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

Lehrstuhl für Informatik 10 (Systemsimulation)

Optimierung des Red-Black-Gauss-Seidel-Verfahrens auf ausgewählten x86-Prozessoren

Markus Stürmer

Studienarbeit

Optimierung des Red-Black-Gauss-Seidel-Verfahrens auf ausgewählten x86-Prozessoren

Markus Stürmer

Studienarbeit

Aufgabensteller:	Prof. Dr. U. Rüde
Betreuer:	Dipl.-Inf. J. Treibig
Bearbeitungszeitraum:	Dezember 2004 – August 2005

Erklärung:

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

Erlangen, den 4. August 2005 .....

Inhaltsverzeichnis

Kapitel 1

Motivation

Die numerische Simulation von physikalischen Zusammenhängen ist heute ein unentbehrliches Werkzeug für Wissenschaft und Wirtschaft, ist aber wohl eine der rechenintensivsten Anwendungen. Sie ermöglicht Experimente am Computer nachzustellen, die in der Praxis nur mit großem Aufwand oder gar nicht durchgeführt werden können.

Bei der numerischen Simulation werden die physikalischen Zusammenhänge zunächst in mathematischen Modellen erfasst. Diese sind im Allgemeinen nicht analytisch lösbar, und man beschränkt sich auf eine angenäherte Lösung an endlich vielen Punkten. Meist resultiert dies in sehr großen Gleichungssystemen, die entweder direkt oder iterativ gelöst werden müssen. Dies ist üblicherweise der für den Computer aufwendigste Teil einer Simulation, sodass durch dessen Optimierung Probleme schneller oder detaillierter behandelt werden können.

Diese Arbeit beschäftigt sich vorwiegend mit der Optimierung des Red-Black Gauss-Seidel-Verfahrens, das eine Vielzahl von linearen Gleichungssystemen iterativ lösen kann, auf zwei verbreiteten Prozessorarchitekturen. Dabei wird besonders auf in aktuellen Prozessoren verwirklichte Konzepte und ihre Folgen auf Auswahl und Anwendung der Optimierungstechniken eingegangen, um eine Übertragung auf ähnliche Algorithmen und folgende Prozessorgenerationen zu ermöglichen.

Für die Beschäftigung mit dem Red-Black Gauss-Seidel spricht nicht nur seine Vielseitigkeit und die Möglichkeit, ihn relativ einfach in Clustern zu parallelisieren. Er wird auch häufig bei Multi-Grid-Verfahren eingesetzt, die zu den schnellsten verfügbaren Lösungsverfahren zählen. Obwohl hierbei nur wenige Iterationen auf denselben Daten zum Glätten der Fehler notwenig sind, wird hierfür die meiste Zeit verwendet. [Wei01]

Kapitel 2

Modellproblem und Red-Black Gauss-Seidel

Als Modellproblem, um den Red-Black Gauss-Seidel implementieren und optimieren zu können, wurde für diese Arbeit die Poisson-Gleichung

$$\Delta\phi = f$$

gewählt, eine partielle Differentialgleichung zweiter Ordnung. Das aus der Gleichung folgende Randwertproblem kann viele physikalische Zusammenhänge beschreiben und ist deshalb im Bereich der Simulation häufig anzutreffen, z.B. bei der Berechnung von elektrischen und Gravitationsfeldern, sie modelliert aber auch die Verteilung und Auswirkung des Drucks bei Strömungen. Δ ist hierbei der Laplace-Operator, der definiert ist als

$$\Delta = \sum_k \frac{\delta^2}{\delta x_k^2}$$

Um die Algorithmen und ihre Optimierung verstehen zu können, werden in diesem Kapitel zunächst die Grundlagen des Gauss-Seidel-Verfahrens vermittelt. Anschließend wird die Diskretisierung des Modellproblems im zwei- und dreidimensionalen Fall und die Anwendung des Gauss-Seidel-Verfahrens beschrieben. Zuletzt wird gezeigt, wie man vom allgemeinen Gauss-Seidel-Verfahren zum Red-Black Gauss-Seidel gelangt, und es werden einfache Implementationen für den zwei- und dreidimensionalen Fall vorgestellt.

2.1 Gauss-Seidel-Verfahren

Das Gauss-Seidel-Verfahren kann zur Lösung von linearen Gleichungssystemen der Form

$$Ax = y$$

genutzt werden.

Dabei wird die Matrix A nun zerlegt in

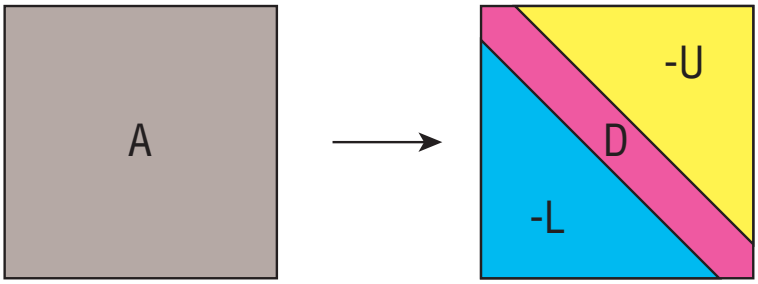
$$A = D - L - U$$

wobei D aus den Werten der Hauptdiagonalen besteht, und L und U eine linke und obere strenge Dreiecksmatrix sind (vgl. Abbildung 2.1).

Durch Einsetzen ergibt sich dann:

$$(D - L) \cdot x = U \cdot x + y$$

Abbildung 2.1: Zerlegung von A in D, L und U



Das daraus abgeleitete Iterationsverfahren

$$D \cdot x^{k+1} - L \cdot x^{k+1} = U \cdot x^k + y$$

lässt sich dann – da L und U Dreiecksmatrizen sind – auch darstellen als

$$d_{i,i} \cdot x_i^{k+1} - \sum_{j=1}^{i-1} l_{j,i} \cdot x^{k+1} = \sum_{j=i+1}^{i_{max}} u_{j,i} \cdot x^k + y_i$$

Da die Werte von D, L und U aus A entstanden sind, entspricht dies

$$x_i^{k+1} = \frac{\sum_{j=1}^{i-1} -a_{j,i} \cdot x_j^{k+1} + \sum_{j=i+1}^{i_{max}} -a_{j,i} \cdot x_j^k + y_i}{a_{i,i}}$$

Auf eine genaue Untersuchung, wann und wie schnell das Verfahren konvergiert, wird hier verzichtet und auf die einschlägige Literatur der Linearen Algebra verwiesen. Das Gauss-Seidel-Verfahren ist allgemein anwendbar und konvergiert, wenn die Matrix A positiv definit oder streng diagonal dominant ist.

Die Iterationsregel lässt sich nun folgendermaßen interpretieren: Ausgehend von einer beliebigen Anfangslösung x^0 werden die Näherungen x^1 bis x^n berechnet. Zumindest im allgemeinen Fall beginnt jeder Iterationsschritt mit der Berechnung von x_1^{k+1}, da dieses ausschließlich von Werten aus x^k abhängt. Danach können sukzessive x_2^{k+1}, das von x_1^{k+1} abhängen kann, bis $x_{i_{max}}^{k+1}$ berechnet werden. Wirklich effizient ist dieses Verfahren allerdings nur, wenn A dünn besetzt ist, und dies auch zur Reduzierung der Berechnungen genutzt werden kann. Meist kennt man allerdings die Struktur des Gleichungssystems und muss es nicht explizit aufstellen. Die Berechnung eines neuen Wertes x_i^{k+1} entspricht nichts anderem, als die Einzelgleichung, die zu Zeile i in A geführt hat, lokal zu lösen; dafür werden sowohl die in der aktuellen Iteration schon berechneten Werte x_1^{k+1} bis x_{i-1}^{k+1}, als auch x_i^k bis $x_{i_{max}}^k$ aus der vorhergehenden verwendet.

2.2 Diskretisierung der Poisson-Gleichung

Zur iterativen Lösung der Poisson-Gleichung

$$\Delta\phi = f$$

mit Randwertbedingungen

$$\phi(\vec{x}) = \Gamma(\vec{x}) \quad \forall \vec{x} \in \delta\Omega$$

muss diese zunächst auf einem Gitter diskretisiert werden.

Es wird hier davon ausgegangen, dass Ω rechtwinklig begrenzt ist, also im Zweidimensionalen rechteckig und im Dreidimensionalen quaderförmig ist. Des weiteren soll eine Diskretisierung auf einem regelmäßigen Gitter möglich sein, die Größe von Omega muss dafür in jeder Dimension ein Vielfaches vom Gitterabstand dh sein.

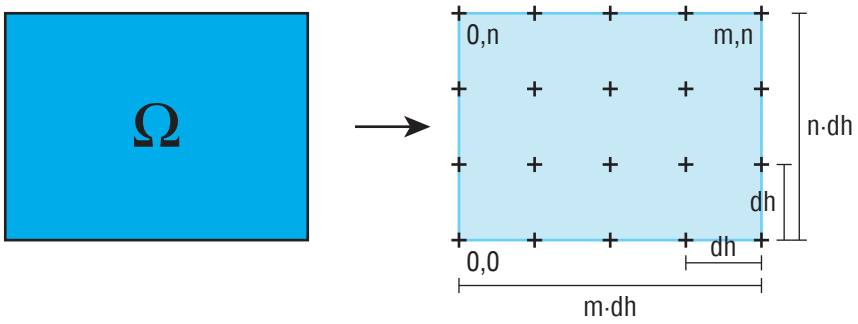
2.2.1 2D Fall

Die zweidimensionale Poissongleichung

$$\frac{\delta^2}{\delta x^2}\phi(x,y) + \frac{\delta^2}{\delta y^2}\phi(x,y) = f(x,y)$$

wird auf dem Gitter der Größe $m \cdot dh$ in x-Richtung und $n \cdot dh$ in y-Richtung diskretisiert. Es entstehen die Werte $U(i,j)$ mit $i = 0..m$ und $j = 0..n$ aus ϕ und aus f die Werte $F(i,j)$ mit $i = 1..m-1$ und $j = 1..n-1$. Die Werte $U(i,j)$ enthalten für $i = 0 \vee i = m \vee j = 0 \vee j = n$ die diskretisierten Randwerte (vgl. Abbildung 2.2).

Abbildung 2.2: Diskretisierung von Ω in 2D



Die Ableitungen zweiter Ordnung für den Laplace-Operator können durch die Methode der Finiten Differenzen im diskreten Fall angenähert werden:

Die Addition der Taylor-Reihen

$$\phi(x+h,y) = \phi(x,y) + h \cdot \frac{\delta}{\delta x}\phi(x,y) + h^2 \cdot \frac{\delta^2}{\delta x^2}\phi(x,y) + h^3 \cdot \frac{\delta^3}{\delta x^3}\phi(x,y) + \mathcal{O}(h^4)$$

$$\phi(x-h,y) = \phi(x,y) - h \cdot \frac{\delta}{\delta x}\phi(x,y) + h^2 \cdot \frac{\delta^2}{\delta x^2}\phi(x,y) - h^3 \cdot \frac{\delta^3}{\delta x^3}\phi(x,y) + \mathcal{O}(h^4)$$

ergibt dann nach Division der Summe durch h^2:

$$\frac{\phi(x-h,y) - 2 \cdot \phi(x,y) + \phi(x+h,y)}{h^2} = \frac{\delta^2}{\delta x^2}\phi(x,y) + \mathcal{O}(h^2)$$

und analog

$$\frac{\phi(x,y-h) - 2 \cdot \phi(x,y) + \phi(x,y+h)}{h^2} = \frac{\delta^2}{\delta y^2}\phi(x,y) + \mathcal{O}(h^2)$$

Der zweidimensionale Laplace-Operator kann dann angenähert werden durch

$$\Delta^\star{}_{2D} = \frac{\phi(x-h,y) - 2 \cdot \phi(x,y) + \phi(x+h,y)}{h^2} + \frac{\phi(x,y-h) - 2 \cdot \phi(x,y) + \phi(x,y+h)}{h^2}$$

Setzt man $h = dh$ und vernachlässigt die Diskretisierungsfehler, ergibt sich das lineare Gleichungssystem

$$\frac{U(i-1,j) - 2 \cdot U(i,j) + U(i+1,j)}{dh^2} + \frac{U(i,j-1) - 2 \cdot U(i,j) + U(i,j+1)}{dh^2} = F(i,j)$$

oder anders ausgedrückt

$$U(i,j) = \frac{1}{4} \cdot \left[U(i+1,j) + U(i-1,j) + U(i,j+1) + U(i,j-1) - dh^2 \cdot F(i,j)\right] \quad 0 < i < m, 0 < j < n$$

Das Gleichungssystem lässt sich in Matrixschreibweise darstellen als

$$Ax = y$$

wobei A eine Matrix der Größe $((m-1)\cdot(n-1))\times((m-1)\cdot(n-1))$ ist. A, x und y können zum Beispiel folgendermaßen belegt sein:

$$x = \begin{pmatrix} U(1,1) \\ \vdots \\ U(m-1,1) \\ U(1,2) \\ \vdots \\ \vdots \\ U(m-1,n-1) \end{pmatrix}$$

x enthält somit die sortierten Unbekannten von U.

$$A = \begin{pmatrix} A_1 & I & & & \\ I & A_2 & I & & \\ & I & \ddots & \ddots & \\ & & \ddots & \ddots & I \\ & & & I & A_{n-1} \end{pmatrix} \text{ mit } A_n = \begin{pmatrix} -4 & 1 & & & \\ 1 & -4 & 1 & & \\ & 1 & \ddots & \ddots & \\ & & \ddots & \ddots & 1 \\ & & & 1 & -4 \end{pmatrix}$$

Die I sind Einheitsmatrizen, die ebenso wie die A_n die Größe $(m-1)\times(m-1)$ besitzen. Die Werte auf der Hauptdiagonalen sind ausschließlich -4, die restlichen Werte $a_{i,j}$ sind dort 1, wo x_j ein Nachbar des durch x_i spezifierten U sind, sonst 0.

$$y = \begin{pmatrix} F(1,1) - r_1(1,1) - r_2(1,1) \\ \vdots \\ F(m-1,1) - r_1(m-1,1) - r_2(m-1,1) \\ \vdots \\ \vdots \\ F(m-1,n-1) - r_1(m-1,n-1) - r_2(m-1,n-1) \end{pmatrix}$$

$$\text{mit } r_1(i,j) = \begin{cases} U(0,j) & \text{für } i = 1 \\ U(m,j) & \text{für } i = m-1 \\ 0 & \text{sonst} \end{cases}$$

$$\text{und } r_2(i,j) = \begin{cases} U(i,0) & \text{für } j = 1 \\ U(i,n) & \text{für } j = n-1 \\ 0 & \text{sonst} \end{cases}$$

y besteht aus den wie die U in x sortierten diskretisierten Werten aus F, wobei bei Werten am Rand die anliegenden Randwerte abgezogen werden müssen.

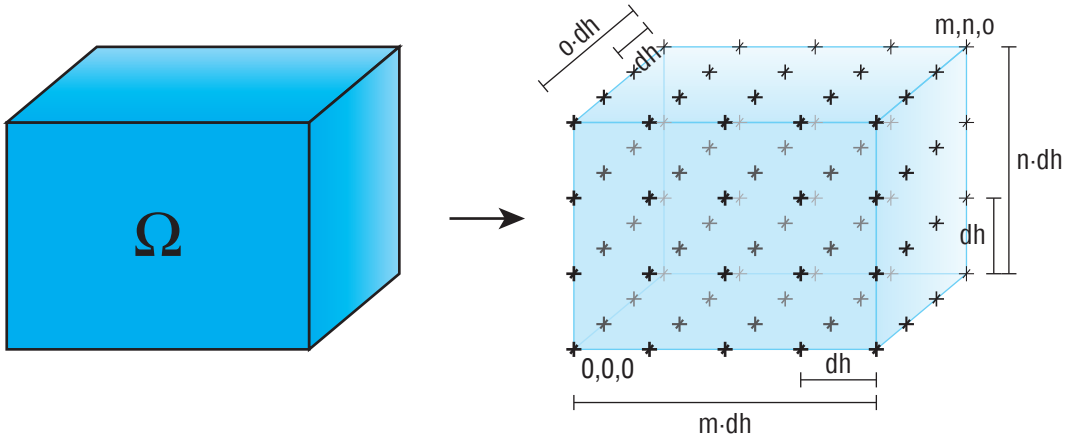
2.2.2 3D Fall

Mit ausgeschriebenem Laplace-Operator lautet die Poisson-Gleichung im dreidimensionalen Fall

$$\frac{\delta^2}{\delta x^2}\phi(x,y,z) + \frac{\delta^2}{\delta y^2}\phi(x,y,z) + \frac{\delta^2}{\delta z^2}\phi(x,y,z) = f(x,y,z)$$

Die Diskretisierung wird analog zum zweidimensionalen Fall durchgeführt (vgl. Abbildung 2.3) und es entstehen aus ϕ die Werte $U(i,j,k)$ mit $i = 0..m$, $j = 0..n$, $k = 0..o$ und $F(i,j,k)$ mit $i = 1..m-1$, $j = 1..n-1$ und $k = 1..o-1$.

Abbildung 2.3: Diskretisierung von Ω in 3D



Auch der diskretisierte dreidimensionale Laplace-Operator kann analog angenähert werden, indem die Finiten Differenzen auch in z-Richtung gebildet und eingesetzt werden.

$$\begin{aligned}\Delta^{\star}{}_{3D} = & \frac{\phi(x-h,y,z)-2\cdot\phi(x,y,z)+\phi(x+h,y,z)}{h^2} \\ & +\frac{\phi(x,y-h,z)-2\cdot\phi(x,y,z)+\phi(x,y+h,z)}{h^2} \\ & +\frac{\phi(x,y,z-h)-2\cdot\phi(x,y,z)+\phi(x,y,z+h)}{h^2}\end{aligned}$$

Das Gleichungssystem ergibt sich dann zu

$$\begin{aligned}F(i,j,k) = & \frac{U(i-1,j,k)-2\cdot U(i,j,k)+U(i+1,j,k)}{dh^2} \\ & +\frac{U(i,j-1,k)-2\cdot U(i,j,k)+U(i,j+1,k)}{dh^2} \\ & +\frac{U(i,j,k-1)-2\cdot U(i,j,k)+U(i,j,k+1)}{dh^2}\end{aligned}$$

oder

$$\begin{aligned}U(i,j,k) = & \frac{1}{6}\cdot\big[U(i+1,j,k)+U(i-1,j,k)+U(i,j+1,k)+U(i,j-1,k \\ & +U(i,j,k+1)+U(i,j,k-1)-dh^2\cdot F(i,j,k)\big]\end{aligned}$$

mit $0<i<m$, $0<j<n$, $0<k<o$

Die Matrixgleichung $Ax=y$ lässt sich dann aufstellen mit

$$A=\begin{pmatrix} A_1^{2D} & I & & & \\ I & A_2^{2D} & I & & \\ & I & \ddots & \ddots & \\ & & \ddots & \ddots & I \\ & & & I & A_{o-1}^{2D}\end{pmatrix}$$

Die A_n^{2D} entsprechen einer Matrix A aus dem zweidimensionalen Fall mit $m_{2D} := m_{3D}$ und $n_{2D} := n_{3D}$.

$$x = \begin{pmatrix} U(1,1,1) \\ \vdots \\ U(m,1,1) \\ U(1,2,1) \\ \vdots \\ U(m,n,1) \\ U(1,1,2) \\ \vdots \\ U(m-1,n-1,o-1) \end{pmatrix}$$

y enthält – wie im zweidimensionalen Fall – die diskretisierten Werte aus F in derselben Sortierung wie die U in x, wobei an den Ecken, Kanten und Seiten die anliegenden Randwerte abgezogen werden müssen.

2.3 Anwendung auf das Modellproblem und Variante des Red-Black Gauss-Seidel

Der Beweis, dass die durch die oben beschriebenen Verfahren erzeugten A positiv definit sind, kann z. B. in [Hac93] nachgelesen werden.

Bei der Implementierung des Gauss-Seidel-Verfahrens ist es nicht nötig, das Gleichungssystem tatsächlich explizit aufzustellen. Entscheidend ist nur, die neuen Werte von U in der Reihenfolge im gedachten x neu zu berechnen. Da nach seiner Berechnung in U^{k+1} der Wert aus U^k nicht mehr benötigt wird, kann dieser im Speicher überschrieben werden.

Beim normalen Gauss-Seidel-Verfahren wird die Matrix-Gleichung wie in den Abschnitten 2.2.1 und 2.2.2 zugrunde gelegt. Die damit zusammenhängende Abarbeitungsreihenfolge beginnt zunächst mit der Berechnung eines neuen $U(1,1)$ bzw. $U(1,1,1)$ und fährt dann mit $U(2,1)$ bzw. $U(2,1,1)$ die ganze Zeile bis $U(m-1,1)$ bzw. $(m-1,1,1)$ hindurch fort. Anschließend wird die Zeile $U(x,2)$ bzw. $U(x,2,1)$ berechnet und so weiter, bis im zweidimensionalen Fall das ganze Gitter oder in 3D eine ganze Ebene fertiggestellt ist. Im dreidimensionalen Fall wird dann Ebene für Ebene ebenso verfahren.

Dem Red-Black Gauss-Seidel liegt gedanklich eine andere Sortierung der Werte U in x zugrunde. Hierzu wird das Innere des Gitters U schachbrettartig in sogenannte rote und schwarze Punkte aufgeteilt, wobei für den zweidimensionale Fall üblicherweise gilt

$$U(i,j) \quad \begin{cases} \text{rot für } (i+j)\ mod\ 2 = 0 \\ \text{schwarz für } (i+j)\ mod\ 2 = 1 \end{cases}$$

und im dreidimensionalen

$$U(i,j,k) \quad \begin{cases} \text{rot für } (i+j+k)\ mod\ 2 = 1 \\ \text{schwarz für } (i+j+k)\ mod\ 2 = 0 \end{cases}$$

Die Reihenfolge der Berechnungen entspricht einer Matrixgleichung, bei der x_1 bis x_i nur roten und x_{i+1} bis $x_{i_{max}}$ nur schwarzen Punkten entspricht. In diesem Fall hängen die ersten i roten Werte ausschließlich von schwarzen Werten aus x^k ab, die folgenden schwarzen Werte ausschließlich von roten Werten aus x^{k+1}. Eine Iteration teilt sich damit in eine rote und eine schwarze Halbiteration auf. Für eine Implementierung kann vorteilhaft sein, dass die Berechnungen innerhalb einer Halbiterationen voneinander unabhängig sind und in beliebiger Reihenfolge oder parallel abgearbeitet werden können.

Der Red-Black Gauss-Seidel ist demnach nur eine bestimmte Methode, das Gauss-Seidel-Verfahren anzuwenden. Während beim Modellproblem das Gitter in zwei „Farben“ aufgeteilt werden kann, so kann im Allgemeinen eine Aufteilung nur mit mehreren Gruppen oder gar nicht erreicht werden.

2.4 Implementierung

Die beschriebenen Gauss-Seidel-Verfahren lassen sehr direkt implementieren. Die auf dem Gitter diskretisierten Werte U und F werden dazu in mehrdimensionalen Arrays gespeichert.

In der Praxis wird man die Größe von F genauso wählen wie die von U. Denn zum einen unterstützen viele Programmiersprachen keine beliebigen Arraygrenzen, zum anderen erleichtert dies dem Compiler die Adressberechnung; der für eine Berechnung benötigte Wert aus F und ihr Ziel in U haben damit nämlich dieselbe Adresse relativ zum Anfang der Arrays.

beispielhafte Implementation des 2D-Gauss-Seidel

```
#Arrays der benötigten Größe anlegen
alloc array U[0..m;0..n]
alloc array F[1..m-1;1..n-1]

#diskrete Werte auf dem Gitter in U und F berechnen
call init

#gewünschte Anzahl an Gauss-Seidel-Iterationen ausführen
repeat number of iterations times
   for j=1..(n-1)
      for i=1..(m-1)
         U[i;j]=( U[i-1;j]+U[i+1;j]+U[i;j-1]+U[i;j+1] \
                 -dh*dh*F[i;j] )/4.0
      next i
   next j
end repeat
```

beispielhafte Implementation des 3D-Gauss-Seidel

```
#Arrays der benötigten Größe anlegen
alloc array U[0..m;0..n;0..o]
alloc array F[1..m-1;1..n-1;1..o-1]

#diskrete Werte auf dem Gitter in U und F berechnen
call init

#gewünschte Anzahl an Gauss-Seidel-Iterationen ausführen
repeat number of iterations times
   for k=1..(o-1)
      for j=1..(n-1)
         for i=1..(m-1)
            U[i;j]=( U[i-1;j;k]+U[i+1;j;k]+U[i;j-1;k] \
                    +U[i;j+1;k]+U[i;j;k-1]+U[i;j;k+1] \
                    -dh*dh*F[i;j] )/6.0
         next i
      next j
   next k
end repeat
```

beispielhafte Implementation des 2D-Red-Black-Gauss-Seidel

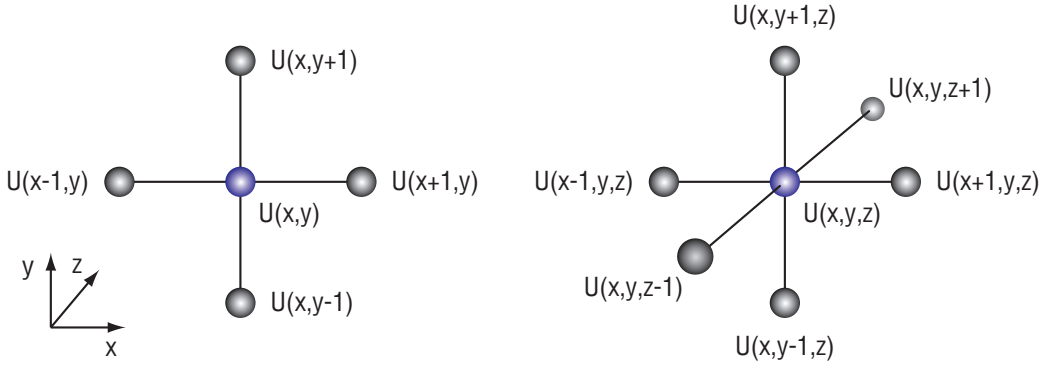
```
#Arrays der benötigten Größe anlegen
alloc array U[0..m;0..n]
alloc array F[1..m-1;1..n-1]

#diskrete Werte auf dem Gitter in U und F berechnen
call init

#gewünschte Anzahl an Gauss-Seidel-Iterationen ausführen
repeat number of iterations times
   # rote Halbiteration
   for j=1..(n-1)
      for i=1..(m-1)
         if ( (i+j) mod 2 = 0) then
            U[i;j]=( U[i-1;j]+U[i+1;j]+U[i;j-1]+U[i;j+1] \
                    -dh*dh*F[i;j] )/4.0
         fi
      next i
   next j

   # schwarze Halbiteration
   for j=1..(n-1)
      for i=1..(m-1)
         if ( (i+j) mod 2 = 1) then
            U[i;j]=( U[i-1;j]+U[i+1;j]+U[i;j-1]+U[i;j+1] \
                    -dh*dh*F[i;j] )/4.0
         fi
      next i
   next j
end repeat
```

beispielhafte Implementation des 3D-Red-Black-Gauss-Seidel

```
#Arrays der benötigten Größe anlegen
alloc array U[0..m;0..n,0..o]
alloc array F[1..m-1;1..n-1;1..o-1]

#diskrete Werte auf dem Gitter in U und F berechnen
call init

#gewünschte Anzahl an Gauss-Seidel-Iterationen ausführen
repeat number of iterations times
   # rote Halbiteration
   for k=1..(o-1)
      for j=1..(n-1)
         for i=1..(m-1)
            if ( (i+j+k) mod 2 = 1) then
               U[i;j;k]=( U[i-1;j;k]+U[i+1;j;k]+U[i;j-1;k] \
                        +U[i;j+1;k]+U[i;j;k-1]+U[i;j;k+1] \
                        -dh*dh*F[i;j;k] )/6.0
            fi
         next i
      next j
   next k

   # schwarze Halbiteration
   for k=1..(o-1)
      for j=1..(n-1)
         for i=1..(m-1)
            if ( (i+j+k) mod 2 = 0) then
               U[i;j;k]=( U[i-1;j;k]+U[i+1;j;k]+U[i;j-1;k] \
                        +U[i;j+1;k]+U[i;j;k-1]+U[i;j;k+1] \
                        -dh*dh*F[i;j;k] )/6.0
            fi
         next i
      next j
   next k
end repeat
```

Aufgrund seiner „Form“ wird der für die Berechnung neuer Werte benötigte, durch Finite Differenzen angenäherte, diskrete Laplace-Operator häufig als 5-Punkt- im zweidimensionalen bzw. 7-Punkt-Stempel im dreidimensionalen Fall bezeichnet (vgl. Abbildung 2.4).

