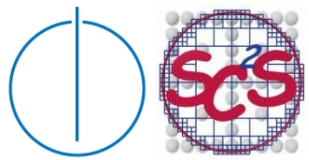
Masterpraktikum Scientific Computing

High-Performance Computing

Thomas Auckenthaler
Wolfgang Eckhardt
Prof. Dr. Michael Bader

Technische Universität München, Germany



Outline

Ebenen der Parallelität

Aufbau eines Einprozessorsystems

- Von-Neumann-Prinzip
- Beschleunigung des Ablaufs: Pipelining
- Schleifen Optimierung
- Pipeline-Konflikte
- Abhängigkeitsanalyse

Aufbau eines Vektorrechner

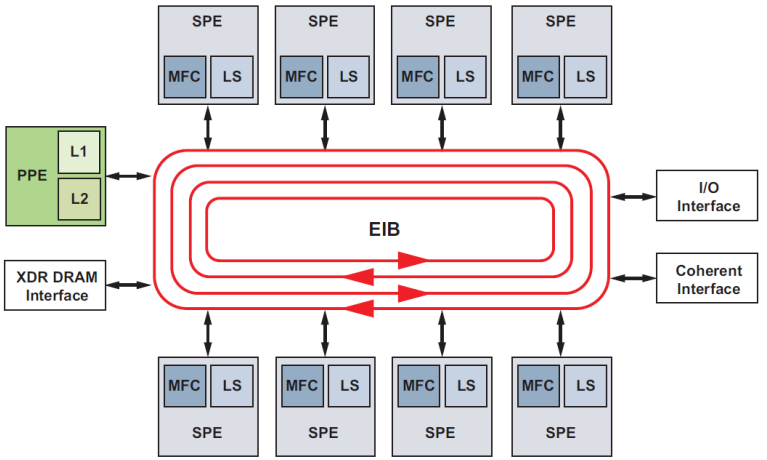
- Aufbau eines Vektorrechners?
- Beispiel: SSE-Erweiterung des Intel Pentium 4
- Abhängigkeitsanalyse und Vektorisierung
- Vektorisierung rekursiver Probleme

Leistungsanalyse - Maßzahlen eines Vektorrechners

Ebenen der Parallelität

Parallele Programme lassen sich nach der Granularität ihre Parallelität klassifizieren und Rechnertypen gegenüberstellen, welche durch ihre Hardware die entsprechende Ebene unterstützen.

- Suboperationsebene
  - Vektorrechner (z. B. NEC-SX, Cray-SV)
  - Feldrechner (z. B. MasPar-Rechner)
  - Cell-Processor
  - SSE$[1-4]$
  - ...



- Anweisungsebene
  - Superskalarrechner (z. B. Workstation/PC)
  - VLIW-Rechner (very long instruktion word) (z. B. Itanium)
  - Pipeline-Architekturen
  - ...

ILP – **I**nstruction **L**evel **P**arallelism

- Blockebene (Threads)
  - Shared-Memory-Rechner (z. B. SUN-UltraSparc, SGI-Altix)
  - ...
- Prozessebene
  - Distributed-Memory-Rechner (Cray-T3E, IBM-SP)
  - ...
- Programmebene
  - Workstation-Cluster
  - Grid-Computer
  - ...

Klassifikation nach Flynn

- Betrachte: Befehlsstrom und Datenstrom
- Ein Rechner bearbeitet zu einem Zeitpunkt einen oder mehrere Befehle
- Ein Rechner bearbeitet zu einem Zeitpunkt einen oder mehrere Datenwert

SISD Einprozessorsystem	**SIMD** Vektorrechner Feldrechner
MISD	**MIMD** Multiprozessorsysteme

