3.4 Verbesserung der Speicher-Organisation

Ziel:

- Ausnutzung von Caches

  $\Longrightarrow$ Verringerung der Anzahl der Cache-Misses

- Verringerung der Allokations / Deallokations-Kosten

  $\Longrightarrow$ Ersetzung von Heap-Allokation durch Stack-Allokation

  $\Longrightarrow$ Unterstützung der Freigabe überflüssiger Heap-Objekte

- Verringerung der Zugriffskosten

  $\Longrightarrow$ 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:

$$\begin{array}{l} \textsf{for}\ \ (j = 1; j < n; j{+}{+}) \\ \qquad \textsf{for}\ \ (i = 1; i < m; i{+}{+}) \\ \qquad\qquad a[i][j] = a[i-1][j-1] + a[i][j]; \end{array}$$

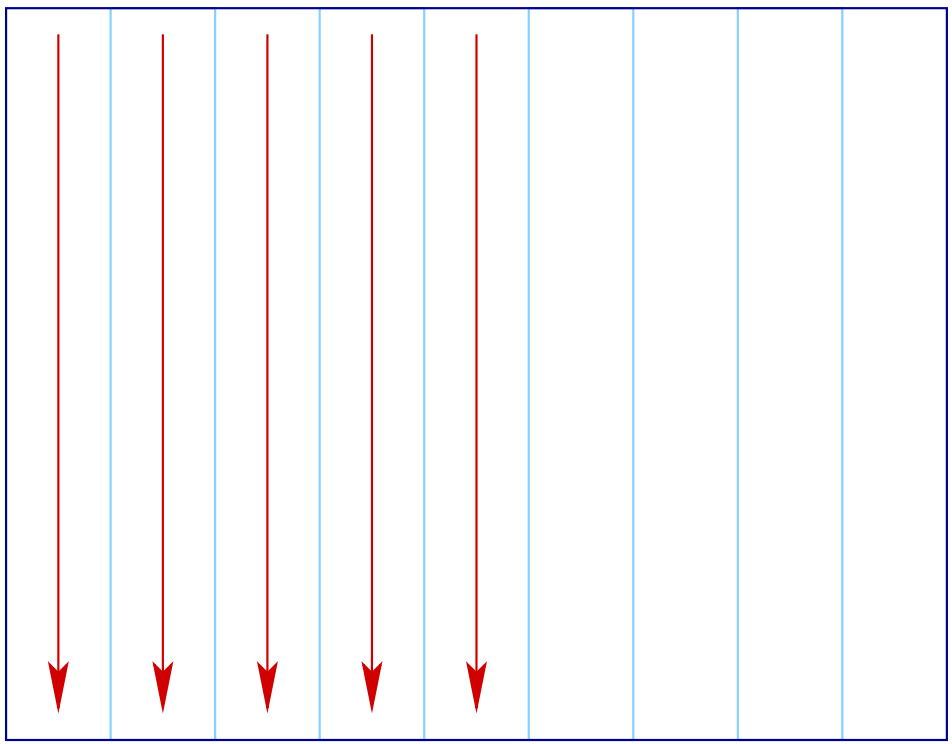
$\Longrightarrow$ Iteriere stets erst über die Zeilen!

$\Longrightarrow$ Vertausche die Reihenfolge der Iterationen:

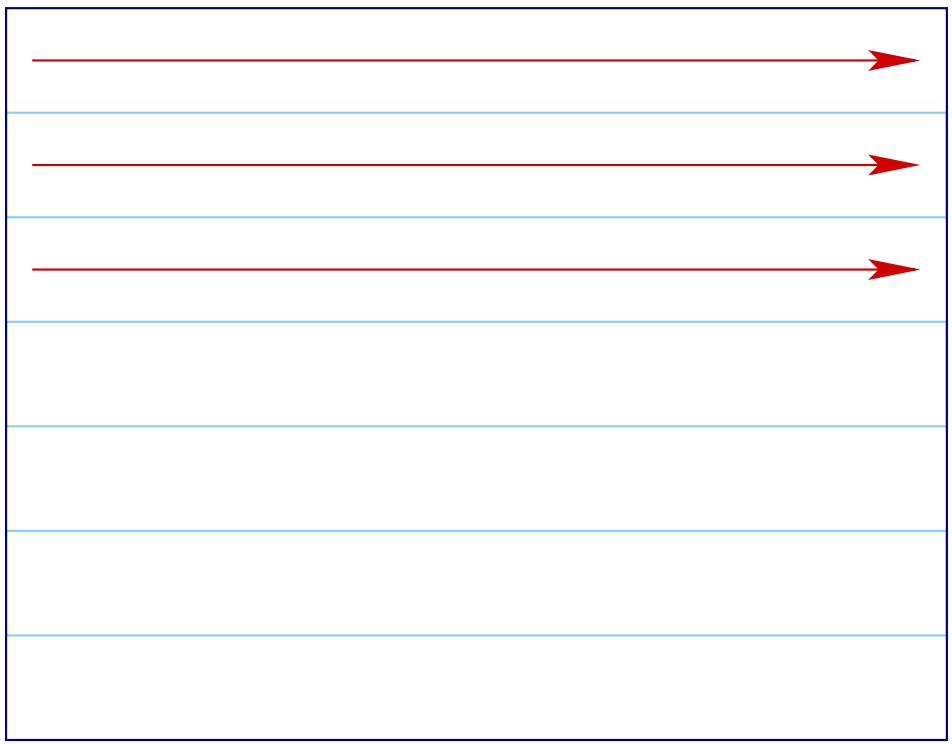
$$\begin{array}{l} \textsf{for}\ \ (i = 1; i < m; i{+}{+}) \\ \qquad \textsf{for}\ \ (j = 1; j < n; j{+}{+}) \\ \qquad\qquad a[i][j] = a[i-1][j-1] + a[i][j]; \end{array}$$

