and row, and the stencil which is used to perform the collide and stream operations.

This method manages to reduce the amount of memory required by overwriting information no longer needed to process the newest time step. By not storing the old and new values of every single site, as the array merging method does, this method drastically cuts down on the amount of memory required. Each lattice site requires only ten double precision values, nine to store the distribution function values, and one double to determine the type of lattice site (technically, a simple integer would do, but in this manner, only a single double array must be allocated in order to implement the entire algorithm). Additionally, one extra row and column must be allocated.

Figure 5.3 illustrates the performance measurements taken on the four test architectures. As can be seen, the compressed grid version of the LBM runs faster than the merged and non–merged version on every single architecture. Figure 5.4 shows the cache behavior exhibited by the compressed grid LBM. When compared to the cache behavior of the merged algorithm in Figure 4.3, the benefit of the compressed grid LBM is easy to be seen. The lower memory requirement of the compressed grid version allows more useful information to be kept in cache than is the case with the other versions, which greatly reduces the number of cache misses and therefore reduces the total amount of performance robbing memory traffic.

5.2 1–Way Blocking

In principle, the application of 1–way blocking to the compressed grid version of the LBM is no different than 1–way blocking the merged version in the previous chapter. Once the data dependencies of a row are satisfied, it can be processed. However, due to the fact that much of this information is discarded by the compressed grid implementation, care must be taken to preserve the information needed when performing several time–steps in a single pass through the lattice. Additionally, each of the two phases performed by the compressed grid implementation of the LBM must be blocked separately. An example of 1–way blocking 2 rows when performing the first phase of the compressed grid LBM version is shown in Figure 5.5. It is important to note the *two* extra rows and columns at the top and right of the grid, as opposed to simply one. These are necessary so that the information required to fulfill the data dependencies of each time step. The number of extra rows and columns required is always equal to the number of time–steps being blocked. The pseudocode for an implementation using 1-way blocking is shown in Appendix B, Section B.3.

5.3 Performance Using 1–Way Blocking

Figure 5.6 presents the performance results for the 1–way blocked LBM implementation. Overall, the results are similar to those presented in section refsec:1way-perf in the previous chapter, except that this implementation results in somewhat higher performance levels.

5.3.1 AMD Athlon

The Athlon displays exactly the results which are to be expected from the application of 1–way blocking. The drop in performance due to the lattice site spilling over the L2 cache boundary and into memory is shifted towards larger grid sizes. For very large grids however, the 1–way blocked implementation is no faster than the unblocked implementation. Figure 5.4 displays the cache behavior of the 1–way blocked implementation.

5.3.2 Intel Pentium 4

Once again, the Pentium performs worse with blocking than without. According to [Int02], the Pentium 4 can perform automatic prefetching in hardware. This would explain the strange performance results exhibited by the blocked code. The hardware prefetch unit searches for regular access patterns with which to perform prefetching. The access pattern of the loop blocked code merely serves to confuse the prefetch unit since the accesses are much less regular than in the unblocked code.

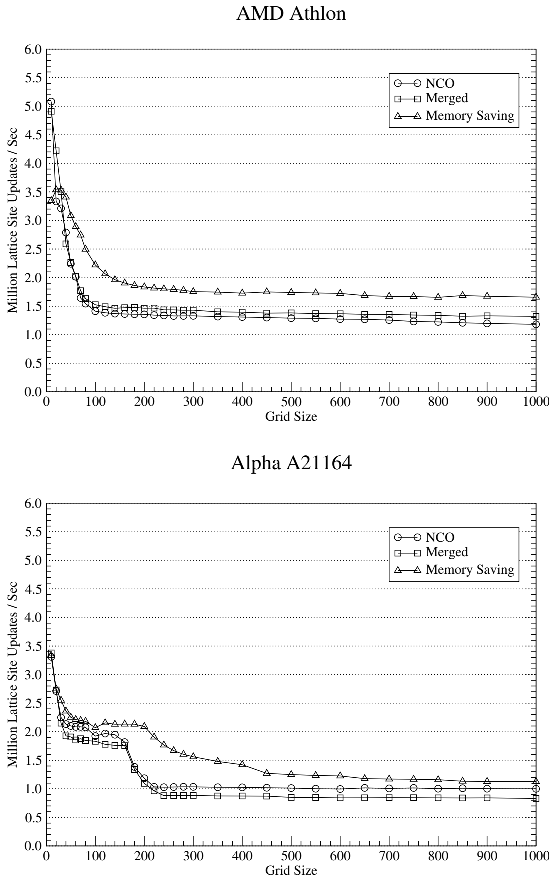


Figure 5.3: Performance of the Compressed Grid version of the LBM as opposed to the non-merged algorithm and merged algorithm.

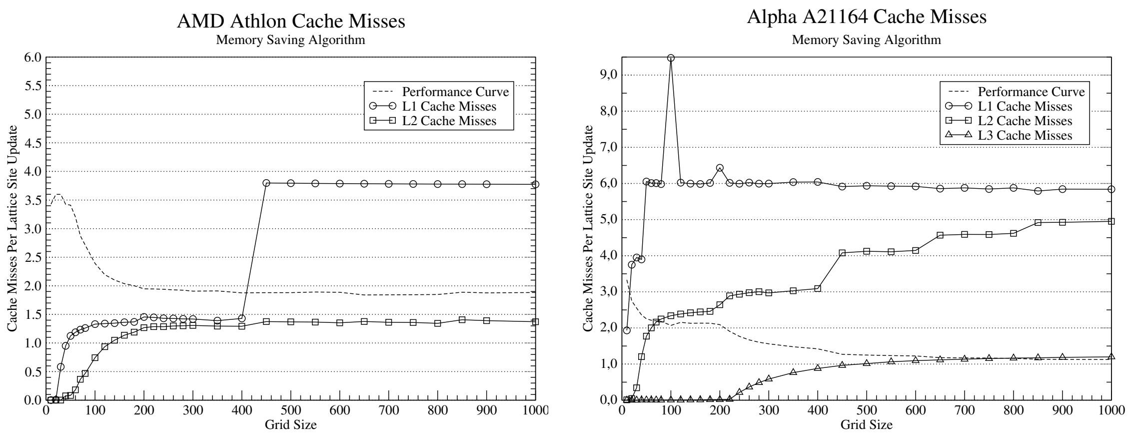


Figure 5.4: Number of cache misses that occur per lattice site update as measured with PAPI on the Athlon and DCPI on the A21164.