Outline

Von-Neumann-Prinzip

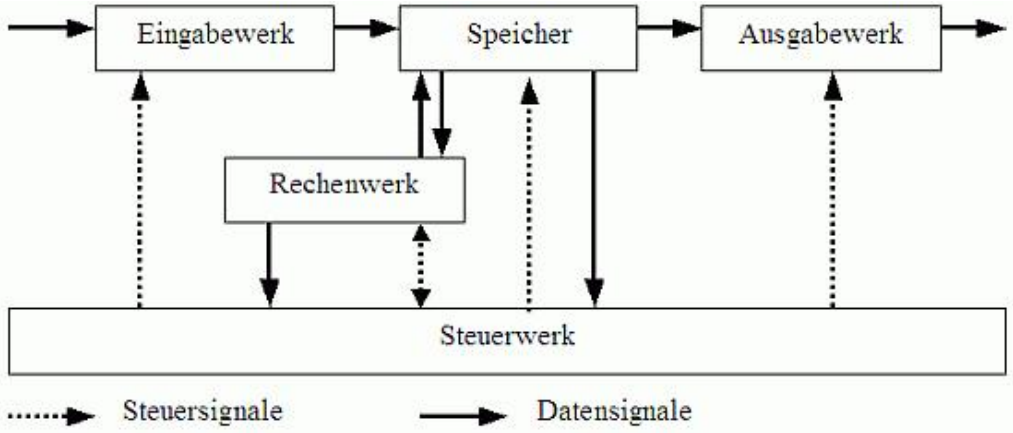


Figure: Aufbau des Von-Neumann-Rechners

IF: instruction fetch
DEC: instruction decode
register fetch
EXEC: execution
effective address
branch output
MEM: memory access
branch comptetion
WB: write back

Figure: Phasen des Befehlszyklus

Pipelining

- Instruktion Pipelining: Überlappende Ausführung von unabhängigen Befehlen
- Erhöhung des Instruktionsdurchsatzes
- Bei einer Pipeline mit k Stufen wird nach einer Einschwingzeit, ab dem Systemzyklus k zu jedem Takt ein Ergebnis geliefert.

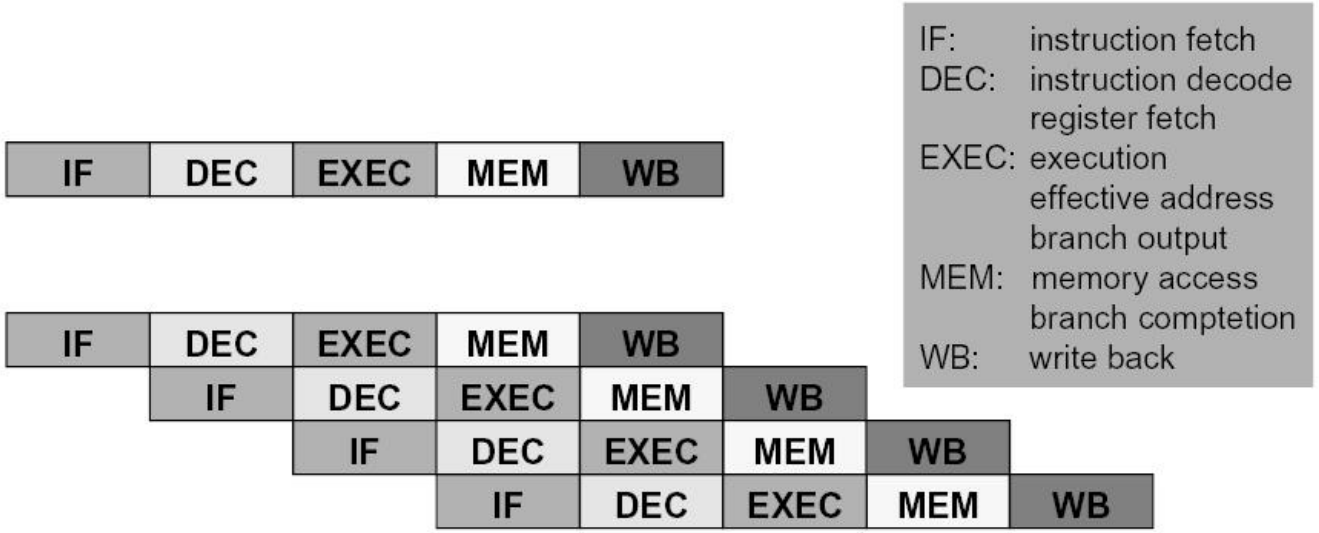


Figure: Befehlsausführung ohne und mit Pipelining

Arithmetisches Pipelining

Beispiel: Fließkomma-Addition in vier Schritten

Arithmetisches Pipelining

Beispiel: Fließkomma-Addition in vier Schritten

- Ein Paar Gleitpunktzahlen aus dem Vektorregister laden,
- die Exponenten vergleichen und eine der Mantissen verschieben,
- die ausgerichtetem Mantissen addieren und
- das Ergebnis normalisieren und in ein Register zurückschreiben.