Abbildung 2.4: Der „Laplace-Stempel“



Da man die beschriebenen Verfahren sowohl aus einer mathematischen als auch rein algorithmischen Sicht betrachten kann, sollte noch auf die unterschiedliche Verwendung bestimmter Begriffe und Bezeichnungen hingewiesen werden. Insbesondere ist der Begriff der Größe eines Problems nicht eindeutig. Geht man von einer diskretisierten partiellen Differentialgleichung aus, wird man die Problemgröße an Ω festmachen und von einem $m \times n$ bzw. $m \times n \times o$ großen Problem sprechen. Ausgehend von einer rein algorithmischen oder allgemein auf das Lösen von Gleichungssystemen bezogenen Sicht ist dasselbe Problem allerdings $(m-1) \times (n-1)$ bzw. $(m-1) \times (n-1) \times (o-1)$ groß, da dies die Menge der Unbekannten darstellt. In dieser Arbeit wird die letztere Art der Bezeichnung bevorzugt.

Zum anderen ist es üblich, sich auf die Nachbarn des Wertes $U_{i,j}$ bzw. $U_{i,j,k}$ der Einfachheit halber über eine Form von Richtungsangabe zu beziehen, wobei sich hier keine eindeutige und für zwei- und dreidimensionale Gitter einheitliche Bezeichnung durchgesetzt hat. In dieser Arbeit werden die Dimensionen mit der üblichen Vorstellung von programmiersprachlichen mehrdimensionalen Feldern im Speicher assoziiert. Somit gilt: $U_{x+1,y}$ *rechts* von $U_{x,y}$, $U_{x,y+1}$ *unter* $U_{x,y}$ und im Dreidimensionalen $U_{x,y,z+1}$ *hinter* $U_{x,y,z}$.

Kapitel 3

Wichtige Konzepte zur Steigerung der Prozessorleistung

Während es durch die Verbesserung der Fertigungsverfahren von Halbleitern möglich ist, immer mehr Schaltungen auf kleinerem Raum unterzubringen, so verdanken moderne Prozessoren ihre hohe Geschwindigkeit viel mehr Konzepten, die die dadurch mögliche höhere Komplexität effektiv in schnellere Verarbeitung verwandeln.

Zudem werden Hauptspeicher zwar immer größer, doch konnte ihre Fähigkeit, den Prozessor auch rechtzeitig und ausreichend mit Daten zu versorgen, nicht entsprechend gesteigert werden. Deshalb wurden verschiedene Methoden entwickelt, um die großen Rechenkapazitäten besser zu nutzen.

In diesem Kapitel sollen einige dieser Konzepte und Methoden vorgestellt werden, die für das Verständnis von Optimierungstechniken notwendig sind. Deren Kenntnis wird bei der Vorstellung der Testsysteme im folgenden Kapitel vorausgesetzt.

3.1 Erhöhung des Instruktionsdurchsatzes

3.1.1 Pipelining und Superpipelining

Die Abarbeitung einer Instruktion umfasst im allgemeinen Laden und Dekodieren der Instruktion, Laden der Operanden, Ausführung der Operation und Schreiben des Ergebnisses. Beim Fließbandverfahren (*Pipelining*) beginnt die Abarbeitung der nächsten Operation bereits, wenn der erste Schritt der Abarbeitung des vorhergehendes Befehls beendet ist. Im Idealfall findet auf jeder Stufe der Pipeline die Teilverarbeitung jeweils verschiedener Instruktionen gleichzeitig statt.

Es besteht auch die Möglichkeit, diese Stufen der Pipeline noch feiner zu granulieren; die Schaltungen für den einzelnen Bearbeitungsschritt werden damit weniger komplex und können dadurch mit einer höheren Taktfrequenz betrieben werden. Dies wird als *Superpipelining* bezeichnet.

Pipelining und Superpipelining führen damit nicht einzelne Befehle schneller aus, sondern ermöglichen durch eine gewisse Parallelität unter günstigen Umständen einen deutlich höheren Instruktionsdurchsatz. Sind für einen Befehl allerdings die Operanden nicht rechtzeitig vorhanden (weil sie aus unteren Speicherebene geladen werden oder von anderen in der Pipeline befindlichen Operationen abhängen), müssen Teile des Fließbandes vorübergehend angehalten werden (*Pipeline Stalls*). Noch gravierender ist das Problem bei bedingten Sprüngen, da hier erst dann der nächste Befehl geholt werden kann, wenn diese Instruktion vollständig abgearbeitet und damit das Sprungziel bekannt ist.

3.1.2 Speculative Execution

Da bei bedingten Sprüngen die Pipeline einmal vollständig geleert wird, verliert Pipelining bei kurzen Schleifen oder verschachtelten Kontrollstrukturen seine große Effizienz. Deshalb ist man dazu übergegangen, bei bedingten Sprüngen einen Entscheidungszweig zunächst „auf Verdacht“ zu bearbeiten (*Speculative Execution*). Allerdings: Wenn der falsche Zweig gewählt wurde, müssen die begonnenen Berechnungen in der Pipeline verworfen, und es muss mit den richtigen Instruktionen neu begonnen werden.

Meist ist entweder spezifiziert, bei welchen Befehlen welcher Zweig bevorzugt wird, oder es existieren mehrere Instruktionsvarianten, bei denen jeweils ein bestimmter Zweig spekulativ ausgeführt wird. Auf jeden Fall müssen Compiler oder Programmierer dies bei der Code-Generierung beachten, um eine möglichst schnelle Ausführung zu ermöglichen.

Auch eine Einheit zum Voraussagen der Sprungziele (*Branch Prediction*) ist üblich, die bei mehrfacher Ausführung den statistisch wahrscheinlicheren Zweig ermittelt.

Der Vollständigkeit halber sollte eine spezielle Form der Branch Prediction erwähnt werden, die häufig bei Unterfunktionsaufrufen angewendet wird: Während „return“-Anweisungen immer Sprünge ausführen, hängt dieses von der im Stapel gespeicherten Rücksprungadresse ab. Die letzten Rücksprungadressen werden zur schnellere Verfügbarkeit in einem Puffer gespeichert. Da die Rücksprungadresse, obwohl schlechter Programmierstil, im Stapelspeicher geändert worden sein kann, muss der Sprung auf die gepufferte Adresse zunächst spekulativ ausgeführt werden.

3.1.3 Out of Order Execution

Um auch Verzögerung durch noch nicht verfügbare Operatoren zu reduzieren, können moderne Prozessoren auch die Abarbeitung von Instruktionen vorziehen, deren Operatoren bereits verfügbar sind (*Out of Order Execution*).

Hierbei werden die ursprünglichen Maschinensprachebefehle nach dem Dekodieren zunächst in eine Warteschlange gestellt. Ein Scheduler wählt hieraus Instruktionen, deren Operanden vorhanden sind und für die er eine Abhängigkeit mit laufenden Instruktionen ausschließen kann. Diese übergibt er dann den dafür vorgesehenen Recheneinheiten. Nach Ende der Berechnung muss allerdings sichergestellt werden, dass die Instruktionen in ihrer logischen Reihenfolge wirksam werden (*Retirement*).

Die Möglichkeiten des Schedulers werden dadurch weiter unterstützt, dass entgegen der Programmierer-Sicht eine größere Anzahl von Registern vorhanden (*Register File*) ist. So können voneinander unabhängige, aber für dasselbe Register vorgesehene Instruktionen parallel ausgeführt werden. Dieses Verfahren kann Daten-Abhängigkeiten, die nur scheinbar wegen der geringen Registeranzahl bestehen, bei der Programmausführung beseitigen und wird als *Register Renaming* bezeichnet.

3.1.4 Superscalarity

Superskalare Prozessoren beinhalten für jede Operationsart nicht nur eine, sondern mehrere Recheneinheiten. Bei entsprechender Versorgung mit Instruktionen und Operanden und dem Fehlen von Abhängigkeiten kann der Scheduler dadurch in einem Takt die Berechnung mehrerer Instruktionen gleichzeitig starten. Superskalare Prozessoren sind damit in der Lage, effektiv mehr als eine Operation pro Takt auszuführen.

3.1.5 Chaining

Unter *Chaining* versteht man Verfahren, die die Verfügbarkeit benötigter Operanden beschleunigen. Muss eine Instruktion z. B. auf das Ergebnis einer laufenden Berechnung warten, kann dieses sofort nach Abschluss der Berechnung, aber noch vor Retirement und Ausschreiben in Register oder den Hauptspeicher, genutzt werden.

3.1.6 SIMD

Insbesondere bei mathematischen Anwendungen wird dieselbe Operationen auf eine große Menge von gleichartigen Daten angewendet. Bereits um 1960 starteten die ersten Versuche zur Entwicklung von Vektorrechnern, die, anstatt dieselbe Operation immer wieder auf die verschiedenen Werte (Skalare) anzuwenden (Single Instruction Single Data, SISD), dieselbe Operation auf ein ganzes Array mehrerer gleichartiger Werte (Vektor) ausführten (Single Instruction Multiple Data, SIMD).

Mittlerweile sind reine Vektorprozessoren unüblich – meist lässt sich dieselbe Leistung mit Clustern aus üblichen Allzweck-Prozessoren günstiger erreichen. Dennoch hat dieses Konzept zunächst als Vektor-Koprozessoren, später aber auch als in den Prozessor integrierte Vektor- oder SIMD-Einheiten Einzug gefunden.

Die heute üblichen SIMD-Einheiten können Vektoren von bis zu 128 Bit Größe verarbeiten, also z. B. Berechnungen auf vier Fließkommazahlen einfacher Genauigkeit oder logische und arithmetische Operationen auf 1 Byte-Werte ausführen. Dies bietet Vorteile gegenüber einer superskalaren Berechnung: Es muss nur ein Befehl dekodiert und bei Operanden im Speicher nur eine Adresse berechnet werden. Außerdem können diese Vektoreinheiten auch selbst superskalar ausgeführt sein.

3.2 Erhöhung des Speicherdurchsatzes

3.2.1 Caches

Eine der ersten Maßnahmen, die Kluft zwischen dem Bedarf des Prozessors an Instruktionen und Daten auf der einen Seite und der Möglichkeit des Hauptspeichers, diesen auch zu erfüllen, andererseits zu verringern, war die Einführung von Zwischenspeichern (*Caches*) oder Hierarchien von Caches.

Einzelne Caches lassen sich zunächst wie Hauptspeicher durch äußere Eigenschaften wie ihre Größe, ihre Latenz und Bandbreite charakterisieren. Die Latenz entspricht hierbei der Zeit oder der Zahl an Taktzyklen, bis Lese- oder Schreibanfragen erfüllt werden. Die Bandbreite gibt an, welche Datenmenge nach der Latenzzeit dauerhaft übertragen werden kann. Angewandt auf einzelne Algorithmen wird allerdings auch entscheidend, welchen Kompromiss der Hersteller bei Cache-Zeilen-Länge und Assoziativität gewählt hat: Größere Cache-Zeilen benötigen weniger Logik zur Verwaltung und ermöglichen bessere Nutzung lokaler Nähe bei benachbarten Speicherzugriffen, erhähen den tatsächlichen Speicherbedarf aber bei weit verstreuten und einmaligen Zugriffen. Auch höhere Assoziativität kann wegen des steigenden Organisationsbedarfs oft nur auf Kosten der Geschwindigkeit erreicht werden. Zu beachten sind auch die Verdrängungsstrategie und das Verhalten bei Schreiboperationen, also ob die Konsistenz mit der darunter liegenden Speicherebene sofort (*Write-Through*) oder erst bei Verdrängung (*Write-Back*) hergestellt wird.

Insgesamt muss noch die Zusammensetzung der Speicherhierarchie beachtet werden. So ist es heute üblich, auf den unteren Cache-Ebenen Daten für Instruktionen und Operanden gleich zu behandeln (*Unified*), aber zumindest in der höchsten Cacheebene getrennte Caches zu verwenden (*Instruction* und *Data Cache*). Durch der Zugriff auf den Hauptspeicher selbst kann durch die Verwendung von Caches beschleunigt werden: Moderner SD-RAM erreicht seinen höchsten Datentransfer bei größeren, aneinander hängenden Speicherblöcken (*Burst Mode*). Dies kann zum schnellen Füllen oder Ausschreiben der Cache-Zeilen genutzt werden, welche dieser Beschränkung nicht unterliegen.

3.2.2 Prefetching

Während Caches Latenz und Bandbreite bei Nutzung von temporaler und lokaler Nähe deutlich verbessern können, so können sie die Latenz bei Zugriffen auf bisher nicht in Caches vorhandene Speicherbereiche bestenfalls nicht verschlechtern. Es wäre deshalb wünschenswert, wenn Daten sich bereits in einem der Caches befinden, wenn sie tatsächlich von einer Instruktion angefordert werden.

Hierzu werden spezielle Einheiten im oder am Prozessor placiert, die Daten nach Möglichkeit vor ihrer tatsächlichen Anforderung in die Cache-Hierarchie laden (*Hardware Prefetcher*).

Hardware-Prefetcher für Instruktionen sind die einfachere und ältere Variante, da sie nur einem klar definierten und voraussehbaren Lesestrom folgen müssen. Moderne Instruktions-Prefetcher können auch spekulativ zusammen mit einer Branch Prediction arbeiten.

Hardware-Prefetcher für Operanden sind komplexer und können anhand einer Analyse der Speicher-Zugriffsmuster eines laufenden Programmes nur vermutlich benötigte Daten voraussagen. Im Allgemeinen können sie nur eine begrenze Anzahl kontinuierlicher Lese- und Schreibströme erkennen.

Oft besteht auch die Möglichkeit, über spezielle Maschinensprachebefehle Daten in die Caches zu laden oder das Verhalten der Hardware-Prefetcher zu beeinflussen.

3.2.3 Write Combining

Die Organisation der Caches in Zeilen kann insbesondere bei Schreiboperationen zu Geschwindigkeitsverlust führen. Werden einzelne Werte geschrieben und sollen in einem Cache eingelagert werden, muss die gesamte Zeile aus dem Speicher geladen und entsprechend im Cache kombiniert werden. Über *Write Combining Buffer* können aufeinander folgende Schreibzugriffe gesammelt werden. Wenn der Inhalt eines Puffer den kompletten Inhalt einer Cache-Zeile fällt, müssen keine Werte aus dem Hauptspeicher mehr gelesen werden. Bei Multiprozessor- oder zumindest -tauglichen System sind hier dennoch Maßnahmen zur Synchronisation notwendig, damit keine andere Einheiten veraltete Daten aus dem Speicher laden. Besonders effizient ist dieses Verfahren, wenn Daten sequentiell geschrieben werden, und die Anzahl der Schreibströme die Anzahl der Write Combining Buffer nicht übersteigt.

Kapitel 4

Vorstellung der Test-Systeme

4.1 Intel Pentium 4 Prescott 3,2 GHz fauia42.informatik.uni-erlangen.de

Intel bezeichnet mit „Pentium 4 Prozessor“ eigentlich keinen Prozessor, sondern eine ganze Prozessorfamilie zum Einsatz in Servern und leistungsfähigen Desktop-Computern, die im Jahr 2000 mit 1,4 und 1,5 GHz Prozessortakt eingeführt wurde. Die Pentium 4-Prozessoren implementieren Intels NetBurst-Architektur. Deren „Philisophie“ ist es, durch eine lange, fein granulierte Pipeline hohe Taktraten zu ermöglichen. Eine schnelle Ausführung ist dann möglich, wenn die Prozessorsteuerung und das Speichersystems die Pipeline schnell genug mit Instruktionen und Operanden beschicken kann. Dies hängt dann vom auszuführenden Befehlsstrom ab.

Neben der Steigerung der Taktrate und mehreren Ausführungen mit verschiedener Cache-Konfiguration, wurde auch der Prozessorkern überarbeitet und dabei Befehlssatz und Fähigkeiten erweitert. Die Beschreibung bezieht sich deshalb auf die im Testrechner vorhandene Version, der die dritte Überarbeitung des Prozessor-Kerns mit dem Namen Prescott enthält.

4.1.1 Befehls- und Registersatz

Der Intel Pentium 4 führt den IA-32-Befehlssatz (Intel 80386-Prozessor) mit einigen Erweiterungen aus. Hierfür stehen acht 32 Bit-Register zur Verfügung, die teilweise als Allzweck-Register genutzt werden können, bei einigen Instruktionen allerdings auch spezielle Bedeutung erhalten. Des weiteren existieren acht stapel-ähnlich organisierte 80 Bit breite Fließkommaregister.

Erweiterungen waren insbesondere die schrittweise eingeführten SIMD-Instruktionen: Die im Pentium MMX eingeführten Befehle mit Verwendungsschwerpunkt Multimedia ermöglichen Berechnungen mit Vektoren aus Integerzahlen auf acht 64 Bit breiten Vektorregistern, die intern allerdings gemeinsam mit den x87-FPU-Registern realisiert und deshalb nicht unabhängig sind (*Multi Media Extension*). Im Pentium III wurde dann die SSE (*Streaming SIMD Extension*) eingeführt, die zunächst nur die Verarbeitung von Vektoren aus maximal vier Fließkommawerten einfacher Genauigkeit auf acht neuen 128 Bit breiten Vektorregistern, auch mit Operanden aus dem Speicher, ermöglichte und neue MMX-Befehle hinzufügte (zum Teil als *MMX2* bezeichnet). Die mit den ersten Pentium 4 eingeführte SSE2 erweiterte den Vektor-Befehlssatz auf die 128-bittigen Vektorregister deutlich und ermöglichte dann die Verarbeitung von einzelnen Werten oder kompletten 16 Byte-Vektoren aller üblichen Datentypen, also 1, 2, 4 und 8 Byte großen Ganzzahlen und Fließkommazahlen einfacher und doppelter Genauigkeit. Die SSE3 des Prescott-Kerns brachte einige spezielle SIMD-Befehle, insbesondere horizontal auf den Vektoren arbeitende Operationen mit Einsatzschwerpunkt Video-Bearbeitung.

4.1.2 Prozessorkern

Die CISC-Befehle des IA-32-Befehlssatzes eignen sich wegen ihrer verschiedenen Länge und ihrer stark unterschiedlichen Komplexität nicht direkt zu Pipelining. Befehle werden deshalb direkt nach dem Laden von einem Decoder in eine Folge mehrerer einfacher, RISC-ähnlicher und direkt ausführbarer Instruktionen übersetzt (μops) und unter Beibehaltung der Reihenfolge in einem speziellen Instruktionscache (dem sogenannten *Execution Trace Cache*) gespeichert.

Eine Einheit aus fünf Schedulern wählt hieraus ausführbereite und nicht durch Abhängigkeiten blockierte μops aus und übergibt diese an die superskalar ausgelegten Recheneinheiten. Die Retirement-Einheit ordnet die Ergebnisse der μops (*Commitment*) und sorgt dafür, dass die IA-32-Instruktionen in ihrer ursprünglichen Reihenfolge abgeschlossen (*Retirement*) und mögliche Ausnahmen angezeigt werden. Die *Rapid Execution Engine* (Rechenkern abzüglich der Dekodiereinheit) ist als 31-stufige Pipeline ausgelegt (ältere Pentium 4: 20 Stufen).

Der Decoder kann pro Takt eine IA-32-Instruktion, der maximal 3 μops entsprechen, übersetzen und im Execution Trace Cache ablegen. Komplexere IA-32-Instruktionen müssen allerdings mit Hilfe eines speziellen ROMs in *Microcode*-Folgen übersetzt werden, deren Dekodierung und Ausführung ineffizienter ist.

Die Schedulereinheit aus fünf Einzelschedulern kann maximal 6μops pro Takt in Auftrag geben, was allerdings die Leistungsfähigkeit des Trace Cache übersteigt. Zur Verfügung stehen folgende Einheiten:

- eine doppelt getaktete Integer ALU für Addition/Subtraktion
- eine weitere doppelt getaktete Integer ALU, die zusätzlich für Logik, Sprünge und Speichern genutzt werden kann
- eine ALU für Integer-Multiplikationen
- eine Fließkommaeinheit zum Bewegen, Speichern und Tauschen von Fließkommawerten
- eine Cluster-artig ausgelegte FPU mit sieben Untereinheiten, die unter anderem auch für SIMD inklusive Integer zuständig sind
- eine Adress-Einheit (*Address Generation Unit*) zum Laden und Prefetchen
- ein weiteres AGU zum Speichern.

Eine der beiden doppelt getakteten Fließkommaeinheiten ist auch zu Schiebe- und Rotieroperationen fähig.

Zu Taktbeginn können die Scheduler durch insgesamt vier Ports vier Operationen starten und zur Takthälfte nochmals durch zwei Ports die doppelt getakteten ALUs beschicken. Allerdings können einige Untereinheiten nach Beginn einer Ausführung erst mit wenigen Takten Verzögerung eine neue Instruktion annehmen.

Pro Takt können eine Lese- und eine Schreiboperation auf den Speicher gestartet werden, auch Out of Order und, soweit keine TLB-Misses auftreten, bei spekulativer Ausführung. Acht ausstehende Leseoperationen können verwaltet werden. Für Schreiboperationen werden insgesamt 48 *Store Buffer* genutzt, wodurch Befehle bereits vor Ausschreiben in die Speicherhierarchie intern abgeschlossen werden können. Auf 36 ausstehende Schreiboperationen kann gleichzeitig gewartet werden. Werden Operanden aus dem Hauptspeicher benötigt, die wegen einer vorhergehenden Schreiboperation noch in einem Store Buffer verweilen, können diese ohne Umweg über die Caches als Operanden verwendet werden (*Store Forwarding*).

Der Scheduler geht grundsätzlich davon aus, dass benötigte Daten im Level 1 Cache vorhanden sind. Ist dies nicht der Fall, arbeiten die μops zunächst mit falschen Operanden und müssen später mit den korrekten Operanden wiederholt werden (*Replay*).

4.1.3 Speicherhierarchie

Der Pentium 4 der fauia42 realisiert eine zweistufige Speicherhierarchie mit getrennten Level 1-Caches und einem gemeinsame Level 2-Cache. Der Level 1-Cache besteht aus dem in Absatz 4.1.2 erwähnten Execution Trace Cache für Instruktionen und einem Level 1 Data Cache. Die Caches sind nicht inklusiv. Schreiboperationen, deren Ziel noch nicht im Level 1-Cache vorhanden ist, werden ausschließlich in den Level 2-Cache geschrieben, im anderen Fall in beiden Cache-Levels aktualisiert (Write-Through). Die Daten-Caches realisieren eine Pseudo-Least-Recently-Used Verdrängungstrategie. Eine Aufstellung weiterer Eigenschaften ist in Tabelle 4.1 zu finden.

Tabelle 4.1: Überblick über die Caches der fauia42

	L1 Trace Cache	**L1 Data Cache**	**L2 Cache**
Größe	12 kμops	16 kB	1 MB
Cachezeile	keine Angabe	64 Byte	64 Byte (effektiv 128 Byte)
Assoziativität	achtfach	achtfach	achtfach
Anbindung	keine Angabe	64 Bit (128 Bit zum Laden der Vektorregister)	256 Bit
Operationen pro Takt	je 3μop Lesen und Schreiben	ein Load und ein Store	ein Load oder ein Store
Latenz	keine Angabe	nach Größe 4 bis 12 Zyklen	7 Zyklen

Zudem verfügt der Pentium 4 Prescott über acht *Write Combining Buffers*. Diese werden auch genutzt, wenn Store Misses im Level 1-Cache auftreten, um den Transfer zum Level 2-Cache zu erhöhen.

Der Pentium 4 besitzt außerdem getrennte Translation Lookaside Buffer für Instruktionen und Daten. Diese besitzen jeweils 64 Einträge für 4 kB große Seiten und sind voll assoziativ. Größere Seiten müssen fragmentiert gespeichert werden.

4.1.4 Prefetcher

Der Prescott-Kern verfügt über Hardware-Prefetcher für Instruktionen und Daten. Der Instruktionsprefetcher lädt die nächsten 64 Byte des IA-32-Bytecode in 32 Byte-Blöcken, auch in Kooperation mit der Branch Prediction. Der Daten-Prefetcher kann bis zu acht kontinuierliche Lese- oder Schreibströme erkennen, allerdings maximal einen pro 4 kB-Speicherblock. Er lädt 128 Byteweise, allerdings maximal bis zum Ende eines 4 kB-Blocks. Er versucht dabei, 256 Byte vor dem tatsächlichen Datenstrom zu bleiben. Beide Prefetcher laden Cache-Zeilen in den Level 2-Cache.

Der erweiterte Instruktionssatz enthält außerdem spezielle Prefetch-Instruktionen, die dem Prefetcher allerdings nur als Hinweis dienen, und die er – z. B. bei fehlender freier Bandbreite – ignorieren kann.

Ein implizites Prefetching findet außerdem immer statt, wenn Daten vom Hauptspeicher in den Level 2-Cache gelesen werden, da hier stets eine zweite benachbarte Zeile mit eingelagert wird.

4.2 AMD Athlon64 4000+ 2,4 GHz fauia49.informatik.uni-erlangen.de

AMDs Athlon Prozessoren können zum einen als Konkurrenzprodukt zu Intels Prozessorserie gesehen werden, die denselben Bytecode aktzeptieren. Intern haben sie allerdings ein vollständig eigenes Design und bietet zusätzliche Fähigkeiten und einen noch größeren Befehlssatz.

4.2.1 Befehls- und Registersatz

Der AMD Athlon64 4000+ der fauia49 akzeptiert zunächst den IA-32 Bytecode des Intel Pentium 4 (mit Ausnahme der neuen SSE3-Instruktionen) und bietet zunächst denselben Registersatz. Die von Intel mit MMX, SSE und SSE2 eingeführten Instruktionen sind dort Teil der 3Dnow! und 3Dnow! Professional Erweiterungen. Allerdings wurden dabei auch weitere Instruktionen hinzugefügt. Der Athlon 4000+ der fauia49 enhält den „Hammer“ Prozessorkern. Mittlerweile sind auch neuere Kerne, unter derselben Prozessorbezeichnung, verfügbar, die auch SSE3-Instruktionen ausführen können.

