

# 6. Speicherhierarchie und Speicheroptimierung

AlDaBi Praktikum

# Inhalt

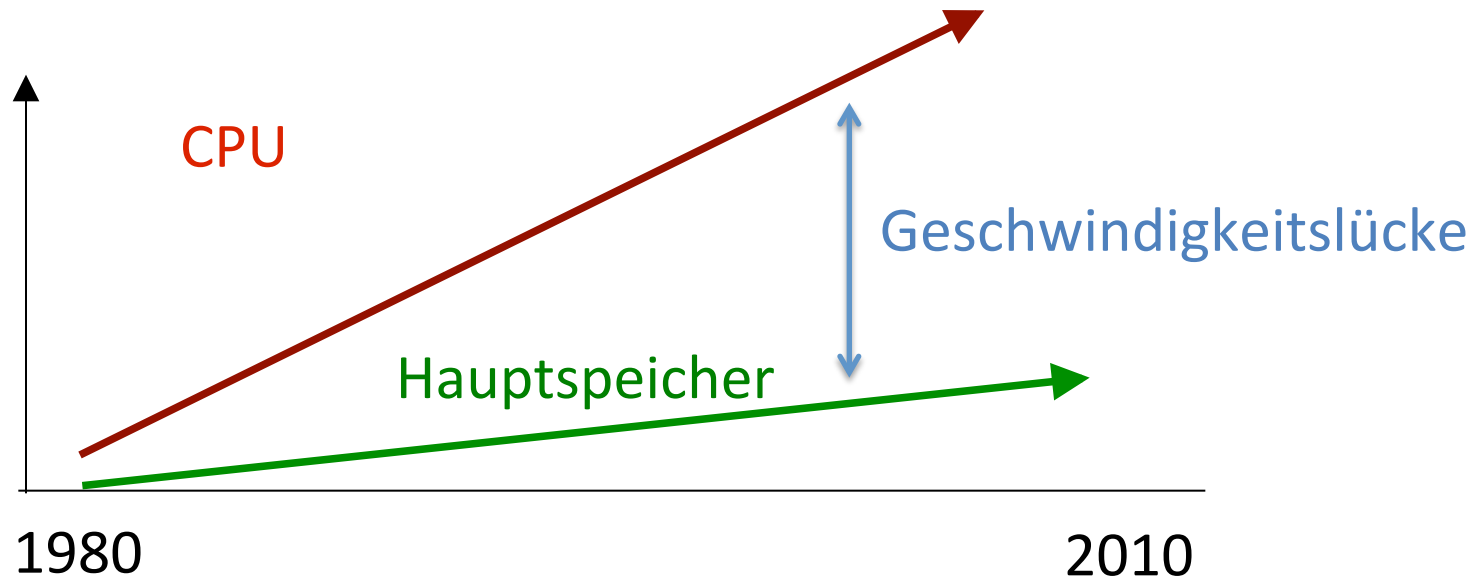
- Speicherhierarchie
- Speicheroptimierung
- Bemerkungen zur P-Aufgabe

# SPEICHERHIERARCHIE

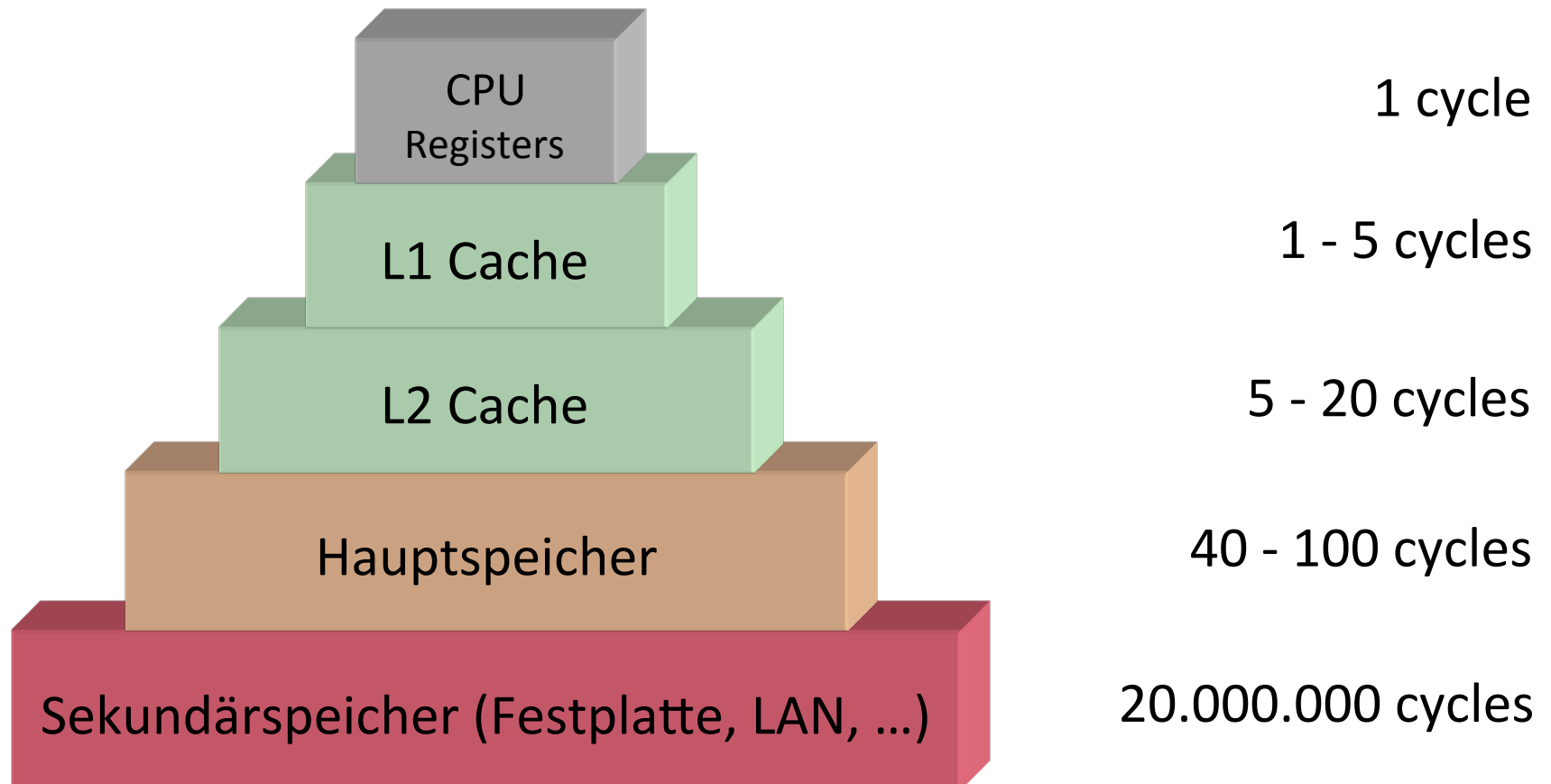
Folien z.T. aus VL „Programmierung von Hardwarebeschleunigern“ von Alexander Reinefeld und Thomas Steinke, WS09