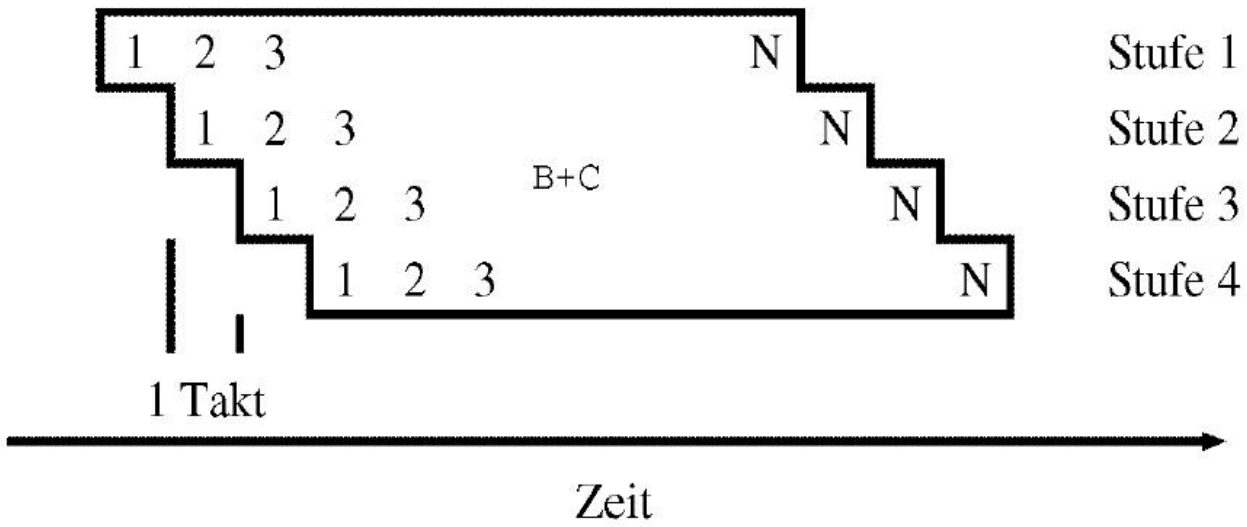


Figure: Zeitl. Verlauf einer Vektoroperation in einer Pipeline

Code-Optimierungen für Pipelining

- loop unrolling

```
for(i=0;i<3;i++) {
  for(k=0;k<m;k++) {
    c[i,k]=a[i,k]*b[i,k];
  }
}
```

```
for(k=0;k<m;k++) {
    c[0,k]=a[0,k]*b[0,k];
    c[1,k]=a[1,k]*b[1,k];
    c[2,k]=a[2,k]*b[2,k];
}
```

- loop fusion

```
for(i=0;i<n;i++) {
    d[i]=d[i-1]*e[i];
}

for(i=0;i<n;i++) {
    f[i]=f[i]+c[i-1];
}
```

```
for(i=0;i<n;i++) {
    d[i]=d[i-1]*e[i];
    f[i]=f[i]+c[i-1];
}
```

- Weitere Optimierungen siehe **Intel C++ Compiler User Guide**

Pipeline - Konflikte

- **Strukturkonflikte:** Parallele Abarbeitung durch Limiterung der Hardware eingeschränkt (z.B. gleichzeitiger Speicherzugriff in “Instruktion Fetch”– und “Memory Access” – Phase)
- **Steuerkonflikte:** Verursacht durch Sprünge im Code
  $\Rightarrow$ Branchprediction, spekulative Ausführung
- **Datenkonflikte:** Datenkonflikte werden durch Datenabhängigkeiten verursacht.

Der Compiler bzw. Hardware zur Laufzeit müssen diese Konflikte erkennen und beheben können.

