Institut für Parallele und Verteilte Systeme

Abteilung Simulation großer Systeme

Universität Stuttgart
Universitätsstraße 38
D - 70569 Stuttgart

Studienarbeit Nr. 2020

Entwicklung eines Cache- und SSE2-optimierten Lattice-Boltzmann-Strömungssimulationsprogramms

Ralf Kible

Studiengang:	Informatik
Prüfer:	Prof. Dr. rer. nat. Hans-Joachim Bungartz
Betreuer:	Dr.-Ing. Martin Bernreuther
begonnen am:	09.05.2005
beendet am:	09.11.2005
CR-Klassifikation:	C.4, I.6.3, C.1.2

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Quellcodeausschnitte

1 Einleitung

Die Ende der 1980er Jahre entwickelte Lattice-Boltzmann-Methode ist ein etabliertes Verfahren der numerischen Strömungssimulation. Die numerische Strömungssimulation hat ihre Anwendungen z. B. im Fahrzeug- und Flugzeugbau, in der Meteorologie aber auch in der Geophysik.

Obwohl die Lattice-Boltzmann-Methode im Verhältnis zu klassischen Ansätzen (beispielsweise Lösen der Navier-Stokes-Gleichungen mittels Finite Differenzen-Verfahren) recht effizient ist, so bedeuten besonders dreidimensionale Simulationen mit ausreichender Auflösung einen hohen Speicher- und Rechenaufwand. Da die Berechnungen der Lattice-Boltzmann-Methode besonders gut parallelisierbar sind, ist sie eine prototypische Anwendung für moderne Hoch- und Höchstleistungsrechner.

In den letzten Jahren ist in diesem Bereich ein Trend weg von speziellen Architekturen hin zu Clustern aus normalen PCs oder Workstations zu erkennen. In der aktuellen Top500-Liste ([Top05]) der 500 schnellsten Supercomputer sind 253 Rechner mit Intel Xeon oder Pentium 4 und 35 Rechner mit AMD Opteron Prozessoren vertreten. Es liegt also nahe, sich auch im Bereich von Hochleistungsrechnern mit den Spezialitäten dieser Architekturen auseinanderzusetzen.

Zu diesen Spezialitäten gehören die von Intel entwickelten SSE- und SSE2-Befehlssatzerweiterungen. Diese Erweiterungen können auf unterschiedliche Art und Weise eingesetzt werden. Die einfachste Möglichkeit ist die automatische Verwendung mittels Compiler-Optimierungen, wobei hier der Programmierer keinerlei Einfluss auf den Einsatz von SSE2-Befehlen hat. Die komplexeste Möglichkeit ist die direkte Programmierung in Assembler oder innerhalb des Quellcodes mittels Inline-Assembler, wobei der Programmierer hier die vollständige Kontrolle aber auch Verantwortung hat, da Compiler keine Optimierungen oder Überprüfungen bei Inline-Assembler mehr vornehmen (können), siehe dazu auch [Hau05].

Die hier untersuchte Möglichkeit ist der Einsatz von sogenannten intrinsischen Funktionen (*intrinsic functions* oder kurz *intrinsics*). Intrinsische Funktionen erscheinen im Quellcode wie normale Funktionsaufrufe, werden aber vom Compiler komplett und direkt in entsprechenden Maschinencode umgesetzt, wobei der Compiler die Verwaltung der Register übernimmt und typische Optimierungen anwenden kann.

Das Ziel dieser Arbeit ist es, das Potential der SSE2-Erweiterungen in einem einfachen Lattice-Boltzmann Simulationsprogramm zu untersuchen, das im Rahmen dieser Arbeit zunächst entwickelt wurde.

1.1 Überblick

Kapitel 2 beinhaltet zunächst eine kurze Einführung in die theoretischen Grundlagen der Lattice-Boltzmann-Methode, um darauf aufbauend einige Hinweise zur Implementierung zu geben. Kapitel 3 beschreibt die SSE- und SSE2-Erweiterungen zunächst unabhängig von der Lattice-Boltzmann-Methode.

Kapitel 4 beschreibt Optimierungen, die gewissermaßen zum Standardrepertoire gehören, bevor in Kapitel 5 die speziellen Anforderungen des Lattice-Boltzmann-Verfahrens bezüglich Speicherbandbreite behandelt werden. Dabei wird zunächst die Problematik der Speichergeschwindigkeit erläutert und danach werden unterschiedliche Methoden vorgestellt, um den Speicherzugriff zu beschleunigen.

Kapitel 6 beschreibt Optimierungen durch den Einsatz von SSE und SSE2, wobei sowohl der Speicherzugriff als auch die Berechnungen selbst betrachtet werden.

Die Auswirkungen der einzelnen Optimierungen werden in Kapitel 7 diskutiert und abschließend wird in Kapitel 8 eine kurze Zusammenfassung der wichtigsten Ergebnisse der Arbeit gegeben.

2 Die Lattice-Boltzmann-Methode

Da eine umfassende Vorstellung und Herleitung der theoretischen Grundlagen der Lattice-Boltzmann-Methode bereits umfangreiche Bücher füllt und diese Arbeit vielmehr die Implementierung und Algorithmik des Verfahrens in den Mittelpunkt stellt, wird die theoretische Einführung knapp gehalten. Für eine umfangreiche Einführung sei auf die Literatur, z. B. [WG00], verwiesen.

2.1 Theoretische Grundlagen

In der Strömungslehre wird das Verhalten „einfacher“ Fluide (einfach hier im Sinne Newtonscher Fluide, deren Materialeigenschaften nicht vom derzeitigen Fließzustand abhängen) mittels der Navier-Stokes-Gleichungen beschrieben. Die Navier-Stokes-Gleichungen sind ein System von nichtlinearen partiellen Differentialgleichungen zweiter Ordnung. Damit können klassische Verfahren zur numerischen Lösung von Differentialgleichungen wie die Finite-Differenzen-Methode oder die Finite-Elemente-Methode auch im Bereich der numerischen Strömungssimulation eingesetzt werden. Um die Navier-Stokes-Gleichungen benutzen zu können, müssen jedoch oftmals Materialparameter und Ergänzungsterme experimentell bestimmt werden.

Eine andere Herangehensweise ermöglicht die statistische Mechanik. Darin werden Vielteilchensysteme betrachtet und es wird versucht, aus mikroskopischen Eigenschaften der Teilchen und deren Wechselwirkungen mittels statistischer Methoden Aussagen über das makroskopische Verhalten der Systeme zu machen. Eine dieser verwendeten statistischen Gleichungen ist die Boltzmann-Gleichung.

Mit der Chapman-Enskog-Multiskalenentwicklung ist es nun unter gewissen Voraussetzungen möglich, die Navier-Stokes-Gleichungen und deren Parameter aus der Boltzmann-Gleichung zu erhalten (für Details siehe [Kra01]), was bedeutet, dass (unter gewissen Voraussetzungen) Lösungen der Boltzmann-Gleichung auch Lösungen der Navier-Stokes-Gleichungen darstellen.

Boltzmann-Gleichung

Der Boltzmann-Gleichung 2.1 liegt die Vorstellung zugrunde, dass sich das Verhalten des Vielteilchenmodells durch eine Verteilungsfunktion in einem Phasenraum beschreiben lässt, der durch die Raumkomponenten x und die Geschwindigkeitskomponenten ξ aufgespannt wird:

$$\frac{\partial f}{\partial t} + \xi \cdot \frac{\partial f}{\partial x} + F \cdot \frac{\partial f}{\partial \xi} = \Omega(f) \tag{2.1}$$

Dabei gibt die Verteilungsfunktion $f = f(x, \xi, t)$ die Wahrscheinlichkeit an, ein Teilchen zum Zeitpunkt t am Ort x mit der Geschwindigkeit ξ anzutreffen, F ist eine äußere Kraft. Die ersten beiden Terme der linken Seite modellieren den Teilchentransport, der dritte Term der linken Seite stellt den Einfluss der äußeren Kraft dar. Die rechte Seite besteht aus dem Kollisionsoperator Ω, der die Wechselwirkung der Teilchen beschreibt.

Vereinfachte Boltzmann-Gleichung

Der Kollisionsoperator Ω besteht aus einem Integral, welches die Boltzmann-Gleichung zu einer Integro-Differentialgleichung macht, die im Allgemeinen schwieriger zu lösen ist als die Navier-Stokes-Gleichungen.

Die Beobachtung, dass sich die Verteilung durch die Kollisionen der Maxwellschen Gleichgewichtsverteilung annähert, führte zu dem vereinfachten Ansatz 2.2 für den Kollisionsoperator.

$$\Omega(f) = -\frac{1}{\tau}(f - f^{eq}) \tag{2.2}$$

Dabei ist f^{eq} die Maxwellsche Gleichgewichtsverteilung und die Relaxationszeit τ steuert die Geschwindigkeit, mit der sich die derzeitige Verteilung an die Gleichgewichtsverteilung annähert. τ kann als mittleres Zeitintervall zwischen den Teilchenstößen gedeutet werden.

Dieser vereinfachte Kollisionsoperator geht auf Bhatnagar, Gross und Krook zurück und wird deswegen BGK-Approximation genannt.

Falls man zusätzlich äußere Kräfte vernachlässigt, so gelangt man zur vereinfachten Boltzmann-Gleichung 2.3:

$$\frac{\partial f}{\partial t} + \xi \cdot \frac{\partial f}{\partial x} = -\frac{1}{\tau}(f - f^{eq}) \tag{2.3}$$

Lattice-Boltzmann-Gleichung

Um diese Gleichung numerisch zu lösen diskretisiert man zunächst die mikroskopische Geschwindigkeit ξ, indem man eine Menge $\{\xi_i | i = 1..N\}$ Kollokationspunkte der Geschwindigkeitskomponenten wählt und die Verteilungsfunktion $f(x, \xi, t)$ durch die Werte $f_i = f(x, \xi_i, t)$ ausdrückt. Für die Wahl dieser Kollokationspunkte existieren viele Modelle, die in der Literatur einheitlich mit DxQy bezeichnet werden, wobei x die Dimension des Raumes darstellt und y die Anzahl der Kollokationspunkte. Die Bezeichnung DxQy geht zurück auf Qian ([QdL92]). Gängige Modelle sind D2Q9, D3Q15 oder D3Q19. Man erhält die diskrete Boltzmann-Gleichung 2.4:

$$\frac{\partial f_i}{\partial t} + \xi_i \cdot \frac{\partial f_i}{\partial x} = -\frac{1}{\tau}(f_i - f_i^{eq}), i = 1, \cdots, N \tag{2.4}$$

Eine Diskretisierung in Raum und Zeit mittels finiter Differenzen führt nach einigen Vereinfachungen zur Lattice-Boltzmann-Gleichung 2.5:

$$f_i(x + \xi_i \Delta t, t + \Delta t) = f_i(x, t) - \frac{\Delta t}{\tau}(f_i(x, t) - f_i^{eq}(x, t)), i = 1, \cdots, N \tag{2.5}$$

Man erhält aus den diskreten Werten der Verteilungsfunktion die makroskopische Dichte

$$\rho = \sum_{i=1}^{N} f_i(x,t) \tag{2.6}$$

und die makroskopische Geschwindigkeit

$$u = \frac{1}{\rho} \sum_{i=1}^{N} \xi_i f_i(x,t) \tag{2.7}$$

Gleichgewichtsverteilung

Für die numerische Lösung muss noch die Gleichgewichtsverteilung nur unter Verwendung der Kollokationspunkte ausgedrückt werden. Hierzu wählt man den Ansatz 2.8 (für eine Begründung siehe [Kra01]):

$$f_i^{eq}(x,t) = w_p \rho \left(1 + \frac{< \xi_{i\alpha}, u_\alpha >}{c_s^2} + \frac{< u_\beta, u_\alpha >}{2c_s^2} \left(\frac{< \xi_{i\alpha}, \xi_{i\beta} >}{c_s^2} - \delta_{\alpha\beta} \right) \right) \tag{2.8}$$

Dabei folgen die griechischen Indizes α, β der Einsteinschen Summenkonvention. Des Weiteren bezeichnet $\delta_{\alpha\beta}$ das Kronecker-Delta, c_s ist die Schallgeschwindigkeit und w_p sind Konstanten, die abhängig vom Modell und den Geschwindigkeitskomponenten bestimmt werden müssen.

Für das D3Q19-Modell beispielsweise (siehe dazu auch Kapitel 2.1), gelten: $w_0 = \frac{1}{3}$, $w_1 = \frac{1}{18}$, $w_2 = \frac{1}{36}$ und $c_s^2 = \frac{1}{3}$.

Randbedingungen

An dieser Stelle sollen nur die in dieser Arbeit verwendeten Randbedingungen kurz beschrieben werden. Tatsächlich gibt es noch einige weitere Arten von Randbedingungen. Für eine umfangreiche Übersicht sei einmal mehr auf die Literatur ([Kra01]) verwiesen.

Periodische Randbedingung

Die periodischen Randbedingungen (*periodic boundary conditions, pbc*) werden eingesetzt, um das Verhalten eines wesentlich größeren Systems zu simulieren. Dabei wird davon ausgegangen, dass sich das Simulationsgebiet periodisch wiederholt, und Teilchen, die das Simulationsgebiet auf einer Seite verlassen, werden an der gegenüberliegenden Seite wieder eingesetzt. Die beiden gegenüberliegenden Ränder werden also identifiziert, in 2D entsteht dadurch eine Torustopologie, wenn an allen Rändern periodische Randbedingungen benutzt werden.

In dieser Arbeit werden an allen Rändern des Simulationsgebiets periodische Randbedingungen verwendet, falls man also Wände in gewissen Raumrichtungen benötigt, so müssen diese explizit als Hindernisse modelliert werden.

Haftrandbedingung

Die Haftrandbedingung (*no-slip condition*) fordert, dass die Geschwindigkeit des Flusses gleich der Randgeschwindigkeit ist. In dieser Arbeit wird diese Randbedingung für feststehende Hindernisse benutzt, damit ergibt sich eine Geschwindigkeit $u = 0$. Dies wird realisiert, indem die Richtung der auftreffenden Teilchen umgekehrt wird (in der Literatur *bounceback* genannt).

Da eine Zelle nun entweder ein Hindernis mit Haftrandbedingung sein kann oder eben nicht, können schräg im Gitter liegende Wände nur schwer modelliert werden.

Druck-Geschwindigkeitsrandbedingung

Ohne diese Randbedingung würden keine Strömungen auftreten, da nur hier Druck und Geschwindigkeit vorgegeben werden können, wobei natürlich weiterhin die Massenerhaltung gelten muss. Dies wird dadurch realisiert, dass die Teilchen innerhalb einer Zelle in die vorgegebene Richtung verschoben werden, um eine Strömung in diese Richtung zu erreichen.

Das D3Q19-Modell

Grundlage dieser Arbeit ist das D3Q19-Modell, das konkret vorgestellt werden soll. Wie bereits gesagt basiert das Modell auf einem dreidimensionalen Raum mit 19 Kollokationspunkten pro Zelle. Zum besseren Verständnis sind die 19 Kollokationspunkte nicht durchnumeriert, sondern werden angelehnt an [Hau05] bezeichnet als C (center), N (north), S (south), E (east), W (west), T (top), B (bottom), NE (north east), NW (north west), SE (south east), SW (south west), TN (top north), BN (bottom north), TS (top south), BS (bottom south), TE (top east), BE (bottom east), TW (top west) und BW (bottom west). Definiert sind die Geschwindigkeiten wie folgt:

$$\xi_i := \begin{cases} (0,0,0) & i = C \\ (\pm 1,0,0) & i = E, W \\ (0,\pm 1,0) & i = N, S \\ (0,0,\pm 1) & i = T, B \\ (\pm 1,\pm 1,0) & i = NE, SE, NW, SW \\ (\pm 1,0,\pm 1) & i = TE, BE, TW, BW \\ (0,\pm 1,\pm 1) & i = TN, BN, TS, BS \end{cases}$$

Für eine anschauliche grafische Darstellung siehe Abbildung 2.1.

Die lokale Gleichgewichtsverteilung ergibt sich zu:

$$f_i^{eq}(x,t) = \rho w_i \left(1 + 3 < \xi_i, u > + \frac{9}{2} < \xi_i, u >^2 - \frac{3}{2} < u, u >\right) \tag{2.9}$$

Dabei ist u die makroskopische Geschwindigkeit und ρ die Dichte der Zelle (siehe Glei-

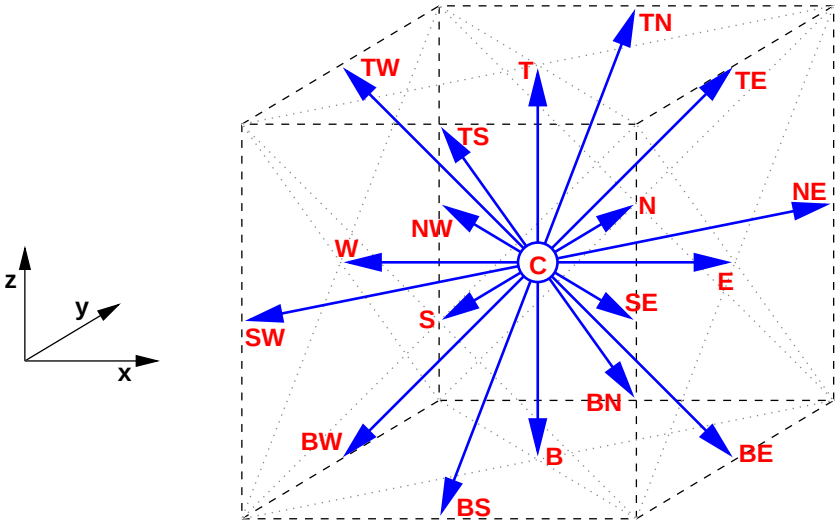


Abbildung 2.1: Diskrete Geschwindigkeiten des D3Q19-Modells

chung 2.6), w_i ist ein Gewichtungsfaktor:

$$w_i = \begin{cases} \frac{1}{3} & i = C \\ \frac{1}{18} & i = E, W, N, S, T, B \\ \frac{1}{36} & i = NE, NW, SE, SW, TE, TW, TN, TS, BE, BW, BN, BS \end{cases}$$

Zum Zeitpunkt $t = 0$ werden folgende Anfangsbedingungen verwendet, die sich aus der Gleichgewichtsverteilung ergeben, indem zu Beginn $u = 0$ angenommen wird, dabei bezeichnet ρ_0 die initiale Dichte der Zelle:

$$f_i(x, t = 0) = \rho_0 w_i$$

Gleichung 2.5 lässt sich nun leicht in zwei Schritten berechnen:

$$\text{Kollision:} \quad \tilde{f}_i(x,t) = f_i(x,t) - \frac{\Delta t}{\tau}(f_i(x,t) - f_i^{eq}(x,t)) \tag{2.10}$$

$$\text{Propagation:} \quad f_i(x + \xi_i \Delta t, t + \Delta t) = \tilde{f}_i(x,t) \tag{2.11}$$

In der Literatur wird diese Reihenfolge der beiden Schritte als *collide-stream*[1]- oder *push*-Methode bezeichnet, da die beiden Schritte auch vertauscht werden können. Die Vertauschung der beiden Schritte bezeichnet man als *stream-collide*-Methode.

2.2 Hinweise zur Implementierung

Die Realisierung dieser Arbeit erfolgte in C, wobei der ANSI-C99-Standard zugrunde gelegt wurde[2]. Die gegebenen Quellcodebeispiele orientieren sich deswegen an der C-Syntax.

Die Gleichungen 2.10 und 2.11 können direkt implementiert werden, indem man für $\tilde{f}_i$ ein zweites Feld vorsieht (`pdf_cp`).