# Technologie-Trends

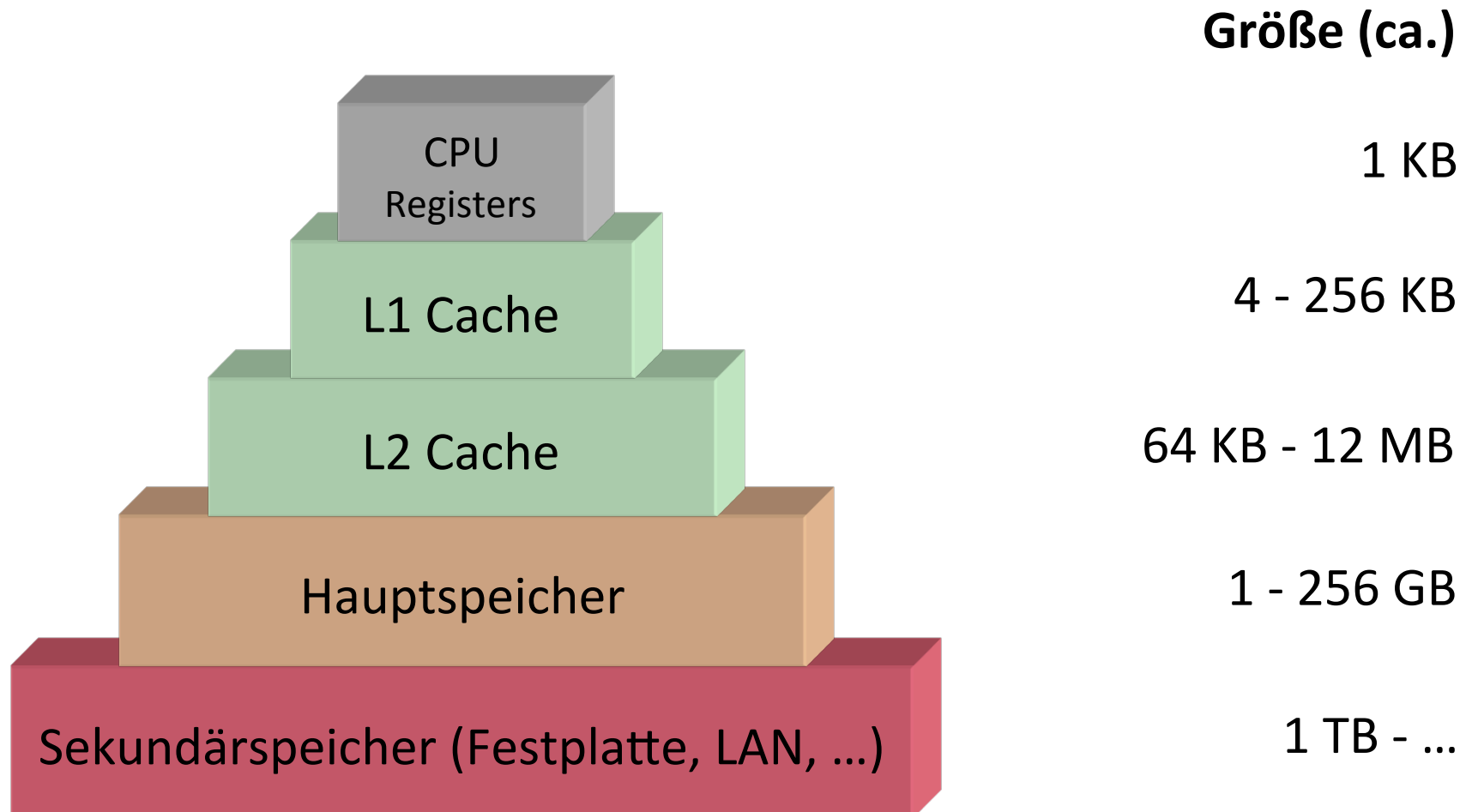
- Seit 1980 wächst die Geschwindigkeit des ...
  - ... Hauptspeichers um 7% / Jahr
  - ... Prozessors um 55% / Jahr



# Die Speicherhierarchie



# Die Speicherhierarchie



# Lokalitätsprinzip

- Ein Programm verbringt 90% der Laufzeit in 10% des Codes
- Es wird unterschieden:
  - Zeitliche Lokalität
    - Auf Speicher, auf den kürzlich zugegriffen wurde, wird bald wieder zugegriffen (Schleifen, lokale Variablen)
  - Örtliche Lokalität
    - Auf Speicher in der Nachbarschaft von kürzlich benutzten Speicher wird bald wieder zugegriffen (Linearer Programmcode, Array-Zugriffe)

# Lokalität

- Caches beschleunigen lokale Speicherzugriffe
- Kann das überhaupt effizient funktionieren?
  - Antwort: meistens!
  - Grund: Räumliche und zeitliche Lokalität von Programmen, d.h. Abarbeitung während kürzerer Zeit bewegt sich häufig in engen Adressbereichen.
    - Abarbeitung von Schleifen
    - In zeitlich engem Abstand Zugriff auf gleiche Daten
    - Zugriffe auf benachbarte Daten
  - Aufgabe des Programmierers



