6 Vektorisierung

6.1 SIMD

Das Konzept der "Fließbandarbeit" ist auch auf dem Computer geeignet, besonders effiziente Abläufe zu organisieren. Eine entscheidende Voraussetzung ist, dass man das Programm (ggf. durch Überarbeitung des Algorithmus) so organisieren kann, dass einfache Rechenoperationen gleichförmig und ohne Rekursion über lange Datensätze (Vektoren) laufen. Wir haben es also mit Problemen vom Typ SIMD (**S**ingle **I**nstruction **M**ultiple **D**ata) zu tun.
In der 1980er Jahren waren **Vektorrechner** populär, die mit speziellen Prozessorstrukturen (*vector pipeline, array processor*) auf langen Vektoren sehr hohe Performance erzielten. Dazu war es nötig, besondere Register und Lademechanismen einzubauen, die den schnellen Datentransfer zwischen Prozessor und RAM ermöglichten – das machte die Rechner sehr teuer.
In modernen Prozessoren ist, in kleinerem Maßstab und viel billiger, dieses Konzept wieder anzutreffen unter der Bezeichnung **SSE** (*Streaming SIMD Extensions*). Spezielle Vektorregister gibt es nicht, aber der mehrstufige Aufbau des **Cache** erlaubt es, größere Datensätze "in der Nähe" des Prozessors zu halten. Außerdem werden die Daten als *cache lines* (z.B. 128B zusammenhängend) vom RAM in den Cache geladen, das erleichtert die Bearbeitung von fortlaufend im RAM gespeicherten Datensätzen (*contiguous storage*). Das ist wertvoll, denn die Anbindung des Prozessors an den Speicher ist heutzutage der kritische Engpass in der Rechnerarchitektur.

6.2 Vektorisierung

Unter SIMD–Bedingungen bietet es sich an, möglichst vektorisiert zu programmieren. Das heißt

- Betroffene Felder möglichst **fortlaufend speichern**.
  Bei mehrfach indizierten Feldern beachten: in Fortran läuft der *erste* Index am schnellsten, in C der *letzte*.
- Vektorielle Berechnungen in die **innerste Schleife** setzen.
- Keine **Rekursionen** innerhalb der Vektorschleife.
  Ausnahme: für bestimmte Standard–Operationen wie Summation gibt es spezielle Macros.
- Keine **Funktionsaufrufe**.
  Ausnahmen: bestimmte mathematische Funktionen, Inline–Funktionen etc.
- Kein plötzliches Ende der Schleife, z.B. durch **break**.
- Kein **I/O** in der Vektorschleife.

Wenn es der Algorithmus erlaubt, kann man auch versuchen, mit einfacher Genauigkeit zu rechnen (*float* statt *double*), denn sowohl die Arithmetik als auch die Datentransfers sind dann doppelt so schnell.

6.3 Auto–Vektorisierung

Moderne Compiler sind in der Lage, die Vektorisierung automatisch vorzunehmen. Man braucht sich also nicht mit speziellen (von Hardware und Compiler abhängigen) Instruktionen zu plagen. Der Compiler ist auch in der Lage, Schleifenstrukturen umzubauen, um die Vektorisierung zu fördern, aber dem sind Grenzen gesetzt, weil auf keinen Fall fehlerhafter Code entstehen darf. Es liegt also in der Hand des Programmierers, die Vektorisierung gut vorzubereiten:

- Einfache Datenstruktur mit fortlaufend gespeicherten Vektordaten.
- Übersichtlicher Programmablauf mit vektorisierbaren, innersten Schleifen.
- Die Iterationszahl der Vektorschleife soll zur Laufzeit, aber vor Beginn der Schleife berechenbar sein.
- Bei unberechtigtem Verdacht auf Rekursion: mit Direktiven (s.u.) nachhelfen.
- Weitere Einschränkungen (keine Mischung von Datentypen etc) hängen vom Entwicklungsstand des Compilers ab...