[1] *stream* bezeichnet die Propagation
[2] soweit möglich, da die Compiler-Unterstützung an einigen Stellen immer noch mangelhaft ist

```
typedef enum {C=0, E=1, W=2, N=3, S=4, T=5, B=6,
              NE=7, NW=8, SE=9, SW=10, TE=11, TW=12, BE=13,
              BW=14, TN=15, TS=16, BN=17, BS=18} directions;
typedef enum {fluid=0, obstacle=1, acceleration=2} nodetype;

double pdf[xdim][ydim][zdim][19];      // speichert f
double pdf_cp[xdim][ydim][zdim][19];   // Kopierfeld für f
nodetype type[xdim][ydim][zdim];       // speichert den Zellentyp
```

Quellcodeausschnitt 2.1: Grundlegende Definitionen zur Implementierung

Ausgehend von den Definitionen in Quellcodeausschnitt 2.1 kann ein Schritt der Simulation wie in Quellcodeausschnitt 2.2 durchgeführt werden.

```
for (int x=0; x<dimx; ++x) {
  for (int y=0; y<dimy; ++y) {
    for (int z=0; z<dimz; ++z) {
      switch (type[x][y][z]) {
        case fluid:
          collide(x,y,z);
          break;
        case obstacle:
          bounceback(x,y,z);
          break;
        case acceleration:
          collide(x,y,z);
          redistribute(x,y,z);
          break;
      }
    }
  }
}
for (int x=0; x<dimx; ++x) {
  for (int y=0; y<dimy; ++y) {
    for (int z=0; z<dimz; ++z) {
      stream(x,y,z);
    }
  }
}
```

Quellcodeausschnitt 2.2: Simulationsschritt

Die unoptimierten Funktionen ergeben sich direkt aus den Gleichungen 2.10, 2.11 bzw. den Randbedingungen aus Kapitel 2.1 und werden nur verkürzt wiedergegeben:

```
void collide(const int x, const int y, const int z) {
  double rho,ux,uy,uz;
  double feq[19];

```

```
5   rho = pdf[x][y][z][C] + pdf[x][y][z][E] + ...
6         + pdf[x][y][z][BS];
7   ux =  pdf[x][y][z][E] + pdf[x][y][z][NE] + ...
8       - pdf[x][y][z][W] - pdf[x][y][z][NW] - ... ;
9   uy =  pdf[x][y][z][N] + pdf[x][y][z][NE] + ...
10      - pdf[x][y][z][S] - pdf[x][y][z][SE] - ... ;
11  uz =  pdf[x][y][z][T] + pdf[x][y][z][TN] + ...
12      - pdf[x][y][z][B] - pdf[x][y][z][BN] - ... ;
13  ux /= rho;
14  uy /= rho;
15  uz /= rho;
16
17  feq[C] = 1.0/ 3.0*invtau*(1.0
18                                -3.0/2.0*(ux^2+uy^2+uz^2))
19  feq[E] = 1.0/18.0*invtau*(1.0+3.0*ux+9.0/2.0*ux*ux
20                                -3.0/2.0*(ux^2+uy^2+uz^2))
21  ...
22
23  pdf_cp[x][y][z][C]
24        = (1.0-invtau)*pdf[x][y][z][C]+invtau*feq[C];
25  pdf_cp[x][y][z][E]
26        = (1.0-invtau)*pdf[x][y][z][E]+invtau*feq[E];
27  ...
28
29 }
```

Quellcodeausschnitt 2.3: `collide` (unoptimiert)

```
1 void stream(const int x, const int y, const int z) {
2   pdf[x][y][z][C]       = pdf_cp[x][y][z][C];
3   pdf[x+1][y][z][E]     = pdf_cp[x][y][z][E];
4   ...
5   pdf[x][y-1][z-1][BS] = pdf_cp[x][y][z][BS];
6 }
```

Quellcodeausschnitt 2.4: `stream`

Die Funktionen `bounceback` (Quellcodeausschnitt 2.5) und `redistribute` (Quellcodeausschnitt 2.6) sind hier der Vollständigkeit halber ebenfalls wiedergegeben. Sie werden im folgenden nicht weiter betrachtet, da sich eine Optimierung hier nicht lohnen würde, siehe dazu Diagramm 4.1.

```
1 void bounceback(const int x, const int y, const int z) {
2   pdf_cp[x][y][z][W]  = pdf[x][y][z][E]
3   pdf_cp[x][y][z][E]  = pdf[x][y][z][W]
4   ...
5   pdf_cp[x][y][z][TN] = pdf[x][y][z][BS]
6 }
```

Quellcodeausschnitt 2.5: `bounceback`

```
1  void redistribute(const int x, const int y, const int z) {
2    double accel;
3
4    accel = acceleration_coefficient*initial_rho/18.0;
5
6    if ((pdf_cp[x][y][z][W] <= accel) ||
7        (pdf_cp[x][y][z][SW] <= accel/2) ||
8        ... ) {
9      return;
10   }
11
12   pdf_cp[x][y][z][W](x,y,z,E)  += accel;
13   pdf_cp[x][y][z][W](x,y,z,SE) += accel/2;
14   ...
15
16   pdf_cp[x][y][z][W](x,y,z,W)  -= accel;
17   pdf_cp[x][y][z][W](x,y,z,SW) -= accel/2;
18   ...
19 }
```

Quellcodeausschnitt 2.6: `redistribute`

Halo-Zellen

Wie bereits in Kapitel 2.1 beschrieben, werden an allen Rändern periodische Randbedingungen benutzt. Diese könnten realisiert werden, indem bei jeder Adressierung der Felder `pdf` und `pdf_cp` bei Bedarf modulo-Rechnungen durchgeführt werden, um zu verhindern, dass außerhalb der Feldgrenzen geschrieben wird.

Dies ist jedoch nicht nur im Hinblick auf die Rechenleistung von Nachteil. Auch im Hinblick auf eine spätere Parallelisierung bietet sich eine andere Vorgehensweise an: An jedem Rand wird eine zusätzliche Schicht Zellen angefügt, die sogenannten Ghost- oder Halo-Zellen. Diese werden nie kollidiert, jedoch kann in der Propagationsphase in diese geschrieben werden. Am Ende eines Simulationsschrittes müssen dann die Werte in den Halo-Zellen an die richtigen Positionen in den normalen Zellen geschrieben werden, siehe Abbildung 2.2. Im Falle einer Parallelisierung mittels Gebietszerlegung können die Halo-Zellen als Puffer zu den benachbarten Prozessen benutzt werden.

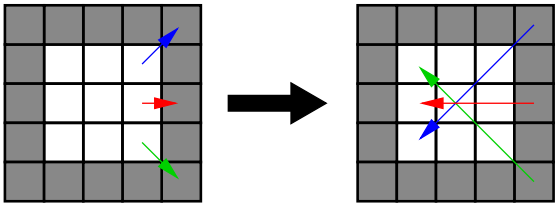


Abbildung 2.2: Zurückkopieren der Halo-Zellen, links: Propagation in die Halo-Zellen (grau), rechts: Kopieren der Werte in den Halo-Zellen an die richtige Position

3 SSE

In diesem Kapitel sollen die Grundlagen der SSE-Befehlssatzerweiterungen zunächst unabhängig von der Lattice-Boltzmann-Methode betrachtet werden. Die Anwendung der SSE-Erweiterungen auf die Lattice-Boltzmann-Methode folgt in Kapitel 6.

3.1 Entwicklung und Geschichte der SSE

Im Jahre 1997 führte Intel unter großem Medieneinsatz den Pentium-Prozessor mit MMX-Technologie ein. MMX (*Multimedia Extensions*) war die erste SIMD-Befehlssatzerweiterung in den Produktreihen von Intel und ist somit der Vorgänger der SSE-Erweiterungen, wobei MMX in modernen Prozessoren immer noch vorhanden ist. SIMD bedeutet *single instruction multiple data*, es ist also möglich mit einem einzelnen Maschinenbefehl eine Operation auf mehrere Daten gleichzeitig durchzuführen.

Zu diesem Zweck wurden 8 neue 64 Bit-Register in die CPU integriert. In jedem dieser Register können 2 32 Bit-Integerwerte, 4 16 Bit-Integerwerte oder auch 8 8 Bit-Integerwerte gespeichert und gleichzeitig verarbeitet werden, man nennt dies gepackte Operationen (*packed operations*). Außerdem beinhaltet MMX Befehle, bei deren Ausführung das Ergebnis den Wertebereich nicht über- bzw. unterschreiten konnte, sondern dann den größt- bzw. kleinstmöglichen Wert behält(*saturation*). Dies alles ist auf die Verarbeitung von Audio- und Videodaten zugeschnitten.

Der größte Nachteil von MMX ist jedoch, dass die Register der Fließkomme-Einheit ebenfalls benutzt werden und somit nicht mehr verfügbar sind, sobald MMX-Befehle eingesetzt werden. Vielmehr muss der Programmierer sicherstellen, dass nicht normale und MMX-Befehle gemischt sind und am Ende einer Berechnung mit MMX muss der sogenannte MMX-Zustand zurückgesetzt werden, um die Register wieder als verfügbar zu markieren.

Diesen Mangel beseitigte Intel im Jahre 1999 mit der Einführung des Pentium III. Der Pentium III brachte eine zusätzliche Befehlssatzerweiterung mit sich, die sogenannten *Internet Streaming SIMD Extensions* (ISSE). Da die Erwähnung des Internets nur zu Werbezwecken geschah und durch ISSE keine besondere Beschleunigung des Internetsurfens zu erwarten war, wurde das I in den kommenden Generationen gestrichen, man spricht heutzutage also von SSE.

SSE benutzt eigenständige Register, welche gleichzeitig auf 128 Bit vergrößert wurden. Außerdem wurden mit SSE Befehle zur Verarbeitung von Fließkommazahlen mit einfacher Genauigkeit hinzugefügt.

Bereits Ende 2000 schob Intel mit dem Pentium 4 die SSE2-Erweiterungen nach, die auch die Verarbeitung von Fließkommazahlen mit doppelter Genauigkeit ermöglichen, was sie interessant für numerische Simulationen macht. Außerdem wurde die Anzahl

der Register verdoppelt. 2004 wurden schließlich mit dem Pentium 4 mit Prescott-Kern die SSE3-Erweiterungen eingeführt, die jedoch keine technischen Neuerungen gegenüber SSE2 brachten, sondern nur komplexere arithmetische Befehle hinzufügten, wie Addition und Subtraktion in einem Maschinenbefehl.

Ein Nachteil der SSE2-Erweiterungen soll nicht verschwiegen werden. Normalerweise werden Operationen auf 64 Bit `double`-Werten intern mit einer erweiterten Genauigkeit von 80 Bit durchgeführt. Da ein SSE-Register jedoch eine Breite von 128 Bit besitzt und darin Operationen auf zwei `double`-Werten mit je 64 Bit durchgeführt werden, kann nicht mit der erweiterten Genauigkeit von 80 Bit gearbeitet werden. Leider findet sich hierzu keine verlässliche Quelle, sondern dieser Punkt wird nur an verschiedenen Stellen im Internet, beispielsweise `http://www.fftw.org/accuracy/comments.html` diskutiert. Dazu ist noch zu sagen, dass manche Optimierungsstufen des Compilers bereits die erweiterte Genauigkeit aufgeben, hier ist also Vorsicht geboten, sofern die erweiterte Genauigkeit benötigt wird.

Wie dargestellt handelt es sich bei SSE und SSE2 um Befehlssatzerweiterungen. Die neuen Befehle können also prinzipiell in einer Hochsprache mittels Inline-Assembler genutzt werden. Abgesehen davon, dass das Programmieren in Assembler deutlich komplizierter ist als in einer Hochsprache, hat dies noch den weiteren Nachteil, dass der Compiler in Assembler-Blöcken keinerlei Optimierungen vornimmt. Um dem Abhilfe zu schaffen, bieten moderne Compiler sogenannte intrinsische Funktionen (*intrinsic functions*), die vom Compiler speziell behandelt werden. Dies ist der Weg, der im Folgenden beschritten werden soll.

3.2 Intrinsische Funktionen und gepackte Operationen

Eine intrinsische Funktion ist eine Funktion, über die der Compiler spezielle Kenntnisse hat und direkt optimierten Maschinencode einfügen kann. So sind in [Int04b] die Maschinenbefehle aufgeführt, die bei den intrinsischen Funktionen verwendet werden. Der Vorteil von intrinsischen Funktionen gegenüber der Verwendung von Inline-Assembler ist, dass der Compiler das Laden und die Verwaltung der Register übernimmt.

Im Gegensatz dazu stehen Inline-Funktionen. Diese werden jedoch nicht direkt durch Maschinenbefehle ersetzt, sondern an der Stelle des Aufrufs wird die Implementierung eingesetzt und danach wird der Maschinencode erzeugt und dabei optimiert.

Intrinsische Funktionen können für unterschiedlichste Operationen benutzt werden können, beispielsweise sind Funktionen wie `memset` oder `strlen` in manchen Compilern intrinsisch ausgeführt. Dennoch sind im folgenden mit intrinsischen Funktionen immer speziell die intrinsischen Funktionen zum Zugriff auf SSE- und SSE2-Befehle gemeint.

Wie bereits in der Einführung genannt, bieten gepackte Operationen die Möglichkeit, eine Operation auf mehreren Daten gleichzeitig durchzuführen. Für einen umfangreichen Überblick über die intrinsischen Funktionen zu diesem Zweck siehe [Int04b] und Anhang C, hier eine kurze Einführung.

Ein 128 Bit SSE-Register kann zwei Fließkommazahlen doppelter Genauigkeit aufnehmen. Im Quellcode werden zwei derart gepackte `double`-Werte durch den Datentyp

`__m128d` repräsentiert. Es sind zahlreiche weitere Datentypen verfügbar, diese interessieren hier jedoch nicht weiter.

Die Rechenoperationen, die auf `__m128d`-Variablen durchgeführt werden können, haben alle die Form `__m128d _mm_`*op*`_pd(__m128d a, __m128d b)`, wobei *op* entweder `add`, `sub`, `mul` oder `div` für die jeweilige Rechenoperation sein kann.

Die intrinsische Funktion `__mm128d _mm_load_pd(double *p)` lädt zwei `double`-Werte ab Adresse `*p`, analog dazu speichert `_mm_store_pd(double *p, __128d a)` die beiden Werte aus `a` an die Stelle `*p`. Zu beachten ist, dass sowohl zum Laden wie auch zum Speichern die Adressen `*p` an 16 Byte Grenzen ausgerichtet sein müssen.

Nützlich ist `__mm128d _mm_shuffle_pd(__128d a, __128d b, _MM_SHUFFLE2(x, y))`, das aus `a` und `b` einen neuen `__mm128d` Wert erzeugt, wobei `_MM_SHUFFLE2(x,y)` ein Makro ist, mit dem ausgewählt wird, welche Werte aus a und b im Ergebnis auftauchen, siehe dazu Anhang C.2.3.

3.3 Ein Beispiel

An dieser Stelle soll an einem einfachen Beispiel die Verwendung von intrinsischen Funktionen im Quellcode verdeutlicht werden.

Die Aufgabenstellung ist eine einfache Vektoroperation. Gegeben seien ein Vektor `a` und ein Skalar `b`, berechnet werden soll die ganzzahlige Division der Werte von `a` durch `b` und der Rest, also $a[i] = b \cdot ergebnis[i] + rest[i]$. Die Umsetzung ist sehr einfach und könnte aussehen wie in Quellcodeausschnitt 3.1.

```
1 for (i=0; i<N; i++) {
2   rest[i] = a[i] / b;
3   ergebnis[i] = floor(rest[i]);
4   rest[i] = (rest[i] - ergebnis[i])*b;
5 }
```

Quellcodeausschnitt 3.1: Konventionelle Berechnung der Vektoroperation

3.3.1 Gepackte Operationen

Diese Schleife könnte für eine Vektorisierung von Hand mittels intrinsischer Funktionen geeignet sein. Dazu muss zunächst das Speicherlayout der Daten betrachtet werden. Sofern dies möglich ist, sollten die Daten, die in der SSE-Einheit verarbeitet werden sollen, auf 16 Byte Grenzen ausgerichtet sein, da dann die größte Leistung erzielt werden kann.

Alle gängigen Compiler bieten Erweiterungen an, um dies bei der Deklaration einer Variablen sicherzustellen. Leider unterscheidet sich die Syntax der Compiler-spezifischen Erweiterungen. In Quellcodeausschnitt 3.2 finden sich am Anfang ein Präprozessor-Konstrukt, durch das bei verschiedenen Compilern Variablen ausgerichtet werden können, und die im Weiteren verwendeten Definitionen für die vektorisierte Version.

```
1  /* ALIGN(variable) ist ein Makro, dass die Adresse der
2     Variable auf eine 16 Byte Grenze ausrichtet */
3  #ifdef __INTEL_COMPILER  /* Intelcompiler */
4  #define ALIGN(var) __declspec(align(16)) var
5  #else  /* sonst: Vermutlich GCC */
6  #define ALIGN(var) var __attribute__ ((__aligned__ (16)))
7  #endif
8
9  double ALIGN(a[N]);
10 double ALIGN(ergebnis[N]);
11 double ALIGN(rest[N]);
12
13 double b = 10.0;
14
15 __m128d divisor, dividend, ganzzahlig, temp;
16
17 /* b in beide Teilworte von divisor laden */
18 divisor = _mm_load1_pd(&b);
```

Quellcodeausschnitt 3.2: Definitionen für die Vektoroperation aus Quellcodebeispiel 3.1 mit intrinsischen Funktionen

Als kurzer Einschub wird in Quellcodeausschnitt 3.3 gezeigt, wie unabhängig vom verwendeten Compiler dynamisch mit `malloc` angeforderter Speicher ausgerichtet werden kann. Dabei ist es sehr wichtig zu beachten, dass der Zeiger `puffer` der ursprünglichen Speicheranforderung nicht überschrieben wird, da sonst der angeforderte Speicher nicht mehr mit `free` freigegeben werden kann.

```
1  char *puffer, *puffer2;
2  /* Benötigten Speicher und 12 Byte zusätzlich anfordern.
3     Ausgegangen davon, dass malloc auf 4 Byte ausgerichtete
4     Zeiger zurückliefert, so werden im schlimmsten Fall
5     12 Byte zur nächsten 16 Byte Grenze benötigt. */
6  puffer = (char *) malloc(1024*sizeof(char) + 12);
7  if (puffer % 16) {
8    /* puffer ist nicht auf 16 Byte ausgerichtet, suche nun die
9       nächste 16 Byte Grenze. */
10   puffer2 = (puffer & 0xFFFFFFF0) + 16;
11 } else {
12   puffer2 = puffer;
13 } /* nun puffer2 verwenden */
```

Quellcodeausschnitt 3.3: Ausrichten einer dynamischen Speicheranforderung

In der letzten Zeile in Quellcodeausschnitt 3.2 wird in beide Werte der Variablen `divisor` der Wert von `b` geladen. Nun kann die Schleife aus Quellcodeausschnitt 3.1 unter Benutzung von intrinsischen Funktionen umgeschrieben werden. Dabei ist zu beachten, dass

nun jeder Schleifendurchlauf zwei Durchläufen der alten Schleife entspricht, deswegen läuft `i` nun in Zweierschritten, siehe Quellcodeausschnitt 3.4.

```
1  for (i=0; i<N; i+=2) {
2    dividend = _mm_load_pd(a+i);
3    temp = _mm_div_pd(dividend, divisor);
4    /* Trick zum Abschneiden: Nach Integer konvertieren, dabei
5       abschneiden und wieder zurück konvertieren */
6    ganzzahlig = _mm_cvtepi32_pd(_mm_cvttpd_epi32(temp));
7    _mm_store_pd(ergebnis+i, ganzzahlig);
8    temp = _mm_sub_pd(temp, ganzzahlig);
9    temp = _mm_mul_pd(temp, divisor);
10   _mm_store_pd(rest+i, temp);
11 }
```

Quellcodeausschnitt 3.4: Berechnung der Vektoroperation mit intrinsischen Funktionen

Betrachtet man die Leistung der beiden Versionen, so ergeben sich auf Testplattform 1 (A.1.1) die in Tabelle 3.1 aufgeführten Messwerte für N=10 000 000.

Compileroptionen	Laufzeit konventionelle Version	Laufzeit vektorisierte Version
`-O0`	0,54	0,36
`-O3 -axW`	0,37	0,32

Tabelle 3.1: Laufzeiten der Quellcodeausschnitte 3.1 und 3.4 in Sekunden, Testplattform 1 (Pentium 4), Intelcompiler, N=10 000 000

Interessant ist dabei, dass bei starker Compileroptimierung (`-axW` weist den Compiler an, automatisch vektorisierten Code für Pentium 4 zu erzeugen) der Geschwindigkeitsvorteil der SSE-Version sinkt. Dies liegt daran, dass dieses sehr einfache Beispiel bereits vom Compiler in SSE-Befehle umgesetzt wird, siehe dazu Anhang B.

Es ist jedoch so, dass diese automatische Vektorisierung nur bei einfachen Schleifen zuverlässig funktioniert. Endgültigen Aufschluss darüber, ob eine Schleife vektorisiert wurde, kann im konkreten Fall nur der vom Compiler generierte Bericht geben.

3.3.2 Prefetching

Die SSE-Erweiterungen beinhalten nicht nur gepackte arithmetische Operationen. Als Vorgriff auf die Kapitel 5 und 6.1 soll hier die Verwendung der intrinsischen Funktion `_mm_prefetch()` demonstriert werden. Diese Funktion dient dazu, Daten vorab in die Pufferspeicher (Caches, siehe Kapitel 5.2) des Prozessors zu laden, um schneller auf diese zugreifen zu können. Man spricht dabei von Prefetching.

Nun soll am gegebenen Beispiel die manuelle Verwendung der Prefetching-Anweisung gezeigt werden. Dazu werden zunächst vier Durchläufe der Schleife aus Quellcodeausschnitt 3.4 entrollt, der Schleifenrumpf wird also viermal kopiert, wobei die Laufvariable `i` entsprechend angepasst wird, siehe Quellcodeausschnitt 3.5.

