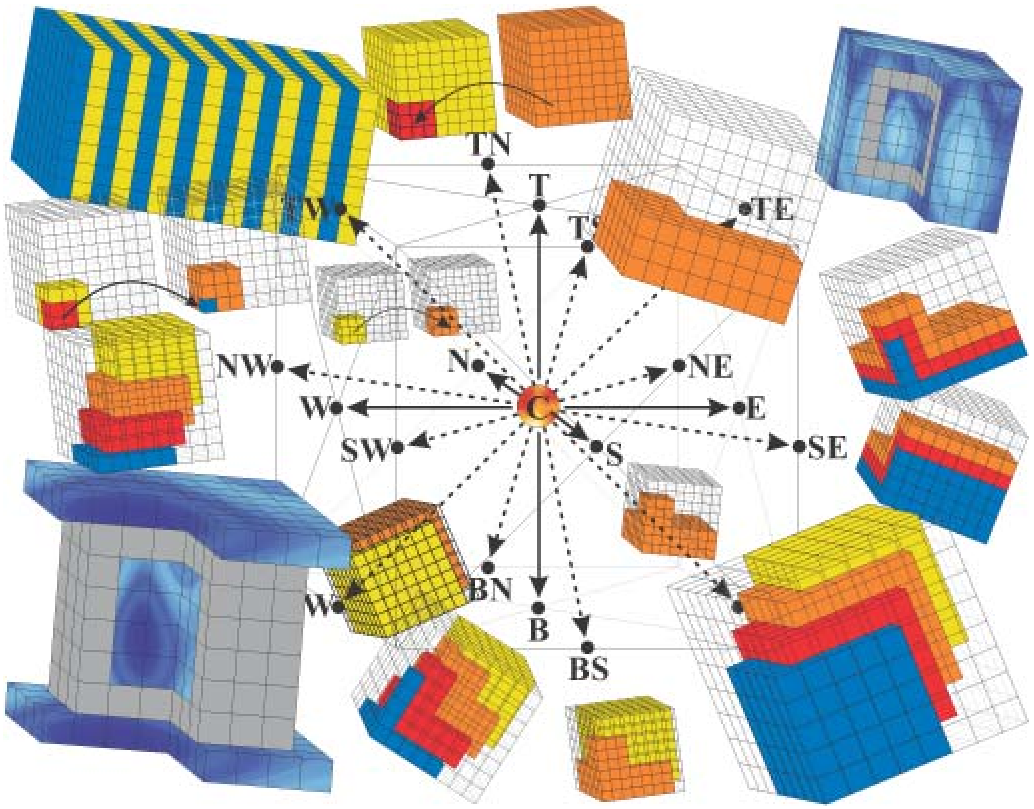
FRIEDRICH-ALEXANDER-UNIVERSITÄT ERLANGEN-NÜRNBERG
INSTITUT FÜR INFORMATIK (MATHEMATISCHE MASCHINEN UND DATENVERARBEITUNG)

Lehrstuhl für Informatik 10 (Systemsimulation)

Cache Optimizations for the Lattice Boltzmann Method in 3D

Klaus Iglberger



Bachelor Thesis

Cache Optimizations for the Lattice Boltzmann Method in 3D

Klaus Iglberger

Bachelor Thesis

Aufgabensteller:	Prof. Dr. Ulrich Rüde
Betreuer:	Dipl.–Phys. T. Pohl, Dipl.–Inf. M. Kowarschik
	Dipl.–Ing. F. Deserno
Bearbeitungszeitraum:	Mai 2003 – August 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 16. September 2003 ....

Contents

List of Figures

Abstract

The simulation of fluid dynamics has been a major topic of research for the last fifty years. One of the most modern techniques to simulate fluids is the Lattice Boltzmann Method (LBM), which evolved from the techniques of lattice cellular gas automata. Compared to earlier simulation methods, the LBM provides stable and efficient numerical calculations for the macroscopic behaviour of fluids, although describing the fluid in a microscopic way. Space is divided into a regular grid of cells, where each cell holds information about the behaviour of the fluid. The additional advantage of easy parallelization contributed to the popularity of the LBM. One of the major drawbacks for virtually all cellular automata is the great dependency of their performance on memory architectures, especially for the 3D case. Since the gap between CPU speed and memory performance grows bigger each year, a lot of work has to be done to speed up the LBM in terms of memory behaviour. Since it has already been shown that common cache optimization techniques can be adapted to the LBM in 2D, this thesis will show, that implementing these techniques in 3D can also efficiently reduce performance drawbacks due to the memory bottleneck. Additionally, this thesis gives insight into improved software engineering techniques, which show, that structured programming and performance can be efficiently combined.

Chapter 1

Introduction

1.1 The Lattice Boltzmann Method

An early method to simulate computational dynamic fluids was based on solving the governing partial differential equations numerically, in most implementations the time-dependent Navier-Stokes equations. The idea behind this approach is to discretize the computational domain using finite differences, finite elements, or finite volumes, to derive algebraic systems of equations and to solve these systems numerically [WPKR03].

The **lattice gas cellular automata** divide time and space into steps to form a lattice and discretize the fluid as particles, which are positioned on certain points, called lattice sites or cells. Within the cells, the current state of the cell is defined by an array of floating-point numbers that represent the distribution functions w.r.t. the discrete directions of velocity [WPKR03].

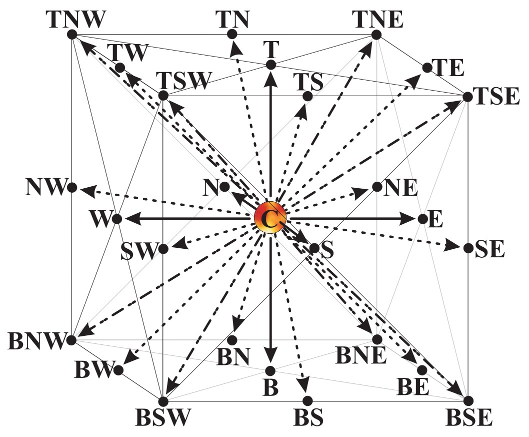
One of the methods relying on this microscopic model is the **Lattice Boltzmann Method**. During one time step, the particles move according to the distribution functions into the proximate cells and collide with all other particles, that stream into this cell from different directions. This collision is simulated by solving the Lattice Boltzmann equation, which conserves mass, momentum and energy of the particles. One of the greatest numerical advantages of this model is obviously, that only adjacent cells can influence each other. This particle-based microscopic model is a good approximation and is a very efficient method to simulate macroscopic behaviour of dynamic fluids.

1.2 The Lattice Boltzmann Method in 3D

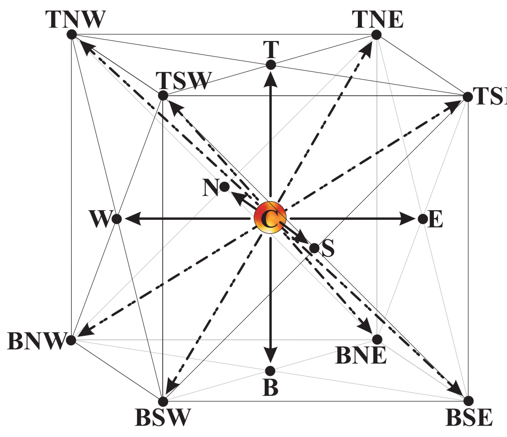
The Lattice Boltzmann Method in 3D is very similar to the LBM in 2D [Wil03] and the extension to 3D is very straightforward: just as in 2D, a 3D lattice has to be chosen, consisting of 3D lattice sites, called cells, which contain the distribution functions of the particles. As in 2D, these distribution functions describe the distribution or velocity of the particles in the cell and therefore represent the flowing direction of the discretized fluid. For the 3D case, three different sets of distribution functions are quite common.

In the D3Q27 model, every possible direction is used: one center, six for the axes in both directions respectively, twelve for all combinations of two axes and eight for all combinations of three axes (see Figure 1.1(a)). Though this model promises the best accuracy in terms of approximation, it does not show the best numerically stable behaviour. In contrast, the D3Q15 model consist only of 15 velocities (one center, six for the axes and eight for all combinations of three axes) and promises a very fast LBM, but due to the small number of distribution functions, this model tends to anisotropic effects (see Figure 1.1(b)).

The model used in this thesis is the D3Q19 model: it consists of 19 velocities and combines fast calculation and good approximation (see Figure 1.2). It also shows less instability than the other



(a) Lattice site with 27 velocities



(b) Lattice site with 15 velocities

Figure 1.1: D3Q27 and D3Q15 model of the 3D LBM.

two models. The 19 velocity values are implemented as double precision values (in the C language a double value with 8 bytes). Together with a double precision value for the general behaviour of the cell, called flag, which can indicate a fluid cell, an obstacle cell or an acceleration cell, a cell contains 20 double values (see Figure 1.3).

The 19 velocities consist of the center of the cell, where a velocity of zero can be assumed, the six velocities along the x, y and z axis in both directions respectively and all combinations of two directions, all together 12. As defined in Figure 1.2, the velocities will henceforth be called C (center), N (north), S (south), W (west), E (east), T (top), B (bottom), NW (northwest), NE (northeast), SW (southwest), SE (southeast), TN (top north), TS (top south), TW (top west), TE (top east), BN (bottom north), BS (bottom south), BW (bottom west), and BE (bottom east). In [WG00] the lattice velocities c_i are defined as follows:

$$c_C = (0,0,0) \tag{1.1}$$

$$c_{W,E}, c_{N,S}, c_{T,B} = (\pm1,0,0),(0,\pm1,0),(0,0,\pm1) \tag{1.2}$$

$$c_{NW,NE,SW,SE}, c_{TW,TE,BW,BE}, c_{TN,TS,BN,BS} = (\pm1,\pm1,0),(\pm1,0,\pm1),(0,\pm1,\pm1) \tag{1.3}$$

As in the 2D case, for velocities near or equal zero, a global equilibrium distribution W_i has to be defined for all velocities. These values are used in the case of a very small flow in the cell or as initial state. The sum of the W_i in a cell is 1. In the D3Q19 model, the W_i are defined as follows:

$$W_0 = 1/3 \tag{1.4}$$

$$W_1 = 1/18 \tag{1.5}$$

$$W_2 = 1/36 \tag{1.6}$$

W_0 is used only for the center, W_1 for N, S, W, E, T and B, and W_2 counts for NW, NE, SW, SE, TN, TS, TW, TE, BN, BS, BW, and BE. The W_i can be seen as weights to the velocities. Therefore all similar velocities are weighted equally for the reason of symmetry.

$$F_i^{(0)} = W_0 \cdot \rho \cdot [1 - \frac{3}{2}\frac{u^2}{c^2}] \tag{1.7}$$

$$F_i^{(0)} = W_1 \cdot \rho \cdot [1 + 3\frac{c_i \cdot u}{c^2} + \frac{9}{2}\frac{(c_i \cdot u)^2}{c^4} - \frac{3}{2}\frac{u^2}{c^2}] \tag{1.8}$$

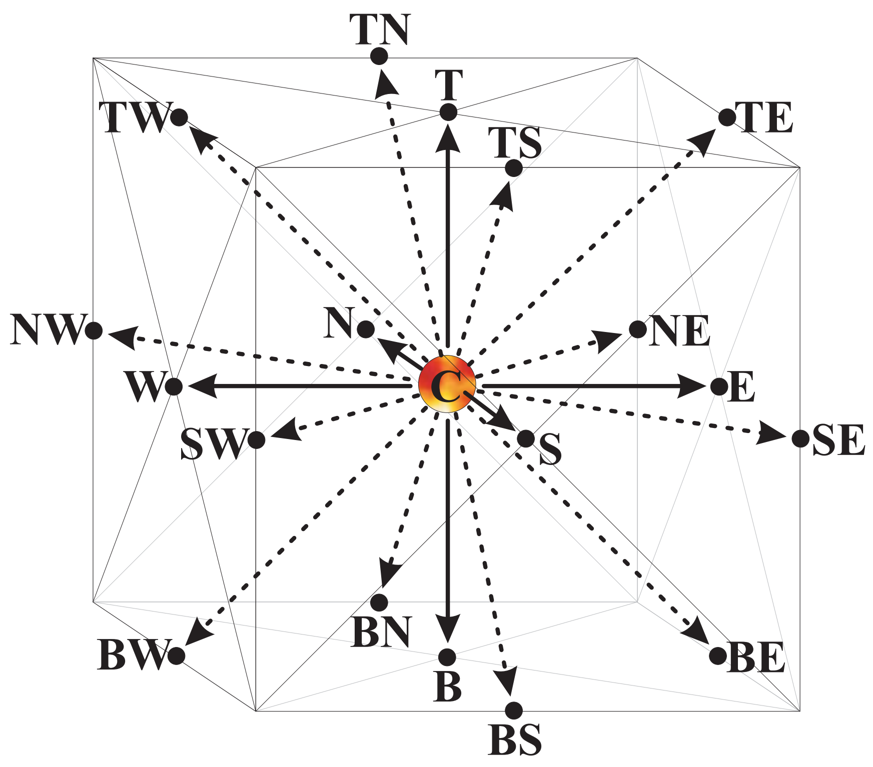


Figure 1.2: D3Q19 model of the 3D LBM. This model is used for this thesis.

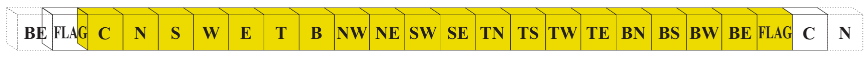


Figure 1.3: A cell consists of 20 values, which lie consecutively in the memory. The same holds for several cells of the grid, which lie consecutively in front and behind each other.

$$F_i^{(0)} \quad = \quad W_2 \cdot \rho \cdot [1 + 3\frac{c_i \cdot u}{c^2} + \frac{9}{2}\frac{(c_i \cdot u)^2}{c^4} - \frac{3}{2}\frac{u^2}{c^2}] \tag{1.9}$$

Equations 1.7 to 1.9 show the local equilibrium distribution functions, which are used in the Lattice Boltzmann Equation to calculate the new equilibrium distribution. For velocities of zero, these functions result in W_0, W_1 and W_2.

$$F_i(x + c_i\Delta t, t + \Delta t) - F_i(x,t) \quad = \quad (1-\omega)F_i(x,t) + \omega F_i^{(0)}(x,t) \tag{1.10}$$

Equation 1.10 shows the slightly simplified **Lattice Boltzmann Equation**: The difference between the old velocity value and the new, already streamed velocity value is the weighted summation of the equilibrium distribution $F_i^{(0)}(x,t)$ and the non-equilibrium distribution $F_i(x,t)$. This equation can be transformed to a simpler, more intuitive set of two equations:

$$\tilde{F}_i(x, t+\Delta t) \quad = \quad F_i(x + c_i\Delta t, t) \tag{1.11}$$

$$F_i(x, t+\Delta t) \quad = \quad (1-\omega)\tilde{F}_i(x, t+\Delta t) + \omega\tilde{F}_i^{(0)}(x, t+\Delta t) \tag{1.12}$$

For each cell update, these two equations have to be calculated: Equation (1.11) is called the **stream step**. In this step, the velocities from the 18 surrounding cells according to Figure 1.2

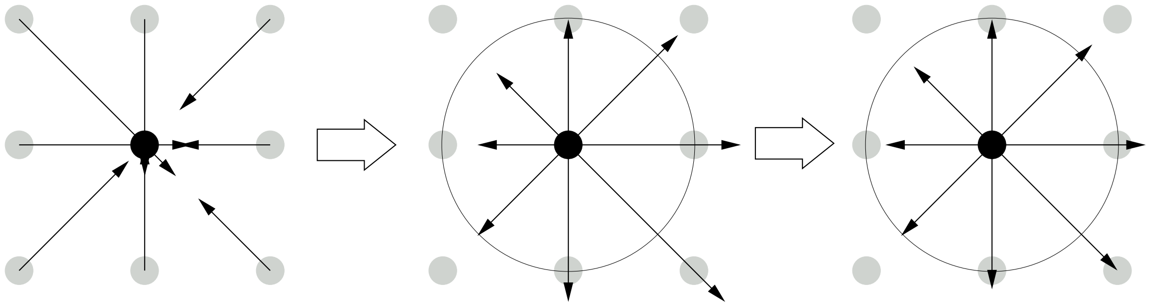

(a) Stream and Collide step of one cell update

Figure 1.4: A lattice site is updated in two steps: the stream- and the collide step.

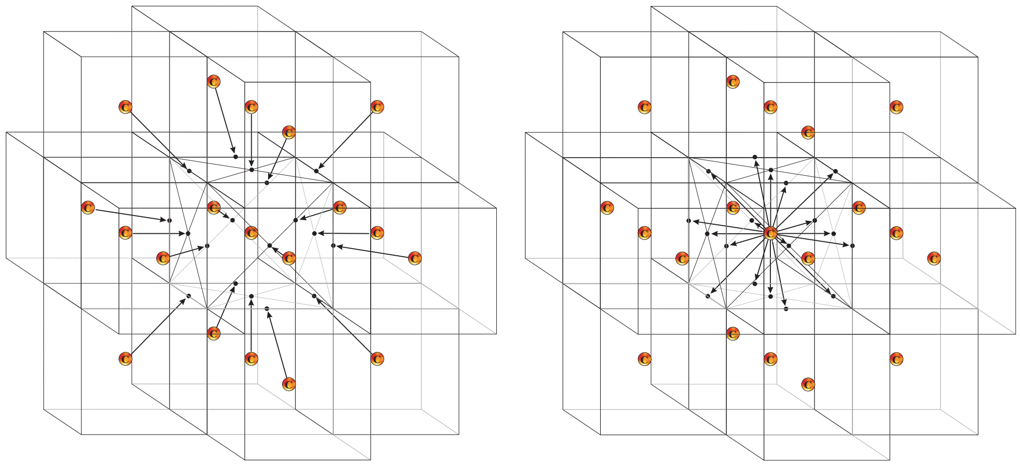

(a) Stream step of the 3D Lattice Boltzmann (b) Collide step of the 3D Lattice Boltzmann

Figure 1.5: Stream and collide step in the case of 3D Lattice Boltzmann.

are gathered and written to the cell being updated. The center does not have to be moved and represents the 19th velocity value. Equation (1.12) is called the **collide step**. This equation is executed afterwards and smoothes the distribution functions towards a total sum of 1. Figure 1.4 illustrates the two steps of a cell update in a clearer 2D version: The first figure shows the stream step, which results in an unbalanced distribution (second figure). The collide step takes the collision of the particle velocities into account and therefore smoothes the distribution towards the sum of 1, which is represented by the circle. Figure 1.5 illustrates the stream and collide step in 3D: the procedures during the steps are exactly the same as in the 2D case, but much harder to imagine.

Obviously, the order of the stream and the collide step can be reversed: in the order collide-stream, the update of a cell would start by calculating the new distribution functions and end with spreading the new values to the surrounding neighbours. In contrast to the stream-collide order, where the old distribution functions are "pulled" into the updating cell from the surrounding neighbours, the newly calculated distribution functions are "pushed" to the neighbouring cells. Though only the order of stream-collide was used in [Wil03], both versions will be benchmarked in this thesis. One reason for this double effort is testing, which order provides faster code, the second reason is, that from a software engineering point of view, the switching between a stream-collide and collide-stream version can be done by replacing a single macro. For these software engineering techniques are of major interest in this thesis, the benchmarking of both versions gives a lot more insight in the performance of Lattice Boltzmann code with nearly no developing effort. For the following code examples, the stream-collide order was preferred for two reasons: first, this order was also preferred in [Wil03], which will provide a better comparison of the codes and second, because the stream-collide versions is probably the more intuitive version of the LBM. But in contrast to cache optimizations like 3-way blocking or 4-way blocking, which will be introduced in Chapter 4, the transformation from the stream-collide to the collide-stream order changes the results of the algorithm depending on the starting conditions. This means, that in contrast to all other cache optimization techniques, this transformation results in different numerical results!

In summary, the LBM consists of three components [WG00]:

1. the lattice
2. the equilibrium distributions
3. the kinetic equation for the collision

After implementing the LBM, Equation (1.11) results in Algorithm 1.1. Here, all values from the 18 surrounding cells are gathered and written to the cell, which is currently updated. Equation (1.12) results in Algorithm 1.2, where the complete cell update takes place. The first part takes care of the obstacle cells: all directions are exactly reversed. This implementation of an obstacle cell is known as **no-slip** condition. The second part deals with the acceleration cells. The characteristic of an acceleration cell is, that here the velocity does not have to be computed, but rather is given from an external source. So here, other than in the third part, fixed values are used for the density rho and the three velocities u_x, u_y, and u_z. The third part is for the fluid cells. Here the values for the density and the velocities have to be computed. The last part corresponds to Equation (1.12): this part sets the equilibrium distribution against the non-equilibrium distribution and computes new values for the 19 velocities of the updated cell.

For the sake of completeness, the computations of rho (1.3), u_x (1.4), u_y (1.5) and u_z (1.6) are mentioned. The implementation of these functions is the exact continuation of the Lattice Boltzmann in 2D, as can be seen in [Wil03]. For more detailed information about the LBM, see [Suc01].

Algorithm 1.1 Stream step of the LBM in 3D

```
for $i = 1$ to $dim$ by 1 do
  for $j = 1$ to $dim$ by 1 do
    for $k = 1$ to $dim$ by 1 do
      dst[i][j][k][N] = src[i][j-1][k][N]
      dst[i][j][k][S] = src[i][j+1][k][S]
      dst[i][j][k][W] = src[i+1][j][k][W]
      dst[i][j][k][E] = src[i-1][j][k][E]
      dst[i][j][k][T] = src[i][j][k-1][T]
      dst[i][j][k][B] = src[i][j][k+1][B]
      dst[i][j][k][NW] = src[i+1][j-1][k][NW]
      dst[i][j][k][NE] = src[i-1][j-1][k][NE]
      dst[i][j][k][SW] = src[i+1][j+1][k][SW]
      dst[i][j][k][SE] = src[i-1][j+1][k][SE]
      dst[i][j][k][TN] = src[i][j-1][k-1][TN]
      dst[i][j][k][TS] = src[i][j+1][k-1][TS]
      dst[i][j][k][TW] = src[i+1][j][k-1][TW]
      dst[i][j][k][TE] = src[i-1][j][k-1][TE]
      dst[i][j][k][BN] = src[i][j-1][k+1][BN]
      dst[i][j][k][BS] = src[i][j+1][k+1][BS]
      dst[i][j][k][BW] = src[i+1][j][k+1][BW]
      dst[i][j][k][BE] = src[i-1][j][k+1][BE]
    end for
  end for
end for
```

Algorithm 1.2 Collide step of the LBM in 3D

```
for $i = 1$ to $dim$ by 1 do
  for $j = 1$ to $dim$ by 1 do
    for $k = 1$ to $dim$ by 1 do
      if FLAG == OBSTACLE then
        SWAP(dst[i][j][k][N], dst[i][j][k][S])
        SWAP(dst[i][j][k][W], dst[i][j][k][E])
        SWAP(dst[i][j][k][T], dst[i][j][k][B])
        SWAP(dst[i][j][k][NW], dst[i][j][k][SE])
        SWAP(dst[i][j][k][NE], dst[i][j][k][SW])
        SWAP(dst[i][j][k][TN], dst[i][j][k][BS])
        SWAP(dst[i][j][k][TS], dst[i][j][k][BN])
        SWAP(dst[i][j][k][TW], dst[i][j][k][BE])
        SWAP(dst[i][j][k][TE], dst[i][j][k][BW])
        continue;
      end if
      if FLAG == ACCELERATION then
        rho = 1.0
        u_x = X_VELOCITY
        u_y = Y_VELOCITY
        u_z = Z_VELOCITY
      else
        rho = Rho()
        u_x = U_x()
        u_y = U_y()
        u_z = U_z()
      end if
      u_sqr_trm = 1.5 · (u_x · u_x + u_y · u_y + u_z · u_z)
      dst[i][j][k][C] = (1.0-OMEGA) · dst[i][j][k][C] +
           + OMEGA · (W_0 · rho · (1.0-u_sqr_trm))
      dst[i][j][k][N] = (1.0-OMEGA) · dst[i][j][k][N] +
           + OMEGA · (W_1 · rho · (1.0+3.0 · u_y+4.5 · (u_y · u_y)-u_sqr_trm))
      dst[i][j][k][S] = (1.0-OMEGA) · dst[i][j][k][S] +
           + OMEGA · (W_1 · rho · (1.0-3.0 · u_y+4.5 · (u_y · u_y)-u_sqr_trm))
      dst[i][j][k][W] = (1.0-OMEGA) · dst[i][j][k][W] +
           + OMEGA · (W_1 · rho · (1.0-3.0 · u_x+4.5 · (u_x · u_x)-u_sqr_trm))
      dst[i][j][k][E] = (1.0-OMEGA) · dst[i][j][k][E] +
           + OMEGA · (W_1 · rho · (1.0+3.0 · u_x+4.5 · (u_x · u_x)-u_sqr_trm))
      dst[i][j][k][T] = (1.0-OMEGA) · dst[i][j][k][T] +
           + OMEGA · (W_1 · rho · (1.0+3.0 · u_z+4.5 · (u_z · u_z)-u_sqr_trm))
      dst[i][j][k][B] = (1.0-OMEGA) · dst[i][j][k][B] +
           + OMEGA · (W_1 · rho · (1.0-3.0 · u_z+4.5 · (u_z · u_z)-u_sqr_trm))
      dst[i][j][k][NW] = (1.0-OMEGA) · dst[i][j][k][NW]
           + OMEGA · (W_2 · rho · (1.0+3.0 · (u_y-u_x)+4.5 · ((u_y-u_x) · (u_y-u_x))-u_sqr_trm))
      dst[i][j][k][NE] = (1.0-OMEGA) · dst[i][j][k][NE] +
           + OMEGA · (W_2 · rho · (1.0+3.0 · (u_y+u_x)+4.5 · ((u_y+u_x) · (u_y+u_x))-u_sqr_trm))
      dst[i][j][k][SW] = (1.0-OMEGA) · dst[i][j][k][SW] +
           + OMEGA · (W_2 · rho · (1.0+3.0 · (-u_y-u_x)+4.5 · ((-u_y-u_x) · (-u_y-u_x))-u_sqr_trm))
      dst[i][j][k][SE] = (1.0-OMEGA) · dst[i][j][k][SE] +
           + OMEGA · (W_2 · rho · (1.0+3.0 · (-u_y+u_x)+4.5 · ((-u_y+u_x) · (-u_y+u_x))-u_sqr_trm))
      dst[i][j][k][TN] = (1.0-OMEGA) · dst[i][j][k][TN] +
           + OMEGA · (W_2 · rho · (1.0+3.0 · (u_y+u_z)+4.5 · ((u_y+u_z) · (u_y+u_z))-u_sqr_trm))
      dst[i][j][k][TS] = (1.0-OMEGA) · dst[i][j][k][TS] +
           + OMEGA · (W_2 · rho · (1.0+3.0 · (-u_y+u_z)+4.5 · ((-u_y+u_z) · (-u_y+u_z))-u_sqr_trm))
      dst[i][j][k][TW] = (1.0-OMEGA) · dst[i][j][k][TW] +
           + OMEGA · (W_2 · rho · (1.0+3.0 · (-u_x+u_z)+4.5 · ((-u_x+u_z) · (-u_x+u_z))-u_sqr_trm))
      dst[i][j][k][TE] = (1.0-OMEGA) · dst[i][j][k][TE] +
           + OMEGA · (W_2 · rho · (1.0+3.0 · (u_x+u_z)+4.5 · ((u_x+u_z) · (u_x+u_z))-u_sqr_trm))
      dst[i][j][k][BN] = (1.0-OMEGA) · dst[i][j][k][BN] +
           + OMEGA · (W_2 · rho · (1.0+3.0 · (u_y-u_z)+4.5 · ((u_y-u_z) · (u_y-u_z))-u_sqr_trm))
      dst[i][j][k][BS] = (1.0-OMEGA) · dst[i][j][k][BS] +
           + OMEGA · (W_2 · rho · (1.0+3.0 · (-u_y-u_z)+4.5 · ((-u_y-u_z) · (-u_y-u_z))-u_sqr_trm))
      dst[i][j][k][BW] = (1.0-OMEGA) · dst[i][j][k][BW] +
           + OMEGA · (W_2 · rho · (1.0+3.0 · (-u_x-u_z)+4.5 · ((-u_x-u_z) · (-u_x-u_z))-u_sqr_trm))
      dst[i][j][k][BE] = (1.0-OMEGA) · dst[i][j][k][BE] +
           + OMEGA · (W_2 · rho · (1.0+3.0 · (u_x-u_z)+4.5 · ((u_x-u_z) · (u_x-u_z))-u_sqr_trm))
    end for
  end for
end for
```

Algorithm 1.3 Calculation of the density rho

```
1: Procedure Rho() returns double
2:     double rho
3:     double* p = dst[i][j][k]
4:     rho = p[C] + p[N] + p[S] + p[W] + p[E] + p[T] + p[B] + p[NW] + N p[E] + p[SW] + p[SE]
5:     rho += p[TN] + p[TS] + p[TW] + p[TE] + p[BN] + p[BS] + p[BW] + p[BE]
6:     return rho
```

Algorithm 1.4 Calculation of the velocity u_x

```
1: Procedure U_x() returns double
2:     double u_x
3:     double* p = dst[i][j][k]
4:     u_x = (- p[W] + p[E] - p[NW] + p[NE] - p[SW] + p[SE] - p[TW] + p[TE] - p[BW] + p[BE]) / Rho()
5:     return u_x
```

Algorithm 1.5 Calculation of the velocity u_y

```
1: Procedure U_y() returns double
2:     double u_y
3:     double* p = dst[i][j][k]
4:     u_y = (p[N] - p[S] + p[NW] + p[NE] - p[SW] - p[SE] + p[TN] - p[TS] + p[BN] - p[BS]) / Rho()
5:     return u_y
```

Algorithm 1.6 Calculation of the velocity u_z

```
1: Procedure U_z() returns double
2:     double u_z
3:     double* p = dst[i][j][k]
4:     u_z = (p[T] - p[B] + p[TN] + p[TS] + p[TW] + p[TE] - p[BN] - p[BS] - p[BW] - p[BE]) / Rho();
5:     return u_z
```

1.3 Model Problem: Lid-Driven Cavity

To test and to benchmark the Lattice Boltzmann Algorithm in 3D, the same problem as in 2D was used: the lid-driven cavity. This test problem consists of a cubic, equilateral box, whose top, called lid, is endlessly dragged in the same direction over the fluid. The movement of the lid stimulates the fluid to move in a circular motion. To illustrate the general behaviour of the fluid in the 2D case, Figure 1.6 was taken from [Wil03]. This picture could well be a cross section of the 3D lid-driven cavity, as the fluid will behave very similarly to the 2D case.

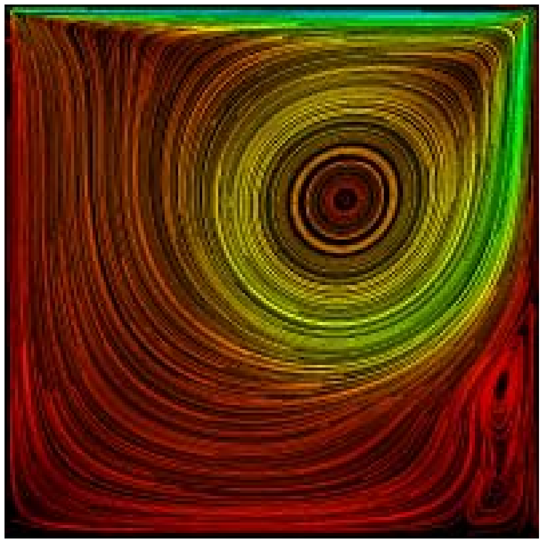


Figure 1.6: The Lid-driven Cavity Problem in 2D.

Although the problem is well known, the LBM algorithm was not adapted to this specific problem. It would be easy to improve the performance of the code by simply adjusting to the given structure of the problem, for example to use the information of the given obstacle structure to eliminate the if-statement, which checks whether the cell is an obstacle or something else. However, this kind of performance improvements were not the topic of this thesis. All optimization techniques used in this thesis were applied to the stream- and collide step as shown in Algorithms 1.1 and 1.2. Therefore, the code remains universal for most Lattice Boltzmann settings.

