

3.4 Verbesserung der Speicher-Organisation

Ziel:

- Ausnutzung von Caches
 - ⇒ Verringerung der Anzahl der Cache-Misses
- Verringerung der Allokations / Deallokations-Kosten
 - ⇒ Ersetzung von Heap-Allokation durch Stack-Allokation
 - ⇒ Unterstützung der Freigabe überflüssiger Heap-Objekte
- Verringerung der Zugriffskosten
 - ⇒ Verkürzung der Indirektionsketten (**Unboxing**)

1. Cache-Optimierung:

Idee: **lokale Speicherzugriffe**

- Laden aus dem Speicher lädt nicht nur ein Byte, sondern füllt eine ganze Cache-Zeile.
- Zugriff auf benachbarte Zellen werden billiger.
- Passen alle Daten einer inneren Schleife in den Cache, wird die Iteration extrem speicher-effizient ...

Mögliche Lösungen:

- Organisiere Zugriffe auf die vorhanden Daten um !
- Organisiere die Daten um !

Solche Optimierungen funktionieren i.a. automatisch nur für
Felder :-)

Beispiel:

```
for (j = 1; j < n; j++)  
    for (i = 1; i < m; i++)  
        a[i][j] = a[i - 1][j - 1] + a[i][j];
```



Iteriere stets erst über die **Zeilen!**

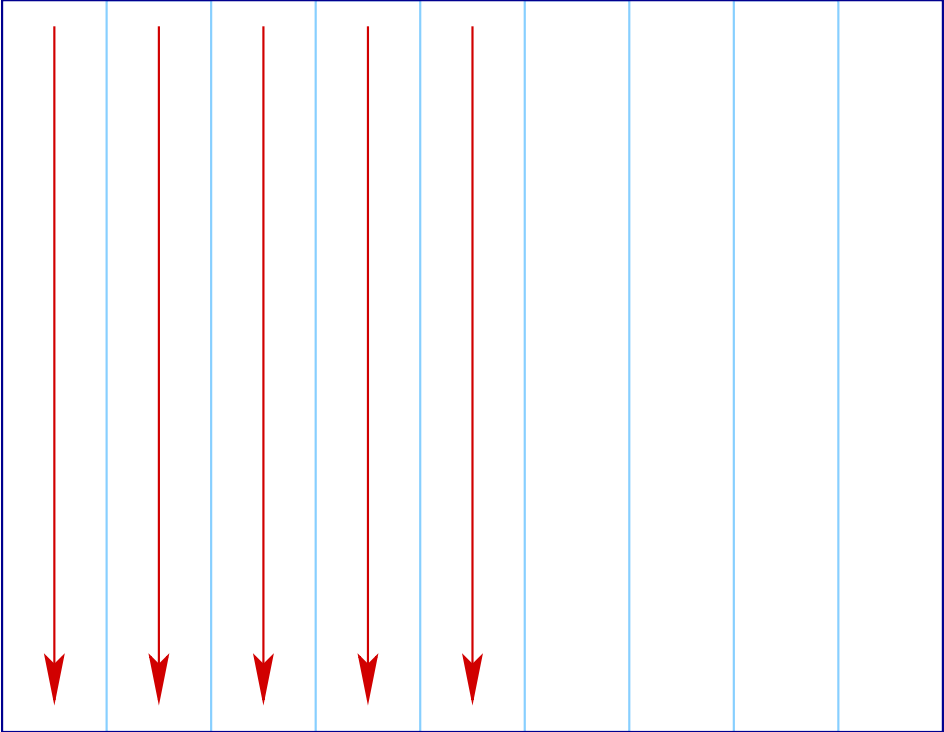


Vertausche die Reihenfolge der Iterationen:

```
for (i = 1; i < m; i++)  
    for (j = 1; j < n; j++)  
        a[i][j] = a[i - 1][j - 1] + a[i][j];
```

Wann ist das erlaubt ???