Der Athlon64 ist grundsätzlich als 64 Bit-Prozessor konzipiert, der bei entsprechender Betriebssystemunterstützung einen größeren Adressraum und einen erweiterten Registersatz bietet. Im 64 Bit-Modus können die acht Allzweckregister auch 64 Bit breit genutzt werden, außerdem stehen acht weitere 64-bittige Allzweckregister und acht zusätzliche 128 Bit-Vektorregister für SIMD-Operationen zur Verfügung.

4.2.2 Prozessorkern

Um die komplexen Operationen des Befehlssatzes effizient verarbeiten zu können, werden diese auch vom Athlon64 in kleinere, RISC-ähnliche Instruktionen zerlegt. AMD hat sich hier allerdings für ein mehrstufiges Konzept entschieden.

Die Instruktionen werden nach dem Laden mit Zusatzinformationen, die effizientere Dekodierung und Ausführung ermöglichen, im Level 1-Instruktions-Cache abgelegt. Von einer Hol- und Dekodiereinheit wird der daraus gelesene Instruktionsstrom in ursprünglicher Reihenfolge in *macro-ops* umgewandelt und einer zentralen Kontrolleinheit (*Instruction Control Unit*) übergeben, wo sie in einem Puffer zwischengespeichert werden. Aus diesem Puffer kann das ICU macro-ops an die Integer- oder Fließkomma-Scheduler unter Beachtung von Abhängigkeiten und vorhandenen Resourcen zur Ausführung übergeben. Das ICU ist auch für das Retirement der macro-ops, Ausnahmen, Unterbrechungen und die Behandlung falscher Sprungvorhersagen zuständig, sowie das Register Renaming der Integer-Einheiten.

Die Dekodiereinheit besteht aus drei Einzeldekodern und kann pro Takt maximal 3 macro-ops erzeugen. Einfache IA-32 bzw. AMD64-Instruktionen können direkt in ein oder zwei macro-ops übersetzt werden (*direct path*, *double*), komplexe Instruktionen müssen in einem ROM nachgeschlagen werden (*vector path*) und können das gleichzeitige Dekodieren von Direct Path-Instruktionen verhindern.

Die drei Integer-Scheduler, denen jeweils eine ALU und ein Adressiereinheit angeschlossen sind, zerteilen die macro-ops in kleinere *micro-ops*, die wiederum bei verfügbaren Operanden Out of Order direkt von der Hardware ausgeführt werden können. Alle drei ALU/AGU-Einheiten beherrschen alle logischen und arithmetischen Operationen und Adressierfunktionen mit Ausnahme der Multiplikation; hierfür steht eine spezielle Einheit zur Verfügung, zu deren Nutzung zwei ALUs synchron genutzt werden müssen.

Die Fließkommaeinheit ist als eigenständiger mathematischer Koprozessor ausgelegt, der ein eigenes Register File verwaltet. Aufgabe der FPU ist die Ausführung aller x87, MMX, SSE und SSE2-Instruktionen und der AMD-eigenen 3Dnow! Erweiterungen. Dies schließt auch einige Integer-Operationen mit ein. Der Fließkomma-Scheduler übersetzt die macro-ops in micro-ops und kann maximal drei micro-ops pro Takt an folgende Einheiten übergeben:

- eine Einheit für einfache logische und arithmethische Operationen
- eine Einheit für komplexere Operationen wie Multiplikation, aber auch Division und Wurzel
- eine Einheit, die zum Schreiben und für einige komplexe Vector Path-Instruktionen genutzt wird

Die Fließkommaeinheit unterstützt außerdem ein spezielles Chaining: Werden Operanden durch eine reine Ladeoperation vom Speicher in ein Register geholt und anschließend durch die FPU

verarbeitet, können diese Daten bereits vor Abschluss der Lade-Instruktion genutzt werden (*Super Forwarding*).

Speicherzugriffe werden zentral durch das *Load-Store Unit* verwaltet. Dieses kann zwölf Zugriffe auf den Level 1-Cache verwalten. Bei Cache Misses werden diese Operationen in einer weiteren Warteschlange mit 32 Einträgen über den Level 2-Cache abgehandelt oder initiieren Hauptspeicherzugriffe. Das LSU ist auch für die Einhaltung der semantischen Ausführungsreihenfolge verantwortlich. Die LSU kann maximal zwei Operationen über jeweils maximal 64 Bit pro Taktzyklus durchführen und beschränkt damit den Datendurchsatz des Prozessors auf maximal 16 Byte pro Takt.

4.2.3 Speicherhierarchie

Die Cachehierarchie ist zweistufig mit getrennten Instruktions- und Datencaches auf Level 1 und einem gemeinsamen Level 2-Cache. Im Level 1-Instruktions-Cache wird ein kommentierter Bytecode, der neben Hinweisen für eine schnellere Dekodierung auch Daten über das bisherige Verhalten bei bedingten Sprüngen für die Branch Prediction enthält, abgelegt. Somit bleiben diese Informationen auch bei Verdrängung in den Level 2-Cache erhalten.

Instruktionen und Daten werden primär nur in den Level 1-Cache geladen. Der Level 2-Cache ist exklusiv und enthält nur Daten, die entweder aus dem Level 1-Cache verdrängt oder verändert wurden und später in den Hauptspeicher geschrieben werden müssen (*Victim or Copy-Back*).

Im Gegensatz zu den meisten anderen Prozessoren benötigt der Athlon64 keine Northbridge auf der Hauptplatine. Stattdessen besitzt er einen integrierten Memory Controller, der dadurch die Latenz zum DDR-SDRAM verkürzt. Zur Kommunikation mit Peripherie-Geräten wird die HyperTransport-Technologie eingesetzt. Weitere Informationen über die Caches sind Tabelle 4.2 zu entnehmen.

Tabelle 4.2: Überblick über die Caches der fauia49

	L1 Instr. Cache	**L1 Data Cache**	**L2 Cache**
Größe	64 kB	64 kB	1 MB
Cachezeile	64 Byte	64 Byte	64 Byte
Assoziativität	zweifach	zweifach	16-fach

Der Athlon64 organisiert auch die Translation Lookaside Buffer hierarchisch. Die getrennten Level 1 Instruction und Data TLBs sind voll assoziativ und können 16 4 kB- sowie acht 2 MB- oder vier 4 MB-Seiten verwalten. Der exklusive Level 2 TLB umfasst 256 4 kB-Seiten.

Athlon64 Prozessoren besitzen mindestens vier Write Combining Buffer. Diese werden allerdings nur für spezielle Speicherbefehle genutzt (siehe 5.3).

4.2.4 Prefetcher

Der AMD Athlon64 bietet ein implizites Instruktions-Prefetching, indem bei einem Cache Miss stets die folgende Cache-Zeile mit eingelagert wird.

Der Hardware-Prefetcher orientiert sich an Zugriffen auf aneinander liegende Cachelines: Werden Cachzeile n und Cachezeile $n+1$ nacheinander adressiert, lädt er Cachezeile $n+3$ in den Speicher. Über die Anzahl erkennbarer Datenströme, oder ob dieser Prefetcher auch den Instruktions-Strom berücksichtigt, ist der Dokumentation nichts zu entnehmen. Maximal auf acht Prefetches kann gewartet werden, die allerdings nur bei freien Resourcen durchgeführt werden.

Der Athlon64 unterstützt zum einen die Prefetch-Instruktionen des erweiterten IA-32-Befehlssatzes und lädt damit Daten in den Level 1 Data Cache. AMDs 3Dnow! bietet außerdem zwei weitere Prefetch-Befehle, die ebenfalls Daten in den Level 1 Cache laden. Hierunter ist auch ein spezielle Anweisung zum Prefetchen von Daten, die geändert werden sollen.

Kapitel 5

Angewandte Optimierungsverfahren

5.1 Datenlayout

Ein großes Potential liegt in den Vektoreinheiten der Testsysteme. Ein Datenlayout sollte deshalb deren Verwendung nicht nur als schnelle skalare FPU, sondern auch eine effiziente Verarbeitung von ganzen Vektoren von Werten ermöglichen. In den folgenden Kapiteln sollen deshalb Voraussetzungen für solch ein Layout sowie deren genaue Realisierung in zwei und drei Dimensionen erklärt werden.

Grundsätzlich profitieren Prozessoren von einer angepassten Ausrichtung von Daten im Speicher. Da moderne Caches nicht Byte-genau adressiert werden können, müssen ungünstig liegende Operanden möglicherweise in zwei Teilen getrennt übertragen werden. Werden Daten *natürlich* im Speicher ausgerichtet, das entspricht bei den üblichen 2^n Byte großen Datentypen einer Ausrichtung an einem Vielfachen ihrer Größe, kann dies ausgeschlossen und eine schnelle Verarbeitung sichergestellt werden.

Da der Zugriff auf nicht ausgerichtete Daten auch eine entsprechend aufwendige Unterstützung durch die Hardware voraussetzt, bieten die SIMD-Einheiten beider Prozessoren nur spezielle, langsamere Lade- und Schreiboperationen, mit denen Vektoren vollständig oder in Teilen übertragen werden können. Als Operanden für Vektorberechnungen können nur natürlich ausgerichtete Vektoren oder Registerinhalte genutzt werden.

Das vorgeschlagene Datenlayout geht bei der Unterstützung der vektorisierten Berechnung von 128 Bit großen Vektoren aus zwei Fließkommazahlen doppelter Genauigkeit aus. Der Ansatz lässt sich aber auch für andere Vektorgrößen und Datentypen verallgemeinern.

5.1.1 Anpassung des Datenloyouts für 2D

Ziel ist es, statt der aufeinander folgenden getrennten Berechnungen

$$U_{x,y} = \frac{1}{4}\big(U_{x-1,y} + U_{x+1,y} + U_{x,y-1} + U_{x,y+1} - (dh^2 \cdot F_{x,y})\big)$$

und

$$U_{x+2,y} = \frac{1}{4}\big(U_{x+1,y} + U_{x+3,y} + U_{x+2,y-1} + U_{x+2,y+1} - (dh^2 \cdot F_{x+2,y})\big)$$

die Vektorrechnung

$$\begin{bmatrix} U_{x,y} \\ U_{x+2,y} \end{bmatrix} = \frac{1}{4}\left(\begin{bmatrix} U_{x-1,y} \\ U_{x+1,y} \end{bmatrix} + \begin{bmatrix} U_{x+1,y} \\ U_{x+3,y} \end{bmatrix} + \begin{bmatrix} U_{x,y-1} \\ U_{x+2,y-1} \end{bmatrix} + \begin{bmatrix} U_{x,y+1} \\ U_{x+2,y+1} \end{bmatrix} - dh^2 \cdot \begin{bmatrix} F_{x,y} \\ F_{x+2,y} \end{bmatrix} \right)$$

auszuführen. Für den dreidimensionalen Fall gilt dies analog. Dieser wird in Abschnitt 5.1.2 genauer behandelt.

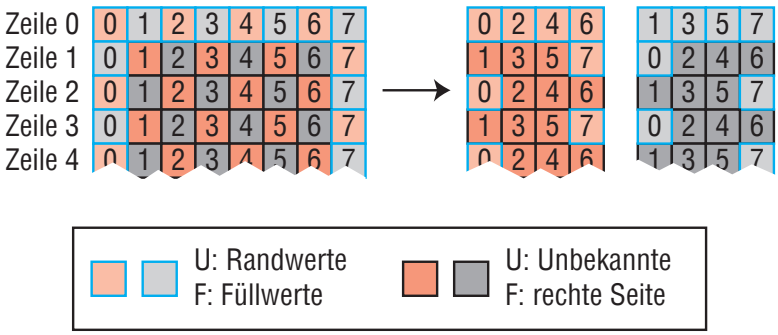
Die Anforderungen an das Speicherlayout sind folgende:

1. die Paare eines Vektors sollen direkt nebeneinander im Speicher liegen
2. möglichst viele für eine Berechnung verwendete Vektoren sollen natürlich, also an 16 Byte-Grenzen, ausgerichtet sein

Anforderung 1 lässt sich durch zeilenweises Auftrennen der beiden Felder U mit den Unbekannten und Randwerten und F mit der rechten Seite der Gleichungen in jeweils zwei neue Arrays bewerkstelligen, die entweder nur rote oder schwarze Werte enthalten. Die Bezeichnungen „rot“ und „schwarz“ werden hier auch auf die Werte aus F und die Randwerte in U verallgemeinert (vgl. Abbildung 5.1).

Bei einem $x \times y$ großen Problem werden mit Randwerten sonst $(x + 2) \times (y + 2)$ große Felder benötigt. Durch die Auftrennung entstehen zwei neue Felder mit $(y+2)$ *Halbzeilen*. Ist x gerade, so enthält jede Zeile gleich viele rote und schwarze Werte, und jede Halbzeile enthält $x/2$ Unbekannte und einen Randwert zu Beginn oder am Ende. Ist x allerdings ungerade, so enthalten die Zeilen unterschiedlich viele rote und schwarze Werte, und dementsprechend sind die *geraden* Halbzeilen mit $y = 0, 2, \ldots$ und die *ungeraden* Halbzeilen unterschiedlich lang. Mit Ausnahme der Halbzeilen 0 und $y+1$, die ausschließlich Randwerte enthalten, enthält eine Halbzeile dann entweder nur $(x+1)/2$ Unbekannte oder $(x - 1)/2$ Unbekannte und zwei Randwerte. In der Praxis sollte hier allerdings eine gleiche Zeilenbreite gewählt werden und in den kürzeren Zeilen Speicher ungenutzt bleiben, um das Feld auch in Hochsprache als Array ansprechen zu können.

Abbildung 5.1: Trennung der roten und schwarzen Werte in 2D



Speicherausrichtung lässt sich für alle Paare mit Ausnahme der Vektoren $(U_{x-1,y}; U_{x+1,y})$ und $(U_{x+1,y}; U_{x+3,y})$ erreichen, die offensichtlich nicht beide gleichzeitig im Speicher entsprechend ausgerichtet sein können. Um Berechnungen möglichst über die ganze Zeile vollständig mit Vektoroperationen durchführen zu können, werden die ersten Unbekannten jeder Zeile an 16 Byte-Grenzen ausgerichtet.

Für den beschriebenen Fall mit Vektoren aus zwei Fließkommazahlen kann man dies sehr leicht erreichen, indem man zunächst mit Padding die Länge einer Halbzeile auf eine ungerade Anzahl von Werten bringt, wenn dies nötig ist. Dann wird der Beginn der schwarzen Felder an 16 Byte-Grenzen und der roten Felder an 16 Byte-Grenzen mit einem Offset von 8 Byte ausgerichtet.

Ist die Anzahl an Unbekannten in einer Zeile x ganzzahlig durch vier teilbar, so können alle Berechnungen vektorisiert durchgeführt werden. Andernfalls kann am Ende einer Zeile noch eine Berechnung mit nur einem Teilvektor notwendig sein, was in diesem Fall einer skalaren Berechnung entspricht. Abbildung 5.2 zeigt hierfür ein Beispiel.

Die Punkte können getrennt dann nicht mehr mit den bisherigen Indizes adressiert werden. Abbildung 5.3 zeigt die Zuordnung zwischen den gemeinsamen und den getrennten Feldern. In der Praxis ist eine transparente Zuordnung zwischen der getrennten Speicherung und dem zu Grunde liegenden Gitter – z.B. über Unterfunktionen – innerhalb des Lösers nicht sinnvoll.

Abbildung 5.2: Layout der roten und schwarzen 2D-Arrays bei $x = 9$

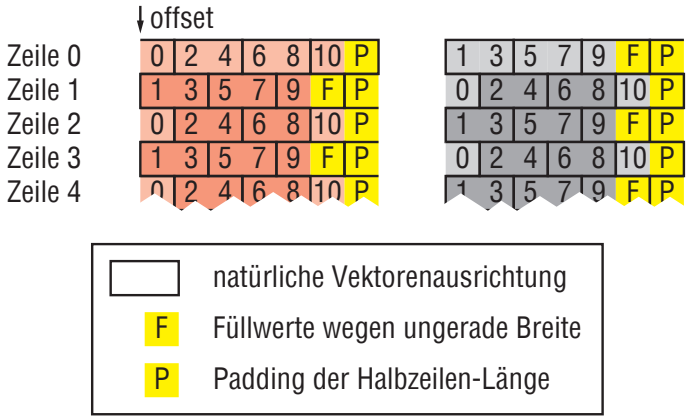
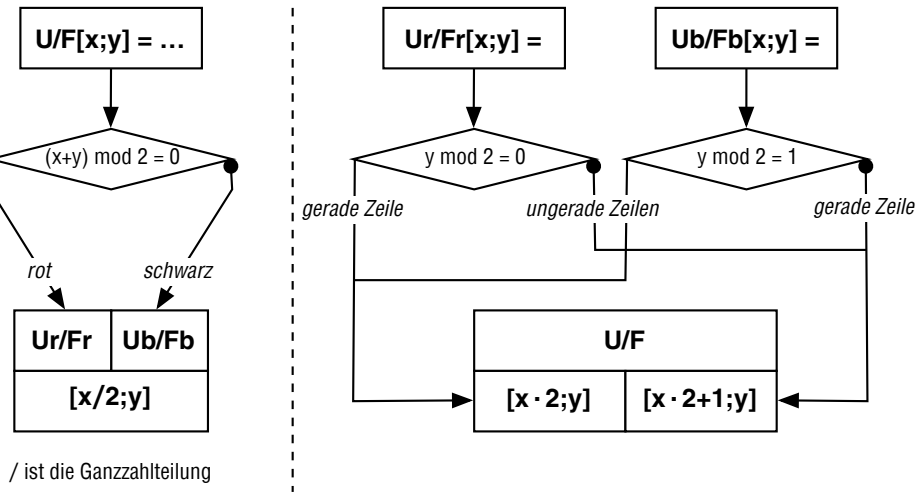


Abbildung 5.3: Zuordnung zwischen getrennten und gemeinsamen Feldern



Für die Implementierung ist die Sicht auf gemeinsame Felder auch nicht nötig. Eine Iteration bedeutet hier nur, zunächst alle Unbekannten in U_r und danach alle Unbekannten in U_b zu berechnen. Dazu müssen allerdings zwei leicht unterschiedliche Stempel genutzt werden (siehe Abbildung 5.4), die jeweils auch noch in zwei Varianten entweder Werte in U_r aus U_b oder umgekehrt berechnen. Von ihrer „Form“ und ihren ausgerichteten und unausgerichteten Zugriffen gleichen sich die Berechnungen von „roten ungerade“ und „schwarzen geraden“ Zeilen einerseits, und den anderen Zeilen andererseits. Bei getrennten Arrays lautet die Berechnung für Halbzeilen, die mit einem Randwert beginnen

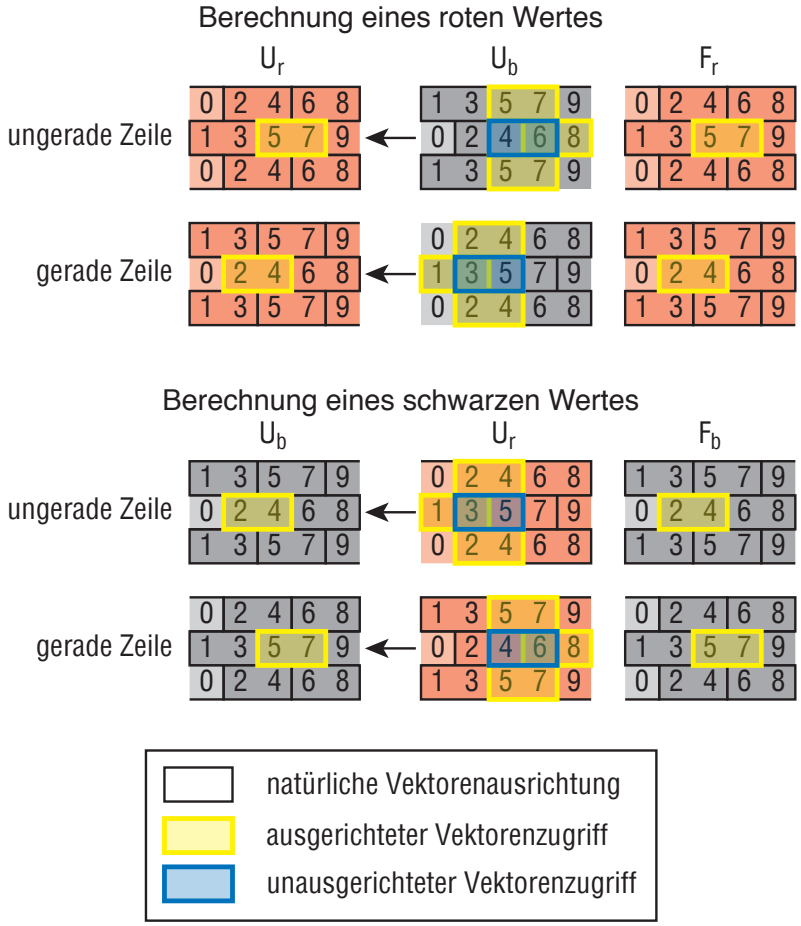
$$U_{r/b}[x;y] = \frac{1}{4}\left(U_{b/r}[x,y-1] + U_{b/r}[x,y+1] + U_{b/r}[x;y] + U_{b/r}[x+1;y] - dh^2 \cdot F_{r/b}[x;y]\right)$$

und für die anderen

$$U_{r/b}[x;y] = \frac{1}{4}\left(U_{b/r}[x,y-1] + U_{b/r}[x,y+1] + U_{b/r}[x-1;y] + U_{b/r}[x;y] - dh^2 \cdot F_{r/b}[x;y]\right)$$

Bei vektorsierten Berechnungen ist jeweils der Vektor $(U_{b/r}[x;y]/U_{b/r}[x+2;y])$ nicht natürlich im Speicher ausgerichtet.

Abbildung 5.4: Diskreter Fünf-Vektor-Stempel in 2D



Für beliebig große Vektoren aus beliebigen Datentypen ist dieser Ansatz jedoch nicht ohne weiteres übertragbar. Eine Möglichkeit wäre es hier, die effektive Zeilenlänge auf ein Vielfaches der Vektorgröße zu padden und jede Zeile mit einer Unbekannten beginnen zu lassen. Randwerte an der linken Seite werden dann an das Ende der vorhergehenden Halbzeile geschoben. In einigen Programmiersprachen ist der Arrayzugriff mit negativen Indizes durchaus möglich. In *C* z. B., wo man die Indizes wegen der *column first*-Schreibweise bei Arrays umdrehen sollte, ist ein Zugriff auf $Ur[3][-1]$ möglich.

Die Auftrennung der Werte in Halbfelder führt auch zu einer effizienteren Cache-Nutzung: Bei einer Berechnung mit gemischten Feldern liegen schwarze und rote Werte innerhalb einer Zeile alternierend im Speicher. So werden bei jeder Halbiteration auch unbenötigte Werte aus F in den Speicher geladen. Aber auch für die Werte aus U bringt dies Vorteile, wenn nicht alle Werte der vorher berechneten Zeile im höchsten Cachelevel verweilen können. Für die Berechnung der Zeile y werden bei gemischten Feldern die vollständigen Zeilen $y-1$ und $y+1$ in den Level 1 Cache geladen, obwohl nur eine Farbe davon benötigt würde.

5.1.2 Erweiterung auf 3D

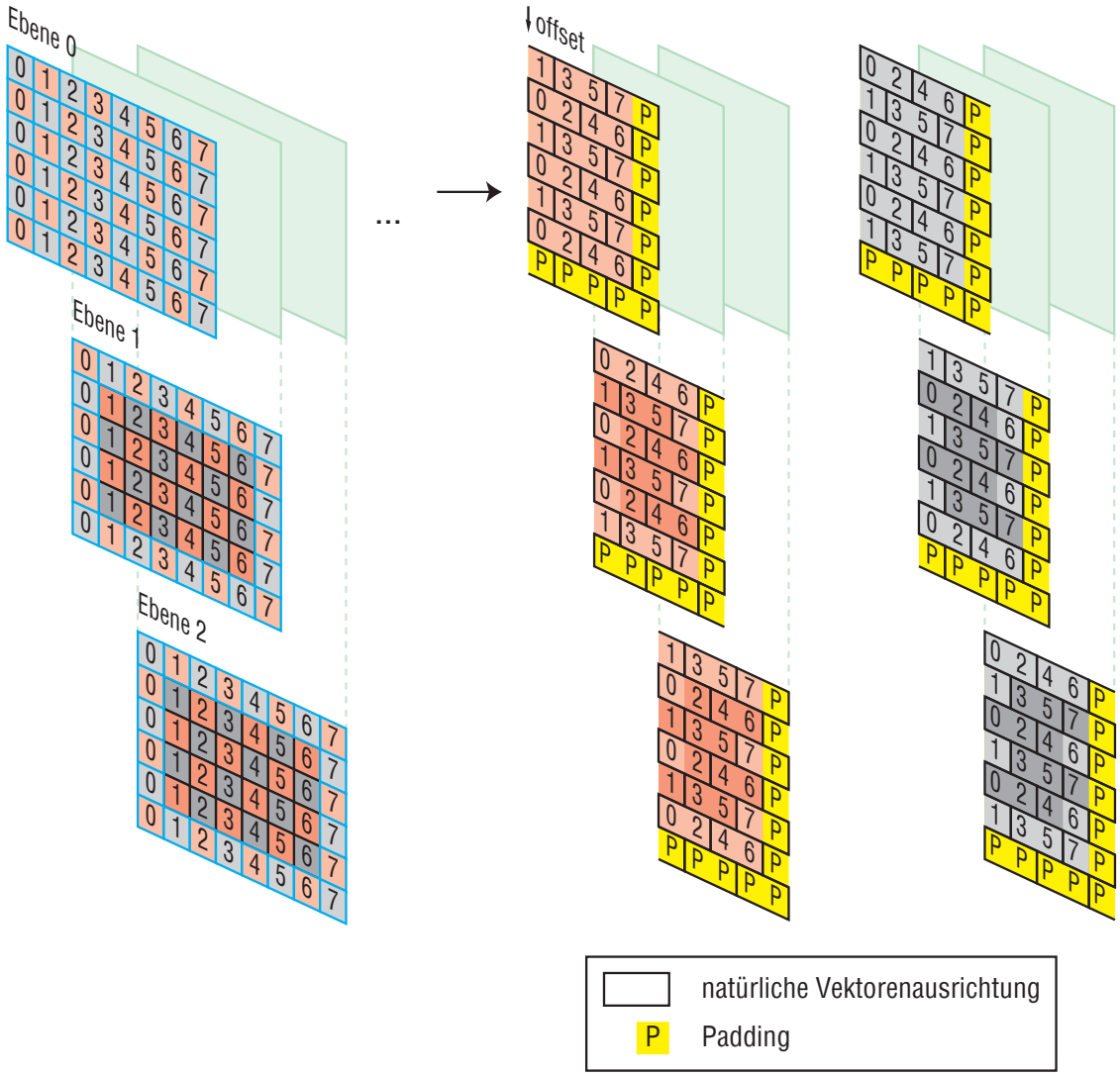
Dieses Speicherlayout mit getrennten Feldern kann auch auf den dreidimensionalen Fall erweitert werden. Hier lässt sich das dreidimensionale Gitter als Aneinanderreihung mehrerer zweidimensionaler Ebenen betrachten, innerhalb derer sich das Speicherlayout wie in Abschnitt 5.1.1 beschrieben anpassen lässt. *Ungerade* Ebenen mit $z = 1, 3, \ldots$ entsprechen hier dem zweidimensionalen Fall, in den *geraden* Ebenen mit $z = 0, 2, 4, \ldots$ ist die Verteilung von roten und schwarzen Punkten genau invertiert. Um die Bezeichnung gerader und ungerader Zeilen konsistent zu halten, werden im folgenden die Zeilen mit $y + z \; mod \; 2 = 0$ als ungerade Zeilen bezeichnet, da der Randwert am Anfang dieser Zeilen wie in 2D den schwarzen Halbfelder zugeordnet wird.

