Ist Effizienz nötig?

- Manche Software ist schnell genug
- Andere noch immer nicht
- Häufigere Aufrufe, andere Arbeitsabläufe
- Größere Eingaben
- Bessere Funktionalität
- Energie sparen

Arten von Effizienz

Laufzeit

- CPU
- Festplatte
- Netzwerk
- andere I/O

Speicher

- RAM
- ROM
- Platte
- Externer Speicher

Kosten von Ineffizienz

- Zeitverlust beim Benutzer
- Andere Arbeitsabläufe
- Unbrauchbarkeit bei Echtzeitanwendungen
- Teurere Hardware
- Energie

Wieviel Effizienz ist sinnvoll?

- Befehl-Antwort-Interaktion: 300ms
- Musik: 20ms
- Animierte Software: Bildwiederholrate (7-16ms).
- Andere Komponente dominiert
- Kommerzielle Überlegungen

Andere Ziele

- Korrektheit
- Klarheit, Einfachkeit
- Entwicklungsaufwand
- Wartungsaufwand
- Time-to-market
- Security

Extrempositionen

- Keine Effizienz-Überlegungen
- Wir optimieren alles

Beobachtungen

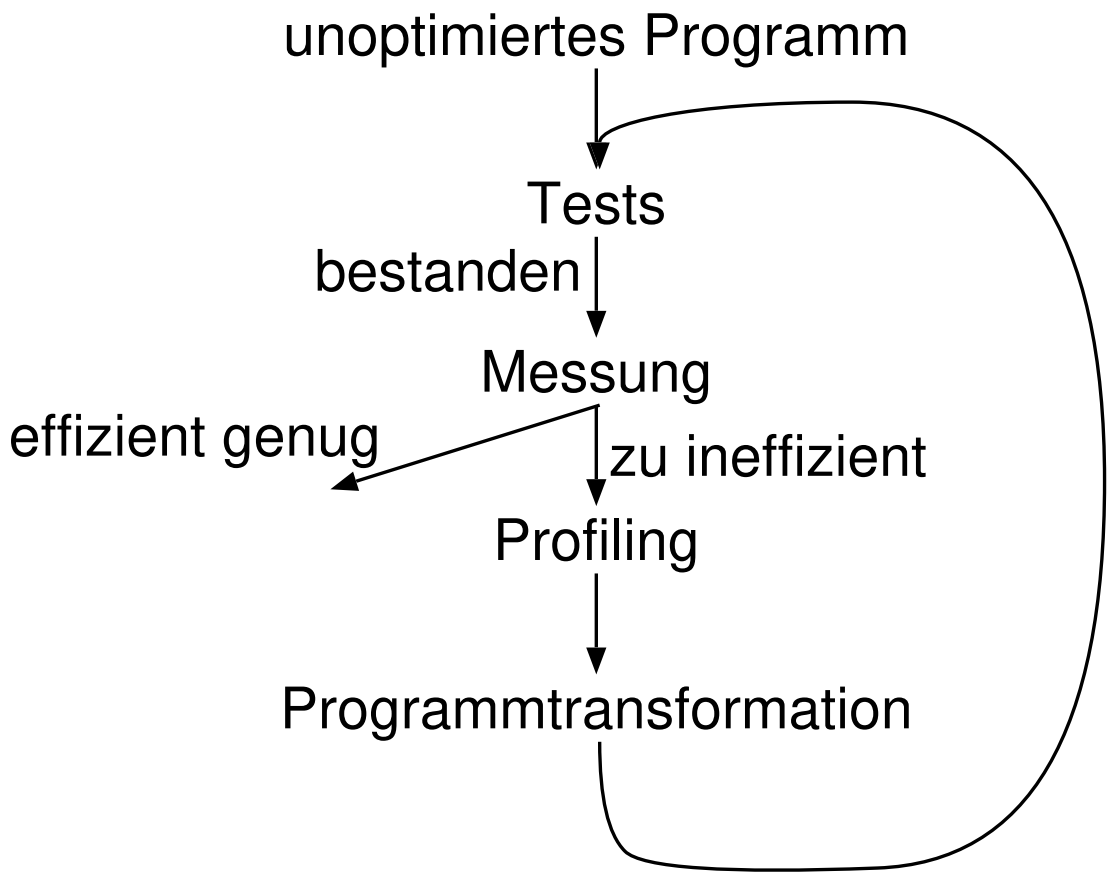
- 80-20 Regel
- Vorhersage von Hot Spots unzuverlässig

Allgemeiner Ansatz

- Zunächst Einfachkeit, Flexibilität, Wartbarkeit
- Messen
- Kritische Teile optimieren

Problem: Effizienzprobleme in Spezifikation und Design

Methode


unoptimiertes Programm
Tests
bestanden
Messung
effizient genug
zu ineffizient
Profiling
Programmtransformation

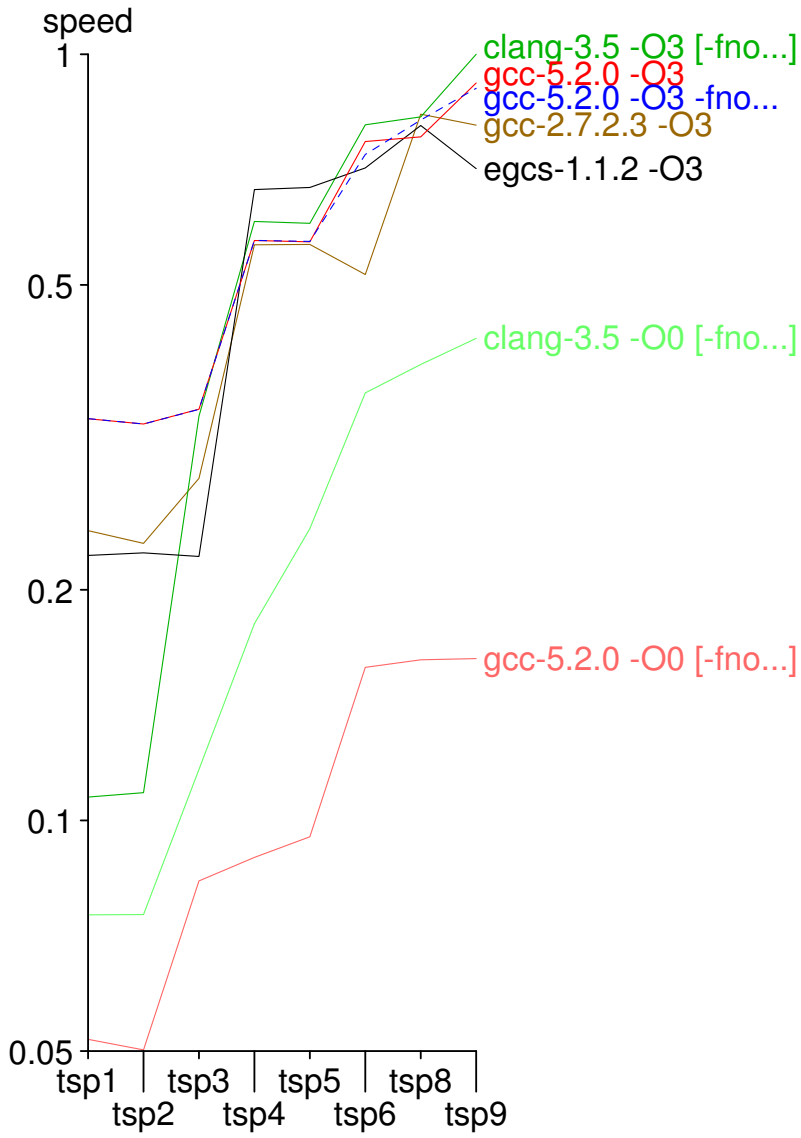
Warum macht das nicht der Compiler?