Abhängigkeitsanalyse

```
S1: A = C - A
S2: A = B - C
S3: B = A + C
```

- echte Abhängigkeit (true dependence) (z.B. **S3** bzgl. **S2** für **A**)
  - **S2** zeigt eine true dependence bzgl. **S1**, **S2** δ **S2**, wenn **OUT(S1)** $\cap$ **IN(S2)** $\neq \emptyset$;
    $\Rightarrow$ **S2** benutzt das Ergebniss von **S1**
- Gegenabhängigkeit (anti dependence) (z.B. **S3** bzgl. **S2** für **B**)
  - **S2** zeigt eine anti dependence bzgl. **S1**, **S2** δ^{-1} **S2**, wenn **IN(S1)** $\cap$ **OUT(S2)** $\neq \emptyset$;
    $\Rightarrow$ werden **S2** und **S1** vertauscht, so wird **S1** fälschlicher Weise das Ergebniss von **S2** benutzen

Abhängigkeitsanalyse

```
S1: A = C - A
S2: A = B - C
S3: B = A + C
```

- Ausgabeabhängigkeit (output dep.) (z.B. **S2** bzgl. **S1** für **A**)
  - **S2** und **S1** zeigen eine output dependence, **S2** δ^O **S2**, wenn **OUT(S1)** $\cap$ **OUT(S2)** $\neq \emptyset$;
    $\Rightarrow$ werden **S2** und **S1** vertauscht, so enthält die Zuweisungsvariable bei Wiederverwendung den falschen Wert

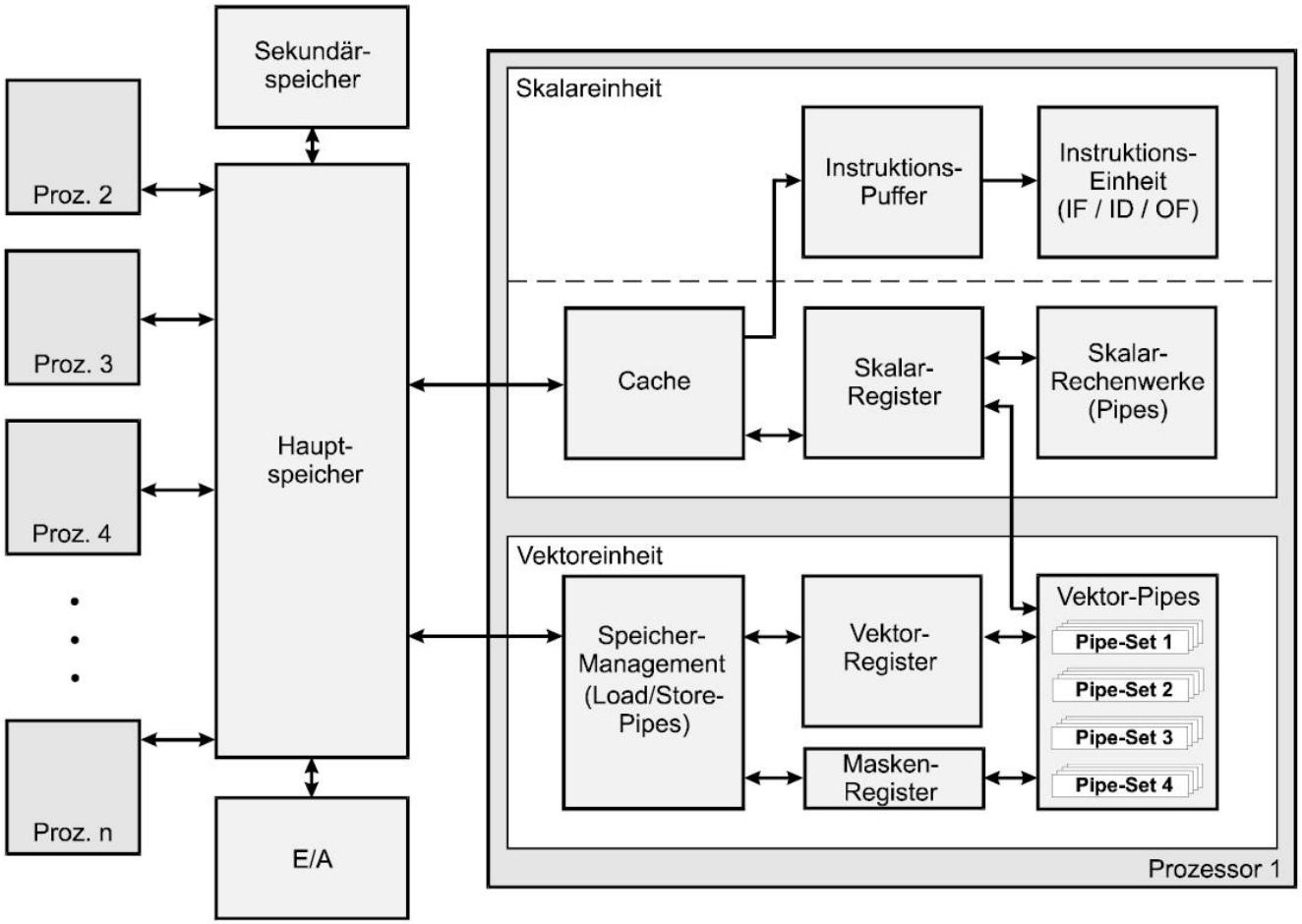
Die Abhängigkeitsanalyse gibt Aufschluss darüber, ob eine Pipeline-Arbarbeitung bzw. Vektorisierung möglich ist.