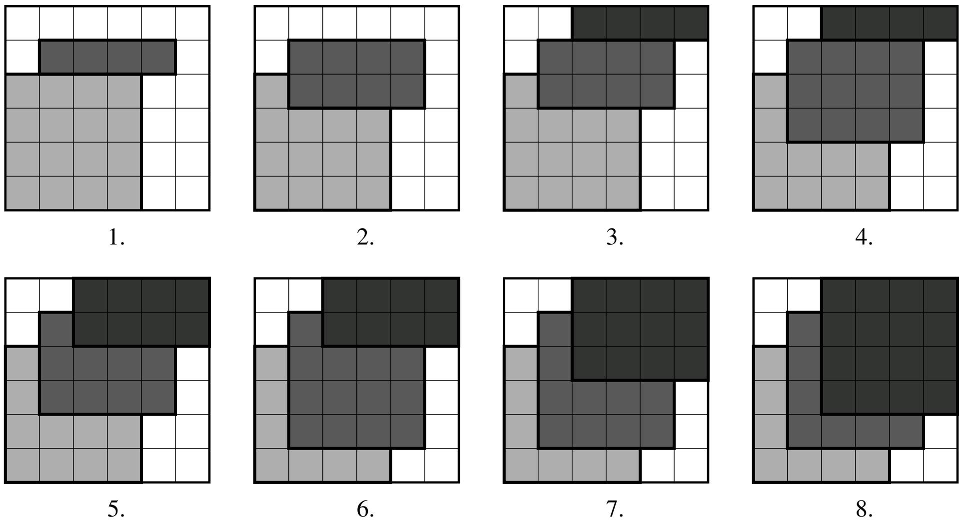


Figure 5.5: Phase 1 of the compressed grid version of the LBM using 1–way blocking. The information from one time–step to the next always flows one row up and to the left.

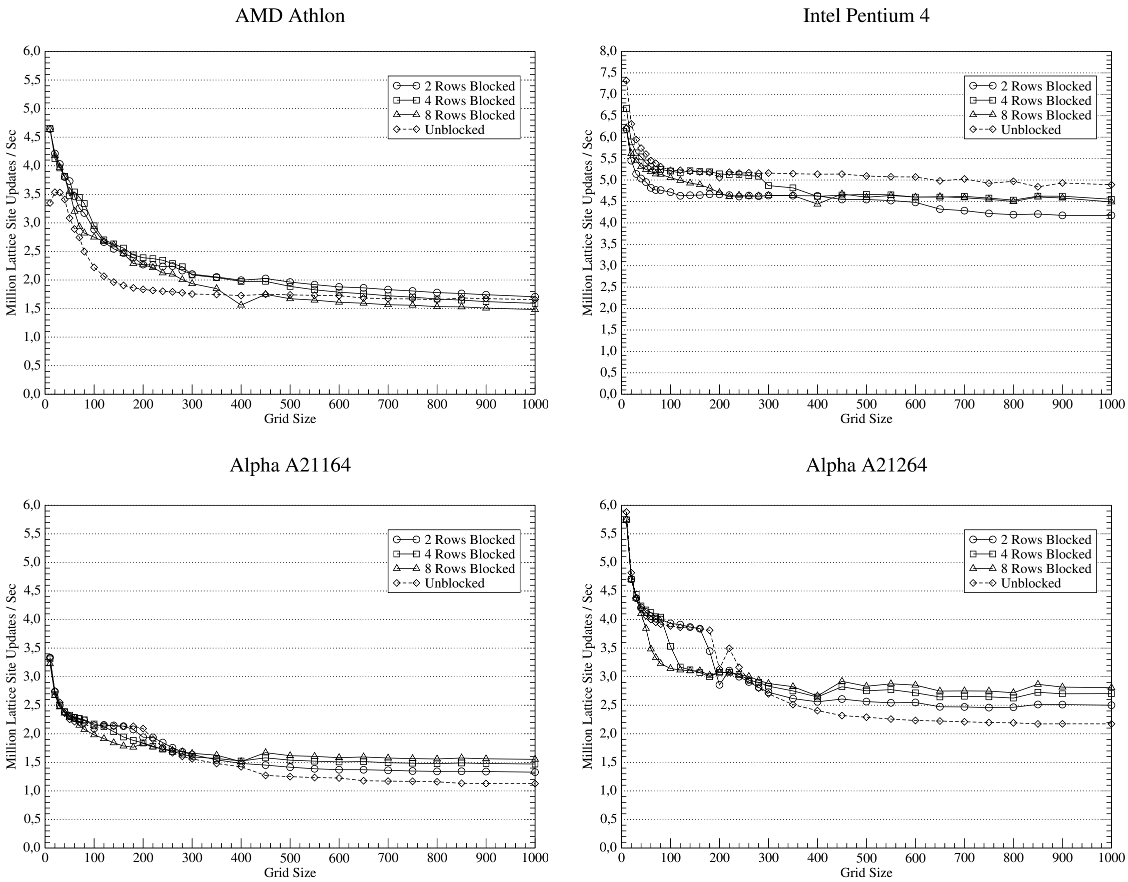


Figure 5.6: Performance of the Compressed Grid version of the LBM using 1–way Blocking as opposed to the unblocked implementation.

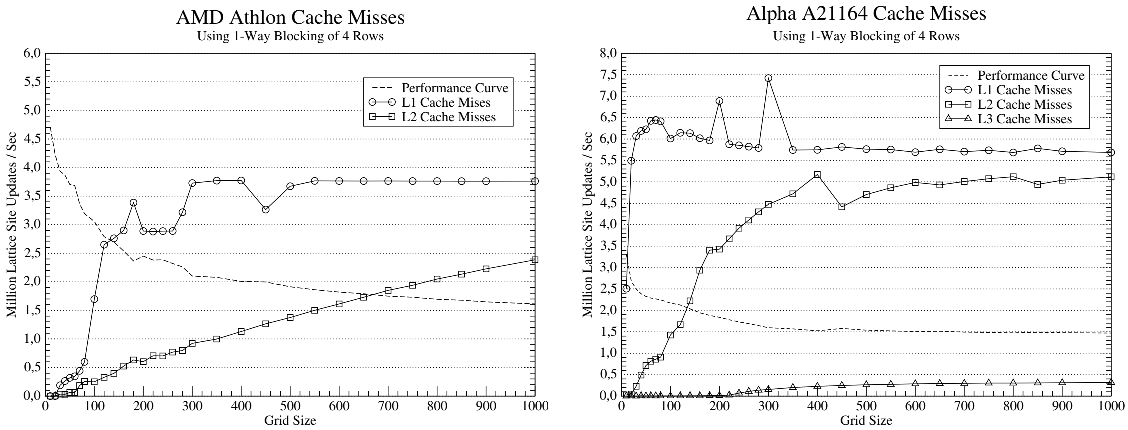


Figure 5.7: Number of cache misses that occur per lattice site update as measured with PAPI on the Athlon and DCPI on the A21164.

5.3.3 Alpha A21164

The large L3 cache of this Alpha once again plays a role in the performance levels of very large grids. The working set still fits into this cache, which gives the blocked implementation a higher performance level than in the unblocked implementation. Curiously, no performance benefit is seen at smaller grid sizes, where one would expect some increase in performance. This was the case in the merged algorithm. When comparing the cache behavior of the unblocked implementation (Figure 5.4) with that of the blocked implementation, it is apparent that 1–way blocking greatly increased the number of L2 misses incurred (see Figure 5.7 . In the merged implementation from the previous chapter, the increase in L2 misses from the unblocked to the blocked implementation was much lower.

5.3.4 Alpha A21264

The faster Alpha exhibits similar behavior as the slower Alpha, described above.

5.4 3–Way Blocking

This section describes the effect of 3–way blocking on the compressed grid implementation. The principle is the same as it was in the previous chapter, with the added complication that the two separate phases performed by the compressed grid implementation must be blocked. Additionally, since the information for each time–step "flows" up and right or down and left, depending on which phase is being performed, additional rows and columns must be introduced. This is similar to what was done in the 1–way blocked implementation. The number of additional rows and columns introduced to the grid depends upon the number of time steps that are blocked.