# Terminologie

- Cache
  - Verhalten sich wie Puffer, bspw. File Cache, DNS Cache
  - Speichern Blöcke des nächst-niedrigeren (langsameren) Levels
- Block
  - Datenmenge fester Größe, die zwischen Cache und Hauptspeicher/Cache transportiert wird
- Latenz (Zugriffszeit) eines Speicherzugriffs
  - Zeit um das erste Wort eines Blocks zu übermitteln
- Bandbreite (Durchsatz) eines Speicherzugriffs
  - Blockgröße in Bytes / Zeit um alle Wörter eines Blocks zu übermitteln

# Cache-Funktionsweise

- Prozessor/Caches stellen Speicheranfragen
  - Beispiel: Lesen eines Bytes an Adresse 0x11223344
- Cache Hit
  - Eine Speicheranfrage kann erfolgreich vom Cache beantwortet werden = entsprechender Block ist im Cache vorhanden
- Cache Miss
  - Eine Speicheranfrage kann nicht allein vom Cache beantwortet werden = entsprechender Block ist nicht im Cache vorhanden
    - Block mit geforderter Speicheradresse muss von einer niedrigeren Ebene (L2 Cache, RAM) geladen werden
    - Geladener Block wird im Cache gespeichert, dazu muss vorher ein anderer Block verdrängt werden
- Verdrängen
  - Zurückschreiben des Blocks in eine niedrigere Ebene

# Cache-Arten

- Direkt abgebildet (direct mapped)
  - Jedem Block wird genau eine Position im Cache zugewiesen:
    - $(\text{block\_addr}) \bmod (\text{number\_of\_blocks\_in\_cache})$
- Vollassoziativ (fully associative)
  - In Block kann überall im Cache platziert werden
- n-fach assoziativ (n-way set associative)
  - Cache ist unterteilt in **sets** mit jeweils n Blöcken
  - Erst wird ein Block einem set zugewiesen
    - $(\text{block\_addr}) \bmod (\text{number\_of\_sets\_in\_cache})$
  - Innerhalb des sets kann er platziert werden

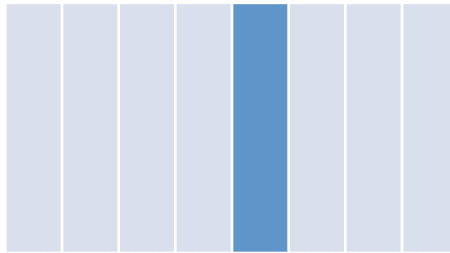
# Cache-Arten (II)

- Wohin wird ein Block im Cache platziert?

## Caches mit 8 Blöcken

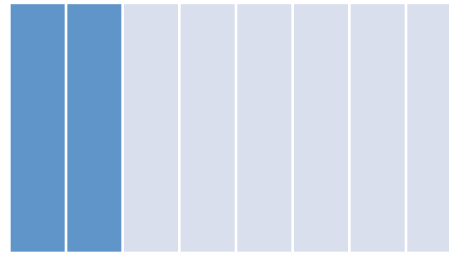
direkt abgebildet  
 $(12 \bmod 8) = 4$

0 1 2 3 4 5 6 7



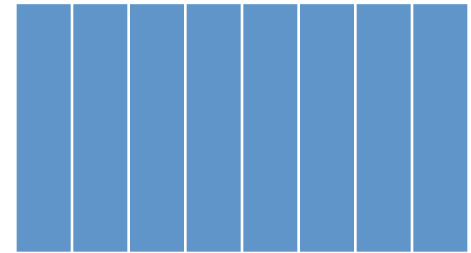
2-fach assoziativ  
 $(12 \bmod 4) = 0$

0 0 1 1 2 2 3 3



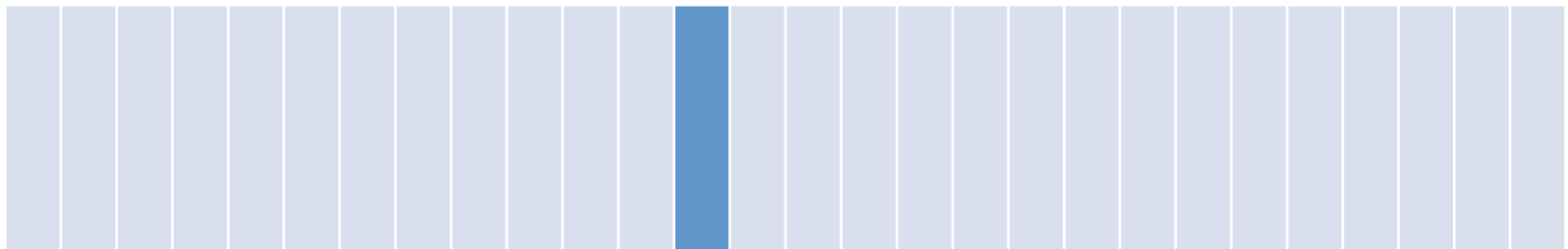
vollassoziativ

0 0 0 0 0 0 0 0



# Hauptspeicher

											1	1	1	1	1	1	1	1	1	2	2	2	2	2	2	2	2
0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7



# Feinde: Drei C's der Cache Misses

- Compulsory Misses
  - Unvermeidliche Misses wenn auf Speicher zum ersten Mal zugegriffen wird
- Capacity Misses
  - Unzureichender Cache-Speicher um alle aktiven Daten zu speichern
  - Auf zu viele Daten wird aufeinanderfolgend zugegriffen
- Conflict Misses
  - Blöcke werden verdrängt, weil sie auf dieselbe Cache-Zeile abgebildet werden