Outline

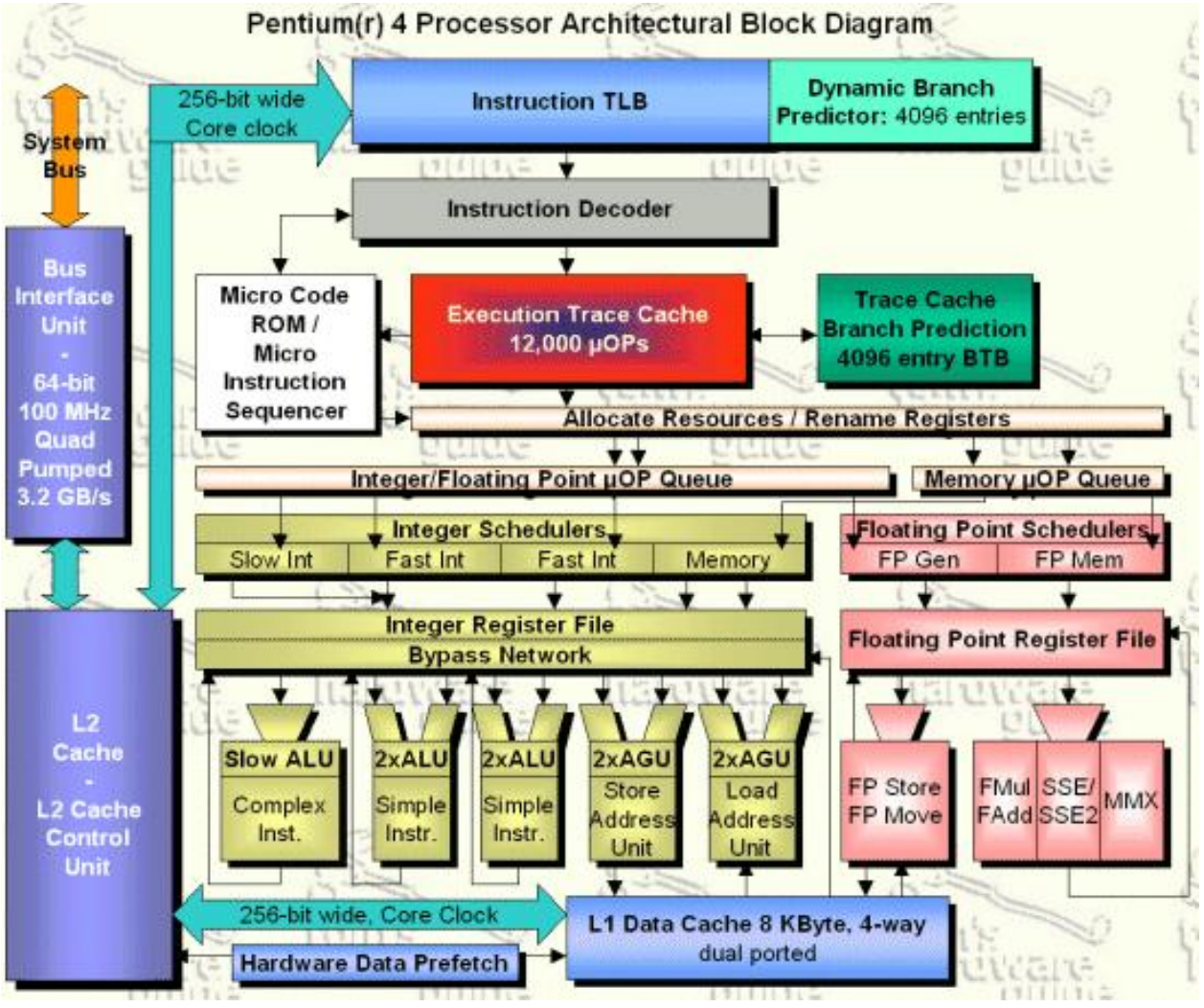
Aufbau eines Vektorrechner - Was ist ein Vektorrechner?

