Vektorisierung
für den Vektorrechner
Siemens/Fujitsu VPP300/700

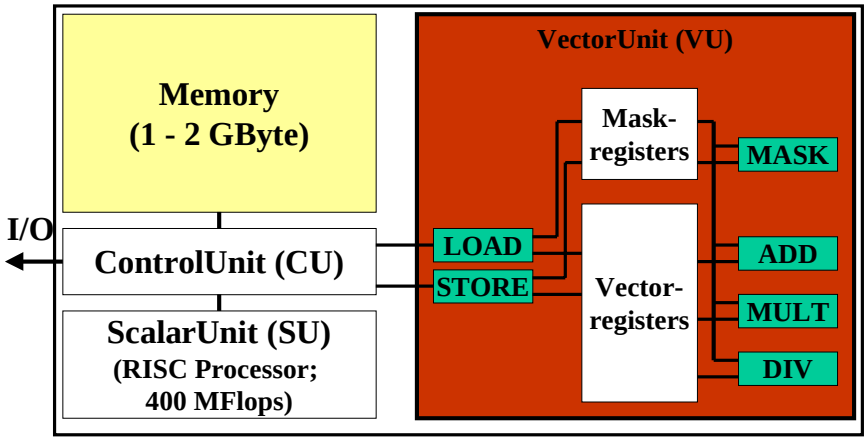
Einsatzgebiete von Hochleistungsrechnern

- Wettervorhersage (Deutscher Wetterdienst)
- Computional Fluid Dynamics (CFD)
- Crash Tests
- Hochenergiephysik
- Reaktorsicherheit
- Atombombentests ????

Hochleistungsrechner-architekturen

- Parallelrechner: Mehrere Prozessoren arbeiten gleichzeitig an einer Problemstellung.
- Vektorprozessoren: Für sogenannte Vektoroperationen ist die *Flops* Leistung etwa 5-15 mal größer als bei RISC Prozessoren.
- VektorParallelrechner: NEC SX4, CRAY YMP/ T90, Fujitsu VPP300/700 .....

Vektorprozessor: Schematischer Überblick



Vektorprozessor: Bestandteile der VU

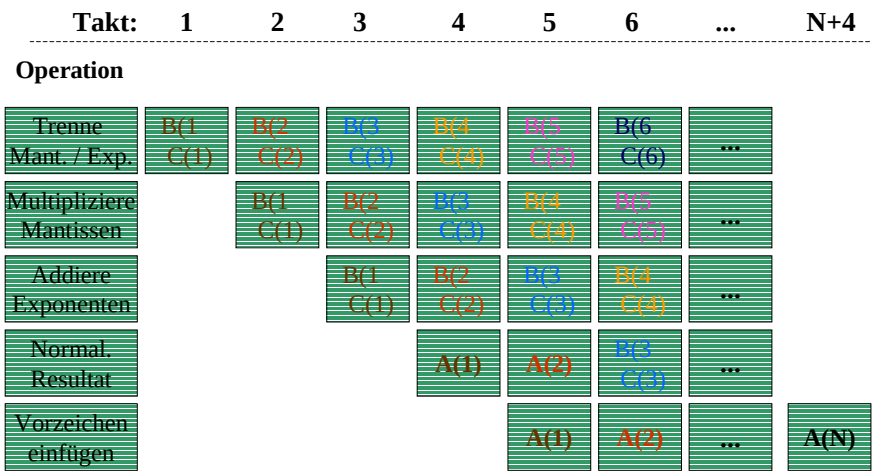
- Pipes: Recheneinheiten die in verschiedene Segmente unterteilt sind. Jedes Segment führt eine genau definierte Teiloperation pro Takt aus. Im allgemeinen gibt es Pipes für:

  Multiplikation (MULT), Addition (ADD), Division (DIV), Lade- (LOAD) und Speicher- (STORE) Zugriffe, sowie für Maskenoperationen (MASK).

- Vektorregister:Speicher für Pipes mit schnellen Zugriffszeiten. Die Länge der Register bezeichnet man als *Vektorlänge*.

Vektorprozessor: Arbeitsweise einer *MULT*Pipe.

A(i)=B(i)*C(i) ; i=1,...,N



Vektorprozessor: Eigenschaften von Pipes (1)

Pipelänge p: Anzahl der Segmente (Im Beispiel p=5)

StartUp Zeit: p Takte (Erstes Resultat nach p Takten, danach 1 Ergebnis pro Takt)

- SpeedUp gegenüber sequentieller Ausführung der Multiplikation

  Rechenzeiten: Sequentieller Rechner: **N** * p Takte

  Pipelining: **N** + p - 1 Takte

  ⇒ SpeedUp= (**N** * p)/(**N** + p - 1) ~ p (für **N** >> p)

Vektorprozessor: Eigenschaften von Pipes (2)

- Eine Pipe kann keine Rekursionen verarbeiten !

  Beispiel: A(i+1) = A(i) * B(i+1) kann nicht im Pipelining berechnet werden, da A(i) und A(i+1) gleichzeitig in der Pipe bearbeitet werden.

- Mehrere Pipes können gleichzeitig arbeiten!
- Verkettung von Pipes (*Chaining*): Resultat einer Pipe wird an andere Pipe weitergereicht.

  Beispiel *MULTADD:* A(i)=A(i)+ s * B(i) ; i=1,...,N

MULT: s*B(n) → **ADD: {s*B(m)} +A(m)** → **STORE: A(l)**

⇒ (2 Flop + 1 Ergebnis) / Takt

Fujitsu VPP300: Konfiguration der Vektoreinheit

- Anzahl der Pipes: 7
  (MULT / ADD /DIV / LOAD / STORE / MASK(2))
- Zwei der drei arithmetischen Pipes, sowie alle anderen Pipes können gleichzeitig arbeiten.
- Jede Pipe ist 8-fach ausgelegt => Max. 8 Ergebnisse pro Pipe und Takt.
- Taktfrequenz: 143 MHz <=> Taktzeit: 7 ns