Damit wird erreicht, dass in einem Durchlauf der entstandenen Schleife 64 Byte der

Felder `a`, `ergebnis` und `rest` verarbeitet werden, wobei 64 Byte gerade der Länge einer Cache-Line (ein Block im Pufferspeicher, siehe Kapitel 5.2) entsprechen.

```
1  for (i=0; i<N; i+=8) {
2    _mm_prefetch( a+i+32, _MM_HINT_NTA);
3    _mm_prefetch(fc+i+32, _MM_HINT_NTA);
4    _mm_prefetch(mc+i+32, _MM_HINT_NTA);
5
6    dividend = _mm_load_pd(a+i);
7    temp = _mm_div_pd(dividend, divisor);
8    ganzzahlig = _mm_cvtepi32_pd(_mm_cvttpd_epi32(temp));
9    _mm_store_pd(ergebnis+i, ganzzahlig);
10   temp = _mm_sub_pd(temp, ganzzahlig);
11   temp = _mm_mul_pd(temp, divisor);
12   _mm_store_pd(rest+i, temp);
13
14   [...]
15
16   dividend = _mm_load_pd(a+i+6);
17   temp = _mm_div_pd(dividend, divisor);
18   ganzzahlig = _mm_cvtepi32_pd(_mm_cvttpd_epi32(temp));
19   _mm_store_pd(ergebnis+i+6, ganzzahlig);
20   temp = _mm_sub_pd(temp, ganzzahlig);
21   temp = _mm_mul_pd(temp, divisor);
22   _mm_store_pd(rest+i+6, temp);
23 }
```

Quellcodeausschnitt 3.5: Berechnung der Vektoroperation mit Prefetching

Die ersten drei Anweisungen in Quellcodeausschnitt 3.5 realisieren das Prefetching und bewirken, dass die Teile der Felder, die in vier Schleifendurchläufen benötigt werden, in den Cache transportiert werden. Da `i` pro Durchlauf um 8 inkrementiert wird, entspricht `i+32` gerade den Werten, die vier Durchläufe später gebraucht werden. Die Konstante `_MM_HINT_NTA` ist ein Hinweis für den Prozessor, siehe Anhang C.

An dieser Stelle kann experimentiert werden, denkbar ist auch `i+16`, was bedeuten würde, dass die Werte für den übernächsten Schleifendurchlauf geladen werden. Zu bedenken ist, dass immer genug Zeit bleiben sollte, die Prefetching-Operationen fertig zu stellen, bevor die Daten tatsächlich benötigt werden.

Compileroptionen	Laufzeit ohne Prefetching	Laufzeit mit Prefetching
`-O0`	0,36	0,33
`-O3 -axW`	0,32	0,32

Tabelle 3.2: Laufzeiten der Quellcodeausschnitte 3.4 und 3.5 in Sekunden, Testplattform 1 (Pentium 4), Intelcompiler, N=10 000 000

In Tabelle 3.2 sind die Laufzeiten der vektorisierten Version mit und ohne Prefetching dargestellt. Es ist zu erkennen, dass ohne Compileroptimierung ein Geschwindigkeitsvorteil durch das Prefetching von knapp 10% entsteht und beinahe die Geschwindigkeit der

vom Compiler vollständig optimierten Version erreicht wird. Mit hoher Compileroptimierung sind beide Versionen jedoch gleich schnell.

Dies liegt daran, dass der Compiler bei hoher Optimierung selbstständig versucht, Prefetching-Operationen einzufügen. In diesem einfachen Beispiel macht dies dem Compiler keine Probleme, bei komplizierteren Speicherzugriffen kann ein manuelles Einfügen von Prefetching-Anweisungen jedoch hohe Geschwindigkeitssteigerungen hervorrufen.

4 Grundlegende Optimierungen

In diesem Kapitel werden einige Optimierungen vorgestellt, die sehr verbreitet und in vielen Lattice-Boltzmann Implementierungen zu finden sind. Das Ziel der Kapitel 4 und 5 ist es, dass ein bereits optimierter Code als Basis für die SSE-Optimierungen in Kapitel 6 zur Verfügung steht, so dass eine angemessene Beurteilung der SSE-Optimierungen möglich ist.

Einige der in diesem Kapitel genannten Optimierungen sind so verbreitet, dass diese bereits in der Grundversion realisiert wurden und deswegen keine gesonderten Vergleichsmessungen durchgeführt wurden.

4.1 Verschmelzen von Kollision und Propagation

Die Gleichungen 2.10 und 2.11 und die in Kapitel 2.2 gegebenen Hinweise deuten an, dass das Berechnungsgitter zweimal komplett durchlaufen wird. Dies ist jedoch nicht notwendig, denn die in der `collide`-Funktion berechneten Werte können direkt an die durch `stream` vorgegebenen Stellen geschrieben werden (bei `bounceback` ist dies genauso möglich).

Das bedeutet, dass am Ende eines Simulationsschritts die Ergebnisse im zweiten Feld `pdf_cp` vorliegen. Für den nächsten Schritt müssen also die beiden Felder vertauscht werden, was aber nahezu ohne Rechen- oder Speicheraufwand durch ein Umsetzen der Zeiger erreicht werden kann.

Falls das Berechnungsgitter nicht mehr vollständig im Cache gehalten werden kann, wird das Gitter durch das Verschmelzen der beiden Schritte einmal weniger in den Hauptspeicher zurückgeschrieben und einmal weniger gelesen. Diese Optimierung ist weit verbreitet und die Basis für andere Optimierungen (z.B. Kapitel 5.4).

Im Folgenden wird diese Optimierung als obligatorisch angesehen, die Funktion `collide` führt also stets Kollision und Propagation gemeinsam aus.

4.2 Arithmetische Optimierungen

Auch wenn die Lattice-Boltzmann-Methode hohe Anforderungen an den Speicherdurchsatz stellt, so ist es dennoch sinnvoll, die Anzahl der Gleitkomma-Berechnungen möglichst gering zu halten. Dass dies wichtig ist, zeigt Diagramm 4.1. Dargestellt ist der relative Zeitbedarf der vier Routinen, die innerhalb eines Simulationsschrittes ausgeführt werden:

- `Redistribute`: Setzen der Druck-Geschwindigkeitsrandbedingung

- `Switchfields`: Tauschen des primären Feldes und des Kopierfeldes und kopieren der Halo-Zellen
- `Bounceback`: Verarbeiten der Hindernis-Zellen
- `Collide`: Verarbeiten der Fluid-Zellen, dabei werden Kollision und Propagation verschmolzen durchgeführt

Die Grafik entstand aus einer Simulation mit Callgrind eines nicht weiter optimierten Programms. Als beispielhaftes Problem diente ein Würfel mit einer Kantenlänge von zehn Zellen in einem Windkanal der Abmessungen $96 \times 32 \times 32$ Zellen. Zwar handelt es sich nur um eine Simulation und kein Profiling unter Benutzung von Hardware-Zählern, dennoch ist klar zu erkennen, dass Optimierungen in erster Linie an der `collide`-Funktion ansetzen müssen.

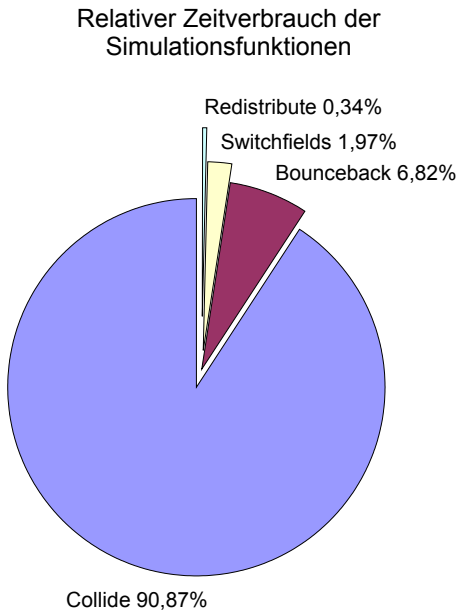


Abbildung 4.1: Relativer Zeitbedarf der Funktionen eines Simulationsschritts

In Gleichung 2.9 sind einige Möglichkeiten offensichtlich, Gleitkomma-Operationen einzusparen: Der Term $\frac{3}{2c^2} < u, u >$ ist innerhalb einer Zelle konstant, kann also zu Beginn der `collide`-Funktion berechnet werden. Auch $\frac{9}{2c^4} < \xi_i, u >^2$ ist für zwei engegengesetzte diskrete Geschwindigkeiten gleich und $\frac{3}{c^2} < \xi_i, u >$ unterscheidet sich nur durch das Vorzeichen.

Viele arithmetische Optimierungen werden bereits von modernen Compilern durchgeführt, beispielsweise wird eine Division durch eine Konstante durch eine Multiplikation mit dem Inversen ersetzt. Optimierungen hingegen, die sich aus Besonderheiten des Modells, wie der Verteilung der Kollokationspunkte, ergeben, müssen weiterhin von Hand durchgeführt werden.

Die durch die in Kapitel 4.1 und 4.2 vorgestellten Optimierungen verbesserte Version der `collide`-Funktion findet sich in Quellcodeausschnitt 4.1. In Hinblick auf Kapitel

5.3 wurde der Zugriff auf die Felder `pdf` und `pdf_cp` bereits durch Makros (`getf` und `getf_cp`) ersetzt, um verschiedene Speicheranordnungen zu ermöglichen.

```
1  void collide(const int x, const int y, const int z) {
2
3    double rho,ux,uy,uz;
4    double temp, temp2;
5
6    rho = getf(x,y,z,C);
7    ux = getf(x,y,z,E)  + getf(x,y,z,NE) + getf(x,y,z,SE);
8    temp = getf(x,y,z,W) + getf(x,y,z,NW) + getf(x,y,z,SW);
9    rho = rho + ux + temp;
10   ux = ux + getf(x,y,z,TE) + getf(x,y,z,BE) -
11        temp - getf(x,y,z,TW) - getf(x,y,z,BW);
12   ...
13   ux /= rho;
14   uy /= rho;
15   uz /= rho;
16
17   double usqr = -1./2.*(ux*ux + uy*uy + uz*uz) + 1./3.;
18   double pre1 = (1. - invtau);
19   double pre2 = invtau*rho;
20
21   getf_cp(x,y,z,C) = pre1*getf(x,y,z,C) + pre2*usqr;
22
23   pre2 /= 6.;
24   temp = 3./2.*ux*ux + usqr;
25   getf_cp(x+1,y,z,E) = pre1*getf(x,y,z,E) + pre2*(temp + ux);
26   getf_cp(x-1,y,z,W) = pre1*getf(x,y,z,W) + pre2*(temp - ux);
27   ...
28
29   pre2 /= 2.;
30   temp = ux+uy;
31   temp2= 3./2.*temp*temp + usqr;
32   getf_cp(x+1,y+1,z,NE) = pre1*getf(x,y,z,NE) +
33                           pre2*(temp2 + temp);
34   getf_cp(x-1,y-1,z,SW) = pre1*getf(x,y,z,SW) +
35                           pre2*(temp2 - temp);
36   ...
37 }
```

Quellcodeausschnitt 4.1: `collide` (optimiert), Kollision und Propagation werden verschmolzen durchgeführt

4.3 Optimierungen durch den Compiler

Der Compiler übersetzt den Quellcode in Maschinenbefehle und hat daher weitreichende Optimierungsmöglichkeiten. Moderne Compiler wenden eine Vielzahl von Optimierungen an, siehe hierzu [Lei95] für eine umfangreiche Beschreibung unterschiedlicher Optimierungstechniken und [D+99] für eine konkrete Übersicht der im Intel IA-64 Compiler automatisch angewendeten Optimierungen. Zusätzlich können dem Compiler mittels Kommandozeilenparametern und Schlüsselwörtern im Quellcode Hinweise, z. B. bezüglich der Zielplattform, gegeben werden.

Im C99-Standard werden zwei neue Schlüsselwörter eingeführt, die dem Compiler das Optimieren erleichtern sollen, siehe Tabelle 4.1.

restrict	kann nur auf Pointer-Typen angewendet werden und garantiert dem Compiler, dass kein *aliasing* stattfindet, es zeigt also kein anderer Zeiger auf diese Speicherstelle
inline	kann nur auf Funktionen angewendet werden und weist den Compiler an, keinen Funktionsaufruf zu machen, sondern den Quellcode dieser Funktion direkt an die Stelle des Aufrufs einzubetten

Tabelle 4.1: Wichtige Compiler-Schlüsselwörter des C99-Standard

Die Auswahl an Compiler-Parametern ist sehr umfangreich, hier soll lediglich eine kleine Auswahl der Parameter für den Intelcompiler 8.0 (Tabelle 4.2) und den GCC 4.0.1 (Tabelle 4.3) gezeigt werden.

-O3	höchste vordefinierte Optimierungsstufe
-xW	generiere speziell auf Intel Pentium 4 abgestimmten Code, beim Start auf anderen Prozessoren können Laufzeitfehler auftreten
-axW	erzeuge normalen IA-32-Code und zusätzlich auf Pentium 4 optimierter Code, zur Laufzeit wird entschieden, welcher Code ausgeführt wird
-ipo	nehme interprozedurale Optimierungen zwischen unterschiedlichen Dateien vor, z. B. *inlining* von Funktionen aus unterschiedlichen Quelldateien oder erweitertes *instruction scheduling*
-Ob2	behandle alle Funktionen als Inline-Funktionen
-static	benutze statische Bibliotheken (anstatt dynamischer)

Tabelle 4.2: Kommandozeilenparameter Intelcompiler

-O3	höchste vordefinierte Optimierungssstufe
-march=pentium4	optimiere speziell auf Pentium 4, es kann auch eine andere Architektur angegeben werden, z. B. Opteron oder AthlonXP
-ftree-vectorize	vektorisiere (einfache) Schleifen durch SSE-Befehle
-fprefetch-loop-arrays	lade Array-Werte, die in Schleifen verwendet werden, im Voraus
-msse[2]	benutze die SSE- oder SSE2-Befehlssatzerweiterungen
-static	benutze statische Bibliotheken (anstatt dynamischer)

Tabelle 4.3: Kommandozeilenparameter GCC

5 Speicher- und Cache-Optimierungen

Die Geschwindigkeit vieler numerischer Programme wird nur durch die Rechengeschwindigkeit des Prozessors beschränkt. Bei der Lattice-Boltzmann-Methode ist dies aufgrund des großen Speicherplatzbedarfs nicht der Fall, denn bereits ein Gitter der Größe 12^3 füllt den Cache eines Pentium 4 vollständig, denn es benötigt 12^3(Anzahl Zellen)·2(Kopierfeld)·19(Werte pro Zelle)·8(Bytes pro Wert) Bytes $\approx$ 512kB Speicher. Ein Feld der Größe 100^3 benötigt bereits beinahe 300 MB Speicher.

Die Geschwindigkeit der Lattice-Boltzmann-Methode wird also zunehmend durch die Speicherbandbreite beschränkt, was durch beispielhafte Berechnungen belegt wird. Ein Grund dafür ist, dass die Hauptspeicher nicht im gleichen Maße schneller geworden sind wie die Prozessoren. Eine Möglichkeit diesem Problem zu begegnen ist der Einsatz von Caches auf Hardware-Seite und die optimalen Nutzung von Speicher und Caches auf Software-Seite.

5.1 Speicher-Bandbreite als Flaschenhals

Um zu demonstrieren, dass die Speicherbandbreite die Geschwindigkeit stärker beeinflusst als die arithmetische Leistung des Prozessors werden zunächst die maximal erreichbaren Geschwindigkeiten berechnet. Dabei wird die Leistung in MLSUPS (*Mega Lattice Site Updates Per Second*) gemessen. Diese Einheit hat sich im Umfeld der Lattice-Boltzmann-Methode durchgesetzt, da sie unabhängig von konkreten Architekturen und Implementierungen ist.

Die maximal erreichbare Leistung abhängig vom Prozessor ergibt sich wie folgt:

$$\text{maximale Leistung (Prozessor)} = \frac{\text{Gleitkommaleistung des Prozessors}}{\text{Anzahl Gleitkommaoperationen pro Zelle}}$$

Plattform 2 (siehe A.1.2) hat eine Taktfrequenz von 3,066 GHz und verarbeitet im optimalen Fall 2 Gleitkommaoperationen pro Takt (durch SSE). Eine `collide`-Operation benötigt theoretisch 156 Gleitkommaoperationen. Davon ausgegangen, dass nur Fluid-Zellen betrachtet werden und sämtliche Gleitkommaoperationen optimal parallel, gleich schnell und ohne Abhängigkeiten ausgeführt werden können, erhält man:

$$\text{maximale Leistung (Prozessor)} = \frac{2 \cdot 3,066 \cdot 10^9}{156} \cdot 10^{-6}\text{MLSUPS} = 39,3 \text{ MLSUPS}$$

Die maximal erreichbare Leistung abhängig von der Speicherbandbreite berechnet sich zu:

$$\text{maximale Leistung (Speicher)} = \frac{\text{Speicherbandbreite}}{\text{Speicherzugriffe pro Zelle}}$$

Der maximale Speicherdurchsatz von Plattform 2 beträgt $4,3\frac{\text{GB}}{\text{s}}$. Für eine `collide`-Operation müssen zunächst alle 19 Werte geladen werden und danach (an andere Stellen)

zurückgeschrieben werden, es werden also $19 \cdot 2 \cdot 8 = 304$ Bytes von oder zum Speicher transferiert. Angenommen, Bytes könnten einzeln von und zum Speicher transferiert werden, so ergibt sich:

$$\text{maximale Leistung (Speicher)} = \frac{4,3 \cdot 10^9}{304} \cdot 10^{-6}\text{MLSUPS} = 14,1 \text{ MLSUPS}$$

Natürlich sind die getroffenen Annahmen für moderne Architekturen nicht realistisch und viel zu optimistisch, dennoch zeigt das Verhältnis der beiden maximalen Leistungen deutlich, dass die Speicherbandbreite der limitierende Faktor ist.

Anschaulich kann die Abhängigkeit der maximalen Leistung von der Speicherbandbreite an Mehrprozessor-Rechnern gezeigt werden, bei denen sich die Prozessoren einen Speicherbus teilen müssen. Dies ist bei Testplattform 2 (Anhang A.1.2) der Fall. Hier bricht die Leistung stark ein, wenn mehrere Prozesse gleichzeitig gestartet werden, obwohl für jeden Prozess ein eigener Prozessor zur Verfügung steht (siehe Kapitel 7.2). Dies sollte ebenso bei modernen Dual-Core Prozessoren der Fall sein, sofern diese nur über einen Speichercontroller verfügen.

5.2 Die Cache-Hierarchie

Moderne Architekturen besitzen nicht nur den großen aber verhältnismäßig langsamen Hauptspeicher, sondern dazwischengeschaltet ist eine typischerweise zwei- oder dreistufige Hierarchie von schnellen Pufferspeichern, den sogenannten *Caches*. Der Zweck von Caches ist ausschließlich die Verringerung der Latenz des Speichers.

Die Stufen der Hierarchie (*cache levels*) werden von den Registern des Prozessors bis zum Hauptspeicher immer langsamer, aber auch größer, für eine schematische Darstellung siehe Abbildung 5.1.

Findet ein Speicherzugriff statt, so werden zuerst der Reihe nach die Caches abgefragt, ob diese die gesuchte Speicherstelle beinhalten. Sollte dies nicht der Fall sein, so spricht man von einem *cache miss* und es wird ein Block aus dem Speicher in einen der Caches geladen. Dieser Block besitzt eine feste Größe (auf gängigen Architekturen 64 Byte) und auch wenn nur 1 Byte gelesen werden soll, so wird dennoch ein ganzer Block,eine sogenannte Cache-Line, geladen.

Caches unterscheiden sich unter anderem in den Strategien, wo eine Cache-Line im Cache abgelegt werden kann und wie das Schreiben von Daten stattfindet.

Falls eine Speicherstelle nur an genau einer Stelle im Cache abgelegt werden kann, so spricht man von einem direkt abgebildeten Cache. Da der Cache viel kleiner ist als der Hauptspeicher, werden viele Speicherstellen auf die gleiche Stelle des Caches abgebildet. Falls eine Speicherstelle an jede beliebige Stelle im Cache geschrieben werden kann, so spricht man von einem voll-assoziativen Cache. Die Vor- und Nachteile sind offensichtlich: Bei einem direkt abgebildeten Cache werden oft Verdrängungen stattfinden, obwohl noch Platz im Cache wäre, ein voll-assoziativer Cache muss bei jedem Zugriff vollständig durchsucht werden.