- Einen Rechner mit **pipelineartig** aufgebautem/n Rechenwerk/en, der **Arrays** von Gleitpunktzahlen verarbeitet.
- **Vektor** = Array (Feld) von Gleitpunktzahlen.
- **Vektorbefehle** werden auf ganze Vektoren angewendet, die in **Vektorregistern** gespeichert sind.
- Ein Vektorrechner enthält neben der Vektoreinheit auch noch eine oder mehrere **Skalareinheiten**. Dort werden die skalaren Befehle parallel zu den Vektorbefehlen ausgeführt.

Prinzipskizze eines Vektorprozessors



Beispiel: SSE-Erweiterung des Intel Pentium 4



SSE-Register

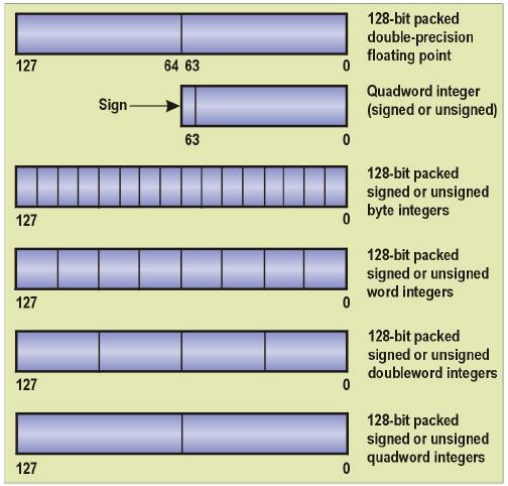


Figure: Streaming SIMD Extensions data types

- Streaming SIMD Extension (SSE)
  - 8 Register (xmm0 bis xmm7), je 128 bit breit
  - halten 2 double-precision floating point Werte (je 64 bit)
  - halten 4 single-prescision floating point Werte (je 32 bit)
  - halten 2 64-bit integer Werte bis zu 16 8-bit Werte

Automatische Vektorisierung

```
#pragma vector always
#pragma ivdep
for(i=1;i<n;i++) {
   c[i]=s*c[i];
}
```

Intel Compiler Pragmas für IA-32 Vektorisierung:

- #pragma vector always $\Rightarrow$ Vektorisierung unabhängig von der Effizienzeinschätzung des Compilers.
- #pragma ivdep $\Rightarrow$ Vektorisierung unabhängig von der Abhängigkeitsanalyse den Compilers

Informationen zu SSE2, Vektorisierung

- Intel Architecture Optimization Reference Manual
- Intel C++ Compiler User's Guide

Abhängigkeitsanalyse und Vektorisierung

- Schleife I

```
for(i=0;i<n;i++) {
        a[i]=b[i]+c[i]
}
```

  Keine Abhängigkeiten zwischen den einzelnen Iterationen $\Rightarrow$ kann optimal in einer Pipeline ausgeführt werden.