Chapter 2

Combining Software Modularity and Performance

The common prejudice is, that using a very structured and well ordered code has the flaw of bad performance, whereas fast code has to be written in an extremely complicated style. One of the initial ideas of this thesis was to provide a modular code infrastructure, which guarantees a very intuitive understanding of procedures, but still developing the fastest code possible. This can be done in C, which is explained in this chapter.

2.1 Using Macros

Algorithm 2.1 Unblocked and 4-way blocked LBM

```
// Unblocked LBM:
double src[dim,dim,dim]
double dst[dim,dim,dim]
for $t = 0$ to $t = timesteps$ by 1 do
  for $i = 1$ to $i = dim - 1$ by 1 do
    for $j = 1$ to $j = dim - 1$ by 1 do
      for $k = 1$ to $k = dim - 1$ by 1 do
        LBMSTEP
      end for
    end for
  end for
  swap( src, dst )
end for
```

```
// 4-way blocked LBM:
double src[dim,dim,dim]
double dst[dim,dim,dim]
for $tt = 0$ to $tt = timesteps$ by $TIMEBLOCK$ do
  for $ii = 1$ to $ii = dim$ by $BLOCKSIZE\_Z$ do
    for $jj = 1$ to $jj = dim$ by $BLOCKSIZE\_Y$ do
      for $kk = 1$ to $kk = dim$ by $BLOCKSIZE\_X$ do
        for $i = I\_START$ to $i = I\_END$ by 1 do
          for $j = J\_START$ to $j = J\_END$ by 1 do
            for $k = K\_START$ to $k = K\_END$ by 1 do
              LBMSTEP
            end for
          end for
        end for
        swap( src, dst )
      end for
    end for
  end for
end for
```

One way to guarantee a modular style in C is the intensive use of macros. In the example, the simplest version of Lattice Boltzmann is compared to a much more complicated version, the 4-way blocked Lattice Boltzmann implementation (see Algorithm 2.1).

The first eye-catching detail is, that the LBMSTEP macro still remains the same. By changing the order of computation nothing changes at the 'heart' of the algorithm. A cell is still updated in the same manner as before. This insight makes the development of faster versions much easier, for only the for-loops have to change, but the update kernel itself remains unchanged and can therefore be replaced by a macro. Obviously, two great advantages of software engineering can be obtained by this approach: first, the simplification of the code, which makes the code more intuitive to read, and second, the modularity and reuse of the code, which makes developing better versions much easier. And, in addition, this technique of software engineering, does not spoil the attempts to create the fastest possible LBM code.

The second interesting detail lies in the head of the three innermost for loops. The start and the end index of the loops can be hidden behind a macro. This detail is version-specific and may change, if major changes of the data layout occur. But, for the basic understanding of the algorithm, the exact values are irrelevant and can be replaced by macros. In addition, most macros can be used in several versions of the LBM algorithm and can therefore be reused efficiently. Again, the two advantages of this software engineering technique come together: simplification and reuse, without paying the cost of bad performance.

In the following chapters this technique will be used in all code sections to help the reader to understand the code more easily and to improve the clarity of the code. Additionally, the macros will also be explained. In some cases it will become obvious, why this approach of software development is much better, because of the complexity of the macros. It will also become clear, that some versions only differ by the exchange of one macro, for example between the stream-collide and the collide-stream versions. Only by exchanging the LBMSTEP macro, one version can be turned into the other.

2.2 Using Macros Hierarchically

With this technique, a macro hierarchy can be created, which also helps to improve the readability of the code.

Algorithm 2.2 First Macro-Definition

```
#define CELLSIZE 20
#define BLOCKSIZE_X 10
```

Algorithm 2.3 Second Macro-Definition

```
#define K_END ((kk+BLOCKSIZE_X · CELLSIZE>=(dim+1) · CELLSIZE)?
    (dim+1) · CELLSIZE :
    (kk+BLOCKSIZE_X · CELLSIZE-t · CELLSIZE))
```

For example, the definition of the macro CELLSIZE (2.2) can be reused in the macros for the for loop headers (2.3) and in the LBMSTEP macros. Creating a hierarchy of macros reduces the amount of macros, which helps the developer to write the code, and also improves the intuitive understanding of the code.

In this thesis the intensive use of macros was implemented as far as possible. In the following code examples an insight into this technique of software engineering is given.

Chapter 3

Non-Cache Optimizations

Beside the cache optimization of the LBM, the code can also be improved in terms of data access and arithmetic. Improvements concerning the data access affect the implementation of the multidimensional grid. Section 3.1 will show, that the decision of *how* to implement and access the data of the grid, can also give great performance increases. Section 3.2 will demonstrate, how the existing Lattice Boltzmann step can be arithmetically improved. The arithmetic improvements of the code do not change the physical model itself, but rather try to speed up the existing calculations by reducing the number of floating-point operations. Section 3.3 will show another non-cache optimization, loop unrolling, which was applied to the LBM code. But, as you will see, this common optimization technique shows rather disappointing results in combination with the Lattice Boltzmann code.

3.1 Data Access

A very important decision concerning performance is the decision about the implementation and data access of the multidimensional data. Although there may be much more possibilities, this thesis only uses the three most important data access patterns:

1. The most basic, even naive implementation of an array using four indices (Section 3.1.1)
2. The manually access of the multidimensional data in a single index array (Section 3.1.2)
3. The Usage of pointers, which are a common programming tool of C (Section 3.1.3)

3.1.1 Data Access with Four Indices

Algorithm 3.1 Four Index Array

```
double src[dimZ][dimY][dimX][20]
double dst[dimZ][dimY][dimX][20]
...
for $t = 1$ to $timesteps$ by $1$ do
    for $i = 1$ to $dim$ by $1$ do
        for $j = 1$ to $dim$ by $1$ do
            for $k = 1$ to $dim$ by $1$ do
                ...
                dst[i][j][k][C] = src[i][j][k][C]
                ...
            end for
        end for
    end for
end for
```

The most intuitive implementation of the three dimensional grid, containing cells with multiple data, is a four dimensional array. Three dimensions refer to the three dimensions of space, the fourth refers to the 20 values of each individual cell. As shown in Algorithm 3.1 this leads to a very simple implementation: the three loop indices i, j and k are used as the first three indices of the array, a macro for the specific cell value is used as the fourth index.

Algorithm 3.2 Three Dimensional Cell Array

```
typedef double cell[20]
cell src[dim][dim][dim]
cell dst[dim][dim][dim]
...
for $t = 1$ to $timesteps$ by 1 do
    for $i = 1$ to $dim$ by 1 do
        for $j = 1$ to $dim$ by 1 do
            for $k = 1$ to $dim$ by 1 do
                ...
                dst[i][j][k][C] = src[i][j][k][C]
                ...
            end for
        end for
    end for
end for
```

Though this is a very intuitive way to represent a three dimensional grid of cells, the performance of this data access was rather disappointing: this basic implementation shows the poorest performance of all versions on all platforms (see Section 6.1). An interesting alternative to this basic implementation is shown in Algorithm 3.2. There, a cell data type is defined, which contains the 20 double values of a cell. With this data type, the two grids are created as three dimensional arrays, although the data access remains exactly the same as in Algorithm 3.1. This implementation runs nearly as fast as the single index access pattern, which is explained in the next section. Obviously, the introduction of the cell data type helps the compiler to create faster code, likely very similar to the single index array data access. The four indices data access will henceforth be called Version A.

3.1.2 Single-Index Access with Integer Arithmetic

Algorithm 3.3 Single Index Array

```
#define CELLSIZE 20
#define ROWSIZE CELLSIZE · dimX
#define PLAINSIZE ROWSIZE · dimY
#define src(i,j,k,l) src[(i) · PLANESIZE + (j) · ROWSIZE + (k) · CELLSIZE + (l)]
...
double src[dimZ · dimY · dimX · CELLSIZE]
double dst[dimZ · dimY · dimX · CELLSIZE]
...
for $t = 1$ to $timesteps$ by 1 do
    for $i = 1$ to $dim$ by 1 do
        for $j = 1$ to $dim$ by 1 do
            for $k = 1$ to $dim$ by 1 do
                ...
                dst(i,j,k,C) = src(i,j,k,C)
                ...
            end for
        end for
    end for
end for
```

The first attempt to improve the performance by using another data access was to allocate an array with only a single-index and manually access the cells. Again, this is done with the help of a macro. Within this macro, i, j and k are weighted with PLANESIZE, ROWSIZE and CELLSIZE, respectively, to allocate a specific cell, l is added to this offset to reach a certain value within the cell (Algorithm 3.3). Although this data access is a lot more complicated and less intuitive than the data access with four indices, the complexity is reduced with the usage of a macro to a very similar version of the four index array: the macro also needs four indices to access a specific distribution function of a specific cell, but the more complicated data access in the single-index array is hidden behind the macro. Again, as in the case of the update kernel of a Lattice Boltzmann step, the complexity of the algorithm can be hidden behind a macro, a change in the order of the cells only results in the change of the macro. This type of data access was also used in [Wil03].

Algorithm 3.4 Alternative Version of Single-Index Array

```
define src(i,j,k,l) src[i + j + k + l]
double src[dim · dim · dim · CELLSIZE]
double dst[dim · dim · dim · CELLSIZE]
...
for $t = 1$ to $timesteps$ by 1 do
    for $i = PLANESIZE$ to dim · PLANESIZE by $PLANESIZE$ do
        for $j = ROWSIZE$ to dim · ROWSIZE by $ROWSIZE$ do
            for $k = CELLSIZE$ to dim · CELLSIZE by $CELLSIZE$ do
                ...
                dst(i,j,k,C) = src(i,j,k,C)
                ...
            end for
        end for
    end for
end for
```

The performance of this implementation of data access is a significant improvement to the data access with four indices, especially on an Intel Pentium 4 machine. On this architecture, the single-index array versions were the fastest.

By reducing the number of multiplications, this version can be improved even further (Algorithm 3.4): the macro is changed to a simple addition of the four indices. Therefore, the indices are increased by the dimensional size respectively. This increase is done by an addition, so no multiplication has to be calculated, neither in the macro nor in the loop heads. That the reduction of multiplications promises another performance increase, can be explained with an example with the Intel Pentium4: an integer multiplication has a latency of 15 cycles and an iteration rate of four cycles, which means, that every four cycles one multiplication can be performed. An integer addition has a latency of only 0.5 cycles and an iteration rate of 0.5 cycles. Consequently, this implementation shows the best results of the single-index array data access and was therefore used for all performance measurements. Henceforth, this type of data access will be referred to as Version B.

3.1.3 Single-Index Access with Pointer Arithmetic

The idea for the third possibility to access the data is described in [PK03]: To avoid multiplications, the cells are referenced by pointers, which are increased by adding CELLSIZE within a row. Algorithm 3.5 shows the implementation for the case of a surrounding fluid layer around the obstacle layer, which will be discussed in Section 4.2. These additional cells force the special cases, where, for example, 2 · CELLSIZE is added, whenever the end of a row of grid cells has been reached.

This version shows very good performance results on the AMD Athlon machine. It will henceforth be called Version C. All results of the basic versions, which differ only in the data access without further improvement, can be found in Section 6.1.

3.2 Arithmetic Optimization

In addition to the cache optimization, one focus, in order to improve overall performance of the algorithm, was to improve the performance of the 'heart' of the algorithm, the arithmetical part. This section explains, which techniques were used to speed up the processing of one cell update.

Algorithm 3.5 Data Access with pointers

```
double src[dim · dim · dim · CELLSIZE]
double dst[dim · dim · dim · CELLSIZE]
double *psrc, *pdst
...
for $t = 1$ to $timesteps$ by 1 do
    psrc = src + PLANESIZE - ROWSIZE - 2 · CELLSIZE
    pdst = src + PLANESIZE - ROWSIZE - 2 · CELLSIZE
    for $i = 1$ to $dim$ by 1 do
        psrc += 2 · ROWSIZE
        pdst += 2 · ROWSIZE
        for $j = 1$ to $dim$ by 1 do
            psrc += 2 · CELLSIZE
            pdst += 2 · CELLSIZE
            for $k = 1$ to $dim$ by 1 do
                psrc += CELLSIZE
                pdst += CELLSIZE
                ...
                *(pdst+C) = *(psrc+C)
                ...
            end for
        end for
    end for
end for
```

3.2.1 Unoptimized Lattice Boltzmann Step

To test the arithmetic performance of the core of the algorithm a special test program was written. In this program, only two cells were allocated, which successively write to and read from each other (Figure 3.1). In this manner most memory traffic was eliminated and only the processing speed of the core could be measured, because all data fitted in the innermost memory banks of all CPUs, which are the registers and the L1 data cache.

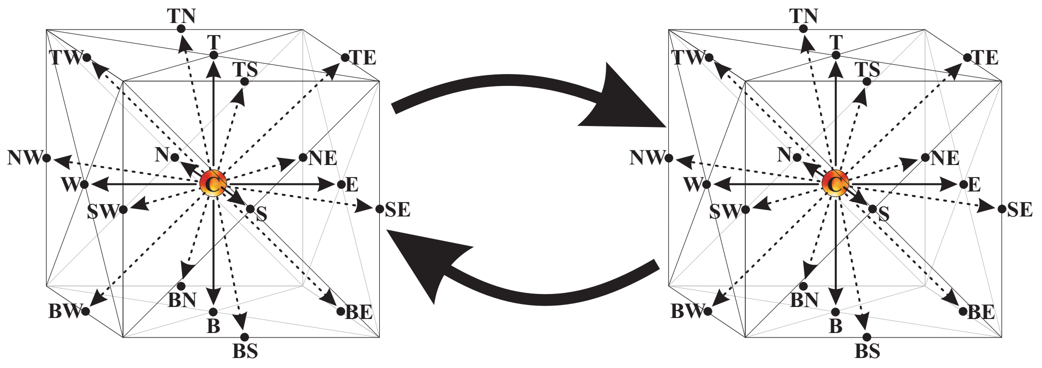


Figure 3.1: The test program for the computational kernel of the algorithm: The data of the left cell is written to the right one and vice versa

Algorithm 3.6 shows the unoptimized Lattice Boltzmann step. This algorithm results from Chapter 1. The goal of this chapter is to improve the arithmetic performance of the Lattice Boltzmann algorithm. Since this arithmetic part of the code will be implemented in a macro, the improvement made in this particular part of the code affects all versions of the Lattice Boltzmann algorithm.

Some improvements of the basic Lattice Boltzmann step, as it was introduced in Chapter 1, have already been included in this code. The most important and very natural improvement is the fusion of the stream and the collide step. The advantage of this procedure is, that the grid has only to be traversed once for each Lattice Boltzmann step instead of twice (once for the stream step and once for the collide step). This improvement shows a very good performance increase as shown in [Wil03]. Additionally, all functions to calculate u_x, u_y, u_z and rho have been inlined in order to

Algorithm 3.6 Unoptimized Lattice Boltzmann Step

```
1: if FLAG == OBSTACLE then
2:     dst(i,j,k,N) = src(i,j+1,k,S)
3:     dst(i,j,k,S) = src(i,j-1,k,N)
4:     dst(i,j,k,W) = src(i,j,k-1,E)
5:     dst(i,j,k,E) = src(i,j,k+1,W)
6:     dst(i,j,k,T) = src(i+1,j,k,B)
7:     dst(i,j,k,B) = src(i-1,j,k,T)
8:     dst(i,j,k,NW) = src(i,j+1,k-1,SE)
9:     dst(i,j,k,NE) = src(i,j+1,k+1,SW)
10:    dst(i,j,k,SW) = src(i,j-1,k-1,NE)
11:    dst(i,j,k,SE) = src(i,j-1,k+1,NW)
12:    dst(i,j,k,TN) = src(i+1,j+1,k,BS)
13:    dst(i,j,k,TS) = src(i+1,j-1,k,BN)
14:    dst(i,j,k,TW) = src(i+1,j,k-1,BE)
15:    dst(i,j,k,TE) = src(i+1,j,k+1,BW)
16:    dst(i,j,k,BN) = src(i-1,j+1,k,TS)
17:    dst(i,j,k,BS) = src(i-1,j-1,k,TN)
18:    dst(i,j,k,BW) = src(i-1,j,k-1,TE)
19:    dst(i,j,k,BE) = src(i-1,j,k+1,TW)
20:    continue # to next iteration
21: end if
22: if FLAG == ACCELERATION then
23:    rho = 1.0
24:    u_x = 0.05
25:    u_y = 0.0
26:    u_z = 0.0
27: else
28:    rho = src(i,j,k,C) + src(i,j-1,k,N) + src(i,j+1,k,S) + src(i,j,k+1,W) + src(i,j,k-1,E) + src(i-1,j,k,T) + src(i+1,j,k,B) + src(i,j-1,k+1,NW) + src(i,j-1,k-1,NE) + src(i,j+1,k+1,SW) + src(i,j+1,k-1,SE) + src(i-1,j-1,k,TN) + src(i-1,j+1,k,TS) + src(i-1,j,k+1,TW) + src(i-1,j,k-1,TE) + src(i+1,j-1,k,BN) + src(i+1,j+1,k,BS) + src(i+1,j,k+1,BW) + src(i+1,j,k-1,BE)
29:    u_x = ( - src(i,j,k+1,W) + src(i,j,k-1,E) - src(i,j-1,k+1,NW) + src(i,j-1,k-1,NE) - src(i,j+1,k+1,SW) + src(i,j+1,k-1,SE) - src(i-1,j,k+1,TW) + src(i-1,j,k-1,TE) - src(i+1,j,k+1,BW) + src(i+1,j,k-1,BE) ) / rho
30:    u_y = ( src(i,j-1,k,N) - src(i,j+1,k,S) + src(i,j-1,k+1,NW) + src(i,j-1,k-1,NE) - src(i,j+1,k+1,SW) - src(i,j+1,k-1,SE) + src(i-1,j-1,k,TN) - src(i-1,j+1,k,TS) + src(i+1,j-1,k,BN) - src(i+1,j+1,k,BS) ) / rho
31:    u_z = ( src(i-1,j,k,T) - src(i+1,j,k,B) + src(i-1,j-1,k,TN) + src(i-1,j+1,k,TS) + src(i-1,j,k+1,TW) + src(i-1,j,k-1,TE) - src(i+1,j-1,k,BN) - src(i+1,j+1,k,BS) - src(i+1,j,k+1,BW) - src(i+1,j,k-1,BE) ) / rho
32: end if
33: u_sqr_trm = 1.5 · (u_x · u_x + u_y · u_y + u_z · u_z)
34: dst(i,j,k,C) = (1.0-OMEGA) · src(i,j,k,C) + OMEGA · (W_0 · rho · (1.0-u_sqr_trm))
35: dst(i,j,k,N) = (1.0-OMEGA) · src(i,j-1,k,N) + OMEGA · (W_1 · rho · (1.0+3.0 · u_y+4.5 · (u_y · u_y)-u_sqr_trm))
36: dst(i,j,k,S) = (1.0-OMEGA) · src(i,j+1,k,S) + OMEGA · (W_1 · rho · (1.0-3.0 · u_y+4.5 · (u_y · u_y)-u_sqr_trm))
37: dst(i,j,k,W) = (1.0-OMEGA) · src(i,j,k+1,W) + OMEGA · (W_1 · rho · (1.0-3.0 · u_x+4.5 · (u_x · u_x)-u_sqr_trm))
38: dst(i,j,k,E) = (1.0-OMEGA) · src(i,j,k-1,E) + OMEGA · (W_1 · rho · (1.0+3.0 · u_x+4.5 · (u_x · u_x)-u_sqr_trm))
39: dst(i,j,k,T) = (1.0-OMEGA) · src(i-1,j,k,T) + OMEGA · (W_1 · rho · (1.0+3.0 · u_z+4.5 · (u_z · u_z)-u_sqr_trm))
40: dst(i,j,k,B) = (1.0-OMEGA) · src(i+1,j,k,B) + OMEGA · (W_1 · rho · (1.0-3.0 · u_z+4.5 · (u_z · u_z)-u_sqr_trm))
41: dst(i,j,k,NW) = (1.0-OMEGA) · src(i,j-1,k+1,NW) + OMEGA · (W_2 · rho · (1.0+3.0 · (u_y-u_x)+4.5 · ((u_y-u_x) · (u_y-u_x))-u_sqr_trm))
42: dst(i,j,k,NE) = (1.0-OMEGA) · src(i,j-1,k-1,NE) + OMEGA · (W_2 · rho · (1.0+3.0 · (u_y+u_x)+4.5 · ((u_y+u_x) · (u_y+u_x))-u_sqr_trm))
43: dst(i,j,k,SW) = (1.0-OMEGA) · src(i,j+1,k+1,SW) + OMEGA · (W_2 · rho · (1.0+3.0 · (-u_y-u_x)+4.5 · ((-u_y-u_x) · (-u_y-u_x))-u_sqr_trm))
44: dst(i,j,k,SE) = (1.0-OMEGA) · src(i,j+1,k-1,SE) + OMEGA · (W_2 · rho · (1.0+3.0 · (-u_y+u_x)+4.5 · ((-u_y+u_x) · (-u_y+u_x))-u_sqr_trm))
45: dst(i,j,k,TN) = (1.0-OMEGA) · src(i-1,j-1,k,TN) + OMEGA · (W_2 · rho · (1.0+3.0 · (u_y+u_z)+4.5 · ((u_y+u_z) · (u_y+u_z))-u_sqr_trm))
46: dst(i,j,k,TS) = (1.0-OMEGA) · src(i-1,j+1,k,TS) + OMEGA · (W_2 · rho · (1.0+3.0 · (-u_y+u_z)+4.5 · ((-u_y+u_z) · (-u_y+u_z))-u_sqr_trm))
47: dst(i,j,k,TW) = (1.0-OMEGA) · src(i-1,j,k+1,TW) + OMEGA · (W_2 · rho · (1.0+3.0 · (-u_x+u_z)+4.5 · ((-u_x+u_z) · (-u_x+u_z))-u_sqr_trm))
48: dst(i,j,k,TE) = (1.0-OMEGA) · src(i-1,j,k-1,TE) + OMEGA · (W_2 · rho · (1.0+3.0 · (u_x+u_z)+4.5 · ((u_x+u_z) · (u_x+u_z))-u_sqr_trm))
49: dst(i,j,k,BN) = (1.0-OMEGA) · src(i+1,j-1,k,BN) + OMEGA · (W_2 · rho · (1.0+3.0 · (u_y-u_z)+4.5 · ((u_y-u_z) · (u_y-u_z))-u_sqr_trm))
50: dst(i,j,k,BS) = (1.0-OMEGA) · src(i+1,j+1,k,BS) + OMEGA · (W_2 · rho · (1.0+3.0 · (-u_y-u_z)+4.5 · ((-u_y-u_z) · (-u_y-u_z))-u_sqr_trm))
51: dst(i,j,k,BW) = (1.0-OMEGA) · src(i+1,j,k+1,BW) + OMEGA · (W_2 · rho · (1.0+3.0 · (-u_x-u_z)+4.5 · ((-u_x-u_z) · (-u_x-u_z))-u_sqr_trm))
52: dst(i,j,k,BE) = (1.0-OMEGA) · src(i+1,j,k-1,BE) + OMEGA · (W_2 · rho · (1.0+3.0 · (u_x-u_z)+4.5 · ((u_x-u_z) · (u_x-u_z))-u_sqr_trm))
```

avoid function calling overhead.

This code is very similar to the code in [Wil03] about cache optimization of Lattice Boltzmann in 2D, though it has been modified with respect to 3D.

The first part of the Lattice Boltzmann step, the if statement, which checks for an obstacle cell (lines 1-27), is obviously not part of the arithmetic optimization, because no arithmetic operations are performed in this part.

Also the second part, which deals with an acceleration cell (lines 22-26), contains no arithmetic operation. All arithmetic operations are contained in the third part (lines 27-52), which calculates the values for rho, u_x, u_y and u_z, respectively, and uses these values to process the new values for all 19 distribution functions.

3.2.2 Optimized Lattice Boltzmann Step

The major attempt to improve performance was to decrease the number of floating-point operations. With closer examination it becomes obvious, that the computations of rho and the u values contain several identical operations. By calculating a part of the u values ahead of rho and by using the result in rho, the performance of the core increased significantly. This approach is similar to [Wil03], but with slight differences.

Another improvement was achieved by the prior calculation of 1/rho. This saves two divisions, which are the most costly arithmetical operations in this algorithm. By reducing the division to only one, an additional performance increase can be observed. Additionally, the values w_0, w_1 and w_2 can be calculated precociously, which brings some performance advantage. As it seems, this handmade precalculations helped the compiler to produce more efficient code. But in most cases, one should expect this work to be done by the compiler.

A small performance increase can be gained by using parenthesis which show the compiler, which values can be added simultaneously. Although this increases the usage of the super-scalar architecture of modern CPUs, this technique can result in different results due to round-off. But in this work, no influence could be observed.

Several other attempts to improve the performance failed. For example, the attempt to "prefetch" all relevant values from the surrounding cells and store them in variables (Algorithm 3.8) showed a clear decrease in performance. Since this technique showed good results in [Wil03], it seems as in the 3D case, the number of variables becomes too big, so not every variable can be saved in latency optimized places like the CPU registers.

Another failed attempt to increase performance was to precalculate several other values, which obviously appear several times (Algorithm 3.9). But all combinations of this precalculations also showed a decrease of performance. Very likely, in this case, the compiler is able to recognize the repeating calculation of these values as obsolete and creates a better precalculation.

Finally the code shown in Algorithm 3.7 showed the best performance. In detail, with all improvements added to the code, the performance for the special test scenario was especially faster on the AMD Athlon architectures as is illustrated in Figure 3.2. The results show a clear performance increase for both architectures. However, the performance results of this test scenario cannot be compared to the real LBM measurements. In this arithmetic test scenario only fluid cells are updated, while in the LBM problem fluid, obstacle and acceleration cells must be updated. Since the update of an obstacle cell counts towards the cell updates, but no arithmetic operations have to be performed, the measured performance will likely be faster than in the arithmetic test scenario. The same counts for acceleration cells, where no rho and no velocity values have to be calculated. This is especially true for smaller grid sizes, where the percentage of obstacle and acceleration cells is highest. Though this may seem like a significant improvement, it is unlikely that this performance

Algorithm 3.7 Optimized Lattice Boltzmann Step

```
if ... then
    ...
else
    u_x = (((src(i,j,k-1,E) + src(i,j-1,k-1,NE)) + (src(i,j+1,k-1,SE) + src(i-1,j,k-1,TE))) + src(i+1,j,k-1,BE))
    u_y = ((src(i,j-1,k,N) + src(i,j-1,k+1,NW)) + (src(i-1,j-1,k,TN) + src(i+1,j-1,k,BN)))
    u_z = ((src(i-1,j,k,T) + src(i-1,j+1,k,TS)) + src(i-1,j,k+1,TW))
    rho = src(i,j,k,C) + src(i,j+1,k,S) + src(i,j,k+1,W) + src(i+1,j,k,B) + src(i,j+1,k+1,SW) + src(i+1,j+1,k,BS) + src(i+1,j,k+1,BW) + u_x + u_y + u_z
    temp = 1 / rho
    u_x = ( u_x - src(i,j,k+1,W) - src(i,j-1,k+1,NW) - src(i,j+1,k+1,SW) - src(i-1,j,k+1,TW) - src(i+1,j,k+1,BW) ) /cdot temp
    u_y = ( u_y - src(i,j+1,k,S) + src(i,j-1,k-1,NE) - src(i,j+1,k+1,SW) - src(i,j+1,k-1,SE) - src(i-1,j+1,k,TS) - src(i+1,j+1,k,BS) ) /cdot temp
    u_z = ( u_z - src(i+1,j,k,B) + src(i-1,j-1,k,TN) + src(i-1,j,k-1,TE) - src(i+1,j-1,k,BN) - src(i+1,j+1,k,BS) - src(i+1,j,k+1,BW) - src(i+1,j,k-1,BE) ) /cdot temp
end if
u_sqr_trm = 1.5 · (u_x · u_x + u_y · u_y + u_z · u_z)
w_0 = OMEGA · W_0 · rho
w_1 = OMEGA · W_1 · rho
w_2 = OMEGA · W_2 · rho
dst(i,j,k,C) = ((1.0-OMEGA) · src(i,j,k,C) + (w_0 · (1.0-u_sqr_trm)))
dst(i,j,k,N) = ((1.0-OMEGA) · src(i,j-1,k,N) + (w_1 · ((1.0-u_sqr_trm)+3.0 · u_y+4.5 · (u_y · u_y))))
dst(i,j,k,S) = ((1.0-OMEGA) · src(i,j+1,k,S) + (w_1 · ((1.0-u_sqr_trm)-3.0 · u_y+4.5 · (u_y · u_y))))
dst(i,j,k,W) = ((1.0-OMEGA) · src(i,j,k+1,W) + (w_1 · ((1.0-u_sqr_trm)-3.0 · u_x+4.5 · (u_x · u_x))))
dst(i,j,k,E) = ((1.0-OMEGA) · src(i,j,k-1,E) + (w_1 · ((1.0-u_sqr_trm)+3.0 · u_x+4.5 · (u_x · u_x))))
dst(i,j,k,T) = ((1.0-OMEGA) · src(i-1,j,k,T) + (w_1 · ((1.0-u_sqr_trm)+3.0 · u_z+4.5 · (u_z · u_z))))
dst(i,j,k,B) = ((1.0-OMEGA) · src(i+1,j,k,B) + (w_1 · ((1.0-u_sqr_trm)-3.0 · u_z+4.5 · (u_z · u_z))))
dst(i,j,k,NW) = ((1.0-OMEGA) · src(i,j-1,k+1,NW) + (w_2 · ((1.0-u_sqr_trm)+3.0 · (u_y-u_x)+4.5 · ((u_y-u_x) · (u_y-u_x)))))
dst(i,j,k,NE) = ((1.0-OMEGA) · src(i,j-1,k-1,NE) + (w_2 · ((1.0-u_sqr_trm)+3.0 · (u_y+u_x)+4.5 · ((u_y+u_x) · (u_y+u_x)))))
dst(i,j,k,SW) = ((1.0-OMEGA) · src(i,j+1,k+1,SW) + (w_2 · ((1.0-u_sqr_trm)+3.0 · (-u_y-u_x)+4.5 · ((-u_y-u_x) · (-u_y-u_x)))))
dst(i,j,k,SE) = ((1.0-OMEGA) · src(i,j+1,k-1,SE) + (w_2 · ((1.0-u_sqr_trm)+3.0 · (-u_y+u_x)+4.5 · ((-u_y+u_x) · (-u_y+u_x)))))
dst(i,j,k,TN) = ((1.0-OMEGA) · src(i-1,j-1,k,TN) + (w_2 · ((1.0-u_sqr_trm)+3.0 · (u_y+u_z)+4.5 · ((u_y+u_z) · (u_y+u_z)))))
dst(i,j,k,TS) = ((1.0-OMEGA) · src(i-1,j+1,k,TS) + (w_2 · ((1.0-u_sqr_trm)+3.0 · (-u_y+u_z)+4.5 · ((-u_y+u_z) · (-u_y+u_z)))))
dst(i,j,k,TW) = ((1.0-OMEGA) · src(i-1,j,k+1,TW) + (w_2 · ((1.0-u_sqr_trm)+3.0 · (-u_x+u_z)+4.5 · ((-u_x+u_z) · (-u_x+u_z)))))
dst(i,j,k,TE) = ((1.0-OMEGA) · src(i-1,j,k-1,TE) + (w_2 · ((1.0-u_sqr_trm)+3.0 · (u_x+u_z)+4.5 · ((u_x+u_z) · (u_x+u_z)))))
dst(i,j,k,BN) = ((1.0-OMEGA) · src(i+1,j-1,k,BN) + (w_2 · ((1.0-u_sqr_trm)+3.0 · (u_y-u_z)+4.5 · ((u_y-u_z) · (u_y-u_z)))))
dst(i,j,k,BS) = ((1.0-OMEGA) · src(i+1,j+1,k,BS) + (w_2 · ((1.0-u_sqr_trm)+3.0 · (-u_y-u_z)+4.5 · ((-u_y-u_z) · (-u_y-u_z)))))
dst(i,j,k,BW) = ((1.0-OMEGA) · src(i+1,j,k+1,BW) + (w_2 · ((1.0-u_sqr_trm)+3.0 · (-u_x-u_z)+4.5 · ((-u_x-u_z) · (-u_x-u_z)))))
dst(i,j,k,BE) = ((1.0-OMEGA) · src(i+1,j,k-1,BE) + (w_2 · ((1.0-u_sqr_trm)+3.0 · (u_x-u_z)+4.5 · ((u_x-u_z) · (u_x-u_z)))))
```

Algorithm 3.8 Prefetching the distribution functions

```
double n, s, w, e, t, b, ...
n = src(i,j-1,k,N)
s = src(i,j+1,k,S)
w = src(i,j,k+1,W)
e = src(i,j,k-1,E)
t = src(i-1,j,k,T)
b = src(i+1,j,k,B)
...
```

Algorithm 3.9 Precalculation of several values

```
temp = 1.0 - u_sqr_trm
temp2 = 1.0 - OMEGA
...
```

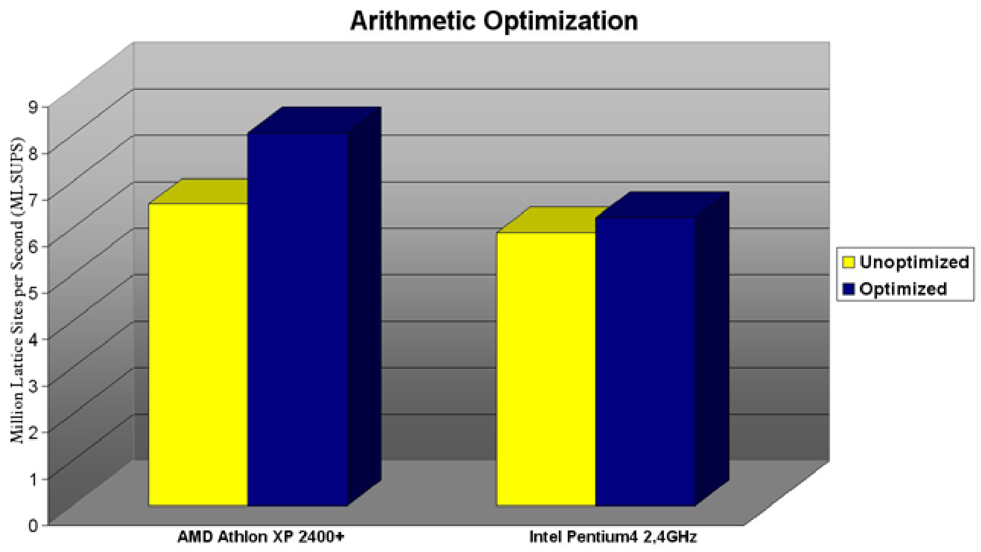


Figure 3.2: The performance increase of the arithmetical optimization on an AMD Athlon XP 2400+ (left) and Intel Pentium4 2.4GHz (right) architecture for the special test scenario.

increase can directly be passed on to the complete Lattice Boltzmann algorithm due to the memory latencies, which were eliminated in this test. But for special problem sizes, which fit into the cache and for which the memory latencies are minimized accordingly, an improvement of the performance can be expected. Consequently the optimized code was used in all measurements of all versions in the later chapters. Although it is unlikely to improve bigger problem sizes, it nevertheless makes sense to use the fastest core available.

