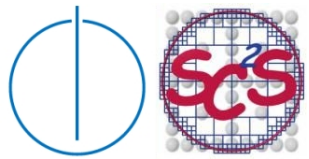
Masterpraktikum Scientific Computing

High-Performance Computing

Thomas Auckenthaler
Wolfgang Eckhardt

Technische Universität München, Germany



Outline

Ebenen der Parallelität

Aufbau eines Einprozessorsystems

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

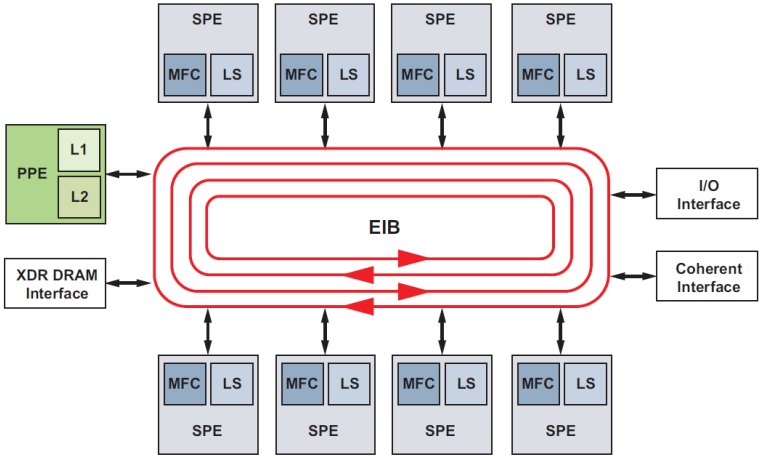
Aufbau eines Vektorrechner

- Aufbau eines Vektorrechners?
- Beispiel: SSE-Erweiterung des Intel Core 2
- Abhängigkeitsanalyse und Vektorisierung
- Vektorisierung rekursiver Probleme

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)
  - Cell-Processor
  - SSE
  - ...



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

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

- Threadebene
  - 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 GPGPU
MISD	**MIMD** Multiprozessorsysteme

Outline

Von-Neumann-Prinzip

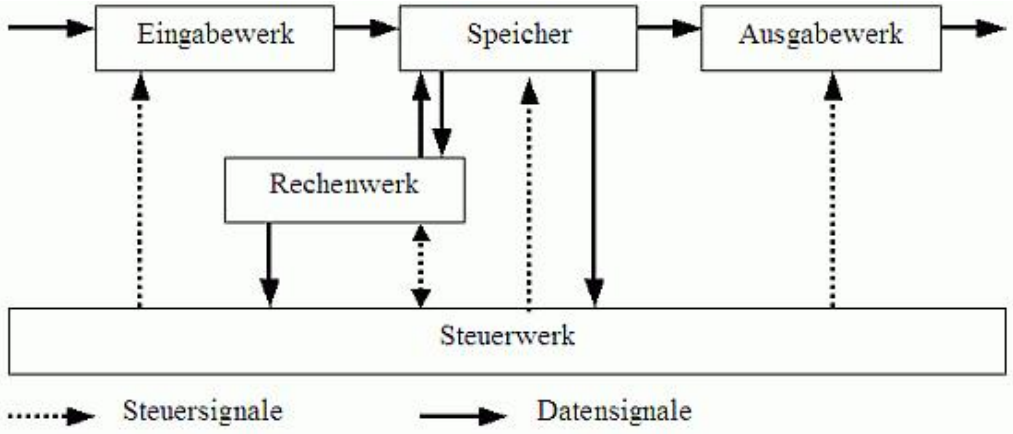


Figure: Aufbau des Von-Neumann-Rechners

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.