Da die erste Ebene mit einem schwarzen (aber nie benötigten) Randwert $U[0;0;0]$ beginnt, müssen in diesem Fall die Ausrichtungen der Arrays gegenüber der 2D-Variante vertauscht werden, also U_r und F_r werden direkt an 16-Byte-Grenzen ausgerichtet, und U_b und F_b mit einem Offset von acht Byte.

Des weiteren muss sichergestellt werden, dass auch in den folgenden Ebenen eine entsprechende Speicherausrichtung besteht. Enthält eine Ebene eine ungerade Anzahl von Halbzeilen, so belegt eine Halbebene wegen der Halbzeilen ungerader Länge auch ein ungerades Vielfaches von 8 Byte, und die folgenden Halbebenen sind jeweils entsprechend ausgerichtet. Im anderen Fall ist ein Padding mit acht Byte am Ende jeder Halbebene theoretisch ausreichend, aber durch das Anhängen einer leeren Halbzeile an jede Ebene bleibt sinnvoller Zugriff auf Hochsprachenebene möglich.

Bei der Berechnung im Dreidimensionalen entsprechen die Zugriffe innerhalb der Ebene auf $U[x \pm 1; y; z]$ und $U[x; y \pm 1; z]$ dem zweidimensionalen Fall. Die Halbzeilen, die die vorderen und hinteren Nachbarn $U[x; y; z \pm 1]$ enthalten, sind ebenso wie die Zeilen mit $U[x; y \pm 1; z]$ ausgerichtet. Damit ist auch auf diese Werte natürlich ausgerichteter Zugriff möglich.

Abbildung 5.5: Speicherlayout in 3D



5.2 Code-Optimierung

Eine große Rolle für die Ausführung des Codes spielt auch die Auswahl und Reihenfolge der Maschinensprache-Operationen. Bei Hochsprache wird diese von einem Compiler erzeugt und kann vom Programmierer oft nur mittelbar oder indirekt beeinflusst werden. Zu den Möglichkeiten des Programmiers zählen hier eine geschickte Auswahl von Hochsprache-Konstrukten und in einigen Fällen spezielle Hinweise an den Compiler, für C sind dies meist `#pragma`-Anweisungen oder Compiler-Flags.

Für große Software-Projekte ist die Programmierung in Assembler aus guten Gründen unüblich: Der Hochsprache-Code ist deutlich einfacher zu lesen und zu warten, während moderne Compiler sehr schnell mit vielen Optimierungen automatisch effiziente Befehlsfolgen erzeugen. Die Generierung von ähnlich „gutem“ Assembler-Code von Hand benötigt meist mehr Zeit und die Fehlerfreiheit ist ungleich schwerer zu sichern. Deshalb lohnt sich Assembler-Programmierung nur für relativ kleine und rechenintensive Programmteile, wie z. B. den Red-Black Gauss-Seidel.

5.2.1 Ablaufkontrolle und Adressierung

Zur Ausführungskontrolle werden alle acht Allzweckregister vorgesehen. Da keinerlei Unterfunktionsaufrufe stattfinden, werden vorübergehend alle Registerwerte auf dem Stapel abgelegt, und auch der Stack Pointer im Speicher ausgelagert. Dadurch ist es möglich, alle Schleifenvariablen und Zeiger im Prozessor zu halten. Die für Schleifen und Adressberechnung benötigten Operanden werden als direkte Operanden der Befehle im Instruktionsstrom eingelagert, so dass der Level 1 Data Cache ausschließlich für die eigentliche Berechnung zur Verfügung steht.

Die augenblickliche Position in den Feldern wird über einen einzigen Zeiger verwaltet, die Operanden im Speicher werden dann durch einen konstanten Wert und dem Zeiger als Offset adressiert. Dynamischer Speicher kann nur eingeschränkt genutzt werden, da hierfür insgesamt vier Zeiger nötig wären, davon drei jeweils für Berechnungen in einer Farbe. Allerdings gibt es die Möglichkeit, einen einzigen großen Speicherblock zu allozieren, der intern in die vier Halbfelder aufgeteilt wird. Seine Anfangsadresse wird in einem reservierten Register abgelegt, und Speicherzugriffe müssen dann mit einem konstanten Wert und zwei Registern als Offset durchgeführt werden. Da die Feldgrößen zur Übersetzungszeit bereits festgelegt sind, können fast alle Änderungen des Zeigers mit einfachen Additionen durchgeführt werden.

5.2.2 Ausführung der Berechnungen

Alle Berechnungen werden auf der Vektoreinheit durchgeführt. Die konstanten Teile der Iterationsformel werden dauerhaft in zwei Vektorregistern abgelegt. Statt einer Division durch 4 bzw. 6 wird eine Multiplikation mit $\frac{1}{4}$ bzw $\frac{1}{6}$ verwendet, da diese Operation äquivalent ist, aber vom Prozessor schneller ausgeführt werden kann. Statt $\ldots - dh^2 \cdot F[i;j]$ wird die äquivalente Berechnung $\ldots + (-dh^2) \cdot F[i;j]$ durchgeführt, in 3D analog. Dadurch werden nur Addition und Multiplikation benötigt. Diese konstanten Teil werden jeweils in der unteren und oberen Hälfte von Registern abgelegt, für die eigentlichen Berechnungen stehen danach noch sechs Vektorregister zur Verfügung.

Für eine vektorisierte Berechnung sind zwei Register notwendig. Der unausgerichtete Vektor aus U muss zunächst speziell geladen werden. In ein weiteres Register wird der Vektor aus F geladen und multipliziert. Alle weiteren Werte aus U sind ausgerichtet, so dass Additionen mit Operanden aus dem Speicher genutzt werden können. Nach Addition der beiden Register und der Multiplikation mit $\frac{1}{4}$ bzw. $\frac{1}{6}$ können diese Werte dann wieder ausgerichtet als Vektor in den Speicher geschrieben werden.

Vektorisierung entspricht implizit einem Loop Unrolling um den Faktor 2. Bei der Generierung des Assembler-Codes können allerdings längere Befehlssequenz aufgerollt werden. Dabei wird die entsprechende Verschiebung im konstanten Teil der Adressierung berücksichtigt, und das Register mit dem Zeiger muss währenddessen nicht geändert werden.

Für den Athlon64 und den Pentium 4 wird jeweils leicht unterschiedlicher Code generiert, dessen Scheduling jeweils an den Prozessor angepasst ist. Längere Code-Sequenzen werden für den Pentium 4 aus Templates für die skalare Berechnung eines Wertes und aus Mustern für die vektorisierte Berechnung von zwei, vier, sechs und zehn Werten zusammengesetzt. Für den Athlon64 stehen nur Codemuster für die einzelne skalare und die vektorisierte Berechnung von zwei Werten zur Verfügung, da durch das Überschneiden der Instruktionen mehrerer Berechnungen keine höhere Geschwindigkeit erreicht werden konnte.

Zudem werden zum Laden des unausgerichteten Vektors unterschiedliche Instruktionen genutzt. Beim Pentium 4 wird das untere Register mit der *movsd*-Anweisung geladen, so dass die höherwertige Registerhälfte gelöscht wird. Er kann Teile eines Registers nur durch eine logische Operation ändern, sodass ein Überschreiben effizienter ist. Der Athlon64 muss Vektoroperationen stets über zwei 64-bittige Zugriffe ausführen. Deshalb ist das ändern nur einer Registerhälfte mit *movlpd* schneller. Die zweite Registerhälfte muss auf beiden Prozessoren über die *movhpd*-Instruktion dazugeladen werden.

5.3 Non-Temporal Moves

Caches basieren ursprünglich auf einem transparenten Konzept, das alle gelesen oder geschriebenen Daten gleich einlagert oder verdrängt. Dabei ist keine Unterscheidung möglich, ob Daten einmalig oder mehrfach benötigt werden. Dadurch können Daten, die nur einmalig gelesen oder geschrieben werden, häufig genutzte Daten aus den Caches verdrängen (*Cache Pollution*).

Um dem entegenwirken zu können, wurden spezielle Schreib-Operationen eingeführt, sogenannte *Non-Temporal Moves*. Diese schreiben die Daten über spezielle Puffer an den normalen zeitabhängigen Cachebereichen vorbei, in denen dadurch mehr Raum für sinnvolle Zwischenspeicherung bleibt. Bei den Testsystemen wird dies über die Write Combining Buffer (vgl. Abschnit 3.2.3) realisiert. Können die so gesammelten Daten einen kompletten RAM-Speicherblock füllen, kann dieser Bereich im Burst Mode überschrieben werden. Es ist in der Spezifikation nicht festgelegt, ab wann diese Daten tatsächlich in den Hauptspeicher ausgeschrieben werden, so dass dies auch erst in freien Bustakten durchgeführt werden kann.

Der erweiterte IA-32-Befehlssatz bietet für die SSE-Register entsprechende Schreibbefehle allerdings nur für vollständige 128 Bit-Vektoren. Bei getrennten Feldern dies für den Red-Black Gauss-Seidel mit getrennten Feldern mit Ausnahme nur skalar berechenbarer Werte am Rand gut nutzbar; für eine rote Halbiteration werden die neuen Werte mit Daten aus U_b und F_r berechnet und in U_r geschrieben und für die schwarze Halbiteration entsprechend entgegengesetzt.

Bei der Realisierung der Non-Temporal Writes in den Testystemen treten allerdings Probleme auf, wenn wider Erwarten doch Daten aus der zu den so geschriebenen Daten gehörenden Cache-Zeile vorzeitig adressiert werden. Für Ein-Prozessor-Systeme droht hier einzig Geschwindigkeitsverlust, für Mehr-Prozessor-Systeme realisieren diese Non-Temporal Moves allerdings ein nur schwach geordnetes Konsistenzmodell. Demnach wird nur sichergestellt, dass auch tatsächlich die letzte Schreiboperation den endgültigen Wert einer Adresse festlegt, allerdings nicht, wann diese Änderungen für andere Prozessoren sichtbar wird. In diesem Fall kann über spezielle *fence*-Operationen sichergestellt werden, dass die Änderungen allgemein wirksam werden. Da durch die schwache Konsistenz kein exklusiver Zugriff nötig ist, können für die Reservierung eingesparte Buszyklen anderweitig genutzt werden, was zu einer höheren Transferrate an Nutzdaten führt.

Non-Temporal Loads sind nicht explizit verfügbar. Dieses Verhalten kann bedingt durch spezielle Prefetch-Instruktionen erreicht werden, wie in Abschnitt 5.5 genauer beschrieben wird.

5.4 Blocking

Unter Blocking versteht man Techniken, die Berechnungen zur besseren Nutzung der Caches in anderer Reihenfolge vornehmen, ohne den zu Grunde liegenden Algorithmus und das Ergebnis

zu ändern. Ist die Verarbeitungsgeschwindigkeit vor allem durch die Speicherbandbreite begrenzt, können solche Verfahren die größte Geschwindigkeitssteigerung bringen.

Für den Red-Black Gauss-Seidel können alle Berechnungen einer Halbiteration in beliebiger Reihenfolge durchgeführt werden, theoretisch ergeben sich für die Berechnung von n Werten einer Farbe $n!$ Möglichkeiten. Das direkte Vorgehen, zeilenweise neue Werte zu berechnen, ist durchaus effizient, solange bei Berechnungen die Werte aus der vorhergehenden Zeile noch in der höchsten Cache-Ebene vorhanden sind. Bei zunehmender Zeilenlänge besteht allerdings die Möglichkeit, dass diese Werte zwischenzeitlich in eine tiefere Cache-Ebene oder den Hauptspeicher verdrängt wurden.

Bei Blocking-Verfahren wird das Gitter gedanklich zerlegt, und die entstehenden Bereiche nacheinander jeweils vollständig abgearbeitet. Im allgemeinen werden hierfür Rechtecke im Zweidimensionalen und Quader im Dreidimensionalen verwendet. Wird die Größe der Blöcke entsprechend der Cache-Hierarchie gewählt, erreicht man hierdurch eine höhere temporale Nähe. Im folgenden zwei einführende Beispiel möglicher Block-Verfahren:

Beispiel A Es werden $1 \times n$ große Blöcke genutzt, also immer Werte aus mehreren Zeilen berechnet. Die rote Halbiteration beginnt dann mit den Berechnungen im ersten Block, also $U_{1,1}$, $U_{1,3}$ ... $U_{1,y}$ mit $y \leq n$. Der Block wird in der Zeile um eins weitergeschoben und $U_{2,2}$, $U_{2,4}$ ... $U_{2,y}$ mit $y \leq n$ aktualisiert. Dies wird dann bis zum Ende der Zeile und in der nächsten Blockzeile mit dem Block $U_{1,n+1} - U_{1,2 \cdot n}$ beginnend fortgesetzt. Am unteren Rand des Gitters müssen möglicherweise noch niedrigere Blöcke genutzt werden, wenn die Höhe y nicht ganzzahlig durch n teilbar ist. Danach folgt ein analoger Durchgang mit der Berechnung der schwarzen Werte.

Beispiel B Es werden $n \times m$ große Blöcke genutzt. Zunächst wird ein Teil der roten Werte in der ersten Zeile innerhalb $U_{1,1} - U_{m,1}$ berechnet. Dies wird in den Zeilen darunter fortgesetzt, bis der gesamte Block $U_{1,1} - U_{m,n}$ berechnet wurde. Der Block wird danach in der Zeile versetzt und mit den roten Werten in $U_{m+1,1} - U_{2 \cdot m,n}$ fortgefahren usw. In diesem Fall können am Ende der Zeilen und am unteren Rand verschieden große Blöcke zum Füllen nötig sein. Die schwarze Halbiteration findet analog statt.

Die genau gewählte Blocking-Technik hängt sehr stark von der ausführenden Hardware ab: Bei niedriger Assoziativität der Caches können Blöcke über viele Zeilen zu Cache-Konflikten führen. Eine Rolle kann auch die Größe des Table Lookaside Buffers spielen, denn Blöcke über mehrere Zeilen greifen auf weiter voneinander im Speicher liegende Adressen zu. Ebenso spielt das Verhalten von Hardware-Prefetchern eine Rolle. Bei einer Berechnung wie in Beispiel A kann dieser häufig keine oder nur scheinbare Datenströme erkennen. Bei Beispiel B kann es zum umgekehrten Effekt kommen: Es werden über die Breite des Blocks hinaus Daten geholt, die bei den Berechnung innerhalb des Blocks wieder verdrängt werden und für den nächsten Block nicht mehr zur Verfügung stehen.

Bei Algorithmen, die iterative oder hintereinander mehrere Berechnungen auf großen Datenmengen ausführen, bietet sich außerdem an, mehrere Berechnungen bzw. Iterationen so bald wie möglich und sinnvoll auszuführen. Beim Red-Black Gauss-Seidel kann ein Wert berechnet werden, sobald all seine Nachbarn entsprechend häufig aktualisiert wurden.

Das oben genannte lokale Blocking kann dazu zum temporalen Blocking erweitert werden. Werden die Blöcke zeilen- oder spaltenweise abgearbeitet, können in derselbe Blockgröße, wegen der Datenabhängigkeiten in den bereits errechneten Bereich jeweils versetzt, bereits weitere Halbiterationen stattfinden. Da sich die Blöcke und die damit benötigten Daten überschneiden, kann die Cachehierarchie noch besser genutzt werden. Zu beachten ist allerdings, dass beim temporalen Blocking eine spezielle Behandlung an den Ränder des Gitters nötig ist.

Für den zwei- und dreidimensionalen Fall werden im folgenden effiziente Blocking-Verfahren vorgestellt, die vektorisiert mit dem in Abschnitt 5.1 vorgestellten Speicherlayout mit getrennten Feldern arbeiten. Diese können in einem Durchgang eine oder mehrere vollständige Iterationen durchführen.

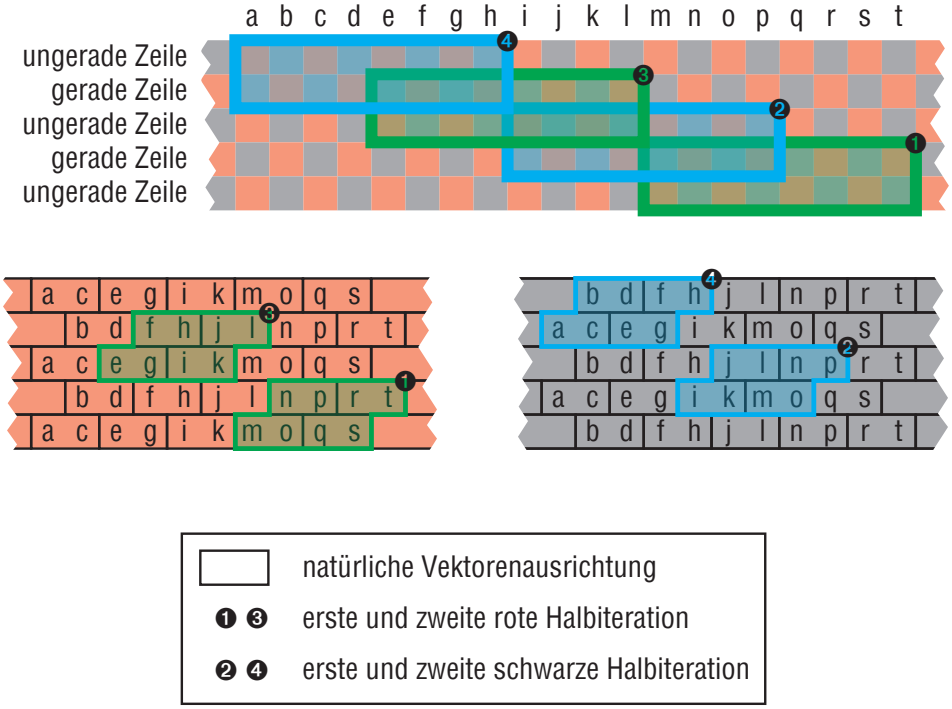
5.4.1 Blocking für den 2D Red-Black Gauss-Seidel

Für den zweidimensionalen Fall wurde ein Blocking-Verfahren mit Rechtecken gewählt, wobei die Blöcke zeilenweise abgearbeitet werden, ebenso die Werte innerhalb eines Blockes. Für gemischte Felder ist dieses Verfahren fast trivial, und für eine möglichst hohe Verarbeitungsgeschwindigkeit sind nur optimale Blöckgrößen zu ermitteln. Bei der Verwendung von getrennten Feldern und Vektorisierung wird das Verfahren aus algorithmischer Sicht komplexer.

Insbesondere für die Vektoreinheit müssen Ausrichtung und Größe der Blöcke berücksichtigt werden. Die Werte am linken Rand der Blöcke sollten an 16 Byte-Grenzen ausgerichtet sein, und die Blöcke in einem Halbfeld eine gerade Breite b besitzen, aus Gittersicht also ein Vielfaches von vier. Bei der Verschmelzung von mehreren Halbiteration zu einer oder mehreren Iterationen ist auch beim Verschieben der Blöcke auf Ausrichtung zu achten, auch wenn hiermit keine ganz optimale Überdeckung erreicht werden kann.

Die Beispiels-Implementation beschränkt sich auf Blöcke mit einer geraden Anzahl von Zeilen als Höhe h. Dadurch haben alle Blöcke dieselbe Verteilung von geraden und ungeraden Zeilen und können innerhalb des Gitters mit derselben Befehlsfolge berechnet werden. Abbildung 5.6 zeigt exemplarisch Form und Ausrichtung der Blöcke in den getrennten Feldern und die Verschiebung der Blöcke beim Verschmelzen mehrerer Halbiterationen.

Abbildung 5.6: Verschmelzen von zwei vollständigen Iterationen mit einem 8×2-Block in 2D



Hinzu kommt ebenfalls eine effiziente, blockartige Behandlung der Ränder. Im folgenden wird ein vollständiger Durchgang in zeitlicher Abfolge dargestellt:

Begonnen wird mit der Vorbereitung des linken oberen Ecks bei $U_{1,1}$. Hier werden bei der Zusammenfassung von i Iterationen zunächst alle roten Werte im Bereich von $U_{1,1} - U_{i \cdot 8-4, i \cdot 2-1}$ berechnet, anschließend alle schwarzen im Bereich $U_{1,1} - U_{i \cdot 8-8, (i-1) \cdot 2-2}$ usw. bis zuletzt die i-te rote Halbiteration auf $U_{1,1}$ und $U_{3,1}$ durchgeführt wird. Wird nur eine Iteration pro Durchgang durchgeführt, beschränkt sich dies auf die letzte rote Halbiteration (vgl. 5.7 Markierung 1 bis 3).

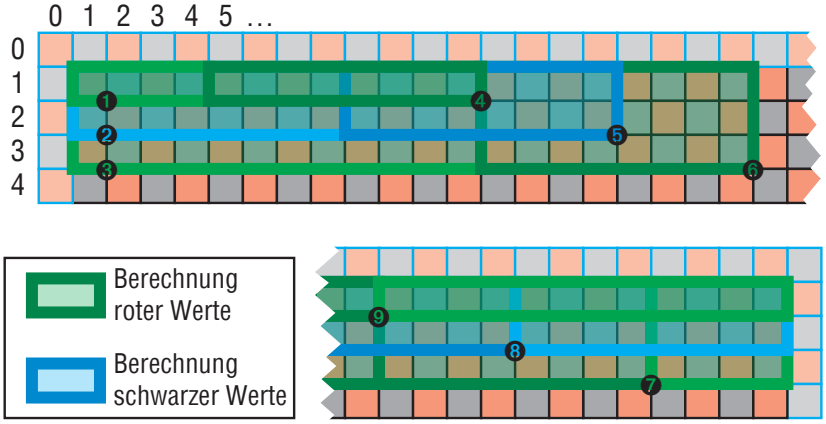
Anschließend wird die obere Kante blockweise abgearbeitet. Der erste Block setzt die Eckbehandlung fort und berechnet die roten Werte in einem $i \cdot 2 - 1$ Zeilen hohen Block mit der üblichen Blockbreite b. Werden mehrere Iterationen zusammengefasst, wird mit jeweils eine Zeile niedrige-

ren, aber ebenso langen Blöcken mit entsprechender Verschiebung fortgefahren (vgl. Markierung 4 bis 6). Dies wird für den Rest der Zeile fortgesetzt, solange vollständige Blöcke möglich sind.

Am Ende der Zeile müssen entsprechend verschieden kurze Blöcke genutzt werden, um eine Füllung bis zum Rand zu erreichen (Markierung 7 bis 9). Kann mit Hilfe der jeweils ersten roten Blöcke die Zeile gefüllt werden, muss kein Block für die erste rote Halbiteration mehr berechnet werden und der Markierung 7 entsprechende Block fällt weg. Soll nur eine Iteration pro Durchgang berechnet werden, reduziert sich diese Randbehandlung darauf, die rote Halbiteration in Zeile $y = 1$ zu berechnen.

Ist die Breite x des Gitters nicht durch vier ganzzahlig teilbar, sind alleine bei der Behandlung des rechten Randes nichtvektorisierte Zugriffe nötig.

Abbildung 5.7: Obere Randbehandlung bei zwei verschmolzenen Iterationen in 2D



Hier wurden Blöcke der Breite vier gewählt

Nach dieser Phase fehlt in der obersten Zeile $y = 1$ allein die letzte schwarze Halbiteration, bei den Zeilen darunter ist jeweils eine Halbiteration weniger durchgeführt worden. Die Behandlung der linken und rechten Ränder wird jeweils vor und nach dem Durchlaufen einer Zeile mit Blöcken durchgeführt.

Ausgehend von $U_{1,i\cdot 2}$ werden am linken Rand zunächst die roten Werte in einem Block der Breite $i \cdot 4 - 2$ und der Höhe h berechnet. Bei der Verschmelzung mehrerer Iterationen folgt dann um eine Zeile nach oben versetzt ein schwarzer Block derselben Höhe mit der Breite $i \cdot 8 - 4$ usw., bis zuletzt der rote Block ausgehend von $U_{1,2}$ mit Breite 4 berechnet wurde (vgl. Abbildung 5.8).

Danach kann mit den eigentlichen Blöcken wie oben beschrieben die Zeile hinweg fortgefahren werden.

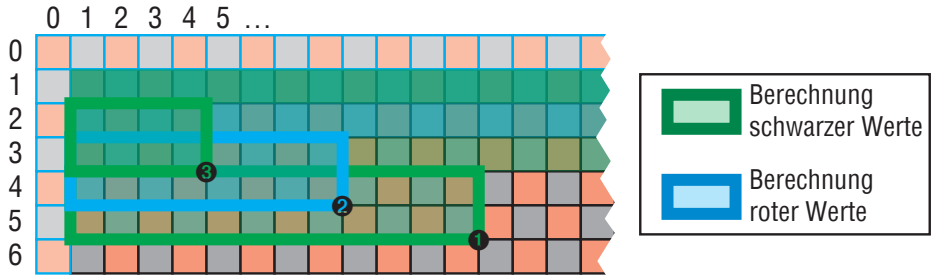
Am Ende der Zeile ist wieder eine spezielle Randbehandlung nötig, die die restlichen Werte bis zum Rand mit Blöcken berechnet.

Nach jeder Blockzeile sind alle verschmolzenen Iterationen in h Zeilen abgeschlossen worden. Darunter bildet sich wieder dieselbe treppenartige Struktur von Zeilen, in denen abnehmend häufig Halbiterationen durchgeführt wurden.

Je nach Blockhöhe können am unteren Rand noch Zeilen vorhanden sein, in denen noch keine Berechnung stattgefunden hat. Dann kann durch die Verwendung einer weiteren Blockzeile mit niedrigeren Blöcken erreicht werden, dass im ungünstigsten Fall – bir gerade Gitterhöhe y – nur in der untersten Zeile noch keine Berechnungen stattgefunden haben. Diese einzelne Zeile kann dann bei der Behandlung des unteren Randes berücksichtigt werden.

Die Behandlung des unteren Randes findet dann ähnlich wie am oberen Rand statt. Hierfür sind allerdings – im Gegensatz zur oberen Randbehandlung – zunächst einzeilige und dann immer höher werdende Blöcke nötig.

Abbildung 5.8: Linke Randbehandlung bei der zwei verschmolzenen Iterationen in 2D



Als Block-Höhe wurden zwei Zeilen gewählt

Die Wahl der Blockbreite hängt insbesondere davon ab, wieviele Iterationen in einem Durchgang ausgeführt werden sollen. Wird mit dem nächsten Block in der Blockzeile begonnen, sollten die Werte der vorhergehenden Block-Kaskade noch im Level 1-Cache vorhanden sein. Bei den verwendeten Testsystemen ist der Level 2-Cache so groß, dass auch bei der Verschmelzung mehrerer Iterationen die Daten aller Blöcke über die Gitterbreite darin gehalten werden können, was weiter unten genauer analysiert wird.