3.3 Loop Unrolling

Loop unrolling is a very common technique to reduce the loop overhead by simply increasing the loop stride and combining several loop iterations in one single loop iteration. This technique is one of the most interesting for a pipeline-aware programming style and it can be implemented in several levels of unrolling and even within a hierarchy of loops: instead of only increasing the stride of the innermost loop, the stride of several loops can be increased. Algorithm 3.10 shows two nested loops, each with stride 2.

Algorithm 3.10 Loop Unrolling

```
1: for $i = 1$ to $dim$ by 2 do
2:   for $j = 1$ to $dim$ by 2 do
3:     do-something(i,j)
4:     do-something(i,j+1)
5:     do-something(i+1,j)
6:     do-something(i+1,j+1)
7:   end for
8: end for
```

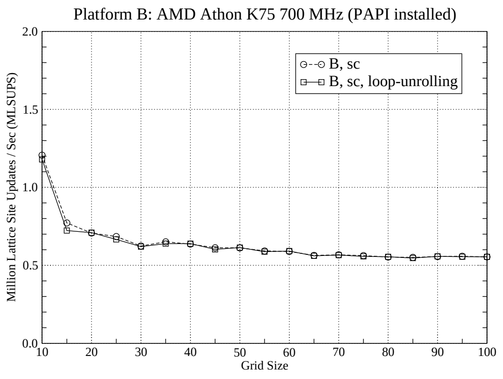
In Algorithm 3.11 the stride of the innermost loop was increased to 2. Therefore, the number of Lattice Boltzmann steps within this loop was also increased to 2. Though not very complicated to implement, this technique often shows at least a slight increase in performance, but unfortunately not in the case of the 3D Lattice Boltzmann. Figure 3.3 shows the performance of the LBM code, measured in "mega lattice site updates per second"(MLSUPS), the level 1 instruction cache misses and level 1 data cache misses for data access Version B with the order of stream-collide, with and without loop-unrolling. With the profiling tool PAPI on Platform B, the ineffectiveness of loop

Algorithm 3.11 Loop Unrolling for 3D LBM

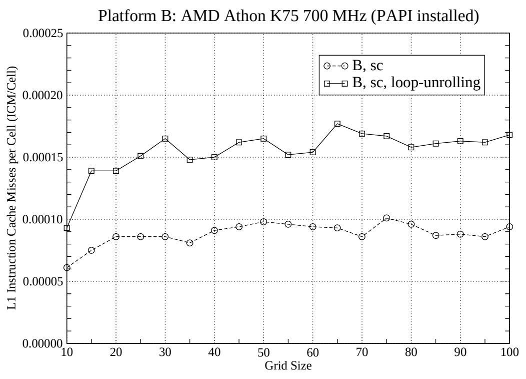
```
1: // Basic LBM without loop unrolling:
2: for $t = 1$ to timesteps by 1 do
3:     for $i = 1$ to dim by 1 do
4:         for $j = 1$ to dim by 1 do
5:             for $k = 1$ to dim by 1 do
6:                 LBMSTEP(i,j,k)
7:             end for
8:         end for
9:     end for
10: end for
```

```
1: // Basic LBM with loop unrolling by 2:
2: for $t = 1$ to timesteps by 1 do
3:     for $i = 1$ to dim by 1 do
4:         for $j = 1$ to dim by 1 do
5:             for $k = 1$ to dim by 2 do
6:                 LBMSTEP(i,j,k)
7:                 LBMSTEP(i,j,k)
8:             end for
9:         end for
10:     end for
11: end for
```

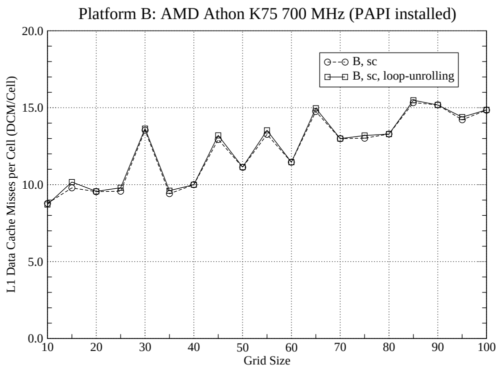
unrolling can be traced back to the increase in instruction cache misses due to the very large loop body. In terms of performance, there are nearly no differences between the two implementations, also the differences for the L1 data cache misses are negligible. Several performance tests lead to the assumption, that also in the LBM algorithm without loop unrolling the time spent in the loop body is large compared to the loop overhead. By even enlarging the loop body further, the advantage of still reducing the loop overhead is negligible, but the number of instructions overwhelm the instruction cache. With this conclusion loop unrolling was not applied in any version of LBM. For more information about the profiling tool PAPI on x86 architectures, see [ABD+97].



(a) Performance in MLSUPS



(b) L1 Instruction Cache Misses



(c) L1 Data Cache Misses

Figure 3.3: Benchmarking results for loop unrolling, measured with PAPI on Platform B.

Chapter 4

Cache Optimization Techniques

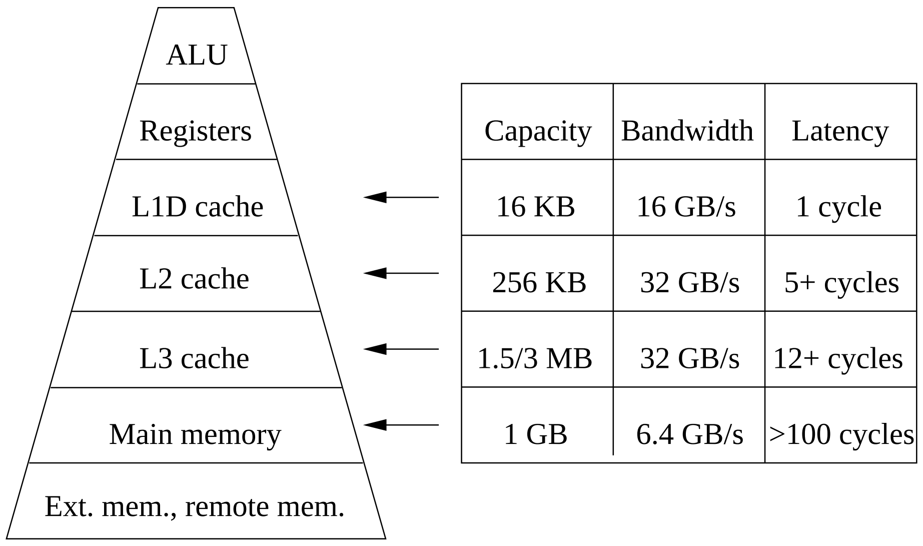
The last 20 years have brought great improvements of the theoretical processing speed of a CPU: super-scalar architectures, pipelines and the impressive improvements in the production process have enabled the CPUs to keep up with Moore's Law. Unfortunately, the memory architectures experienced a much slower increase in performance. The so called speed gap between the theoretical maximum speed of the CPU core and the memory has been growing bigger each year. To prevent idle CPU cores, faster, but more expensive memory was introduced between the CPU and the main memory, the cache. Organized in several hierarchical layers, the cache provides fast data flow towards the CPU, while also communicating with the main memory to prefetch or preload requested data. Compared to the access of the caches, the access to the main memory is about 100 times slower. Figure 4.1 shows the memory hierarchy of the newest Intel Itanium2 processor: the three layers of caches outperform the main memory by far in terms of latency and bandwidth. To speed up numerical algorithms with great dependence on the memory, it is imperative to exploit the memory hierarchy of the CPU and to try to improve the **spatial and temporal locality** of the code. The most ideal case would be to keep the data as close as possible to the arithmetic logical unit (ALU) of the CPU. This would guarantee the by far lowest memory latencies.

In this thesis, two approaches to reach this goal, were taken: first of all, by choosing an effective data layout, the memory traffic can be significantly reduced and by this the sum of memory latencies. Section 4.2 will show, that it is even possible, to nearly halve the memory requirement of the LBM. Second, by implementing a fast data access and changing the pattern of data access, the temporal locality of the code can be effectively increased. That the choice of how to access the data can greatly improve the performance has already been shown in Section 3.1. This chapter will demonstrate, that changing the order of cell updates can efficiently increase the performance of the code. To shortly explain the nomenclature and the properties of caches, the following section will briefly explain the purpose of the cache optimizing techniques. A more detailed explanation of caches can be found in [Sto02], [AK02] and [GH01]. Additional insight in CPU architecture and caches can be found in [Han98] and [HP96]. For an overview of cache optimization techniques, see [KW03].

4.1 Caches

In principle, caches are able to provide requested data by exploiting reference locality. 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 neighbouring 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 (see also [GH01], [KW03], [Wil03].

Admittedly, temporal and spatial locality exclude each other: temporal locality would best be served by loading each single data separately to minimize the probability of overwriting. This would guarantee a long lifetime for every data in cache and therefore increase temporal locality. Spatial locality, however, would greatly profit from contiguous data blocks of the size of the cache, because



Capacity	Bandwidth	Latency
16 KB	16 GB/s	1 cycle
256 KB	32 GB/s	5+ cycles
1.5/3 MB	32 GB/s	12+ cycles
1 GB	6.4 GB/s	>100 cycles

(a)

Figure 4.1: The memory hierarchy of the Intel Itanium2 processor.

then all subsequent data accesses could result in cache hits. As a compromise between these two requirements, cache lines are loaded into the cache, whose sizes differ from architecture to architecture. These cache lines contain several bytes of data, which delivers some spatial locality for the next data accesses and some temporal locality, because not every cache miss would overwrite all data in the cache.

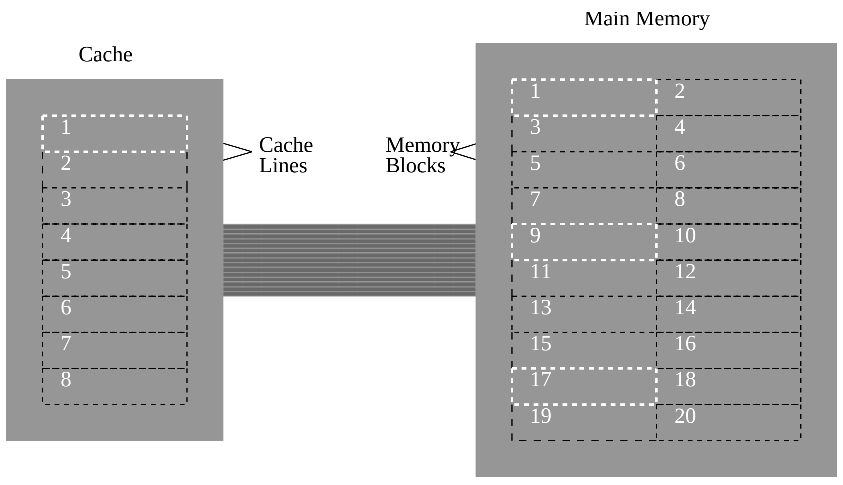
But even the manner of how cache lines are mapped to the cache differs between architectures: Figure 4.2(a) shows a direct mapped cache architecture. In this case, each cache line can only be mapped to exactly one location in cache. Clearly, this could result in very poor performance for specific problems in the case of data access in multiples of the cache size, because these cache lines would overwrite each other with every data access. The second part, Figure 4.2(b), shows a fully associative cache, where every cache line can be mapped to every location. Although this mapping eliminates the problem of a direct mapped cache, it lacks performance because of a different drawback: for each data request the cache management unit has to check, whether the data is already in the cache or not. For a direct mapped cache, this check can be done by a simple modulo operation, but with fully associative caches, each cache line in cache has to be checked.

To combine the short search time of direct mapped caches and the flexibility of fully associative caches, many architectures use n-way set-associative caches. The cache is divided into a number of sets, which consist of n cache lines each. When searching for a specific cache line, first, the set can be determined by a modulo operation and then the cache line has to be searched among a very small number of cache lines. Figure 4.2(c) illustrates a 4-way set-associative cache. In this example, the search for a cache line would first mean to find the according set and then searching among four cache lines at max.

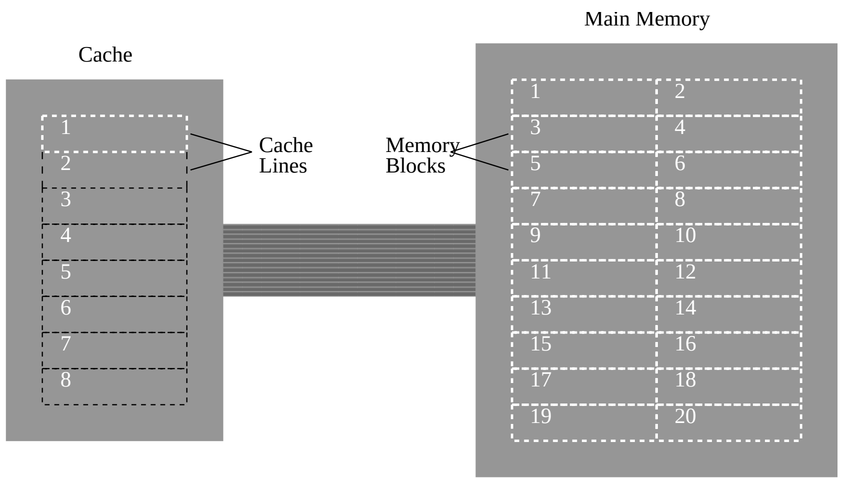
Appendix A shows the cache characteristics of the architectures used for this thesis.

4.2 Data Layout Optimizations

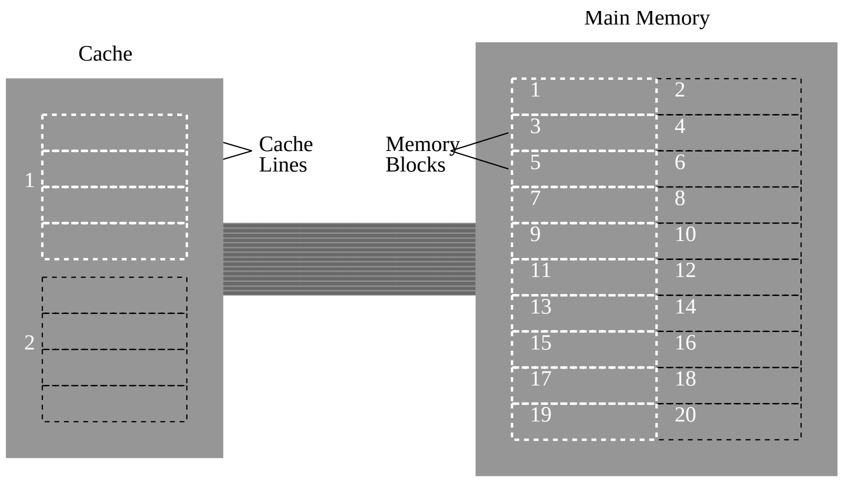
A very important decision concerning performance is the basic data layout of the problem. For this thesis, two different data layouts were implemented:



(a) Direct mapped cache



(b) Fully associative cache



(c) Set-associative cache

Figure 4.2: Cache associativity of a current CPU.

1. two separate grids, which means the memory requirement of two full grids
2. compressed grid, which offers a way to nearly halve this memory requirement

Unfortunately, it is probably impossible, to reduce the memory requirement to only one single grid because of the data dependencies: the old values have to be preserved, until a full Lattice Boltzmann step is completed.

4.2.1 Two Grids

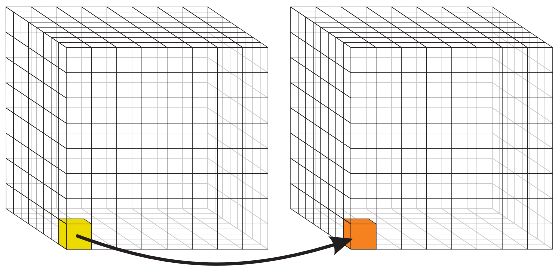
The first intuitive idea is to allocate two separate grids, both large enough to contain each cell of the original grid. The first of the two grids is initially called source grid, the second destination grid. As can be expected from the names, the source grid contains the initial data of the grid, the destination grid receives the first updated cells. After a complete update of the source grid, the destination grid becomes the source grid and the source grid becomes the destination grid. Again all cells of the source grid are updated to the destination grid, an so on.

By allocating two grids and writing from one to the other it is assured, that the data in the particular source grid is not replaced by updated values. Every cell in the grid relies on the data of the preceding time step, which would be overwritten, if every cell update would take place in a single grid. By overwriting the older values, the algorithm would not produce a correct result. For this reason it is important to allocate additional memory in order to maintain the data of the previous Lattice Boltzmann step. In this case, a whole additional grid is allocated.

The Lattice Boltzmann step as shown in Algorithm 3.7 is executed on all cells, even the obstacle cells, which are positioned along the boundaries of the grids. To avoid some special handling for these boundary cells, a layer of fluid was implemented, which encloses the obstacle cells. In this manner, all obstacle cells can stream their distribution functions to all neighboring cells without caring whether they are located at the boundary of the active grid. Figure 4.4 shows the grid with the surrounding fluid layer. For the code examples, in order to provide an easier understanding, these additive cells were neglected, but this additional fluid layer increases the dimension of the grid by 2.

For example, the biggest grid size was actually 102^3 instead of 100^3. For debugging purposes, the distribution functions of these surrounding cells can be initialized with very high and unusual values, so in case of a ”leak”, the values inside the obstacle cells will very fast attain unrealistic values.

Together with these special cells, the total memory requirement of the two grids can be calculated. Figure 4.5 shows the polynomial function of the memory requirement.


(a) The updated cell is written to the destination grid

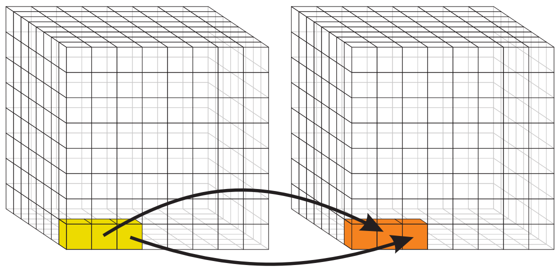

(b) Subsequently, all updated cells are written to the corresponding cell in the destination grid

Figure 4.3: Updating cells in two grids.

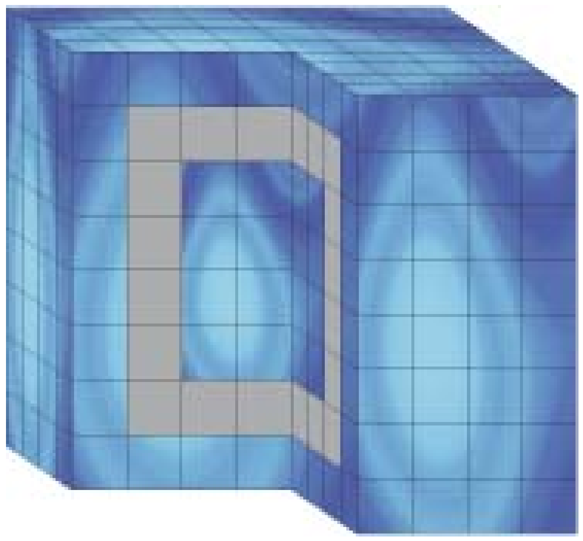


Figure 4.4: The implementation of one grid requires even slightly more memory than theoretically needed, because of a surrounding fluid layer, which helps to avoid a special treatment for the obstacle cells.

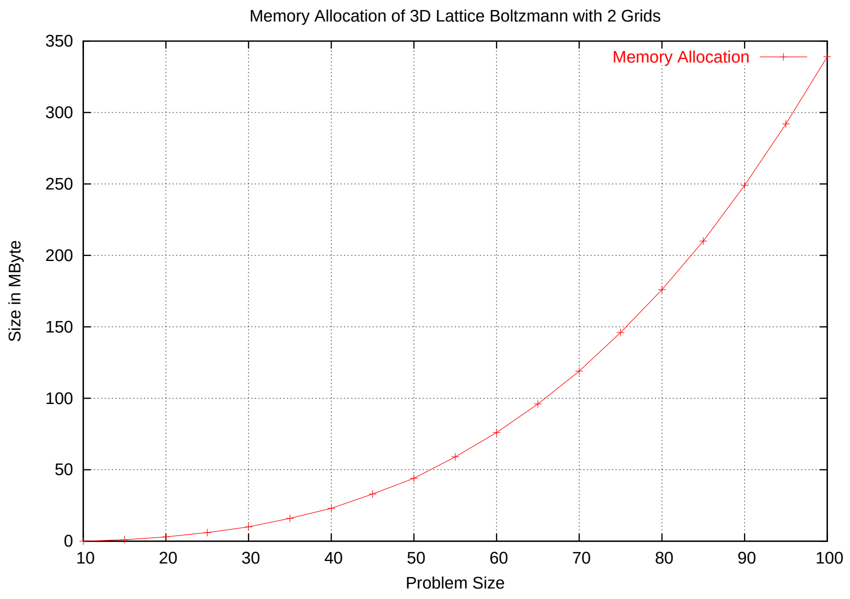


Figure 4.5: The memory requirement of two grids is an polynomial function, which can be derived by the following equation: $2 \cdot (GridSize + 2)^3 \cdot 20 \cdot 8$ Byte.

According to the development of this function it was decided to use grid sizes from 10 to 100 in steps of 5. Sizes below 10 can be considered impractical (the absolute minimum for a grid size to make sense would be 4), sizes above 100 are close to exceed the main memory capacities of the used architectures.

4.2.2 Grid Merging

Grid merging is a cache optimization technique, which aims at combining data, which belong together. For general information about this technique, which is also called array merging, see [HP96]. Since for each cell update in one grid the corresponding cell and its neighbours in the other grid are always needed, the pairwise combination of all cells in one grid with the corresponding cells of the other grid promises an advantage in terms of memory traffic, because fewer cache lines would have to be loaded. In [Wil03] it was shown, that this technique can indeed improve the performance of the 2D algorithm. Consequently, this technique was also applied to the LBM in 3D. But in contrast to the 2D case, 3D grid merging shows no performance improvements. As an example, data access Version B was implemented as grid merged version. This can be done by simply allocating one larger grid and changing the macro for the data access. Again, this shows, how modular the LBM in 3D can be implemented. Both the basic, non-merged version and the grid merged version were benchmarked on Platform B and profiled with PAPI. The benchmarking results can be seen in Figure 4.7. In terms of performance, no differences arise, which would be worth further study. Also the level 1 data cache misses show no improvement with respect to cache behaviour. Finally, the level 2 data cache misses show nearly no differences at all. Considering these results, grid merging was not applied for any version in this thesis.

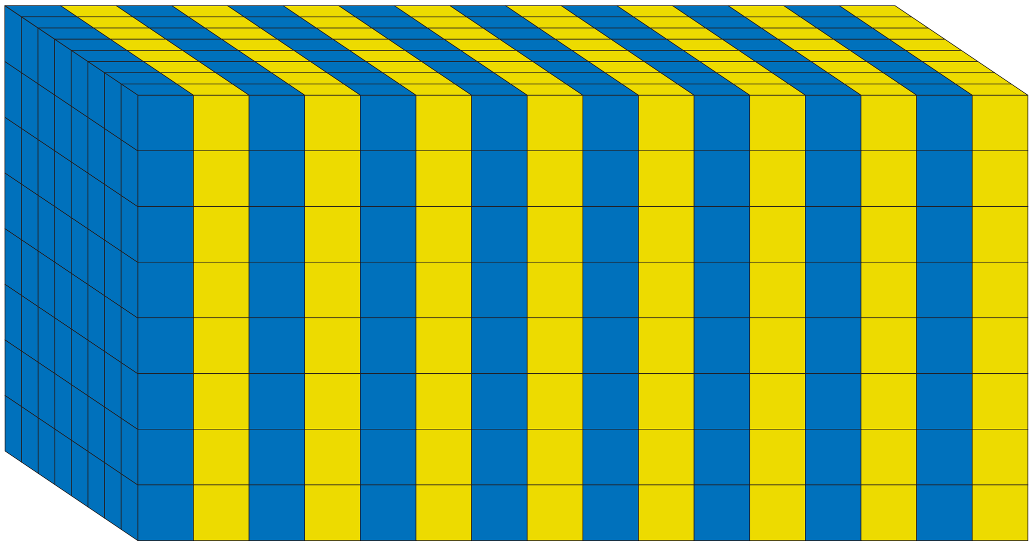
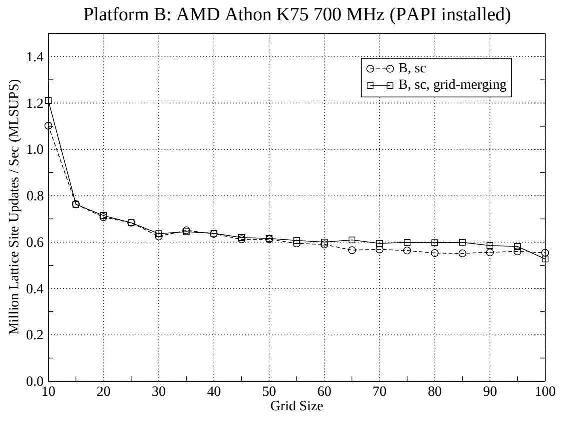


Figure 4.6: The two grids are merged together with grid merging: The cells from the source grid and the cells from the destination grid, which belong together, lie directly abreast. For this case, a single grid with the size of two grids was allocated.

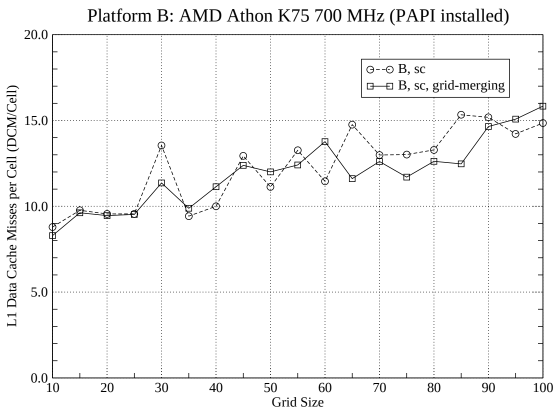
4.2.3 Compressed Grid

The second investigated data layout is compressed grid. This method efficiently reduces the amount of memory needed for the LBM by combining the two grids into one, as can be seen in Figure 4.8: for large grid sizes, compressed grid nearly halves the memory requirement of the problem. But by combining the two grids, the problem of data dependency gets more complex. To avoid overwriting data needed subsequently, the algorithm is divided in two phases (Algorithm 4.3):

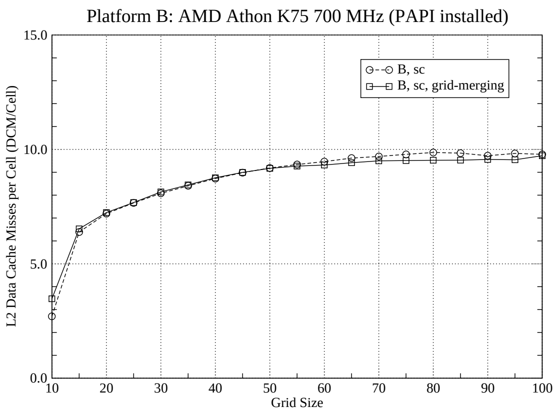
Figures 4.9 and 4.10 illustrate the two phases of the compressed grid data layout. In the first phase, the whole grid will be shifted along the displacement vector (+1,+1,+1). By starting with the top-north-east cell and working virtually reversed, the data dependencies are not violated. After the first complete Lattice Boltzmann step and after the whole grid has been moved, the second phase ”transports” the grid back to the starting position. Starting with the bottom-south-west cell, again to pay attention to the data dependencies, the whole grid is shifted along a (-1,-1,-1) vector, using the original order of cell updates, which has already been used for the two grid algorithms.



(a) Performance in MLSUPS



(b) L1 Date Cache Misses



(c) L2 Data Cache Misses

Figure 4.7: Benchmarking results for grid merging measured with PAPI on Platform B.

Dividing the algorithm in two phases and always starting with the cell positioned at the edge fitting to the displacement vector, guarantees, that no values, which will still be needed, are overwritten. [Wil03] shows, that this is true for the 2D case. In the case of a 3D grid, this fact still holds true, but to illustrate it, is disproportionately more difficult.

In this manner the subsequently needed data are preserved, whereas the memory requirement is significantly reduced, for large grids even nearly by a factor of 2. Figure 4.8 shows the memory requirements of the implementation of this thesis (Figure 4.11). Again, the grid is surrounded by an additional layer of fluids, which helps to avoid special cases for the obstacle cells. On one side, compressed grid makes larger grid sizes possible, because the allocation of the main memory is reduced, on the other side this also reduces capacity misses and memory traffic because of less memory involved.