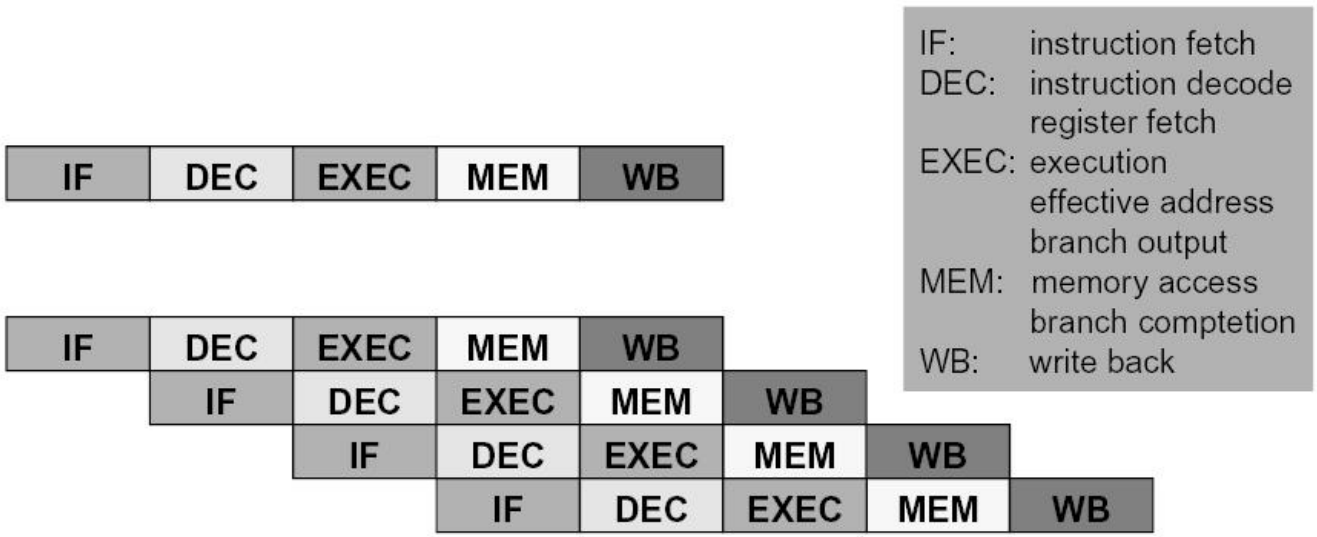


Figure: Befehlsausführung ohne und mit Pipelining

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.

