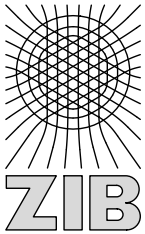
Übungen zur Vorlesung „Hochleistungsrechnen für Naturwissenschaftler“

– Auswertung von Projekt 1 –

Hinnerk Stüben



Konrad-Zuse-Zentrum für Informationstechnik Berlin

Humboldt-Universität zu Berlin – Wintersemester 2010/2011

BLAS daxpy()

- Das Original → `http://www.netlib.org/blas/blas.tgz`

```
      SUBROUTINE DAXPY(N,DA,DX,INCX,DY,INCY)
*     .. Scalar Arguments ..
      DOUBLE PRECISION DA
      INTEGER INCX,INCY,N
*     ..
*     .. Array Arguments ..
      DOUBLE PRECISION DX(*),DY(*)
*     ..
*
*  Purpose
*  =======
*
*     constant times a vector plus a vector.
*     uses unrolled loops for increments equal to one.
*     jack dongarra, linpack, 3/11/78.
*     modified 12/3/93, array(1) declarations changed to array(*)
*
      ...
```


Ein eigenes daxpy-Unterprogramm

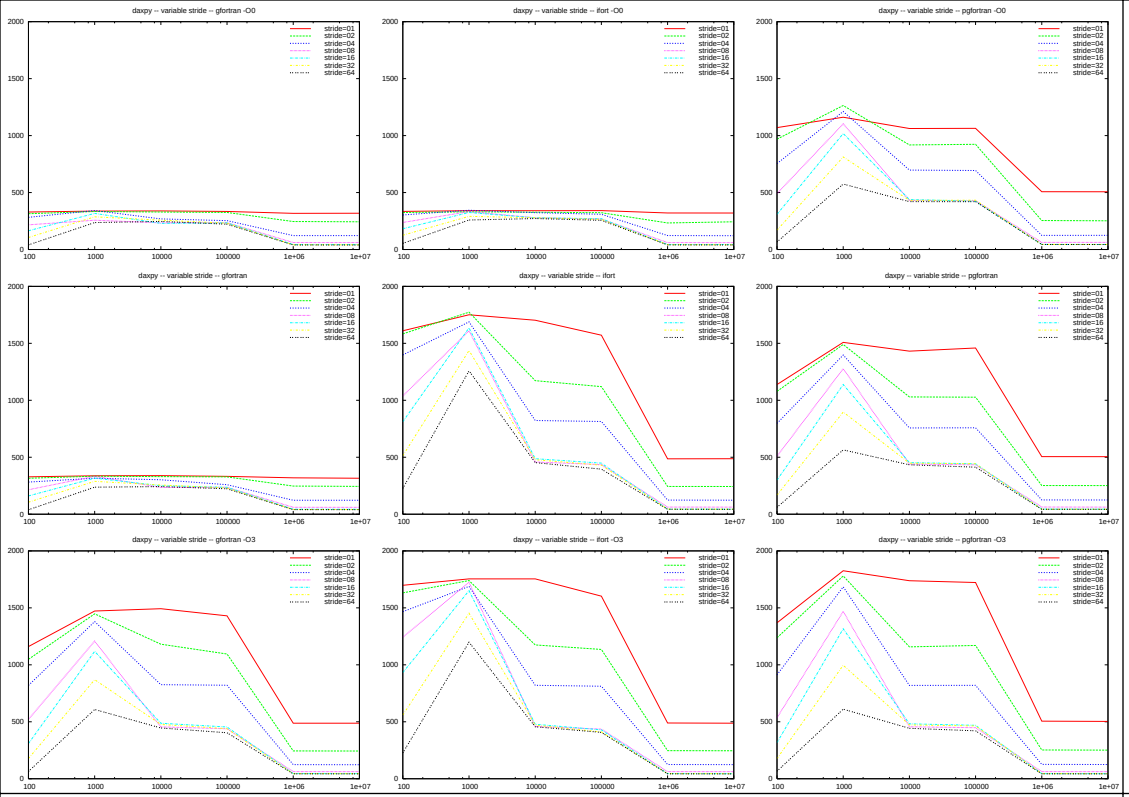
- Fortran

```
subroutine daxpy(inout, in, a, n, stride)
  implicit none
  real(8) :: inout(n), in(n), a
  integer :: i, n, stride

  do i = 1, n, stride
     inout(i) = inout(i) + a * in(i)
  enddo
end
```

- C

```
void daxpy(double inout[], double in[], double a, int n, int stride)
{
    int i;

    for (i = 0; i < n; i += stride)
        inout[i] += a * in[i];
}
```


Ergebnisse
– variable Schrittweite (stride) –



Rechnen im L2-Cache

- Performance in MFlop/s, Vektorlänge 10.000

Schrittweite	Compiler		
	`gfortran -O3`	`ifort -O3`	`pgfortran -O3`
1	1492	1754	1738
2	1181	1174	1157
4	826	820	819
8	455	459	456
16	487	478	481
32	476	470	472
64	445	456	443