Wann ist das erlaubt ???

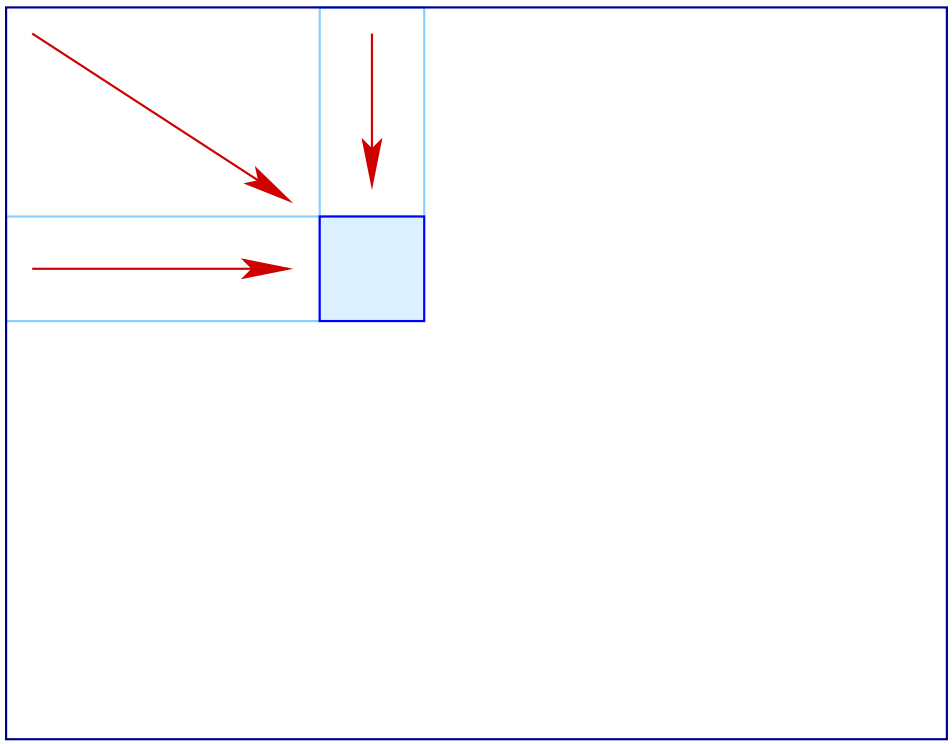
Iterations-Schema: vorher:



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	$\leq$	i_2
j_2	$\leq$	j_1
(i_1, j_1)	$=$	$(i_2 - 1, j_2 - 1)$
i_2	$\leq$	i_1
j_1	$\leq$	j_2

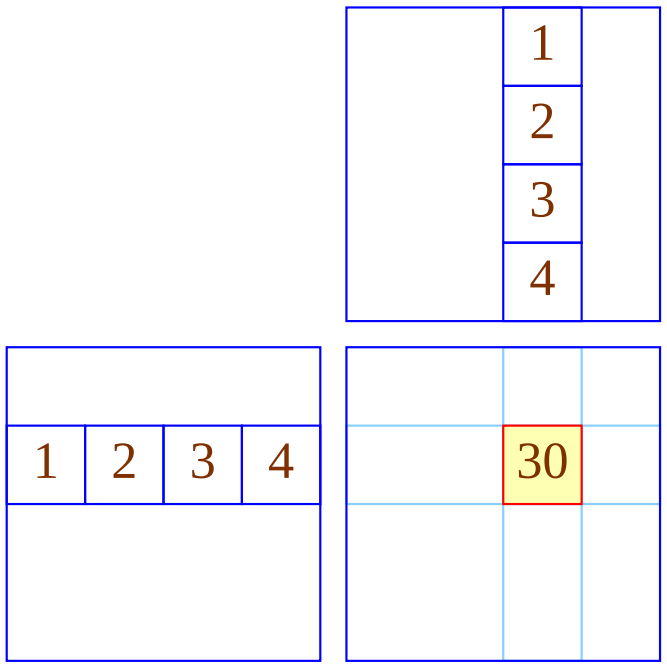
Das erste impliziert: $j_2 \leq j_2 - 1$ Hurra!