- Schleife II

```
for(i=1;i<n-1;i++) {
        c[i]=c[i]*c[i-1];
}
```

  REKURSION! c_{i-1} ist beim Zugriff noch in der Pipeline $\Rightarrow$ keine

Vektorisierung rekursiver Probleme

- Summation ohne Hardwareunterstützung

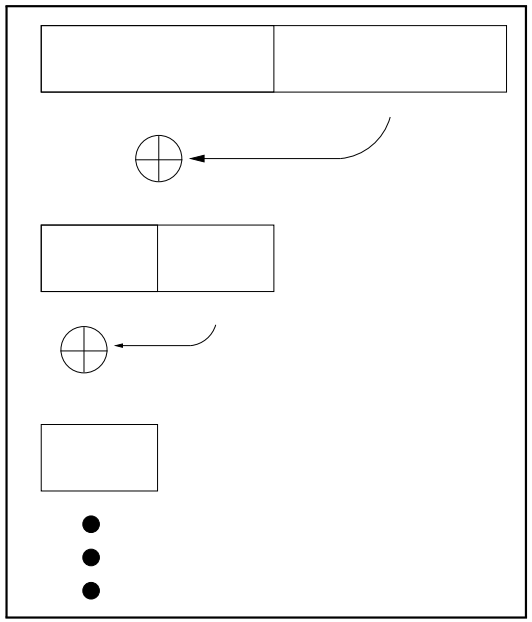


Figure: Cascade-Method: Vektorisierung der Summation

```
s=a[0];
for(i=1;i<n;i++) {
    s=s+a[i];
}
```

- divide and conquer Strategie: Addiere die zweite Hälfte des Vektors zur Ersten.
- Setze dieses Prinzip jeweils mit der ersten Hälfte fort,
- bis das erste Element die Summe enthält.

Vektorisierung rekursiver Probleme

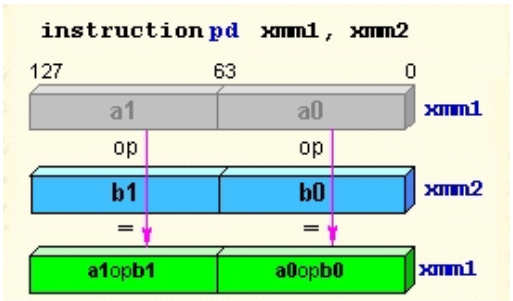
- Lineare Rekursion erster Ordnung

$$x_1 = a_1; \quad x_i = a_i + b_i \cdot x_{i-1}, \quad i = 2 \ldots n \tag{1}$$

- Recursive Doubling (vorgestellt von H.S. Stone 1975) z. B. beschrieben in [1] $\Rightarrow$ Nicht effizient!
- Cyclic Reduction (vorgestellt von R.W. Hockney) und z. B. in [2] beschrieben
- partition method vorgestellt in [3] z. B. in [4] beschrieben
  $\Rightarrow$ Meist effizienteste Methode!

SSE2 Intrinsic Functions

- `__m128i; __m128d; //Register-Variablen`
- `__m128d _mm_`*instr*`_pd(__m128d a, __m128d b)`
- suffixes:
  - ps,pd → packed single- und double-precision floating point Funktionen
  - ss,sd → scalar single- und double-precision floating point Funktionen
  - ...
- alle Funktionen findet man in der “Intel C++ Intrinsics Reference” im **Intel C++ Compiler User’s Guide**



Beispiel Code mit SSE2 Intrinsic Functions

```
#include <emmintrin.h>

int sse_add(int length, double* a,
            double* b, double* c) {
    int i;
    _m128d t0,t1;

    for (i = 0; i < length; i +=2) {
        // lädt 2 64-bit-double-Werte
        t0 = _mm_load_pd(a[i]);
        t1 = _mm_load_pd(b[i]);
        // addiert 2 64-bit-double-Werte
        t0 = _mm_add_pd(t0,t1);
        // speichert 2 64-bit-double-Werte
        _mm_store_pd(c[i],t0);
    }
}
```

Outline

Leistungsanalyse - Maßzahlen eines Vektorrechners

- Der Durchsatz D einer Pipeline mit k Stufen und der Taktzyklusdauer τ ergibt sich, bei der Verarbeitung von n Elementen über den Zeitraum $t = T \cdot \tau$ ($T = \#Takte$), zu

$$D = \frac{n}{T \cdot \tau}. \tag{2}$$

  Mit

$$T = k + (n - 1) \tag{3}$$

  ergibt sich

$$D = \frac{n}{(k + (n-1)) \cdot \tau} = \frac{1}{(\frac{k}{n} + (1 - \frac{1}{n})) \cdot \tau} \underset{n \to \infty}{\longrightarrow} \frac{1}{\tau}. \tag{4}$$

  Damit ist der maximale Durchsatz einer Pipeline $D_{max} = \frac{1}{\tau}$.
- Die Hälfte des maximalen Durchsatzes wird bei der Vektorlänge $n := n_{1/2}$ erreicht. $n_{1/2}$ wird als Half-performance length bezeichnet.