→ Peak Performance: 2 * 8 * 143 MFlops = 2.2 GFlops

Fujitsu VPP300: Zugriff auf den Hauptspeicher

- Kapazität: 2 GBytes (ca. 1.6 - 1.8 GBytes verfügbar)
- Nur 1 LOAD Pipe (CRAY T90 besitzt 2 LOAD Pipes)
- Zugriffszeit 60 ns.
- Bank busy time: 20 Takte.
- 512 Speicherbänke (interleaving)

→ Speicherbandbreite zum Hauptspeicher: 18.2 GBytes/s

(Pro Takt können 8 Worte a 8 Bytes gelesen bzw. geschrieben werden => 2 * 8 * 8 Bytes / 7 ns. Die Bandbreite wird nur bei kontinuierlichen Speicherzugriffen (stride=1) erreicht)

Fujitsu VPP300: Die Register der Vektoreinheit (1)

Vektorregister

- Größe:128 KBytes.
- Dynamisch konfigurierbar von

  8 Register a 2048 Elemente bis

  256 Register a 64 Elemente.

  (Die Anzahl der Register muß eine Potenz von 2 sein)
- „Optimale Vektorlänge“: 2048 Elemente.
- Kein rekursiver Zugriff innerhalb eines Vektorregisters!
- Mask-Register: 2 KBytes

Fujitsu VPP300: Die Register der Vektoreinheit (2)

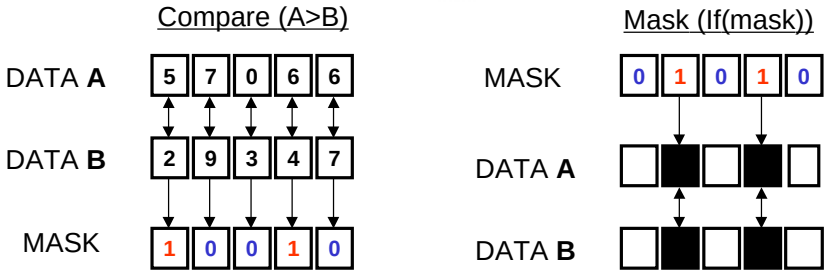
Vektorregister

- *Stripmining*: Sind die Vektoren im Programm länger als die Registerlänge, werden die Vektoren entsprechen den Hardewaregegebenheiten in Teilvektoren aufgespalten.
- Compiler kann und sollte über die maximale Länge eines Loops informiert werden. Die geschieht via Compilerdirektiven.
- Eigenschaften der Vektorregister sind rechnerabhängig !
  (CRAY T90 : 128 Elemente / Register)

Fujitsu VPP300: Die Register der Vektoreinheit (3)

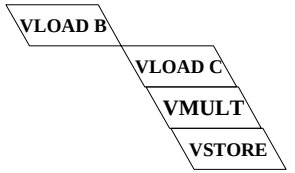
Maskregister

- Größe: 2 KBytes
- Konfigurierbare Register
- Grundlegende Operationen



Fujitsu VPP300: Vektorperformance

- Vektormultiplikation: A(I) = B(I) * C(I)

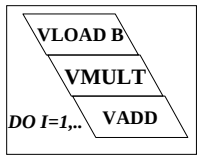


2 Ladeoperationen pro FLOP (N FLOP / 2 N Takte)

=> VECMULT erreicht max. 1/4 der Peak Performance.

- Matrix-Matrix-Multiplikation: A(I,J) = A(I,J) + B(I,K) * C(K,J)

```
DO J=1,..
  DO K=1,..
```



2 FLOP pro Ladeoperation (2 N FLOP / N Takte)

=> MATMULT erreicht (theor.) die Peak Performance.

Vektorisierung auf VPP300/700: Allgemeines

Vektorisierung bezeichnet den Vorgang, ein Programm so zu gestalten, daß die Ausführung im wesentlichen durch Vektoroperationen realisiert wird.

Stufen der Vektorisierung:

- Der **Compiler** versucht diese Aufgabe automatisch zu lösen. Dies gelingt vor allem dann gut, wenn der Programmierer beim Programmentwurf bereits die Vektorarchitektur berücksichtigt und Probleme wie z.B. Rekursionen zu vermeiden sucht.
- Mit **Compilerdirektiven** kann der Programmierer dem Compiler Informationen über das dynamische Verhalten des Codes geben und so in vielen Fällen eine Vektorisierung erzwingen. Der Programmierer trägt jedoch hier die Verantwortung für den korrekten Einsatz der Direktiven.
- **Analysedaten** teilen dem Programmierer mit wie und wo vektorisiert wurde und welche Stellen noch einer Optimierung bedürfen.

Vektorisierung auf VPP300/700: Determinanten der Vektorperformance

- **Anzahl der arithmetischen Operationen in einer Schleife**
- **Verhältnis von Speicherzugriffen zu arithmetischen Operationen**
- **Art der Operationen**
- **Verteilung der Vektorelemente im Speicher**
- **Vektorlänge** (Entspricht oft der Schleifenlänge)
- **Wann sollte welcher Rechner verwendet werden?**

Vektorlängen	Geeigneter Rechner
1 bis 8	PC
1 bis 32	Workstation
32 bis 64	Workstation / CRAY Vektorrechner
64 bis 128	CRAY T90
128 bis 256	VPP besser als Workstation
256 bis 2048	VPP deutlich besser als Workstation
> 2048	Optimale Nutzung

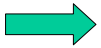
Vektorisierung auf VPP300/700: Vektorisierung von Schleifen (1)

- Anwendung einer Operation auf eine geordnete Menge von Daten des gleichen Typs: *Vektorisierung* i.a. möglich, falls keine
  - Datenabhängigkeiten
  - Rekursionen
  - Reduktionen
  - indirekte Adressierungen
  - IF-Anweisungen
  - GOTO-Anweisungen
  - Unterprogrammaufrufe
  - I/O Operationen

  dies verhindern.
- Zur Vereinfachung kann man von einer parallelen Ausführung von vektorisierten Schleifen ausgehen.

Vektorisierung auf VPP300/700: Vektorisierung von Schleifen (2)

- Skalare Abarbeitung: Sequentielle Ausführung aller Statements einer Schleife bei jeweils festem Index
- Semantische Definition von FORTRAN basiert auf der skalaren Abarbeitung von Schleifen



- Vektorisierung nur bei gleichem Ergebnis !
- Vektorisierende Compiler untersuchen
  - die Semantik,
  - die Durchführbarkeit, aufgrund des Instruktionssatzes des Rechners und
  - ob ein Performancegewinn erzielt wird.