Deswegen benutzen die meisten aktuellen Architekturen einen Mittelweg, die sogenannten n-fach assoziativen Caches. Dabei kann jede Speicherstelle auf eine von n Stellen im

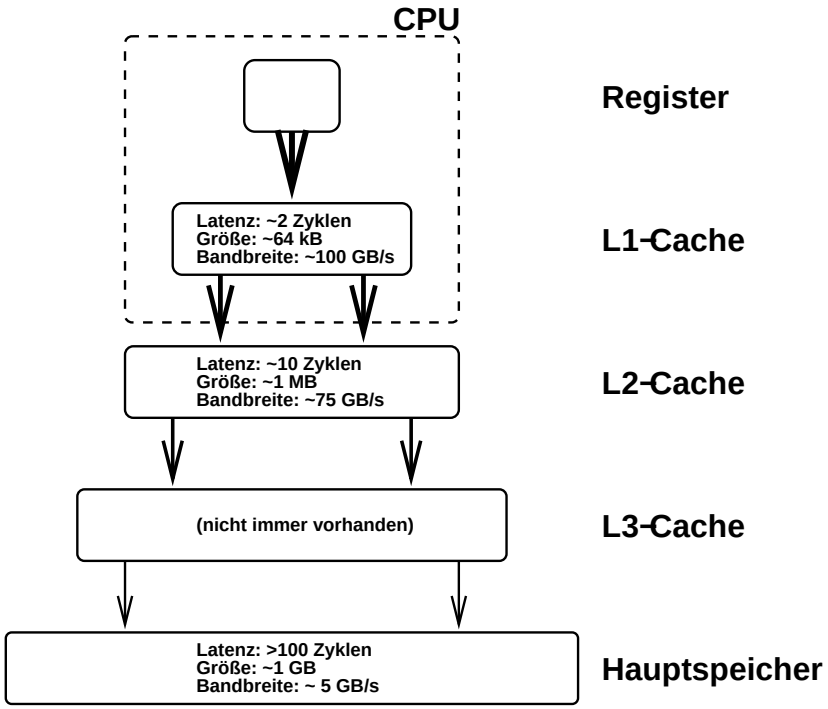


Abbildung 5.1: Schematische Darstellung der unterschiedlichen Caches, zu beachten ist, dass mittlerweile auch L2-Caches oftmals in den Prozessor integriert werden.

Cache geschrieben werden, wobei wieder viele Speicherstellen auf die gleichen n Stellen abgebildet werden. Ein Pentium 4 beispielsweise besitzt einen 8-fach assoziativen L2-Cache.

Um nun Daten zu Schreiben gibt es auch zwei grundsätzliche Möglichkeiten. Entweder die Daten werden direkt in den Hauptspeicher geschrieben (*write through*), wobei die Caches auf den neuen Stand gebracht werden, oder nur der Cache wird geschrieben und erst beim Auslagern einer Cache-Line wird der Hauptspeicher aktualisiert (*write back*). Die meisten aktuellen Architekturen verwenden eine write back Strategie. Dies hat jedoch zur Folge, dass zum Schreiben einer Speicherstelle diese zuerst in den Cache geladen werden muss.

Als Anmerkung sei erwähnt, dass die SSE-Erweiterungen Befehle beinhalten, die ein direktes Schreiben in den Hauptspeicher ermöglichen, ohne die Caches zu füllen. Man nennt dies ein *streaming store*, siehe Kapitel 6.1. Sofern die zu schreibenden Daten bereits im Cache sind, so werden sie auf den neuen Stand gebracht und in den Hauptspeicher geschrieben. Dies entspricht der write through Strategie.

Diese Ausführungen sollen hier genügen, für weitere Informationen zu Caches sei auf spezielle Literatur zur Rechner-Architektur verwiesen, beispielsweise [HP02].

Für Programmierer von Hochsprachen sind Caches normalerweise vollkommen transparent, es kann weder Speicher in einem Cache angefordert werden, noch können Daten aus dem Cache explizit gelöscht werden. Intrinsische Funktionen bieten jedoch die Möglichkeit, direkt die Caches zu manipulieren.

Es existieren aber auch indirekte Methoden, die Nutzung von Caches zu verbessern. Einerseits kann man die Speicherposition von gemeinsam benutzten Variablen ändern, so dass diese dicht im Speicher liegen und auf einmal in den Cache geladen werden. Man spricht dabei von räumlicher Lokalität (*spatial locality*). Andererseits kann man den Zugriff auf Variablen so gestalten, dass alle Operationen auf einer Variablen in möglichst kurzer Zeit stattfinden, so dass die Wahrscheinlichkeit eines Verdrängens aus dem Cache geringer ist. Hierbei spricht man von zeitlicher Lokalität (*temporal locality*).

Im Folgenden werden Methoden vorgestellt, um sowohl die räumliche als auch die zeitliche Lokalität zu erhöhen.

5.3 Wahl des Speicherlayouts

Die Wahl des Speicherlayouts für das Feld, in dem die Werte der Verteilungsfunktion f gespeichert werden, beeinflusst vor allem die räumliche Lokalität. Grundsätzlich kann man die Verteilungsfunktion als vier-dimensionales Feld sehen: Drei Raumdimensionen und zusätzlich eine Dimension, die die 19 unterschiedlichen Werte an den Kollokationspunkten speichert. Damit ergeben sich vier Möglichkeiten des Speicherlayouts (sofern man nicht die Raumdimensionen zusätzlich permutiert), wobei zuerst die beiden naheliegenden Varianten mit einer graphischen Interpretation erläutert werden. Die theoretischen Überlegungen zu Vor- und Nachteilen der verschiedenen Speicherlayouts werden ausführlicher in [Don04] behandelt.

Speicherlayout Array of Structures

Die naheliegendste Anordnung ist es, ein dreidimensionales Feld anzulegen und an jeder Position die 19 zu der Zelle mit den passenden Koordinaten gehörigen Werte abzuspeichern. Man kann also von einem Array of Structures sprechen. Die Darstellung in der Feld-Schreibweise von C ist `pdf[xdim][ydim][zdim][i]`.

Da in C der letzte Index eines Feldes dicht im Speicher liegt, liest die `collide`-Funktion also nur Werte, die dicht beeinander im Speicher liegen. In einem Aufruf von `collide` werden alle 19 Werte einer Zelle zum Lesen benötigt. Davon ausgegangen, dass auf einer heutigen Architektur eine Cache-Line eine Länge von 64 Byte hat und ein `double` 8 Byte benötigt, so müssen im Mittel 2,4 Cache-Lines pro Aufruf von `collide` geladen werden.

Zum Zurückschreiben müssen jedoch Werte in die im Gitter anliegenden 19 Zellen in das Kopierfeld geschrieben werden. Dazu müssen 19 unterschiedliche Speicherstellen aus dem Kopierfeld zunächst in den Cache geladen werden, um die berechneten Werte zu speichern. Keine dieser Speicherstellen sind aneinanderliegend, im schlimmsten Fall muss also für jeden zu schreibenden Wert eine Cache-Line geladen werden. Außerdem ist abhängig von der Größe des Feldes, welche der geladenen Cache-Lines verdrängt werden oder im nächsten Aufruf von `collide` wiederverwendet werden können.

Da für eine Berechnung mit SSE-Operationen, speziell mit gepackten Operationen (Kapitel 6.2), die Eingabedaten dicht im Speicher liegen müssen, könnte bei diesem Layout auch eine automatische Vektorisierung des Compilers erfolgen.

Abschließend ist zu bemerken, dass eine anliegende Zelle in x-Richtung einen anderen Abstand hat als eine anliegende Zelle in z-Richtung. Dies ergibt für das Zurückschreiben ein unregelmäßiges Zugriffsmuster und macht es dem Prozessor schwer, bereits frühzeitig ein Prefetching, ein vorzeitiges Laden später benötigter Speicherstellen, vorzunehmen. Dieses Problem ist jedoch allen Speicherlayouts gemeinsam.

Speicherlayout Structure of Arrays

Die zweite einsichtige Möglichkeit ist es, 19 Felder anzulegen, die jeweils eine diskrete Geschwindigkeit für alle Zellen speichern, als C-Feld also `pdf[i][xdim][ydim][zdim]`. Hier kann man von einer Structure of Arrays sprechen.

Davon ausgegangen, dass der Cache groß genug ist und keine Verdrängung stattfindet, bedeutet das also, dass zum Lesen der Werte nach $\frac{\text{Länge einer cache line}}{\text{Länge eines double}}$ Ausführungen von `collide` ein cache miss pro diskrete Geschwindigkeit stattfindet, sofern die Schleifen der Simulation in der Schachtelung über das Feld laufen, wie es die Indizierung des Feldes vorgibt (hier müsste also die innerste Schleife über z laufen). Diese Überlegungen gelten auch für die Schreibzugriffe.

Der theoretische Vorteil dieses Speicherlayouts ist es, dass keine Daten umsonst geladen werden, sondern alle geladenen Cache-Lines für die folgenden Zellen benötigt werden.

Alternative Speicherlayouts

Nach den beiden vorgeführten Varianten mit einfachen graphischen Deutungen liegt es nahe, auch Speicherlayouts zu benutzen, bei denen die Dimension `i` an anderer Stelle des Feldes liegt, also das Feld `pdf[xdim][ydim][i][zdim]` (Alternative 1) oder `pdf[xdim][i][ydim][zdim]` (Alternative 2) anzuordnen.

Diese beiden Layouts beheben eine Schwäche des Structure of Arrays Layouts, wobei sie die Stärke der guten Cache-Nutzung beibehalten sollen. Die Schwäche ist es, dass die Geschwindigkeiten einer Zelle beim Structure of Arrays Layout im Speicher weitestmöglich auseinander liegen. Damit ist die Gefahr größer, dass bei assoziativen Caches die verschiedenen Speicherstellen auf die gleiche Cache-Line gespeichert werden, die Daten sich also gegenseitig verdrängen.

5.4 Compressed-Grid Methode

Wie in Kapitel 2.2 bereits beschrieben, gestaltet sich die Implementierung der Lattice-Boltzmann-Methode unter Verwendung von zwei vollständigen Gittern recht einfach, man nennt dies auch die Two-Grid Methode. Dies hat verschiedene Nachteile. Zunächst ist klar, dass dafür doppelt soviel Speicher benötigt wird. Andererseits wird aber damit auch mehr Cache-Speicher benötigt und die beiden Gitter sind im Speicher zumeist weit voneinander entfernt.

Da im Propagationsschritt Werte in allen benachbarten Zellen überschrieben werden, die im Kollisionsschritt dieser Zellen noch erforderlich sind, kann nicht ohne Weiteres auf

eine Möglichkeit zum Zwischenspeichern der Werte verzichtet werden.

Dazu genügt jedoch ein Gitter, sofern dieses geeignet vergrößert wird. Die Vorstellung dabei ist, dass beide Gitter gegeneinander verschoben in ein Gitter eingebettet werden, daher die Bezeichnung Compressed-Grid. Für das D3Q19-Modell vergrößert man das Gitter um eine Schicht Zellen in Richtung B (bottom), E (east) und S (south) (siehe Abbildung 2.1).

Beginnt ein Simulationsschritt bei Zelle $(0,0,0)^T$, so können die berechneten Werte um $(-1,-1,-1)^T$ verschoben in die neu angefügten Zellschichten geschrieben werden. Wenn die Zellen weiter verarbeitet werden, so werden durch die Verschiebung um $(-1,-1,-1)^T$ nur Werte von Zellen überschrieben, die bereits verarbeitet sind und deswegen nicht mehr benötigt werden.

Nach Abschluss des Simulationsschrittes ist das gesamte Simulationsgebiet im Gitter um $(-1,-1,-1)^T$ verschoben. Nun sind in den Richtungen T (top), W (west) und N (north) je eine nicht benutzte Schicht Zellen. Im nächsten Simulationsschritt wird also sozusagen am anderen Ende des Simulationsgebiets begonnen, die Zellen werden in entgegengesetzter Richtung verarbeitet und die Werte werden verschoben in Richtung $(1,1,1)^T$ zurückgespeichert.

Dieses Vorgehen ist möglich, da das D3Q19-Modell keine diskrete Geschwindigkeit mit der Richtung $(1,1,1)^T$ kennt. Somit existieren keine Abhängigkeiten zwischen zwei Zellen $(x,y,z)^T$ und $(x-1,y-1,z-1)^T$. Für das D3Q27-Modell beispielsweise muss eine andere Verschiebung gewählt werden, möglich wäre $(2,1,1)^T$.

Das Vorgehen ist in Abbildung 5.2 für das D2Q9-Modell dargestellt. Dabei wurden zwei Schichten Zellen in Richtung W (west) und eine Schicht in Richtung S (south) angefügt und eine Verschiebung von $(-2,-1)^T$ gewählt. Das tatsächliche Simulationsgebiet ist blau eingefärbt.

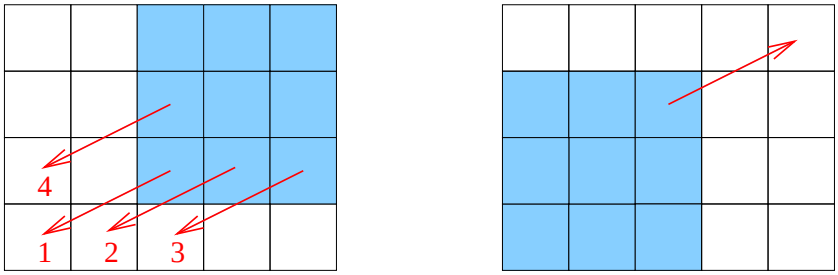


Abbildung 5.2: Verschobene Propagation bei der Compressed-Grid Methode, im ersten Zeitschritt wird nach unten links verschoben, im nächsten Zeitschritt nach oben rechts

In der Implementierung muss also zusätzlich die Position des Simulationsgebiets im gesamten Gitter verwaltet werden, außerdem müssen die Schleifen modifiziert werden, so dass sie in die jeweils passende Richtung laufen. Dafür benötigt nun ein Gitter der Größe 100^3 nicht mehr 308 MB (inklusive Halo-Zellen), sondern nur noch 158 MB (ebenfalls inklusive Halo-Zellen).

5.5 3D-Loop-Blocking

Zuletzt kann bei der Lattice-Boltzmann-Methode noch eine klassische Technik zur Verbesserung der temporalen Lokalität eingesetzt werden, die beispielsweise auch bei der Matrix-Matrix-Multiplikation verwendet wird. Viele Compiler (auch der Intel IA-64 Compiler [Int04b]) nehmen diese Optimierung bereits selbstständig vor.

Die Rede ist vom Loop-Blocking (im deutschen auch Schleifenblockung, dies ist aber ungebräuchlich, deswegen wird im Weiteren die englische Bezeichnung verwendet). Damit bezeichnet man das Zerlegen von mehreren geschachtelten Schleifen in Blöcke, wie in Quellcodeausschnitt 5.1 am Beispiel der Matrix-Matrix-Multiplikation dargestellt, wobei gleichzeitig eine Vertauschung der Schleifen durchgeführt wurde. Natürlich müssen vor dem Blocken eventuelle Abhängigkeiten untersucht werden.

```
1  /* Definitionen: */
2  double a[1000][1000], b[1000][1000], c[1000][1000];
3
4  /* Ungeblockte Version: */
5  for(i=0;i<1000;i++)
6      for(j=0;j<1000;j++)
7          for(k=0;k<1000;k++)
8              c[i][j] = c[i][j] + a[i][k] * b[k][j];
9
10 /* Geblockte Version: */
11 for(jj=0;jj<1000;jj+=blocksize)
12   for(kk=0;kk<1000;kk+=blocksize)
13     for(i=0;i<1000;i++)
14       for(j=jj;j<jj+blocksize;j++)
15         for(k=kk;k<kk+blocksize;k++)
16           c[i][j]=c[i][j]+a[i][k]*b[k][j];
```

Quellcodeausschnitt 5.1: Loop-Blocking bei der Matrix-Matrix-Multiplikation

Der Zweck des Loop-Blocking ist es, durch das Anpassen der Blockgröße sowohl die zeitliche als auch die räumliche Lokalität zu erhöhen, wobei die Blockgröße einerseits so klein gewählt wird, dass die Blöcke noch in den Cache passen, andererseits so groß, dass der Aufwand der zusätzlichen Schleifen verschwindet. In der geblockten Matrix-Matrix-Multiplikation aus Quellcodeausschnitt 5.1 ist durch das Blocken beispielsweise `a[i][k]` bezüglich der `j`-Schleife wiederverwendbar, solange es im Cache bleibt.

Bei der Lattice-Boltzmann-Methode erwies sich in [Don04] ein 3D-Loop-Blocking als vielversprechend. 3D-Blocking bedeutet, dass alle drei Schleifen (also die Schleifen über die Raumdimensionen) in der Simulationsroutine geblockt werden, das Simulationsgebiet wird also sozusagen in Würfel aufgeteilt und diese werden nacheinander abgearbeitet, wie in Abbildung 5.3 für 2D dargestellt.

Bei der Lattice-Boltzmann-Methode wäre auch ein Blocken der Zeit möglich, dies wird in einigen Arbeiten als "4-way-blocking"([Hau05]) bezeichnet. Dadurch befinden sich unterschiedliche Zellen des Simulationsgitters in unterschiedlichen Zeitschritten. Deshalb müssen zum Auswerten der Simulation zu gewissen Zeitpunkten alle Zellen in den gleichen

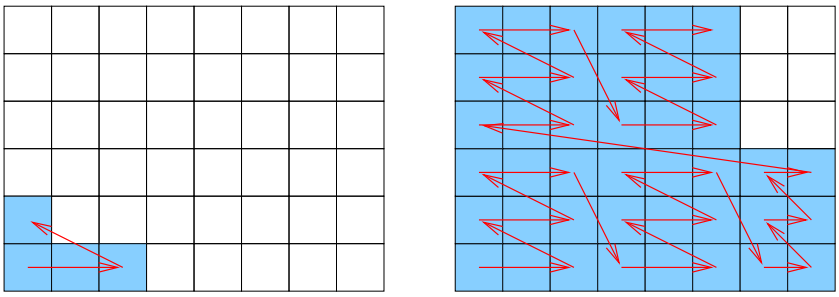


Abbildung 5.3: 2D-Loop-Blocking bei der Lattice-Boltzmann-Methode

Zeitschritt gebracht werden. Außerdem wird ein Austausch von Zwischenergebnissen mit anderen Simulationen und damit eine Kopplung sehr schwierig, da auch hier alle Zellen im selben Zeitschritt sein müssten.

6 Optimierung mit intrinsischen Funktionen

6.1 Beeinflussen des Caching-Verhaltens

Wie bereits in Kapitel 5 gezeigt, wird die Berechnungsgeschwindigkeit bei der Lattice-Boltzmann-Methode in zunehmendem Maße von der Geschwindigkeit des Speicherzugriffs bestimmt. Um die Latenz von Speicherzugriffen zu verringern, besitzen moderne Architekturen Mechanismen, um Speicherzugriffe vorherzusagen und Daten vorab zu laden, man spricht von *Prefetching*.

Ein Pentium 4 Prozessor beispielsweise lädt Speicherstellen im passenden Abstand voraus, sobald zwei Cache-Misses mit gleichem Abstand registriert wurden. Die Verarbeitung von großen Strukturen in Arrays kann damit deutlich beschleuningt werden. Da bei der Lattice-Boltzmann-Methode bei der Verarbeitung jeder Zelle jedoch Zugriffe auf alle benachbarten Zellen nötig sind, ist ein automatisches Prefetching schwierig, da bei keinem Speicherlayout alle benachbarten Zellen in gleichem Abstand im Speicher sind.

Um dieses Problem manuell angehen zu können, wurden mit SSE und SSE2 intrinsische Funktionen eingeführt, um die Caches zu manipulieren. Eine Übersicht der wichtigen Funktionen zu diesem Zweck siehe Tabelle 6.1.

Die Funktion `_mm_stream_pd` kann verwendet werden, falls lange Felder existieren, die nur einmal geschrieben werden. In diesem Fall kann es sich lohnen, die Schreiboperationen direkt in den Hauptspeicher durchzuführen, da so mehr Cache-Speicher für andere Daten zur Verfügung steht. Falls jedoch die Daten auch gelesen oder mehrfach geschrieben werden, wobei eine Wiederbenutzung des Caches möglich scheint, so sollte diese Operation nicht verwendet werden. Eine Verwendung in den Berechnungen der Lattice-Boltzmann-Methode scheint deswegen nicht sinnvoll.

`void _mm_prefetch(char *p , int i )`	Lade die Cache-Line, die die Adresse `*p` beinhaltet, `i` gibt einen Hinweis auf die Cache-Ebene und es werden die Konstanten `_MM_HINT_T0` (lade in alle Caches), `_MM_HINT_T1` (lade in L2- und L3-Cache), `_MM_HINT_T2` (lade in L3-Cache) und `_MM_HINT_NTA` (lade nur in L1-Cache) erwartet
`void _mm_stream_pd(double *p, __m128d a)`	schreibe `a` direkt in den Hauptspeicher zurück, ohne zunächst in den Cache auszulagern
`void _mm_clflush(void const *p)`	lösche die Cacheline, die die Adresse `*p` beinhaltet

Tabelle 6.1: Intrinsische Funktionen zur Manipulation der Caches (Auswahl)

Mithilfe von `_mm_prefetch` können also von Hand Daten in die Caches geladen werden. Für eine Ausführung von `collide` wird offensichtlich die gerade betrachtete Zelle im Cache benötigt, da aber alle betrachteten Plattformen die Schreibstrategie write back benutzen, werden zusätzlich alle benachbarten Zellen im Cache benötigt, da in diese geschrieben wird.

Betrachtet man zunächst das Array of Structures Speicherlayout, so sind die Werte einer Zelle dicht im Speicher, es genügt also, nur so viele Cache-Lines zu laden, dass die 19 aufeinanderfolgenden Werte der Zelle in den Cache transportiert werden. Zum Laden der benachbarten Zellen gibt es jedoch keine Alternative, als alle benötigten Werte explizit anzugeben, da diese verstreut im Speicher sind.