Rechnen im L1-Cache

- Performance in MFlop/s, Vektorlänge 1.000

Schrittweite	Compiler		
	`gfortran -O3`	`ifort -O3`	`pgfortran -O3`
1	1472	1754	1826
2	1447	1739	1780
4	1379	1688	1684
8	1208	1724	1468
16	1118	1651	1316
32	869	1452	996
64	607	1198	610

Laden aus dem Hauptspeicher

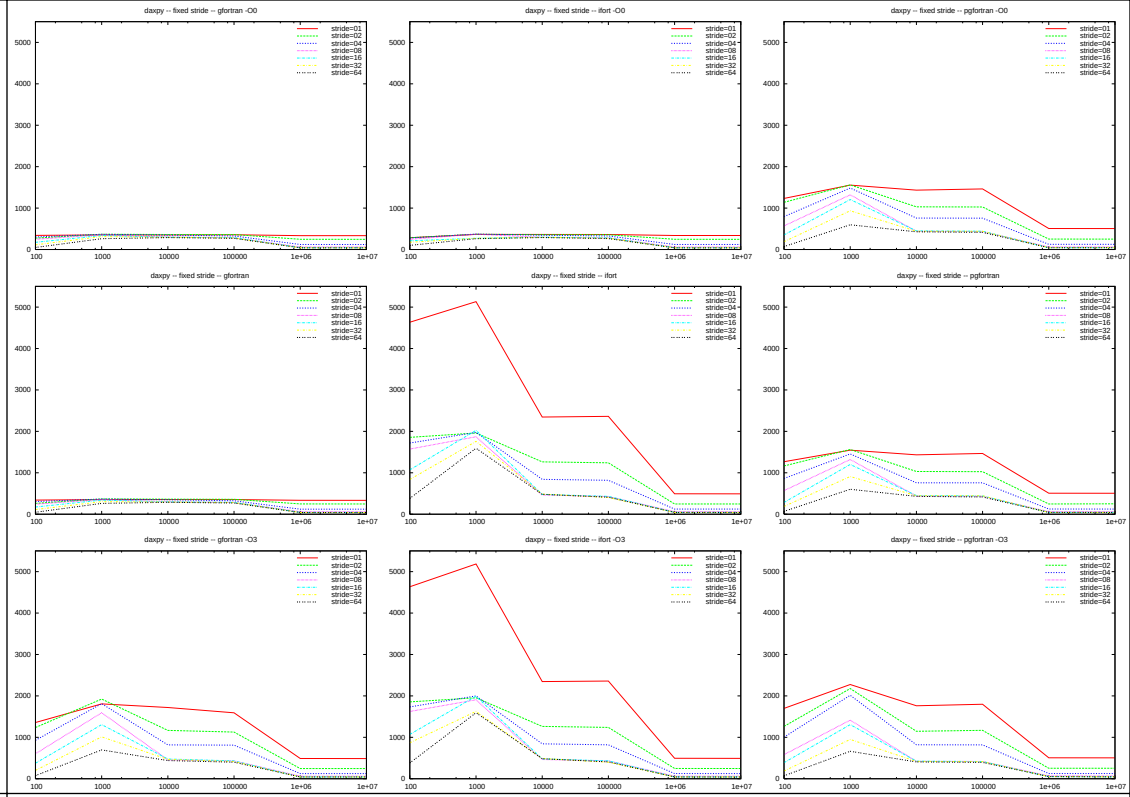
- Performance in MFlop/s, Vektorlänge 10.000.000

Schrittweite	Compiler		
	`gfortran -O3`	`ifort -O3`	`pgfortran -O3`
1	487	487	503
2	243	245	251
4	123	124	124
8	62	62	62
16	44	44	44
32	44	44	44
64	43	43	43

- Die Compiler haben keinen Einfluss auf das Laden aus dem Hauptspeicher

→ Bei kleinen Schrittweiten ist die Rechenleistung proportional zur Schrittweite

Ergebnisse
– fest programmierte Schrittweite –



daxpy-Unterprogramme mit fest programmierter Schrittweite

- Fortran

```
subroutine daxpy_stride4(inout, in, a, n)
  implicit none
  real(8) :: inout(n), in(n), a
  integer :: i, n

  do i = 1, n, 4
     inout(i) = inout(i) + a * in(i)
  enddo
end
```

- C

```
void daxpy_stride4(double inout[], double in[], double a, int n)
{
    int i;

    for (i = 0; i < n; i += 4)
        inout[i] += a * in[i];
}
```


Fest programmierte Schrittweite

→ Bei Rechnung in den Daten-Caches kann insbesondere der Intel-Compiler deutlich besser optimieren.