Auch Compiler verwenden Programmtransformationen, aber

- muss sich an die Semantik der Sprache halten
- vermeidet potentielle Pessimierungen
- probiert nur Dinge, die schnell und mit wenig Speicher gehen
- kann nur Optimierungen, die relativ häufig gebraucht werden
- Abhängigkeit der Optimierungen voneinander

```
*s1==*s2 && *s1!=0 && *s2!=0
```

Optimierung: Compiler vs. Quellcode



Beispiel: Stolpersteine für Compiler

```
for (i=0, best=0; i<n; i++)
  if (a[i]<a[best])
    best=i;
return best;

for (p=a, bestp=a, endp=a+n; p<endp; p++)
  if (*p < *bestp)
    bestp = p;
return bestp-a;

for (i=0, bestp=a; a+i<a+n; i++)
  if (a[i]<*bestp)
    bestp=a+i;
return bestp-a;
```

Typische Stolpersteine für Compiler

- Aliasing

```
*p = ...                    for (i=0; i<n; i++)
... = *q;                     a[i] = a[i]*b[j];
```

- Seiteneffekte, Exceptions

```
if (flag)                   for (i=0; i<n; i++)
  printf(...)                 a[i] = a[i]+1/b[j];
```

Hardware-Eigenschaften

1Z	2–8 unabhängige Befehle
1Z	Latenzzeit eines ALU-Befehls
3–5Z	Latenzzeit eines Load-Befehls (L1-Hit)
14Z	Latenzzeit eines Load-Befehls (L1-Miss, L2-Hit)
50Z	Latenzzeit eines Load-Befehls (L2-Miss, L3-Hit)
50–ns	Latenzzeit eines Load-befehls (L3-Miss, Main Memory access)
3ns	Übertragungszeit einer Cacheline (64B) vom/zu DDR4-2666, DDR5-5200
0–1Z	korrekt vorhergesagter Sprung
20Z	falsch vorhergesagter Sprung (branch misprediction)
4Z	Latenzzeit Integer-Multiplikation
4Z	Latenzzeit FP-Addition/Multiplikation
30–90Z	Latenzzeit Division
100us	IP-Ping über Ethernet
10us	1KB Übertragung über Gb Ethernet
10ms	Latenzzeit Plattenzugriff (seek+rotational delay)
10ms	2500KB sequentieller Plattenzugriff (ohne delay)

Hardware-Eigenschaften: Latenz

```
while (i<n) {
  r+=a[i];
  i++;
}
```

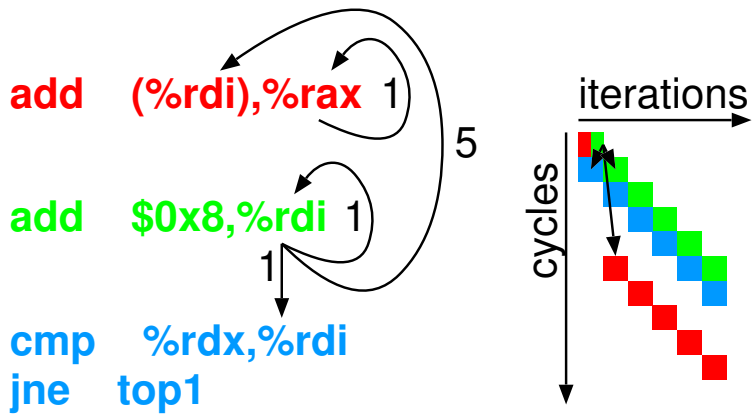
```
add   (%rdi),%rax

add   $0x8,%rdi

cmp   %rdx,%rdi
jne   top1
```

```
while (a!=0) {
  r += a->val;
  a = a->next;
}
```

```
add   0x8(%rdi),%rax

mov   (%rdi),%rdi

test  %rdi,%rdi
jne   top2
```

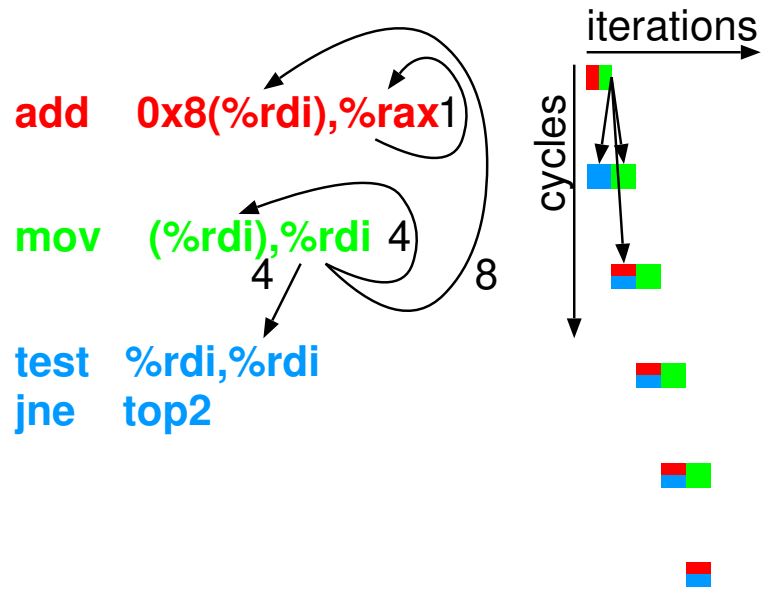
Hardware-Eigenschaften: Latenz

```
while (i<n) {
  r+=a[i];
  i++;
}
```



Skylake: 1.29Z/Iteration

```
while (a!=0) {
  r += a->val;
  a = a->next;
}
```



Skylake: 4Z/iteration

Programm-Eigenschaften: Latenz vs. Durchsatz

```
// double a[], r;
while (i<n) {
  r+=a[i];
  i++;
}
```

Skylake: 4Z/Iteration

```
// double a[], f;
while (i<n) {
  a[i]=a[i]+f;
  i++;
}
```

Skylake: 1.37Z/iteration

mit Vektorisierung
gcc -O3 -mavx:
Skylake: 0.45Z/iteration

Programm-Eigenschaften

Latenz-dominiert

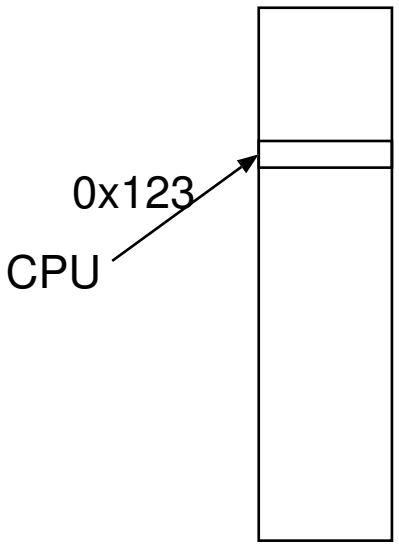
- Abhängige Operationen
  auf den selben Daten
- Daten oft im Cache
- Großteil des Codes
- Hilfreich:
  OoO, Branch Prediction, Caches
- manchmal unabhängige Instanzen
  z.B. Compiler, On-Line-Systeme
  Hilfreich: Multi-Core