Für sehr lange Zeilen lässt sich der Algorithmus entsprechend erweitern. Während sich eine Anpassung der oberen und unteren Randbehandlung nicht lohnt, da die Programmausführung nur relativ wenig Zeit damit verbringt, gibt es im Mittelfeld die Möglichkeit, die Abarbeitung ganzer Cache-Zeilen zu teilen. Die Blockzeilen werden dabei nicht über die gesamte Breite berechnet, sondern eine weitere Blocking-Ebene eingeführt, so dass das Feld in mehreren Spalten abgearbeitet wird. Die Randbehandlung wird wie im einfacheren Fall vor bzw. nach den Blöcken am linken und rechten Rand durchgeführt.

Der Speicherbedarf für die Berechnung einer Blockzeile bei einer Blockhöhe von zwei Zeilen und i verschmolzenen Iterationen lässt sich folgendermaßen abschätzen: Es finden durch das temporale Blocking in $2 \cdot i + 1$ Zeilen Berechnungen statt. Davon werden in $2 \cdot i - 1$ Zeilen sowohl rote als auch schwarze Werte berechnet, wofür jeweils die Halbzeilen aus U_r, U_b, F_r und F_b benötigt werden. In den obersten und untersten Zeile werden nur Werte einer Farbe berechnet, wodurch neben der Halbzeile aus U_r und U_b nur entweder Werte aus F_r oder F_b benötigt werden. Hinzu kommen noch eine Halbzeile aus U_r oberhalb und U_b unterhalb der Blockzeile. Hieraus ergeben sich $(2 \cdot i - 1) \cdot 4 + 2 \cdot 3 + 2 = 8 \cdot i + 4$ Halbzeilen. Damit kann abgeschätzt werden, ob sich eine senkrechte Teilung des Gitters lohnt.

Werden vollständige Blockzeilen abgearbeitet, liegen die Daten der Blöcke in den vier getrennten Feldern jeweils sequentiell hintereinander, sodass in den acht- bzw. 16-fach assozativen Level 2-Caches kaum Konflikte auftreten, und wenig TLB-Einträge benötigt werden. Die Einführung von spaltenartigen Überblöcken hingegen führt zu größeren Adresssprüngen, die Cache-Konflikte provozieren und neben höherem Bedarf an Seiteninformationen auch die Effizienz des Hardware Prefetchers beeinträchtigen.

Der Red-Black Gauss-Seidel berechnet identische Ergebnisse auch mit vertauschten Dimensionen. Häufig ist deshalb die Berechnung vollständiger Cachezeilen mit transponierten Feldern einer Teilung vorzuziehen.

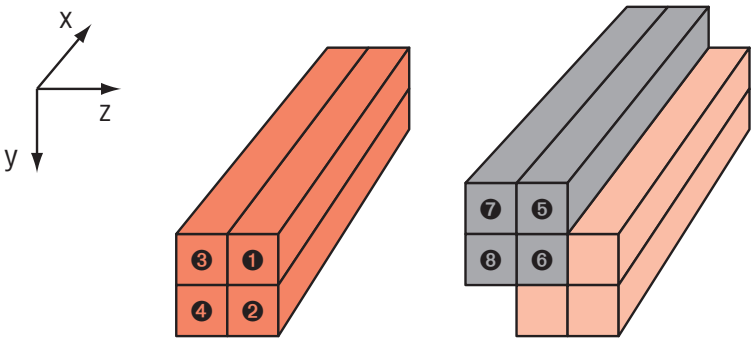
5.4.2 Blocking für den 3D Red-Black Gauss-Seidel

Im Dreidimensionalen wird ein Block mit der Größe von zwei mal zwei (Halb-)Zeilen verwendet. Für die roten Blöcke $U_{1,y,z} - U_{x_{max},y+1,z+1}$ wurde bei der Implementierung der Einfachheit halber y und z als ungerade vorausgesetzt, worauf der schwarze Block $U_{1,x-1,z-1} - U_{x_{max},y,z}$ folgt (vgl. Abbildung 5.9). Bei der Verschmelzung mehrerer Iterationen können weitere Blöcke hinzukommen. Die Verwendung einzeiliger Blöcke ist nicht sinnvoll, da die rote Zeile $U_{1 \ldots x_{max},y,z}$ und der nächstmögliche schwarze Block $U_{1 \ldots x_{max},y-1,z-1}$ bei getrennten Feldern keine gemeinsamen Daten nutzen.

Zur Durchführung einer Halbiteration in einer Zeile werden Daten aus sechs Halbzeilen benötigt. Der nur 16 kB große Level 1-Cache des Pentium 4 fasst demnach alle Werte für Zeilen mit über 600 Werten. Im 32 Bit-Adressraum kann wenn nötig durch Tauschen der Dimension diese Zeilenlänge stets unterschritten werden. Deshalb können stets vollständige Halbzeilen am Stück berechnet werden, um den Hardware Prefetcher zu unterstützen. Dieser muss in 3D mit zwei Datenströmen mehr als in 2D zurecht kommen.

Bei zwei mal zwei Zeilen großen Blöcken ist eine gute Nutzung gemeinsamer Werte auch beim Verschieben der Blöcke beim temporalen Blocking möglich. Bei der Verwendung breiterer Blöcke, die also Zeilen aus vielen Ebenen enthalten, steigt die Gefahr von TLB Misses und Cache-Konflikten. Höhere Blöcke sind dagegen denkbar, haben sich aber im Experiment ineffizienter als die Einführung einer weiteren Blocking-Ebene (s. u.) erwiesen.

Abbildung 5.9: Blöcke und deren Abarbeitungsreihenfolge in 3D



Eine spezielle Randbehandlung muss bei den Randflächen $y = 0$, $y = y_{max} + 1$, $z = 0$ und $z = z_{max} + 1$ durchgeführt werden. Da Blöcke immer über vollständige Zeilen reichen, ist an $x = 0$ und $x = x_{max} + 1$ keine explizite Randbehandlung mehr nötig.

Bei einem Durchgang mit i verschmolzenen Iterationen wird mit der Kante mit $y = 1, z = 1$ begonnen. Hier wird zunächst die erste rote Halbiteration im Block $U_{1,1,1} - U_{x_{max},2\cdot i,2\cdot i}$ durchgeführt, und für $i > 1$ mit der Berechnung der folgenden schwarzen Halbiteration im Block $U_{1,1,1} - U_{x_{max},2\cdot i-1,2\cdot i-1}$ fortgefahren, bis mit der i-ten schwarzen Halbiteration des Blocks $U_{1,1,1} - U_{x_{max},1,1}$ die Behandlung der Kante abgschlossen ist (vgl. Abbildung 5.10, Markierung 1 bis 4).

Die obere Randfläche bei $y = 0$ wird dann blockartig abgearbeitet. Die Berechnungen der Eckbehandlung fortführend wird mit einem zwei Zeilen breiten und $2 \cdot i$ Zeilen hohen roten Block begonnen und temporal geblockt mit entsprechend niedrigeren, aber ebenso breiten Blöcke fortgesetzt (Markierung 5 bis 8).

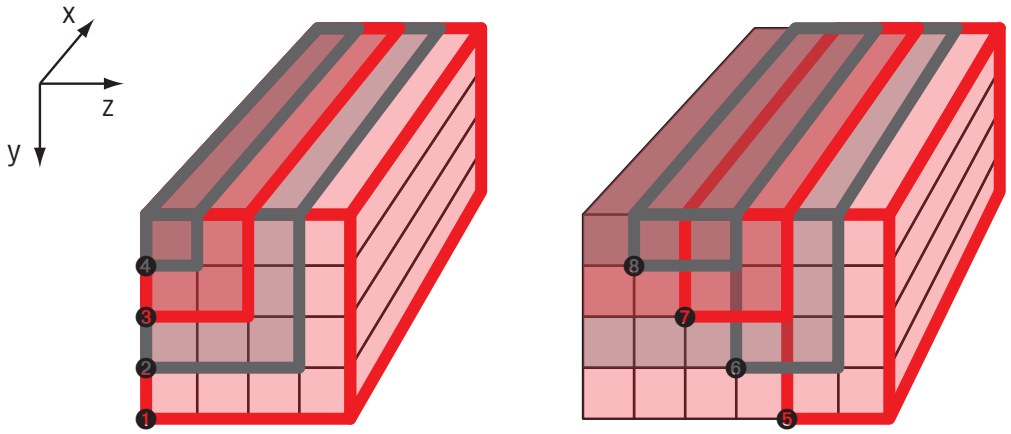
Die Behandlung der oberen Randfläche wird an der Kante $y = 0$, $z = z_{max}$ durch entsprechend lange Blöcke zum Füllen in der Breite abgeschlossen.

Nach Behandlung des oberen Randes sind alle Berechnungen in der obersten Zeilen mit $y = 1$ abgeschlossen, für $y = 2 \ldots$ nimmt die Anzahl der durchgeführten Rechenschritte ab, bis ab der Zeile $y = 2 \cdot i + 1$ noch keine Berechnungen durchgeführt wurden.

Bei der Abarbeitung innerhalb des Gitters wird – zumindest bei größeren Feldern – eine weitere Blocking-Ebene eingeführt. Diese Überblöcke teilen das Gitter in der x-z-Ebene in mehrere gleich hohe Quader und meist einen flacheren Restquader am unteren Rand. Die Behandlung der Randebenen bei $z = 0$ und $z = z_{max} + 1$ findet jeweils am Anfang und am Ende des Durchlaufens eines Überblocks statt.

Der erste Überblock beginnt in der Zeile $2 \cdot i + 1$, in der noch keine Berechnungen stattgefunden haben. Die Randbehandlung wird geblockt durchgeführt: Zunächst wird die erste rote Halbiteration im Block $U_{1,2\cdot i+1,1} - U_{x_{max},2\cdot i+2,2\cdot i}$ durchgeführt, anschließend bei temporalem Blocking die erste schwarze im Block $U_{1,2\cdot i,1} - U_{x_{max},2\cdot i+1,2\cdot i-1}$ usw. bis zuletzt in den Zeilen $U_{1\ldots x_{max},2,1}$ und $U_{1\ldots x_{max},3,1}$ die Berechnungen mit der i-ten schwarzen Halbiteration abgeschlossen werden. Dies

Abbildung 5.10: Behandlung der oberen Randfläche in 3D mit zwei verschmolzenen Iterationen



Links die Behandlung der linken oberen Kante, rechts das Blocking an der oberen Fläche

wird jeweils um zwei Zeilen in y-Richtung nach unten verschoben fortgesetzt, bis der Rand in der gesamten Höhe des Überblocks vorbereitet ist.

Darauf wird der mittlere Teil des Überblocks mit oben beschriebenen Blöcken durchlaufen. Die Blöcke werden jeweils um zwei Zeilen in y-Richtung bis zum Ende des Überblocks verschoben, so dass in einem Bereich der Höhe des Überblocks in zwei Ebenen die Berechnungen abgeschlossen werden. So wird jeweils mit zwei x-y-Ebenen breiten Streifen aus Blöcken die gesamte z-Dimension durchschritten, bis der Überblock mit einer Randbehandlung analog zu $z = 0$ bei der Randebene $z = z_{max}$ abgeschlossen wird.

Wenn nicht das gesamte Feld abzüglich oberer Randbehandlung durch die gleichmäßig großen Überblöcke komplett abgedeckt werden kann, ist noch die Berechnung eines flacheren Überblocks notwendig. Danach wurden nur in der x-z-Ebene mit $y = y_{max}$ noch keine Berechnungen durchgeführt, wenn y ungerade ist. Die Randbehandlung an der Randebene bei $y = y_{max}$ findet, unter spezieller Beachtung der Kanten, entsprechend konträr zur Randbehandlung bei $y = 0$ statt.

Grund für die Einführung der Überblöcke ist eine bessere Nutzung des Level 2-Caches. Um entscheiden zu können, ob und in welcher Höhe Überblöcke genutzt werden sollen, muss der Speicherbedarf einer Spalte aus Blöcken über die gesamte Höhe oder innerhalb einer Überblockzeile betrachtet werden. Die Anzahl an Halbzeilen n, die zur Berechnung einer Spalte aus Blöcken der Höhe h (also mit $\frac{h}{2}$ Blöcken übereinander) kann über folgende Formel berechnet werden:

$$n = 8 \cdot h \cdot i + 6 \cdot h + 16 \cdot i - 4$$

Eine darauf basierende Abschätzung der Überblockhöhen führt allerdings nicht immer zu einem optimalen Ergebnis. Neben der Assoziativität muss auch berücksichtigt werden, dass keine strikte Least-Recently-Used Verdrängungsstrategie realisiert wird und Cache-Zeilen deshalb nicht in exakter Reihenfolge verdrängt werden. Eine – auch in der Praxis erfolgreiche – vernünftige Faustregel ist es, die Höhe so wählen, dass zwei vollständige Blockstreifen im Level 2-Cache Platz hätten. Optimale Überblockhöhen wird man allerdings nur experimentell für eine bestimmte Problemgröße ermitteln können, die meist leicht größer als die so abgeschätzte Höhe ausfällt.

5.5 Software Prefetching

Insbesondere bei Blocking-Verfahren findet der Datentransfer aus dem Hauptspeicher nicht gleichmäßig, sondern in Schüben statt. Ein Teil der Operationen kann zwar fast vollständig auf Daten aus den Caches in hoher Geschwindigkeit operieren, andere Teile adressieren allerdings noch nicht in Caches befindliche Bereiche des Speichers. Werden später benötigte Daten zumindest teilweise durch

Prefetching in die Cache-Hierarchie geladen, während keine oder wenige Speicherzugriffe stattfinden, kann der die Geschwindigkeit begrenzende Hauptspeicherdurchsatz geglättet und insgesamt erhöht werden.

Wie in Abschnitt 3.2.2 bereits beschrieben, verfügen die Test-Systeme über Hardware-Prefetcher, unterstützen aber auch spezielle Prefetch-Instruktionen. Da die Hardware-Prefetcher nur begrenzte Fähigkeiten beim Erkennen von Datenströmen und weitere Einschränkungen besitzen – der Prefetcher des Penium 4 stoppt stets an 4 kB-Grenzen –, scheint die Nutzung von Software Prefetches sinnvoll. Deshalb wurde die Möglichkeit geschaffen, bei der Generierung der Assembler-Programme auch Prefetch-Instruktionen einfließen zu lassen.

Die Prefetch-Instruktionen werden jeweils im Anschluss an ausgerollte Berechnungen (vgl. Abschnitt 5.2.2) eingefügt.

Es bestehen auch spezielle Non-Temporal Prefetches, die ein Gegenstück zu den in Abschnitt 5.3 beschriebenen Non-Temporal Moves bilden. Mit diesen speziellen Prefetch-Instruktionen werden Daten nicht in die normalen Caches geladen, sondern in spezielle Puffer. Werden diese Adressen durch übliche Instrukionen gelesen, können damit „Non-Temporal Loads“ emuliert werden.

Dieses Verfahren ist speziell für die Operanden gedacht, die nur einmalig genutzt und nicht geändert werden. Sie verweilen nur kurz in den Puffern, unterliegen keiner speziellen Verdrängungsstrategie und können dann verworfen werden. Diese Instruktionen sind v.a. sinnvoll, um Cache Pollution zu vermeiden. Bei der Implementierungen des Red-Black Gauss-Seidel sinnvoll angewendet werden können diese Non-Temporal Prefetches für die Werte der rechten Gleichungsseite in F_r und F_b, wenn nur einzelne Iterationen pro Durchgang ausgeführt werden.

Kapitel 6

Benchmarking

Der Erfolg einer Optimierung kann nur durch Implementierung und Messung des Laufverhaltens beurteilt werden. In diesem Kapitel werden zunächst die für die beschriebenen Optimierungstechniken hergestellten Implemenationen beschrieben. Um die erzielten Leistungen bewerten zu können, werden im folgenden Vergleichspunkte geschaffen. Diese werden zum einen anhand theoretischer Überlegungen gewonnen, zum anderen werden bestehende optimierte Implementationen zum Vergleich herangezogen.

6.1 Getestete Implementationen

Die implementierten Optimierungsverfahren für den Red-Black Gauss-Seidel wurden über den m4-Macro-Prozessor generiert und mit dem GNU Assembler *as* als Unterfunktion übersetzt. Der erzeugte Assembler-Code besteht aus den Instruktionen des IA-32 Befehlssatzes und nutzt die SSE- bzw. SSE2-Erweiterungen, allerdings wurden jeweils auf den Prozessor optimierte, leicht unterschiedliche Instruktionssequenzen für die Berechnungen generiert (siehe Abschnitt 5.2.2).

Die Funktionen werden jeweils aus einem in 2D und 3D jeweils identischem Hauptprogramm in C aufgerufen, in dem auch die Laufzeitmessung und die dabei nicht berücksichtigte Initalisierung der Felder stattfindet. Auf der fauia42 stand dafür das Performance Application Programming Interface (*PAPI*) zur Verfügung, auf der fauia49 wurde der *clock* Systemaufruf zur Messung der Prozessorzeit verwendet.

Die Tests wurden jeweils ohne weitere Systemlast durchgeführt. Die HyperThreading genannte Technologie, mit der der Pentium 4 auf zwei logische Prozessoren aufgeteilt und für Symmetrisches Multi Threading genutzt werden kann, wurde deaktiviert.

Um den Erfolg der Optimierungsmethoden besser beurteilen zu können, wurden die Implementationen mit verschiedenen Problemgrößen getestet. Dabei wurde sich auf in allen Dimensionen gleich große Probleme konzentriert, da die Referenz-Implementationen nur dies unterstützen. Die optimalen Parameter wurden dabei bei einem größeren Problem möglichst optimal bestimmt und dann jeweils für den gesamten Testlauf verwendet.

Durch das auf der fauia42 vorhandene PAPI wurden während der Testläufe Daten aus den Hardware Performance Countern gesammelt, worauf zum Teil Bezug genommen wird. Auf der fauia49 stand diese Möglichkeit leider nicht zur Verfügung.

6.1.1 Red-Black Gauss-Seidel in 2D

Getestet wurden eine direkte Implementation des Algorithmus mit einzelnen Halbiterationen sowie das in Abschnitt 5.4.1 beschriebene Blocking-Verfahren, das mehrere vollständige Iterationen pro

Durchgang berechnen kann. In beiden Fällen wird das Datenlayout mit farblich getrennten Feldern aus Abschnitt 5.1 verwendet. Für jeden Prozessor wird entsprechend Abschnitt 5.2.2 leicht unerschiedlicher Code für die Berechnungen erzeugt.

Für die Verfahren mit getrennten Halbiterationen steht sowohl eine Version mit üblichen, temporalen Schreiboperationen, als auch eine Fassung mit Non-Temporal Writes zu Verfügung.

Zudem können in allen Varianten Prefetch-Instruktionen erzeugt werden, wobei hier nach Version und Prozessor auch unterschiedliche Datenströme verfolgt werden. Die Prefetch-Instruktionen werden jeweils nach dem Aufrollen von Berechnungen ausgeführt. Die Block-Version berechnet mehrere Zeilen, so dass am Ende einer Cache-Zeile größere Sprünge in den Datenströmen entstehen; die generierten Prefetch-Instruktionen berücksichtigen diese Sprünge soweit möglich.

Die folgenden Parameter werden vor der Code-Generierung festgelegt. Dadurch können viele Berechnungen und Fallunterscheidungen von Macro-Prozessor und Assembler übernommen werden:

Problemgröße: In x- und y-Dimension können verschiedene Größen angegeben werden. Um die Implementierung zu vereinfachen, müssen die Größe in x- ganzzahlig durch vier, in y- Richtung durch zwei teilbar sein.

Anzahl der Durchgänge und Iterationen pro Durchgang: Beeinflusst das Verhalten der zu temporalem Blocking fähigen Version. Die einfache Version führt entsprechend entsprechend mehr Durchgänge aus.

Anzahl aufgerollter Berechnungen: Wenn möglich, wird innerhalb von Schleifen Code für entsprechend viele Berechnungen ausgerollt. Dies stellt dann auch die Breite eines Blockes dar. Dieser Parameter muss gerade sein, um Zugriffe mit entsprechender Speicherausrichtung zu sichern. Dieser Parameter kann zum einen bei der Randbehandlung überschritten werden, zum anderen können nur einmalig durchlaufene Schleifen mit einem möglichen Rest verschmolzen werden.

Blockhöhe: Die Höhe der Blöcke wird gerade angenommen, damit nicht unterschiedliche Code-Sequenzen für Blöcke mit unterschiedlicher Verteilung von geraden oder ungeraden Zeilen nötig sind. Für den Pentium 4 war eine Höhe von zwei Zeilen grundsätzlich optimal, der Athlon 64 erreichte insbesondere bei der Verschmelzung mehrerer Iterationen erst mit höheren Blöcken optimale Ergebnisse.

Spaltenbreite: Das Gitter kann wie Abschnitt 5.4.1 mit einer weiteren Blocking-Ebene in Säulen geteilt werden. In der Praxis ist dies nur selten sinnvoll.

Prefetching und Prefetch-Distanz: Wenn gewünscht, können Prefetch-Instruktionen für entsprechend entfernte Daten erzeugt werden. Dieser Wert und muss das Verfahren an die anderen Parameter angepasst werden.

6.1.2 Red-Black Gauss-Seidel in 3D

Wie bei der Optimierung in 2D wurden hier mit dem in Abschnitt 5.1.2 beschriebenen Datenlayout sowohl eine einfache Version mit einzelnen Halbiterationen als auch das in Abschnitt 5.4.2 beschriebenen Blocking-Verfahrens implementiert. Ebenso kann bei der Version mit Halbiterationen zwischen üblichen und Non-Temporal Stores gewählt werden, und in allen Versionen angepasste Prefetch-Instruktionen erzeugt.

Prefetching, Anzahl der Durchgänge und temporal zu blockende Iterationen werden wie im Zweidimensionalen festgelegt. Desweiteren müssen folgende Parameter beachtet werden:

Größe: Für alle Dimensionen kann eine unterschiedliche Anzahl von Unbekannten gewählt werden. Zur Vereinfachung der Randbehandlungs-Generierung müssen alle Größen gerade sein.

Anzahl der aufgerollten Berechnung: Der Parameter hat dieselbe Bedeutung wie im Zweidimensionalen. Da in allen Versionen vollständige Halbzeilen berechnet werden, werden allerdings nie längere Berechnungen ausgerollt, solange Halbzeilen mindestens doppelt so viele

Unbekannte besitzen. Durch Setzen auf 1 können auch skalare Berechnungen erzwungen werden.

Überblock-Höhe: Bei der zu Blocking fähigen Version wird das Feld in entsprechend hohe Überblöcke eingeteilt. Außerdem kann die Abschätzung der Blockhöhe auch über die in Kapitel 5.4.2 vorgeschlage „Faustregel“ stattfinden, oder eine Unterteilung in Überblöcke verhindert werden.

6.2 Referenz-Impementationen

Der Erfolg oder Misserfolg von Optimierungen zeigt sich am deutlichsten im Vergleich zu anderen Implementationen.

6.2.1 Referenz für den 2D Red-Black Gauss-Seidel

Für 2D werden die Implementationen von Christan Weiss [Wei01] herangezogen, die im Rahmen des DiME Projektes[1] entwickelt wurden. Im folgenden werden die insgesamt neun Fortran-Programme mit ursprünglicher Bezeichnung – mit Aussparung der Randbehandlungen – kurz beschrieben:

rb1: Dies ist eine direkte Implementierung, die pro Durchlauf eine Halbiteration ausführt.

rb2 und rb3: Diese Versionen führen jeweils eine vollständige Iteration während dem Durchlaufen des Gitters aus. rb2 bewerkstelligt dies, indem direkt auf die Berechnung eines neuen Wertes des roten Punktes $U_{x,y}$ jeweils der darüber liegende schwarze Wert $U_{x,y-1}$ berechnet wird. rb3 verwendet vollständige Zeilen als Blöcke und berechnet jeweils nach den roten Werten in Zeile y die schwarzen Werte in Zeile $y-1$.

rb4, rb5 stellen die Erweiterung von rb2 und rb3 zur Berechnung von i Iterationen in einem Durchgang dar. rb4 beginn dazu wie rb3, fährt dann aber entsprechend häufig in den darüber liegenden Zeilen mit den abwechselnd roten und schwarzen Berechnungen fort. rb5 kombiniert dies mit rb2. In zwei übereinanderliegenden Zeilen wird direkt nach dem roten Wert in der unteren der schwarze Wert in der darüberliegenden Zeile berechnet.

rb6, rb7 und rb8: rb6 stellt eine direkte Erweiterung von rb2 dar, indem nach der Berechnung eines roten Punktes jeweils entsprechend viele schwarze und rote Punkte dahinter aktualisiert werden. rb7 und rb8 führen die Berechnungen in identischer Reihenfolge wie rb6 aus. Allerdings wurden hier mehrere Berechnungen in Fortran ausgerollt, um ein besseres Scheduling durch den Compiler zu ermöglichen. In rb8 wurde die Reihenfolge der Berechnungsanweisungen zudem von Hand angepasst, um den Compiler beim Schedulen weiter zu unterstützen.

rb9: rb9 implementiert ein Block-Verfahren mit Über-Blöcken in Parallelogramm-Form, das in Abbildung 6.1 grob skizziert wird (*Skewed Blocking*). Für eine genaue Erläuterung wird auf [Wei01] verwiesen.

Allerdings musste die eigentliche Berechnung im Code verändert werden, da dieser den Red-Black Gauss-Seidel für ein anderes Problem modellierte. Die ursprüngliche Berechnung lautete

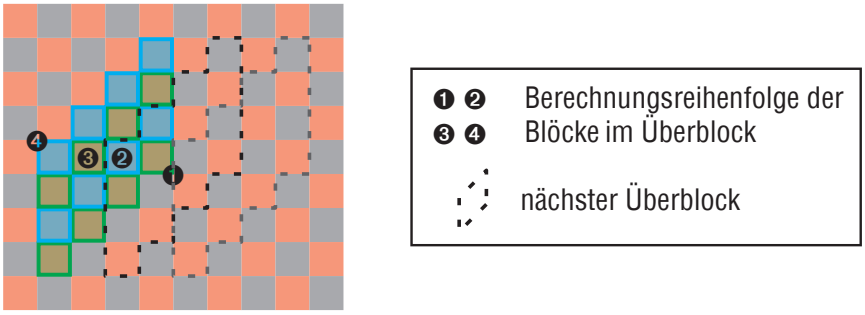
$$U_{x,y} = c_N \cdot U_{x,y-1} + c_S \cdot U_{x,y+1} + c_E \cdot U_{x+1,y} + c_W \cdot U_{x-1,y} + F_{x,y}$$

wobei c_x verschiedene, während der Ausführung konstante Koeffizienten darstellt.

Als Vergleichswert für eine bestimmte Problemgröße wird jeweils die schnellste, vergleichbare Version genutzt. Für die einfache Version, die Halbiterationen jeweils auf dem ganzen Gitter ausführt, und beim Blocking mit nur einer Iteration pro Durchgang sind dies rb1 bis rb3. Werden i Iterationen temporal geblockt, werden ebenfalls die Versionen rb4 bis rb9 mit i und weniger geblockten Iterationen zum Vergleich herangezogen.

[1] DiME steht für „Data Local Iterative Methods For The Efficient Solution of Partial Differential Equations“. Homepage http://www10.informatik.uni-erlangen.de/Research/Projects/DiME-new/index.html

Abbildung 6.1: Skizziertes 2D Skewed Blocking bei zwei verschmolzenen Iterationen



Die gemessenen Geschwindigkeiten sind den Abbildungen 6.2, 6.3 und 6.4 zu entnehmen. In letzteren sind jeweils die Maximalwerte der vorhergehenden Versionen ohne Bezeichnung eingetragen.