Unfortunately, compressed grid is a slight drawback for the intention to create absolutely reusable code. Compressed grid needs, in comparison to the standard two grids, a new loop ordering. Compared to the standard algorithm, the stride of the time-steps is increased to two, because two time-steps have to be calculated until a whole phase of the compressed grid algorithm is executed. Therefore two groups of for loops are required, one to shift the grid along the (+1,+1,+1) vector, the other one to shift the grid backwards. Accordingly, two new LBMSTEP macros have to be introduced, which contain the different "shifting information". But among compressed grid

Algorithm 4.1 Lattice Boltzmann step 1 for compressed grid (Obstacle part)

```
if FLAG == OBSTACLE then
    grid(i+1,j+1,k+1,FLAG) = grid(i,j,k,FLAG)
    grid(i+1,j+1,k+1,C) = grid(i,j,k,C)
    grid(i+1,j+1,k+1,N) = grid(i,j+1,k,S)
    grid(i+1,j+1,k+1,S) = grid(i,j-1,k,N)
    grid(i+1,j+1,k+1,W) = grid(i,j,k-1,E)
    grid(i+1,j+1,k+1,E) = grid(i,j,k+1,W)
    grid(i+1,j+1,k+1,T) = grid(i+1,j,k,B)
    grid(i+1,j+1,k+1,B) = grid(i-1,j,k,T)
    grid(i+1,j+1,k+1,NW) = grid(i,j+1,k-1,SE)
    grid(i+1,j+1,k+1,NE) = grid(i,j+1,k+1,SW)
    grid(i+1,j+1,k+1,SW) = grid(i,j-1,k-1,NE)
    grid(i+1,j+1,k+1,SE) = grid(i,j-1,k+1,NW)
    grid(i+1,j+1,k+1,TN) = grid(i+1,j+1,k,BS)
    grid(i+1,j+1,k+1,TS) = grid(i+1,j-1,k,BN)
    grid(i+1,j+1,k+1,TW) = grid(i+1,j,k-1,BE)
    grid(i+1,j+1,k+1,TE) = grid(i+1,j,k+1,BW)
    grid(i+1,j+1,k+1,BN) = grid(i-1,j+1,k,TS)
    grid(i+1,j+1,k+1,BS) = grid(i-1,j-1,k,TN)
    grid(i+1,j+1,k+1,BW) = grid(i-1,j,k-1,TE)
    grid(i+1,j+1,k+1,BE) = grid(i-1,j,k+1,TW)
    continue
end if
...
```

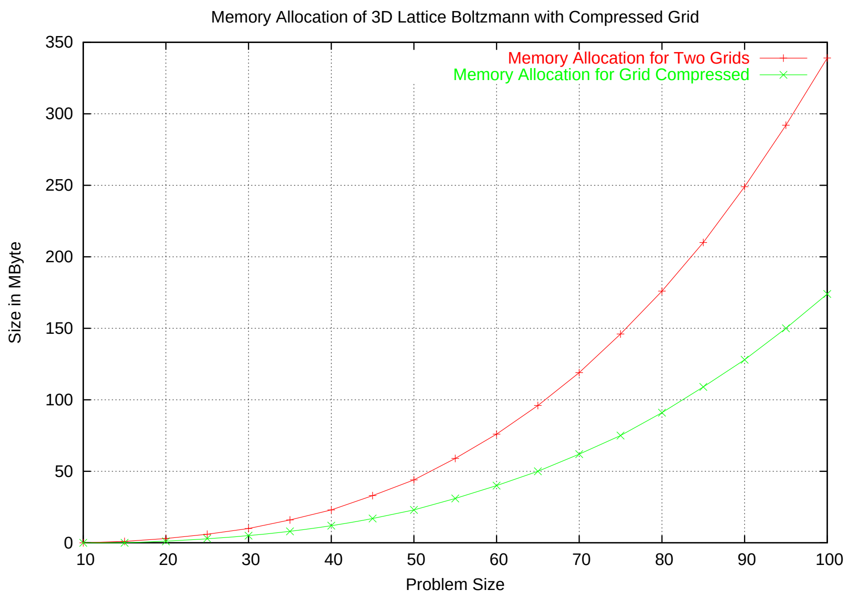


Figure 4.8: The memory requirement of grid compression is nearly halved compared to the two grids data layout and can be derived by the following equation: $(GridSize + 3)^3 \cdot 20 \cdot 8$ Byte. This equation is only valid, if no blocking technique is applied!

versions, the usage of macros can again be done as before in the two grid algorithms: to switch from one version to another, only new macros for the for loop headers have to be developed and the LBMSTEP macro for the stream-collide or collide-stream version has to be inserted twice (one for the (+1,+1,+1) vector, the other for the (-1,-1,-1) vector). It is possible to introduce a variable, which contains the information of the shifting direction. With this variable it could be possible, to reduce the two Lattice Boltzmann step macros to one. This could be done very easily for the stream-collide macro, but not as easily in the collide-stream macro. In this case, the introduction of the variable would mean some additional floating-point operations. Due to this fact, it was decided to use two macros instead, which provides faster code and which means no additional code after compiling.

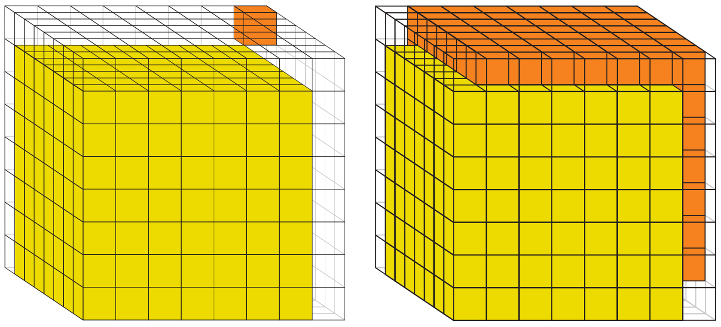
Algorithm 4.2 Lattice Boltzmann step 1 for compressed grid (Acceleration and Fluid Part)

```
...
if FLAG == ACCELERATION then
    rho = 1.0
    u_x = 0.05
    u_y = 0.0
    u_z = 0.0
else
    u_x = (((grid(i,j,k-1,E) +grid(i,j-1,k-1,NE)) +(grid(i,j+1,k-1,SE) +grid(i-1,j,k-1,TE))) +grid(i+1,j,k-1,BE))
    u_y = ((grid(i,j-1,k,N) +grid(i,j-1,k+1,NW)) +(grid(i-1,j-1,k,TN) +grid(i+1,j-1,k,BN)))
    u_z = ((grid(i-1,j,k,T) +grid(i-1,j+1,k,TS)) +grid(i-1,j,k+1,TW))
    rho = ((((grid(i,j,k,C) +grid(i,j+1,k,S)) +(grid(i,j,k+1,W) +grid(i+1,j,k,B))) +((grid(i,j+1,k+1,SW) +grid(i+1,j+1,k,BS)) +(grid(i+1,j,k+1,BW) +u_x))) +(u_y +u_z))
    temp = 1 / rho
    u_x = (u_x -grid(i,j,k+1,W) -grid(i,j-1,k+1,NW) -grid(i,j+1,k+1,SW) -grid(i-1,j,k+1,TW) -grid(i+1,j,k+1,BW)) · temp
    u_y = (u_y -grid(i,j+1,k,S) +grid(i,j-1,k-1,NE) -grid(i,j+1,k+1,SW) -grid(i,j+1,k-1,SE) -grid(i-1,j+1,k,TS) -grid(i+1,j+1,k,BS)) · temp
    u_z = (u_z -grid(i+1,j,k,B) +grid(i-1,j-1,k,TN) +grid(i-1,j,k-1,TE) -grid(i+1,j-1,k,BN)-grid(i+1,j+1,k,BS)-grid(i+1,j,k+1,BW) -grid(i+1,j,k-1,BE)) · temp
end if
u_sqr_trm = 1.5 · (u_x · u_x + u_y · u_y + u_z · u_z)
w_0 = OMEGA · w_0 · rho
w_1 = OMEGA · w_1 · rho
w_2 = OMEGA · w_2 · rho
grid(i+1,j+1,k+1,FLAG) = grid(i,j,k,FLAG)
grid(i+1,j+1,k+1,C) = ((1.0-OMEGA) · grid(i,j,k,C) + (w_0 · (1.0-u_sqr_trm)))
grid(i+1,j+1,k+1,N) = ((1.0-OMEGA) · grid(i,j-1,k,N) + (w_1 · ((1.0-u_sqr_trm)+3.0 · u_y+4.5 · (u_y · u_y))))
grid(i+1,j+1,k+1,S) = ((1.0-OMEGA) · grid(i,j+1,k,S) + (w_1 · ((1.0-u_sqr_trm)-3.0 · u_y+4.5 · (u_y · u_y))))
grid(i+1,j+1,k+1,W) = ((1.0-OMEGA) · grid(i,j,k+1,W) + (w_1 · ((1.0-u_sqr_trm)-3.0 · u_x+4.5 · (u_x · u_x))))
grid(i+1,j+1,k+1,E) = ((1.0-OMEGA) · grid(i,j,k-1,E) + (w_1 · ((1.0-u_sqr_trm)+3.0 · u_x+4.5 · (u_x · u_x))))
grid(i+1,j+1,k+1,T) = ((1.0-OMEGA) · grid(i-1,j,k,T) + (w_1 · ((1.0-u_sqr_trm)+3.0 · u_z+4.5 · (u_z · u_z))))
grid(i+1,j+1,k+1,B) = ((1.0-OMEGA) · grid(i+1,j,k,B) + (w_1 · ((1.0-u_sqr_trm)-3.0 · u_z+4.5 · (u_z · u_z))))
grid(i+1,j+1,k+1,NW) = ((1.0-OMEGA) · grid(i,j-1,k+1,NW) + (w_2 · ((1.0-u_sqr_trm)+3.0 · (u_y-u_x)+4.5 · ((u_y-u_x) · (u_y-u_x)))))
grid(i+1,j+1,k+1,NE) = ((1.0-OMEGA) · grid(i,j-1,k-1,NE) + (w_2 · ((1.0-u_sqr_trm)+3.0 · (u_y+u_x)+4.5 · ((u_y+u_x) · (u_y+u_x)))))
grid(i+1,j+1,k+1,SW) = ((1.0-OMEGA) · grid(i,j+1,k+1,SW) + (w_2 · ((1.0-u_sqr_trm)+3.0 · (-u_y-u_x)+4.5 · ((-u_y-u_x) · (-u_y-u_x)))))
grid(i+1,j+1,k+1,SE) = ((1.0-OMEGA) · grid(i,j+1,k-1,SE) + (w_2 · ((1.0-u_sqr_trm)+3.0 · (-u_y+u_x)+4.5 · ((-u_y+u_x) · (-u_y+u_x)))))
grid(i+1,j+1,k+1,TN) = ((1.0-OMEGA) · grid(i-1,j-1,k,TN) + (w_2 · ((1.0-u_sqr_trm)+3.0 · (u_y+u_z)+4.5 · ((u_y+u_z) · (u_y+u_z)))))
grid(i+1,j+1,k+1,TS) = ((1.0-OMEGA) · grid(i-1,j+1,k,TS) + (w_2 · ((1.0-u_sqr_trm)+3.0 · (-u_y+u_z)+4.5 · ((-u_y+u_z) · (-u_y+u_z)))))
grid(i+1,j+1,k+1,TW) = ((1.0-OMEGA) · grid(i-1,j,k+1,TW) + (w_2 · ((1.0-u_sqr_trm)+3.0 · (-u_x+u_z)+4.5 · ((-u_x+u_z) · (-u_x+u_z)))))
grid(i+1,j+1,k+1,TE) = ((1.0-OMEGA) · grid(i-1,j,k-1,TE) + (w_2 · ((1.0-u_sqr_trm)+3.0 · (u_x+u_z)+4.5 · ((u_x+u_z) · (u_x+u_z)))))
grid(i+1,j+1,k+1,BN) = ((1.0-OMEGA) · grid(i+1,j-1,k,BN) + (w_2 · ((1.0-u_sqr_trm)+3.0 · (u_y-u_z)+4.5 · ((u_y-u_z) · (u_y-u_z)))))
grid(i+1,j+1,k+1,BS) = ((1.0-OMEGA) · grid(i+1,j+1,k,BS) + (w_2 · ((1.0-u_sqr_trm)+3.0 · (-u_y-u_z)+4.5 · ((-u_y-u_z) · (-u_y-u_z)))))
grid(i+1,j+1,k+1,BW) = ((1.0-OMEGA) · grid(i+1,j,k+1,BW) + (w_2 · ((1.0-u_sqr_trm)+3.0 · (-u_x-u_z)+4.5 · ((-u_x-u_z) · (-u_x-u_z)))))
grid(i+1,j+1,k+1,BE) = ((1.0-OMEGA) · grid(i+1,j,k-1,BE) + (w_2 · ((1.0-u_sqr_trm)+3.0 · (u_x-u_z)+4.5 · ((u_x-u_z) · (u_x-u_z)))))
```

Algorithm 4.3 Compressed Grid

```
// 2 Grids:
for $t = 1$ to $timesteps$ by 1 do
    for $i = 1$ to $dim$ by 1 do
        for $j = 1$ to $dim$ by 1 do
            for $k = 1$ to $dim$ by 1 do
                LBMSTEP
            end for
        end for
    end for
    swap( src, dst )
end for
```

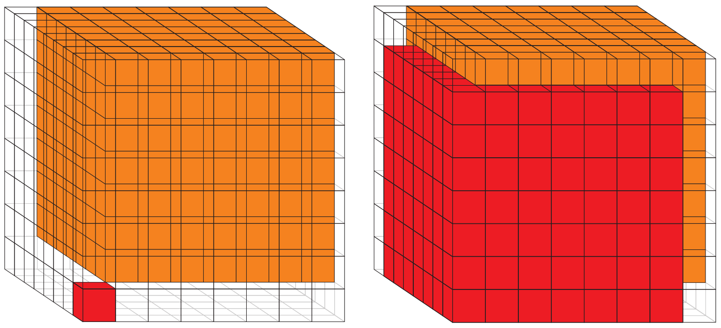
```
// Compressed Grid:
for $t = 1$ to $timesteps$ by 2 do
    // First phase of compressed grid
    for $i = dim$ to 1 by $-1$ do
        for $j = dim$ to 1 by $-1$ do
            for $k = dim$ to 1 by $-1$ do
                LBMSTEP_1
            end for
        end for
    end for
    // Second phase of compressed grid
    for $i = 1$ to $dim$ by 1 do
        for $j = 1$ to $dim$ by 1 do
            for $k = 1$ to $dim$ by 1 do
                LBMSTEP_2
            end for
        end for
    end for
end for
```



(a) The first cell is updated and shifted by the (+1,+1,+1) vector

(b) After the first Lattice Boltzmann step, the whole grid is shifted

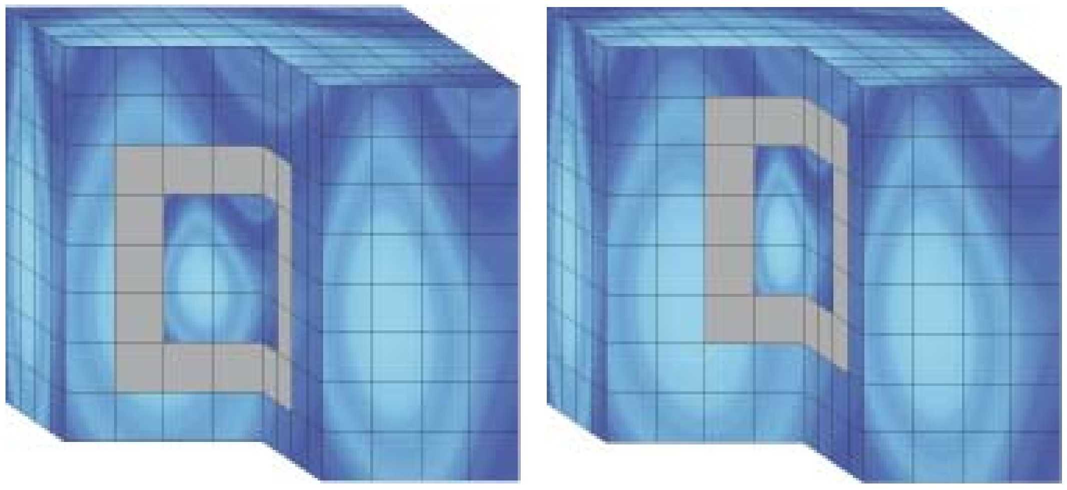
Figure 4.9: The first phase of the grid compressed algorithm.



(a) The first cell is updated and shifted by the (-1,-1,-1) displacement vector

(b) After the second Lattice Boltzmann step, the whole grid is at the starting position again

Figure 4.10: The second phase of the grid compressed algorithm.



(a) Grid Compression before Phase 1 (b) Grid Compression before Phase 2

Figure 4.11: The implementation of Compressed Grid requires nearly half of the memory of two grids.

Algorithm 4.1 shows the obstacle part of the grid compressed algorithm. In this part, the differences between the grid compressed and the two grids algorithms become already obvious: there is no distinction between a source and a destination grid, but only a single "grid". Also, the new values are written to the cell in the (+1,+1,+1) direction in the case of the stream-collide order. Algorithm 4.2 shows the acceleration and fluid part of the grid compressed algorithm, which shows the same differences compared to the two grids algorithm. Although grid compressed algorithms seem much more complicated, this data layout offers exactly the same possibilities for cache optimization techniques as the two grid data layout. Consequently, in the following chapter all techniques will be applied to both data layouts.

4.3 Access Pattern Optimization by Loop Blocking

Algorithm 4.4 1-way blocking of the LBM

```
1: for $tt = 1$ to $timesteps$ by $TIMEBLOCK$ do
2:   for $t = 1$ to $TIMEBLOCK$ by $1$ do
3:     for $i = 1$ to $dim$ by $1$ do
4:       for $j = 1$ to $dim$ by $1$ do
5:         for $k = 1$ to $dim$ by $1$ do
6:           LBMSTEP
7:         end for
8:       end for
9:     end for
10:    swap( src, dst )
11:  end for
12: end for
```

To improve performance, it is very important to remember how slow the main memory actually is. In order to speed up an application, which relies on the main memory, it is advisable to reduce memory transfer between the main memory and the CPU. Data should be provided from the much faster caches in order to keep the CPU busy. One possibility is to use the data still in cache several times before replacing it with new data. This can be done by the cache optimization technique of blocking, which generally changes the order of data access. There are several levels of blocking: for each loop, which is divided into two loops, the level of blocking is increased by one. For example, if only one of the four basic loops of the LBM, e.g. the time loop, is divided into two loops, then one would call this technique 1-way blocking (Algorithm 4.4). Although the 1-way blocking technique is used in [Wil03], it obviously makes no sense to apply this technique for the 3D-LBM, because of the memory requirements of a 3D grid. In this thesis, only 3-way blocking, where the three dimensions of space were blocked, and 4-way blocking, where all four loops were blocked (the three dimensions of space and the time loop), were implemented. While developing these versions, 3-way blocking should only provide an easier way to the technique of 4-way blocking and was never expected to be a useful technique, but the performance measurements showed some interesting results. This, and the fact, that 3-way blocking might provide a better understanding of 4-way blocking, are the reasons, why both 3-way and 4-way blocking are explained in detail.

Since blocking techniques are a common cache optimization technique, even compilers can handle similar attempts to increase the performance of a code. In this thesis, all blocking attempts were done manually, but for more information about compiler optimizations see [AK01] and [BGS94].

4.3.1 3-Way Blocking

Algorithm 4.5 3-way blocking of the LBM

```
for $t = 1$ to $timesteps$ by 1 do
  for $ii = 1$ to $dim$ by $BLOCKSIZE\_Z$ do
    for $jj = 1$ to $dim$ by $BLOCKSIZE\_Y$ do
      for $kk = 1$ to $dim$ by $BLOCKSIZE\_X$ do
        for $i = ii$ to $I\_END$ by 1 do
          for $j = jj$ to $J\_END$ by 1 do
            for $k = kk$ to $K\_END$ by 1 do
              LBMSTEP
            end for
          end for
        end for
      end for
    end for
  end for
  swap( src, dst )
end for
```

Algorithm 4.6 The macros for the 3-way blocking

```
#define MIN(a,b) ((a¡b)?(a):(b))
#define I_END MIN(ii+BLOCKSIZE_Z,(dim))
#define J_END MIN(jj+BLOCKSIZE_Y,(dim))
#define K_END MIN(kk+BLOCKSIZE_X,(dim))
```

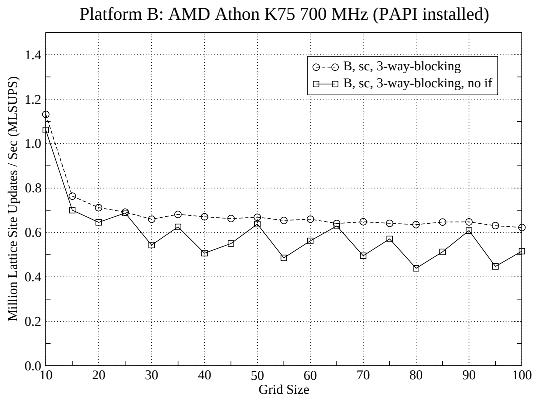
With the technique of 3-way blocking, the three loops for the spatial dimensions are splitted into altogether six loops (Algorithm 4.5). The advantage of this is the reuse of several already loaded cache lines: by choosing a reasonable BLOCKSIZE for each dimension, the currently updated block will fit entirely into the cache (this BLOCKSIZE is dependent on the used data layout). This means, while calculating new values in this block, ideally no new cache lines have to be loaded until a new block is started. Every value needed for the calculations is already in the cache, for a cell in plane i and also for a cell in plane i+1. Also, nearly all cache lines could be completely used in the ideal case. This would not be possible for the whole grid, due to the memory requirement of the entire grid, which can be seen in Figures 4.5 and 4.8. Though this is not the best possible cache reuse, this is the best technique, if every time-step is important and is to be visualized for example. With the technique of 4-way blocking, only a fraction of all time-steps could be visualized. Figures 4.13 and 4.14 illustrate the technique of 3-way blocking. Important to notice is the fact, that every small block, which is cut out of the entire grid, is updated in exactly the same manner as the whole grid in the unblocked versions. Subsequently, the technique of 3-way blocking will be applied to both data layouts, two grids and grid compressed. These illustrations are supposed to create a better understanding of the quite complex processes during a Lattice Boltzmann step in 3D.

Algorithm 4.7 3-way blocking without if statements in the loop headers (part 1)

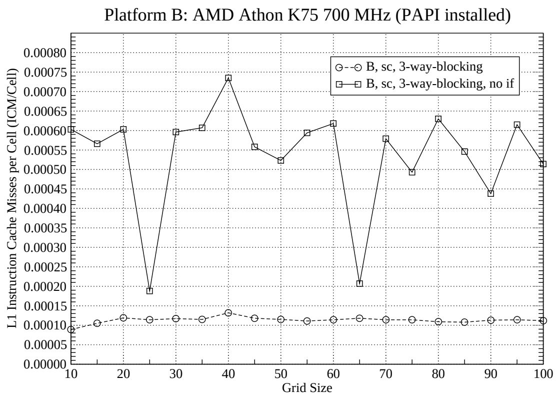
```
1: for $t = 1$ to $timesteps$ by 1 do
2:   for $ii = 1$ to $dim - BLOCKSIZE\_Z$ by $BLOCKSIZE\_Z$ do
3:     for $jj = 1$ to $dim - BLOCKSIZE\_Y$ by $BLOCKSIZE\_Y$ do
4:       for $kk = 1$ to $dim - BLOCKSIZE\_X$ by $BLOCKSIZE\_X$ do
5:         for $i = ii$ to $ii + BLOCKSIZE\_Z$ by 1 do
6:           for $j = jj$ to $jj + BLOCKSIZE\_Y$ by 1 do
7:             for $k = kk$ to $kk + BLOCKSIZE\_X$ by 1 do
8:               LBMSTEP
9:             end for
10:          end for
11:        end for
12:      end for
13:      for $i = ii$ to $ii + BLOCKSIZE\_Z$ by 1 do
14:        for $j = jj$ to $jj + BLOCKSIZE\_Y$ by 1 do
15:          for $k = kk$ to $dim$ by 1 do
16:            LBMSTEP
17:          end for
18:        end for
19:      end for
20:    end for
21:    for $kk = 1$ to $dim - BLOCKSIZE\_X$ by $BLOCKSIZE\_X$ do
22:      for $i = ii$ to $ii + BLOCKSIZE\_Z$ by 1 do
23:        for $j = jj$ to $dim$ by 1 do
24:          for $k = kk$ to $kk + BLOCKSIZE\_X$ by 1 do
25:            LBMSTEP
26:          end for
27:        end for
28:      end for
29:    end for
30:    for $i = ii$ to $ii + BLOCKSIZE\_Z$ by 1 do
31:      for $j = jj$ to $dim$ by 1 do
32:        for $k = kk$ to $dim$ by 1 do
33:          LBMSTEP
34:        end for
35:      end for
36:    end for
37:  end for
38:  for $jj = 1$ to $dim - BLOCKSIZE\_Y$ by $BLOCKSIZE\_Y$ do
39:    for $kk = 1$ to $dim - BLOCKSIZE\_X$ by $BLOCKSIZE\_X$ do
40:      for $i = ii$ to $dim$ by 1 do
41:        for $j = jj$ to $jj + BLOCKSIZE\_Y$ by 1 do
42:          for $k = kk$ to $kk + BLOCKSIZE\_X$ by 1 do
43:            LBMSTEP
44:          end for
45:        end for
46:      end for
47:      for $i = ii$ to $dim$ by 1 do
48:        for $j = jj$ to $jj + BLOCKSIZE\_Y$ by 1 do
49:          for $k = kk$ to $dim$ by 1 do
50:            LBMSTEP
51:          end for
52:        end for
53:      end for
54:    end for
55:  end for
56:  for $kk = 1$ to $dim - BLOCKSIZE\_X$ by $BLOCKSIZE\_X$ do
57:    for $i = ii$ to $dim$ by 1 do
58:      for $j = jj$ to $dim$ by 1 do
59:        for $k = kk$ to $kk + BLOCKSIZE\_X$ by 1 do
60:          LBMSTEP
61:        end for
62:      end for
63:    end for
64:  end for
65:  ...
66: end for
```

Algorithm 4.8 3-way blocking without if statements in the loop headers (part 2)

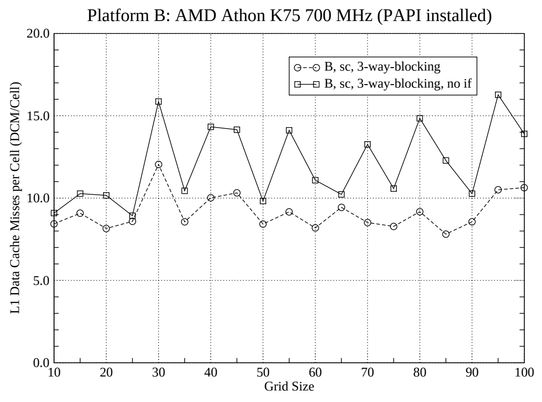
```
...
for $i = ii$ to $dim$ by $1$ do
    for $j = jj$ to $dim$ by $1$ do
        for $k = kk$ to $dim$ by $1$ do
            LBMSTEP
        end for
    end for
end for
swap( src, dst )
...
```



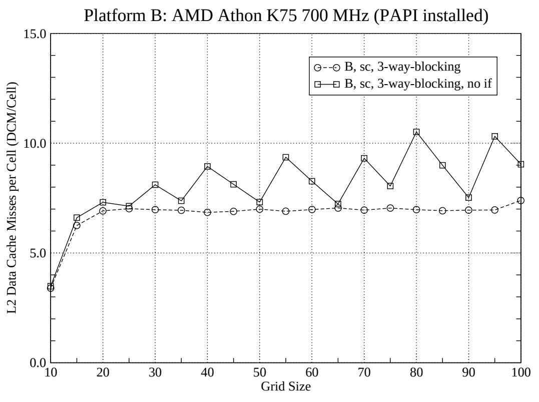
(a) Performance in MLSUPS



(b) L1 Instruction Cache Misses



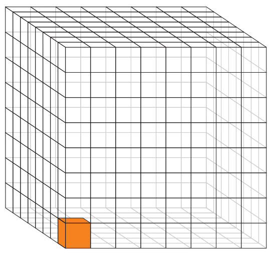
(c) L1 Data Cache Misses

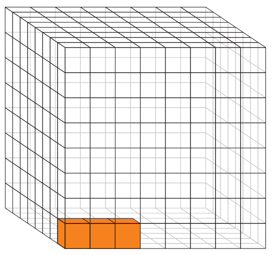


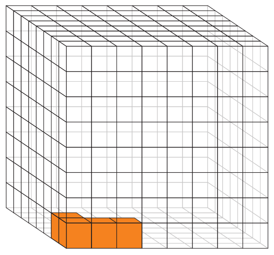
(d) L2 Data Cache Misses

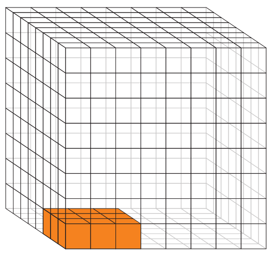
Figure 4.12: Benchmarking results for 3-way blocking without if statement in the loop heads, measured with PAPI on Platform B.

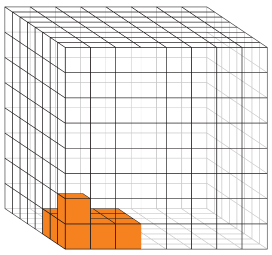
Algorithm 4.6 shows the macros, which were used in the loop headers. The usage of macros is a very good idea in terms of performance as can be seen in Figure 4.12. Both the instruction cache misses and the data cache misses are highly increased, if instead of if statements in the loop heads every exception at the grid boundaries is handled in special cases, as is shown in Algorithm 4.7. This increase in cache misses can easily be noticed by the very unstable and decreased performance of the LBM. With this insight, both 3-way blocking and 4-way blocking, which would be even worse in terms of special case handling, were implemented with the help of macros in the loop heads, which contain if statements in the MIN(.,.) macro. Obviously, the number of evaluations of the if statements is very small compared to the calculations done in the loop body. In respect of readability and software engineering, the 3-way blocked code with macros also offers much more manageability and, yet again, reuse of the macros.

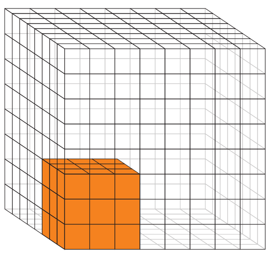

(a) First updated cell

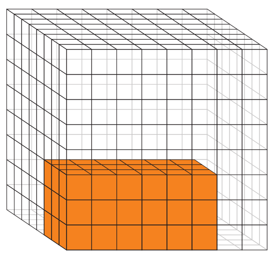

(b) First updated row


(c) Next row is started


(d) First updated plane


(e) Next plane is started


(f) First block is updated


(g) Two blocks are completely updated

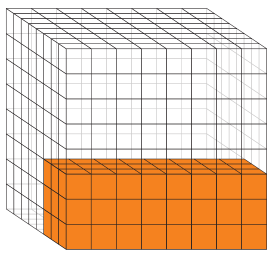
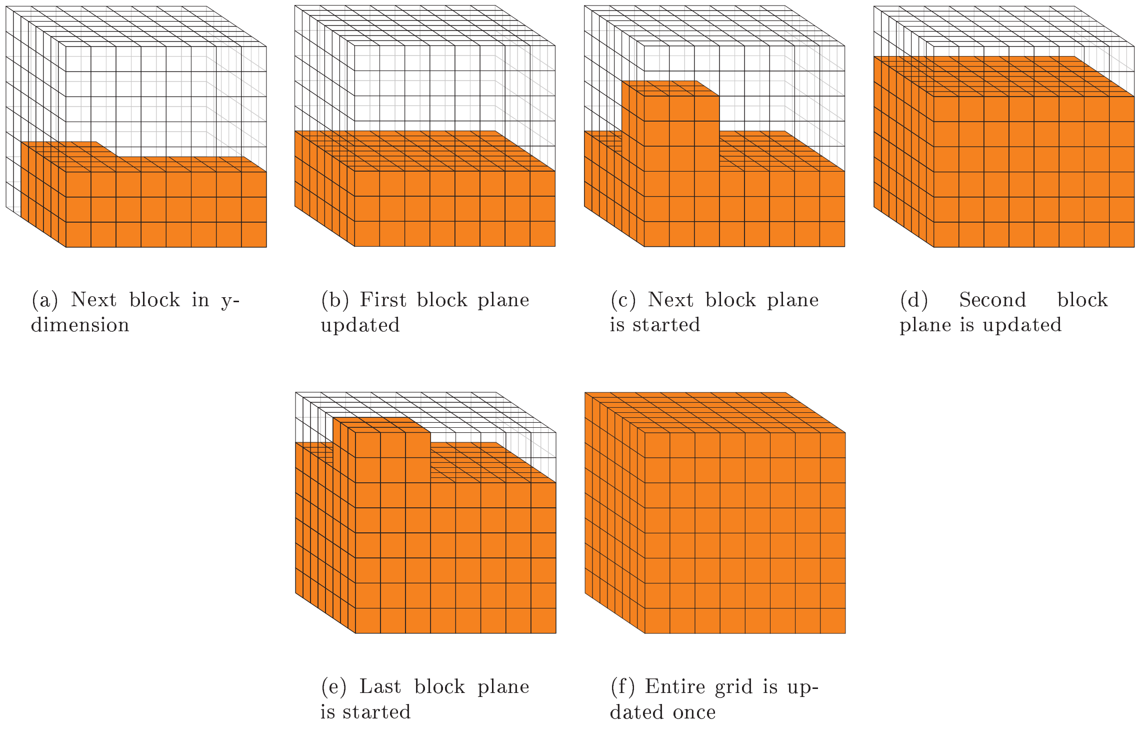

(h) Third block is updated

Figure 4.13: 3-way blocking, part 1.