Figures 5.8 to 5.12 demonstrate 3–way blocking using a block size of 4×4 and a time block of 3, hence the additional 3 rows and columns. Only the first phase is shown. Upon completion of the first phase, a complete $n \times n$ lattice has moved from the lower, left $n \times n$ sites to the upper, right $n \times n$ sites, as is shown in figure 5.12. The sites are processed from right to left, all rows are processed from the top to the bottom. In the second phase, one would proceed left to right, bottom to top, with the grid being back in the lower, left $n \times n$ sites.

In the In all benchmarks performed using the 3–way blocked compressed grid implementation, the number of times the time loop was blocked was equal to the number of times the x and y loop were blocked. However, different combinations may yield better performance results. The pseudocode for an implementation using equal size time, x, and y blocks is shown in Appendix B, Section B.4.

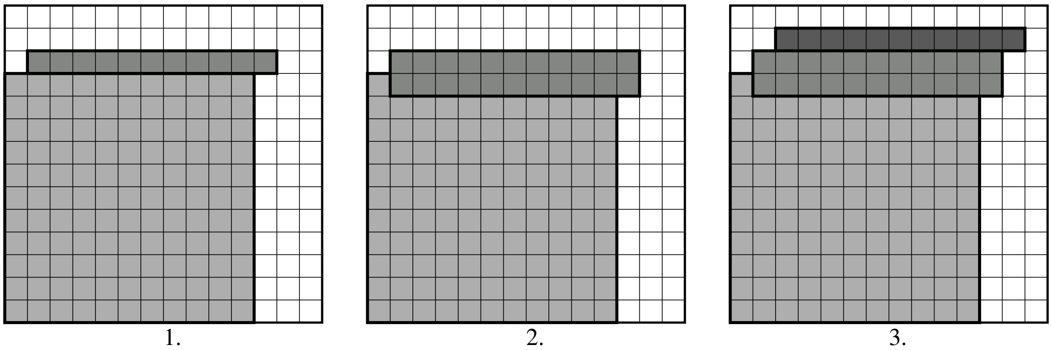


Figure 5.8: Initialization of the first rows is the same as in 1–way blocking.

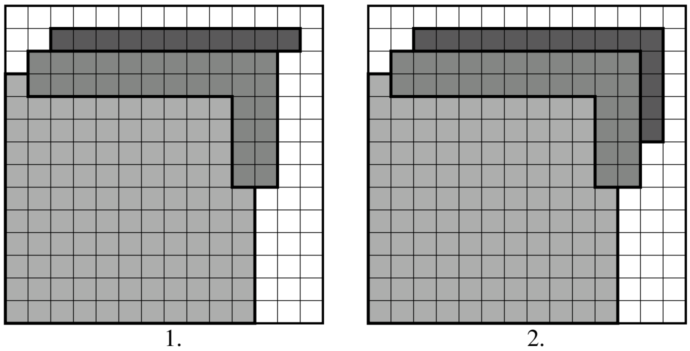


Figure 5.9: Initializing a row of blocks.

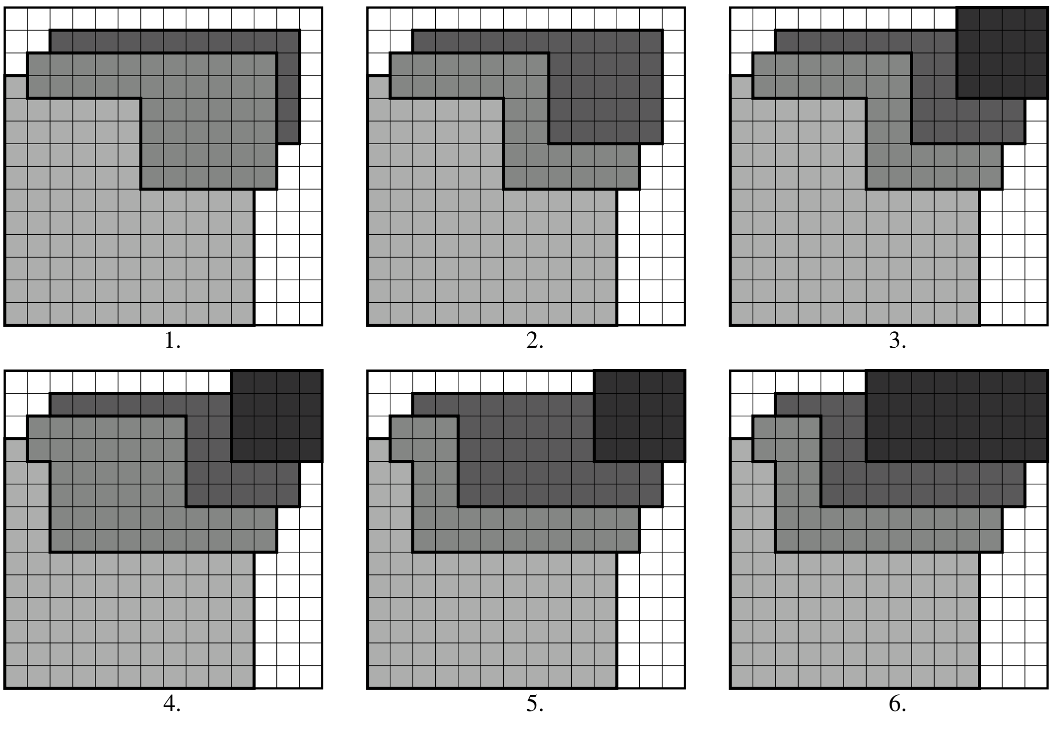


Figure 5.10: As many 4×4 blocks as possible are processed in one row of blocks.

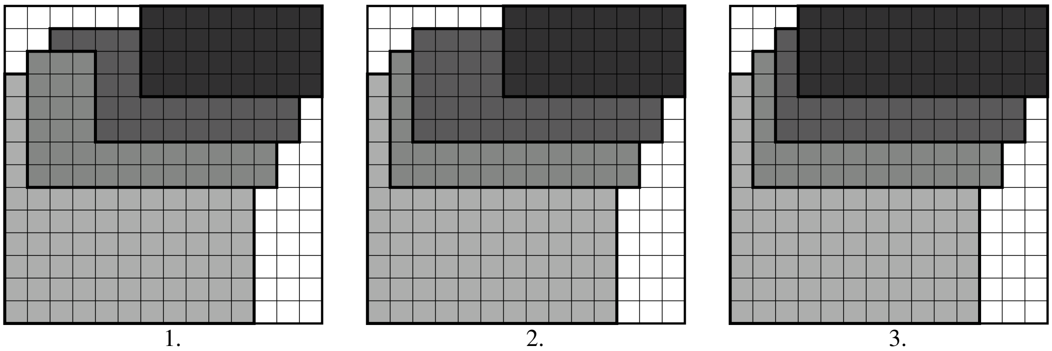


Figure 5.11: All the remaining sites that could not be updated in a block must be processed.