6.4 Programmbeispiele

6.4.1 Einfache Schleifen

```
for (i=0; i<n; i++) {                /* ok */
   z[i] = z[i] + a[i]*b[i];
}

for (i=0; i<100; i++) {              /* ok */
   a[i] = b[i] * c[i];
   if (a[i] < 0.0) a[i] = 0.0;
}

s = 0.;
for (i=0; i<n; i++) {                /* ok */
   s += v[i]*w[i];                   /* Summation wird erkannt */
}

for (i=0; i<n-1; i++) {              /* ok */
   v[i] = w[i] + p*v[i+1];           /* v[i+1] greift voraus */
}
```

```
for (i=1; i<n; i++) {             /* nicht vektorisierbar */
   v[i] = w[i] + p*v[i-1];        /* rekursiv! */
}

for (i=0; i<n; i++) {             /* nicht ok */
   v[i] = w[i] + p*v[i+k];        /* rekursiv bei k<0 ! */
}
```

Im letzten Beispiel müsste man mit einer Direktive wie `ivdep` (s.u.) eingreifen, wenn man sicher weiß, dass der Fall $k < 0$ nicht eintritt.

6.5 Intel–Compiler

6.5.1 Dokumentation

```
file:///usr/global/intel/Documentation/en_US/documentation_c.htm
file:///usr/global/intel/Documentation/en_US/documentation_f.htm
```

6.5.2 Aufruf

```
icc   -fast -fno-alias -vec-report3
ifort -fast -vec-report3
```

Durch `-fast` wird die Auto–Vektorisierung aktiviert (das gilt auch schon bei der Default–Optimierung `-O bzw. -O2` !) und für die Hardware optimiert, auf der der Compiler aufgerufen wird. Dem ist in der Regel nichts hinzuzufügen. [Der Schalter `-fno-alias` bedeutet *no pointer aliasing* und signalisiert dem C–Compiler, dass man *nicht* von der Möglichkeit Gebrauch gemacht hat, über verschiedene Indizierungen auf denselben Speicherplatz zuzugreifen.]
Zugleich entsteht ein Vektorisierungs–Report (hier in der ausführlichsten Form). Er ist nicht leicht lesbar, man muss die angegebenen Zeilen im Quellprogramm suchen und dann kontrollieren, ob die erwartete Vektorisierung der inneren Schleifen gelungen ist.
Wenn das nicht automatisch funktioniert hat, versucht man es mit...

6.5.3 Direktiven

Die Auto–Vektorisierung kann man durch Direktiven beeinflussen, die man unmittelbar *vor* der betreffenden Schleife einfügt. Die wichtigsten sind

```
ivdep              "ignore vector dependencies"
vector always      "vectorize; ignore heuristics"
```

Syntax in Fortran und C:

```
!dec$ ivdep
!dec$ vector always

#pragma ivdep
#pragma vector always
```

Formal sind das Kommentare bzw. Preprozessor–Anweisungen, sie stören also nicht, wenn man das Programm "normal" übersetzt.
Mehr dazu in der Compiler–Dokumentation.
Bei neueren Versionen der Intel–Compiler sind Direktiven nur noch selten nötig.

6.6 Portland–Compiler

6.6.1 Dokumentation

Die Portland–Compiler erzeugen ebenfalls sehr schnellen Code, insbesondere auf AMD–Prozessoren koennen sie mit dem Intel–Compiler konkurrieren.

```
file:///usr/global/pgi/doc/index.htm
```

6.6.2 Aufruf

```
pgcc      -fast -Msafeptr -Minfo=vect
pgfortran -fast -Minfo=vect
```

6.6.3 Direktiven

..ähnlich wie Intel–Compiler...