Vektorisierung auf VPP300/700: Vektorisierung von Schleifen (3)

- Vektorisierung durch den Compiler ist nicht auf innere Schleifen beschränkt
- Compiler kann
  - Umordnungen,
  - Zusammenfassungen und
  - Aufrollen von Schleifen bzw. Teilschleifen

  durchführen, um eine bessere Vektorisierbarkeit zu erreichen.

- Der vektorisierte Code ist hochgradig rechnerabhängig!

Vektorisierung auf VPP300/700: Vektorisierung und Compiler

Überblick

- Vektorisierender Compiler:
  - FORTRAN: frt
  - C: vcc
- Höchste Optimierungsstufe:
  - FORTRAN: -Of
  - C: -K4

- Compilerdirektiven ins Programm einfügen:
  FORTRAN: !OCL ..... / C: #pragma .....
  Informationen für den Compiler über dynamisches Verhalten des Programms

Vektorisierung auf VPP300/700: Vektorisierung von Schleifen (4)

- Beispiel:

```
DO i = 1 , N
    a( i ) = b( i ) + c( i ) * s
    d( i ) = b( i ) * c( i )
ENDDO
```

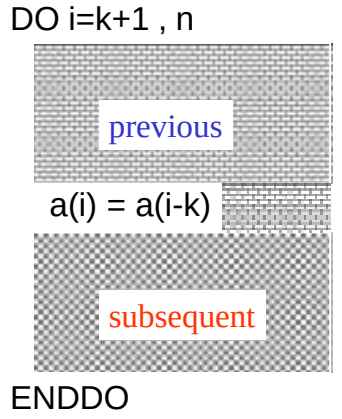
- Die Schleife wird vektoriell in drei Schritten abgearbeitet:
  1. **a(i) = b(i) + c(i) * s** ; (parallel) für alle i
  2. **d(i) = b(i) * c(i)** ; (parallel) für alle i
  3. **i = n + 1**

- Gut geeignet ist die FORTRAN90 Notation:

```
a( 1:n ) = b( 1:n ) + c( 1:n ) * s
d( 1:n ) = b( 1:n ) * c( 1:n )
```


Vektorisierung auf VPP300/700: Datenabhängigkeiten (1)

- Grundlegende Arten von Datenabhängigkeiten:
  - previous minus
  - previous plus
  - subsequent minus
  - subsequent plus



Vektorisierung auf VPP300/700: Datenabhängigkeiten (2)

- Datenabhängigkeiten (*previous* minus):

```
a_0( 1:n ) = a( 1:n )   ! Sichere urspr. Werte
DO i = k+1 , n
    a( i ) = a( i - k )
ENDDO
```

Beispiel: k=3 , n = 10

Parallel/Vektorisiert:

a(4:10)=a(1:7)

=> a(7) = a_0(4)

Falsches Ergebnis!

Beispiel: k=3 , n = 10

Sequentiell:

a(4)=a(1)
a(5)=a(2)
a(6)=a(3)
a(7)=a(4)=a_0(1)
...
a(10)=a(7)=a_0(4)

! k>= n/2 prinzipiell vektorisierbar !

Vektorisierung auf VPP300/700: Datenabhängigkeiten (3)

- Datenabhängigkeiten (*previous* plus):

```
a_0( 1:n ) = a( 1:n )   ! Sichere urspr. Werte
DO i = 1 , n - k
    a( i ) = a( i + k )
ENDDO
```

Beispiel: k=3 , n = 10

Parallel/Vektorisiert:

a(1:7)=a(4:10)

=> a(7) = a_0(10)

Identisches Ergebnis!

Beispiel: k=3 , n = 10

Sequentiell:

a(1)=a(4)
a(2)=a(5)
a(3)=a(6)
a(4)=a(7)
...
a(7)=a(10)

! Immer vektorisierbar !

Vektorisierung auf VPP300/700: Datenabhängigkeiten (4)

- Datenabhängigkeiten (*subsequent* minus):

```
a_0( 1:n ) = a( 1:n )   ! Sichere urspr. Werte
DO i = k + 1 , n
     a( i ) = c(i)
     b( i ) = a( i - k )
ENDDO
```

Beispiel: k=3 , n = 10

Parallel/Vektorisiert:

a(4:10) = c(4:10)

b(4:10) = a(1:7)

=> b(4:6) <=> a_0(1:3)

b(7:10)<=> c(4:7)

Identisches Ergebnis!

Beispiel: k=3 , n = 10

Sequentiell:

a(4)=c(4)
b(4)=a(1)
...
b(6)=a(3)

a(7)=c(7)
b(7)=a(4)
...
a(10)=c(10)
b(10)=a(7)

! Immer vektorisierbar !

Vektorisierung auf VPP300/700: Datenabhängigkeiten (5)

- Datenabhängigkeiten (*subsequent* plus):

```
a_0( 1:n ) = a( 1:n )   ! Sichere urspr. Werte
DO i = 1 , n-k
     a( i ) = c(i)
     b( i ) = a( i + k )
ENDDO
```

Beispiel: k=3 , n = 10

Parallel/Vektorisiert:

a(1:7) = c(1:7)

b(1:7) = a(4:10)

=> b(1:3) <=> c(1:3)

b(4:7)<=> a_0(7:10)

Falsches Ergebnis!

Beispiel: k=3 , n = 10

Sequentiell:

a(1) = c(1)
b(1) = a(4) (= a_0(4))
a(2) = c(2)
b(2) = a(5) (=a_0(5))
...
a(7) = c(7)
b(7) = a(10) (=a_0(10))

! k>= n/2 prinzipiell vektorisierbar !

Vektorisierung auf VPP300/700: Aufspalten von Schleifen

- Beispiel:

```
DO i = 2 , n
     a(i) = SQRT( a(i) )
     b(i) = b(i-1)
ENDDO
```

- Rekursive Referenz bezüglich b verhindert Vektorisierung der Schleife!
- Aufspalten: Schleife wird in zwei Einzelschleifen zerlegt:
  - a Schleife vektorisiert.
  - b Schleife wird X fach entrollt
- *Unrolling*: Ergebnis der vorhergehenden Operation direkt nutzen!