Sofern die Schleife über `x` als innerste durchlaufen wird, kann man wie im Makro in Quellcodeausschnitt 6.1 ausgeführt, die Werte laden, die in `DIST` Schleifendurchläufen benötigt werden. Die ersten drei Zeilen sind dabei die Werte der gerade zu bearbeitenden Zelle, die restlichen Zeilen sind die Stellen, in die nach der Propagation geschrieben werden muss. Der optimale Abstand `DIST` zu den vorab geladenen Zellen kann nun experimentell auf die vorliegende Architektur angepasst werden.

```
1  #define DOPREFETCH \
2    _mm_prefetch((char *) &getf(x+DIST,y,z,0), HINT); \
3    _mm_prefetch((char *) &getf(x+DIST,y,z,8), HINT); \
4    _mm_prefetch((char *) &getf(x+DIST,y,z,16), HINT); \
5    _mm_prefetch((char *) &getf_cp(x+DIST,y,z,C), HINT); \
6    _mm_prefetch((char *) &getf_cp(x+DIST+1,y+1,z,NE), HINT); \
7    _mm_prefetch((char *) &getf_cp(x+DIST+1,y-1,z,SE), HINT); \
8    _mm_prefetch((char *) &getf_cp(x+DIST+1,y,z+1,TE), HINT); \
9    _mm_prefetch((char *) &getf_cp(x+DIST+1,y,z-1,BE), HINT); \
10   _mm_prefetch((char *) &getf_cp(x+DIST,y+1,z,N), HINT); \
11   _mm_prefetch((char *) &getf_cp(x+DIST,y,z+1,T), HINT); \
12   _mm_prefetch((char *) &getf_cp(x+DIST,y+1,z+1,TN), HINT); \
13   _mm_prefetch((char *) &getf_cp(x+DIST,y-1,z,S), HINT); \
14   _mm_prefetch((char *) &getf_cp(x+DIST,y,z-1,B), HINT); \
15   _mm_prefetch((char *) &getf_cp(x+DIST,y-1,z-1,BS), HINT); \
16   _mm_prefetch((char *) &getf_cp(x+DIST,y-1,z+1,TS), HINT); \
17   _mm_prefetch((char *) &getf_cp(x+DIST,y+1,z-1,BN), HINT);
```

Quellcodeausschnitt 6.1: Makro für Prefetching bei Array of Structures Speicherlayout, DIST ist der Abstand der zu ladenden Zelle, HINT gibt an, in welchen Cache geladen werden soll

Bei den anderen Speicherlayouts ist eine einzelne Zelle nicht mehr dicht im Speicher, es müssen also alle Werte einer Zelle in den Cache transportiert werden. Dieses muss jedoch nicht mehr in jedem Schritt geschehen, da mit dem Laden einer Cache-Line auch die folgenden Zellen in den Cache transportiert werden. Andererseits führt eine Berücksichtigung dieser Tatsache zu zusätzlichen Sprüngen, was wiederum Zeit kostet.

6.2 Verwenden von gepackten Operationen

Zur Verwendung von gepackten Operationen (*packed operations*) bei der Lattice-Boltzmann-Methode bieten sich zwei Vorgehensweisen an: Das Zusammenfassen von Rechenoperationen verschiedener Raumrichtungen innerhalb der `collide`-Funktion und das Erstellen einer anderen `collide`-Funktion, in der mehrere Zellen gleichzeitig kollidiert werden können. Die Voraussetzungen, Vor- und Nachteile dieser beiden Möglichkeiten werden in den folgenden Abschnitten betrachtet.

6.2.1 Simultane Berechnung entgegengesetzter diskreter Geschwindigkeiten

Zunächst ist zu sagen, dass für die optimale Nutzung von gepackten Operationen die Daten an 16 Byte Grenzen ausgerichtet sein müssen und die gemeinsam zu verarbeitenden Daten dicht im Speicher liegen müssen, so dass sie direkt in ein SSE-Register geladen werden können. Dies bedeutet, dass nur das Array of Structures Speicherlayout für diese Verwendung von gepackten Operationen in Frage kommt, denn nur hier liegen die zu den unterschiedlichen diskreten Geschwindigkeiten gehörigen Werte einer Zelle dicht im Speicher.

Da eine Zelle aus 19 `double`-Werten besteht, würde jede zweite Zelle nicht an einer 16 Byte Grenze beginnen. Deswegen wird eine Zelle auf 20 `double`-Werte aufgefüllt (*padding*), das resultierende Speicherlayout ist in Abbildung 6.1 dargestellt.

Bei der Betrachtung von Gleichung 2.9 fällt auf, dass sich die lokale Gleichgewichtsverteilung für zwei entgegengesetzte Raumrichtungen nur an einer Stelle um ein Vorzeichen unterscheidet, denn beispielsweise gilt:

$$
\begin{aligned}
f_N^{eq}(x,t) &= \rho w_N \left(1 + 3 < \xi_N, u > + \frac{9}{2} < \xi_N, u >^2 - \frac{3}{2} < u, u >\right) \\
&= \rho w_S \left(1 + 3 < -\xi_S, u > + \frac{9}{2} < -\xi_S, u >^2 - \frac{3}{2} < u, u >\right) \\
&= \rho w_S \left(1 - 3 < \xi_S, u > + \frac{9}{2} < \xi_S, u >^2 - \frac{3}{2} < u, u >\right) \\
&= \rho w_S \left(1 + 3 < \xi_S, -u > + \frac{9}{2} < \xi_S, -u >^2 - \frac{3}{2} < -u, -u >\right) \\
f_S^{eq}(x,t) &= \rho w_S \left(1 + 3 < \xi_S, u > + \frac{9}{2} < \xi_S, u >^2 - \frac{3}{2} < u, u >\right)
\end{aligned}
$$

Zeile drei und fünf unterscheiden sich wie gesagt nur an einer Stelle um ein Vorzeichen. Die weitere Umformung in Zeile vier unterscheidet sich von Zeile fünf nur noch darin, dass u an allen Stellen durch $-u$ ersetzt ist. Man kann also in drei SSE-Registern komponentenweise $-u$ und u abspeichern und dann die Gleichgewichtsverteilung für die jeweils entgegengesetzten diskreten Geschwindigkeiten gleichzeitig berechnen.

Die Vorteile dieses Vorgehens sind klar: Es werden deutlich weniger Rechenoperationen benötigt. Jedoch kann der volle Vorteil der gepackten Operationen nicht ausgespielt wer-

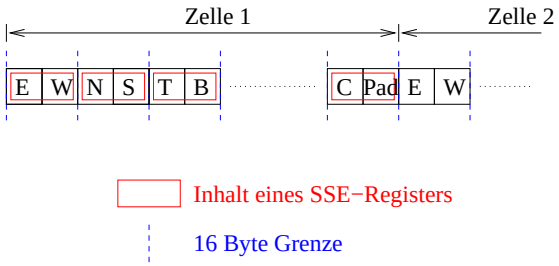


Abbildung 6.1: Speicherlayout zur simultanen Kollision mehrerer Raumrichtungen

den, da die Ergebnisse, die gepackt in einem SSE-Register vorliegen, an unterschiedlichen Stellen gespeichert werden müssen. Außerdem ist jeweils eine der beiden Speicherstellen nicht an einer 16 Byte Grenze ausgerichtet.

Das Speichern eines einzelnen Wertes aus einem `__m128d` kann auf zwei unterschiedliche Arten geschehen: Einerseits durch `_mm_store_sd()`, das den niederwertigen `double`-Wert speichert, in Verbindung mit Shuffling, um den richtigen Wert an die niederwertige Position zu bringen, andererseits durch die Befehle `_mm_storel_pd()` und `_mm_storeh_pd()`, die direkt den nieder- bzw. höherwertigen `double`-Wert speichern. Erstere Methode findet sich im Quellcodeausschnitt 6.2, da sie interessanterweise (wenn auch nur minimal) schneller ist.

```
1  __m128d EW, NS, TB, NESW, NWSE, TEBW, TWBE, TNBS, TSBN, CPAD;
2  __m128d rho, ux, uy, uz;
3  [...]
4
5  EW    = _mm_load_pd(&getf(x,y,z,E));   //beinhaltet nun E und W
6  [...]
7
8  /* Auskommentiert jeweils die korrespondierenden Zeilen der
9     konventionellen Version zum Vergleich
10    c1_6 und c3_2 sind Konstanten
11    pre1 und pre2 sind  bereits berechnete Vorfaktoren */
12 //pre2 /= 6.;
13 pre2 = _mm_mul_pd(pre2, c1_6);
14 //temp = 3./2.*ux*ux + usqr;
15 reg = _mm_add_pd(_mm_mul_pd(_mm_mul_pd(c3_2, ux), ux) , usqr);
16 //getf_cp(x+1,y,z,E) = pre1*getf(x,y,z,E) + pre2*(temp + ux);
17 //getf_cp(x-1,y,z,W) = pre1*getf(x,y,z,W) + pre2*(temp - ux);
18 reg2 = _mm_add_pd(_mm_mul_pd(pre1, EW),
19        _mm_mul_pd(pre2, _mm_add_pd(reg, ux)));
20 _mm_store_sd(&getf_cp(x+1,y,z,E), reg2);
21 _mm_store_sd(&getf_cp(x-1,y,z,W),
22                _mm_shuffle_pd(reg2, reg2,_MM_SHUFFLE2(0,1)));
23 [...]
```

Quellcodeausschnitt 6.2: Zusammengefasste Berechnung der Werte für E und W

6.2.2 Simultane Kollision mehrerer Zellen

Wie im vorigen Kapitel beschrieben, liegen beim Array of Structures Speicherlayout die verschiedenen Geschwindigkeitswerte einer Zelle dicht im Speicher, so dass diese geschickt durch gepackte Operationen verarbeitet werden können. Andererseits liegen bei allen anderen Speicherlayouts die gleichen diskreten Geschwindigkeiten verschiedener Zellen dicht im Speicher. Dies bedeutet, dass beispielsweise die E-Komponenten zweier aufeinanderfolgender Zellen mit einem Maschinenbefehl in ein SSE-Register geladen werden können. Auch hier sollten die Speicherzugriffe ausgerichtet auf 16 Byte Grenzen erfolgen, Graphik 6.2 verdeutlicht das verwendete Speicherlayout.

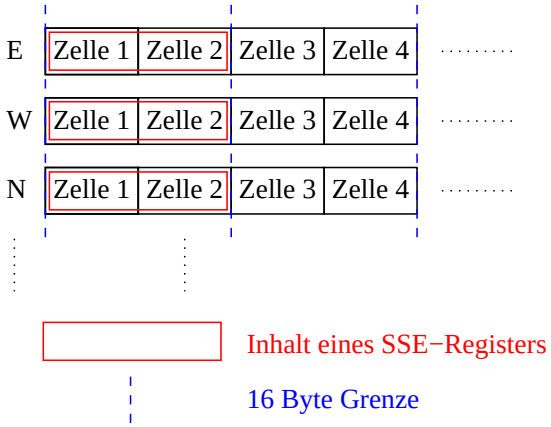


Abbildung 6.2: Speicherlayout zur simultanen Kollision mehrerer Zellen

Anstatt also die Rechenoperationen innerhalb der `collide`-Funktion durch intrinsische Funktionen zu ersetzen, können dadurch zwei Zellen gleichzeitig kollidiert werden, indem zwei Schleifendurchläufe der innersten Schleife (die Schleife über `x` in Quellcodeausschnitt 6.3) zusammengefasst werden.

```
1  for (coord z=1; z<dimz+1;++z) {
2    for (coord y=1; y<dimy+1;++y) {
3      for (coord x=1; x<dimx+1;x+=2) {
4        if ((gettype(x,y,z) == fluid)
5            && (gettype(x+1,y,z) == fluid)) {
6          collide2(x,y,z);
7        } else {
8          /* ... */
9        }
10     }
11   }
12 }
```

Quellcodeausschnitt 6.3: Modifikation der Simulationsschleife zur simultanen Kollision mehrerer Zellen

Quellcodebeispiel 6.3 illustriert das Vorgehen: In der innersten Schleife wird in Zweier-

schritten vorangegangen. Sofern es sich um zwei aufeinanderfolgende Fluid-Zellen handelt, wird die Routine `collide2` aufgerufen, die beide Zellen auf einmal kollidiert und propagiert. Wie bereits gesagt, ist dies bei allen Speicherlayouts möglich, bei denen die gleichen Geschwindigkeitskomponenten der gemeinsam kollidierten Zellen dicht im Speicher liegen, also Structure of Arrays, Alternative 1 und Alternative 2 (siehe Kapitel 5.3).

Tatsächlich ist die Funktion `collide2` nichts anderes als `collide`, wobei alle Operationen durch die entsprechenden SSE2-Operationen ersetzt worden sind, siehe dazu Quellcodeausschnitt 6.4 (dabei ist zu beachten, dass die SSE2-Berechnungen als normale Funktionsaufrufe gewissermaßen in Präfixnotation stehen).

```
1  static inline void collide2(const coord x, const coord y,
2                              const coord z) {
3
4    [...]
5
6    __m128d rho  = [...]
7    __m128d ux   = [...]
8    __m128d uy   = [...]
9    __m128d uz   = [...]
10   __m128d usqr = [...]
11   __m128d pre1 = [...]
12   __m128d pre2 = [...]
13   __m128d temp = [...]
14
15
16   /* Originalzeile aus collide zum Vergleich:
17      getf_cp(x+1,y,z,E) = pre1*getf(x,y,z,E)+pre2*(temp + ux);*/
18   _mm_storeu_pd(&(getf_cp(x+1,y,z,E)),
19       _mm_add_pd(_mm_mul_pd(pre1, _mm_load_pd(&getf(x,y,z,E))),
20                  _mm_mul_pd(pre2, _mm_add_pd(temp, ux))));
21
22   /* getf_cp(x-1,y,z,W) = pre1*getf(x,y,z,W)+pre2*(temp - ux);*/
23   _mm_storeu_pd(&(getf_cp(x-1,y,z,W)),
24       _mm_add_pd(_mm_mul_pd(pre1, _mm_load_pd(&getf(x,y,z,W))),
25                  _mm_mul_pd(pre2, _mm_sub_pd(temp, ux))));
26
27   [...]
28
29 }
```

Quellcodeausschnitt 6.4: `collide2` zur simultanen Kollision mehrerer Zellen

Wie bereits mehrfach erwähnt, sollten die Daten an 16 Byte Grenzen ausgerichtet sein. Dies ist bei der Compressed-Grid Methode (Kapitel 5.4) problematisch: Normalerweise werden bei der Compressed-Grid Methode in der Propagationsphase die Werte um eine Zelle in jeder Raumrichtung verschoben. Dies führt dazu, dass alle Zellen in x-Richtung um 8 Byte (die Größe eines `double`) verschoben werden, falls also in einem Simulationsschritt die Zellen mit ungerader x-Komponente an 16 Byte Grenzen ausgerichtet sind,

so sind im nächsten Schritt alle Zellen mit gerader x-Komponente an 16 Byte Grenzen ausgerichtet.

Man kann dies akzeptieren und die Schleifen passend formulieren oder aber man verschiebt die Gitter um zwei Zellen gegeneinander. Somit werden alle Zellen um 16 Byte in x-Richtung verschoben und alle zuvor ausgerichteten Zellen sind wieder ausgerichtet. Ein Nachteil hieran ist, dass mehr Speicher benötigt wird (eine zusätzliche Ebene an Halo-Zellen je Raumrichtung) und die Lokalität etwas schlechter wird.

7 Resultate

7.1 Einfluss der Compileroptimierungen

In diesem Kapitel werden die Auswirkungen verschiedener Compiler-Optionen auf die Ausführungsgeschwindigkeit des Simulationsprogramms untersucht. Dies dient einerseits dazu, sinnvolle Compiler-Einstellungen für die weiteren Auswertungen zu finden, andererseits soll aber auch festgestellt werden, wie sich der Compiler bezüglich der verschiedenen selbst implementierten Optimierungen verhält.

Um letzteres zu testen, wurden drei verschiedene Versionen des Simulationsprogramms mit unterschiedlichen Compiler-Optionen übersetzt und getestet. Die Programmversionen sind im einzelnen:

- Testprogramm 1: Referenzprogramm ohne Optimierungen, Two-Grid Version mit Array of Structures Speicherlayout.
- Testprogramm 2: Compressed-Grid Version mit Array of Structures Speicherlayout, 3D-Cache-Blocking mit Blockgröße 22, manuelles Prefetching der nächsten benötigten Zelle mit SSE
- Testprogramm 3: Two-Grid Version mit Structure of Arrays Speicherlayout, simultane Kollision zweier Zellen mit SSE2

Zu den verwendeten Compiler-Optionen siehe Kapitel 4.3, genaue Informationen zu den Compilern selbst finden sich in Anhang A.2. Die Messungen wurden alle mit kubischen Simulationsgittern der Größe 64^3 durchgeführt. Die Ergebnisse für den Intelcompiler finden sich für die Testplattformen aus Anhang A.1 in den Tabellen 7.1, 7.2 und 7.3, für den GCC in Tabelle 7.4.

Compileroptionen	Programm 1	Programm 2	Programm 3
-O0 -static	1,36	1,18	2,33
-O1 -static	2,61	3,40	3,26
-O3 -static	2,60	3,39	3,26
-O3 -static -axW	2,61	3,46	3,40
-O3 -static -ipo	2,62	3,40	3,29
-O3 -static -Ob2	2,60	3,38	Fehler
-O3 -static -axW -ipo -Ob2	**2,62**	**3,49**	**3,45**

Tabelle 7.1: Einfluss verschiedener Compileroptimierungen, Intelcompiler, Testplattform 1 (Pentium 4), Feldgröße 64^3, alle Werte in MLSUPS

Compileroptionen	Programm 1	Programm 2	Programm 3
-O0 -static	1,31	1,13	1,96
-O1 -static	1,57	3,07	2,61
-O3 -static	1,57	3,06	2,61
-O3 -static -axW	1,58	**3,22**	2,68
-O3 -static -ipo	1,58	3,08	2,61
-O3 -static -Ob2	1,58	3,05	Fehler
-O3 -static -axW -ipo -Ob2	**1,58**	3,16	**2,71**

Tabelle 7.2: Einfluss verschiedener Compileroptimierungen, Intelcompiler, Testplattform 2 (Xeon), Feldgröße 64^3, alle Werte in MLSUPS

Compileroptionen	Programm 1	Programm 2	Programm 3
-O0 -static	0,99	1,09	1,72
-O1 -static	1,98	3,30	2,75
-O3 -static	1,98	3,30	2,75
-O3 -static -axW	**2,16**	**3,66**	2,80
-O3 -static -ipo	1,99	3,30	2,81
-O3 -static -Ob2	2,00	3,29	Fehler
-O3 -static -axW -ipo -Ob2	2,05	3,53	**2,86**

Tabelle 7.3: Einfluss verschiedener Compileroptimierungen, Intelcompiler, Testplattform 3 (Opteron), Feldgröße 64^3, alle Werte in MLSUPS

Bei den Messungen mit dem Intelcompiler ist zunächst auffällig, dass für die Compiler-Optionen `-O3 -static -Ob2` fehlerhafter Code generiert wird. Es trat auf allen Testplattformen ein Segmentierungsfehler auf. Dies passierte jedoch nicht, wenn weitere Compiler-Optionen hinzugefügt wurden. Ein Grund für dieses Verhalten konnte nicht festgestellt werden.

Ebenso auffällig ist, dass nahezu kein Unterschied zwischen den Optimierungsstufen `O1` (diese ist identisch mit `O2`) und `O3` besteht. Insgesamt ist zu sagen, dass nur die Option `axW` noch einen merklichen Vorteil gegenüber der höchsten standardmäßigen Optimierungsstufe bringt.

Interessant ist, dass Testprogramm 3 ohne Compileroptimierungen deutlich schneller ist als die Testprogramme 1 und 2 ohne Optimierungen. Jedoch profitiert dieses nicht mehr so stark von den Optimierungen des Compilers, so dass Testprogramm 3 etwas von seinem Geschwindigkeitsvorteil einbüßt.

Auch bei der Verwendung des GCC (Tabelle 7.4) trat zunächst ein seltsamer Fehler auf, der nicht geklärt werden konnte. Auf Testplattform 2 ließen sich keine statisch gelinkten Programme ausführen. Die Messungen in Tabelle 7.4 wurde für Plattform 2 also ohne `-static` durchgeführt.

Auch hier besteht kein merklicher Unterschied zwischen den Stufen `O2` und `O3`. Interessant ist, dass auf allen Plattformen, speziell auch den Intel-basierten, die Optimierung für Opteron-Systeme beinahe 10% Geschwindigkeitszuwachs bringt. Hingegen haben die Optionen zum automatischen Vektorisieren und Prefetching keinen merklichen Effekt.