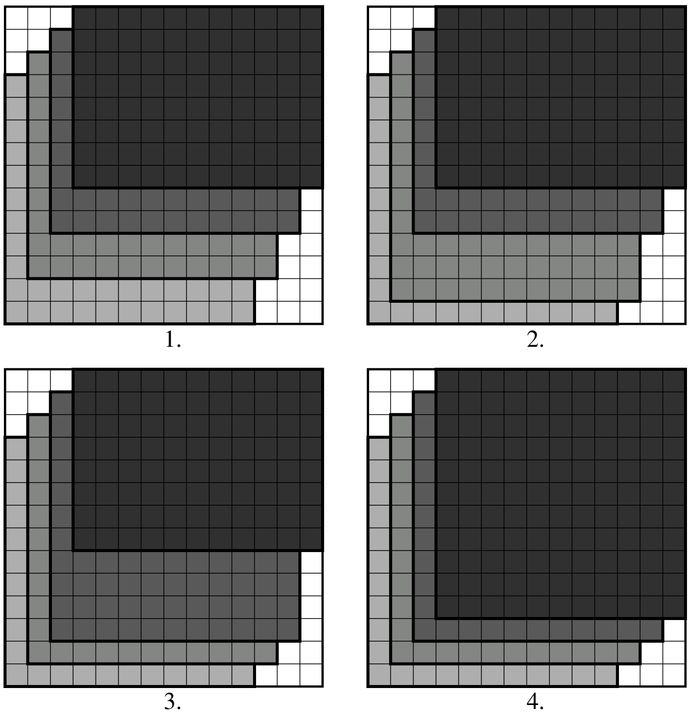


Figure 5.12: After all possible block rows have been processed, the remaining rows must be completed.

5.5 Performance Using 3–Way Blocking

The 3–way blocked, compressed grid implementation of the LBM is altogether the highest performing of all implementations presented. It avoids the drop in performance usually exhibited when processing large grid sizes, and is marginally faster than the 3–way blocked, merged implementation of the previous chapter.

5.5.1 AMD Athlon

The 3-way blocked, grid compressed implementation is the fastest implementation of the LBM as measured on the AMD Athlon. The performance curve is, in general, very flat, which indicates that it avoids the large number of capacity misses incurred by the simpler implementations. When comparing the cache misses of this implementation (Figure 5.14) with the 3-way blocked implementation in the last chapter (Figure 4.13), it is apparent that this version incurs fewer cache misses overall, explaining its slightly higher performance. The sharp local drops, for example at a grid size of 400^2 and 800^2, in performance are due to conflict misses and will be further described in Section 5.6.

5.5.2 Intel Pentium 4

The 3-way blocked implementation shows definite improvement over the 1-way blocked implementation, yet the unblocked grid compressed implementation is the fastest overall on the Intel P4. As mentioned above, according to [Int02], the Pentium 4 can perform automatic prefetching in hardware. This would explain the strange performance results exhibited by the blocked code. The hardware prefetch unit searches for regular access patterns with which to perform effective prefetching. The access pattern of the loop blocked code merely serves to confuse the prefetch unit since the accesses are much less regular than in the unblocked code. In the case of the 3-way blocked code, the cache benefits of blocking are almost as effective as the performance benefits of the hardware prefetch unit.

5.5.3 Alpha A21164

The Alpha A21164 shows similar results as the AMD Athlon. This implementation is the fastest of all implementations tested. A comparison of the cache misses of this implementation (Figure 5.14) and the 3-way blocked implementation in the last chapter (Figure 4.13) show that this implementation manages to further reduce L2 and L3 cache misses, although L1 misses increase slightly. The sharp local drops, for example at a grid size of 400^2 and 800^2, in performance are due to conflict misses and will be further described in Section 5.6.

5.5.4 Alpha A21264

The faster Alpha A21264 shows similar performance with this implementation as with the 3-way blocked merged implementation in the last chapter. The only benefit of this implementation would be the reduced memory requirements.

5.6 Padding

As was the case with the 3–way blocked, merged implementation, the 3–way blocked memory saving implementation exhibits several sharp, local drops in performance when using some block sizes. This occurs on all architectures at a grid size of about 800^2, using a 16×16 block, for example. On closer examination, using the profiling tools on the Athlon and the A21164, a sharp rise in cache misses is observed. The increase in cache misses corresponds exactly with the drop in performance. Figure 5.15 shows the relationship between the performance drop and the cache misses. At a size of 802^2, using a block size of 16×16, a single grid row requires 65600 bytes of memory. This corresponds very closely to the size of the L1 cache on the Athlon, and is a multiple of the L1 cache size on the Alpha. Hence, the cache misses incurred here are conflict misses and can be eliminated by intra–array padding.

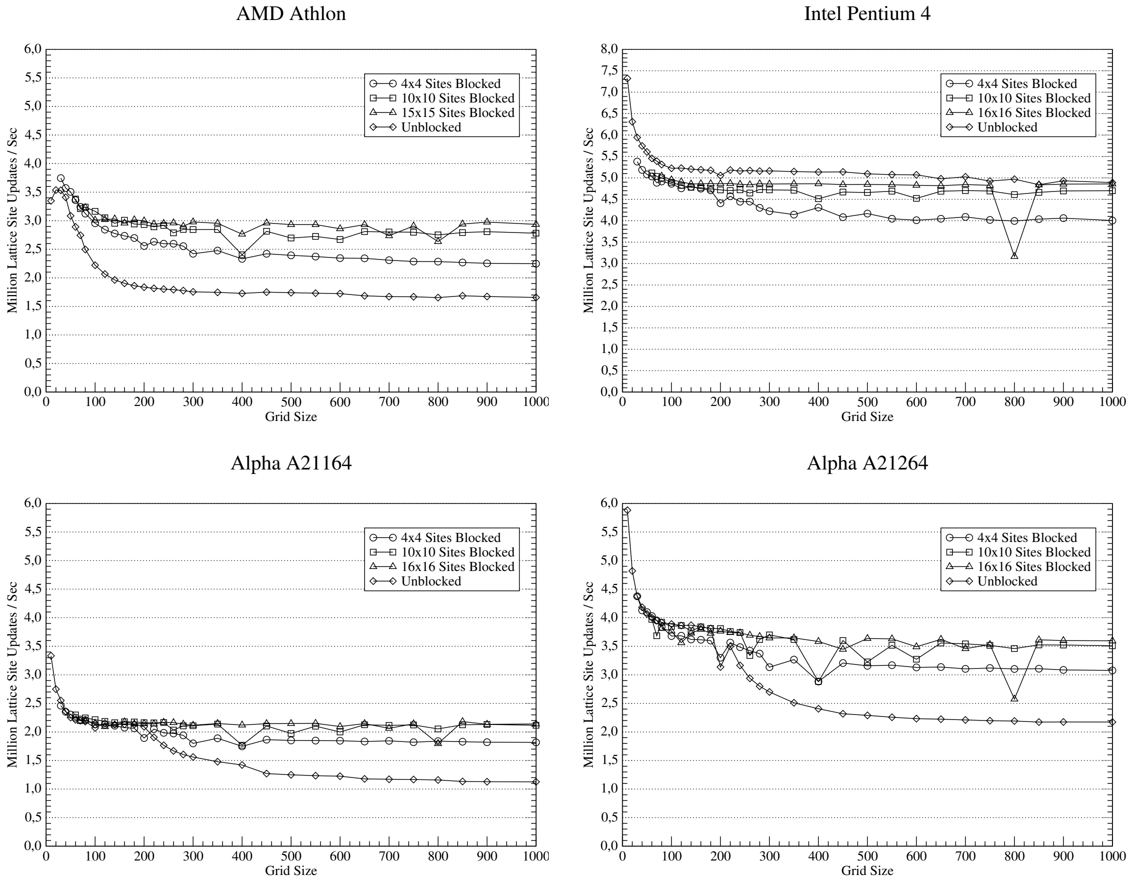


Figure 5.13: Performance of the 3–way Blocked Compressed Grid version of the LBM as opposed to the non-blocked compressed grid version.