Vektorisierung auf VPP300/700: Umordnung von Schleifen (1)

Vektorisierung durch Umordnung

- Beispiel:

```
DO i = 2 , n
     b(i) = a(i-1)
     a(i) = 2.0 * a(i)
ENDDO
```

- *previous minus* Datenabhängigkeit

  ➡ In dieser Form nicht vektorisierbar

- Aber: Vertauschen der Anweisungen für i > 2 möglich !
- Diese Option wird vom Compiler erkannt

  ➡ Schleife wird vektorisiert

Vektorisierung auf VPP300/700: Umordnung von Schleifen (2)

- Vorsicht mit Compilerdirektive!

```
!OCL NOVREC
DO i = 2 , n
      b(i) = a(i-1)
      a(i) = 2.0 * a(i)
ENDDO
```

→ Falsches Ergebnis!

- Vorsicht bei F90 Array-Syntax:

  Falsch:

```
b(2:n) = a(1:(n-1))
a = 2.0 * a
```

  Korrekt:

```
b(2) = a(1)
a(2:n) = 2.0 * a(2:n)
b(3:n) = a(2:(n-1))
```


Vektorisierung auf VPP300/700: Reduktion und Rekursion (1)

Vektorisierung von Reduktionen

- Reduktion:

  Zuweisung, die den Rang einer Variablen reduziert

- Beispiele:
  - Vektorsumme
  - Skalarprodukt
  - Maximum / Minimum eines Vektors

- Reduktionen sind im allgemeinen nicht vektorisierbar, da es sich um Rekursionen vom Typ *previous minus* handelt!
- Beispiel: Vektorsumme

Vektorisierung auf VPP300/700: Reduktion und Rekursion (2)

Vektorisierung der Vektorsumme

```
s = 0.0
DO i = 1 , n
      s = s + a(i)
ENDDO
```

→

```
DO i = 1 , i_teilsumme
     temp(i) = 0.0
ENDDO

DO j = 1 , n , i_teilsumme
     DO i = 1 , 128
          temp(i) = temp(i) + a( j + i - 1 )
     ENDDO
ENDDO

s = 0.0
DO j = 1 , i_teilsumme
     s = s + temp(j)
ENDDO
```

!Vektoriell !

Vektorisierung auf VPP300/700: Reduktion und Rekursion (3)

- Vektorsumme und andere Reduktionen: Auf hochoptimierte Bibliotheksfunktion zurückgreifen!

Vektorisieren von Rekursionen

- Linear Rekursion 1. Ordnung:

```
DO i = 2 , n
      a(i) = a(i - 1) + s
ENDDO
```

!Sequentiell !

→

```
temp = a(1)
DO i = 2 , n
      a(i) = temp + (i - 1) * s
ENDDO
```

!Vektoriell !

Vektorisierung auf VPP300/700: Indirekte Adressierung (1)

Indirekte Adressierung über *index* Feld

- Indirekte Adressierung, sowie nichtlineare Adreßfortschaltung erschweren bzw. verhindern in vielen Fällen eine Vektorisierung
- Beispiele:

```
DO i = 1 , m
     b(i) = a(index1(i)) + c(i)              ! Indirekte Adressierung
ENDDO
DO i = 1 , m
     b(index2(i)) = a(index1(i)) + c(i)      ! Indirekte Adressierung
ENDDO
DO i = 1 , k
     b(i**2) = a(i) + c(i)                   ! Nichtlineare Adreßfort.
ENDDO
```

Indirekte LOAD/STORE Operationen ermöglichen eine Vektorisierung dieser Beispiele auf der VPP.

Vektorisierung auf VPP300/700: Indirekte Adressierung (2)

Indirekte Adressierung über Indexfunktionen

- Bei *intrinsic_function* kann der Compiler u. U. eine Vektorisierung der Schleife durchführen
- Bei *external_function* keine automatische Vektorisierung !

  ⇒ Compiler durch OPTION zum inlining veranlassen !
- Beispiel

```
DO i = 1 , m
     b(i) = a(index1(i)) * s                 ! Indirekte Adressierung
ENDDO
C Mit der inde   Funktion:
INTEGER FUNCTION index1(i)
      INTEGER i
      index1 = i
      if( i < 100 ) index1 = 100
END FUNCTION
```


Vektorisierung auf VPP300/700: Indirekte Adressierung (3)

- Compileroption zum inlining der Funktion index1:
  -Ne, index1 (bei **-Of** nicht nötig!)

- Der Compiler liefert dann folgende Analyse:

```
0000020 v     DO i = 1 , m
0000021 vi        b(i) = a(index1(i)) * s
0000022 v     ENDDO
```

vi: Schleife wurde vektorisiert und für die Indexfunktion wurde ein *inlinin* des Funktionsaufrufs vorgenommen.

Vektorisierung auf VPP300/700: Indirekte Adressierung (4)

- Indirekte Adressierung:

```
DO i = 1 , m
   a  index(i) ) = a( index(i) ) + b(i)
ENDDO
```

- Die Schleife wird nicht automatisch vektorisiert, da möglicherweise auf gleiche Elemente von **a** zugegriffen wird. (Zum Beispiel: **index(i) = 1** ; i=1,...,m).
- Ist dies nicht der Fall, kann der Programmierer dies dem Compiler mit Hilfe der folgenden Compilerdirektive mitteilen:
- **!OCL DISJOINT(a)**
- Die Vektorisierung der Schleife kann ebenfalls durch **!OCL NOVREC** erzwungen werden.