Compileroptionen (-msse -msse2 immer verwendet)	Plattform 1 Pentium 4	Plattform 2 Xeon	Plattform 3 Opteron
-O0	1,19	1,25	1,13
-O1	1,96	1,98	2,50
-O2	2,72	2,58	3,08
-O3	2,73	2,58	3,08
-O3 -static	2,74	Fehler	3,09
-O3 -static -march=opteron	**3,28**	**3,01**	**3,33**
-O3 -static -march=pentium4	2.96	2,78	3,07
-O3 -static -fprefetch-loop-arrays	2,74	2,60	3,09
-O3 -static -ftree-vectorize	2,74	2,57	3,08

Tabelle 7.4: Einfluss verschiedener Compileroptimierungen, GCC, verschiedene Testplattformen, Testprogramm 2, Feldgröße 64^3, alle Werte in MLSUPS

Insgesamt sind die Programme, die mit dem GCC erzeugt wurden, um etwa 5%-10% langsamer als die des Intelcompilers. Deswegen wurden alle weiteren Messungen mit dem Intelcompiler durchgeführt.

7.2 Einfluss der Speicherbandbreite

Wie bereits in der Einleitung erwähnt, ist die Lattice-Boltzmann-Methode eine typische Anwendung für Cluster. In neuerer Zeit werden Cluster-Knoten immer häufiger mit mehreren Prozessoren ausgestattet, wie Testplattform 2, welche ein Knoten des institutseigenen Clusters ist und zwei Intel Xeon Prozessoren besitzt.

Daraus ergab sich die Fragestellung, wie sich die Geschwindigkeit des Simulationsprogramms verhält, wenn zwei Prozesse gleichzeitig auf einem Rechner ablaufen. Dieses Szenario kann beispielsweise bei einer Parallelisierung mittels MPI auftreten, wenn keine speziellen Vorkehrungen zur Ausnutzung mehrerer Prozessoren getroffen werden, sondern nur mehrere Prozesse gestartet werden. MPI steht für Message Passing Interface und ist ein Standard zum Austauschen von Nachrichten in Fortran-, C++- und C-Programmen, der speziell auf die Verwendung auf Clustern abgestimmt ist.

In Tabelle 7.5 sind die Ergebnisse der Messungen dargestellt, die sich aus dieser Fragestellung ergaben. Es wurden auf den Testplattformen 2 (Dual-Xeon) und 3 (Dual-Opteron) jeweils ein einzelner Prozess und zwei Prozesse gleichzeitig ausgeführt, die Werte für zwei Prozesse sind die Mittelwerte der beiden Einzelgeschwindigkeiten.

	Plattform 2 Dual-Xeon		Plattform 3 Dual-Opteron	
Feldgröße	8^3	64^3	8^3	64^3
1 Prozess	5,41	2,98	6,24	3,52
2 Prozesse	5,39	1,67	6,25	3,51

Tabelle 7.5: Simultane Ausführunge mehrerer Prozesse, Testprogramm 2, Feldgröße 64^3

Auffällig ist der Wert für zwei simultan ausgeführte Prozesse auf Testplattform 2 bei einer Gittergröße von 64^3. Beide Prozesse zusammen leisten nur 12% mehr als ein einzelner Prozess. Auf Testplattform 3 ist dies nicht der Fall: Zwei Prozesse leisten hier das doppelte.

Dieses Verhalten lässt sich durch einen genaueren Blick in die Konfigurationen der Testplattformen 2 und 3 erklären und die Erklärung wurde bereits im Titel des Abschnittes verraten. Testplattform 2 verfügt nur über ein gemeinsames Speicherinterface für beide Prozessoren und somit müssen sich diese bei Gittergrößen, die nicht mehr in den Cache passen, die Speicherbandbreite teilen.

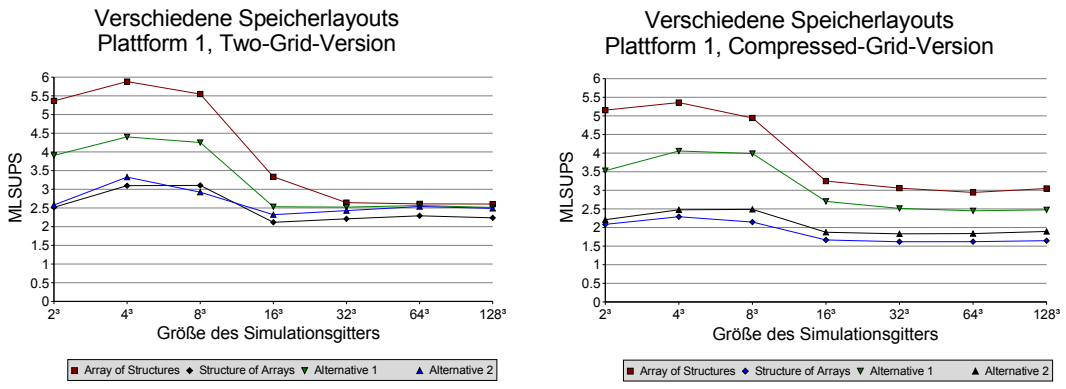
Dadurch erreichen die Prozesse nur noch 56% der maximalen Ausführungsgeschwindigkeit, die sie bei der vollen Speicherbandbreite erreichen. Man sieht hier also ganz deutlich, dass die Speicherbandbreite die Ausführungsgeschwindigkeit limitiert. Die Messwerte bei der Gittergröße 8^3 stützen diese These: Hier findet kein Leistungseinbruch statt, da ein Gitter der Größe 8^3 vollständig in den prozessoreigenen Cache passt. In diesem Fall müssen sich die Prozessoren nur einmal die Speicherbandbreite teilen, nämlich um die Daten in ihren Cache zu laden. Danach können beide Prozesse vollständig im Cache ausgeführt werden und erreichen die maximal mögliche Geschwindigkeit.

Testplattform 3 hingegen besteht aus zwei Opteron-Prozessoren. Jeder dieser Prozessoren besitzt einen eigenen integrierten Speichercontroller, so dass beide Prozesse auf je einem Prozessor mit voller Geschwindigkeit ablaufen können.

7.3 Einfluss der Optimierungen ohne SSE

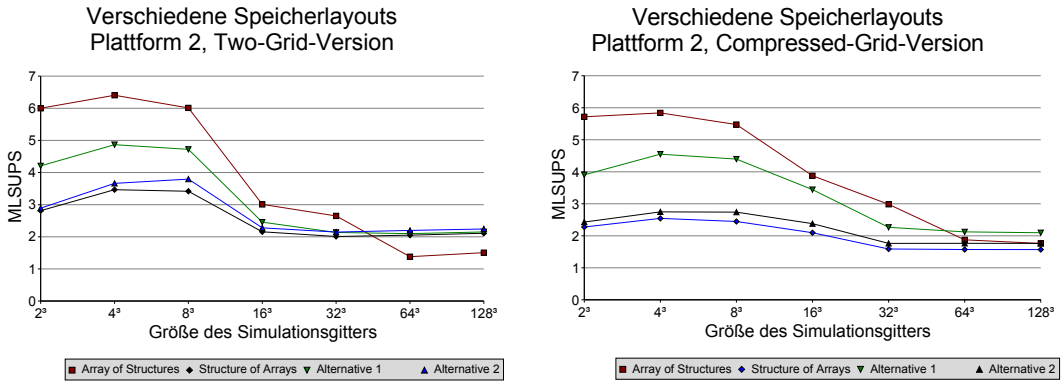
7.3.1 Wahl des Speicherlayouts und Compressed-Grid Methode

In diesem Kapitel werden zunächst die Effekte der verschiedenen Optimierungen aus Kapitel 5 dargestellt. In den Diagrammen 7.1, 7.2 und 7.3 sind die Auswirkungen für die verschiedenen Testplattformen graphisch aufbereitet. Für eine Übersicht der Hardwarekonfigurationen der einzelnen Testplattformen siehe Anhang A.1.



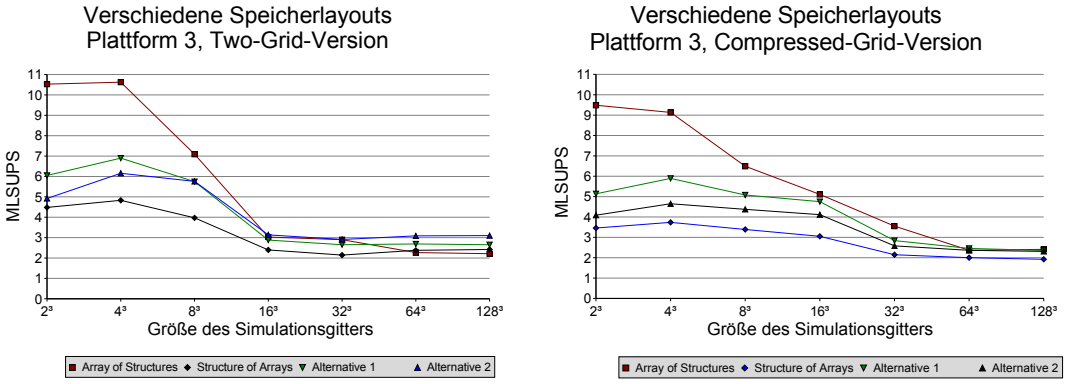
(a) Two-Grid (b) Compressed-Grid

Abbildung 7.1: Einfluss der Speicherlayouts aus Kapitel 5.3, Plattform 1 (Pentium 4)



(a) Two-Grid (b) Compressed-Grid

Abbildung 7.2: Einfluss der Speicherlayouts aus Kapitel 5.3, Plattform 2 (Xeon)



(a) Two-Grid (b) Compressed-Grid

Abbildung 7.3: Einfluss der Speicherlayouts aus Kapitel 5.3, Plattform 3 (Opteron)

Zunächst fällt auf, dass alle Testplattformen bei kleinen Systemgrößen bis etwa 8^3 Zellen deutlich höhere Rechenleistungen erzielen als bei größeren Systemen. Dies ist darauf zurückzuführen, dass solch kleine Systeme noch vollständig im Cache vorgehalten werden können und die Speicherbandbreite keinen Flaschenhals darstellt. Ebenso zeigt das einfachste und naheliegendste Speicherlayout, Array of Structures, bei kleinen Systemgrößen auf allen Plattformen die höchste Leistung.

Die maximale Gittergröße, die vollständig im Cache gehalten werden kann, lässt sich leicht wie in Gleichung 7.1 berechnen. Dabei ist die Anzahl Gitter für die Two-Grid Version zwei, für Compressed-Grid eins. Zu beachten ist, dass die Gittergröße ebenfalls die Halo-Zellen umfassen muss, die tatsächlichen Simulationsgitter sind also kleiner.

$$\text{Maximale Gittergröße für Berechnungen im Cache} = \frac{\text{Cachegröße in Byte}}{\text{Anzahl diskreter Geschwindigkeiten} \cdot \text{Größe des Datentyps} \cdot \text{Anzahl Gitter}} \tag{7.1}$$

Betrachtet man beispielsweise Abbildung 7.1(a), so fällt auf, dass die Rechengeschwindigkeit ab einer Größe von 8^3 einbricht. Nach Gleichung 7.1 ergibt sich bei Plattform 1 (Pentium 4) mit einem L2-Cache von 512 kB die maximale Gittergröße, die bei einer Berechnung der Two-Grid Version im Cache gehalten werden kann, zu $\frac{512 \cdot 1024}{19 \cdot 8 \cdot 2} \approx 12^3$. Da Halo-Zellen in jeder Richtung benötigt werden, kann also ein kubisches Simulationsgitter maximal die Größe 10^3 besitzen. Damit ist klar, dass ein Gitter der Größe 8^3 noch in den Cache passt, ein Gitter der Größe 16^3 jedoch nicht mehr, weswegen die Leistung an dieser Stelle einbricht.

In Abbildung 7.3(a) sind zwei Einbrüche zu erkennen. Dies ist darauf zurückzuführen, dass Plattform 3 (Opteron) einen deutlich größeren L1-Cache hat als die Plattformen 1 und 2 (Pentium 4 und Xeon). Die Größe des L1-Caches ist 64 kB, es können damit Gitter bis zu einer Größe von 4^3 bei der Two-Grid Version vollständig im L1-Cache simuliert werden. Bis zu einer Gittergröße von 13^3 für die Two-Grid Version bzw. 17^3 für die Compressed-Grid Version kann der L2-Cache mit einer Kapazität von 1 MB genutzt werden, deswegen zeigt die Compressed-Grid Version auf dieser Plattform für ein System der Größe 16^3 eine deutlich größere Leistung als die Two-Grid Version (vergleiche Abbildungen 7.3(a) und 7.3(b)).

Plattform 1 zeigt dieses Verhalten nicht. Da Systeme der Größe 16^3 sowohl für Two-Grid wie auch für Compressed-Grid nicht in den Cache passen, sind die Leistungen hier ungefähr gleich. Plattform 2 besitzt einen L3-Cache der Kapazität 1 MB, aber nur einen sehr kleinen L1-Cache für Daten. Deswegen zeigt diese Plattform für kleine Systeme das gleiche Verhalten wie Plattform 1, speziell für eine Gittergröße von 16^3 ist aber das gleiche Verhalten wie bei Plattform 3 erkennbar.

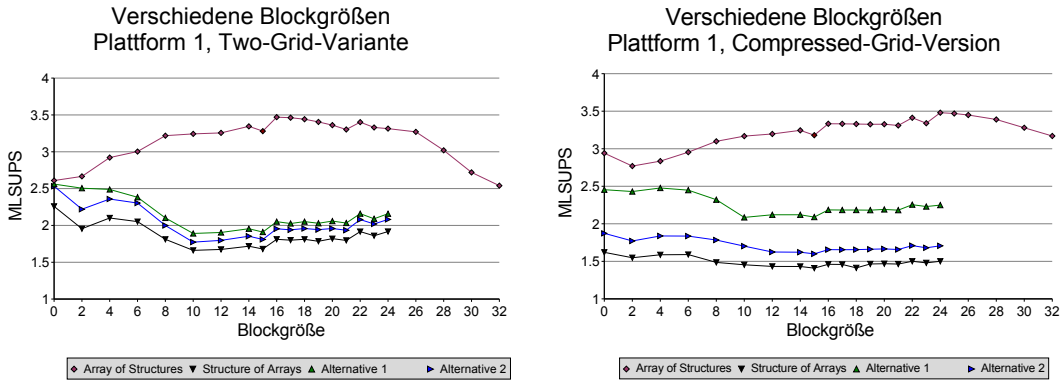
Ab einer Systemgröße von etwa 32^3 Zellen bleiben für alle Plattformen die Werte für die Two-Grid Version auf konstantem Niveau, da ab dieser Systemgröße keine Cache-Effekte mehr auftreten. Dies ist ab einer Feldgröße von 64^3 auch bei der Compressed-Grid Version der Fall. Bei diesen Feldgrößen sind je nach Plattform unterschiedliche Speicherlayouts am geeignetsten, die Unterschiede in der Leistung sind jedoch zumeist gering.

Bei Systemgrößen unter 16^3 Zellen zeigt die Compressed-Grid Version generell geringfügig schwächere Leistungen, was auf die kompliziertere Organisation der Simulationsschleifen zurückzuführen ist. Die Plattformen 1 und 2 zeigen bei Systemgrößen über 16^3 Zellen deutlich höhere Leistungen bei der Compressed-Grid Version, interessanterweise bei verschiedenen Speicherlayouts. Nur Testplattform 3 profitiert nicht von der Compressed-Grid Technik. Dies könnte auf die deutlich schnellere Speicheranbindung des Opteron-Systems zurückzuführen sein.

7.3.2 3D-Loop-Blocking

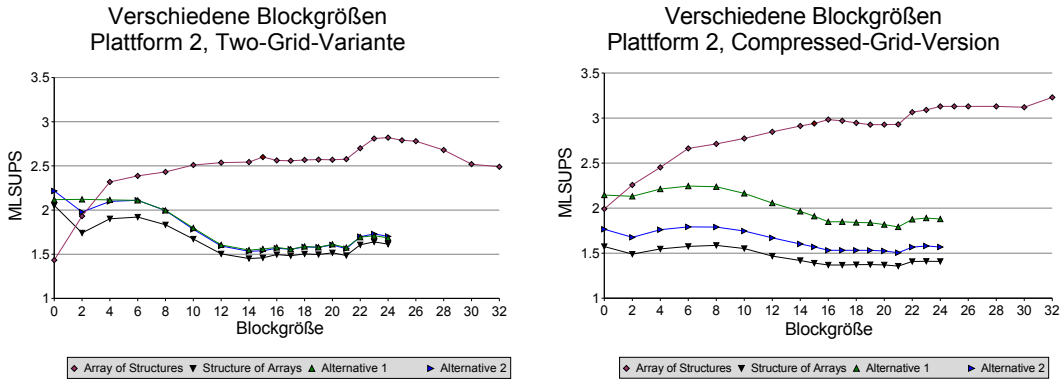
Die Auswirkungen des 3D-Loop-Blocking (siehe 5.5) sind in den Diagrammen 7.4, 7.5 und 7.6 zu sehen. Dabei bedeutet eine Blockgröße von x, dass das Simulationsgitter in Würfel von $x \times x \times x$ Zellen aufgeteilt wird.

Auf allen verwendeten Testplattformen profitiert nur das Array of Structures Speicherlayout von 3D-Loop-Blocking. Dies ist nicht verwunderlich, denn nur bei diesem Speicherlayout entsteht durch das 3D-Loop-Blocking ein einzelner zusammenhängender Speicherblock. Im Gegensatz dazu entstehen beim Structure of Arrays Speicherlayout beispiels-



(a) Two-Grid (b) Compressed-Grid

Abbildung 7.4: Einfluss verschiedener Blockgrößen bei 3D-Loop-Blocking, Plattform 1 (Pentium 4), Gittergröße 64^3



(a) Two-Grid (b) Compressed-Grid

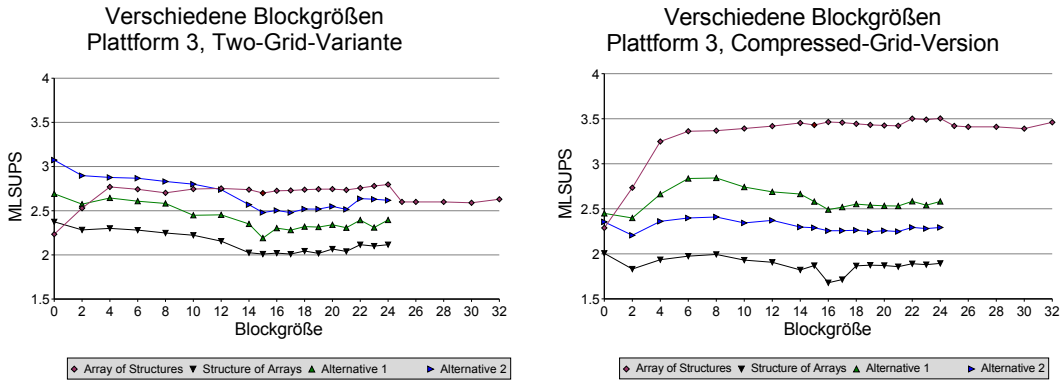
Abbildung 7.5: Einfluss verschiedener Blockgrößen bei 3D-Loop-Blocking, Plattform 2 (Xeon), Gittergröße 64^3

weise 19 kleinere Blöcke (einer für jede diskrete Geschwindigkeit), die sich gegenseitig aus dem Cache verdrängen können.

Tatsächlich zeigen auf den Plattformen 1 und 2 alle Speicherlayouts außer Array of Structures eine schlechtere Leistung. Da für Blockgrößen über 24 keine grundsätzliche Veränderung des Verhaltens bei den Speicherlayouts Structure of Arrays, Alternative 1 und Alternative 2 zu erwarten ist, sind keine weiteren Messwerte aufgeführt. Diese Speicherlayouts zeigen ihr optimales Verhalten bei sehr kleinen Blockgrößen und nähern sich für Blockgrößen größer als ca. 16 von unten dem Wert ohne 3D-Loop-Blocking an.

Nur für Plattform 3 (Opteron) ist bei der Compressed-Grid Version (Abbildung 7.6(b)) die Leistung dieser Speicherlayouts deutlich höher als ohne 3D-Loop-Blocking. Dies tritt bei einer Blockgröße von ca. 6^3 auf, was ungefähr der Systemgröße entspricht, die im L1-Cache gehalten werden kann.

Interessant ist, dass auf Plattform 3 auch das Array of Structures Speicherlayout bei



(a) Two-Grid (b) Compressed-Grid

Abbildung 7.6: Einfluss verschiedener Blockgrößen bei 3D-Loop-Blocking, Plattform 3 (Opteron), Gittergröße 64^3

der Two-Grid Version an Geschwindigkeit einbüßt. Dies könnte daran liegen, dass das automatische Prefetching des Opteron (der Prozessor versucht, in Zukunft benötigte Werte vorab zu laden), durch das 3D-Loop-Blocking gestört wird, da dies dazu führt, dass die Speicherzugriffe weniger regelmäßig stattfinden. Die Compressed-Grid Version gewinnt jedoch auch auf Testplattform 3 deutlich an Geschwindigkeit.

Das 3D-Loop-Blocking im Zusammenspiel mit der Compressed-Grid Technik erweist sich auf allen Plattformen als bisher schnellste Programmversion. Dabei gilt es jedoch für jede Plattform die optimale Blockgröße zu bestimmen. Die optimale Blockgröße sollte einerseits möglichst in den Cache passen, andererseits jedoch nicht zu klein gewählt werden, da sonst die Nachteile der sehr kurzen Schleifen überwiegen.