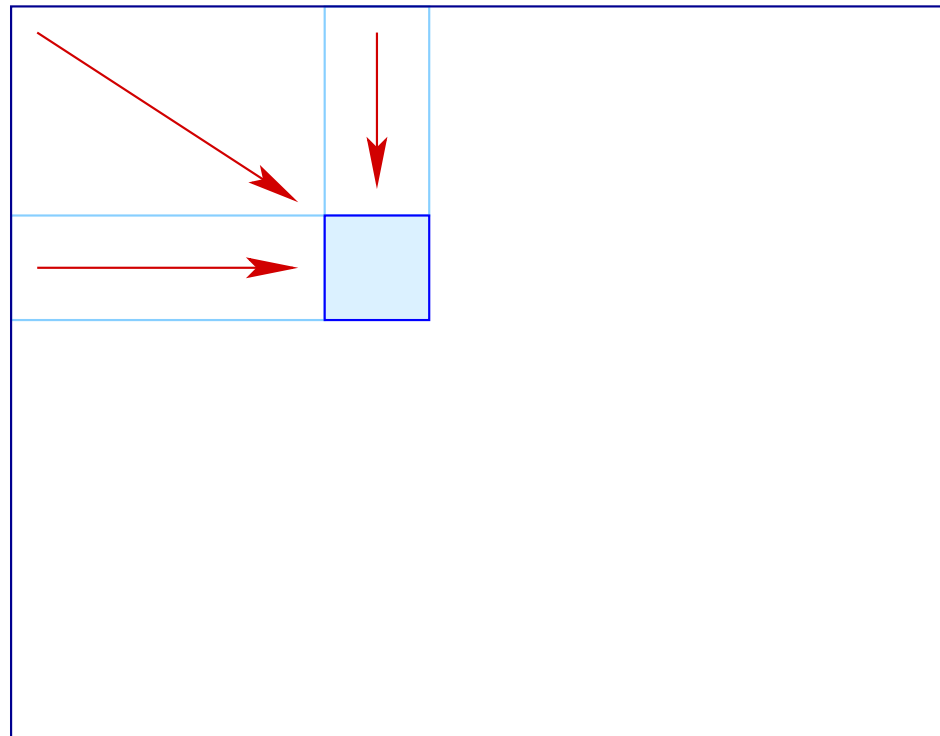
Iterations-Schema: vorher:



Iterations-Schema: *nachher:*



Iterations-Schema: erlaubte Abhängigkeiten:



In unserem Fall müssen wir überprüfen, dass die folgenden Gleichungs-Systeme **keine** Lösung haben:

| Schreiben | | Lesen |
|--------------|---|----------------------|
| (i_1, j_1) | = | $(i_2 - 1, j_2 - 1)$ |
| i_1 | ≤ | i_2 |
| j_2 | ≤ | j_1 |
| (i_1, j_1) | = | $(i_2 - 1, j_2 - 1)$ |
| i_2 | ≤ | i_1 |
| j_1 | ≤ | j_2 |

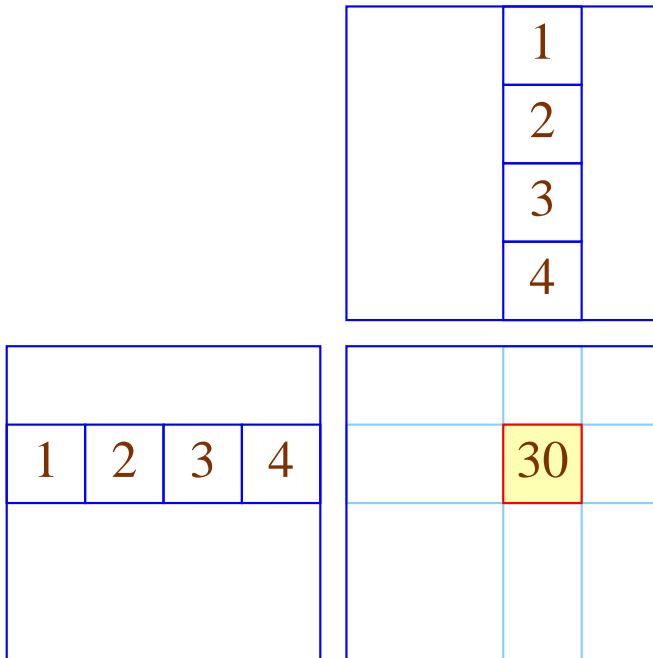
Das erste impliziert: $j_2 \leq j_2 - 1$ **Hurra!**