Vektorisierung auf VPP300/700: IF Statements (1)

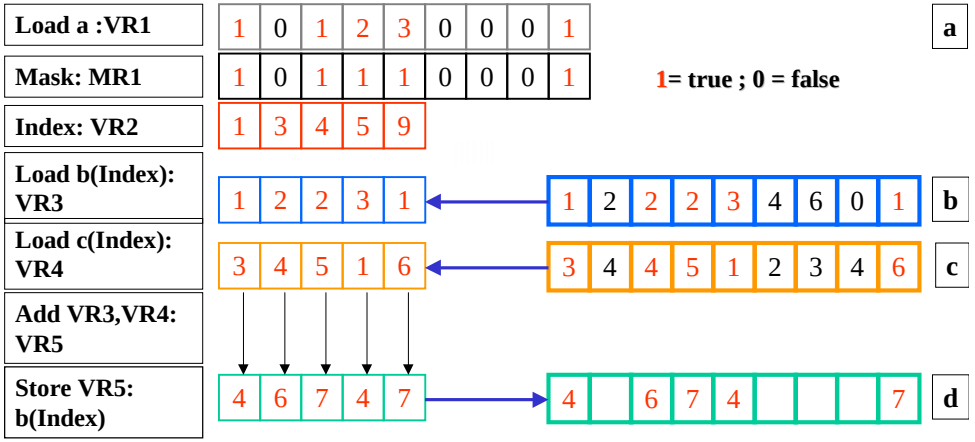
Vektorisierung bei Schleifen mit IF Anweisungen

- IF Anweisungen sind bei Vektorisierung immer problematisch !
  ➡ Optimale Lösung: keine IF Anweisung in der Schleife
- Alternativen:
  - *List Vector* Methode
  - *Compress/Expand* Methode
  - *Mask* Methode
- Namensgebung variiert ! (Rechnerabhängig)
- Prinzip ist aber ähnlich !
- Beispiel:

```
DO i = 1 , m
      IF (  a(i) > 0.0 ) d(i) = b(i) + c(i)
ENDDO
```

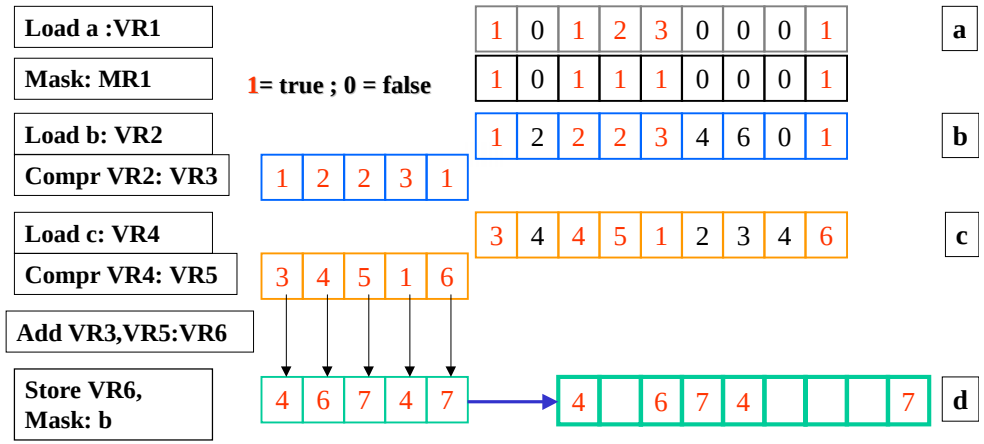

Vektorisierung auf VPP300/700: IF Statements (2)

List Vector Methode



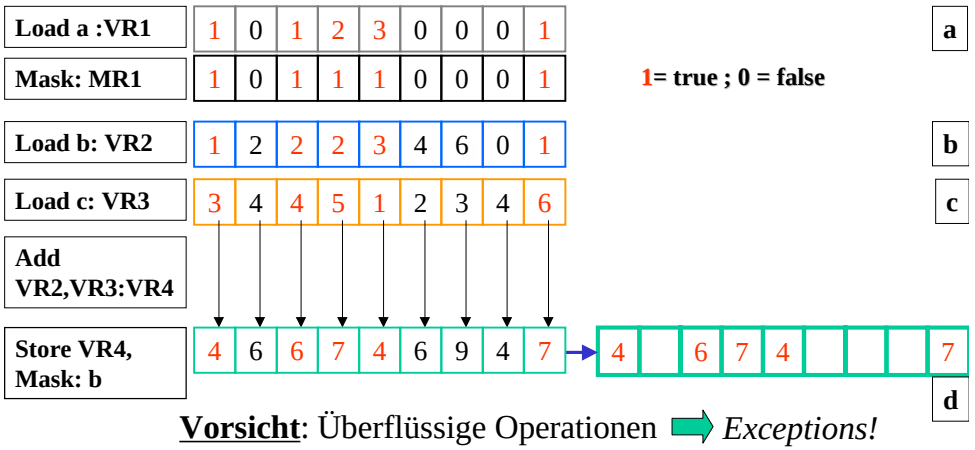
Vektorisierung auf VPP300/700: IF Statements (3)

Compress/Expand Methode



Vektorisierung auf VPP300/700: IF Statements (4)

Mask Methode



Vorsicht: Überflüssige Operationen ➡ *Exceptions!*

Vektorisierung auf VPP300/700: IF Statements (5)

Anwendungsgebiete

Die VPP setzt die drei Methoden in Abhängigkeit von der *true ratio* des *IF* Statements wie folgt ein:

- *List Vector*: Geringe *true ratio* (Overhead durch indirektes Laden!)
- *Compress/Expand:*Geringe bis moderat hohe *true ratio. (*VPP verwendet diese Methode sehr selten*)*
- *Mask:*Sehr hohe *true ratios*. Aber: Test auf *Exceptions* nötig!

Vektorisierung auf VPP300/700: IF Statements (6)

Benutzersteuerung

Auswahl der Methode durch den Programmierer via:

- Compileroptionen (auf Subroutinen-Ebene):

  frt -Wv, -l[m | l | c]
- Besser: Compilerdirektiven (auf Statement-Ebene):
  - !OCL VCT(LIST)
  - !OCL VCT(CEX)
  - !OCL VCT(MASK)
- Am besten: *True ratio* via Compilerdirektive angeben und Auswahl dem Compiler überlassen:

```
!OCL  IF(62)                ! True ratio is approx. 62%
```


Vektorisierung auf VPP300/700: Speicherzugriff (1)

Organisation des Speicher

- Standard-Speicherbausteine: SDRAMS
- Relativ lange Zugriffszeiten: Mindestens 20 Taktzyklen zwischen zwei aufeinander folgenden Speicherzugriffen

