18 Effiziente Programmierung

- Etwas Rechnerarchitektur
- Befehlsparallelität erhöhen
- Cacheslokalität verbessern
- Compileroptimierungen erleichtern
- Werkzeuge

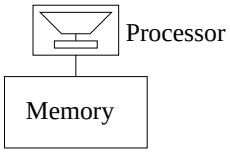
Literatur: Theo Ungerer, Mikroprozessortechnik
J. L. Hennessy und D. A. Patterson, Computer Architecture – a Quantitative Approach.

18.1 Maschinenmodell

18.2 DAS traditionelle Modell:

Random **A**ccess **M**achine

John von Neumann, First Draft of a Report on the EDVAC, 1945 (Techn. Ber. d. University of Pennsylvania)



18.2.1 Modellparameter

- "genug" Speicherzellen
- Festgelegte Anzahl Taktzyklen pro Operation (Laden, Speichern, addieren, ...)

\+ Einfach

\+ Andere einfache Modelle (Turingmachinen, Grammatiken, ... vermutlich ungeeignet für Effizientbetrachtungen

18.2.2 Algorithmenanalyse:

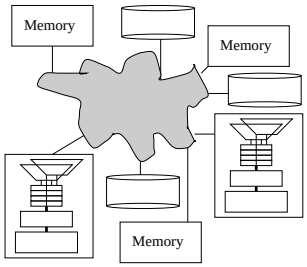
- Zähle Befehle
- $\mathcal{O}(\cdot)$ Notation unterdrückt maschinenabhängige Konstanten
- Dann reicht es manchmal, Schlüsseloperationen wie Vergleiche, oder Gleitkomma-Multiplikationen zu zählen.

18.2.3 Einfache RISC-Maschinen

- Es gibt 32–64 universell **Register**
- Alle **Rechenoperationen** haben die Form `op` Quellregister1 Quellregister2 Zielregister. (`add`, `sub`, `shift`, `and`, `not` (?), ...
- Zusätzlich gibt es die Befehle `load` Adressregister, Zielregister und `store` Adressregister, Datenregister
- (Bedingte) Sprünge

Logisch entspricht das fast dem RAM-Modell.

18.3 Die grausame Realität



- Pipelining und Befehlsparallelismus: Befehlsausführungszeiten nicht unabhängig
- Was macht der Compiler aus meinem Code?
- Register, Caches, Haupspeicher, Platten, Bänder: unterschiedliche Zugriffszeiten, z.T. implizite Datenmigration
- logische, virtuelle und physikalische Speicheradressen müssen umgesetzt werden
- Mehrere Prozessoren und Speicher
- Komplizierte Verbindungen (Busse, Cluster, LAN, MAN, WAN, ...)

18.4 Pipelining

- Die Abarbeitung eines Maschinenbefehls wie `add` erfolgt in 5–10 Schritten (je nach Prozessor). (Preisfrage: Welche Proz. brauchen die mesten Schritte?)
- Für jeden Schritt gibt es eine eigene Bearbeitungstation.
- Die Stationen sind zu einem Fließband angeordnet.

⇒ In jedem Taktzyklus kann das Fließband einen neuen Befehl in Angriff nehmen

18.5 Befehlsparallelismus

- Es gibt mehrere sagen wir vier parallele Pipelines.

⇒ In jedem Taktzyklus können bis zu vier Befehle auf die Reise geschickt werden

⇒ Es können mehrere dutzend Befehle gleichzeitig in Bearbeitung sein.

18.6 Beispiel: Skalarprodukt

```
double scalarProduct0(const double * const a,
                      const double * const b, int n)
{ double accu = 0;
  for (int i = 0;  i < n;  i++) {
    accu = accu + a[i] * b[i];
  }
  return accu;
}

double scalarProduct1(double *a,
                      double *b, int n)
{ double accu = 0;

  double *beyond = a + n;
  for (beyond=a+n;  a<beyond;  a++,b++) {
    accu += *a + *b;
  }
  return accu;
}
```

18.6.1 Einfache Compilerspielereien

Ohne Optimierung, $N = 1024$

```
\>CC sprod.C
\>a.out
scalarProduct0 17.2101MFlop/s
scalarProduct1 20.898MFlop/s
```

Hackerversion besser?

```
\>CC -fast sprod.C
\>a.out
scalarProduct0 InfMFlop/s
```

OOps was war das?


```
>>> einfuegen >>>>>>
  // fool optimizer
  if (result < 0) { cout << result; }
>>>>>>>>>>>>>>>>>>>>
\>CC -fast -O4 sprod.C
\>a.out
scalarProduct0 186.182MFlop/s
scalarProduct1 97.5238MFlop/s
```

Hackerversion schlechter!
Optimierung "bringts"

PD Compiler reichen?

```
\>g++ -O6 sprod.C
\>a.out
scalarProduct0 85.3333MFlop/s
scalarProduct1 75.8519MFlop/s
```

Jedenfalls nicht immer.