Durchsatz-dominiert

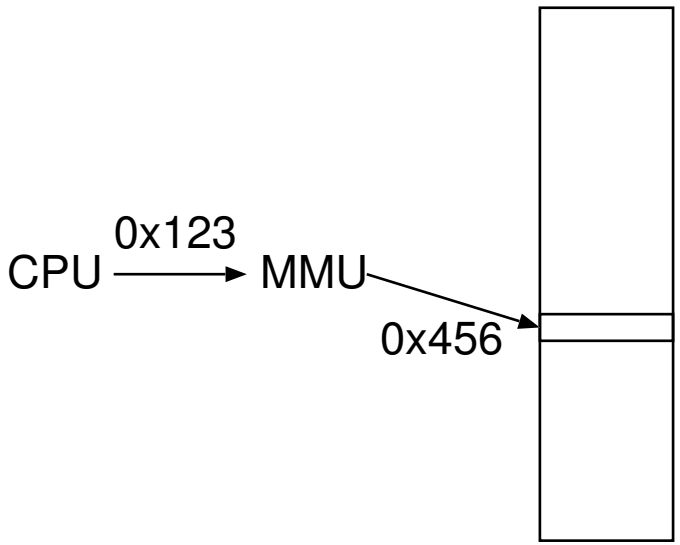
- Gleiche Operationen auf vielen Daten
  z.B. Bilder, Audio, Grafik,
  neuronale Netze, Matrizen
- Braucht oft Hauptspeicherbandbreite
- Relativ wenig Code
  aber viel Laufzeit
- Hilfreich: SIMD, multi-cores, GPUs

Hardware-Eigenschaften: Speicher/Cache

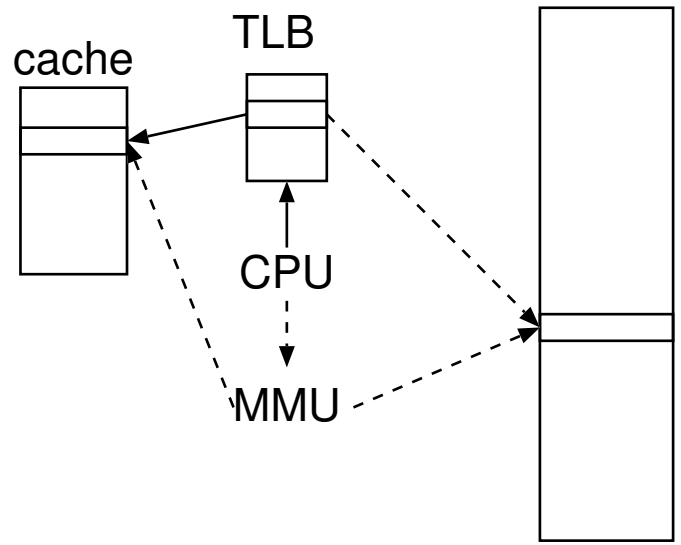
Einfache Ansicht



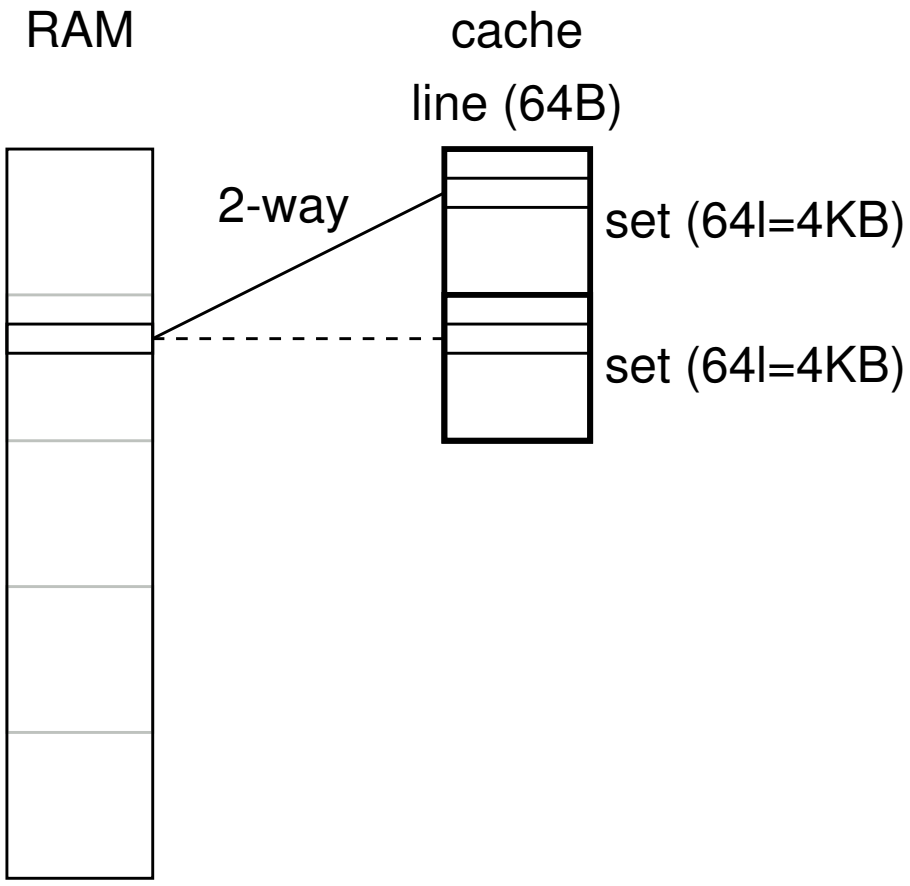
Virtueller Speicher (VM)



Performance



Hardware-Eigenschaften: Speicher/Cache



- temporal locality (Programmeigenschaft)
  spatial locality (Programmeigenschaft)
- compulsory misses (Programmeigenschaft)
  capacity misses
  conflict misses
- Intel Skylake (Core ix-6xxx):
  data cache (L1): 32KB, 64B/line, 8-way, 4c
  instruction cache (L1): 32KB, 64B/line, 8-way
  L2 cache: 256KB, 64B/line, 4-way, 12c
  L3 cache: 2-8MB, 64B/line, 4-16-way, ≥ 42c
  RAM: ≈ 50ns
  DTLB L1: 64 entries (4KB), 4-way
  DTLB L1: 32 entries (2MB), 4-way
  DTLB L2: 1536 e. (4KB, 2MB), 12-way, 9c

Datenstrukturen und Algorithmen

- Effiziente Implementierung eines ineffizienten Algorithmus ist Zeitverschwendung
- Effiziente Implementierung eines effizienten Algorithmus
- Ziele: Einfachkeit, Effizienz, Flexibilität
- Problem: Einschätzung im vorhinein
- Datenstruktur zieht sich durch große Teile
- Abstrakte Datentypen