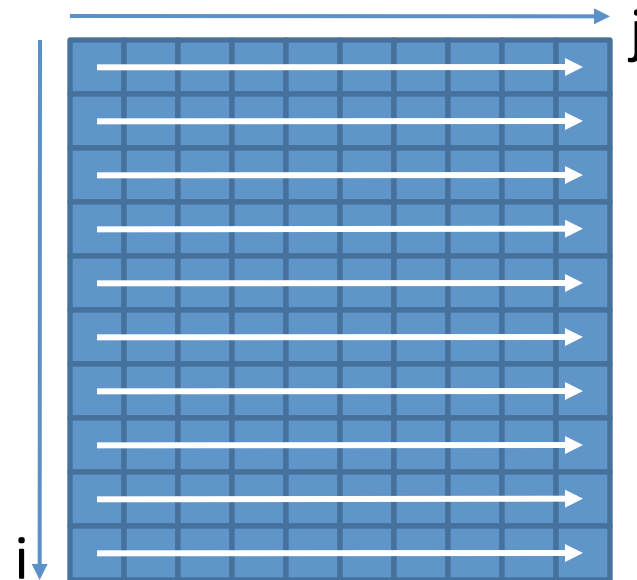
# Freunde: Die drei R's

- Rearrange (Code, Daten)
  - Ändern des Layouts von Code/Daten um örtliche Lokalität zu erhöhen
- Reduce (Größe, # gelesenen Cache-Zeilen)
  - Kleine/cleverere Formate, Kompression
- Reuse (Cache-Zeilen)
  - Erhöhen der zeitlichen und örtlichen Lokalität

	Compulsory	Capacity	Conflict
Rearrange	X	(x)	X
Reduce	X	X	(x)
Reuse	(x)	X	

# Compulsory Misses

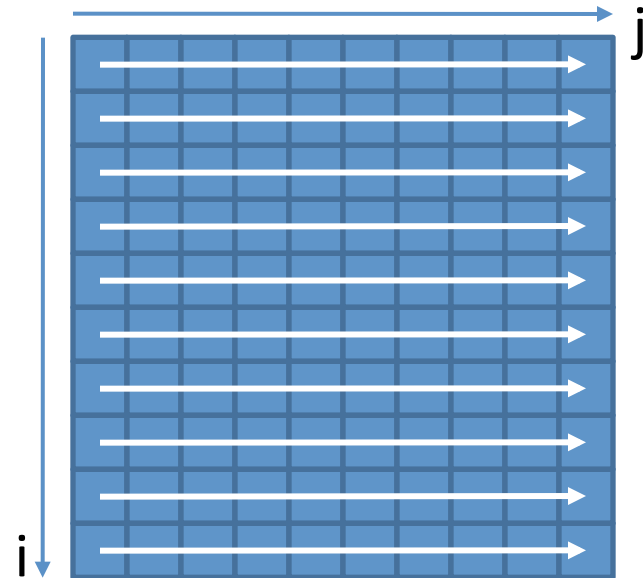
- Matrix  $M$  mit  $m \times n$  Elementen
  - $m$  Zeilen
  - $n$  Spalten
- Arbeitsspeicher „1-dimensional“, daher:
  - Ordne jede Zeile nacheinander im Speicher an
  - Zugriff auf Element  $M_{i,j}$  in Zeile  $i$  und Spalte  $j$ :
    - $M[i * n + j]$



# Compulsory Misses (II)

- Beispiel: Initialisiere alle Elemente
  - Variante 1:

```
// jede Zeile
for(i = 0; i < m; ++i)
{
    // jede Spalte
    for(j = 0; j < n; ++j)
    {
        pos = i * n + j;
        M[pos] = 0;
    }
}
```



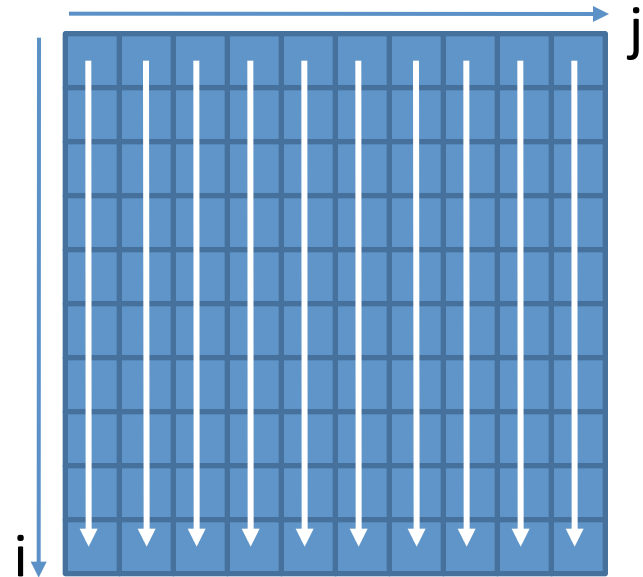
- Für  $n, m = 300$  auf einer aktuellen Intel CPU (IA32):  $\sim 0.048$  s



# Compulsory Misses (III)

- Beispiel: Initialisiere alle Elemente
  - Variante 2: Vertausche Zeilenindex mit Spaltenindex

```
// jede Spalte
for(j = 0; j < n; ++j)
{
    // jede Zeile
    for(i = 0; i < m; ++i)
    {
        pos = i * n + j;
        M[pos] = 0;
    }
}
```



- Für  $n, m = 300$  auf einer aktuellen Intel CPU (IA32):  **$\sim 0.160$  s**

# Capacity Misses

- Ein einfaches Programm<sup>1</sup>:

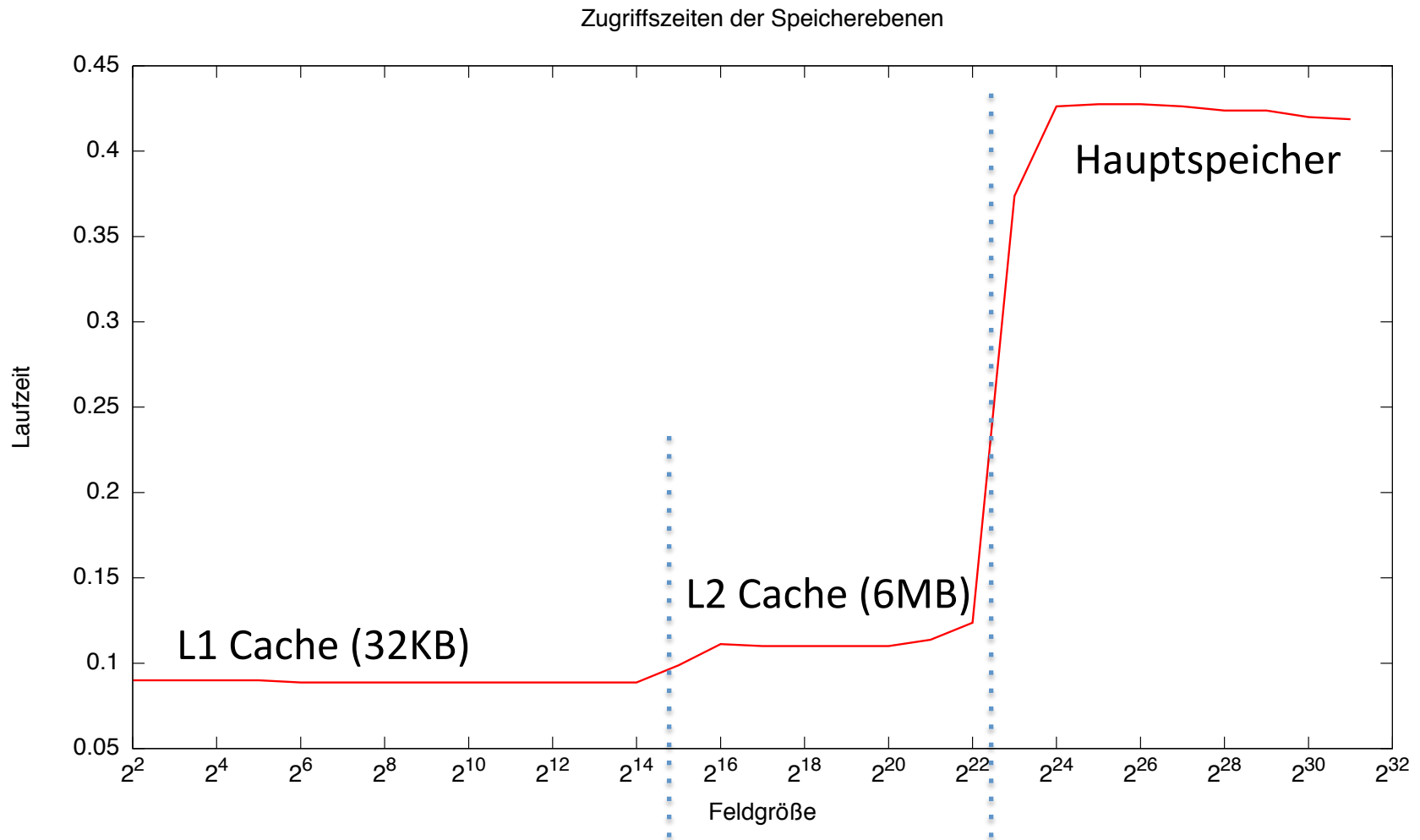
```
int steps = 64*1024*1024; // Arbitrary number of steps
int lengthMod = buffer_size - 1;
for (int i = 0; i < steps; i++)
    arr[(i * 16) & lengthMod]++;
```

- Durchlaufe das Programm mit verschiedenen Werten für `buffer_size`  
 $2^0, 2^1, 2^2, 2^3, \dots$
- Immer gleich viele Zugriffe (steps) aber innerhalb verschieden großer Bereiche

<sup>1</sup> [https://svn.imp.fu-berlin.de/aldabi/WS12/material/cache\\_test/](https://svn.imp.fu-berlin.de/aldabi/WS12/material/cache_test/)

# Capacity Misses (II)

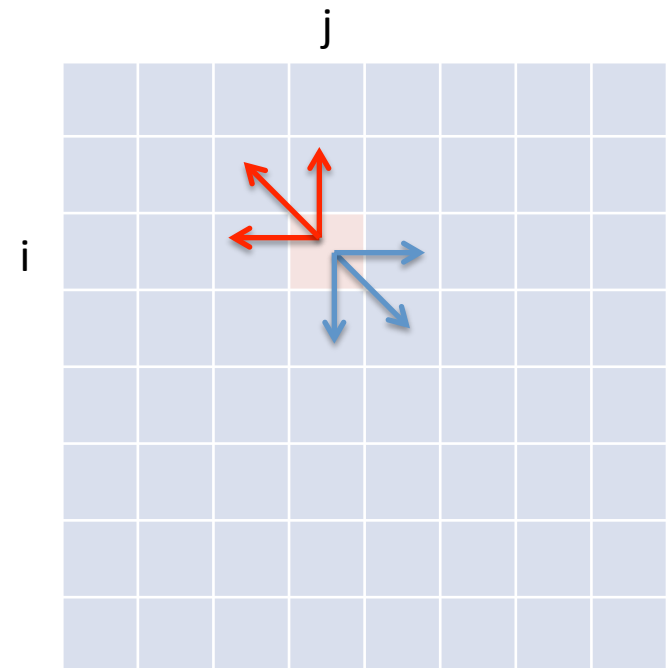
- Von Feldgröße abhängige Laufzeiten



# **SPEICHEROPTIMIERUNG**

# Cache-effiziente DP-Algorithmen

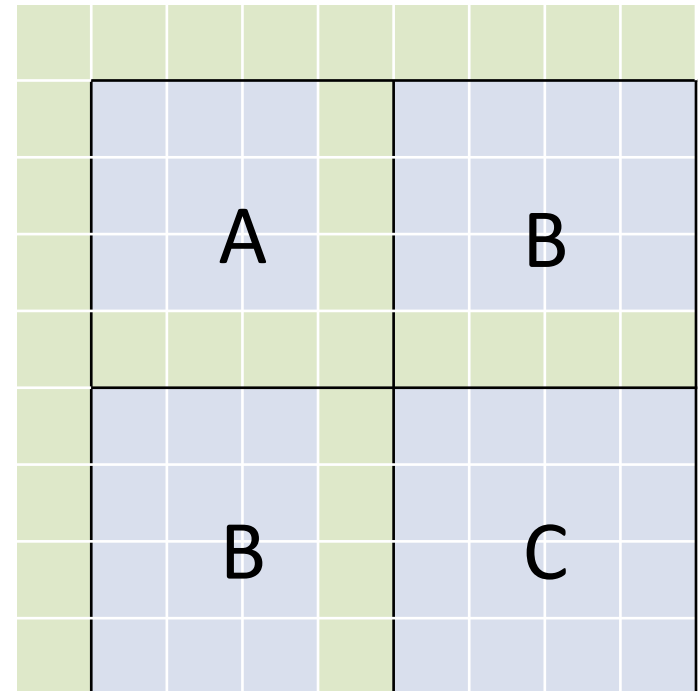
- Beispiel: Needleman-Wunsch Alignment
  - $M_{i,j}$  **hängt** ab von  $M_{i-1,j-1}$ ,  $M_{i,j-1}$  und  $M_{i-1,j}$
  - $M_{i,j}$  **wird** von  $M_{i+1,j+1}$ ,  $M_{i,j+1}$  und  $M_{i+1,j}$  **benötigt**
- Optimierungsziele:
  - Minimiere Capacity Misses
    - $M_{i,j}$  sollte möglichst noch im Cache sein, wenn es wieder benutzt wird
    - Wähle Berechnungsreihenfolge, so dass Zeit zwischen Berechnung und Benutzung von  $M_{i,j}$  minimiert wird
  - Minimiere Compulsory Misses
    - Wähle Datenstruktur für  $M$ , so dass aufeinanderfolgende Zugriffe auch im Speicher aufeinander folgen