18.7 Sentinels

Eine Methode zur Vermeidung zusätzlicher `if`s.

Ein spezieller Wert wird an das „Ende“ einer Datenstruktur plaziert und verhindert implizit, daß eine Traversierung der Daten über das Ende hinausschießt.

Beispiel: Suche in einem Array: Plaziere gesuchtes Element am Ende.

- Hack-Alert! Wirklich nötig? Kenne ich einen passenden Wert? Darf ich implantieren?
- Sorgfältig dokumentieren und durch Assertions absichern.
- Kann den Code manchmal vereinfachen und damit Fehlerquellen ausschalten
- Spart nur leicht vorhersagbare Sprünge

18.7.1 Beispiel Merge

```
//Pre: f0 und f1 enthalten mindestens sz Werte
//  kleiner dem Wert SENTINEL und
//  sind durch einen Wert >= SENTINEL beendet
void merge(double **f0, double **f1,
           double *to, int sz)
{ double *from0=*f0;  double *from1=*f1;
  double key0=*from0; double key1=*from1;
  double *done    = to + sz;
  while (to < done) {
    if (key1 <= key0) {
      *to   = key1; from1++; key1 = *from1;
    } else {
      *to   = key0; from0++; key0 = *from0;
    }
    Assert(*to < SENTINEL);
    to++;
  }
  *f0   = from0;
  *f1   = from1;
}
```

Diskussion

- Nur ein schwer vorhersagbares `if` (unvermeidlich)
- Endetest könnte duch Schleifenausrollen billiger gemacht werden (Erfolg zweifelhaft)
- Sehr Cache-freundlich (Im Gegensatz zu merge auf Listen)
- Arrays brauchen weniger Speicher als Listen
- Lokale Variablen `from0/1, done, key0/1` erleichtern dem Optimierer seine Aufgabe.

18.8 Speicherhierarchie

Ein (typischer?) 4-fach superskalarer 500MHz RISC-Prozessor.

	Verzögerung [s]	Bandbreite [Byte/s]	Kapazität [Byte]
Register	$2 \cdot 10^{-9}$	$64 \cdot 10^{9}$	512
L-1 Cache	$4 \cdot 10^{-9}$	$8 \cdot 10^{9}$	2 x 8–32K
L-2 Cache	$20 \cdot 10^{-9}$	$2 \cdot 10^{9}$	512K–4M
DRAM	$400 \cdot 10^{-9}$	$500 \cdot 10^{6}$	64M – 1G

18.9 Was ist ein Cache?

- Speicherstellen werden in **Cache lines** zu je B Bytes zusammengefäßt (z.Zt $B \in 16..256$)
- Ein Cache der Größe M speichert eine Teilmenge von M/B Cache Lines
- Die *Hardware* entscheidet welche Teilmenge das ist

18.9.1 Lesen (`load` R_i, A)

(Wir vernachlässigen Addressübersetzungen)

$L := \lfloor \frac{A}{B} \rfloor$ (* Welche Cache Line? *)
if Line $L \in$ Cache **then** (* Cache Hit *)
  Lese aus dem Cache (Wort A mod B von Line L)
else
  Schaffe Platz im Cache
  Bringe Line L in den Cache
  liefere Wort A mod B von Line L
end
Schreibe Ergebnis in Register R_i

18.9.2 Schreiben (`store` R_i, A)

Write-Back Schreibe nur in Cache (und verdrängte Cache Lines)

Write-Through Schreibe sowohl in Cache als auch in Hauptspeicher

18.9.3 Cache-Ersetzungstrategien

Direct Mapped

`CacheLine l1-Cache[M/B]`

Cache-Line L ist immer im Eintrag L mod M/B gespeichert.

$\Rightarrow$ häufige **Interference Misses**. Beispiel: Array a beginnt an Adresse $5 \cdot 1024B$, Array b beginnt an Adresse $42 \cdot 1024B$, beide sind $n \ll M$ Bytes gross. Trotzdem wird `scalarProduct(a,b)` fast $2n$ Cache misses produzieren.