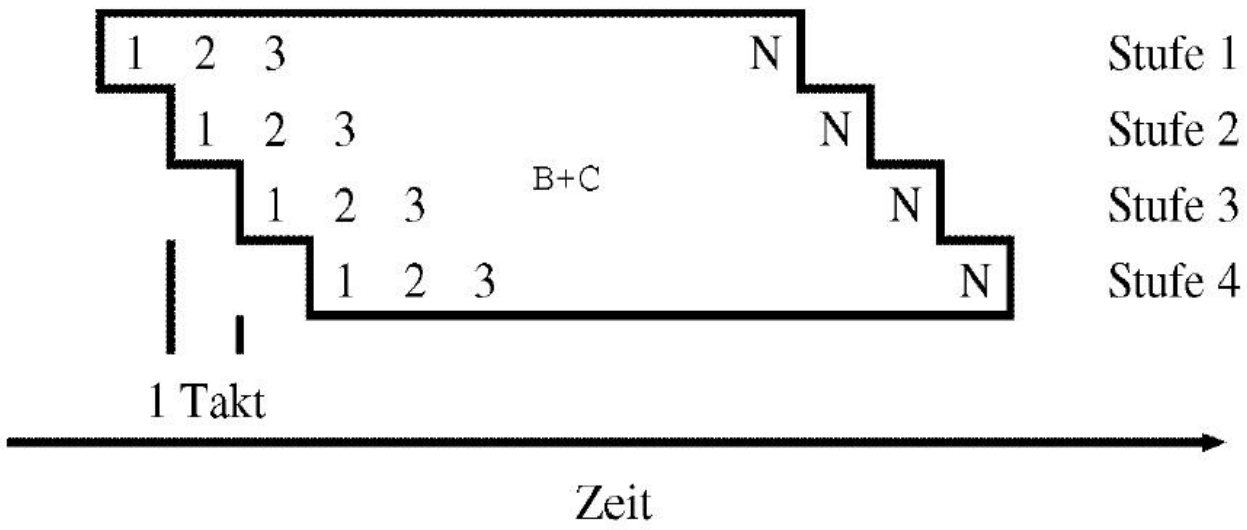


Figure: Zeitl. Verlauf einer Vektoroperation in einer Pipeline

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
```

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

Abhängigkeitsanalyse

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

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

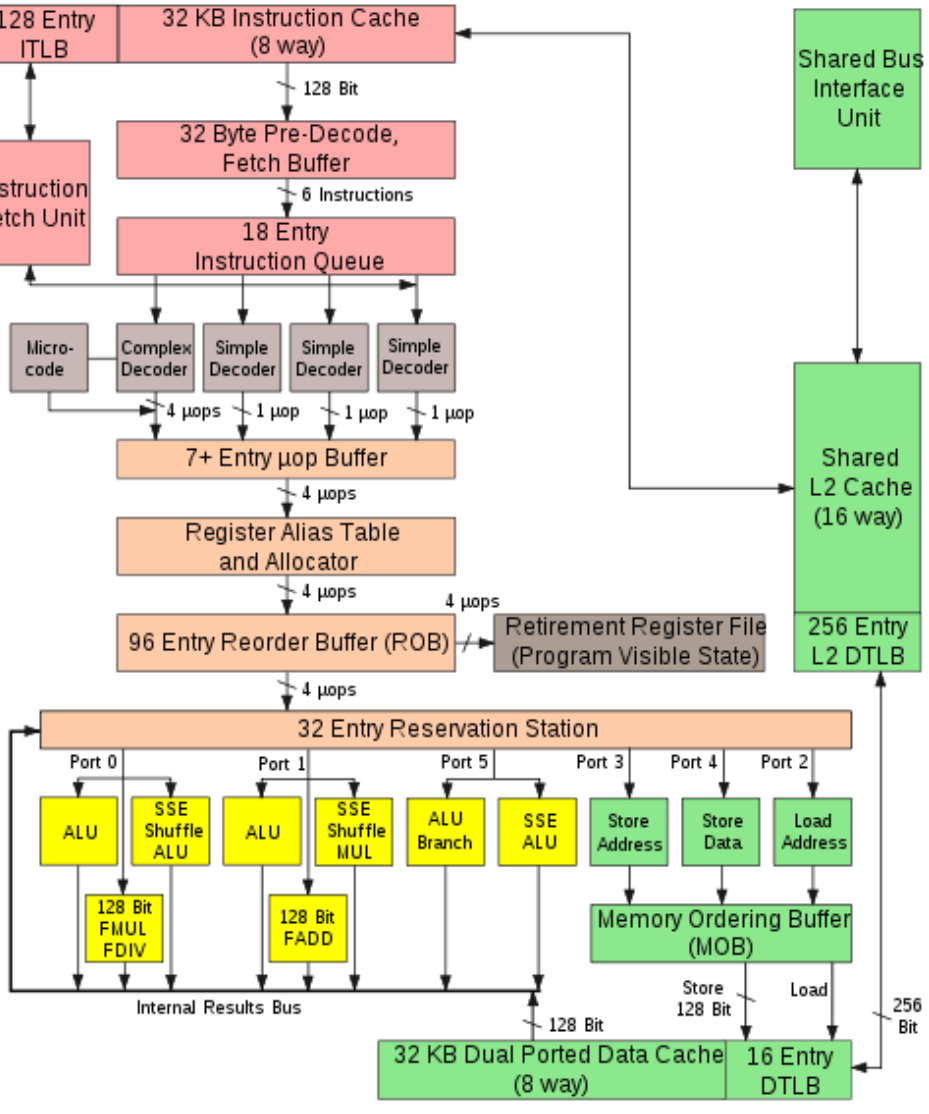
Die Abhängigkeitsanalyse gibt Aufschluss darüber, ob eine Pipeline-Arbarbeitung bzw. Vektorisierung möglich ist.

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.

Beispiel: SSE-Erweiterung des Intel Core 2



Intel Core 2 Architecture

SSE-Register

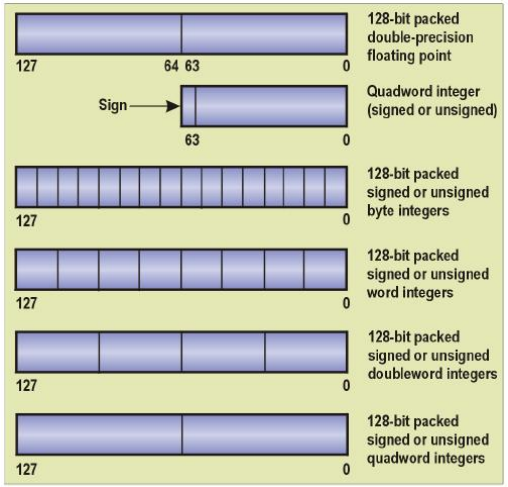


Figure: Streaming SIMD Extensions data types

- 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 des 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

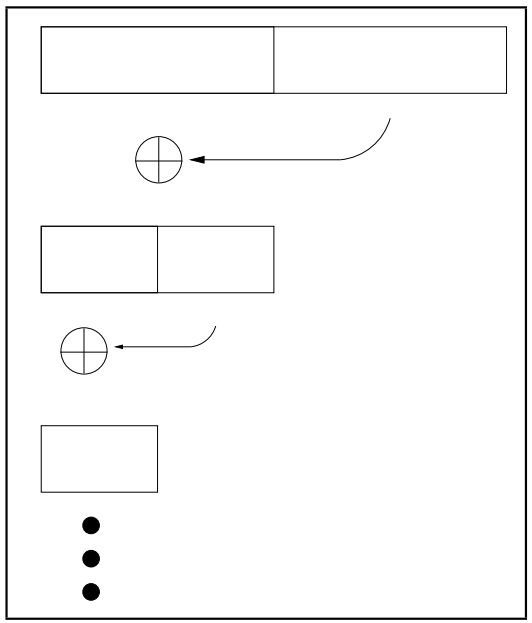


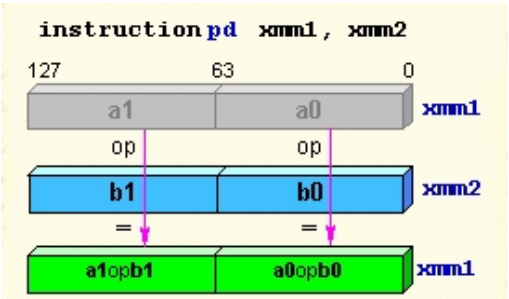
Figure: Cascade-Method: Vektorisierung der Summation

```
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.

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);
    }
}
```