1. **Cache Optimization:**

**Idea:** local memory access

- Loading from memory fetches not just one byte but fills a complete cache line.
- Access to neighbored cells become cheaper.
- If all data of an inner loop fits into the cache, the iteration becomes maximally memory-efficient ...
Possible Solutions:

→ Reorganize the data accesses!
→ Reorganize the data!

Such optimizations can be made fully automatic only for arrays :-(

Example:

\[
\begin{align*}
\text{for } (j = 1; j < n; j++) & \\
\text{for } (i = 1; i < m; i++) & \\
& a[i][j] = a[i - 1][j - 1] + a[i][j];
\end{align*}
\]
At first, always iterate over the rows!

Exchange the ordering of the iterations:

\[
\text{for } (i = 1; i < m; i++)
\]

\[
\text{for } (j = 1; j < n; j++)
\]

\[a[i][j] = a[i-1][j-1] + a[i][j];\]

When is this permitted???
Iteration Scheme: before:
Iteration Scheme: after:
Iteration Scheme:  allowed dependencies:
In our case, we must check that the following equation systems have **no** solution:

<table>
<thead>
<tr>
<th>Write</th>
<th>Read</th>
</tr>
</thead>
<tbody>
<tr>
<td>((i_1, j_1) = (i_2 - 1, j_2 - 1))</td>
<td>((i_2 - 1, j_2 - 1))</td>
</tr>
<tr>
<td>(i_1 \leq i_2)</td>
<td>(j_2 \leq j_1)</td>
</tr>
<tr>
<td>(j_2 \leq j_2 - 1)</td>
<td>(i_1 \leq i_1)</td>
</tr>
</tbody>
</table>

The first implies: \(j_2 \leq j_2 - 1\) **Hurra!**

The second implies: \(i_2 \leq i_2 - 1\) **Hurra!**
Example: Matrix-Matrix Multiplication

\[
\begin{align*}
\text{for } (i = 0; i < N; i++) & \\
& \quad \text{for } (j = 0; j < M; j++) \\
& \quad \quad \text{for } (k = 0; k < K; k++) \\
& \quad \quad \quad c[i][j] = c[i][j] + a[i][k] \cdot b[k][j];
\end{align*}
\]

Over \( b[][] \) the iteration is columnwise :-(
Exchange the two inner loops:

\[
\begin{align*}
&\text{for } (i = 0; i < N; i++) \\
&\quad \text{for } (k = 0; k < K; k++) \\
&\quad\quad \text{for } (j = 0; j < M; j++) \\
&\quad\quad\quad c[i][j] = c[i][j] + a[i][k] \cdot b[k][j];
\end{align*}
\]

Is this permitted ???
Discussion:

- Correctness follows as before  :-)
- A similar idea can also be used for the implementation of multiplication for row compressed matrices  :-))
- Sometimes, the program must be massaged such that the transformation becomes applicable :-(
- Matrix-matrix multiplication perhaps requires initialization of the result matrix first ...
for \((i = 0; i < N; i++)\)

\[
\text{for } (j = 0; j < M; j++) \quad \{
\]
\(c[i][j] = 0;\)

\[
\text{for } (k = 0; k < K; k++) \quad \{
\]
\(c[i][j] = c[i][j] + a[i][k] \cdot b[k][j];\)

\[
\}\]

- Now, the two iterations can no longer be exchanged \:-(\)
- The iteration over \(j\), however, can be \textit{duplicated} ...

761
for \((i = 0; i < N; i++)\) {
    for \((j = 0; j < M; j++)\) \(c[i][j] = 0;\)
    for \((j = 0; j < M; j++)\)
        for \((k = 0; k < K; k++)\)
            \(c[i][j] = c[i][j] + a[i][k] \cdot b[k][j];\)
}  

**Correctness:**

\[\implies\] The read entries (here: no) may not be modified in the remaining body of the loop !!!

\[\implies\] The ordering of the write accesses to a memory cell may not be changed :-)

762
We obtain:

\[
\begin{align*}
&\text{for } (i = 0; i < N; i++) \{ \\
&\quad \text{for } (j = 0; j < M; j++) \ c[i][j] = 0; \\
&\quad \text{for } (k = 0; k < K; k++) \\
&\quad \quad \text{for } (j = 0; j < M; j++) \\
&\quad \quad \quad c[i][j] = c[i][j] + a[i][k] \cdot b[k][j];
\end{align*}
\]

Discussion:

- Instead of fusing several loops, we now have distributed the loops :-)  
- Accordingly, conditionals may be moved out of the loop \[=>\] if-distribution ...
Warning:

Instead of using this transformation, the inner loop could also be optimized as follows:

```plaintext
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] \cdot b[k][j];
        c[i][j] = t;
    }
```
Idea:

If we find heavily used array elements $a[e_1] \ldots [e_r]$ whose index expressions stay constant within the inner loop, we could instead also provide auxiliary registers :-)

Warning:

The latter optimization prohibits the former and vice versa ...
Discussion:

- so far, the optimizations are concerned with iterations over arrays.
- Cache-aware organization of other data-structures is possible, but in general not fully automatic ...

Example: Stacks
Advantage:

+ The implementation is simple  :-)
+ The operations push / pop require constant time  :-)
+ The data-structure may grow arbitrarily  :-)

Disadvantage:

– The individual list objects may be arbitrarily dispersed over the memory  :-(
Alternative:

Advantage:

+ The implementation is also simple :-) 
+ The operations push / pop still require constant time :-) 
+ The data are consecutively allocated; stack oscillations are typically small

⇒⇒⇒ better Cache behavior !!!
Disadvantage:

− The data-structure is bounded  :-(

Improvement:

• If the array is full, replace it with another of double size !!!
• If the array drops empty to a quarter, halve the array again !!!

⇒ The extra amortized costs are constant  :-)
⇒ The implementation is no longer so trivial  :-}
Discussion:

→ The same idea also works for queues :-(
→ Other data-structures are attempted to organize blockwise.

Problem: how can accesses be organized such that they refer mostly to the same block ???

⇒ Algorithms for external data