Dabei ist es nicht ohne Weiteres möglich, die Ergebnisse von Kapitel 7.3.1 zu übertragen. Bei einer sehr kleinen Systemgröße werden die Werte in den Caches über mehrere Zeitschritte hinweg wieder verwendet. Das Ziel des 3D-Loop-Blocking ist jedoch, die Wiederverwendbarkeit der Werte innerhalb eines Zeitschrittes zu erhöhen, da die Werte nicht bis zum nächsten Zeitschritt im Cache verbleiben können, weil das Simulationsgebiet insgesamt zu groß ist. Erst die Technik des 4-way-blocking aus [Hau05] ermöglicht es, die Werte bei großen Simulationsgebieten über mehrere Zeitschritte im Cache wieder zu verwenden, siehe dazu das Ende von Kapitel 5.5.

Bei Plattform 1 (Pentium 4) können in den Abbildungen 7.4(a) und 7.4(b) deutlich die optimalen Blockgrößen abgelesen werden. Bei Plattform 2 (Xeon) ist bei der Two-Grid Version (Abbildung 7.5(a)) auch deutlich das Optimum zu erkennen. Bei der Compressed-Grid Version in Abbildung 7.5(b) hingegen scheint die Performance immer weiter zu steigen. Dies ist jedoch nicht der Fall, das Optimum liegt bei ca. 24, danach fällt die Leistung sehr langsam ab. Der hohe Wert bei Blockgröße 32 ist ein Ausreißer nach oben, der auf die Größe des Simulationsgitters von 64^3 zurückzuführen ist: Hier lässt sich das Simulationsgebiet in exakt vier Blöcke aufteilen. Bei anderen Gittergrößen tritt dies nicht auf und die Leistung nimmt ab Blockgröße 24 ab.

Die optimale Blockgröße für das Speicherlayout Array of Structures bei Plattform 3 (Opteron) ist in der Two-Grid Version 24, danach tritt ein deutlicher Einbruch auf.

Bei der Compressed-Grid Version ist der gleiche Effekt wie auf Plattform 2 zu sehen: Blockgröße 32 ist ein Ausreißer nach oben, das Optimum liegt bei 22 bis 24.

Zu den konventionellen Optimierungen ist abschließend festzustellen, dass sich sowohl 3D-Loop-Blocking wie auch Compressed-Grid deutlich auszahlen, besonders in Verbindung miteinander. Alle diese Optimierungen sind jedoch speziell auf das Array of Structures Speicherlayout zugeschnitten, weswegen die anderen Speicherlayouts hierdurch kaum Vorteile verbuchen können.

7.4 Einfluss der Optimierungen mit SSE

Die Auswirkungen der Optimierungen mit SSE durch die Verwendung von intrinsischen Funktionen sollen nun in der gleichen Reihenfolge dargestellt werden, in der sie in Kapitel 6 besprochen wurden, beginnend mit der manuellen Beeinflussung des Caching-Verhaltens.

7.4.1 Beeinflussen des Caching-Verhaltens

Die Diagramme 7.7, 7.8 und 7.9 zeigen die Effekte des manuellen Prefetchings auf den unterschiedlichen Testplattformen. Dabei wurde aufbauend auf den konventionellen Optimierungen stets das Array of Structures Speicherlayout verwendet. Dargestellt sind jeweils Two-Grid und Compressed-Grid Version mit und ohne 3D-Loop-Blocking. Eine Prefetching-Distanz von x bedeutet, dass bei der Simulation einer Zelle bereits die Werte der Zelle geladen werden, die in x weiteren Schritten benötigt werden.

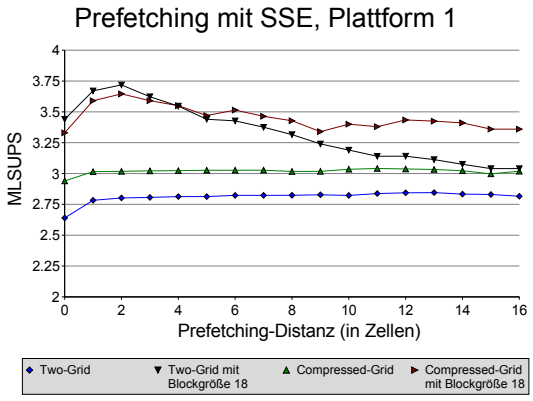


Abbildung 7.7: Einfluss von SSE-Prefetching, Plattform 1 (Pentium 4), Gittergröße 64^3, Array of Structures Speicherlayout

Interessant ist bei den Plattformen 1 und 2, dass ohne 3D-Loop-Blocking die gleiche Geschwindigkeit erzielt wird, unabhängig davon, welche Zelle vorab geladen wird. Dennoch ist die Geschwindigkeit geringfügig höher als ohne Prefetching.

Mit 3D-Loop-Blocking steigt die Leistung bei allen Plattformen durch das Prefetching deutlich. Die größte Geschwindigkeit wird erzielt, wenn die jeweils nächste benötigte Zelle (bzw. die übernächste benötigte Zelle für Testplattform 2) vorab geladen wird. Zu große

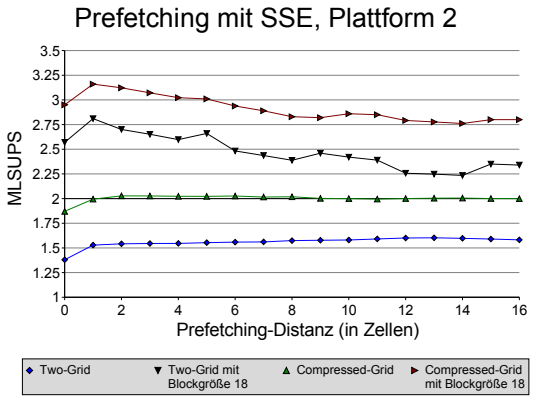


Abbildung 7.8: Einfluss von SSE-Prefetching, Plattform 2 (Xeon), Gittergröße 64^3, Array of Structures Speicherlayout

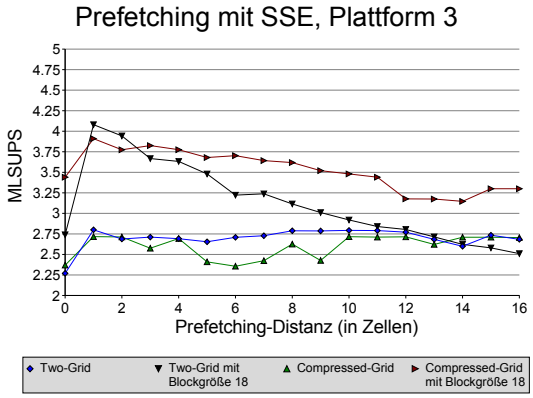


Abbildung 7.9: Einfluss von SSE-Prefetching, Plattform 3 (Opteron), Gittergröße 64^3, Array of Structures Speicherlayout

Prefetching-Distanzen führen dazu, dass oftmals Werte geladen werden, die nicht mehr im gerade durch das 3D-Loop-Blocking behandelten Speicherblock liegen, so dass hier die Geschwindigkeit wieder sinkt. Außerdem fällt auf, dass auf den Testplattformen 1 und 3 die Two-Grid Version durch das Prefetching den Vorteil der Compressed-Grid Version wett macht und diese sogar überholt.

Dieser überraschend große Effekt des Prefetching im Zusammenspiel mit 3D-Loop-Blocking liegt möglicherweise darin begründet, dass durch das 3D-Loop-Blocking die Speicherzugriffe wesentlich unregelmäßiger stattfinden, so dass der in die Prozessoren integrierte Hardware-Prefetching-Mechanismus nur schlecht funktioniert. Jedenfalls lassen sich im extremsten Fall auf Testplattform 3 beinahe 50% Geschwindigkeitssteigerung erzielen, wobei der ursprüngliche Quellcode an keiner Stelle verändert werden musste. Es wurden lediglich an einigen Stellen die in Kapitel 6.1 gezeigten Anweisungen zusätzlich eingefügt.

7.4.2 Simultane Berechnung entgegengesetzter diskreter Geschwindigkeiten

Die Ergebnisse der Optimierung aus Kapitel 6.1 können kurz gefasst werden. In Diagramm 7.10 sind die Ergebnisse für die Two-Grid Version ohne 3D-Loop-Blocking dargestellt.

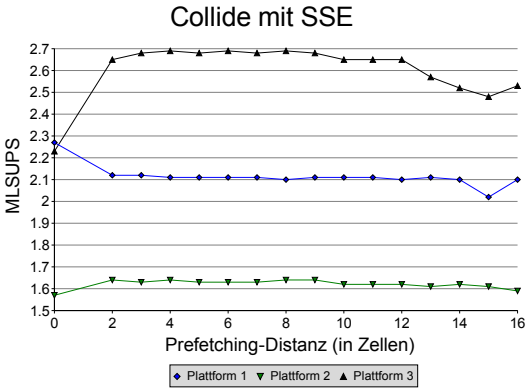


Abbildung 7.10: Simultane Berechnung mehrerer Raumrichtungen, Gittergröße 64^3, Two-Grid Version

Auf keiner der Testplattformen kann diese Optimierung einen Vorteil verbuchen, tatsächlich sind die erzielten Geschwindigkeiten sogar langsamer als ohne diese Optimierung. Auch der Einsatz von 3D-Loop-Blocking ändert dies nicht. Das kann an den in Kapitel 6.1 beschriebenen Problemen beim Speicherzugriff liegen: Die beiden Werte, die simultan in einem SSE-Register berechnet werden, müssen an verschiedene Stellen im Speicher geschrieben werden.

Tabelle 7.6 lässt jedoch auch einen anderen Schluss zu. Bei Verzicht auf sämtliche Compileroptimierungen ist die von Hand optimierte Version ca. 22% schneller als die herkömmliche Version. Möglicherweise haben die Compileroptimierungen für den Einsatz von SSE2 noch nicht das Niveau der Optimierungen für herkömmlichen Quellcode erreicht.

Programmbeschreibung	MLSUPS
Two-Grid, Array of Structures Speicherlayout	1,33
Two-Grid, Array of Structures Speicherlayout, zusammengefasste Berechnung unterschiedlicher Richtungen	1,62

Tabelle 7.6: Simultane Berechnung entgegengesetzter diskreter Geschwindigkeiten ohne Compileroptimierungen, Testplattform 1 (Pentium 4), Gittergröße 64^3

7.4.3 Simultane Kollision mehrerer Zellen

Zuletzt werden hier die Ergebnisse der Optimierung aus Kapitel 6.2.2 dargestellt. Obwohl sich in Kapitel 7.3 die Speicherlayouts Structure of Arrays und Alternative 1 als nicht konkurrenzfähig erwiesen, zeigt sich in den Diagrammen 7.11, 7.12 und 7.13 bereits ohne

Prefetching (Prefetching-Distanz 0), dass die Vektorisierung durchaus mit den schnellsten konventionellen Varianten mithalten kann.

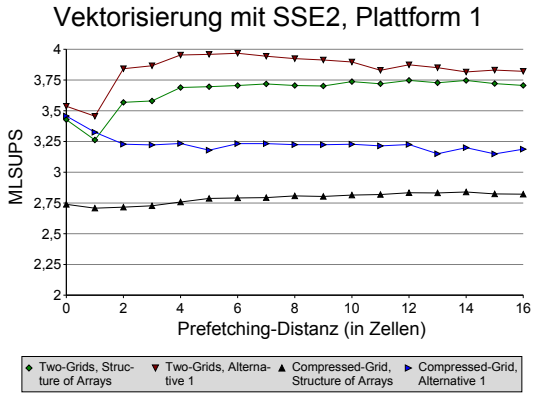


Abbildung 7.11: Einfluss der manuellen Vektorisierung mit SSE2, Plattform 1 (Pentium 4), Gittergröße 64^3

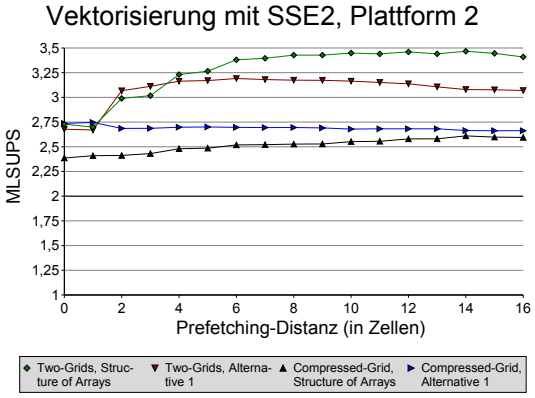


Abbildung 7.12: Einfluss der manuellen Vektorisierung mit SSE2, Plattform 2 (Xeon), Gittergröße 64^3

Zusammen mit manuellem Prefetching ist die von Hand vektorisierte Version die mit Abstand schnellste auf allen Plattformen. Dabei ist auf allen Plattformen die Two-Grid Version die schnellere, jedoch mit unterschiedlichen Speicherlayouts. Plattform 1 und 3 erreichen die höchste Geschwindigkeit mit dem Speicherlayout Alternative 1, Plattform 2 hingegen mit dem Speicherlayout Structure of Arrays.

Insgesamt zeigt die SSE-Einheit hier einige typische Merkmale eines Vektorrechners:

- das Speicherlayout muss so gewählt sein, dass zusammenhängende lange Arrays verarbeitet werden können: Structure of Arrays oder Alternative 1
- die Schleifen müssen möglichst einfach sein: Two-Grid Version
- Loop-Blocking ist schädlich, da die Vektorlänge dadurch geringer wird (siehe Tabelle 7.7)

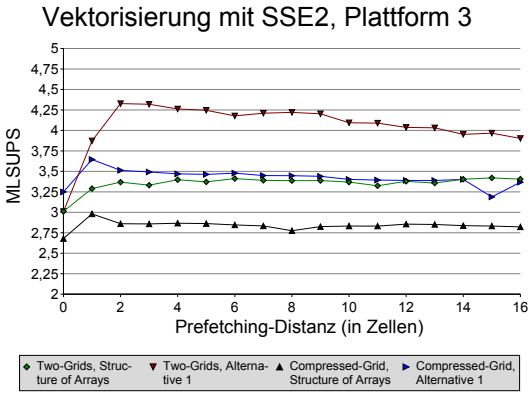


Abbildung 7.13: Einfluss der manuellen Vektorisierung mit SSE2, Plattform 3 (Opteron), Gittergröße 64^3

Blockgröße	MLSUPS
kein Blocking	3,46
8	2,38
16	2,36
24	2,56

Tabelle 7.7: Einfluss der Blockgröße auf die vektorisierte `collide`-Funktion, Plattform 1 (Pentium 4), Structure of Arrays Speicherlayout, Gittergröße 64^3

7.4.4 Zusammenstellung der schnellsten Versionen

Abschließend sollen in einer tabellarischen Übersicht die schnellsten Programmversionen auf jeder Plattform dargestellt werden (Tabellen 7.8, 7.9 und 7.10). Hier herrscht über die grundsätzlichen Versionen Einigkeit zwischen allen Plattformen, jedoch unterscheiden sich einige Parameter wie die beste Blockgröße und die beste Prefetching-Distanz. Außerdem findet sich in den Tabellen eine qualitative Übersicht der Geschwindigkeitssteigerungen, die die beste konventionelle und die beste mit SSE und SSE2 optimierte Version mit sich bringen.

Programmbeschreibung	MLSUPS
Referenzprogramm: Two-Grid Version, Array of Structures Speicherlayout	2,61 (100%)
Schnellstes Programm ohne SSE: Compressed-Grid Version, Array of Structures Speicherlayout, 3D-Blocking mit Blockgröße 24	3,48 (133%)
Schnellstes Programm mit SSE: Two-Grid, Alternative 1 Speicherlayout, simultane Kollision zweier Zellen, Prefetching im Abstand 6 Zellen	3,97 (152%)

Tabelle 7.8: Vergleich der schnellsten Programmversionen, Plattform 1 (Pentium 4)

Programmbeschreibung	MLSUPS
Referenzprogramm: Two-Grid Version, Array of Structures Speicherlayout	1,38 (100%)
Schnellstes Programm ohne SSE: Compressed-Grid Version, Array of Structures Speicherlayout, 3D-Blocking mit Blockgröße 23	2,98 (216%)
Schnellstes Programm mit SSE: Two-Grid, Structure of Arrays Speicherlayout, simultane Kollision zweier Zellen, Prefetching im Abstand 14 Zellen	3,47 (251%)

Tabelle 7.9: Vergleich der schnellsten Programmversionen, Plattform 2 (Xeon)

Programmbeschreibung	MLSUPS
Referenzprogramm: Two-Grid Version, Array of Structures Speicherlayout	2,27 (100%)
Schnellstes Programm ohne SSE: Compressed-Grid Version, Array of Structures Speicherlayout, 3D-Blocking mit Blockgröße 22	3,50 (154%)
Schnellstes Programm mit SSE: Two-Grid, Alternative 1 Speicherlayout, simultane Kollision zweier Zellen, Prefetching im Abstand 2 Zellen	4,33 (191%)

Tabelle 7.10: Vergleich der schnellsten Programmversionen, Plattform 3 (Opteron)

8 Zusammenfassung und Ausblick

Aktuelle Prozessoren für den Verbrauchermarkt haben durch ihr gutes Preis-/Leistungsverhältnis den Weg in moderne Hochleistungsrechner gefunden. Damit sind auch die SSE-Befehlssatzerweiterungen in vielen modernen Clustern zugänglich und da diese seit SSE2 Fließkommaberechnungen mit doppelter Genauigkeit unterstützen, verdienen sie größere Aufmerksamkeit im Hinblick auf ihren Einsatz in Simulationsprogrammen.

In dieser Arbeit sollte das Potential der Verwendung von SSE- und SSE2-Befehlen in einem Lattice-Boltzmann-Simulationsprogramm ausgelotet werden. Die Verwendung der SSE-Befehle sollte dabei mittels intrinsischer Funktionen geschehen, da diese einen leichteren Zugang bieten als die direkte Programmierung auf Assembler-Ebene.

Auch im Sinne des Software-Engineering ist die Verwendung intrinsischer Funktionen dem Einsatz von Inline-Assembler vorzuziehen, da der Compiler durch Typprüfungen und ähnliches Unterstützung bei der Programmierung mit intrinsischen Funktionen bietet. Dennoch darf beim Einsatz von intrinsischen Funktionen nicht vergessen werden, dass diese plattformspezifisch sind und die SSE-Erweiterungen nicht auf allen gängigen Architekturen verfügbar sind.

Um den Nutzen des Einsatzes von SSE und SSE2 qualitativ festzustellen, wurden zunächst gängige Optimierungstechniken wie verschiedene Speicheranordnungen, die Compressed-Grid Methode und 3D-Loop-Blocking implementiert. Danach wurden Optimierungen unter Verwendung von SSE und SSE2 entwickelt, um sowohl den Speicherzugriff als auch die Rechengeschwindigkeit zu erhöhen.

Ein wichtiges Ergebnis dieser Arbeit ist, dass eine sehr einfache Optimierung wie das manuelle Einfügen von Prefetch-Anweisungen bereits starke Geschwindigkeitssteigerungen hervorrufen kann. Diese Technik sollte sich gut auf andere Bereiche übertragen lassen, da der vorhandene Quellcode nicht verändert werden muss, es genügt, einige zusätzliche Zeilen einzufügen.

Weiterhin wurde festgestellt, dass es für den Einsatz von SSE-Befehlen wichtig ist, die Speicheranordnung der Daten an die Eigenschaften der SSE-Einheit anzupassen. Dazu genügt es nicht, von der bisher besten Variante auszugehen. Tatsächlich erwiesen sich einige der konventionellen Optimierungstechniken als kontraproduktiv im Zusammenspiel mit SSE, beispielsweise das 3D-Loop-Blocking. Die größte Geschwindigkeit wurde nicht aufbauend auf den konventionell optimierten Programmen erzielt, sondern vielmehr durch die Ausnutzung der Eigenschaften von Vektorrechnern.

Da es das Fernziel dieser Arbeit war, Optimierungen zu entwickeln, die auf modernen Clustern eingesetzt werden können, bleibt als nächster Schritt, die Optimierungen in ein paralleles Programm zu übertragen. Dabei sollte den Datenstrukturen besondere Aufmerksamkeit geschenkt werden, um Datenstrukturen zu entwickeln, die sich gleichermaßen für eine Parallelisierung mit MPI eignen, wie auch für die Verwendung von SSE.

Außerdem bleibt noch zu untersuchen, inwiefern die neuen arithmetischen Operationen der SSE3-Erweiterungen bei den Berechnungen der Lattice-Boltzmann-Methode gewinnbringend eingesetzt werden können.