Abbildung 6.2: Referenzwerte vom 2D Red-Black Gauss-Seidel mit getrennten Halbiterationen oder einer verschmolzenen Iteration

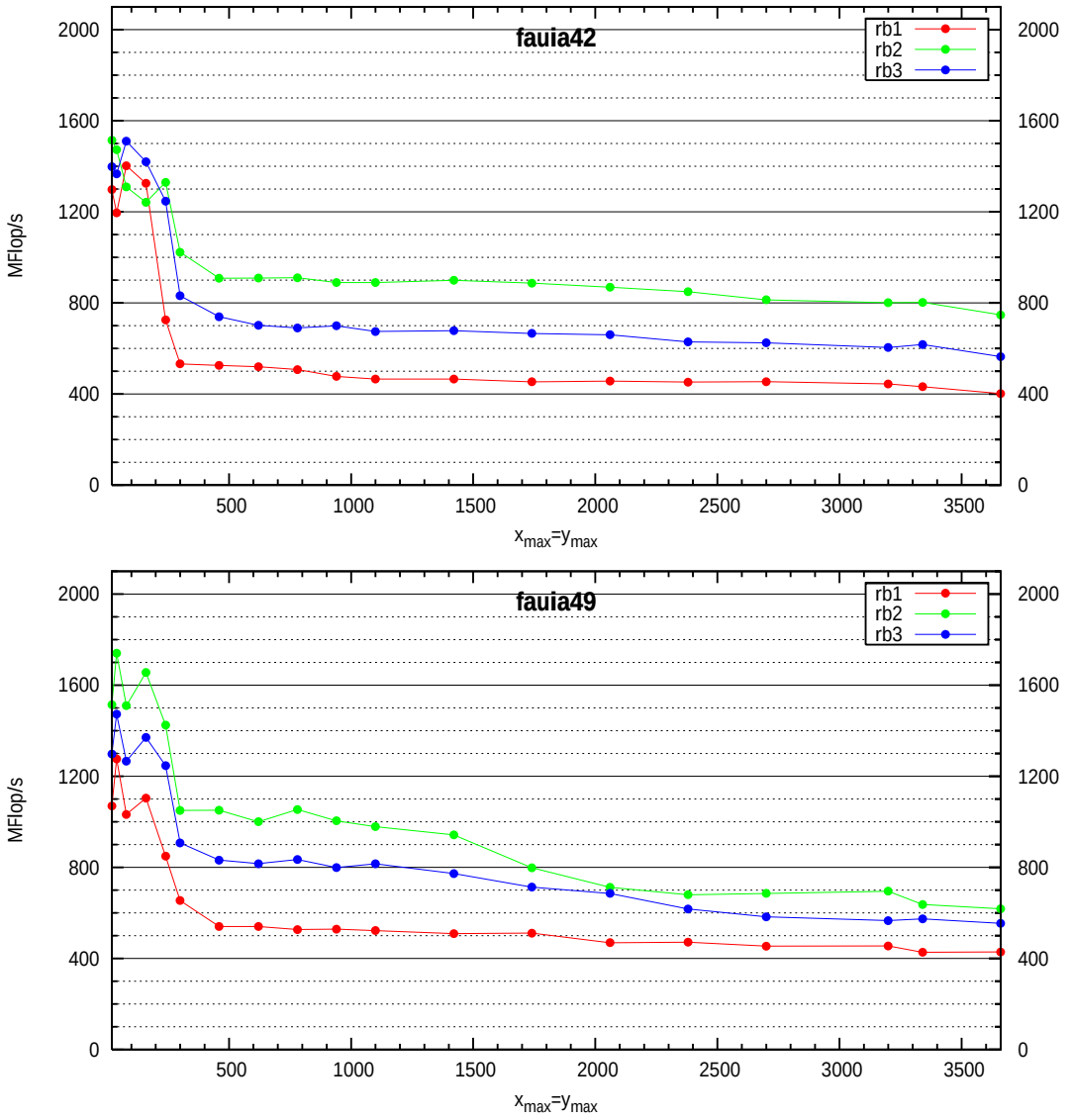


Abbildung 6.3: Referenzwerte vom 2D Red-Black Gauss-Seidel mit zwei verschmolzenen Iterationen

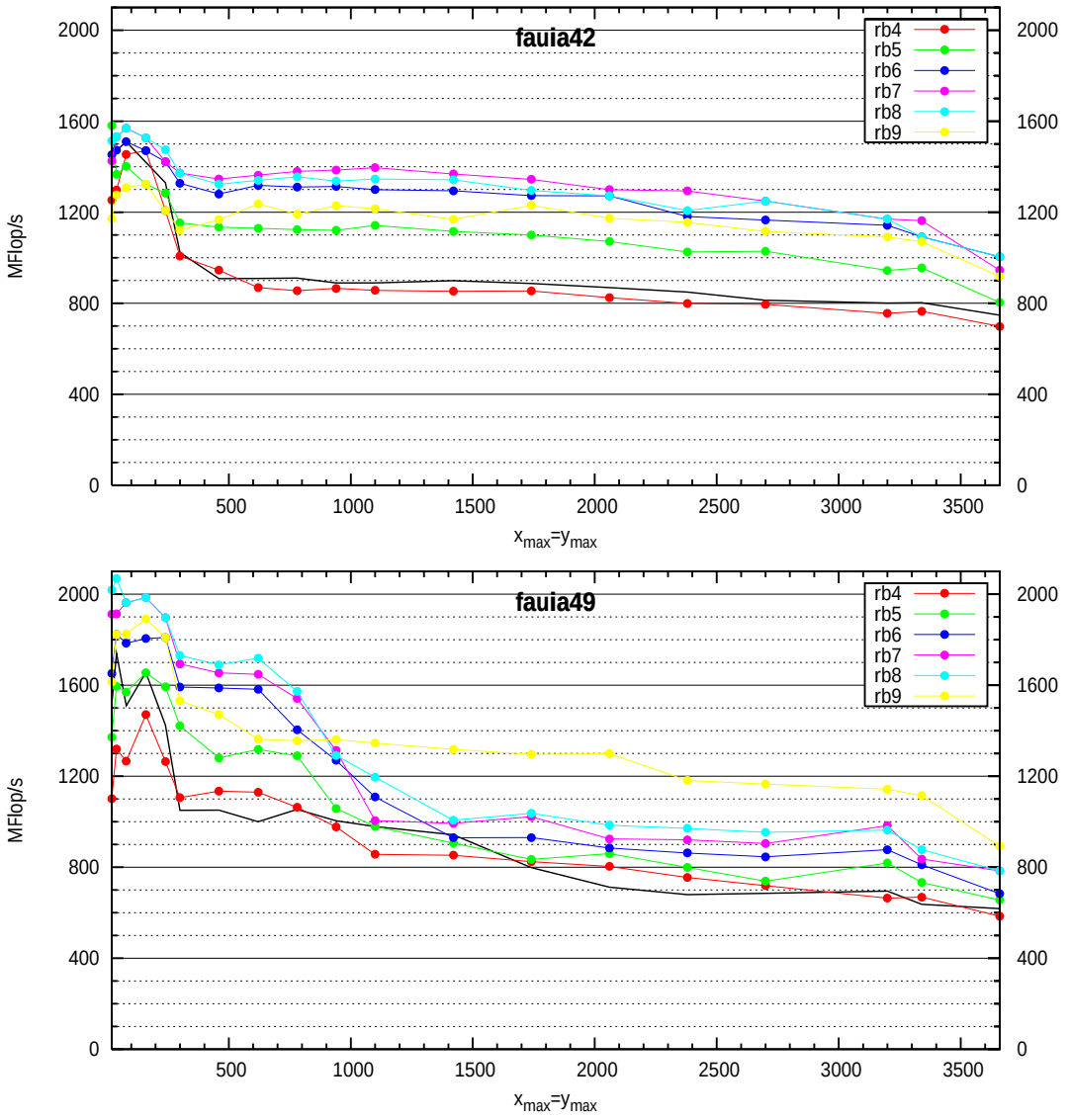
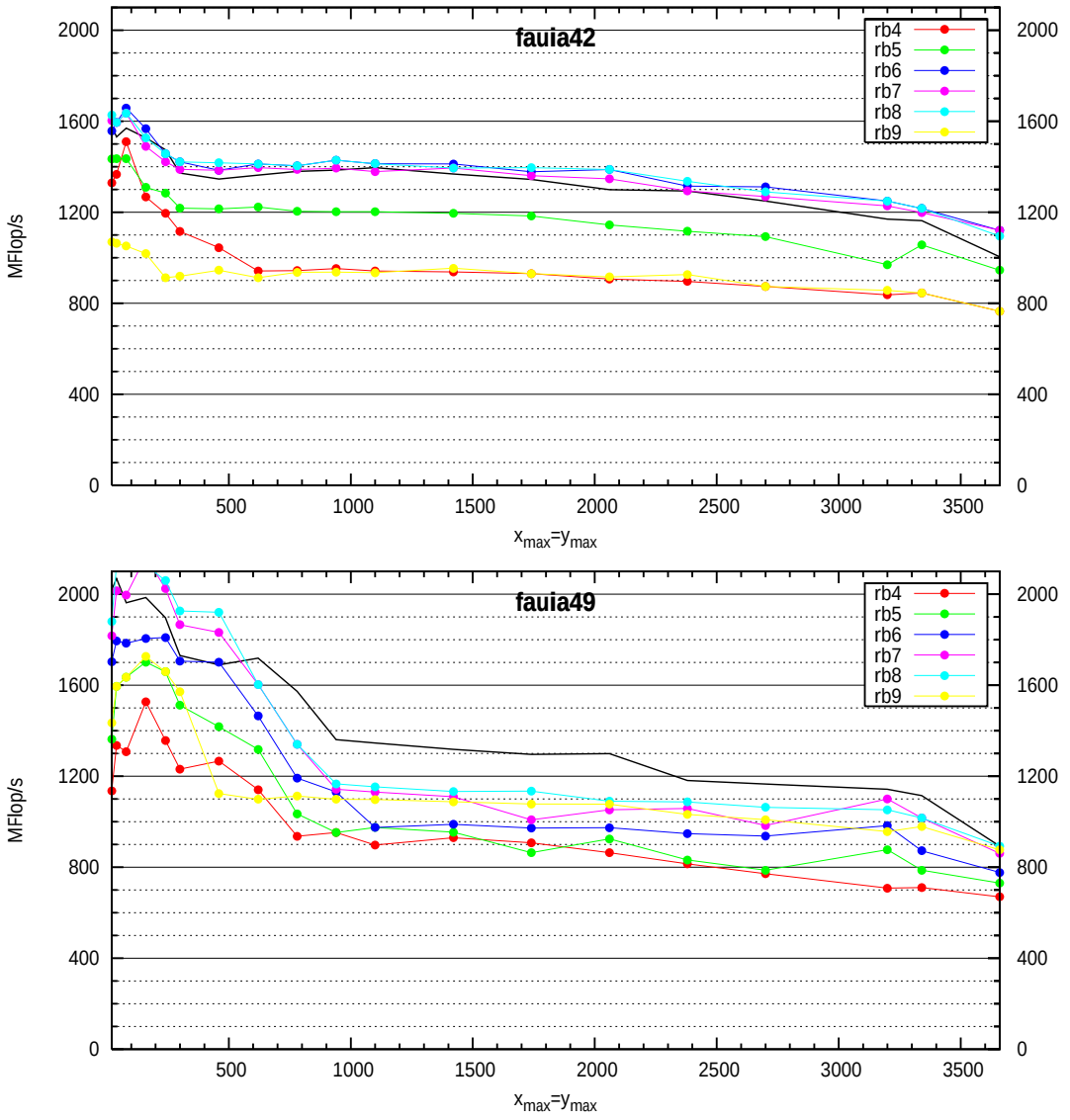


Abbildung 6.4: Referenzwerte vom 2D Red-Black Gauss-Seidel mit drei verschmolzenen Iterationen



6.2.2 Referenz für den 3D Red-Black Gauss-Seidel

Die Referenzwerte wurden hier aus den Implementationen von Nils Thürey [Thü02] gewonnen. Es stehen folgende Versionen zur Verfügung:

Standard: Deine direkte Implementierung des dreidimensionalen Red-Black Gauss-Seidel mit getrennten Halbiterationen.

Fused: Eine vollständige Iteration wird in einen Durchgang zusammengefasst, indem direkt nach dem roten Wert $U_{x,y,z}$ der schwarze Wert $U_{x,y,z-1}$ aktualisiert wird.

1-Way Blocking: Dies stellt die Erweiterung der Fused-Variante dar. Für i Iterationen werden nach $U_{x,y,z}$ auch die dahinter liegenden Werte $U_{x,y,z-1\ldots z-2\cdot i}$ berechnet, danach wird mit $U_{x+2,y,z}$ ebenso fortgefahren. Dies entspricht in der Terminologie von Abschnitt 5.4 Blöcken von einem Punkt Größe. Es können allerdings Varianten erzeugt werden, bei denen andere Blöcke wie vollständige Zeilen oder Ebenen gewählt werden, indem die zur Ausführung nötigen Schleifen getauscht werden (*Loop Interchange*).

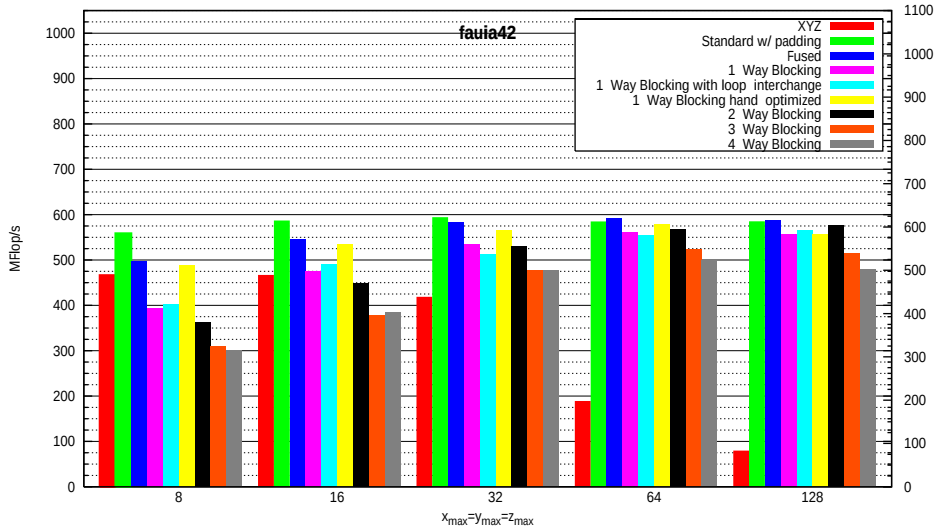
2-Way Blocking, 3-Way Blocking: Bei diesen Varianten werden weitere Blocking-Ebenen eingeführt. Beim 2-Way Blocking wird die x-Schleife geteilt, so dass Blöcke der Größe einer Teilzeile entstehen. Beim 3-Way Blocking wird eine Ebene in Rechtecke zerlegt.

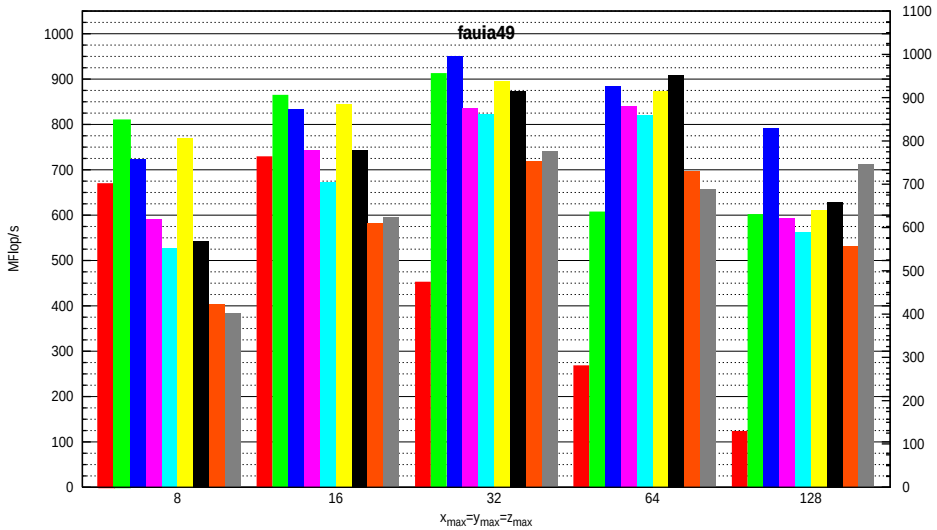
4-Way Blocking: Wird eine weitere Blocking-Ebene eingeführt, werden jeweils Quader innerhalb des Gitters als Block genutzt. In diesem Fall genügt es allerdings nicht, den Block zum Verschmelzen mehrerer Halbiterationen nur in eine Richtung zu verschieben, um den Datenabhängigkeiten genüge zu tun. In diesem Fall wird in allen Dimensionen um -1 verschoben.

Die Berechnung musste geändert werden, da sie bisher unserem Modellproblem mit $dh = 1$ entsprach und damit eine Multiplikation weniger durchgeführte. Deshalb werden die Werte auf F jeweils mit einer Konstante multipliziert.

Dem Source-Code liegen Skripte bei, die die verschiedenen Versionen mit unterschiedlichen Parametern für verschiedenen Blockgroßen und verschiedenen Padding-Varianten zum Reduzieren von Cache-Konflikten mit wenigen Problemgrößen testet. Maximal werden dabei zwei Iterationen pro Durchgang zusammengefasst. Als Vergleichswert wird die höchste Geschwindigkeit aus allen Versionen bei einer bestimmten Größe verwendet. Auf beiden Plattformen wurde der Intel Fortran Compiler genutzt, auf der fauia49 mit 64-Bit-Erweiterungen.

Abbildung 6.5: Referenzwerte für den 3D Red-Black Gauss-Seidel





6.3 Theoretische Vergleichswerte

Referenzwerte anhand von Überlegungen sind meist nicht zu erreichende Obergrenzen. Ihr Bestimmung wird dadurch erschwert, dass zum einen nur beschränkt Informationen über die Hardware vorhanden sind, es aber auch nicht möglich ist, das Zusammenwirken der Einheiten im Prozessorkern untereinander oder gar mit der Speicherhierarchie einfach zu modellieren. Deshalb können die im folgenden hergeleiteten Vergleichswerte auch nur eine Richtschnur sein. An Aussagekraft gewinnen sie allerdings, wenn man sie zusätzlich zum Vergleich verschiedener Implementierungen und Optimierungsverfahren nutzt.

6.3.1 Instruktionsdurchsatz

Die gängige Einheit zur Messung von Geschwindigkeit ist die Anzahl der Fließkomma-Operationen pro Sekunde, wobei bei der Bewertung von numerischen Codes hier mathematische Operatoren gemeint sind. Die Anzahl der vom Prozessor tatsächlich ausgeführten Instruktionen kann je nach Modell und Code-Erzeugung durchaus variieren: So können bei RISC-Prozessoren Divisionen meist nur durch Multiplikation mit dem Kehrwert erreicht werden, oder Vektorrechner können mehrere Berechnungen in einer Operation durchführen. Wird dieselbe Berechnung in Schleifen mehrfach ausgeführt, so bleibt das Verhältnis von logischen und internen Operationen konstant.

Eine Obergrenze kann über eine Analyse der verwendeten Instruktionen und ihrer Verarbeitung in den verschiedenen Ausführungseinheiten der Prozessoren gefunden werden. Es wird von ausschließlich vektorisierten Berechnungen ausgegangen, sowohl Datenabhängigkeiten als auch die Ablaufsteuerung werden hier nicht berücksichtigt. Es wird davon ausgegangen, dass der Scheduler aus einem unbegrenzten Instruktionsfenster auf ein unendliches Register File Instruktionen starten kann.

Folgende Instruktionen müssen für die vektorisierte Berechnung eines neuen Wertes ausgeführt werden:

- Laden des unausgerichteten Vektors (über zwei Befehle)
- Laden des Vektor aus F
- vier Additionen (in 3D: sechs), davon drei (in 3D: fünf) mit Operanden im Speicher
- zwei Multiplikationen
- Schreiben des Ergebnisvektors

Die Ausführung dieser Instruktionen entspricht im mathematischen Sinne zwölf Fließkommaoperationen (im Dreidimensionalen: 16).

Für Vektoradditionen steht beim Pentium 4 Prescott nur eine Einheit zur Verfügung. Da diese erst nach einem Takt Pause neue Additionen beginnen kann, sind mindestens acht (3D: zwölf Takte) zur Ausführung nötig.

Bei diesem vereinfachten Modell kann keine stärkere Einschränkung bewiesen werden. Multipliziert mit der Taktfrequenz von 3,2 GHz ist die Obergrenze für den zweidimensionalen Algorithmus 4,8 GFlop/s und für den dreidimensionalen knapp 4,3 GFlop/s. Dennoch scheinen diese Grenzen unrealistisch. Zum einen muss hierfür ein optimales Scheduling erfolgen. Zum anderen startet der Pentium 4 Operationen in der Hoffnung, dass die Operanden rechtzeitig aus dem Level 1-Cache eintreffen und muss diese sonst zweimal, einmal mit falschen und einmal mit den korrekten Operanden ausführen. Kann der Level 1-Cache für eine Berechnung nur einmal einen Operanden mit nur einem Takt Verzögerung liefern, erhöht sich diese Grenze um zwei Takte, und das Maximum läge bei 3,84 bzw. 3,66 GFlop/s.

Auch beim Athlon64 stellen die Additionen den begrenzen Faktor dar. Zwar kann die Adder-Einheit jeden Takt beschickt werden, allerdings werden die 128 Bit-Vektoren intern durch zwei 64 Bit-Berechnungen verarbeitet. Bei 2,4 GHz Prozessortaktung sind dadurch 3,6 GFlops für 2D und

3,2 GFlops in 3D möglich. Da beim Athlon64 mehrere Speicher- und Fließkommaoperationen pro Takt gestartet werden können, erscheinen diese Grenzen hier realistischer.

6.3.2 Minimaler Hauptspeicherdurchsatz

Ab einer gewissen Problemgröße kann davon ausgegangen werden, dass die zu Beginn eines Durchgangs benötigten Daten am Ende vollständig verdrängt wurden. Deshalb entspricht die Datenmenge, die pro Durchgang transferiert werden muss, einem Vielfachen des Speicherbedarfs eines Halbfeldes. Die Größe eines Halbfeldes kann als mindestens $m_{2D} = \frac{1}{2}x \cdot y \cdot 8$ Bytes bzw $m_{3D} = \frac{1}{2}x \cdot y \cdot z \cdot 8$ Bytes angenommen werden.

Ebenso kann die Anzahl der Berechnungen für eine Iteration und bei gleicher Anzahl von roten und schwarzen Unbekannten auch für eine Halbiterationen berechnet werden: $o_{2D} = \frac{1}{2}x \cdot y \cdot 6$ Flop/s und $o_{3D} = \frac{1}{2}x \cdot y \cdot z \cdot 8$ Flop/s.

Müssen für einen Durchlauf, in dem $n_{i\star}$ Halbiterationen durchgeführt werden, also n_m Halbfelder transferiert werden, bildet sich ein konstantes Verhältnis, wieviele Daten für einen Flop durchschnittlich übertragen werden müssen.

$$c = \frac{n_m \cdot m}{n_{i\star} \cdot o}$$

$$c_{2D} = \frac{n_m}{n_{i\star}} \cdot \frac{4 \text{ Bytes}}{3 \text{ Flop/s}}$$

$$c_{3D} = \frac{n_m}{n_{i\star}} \cdot \frac{\text{Bytes}}{\text{Flop/s}}$$

Für die einfache Variante mit getrennten roten und schwarzen Durchgängen werden für eine Halbiteration mindestens zwei Halbfelder gelesen und ein Halbfeld geschrieben. Werden keine Non-Temporal Moves eingesetzt, besteht allerdings die Gefahr, dass auch das Zielfeld in den Speicher geladen werden muss. Daraus folgt (die pessimistische Annahme in eckigen Klammern):

$$c_{2D} = 4\frac{\text{Bytes}}{\text{Flops}} \quad \left[\frac{16}{3}\frac{\text{Bytes}}{\text{Flops}}\right]$$

$$c_{3D} = 3\frac{\text{Bytes}}{\text{Flops}} \quad \left[4\frac{\text{Bytes}}{\text{Flops}}\right]$$

Werden i Iterationen in einem Durchgang zusammengefasst, müssen pro Durchgang zumindest die Felder U_r, F_r und F_b gelesen und U_r und U_b geschrieben werden. Insbesondere in diesem Fall ist es unwahrscheinlich, dass komplette Cache-Zeilen über Write Combining Store Buffer gesammelt werden können. Dennoch auch hier die pessimistische Annahme in Klammern:

$$c_{2D} = \frac{10}{3 \cdot i}\frac{\text{Bytes}}{\text{Flops}} \quad \left[\frac{4}{i}\frac{\text{Bytes}}{\text{Flops}}\right]$$

$$c_{3D} = \frac{5}{2 \cdot i}\frac{\text{Bytes}}{\text{Flops}} \quad \left[\frac{3}{i}\frac{\text{Bytes}}{\text{Flops}}\right]$$

Ist der maximale Speicherdurchsatz t bekannt, kann so errechnet werden, welche GFlop-Rate so theoretisch erreichbar ist.

$$r_{max} = \frac{t}{c}$$

Wurde allerdings die Geschwindigkeit eines Programmes gemessen, kann sich dadurch errechnen lassen, welcher Hauptspeicherdurchsatz mindestens stattgefunden hat.

$$t_{min} = c \cdot r$$

Das theoretische Maximum beim verwendeten DDR-SDRAM mit 400 MHz liegt bei 6,4 GB/s. In der Praxis ist dies allerdings nicht zu erreichen. Tabelle 6.1 zeigt durch das Testprogramm lssbench[2] ermittelte, erreichbare Transferraten auf den Testsystemen.

Tabelle 6.1: Hauptspeicherdurchsatz der Testsysteme

	fauia42	**fauia49**
Lesen	5,8	5,8
Lesen mit Prefetching	5,9	6,1
Schreiben	1,8	2,6
Non-Temporal Writes	4,8	6,0

6.3.3 Prozessor I/O

Die Lese- und Schreiboperationen sind anhand des Assemblercodes genau herauszufinden. Demnach werden zwei Halbvektoren, und vier volle (3D: sechs) Vektoren gelesen und ein Vektor geschrieben. Da davon ausgegangen werden kann, dass bei diesem Verfahren berechnete Werte erst nach einiger Zeit wieder als Operanden genutzt werden, kann Store Forwarding ausgeschlossen werden.

Das Speicherinterface des Athlon64 kann pro Takt zwei beliebige, maximal 8 Byte große Transfers durchführen, was hier theoretisch zwei Flop/Takt zuließe.

Beim Pentium 4 muss die Art der Speicherzugriffe genauer berücksichtigt werden. Pro Takt können je ein Lese- und ein Schreib-Befehl abgesetzt werden. Aufgrund der speziellen 128 Bit-Datenleitung zur SSE kann damit auch ein kompletter Vektor geladen werden. Weil in diesem Fall ein Vektor der Operanden nicht ausgerichtet ist, benötigt der Prozessor mindestens sechs Takte, um alle Daten für die Berechnung eines Vektors in 2D anzufordern, und acht Takte in 3D. Das Ausschreiben eines Ergebnisses kann gleichzeitig mit den Lese-Operation stattfinden.