Algorithmische Komplexität

- Betrachtet oft den worst case
- Zählt bestimmte Operationen, nicht immer relevant für die Laufzeit
- Ignoriert konstante Faktoren
- logarithmische Faktoren
- Beispiel: Suche nach Substring (Länge m) in String (Länge n)
  einfacher Algorithmus: $\mathrm{O}(mn)$ (worst), $\mathrm{O}(n)$ (best)
  KMP: $\mathrm{O}(n)$, aber meist langsamer als der einfache
  BM: $\mathrm{O}(n)$ (worst), $\mathrm{O}(n/m)$ (best)
- Quicksort: $\mathrm{O}(n^2)$ (worst), $\mathrm{O}(n \ln n)$ (typ), räumliche und zeitliche Lokalität
  Heapsort: $\mathrm{O}(n \ln n)$, schlechte Lokalität
  Mergesort: $\mathrm{O}(n \ln n)$, gute Lokalität

Wie spezifizieren: Speicherblock kopieren

	`cmove` (Forth) `rep movsb` (AMD64)	`memcpy()` (C)	`memmove()` (C) `move` (Forth)
keine Überlappung	Quelle → Ziel	Quelle → Ziel	Quelle → Ziel
Zielanfang in Quelle	Musterreplikation	undefiniert	Quelle → Ziel
Quellenanfang in Ziel	Quelle → Ziel	undefiniert	Quelle → Ziel
Implementierung	byteweise vorwärts	grössere Einheiten	Unterscheidung
Alternative	Unterscheidung		
	überspezifiziert	unterspezifiziert	gut

Programmiersprachen

- Eingebaute Ineffizienz
- Idiomatische Ineffizienz
- Effizienz durch Compiler
- Effizienz durch Programmiereffizienz
- Assembler?

Programmiersprachen: Beispiele

- Aliasing: C vs. Fortran (eingebaut)

```
void f(double a[], double b[], double c[], long n) {
  for (long i=0; i<n; i++)
    c[i]=a[i]+b[i];
}
```

Programmiersprachen: Beispiele

- Verschachtelte Objekte: Java vs. C(++) (eingebaut)

```
struct mystruct { int a; float b; double c; }
struct mystruct a[10000];
struct mystruct *b[10000];
```

- Skalieren bei Zeigerarithmetik: C vs. Forth (eingebaut/idiomatisch)

```
mystruct *p = a+i;      a i cells + constant p
mystruct *q = a+j;      a j cells + constant q
...                     ...
long d = q-p;           q p - constant d1
mystruct *r = p+d;      p d1 + constant r
```

Programmiersprachen: Beispiele

- 0-terminierte Strings in C (eingebaut/idiomatisch)

```
l=strlen(s);
strcat(strcat(strcat(s,s1),s2),s3);
```

- „C++ ist langsam“

- Mikrobenchmarks vs. Programmierwettbewerbe

- Flughafen von Riad

Code motion out of loops

```
for (...) {
  .... Berechnung ...
}
```

Berechnung hat keine Seiteneffekte
Berechnung benötigt keine Resultate aus der Schleife.

```
temp = Berechnung;
for (...) {
  .... temp ...
}
```

Combining Tests

z.B. Sentinel in Suchschleifen

```
for (i=0;  i<n && a[i]!=key; i++)
```

a[n] darf geschrieben werden

```
a[n] = key;
for (i=0; a[i]!=key; i++)
   ;
```

Verringert die Wartbarkeit, Reentrency

Loop Unrolling

```
for (i=0; i<n; i++)
  body(i);

for (i=0; i<n-1; i+=2) {
  body(i);
  body(i+1);
}
for (; i<n; i++)
  body(i);
```

Die Optimierung des entrollten Codes können Menschen besser

Transfer-Driven Unrolling/Modulo Variable Renaming

```
new_a = ...
... = ... a ...
a = new_a
```

Unrolling um Faktor 2

```
a2 = ...;
... = ... a1 ...;
a1 = ...;
... = ... a2 ...;
```

Software Pipelining

```
for (...) {
  a = ...;
  ... = ... a ...;
}
```

Berechnung von a hat keine Seiteneffekte

```
a = ...;
for (...) {
  ... = ... a ...;
  a = ...;
}
```

```
new_a = ...;
for (...) {
  a = new_a;
  new_a = ...;
  ... = ... a ...;
}
```

Unconditional Branch Removal

```
while (test)
  code;

if (test)
  do
    code;
  while (test);
```

Machen Compiler heute selbst

Loop Peeling

```
while (test)
  code;
```

```
if (test) {
  code;
  while (test)
    code;
}
```

Loop Fusion

```
for (i=0; i<n; i++)
  code1;
for (i=0; i<n; i++)
  code2;
```

Iteration k in code2 hängt nicht von Iteration $j > k$ in code1 ab.
Code2 überschreibt nicht Daten, die code1 liest.

```
for (i=0; i<n; i++) {
  code1;
  code2;
}
```

Exploit Algebraic Identities

```
~a&~b
```

```
~(a|b)
```

Computerarithmetik ist nich ganzzahlige Arithmetik und nicht Real-Arithmetik:

Integer: Overflow: $a > b \not\Leftrightarrow a + n > b + n$

FP: Rundungsfehler: $a + (b + c) \neq (a + b) + c$

Short-circuiting Monotone Functions

```
for (i=0, sum=0; i<n; i++)
  sum += x[i];
flag = sum > cutoff;
```

Alle `x[i]>=0`, sum und i werden danach nicht gebraucht.

```
for (i=0, sum=0; i<n && sum <= cutoff; i++)
  sum += x[i];
flag = sum > cutoff;
```

Unrolling für weniger Vergleiche und Verzweigungen.

Long-circuiting

```
A && B
```

A und B berechnen flags, B hat keine Seiteneffekte

```
A & B
```

Einsatzgebiet: Wenn B billig ist und A schwer vorhersagbar.

Arithmetik mit Flags

```
if (flag)
  x++;
```

```
x += (flag != 0);
```

Andere Flag-Repräsentation

```
(a<0) != (b<0)
```

```
(a^b) < 0
```

Reordering Tests

```
A && B
```

A und B haben keine Seiteneffekte

```
B && A
```

Welche Reihenfolge? Zuerst:

- Billiger
- Vorhersagbarer
- höhere Abkürzwahrscheinlichkeit

Reordering Tests

```
if (A)
  ...
else if (B)
  ...
```

A und B haben keine Seiteneffekte, $\neg(A \wedge B)$

```
if (B)
  ...
else if (A)
  ...
```

Precompute Functions

```
int foo(char c)
{
   ...
}
```

foo() hat keine Seiteneffekte.

```
int foo_table[] = {...};

int foo(char c)
{
  return foo_table[c];
}
```

Boolean/State Variable Elimination

```
flag = exp();
S1;
if (flag)
  S2;
else
  S3;
```

flag wird nachher nicht gebraucht.

```
if (exp()) {
  S1;
  S2;
} else {
  S1;
  S3;
}
```

Collapsing Procedure Hierarchies

- Inlining

- Specialization

```
foo(int i, int j)
{
...
}
... foo(1, a);
```

```
foo_1(int j)
{
  ...
}
```

Exploit Common Cases

Handle all cases correctly and common cases efficiently.

- Memoization: Bei teuren Funktionen: schon berechnete Resultate merken.
- Vorberechnete Tabellen/Codesequenzen für häufige Parameter

Coroutines

Statt Multi-Pass Verarbeitung:

```
coroutine producer {
  for (...)
    ... consumer(x); ...
}

coroutine consumer {
  for (...)
    ... x = producer(); ...
}
```

Auch Pipelines, Iteratoren, etc.