A Testplattformen und Compiler

A.1 Testplattformen

A.1.1 Plattform 1

Prozessor Intel Pentium 4

Taktfrequenz 2,8 GHz

L1-Cache 8 kB Daten und 12000 μOps Instruktionen

L2-Cache 512 kB

Hauptspeicher 2 GB DDR PC266

A.1.2 Plattform 2

Testplattform 2 ist ein Knoten des institutseigenen Clusters Mozart, der aus 64 Knoten der angegebenen Hardwarekonfiguration und einem Verwaltungsknoten besteht, die mit einem Infiniband 4x Netzwerk verbunden sind. Weitere Details können unter http://www.ipvs.uni-stuttgart.de/abteilungen/sgs/abteilung/ausstattung/mozart/ betrachtet werden.

Prozessor 2 $\times$ Intel Xeon

Taktfrequenz 3,066 GHz

L1-Cache 8 kB Daten und 12000 μOps Instruktionen

L2-Cache 512 kB

L3-Cache 1 MB

Hauptspeicher 4 GB DDR PC266 ECC

A.1.3 Plattform 3

Prozessor 2 $\times$ AMD Opteron 248

Taktfrequenz 2,2 GHz

L1-Cache 64 KB Daten und 64 KB Instruktionen

L2-Cache 1 MB

Hauptspeicher 15 GB

A.2 Compiler

Sofern nicht anders angegeben wurden alle Messungen mit dem Intelcompiler durchgeführt.

A.2.1 Intelcompiler 8.0

- Genaue Bezeichnung: Intel(R) C++ Compiler for 32-bit applications, Version 8.0 Build 20031016Z
- Linker: GNU ld version 2.13.90.0.18 20030206
- Standardoptionen (sofern nicht anders angegeben):
  -std=c99 -O3 -static -ipo -axW -Ob2

A.2.2 GCC 4.0.1

- Genaue Bezeichnung: gcc version 4.0.1 20050727 (Red Hat 4.0.1-5)
- Linker: GNU ld version 2.15.94.0.2.2 20041220
- Standardoptionen (sofern nicht anders angegeben):
  -std=c99 -O3 -mtune=pentium4/opteron -msse -msse2

B Autovektorisierung des Compilers

In diesem Anhang soll ein kurzes Beispiel für eine Schleife gegeben werden, die der Compiler automatisch vektorisiert. Es werden keine Assembler-Kenntnisse benötigt, jedoch kann das gezeigte Vorgehen genutzt werden, um festzustellen, ob sich der manuelle Einsatz von SSE-Befehlen lohnt oder ob der Compiler bereits SSE-Befehle eingesetzt hat. Zur Vorgeschichte dieser Untersuchung siehe Kapitel 3.3.

Um ein leichteres Disassemblieren zu ermöglichen wurde das Beispiel aus Kapitel 3.3 in eine eigene Funktion ausgelagert, der verwendete Quellcode findet sich in Quellcodeausschnitt B.1.

```
1  void calc_conv(int i) {
2    rest[i] = a[i] / b;
3    ergebnis[i] = floor(rest[i]);
4    rest[i] = (rest[i] - ergebnis[i])*b;
5  }
6
7  int main(void){
8    [...]
9    for (i=0; i<N; i++) {
10     calc_conv(i)
11   }
12   [...]
13 }
```

Quellcodeausschnitt B.1: Konventionelle Berechnung der Vektoroperation

Kompiliert man das Programm mit dem Intelcompiler und der Option `-O0` (also ohne Optimierung), so entsteht für die Funktion `calc_conv` der in Quellcodeausschnitt B.2 dargestellte Maschinencode. Es ist nicht wichtig, die konkreten Assembler-Befehle lesen zu können, es fällt auf, dass nur die „normalen“ Register (wie `eax`) und keine SSE-Register (`xmm0-15`) benutzt werden, die Schleife also nicht vektorisiert wurde.

```
1  <calc_conv>:
2  push   %ebp
3  mov    %esp,%ebp
4  mov    0x8(%ebp),%eax
5  fldl   0x1652e560(,%eax,8)
6  fldl   0x804c190
7  fdivrp %st,%st(1)
8  mov    0x8(%ebp),%eax
9  fstpl  0x1b179980(,%eax,8)
10 add    $0xfffffff8,%esp
11 mov    0x8(%ebp),%eax
```

```
12 fldl   0x1b179980(,%eax,8)
13 fstpl  (%esp)
14 call   8048f68 <floor>
15 add    $0x8,%esp
16 mov    0x8(%ebp),%eax
17 fstpl  0x118e3160(,%eax,8)
18 mov    0x8(%ebp),%eax
19 mov    0x8(%ebp),%edx
20 fldl   0x1b179980(,%eax,8)
21 fldl   0x118e3160(,%edx,8)
22 fsubrp %st,%st(1)
23 mov    0x8(%ebp),%eax
24 fstpl  0x1b179980(,%eax,8)
25 mov    0x8(%ebp),%eax
26 fldl   0x1b179980(,%eax,8)
27 fldl   0x804c190
28 fmulp  %st,%st(1)
29 mov    0x8(%ebp),%eax
30 fstpl  0x1b179980(,%eax,8)
31 leave
32 ret
33 nop
```

Quellcodeausschnitt B.2: Maschinencode der Routine `calc_conv`, kompiliert mit -O0

Kompiliert man den gleichen Quelltext mit den Optionen `-O3 -axW`, wobei `-axW` den Compiler anweist, vektorisierten Code für die Pentium 4 Architektur zu erzeugen, dann ergibt sich der in Quellcodeausschnitt B.3 dargestellte Maschinencode. Auch hier benötigt man keine größeren Assembler-Kenntnisse. Zunächst wird geprüft, ob SSE2 auf dem ausführenden Computer zur Verfügung steht (Zeile 3), falls ja, so wird `calc_conv.J` aufgerufen. Man sieht sofort, dass die Berechnung mittels SSE2 stattfindet.

```
1 <calc_conv>:
2 testl  $0xfffffe00,0x804c8dc
3 jne    8048d0c <calc_conv.J>
4 testl  $0xffffffff,0x804c8dc
5 jne    8048d8c <calc_conv.A>
6 call   8048f78 <__intel_cpu_indicator_init>
7 jmp    8048ce4 <calc_conv>
8 ret
9
10
11 <calc_conv.J>:
12 push   %ebx
13 mov    %esp,%ebx
14 and    $0xfffffff8,%esp
15 sub    $0x10,%esp
16 mov    0x8(%ebx),%eax
17 movsd  0x804c190,%xmm0
```

```
18 movsd   0x1652e560(,%eax,8),%xmm1
19 divsd   %xmm0,%xmm1
20 movsd   0x804be90,%xmm4
21 movsd   0x804be88,%xmm3
22 movsd   0x804be80,%xmm5
23 andpd   %xmm1,%xmm4
24 orpd    %xmm4,%xmm3
25 movapd  %xmm3,%xmm7
26 addsd   %xmm1,%xmm7
27 subsd   %xmm3,%xmm7
28 movapd  %xmm7,%xmm6
29 subsd   %xmm1,%xmm6
30 cmpnlesd %xmm4,%xmm6
31 andpd   %xmm5,%xmm6
32 subsd   %xmm6,%xmm7
33 movsd   %xmm7,0x118e3160(,%eax,8)
34 subsd   %xmm7,%xmm1
35 mulsd   %xmm0,%xmm1
36 movsd   %xmm1,0x1b179980(,%eax,8)
37 mov     %ebx,%esp
38 pop     %ebx
```

Quellcodeausschnitt B.3: Maschinencode der Routine `calc_conv`, kompiliert mit Parametern -O3 -axW

Zum Vergleich folgt in Quellcodeausschnitt B.4 die von Hand vektorisierte Funktion aus Quellcodeausschnitt 3.4. Auffällig ist, dass der Compiler in der automatisch vektorisierten Funktion deutlich weniger gepackte Operationen (Maschinenbefehle, die auf `pd` enden: `mulpd, addpd` ...) benutzt, sondern SSE2-Befehle, die nur auf einem Wert arbeiten (Maschinenbefehle, die auf `sd` enden: `mulsd, addsd` ...)

```
1 <calc_intrin>:
2 push    %ebx
3 mov     %esp,%ebx
4 and     $0xfffffff0,%esp
5 mov     0x8(%ebx),%eax
6 movapd  0x1652e560(,%eax,8),%xmm6
7 movapd  %xmm6,0x1b179970
8 movapd  0x804c920,%xmm5
9 divpd   %xmm5,%xmm6
10 movapd %xmm6,0x1b179960
11 cvttpd2dq %xmm6,%xmm3
12 cvtdq2pd %xmm3,%xmm4
13 subpd   %xmm4,%xmm6
14 mulpd   %xmm5,%xmm6
15 movapd  %xmm4,0x804c910
16 movapd  %xmm4,0xcc97d40(,%eax,8)
17 movapd  %xmm6,0x804c900
```

```
18 movapd %xmm6,0x804c940(,%eax,8)
19 mov    %ebx,%esp
20 pop    %ebx
21 ret
```

Quellcodeausschnitt B.4: Maschinencode der Routine `calc_intrin`, kompiliert mit Parametern -O3 -axW

C Übersicht über wichtige intrinsische Funktionen

Hier findet sich eine Übersicht der wichtigen intrinsischen Funktionen, die in den Quellcodebeispielen verwendet werden. Diese Übersicht ist keineswegs vollständig, siehe zusätzlich [Int04b].

C.1 Intrinsische Funktionen für die Verwendung von SSE-Befehlen

Die SSE-Erweiterungen stellen Operationen für die Manipulation von `float`- und `int`-Werten mit einfacher Präzision bereit, die hier keine Verwendung finden. Deswegen findet sich bei den SSE-Erweiterungen nur eine Funktion von Interesse.

```
void _mm_prefetch(char const *a, int sel)
```

Diese Funktion lädt eine Cache Line Daten, die die Daten an Adresse `a` beinhaltet, in den Cache. In welchen Cache die Daten geladen werden, lässt sich durch `sel` bestimmen, indem die Konstanten _MM_HINT_T0 (in alle Caches laden), _MM_HINT_T1 (in L2- und L3-Cache laden), _MM_HINT_T2 (in L3-Cache laden, sofern vorhanden) und _MM_HINT_NTA (nur in L1-Cache laden, nicht in L2-Cache auslagern) übergeben werden.

C.2 Intrinsische Funktionen für die Verwendung von SSE2-Befehlen

Der für diese Arbeit wichtigste Datentyp der SSE2-Erweiterungen ist `__m128d`, welcher zwei double-Werte beinhaltet. Für die Beschreibung der einzelnen Funktionen seien für eine Variable `a` vom Typ `__m128d` `a0` und `a1` die beiden darin enthaltenen double-Werte.

C.2.1 Arithmetische Operationen

```
__m128d r _mm_add_sd(__m128d a, __m128d b)

r0 = a0 + b0
r1 = a1
```

```
__m128d r _mm_sub_sd(__m128d a, __m128d b)

r0 = a0 - b0
r1 = a1


__m128d r _mm_nul_sd(__m128d a, __m128d b)

r0 = a0 * b0
r1 = a1


__m128d r _mm_div_sd(__m128d a, __m128d b)

r0 = a0 / b0
r1 = a1


__m128d r _mm_add_pd(__m128d a, __m128d b)

r0 = a0 + b0
r1 = a1 + b1


__m128d r _mm_sub_pd(__m128d a, __m128d b)

r0 = a0 - b0
r1 = a1 - b1


__m128d r _mm_mul_pd(__m128d a, __m128d b)

r0 = a0 * b0
r1 = a1 * b1


__m128d r _mm_div_pd(__m128d a, __m128d b)

r0 = a0 / b0
r1 = a1 / b1
```

C.2.2 Lade- und Speicheroperationen

```
__m128d r _mm_load_pd(double const*dp)

r0 = dp[0]
r1 = dp[1]
```

dp muss an einer 16 Byte Grenze ausgerichtet sein!

```
__m128d r _mm_load1_pd(double const*dp)

r0 = dp[0]
r1 = dp[0]


__m128d r _mm_loadr_pd(double const*dp)

r0 = dp[1]
r1 = dp[0]
```

dp muss an einer 16 Byte Grenze ausgerichtet sein!

```
__m128d r _mm_loadu_pd(double const*dp)

r0 = dp[0]
r1 = dp[1]


__m128d r _mm_load_sd(double const*dp)

r0 = p[0]
r1 = 0.0


__m128d r _mm_loadh_pd(__m128d a, double const*dp)

r0 = a0
r1 = dp[0]


__m128d r _mm_loadl_pd(__m128d a, double const*dp)

r0 = dp[0]
r1 = a1


__m128d r _mm_set_sd(double w)

r0 = w
r1 = 0.0


__m128d r _mm_set1_pd(double w)

r0 = w
r1 = w
```

```
__m128d _mm_set_pd(double w, double x)

r0 = x
r1 = w
```

```
void _mm_store_sd(double *dp, __m128d a)

dp[0] = a0
```

```
void _mm_store1_pd(double *dp, __m128d a)

dp[0] = a0
dp[1] = a0
```

dp muss an einer 16 Byte Grenze ausgerichtet sein!

```
void _mm_store_pd(double *dp, __m128d a)

dp[0] = a0
dp[1] = a1
```

dp muss an einer 16 Byte Grenze ausgerichtet sein!

```
void _mm_storeu_pd(double *dp, __m128d a)

dp[0] = a0
dp[1] = a1
```

```
void _mm_storer_pd(double *dp, __m128d a)

dp[0] = a1
dp[1] = a0
```

dp muss an einer 16 Byte Grenze ausgerichtet sein!

```
void _mm_storeh_pd(double *dp, __m128d a)

dp[0] = a1
```

```
void _mm_storel_pd(double *dp, __m128d a)

dp[0] = a0
```

C.2.3 Sonstige Operationen

```
__m128d r _mm_shuffle_pd(__m128d a, __m128d b, int i)
```

Diese Operation dient dazu, aus den in `a` und `b` enthaltenen `double`-Werten einen neuen `__m128d`-Wert aufzubauen. Zur einfacheren Handhabung wird ein Makro bereitgestellt, dass den Wert für `i` liefert: `_MM_SHUFFLE2(x,y)`, wobei `x` und `y` 0 oder 1 sein können und als Selektor für die `double`-Werte aus `a` und `b` zu sehen sind.

Beispiel:

```
a0 = w
a1 = x
b0 = y
b1 = z
```

Dann ergibt:

```
c  = _mm_shuffle_pd(a, b, _MM_SHUFFLE2(0,1));
c0 = x
c1 = y
c  = _mm_shuffle_pd(a, b, _MM_SHUFFLE2(1,0));
c0 = w
c1 = z
c  = _mm_shuffle_pd(a, b, _MM_SHUFFLE2(1,1));
c0 = x
c1 = z
```

D Der `cpuid` Maschinenbefehl

Wenn ein Programm entwickelt wird, das hardwarenahe Optimierungen beinhaltet, stellt sich die Frage, wie man sicherstellen kann, dass die gerade benutze Hardware alle Voraussetzungen erfüllt. Zu diesem Zweck besitzen alle modernen IA-32 und nachfolgenden Architekturen einen Maschinenbefehl: `cpuid`.

Dieser Befehl ermöglicht es, weitreichende Informationen über die verwendete Konfiguration einzuholen, beispielsweise unterstützte Befehlssatzerweiterungen wie MMX, SSE, SSE2 oder SSE3, aber auch Größe und Assoziativität der Caches.

Am Rande der Bearbeitung dieser Studienarbeit ist eine sehr einfache Bibliothek entstanden, die es ermöglicht zu überprüfen, ob MMX, SSE und SSE2 vorhanden sind. Es wäre nicht schwierig, diese Bibliothek zu erweitern, um weitere Informationen über die verwendete CPU einzuholen.

Das Fernziel dabei ist es, adaptive Programme zu ermöglichen, die Parameter wie beispielsweise die Blockgröße bei Loop-Blocking-Methoden automatisch der gerade verwendeten Hardware anpassen.

Das Kernstück der Bibliothek ist eine Routine mit wenigen Zeilen Inline-Assembler:

```
1  /* Globale Variablen zum Speichern der Registerwerte */
2  unsigned int reg_eax, reg_ebx, reg_ecx, reg_edx;
3
4  static void callcpuid(unsigned int eax) {
5    asm(" movl %0, %%eax " :: "m" (eax) );
6    asm(" cpuid ");
7    asm(" mov %%eax, %0 " : "=m" (reg_eax) );
8    asm(" mov %%ebx, %0 " : "=m" (reg_ebx) );
9    asm(" mov %%ecx, %0 " : "=m" (reg_ecx) );
10   asm(" mov %%edx, %0 " : "=m" (reg_edx) );
11 }
```

Quellcodeausschnitt D.1: Benutzung des `cpuid`-Befehls

Im Quellcodeausschnitt D.1 bezeichnet der Eingabeparameter `eax` den gewünschten Informationslevel. Zunächst wird der Wert aus dem Eingabeparameter `eax` in das Register EAX geladen. Danach wird der `cpuid`-Befehl aufgerufen und die Ausgabewerte aus den Registern in die entsprechenden globalen Variablen geladen. Dort müssen sie entsprechend verarbeitet werden.

In Tabelle D.1 findet sich eine Kurzbeschreibung der gängigen Informationslevel, für eine umfangreiche Übersicht siehe [San].

Der Quellcodeausschnitt D.2 zeigt als Beispiel, wie getestet werden kann, ob der vorliegende Prozessor die SSE-Erweiterungen unterstützt. Zunächst wird dabei getestet, ob

Level:	Informationen über:
0	Maximalen Informationslevel und Hersteller
1	Prozessor-Familie, -Typ, -Modell und -Stepping, außerdem besondere Erweiterungen
2	Konfiguration des Prozessors (Caches, TLBs, ...)
3	Prozessor-Seriennummer
4	erweiterte Cache-Informationen

Tabelle D.1: Verschiedene Ebenen des cpuid-Befehls

Informationslevel 1 verfügbar ist. Sofern dies der Fall ist, wird cpuid mit Level 1 aufgerufen und das ensprechende Bit geprüft.

```
1 int checkSSE(void) {
2   if (cpuidLevel() > 0) {
3     callcpuid(1);
4     return (int) ((reg_edx>>25) & 1);
5   } else {
6     return 0;
7   }
8 }
```

Quellcodeausschnitt D.2: Test auf Verfügbarkeit der SSE-Erweiterungen

Zu beachten ist bei der Benutzung der Bibliothek, dass die GCC-Syntax bei den Assembleraufrufen benutzt wird, die nicht alle aktuellen Compiler unterstützen. Beim Kompilieren sollte außerdem darauf geachtet werden, dass keine Optimierungen benutzt werden, denn hohe Optimierungsstufen führten zu Fehlern (der Informationslevel wurde nicht korrekt gesetzt).

Literaturverzeichnis

[D+99] Carole Dulong et al. An Overview of the Intel IA-64 Compiler. Intel Technology Journal, Q4, 1999. http://www.intel.com/technology/itj/archive/1999.htm.

[Don04] Stefan Donath. On Optimized Implementations of the Lattice Boltzmann Method on Contemporary High Performance Architectures. http://www10.informatik.uni-erlangen.de/Publications/Theses/Donath.pdf, August 2004. Bachelor Thesis, Friedrich Alexander-Universität Erlangen-Nürnberg, Institut für Informatik, Lehrstuhl für Informatik 10 (Systemsimulation).

[Hau05] Simon Hausmann. Optimization and Performance Analysis of the Lattice Boltzmann Method on x86-64 based Architectures. http://www10.informatik.uni-erlangen.de/Research/Projects/DiME-new/pubs/hausmann.pdf, April 2005. Bachelor Thesis, Friedrich Alexander-Universität Erlangen-Nürnberg, Institut für Informatik, Lehrstuhl für Informatik 10 (Systemsimulation).

[HP02] John L. Hennessy and David A. Patterson. *Computer architecture: a quantitative approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002.

[Int04a] Intel Corporation. *IA-32 Intel Architecture Optimization Reference Manual*, 2004. Document Number 248966-011,http://developer.intel.com.

[Int04b] Intel Corporation. *Intel® C++ Compiler for Linux Systems User's Guide*, August 2004. Document Number 253254-031,http://developer.intel.com.

[Kra01] Manfred Krafczyk. Gitter-Boltzmann-Methoden: Von der Theorie zur Anwendung. http://www.cab.bau.tu-bs.de/institut/mitarbeiter/grund/krafczyk/kr_habilit.htm, Januar 2001. Habilitationsschrift, Technische Universität München, Lehrstuhl für Bauinformatik.

[Lei95] Ernst L. Leiss. *Parallel and vector computing: A practical introduction*. McGraw-Hill, Inc, 1995.

[QdL92] Y.H. Qian, D. d'Humieres, and P. Lallemand. Lattice BGK for Navier-Stokes equation. *Europhysics Letters*, 17(6):479–484, 1992.

[San] IA-32 architecture CPUID. http://www.sandpile.org/ia32/cpuid.htm. Zuletzt besucht: 09. September 2005.

[Top05] Top500 Supercomputer Sites. http://www.top500.org/, Juni 2005.

[WG00] Dieter A. Wolf-Gladrow. *Lattice gas cellular automata and lattice Boltzmann models*. Springer, 2000.

Erklärung

Hiermit versichere ich, diese Arbeit selbstständig verfasst und nur die angegebenen Quellen benutzt zu haben.

Unterschrift:

Gärtringen, den 09. November 2005