6.7 GNU–Compiler

Die bei uns installierten GNU–Compiler (z. Zt. Version 4.7.2) sind, was die Performance des erzeugten Codes anbelangt, inzwischen ziemlich ausgereift (obwohl die Hardware–spezifische Optimierung in der Entwicklung immer etwas nachhinkt). Auf jeden Fall sind sie wertvolle Entwicklungswerkzeuge, denn man kann sie überall frei installieren.

6.7.1 Aufruf

```
gcc -O3 -ftree-vectorizer-verbose=5
gfortran -O3 -ftree-vectorizer-verbose=5
```

Wenn es um optimale Performance geht, sollte man bei `gfortran` auch die (neuen) Optionen testen:

```
gfortran -fstack-arrays ...
```

oder noch umfassender:

```
gfortran -Ofast ...
```

6.8 Vektorisierung des Metropolis–Updates

6.8.1 Erster Versuch

Neben dem Spinfeld (`phi_r[], phi_i[]`) legt man auch für die diversen Hilfsgrößen, die beim Update auftreten, jeweils Vektoren an und organisiert alle rechenaufwendigen Operationen als (innere) Schleifen über diesen Vektorindex:

```
double b_r[nvol], b_i[nvol], phi_new_r[nvol], phi_new_i[nvol];
double sq[nvol], rho[nvol], rho_new[nvol];
double r1[nvol], r2[nvol], r3[nvol];
...

/* Hintergrundfeld */
for (i=0; i<nvol; i++) {
  b_r[i] = hext;
  b_i[i] = 0.;
}
for (k=1; k<=2*ndim; k++){
  for (i=0; i<nvol; i++) {
     b_r[i] += kappa * phi_r[nn[k][i]];
     b_i[i] += kappa * phi_i[nn[k][i]];
  }
}

/* Metropolis hits */

for (i=0; i<nvol; i++) {
  sq[i] = phi_r[i]*phi_r[i] + phi_i[i]*phi_i[i];
  rho[i] = exp(2*(b_r[i]*phi_r[i] + b_i[i]*phi_i[i])
               - sq[i] - lambda*(sq[i] - 1.)*(sq[i] - 1.));
}

for (ihit = 0; ihit < nhits; ihit++) {
  dfrngv(nvol, r1);
  dfrngv(nvol, r2);
  dfrngv(nvol, r3);

  /* Vorschlag */
  ...
  /* Akzeptanzschritt */
  ...
}
```

Man beachte, dass die Definition des Indexfeldes `nn` in Fortran und C verschieden ist: `nn(i,k)` bzw. `nn[k][i]`, so dass der Punktindex `i` jeweils "am schnellsten" läuft und Zugriffe in der Vektorschleife an fortlaufende Speicherplätze (*stride 1*) gehen.

6.8.2 Probleme

- Es befinden sich Nachbarpunkte im Vektor, die beim Update voneinander abhängen – das vekorielle Update ist rekursiv.
- Die Vektorlänge ist gleich dem Volumen des Systems und damit zu groß für den Cache.

Insbesondere wegen des implizit rekursiven Charakters der lokalen Updates ist die obige Implementierung **falsch**!

Die Problem der Rekursion (Abhängigkeit) kann man am einfachsten lösen, indem man das Gitter wie ein Schachbrett zerlegt. Dabei hat jeder weiße (gerade) Punkt nur schwarze (ungerade) Nachbarn und umgekehrt. [Bei periodischen Randbedingungen müssen alle Gitterlängen *gerade* sein.] Wenn man nun die Gitterkonfiguration in *zwei* Vektoren (weißen/schwarz) aufteilt, kann man erst alle Spins des geraden Gitters vektoriell bearbeiten, wobei die Spins auf ungeraden Stellen den Hintergrund bilden, und dann umgekehrt.
Die rekursive Natur des lokalen Updates ist damit aufgelöst, aber die Vektoren umfassen jetzt das *halbe* Volumen und sind damit immer noch zu lang. Deshalb geht man einen Schritt weiter und zerlegt das Gitter in viele, relativ kurze Vektoren. Damit die Geometrie nicht zu kompliziert wird, gibt man sich eine regelmäßige Zerlegung des Gitters in gleiche **Zellen** vor und verteilt die Vektorkomponenten gleichmäßig über die so entstehende periodische Struktur.
[Nebenbei: Wir werden sehen, dass sich dieses Schema auch gut eignet zur *Parallelisierung* mit OpenMP.]