- *Interleaving* in 512 Speicherbänke
- Aufeinanderfolgende Speicheradressen werden über die Speicherbänke verteilt
- Optimale Ausnutzung nur bei kontinuierlichem (*stride=1)* Zugriffen möglich

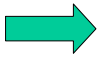
Vektorisierung auf VPP300/700: Speicherzugriff (2)

Schrittweise des Zugriffs

- Beispiel:

```
DO i = 1 , m-1 , memstride
    a(i) = b(i) * s + c(i)
ENDDO
```

- VPP besitzt nur einen Load- und einen Storepfad zum Hauptspeicher (CRAY T90: 2 x Load + 1 x Store)

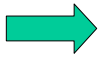


- Hohe Speicherbandbreite i.a. nur für *memstride=1* Zugriff möglich

Vektorisierung auf VPP300/700: Speicherzugriff (3)

Allgemeines

- Mehrdimensionale Felder: Die Organisation von Matrizen in FORTRAN (C) spaltenweise (zeilenweise) erfordert eine Bearbeitung des ersten Index in der innersten Schleife.
- Vektorisierung der äußeren Schleife: Adreßfortschaltung entspricht der Länge des Feldes in der ersten Dimension. Insbesondere bei Feldlängen von 2er-Potenzen treten oft Speicherbankkonflikte auf.



- *Worst Case* für VPP: Feldlängen von 64 * DOUBLE PRECISION

Vektorisierung auf VPP300/700: Speicherzugriff (4)

Überdimensionierung

- Überdimensionierung von Feldern bei nicht kontinuierlichem Speicherzugriff:

Bzgl. i nicht vektorisierbar !

```
REAL a(128,n), b(128,n)
DO i = 1 , m
    i1 = index(i)
    DO j = 1 , n
        a( i1 , j ) = a( i1  , j )
    ENDDO
ENDDO
```

Bzgl. i nicht vektorisierbar !

Überdimensioniert!

```
REAL a(129,n), b(129,n)
DO i = 1 , m
    i1 = index(i)
    DO j = 1 , n
        a( i1 , j ) = a( i1 , j )
    ENDDO
ENDDO
```


Vektorisierung auf VPP300/700: Compiler und Schleifen (1)

- Vertauschen von Schleifen

```
DO i = 1 , n
    DO j = 2 , m
        a( i , j ) = a( i , j-1 ) + s
    ENDDO
ENDDO
```

- Problem: Bezüglich **j** kann nicht vektorisiert werden, da diese Schleife eine Rekursion enthält.
- Lösung: Der Compiler vektorisiert die äußere (**i-**) Schleife und entrollt die innere (**j-**) Schleife zweifach (*unrolling*). Es wird folgende (äquivalente) Schleifenkonstruktion generiert:

```
DO j = 2 , m , 2 !  Falls m durch 2 teilbar
    DO i = 1 , n
        a( i , j )   = a( i , j-1 ) + s
        a( i , j+1 ) = a( i , j )   + s
    ENDDO
ENDDO
```


Vektorisierung auf VPP300/700: Compiler und Schleifen (2)

- Die vom Compiler vorgenommenen Änderungen werden in einem Analyse-Listing dokumentiert, das beim Compilieren des Programmes ausgegeben werden kann.
- Beachte: Unterschiedliche Compiler können dieses Problem verschiedenartig lösen !
- Beispiel *CRAY T90*. Für n > 128 (=Registerlänge der *CRAY T90*) erzeugt der *CRAY* Compiler folgende Schleifenkonstruktion (*Loop Unwinding*):

```
DO i = n+1 , n*m
    a( i , 1 )   = a( i -n , 1) + s
ENDDO
```

Vorteile: Kontinuierlicher Speicherzugriff und Vergrößerung der Vektorlänge.

Vektorisierung auf VPP300/700: Compilerdirektiven (1)

Compilerdirektiven sind Informationen des Programmierers über das dynamische Verhalten des Programms, die der Compiler nicht am Quellcode erkennen kann.

- Die alleinige Verantwortung für den korrekten Gebrauch trägt der Programmierer!

- Compilerdirektiven sind in FORTRAN als Kommentarzeilen angelegt und beginnen am Zeilenanfang mit

  !OCL (=*Optimization Control Line*).

Vektorisierung auf VPP300/700: Compilerdirektiven (2)

- Compilerdirektiven können sich auf Programmeinheiten (P), DO-Loops (D), Array-Expressions (A) und Statements (S) beziehen.
  Beispiele:

```
!OCL  NOVREC(a)                          !.....Programmeinheiten
      SUBROUTINE sub1(a,b)

!OCL  REPEAT (100)                       !.....DO-Loop
      DO i = 1 , m

!OCL  IF(95)                             !.....Statement
      IF(w(i) > 0 ) THEN
```

- Überblick über verfügbare Compilerdirektiven: Vgl. beiliegende Übersicht

Vektorisierung auf VPP300/700: Compilerdirektiven (3)

Compilerdirektiven in C

- Syntax:

  #pragma <scope_spec> <opt_spec>

- <scope_spec> = loop: Anweisung bezieht sich auf die folgende Schleife!

- <opt_spec>=
  - novrec
  - disjoint
  - repeat
  - scalar
  - .....

Vektorisierung auf VPP300/700: Allgemeine Hinweise (1)

- Auswahl eines geeigneten Algorithmus unter Berücksichtigung von Bibliotheksroutinen unt anderen Quellen wie z.B.:
  - http://gams.nist.gov
  - htpp://www.netlib.org
  - http://www.nhse.org
- Übersichtliche, wohlkonstruierte, lesbare Programme, frei von sinnwidrigen bzw sinnfremden Konstrukten!
- Davon abweichende Modifikationen, die zur Vektorisierung vorgenommen wurde müssen ausführlich dokumentiert sein!