Das zweite impliziert: $i_2 \leq i_2 - 1$ **Hurra!**

Beispiel: Matrix-Matrix-Multiplikation

```
for (i = 0; i < N; i++)  
    for (j = 0; j < M; j++)  
        for (k = 0; k < K; k++)  
            c[i][j] = c[i][j] + a[i][k] · b[k][j];
```

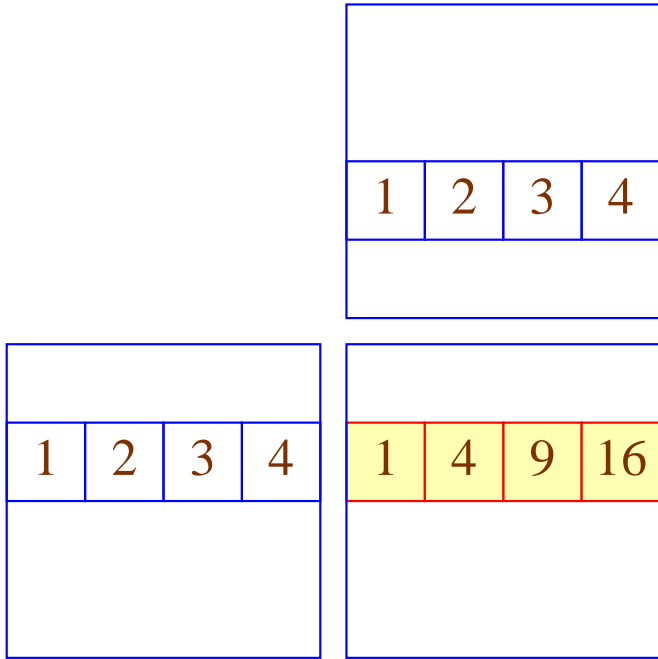
Über $b[][]$ iterieren wir **spaltenweise** :-)



Vertausche die beiden inneren Schleifen:

```
for (i = 0; i < N; i++)  
    for (k = 0; k < K; k++)  
        for (j = 0; j < M; j++)  
            c[i][j] = c[i][j] + a[i][k] · b[k][j];
```

Ist das erlaubt ???



Diskussion:

- Die Korrektheit folgt genauso wie eben :-)
- Eine ähnliche Idee lässt sich auch zur Implementierung von Matrix-Multiplikation **zeilen-komprimierter** Matrizen benutzen :-))
- Möglicherweise muss das Programm erst **konditioniert** werden, damit die Anwendbarkeit der Transformation erkannt wird :-)
- Matrix-Multiplikation benötigt evt. erst eine Initialisierung der Ergebnis-Matrix ...

```

for (i = 0; i < N; i++)
    for (j = 0; j < M; j++) {
        c[i][j] = 0;
        for (k = 0; k < K; k++)
            c[i][j] = c[i][j] + a[i][k] · b[k][j];
    }

```

- Jetzt können wir die beiden Iterationen nicht einfach vertauschen :-)
- Wir können aber die Iteration über j duplizieren ...

```

for (i = 0; i < N; i++) {
    for (j = 0; j < M; j++) c[i][j] = 0;
    for (j = 0; j < M; j++)
        for (k = 0; k < K; k++)
            c[i][j] = c[i][j] + a[i][k] · b[k][j];
}

```

Zur Korrektheit:

- ⇒ Die gelesenen Einträge (hier: keine) dürfen im Rest des Rumpfs nicht modifiziert werden !!!
- ⇒ Die Reihenfolge der Schreibzugriffe einer Zelle darf sich nicht ändern :-)

Man erhält:

```
for (i = 0; i < N; i++) {  
    for (j = 0; j < M; j++) c[i][j] = 0;  
    for (k = 0; k < K; k++)  
        for (j = 0; j < M; j++)  
            c[i][j] = c[i][j] + a[i][k] · b[k][j];  
}
```

Diskussion:

- Statt mehrere Schleifen zusammen zu fassen, haben wir Schleifen **distribuiert** :-)
- Desgleichen zieht man Abfragen vor die Schleife \implies if-Distribution ...

Achtung:

Statt dieser Transformation könnte man die innere Schleife auch anders optimieren:

```
for (i = 0; i < N; i++)  
    for (j = 0; j < M; j++) {  
        t = 0;  
        for (k = 0; k < K; k++)  
            t = t + a[i][k] · b[k][j];  
        c[i][j] = t;  
    }
```

Idee:

Finden wir ein **heftig benutztes** Feld-Element $a[e_1] \dots [e_r]$, dessen Index-Ausdrücke e_l innerhalb der inneren Schleife **konstant** sind, können wir stattdessen ein Hilfsregister spendieren :-)

Achtung:

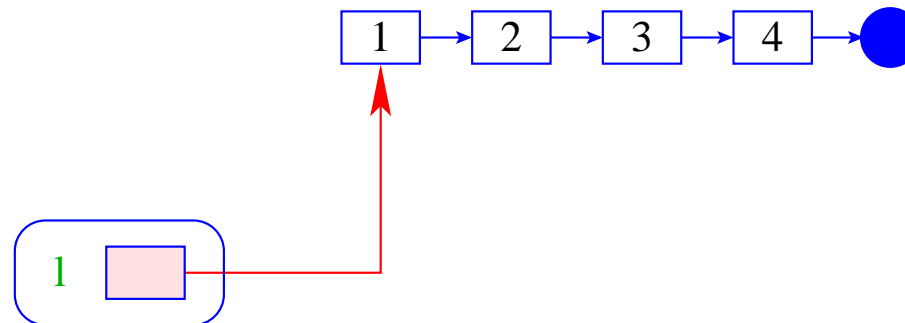
Diese Optimierung verhindert die vorherige und umgekehrt ...

Diskussion:

- Die bisherigen Optimierungen beziehen sich auf Iterationen über Feldern.
- Cache-sensible Organisation anderer Datenstrukturen ist möglich, aber i.a. nicht vollautomatisch möglich ...

Beispiel:

Keller



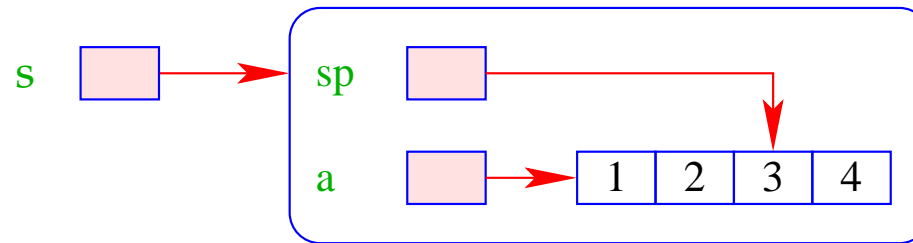
Vorteil:

- + Die Implementierung ist einfach :-)
- + Die Operationen **push** / **pop** erfordern konstante Zeit :-)
- + Die Datenstruktur ist potentiell beliebig groß :-)

Nachteil:

- Die einzelnen Listenknoten können beliebig über den Speicher verteilt sein :-)

Alternative:



Vorteil:

- + Die Implementierung ist auch einfach :-)
 - + Die Operationen **push** / **pop** erfordern konstante Zeit :-)
 - + Die Daten liegen konsequent; Stack-Schwankungen sind im **Mittel** gering
- ⇒ gutes Cache-Verhalten !!!