- Beim Laden aus dem Hauptspeicher sind die Ergebnisse unverändert.

Diskussion und Anmerkungen

Diskussion und Anmerkungen

- Implementierung
- Beobachtungen zur Rechenleistung
- Unterschiede der Compiler
- Fehlerdiskussion

Implementierung – daxpy()-Unterprogramm

- C

```
void daxpy(double y[], double x[], double a, int n, int s)
{
    int i;

    for (i = 0; i < n; i += s)
        y[i] += a * x[i];
}
```

→ Der Faktor a sollte vom Typ `double` sein,
n sollte vom Typ `int` sein.

Implementierung – Berechnung der Rechenleistung

- Anzahl Schleifendurchläufe: $\lceil n/s \rceil$
  - Fortran: `ceiling(real(n, kind = 8) / real(s, kind = 8))`
  - C: `ceil((double) n / (double) s)`
- Anzahl der Operationen pro Schleifendurchlauf: `2`
- Wiederholungen: `r`

⇒ Anzahl Rechenoperationen: `2 * ⌈n/s⌉ * r`

- Alternativen:
  - eigene Implementierung von `ceil`, z.B. mit `if`
  - Auswertung der Laufvariablen `i`

Qualitative Beobachtungen

- Man sollte erkennen:
  - den Einfluss der Daten-Caches,
  - den Einfluss der Schrittweite.
- Der Einfluss der Schrittweite ist von grundlegender Bedeutung, weil
  - alle Daten immer erst aus dem Speicher geladen werden können,
  - nur vergleichsweise wenige Anwendungsklassen extrem von den Caches profitieren.

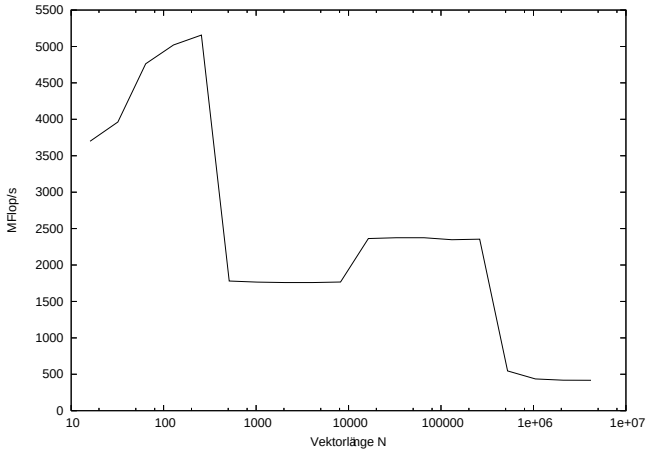
→ Merkspruch: Speicherzugriffe am besten mit Schrittweite 1.

Quantitative Beobachtungen

- Angabe der Größen der Daten-Caches
  - ist mit den vorgegebenen Schrittweiten nur grob möglich
- Angabe der Cache-Zeilenlänge
  - lässt sich recht gut ablesen

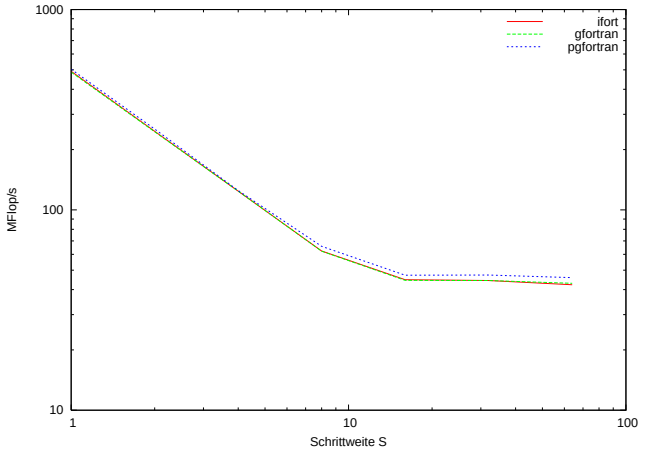
Größen der Daten-Caches

- Messung mit Zweierpotenzen als Vektorlängen, Compiler: `ifort -O3`



Cache-Zeilenlänge

- Vektorlänge: 10.000.000, Optimierung: `-O3`



Unterschiede der Compiler

- Die Compiler zeigen im Detail Unterschiede in der Rechenleistung.
- Insbesondere beim Rechnen im L1-Cache ist der vom Intel-Compiler generierte Code deutlich schneller.
- Wichtige Beobachtung:
  - die Bezeichnung der Optimierungsstufen ist nicht standardisiert

(Mess-) Fehlerdiskussion

- Das vorgeschlagene Verfahren war im wesentlichen genau genug . . .
- . . . bis auf ggf. folgende Situationen:
  - Fehler sollten diskutiert werden bei unsystematischen Kurvenverläufen
  - Fehler könnten für die kleine Vektorlänge 100 diskutiert werden