Vektorisierung auf VPP300/700: Allgemeine Hinweise (2)

- Anordnung von Schleifen:
  - Indexraum der innersten Schleife maximal wählen (<=> Vektorlänge)
  - Innerste Schleife sollte (in Fortran Programmen) immer auf dem 1. Index arbeiten.
  - In geschachtelten Schleifen sollten die weiteren Indizes von der innersten zu äußersten Schleife von links nach rechts abgearbeitet werden!
- Angabe der maximalen Länge einer Schleife via Compilerdirektive:

  !OCL REPEAT(maxlength)

Vektorisierung auf VPP300/700: Allgemeine Hinweise (3)

- Vertauschen von Schleifen:
  - Wird in der Regel vom Compiler erkannt
  - Oft wird der Code leichter lesbar und analysierbar
- IF-Abfragen:
  - Sollten in Schleifen vermieden werden.
  - Wenn sie unausweichlich sind, sollte man die IF-Blöcke in der Reihenfolge der *true-ratio* anordnen und dem Compiler die *true-ratio* mitteilen.
- *Mixe mode expressions* vermeiden!

Vektorisierung auf VPP300/700: Allgemeine Hinweise (4)

- Konstanten an Stelle von Variablen!

  PARAMETER Anweisungen erleichtern dem Compiler die Vektorisierung erheblich.
- Dynamische Speicherverwaltung vermeiden!

  Dynamische Arrays erschweren die Vektorisierung.
- Loop Unrolling dem Compiler überlassen!
- Loop Unwinding durchführen!

  Zusammenfassen einer (kleinen) inneren Schleife mit einer äußeren Schleife kann die Vektorlägen deutlich vergrößeren.
- Loop Fusion!

  Vereinigung mehrerer unabhängiger Schleifen mit gleichem Indexraum. Kann vom Compiler übernommen werden. (-Wv,-fusion)

Vektorisierung auf VPP300/700: Allgemeine Hinweise (5)

- Klare Programmierung und Indizierung!

  Zwischenrechnungen vermeiden und den Code nicht unnötig komplizieren:

  a(i+4,j-3) = ...
  ! an Stelle von
  ia = i + 4
  ja = j - 3
  a(ia,ja)= ....
- Strength Reduction!

  Zeitaufwendige Operationen vermeiden!

  a ** 2 durch a * a
  a ** 0.5 durch dsqrt(a)
  a / 2 durch a * 0.5

Vektorisierung auf VPP300/700: Allgemeine Hinweise (6)

- Genauigkeiten:

  VPP Standard: 32 Bit (CRAY: 64 Bit)
  - REAL*16 /COMPLEX*16 sowie
  - INTEGER*1 , INTEGER*2, INTEGER*8

  werden von der VPP nicht vektorisiert.
- Modularisierung nicht übertreiben!

  Data, *Commo* und *Save* Anweisungen nicht verwenden!
- FORTRAN Code für rechenintensive Programmteile!
- Kein vektorisierender C++ Compiler!

Vektorisierung auf VPP300/700: Allgemeine Hinweise (7)

- Compilerdirektiven:

  Alle Informationen dem Compiler mitteilen!

  Vorsicht: Den Compiler nicht hintergehen!

- Unterprogramme:
  - Einbinden durch den Programmierer
  - Einbinden durch Compiler via *Inlining*

Vektorisierung auf VPP300/700: Allgemeine Hinweise (8)

- Standardbibliotheken verwenden!

  Für viele Programmkernels - insbesondere aus der Linearen Algebra - existieren Standardbibliotheken, die bereits speziell für die VPP optimiert sind. Die Portabilität wird dadurch nicht berührt, da Programmbibliotheken wie IMSL, NAG, LAPACK oder SCALAPACK auf jedem gebräuchlichen Rechner vorhanden sind!

  Diese Bibliotheken enthalten die optimierten BLAS-Routinen (=*Basic Linear Algebraic Subroutines*), welche wiederum gemäß ihrer Ordnung unterschieden werden:

  - BLAS - Level 1 (**O(N)** : Vektorsumme , Skalarprodukt, ...)
  - BLAS - Level 2 (**O(N**2)** : Matrix-Vektor-Multiplikation, ...)
  - BLAS - Level 3 (**O(N**3)** : Matrix-Matrix-Multiplikation, ...)

Vektorisierung auf VPP300/700: Analyse Listing

- Analyse der vom Compiler vorgenommenen Vektorisierungen anhand des Analyselistings
- Erzeugen des Analyselistings:

  frt ... -Post -Wv,-m3 -Z <Analysefile> -c <Sourcefile>

  (vcc -Wv,-m3,-Pst -Z <Analysefile> -c <Sourcefile>)
- Liste ausgewählter Vektorisierungskürzel
  - v : Vektorisiert
  - vi: Vektorisiert mit inling code
  - vv: Vektorisiert mit array option
  - s : Skalar
  - m: Mixed mode

Vektorisierung auf VPP300/700: Der Vektorcompiler (1)

- Aufruf auf der VPP: **frt** <Optionen> *file.f*
- Empfohlene Optionen:

Testphase:	**-On**	**-g**	**-Wv,-m3**	**-sc -A2 -AR**	**-Post**
Optimierungsphasen:	**-Ob**	**-NI**	**-Wv,-m3,-qs,**	**-AR**	**-Post**
	-Oe	**-NI**	**-Wv,-m3,-qm,**	**-AR**	**-Post**
	-Of	**-NI**	**-Wv,-m3,-qm**	**-Sw**	**-Post**

- Von Systemseite sind einige Optionen bereits voreingestellt. Alle aktiven Compilerswitches werden mit -Po ausgegeben.
- Es ist sinnvoll die Compilerausgabe mit **-Z** *file.lst* in das File *file.lst* umzulenken. Dort sind dann auch eventuelle Fehlermeldungen zu finden.

Vektorisierung auf VPP300/700: Der Vektorcompiler (2)

- Ausgewählte Optionen (1):
  - **-On; -Ob; -Oe; -Of** : Optimierungsstufen: *no; basic; extended; full*
  - **-sc** : Keine Vektorisierung
  - **-g**: Debugging-Information für fdb generieren
  - **-Dxxx**: Diverse Debug-Optionen
  - **-A2** : Statementnummern bei Fehlern angeben
  - **-AR**: Konstante Daten in schreibgeschützten Bereichen
  - .....

Vektorisierung auf VPP300/700: Der Vektorcompiler (3)

- Ausgewählte Optionen (2)
  - **-Wv:** Folgende Optionen können an den Vektorisierer weitergegeben werden:

    ,-m3: Informationen über die vom Compiler vorgenommene Vektorisierung bzw. „Nichtvektorisierung“.