6.9 Vektorisierung über Zellen

Im folgenden wird das vorgeschlagene Indizierungssschema am Beispiel eines (6×9)–Gitters mit periodischen Randbedingungen und einer Zellengröße von (3×3) erläutert. Bei der Zerlegung entstehen 9 Vektoren der Länge 6.

.	.	.	.	.	.
.	.	.	.	.	.
.	.	.	.	.	.
.	.	.	.	.	.
.	.	.	.	.	.
.	.	.	.	.	.
6	7	8	.	.	.
3	4	5	.	.	.
0	1	2	.	.	.

.	.	.	.	.	.
.	u_4	v_4	.	u_5	v_5
.	.	.	.	.	.
.	.	.	.	.	.
.	u_2	v_2	.	u_3	v_3
.	.	.	.	.	.
.	.	.	.	.	.
.	u_0	v_0	.	u_1	v_1
.	.	.	.	.	.

.	w_4	.	.	w_5	.
.	.	.	.	.	.
.	z_4	.	.	z_5	.
.	w_2	.	.	w_3	.
.	.	.	.	.	.
.	z_2	.	.	z_3	.
.	w_0	.	.	w_1	.
.	.	.	.	.	.
.	z_0	.	.	z_1	.

Die linke Abbildung zeigt das Gitter mit der **Basiszelle** links unten. Dort hat jeder Vektor seinen **Fußpunkt**, und der eingetragene (Basis–)Index `ib = 0..8` charakterisiert den gesamten Vektor.
Im mittleren Bild sind zwei Vektoren eingetragen, der Übersichtlichkeit halber mit Namen statt Basisindizes: $u_i, i = 0..5$ mit `ib = 4` und $v_i, i = 0..5$ mit `ib = 5`. v_i ist der rechte Nachbar (Richtung `k = 1`) von u_i, eine Permutation der Indizes ist nicht nötig. Das beschreiben wir mit zwei neuen Indexfeldern `nnstep` und `nnflag` folgendermaßen:

```
nnstep[1][4] = 5
nnflag[1][4] = 0
```

Der Schritt von v_i zu u_i (Richtung -1) ist genauso einfach.
Anders im rechten Bild: der Schritt von w_i (`ib = 7`) nach oben (Richtung `k = 2`) führt zu z_j (`ib = 1`), aber mit permutierten Indizes:

```
nnstep[2][7] = 1
nnflag[2][7] = 2
nn[2][0] = 2
```

```
nn[2][1] = 3
nn[2][2] = 4
nn[2][3] = 5
nn[2][4] = 0
nn[2][5] = 1
```

denn der "obere" Nachbar von w_0 ist z_2, von w_4 ist z_0 etc.
Das letzte Beispiel lehrt zweierlei:

- Die Basiszelle hat *de facto* periodische Randbedingungen.
- *Wenn* eine Permutation des Vektors erforderlich ist, dann sieht das Indexfeld dazu genauso aus wie bei einem Gitter der Größe $(s_1 \times s_2 \times ..)$ mit periodischen Randbedingungen.
- Der Permutationsvektor hängt ggf. nur von der Richtung ab, nicht vom Basispunkt des Vektors: im Beispiel wurde der Schritt `ib = 7 -> 1` betrachtet, aber `ib = 6 -> 0` und `ib = 8 -> 2` sehen genauso aus, was die Vektorindizes betrifft.