Transformation on Recursive Procedures

- Tail call optimization
- Inlining
- Ein rekursiver Aufruf: durch Zähler ersetzen
- Allgemein: expliziten Stack verwenden
- Für kleine Problemgrößen andere Methode
- Rekursion statt Iteration für automatisches Cache-blocking

Tail Call Optimization

```
void traverse_simple( PNODE p )
{
        if ( p!=0 )
        {
                traverse_simple( p->l );
                ...
                traverse_simple( p->r );
        }
}

start:
        if ( p!=0 )
        {
                traverse_simple( p->l );
                ...
                p = p->r; goto start;
        }
```

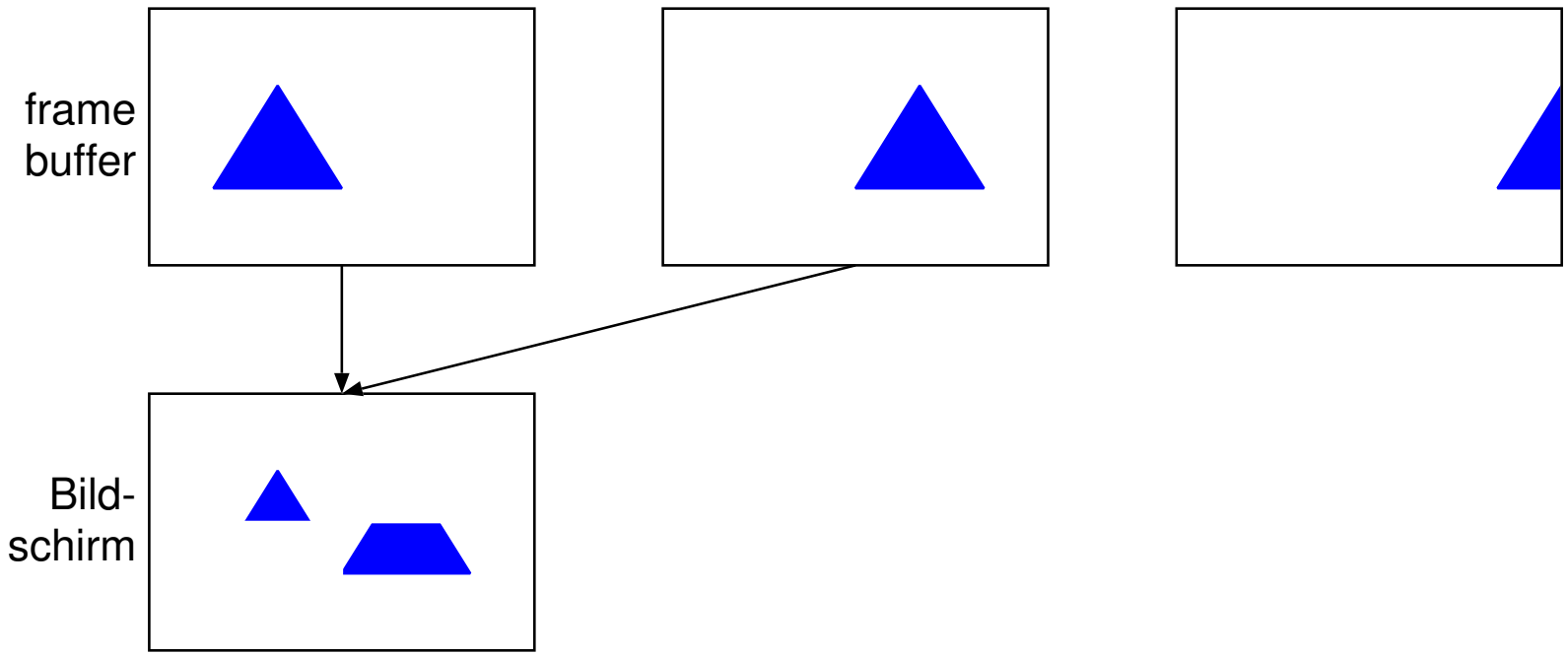
Zählerverwendung

```
foo()
{
  if (...) {
    code1;
    foo();
    code2;
  }
}

  while (...) {
    count++;
    code1;
  }
  for (i=0; i<count; i++)
    code2;
```

Parallelism

- Zwischen mehreren CPUs: multithreading
- Zwischen CPU und Platte: prefetching, write buffering
- Zwischen CPU und Graphik-Karte: triple buffering
- Zwischen CPU und Speicher: prefetching
- Zwischen verschiedenen Befehlen: instruction scheduling
- SIMD

Triple Buffering



- Double Buffering ohne Vertikalsynchronisation: Tearing
- Double Buffering mit vsync: Warten auf vsync
- Triple-Buffering: kein Tearing und kein Warten

Exploit Word Parallelism/SIMD

```
for (count=0; x > 0; x >>= 1)
  count += x&1;

/* 64-bit-spezifisch */
x = (x & 0x5555555555555555L) + ((x>>1) & 0x5555555555555555L);
x = (x & 0x3333333333333333L) + ((x>>2) & 0x3333333333333333L);
x = (x+(x>>4))   &0x0f0f0f0f0f0f0f0fL;
x = (x+(x>>8)) /*&0x001f001f001f001fL*/;
x = (x+(x>>16))/*&0x0000003f0000003fL*/;
x = (x+(x>>32))  &0x7fL;
count = x;

0|0|0|1|1|0|1|1
  0|  1|  1|  2
      1|      3
               4
```

Compile-Time Initialization

- Initialize tables at compile-time instead of at run-time
- CPU time vs. load time from disk

Strength Reduction/Incremental Algorithms/Differentiation

```
y = x*x;
x += 1;
y = x*x;
```

```
y = x*x;
x += 1;
y += 2*x-1;
```

Common subexpression elimination/Partial Redundancy Elimination

```
a = Exp;
b = Exp;
```

Exp hat keine Seiteneffekte

```
a = Exp;
b = a;
```

Pairing Computation

- Zusätzliches Resultat für geringen Aufwand

- Z.B. Division und Rest (C: `div`)
  sin und cos (glibc: `sincos`)

Data Structure Augmentation

- Felder mit redundanten Daten zur Beschleunigung gewisser Operationen
- größere Gefahr inkonsistenter Datenstrukturen
- Hints, die stimmen können, aber nicht müssen
- Memoization
- Caching

Automaten

- Zustand repräsentiert etwas Komplizierteres
- Endlicher Automat für Scannen
- Stackautomat für Parsen
- Baumautomat für tree parsing
  `iburg` (kein Automat) $\Rightarrow$ `burg`

Lazy Evaluation

- Beispiel: Automat für regular expression
- Beispiel: Tree-parsing automaton (burg)

Packing

- Keine überflüssigen Bytes/Bits (bitfields in C, `packed` in Pascal)
- Datenkompression
- Codegröße
- Cache-Verhalten

Interpreters, Factoring

- Ähnliche Codestücke als Prozeduren abstrahieren
- Schematische Programme per Interpreter implementieren

Programmbeispiel: Traveling Salesman Problem

- Eine Reihe von Städten besuchen, jede genau ein mal
  Reiseentfernung minimieren