The following figures explain the 3-way blocking algorithm (see Algorithm 4.5) further: The first six pictures (Figures 4.13(a)-(f)) show how to process a whole block. Instead of calculating an entire row, only BLOCKSIZE_X cells are updated. This limit is set by the macro in the innermost loop. Grid (d) shows, that the entire first plane of the block has been updated. Only BLOCKSIZE_Y rows have been updated, as has been specified in the second innermost loop. The next loop sets the limit of planes of the block to BLOCKSIZE_Z, as is shown in Grid (f). This order of computation is applied for every block. Whenever the computation reaches a block, it is updated exactly in this pattern.

The next picture (Figure 4.13(g)) show, that the next block is also updated in the described manner.

The next block is smaller than the two blocks before, because the boundary of the grid has been reached in x dimension (Figure 4.13(h)). Only two cells in x dimension are updated. This behaviour is determined in the macros, which decide, if the stride can be a full BLOCKSIZE or if the end of the loop range has to be the dimension of the grid, which is the highest value the four loop indices may reach. Although this control means the introduction of three if statements ((...)?(...):(...) is an equivalent construction in C), these if statements do not decrease performance. The innermost if has to be calculated only once for three cells in this example, the second innermost if only once for nine cells and the next only once for all 27 cells. For bigger block sizes, the number of evaluations for these if statements are significantly reduced. Additionally, the CPU has lots of time to precalculate these values, because of the big loop body. And, as a very positive development, modern CPUs do not suffer too badly from if statements, because of much better jump predictions.


(a) Next block in y-dimension

(b) First block plane updated

(c) Next block plane is started

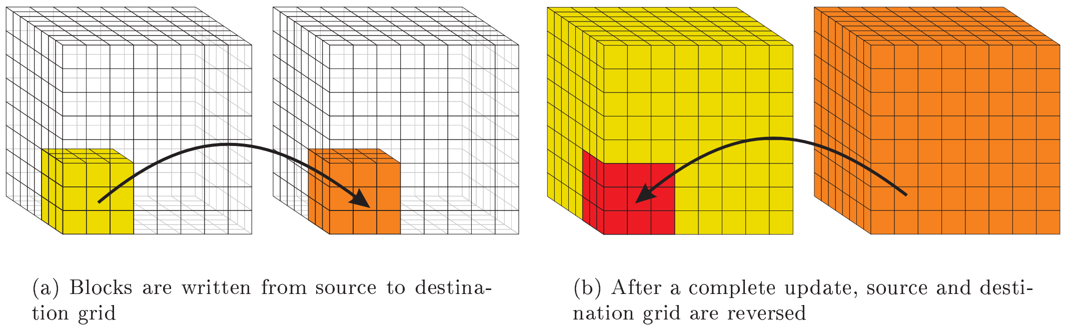
(d) Second block plane is updated

(e) Last block plane is started

(f) Entire grid is updated once

Figure 4.14: 3-way blocking, part 2.

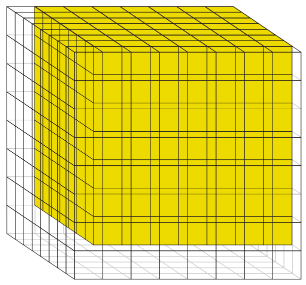
The next illustrations (Figure 4.14) show the remaining block updates, until the whole grid is updated once. After that, the entire grid is updated a second time.

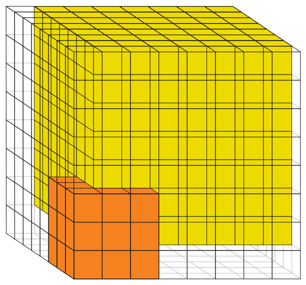

(a) Blocks are written from source to destination grid

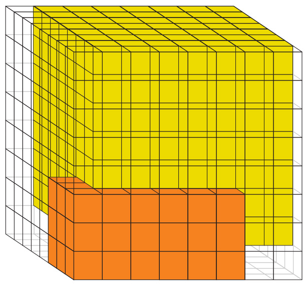
(b) After a complete update, source and destination grid are reversed

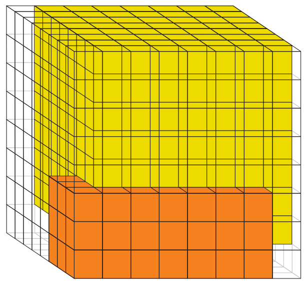
Figure 4.15: 3-way blocking with two grids.

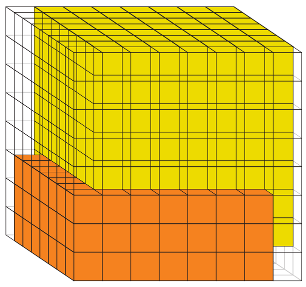
The two parts of Figure 4.15 show, how 3-way blocking works with two grids. First, all cells in the source grid to the left are updated once in blocks and written to the destination grid to the corresponding cells. After the entire grid has been updated once, the source and destination grids are reversed and again and all cells from the source grid are updated into the destination grid.

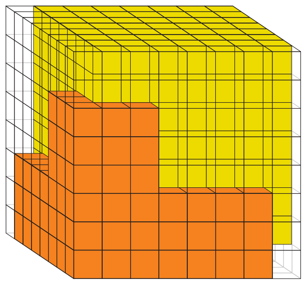

(a) The initial state of the grid


(b) The first block is updated


(c) The second block is updated


(d) The third block is updated


(e) After the first ii iteration


(f) The first block of the second ii iteration is updated

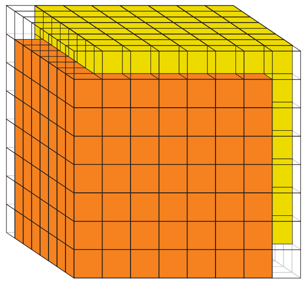

(g) The whole grid is updated and shifted by the (-1,-1,-1) displacement vector

Figure 4.16: 3-way blocking with grid compression.

Figure 4.16 shows, how 3-way blocking works with the grid compression data layout. In this case, the second phase, where the grid is shifted along a (-1,-1,-1) vector, is used as an example, because this perspective allows more insight. As described, all blocks are updated in the same manner. But instead of writing to another grid, the updated cells are shifted along the (-1,-1,-1) vector, until the whole grid has been moved (g).

4.3.2 4-Way Blocking

Algorithm 4.9 4-way blocking of the LBM

```
for $t = 1$ to $timesteps$ by $TIMEBLOCK$ do
  for $ii = 1$ to $dim$ by $BLOCKSIZE\_Z$ do
    for $jj = 1$ to $dim$ by $BLOCKSIZE\_Y$ do
      for $kk = 1$ to $dim$ by $BLOCKSIZE\_X$ do
        for $t = 1$ to $TIMEBLOCK$ by $1$ do
          for $i = I\_START$ to $I\_END$ by $1$ do
            for $j = J\_START$ to $J\_END$ by $1$ do
              for $k = K\_START$ to $K\_END$ by $1$ do
                LBMSTEP
              end for
            end for
          end for
          swap( src, dst )
        end for
      end for
    end for
  end for
end for
```

Algorithm 4.10 The macros for the 4-way blocking

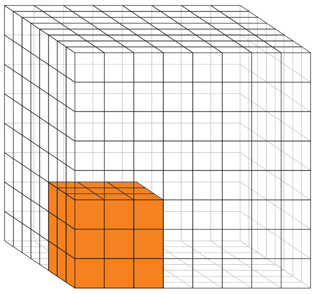
```
#define I_START ((1>ii-t*planeSize)?1:(ii-t))
#define J_START ((1>jj-t*rowSize)?1:(jj-t))
#define K_START ((1>kk-t*CELLSIZE)?1:(kk-t))
#define I_END ((ii+BLOCKSIZE_Z >= (dim+1)) ? (dim+1) : (ii+BLOCKSIZE_Z-t))
#define J_END ((jj+BLOCKSIZE_Y >= (dim+1)) ? (dim+1) : (jj+BLOCKSIZE_Y-t))
#define K_END ((kk+BLOCKSIZE_X >= (dim+1)) ? (dim+1) : (kk+BLOCKSIZE_X-t))
```

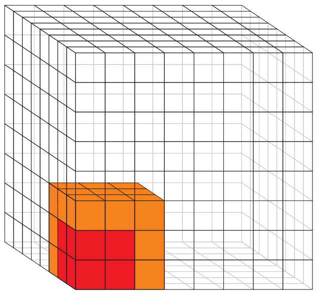
4-way blocking is very much related to 3-way blocking, but in addition the time loop is blocked (Algorithm 4.9). This means, that a block is updated several times, before a new block is started. Using this technique, the cache reuse can be efficiently increased, because, as described with 3-way blocking, until the next block is started, ideally no memory traffic is needed and the block is updated several times. The disadvantage of this technique is that a single time-step cannot be visualized, because while updating the grid, several time-steps are merged into single passes through the grid. In order to respect all data dependencies, the block sizes have to be adapted. This is done in the macros, which control the start and the end value of the loop header indices. In this implementation, the number of if statements has grown to six, but again, the number of evaluations of the if statements is relatively small and does not decrease performance. How this blocking technique works and how the block sizes have to be adapted, is illustrated from Figure 4.17 to Figure 4.22. As in the case of 3-way blocking, the appliance to the two data layouts will be presented afterwards.

A single block is updated exactly like explained in the 3-way blocking case (Figure 4.17). After the first block update, the second update of the first block takes place within this block. The size of the block has to decrease, because all cells, which have to be updated again, must be surrounded by cells, which have already been updated once. This decrease in size has to occur in all three dimensions. After the second update, the block size has again to decrease in order to execute the third update. The third picture shows the first block after three updates.

The second block of the grid is again updated as described with 3-way blocking (Figure 4.18). After that, all values, which lie within the cells, which have been updated once and are not yet updated twice, are updated for the second time (Figure 4.18(b)). Note, that the block size in x dimension does not decrease in order to attach to the already twice updated cells. After that, the cells completely surrounded by twice updated cells are updated a third time (Figure 4.18(c)). Again, the block size in x dimension does not decrease.

First, the third block is updated like in 3-way blocking (Figure 4.19(a)). Then, all cells, which are completely surrounded by updated cells, are updated for the second time (Figure 4.19(b)). In this case, the third block has reached the boundary in x dimension. This means, that, in order to


(a) First block is updated once


(b) Smaller block in the first block is updated for the second time

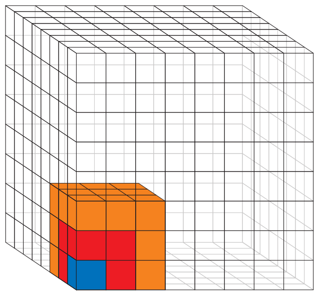
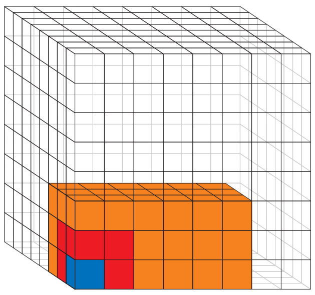
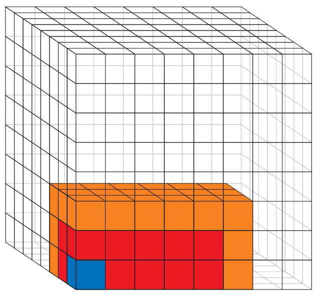

(c) An even smaller block is updated for the third time

Figure 4.17: Three updates within the first block.


(a) The second block is updated once


(b) Within the second block, the second update takes place

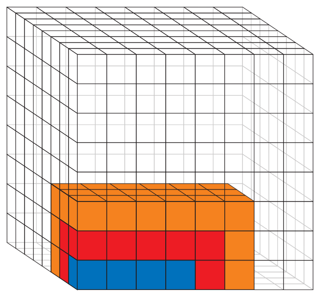
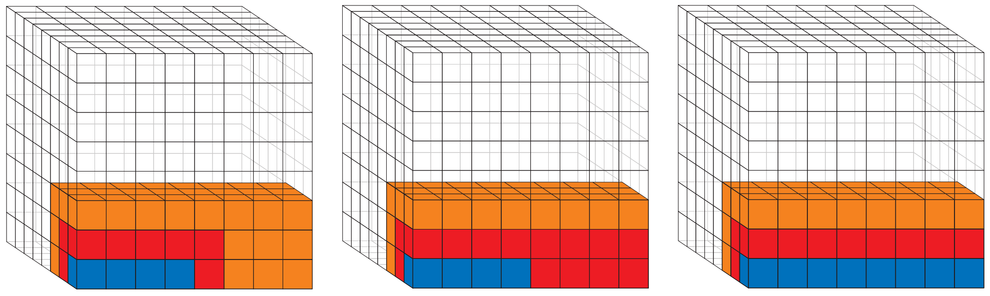

(c) The third update within the second block

Figure 4.18: The second block is updated thrice.

attach to the already twice updated cells, the block size in x dimension has to increase by one. Not two cells in x dimension as in the first update, but three cells have to be updated. Now, all cells within the once updated blocks, are updated twice. After that, the same technique is applied to the third update (Figure 4.19(c)). Again, the x block size has to increase by one in order to attach to the already updated cells.

The next block, which is updated, lies in the y direction (Figure 4.20(a)). It is also updated accordingly to the already explained patterns. Now, the block size in the y dimension has to be adapted to ensure, that all cells are updated. The macro for the y direction turns out to be very similar to the macro for the x direction (4.10). This fact shows again, that developing better versions of Lattice Boltzmann can be done by using the components of a macro tool kit, which were developed separately. The next picture shows the grid after a complete iteration of the outermost spatial loop, the ii-loop (Figure 4.20(b)). The first plane has been updated thrice, the second plane twice and the third and now uppermost plane has been updated only once.

The second iteration of ii works in exactly the same pattern (Figure 4.21): first of all the first block is updated once, then all cells within are updated the second time and after that, all cells, which are surrounded by cells, which have already been updated twice, are updated the third time (Figure 4.21(a)). In this case, the block size in z direction remains three in order to calculate all necessary cells. Again, this can be done by using a macro, which resembles the macros for the x and y direction. After the first block, the second block is updated accordingly (Figure 4.21(b)). Only

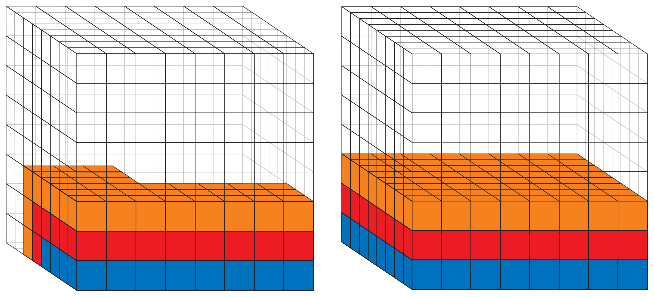


(a) The third block is updated like in 3-way blocking

(b) The block size in x dimension increases in order to update all cells within the second time

(c) Again the x block size is increased to calculate the third update

Figure 4.19: The third block is updated thrice.



(a) The next block in y direction is updated

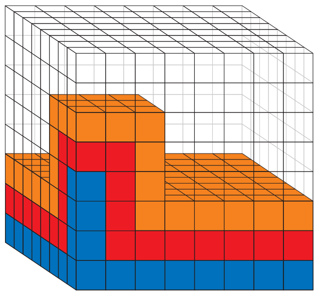
(b) After the first iteration of ii

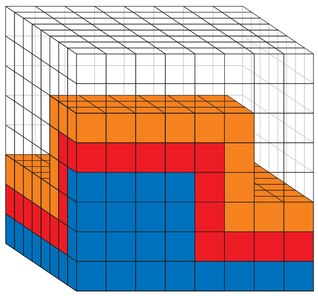
Figure 4.20: After the first iteration of the ii loop, the first plane is updated thrice, the second plane twice and the third plane only once.

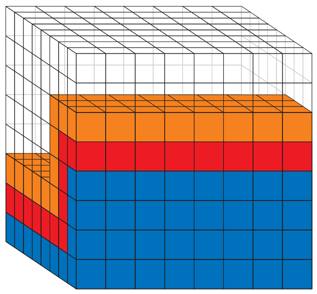
the block size in x dimension has to be reduced in order to stay within the grid. Thereafter, the third block is updated by increasing the block size in x dimension as described in Figure 4.19(c). The next block is again in y direction (Figure 4.21(d)) and so on, until the second iteration of ii is finished (Figure 4.21(e)). The uppermost updated plane was only updated once, the plane below however, could be updated a second time and all planes below that one could even be updated a third time.

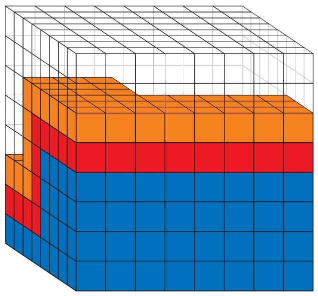
Primarily, the first block of the third iteration is updated (Figure 4.22). Now, for the first time, the block size in z dimension has to be increased. By continuing with the algorithm as described above, the entire grid is updated thrice after the last iteration of ii. This explains, why this blocking technique is not able to visualize every single Lattice Boltzmann step: By combining three time-steps, only every third time-step is observable.

The three Figures 4.23(a) to 4.23(c) illustrate, how 4-way blocking is applied to two grids: After updating the first block for the first time, as it is done in 3-way blocking, the grids are reversed and all cells, which can be updated for the second time, are written to the former source grid, which is now the destination grid. After that, the grids are reversed again and all cells, which can be updated for the third time, are written to the destination grid again. After that, the second block would be updated. Obviously, the destination grid contains the first and the third update step; the source grid contains the second update step and those cells, which are not updated.


(a) The first block of the second iteration of ii is updated


(b) The second block is updated


(c) The third block is updated


(d) The block one step in y direction is updated

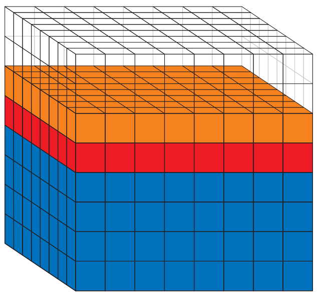
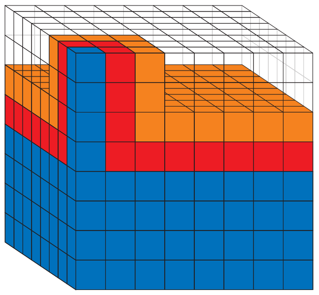
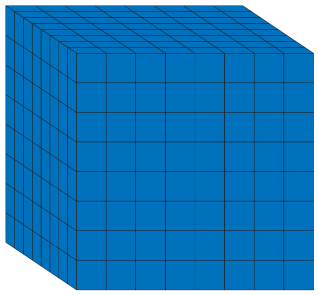

(e) After the second iteration of ii

Figure 4.21: After the second iteration of ii, four planes are updated thrice, the plane above is updated twice and the uppermost plane is only updated once.

To illustrate 4-way blocking with grid compression, the following example was created (Figure 4.24): the initial grid has a dimension of 5 × 5 × 5, three time-steps are to be applied. Figure (a) shows the initial state of the grid and the very high waste of memory, which comes with the small dimension of 5. But for bigger dimensions, the percentage of wasted memory is reduced significantly. The Figures 4.24(b), 4.24(c) and 4.24(d) show the complete update of the first block. Consequently, the newer updates are again shifted along the (-1,-1,-1) vector. All following block updates are equivalent to the already explained 4-way blocking algorithm, but again every new update is shifted by a (-1,-1,-1) vector.

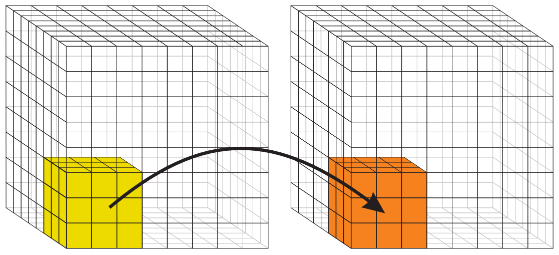


(a) Now the block size in z dimension has to be increased

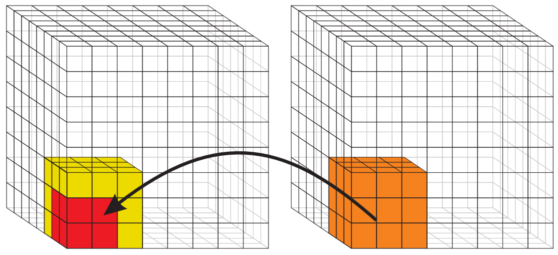


(b) The whole grid is updated thrice

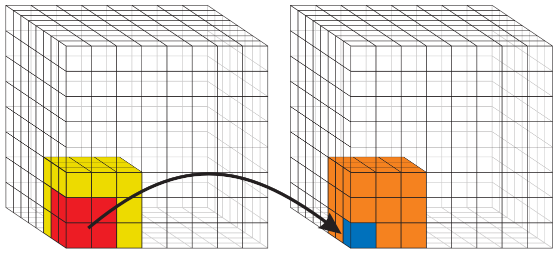
Figure 4.22: In the end, the whole grid is updated thrice and three time-steps have been performed within a single sweep over the entire grid.



(a) One block is updated and written from the source to the destination grid

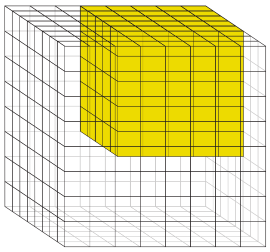


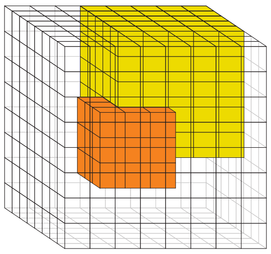
(b) After reversing the grids, all cells completely surrounded by updated cells are updated for the second time

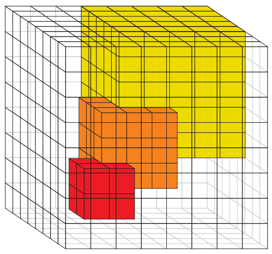


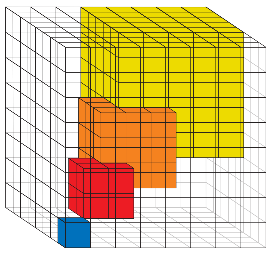
(c) Again the grids are reversed and all inner cells updated

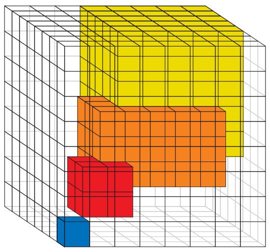
Figure 4.23: 4-way blocking with two grids.

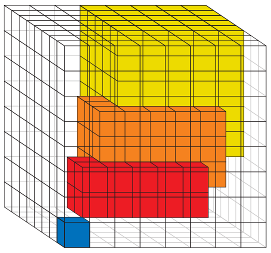

(a) The initial state of the grid

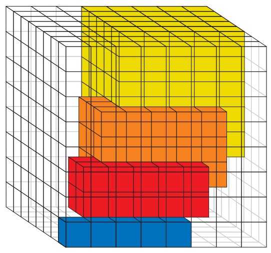

(b) The first block is updated once

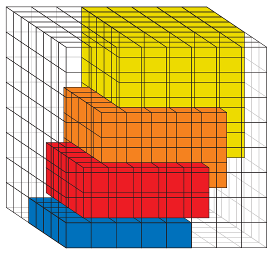

(c) The first block is updated for the second time

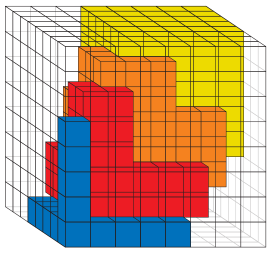

(d) The first block is updated for the third time

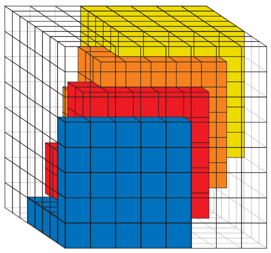

(e) The second block is updated once


(f) The second block is updated for the second time


(g) The second block is updated for the third time


(h) After the first iteration of ii


(i) The first block of the second ii iteration is completely updated


(j) The second block of the second ii iteration is completely updated

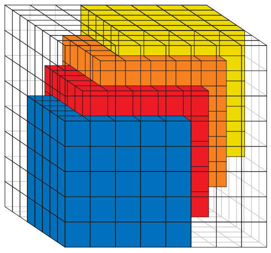

(k) The whole grid is updated thrice

Figure 4.24: 4-way blocking with grid compression.

Chapter 5

Further Optimization Aspects

The behaviour of vector machines is different from the aforementioned architectures, which profit greatly from the so far discussed cache optimization techniques: for vector machines, pipeline-aware programming style should be preferred to cache-aware (for more detailed informations about vector machines, see [HP96]). Vector architectures benefit greatly from long loops. In the case of the LBM, this would mean, that the innermost loop, the loop in x dimension, should be as long as possible. In the extreme case, this would mean, that BLOCKSIZE_X is equivalent to the dimension of the grid. But still this loop length is much too small: instead of 100, a vector architecture would prefer a loop length of several thousand. To fullfill this requirement, a special version of LBM was developed, which guarantees a maximum possible loop length of "$\dim^3$", which is equivalent to the number of cells in the grid.

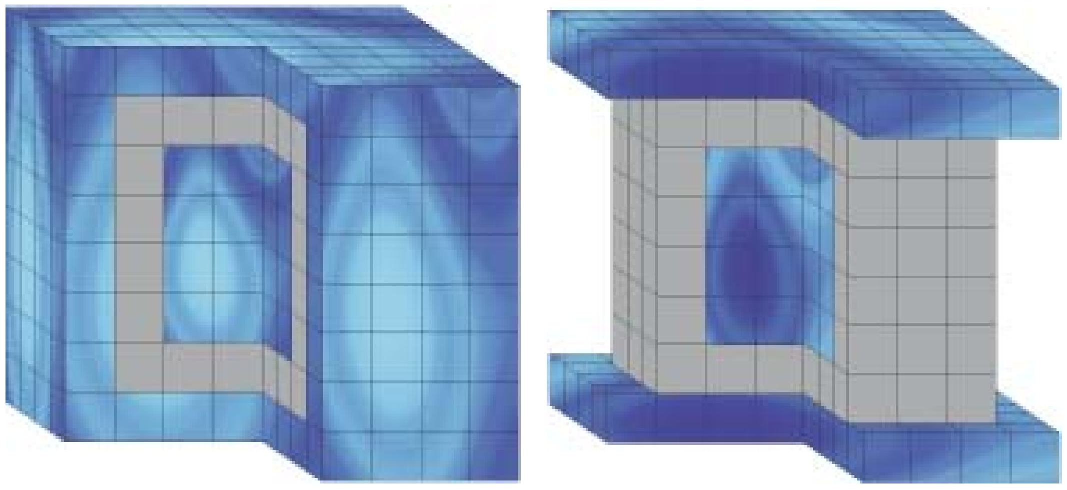
5.1 Vector Machine Version of Lattice Boltzmann

Algorithm 5.1 Vector Optimized Version of LBM

```
// Three inner loops:
double src[dim · dim · dim· CELLSIZE]
double dst[dim · dim · dim· CELLSIZE]
for $t = 1$ to $timesteps$ by 1 do
    #pragma omp parallel for private (#all variables and j and k#)
    for $i = 1$ to $dim$ by 1 do
        for $j = 1$ to $dim$ by 1 do
            for $k = 1$ to $dim$ by 1 do
                LBMSTEP
            end for
        end for
    end for
    swap( src, dst )
end for
```

```
// One inner loop:
double src[dim · dim · dim· CELLSIZE + EXTRA]
double dst[dim · dim · dim· CELLSIZE + EXTRA]
for $t = 1$ to $timesteps$ by 1 do
    #pragma omp parallel for private (#all variables#)
    for $i = 1$ to $dim^3$ by 1 do
        LBMSTEP
    end for
    swap( src, dst )
end for
```

Algorithm 5.1 shows the vector-optimized Lattice Boltzmann code. This code can be based on the integer arithmetic data access version or the pointer arithmetic data access version of the cache-optimized LBM. At first sight the most significant change becomes obvious: the three dimensional loops have been combined into a single loop, which traverses the whole grid. To make this fusion possible, the implementation of the grid had to change: Figure 5.1(b) shows the grid for the vector version of LBM. In contrast to the prior implementations (see Figure 5.1(a)), the grid is not completely surrounded by an additional layer of fluid. Instead, only at the top and bottom of the grid extra fluid cells are positioned, which are symbolized as "EXTRA"in the code. In terms of memory requirement, the vector implementation therefore even slightly reduces the amount of needed memory. Although this implementation might seem as if some special treatment of the obstacle cells has been introduced, the macro for the Lattice Boltzmann step, LBMSTEP, remains unchanged compared to the cache-optimized versions. Intuitively, this would result in a violation of the data dependencies, but in fact, all data dependencies are still respected. The reason for



(a) The implementation of the basic LBM

(b) The implementation of the vector optimized LBM

Figure 5.1: Implementation for the two Grids Versions and for the Vector Machine Optimized Versions.

the correct behaviour lies in two implementation aspects: first, the complete data of the grid is stored in a single-index array in a row-wise manner. This means, that the last obstacle of a row is directly adjacent to the first obstacle in the next row. Because the whole grid is arranged in this manner, the right neighbour of an obstacle cell at the right border will always be an obstacle cell and the left neighbour of an obstacle cell at the left border will also always be an obstacle cell. The second reason is, that the obstacle cells always reverse the streamed values (no-slip condition). If, for example, an obstacle cell at the left border accesses the cell to the TW, which would be a cell which lies on the right border, then the streamed values from the accessed cell would be streamed backwards, in other words, the streaming direction would be reversed. In this manner, values, which are streamed within the grid, still cannot leave the grid and values, which are streamed outside the grid, are bound to the outer grid.

5.2 Parallelization using OpenMP

To benchmark the vector optimized code, Algorithm 5.1 was parallelized with OpenMP. OpenMP is in this case a very simple way to parallelize the code for shared memory architectures: before the innermost loop, which covers the whole grid, a pragma directive has to be included in order to tell the compiler to prepare the code for several CPUs. During runtime, each CPU will compute an equally sized part of the loop. Additionally, the three loop version was parallelized in the same manner in order to benchmark the performance increase for the vector-optimized 1 loop version (see Algorithm 5.1). In this case, only the data access Version B could be parallelized, because of the too complex loop structure of the 3 loops C version, which posed too many problems for the compiler.

As target architecture, the Hitachi SR8000 at the Leibniz-Rechenzentrum München was chosen. Although the SR8000 is not a real vector machine, it offers the best possibility to benchmark the parallelized versions of 3D LBM. Within one node with 8 CPUs, the memory remains shared, while across nodes the memory architecture becomes distributed. For this reason, the optimized code was benchmarked for 1 up to 8 CPUs for the shared memory architecture in one node for two problem cases: first, a constant problem of grid size 150^3 with 100 iterations and second a scaling problem with a grid size of 100^3 for 1 CPU, 126^3 for 2 CPUs, 144^3 for 3 CPUs, 159^3 for 4 CPUs, 171^3 for

5 CPUs, 182^3 for 6 CPUs, 191^3 for 7 CPUs, 200^3 for 8 CPUs and 100 iterations. In this case, the resulting time was supposed to remain equal, since the overall number of cells is approximately proportional to the number of CPUs. The results of these benchmarks can be found in Section 6.8. For more information on OpenMP, see [Qui03].

Chapter 6

Results

This chapter will present all benchmark results of all LBM versions introduced in the preceding chapters. The results are ordered according to the data layout and the applied blocking technique: First, all benchmarks for the two grids data layout without any additional cache optimization technique will be presented, subsequently 3-way blocking and 4-way blocking will be applied to this data layout. Afterwards, the grid compressed data layout without blocking, with 3-way and 4-way blocking will be presented. In each subsection, there will be a benchmark result for each CPU architecture tested. For the two Intel Pentium4 machines, always two figures will be presented, one for the GNU gcc compiler and one for the Intel icc compiler.