Die Konstruktion der Geometriefelder kann also auf schon geläufige Techniken zurückgreifen.
Als erstes zerlegen wir das Gitter:

$$\begin{aligned} L_i &= l_i\, s_i \qquad i = 1..d \\ & \quad 2 \le l_i \le L_i \\ & \quad 1 \le s_i < L_i \end{aligned} \tag{6.1}$$

Wir haben eine Basiszelle der Grösse $(l_1 \times l_2 \times ..)$., darauf sind Vektoren definiert, die jeweils ein "ausgedünntes" Gitter (Untergitter) der Größe $(s_1 \times s_2 \times ...)$ bilden. Die Geometrie wird in zwei Schichten erledigt: man muss die Nachbarn der Basispunkte finden können, und entlang der Vektoren kann es nötig sein, die Indizes zu permutieren – nämlich dann, wenn der Schritt zum Nachbarn in der Basiszelle von Rand zu Rand springt.
Das vorbereitete Programm `geom_vec.c` erledigt das. Hier der Anfang mit der Beschreibung der Geometriefelder:

```
void geom_vec() {
  /*
  Angelegt und besetzt ist                                  B Bunk 6/2010
      Dimension     ndim                                    rev    6/2011
      Gittergroesse lsize[k], k=1..ndim

  Angelegt und berechnet wird
      Volumen                     nvol

      Volumen der Basiszelle      nvcell
      NN-Schritt fuer Basispunkt nnstep[k][ib], ib=0..(nvcell-1), k=0..2*ndim
      Permutationsflag  dazu      nnflag[k][ib], ib=0..(nvcell-1), k=0..2*ndim

      Vektorlaenge                lvec
      NN-Indexfeld im Vektor      nn[k][iv], iv=0..(lvec-1), k=0..2*ndim
```

```
  nnstep[k][ib] gibt den Nachbarn des Basispunkts ib bei einem Schritt in
      Richtung +k.
      Fuer einen Schritt in Richtung -k setze man den Index auf (ndim+k).
      nnstep[0][ib] ist reserviert.
  nnflag[k][ib] gibt die zugehoerige Permutationsnummer:
      nnflag[k][ib] = k, (k=1..2*ndim), wenn der Schritt in der Basiszelle
                           ueber den Rand springt;
      nnflag[k][ib] = 0 sonst.
      nnflag[0][ib] = 0,1 gibt die Farbe von ib (Schachbrettmuster).

  nn[k][iv] gibt, ausgehend vom Vektorindex iv, den Index im Nachbarvektor
      bei einer Verschiebung (des gesamten Vektors) in Richtung +k.
      Fuer eine Verschiebung in Richtung -k setze man den Index auf (ndim+k).
      nn[0][iv] = iv (keine Permutation).
  */
  ...
}
```