# Tiling

- Lösung:
  - Zerlege DP-Matrix in Blöcke (Tiles) der Größe des Caches
  - Berechne jeden Block einzeln
    - Benutze letzte Spalte/Zeile/Element des linken/oberen/diagonalen Nachbarn

- Idee lässt sich auch benutzen für:
  - Paralleles Berechnen der DP-Matrix
    - Antidiagonal-Blöcke (B) sind unabhängig
  - Matrizen die nicht in den Hauptspeicher passen



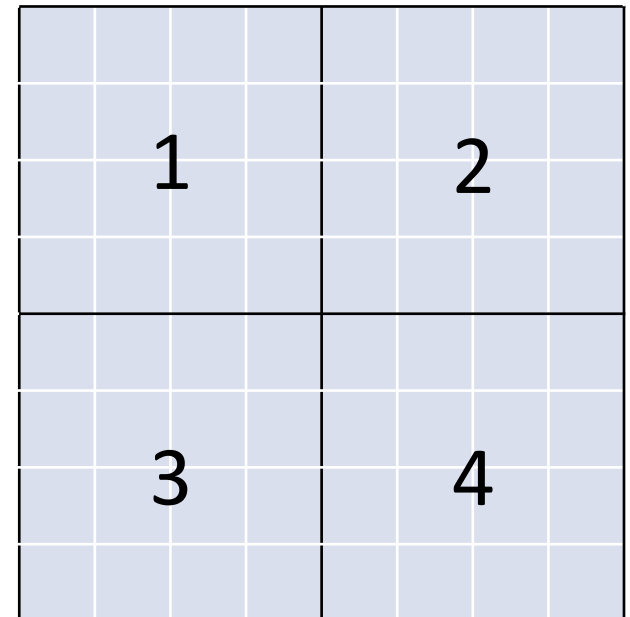
- *Cache-efficient Dynamic Programming Algorithms for Multicores*, Chowdhury and Ramachandran, 2008

# Cache-Optimierte Algorithmen

- Wie groß muss ein Block gewählt werden = Wie groß ist der Cache?
- Cache-aware Algorithmus
  - Hat Kenntnis über Größe und Art der Prozessorcaches
  - Nutzt diese explizit zur Zugriffsbeschleunigung
- Cache-oblivious Algorithmus
  - Hat keine Kenntnis über Prozessorcaches
  - Nutzt trotzdem durch Orts- und Zeitlokalität Caches effizient aus

# Cache-oblivious Tiling

- Wenn Cache-Größe nicht bekannt ist:
  - Zerlege und berechne DP-Matrix rekursiv in Viertel-Blöcken
- Warum funktioniert das?
  - In Rekursionstiefe  $t$  wird ein Quadrat der Größe  $m \times m$  berechnet mit  $m = n/2^t$
  - Ab einem gewissen  $t$  finden alle Berechnungen im Cache statt



- *Cache-Oblivious Dynamic Programming for Bioinformatics*, Chowdhury, Le and Ramachandran, 2010



# Cache-oblivious Tiling

- Wenn Cache-Größe nicht bekannt ist:
  - Zerlege und berechne DP-Matrix rekursiv in Viertel-Blöcken
- Warum funktioniert das?
  - In Rekursionstiefe  $t$  wird ein Quadrat der Größe  $m \times m$  berechnet mit  $m = n/2^t$
  - Ab einem gewissen  $t$  finden alle Berechnungen im Cache statt

1	2	5	6
3	4	7	8
9	10	13	14
11	12	15	16

- *Cache-Oblivious Dynamic Programming for Bioinformatics*, Chowdhury, Le and Ramachandran, 2010

# Cache-oblivious Tiling

- Wenn Cache-Größe nicht bekannt ist:
  - Zerlege und berechne DP-Matrix rekursiv in Viertel-Blöcken
- Warum funktioniert das?
  - In Rekursionstiefe  $t$  wird ein Quadrat der Größe  $m \times m$  berechnet mit  $m = n/2^t$
  - Ab einem gewissen  $t$  finden alle Berechnungen im Cache statt
- Rekursionsabbruch bei kleinen Quadraten
  - Bspw. bei  $m=8$
  - Nicht-rekursives Berechnen des Quadrats wie bei Cache-aware Tiling
- *Cache-Oblivious Dynamic Programming for Bioinformatics*, Chowdhury, Le and Ramachandran, 2010

1	2	5	6	17	18	21	22
3	4	7	8	19	20	23	24
9	10	13	14	25	26	29	30
11	12	15	16	27	28	31	32
33	34	37	38	49	50	53	54
35	36	39	40	51	52	55	56
41	42	45	46	57	58	61	62
43	44	47	48	59	60	63	64

# Code-Cache Optimierung

- Ortslokalität (Compulsory Misses vermeiden)
  - Umordnen von Funktionen
    - Manuell innerhalb des Quelltexts
    - Umordnen von Object-Dateien während des Linkens (Reihenfolge im Makefile)
    - `__attribute__((section("xxx")))` in gcc
  - Programmierstil anpassen
    - Monolithische Funktionen statt starker Verschachtelung
    - Kurze Funktionen mit inline einsetzen statt anspringen
    - Kapselung/OOP ist wenig code-cache-freundlich