18.9.4 Hypothetischer Fully Associative Cache

Verdränge *least recently used* cache line

+ Robuster, Weniger Cache faults
– Braucht viel Platz (d.h. teuer)
– Selbst Hardwarerealisierung ist langsamer
– Hoher Energieverbrauch
– Nur für sehr kleine Caches realisierbar

18.9.5 In der Praxis

Irgendwas zwischen direct mapped and fully associative (oft sehr nah an direct mapped)

18.9.6 Speicherbandbreitenbegrenzung

$N = 2^{20}$

```
\>a.out
scalarProduct0 24.8686MFlop/s
scalarProduct1 19.3422MFlop/s
scalarProduct2 24.8686MFlop/s
```


18.9.7 Interference Misses

$N = 2^{17}$

```
scalarProduct0 16.8041MFlop/s
scalarProduct1 14.247MFlop/s
scalarProduct2 17.9551MFlop/s
```

Hier: Hinweis auf eine tumben direct mapped Cache.

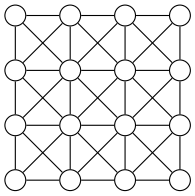
18.9.8 Padding

```
double a[N+64*13];
double b[N];
scalarProduct0 28.4939MFlop/s
scalarProduct1 23.6166MFlop/s
scalarProduct2 25.4509MFlop/s
```

18.10 Beispiel: Zellularautomat

zweidimensional, zwei Zustände pro Sekunde

- Zweidimensional
- Zwei Zustände pro Zelle Q
- Nachbarschaft N: Eine Zelle und ihre 8 Nachbarn in horizontaler und diagonaler Richtung. $(c, n, o, s, w, no, so, sw, nw)$
- Überführungsfunktion $\delta : Q^{|N|} \to Q$.



Arbeitsweise

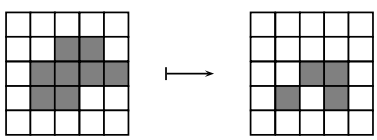
- synchron agierende
- identische
- kommunizierende
- endliche Automaten

In einem Zeitschritt führen alle endlichen Automaten synchron einen Zustandsübergang aus.

18.10.1 Eine (Fast-)Majoritätsregel

- $Q = \{0, 1\}$
- $\delta(c, n, o, s, w, no, so, sw, nw) =$
  $f(c + n + o + s + w + no + so + sw + nw)$

n	0	1	2	3	4	5	6	7	8	9
$f(n)$	0	0	0	0	1	0	1	1	1	1



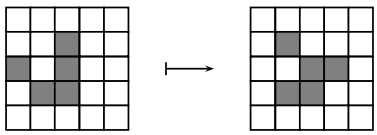
Ein Modell für "Entmischung"

Huebsche Bilder die der Drucker nicht mag...

LIFE

$$\delta(q_{\mathbf{c}}, q_{\mathbf{n}}, q_{\mathbf{o}}, q_{\mathbf{s}}, q_{\mathbf{w}}, q_{\mathbf{no}}, q_{\mathbf{so}}, q_{\mathbf{sw}}, q_{\mathbf{nw}})$$
$$= g(q_{\mathbf{c}}, q_{\mathbf{n}} + q_{\mathbf{o}} + q_{\mathbf{s}} + q_{\mathbf{w}} + q_{\mathbf{no}} + q_{\mathbf{so}} + q_{\mathbf{sw}} + q_{\mathbf{nw}}) \ .$$

n	0	1	2	3	4	5	6	7	8
$g(0,n)$	0	0	0	1	0	0	0	0	0
$g(1,n)$	0	0	1	1	0	0	0	0	0



Simulation

18.10.2 Eine einfache Version

caseq-slow.c

```
/* simulate a cellular automaton
 * periodic boundaries
 *
 * uses graphical display
 *
 * command line arguments:
 *
 * #1: number of rows of configurations;
 * #2: number of iterations to be simulated;
 * #3: length of period (in iterations) between
 *     displaying configurations;
 *     0 means that no states are displayed;
 * #4: name of X Display; e.g. "mpii01666:0.0";
 */
```


Datentypen

```
#define CONFIGCOLS 256

typedef char State;
typedef State Row[CONFIGCOLS + 2];
```

Idee: Flexibleres C++ Programm? Effizienz?

Überführungsfunktion

```
/* transition table for the annealing rule */
static State anneal[10] =
   {0, 0, 0, 0, 1, 0, 1, 1, 1, 1};

/*  compute the new state of a cell */
#define transition(a, x, y)
   (anneal[(a)[(y)-1][(x)-1] + \
           (a)[(y)  ][(x)-1] + \
           (a)[(y)+1][(x)-1] + \
           (a)[(y)-1][(x)  ] + \
           (a)[(y)  ][(x)  ] + \
           (a)[(y)+1][(x)  ] + \
           (a)[(y)-1][(x)+1] + \
           (a)[(y)  ][(x)+1] + \
           (a)[(y)+1][(x)+1]   \
          ])
```

Idee: C++ mit `inline` Funktion.