Offensichtlich wären dies keine begrenzenden Faktoren, wenn man davon ausgeht, dass in jedem Takt auch ein Datum geliefert wird. Tabelle 6.2 zeigt je nach Aktion den von lssbench ermittelten maximalen Durchsatz in die verschiedenen Cache-Ebenen, der in der Praxis erreicht werden kann. Da die Level 1-Caches der Systeme Lese- und Schreiboperationen gleichzeitig durchführen können, sollte der Durchsatz in jeder Richtung getrennt betrachtet werden.

Das Verhältnis zwischen gelesener Datenmenge und damit ausgeführten Berechnungen ergibt sich dann zu $\frac{5 \cdot 8 \text{ Bytes}}{6 \text{ Flop}}$ in 2D und $\frac{7 \cdot 8 \text{ Bytes}}{8 \text{ Flop}}$ in 3D, geschrieben werden jeweils $\frac{8 \text{ Bytes}}{6 \text{ Flop}}$ bzw $\frac{8 \text{ Bytes}}{8 \text{ Flop}}$.

Tabelle 6.2: Cache-Durchsatz der Testsysteme

	fauia42	**fauia49**
Lesen aus L1 [GB/s]	23,8 (46,4[1])	34,5[2]
Schreiben in L1 [GB/s]	writethrough	34,3[2]
Lesen aus L2 [GB/s]	14,7 (25,5[1])	11,8
Schreiben in L2 [GB/s]	8,0 (10,9[1])	9,2

[1] Vektoroperationen 16 Byte
[2] kann für SSE nur mit Chaining erreicht werden

[2] lssbench wurde und wird am Lehrstuhl für Systemsimulation der Friedrich-Alexander Universität in Erlangen/Nürnberg entwickelt

Kapitel 7

Bewertung der Optimierungen

Die Testergebnisse der verschiedenen Implementationen ermöglichen es, den Erfolg der verschiedenen Optimierungsmethoden auf den beiden Systemen zu bewerten.

Die Analyse der Versionen mit getrennten Halbiterationen sind hierbei besonders aussagekräftig für die Effizienz der Vektorisierung und des dafür nötigen Speicherlayouts. Nur im Vergleich zu diesen Ergebnissen ist zudem der Erfolg der Blocking-Verfahren zu beurteilen.

Das Scheduling zeigt sich an den schnellsten gemessenen Verarbeitungsgeschwindigkeiten. Insbesondere bei geblockten Codes ist ein effektives Scheduling notwendig, da nur so beim Verschmelzen von zwei oder mehreren Halbiterationen die beabsichtigte hohe temporale Nähe auch in höhere Verarbeitungsgeschwindigkeit resultiert. Allerdings hat das Scheduling selbst bei rein serieller Verarbeitung von Daten Einfluss, da hier wegen lokaler Nähe ein Teil der Berechnungen direkt auf den Caches stattfindet. Diese sind zusammen mit weiteren Kennwerten in Tabelle 7.1 am Ende des Kapitels aufgeführt.

Dort sind in in den Tabellen 7.2 und 7.3 auch Kennwerte für die jeweils schnellsten Versionen und die Referenzwerte bei jeweils einer Problemgröße berechnet und gegenübergestellt worden. Diese können mit den ermittelten Grenzen und gemessenen Transferraten aus Abschnitt 6.3 verglichen werden.

7.1 Vektorisierung und Speicherlayout

7.1.1 2D Red-Black Gauss-Seidel mit einzelnen Halbiterationen

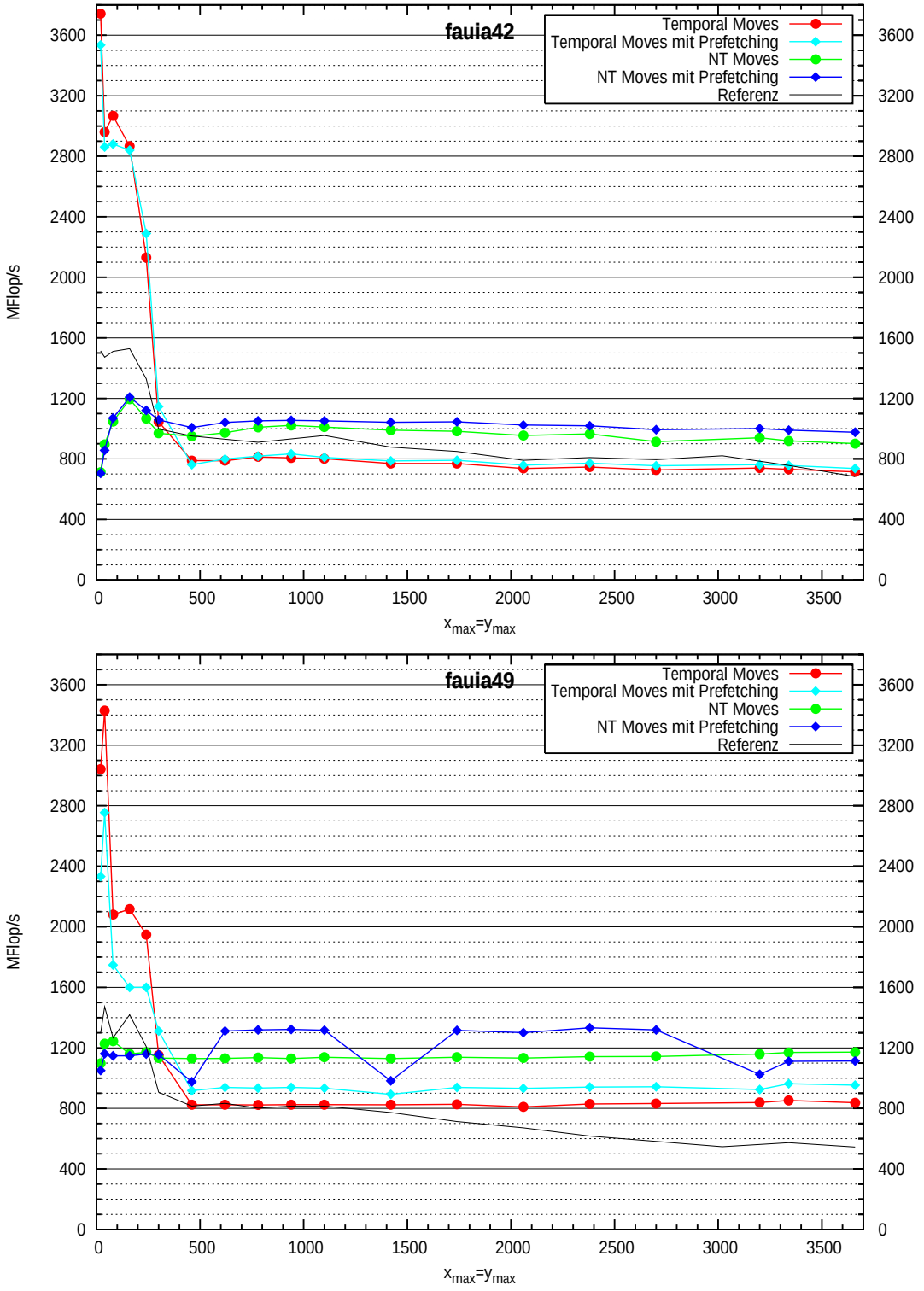
Die gemessenen Geschwindigkeiten sind für den 2D Red-Black Gauss-Seidel in Abbildung 7.1 zu finden. Als Referenz wird jeweils der schnellste Wert von rb1, rb2 und rb3 verwendet, wobei dies auf beiden Prozessoren die Werte von rb2 sind, sobald nicht mehr alle Felder in den Caches gehalten werden können (ab ca. 300×300).

Innerhalb der Caches werden deutlich höhere Geschwindigkeiten als bei den Referenzimplementierungen erreicht. Beim Pentium 4 wirkt sich hier zum einen der häufig benötigte, sehr schnelle Level 2-Cache positiv aus, auch können die ausgerichteten Vektoren hier über den doppelt breiten Bus von der Vektoreinheit gelesen werden.

Obwohl rb2 eine vollständige Iteration pro Durchgang ausführt, können bereits die einfachen Versionen ohne Prefetching oder Non-Temporal Stores ähnliche Geschwindigkeiten vorweisen. Diese liegt jedoch deutlich über der von rb1, das ebenfalls nur eine Halbiteration pro Durchlauf berechnet (vgl. Abbildung 6.2). Inbesondere müssen durch das Speicherlayout mit getrennten Feldern hier weniger unbenötigte Werte übertragen werden müssen.

Bei größeren Problemen kann hier jedoch am meisten von der Verwendung von Non-Temporal Stores profitiert werden, die bei temporalem Blocking nicht eingesetzt werden können. Bei einer

Abbildung 7.1: Optimierter 2D-Red-Black Gauss-Seidel mit einzelnen Halbiterationen



Problemgröße von 2060×2060 konnte so die Geschwindigkeit auf dem Pentium 4 um 25 % auf 0,94 GFlop/s, auf dem Athlon64 sogar um 33 % auf 1,32 GFlop/s gesteigert werden.

Ursachen hierfür sind insbesondere der gleichmäßige Transfer in und aus dem Hauptspeicher und das vom Hardware Prefetcher leicht verfolgbare Zugriffsmuster. Indem berechnete Werte mit hohem Durchsatz an den Caches vorbei geschrieben werden, müssen hier nur vier Halbzeilen statt sechs vollständiger Zeilen bei rb2 zur Wiederverwendung gehalten werden. Bei der genannten Problemgröße wurde die Anzahl der Level 2 Load Misses auf dem Pentium annähernd halbiert.

Die Verwendung von Software Prefetches konnte auf dem Pentium 4 bei größeren Problemen die Geschwindigkeit maximal um gut 9 % steigern, beim Athlon maximal 15 % bei Temporal und 17 % bei Non-Temporal Stores. Überstiegen die benötigten Daten die Cache-Größe nicht, wurden durch das Einbringen dann unnötiger Prefetch-Instruktionen erwartungsgemäß schlechtere Ergebnisse erzielt.

7.1.2 3D Red-Black Gauss-Seidel mit einzelnen Halbiterationen

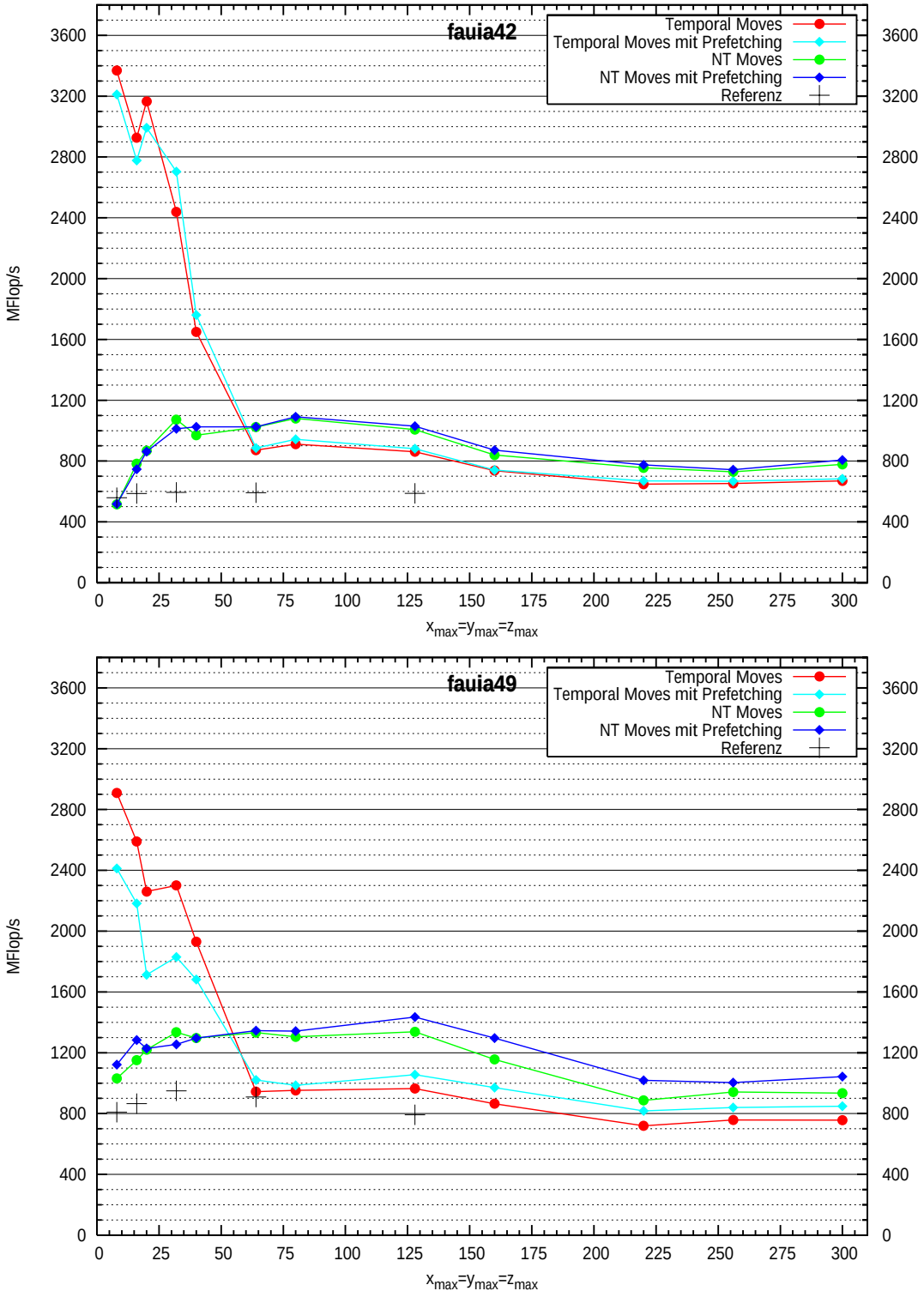
Wie in Abbildung 7.2 zu sehen ist, konnten durch Vektorisierung und das angewendete Speicherlayout bei Problemen in Cache-Größe sehr hohe Geschwindigkeiten erreicht werden.

Allerdings sinkt die Ausführungsgeschwindigkeit bei größer werdenden Problemen noch einmal bei ca. $128 \times 128 \times 128$, wenn bei der Berechnung der Ebene z die Daten aus der vorhergehenden Ebene $z-1$ nicht mehr in den Caches vorhanden sind. Während – soweit keine Cache-Konflikte auftreten – bei kleineren Problemen bei jeder Berechnung nur der Nachbar aus Ebene $z+1$ und der Wert F benötigt werden, müssen in diesem Fall auch der untere Nachbar und der Nachbar aus Ebene $z-1$ neu in den Speicher geladen werden.

Bei einer Problemgröße von $300 \times 300 \times 300$ konnte durch Non-Temporal Writes die GFlop-Rate auf dem Pentium 4 von 0,67 um 16 % auf 0,78, beim Athlon64 von 0,76 um 24 % auf 0,93 gesteigert werden. Da in 3D die geschriebenen Daten einen kleineren Anteil am Gesamtaufkommen haben, wirkt sich dies hier weniger stark als in 2D aus.

Während auf dem Athlon64 durch die Verwendung von Prefetch-Instruktionen bis zu 12 % an Geschwindigkeitssteigerung möglich war, konnten am Pentium 4 bei größeren Problemen nur zwei bis vier Prozent mehr erreicht werden.

Abbildung 7.2: Optimierter 3D-Red-Black Gauss-Seidel mit einzelnen Halbiterationen



7.2 Blocking-Verfahren

Beim temporalen Blocking wurden die Verschmelzung von ein, zwei und drei Iterationen betrachtet. Zum nimmt die Ausführungsgeschwindigkeit bei mehr verschmolzenen Iterationen nur noch gering zu, zum anderen sind bei Multi-Grid-Verfahren, die den Red-Black Gauss-Seidel häufig als Glätter einsetzen, nur selten mehr Schritte nötig.

7.2.1 Blocking in 2D

Die gemessenen Geschwindigkeiten für das zweidimensionale Blocking-Verfahren sind in den Abbildungen 7.3, 7.4 und 7.5 dargestellt.

Für den Pentium 4 wurden die höchsten Geschwindigkeiten stets mit einer Blockhöhe von zwei Zeilen erreicht. Eine mögliche Erklärung liefern die durch PAPI gesammelten Daten: Während bei der Verwendung höherer Blöcke deutlich weniger Level 1 Cache Misses auftreten, steigt die Anzahl der Level 2 Load und Store Misses deutlich an. Scheinbar ist dies kein singuläres Problem und kann deshalb wohl auch nicht durch einfache Padding-Techniken beseitigt werden. Durch die sehr schnelle Anbindung des Level 2-Caches kann auch eine weniger intensive Nutzung des Level 1-Caches in Kauf genommen werden, wenn dafür weniger Zeilen aus der Cache-Hierarchie verdrängt werden.

Für den Athlon64 trifft dies offensichtlich nicht zu, er erreichte die beste Performance meist bei sechs Zeilen hohen Blöcken ohne und vier Zeilen hohen Blöcken mit Software Prefetching. Eine Ausnahme stellt das temporale Blocking von nur einer Iteration dar, wo mit zwei Zeilen hohen Blöcken ähnliche Ergebnisse erreicht wurden.

Bei nur einer Iteration pro Durchgang konnten auf dem Pentium 4 die Referenzwerte nur für die Problemgrößen deutlich übertroffen werden, die entweder annähernd vollständig in den Caches gehalten werden können, oder die sehr groß sind. In einem Zwischenbereich liegen beide Löser gleichauf.

Der Athlon64 erreicht durchweg schnellere GFlop-Raten als die Referenz-Implementationen und die Berechnung einzelner Halbiterationen mit Temporal Moves. Mithilfe von Software Prefetching, das beim Pentium 4 die Geschwindigkeit nur um zwei bis vier Prozent erhöht, kann diese um 15 bis 20 % gesteigert werden.

Dennoch können auf beiden Systemen die Versionen mit Non-Temporal Moves kaum erreicht werden.

Bei der Verschmelzung mehrerer Iterationen kann im günstigstens Fall das entsprechende Vielfache erreicht werden. Auf dem Pentium 4 Prozessor skaliert das Blocking-Verfahren deutlich besser als auf dem Athlon64 und erreicht bei einer Problemgröße von 2060×2060 annähernd die $1,8$-fache bei zwei und über die $2,4$-fache Geschwindigkeit bei drei zusammen berechneten Iterationen. Der Geschwindigkeitszuwachs bei Software Prefetching liegt hier nur bei wenigen Prozent. Beim Athlon64 ist dieser deutlich höher und kann bei drei verschmolzenen Iterationen die GFlop-Rate um etwa 30 %, bei drei verschmolzenen Iterationen noch um bis zu 20 % steigern. Dennoch skaliert das temporalen Blocking nur mit dem gut $1,6$- respektive $1,8$-fachen.

Während die Performance insbesondere ohne Prefetching über die getesteten Problemgrößen hinweg relativ stabil bleibt, weisen alle Versionen des Blocking-Verfahrens auf beiden Plattformen einen Einbruch bei einer Problemgröße um 460×460 auf. Auf der fauia42 konnte hier ein ungewöhnlich häufiges Auftreten von Level 2 Load und Store Cache Misses festgestellt werden, und tritt auch bei Problemen mit anderer Höhe in diesem Bereich auf. Vermutlich handelt es sich hierbei um Cache-Konflikte, die bei bestimmen Zeilen oder Array-Größen auftreten.

Abbildung 7.3: Optimierter 2D-Red-Black Gauss-Seidel mit einer verschmolzenen Iterationen

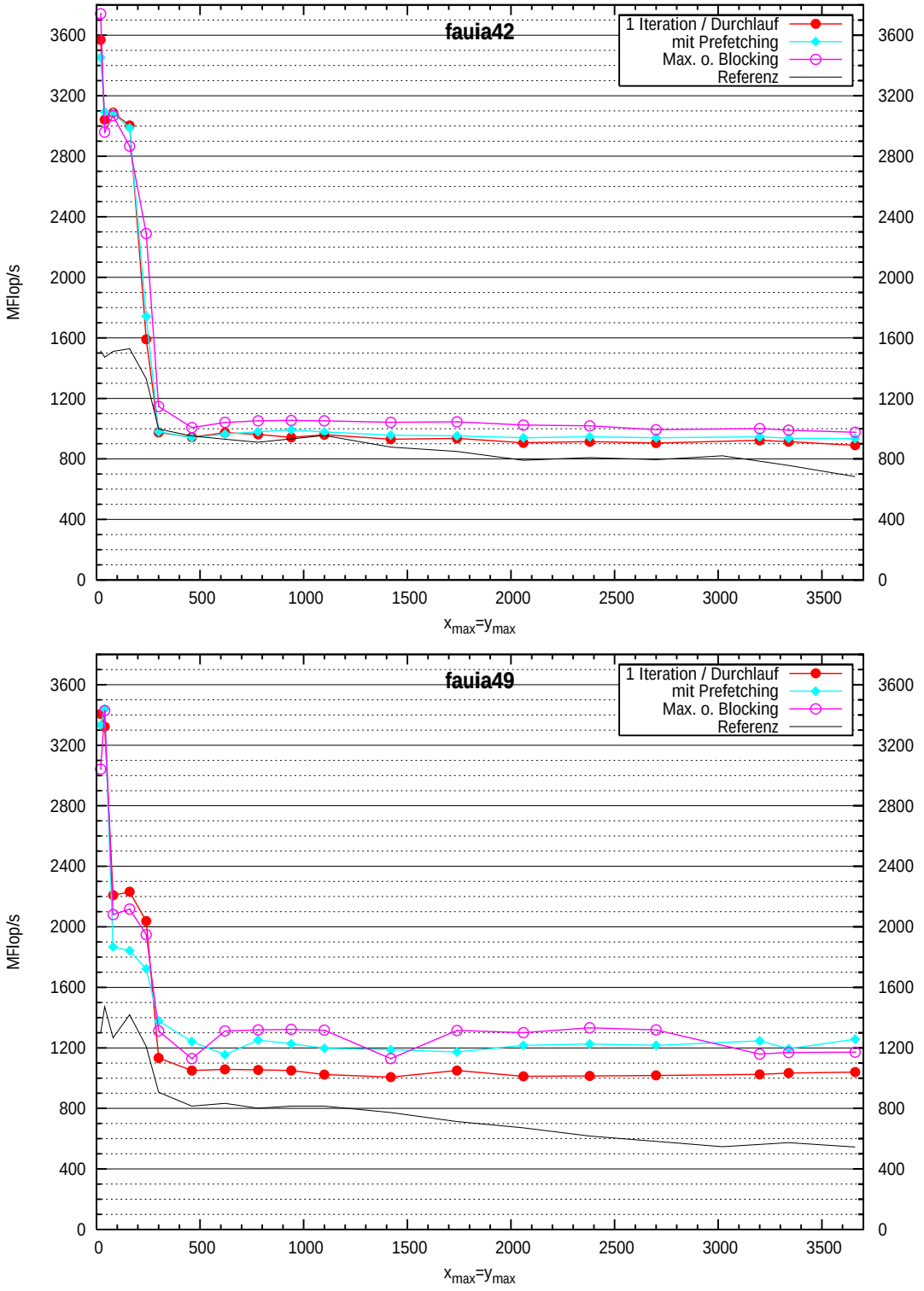


Abbildung 7.4: Optimierter 2D-Red-Black Gauss-Seidel mit zwei verschmolzenen Iterationen

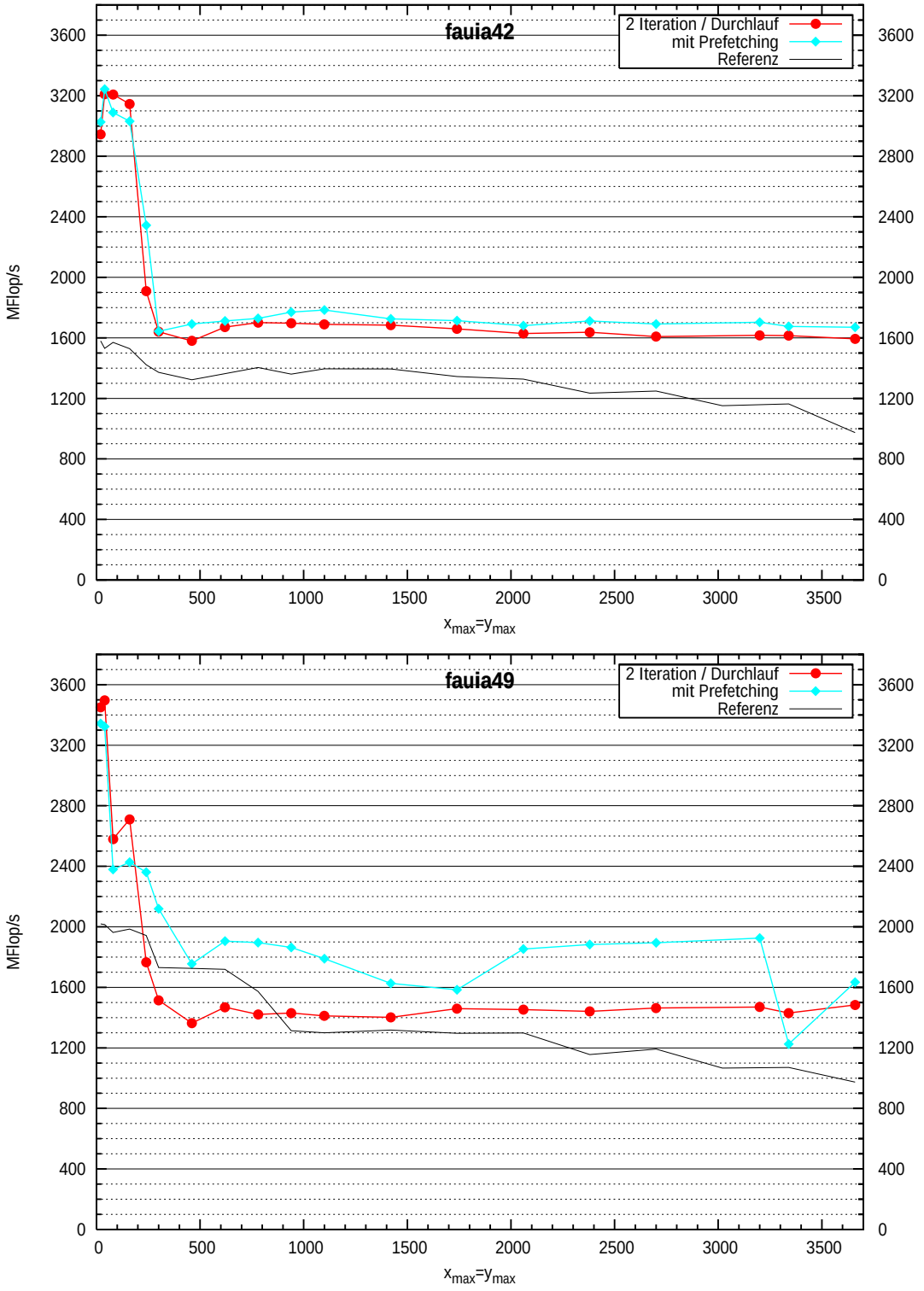
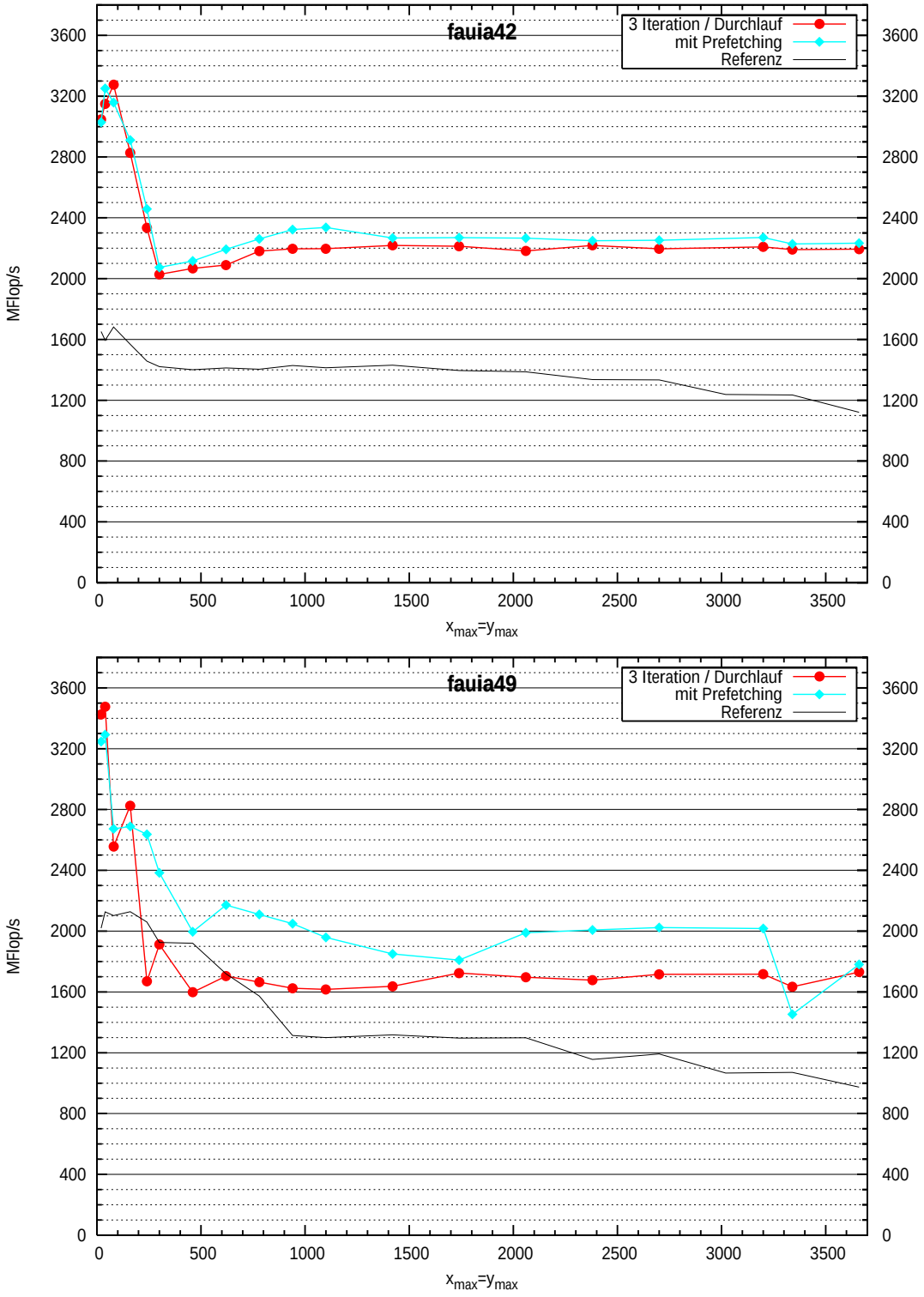


Abbildung 7.5: Optimierter 2D-Red-Black Gauss-Seidel mit drei verschmolzenen Iteration



7.2.2 Blocking in 3D

Die gemessenen Geschwindigkeiten des temporalen Blocking-Verfahrens für den dreidimensionalen Red-Black Gauss-Seidel sind in den Abbildungen 7.6, 7.7 und 7.8 dargestellt. Referenzwerte sind leider nur für wenige Problemgrößen vorhanden.