To make the legend of each picture more readable, some abbreviations were used. As it was first used in Section 3.1, the three different data access schemes are called A for the four array indices, B for the single-index array version with macro access and C for the offset version. Additionally, the differentiation between the "pulling" stream-collide and the "pushing" collide-stream versions is done with the abbreviations **sc** for stream-collide and **cs** for collide-stream, respectively. Since the common data layout is the usage of two grids, only the special case of compressed grid is marked with **GC** for grid compression. The blocking techniques of 3- and 4-way blocking are marked accordingly.

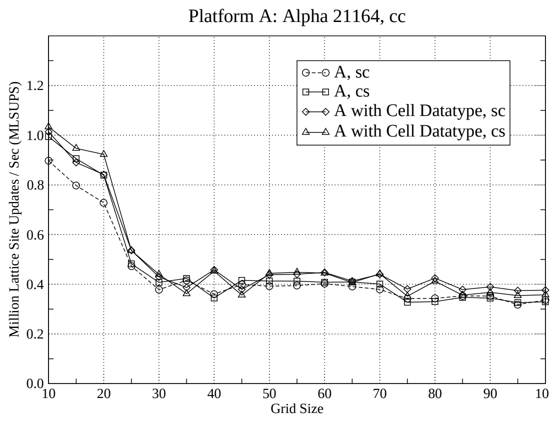
The choice of the block sizes presented another challenge: intuitively, for the optimal block size, the blocks do not have to be cubic. Initial tests were run for both 3-way blocking and 4-way blocking to determine the best possible block size, but unfortunately, the results were rather disappointing in terms of uniqueness. For several combinations of x-, y- and z-side lengths the performance seemed to be maximal, sometimes even for absolutely unexpected combinations. For this reason, constant block sizes of 8^3 for the two grids data layout and 16^3 for the grid compressed data layout were used. The choice of the number of iterations for one block in the case of 4-way blocking was much more easy: since 2 iterations were slower than 4 and 6 were probably too much (the number of iterations had to be even in this implementation), 4 iterations for all versions were chosen. This means, that the block, which is just updated, is updated 4 times, before the next block is loaded to the cache. The technique of 4-way blocking, which requires this iteration number, is explained in detail in Section 4.3.2.

All benchmark results are shown as MLSUPS (mega lattice site updates per second). One cell update consists of 273 floating-point operations, which would result in 273.0 MFLOPS for a CPU with 1.0 MLSUPS.

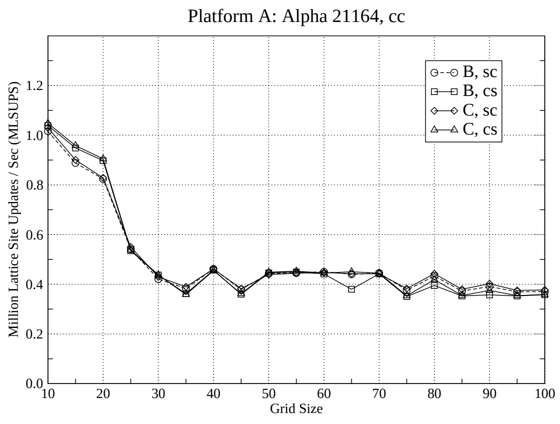
6.1 Two Grids

In this section, the results for the basic data access schemes A, B and C are compared using the common data layout of two grids. For all architectures, the two possible versions of A, as introduced in Section 3.1, and B and C are compared separately to improve the clarity of the pictures. Each version was tested as stream-collide (sc) and collide-stream version (cs).

A common fact for all versions with the data layout of two grids, is that nearly no cache effects can be monitored. This is because of the memory requirement of this data layout (see Figure 4.5): only the smallest problem size fits into the cache of the CPUs. Because of this, the performance drops immediately after the 10^3 problem to a much smaller value and keeps decreasing until the problem size of 30^3, where it reaches a very constant level up to the maximum problem size of 100^3. Another common result is the sometimes significant drop in performance for the problem size of 30^3. This behaviour can be easily explained with the implementation of the two grids: with the additional, surrounding fluid layer, the grids have a total size of 32^3. Since 32 is a multiple of 2, the neighbouring cells to the north, south, top and bottom direction will very likely replace the center cell in the cache. This behaviour is called *cache thrashing*. With the cache optimization technique of *array padding* the cache thrashes can be avoided, but since the problem size of 30^3 is the only grid size, where cache thrashing occurs, padding was not used. For an short overview of the technique of array padding, refer to [KW03]. For a more detailed insight, see [RT00].



(a)



(b)

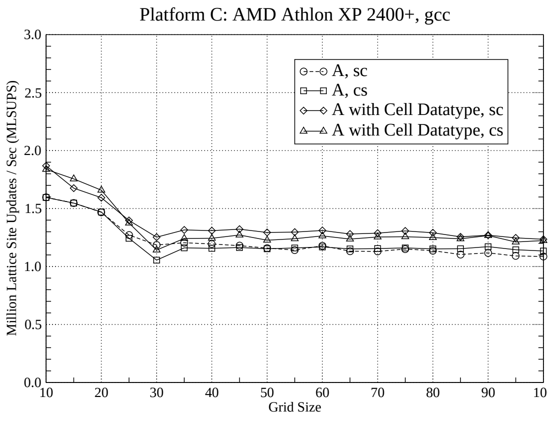
Figure 6.1: Two grids for Alpha 21164.

The A, B and C data access versions show no significant differences on the Alpha 21164 (Figure 6.1). Only the introduction of the cell data type helps A to catch up with B and C in terms of performance. Obviously, the additional *typedef* command in the C language helps the compiler to produce faster code: only a three dimensional cell data structure remains visible instead of a four dimensional double data structure. The performance for the maximum problem for all data access versions reaches nearly 0.4 MLSUPS.

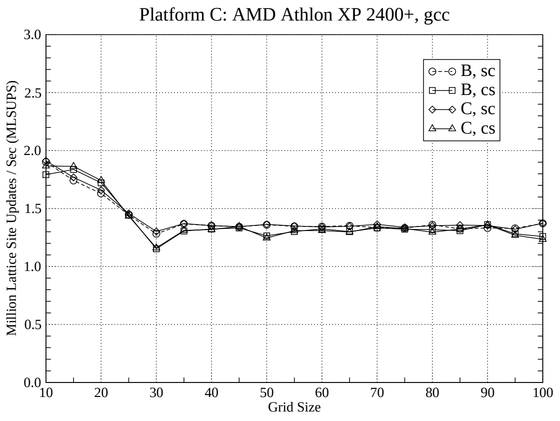
On the AMD Athlon architecture (Figure 6.2), the same relative performance increase for the cell data type in version A can be monitored, but even this improvement does not raise version A to the performance level of B and C. As in the case of nearly all basic versions, stream-collide performs slightly better than collide-stream. In comparison to the single channel Pentium4, the Athlon shows a rather disappointing performance: the P4 is 30 to 40% faster than the Athlon, although the two memory architectures can be directly compared. The Pentium4 uses the Intel i845 chip set, the Athlon the VIA KT333 chip set, both working with Single Channel DDR333 memory.

For the Pentium 4 (Figure 6.3), the basic versions show no significant differences. A, B and C are very similar in terms of performance. Differences exist between the stream-collide and the collide-stream versions. Stream-Collide always seems to perform better. For A, the definition of the cell data type seems to clearly improve the performance to the level of B and C. Also, it does not make much differences, whether the Gnu gcc or the Intel icc compiler is used.

A much better result in terms of performance delivers the Pentium 4 with the faster memory architecture (Figure 6.4). About 2.5 MLSUPS for the basic B and C versions makes this P4 archi-

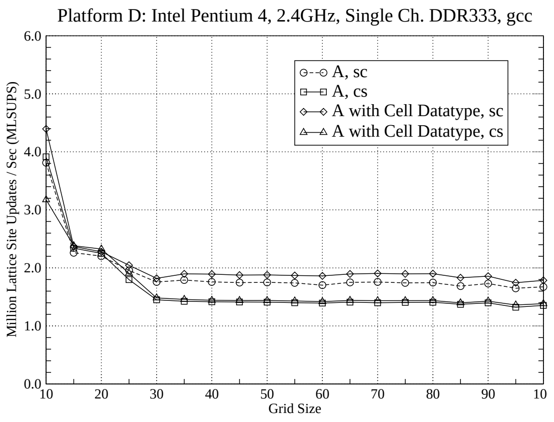


(a)

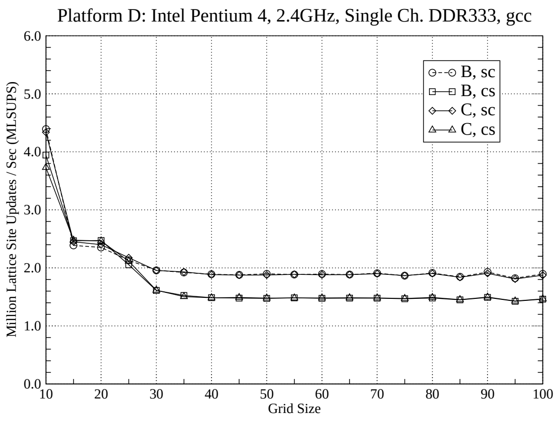


(b)

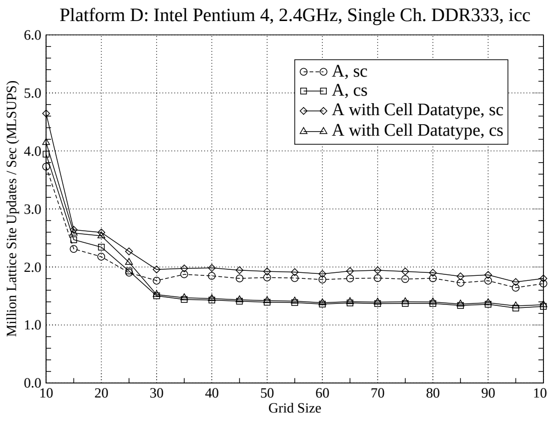
Figure 6.2: Two grids for AMD Athlon XP.



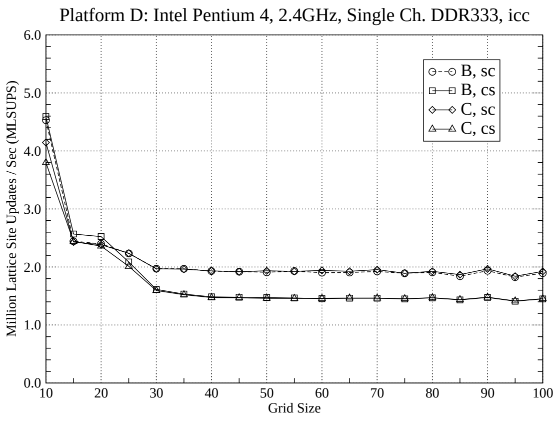
(a)



(b)

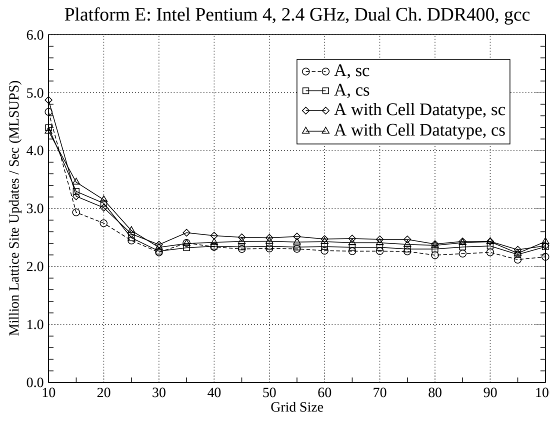


(c)

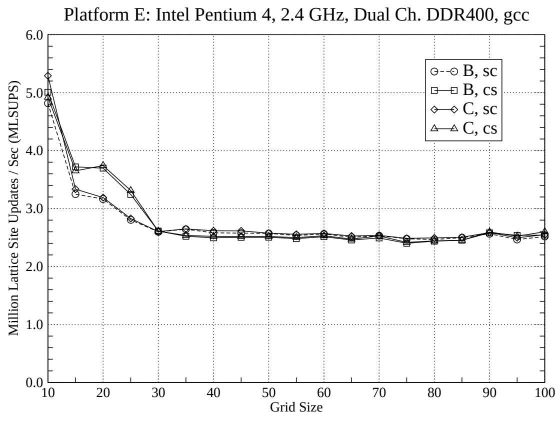


(d)

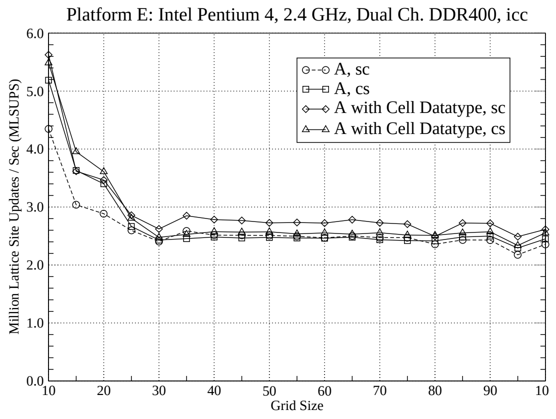
Figure 6.3: Two grids for Intel P4, Single Channel DDR333.



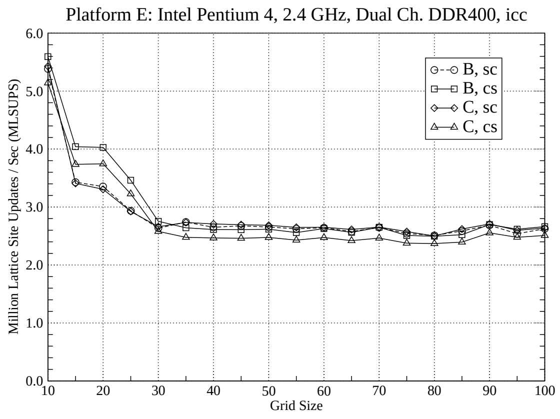
(a)



(b)



(c)

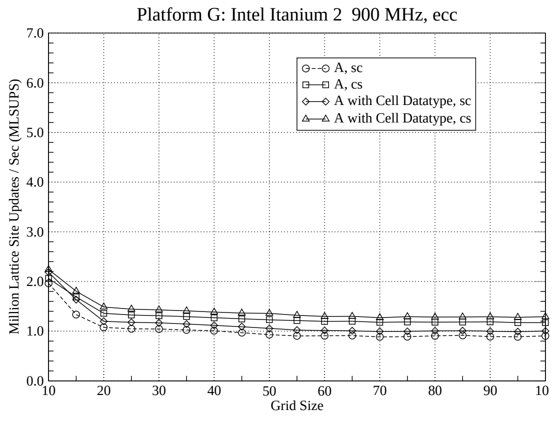


(d)

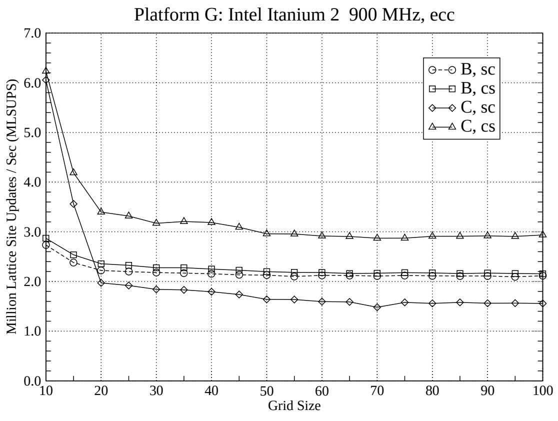
Figure 6.4: Two grids for Intel P4, Dual Channel DDR400.

tecture about 30% faster than the single channel architecture. This performance increase can be easily explained by i865 chip set used with this Pentium4 CPU: In comparison to the older, i845 chip set, the Springdale (i865) offers a front side bus (FSB) of 800MHz (200MHz·4) instead of only 533MHz (133·4). This raises the system bandwidth to 6.4GByte/s (800MHz·8Byte). In combination with dual channel DDR400 main memory, which virtually combines two RAM modules to one and therefore can provide up to 6.4GByte/s (400MHz·2·8Byte), no memory bottleneck as in the older chip set exists (4.2GByte/s System Bandwidth to 2.6GByte/s Memory Bandwidth). This advantage directly results in the 30% faster performance, since the CPU core remains the same. In case of version A, only the stream-collide version with the cell data type can catch up with B and C. Again, stream-collide outperforms collide-stream for each basic data access pattern. In comparison between the Intel icc and the GNU gcc compiler, both reach nearly identical performance levels.

The Intel Itanium2 shows one of the most interesting results (Figure 6.5). While A is outperformed by B and C, the fastest version is always the one based on the collide-stream update step. The biggest surprise is the performance of the C collide-stream version: nearly 3.0 MLSUPS with no blocking techniques applied is an impressive result for a basic version. Also very interesting is the declension of version C stream-collide, which is only half as fast as the collide-stream version. The Itanium2 seems to like indirect writing operations, which are contained in the collide-stream versions, much better than indirect reading operations, which are performed in the stream-collide versions. Probably, the collide-stream order is much better supported by the EPIC-architecture



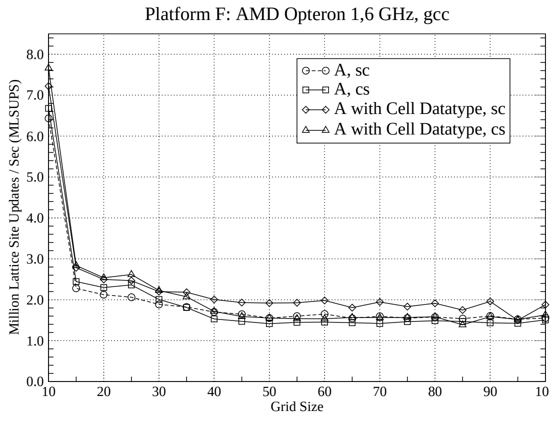
(a)



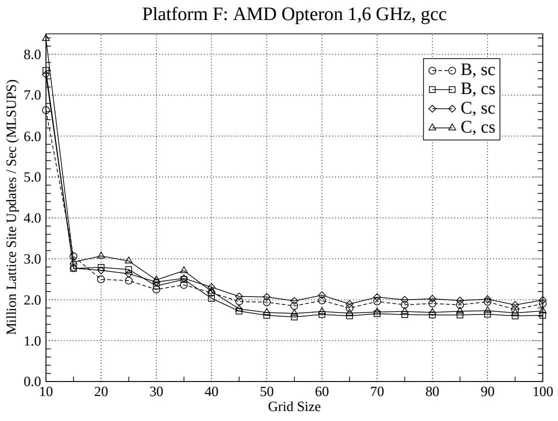
(b)

Figure 6.5: Two grids for Intel Itanium2.

(Explicit Parallel Instruction Computing) of the Intel Itanium2. This behaviour clearly shows the strong dependency of the Itanium2 on the compiler, which has to create code optimized for the EPIC-architecture.



(a)



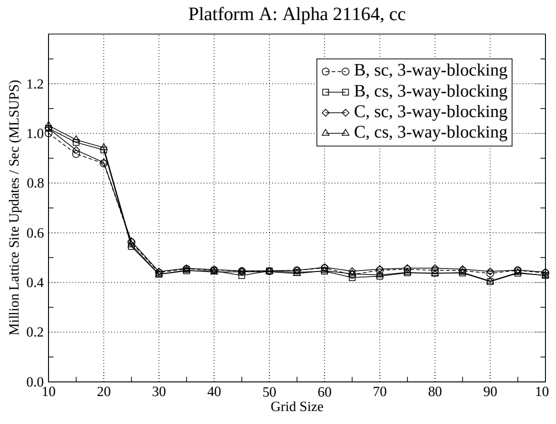
(b)

Figure 6.6: Two grids for AMD Opteron.

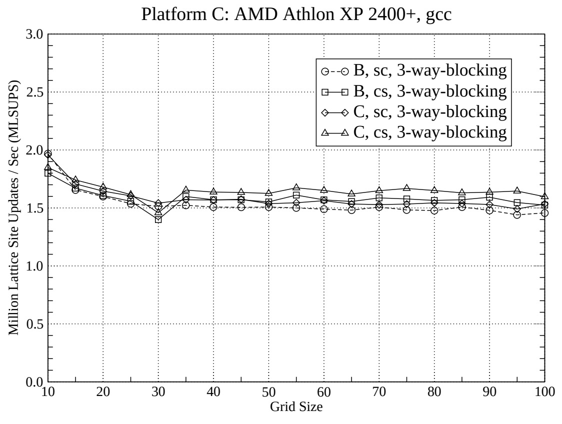
The AMD Opteron is AMD's newest CPU creation. This 64bit-CPU keeps the DDR333 memory controller on chip and therefore improves the memory access latencies.
For the basic versions, the AMD Opteron shows an interesting slower decrease of performance in the beginning (Figure 6.6): it reaches the constant level of performance not until the problem size of 45^3. This is probably the result of the large caches of the AMD Opteron (128kByte L1 Cache and 1024 kByte L2 cache). As with the Intel Pentium4, stream-collide seems to be better for the AMD Opteron. Again, the cell data type improves the performance of A to the level of B and C, but overall, the AMD Opteron cannot reach the performance results of the newest Intel machines, at least for those code versions, where no blocking techniques have been introduced.

6.2 3-way Blocking in the Two-Grid Data Layout

In this section, the common data layout of two grids is combined with the 3-way blocking technique. Since data access A proved to be the slowest, 3-way blocking was only applied to the data access versions B and C. These two versions are both benchmarked as stream-collide (sc) and collide-stream (cs) versions.



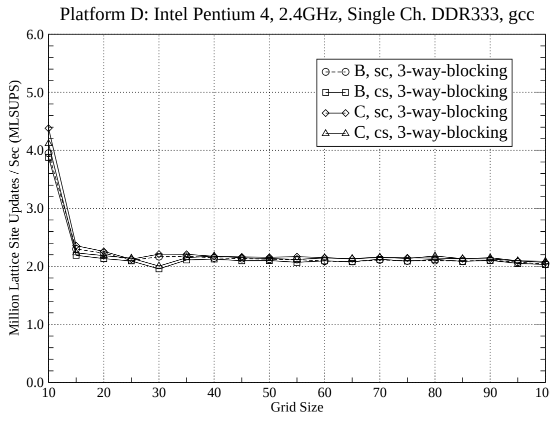
(a)



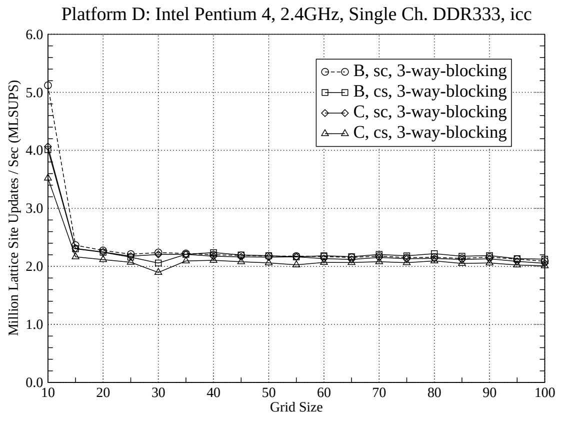
(b)

Figure 6.7: Two grids with 3-way blocking for Alpha 21164 and AMD Athlon.

The Alpha 21164 benefits only marginally from the 3-way blocking (Figure 6.7(a)): the performance for the maximum grid size climbs above 0.4 MLSUPS. The differences between the B and C versions are negligible. For bigger grid sizes, the Athlon architecture shows performance increases primarily for the collide-stream versions of about 30% (Figure 6.7(b)). The stream-collide version does not benefit as clearly from 3-way blocking as the collide-stream versions. For small grid sizes, nearly no improvement can be observed.



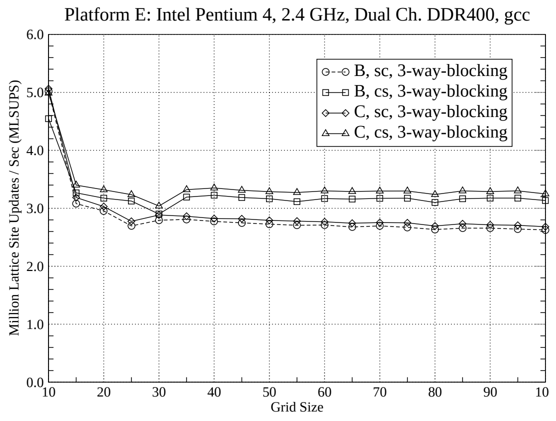
(a)



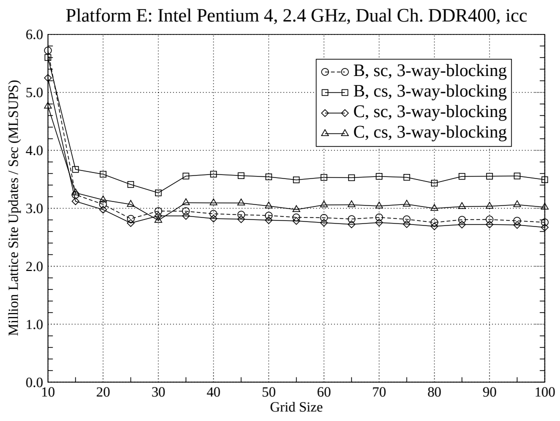
(b)

Figure 6.8: Two grids with 3-way blocking for Intel P4, Single Channel DDR333.

For the single channel P4 (Figure 6.8), all versions show nearly the same result: slightly above 2 MLSUPS with gcc and Intel compiler. Stream-Collide seems to have a very small advantage to the collide-stream versions, while in the case of no blocking technique stream-collide clearly performed better (see Figure 6.3(b)+(d)).



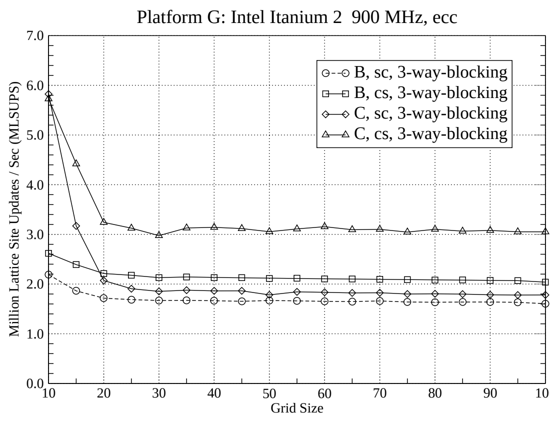
(a)



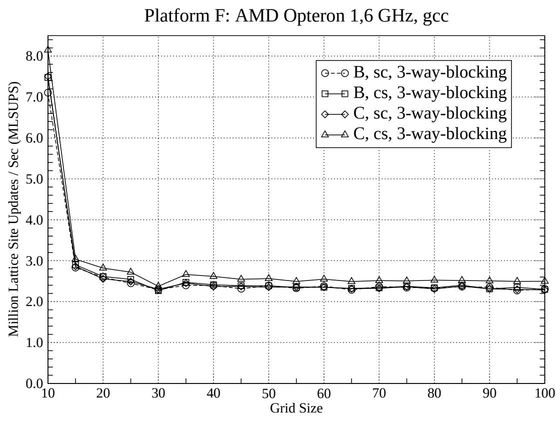
(b)

Figure 6.9: Two grids with 3-way blocking for Intel P4, Dual Channel DDR400.

For the dual channel Pentium4 (Figure 6.9), the results are much more distinct: obviously, collide-stream versions dominate the stream-collide versions, but the GNU gcc compiler produces better results with the C version, the Intel compiler with the B version. The Intel compiler also produces the fastest version of two-grid 3-way blocked code with 3.5 MLSUPS.



(a)



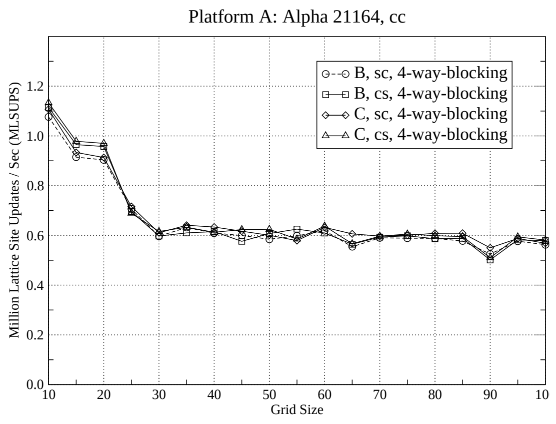
(b)

Figure 6.10: Two grids with 3-way blocking for Intel Itanium2 and AMD Opteron.

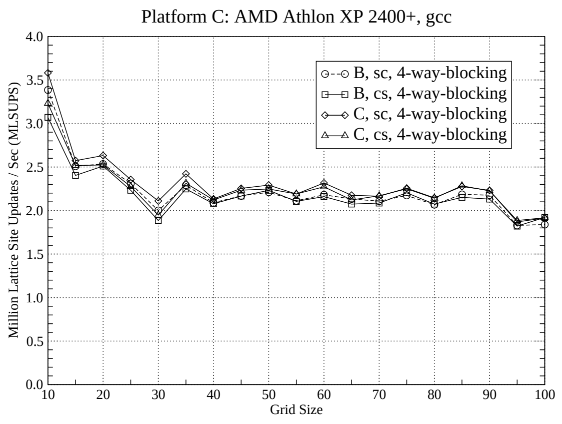
The performance for the Intel Itanium2 is hardly increased by 3-way blocking (Figure 6.10(a)). Again, the obvious dominance of the C collide-stream version can be observed. For the AMD Opteron (Figure 6.10(b)), 3-way blocking improves the performance of all implementations of about 25% and, as with the Itanium2, the C collide-stream version dominates all other versions.

6.3 4-way Blocking in the Two-Grid Data Layout

In this section, the 4-way blocking technique is applied to the data layout of two grids. The data access versions B and C are each benchmarked as stream-collide (sc) and collide-stream (cs) versions.



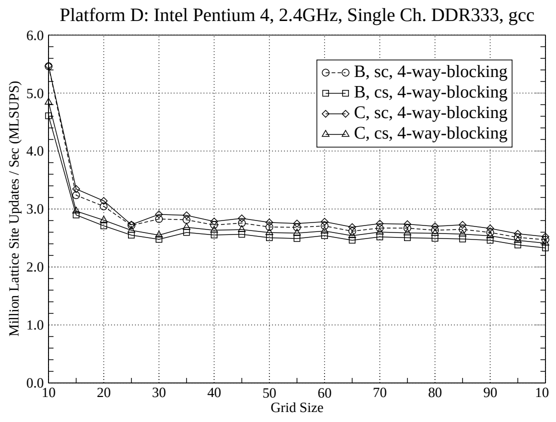
(a)



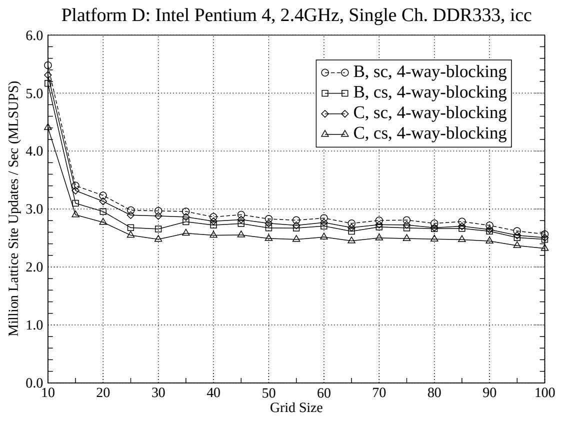
(b)

Figure 6.11: Two grids with 4-way blocking for Alpha 21164 and AMD Athlon.

The Alpha 21164 benefits significantly from the cache optimization technique of 4-way blocking (Figure 6.11(a)): the performance is increased by nearly 50% to nearly 0.6 MLSUPS. Again, the differences between the different versions are negligible, although the collide-stream versions seem to have the advantage against the stream-collide versions for smaller grid sizes. For the AMD Athlon (Figure 6.11(b)), 4-way blocking means a great performance increase: about 40% more performance can be monitored, small grid sizes benefit even more from this cache optimization technique. The differences between the versions are quite small, but the C versions show a slightly better overall performance than the B versions. But again, compared to the corresponding code variants employing the 4-way blocking technique on the Intel Pentium4 with single channel DDR333 memory, the Athlon shows disappointing results.