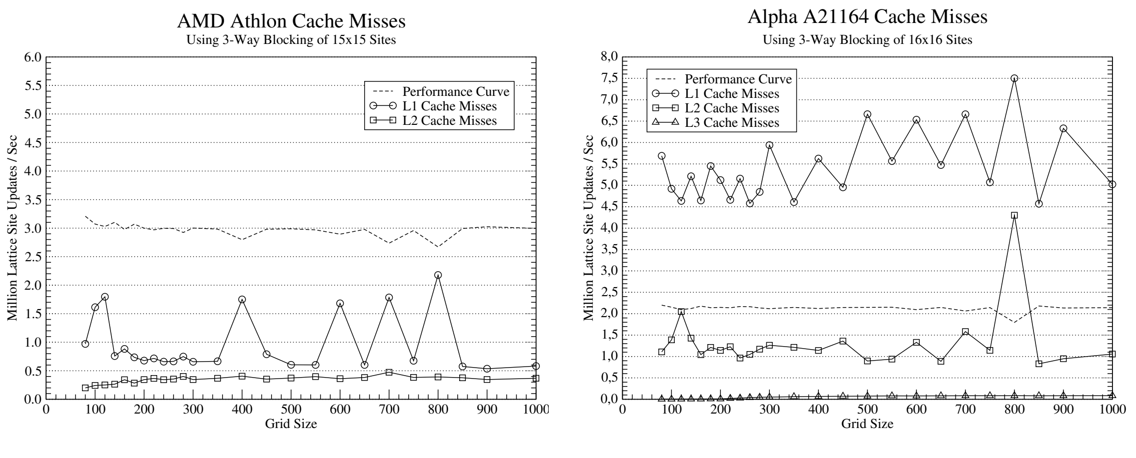


Figure 5.14: Number of cache misses that occur per lattice site update as measured with PAPI on the Athlon and DCPI on the A21164.

It is important to note that since the number of times the *time* loop is blocked directly influences the overall size of the grid, different block sizes will exhibit performance drops at different grid sizes. This explains why the 16x16 block exhibited a drop in performance at 802^2, and the 10×10 and 4×4 block did not. These sizes suffer from performance drops as well, simply at different sizes.

Figure 5.16 demonstrates the effect that a pad of 10 doubles at the end of each grid line had on the performance. The performance drop has been shifted from a size of 802^2 to 792^2, exactly the size of the pad applied.

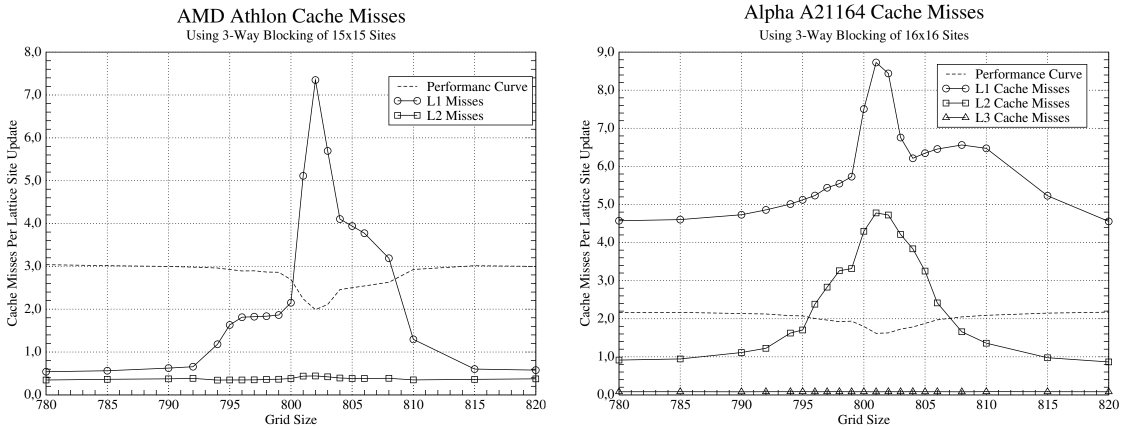


Figure 5.15: Closer examination of the performance drop exhibited on all architectures at a grid size around 800^2 using PAPI and DCPI.

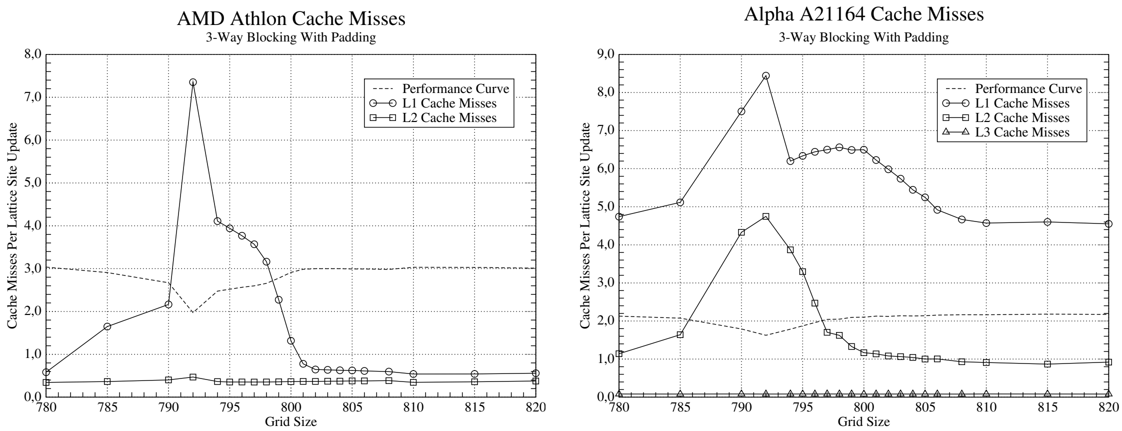


Figure 5.16: The effect of padding on the performance and cache behavior. The performance drop has been shifted left to a smaller size.

Chapter 6

Conclusions

The Lattice Boltzmann Method is a fairly new method of simulating dynamic fluid flows using a particle approach based on the Boltzmann Equation instead of a discretization of the Navier-Stokes Equation. Since an implementation of the LBM generally works on a regular grid in memory, it is an ideal algorithm on which to perform cache optimizations in order to increase performance levels. This paper specifically examines the suitability of a 2D implementation of the LBM for cache awareness. In addition, a new storage format is introduced, which is designed to reduce the amount of memory required, and therefore the amount of memory traffic as well, and its suitability for cache optimizations.

In general, the use of cache optimization techniques such as blocking and padding greatly increased the performance of a standard implementation of the LBM. *3-way blocking* especially resulted in algorithms that ran at consistently significantly higher performance levels than non-optimized code. This effect was seen on an AMD Athlon architectures and Alpha A21164 and A21264 architectures, the results of which are thoroughly discussed in this paper. When tested on additional architectures, namely an AMD Athlon XP and an Intel Itanium 2, similar increases in performance were seen. Only on one architecture, namely the Intel Pentium 4 architecture did these cache optimizations not increase the performance level. The Intel P4 architecture may be performing prefetching of data in hardware [Int02], which would explain the observed behavior and obviate the need for cache optimizations such as loop blocking.