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

Beispiel: Matrix-Matrix-Multiplikation

$$
\begin{aligned}
&\textsf{for}\ \ (i = 0; i < N; i{+}{+}) \\
&\qquad \textsf{for}\ \ (j = 0; j < M; j{+}{+}) \\
&\qquad\qquad \textsf{for}\ \ (k = 0; k < K; k{+}{+}) \\
&\qquad\qquad\qquad c[i][j] = c[i][j] + a[i][k] \cdot b[k][j];
\end{aligned}
$$

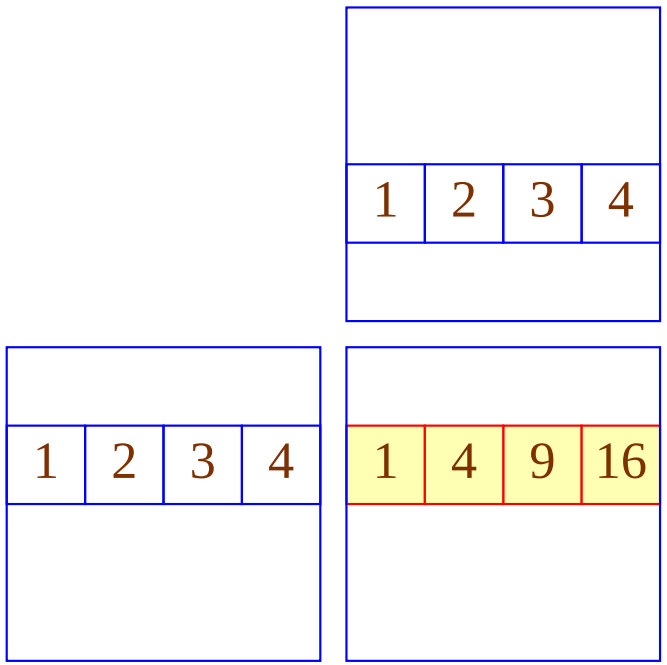
Über $b[][]$ iterieren wir spaltenweise :-(


1
2
3
4
1
2
3
4
30

Vertausche die beiden inneren Schleifen:

$$
\begin{array}{l}
\textsf{for}\ \ (i = 0; i < N; i{+}{+}) \\
\qquad \textsf{for}\ \ (k = 0; k < K; k{+}{+}) \\
\qquad\qquad \textsf{for}\ \ (j = 0; j < M; j{+}{+}) \\
\qquad\qquad\qquad c[i][j] = c[i][j] + a[i][k] \cdot b[k][j];
\end{array}
$$

Ist das erlaubt ???


1 2 3 4
1 2 3 4
1 4 9 16

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:

$\Longrightarrow$ Die gelesenen Einträge (hier: keine) dürfen im Rest des Rumpfs nicht modifiziert werden !!!

$\Longrightarrow$ Die Reihenfolge der Schreibzugriffe einer Zelle darf sich nicht ändern :-)

Man erhält:

$$
\begin{array}{l}
\textsf{for}\ \ (i = 0; i < N; i{+}{+})\ \ \{ \\
\qquad \textsf{for}\ \ (j = 0; j < M; j{+}{+})\ \ c[i][j] = 0; \\
\qquad \textsf{for}\ \ (k = 0; k < K; k{+}{+}) \\
\qquad\qquad \textsf{for}\ \ (j = 0; j < M; j{+}{+}) \\
\qquad\qquad\qquad c[i][j] = c[i][j] + a[i][k] \cdot b[k][j]; \\
\}
\end{array}
$$

Diskussion:

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

Achtung:

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

$$
\begin{array}{l}
\textsf{for}\ \ (i = 0; i < N; i{+}{+}) \\
\qquad \textsf{for}\ \ (j = 0; j < M; j{+}{+})\ \ \{ \\
\qquad\qquad t = 0; \\
\qquad\qquad \textsf{for}\ \ (k = 0; k < K; k{+}{+}) \\
\qquad\qquad\qquad t = t + a[i][k] \cdot b[k][j]; \\
\qquad\qquad c[i][j] = t; \\
\qquad \}
\end{array}
$$

Idee:

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

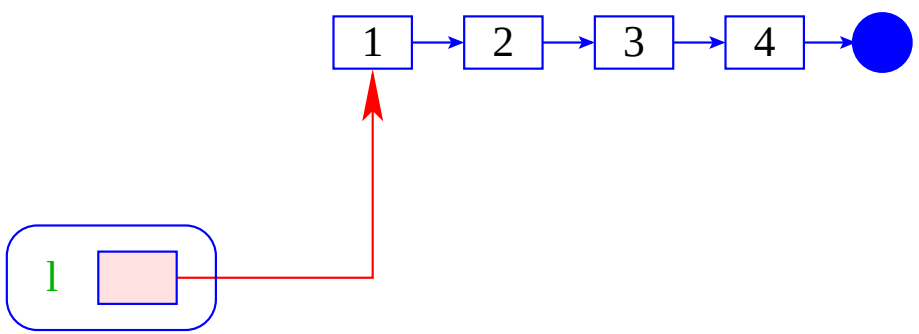
Achtung:

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



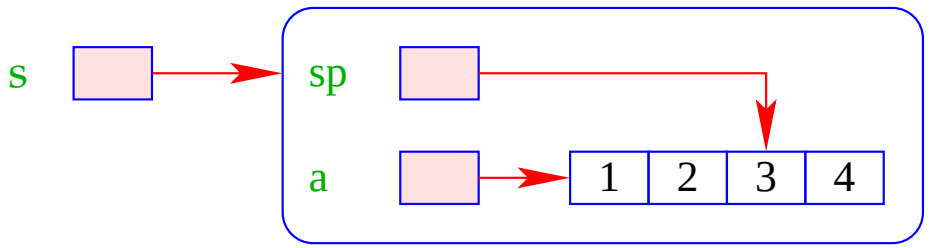
Vorteil:

+ 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 konsequtiv; Stack-Schwankungen sind im Mittel gering

  $\Longrightarrow$ 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 !!!

$\Longrightarrow$ Die Extra-Kosten sind amortisiert konstant :-)

$\Longrightarrow$ Die Implementierung ist nicht mehr ganz so trivial :-}

Diskussion:

$\rightarrow$ Die gleiche Idee klappt auch für Schlangen :-)

$\rightarrow$ Andere Datenstrukturen bemüht man sich, blockweise aufzuteilen.

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

$\Longrightarrow$ 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 allokieren :-)

$\Longrightarrow$ Escape-Analyse

Idee:

Berechne Alias-Information.

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

Beispiel: unsere Pointer-Sprache

$$
\begin{aligned}
x &= \mathsf{new}();\\
y &= \mathsf{new}();\\
x &\rightarrow a = y;\\
z &= y;\\
&\mathsf{return}\ \ z;
\end{aligned}
$$

... 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:

$$x = \mathsf{new}();$$
$$y = \mathsf{new}();$$
$$x \to a = y;$$
$$z = y;$$
$$\mathsf{return}\ \boxed{z};$$

Von außen zugänglich sind Objekte, die:

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

... im Beispiel:

$$
\begin{aligned}
x &= \mathsf{new}();\\
y &= \mathsf{new}();\\
x &\rightarrow a = y;\\
z &= \boxed{y};\\
&\mathsf{return}\ \boxed{z};
\end{aligned}
$$

Von außen zugänglich sind Objekte, die:

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

... im Beispiel:

$$
\begin{aligned}
& x = \mathsf{new}();\\
& y = \mathsf{new}();\\
& \boxed{x \rightarrow a} = y;\\
& z = \boxed{y};\\
& \mathsf{return}\ \boxed{z};
\end{aligned}
$$

Von außen zugänglich sind Objekte, die:

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

... im Beispiel:

$$
\begin{aligned}
& x = \mathsf{new}(); \\
& y = \boxed{\mathsf{new}()}; \\
& \boxed{x \rightarrow a} = y; \\
& z = \boxed{y}; \\
& \mathsf{return}\ \boxed{z};
\end{aligned}
$$

Wir schließen:

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

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 $y_1, y_2, \ldots$ 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 :-((

3.5 Zusammenfassung

Wir haben jetzt diverse Optimierungen kennen gelernt zur besseren Ausnutzung der Hardware-Gegebenheiten.

Reihenfolge ihrer Anwendung:

- Erst globale Restrukturierungen der Prozeduren/Funktionen sowie der Schleifen für besseres Speicherverhalten ;-)
- Dann lokale Umstrukturierung für optimale Nutzung des Instruktionssatzes und der Prozessor-Parallelität :-)
- Dann Registerverteilung und schließlich
- Peephole-Optimierung für den letzten Schliff ...

Funktionen:	Endrekursion + Inlining Stack-Allokation
Schleifen:	Iterationsverbesserung → if-Distribution → for-Distribution Werte-Caching
Rümpfe:	Life-Range-Splitting Instruktions-Auswahl Instruktions-Anordung mit → Schleifen-Abwicklung → Schleifen-Verschmelzung
Instruktionen:	Register-Verteilung Peephole-Optimierung