Das Programm berechnet zunächst die Zerlegung (6.1) nach folgendem Schema: es versucht $s_i = 10, 9, 8, ..., 1$, so dass $L_i = l_i s_i$ mit *geradem* l_i. In $d = 2, 3$ Dimensionen entsteht so (meist) eine Vektorlänge $\prod_i s_i = \mathcal{O}(100)$. Ein Double–Vektor braucht dann ca. 1kB, bei einem Cache von einigen MB ist das in Ordnung.
Dann werden die Geometriefelder berechnet. Zu bemerken ist, dass der Richtungsindex `k` immer vorne steht und der Laufindex hinten, wie es sich für C–Felder gehört.
Die Fortran–Version sieht ähnlich aus, nur sind dort wieder die Richtungen mit $\pm1, \pm2, ..$ bezeichnet, die Laufindizes stehen vorne und beginnen bei 1.
In Anwendungen operiert man immer auf ganzen Vektoren, d.h. die innerste Schleife geht über den Vektorindex. Weiter außen gibt es eine Schleife über die Basispunkte, so überstreicht man das gesamte Gitter. Die Berechnung des Hintergrunds $B(x)$, mit dem das Metropolis–Update beginnt, könnte also etwa so aussehen:

```
double b_r[lvec], b_i[lvec];     /* B-Vektor fuer Hintergrundfeld */

for (ib=0; ib<nvcell; ib++) {    /* Schleife ueber Basispunkte */

   for (i=0; i<lvec; i++) {      /* VEKTORSCHLEIFE */
      b_r[i] = hext;             /* externes Magnetfeld */
      b_i[i] = 0.;
   }
   for (k=1; k<=2*ndim; k++){
      jb = nnstep[k][ib];              /* Basis des Nachbarvektors */
      if (nnflag[k][ib]) {             /* Permutation noetig? */
         for (i=0; i<lvec; i++) {      /* VEKTORSCHLEIFE mit Perm. */
            b_r[i] += kappa * phi_r[jb][nn[k][i]];
            b_i[i] += kappa * phi_i[jb][nn[k][i]];
         }
      }
      else {
```

```
          for (i=0; i<lvec; i++) {      /* VEKTORSCHLEIFE ohne Perm. */
             b_r[i] += kappa * phi_r[jb][i];
             b_i[i] += kappa * phi_i[jb][i];
          }
       }
    }
    /* Metropolis hits ... */
 }                            /* Ende der Schleife ueber Basispunkte */
```

Und dasselbe in Fortran:

```
real(8), dimension(lvec) :: b_r, b_i

do ib=1,nvcell        ! Schleife ueber Basispunkte

   ! Hintergrundfeld

   b_r = hext
   b_i = 0.

   do k=-ndim,ndim
      if (k == 0) cycle
      jb = nnstep(ib,k)                       ! Basis des Nachbarvektors
      if (nnflag(ib,k) == 0) then             ! Permutation noetig?
         b_r = b_r + kappa * phi_r(:,jb)      ! Vektorschleife ohne Perm.
         b_i = b_i + kappa * phi_i(:,jb)
      else
         b_r = b_r + kappa * phi_r(nn(:,k),jb) ! Vektorschleife mit Perm.
         b_i = b_i + kappa * phi_i(nn(:,k),jb)
      endif
   enddo

   ! Metropolis hits ...

enddo                  ! Ende der Schleife ueber Basispunkte
```

Das Spinfeld hat jetzt zwei Indizes (`phi_r[ib][i]` bzw. `phi_r(i,ib)`): den Basisindex des Vektors und den Vektorindex. Letzterer steht in C hinten, in Fortran aber vorne.
Außerdem braucht man noch einige Hilfsvektoren (hier `b_r[lvec]` und `b_i[lvec]`), die sich aber immer nur über die Vektorlänge `lvec` erstrecken, nicht über das ganze Gittervolumen. Deshalb nehmen sie nicht so viel Speicherplatz ein, und man kann ohne Bedenken eine größere Zahl davon anlegen.
Auch die Geometriefelder sind jetzt nicht mehr so groß wie im skalaren Fall: der Laufindex geht jeweils nur über das Zellvolumen *oder* über die Vektorlänge.
Letztlich erstreckt sich nur das Spinfeld über das ganze Gittervolumen, alle anderen Felder sind deutlich kleiner. Die "ausgedünnte" Vektorisierung über die Untergitter entlang der Zellen spart also nebenbei auch noch Speicherplatz.

6.10 Dynamische Arbeitsvektoren in C

Wenn die Vektorlänge `n` zur *Laufzeit* bekannt ist, kann man einen Arbeitsvektor dynamisch anlegen:

```
/* ANSI C / C90: */
   double *s;
   s = (double *) malloc(n * sizeof(double));
   /* work... */
   free(s);

/* C99: */
   double s[n];
   /* work... */
```

Die **zweite** Möglichkeit ist vorzuziehen, denn damit wird der Arbeitsvektor auf dem Stack angelegt, und der Zugriff ist schnell. Das geht aber *nicht* für *global* deklarierte Felder.