The introduction of the new storage format, here dubbed the *compressed grid* method greatly reduces the amount of memory required by the LBM while adding only very little computational complexity. The reduction in memory requirements translates directly into a reduction in memory traffic and therefore results in significant performance increases for all architectures. Once cache optimizations were applied to and LBM using compressed grid storage, it resulted in the highest performing code measured on the AMD and Alpha architectures. The Intel P4, once again, performed best with the non-optimized code.

Overall, the regular grid used by the LBM makes it a highly suitable algorithm on which to perform cache optimizations in itself. Using the compressed grid technique will also result in significant performance gains on almost all architectures. The application of the compressed grid technique to other algorithms using stencil accesses to regular grids and the resulting performance behavior remains to be further investigated.

Appendix A

Machine Specifications

A.1 Platform A

- CPU: AMD Athlon 700MHz
- Name: fauia28.informatik.uni-erlangen.de
- L1 Cache: 64KB Data, 64KB Instructions, 2-way set-associative
- L2 Cache: 512KB, direct-mapped, off chip
- TLB Size: 32 entries L1-TLB, 256 entries L2-TLB
- Operating System: Linux with Kernel 2.4.10
- Compiler: GNU g++ 3.2.2
- Compiler flags: `-O3 -ffast-math`

A.2 Platform B

- CPU: Intel Pentium 4 1.5GHz
- Name: fauia43.informatik.uni-erlangen.de
- L1 Cache: 8KB Data
- L2 Cache: 256KB
- TLB Size: 64 entries
- Operating System: Linux with Kernel 2.4.10
- Compiler: GNU g++ 3.2.2
- Compiler flags: `-O3 -ffast-math`

A.3 Platform C

- CPU: Alpha 21164
- Workstation: DEC PWS 500au
- Name: fauia30.informatik.uni-erlangen.de
- L1 Cache: 8KB Data, 8KB Instructions, direct mapped, line size 32 bytes
- L2 Cache: 96KB, 3-way set-associative, line size 32/64 bytes

- L3 Cache: 4MB, direct-mapped, off chip module
- TLB Size: 128 entries
- Operating system: Compaq Tru64 UNIX V5.0A
- Compiler: Compaq C++ V6.5-014
- Compiler flags: `-O4 -arch host -tune host -fast -g3`

A.4 Platform D

- CPU: Alpha 21264
- Workstation: Compaq XP 1000 professional
- Name: fauia35.informatik.uni-erlangen.de
- L1 Cache: 64KB Data, 64KB Instructions, 2-way set-associative, line size 32 bytes
- L2 Cache: 4MB, direct-mapped, off chip
- TLB Size: 128 entries
- Operating System: Digital UNIX V4.0F
- Compiler: Compaq C++ V6.5-014
- Compiler flags: `-O4 -arch host -tune host -fast -g3`

Appendix B

Implementations in Pseudocode

Contrary to what is written in the pseudo-code, the lattice itself is a dynamically allocated array of doubles, and all accesses performed by hand. An access to the "N" distribution function of the site located at (j,i) resembles something like this:

```
grid[numRows * CELLSIZE * i + CELLSIZE * j + N_OFF],
```

where CELLSIZE is a constant representing the number of doubles required by one cell, and N_OFF is a constant representing an offset to the double containing the value of the "N" distribution function. This results in fairly unreadable code however, so the pseudo-code uses a object oriented like notation when accessing values in the grid. This is done purely for readability purposes only. Despite this, the structure of the algorithm remains unchanged.

Additionally, since the original code was written in C++, all pseudocode assumes a column major access format.

B.1 3–Way Blocked Merged LBM

Algorithm B.1 3–Way Blocked Merged LBM

```
for tt = 1 to MAX_TIME by 2 * BS do
  // Special handling for the bottom rows
  for i = 1 to BS − 1 do
    for t = 0 to i − 1 by 1 do
      for j = 1 to n do
        // Stream_and_Collide(i-t,j)
      end for
    end for
  end for
  for ii = BS to m − BS + 1 by BS do
    // Special handling for beginning a block row
    for t = 0 to BS − 2 by 1 do
      for i = 0 to BS − 1 by 1 do
        for j = 1 to BS − 1 − t by 1 do
          // Stream_and_Collide(ii+i-t,j)
        end for
      end for
    end for
    // Handle main body using blocks
    for jj = BS to n − BS + 1 by BS do
      for t = 0 to BS − 1 by 1 do
        for i = 0 to BS − 1 by 1 do
          for j = 0 to BS − 1 by 1 do
            // Stream_and_Collide(ii+i-t,jj+j-t)
          end for
        end for
      end for
    end for
    // Special handling for finishing a block row
    s = n − [(n + 1)modBS] + 1;
    for t = 0 to BS − 1 by 1 do
      for i = 0 to BS − 1 by 1 do
        for j = s − t to n by 1 do
          // Stream_and_Collide(ii+i-t,j)
        end for
      end for
    end for
  end for
  // Special handling for the top rows
  s = m − [(m + 1)modBS] + 1;
  for t = 0 to BS − 1 by 1 do
    for i = s − t to m by 1 do
      for j = 1 to n by 1 do
        // Stream_and_Collide(m-t,j)
      end for
    end for
  end for
end for
```