- Optimale Lösung: NP-vollständig

- Beispiel nach Jon Bentley: suboptimaler Algorithmus
  Von jeder Stadt zur nahesten (gieriger Algorithmus)
  $\mathrm{O}(n^2)$, ca. 25% schlechter als optimale Lösung

Werkzeuge

- `gprof`: Profiling auf Funktionsebene

```
gcc -pg -O tsp1.c -lm -o tsp1
tsp1 10000 >/dev/null
gprof tsp1
```

- `gcov`: Profiling auf Zeilenebene

```
gcc -O --coverage tsp1.c -lm -o tsp1
tsp1 10000 >/dev/null
gcov tsp1
cat tsp1.c.gcov
```

Werkzeuge

- `perf stat`: Performance counters

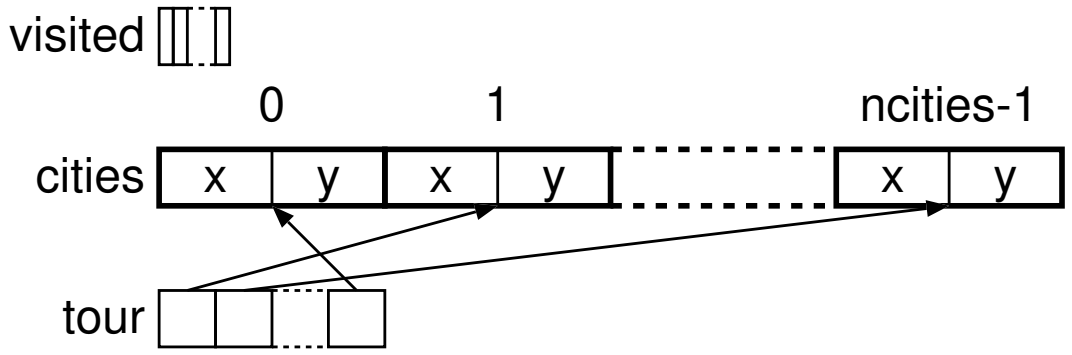
```
gcc -O tsp1.c -lm -o tsp1
perf list
perf stat -e cycles:u -e instructions:u -e L1-dcache-load-misses:u \
 -e dTLB-load-misses:u tsp1 10000 >/dev/null
```

- `perf`-basiertes Profiling

```
perf record -e cycles:u tsp1 10000 >/dev/null
perf annotate -s tsp
perf report
```

Traveling Salesman Problem: Heisser Code

```
for (i=1; i<ncities; i++) {
  CloseDist = DBL_MAX;
  for (j=0; j<ncities-1; j++) {
    if (!visited[j]) {
      if (dist(cities, ThisPt, j) < CloseDist) {
        CloseDist = dist(cities, ThisPt, j);
        ClosePt = j;
      }
    }
  }
  tour[endtour++] = ClosePt;
  visited[ClosePt] = 1;
  ThisPt = ClosePt;
}
```



tsp1 → tsp2: Common Subexpression Elimination

```
                                                    double ThisDist = dist(cities, ThisPt, j);
if (dist(cities,ThisPt,j) < CloseDist) { if (ThisDist < CloseDist) {
  CloseDist = dist(cities, ThisPt, j);     CloseDist = ThisDist;
```

tsp2 → tsp3: sqrt eliminieren

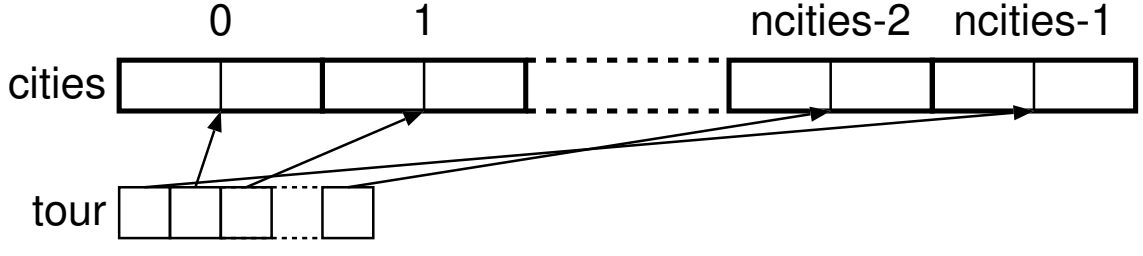
```
double dist(point cities[],
            int i, int j) {
  return sqrt(
       sqr(cities[i].x-cities[j].x)+
       sqr(cities[i].y-cities[j].y));
}
double ThisDist =
   dist(cities, ThisPt, j);
```

```
double DistSqrd(point cities[],
                int i, int j) {
  return (sqr(cities[i].x-cities[j].x)+
          sqr(cities[i].y-cities[j].y));
}

double ThisDist =
   DistSqrd(cities, ThisPt, j);
```

tsp3 → tsp4: visited eliminieren

```
for (i=0; i<ncities; i++)
  visited[i]=0;
...
for (j=0; j<ncities-1; j++) {
  if (!visited[j]) {
    double ThisDist =
      DistSqrd(cities, ThisPt, j);
    ...
  }
}
ThisPt = ClosePt;
tour[endtour++] = ClosePt;
visited[ClosePt] = 1;
```

```
for (i=1; i<ncities; i++)
  tour[i]=i-1;
...
for (j=i; j<ncities; j++) {

  double ThisDist =
    DistSqrd(cities, ThisPt, tour[j]);
  ...

}
ThisPt = tour[ClosePt];
swap(&tour[i],&tour[ClosePt]);
```



tsp4 → tsp5: DistSqrd inlinen

```
for (j=i; j<ncities; j++) {
  double ThisDist =
    DistSqrd(cities, ThisPt, tour[j]);
```

```
double ThisX = cities[ThisPt].x;
double ThisY = cities[ThisPt].y;
for (j=i; j<ncities; j++) {
  double ThisDist =
    sqr(cities[tour[j]].x-ThisX)+
    sqr(cities[tour[j]].y-ThisY);
```

tsp5 → tsp6: y-Distanz nur bei Bedarf berechnen

```
double ThisDist =
  sqr(cities[tour[j]].x-ThisX)+

  sqr(cities[tour[j]].y-ThisY);
if (ThisDist < CloseDist) {
  CloseDist = ThisDist;
  ClosePt = j;

}
```

```
double ThisDist =
  sqr(cities[tour[j]].x-ThisX);
if (ThisDist < CloseDist) {
  ThisDist += sqr(cities[tour[j]].y-ThisY);
  if (ThisDist < CloseDist) {
    CloseDist = ThisDist;
    ClosePt = j;
  }
}
```