Simulation

```
/*  simulate one step of the CA
 *  IN:
 *  from:       pointer to old configuration
 *  to:         pointer to new configuration
 *  configRows: number of rows of configurations
 *  OUT:
 *  to:         the memory is modified
 */
static void
simulate(Row *from, Row *to, int configRows)
{
   int x,y;

   boundary(from, configRows);
   for (y = 1;  y <= configRows;  y++) {
      for (x = 1;  x <= CONFIGCOLS;  x++) {
         to[y][x  ] = transition(from, x  , y);
      }
   }
}
```

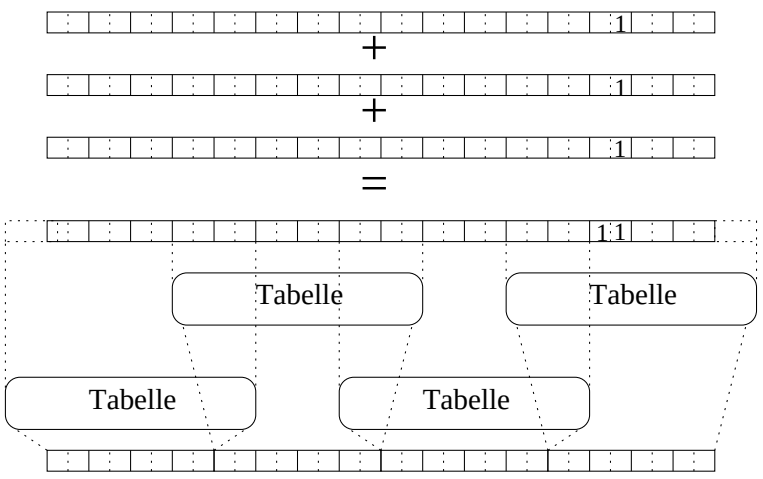

Periodische Randbedingungen

```
/*  treat boundary conditions
 *  IN:
 *  buf:       pointer to a stored configuration
 *  configRows:number of rows of configurations
 *  OUT:
 *  buf:       configuration is modified in plac
 */
static void boundary(Row *buf, int configRows)
{ int x,y;

  for (y = 1;  y <= configRows;  y++) {
    buf[y][0           ] = buf[y][CONFIGCOLS];
    buf[y][CONFIGCOLS+1] = buf[y][1         ];
  }

  for (x = 1;  x <= CONFIGCOLS;  x++) {
    buf[0           ][x] = buf[configRows][x];
    buf[configRows+1][x] = buf[1         ][x];
  }
}
```


18.10.3 Bitparallele Simulation



Vergleich

- Parallele Addition für 16 Zellen
- Paralleler Lookup für 4 Zellen
- 4-fach geringerer Speicherbedarf (Cache?)
- 4-fach mehr Zellen pro Cache-miss
- 16-fach mehr Zellen pro Load- (Store-) Befehl
- Pentium-II-300: 20M Zellen/s $\Longrightarrow$ 160M Zellen/s (512×512 Zellen)

Umsetzung

```
static void simulate(Line *from, Line *to, int lines)
{ int x, y;
  unsigned int l, m, r;
  boundary(from, lines);
  for (y = 1;  y <= lines;  y++) {
    l = from[y-1][0] + from[y][0] + from[y+1][0];
    m = from[y-1][1] + from[y][1] + from[y+1][1];

    for (x = 1;  x <= XSIZE; x++) {
      r=from[y-1][x+1]+from[y][x+1]+from[y+1][x+1];
      to[y][x] =
        (transition[((l& 3u        )<<10)|
                    ((m&(1023u<<22))>>22) ]<<24)|
        (transition[((m&(4095u<<14))>>14) ]<<16)|
        (transition[((m&(4095u<< 6))>> 6) ]<< 8)|
         transition[((m& 1023u     )<< 2)|
                    ((r&(   3u<<30))>>30)  ];
      l = m; m = r;
    }
  }
}
```


Weitere Maßnahmen:

- 64 Bit
- Paralleler Lookup für 8 Zellen. Wirklich?
- 1 Bit pro Zustand
  - $-$ δ als boolesche Funktion `& | ^ ~`. (LIFE)
  - $+$ 2–16 mal mehr Parallelismus
  - $+$ Keine Tabellenzugriffe
  - $-$ Viele Instruktionen
- Ausrichtung nach Cache-Line-Grenzen
- Prefetching, Zeilenpuffer
- MMX, Grafikhardware?
- Einfachere Behandlung stationärer Bereiche. Datenstrukturen ...