# Code Cache Optimierung (II)

- Capacity Misses vermeiden:
  - Aufpassen beim Gebrauch von:
    - inline bei Funktionen
    - ausgerollten Schleifen (loop unrolling)
    - großen Makros
  - KISS (keep it simple stupid)
    - Featuritis vermeiden
    - Template-Spezialisierungen verwenden
  - Schleifen zerlegen oder vereinen
  - Mit Code-Größenoptimierung compilieren (“-Os” beim gcc)
  - Umschreiben in Assembler (wo es viel bringt)

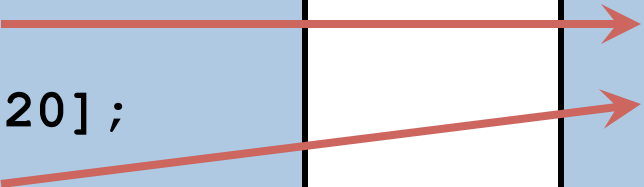
# Data-Cache Optimierung

- Cache-oblivious Layout
  - Umordnen der Elemente (sind normalerweise konzeptuell geordnet)
  - Hot/cold Splitting
- Wahl der richtigen Datenstruktur
  - Feld von structs
  - struct mit Feldern
- Wenig Unterstützung vom Compiler
  - Einfacher für Sprachen ohne Pointer (Java)
  - C/C++: Aufgabe des Programmierers

# Umordnen der Elemente

```
struct S {  
    void *key;  
    int count[20];  
    S *pNext;  
};
```

```
struct S {  
    void *key;  
    S *pNext;  
    int count[20];  
};
```



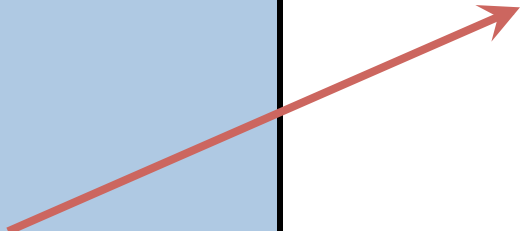
```
void Foo(S *p, void *key, int k) {  
    while (p) {  
        if (p->key == key) {  
            p->count[k]++;  
            break;  
        }  
        p = p->pNext;  
    }  
}
```

Wird auf Elemente  
gemeinsam zugegriffen,  
sollten sie ortslokal  
gespeichert werden

# Hot/Cold Splitting

Hot fields:

```
struct S {  
    void *key;  
    S *pNext;  
    S2 *pCold;  
};
```

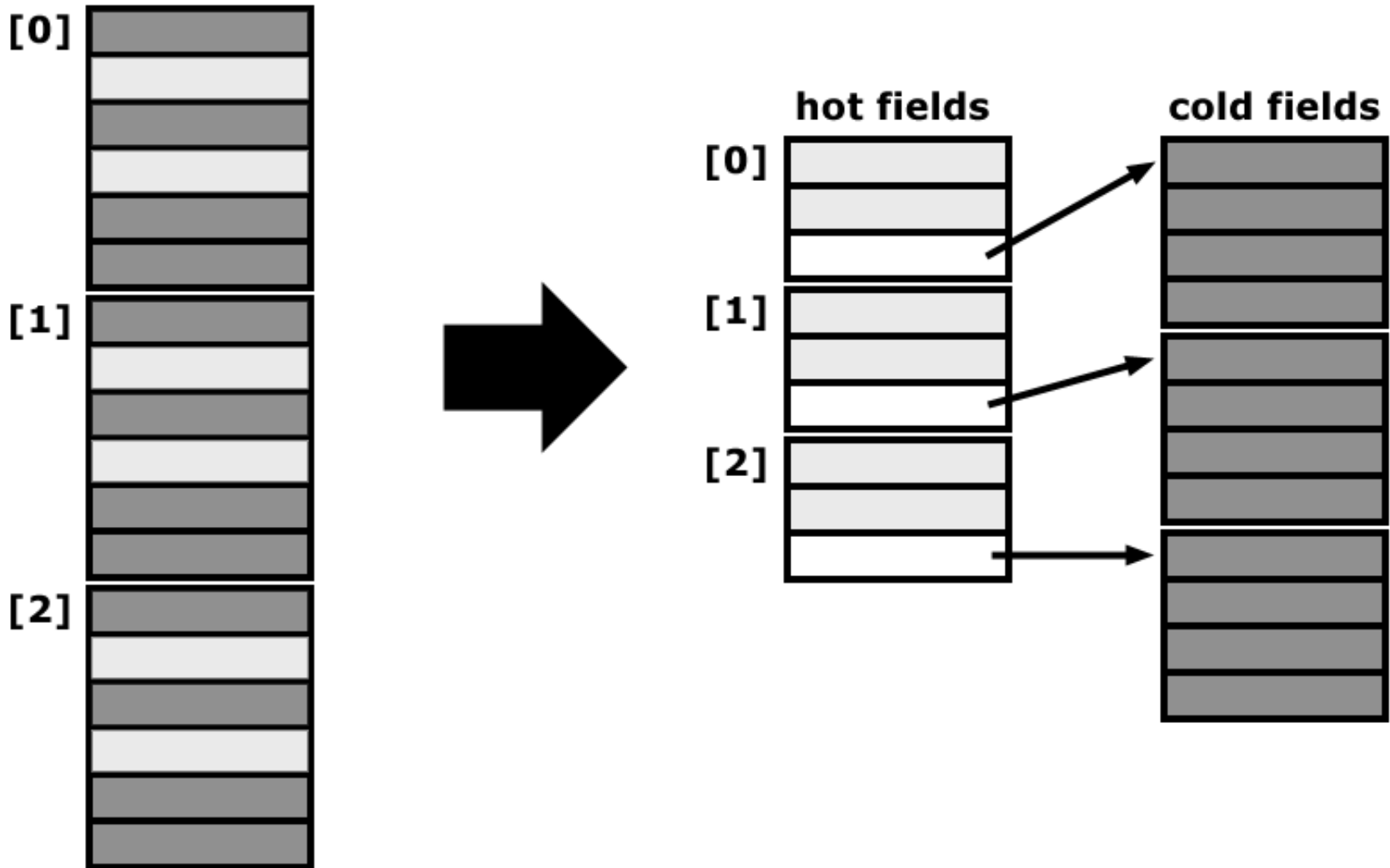


Cold fields:

```
struct S2 {  
    int count[10];  
};
```

- Alloziere 'struct S' von einem Memory Pool
  - Schneller als new
  - erhöht Cache-Kohärenz
- Bevorzuge Feld von structs
  - Erfordert keinen extra Zeiger pCold