    ,-qm: Partielle Vektorisierung erlaubt.

    ,-fusion: *Loop fusion* erlaubt.

    ,-ilfunc: *Inlining* von Funktionen.

Vektorisierung auf VPP300/700: Der Vektorcompiler (4)

- Ausgewählte Optionen (3)
  - **-Exxxx: Diverse Optionen zu Programmanalyse-Informationen**
  - **-Pxxxx**: Ausgabe von:
    - *Optionsliste (o)*
    - *Source Code (s)*
    - *Statistik (t)*
    - *Cross Reference List (x)*
    - *Include File Name-List (d)*
  - **-Z <filename> :** Umlenken des Outputs
  - **-o <filename> :** Name des Objekts
  - -c: Nur Objektfile generieren (.o)
- Mehr unter **man frt** oder **http://www.lrz-muenchen.de/services/compute/vpp/compiler/einstellungen.html**

Vektorisierung auf VPP300/700: Der Vektorcompiler (5)

- Mehr unter
  - **man frt**
  - **http://www.lrz-muenchen.de/services/compute/vpp/compiler/einstellungen.html**
  - RRZE Homepage

Vektorisierung auf VPP300/700: Analysewerkzeuge (1)

PEPA: Messung des Gesamtzeitverhaltens eines Programmes:

setenv FJPEPA ON
jobexec a.out

liefert Informationen über: **VU busy ratio**
Average Performance [Mflops]
Operation count in SU and VU
Operation count in VU
Measurement time [sec]

- Overhead durch diese Messung ist vernachlässigbar.
- Diese Informationen können für bestimmte Programmteile mit Hilfe spezieller Subroutinen (VPARESET/-START/-STOP/-RET) gewonnen werden.

Vektorisierung auf VPP300/700: Analysewerkzeuge (2)

SAMPLER: FJSAMP ist ein dynamischer Profiler, der mißt auf welche Adreßbereiche zugegriffen wird.

setenv FJSAMP „file:samp.dat,interval=10,type=vtime“
jobexec ./a.out
fjsamp a.out

- Der Output enthält Aussagen über absolute und prozentuale „Treffer“
  - in Unterprogrammen,
  - in Schleifen,
  - in Array-Konstrukten,
  - sowie Aussagen über Vektorlängen.
- Der Overhead durch die Messung beträgt etwa 10% der urspr. Rechenzeit.
- Übersicht über alle verfügbaren Werkzeuge:
  http://www.lrz-muenchen.de/services/compute/vpp/optvecpa /tools.html

Vektorisierung auf VPP300/700: Analysewerkzeuge (3)

Analyse der Operationscounts:

- Compilieren + Linken:

  frt -Of -Wv,-m3,-qm -Wc ... -lfjcnt

- Starten + Auswerten:

  setenv FJCNT „file:count “
  jobexec a.out
  fjsamp *.ainf

Vektorisierung auf VPP300/700: Referenz

Dank für viele Anregungen geht an:

Dr. Ing. Claus-Uw Linster

Dr. Matthias Breh , LRZ München

Vektorisierung auf VPP300/700: Übungen (1)

Allgemein:

- Verwenden Sie, soweit nicht anders angegeben die jeweils höchste Optimierungsstufe!
- Führen Sie immer eine Analyse Ihres Compilerlistings durch!
- Verwenden Sie immer den FJPEPA (setenv FJPEPA ON)
- Führen Sie immer mehrere Läufe durch und notieren di jeweiligen Laufzeiten!
- C Programmierer werden gebeten, alle Problemstellungen auch als C Code verfassen und mit den FORTRAN Resultaten zu vergleichen!
- Testen Sie möglichst viele Beispiele aus der Vorlesung!

Vektorisierung auf VPP300/700: Übungen (2)

A) Datenabhängigkeiten

Das Programm **PrevMin.f** führt eine Schleife mit *previous minus* Abhängigkeit vektorisiert und sequentiell aus.

- Vergleichen Sie die Ergebnisse für verschiedene k !
- Ändern Sie das Programm so ab, daß Sie auch die *previous plus, subsequent minus , subsequent minus* Abhängigkeiten simulieren können.
- Vergleichen Sie auch hier die Ergebnisse von sequentieller und vektorisierter Schleife

Vektorisierung auf VPP300/700: Übungen (3)

B) *Inlining*

Das Programm **Inline.f** führt eine Schleife mit mit einer indirekten Adressierung via Funktionsaufruf aus.

- Übersetzen Sie jeweils das Programm ohne und mit -Of und vergleichen Sie sowohl Analyse-Listing und Laufzeiten.

C) Vertauschen von Schleifen:

Das Programm **Loop.f** führt enthält eine Doppelschleife, deren Reihenfolge vom Compiler vertauscht wird, um eine Vektorisierung zu ermöglichen.

- Bestimmen Sie die Performance für die drei Beispiel - Parametersätze

Vektorisierung auf VPP300/700: Übungen (4)

D) Überdimensionierung

Das Programm **Ueberdim.f** zeigt den Effekt der Überdimensionierung von Feldern

Das Schleifenstatement lautet (im Gegensatz zur Vorlesung!)

a(index(i),j) = a(index(i),j) + s

- Vergleichen Sie die Performance der beiden Methoden!
- Untersuchen Sie auch andere Dimensionierungen!

Vektorisierung auf VPP300/700: Übungen (5)

E) Matrix-Vektor-Multiplikation (MVM) -1 -

Das Programm **MVM_Stand.f** führt MVM‘s mit maximaler Dimension N=3000 durch.

- Was fällt am Analyse-Listing auf?
- Bestimmen Sie die Performance als Funktion der Dimension!
- Was fällt auf?
- Analysieren Sie die Schleifenlänge mit FJSAMP!

Vektorisierung auf VPP300/700: Übungen (6)

E) Matrix-Vektor-Multiplikation (MVM) - 2 -

Das Programm **MVM_Blas.f** verwendet die Standard-BLAS Routine für die MVM.

- Bestimmen Sie die Performance als Funktion der Dimension!
- Versuchen Sie **MVM_Stand.f** so zu verändern, daß Sie die Performance der BLAS Routine erhalten.