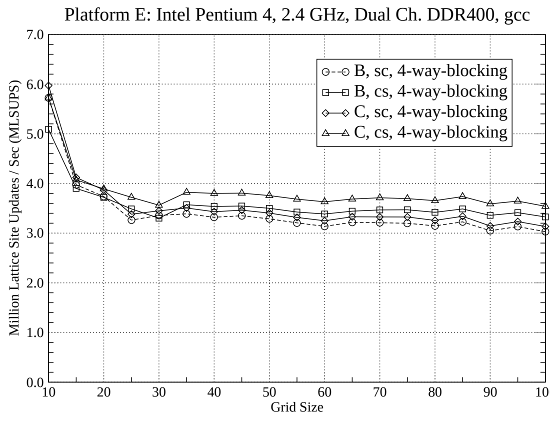
(a)



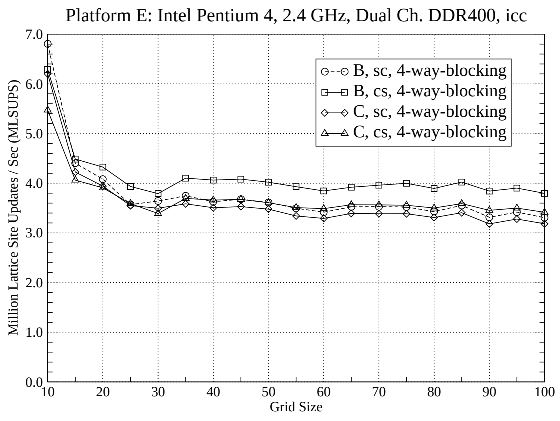
(b)

Figure 6.12: Two grids with 4-way blocking for Intel P4, Single Channel DDR333.

For the single channel Pentium4 (Figure 6.12), 4-way blocking considerably increases the performance: 2.5 MLSUPS for the 100^3 problem are possible with both compilers and a stream-collide version. The differences between the versions are marginal.



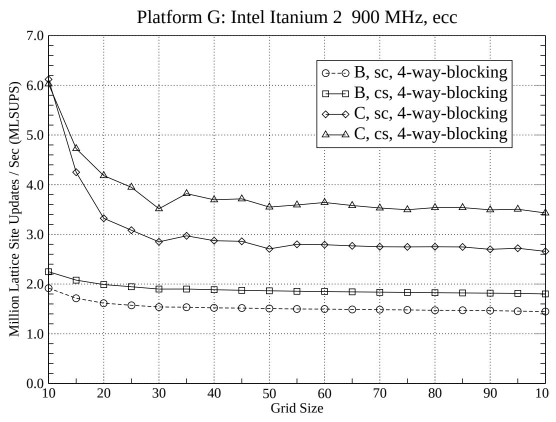
(a)



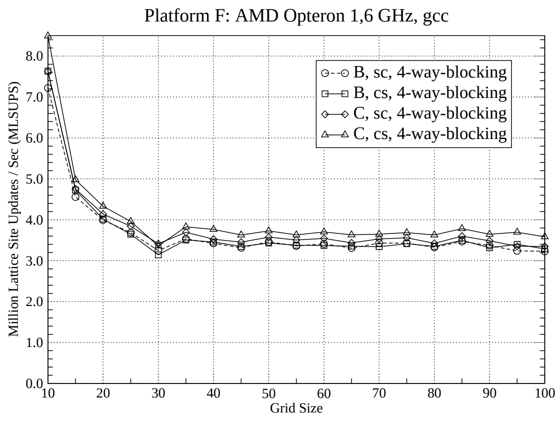
(b)

Figure 6.13: Two grids with 4-way blocking for Intel P4, Dual Channel DDR400.

For the faster dual channel Pentium4 (Figure 6.13), 4-way blocking is able to increase performance to nearly 4.0 MLSUPS. Thereby, the Intel compiler produces slightly faster code. Again, with the faster memory architecture, collide-stream versions seem to perform better than the stream-collide versions.



(a)



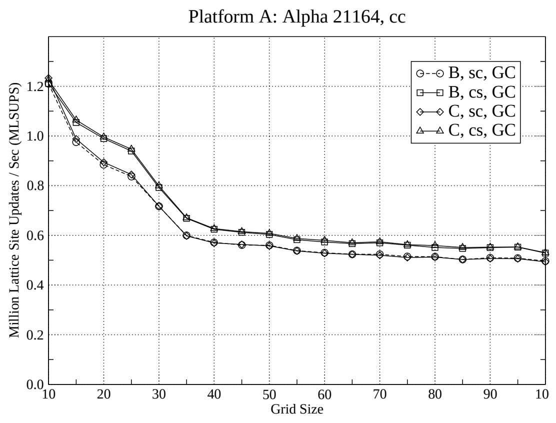
(b)

Figure 6.14: Two grids with 4-way blocking for Intel Itanium2 and AMD Opteron.

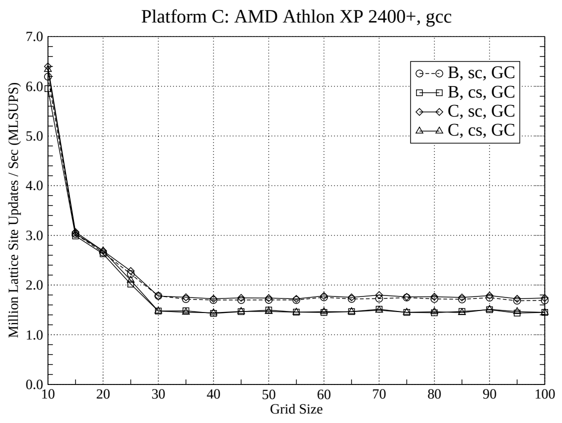
The Intel Itanium2 obviously prefers the C version (Figure 6.14(a)): with 4-way blocking it nearly reaches 3.5 MLSUPS. Again, collide-stream performs much better than the stream-collide version. Version B stays below 2 MLSUPS, but here also collide-stream dominates stream-collide. For the AMD Opteron (Figure 6.14 (b)), 4-way blocking seems to be an absolutely great idea: from 2.5 MLSUPS with 3-way blocking to 3.5 MLSUPS with 4-way blocking, the performance is increased by 40%. Also with the Opteron, collide-stream seems to be the better choice.

6.4 Grid Compression

In this section, only the two data access schemes B and C are used with the data layout of grid compressed, since the data access scheme A barely performed as good as the other two in the two-grid data layout. Each scheme was tested as stream-collide (sc) and collide-stream (cs) version. Common for all grid compressed versions is, that in contrast to the data layout of two grids, the constant performance for large grid sizes is reached at a larger grid size. This is because of the effective reduction of memory requirement (see Figure 4.8), which means, that even grids larger than 10^3 lattice sites can fit into the cache. Grid compression therefore improves the data locality by construction.



(a)



(b)

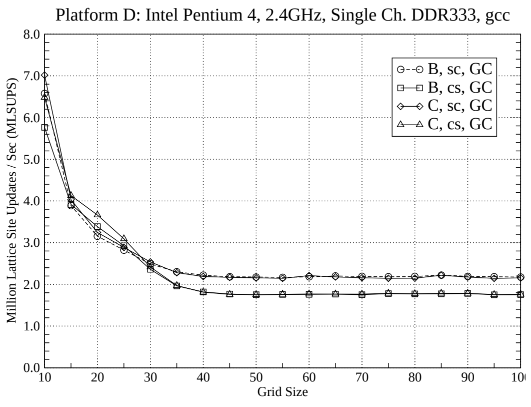
Figure 6.15: GC for Alpha 21164 and AMD Athlon.

Grid compression is a very efficient cache optimization technique for the Alpha 21164 (Figure 6.15(a)). In comparison with the basic two grid versions, the performance is immediately increased by 25%. Also, the constant performance level is reached not until the problem size of 60^3, because of the efficient reduction of memory requirement. With GC, collide-stream versions are faster than stream-collide versions. Also for the AMD Athlon (Figure 6.15(b)), grid compression offers an obvious performance increase, especially for small grid sizes. Stream-collide clearly dominates the collide-stream versions.

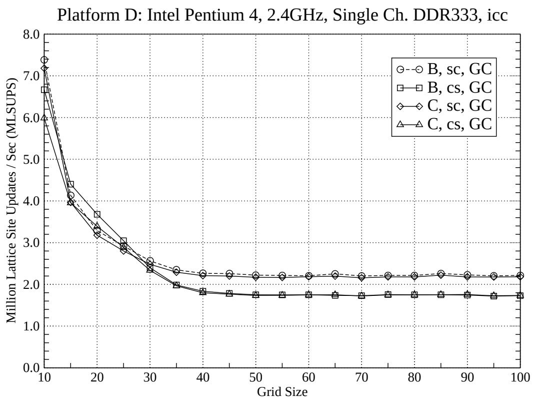
In contrast to two grids, grid compression definitely offers a better performance for the single channel Pentium4 (Figure 6.16): From below 2.0 MLSUPS to 2.2 MLSUPS with both compilers means a 10% increase in performance. Again, stream-collide versions are ahead of the collide-stream versions.

For the dual channel Intel Pentium4 (Figure 6.17), grid compression means a performance increase of about 15-20%: from about 2.6 MLSUPS to slightly above 3.0 MLSUPS, where the Intel compiler produces the slightly better result. Again, collide-stream versions perform better on the faster memory architecture.

For the Intel Itanium2 (Figure 6.18(a)), grid compressed improves the performance of version C, where again the collide-stream version performs best, but decreases the performance of B. However, the biggest performance increase takes place for small grid sizes, larger grids are only slightly faster computed. The AMD Opteron also shows only significant performance improvements for small grid sizes (Figure 6.18(b)). Like in the case of the two-grid data layout, without blocking the collide versions perform best.

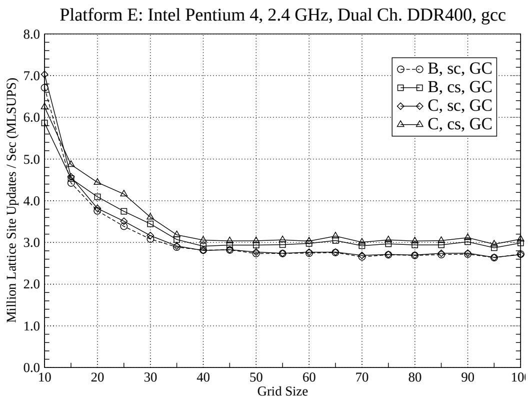


(a)

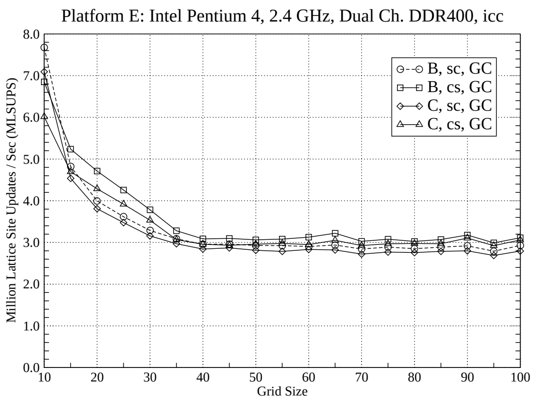


(b)

Figure 6.16: GC for Intel P4, Single Channel DDR333.

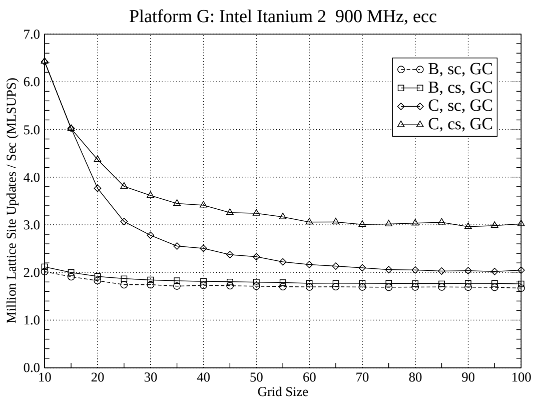


(a)

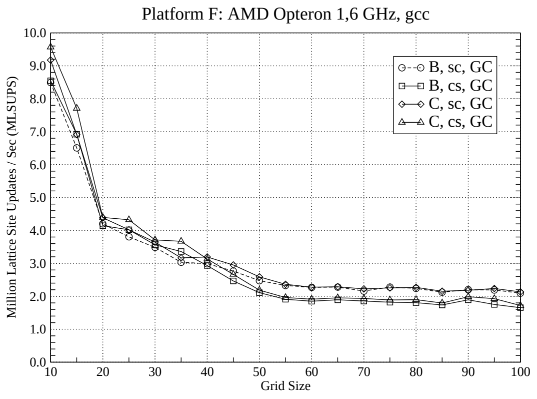


(b)

Figure 6.17: GC for Intel P4, Dual Channel DDR400.



(a)

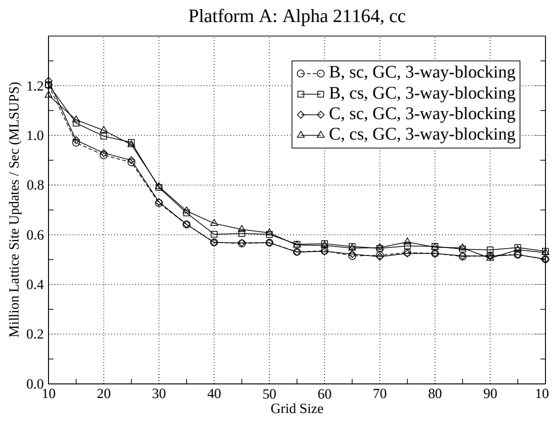


(b)

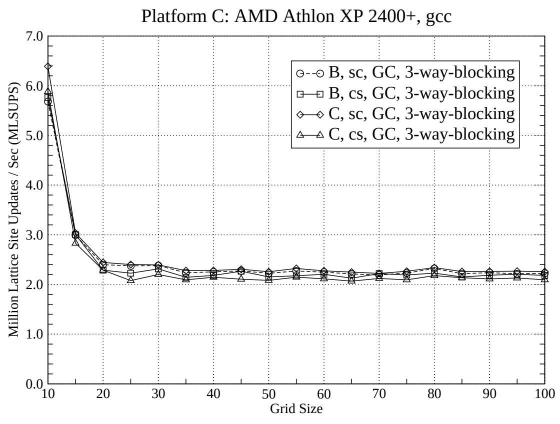
Figure 6.18: GC for Intel Itanium2 and AMD Opteron.

6.5 3-way Blocking in the Grid Compressed Data Layout

In this section, the technique of 3-way blocking was applied to the data layout of grid compressed. For each of the two data access schemes B and C a measurement for stream-collide (sc) and collide-stream (cs) was taken.



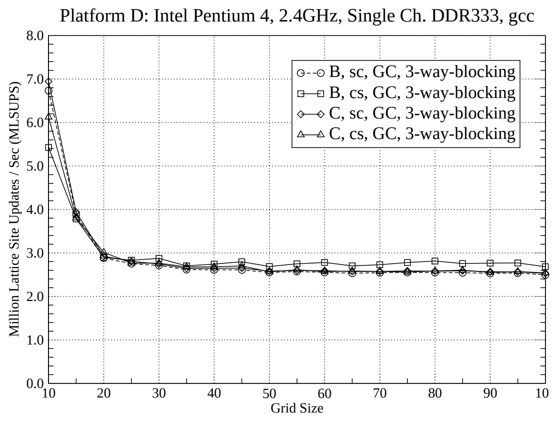
(a)



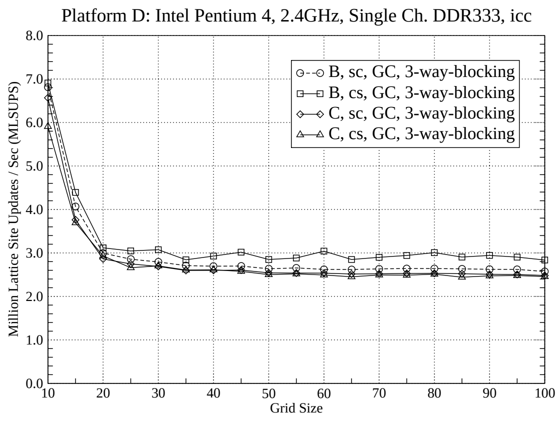
(b)

Figure 6.19: GC with 3-way blocking for Alpha 21164 and AMD Athlon.

3-way blocking with grid compression seems to have no performance effect for the Alpha 21164 in comparison to the non-blocked grid-compressed versions (Figure 6.19(a)). But again, the collide-stream versions are faster. In contrast, the AMD Athlon profits greatly from the cache optimization technique of 3-way blocking (Figure 6.19(b)). All versions reach a stable performance of about 2.2 MLSUPS for bigger grid sizes. Additionally, the performance difference between the versions is negligible.



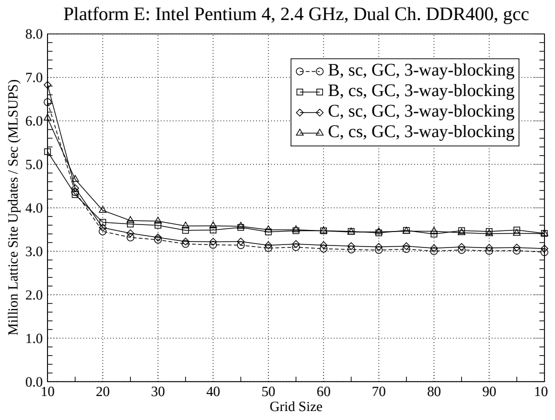
(a)



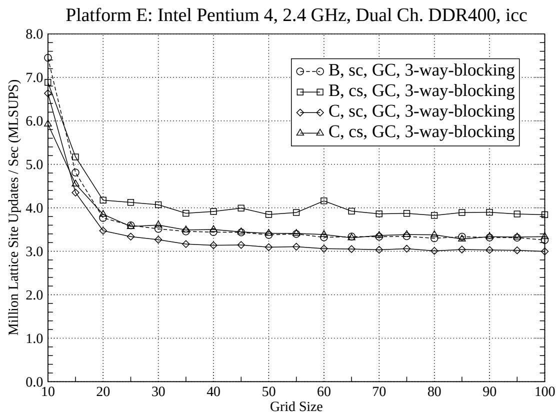
(b)

Figure 6.20: GC with 3-way blocking for Intel P4, Single Channel DDR33.

For the slower P4 memory architecture (Figure 6.20), 3-way blocking in combination with grid compressed seems to be a great improvement. Especially with the Intel compiler, nearly 3.0 ML-SUPS are reached. And suddenly, version B with collide-stream outperforms the other versions on both compilers, where without blocking, only the Intel compiler showed this version as the fastest.



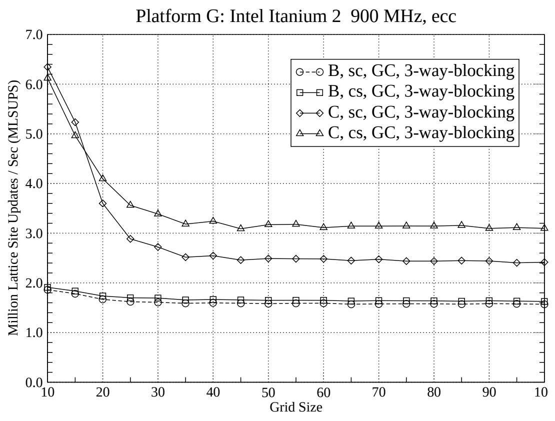
(a)



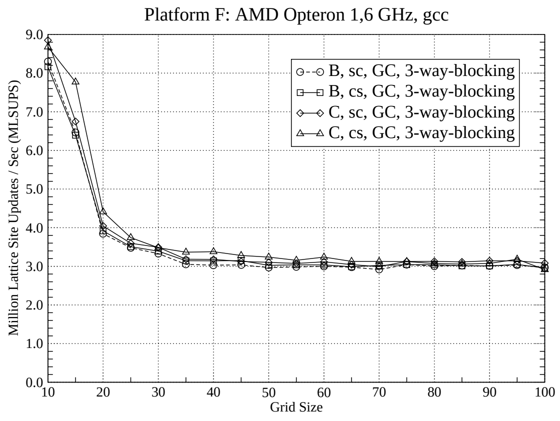
(b)

Figure 6.21: GC with 3-way blocking for Intel P4, Dual Channel DDR400.

The faster Pentium4 memory architecture also shows great performance increase of about 25% on the Intel compiler (Figure 6.21). Here also the B collide-stream version performs best.



(a)



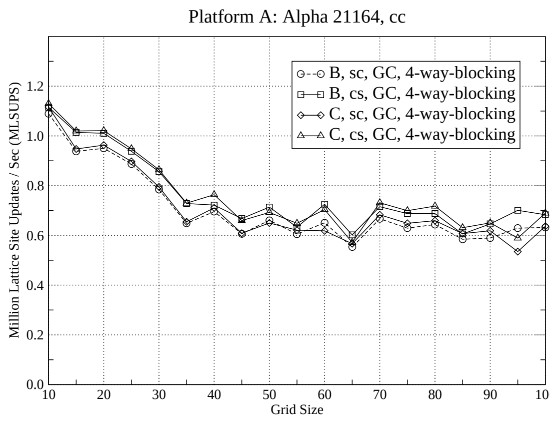
(b)

Figure 6.22: GC with 3-way blocking for Intel Itanium2 and AMD Opteron.

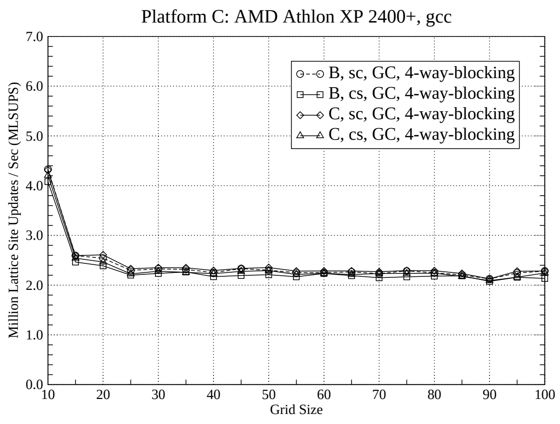
For the Intel Itanium2 (Figure 6.22(a)), 3-way blocking with grid compressed means only a slight increase for the fastest version C with collide-stream, a litter better improvement for the second best C with stream-collide and a performance decrease for the B versions. The AMD Opteron (Figure 6.22(b)), however, performs differently: nearly 50% performance increase with 3-way blocking can be monitored. But for this architecture, all version perform nearly the same.

6.6 4-way Blocking in the Grid Compressed Data Layout

In this section, the cache optimization technique of 4-way blocking was applied to the data layout of grid compressed. For both data access schemes B and C, stream-collide (sc) and collide-stream (cs) was benchmarked.



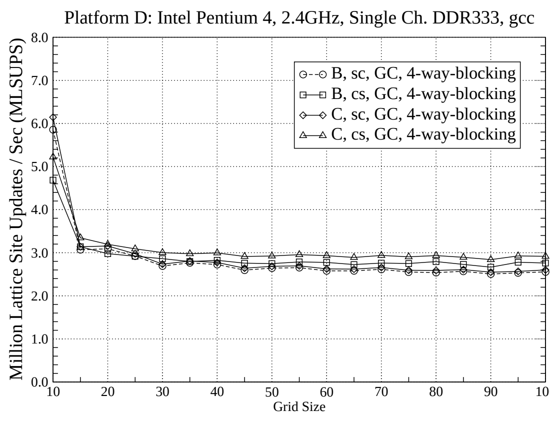
(a)



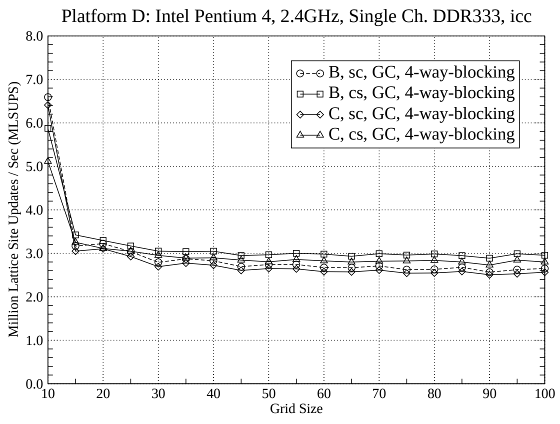
(b)

Figure 6.23: GC with 4-way blocking for Alpha 21164 and AMD Athlon.

As with the two grids data layout, 4-way blocking offers a great performance increase for the Alpha 21164 (Figure 6.23(a)). Although no stable performance level is reached, the maximum problem of 100^3 shows the highest performance for the Alpha and improves the performance in comparison to 3-way blocking with GC by about 40%. For the AMD Athlon (Figure 6.23(b)), 4-way blocking offers nearly no performance increase, but a significant performance drop for small grid sizes. As in the case of 3-way blocking, the performance difference between the versions is negligible.



(a)



(b)

Figure 6.24: GC with 4-way blocking for Intel P4, Single Channel DDR333.

The slower P4 memory architecture shows no impressive performance increase for the step from 3-way blocking to 4-way blocking (Figure 6.24). For both compilers, the performance gets nearer to 3.0 MLSUPS and again, the differences between the versions are small.

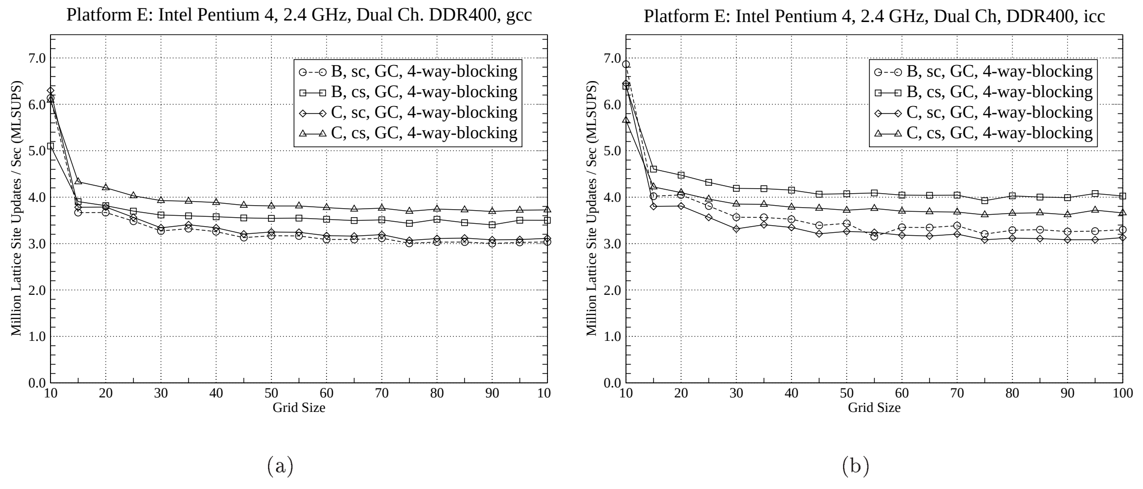


Figure 6.25: GC with 4-way blocking for Intel P4, Dual Channel DDR400.

For the fast Intel Pentium4 memory architecture (Figure 6.25), the performance increase for 4-way blocking is also very small. As for 3-way blocking, the gcc compiler favors the B collide-stream version and the Intel compiler likes the C collide-stream version best.

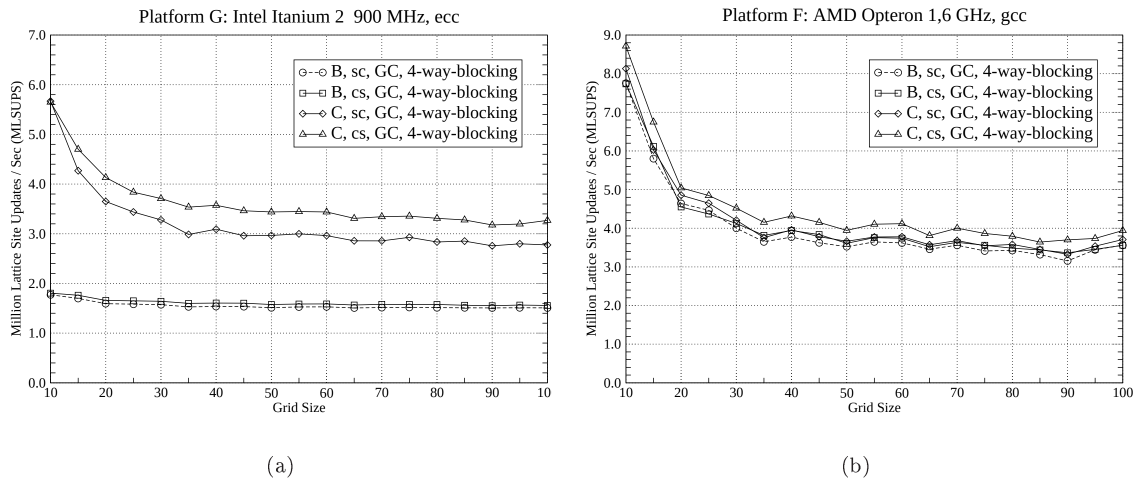


Figure 6.26: GC with 4-way blocking for Intel Itanium2 and AMD Opteron.

The performance of the Intel Itanium2 is slightly improved for the C versions in comparison to the grid compressed 3-way blocked version (Figure 6.26(a)). The B versions stay on the low performance level, obviously, the Itanium2 has problems with this data access. For the AMD Opteron, the performance is greatly improved from slightly above 3.0 MLSUPS to nearly 4.0 MLSUPS, which means nearly 30% performance improvement. Now also the C collide-stream version works fastest.

6.7 Profiling results for 4-way Blocking

To show the success of the cache optimization for the LBM in 3D, some profiling measurements on Platform A (Alpha 21164) and on Platform B (AMD Athlon) were made. On Platform A, the best basic version and the fastest version with 4-way blocking and the grid compressed data layout were profiled with DCPI and a constant grid size of 100^3. In this case, L1 Data Cache Misses (dmisses), L2 Data Cache Misses (smisses), L3 Data Cache Misses (bmisses), TLB misses (dtbmisses) and Instruction Cache Misses (imisses) were measured. On Platform B, the 4-way blocked B and C versions with both data layouts were benchmarked with the profiling tool PAPI. Here, level 1 data cache misses (L1_DCM), level 2 data cache misses (L2_DCM) and TLB data misses (TLB_DM) were measured to give an impression of the efficiency of 4-way blocking. For more detailed information on DCPI and PAPI see also [ABD+97] and [BDG+00].

6.7.1 Profiling with DCPI

```
Where have all the cycles gone?

 I-cache (not ITB)   0.1% to  5.4%
  ITB/I-cache miss   0.0% to  0.0%
      D-cache miss  31.8% to 32.0%
          DTB miss   0.0% to  1.6%
      Write buffer  22.3% to 28.9%
   Synchronization   0.0% to  0.0%

 Branch mispredict   0.0% to  0.0%
         IMUL busy   0.0% to  0.0%
         FDIV busy   0.0% to  0.0%
             Other   0.0% to  0.0%

 Unexplained stall   0.2% to  0.2%
  Unexplained gain  -1.0% to -1.0%
----------------------------------------
  Subtotal dynamic                 60.3%

          Slotting      4.8%
     Ra dependency     14.0%
     Rb dependency      4.0%
     Rc dependency      0.0%
     FU dependency      0.0%
----------------------------------------
   Subtotal static                 22.8%
----------------------------------------
       Total stall                 83.1%

            Useful     14.8%
              Nops      1.8%
----------------------------------------
   Total execution                 16.6%

Net sampling error                  0.3%
----------------------------------------
     Total tallied                100.0%
(227946, 62.5% of all samples)
```

```
Where have all the cycles gone?

 I-cache (not ITB)   0.3% to  1.8%
  ITB/I-cache miss   0.0% to  0.0%
      D-cache miss  37.4% to 37.8%
          DTB miss   0.1% to  3.0%
      Write buffer   3.6% to  6.0%
   Synchronization   0.0% to  0.0%

 Branch mispredict   0.0% to  0.7%
         IMUL busy   0.0% to  0.0%
         FDIV busy   0.0% to  0.0%
             Other   0.0% to  0.0%

 Unexplained stall   0.5% to  0.5%
  Unexplained gain  -2.0% to -2.0%
----------------------------------------
  Subtotal dynamic                 43.4%

          Slotting      5.4%
     Ra dependency     19.9%
     Rb dependency      5.4%
     Rc dependency      0.0%
     FU dependency      0.0%
----------------------------------------
   Subtotal static                 30.7%
----------------------------------------
       Total stall                 74.1%

            Useful     22.2%
              Nops      3.2%
----------------------------------------
   Total execution                 25.4%

Net sampling error                  0.5%
----------------------------------------
     Total tallied                100.0%
(168755, 90.1% of all samples)
```

Figure 6.27: DCPI "Where have all the cycles gone?" for both the basic version C, cs (left) and C, cs, GC, 4-way blocking (right).