B.2 Compressed Grid LBM

Algorithm B.2 Compressed Grid LBM

```
// For grid compression, an additional row and column are allocated.
site grid[n + 3, m + 3]
for i = 1 to i = n − 1 by 1 do
  for j = 1 to j = m − 1 by 1 do
    grid[i,j].init_values();
  end for
end for
for tt = 1 to MAX_TIME/2 by BS do
// First phase: proceed from top, right site to bottom left
for i = n + 1 to i = 2 by 1 do
  for j = m + 1 to j = 2 by 1 do
    if grid[i,j].type == "boundary" then
      grid[i,j].S = grid[i-2,j-1].N;
      grid[i,j].W = grid[i-1,j].E;
      grid[i,j].N = grid[i,j-1].S;
      grid[i,j].E = grid[i-1,j-2].W;
      grid[i,j].NE = grid[i,j].SW;
      grid[i,j].NW = grid[i,j-2].SE;
      grid[i,j].SE = grid[i-2,j].NW;
      grid[i,j].SW = grid[i-2,j-2].NE;
      // Make sure to copy the type flag as well!
      grid[i,j].TYPE = grid[i-1,j-1].TYPE
    else
      double rho;
      double u_x;
      double u_y;
      if grid[i,j].type == "acceleration" then
        rho = 1.0;
        u_x = 0.1;
        u_y = 0.0;
      else
        u_x = grid[i,j-1].E + grid[i-1,j-1].NE + grid[i+1,j-1].SE;
        u_y = grid[i-1,j].N + grid[i-1,j-1].NE + grid[i-1,j+1].NW;
        rho = grid[i,j-1].W + grid[i+1,j].S + grid[i+1,j+1].SW + grid[i,j].C - grid[i-1,j-1].NE + u_x + u_y;
        u_x = ( u_x - grid[i,j-1].W - grid[i-1,j+1].NW - grid[i+1,j+1].SW ) / rho;
        u_y = ( u_y - grid[i+1,j].S - grid[i+1,j-1].SE - grid[i+1,j+1].SW ) / rho;
      end if
      double u_sqr_trm = 3/2 · ( u_x · u_x + u_y · u_y );
      rho = 1.0/τ · 1.0/36.0 rho;
      dst[i,j].NE = (1.0 − 1.0/τ)src[i-2,j-2].NE + rho( 1.0 + 3.0(u_x+u_y) + 3.0(u_x+u_y)·3.0(u_x+u_y)·0.5 - u_sqr_trm );
      dst[i,j].SE = (1.0 − 1.0/τ)src[i,j-2].SE + rho( 1.0 + 3.0(u_x-u_y) + 3.0(u_x-u_y)·3.0(u_x-u_y)·0.5 - u_sqr_trm );
      dst[i,j].SW = (1.0 − 1.0/τ)src[i,j].SW + rho( 1.0 + 3.0(-u_x-u_y) + 3.0(-u_x-u_y)·3.0(-u_x-u_y)·0.5 - u_sqr_trm );
      dst[i,j].NW = (1.0 − 1.0/τ)src[i-2,j]].NW + rho( 1.0 + 3.0(-u_x+u_y) + 3.0(-u_x+u_y)·3.0(-u_x+u_y)·0.5 - u_sqr_trm );
      rho=4.0·rho;
      dst[i,j].N = (1.0 − 1.0/τ)src[i-2,j-1].N + rho( 1.0 + 3.0·u_y + 9.0/2.0(u_y · u_y) - u_sqr_trm );
      dst[i,j].S = (1.0 − 1.0/τ)src[i,j-1].S + rho( 1.0 - 3.0·u_y + 9.0/2.0(u_y · u_y) - u_sqr_trm );
      dst[i,j].E = (1.0 − 1.0/τ)src[i-1,j-2].E + rho( 1.0 + 3.0·u_x + 9.0/2.0(u_x · u_x) - u_sqr_trm );
      dst[i,j].W = (1.0 − 1.0/τ)src[i-1,j-2].W + rho( 1.0 - 3.0·u_x + 9.0/2.0(u_x · u_x) - u_sqr_trm );
      dst[i,j].C = (1.0 − 1.0/τ)src[i-1,j-1].C + 4.0·rho( 1.0 - u_sqr_trm );
    end if
  end for
end for
```

Algorithm B.2 continued.

```
// Second phase: proceed from bottom, left site to top right.
for i = 1 to i = n by 1 do
  for j = 1 to j = m by 1 do
    if grid[i,j].type == "boundary" then
      grid[i,j].S = grid[i,j+1].N;
      grid[i,j].W = grid[i+1,j].E;
      grid[i,j].N = grid[i+2,j+1].S;
      grid[i,j].E = grid[i+1,j+2].W;
      grid[i,j].SW = grid[i,j].NE;
      grid[i,j].NW = grid[i+2,j].SE;
      grid[i,j].SE = grid[i,j+2].NW;
      grid[i,j].NE = grid[i+2,j+2].SW;
      // Make sure to copy the type flag as well!
      grid[i,j].TYPE = grid[i+1,j+1].TYPE
    else
      double rho;
      double u_x;
      double u_y;
      if grid[i,j].type == "acceleration" then
        rho = 1.0;
        u_x = 0.1;
        u_y = 0.0;
      else
        u_x = grid[i,j-1].E + grid[i-1,j-1].NE + grid[i+1,j-1].SE;
        u_y = grid[i-1,j].N + grid[i-1,j-1].NE + grid[i-1,j+1].NW;
        rho = grid[i,j-1].W + grid[i+1,j].S + grid[i+1,j+1].SW + grid[i,j].C - grid[i-1,j-1].NE + u_x + u_y;
        u_x = ( u_x - grid[i,j-1].W - grid[i-1,j+1].NW - grid[i+1,j+1].SW ) / rho;
        u_y = ( u_y - grid[i+1,j].S - grid[i+1,j-1].SE - grid[i+1,j+1].SW ) / rho;
      end if
      double u_sqr_trm = 3/2 · ( u_x · u_x + u_y · u_y );
      rho = 1.0/τ · 1.0/36.0 rho;
      dst[i,j].NE = (1.0 − 1.0/τ)src[i,j].NE + rho( 1.0 + 3.0(u_x+u_y) + 3.0(u_x+u_y)·3.0(u_x+u_y)·0.5 - u_sqr_trm );
      dst[i,j].SE = (1.0 − 1.0/τ)src[i+2,j].SE + rho( 1.0 + 3.0(u_x-u_y) + 3.0(u_x-u_y)·3.0(u_x-u_y)·0.5 - u_sqr_trm );
      dst[i,j].SW = (1.0 − 1.0/τ)src[i+2,j+2].SW + rho( 1.0 + 3.0(-u_x-u_y) + 3.0(-u_x-u_y)·3.0(-u_x-u_y)·0.5 - u_sqr_trm );
      dst[i,j].NW = (1.0 − 1.0/τ)src[i,j+2]].NW + rho( 1.0 + 3.0(-u_x+u_y) + 3.0(-u_x+u_y)·3.0(-u_x+u_y)·0.5 - u_sqr_trm );
      rho=4.0·rho;
      dst[i,j].N = (1.0 − 1.0/τ)src[i,j+1].N + rho( 1.0 + 3.0·u_y + 9.0/2.0(u_y · u_y) - u_sqr_trm );
      dst[i,j].S = (1.0 − 1.0/τ)src[i+2,j+1].S + rho( 1.0 - 3.0·u_y + 9.0/2.0(u_y · u_y) - u_sqr_trm );
      dst[i,j].E = (1.0 − 1.0/τ)src[i+1,j].E + rho( 1.0 + 3.0·u_x + 9.0/2.0(u_x · u_x) - u_sqr_trm );
      dst[i,j].W = (1.0 − 1.0/τ)src[i+1,j+2].W + rho( 1.0 - 3.0·u_x + 9.0/2.0(u_x · u_x) - u_sqr_trm );
      dst[i,j].C = (1.0 − 1.0/τ)src[i+1,j+1].C + 4.0·rho( 1.0 - u_sqr_trm );
    end if
  end for
end for
end for
```

B.3 1–Way Blocked Compressed Grid LBM

Algorithm B.3 1–Way Blocked Compressed Grid LBM

```
for tt = 1 to MAX_TIME/2 by BS do
  // First phase: proceed from top, right site to bottom left site.
  // Special handling for the top rows
  for i = 0 to BS − 2 by 1 do
    for t = 0 to i by 1 do
      for j = n + 1 to 2 by −1 do
        // Stream_and_Collide_1(m+1-i+2t,j+t)
      end for
    end for
  end for
  // Blocking
  for i = m + 2 − BS to 2 by −1 do
    for t = 0 to BS − 1 by 1 do
      for j = n + 1 to 2 by −1 do
        // Stream_and_Collide_1(i+2t,j+t)
      end for
    end for
  end for
  // Special handling for the bottom rows
  for i = 0 to BS − 2 by 1 do
    for t = 0 to BS − i − 2 by 1 do
      for j = n + 1 to 2 by −1 do
        // Stream_and_Collide_1(3+i+2t,j+t+1)
      end for
    end for
  end for
  // Second phase: proceed from bottom, left site to top right.
  // Special handling for the bottom rows
  for i = 0 to BS − 2 by 1 do
    for t = 0 to i by 1 do
      for j = BS to n − 1 + BS by 1 do
        // Stream_and_Collide_2(BS+i-2t,j-t)
      end for
    end for
  end for
  // Blocking
  for i = 2 ∗ BS − 1 to m + BS − 1 by 1 do
    for t = 0 to BS − 1 by 1 do
      for j = BS to n − 1 + BS by 1 do
        // Stream_and_Collide_2(i-2t,j-t)
      end for
    end for
  end for
  // Special handling for the top rows
  for i = 0 to BS − 2 by 1 do
    for t = 0 to BS − i − 2 by 1 do
      for j = BS to n − 1 + BS by 1 do
        // Stream_and_Collide_2(m+BS-i-2t-2,j-i-t-1)
      end for
    end for
  end for
end for
```