# Hot/Cold Splitting





# Compiler Padding

- Viele Prozessoren erwarten Datentypen ein bestimmtes Adress-Alignment
  - Adress-Alignment von x byte bedeutet, dass Adresse durch x teilbar ist
  - Misalignment-Zugriffe brauchen länger/führen zum Absturz
  - Alignment ist meist mindestens die Größe des Datentyps
- Beispiel:
  - char Variablen werden byte-aligniert und können an jeder Adresse beginnen
  - short (2 byte) werden 2-byte aligniert, d.h. 0x10004567 ist keine gültige Adresse für short Variablen
- Elemente von structs/classes müssen auch aligniert werden
  - dadurch entsteht “Verschnitt”, sog. Padding
  - das Alignment der struct/class ist das Maximum der Alignments der Elemente

# Compiler Padding (II)

```
struct X {  
    int8 a;  
    int64 b;  
    int8 c;  
    int16 d;  
    int64 e;  
    float f;  
};
```

```
struct Y {  
    int8 a, pad1[7];  
    int64 b;  
    int8 c, pad2[1];  
    int16 d, pad3[2];  
    int64 e;  
    float f, pad4[1];  
};
```

```
struct Z {  
    int64 b;  
    int64 e;  
    float f;  
    int16 d;  
    int8 a;  
    int8 c;  
};
```

Besser: Abnehmende Größe!



- Bei 4-byte großen floats, gilt für die meisten Compiler:
  - sizeof(X) == 40; Summe der Elemente: 24 byte
  - sizeof(Y) == 40;
  - sizeof(Z) == 24

# **BEMERKUNGEN ZUR P-AUFGABE**

# Bemerkungen zu Aufgabe 5

- Typische Fehler:
  - QGramIndex:
    - hash-Funktionen nicht korrekt implementiert
    - SuffixArray falsch konstruiert
    - QGramIndex nicht über das Genom aufgebaut sondern über die Reads
  - Filter:
    - Nicht in überlappenden Blöcken gezählt ( $b = 2 \cdot w$ )
    - $|D| / b$  = Anzahl der Blöcke für die erste Reihe
    - $(|D| - w) / b$  = Anzahl der überlappenden Blöcke in der zweiten Reihe
    - Bei der Suche:
      - $x = \text{sufftab}[\text{dir}[h(Q_r)]] \dots \text{suftab}[\text{dir}[h(Q_r) + 1]]$
      - $x/b \Rightarrow$  Block in der ersten Reihe
      - $(x-w)/b \Rightarrow$  Block in der zweiten Reihe
  - Verifikation:
    - $F[i, 0] = 0$  for  $i < -1 \dots m$ ;  $F(0, j) = F(0, j-1) - 1$  for  $j < -1 \dots n$
    - Suche  $\max$  in  $F(m, j)$ ,  $j < -0 \dots n$
    - Verwenden von Levenshtein-Distanz, dann ist auch kein Traceback notwendig.

# Bemerkungen zu Aufgabe 5 (II)

- Mögliche Verbesserungen:
  - Zielvektoren als Referenz übergeben

```
vector<int> macheWas()  
{  
    vector<int> returnVec;  
    // mache was mit returnVec  
    return returnVec;  
}
```

```
void macheWas(vector<int> &returnVec)  
{  
    // mache was mit returnVec  
}
```

- Benutzen der STL

```
vector<char> q;  
for (int i = 0; i < Q; ++i)  
    q.push_back(D[i]);  
  
for (int i = 0; i < q.size() ++i)  
    // compute hashes ...
```

```
string q = D.substr(0,Q);  
for (int i = 0; i < q.size() ++i)  
    // compute hashes ...  
  
string::iterator it = D.begin();  
for (; it != D.begin()+Q; ++it)  
    // compute hashes
```

# Hinweise zu Aufgabe 6

- Es sollen C-optimale Schnittpositionen der Sequenzen  $s_1, \dots, s_k$  gefunden werden
- Herangehensweise:
  - Berechne Additional-Cost-Matrix für jedes Sequenzpaar mit Needleman-Wunsch in Vorwärts- und Rückwärtsrichtung
    - Rückwärts: Entweder Sequenz umdrehen (und Matrix umgedreht benutzen) oder DP-Algorithmus umdrehen
    - Nicht vergessen, von jedem Element in  $D^f$  und  $D^r$  den Score  $c_{\text{opt}}(s, t)$  des normalen paarweisen Alignments abzuziehen
  - Aufzählen aller möglichen Schnittpositionstupel  $c_1, \dots, c_k$  mit  $0 < c_i < |s_k|$ 
    - `vector<int>` der Länge  $k$  zum Speichern eines Tupels
    - Rekursiv über  $i$  und Aufsummieren der Additional Costs von  $s_i$  zu  $s_j$  mit  $j=1, \dots, i-1$
    - Rekursionsabbruch bei Überschreiten des bisherigen Minimums
  - Parallele Rekursion
    - Threads teilen sich den Wertebereich  $c_i=1, \dots, |s_k|-1$  auf und zählen  $c_2, \dots, c_k$  rekursiv auf
    - Globale Variablen für das bisheriges Minimum der Additional Costs und zug. Schnittpositionen

# Branch & Bound

```
#include <limits>

vector<string> sequences;
int minimum = numeric_limits<int>::max();

void enumerate (vector<int> & pos, int i, int summe)
{
    if (summe >= minimum) return;           // bound
    if (i == pos.size())
    {
        minimum = summe;                   // neues minimum
        return;
    }
    for (pos[i] = 1; pos[i] < sequences[i].size(); ++pos[i])
    {
        int summand = ...;
        enumerate (pos, i + 1, summe + summand); // branch
    }
}

int main(int argc, char *argp[])
{
    ...
    vector<int> p(k);
    enumerate(p, 0, 0);
    ...
}
```