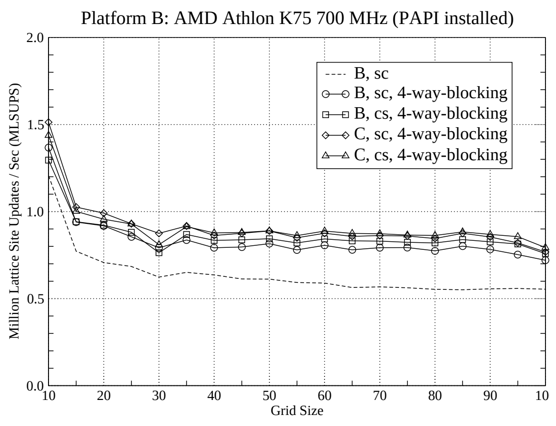
Figure 6.27 shows the output of the DCPI analysis tool "Where have all the cycles gone". The total values have to be interpreted as part of 100%. This means, that these values only represent relative sizes, they do not illustrate total runtimes. In comparison between the basic C version with collide-stream order and the grid-compressed C collide-stream version with 4-way blocking, the percentage of total execution cycles has increased by 9% to about 25%, while the total stall cycles have decreased from 83% to 74%. This values show the optimized cache behaviour of the grid-compressed, 4-way blocked code.

Events	C, cs, no blocking	C, cs, GC	C, cs, 4-way blocking	C, cs, GC, 4-way blocking
L1 DCM (dmiss)	66571	67585	67956	71881
L2 DCM (smiss)	99375	100920	75210	66539
L3 DCM (bmiss)	29705	11188	10715	6295
TLBM (dtbmiss)	357	259	2102	1202
ICM (imiss)	251	314	300	254

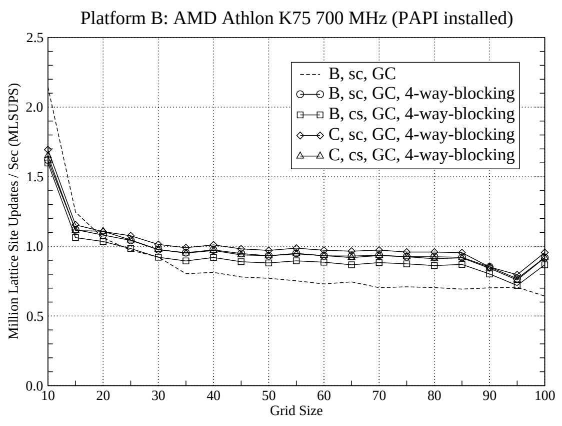
Table 6.1: Number of interrupts of DCPI "dcpiprof" on Platform A (sampling period for corresponding event: 4096).

Another DCPI analysis tool, dcpiprof, is able to extract level 1, level 2 and level 3 data cache misses beside the TLB misses and instruction cache misses. Figure 6.1 shows the number of misses for a sampling period of 4096. This means, that every 4096 misses, an interrupt is created and therefore the total number of misses is about 4096 times the number of interrupts. The comparison between the three fastest versions (basic version, 4-way blocked with the two grids data layout and 4-way blocked with grid compression) shows, that the level 1 data cache misses increase, but both level 2 and level 3 data cache misses decrease. This clearly shows, that the cache optimizations are mainly aimed at larger caches, which are able to hold a whole block. The TLB misses increase, especially for the two grids 4-way blocked version. This behaviour of highly cache optimized code was also monitored for multi-grid methods (see [KRTW02]). The instruction cache misses stay on a very low and constant level.

6.7.2 Profiling with PAPI



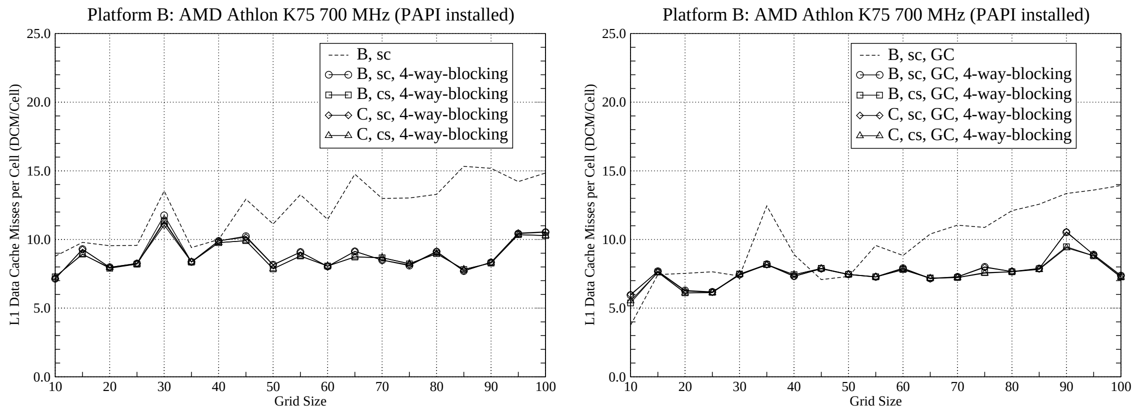
(a)



(b)

Figure 6.28: MLSUPS on AMD Athlon with PAPI installed.

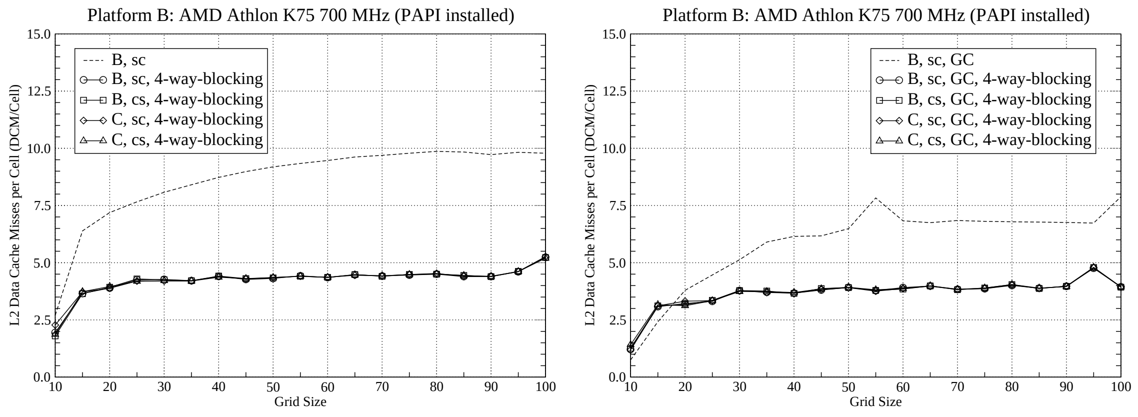
The performance behaviour on the slower AMD Athlon is qualitatively similar to the performance behaviour of the AMD Athlon XP (Figure 6.28): grid compression in combination with 4-way blocking shows an obvious performance increase for all versions and the C versions show a slightly better performance than the B versions.



(a) (b)

Figure 6.29: L1_DCM on AMD Athlon, measured with PAPI.

As the profiling with PAPI shows, the L1_DCM are efficiently reduced with the cache optimization technique of 4-way blocking (Figure 6.29). With the data layout grid compressed, the L1_DCM are even halved compared to the basic B version with stream-collide order. In terms of L1_DCM it makes no difference whether a stream-collide or a collide-stream version is used.

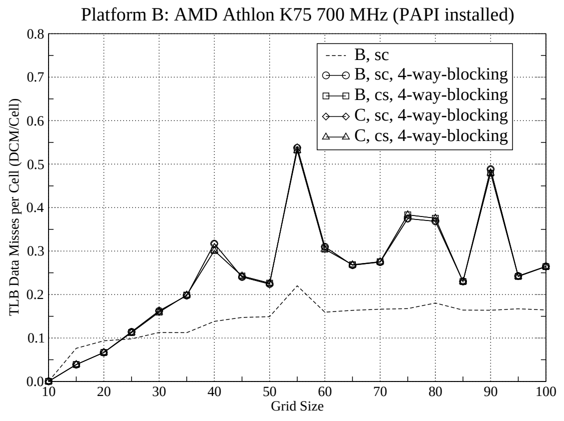


(a) (b)

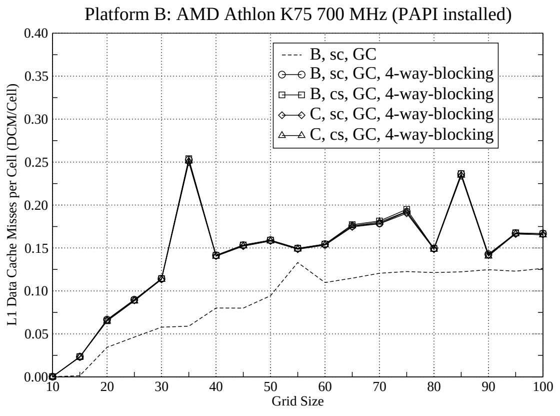
Figure 6.30: L2_DCM on AMD Athlon, measured with PAPI.

The focus on the L2_DCM shows even more clearly the efficiency of the cache optimization techniques (Figure 6.30). Especially with the data layout grid compressed, the L2_DCM are reduced by 60% for large grids. This explains the performance increase of 4-way blocking: the cache content is by far better reused than in the basic versions.

In contrast to the level 1 and level 2 data cache misses, the TLB data misses are increased (Figure 6.31). This result was expected, since in comparison to the unblocked versions, where always successive cache lines have to be loaded into the caches, the cache lines contained in one block are widespread within the main memory. Similar results for multi-grid methods were monitored in [KRTW02].



(a)

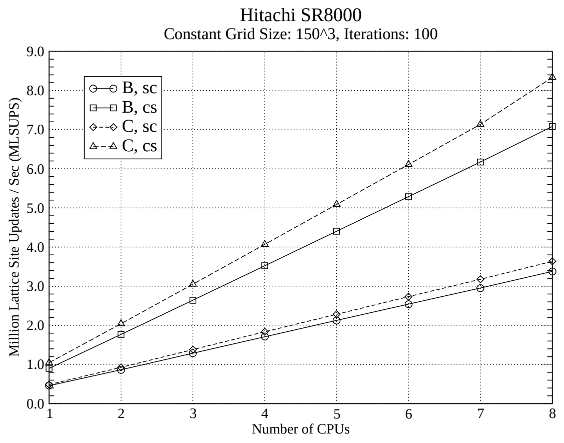


(b)

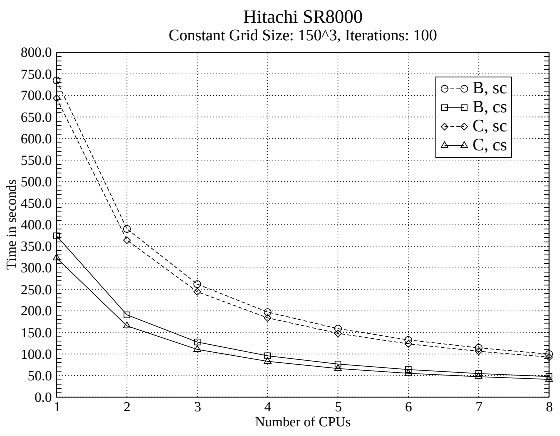
Figure 6.31: TLB_DM on AMD Athlon, measured with PAPI.

6.8 OpenMP-Parallel Version of LBM

The vector-optimized version of the LBM was benchmarked on the Hitachi SR8000 with two problems: first, a problem of constant grid size 150^3 with 100 iterations and, second, a scaling problem with grid sizes of 100^3 for one CPU, 126^3 for two CPUs, 144^3 for three CPUs, 159^3 for four CPUs, 171^3 for five CPUs, 182^3 for six CPUs, 191^3 for seven CPUs, 200^3 for eight CPUs and 100 iterations, respectively. The results are illustrated in both MLSUPS and time.



(a) MLSUPS for the constant problem

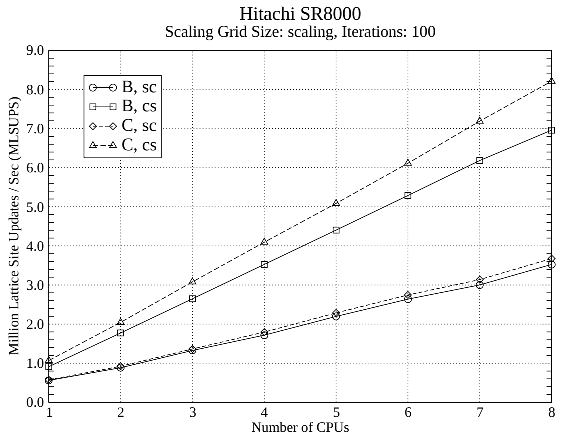


(b) Times for the constant problem

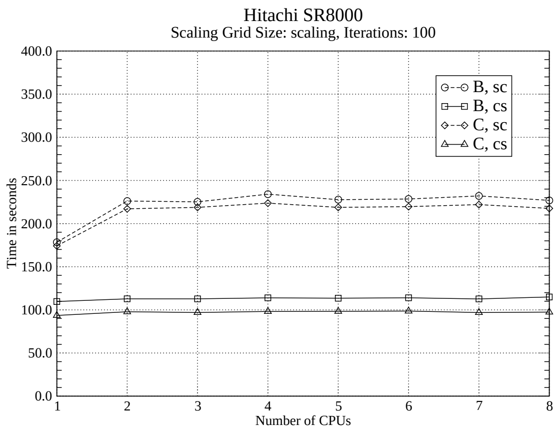
Figure 6.32: MLSUPS and times for the constant problem.

Obviously, the collide-stream versions dominate the stream-collide versions clearly (Figure 6.32). The best overall performance shows the C collide-stream version: 8.33 MLSUPS for 8 CPUs is the highest MLSUPS value reached in this thesis. The nearly perfect linear scalability for all versions can easily be monitored.

For the variable-size problem, the MLSUPS benchmarks result in nearly the exact values as in the constant-size problem (Figure 6.33). In respect of the time, the collide-stream versions show a very constant time level. The stream-collide versions are more than twice as time intensive as the



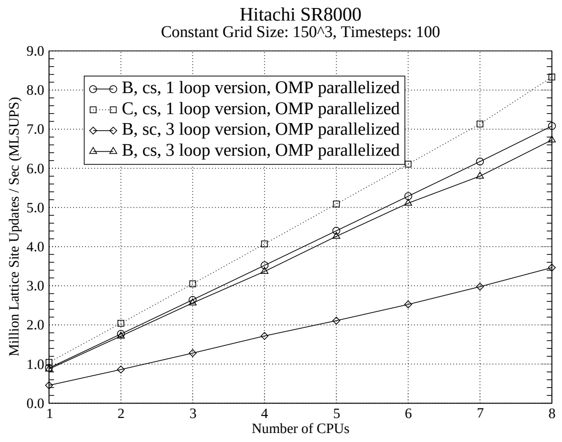
(a) MLSUPS for the scaling problem



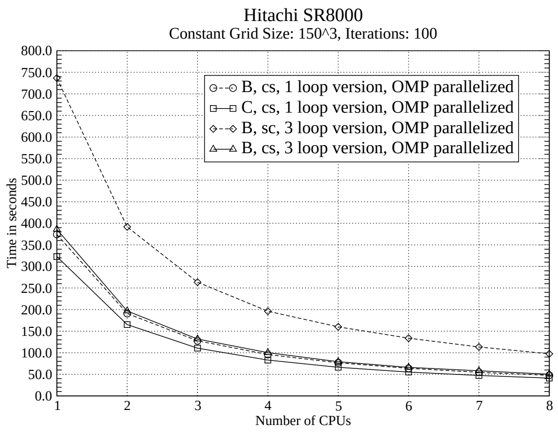
(b) Times for the scaling problem

Figure 6.33: MLSUPS and times for the scaling problem

collide-stream versions and show an obvious increase in time for the step from 1 CPU to 2 CPUs. This behaviour may result from the choice of the grid sizes.



(a) MLSUPS for the 1 loop and 3 loops versions



(b) Times for the 1 loop and 3 loops versions

Figure 6.34: MLSUPS and times for the 1 loop and 3 loops versions.

The comparison between the 1 loop versions and the 3 loops versions show only a slight advantage for the 1 loop versions (Figure 6.34). This may result from the fact, that the SR8000 CPUs are no real vector CPUs, but rather RISC processors. Again, the collide-stream versions dominate the stream-collide versions.

Recapitulating, to increase the performance for the Hitachi SR8000, it is fundamental to use the collide-stream order. Whether the B or C data access is used, the collide-stream order performs more than twice as good as the stream-collide order. For an additional small performance improvement, the C data access should be preferred to the B data access. And since the Hitachi SR8000 is no real vector architecture, the performance levels of the 1-loop version and 3-loops version show only small differences. However, due to compiler problems, only the B 3-loops version could be implemented, which only allows the comparison between the B 1-loop and 3-loops versions.

6.9 Summary of results

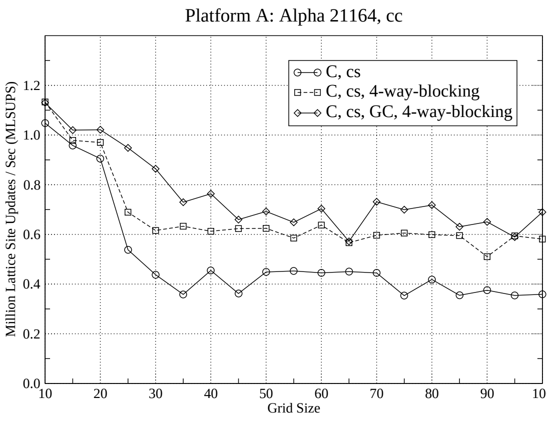
Comparing the different data access schemes, only A clearly falls behind. To decide, whether B or C is faster depends on the according architecture and compiler: for example the Intel icc compiler produces much faster code with the B version for the Pentium4, the GNU gcc compiler favors the C version. All architectures have in common, that in the unoptimized versions stream-collide nearly always performs better, while in the optimized versions, collide-stream wins the performance race.

Although 3-way blocking was not expected to really improve performance, especially for the fast Pentium4 architecture and the AMD Opteron the performance is considerably improved. This performance increase results from the decrease of cache misses, because the data of the entire block in cache can be used several times even without performing several time-steps on one block. However, in nearly all cases, 4-way blocking can again improve performance. For example in the case of the Alpha 21164 or the AMD Opteron, 4-way blocking can significantly increase the MLSUPS value, while in the case of the Pentium4, 3-way blocking results in the biggest performance acceleration. In the direct comparison between the AMD Athlon XP and the Intel Pentium4, the Pentium4 clearly shows the better performance result. Since the memory architecture of both machines is very similar (KT333 chip set for the AMD Athlon XP and i845 chip set for the Intel Pentium4), this result shows, that the Pentium4 has the better floating point unit.

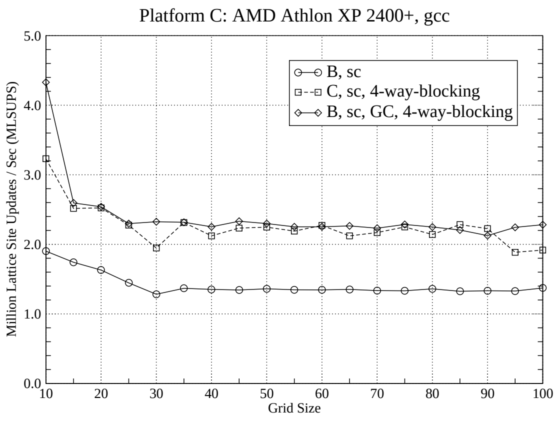
Grid compression is a very efficient way to reduce the memory requirement of the problem and therefore reduce the memory traffic. This advantage becomes obvious for nearly all architectures, except the two 64bit CPUs, the Intel Itanium2 and the AMD Opteron. These two machines do not show as large performance increases as the 32bit architectures, the Itanium2 even shows better results for the two grids versions. In all other cases, grid compression is able to efficiently increase the performance levels, especially for smaller grid sizes.

For a better comparison of the different versions on each architecture, Figure 6.35 shows the fastest basic version, the fastest two-grid 4-way blocked version and the fastest grid-compressed 4-way blocked version. In this comparison it becomes obvious, that with the right choices between the stream-collide and collide-stream cell update order, the data access schemes B or C and 4-way blocking in combination with either the two-grid or the grid-compressed data layout, the performance of the LBM can be increased by up to 100% as in the case of the AMD Opteron.

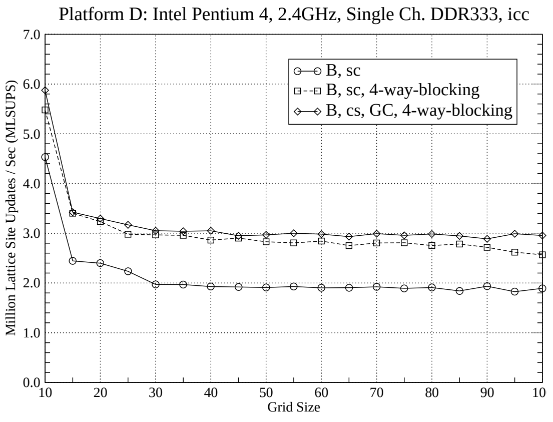
The vector optimized versions offer a very simple way to parallelize the LBM. With only a single pragma-directive, the code can be parallelized for shared memory architectures via OpenMP. For the Hitachi SR8000, the choice of the collide-stream order for the cell update increases the performance by 100%, while the choice between data access schemes B and C and the choice between the 1-loop and 3-loops loop structure is only of minor consequence in terms of performance.



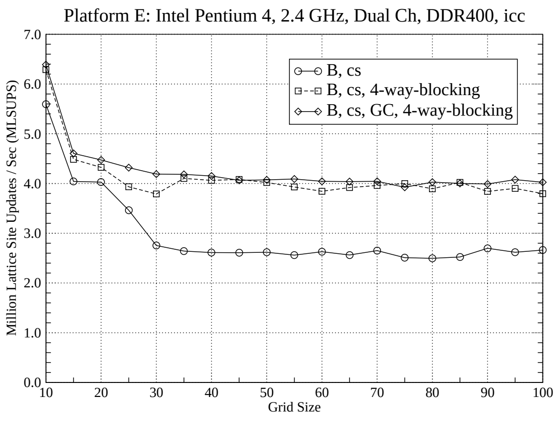
(a)



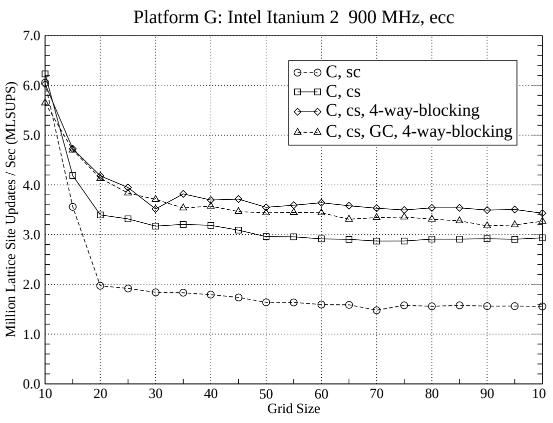
(b)



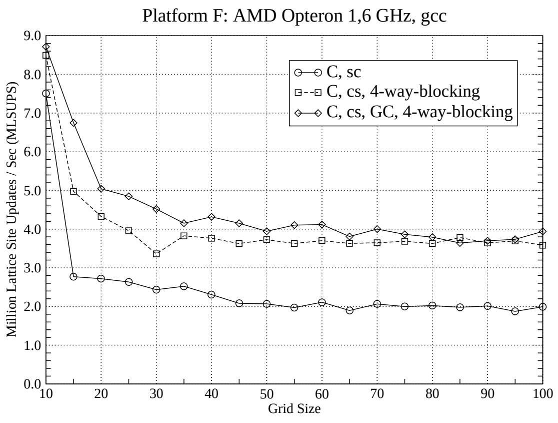
(c)



(d)



(e)



(f)

Figure 6.35: Comparison of the fastest versions on all architectures.

Chapter 7

Conclusion

The Lattice Boltzmann Method, which uses a particle approach based on the Boltzmann Equation to simulate dynamic fluid flows, is a very interesting alternative to the discretisation and the solution of the Navier-Stokes Equations. This is particularly true in terms of cache optimization. In the 2D case as well as in the 3D case, cache optimization techniques like blocking and the data layout of grid compression are able to efficiently increase the performance of the basic LBM code. This thesis examines the applicability of several cache and non-cache optimization techniques in combination with the 3D Lattice Boltzmann Method. Additionally, special attention has been paid to software engineering techniques to combine performance with modularity and reuse of code.

While the optimization techniques of loop unrolling and grid merging showed no performance increase and were therefore discarded, the techniques of *3-way blocking* and *4-way blocking* yielded great performance increases on all platforms. An additional performance increase can be reached on some platforms with the data layout optimization of grid compression. Since this technique efficiently reduces the memory requirement of the problem, it also reduces the memory traffic and accordingly eases the memory bottleneck. In terms of non-cache optimizations, this thesis shows, how the core of the Lattice Boltzmann Method, one cell update, can be improved by simple arithmetic optimizations and different data access schemes.

By the intensive use of macros, this performance optimization can be easily combined with structured, reusable code, which helps to develop different versions of the LBM by simply changing macros. This modular programming style greatly improves the development efforts for faster LBM code.

However, even after these many optimization attempts, some possibilities to improve performance remain: in the case of the blocking techniques, the optimal block sizes can differ from the block sizes used in this thesis. By using the optimal block sizes, the problem can be more efficiently adapted to the memory architecture of the corresponding architecture. Another possibility to create faster LBM code could be to change the principle data layout: instead of keeping the distribution functions of a single cell together, the distribution functions of different cells could be held adjacent in memory. But in this case, other cache optimization techniques would have to be used, because the blocking techniques rely on the data layout used in this thesis. Further, an improved core for the LBM without divisions can be used. This cell update is just a slight change in the physical fundament of the LBM, but the removal of the last remaining division might result in noticeable performance increases. For the Pentium4 architecture, a special approach could be tried: Intel offers special data types for the C language, which exploit the parallelism of the Pentium4 architecture.

In terms of parallelization, the simple OpenMP variant can obviously be improved, since no further optimization efforts were taken. One possibility would be the parallelization with MPI, which offers precise MPI calls for a better control of the parallel code.

Appendix A

Architectures

A.1 Platform A

- 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 V6.1-019
- Compiler flags: *-O4 -fast -tune host -g3*

A.2 Platform B

- CPU: AMD Athlon K75 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 2.4.21
- Compiler: GNU gcc 3.2.2
- Compiler flags: *-Werror -Wall -O3 -DPAPI*

A.3 Platform C

- CPU: AMD Athlon XP 2400+, VIA KT333 Chipset
- Name: fauia22.informatik.uni-erlangen.de
- L1 Cache: 64KB Data, 64KB Instructions, 2-way set-associative
- L2 Cache: 256KB, direct-mapped, on chip
- TLB Size: 32 entries L1-TLB, 256 entries L2-TLB
- Operating System: Linux with Kernel 2.4.21
- Compiler: GNU gcc 3.2.2
- Compiler flags: *-Wall -Werror -O3 -march=athlon-xp -mcpu=athlon-xp*

A.4 Platform D

- CPU: Intel Pentium 4 2.4GHz, Single Channel DDR333, Granite Bay (E7205) Chipset
- Name: fauia23.informatik.uni-erlangen.de
- L1 Cache: 8KB Data 4-way set-associative, 16KB Instruction 8-way set-associative
- L2 Cache: 512KB 8-way set-associative
- TLB Size: 64 entries
- Operating System: Linux with Kernel 2.4.21
- Compiler: Intel icc 7.1 and GNU gcc 3.2.2
- Intel Compiler flags: *-O3 -tpp7 -xW -march=pentium4 -mcpu=pentium4*
- GCC Compiler flags: *-Wall -Werror -O3 -march=pentium4 -mcpu=pentium4*

A.5 Platform E

- CPU: Intel Pentium 4 2.4GHz, Dual Channel DDR400, Springdale (i865PE) Chipset
- Name: fauia46.informatik.uni-erlangen.de
- L1 Cache: 8KB Data 4-way set-associative, 16KB Instruction 8-way set-associative
- L2 Cache: 512KB 8-way set-associative
- TLB Size: 64 entries
- Operating System: Linux with Kernel 2.4.21
- Compiler: Intel icc 7.1 and GNU gcc 3.2.2
- Intel Compiler flags: *-O3 -tpp7 -xW -march=pentium4 -mcpu=pentium4*
- GCC Compiler flags: *-Wall -Werror -O3 -march=pentium4 -mcpu=pentium4*

A.6 Platform F

- CPU: AMD Opteron 1.6Ghz
- Name: Thor.rrze.uni-erlangen.de
- L1 cache: 128KB
- L2 cache: 1024KB
- TLB Size: L1 TLB 32 entries, L2 TLB 512 entries (4kByte Pages)
- Operating System: SuSE Linux Enetrprise Server 8 (AMD 64 bit), Kernel 2.4.19 (NUMA)
- Compiler: GNU gcc 3.2.2
- Compiler flags: *-O3*

A.7 Platform G

- CPU: Intel Itanium 2 900Mhz, 10GB Main Memory
- Name: mc1.rrze.uni-erlangen.de
- L1 cache: 16KB Data, 16KB Instructions, 4-way set-associative, 64byte cache line, bandwidth: 64GB/s
- L2 cache: 256KB, 4-way set-associative, 128byte cache line, bandwidth: 32GB/s
- L3 cache: 3MB onDie, 12-way set-associative, bandwidth: 32GB/s
- TLB Size: L1 TLB 32 entries, L2 TLB 128 entries (16kByte Pages)
- Operating System: Red Hat Linux 7.3, Kernel 2.4.20 (SMP)
- Compiler: Intel ecc 7.1 Build 20030519
- Compiler flags: *-O3 -tpp1 -fno-alias*

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.

[AK01] R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures.* Morgan Kaufmann Publishers, San Francisco, California, USA, 2001.

[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.

[BGS94] D.F. Bacon, S.L. Graham, and O.J. Sharp. Compiler Transformations for High-Performance Computing. *ACM Computing Surveys*, 26(4):345–420, 1994.

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

[Han98] J. Handy. *The Cache Memory Book.* Academic Press, second edition, 1998.

[HP96] J.L. Hennessy and D.A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann Publisher, Inc., San Francisco, California, USA, second edition, 1996.

[KRTW02] M. Kowarschik, U. Rüde, N. Thürey, and C. Weiß. Performance Optimization of 3D Multigrid on Hierarchical Memory Architectures. In *Proc. of the 6th Int. Conference on Applied Parallel Computing (PARA 2002)*, volume 2367 of *Lecture Notes in Computer Science*, pages 307–316, Espoo, Finland, 2002. Springer.

[KW03] M. Kowarschik and C. Weiß. An Overview of Cache Optimization Techniques and Cache-Aware Numerical Algorithms. In U. Meyer, P. Sanders, and J. Sibeyn, editors, *Algorithms for Memory Hierarchies — Advanced Lectures*, volume 2625 of *Lecture Notes in Computer Science*, pages 213–232. Springer, March 2003.

[PK03] T. Pohl and M. Kowarschik. http://www10.informatik.uni-erlangen.de/~pohlt/perf/, 2003.

[Qui03] Micheal J. Quinn. *Parallel Programming in C with MPI and OpenMP.* McGraw-Hill College, 2003.

[RT00] G. Rivera and C.-W. Tseng. Tiling Optimizations for 3D Scientific Computations. In *Proc. of the ACM/IEEE Supercomputing Conference*, Dallas, Texas, USA, 2000.

[Sto02] Jon Stokes. Understanding CPU Caching and Performance. Technical report, Ars Technica, LLC, July 2002.

[Suc01] S. Succi. *The Lattice Boltzmann Equation for Fluid Dynamics and Beyond.* Numerical Mathematics and Scientific Computation. Oxford University Press, 2001.

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

[Wil03] J. Wilke. Cache Optimizations for the Lattice Boltzmann Method in 2D. Lehrstuhl für Informatik 10 (Systemsimulation), Institut für Informatik, University of Erlangen-Nuremberg, Germany, February 2003. http://www10.informatik.uni-erlangen.de.

[WPKR03] Jens Wilke, Thomas Pohl, Markus Kowarschik, and Ulrich Rüde. Cache Performance Optimizations for Parallel Lattice Boltzmann Codes. In *Proc. of the EuroPar03 Conference*, Lecture Notes in Computer Science. Springer, 2003. to appear.