B.4 3-Way Blocked Compressed Grid LBM

Algorithm B.4 3–Way Blocked Grid Compressed LBM

```
for tt = 1 to MAX_TIME/2 by BS do
// First phase: proceed from top, right site to bottom left
// Special handling for the top rows
for i = 0 to BS − 2 by 1 do
  for t = 0 to i by 1 do
    for j = n + 1 + t to 2 + t by −1 do
      // Stream_and_Collide_1(m+1-i+2t,j)
    end for
  end for
end for
// Process main body using rows of blocks.
for ii = m − BS + 2 to BS + 1 by −BS do
  // Special handling for the beginning of a block row
  for t = 0 to BS − 2 by 1 do
    for i = BS − 1 to 0 by −1 do
      for j = n + 1 + t to n + 3 + 2t − BS by −1 do
        // Stream_and_Collide_1(ii-i+2t,j)
      end for
    end for
  end for
  // Handle main body using blocks
  for jj = n − BS + 2 to BS + 1 by −BS do
    for t = 0 to BS − 1 by 1 do
      for i = BS − 1 to 0 by −1 do
        for j = BS − 1 to 0 by −1 do
          // Stream_and_Collide_1(ii-i+2t,jj-j+2t)
        end for
      end for
    end for
  end for
  // Special handling for finishing a block row
  for t = 0 to BS − 1 by 1 do
    for i = 0 to BS − 1 by 1 do
      for j = jj − 2t to 2 + t by −1 do
        // Stream_and_Collide_1(ii-i-2t,j)
      end for
    end for
  end for
end for
// Special handling for the bottom rows
for t = 0 to BS − 1 by 1 do
  for i = 2 + t to ii − 2t by 1 do
    for j = n + 1 + t to 2 + t by −1 do
      // Stream_and_Collide_1(i,j)
    end for
  end for
end for
```

Algorithm B.4 continued.

```
// Second phase: proceed from bottom, left site to top right.
// Special handling for the bottom rows
for i = 0 to BS − 2 by 1 do
  for t = 0 to i by 1 do
    for j = BS − t to n − 1 + BS − t by 1 do
      // Stream_and_Collide_2(BS+i-2t,j)
    end for
  end for
end for
// Process main body using rows of blocks.
for ii = 2 ∗ BS − 1 to m by BS do
  // Special handling for the beginning of a block row
  for t = 0 to BS − 2 by 1 do
    for i = 0 to BS − 1 by 1 do
      for j = BS − t to 2 ∗ BS − 2t − 2 by 1 do
        // Stream_and_Collide_2(ii+i-t,j)
      end for
    end for
  end for
  // Handle main body using blocks
  for jj = 2 ∗ BS − 1 to n by BS do
    for t = 0 to BS − 1 by 1 do
      for i = 0 to BS − 1 by 1 do
        for j = 0 to BS − 1 by 1 do
          // Stream_and_Collide_2(ii+i-2t,jj+j-2t)
        end for
      end for
    end for
  end for
  // Special handling for finishing a block row
  for t = 0 to BS − 1 by 1 do
    for i = 0 to BS − 1 by 1 do
      for j = jj − 2t to n + BS − 1 − t by 1 do
        // Stream_and_Collide_2(ii+i-t,j)
      end for
    end for
  end for
end for
// Special handling for the top rows
for t = 0 to BS − 1 by 1 do
  for i = n + BS − 1 − t to ii − 2t by −1 do
    for j = BS − t to n − 1 + BS − t by 1 do
      // Stream_and_Collide_2(i,j)
    end for
  end for
end for
end for
```

Bibliography

[ABD+97] J.M. Anderson, L.M. Berc, J. Dean, S. Ghemawat, M.R. Henzinger, S.A. Leung, R.L. Sites, M.T. Vandevoorde, C.A. Waldspurger, and W.E. Weihl. Continuous Profiling: Where Have All the Cycles Gone? In *Proc. of the 16th ACM Symposium on Operating System Principles*, pages 1–14, St. Malo, France, 1997.

[AK02] Randy Allen and Ken Kennedy. *Optimizing Compilers for Modern Architectures.* Academic Press, 2002.

[BDG+00] 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.

[BDQ98] 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, Nevada, USA, 1998.

[BF01] R.L. Burden and J.D. Faires. *Numerical Analysis.* Brooks/Cole, 2001.

[CDE94] Shiyi Chen, Gary D. Doolen, and Kenneth G. Eggert. Lattice boltzmann fluid dynamics. *Los Alamos Science*, 22, 1994.

[GH01] Stefan Goedecker and Adolfy Hoisie. *Performance Optimization of Numerically Intensive Codes.* SIAM, 2001.

[HL97] Xiaoyi He and Li-Shi Luo. Lattice Boltzmann Model for the Incompressible Navier-Stokes Equation. *Journal of Statistical Physics*, pages 927–944, 1997.

[Int02] Intel Corporation. *Intel Itanium2 Processor Reference Manual*, 2002. Document Number: 251110–001.

[LRW91] M.S. Lam, E.E. Rothberg, and M.E. Wolf. The Cache Performance and Optimizations of Blocked Algorithms. In *Proc. of the Fourth Int. Conference on Architectural Support for Programming Languages and Operating Systems*, pages 63–74, Palo Alto, California, USA, 1991.

[MSY02] Renwei Mei, Wei Shyy, and Dazhi Yu. Lattice Boltzmann Method for 3D Flows with Curved Boundary. Technical Report NASA/CR-2002-211657, ICASE, NASA Langley Research Center, Hampton, Virginia, June 2002.

[RT98] 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.

[Sto02] Jon Stokes. Understanding cpu caching and performance. Technical report, Ars Technica, LLC, July 2002.

[Wei01] C. Weiß. *Data Locality Optimizations for Multigrid Methods on Structured Grids.* PhD thesis, Lehrstuhl für Rechnertechnik und Rechnerorganisation, Institut für Informatik, Technische Universität München, Munich, Germany, December 2001.

[WG00] Dieter A. Wolf-Gladrow. *Lattice-Gas Cellular Automata and Lattice Boltzmann Models*. Springer, 2000.

[WL91] M.E. Wolf and M.S. Lam. A Data Locality Optimizing Algorithm. In *Proc. of the SIGPLAN'91 Symposium on Programming Language Design and Implementation*, volume 26 of *SIGPLAN Notices*, pages 33–44, Toronto, Canada, 1991.