Nachteil:

- Die Datenstruktur ist **beschränkt** :-)

Verbesserung:

- Ist das Feld **voll**, ersetze es durch ein **doppelt** so großes !!!
- Wird das Feld **leer bis auf ein Viertel**, **halbiere** es wieder !!!

⇒ Die Extra-Kosten sind **amortisiert** konstant :-)

⇒ Die Implementierung ist nicht mehr ganz so trivial :-}

Diskussion:

- Die gleiche Idee klappt auch für **Schlangen** :-)
- Andere Datenstrukturen bemüht man sich, blockweise aufzuteilen.

Problem: wie organisiert man die Zugriffe, dass sie **möglichst lange** auf dem selben Block arbeiten ???

⇒ **Algorithmen auf externen Daten**

2. Stack-Allokation statt Heap-Allokation

Problem:

- Programmiersprachen wie **Java** legen **alle** Datenstrukturen im Heap an — selbst wenn sie nur innerhalb der aktuellen Methode benötigt werden :-)
- Überlebt kein Verweis auf diese Daten den Aufruf, wollen wir sie auf dem Stack allokkieren :-)

⇒⇒⇒ Escape-Analyse

Idee:

Berechne **Alias**-Information.

Bestimme, ob ein erzeugtes Objekt möglicherweise von **außen** erreichbar ist ...

Beispiel: unsere Pointer-Sprache

```
x = new();  
y = new();  
x → a = y;  
z = y;  
return z;
```

... könnte ein möglicher Methoden-Rumpf sein :-)

Von außen zugänglich sind Objekte, die:

- von `return` zurück geliefert werden;
- einer **globalen Variablen** zugewiesen werden;
- von solchen Objekten **erreichbar sind**.

... im Beispiel:

```
x = new();  
y = new();  
x → a = y;  
z = y;  
return z;
```

Von außen zugänglich sind Objekte, die:

- von `return` zurück geliefert werden;
- einer **globalen Variablen** zugewiesen werden;
- von solchen Objekten **erreichbar sind**.

... im Beispiel:

```
x = new();  
y = new();  
x → a = y;  
z = y;  
return z;
```

Von außen zugänglich sind Objekte, die:

- von `return` zurück geliefert werden;
- einer **globalen Variablen** zugewiesen werden;
- von solchen Objekten **erreichbar sind**.

... im Beispiel:

```
x = new();
```

```
y = new();
```

```
x → a = y;
```

```
z = y;
```

```
return z;
```

Von außen zugänglich sind Objekte, die:

- von `return` zurück geliefert werden;
- einer **globalen Variablen** zugewiesen werden;
- von solchen Objekten **erreichbar sind**.

... im Beispiel:

```
x = new();  
y = new();  
x → a = y;  
z = y;  
return z;
```

Wir schließen:

- Die Objekte, die das erste `new()` anlegt, können nicht entkommen.
- Wir können sie darum auf dem Stack allokkieren :-)

Achtung:

Das ist natürlich nur **sinnvoll**, wenn von dieser Sorte nur **wenige** pro Methoden-Aufruf angelegt werden :-)

Liegt deshalb ein solches lokales `new()` in einer Schleife, sollten wir die Objekte **vorsichtshalber** doch im Heap anlegen ;-)

Erweiterung: Formale Parameter

- Wir benötigen eine **interprozedurale** Alias-Analyse :-)
- Kennen wir das gesamte Programm, können wir z.B. die Kontrollflussgraphen der einzelnen Funktionen zu einem einzigen zusammen fassen (durch Hinzufügen geeigneter Kanten) und für diesen Alias-Information berechnen ...
- **Achtung:** benutzen wir die **selben Namen** y_1, y_2, \dots für die formalen Parameter, wird die Information dort notwendig ungenau :-)
- Kennen wir das Gesamtprogramm **nicht**, müssen wir annehmen, dass **jede** Referenz, die einer anderen Funktion bekannt ist, entkommt :-((