Mit

$$\frac{D_{max}}{2} = \frac{1}{2 \cdot \tau} = \frac{n_{1/2}}{(k + (n_{1/2} - 1)) \cdot \tau} \tag{5}$$

folgt

$$n_{1/2} = k - 1 \, . \tag{6}$$

Neben der Einschwingzeit muss auch die Startzeit s betrachtet werden. In der Startzeit werden Vektorbefehle decodiert, Adressen für das erste und letzte Feld der Vektoren berechnet, die Operanden geladen usw. . Damit wird (3) zu

$$T = k + (n - 1) + s \, . \tag{7}$$

In der Praxis wird die Startzeit und die Einschwingzeit unter der Half-performance length zusammengefasst. Für die Ausführungszeit $t(n)$ ergibt sich

$$t(n) = T \cdot \tau = (k + (n - 1) + s) \cdot \tau = (n + n_{1/2})\tau \tag{8}$$

Floatingpoint-Leistung

Die Floatingpoint-Leistung (Performance) R eines (Vektor-)""Rechners ist

$$R = \frac{\text{Zahl der Floatingpoint-Operationen}}{\text{Ausführungszeit}} \,. \tag{9}$$

Wird die effektive Zykluszeit berücksichtigt, ergibt sich mit (8)

$$R(n) = \frac{n}{(n + n_{1/2})\tau_{eff}} = \frac{1}{\tau_{eff}} \cdot \frac{1}{1 + \frac{n_{1/2}}{n}} \,. \tag{10}$$

Im Grenzwert $n \to \infty$ gilt

$$R_\infty = \frac{1}{\tau_{eff}} \,. \tag{11}$$

Der Grenzwert R_∞ wird als Real Peak Performance bezeichnet. R_∞ kann in der Praxis nicht gemessen werden daher ist es sinnvoll die Maximal Performance R_{max} als maximal gemessene Leistung einzuführen. Dabei muss $n \gg n_{1/2}$ gelten.
Damit ist τ_{eff} mit

$$\tau_{eff} = \frac{1}{R_\infty} \approx \frac{1}{R_{max}} \tag{12}$$

gegeben. Die Leistung wird normalerweise in FLOPS und die Zykluszeit in nsec angegeben.

Weiterführende Links:

www http://www.netlib.org/benchweb/hpc.html

www http://www.scl.ameslab.gov/Publications/Gus/TwelveWays.html

www http://docs.cray.com
Dokument S-2312-36: “Cray SV1 Application Optimization Guide“.

- Ein ausführliches Kapitel über Schleifen-Optimierung findet sich z.B. in [5].

SCHÖNAUER, W.:
Scientific Computing on Vector Computers.
Amsterdam : North-Holland, 1987

HOCKNEY, R.W. ; JESSHOPE, C.R.:
Parrallel Computers 2.
Adam Hilger, 1988. –
Ein Klassiker!

VAN DER VORST, H.A. ; DEKKER, K.:
Vectorization of linear recurrence relations.
In: *SIAM J. Sci. Stat. Comput.*
10 (1989), Nr. 1, S. 27–35

SCHÖNAUER, W.:
Scientific Supercomputing.
Karlsruhe : Schönauer, 2000. –
Anwendungs bezogenes Vorlesungsskript, das lange Zeit handschriftlich (?!) im Web zufinden war. Sehr empfehlenswert! . –
ISBN 3–00–005484–7

DOWD, Kevin ; SEVERANCE, Charles:
High Performance Computing.
O'Reilly, 1998. –
Ein Buch für Praktiker! Optimierter Speicherzugriff, Schleifen-Optimierung und Benchmarking sind nur einige Themen aus dem breiten Vorrat, den dieses Buch bereit hällt. –
ISBN 3–931216–76–4