Ausgelassen: ganze statt Gleitkomma-Zahlen

tsp6 → tsp8: Direkte Umordnung der Städte

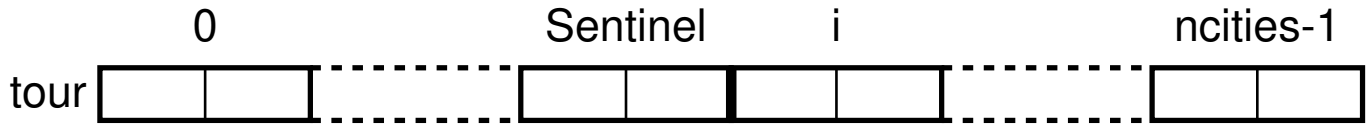
```
void tsp(point cities[], int tour[],
         int ncities)
...
double ThisX = cities[ThisPt].x;
double ThisY = cities[ThisPt].y;
CloseDist = DBL_MAX;
for (j=i; j<ncities; j++) {
  double ThisDist =
    sqr(cities[tour[j]].x-ThisX);
  if (ThisDist < CloseDist) {
    ThisDist +=
    sqr(cities[tour[j]].y-ThisY);
    ...
}
ThisPt = tour[ClosePt];
```

```
void tsp(point cities[], point tour[],
         int ncities)
...
double ThisX = tour[i-1].x;
double ThisY = tour[i-1].y;
CloseDist = DBL_MAX;
for (j=i; j<ncities; j++) {
  double ThisDist =
    sqr(tour[j].x-ThisX);
  if (ThisDist < CloseDist) {
    ThisDist +=
      sqr(tour[j].y-ThisY);
  ...
}
```

tsp8 → tsp9: Sentinel

```
for (j=i; j<ncities; j++) {
  double ThisDist = sqr(tour[j].x-ThisX);
  if (ThisDist < CloseDist) {
    ThisDist += sqr(tour[j].y-ThisY);
    if (ThisDist < CloseDist) {


      CloseDist = ThisDist;
      ClosePt = j;
    }
  }
}
```

```
for (j=ncities-1; ;j--) {
  double ThisDist = sqr(tour[j].x-ThisX);
  if (ThisDist <= CloseDist) {
    ThisDist += sqr(tour[j].y-ThisY);
    if (ThisDist <= CloseDist) {
      if (j < i)
        break;
      CloseDist = ThisDist;
      ClosePt = j;
    }
  }
}
```



Beispiel: Matrizenmultiplikation

$$C = AB$$

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$$

$$\begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{pmatrix} \begin{pmatrix} b_{11} & b_{12} & \cdots & b_{1p} \\ b_{21} & b_{22} & \cdots & b_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n1} & b_{n2} & \cdots & b_{np} \end{pmatrix} \begin{pmatrix} c_{11} & c_{12} & \cdots & c_{1p} \\ c_{21} & c_{22} & \cdots & c_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ c_{m1} & c_{m2} & \cdots & c_{mp} \end{pmatrix}$$

Beispiel: Matrizenmultiplikation

```
for (i=0; i<n; i++)
  for (j=0; j<p; j++) {
    for (k=0, r=0.0; k<m; k++)
      r += a[i*m+k]*b[k*p+j];
    c[i*p+j]=r;
  }
```

$n, p, m = 500$: 4.4Z/Iteration

$n, p, m = 700$: 23.3Z/Iteration

```
for (i=0; i<n; i++)
  for (j=0; j<p; j++)
    c[i*p+j] = 0.0;
for (i=0; i<n; i++)
  for (j=0; j<p; j++)
    for (k=0; k<m; k++)
      c[i*p+j] += a[i*m+k]*b[k*p+j];
```

$n, p, m = 500$: 5.1Z/Iteration

$n, p, m = 700$: 23.1Z/Iteration

Welche Verschachtelung?

```
for (i=0; i<n; i++)
 for (j=0; j<p; j++)
  for (k=0; k<m; k++)
   c[i*p+j]+=a[i*m+k]*b[k*p+j];
```

```
for (i=0; i<n; i++)
 for (k=0; k<m; k++)
  for (j=0; j<p; j++)
   c[i*p+j]+=a[i*m+k]*b[k*p+j];
```

```
for (j=0; j<p; j++)
 for (k=0; k<m; k++)
  for (i=0; i<n; i++)
   c[i*p+j]+=a[i*m+k]*b[k*p+j];
```

```
for (j=0; j<p; j++)
 for (i=0; i<n; i++)
  for (k=0; k<m; k++)
   c[i*p+j]+=a[i*m+k]*b[k*p+j];
```

```
for (k=0; k<m; k++)
 for (i=0; i<n; i++)
  for (j=0; j<p; j++)
   c[i*p+j]+=a[i*m+k]*b[k*p+j];
```

```
for (k=0; k<m; k++)
 for (j=0; j<p; j++)
  for (i=0; i<n; i++)
   c[i*p+j]+=a[i*m+k]*b[k*p+j];
```

Welche Verschachtelung?

```
for (i=0; i<n; i++)
 for (j=0; j<p; j++)
  for (k=0; k<m; k++)
   c[i*p+j]+=a[i*m+k]*b[k*p+j];
```

$n,p,m = 500$ -O2: 5.5Z/It
$n,p,m = 700$ -O2: 23.1Z/It
$n,p,m = 700$ -O3: 23.1Z/It

```
for (i=0; i<n; i++)
 for (k=0; k<m; k++)
  for (j=0; j<p; j++)
   c[i*p+j]+=a[i*m+k]*b[k*p+j];
```

$n,p,m = 500$ -O2: 3.2Z/It
$n,p,m = 700$ -O2: 3.2Z/It
$n,p,m = 700$ -O3: 1.4Z/It

```
for (j=0; j<p; j++)
 for (k=0; k<m; k++)
  for (i=0; i<n; i++)
   c[i*p+j]+=a[i*m+k]*b[k*p+j];
```

$n,p,m = 500$ -O2: 51.8Z/It
$n,p,m = 700$ -O2: 54.1Z/It
$n,p,m = 700$ -O3: 54.2Z/It

```
for (j=0; j<p; j++)
 for (i=0; i<n; i++)
  for (k=0; k<m; k++)
   c[i*p+j]+=a[i*m+k]*b[k*p+j];
```

$n,p,m = 500$ -O2: 4.5Z/It
$n,p,m = 700$ -O2: 23.1Z/It
$n,p,m = 700$ -O3: 23.0Z/It

```
for (k=0; k<m; k++)
 for (i=0; i<n; i++)
  for (j=0; j<p; j++)
   c[i*p+j]+=a[i*m+k]*b[k*p+j];
```

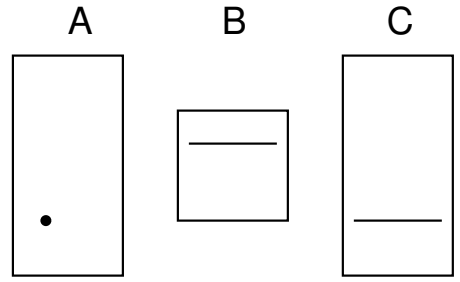
$n,p,m = 500$ -O2: 3.2Z/It
$n,p,m = 700$ -O2: 3.3Z/It
$n,p,m = 700$ -O3: 1.9Z/It

```
for (k=0; k<m; k++)
 for (j=0; j<p; j++)
  for (i=0; i<n; i++)
   c[i*p+j]+=a[i*m+k]*b[k*p+j];
```

$n,p,m = 500$ -O2: 51.6Z/It
$n,p,m = 700$ -O2: 54.0Z/It
$n,p,m = 700$ -O3: 54.1Z/It

Gründe

- räumliche Lokalität
  TLB misses
  cache misses
  j als innerste Schleife
  j erlaubt SIMD-Befehle (Auto-Vektorisierung: `-O3`)
- Recurrences (Abhängigkeiten zwischen Iterationen)
  nicht k als innerste Schleife
- Zeitliche Lokalität
  k als mittlere Schleife: Zeile `c[i*p+j]` wiederverwendet



mm2-ikj → mm3: Explizite Vektorisierung

```
void matmul(
  double a[], double b[], double c[],
  size_t m, size_t n, size_t p)
{
```