Bis Problemgrößen bis etwa $40 \times 40 \times 40$, solange die Felder also im Level 2-Cache gehalten werden können, zeigt das Blocking-Verfahren auf beiden Plattformen ein ähnliches Verhalten wie die Version mit Temporal Writes und getrennten Halbiterationen.

Die nach den Cachegrößen erreichte Geschwindigkeit kann dann auf beiden Plattformen über alle getesteten Problemgrößen hinweg gehalten, mit zunehmender Zeilenlänge sogar leicht gesteigert werden. Dies ist wohl auf eine höhere Effizienz der Hardware Prefetcher beim Durchlaufen längerer Zeilen zurückzuführen.

Bei einer Problemgröße von $300 \times 300 \times 300$ konnte somit die Geschwindigkeit gegenüber der einfachen Version mit Temporal Writes und Software Prefetching auf dem Pentium 4 um etwa zwei Drittel, beim Athlon64 auf gut das $1,4$-fache gesteigert werden.

Der Einsatz von Software-Prefetching konnte auf beiden Plattformen nur geringe Leistungszunahme erreichen, für den Pentium 4 blieb sie meist unter 3 %, auf dem Athlon64 unter 5 %. Insbesondere beim temporalen Blocken mehrerer Iterationen konnte damit ein nur ein geringer oder kein Vorteil erreicht werden.

Bei der Verschmelzung von zwei oder drei Iterationen konnten die Berechnungen auf dem Pentium 4 mit der etwa $1,3$- und $1,4$-fachen Geschwindigkeit ausgeführt werden, und etwas unter bzw. über dem $1,4$-Fachen auf dem Athlon64.

Der Erfolg von Software Prefetching blieb insbesondere bei diesem Blocking-Verfahren aus. In diesem Fall führte auch die Verfolgung vieler Datenströme eher zu einer Verlangsamung. Für den Pentium 4 brachten Prefetch-Instruktionen allgemein nur geringe Beschleunigung. Für den Athlon64 konnten zumindest in einigen Fällen deutliche Vorteile daraus gezogen werden. Insbesondere beim Pentium 4 ist unklar, ob die Software Prefetches in ungünstiger Weise in den Code eingefügt wurden, ob die Verarbeitung der Befehle ineffizient realisiert wurde, oder ob der Hardware Prefetcher bereits fast optimale Leistungen brachte.

Die deutlich höheren Geschwindigkeiten des 3D-Blocking gegenüber den Referenzversionen hängen von mehreren Faktoren ab.

Diejenigen Berechnungen, die hauptsächlich oder nur von bereits in den Caches vorhanden Daten abhängen, profitieren deutlich von der Vektorisierung und dem handoptimierten Scheduling. Auch der effiziente Code für die Schleifen und zur Adressberechnung spielen hier eine Rolle.

Bei der Referenzversion deutet das Verhalten bei Problemen in Cache-Größe darauf hin, dass die Ablaufsteuerung und die Adressberechnungen nicht so gut vom Compiler optimiert werden konnten. Insbesondere bei kleinen Problemen scheint dies so viel Rechenleistung zu benötigen, dass die schnelleren Cache-Zugriffe auf Operanden ausgeglichen werden.

Bei der vorgestellten Blocking-Technik kann durch die Berechnung ganzer Zeilen am Stück zudem der Hardware-Prefetcher gut unterstützt werden. Ein Hinweis, dass dies auch funktioniert, findet sich zum einen bei der Version mit nur einer temporal geblockten Iteration, bei der die Geschwindigkeit mit zunehmender Problemgröße noch leicht ansteigt. Zum anderen haben Versuche ergeben, dass das Padden von Zeilen mit nur wenigen Füllwerten die Ausführungsgeschwindigkeit deutlich reduziert, was mit den größeren Sprüngen in den Datenströmen zu erklären wäre. Ein Padding der Ebenen hingegen ist für die Geschwindigkeit praktisch unerheblich und kann in entsprechenden Fällen zur Reduzierung von Cache-Konflikten genutzt werden.

Bei großen Problemen greift zudem die Einführung einer weiteren Blocking-Ebene. Allerdings hängt desses Effizient von der möglichen Überblock-Höhe ab. Diese ist durch die Cache-Größe begrenzt und sinkt bei höherer Zeilenlänge oder mehr temporal geblockten Iterationen.

Abbildung 7.6: Optimierter 3D-Red-Black Gauss-Seidel mit einer verschmolzenen Iteration

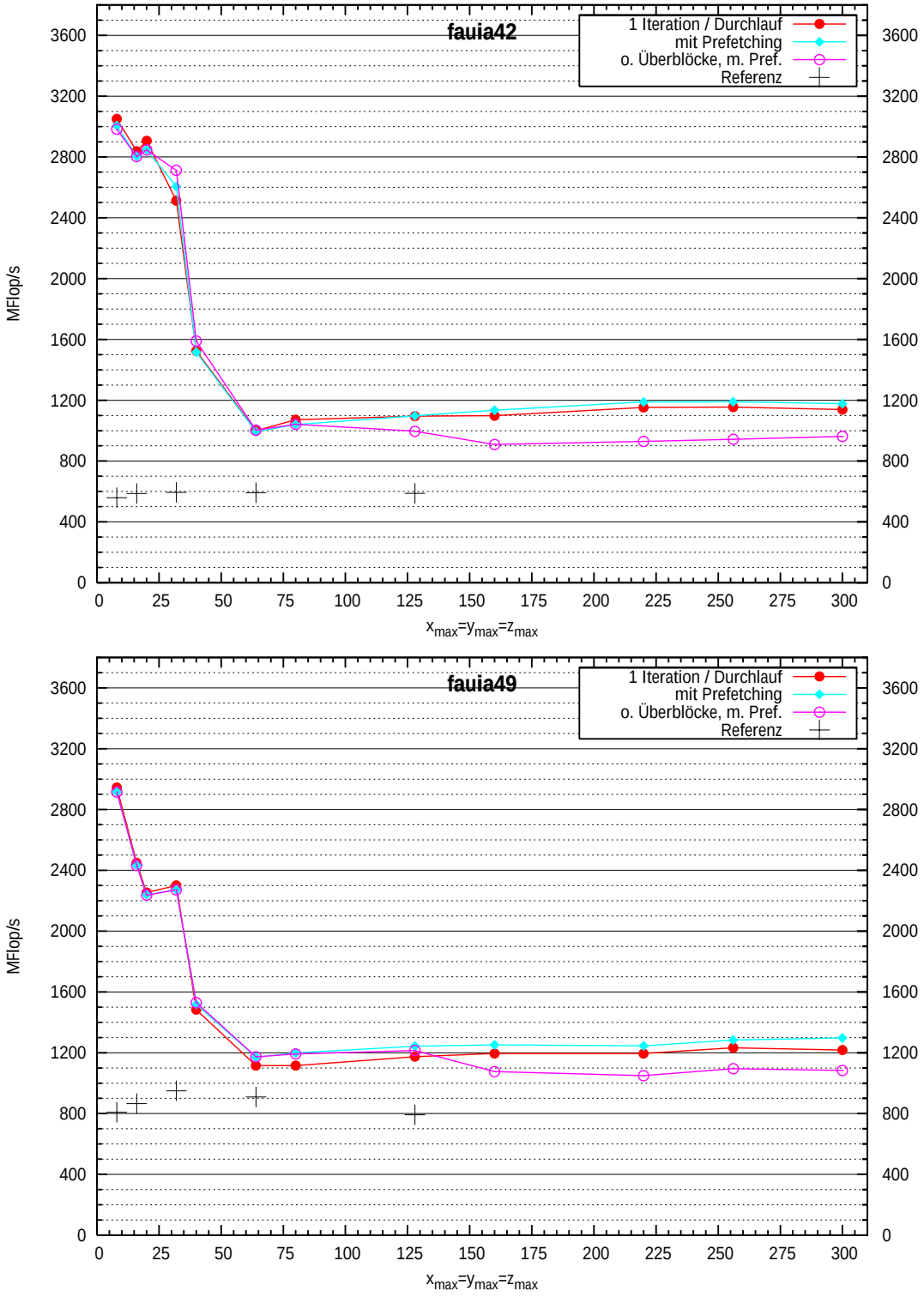


Abbildung 7.7: Optimierter 3D-Red-Black Gauss-Seidel mit zwei verschmolzenen Iterationen

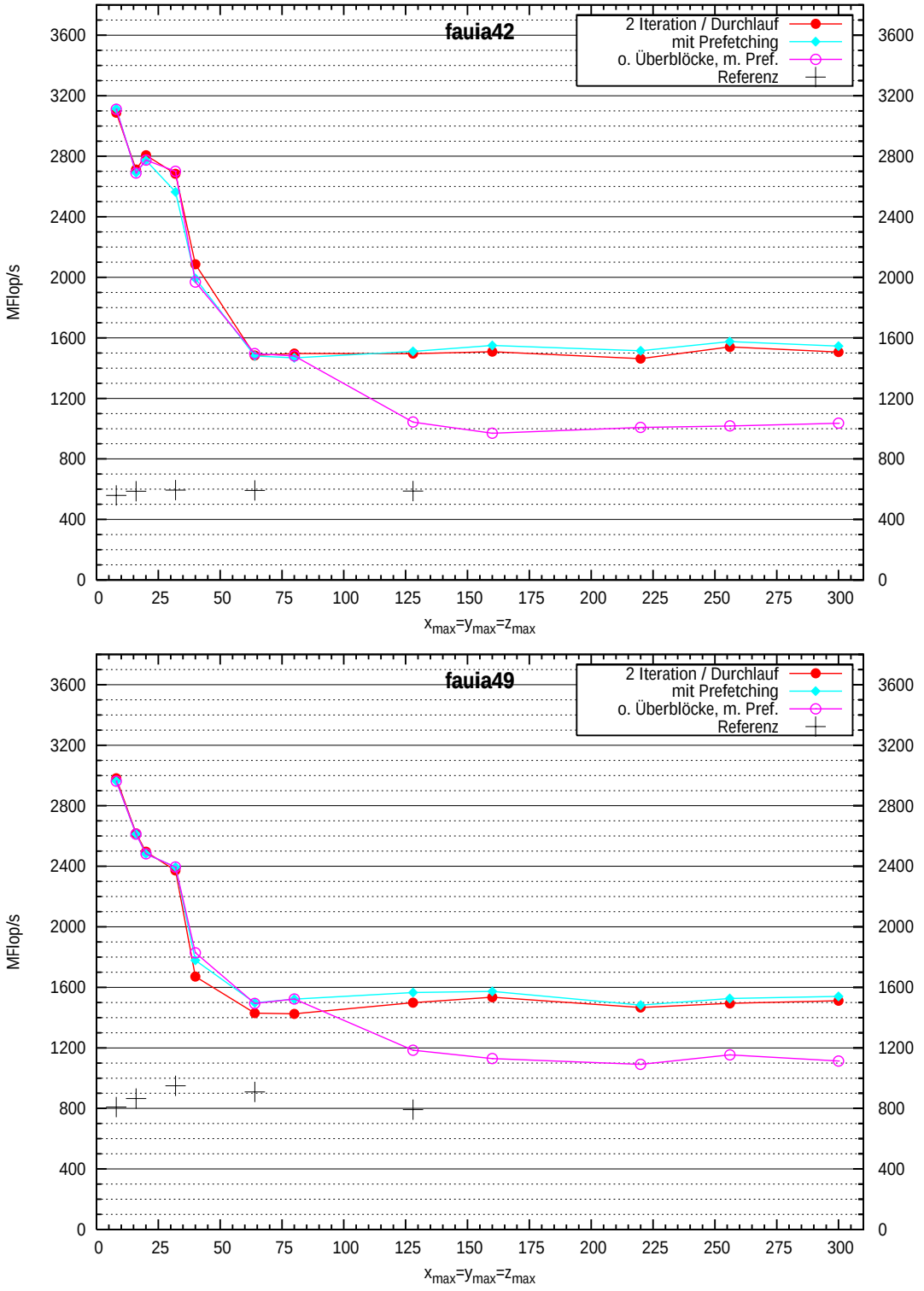
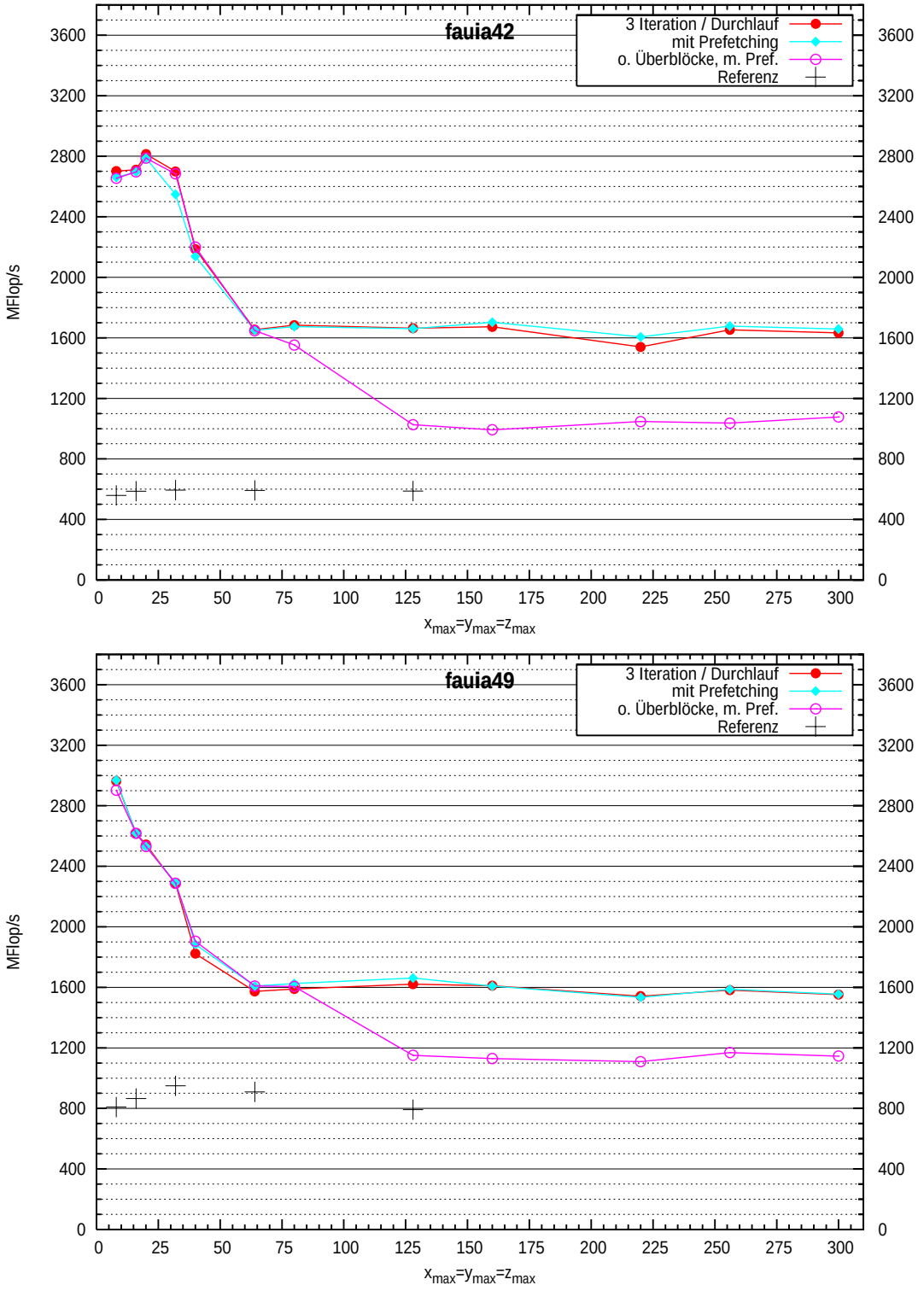


Abbildung 7.8: Optimierter 3D-Red-Black Gauss-Seidel mit drei verschmolzenen Iterationen



7.3 Tabellen

Tabelle 7.1: Maximal erreichte Ausführungsgeschwindigkeiten

	2D		3D	
System	fauia42	fauia49	fauia42	fauia49
Geschwindigkeit [MFlop/s]	3742	3495	3368	2981
Iterationen / Durchgang	1/2	2	1/2	3
Takte pro Vektoraddition	2,57	2,06	2,53	2,15
Prozessor-Input [GB/s]	24,4	22,8	23,0	20,4
Prozessor-Output [GB/s]	4,9	4,6	3,3	2,9
Prozessor-IO [GB/s]	29,2	27,3	26,3	23,3

Tabelle 7.2: Kennwerte für die jeweils schnellste Version des 2D Red-Black Gauss-Seidel bei 2060 × 2060 Unbekannten

fauia42

Anzahl der Iterationen	**1**		**2**		**3**	
MFlop-Rate bei 2060 × 2060	791	1024	1327	1680	1387	2266
Version	rb2	NT	rb7	geblockt	rb6	geblockt
Geschwindigkeitsteigerung	-	129 %	-	127 %	-	163 %
Takte / Vektoraddition	-	9,4	-	5,7	-	4,2
Prozessor-I/O						
Lesen [GB/s]	5,2	6,7	8,6	10,9	9,0	14,8
Schreiben [GB/s]	1,0	1,3	1,7	2,2	1,8	3,0
Gesamt [GB/s]	6,2	8	10,4	13,1	10,8	17,7
minimaler Hauptspeicherdurchsatz						
Lesen [GB/s]	2,1	2,7	1,7	2,2	1,2	2,0
Schreiben [GB/s]	1,0	1,3	0,9	1,1	0,6	1,0
Gesamt [GB/s]	3,1	4,0	2,6	3,3	1,8	3,0

fauia49

Anzahl der Iterationen	**1**		**2**		**3**	
MFlop-Rate bei 2060 × 2060	698	1301	1299	1853	1103	1989
Version	rb2	NT	rb9	geblockt	rb8	geblockt
Geschwindigkeitsteigerung	-	186 %	-	143 %	-	180 % (153 %[1])
Takte / Vektoraddition	-	5,5	-	3,9	-	3,6
Prozessor-I/O						
Lesen [GB/s]	4,5	8,5	8,5	12,1	7,2	12,9
Schreiben [GB/s]	0,9	1,7	1,7	2,4	1,4	2,6
Gesamt [GB/s]	5,5	10,2	10,1	14,5	8,6	15,5
minimaler Hauptspeicherdurchsatz						
Lesen [GB/s]	1,8	3,4	1,7	2,4	1	1,7
Schreiben [GB/s]	0,9	1,7	0,8	1,2	0,5	0,9
Gesamt [GB/s]	2,7	5,1	2,5	3,6	1,4	2,6

[1] gegenüber rb9 bei zwei verschmolzenen Iterationen

Tabelle 7.3: Kennwerte für die jeweils schnellste Version des 3D Red-Black Gauss-Seidel bei 128 × 128 × 128 Unbekannten

fauia42

	Referenz	**1/2 Iter.**	**1 Iter.**	**2 Iter.**	**3 Iter.**
	fused	NT	geblockt	geblockt	geblockt
MFlop-Rate bei 128 × 128 × 128	587	1029	1097	1510	1660
Geschwindigkeitsteigerung	-	175 %	187 %	257 %	283 %
Takte / Vektoraddition	-	8,3	8,8	5,7	5,1
Prozessor-I/O					
Lesen [GB/s]	4,0	7,0	7,5	10,3	11,3
Schreiben [GB/s]	0,6	1,0	1,1	1,5	1,6
Gesamt [GB/s]	4,6	8,0	8,6	11,8	13,0
minimaler Hauptspeicherdurchsatz					
Lesen [GB/s]	1,1	2,0	2,1	1,5	1,1
Schreiben [GB/s]	0,6	1,0	1,1	0,7	0,5
Gesamt [GB/s]	1,7	3,0	3,2	2,2	1,6

fauia49

	Referenz	**1/2 Iter.**	**1 Iter.**	**2 Iter.**	**3 Iter.**
	fused	NT	geblockt	geblockt	geblockt
MFlop-Rate bei 128 × 128 × 128	792	1435	1243	1566	1662
Geschwindigkeitsteigerung	-	181 %	157%	198 %	210 %
Takte / Vektoraddition	-	4,5	5,1	4,1	3,9
Prozessor-I/O					
Lesen [GB/s]	5,4	9,8	8,5	10,7	11,4
Schreiben [GB/s]	0,8	1,4	1,2	1,5	1,6
Gesamt [GB/s]	6,2	11,2	9,7	12,2	13,0
minimaler Hauptspeicherdurchsatz					
Lesen [GB/s]	1,5	2,8	2,4	1,5	1,1
Schreiben [GB/s]	0,8	1,4	1,2	0,8	0,5
Gesamt [GB/s]	2,3	4,2	3,6	2,3	1,6

Abbildungsverzeichnis

Tabellenverzeichnis

Literaturverzeichnis

[AMD01] AMD Corporation: *Eighth Generation Processor Architecture*, Oktober 2001. http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/-Hammer_architecture_WP_2.pdf.

[AMD04] AMD Corporation: *AMD Software Optimization Guide for AMD Athlon 64 and AMD Opteron Processors*, Oktober 2004. http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/25112.PDF.

[AMD05] AMD Corporation: *128 Bit Media Instructions.* In: *AMD64 Architecture Programmer's Manual*, Nummer 4. Februar 2005. http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/26568.pdf.

[B+04] Bloggs, D. et al.: *Microarchitecture of the Intel Pentium 4 Processor on 90nm Technology.* Intel Technology Journal, 8, Februar 2004.

[Cor05] Corporation, AMD: *Application Programming.* In: *AMD64 Architecture Programmer's Manual*, Nummer 1. März 2005. http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/24592.pdf.

[GDN97] Griebel, M., Th. Dornseifer und T. Neunhoeffer: *Numerical Simulations in Fluid Dynmics - A Practical Intruduction.* Society for Industrial and Applied Mathematics, 1997.

[Hac93] Hackebusch, W.: *Iterative Löser großer schwachbesetzter Gleichungssysteme.* Teubner Stuttgart, 2 Auflage, 1993.

[Int04a] Intel Corporation: *Basic Architecture.* In: *Intel Architecture Software Developer's Manual*, Nummer 1 in *3.* 2004. ftp://download.intel.com/design/Pentium4/manuals/25366514.pdf.

[Int04b] Intel Corporation: *Instruction Reference A-M.* In: *Intel Architecture Software Developer's Manual*, Nummer 2a in *3.* 2004. ftp://download.intel.com/design/Pentium4/manuals/25366614.pdf.

[Int04c] Intel Corporation: *Instruction Reference N-Z.* In: *Intel Architecture Software Developer's Manual*, Nummer 2b in *3.* 2004. ftp://download.intel.com/design/Pentium4/manuals/25366714.pdf.

[Int05] Intel Corporation: *Intel Architecture Optimization - Reference Manual*, Juni 2005. ftp://download.intel.com/design/Pentium4/manuals/24896612.pdf.

[Thü02] Thürey, N.: *Cache Optimizations for Multigrid in 3D.* Lehrstuhl für Informatik 10 (Systemsimulation), Institut für Informatik, University of Erlangen-Nuremberg, Germany, Juni 2002. Studienarbeit.

[Wei01] Weiss, C.: *Data Locality Optimizations for Multigrid Methods on Structured Grids.* Doktorarbeit, Lehrstuhl für Rechnertechnik und Rechnerorganisation, Institut für Informatik, Technische Universität München, Munich, Germany, Dezember 2001.