1.42Z/It

```
typedef double v4d
    __attribute__ ((vector_size (32)));
void matmul(
  double a[], v4d b[], v4d c[],
  size_t m, size_t n, size_t p)
{
  p=p/4;
```

1.66Z/It

mm3 → mm4: Loop-invariant code motion

```
for (j=0; j<p; j++)
  c[i*p+j] += a[i*m+k]*b[k*p+j];
```

1.66Z/It

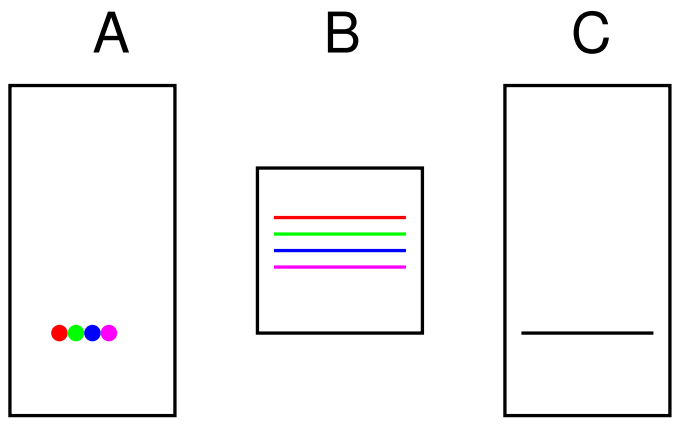
```
double aik = a[i*m+k];
for (j=0; j<p; j++)
  c[i*p+j] += aik*b[k*p+j];
```

1.54Z/It

mm4 → mm5: Loop unrolling, interchange

```
for (k=0; k<m; k++) {
  double aik = a[i*m+k];



  for (j=0; j<p; j++)

    c[i*p+j] += aik*b[k*p+j];

}
```

1.54Z/It

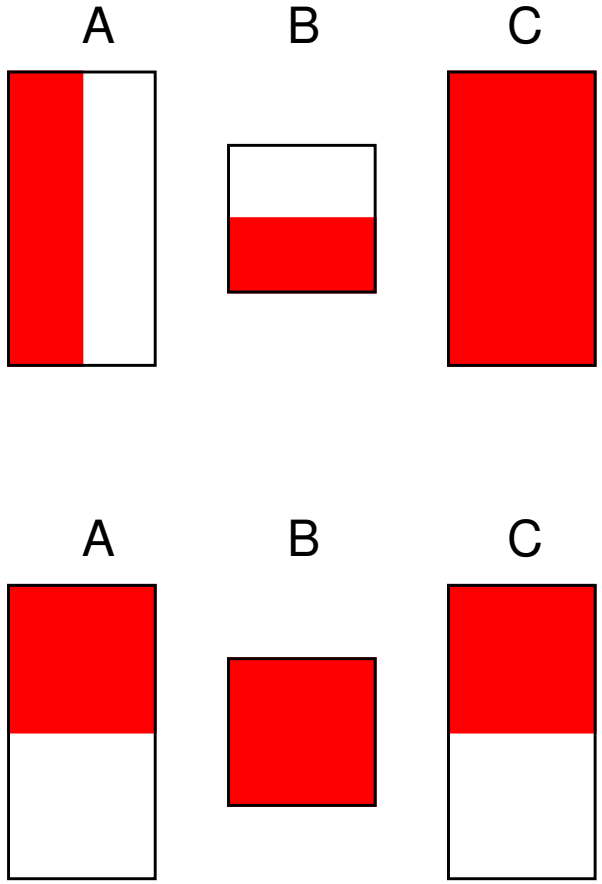



```
for (k=0; k<m; k+=4) {
  double aik0 = a[i*m+k+0];
  double aik1 = a[i*m+k+1];
  double aik2 = a[i*m+k+2];
  double aik3 = a[i*m+k+3];
  for (j=0; j<p; j++) {
    v4d r;
    r  = aik0*b[(k+0)*p+j];
    r += aik1*b[(k+1)*p+j];
    r += aik2*b[(k+2)*p+j];
    r += aik3*b[(k+3)*p+j];
    c[i*p+j] += r;
  }
}
```

1.11Z/It

mm5 → mm6: Rekursion

```
for (i=0; i<n; i++)
  for (k=0; k<m; k+=4)
```



1.11Z/It

```
static void matmul1(
  double a[], v4d b[], v4d c[],
  size_t m, size_t n, size_t p,
  size_t m1, size_t n1)
{
  if (m1>=8) {
    size_t m2 = (m1/2)&~3;
    size_t m3 = m1-m2;
    matmul2(a   ,b     ,c,m,n,p,m2,n1);
    matmul2(a+m2,b+m2*p,c,m,n,p,m3,n1);
  } else {
    matmul2(a,b,c,m,n,p,m1,n1);
  }
}
```

0.64Z/It

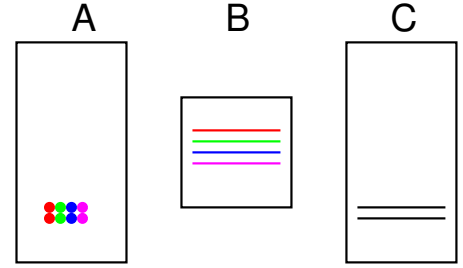
mm6 → mm7: Loop unrolling, interchange

```
for (i=0; i<n1; i++) {
  double aik0 = a[i*m+0];
  double aik1 = a[i*m+1];
  double aik2 = a[i*m+2];
  double aik3 = a[i*m+3];
  for (j=0; j<p; j++) {
    v4d r;
    r  = aik0*b[0*p+j];
    r += aik1*b[1*p+j];
    r += aik2*b[2*p+j];
    r += aik3*b[3*p+j];
    c[i*p+j] += r;
  }
}
```

0.64Z/It

```
for (i=0; i<n1; i+=2) {
  double ai0k0 = a[(i+0)*m+0]; double ai1k0 = a[(i+1)*m+0];
  double ai0k1 = a[(i+0)*m+1]; double ai1k1 = a[(i+1)*m+1];
  double ai0k2 = a[(i+0)*m+2]; double ai1k2 = a[(i+1)*m+2];
  double ai0k3 = a[(i+0)*m+3]; double ai1k3 = a[(i+1)*m+3];
  for (j=0; j<p; j++) {

    v4d bk0j = b[0*p+j]; v4d bk2j = b[2*p+j];
    v4d bk1j = b[1*p+j]; v4d bk3j = b[3*p+j];
    v4d ci0j = ai0k0*bk0j+ai0k1*bk1j+ai0k2*bk2j+ai0k3*bk3j;
    v4d ci1j = ai1k0*bk0j+ai1k1*bk1j+ai1k2*bk2j+ai1k3*bk3j;
    c[(i+0)*p+j] += ci0j; c[(i+1)*p+j] += ci1j;
  }
}
```

0.54Z/It



ATLAS, OpenBLAS

- ATLAS: 0.65Z/It
- OpenBLAS (1 thread): 0.36Z/It